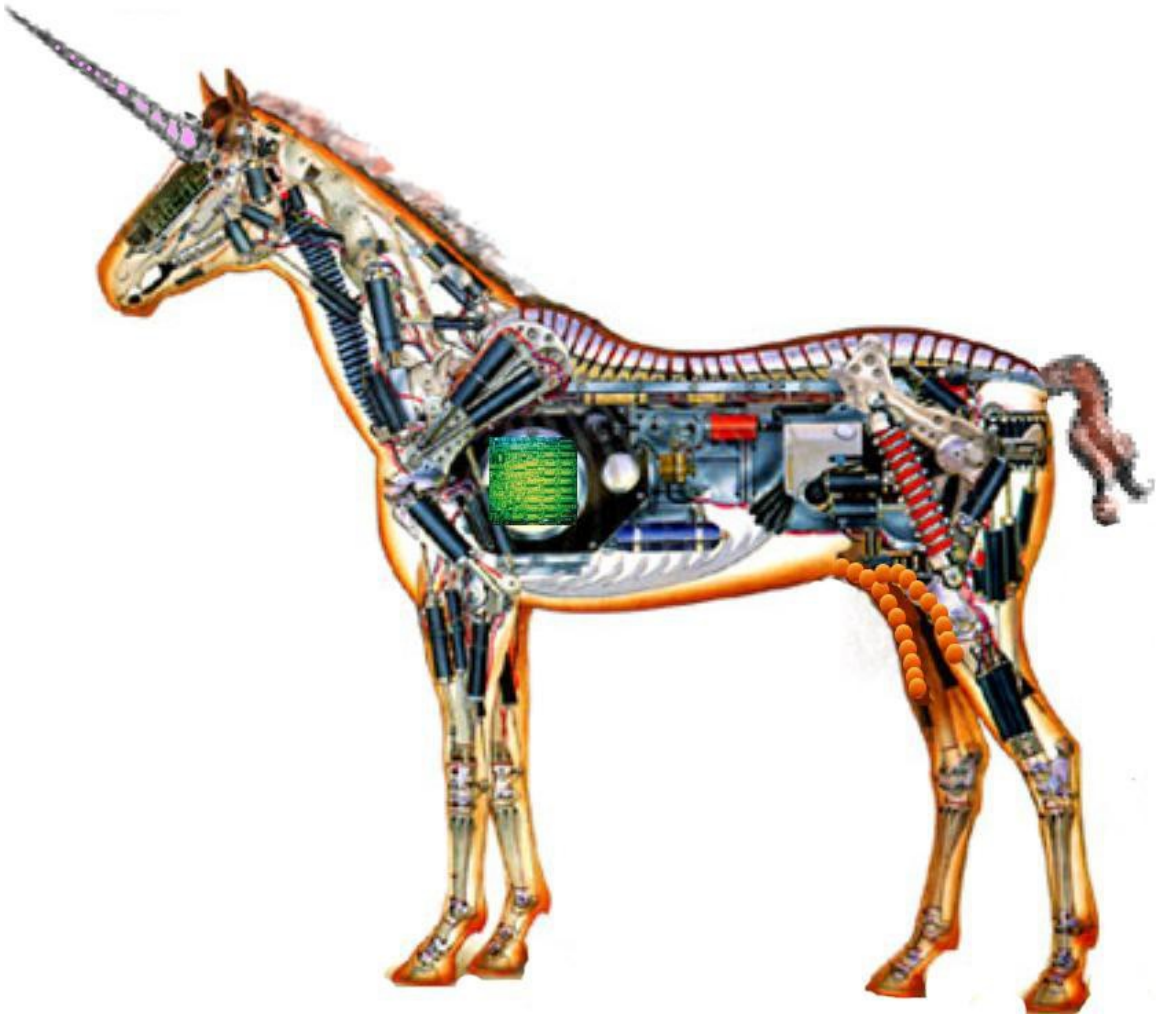


The

Implementation of Icon and Unicon

a Compendium

Clinton Jeffery, editor

The Implementation of Icon and Unicon

Ralph and Madge T. Griswold

Kenneth W. Walker

Clinton L. Jeffery

This is a draft manuscript dated 6/6/2014. Send comments and errata to jeffery@cs.uidaho.edu.

This document was prepared using OpenOffice.org 4.1.

Contents

Preface

This book is a compendium of all documents that describe the implementation of the Icon and Unicon programming languages, an implementation that started with Icon version 3 on a PDP-11 sometime near the year 1980.

Organization of This Book

This book consists of four parts. The first part, Chapters 1-12, present the core of the implementation, focusing on the Icon virtual machine interpreter and runtime system. This material was formerly published as the Implementation of the Icon Programming Language, by Ralph and Madge T. Griswold; at that time it documented Icon Version 6. Many of the details in this book became obsolete with the rewriting of the runtime system for Icon Version 8. After long consideration, I have elected to preserve the authors' style and intent, while updating it to document Icon Version 9.5 and Unicon Version 12. Blue-colored text indicates when necessary Unicon issues and differences, so that Part I remains useful to people who prefer to use the Icon implementation, not just those working with Unicon.

Part II, in Chapters 13-19, describes the optimizing compiler, iconc, and the structuring of the runtime system to support it. This work is the brainchild of Ken Walker, whose dissertation is presented here, along with his technical reports describing the runtime language RTL and its translator, rtt. Ken's compiler has been enhanced significantly by Anthony Jones' B.S. Honors thesis at UTSA on space reduction techniques that reduce the space cost of type inferencing by 2/3rds, and Mike Wilder's M.S. thesis at NMSU and follow-on work at Idaho on adapting iconc to support Unicon. These contributions belong logically to Part II.

Part III describes the implementation of Unicon and the many extensions that transformed the language from a string-and-list-processing language into a modern object-oriented, network-savvy, graphics-rich applications language. Part IV consists of essential reference material presented in several Appendixes.

Acknowledgments

This book would not be possible without the generous contributions and consent of the primary authors of the Icon language implementation documents, Ralph and Madge Griswold, and Kenneth Walker. Ralph Griswold re-scanned and corrected his Icon implementation book manuscript in order to place it in the public domain on the web, a large, selfless, thankless, and valuable undertaking. Ken Walker found and shared his original nroff dissertation source files.

Susie Jeffery provided crucial assistance in the OCR reconstruction of Icon implementation book manuscript from the public domain scanned images. Mike Kemp was a valuable volunteer proofreader in that effort. Responsibility for remaining typographical errors rests with me.

Thanks to the rest of the people who contributed code to the Icon and Unicon Projects over a period of many years, and to those who contributed while obtaining many Ph.D. and M.S. degrees.

The editor wishes to acknowledge generous support from the National Library of Medicine. This work was also supported in part by the National Science Foundation under grants CDA-9633299, EIA-0220590 and EIA-9810732, and the Alliance for Minority Participation.

Clinton Jeffery, Moscow ID, May 2014

Acknowledgments for Chapters 1-12

The implementation of Icon described in Part I owes much to previous work and in particular to implementations of earlier versions of Icon. Major contributions were made by Cary Coutant, Dave Hanson, Tim Korb, Bill Mitchell, a Steve Wampler. Walt Hansen, Rob McConeghy, and Janalee O'Bagy also made significant contributions to this work.

The present system has benefited greatly from persons who have installed Icon on a variety of machines and operating systems. Rick Fonorow, Bob Goldberg, Chris Janton, Mark Langley, Rob McConeghy, Bill Mitchell, Janal O'Bagy, John Polstra, Gregg Townsend, and Cheyenne Wills have made substantial contributions in this area.

The support of the National Science Foundation under Grants MCS7 01397, MCS79-03890, MCS81-0l916, DCR-8320138, DCR-840183I, at DCR-8502015 was instrumental in the original conception of Icon and has bee invaluable in its subsequent development.

A number of persons contributed to this book. Dave Gudeman, Dave Ha son, Bill Mitchell, Janalee O'Bagy, Gregg Townsend, and Alan Wendt contr buted to the exercises that appear at the ends of chapters and the projects given

Appendix E. Kathy Cummings, Bill Griswold, Bill Mitchell, Katie Morse, Mil Tharp, and Gregg Townsend gave the manuscript careful readings and made numerous suggestions. Janalee O'Bagy not only read the manuscript but also supplied concepts for presenting and writing the material on expression evaluation.

Finally, Dave Hanson served as an enthusiastic series editor for this book. His perceptive reading of the manuscript and his supportive and constructive suggestions made a significant contribution to the final result.

Acknowledgments for Chapters 13-24

I would like to thank Ralph Griswold for acting as my research advisor. He provided the balance of guidance, support, and freedom needed for me to complete this research. From him I learned many of the technical writing skills I needed to compose this dissertation. I am indebted to him and the other members of the Icon Project who over the years have contributed to the Icon programming language that serves as a foundation of this research. I would like to thank Peter Downey and Saumya Debray for also serving as members on my committee and for providing insightful criticisms and suggestions for this dissertation. In addition, Saumya Debray shared with me his knowledge of abstract interpretation, giving me the tool I needed to shape the final form of the type inferencing system.

I have received help from a number of my fellow graduate students both while they were still students and from some after they graduated. Clinton Jeffery, Nick Kline, and Peter

Bigot proofread this dissertation, providing helpful comments. Similarly, Janalee O'Bagy, Kelvin Nilsen, and David Gudeman proofread earlier reports that served as a basis for several of the chapters in this dissertation. Janalee O'Bagy's own work on compiling Icon provided a foundation for the compiler I developed. Kelvin Nilsen applied my liveness analysis techniques to a slightly different implementation model, providing insight into dependencies on execution models.

Compendium Introduction

The implementation of the Icon programming language is now old. It inherits ideas from earlier languages, and introduces many of its own. The implementation documentation traditionally revolved around the virtual machine and its runtime system; other parts of the implementation were documented in scattered technical reports or not at all, other than the source code. This volume changes all that, by bringing all the implementation documents together in a single volume.

Icon's public-domain implementation is fairly efficient; for example at one point Keith Waclena of the University of Chicago documented a factor of 4 or more speed advantage of Icon versus Python on multiple benchmarks, and that was for the Icon virtual machine interpreter; the Icon optimizing compiler adds another factor of 2-5 or more in faster execution speed. The design decisions that achieve Icon's very-high level language features (such as generators and goal-directed evaluation) with acceptable performance make for an interesting study. This book is intended for those wanting to learn the implementation in order to add features, improve performance, learn about compilers in general, or glean ideas for their own independent programming language efforts.

Icon traditionally consisted of a virtual machine translator, a linker, and a virtual machine interpreter. The translator and linker were merged long ago, but other tools have been added. The big added components are the optimizing compiler written by Ken Walker, and the Unicon translator written by Clint Jeffery. These additions are now a large part of the story. The trends I hope to see in the future are: merger of components, and gradual replacement of C-based components with ones written in Unicon.

How Many Compilers?

The figure below shows two symmetrically-organized sets of tools. The tools on the left are the compilers end-users employ to translate Icon or Unicon into executable machine code, while the tools on the right show how the underlying run-time system needed in order to execute those programs is built. Of the six rectangles, four are compilers that perform distinct tasks specific to this programming language family. The front-end translation tool, named unicon, is a preprocessor that translates Unicon code into Icon code. Its primary functions are to translate object-orientation (classes, single and multiple inheritance, and packages) down to underlying imperative constructs. Unicon is written in Unicon. Icont and iconc compile Icon code down to virtual machine and C code, respectively. They share a few common front-end components, but are largely independent. Iconx is the name of the Icon (and Unicon) virtual machine, which mostly consists of a large collection of complex high-level data structure and I/O facilities which are built-in to these languages. Most of the source code for iconx is also used in rt.a, the runtime library that is linked to Icon programs compiled with iconc.

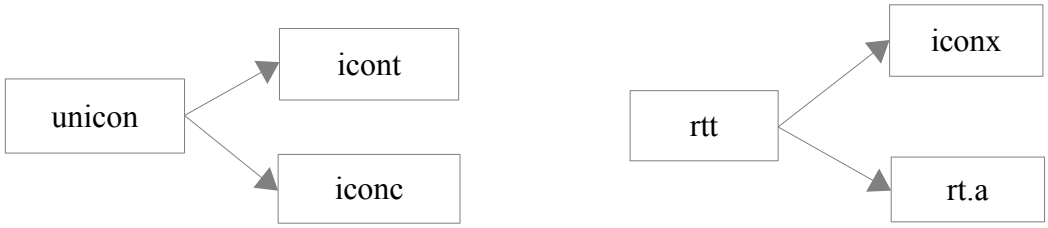


Figure CI-1: three compilers for users (left), one (rtt) for the language implementors

Part I: The Implementation of the Icon Programming Language

by Ralph Griswold and Madge Griswold

discussion updated to Icon 9.5 source code by Clint Jeffery

Chapter 1: Introduction

PERSPECTIVE: The implementation of complex software systems is a fascinating subject-and an important one. Its theoretical and practical aspects occupy the attention and energy of many persons, and it consumes vast amounts of computational resources. In general terms, it is a broad subject ranging from operating systems to programming languages to data-base systems to real-time control systems, and so on.

Past work in these areas has resulted in an increasingly better understanding of implementation techniques, more sophisticated and efficient systems, and tools for automating various aspects of software production. Despite these advances, the implementation of complex software systems remains challenging and exciting. The problems are difficult, and every advance in the state of the art brings new and more difficult problems within reach.

Part I of this book addresses a very small portion of the problem of implementing complex software systems—the implementation of a very high-level programming language that is oriented toward the manipulation of structures and strings of characters.

In a narrow sense, this book describes an implementation of a specific programming language, Icon. In a broader sense, it deals with a language-design philosophy, an approach to implementation, and techniques that apply to the implementation of many programming languages as well as related types of software systems.

The focus of this book is the implementation of programming language features that are at a high conceptual level—features that are easy for human beings to use as opposed to features that fit comfortably on conventional computer architectures. The orientation of the implementation is generality and flexibility, rather than maximum efficiency of execution. The problem domain is strings and structures rather than numbers. It is these aspects that set the implementation of Icon apart from more conventional programming-language implementations.

1.1 Implementing Programming Languages

In conventional programming languages, most of the operations that are performed when a program is executed can be determined, statically, by examining the text of the program. In addition, the operations of most programming languages have a fairly close correspondence to the architectural characteristics of the computers on which they are implemented. When these conditions are met, source-code constructs can be mapped directly into machine instructions for the computer on which they are to be executed. The term *compilation* is used for this translation process, and most persons think of the implementation of a programming language in terms of a compiler.

Writing a compiler is a complex and difficult task that requires specialized training, and the subject of compilation has been studied extensively (Waite and Goos, 1984; Aho, Lam, Sethi and Ullman 2006). Most of the issues of data representation and code generation are comparatively well understood, and there are now many tools for automating portions of the compiler-writing task (Lesk 1975, Johnson 1975).

In addition to the compiler proper, an implementation of a programming language usually includes a run-time component that contains subroutines for performing computations

that are too complex to compile in-line, such as input, output, and mathematical functions.

Some programming languages have features whose meanings cannot be determined statically from the text of a source-language program, but which may change during program execution. Such features include changes in the meaning of functions during execution, the creation of new data types at run time, and self-modifying programs. Some programming languages also have features, such as pattern matching, that do not have correspondences in the architecture of conventional computers. In such cases, a compiler cannot translate the source program directly into executable code. Very high-level operations, such as pattern matching, and features like automatic storage management significantly increase the importance and complexity of the run-time system. For languages with these characteristics--languages such as APL, LISP, SNOBOL4, SETL, Prolog, and Icon-much of the substance of the implementation is in the run-time system rather than in translation done by a compiler. While compiler writing is relatively well understood, run-time systems for most programming languages with dynamic features and very high-level operations are not.

Programming languages with dynamic aspects and novel features are likely to become more important rather than less important. Different problems benefit from different linguistic mechanisms. New applications place different values on speed of execution, memory requirements, quick solutions, programmer time and talent, and so forth. For these reasons, programming languages continue to proliferate. New programming languages, by their nature, introduce new features.

All of this creates difficulties for the implementer. Less of the effort involved in implementations for new languages lies in the comparatively familiar domain of compilation and more lies in new and unexplored areas, such as pattern matching and novel expression-evaluation mechanisms.

The programming languages that are the most challenging to implement are also those that differ most from each other. Nevertheless, there are underlying principles and techniques that are generally applicable, and existing implementations contain many ideas that can be used or extended in new implementations.

1.2 The Background for Icon

Before describing the Icon programming language and its implementation, some historical context is needed, since both the language and its implementation are strongly influenced by earlier work.

Icon has its roots in a series of programming languages that bear the name SNOBOL. The first SNOBOL language was conceived and implemented in the early 1960s at Bell Telephone Laboratories in response to the need for a programming tool for manipulating strings of characters at a high conceptual level (Farber, Griswold, and Polonsky 1964). It emphasized ease of programming at the expense of efficiency of execution; the programmer was considered to be a more valuable resource than the computer.

This rather primitive language proved to be popular, and it was followed by successively more sophisticated languages: SNOBOL2, SNOBOL3 (Farber, Griswold, and Polonsky 1966), and finally SNOBOL4 (Griswold, Poage, and Polonsky 1971). Throughout the development of these languages, the design emphasis was on ease of programming rather than on ease of implementation (Griswold 1981). Potentially valuable features were not

discarded because they might be inefficient or difficult to implement. The aggressive pursuit of this philosophy led to unusual language features and to challenging implementation problems.

SNOBOL4 still is in wide use. Considering its early origins, some of its facilities are remarkably advanced. It features a pattern-matching facility with backtracking control structures that effectively constitutes a sublanguage. SNOBOL4 also has a variety of data structures, including tables with associative lookup. Functions and operators can be defined and redefined during program execution. Identifiers can be created at run-time, and a program can even modify itself by means of run-time compilation.

Needless to say, SNOBOL4 is a difficult language to implement, and most of the conventional compilation techniques have little applicability to it. Its initial implementation was, nonetheless, sufficiently successful to make SNOBOL4 widely available on machines ranging from large mainframes to personal computers (Griswold 1972). Subsequent implementations introduced a variety of clever techniques and fast, compact implementations (Santos 1971; Gimpel 1972a; Dewar and McCann 1977). The lesson here is that the design of programming languages should not be overly inhibited by perceived implementation problems, since new implementation techniques often can be devised to solve such problems effectively and efficiently.

It is worth noting that the original implementation of SNOBOL4 was carried out concomitantly with language design. The implementation was sufficiently flexible to serve as a research tool in which experimental language features could be incorporated easily and tested before they were given a permanent place in the language.

Work on the SNOBOL languages continued at the University of Arizona in the early 1970s. In 1975, a new language, called SL5 ("SNOBOL Language 5"), was developed to allow experimentation with a wider variety of programming-language constructs, especially a sophisticated procedure mechanism (Griswold and Hanson, 1977; Hanson and Griswold 1978). SL5 extended earlier work in pattern matching, but pattern matching remained essentially a sublanguage with its own control structures, separate from the rest of the language.

The inspiration for Icon came in 1976 with a realization that the control structures that were so useful in pattern matching could be integrated with conventional computational control structures to yield a more coherent and powerful programming language.

The first implementation of Icon (Griswold and Hanson 1979) was written in Ratfor, a preprocessor for Fortran that supports structured programming features (Kernighan 1975). Portability was a central concern in this implementation. The implementation of Icon described in this book is a successor to that first implementation. It borrows much from earlier implementations of SNOBOL4, SL5, and the Ratfor implementation of Icon. As such, it is a distillation and refinement of implementation techniques that have been developed over a period of more than twenty years.

Chapter 2: Icon Language Overview

PERSPECTIVE: The implementer of a programming language needs a considerably different understanding of the language from the persons who are going to use it. An implementer must have a deep understanding of the relationships that exist among various aspects of the language and a precise knowledge of what each operation means. Special cases and details often are of particular importance to the implementer. Users of a language, on the other hand, must know how to use features to accomplish desired results. They often can get by with a superficial knowledge of the language, and they often can use it effectively even if some aspects of the language are misunderstood. Users can ignore parts of the language that they do not need. Idiosyncrasies that plague the implementer may never be encountered by users. Conversely, a detail the implementer overlooks may bedevil users. Furthermore, the implementer may also need to anticipate ways in which users may apply some language features in inefficient and inappropriate ways.

Part I of this book is about the implementation of Version 9 of Icon. The description that follows concentrates on aspects of the language that are needed to understand its implementation. Where there are several similar operations or where the operations are similar to those in well-known programming languages, only representative cases or highlights are given. A complete description of Icon for the user is contained in Griswold and Griswold (1997).

Icon is an unusual programming language, and its unusual features are what make its implementation challenging and interesting. The interesting features are semantic, not syntactic; they are part of what the language can do, not part of its appearance. Syntactic matters and the way they are handled in the implementation are of little interest here. The description that follows indicates syntax mostly by example.

This chapter is divided into two major parts. The first part describes the essential aspects of Icon. The second part discusses those aspects of Icon that present the most difficult implementation problems and that affect the nature of the implementation in the most significant ways.

2.1 The Icon Programming Language

Icon is conventional in many respects. It is an imperative, procedural language with variables, operations, functions, and conventional data types. Its novel aspects lie in its emphasis on the manipulation of strings and structures and in its expression-evaluation mechanism. While much of the execution of an Icon program has an imperative flavor, there also are aspects of logic programming.

There are no type declarations in Icon. Instead, variables can have any type of value. Structures may be heterogeneous, with different elements having values of different types. Type checking is performed during program execution, and automatic type conversion is provided. Several operations are polymorphic, performing different operations depending on the types of their arguments.

Strings and structures are created during program execution, instead of being declared and allocated during compilation. Structures have pointer semantics; a structure value is a pointer to an object. Storage management is automatic. Memory is allocated as required, and garbage collection is performed when necessary. Except for the practical

considerations of computer architecture and the amount of available memory, there are no limitations on the sizes of objects.

An Icon program consists of a series of declarations for procedures, records, and global identifiers. Icon has no block structure. Scoping is static: identifiers either are global or are local to procedures.

Icon is an expression-based language with reserved-word syntax. It resembles C in appearance, for example (Kernighan and Ritchie 1978).

2.1.1 Data Types

Icon has many types of data--including several that are not found in most programming languages. In addition to the usual integers and real (floating-point) numbers, there are strings of characters and sets of characters (csets). There is no character data type, and strings of characters are data objects in their own right, not arrays of characters.

There are four structure data types that comprise aggregates of values: lists, sets, tables, and records. Lists provide positional access (like vectors), but they also can be manipulated like stacks and queues. Sets are unordered collections of values on which the usual set operations can be performed. Tables can be subscripted with any kind of value and provide an associative-access mechanism. Records are aggregates of values that can be referenced by name. Record types also add to the built-in type repertoire of Icon.

The null value serves a special purpose; all variables have the null value initially. The null value is illegal in most computational contexts, but it serves to indicate default values in a number of situations. The keyword &null produces the null value.

A source-language file is a data value that provides an interface between the program and a data file in the environment in which the program executes.

Procedures also are data values---"first-class data objects" in LISP parlance. Procedures can be assigned to variables, transmitted to and returned from functions, and so forth. There is no method for creating procedures during program execution, however.

Finally, there is a co-expression data type. Co-expressions are the expression-level analog of coroutines. The importance of co-expressions is derived from Icon's expression-evaluation mechanism.

Icon has various operations on different types of data. Some operations are polymorphic and accept arguments of different types. For example, type(x) produces a string corresponding to the type of x. Similarly, copy(x) produces a copy of x, regardless of its type. Other operations only apply to certain types. An example is:

```
*x
```

which produces the size of x, where the value of x may be a string, a structure, and so on. Another example is ?x, which produces a randomly selected integer between 1 and x, if x is an integer, but a randomly selected one-character substring of x if x is a string, and so on. In other cases, different operations for similar kinds of computations are syntactically distinguished. For example,

```
i = j
```

compares the numeric values of i and j, while

```
s1 == s2
```

compares the string values of s1 and s2. There is also a general comparison operation that determines whether any two objects are the same:

```
x1 === x2
```

As mentioned previously, any kind of value can be assigned to any variable. For example, x might have an integer value at one time and a string value at another:

```
x := 3
...
x := "hello"
```

Type checking is performed during program execution. For example, in

```
i := x + 1
```

the value of x is checked to be sure that it is numeric. If it is not numeric, an attempt is made to convert it to a numeric type. If the conversion cannot be performed, program execution is terminated with an error message.

Various conversions are supported. For example, a number always can be converted to a string. Thus,

```
write(*s)
```

automatically converts the integer returned by *s to a string for the purpose of output.

There also are explicit type-conversion functions. For example,

```
s1 := string(*s2)
```

assigns to s1 a string corresponding to the size of s2.

A string can be converted to a number if it has the syntax of a number. Thus,

```
i := i + "20"
```

produces the same result as

```
i := i + 20
```

Augmented assignments are provided for binary operations such as the previous one, where assignment is made to the same variable that appears as the left argument of the operation. Therefore, the previous expression can be written more concisely as

```
i +:= 20
```

Icon also has the concept of a numeric type, which can be either an integer or a real (floating-point) number.

2.1.2 Expression Evaluation

In most programming languages---Algol, Pascal, PL/I, and C, for example---the evaluation of an expression always produces exactly one result. In Icon, the evaluation of an expression may produce a single result, it may produce no result at all, or it may produce a sequence of results.

Success and Failure. Conventional operations in Icon produce one result, as they do in most programming languages. For example,

```
i + j
```

produces a single result, the sum of the values of i and j. However, a comparison operation such as

```
i > j
```

produces a result (the value of j) if the value of i is greater than the value of j but does not produce a result if the value of i is not greater than j.

An expression that does not produce a result is said to *fail,* while an expression that produces a result is said to *succeed.* Success and failure are used in several control structures to control program flow. For example,

```
if i > j then write(i) else write(j)
```

writes the maximum of i and j. Note that comparison operations do not produce Boolean values and that Boolean values are not used to drive control structures. Indeed, Icon has no Boolean type.

Generally speaking, an operation that cannot perform a computation does not produce a result, and hence it fails. For example, type-conversion functions fail if the conversion cannot be performed. An example is numeric(x), which converts x to a numeric value if possible, but fails if the conversion cannot be performed. Failure of an expression to produce a result does not indicate an error. Instead, failure indicates that a result does not exist. An example is provided by the function find(s1, s2), which produces the position of s1 as a substring of s2 but fails if s1 does not occur in s2. For example,

```
find("it", "They sit like bumps on a log.")
```

produces the value 7 (positions in strings are counted starting at 1). However,

```
find("at", "They sit like bumps on a log.")
```

does not produce a result. Similarly, read(f) produces the next line from the file f but fails when the end of the file is reached.

Failure provides a natural way to control loops. For example,

```
while line := read(f) do
   write(line)
```

writes the lines from the file f until an end of file causes read to fail, which terminates the loop.

Another use of success and failure is illustrated by the operation

```
\expr
```

which fails if *expr* is null-valued but produces the result of *expr* otherwise. Since variables have the null value initially, this operation may be used to determine whether a value has been assigned to an identifier, as in

```
if \x then write(x) else write("x is null")
```

If an expression that is enclosed in another expression does not produce a result, there is no value for the enclosing expression, it cannot perform a computation, and it also produces no result. For example. In

```
write(find("at", "They sit like bumps on a log."))
```

the evaluation of find fails, there is no argument for write, and no value is written.

Similarly, in

```
i := find("at", "They sit like bumps on a log.")
```

the assignment is not performed and the value of i is not changed.

This "inheritance" of failure allows computations to be expressed concisely. For example,

```
while write(read(f))
```

writes the lines from the file f just as the previous loop (the do clause in while-do is optional).

The expression

```
not expr
```

inverts success and failure. It fails if *expr* succeeds, but it succeeds, producing the null value, if *expr* fails.

Some expressions produce variables, while others only produce values. For example,

```
i + j
```

produces a value, while

```
i := 10
```

produces its left-argument variable. The term *result* is used to refer to a value or a variable. The term *outcome* is used to refer to the consequences of evaluating an expression---either its result or failure.

Loops. There are several looping control structures in Icon in addition to while-do. For example,

```
until expr1 do expr2
```

evaluates *expr2* repeatedly until *expr1* succeeds. The control structure

```
repeat expr
```

simply evaluates *expr* repeatedly, regardless of whether it succeeds or fails.

A loop itself produces no result if it completes, and hence it fails if used in a conditional context. That is, when

```
while expr1 do expr2
```

terminates, its outcome is failure. This failure ordinarily goes unnoticed, since loops usually are not used as arguments of other expressions.

The control structure

```
break expr
```

causes the immediate termination of the evaluation of the loop in which it appears, and control is transferred to the point immediately after the loop. The outcome of the loop in this case is the outcome of *expr*. If *expr* is omitted, it defaults to the null value.

An example of the use of break is:

```
while line := read(f) do
   if line == "end" then break
   else write(line)
```

Evaluation of the loop terminates if read fails or if the file f contains a line consisting of "end".

The expression next causes transfer to the beginning of the loop in which it occurs. For example,

```
while line := read(f) do
   if line == "comment" then next
   else write(line)
```

does not write the lines of f that consist of "comment".

The break and next expressions can occur only in loops, and they apply to the innermost loop in which they appear. The argument of break can be a break or next expression, however, so that, for example,

```
break break next
```

breaks out of two levels of loops and transfers control to the beginning of the loop in which they occur.

Case Expressions. The case expression provides a way of selecting one of several expressions to evaluate based on the value of a control expression, rather than its success or failure. The case expression has the form

```
case expr of {
   case clauses
   ...
   }
```

The value of *expr* is used to select one of the case clauses. A case clause has the form

```
expr1 : expr2
```

where the value of *expr* is compared to the value of *expr1,* and *expr2* is evaluated if the comparison succeeds. There is also a default case clause, which has the form

```
default: expr3
```

If no other case clause is selected, expr3 in the default clause is evaluated. An example is

```
case line := read(f) of {
   "end":          write("*** end ***")
   "comment":      write("*** comment ***")
   default:  write(line)
   }
```

If the evaluation of the control clause fails, as for an end of file in this example, the entire case expression fails. Otherwise, the outcome of the case expression is the outcome of evaluating the selected expression.

Generators. As mentioned previously, an expression may produce a sequence of results. This occurs in situations in which there is more than one possible result of a computation. An example is

```
find("e", "They sit like bumps on a log.")
```

in which both 3 and 13 are possible results.

While most programming languages produce only the first result in such a situation, in Icon the two results are produced one after another if the surrounding context requires both of them. Such expressions are called *generators* to emphasize their capability of producing more than one result.

There are two contexts in which a generator can produce more than one result: *iteration* and *goal-directed evaluation*.

Iteration is designated by the control structure

```
every expr1 do expr2
```

in which expr1 is repeatedly resumed to produce its results. For each such result, expr2 is evaluated. For example,

```
every i := find("e", "They sit like bumps on a log.") do
   write(i)
```

writes 3 and 13.

If the argument of an expression is a generator, the results produced by the generator are provided to the enclosing expression—the sequence of results is inherited. Consequently, the previous expression can be written more compactly as

```
every write(find("e", "They sit like bumps on a log."))
```

Unlike iteration, which resumes a generator repeatedly to produce all its results, goal-directed evaluation resumes a generator only as necessary, in an attempt to cause an enclosing expression to succeed. While iteration is explicit and occurs only where specified, goal-directed evaluation is implicit and is an inherent aspect of Icon's expression-evaluation mechanism.

Goal-directed evaluation is illustrated by

```
if find("e", "They sit like bumps on a log") > 10
then write("found")
```

The first result produced by `find()` is 3, and the comparison operation fails. Because of goal-directed evaluation, find is automatically resumed to produce another value. Since this value, 13, is greater than 10, the comparison succeeds, and found is written. On the other hand, in

```
if find("e", "They sit like bumps on a log.") > 20
then write("found")
```

the comparison fails for 3 and 13. When find is resumed again, it does not produce another result, the control clause of if-then fails, and nothing is written.

There are several expressions in Icon that are generators, including string analysis functions that are similar in nature to find. Another generator is

```
i to j by k
```

which generates the integers from `i` to `j` by increments of `k`. If the `by` clause is omitted, the increment defaults to one.

The operation `!x` is polymorphic, generating the elements of `x` for various types. The meaning of "element" depends on the type of `x`. If `x` is a string, `!x` generates the one-character substrings of `x`, so that `!"hello"` generates `"h"`, `"e"`, `"l"`, `"l"`, and `"o"`. If `x` is a file, `!x` generates the lines of the file, and so on.

Generative Control Structures. There are several control structures related to generators. The *alternation* control structure,

```
expr1 | expr2
```

generates the results of expr1 followed by the results of expr2. For example,

```
every write("hello" | "howdy")
```

writes two lines, hello and howdy.

Since alternation succeeds if either of its arguments succeeds, it can be used to produce the effect of logical disjunction. An example is

```
if (i > j) | (j > k) then expr
```

which evaluates expr if i is greater than j or if j is greater than k.

Logical conjunction follows as a natural consequence of goal-directed evaluation. The operation

```
expr1 & expr2
```

is similar to other binary operations, such as *expr1 + expr2,* except that it performs no computation. Instead, it produces the result of *expr2,* provided that both *expr1* and *expr2* succeed. For example,

```
if (i > j) & (j > k) then expr
```

evaluates *expr* only if i is greater than j and j is greater than k.

Repeated alternation,

```
|expr
```

generates the results of *expr* repeatedly and is roughly equivalent to

```
expr | expr | expr | ...
```

However, if *expr* fails, the repeated alternation control structure stops generating results. For example,

```
|read(f)
```

generates the lines from the file f (one line for each repetition of the alternation) but stops when read(f) fails.

Note that a generator may be capable of producing an infinite number of results. For example,

```
|(1 to 3)
```

can produce 1, 2, 3, 1, 2, 3, 1, 2, 3, However, only as many results as are required by context are actually produced. Thus,

```
i := | (1 to 3)
```

only assigns the value 1 to i, since there is no context to cause the repeated alternation control structure to be resumed for a second result.

The *limitation* control structure

```
expr1 \ expr2
```

limits *expr1* to at most *expr2* results. Consequently,

```
| (1 to 3) \ 5
```

is only capable of producing 1, 2, 3, 1, 2.

The Order of Evaluation. With the exception of the limitation control structure, argument evaluation in Icon is strictly left-to-right. The resumption of expressions to produce additional results is in last-in, first-out order. The result is "cross-product" generation of results in expressions that contain several generators. For example,

```
every write((10 to 30 by 10) + (1 to 3))
```

writes 11, 12, 13, 21, 22, 23, 31, 32, 33.

Control Backtracking. Goal-directed evaluation results in control backtracking to obtain additional results from expressions that have previously produced results, as in

```
if find("e", "They sit like bumps on a log.") > 10
then write("found")
```

Control backtracking is limited by a number of syntactic constructions. For example, in

```
if expr1 then expr2 else expr3
```

if *expr1* succeeds, but *expr2* fails, *expr1* is not resumed for another result. (If it were, the semantics of this control structure would not correspond to what "if-then-else" suggests.) Such an expression is called a *bounded expression.* The control clauses of loops also are bounded, as are the expressions within compound expressions:

```
{ expr1; expr2; expr3; ...; exprn }
```

These expressions are evaluated in sequence, but once the evaluation of one is complete (whether it succeeds or fails), and the evaluation of another begins, there is no possibility of backtracking into the preceding one. The last expression in a compound expression is not bounded, however.

Except in such specific situations, expressions are not bounded. For example, in

```
if expr1 then expr2 else expr3
```

neither *expr2* nor *expr3* is bounded. Since Icon control structures are expressions that may return results, it is possible to write expressions such as

```
every write(if i > j then j to i else i to j)
```

which writes the integers from i to j in ascending sequence.

Data Backtracking. While control backtracking is a fundamental part of expression evaluation in Icon, data backtracking is not performed except in a few specific operations. For example, in

```
(i := 3) & read(f)
```

the value `3` is assigned to `i`. Even if `read(f)` fails, the former value of `i` is not restored.

There are, however, specific operations in which data backtracking is performed. For example, the *reversible assignment* operation

```
x <- y
```

assigns the value of y to x, but it restores the former value of x if control backtracking into this expression occurs. Thus,

```
(i <- 3) & read(f)
```

assigns 3 to i but restores the previous value of i if `read(f)` fails.

2.1.3 Csets and Strings

Csets are unordered sets of characters, while strings are sequences of characters. There are 256 different characters, the first 128 of which are interpreted as ASCII. The number and interpretation of characters is independent of the architecture of the computer on which Icon is implemented.

Csets. Csets are represented literally by surrounding their characters by single quotation marks. For example,

```
vowels := 'aeiouAEIOU'
```

assigns a cset of 10 characters to vowels.

There are several built-in csets that are the values of keywords. These include `&lcase`, `&ucase`, and `&cset`, which contain the lowercase letters, the uppercase letters, and all 256 characters, respectively.

Operations on csets include union, intersection, difference, and complement with respect to `&cset`. Csets are used in lexical analysis. For example, the function `upto(c, s)` is

analogous to `find(s1, s2)`, except that it generates the positions at which any character of `c` occurs in `s`. Thus,

```
upto(vowels, "They sit like bumps on a log.")
```

is capable of producing 3, 7, 11, 13, 16, 21, 24, and 27.

Strings. Strings are represented literally by surrounding their characters with double quotation marks instead of single quotation marks. The empty string, which contains no characters, is given by `""`. The size of a string is given by `*s`. For example, if

```
command := "Sit still!"
```

then the value of `*command` is 10. The value of `*""` is 0. Space for strings is provided automatically and there is no inherent limit to the size of a string.

There are several operations that construct strings. The principal one is concatenation, denoted by

```
s1 || s2
```

The function `repl(s, i)` produces the result of concatenating s i times. Thus,

```
write(repl("*!",3))
```

writes `*!*!*!`.

Other string construction functions include `reverse(s)`, which produces a string with the characters of `s` in reverse order, and `trim(s, c)`, which produces a string in which trailing characters of `s` that occur in `c` are omitted. There also are functions for positioning a string in a field of a fixed width. For example, the function `left(s1,i, s2)` produces a string of length `i` with `s1` positioned at the left and padded with copies of `s2` as needed.

Substrings are produced by subscripting a string with the beginning and ending positions of the desired substring. Positions in strings are between characters, and the position before the first character of a string is numbered 1. For example,

```
verb := command[1:4]
```

assigns the string `"Sit"` to `verb`. Substrings also can be specified by the beginning position and a length, as in

```
verb := command[1+:3]
```

If the length of a substring is 1, only the first position need be given, so that the value of `command[2]` is `"i"`.

Assignment can be made to a subscripted string to produce a new string. For example,

```
command[1:4] := "Remain"
```

changes the value of command to `"Remain still!"`.

String operations are applicative; no operation on a string in Icon changes the characters in it. The preceding example may appear to contradict this, but in fact

```
command[1:4] := "Remain"
```

is an abbreviation for

```
command := "Remain" || command[5:11]
```

Thus, a new string is constructed and then assigned to command.

Nonpositive values can be used to specify a position with respect to the right end of a string. For example, the value of `command[-1]` is `"!"`. The value 0 refers to the position after the last character of a string, so that if the value of command is `"Sit still!"`,

```
command[5:0]
```

is equivalent to

```
command[5:11]
```

The subscript positions can be given in either order. Thus,

```
command[11:5]
```

produces the same result as

```
command[5:11]
```

String-analysis functions like find and upto have optional third and fourth arguments that allow their range to be restricted to a particular portion of a string. For example,

```
upto(vowels, "They sit like bumps on a log.", 10, 20)
```

only produces positions of vowels between positions 10 and 20 of its second argument: 11, 13, and 16. If these arguments are omitted, they default to 1 and 0, so that the entire string is included in the analysis.

Mapping. One of the more interesting string-valued functions in Icon is `map(s1, s2, s3)`. This function produces a string obtained from a character substitution on s1. Each character of s1 that occurs in s2 is replaced by the corresponding character in s3. For example,

```
write(map("Remain still!", "aeiou", "*****"))
```

writes R*m**n St*ll!. Characters in s1 that do not appear in s2 are unchanged, as this example shows. If a character occurs more than once in s2, its right-most correspondence in s3 applies. Consequently,

```
s2 := &lcase || &ucase || "aeiou"
s3 := repl("|",26) || repl("u",26) || "*****"
write(map("Remain still!", s2, s3))
```

writes u*|**| ||*||!.

2.1.4 String Scanning

String scanning is a high-level facility for string analysis that suppresses the computational details associated with the explicit location of positions and substring specifications. In string scanning, a subject serves as a focus of attention. A position in this subject is maintained automatically.

A string-scanning expression has the form

```
expr1 ? expr2
```

in which the evaluation of *expr1* provides the subject. The position in the subject is 1 initially. The expression *expr2* is then evaluated in the context of this subject and position.

Although *expr2* can contain any operation, two *matching functions* are useful in analyzing the subject:

`tab(i)`	set the position in the subject to i
`move(i)`	increment the position in the subject by i

Both of these functions return the substring of the subject between the old and new positions. If the position is out of the range of the subject, the matching function fails and the position is not changed. The position can be increased or decreased. Nonpositive values can be used to refer to positions relative to the end of the subject. Thus, tab(0) moves the position to the end of the subject, matching the remainder of the subject.

An example of string scanning is

```
line ? while write(move(2))
```

which writes successive two-character substrings of line, stopping when there are not two characters remaining.

In string scanning, the trailing arguments of string analysis functions such as find and upto are omitted; the functions apply to the subject at the current position. Therefore, such functions can be used to provide arguments for matching functions. An example is

```
line ? write(tab(find("::=")))
```

which writes the initial portion of line up to an occurrence of the string "::=".

If a matching function is resumed, it restores the position in the subject to the value that it had before the matching function was evaluated. For example, suppose that line contains the substring "::=". Then

```
line ?
   ((tab(find("::=") + 3)) & write(move(10)) | write(tab(0)))
```

writes the 10 characters after "::=", provided there are 10 more characters. However, if there are not 10 characters remaining, move(10) fails and tab(find("::=")) is resumed. It restores the position to the beginning of the subject, and the alternative, tab(0), matches the entire subject, which is written.

Data backtracking of the position in the subject is important, since it allows matches to be performed with the assurance that any previous alternatives that failed to match left the position where it was before they were evaluated.

The subject and position are directly accessible as the values of the keywords &subject and &pos, respectively. For example,

```
&subject := "Hello"
```

assigns the string "Hello" to the subject. Whenever a value is assigned to the subject, &pos is set to 1 automatically.

The values of &subject and &pos are saved at the beginning of a string-scanning expression and are restored when it completes. Consequently, scanning expressions can be nested.

2.1.5 Lists

A list is a linear aggregate of values ("elements"). For example,

```
cities := ["Portland", "Toledo", "Tampa"]
```

assigns a list of three strings to cities. Lists can be heterogeneous, as in

```
language := ["Icon", 1978, "The University of Arizona"]
```

An empty list, containing no elements, is produced by []. The function

```
list(i, x)
```

produces a list of i elements, each of which has the value of x. The size operation *x also applies to lists. The value of *cities is 3, for example.

An element of a list is referenced by a subscripting expression that has the same form as the one for strings. For example,

```
cities[3] := "Miami"
```

changes the value of cities to

```
["Portland", "Toledo", "Miami"]
```

The function sort (a) produces a sorted copy of a. For example, sort(cities) produces

```
["Miami", "Portland", "Toledo"]
```

List operations, unlike string operations, are not applicative. While assignment to a substring is an abbreviation for concatenation, assignment to a subscripted list changes the value of the subscripted element.

A list value is a pointer to a structure that contains the elements of the list. Assignment of a list value copies this pointer, but it does not copy the structure. Consequently, in

```
states := ["Nevada", "Texas", "Maine", "Georgia"]
slist := states
```

both states and slist point to the *same* structure. Because of this,

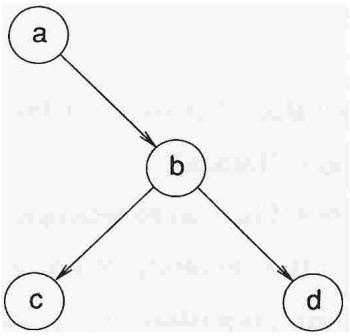
```
states[2] := "Arkansas"
```

changes the second element of slist as well as the second element of states.

The elements of a list may be of any type, including lists, as in

```
tree := ["a", ["b", ["c"], ["d"]]]
```

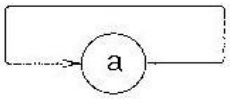
which can be depicted as



Structures also can be used to represent loops, as in

```
graph := ["a", ""]
graph[2] := graph
```

which can be depicted as



Lists are not fixed in size. Elements can be added to them or removed from them at their ends by queue and stack functions.

The function `put(a, x)` adds the value of `x` to the right end of the increasing its size by one. Similarly, `push(a, x)` adds the value of `x` to the left end of `a`. For example,

```
lines := []
while put(lines, read(f))
```

constructs a list of the lines from the file f. Conversely,

```
lines := []
while push(lines, read(f))
```

constructs a list of lines in reverse order.

The functions `pop(a)` and `get(a)` are the same. They both remove an element from the left end of a and return it as the value of the function call, but they fail if a is empty. Consequently,

```
lines := []
while push(lines, read(f))
while write(pop(lines))
```

writes out the lines of `f` in reverse order. The function `pull(a)` is similar, but it removes an element from the right end of `a`.

Other operations on lists include concatenation, which is denoted by

```
a1 ||| a2
```

where `a1` and `a2` are lists. There is no automatic conversion of other types to lists.

List sectioning is denoted by

```
a[i:j]
```

The result is a *new* list containing values `i` through `j` of `a`.

There is no inherent limit to the size of a list, either when it is originally created or as a result of adding elements to it.

2.1.6 Sets

A set is an unordered collection of values. Unlike csets, which contain only characters, sets are collections of Icon values that can be of any type. A set is constructed from a list by `set(a)`. For example,

```
states := set(["Virginia", "Rhode Island", "Kansas",
               "Illinois"])
```

assigns a set of four elements to `states`.

The operation

```
member(s, x)
```

succeeds if the value of `x` is a member of `s` but fails otherwise. The operation

```
insert(s, x)
```

adds the value of `x` to `s` if it is not already a member of `s`, while

```
delete(s, x)
```

deletes the value of `x` from `s`. The operations of union, intersection, and difference for sets also are provided.

Like other structures, sets can be heterogeneous. A set can even be a member of itself, as in

```
insert(s, s)
```

There is no contradiction here, since a set value is a pointer to the structure for the set.

2.1.7 Tables

A table is a set of pairs of values. Tables provide an associative look mechanism as contrasted with positional references to lists. They can be subscripted with an *entry value* to which a value can be assigned to make up a pair called a table element.

A table is created by

```
table(x)
```

Tables are empty initially. The value of x is an assigned default value that is produced if the table is subscripted with an entry value to which no value has been assigned (that is, for an element that is not in the table). For example,

```
states := table(0)
```

assigns to states a table with a default value of 0. An element can be added to states by an assignment such as

```
states["Oregon"] := 1
```

which adds a table element for `"Oregon"` with the value `1` to `states`. On the other hand,

```
write(states ["Utah"])
```

writes `0`, the default value, if there is no element in the table for `"Utah"`.

Tables can be heterogeneous and have a mixture of types for entry and assigned values. Tables grow automatically in size as new elements are added and there is no inherent limit on the size of a table.

2.1.8 Records

A record is an aggregate of values that is referenced by named fields. Each record type has a separate name. A record type and the names of its fields are given in a declaration. For example,

```
record rational(numerator, denominator)
```

declares a record of type rational with two fields: numerator and denominator.

An instance of a record is created by calling a record-constructor function corresponding to the form of the declaration for the record type. Thus,

```
r := rational(3,5)
```

assigns to `r` a record of type rational with a numerator field of 3 and a denominator field of 5. Fields are referenced by name, as in

```
write(r.numerator)
```

which writes 3. Fields can also be referred to by position; `r[1]` is equivalent to `r.numerator`.

There is no inherent limit to the number of different record types. The same field names can be given for different record types, and such fields need not be in the same position for all such record types.

2.1.9 Input and Output

Input and output in Icon are sequential and comparatively simple. The standard input, standard output, and standard error output files are the values of `&input`, `&output`, and `&errout`, respectively. The function

```
open(s1,s2)
```

opens the file whose name is `s1` according to options given by `s2` and produces a value of type file. Typical options are `"r"` for opening for reading and `"w"` for opening for writing. The default is `"r"`. For example,

```
log := open("grade.log", "w")
```

assigns a value of type file to log, corresponding to the data file grade.log, which is opened for writing. The function open fails if the specified file cannot be opened according to the options given. The function `close(f)` closes the file `f`.

The function `read(f)` reads a line from the file `f` but fails if an end of file is encountered. The default is standard input if `f` is omitted.

The result of

```
write(x1,x2, ..., xn)
```

depends on the types of x1, x2, ..., xn. Strings and types convertible to strings are written, but if one of the arguments is a file, subsequent strings are written to that file. The default file is standard output. Thus,

```
write(s1,s2)
```

writes the concatenation of `s1` and `s2` to standard output, but

```
write(log,s)
```

writes `s` to the file grade.log. In any event, write returns the string value of the last argument written.

The function

```
stop(x1, x2, ..., xn)
```

produces the same output as write, but it then terminates program execution.

2.1.10 Procedures

Procedure Declarations. The executable portions of an Icon program are contained in procedure declarations. Program execution begins with a call of the procedure main.

An example of a procedure declaration is:

```
procedure maxstr(slist)
   local max, value
   max := 0
   every value := *!slist do
      if value> max then max := value
   return max
end
```

This procedure computes the longest string in a list of strings. The formal parameter slist and the identifiers max and value are local to calls of the procedure `maxstr()`. Storage for them is allocated when `maxstr()` is called and deallocated when `maxstr()` returns.

A procedure call has the same form as a function call. For example,

```
lines := []
while put(lines, read(f))
write(maxstr(lines))
```

writes the length of the longest line in the file `f`.

A procedure call may fail to produce a result in the same way that a built-in operation can fail. This is indicated by fail in the procedure body in place of return. For example, the following procedure returns the length of the longest string in `slist` but fails if that length is less than limit:

```
procedure maxstr(slist, limit)
   local max, value
   max := 0
   every value := *!slist do
      if value> max then max := value
   if max < limit then fail else return max
end
```

Flowing off the end of a procedure body without an explicit return is equivalent to fail.

A procedure declaration may have static identifiers that are known only to calls of that procedure but whose values are not destroyed when a call returns. A procedure declaration also may have an initial clause whose expression is evaluated only the first time the procedure is called. The use of a static identifier and an initial clause is illustrated by the following procedure, which returns the longest of all the strings in the lists it has processed:

```
procedure maxstrall(slist)
   local value
   static max
   initial max := 0
   every value := *!slist do
      if value> max then max := value
   return max
end
```

Procedures and Functions. Procedures and functions are used in the same way. Their names have global scope. Other identifiers can be declared to have global scope, as in

```
global count
```

Such global declarations are on a par with procedure declarations and cannot occur within procedure declarations.

A call such as

```
write(maxstr(lines))
```

applies the *value* of the identifier `maxstr` to `lines` and applies the *value* of the identifier `write` to the result. There is nothing fixed about the values of such identifiers. In this case, the initial value of `maxstr` is a procedure, as a consequence of the procedure declaration for it. Similarly, the initial value of `write` is a function. These values can be assigned to other variables, as in

```
print := write
         ...
print(maxstr(lines))
```

in which the function that is the initial value of `write` is assigned to `print`.

Similarly, nothing prevents an assignment to an identifier whose initial value is a procedure. Consequently,

```
write := 3
```

assigns an integer to `write`, replacing its initial function value.

Although it is typical to call a procedure by using an identifier that has the procedure value, the procedure used in a call can be computed. The general form of a call is

expr₀(expr₁, expr₂, ..., exprₙ)

where the value of *expr₀* is applied to the arguments resulting from the evaluation of *expr₁ expr₂, ...*, *exprₙ*. For example,

(proclist[i])*(expr₁, , expr₂, ..., exprₙ)*

applies the procedure that is the `i`th element of `proclist`.

Procedures may be called recursively. The recursive nature of a call depends on the fact that procedure names are global. The "Fibonacci strings" provide an example:

```
procedure fibstr(i)
   if i = 1 then return "a"
   else if i = 2 then return "b"
   else return fibstr(i - 1) || fibstr(i - 2)
end
```

An identifier that is not declared in a procedure and is not global defaults to local. Thus, local declarations can be omitted, as in

```
procedure maxstr(slist)
   max := 0
   every value := * !slist do
      if value > max then max := value
   return max
end
```

Procedures as Generators. In addition to returning and failing, a procedure can also suspend. In this case, the values of its arguments and local identifiers are not destroyed, and the call can be resumed to produce another result in the same way a built-in generator can be resumed. An example of such a generator is

```
procedure intseq(i)
   repeat {
      suspend i
      i +:= 1
      }
end
```

A call `intseq(10)`, for example, is capable of generating the infinite sequence of integers 10, 11, 12, ... For example,

```
every f(intseq(10) \ 5)
```

calls f(10), f(11), f(12), f(13), and f(14).

If the argument of suspend is a generator, the generator is resumed when the call is resumed and the call suspends again with the result it produces. A generator of the Fibonacci strings provides an example:

```
procedure fibstrseq()
   local s1, s2, s3
   s1 := "a"
   s2 := "b"
   suspend (s1 | s2)
   repeat {
      suspend s3 := s1 || s2
      s1 := s2
      s2 := s3
```

```
        }
end
```

When this procedure is called, the first suspend expression produces the value of s1, "a". If the call of `fibstrseq()` is resumed, the argument of `suspend` is resumed and produces the value of `s2`, `"b"`. If the call is resumed again, there is no further result for the first `suspend`, and evaluation continues to the `repeat` loop.

Repeated alternation often is useful in supplying an endless number of alternatives. For example, the procedure `intseq(i)` can be rewritten as

```
procedure intseq(i)
   suspend i | (i +:= |1)
end
```

Note that `|1` is used to provide an endless sequence of increments.

Argument Transmission. Omitted arguments in a procedure or function call (including trailing ones) default to the null value. Extra arguments are evaluated, but their values are discarded.

Some functions, such as `write()`, may be called with an arbitrary number of arguments. All arguments to procedures and functions are passed by value. If the evaluation of an argument expression fails, the procedure or function is not called. This applies to extra arguments. Arguments are not dereferenced until all of them have been evaluated. Dereferencing cannot fail. Since no argument is dereferenced until all argument expressions are evaluated, expressions with side effects can produce unexpected results. Thus, in

```
write(s, s := "hello")
```

the value written is `hellohello`, regardless of the value of `s` before the evaluation of the second argument of `write()`.

Dereferencing in Return Expressions. The result returned from a procedure call is dereferenced unless it is a global identifier, a static identifier, a subscripted structure, or a subscripted string-valued global identifier.

In these exceptional cases, the variable is returned and assignment can be made to the procedure call. An example is

```
procedure maxel(a, i, j)
   if i > j then return a[i]
   else return a[j]
end
```

Here a list element, depending on the values of `i` and `j`, is returned. An assignment can be made to it, as in

```
maxel(lines, i, j) := "end"
```

which assigns `"end"` to `lines[i]` or `lines[j]`, depending on the values of `i` and `j`.

Mutual Evaluation. In a call expression, the value of $expr_0$ can be an integer `i` as well as a procedure. In this case, called *mutual evaluation,* the result of the `i`th argument is produced. For example,

```
i := 1(find(s1, line1), find(s2, line2))
```

assigns to `i` the position of `s1` in `line1`, provided `s1` occurs in `line1` and that `s2` occurs in line2. If either call of find fails, the expression fails and no assignment is made.

The selection integer in mutual evaluation can be negative, in which case it is interpreted relative to the end of the argument list. Consequently,

(-1)(expr₁,expr₂, ..., exprₙ)

produces the result of exprn and is equivalent to

expr1 & *expr2* & ... & *exprn*

The selection integer can be omitted, in which case it defaults to -1.

2.1.11 Co-Expressions

The evaluation of an expression in Icon is limited to the site in the program where it appears. Its results can be produced only at that site as a result of iteration or goal-directed evaluation. For example, the results generated by `intseq(i)` described in Section 2.1.10 can only be produced where it is called, as in

```
every f(intseq(10) \ 5)
```

It is often useful, however, to be able to produce the results of a generator at various places in the program as the need for them arises. Co-expressions provide this facility by giving a context for the evaluation of an expression that is maintained in a data structure. Co-expressions can be *activated* to produce the results of a generator on demand, at any time and place in the program.

A co-expression is constructed by

`create` *expr*

The expression *expr* is not evaluated at this time. Instead, an object is produced through which *expr* can be resumed at a later time. For example,

```
label := create ("L" || (1 to 100) || ":")
```

assigns to label a co-expression for the expression

```
"L" || (1 to 100) || ":"
```

The operation `@label` activates this co-expression, which corresponds to resuming its expression. For example,

```
write(@label)
write("    tstl     count")
write(@label)
```

writes

```
L1:
     tstl     count
L2:
```

If the resumption of the expression in a co-expression does not produce a result, the co-expression activation fails. For example, after `@label` has been evaluated 100 times, subsequent evaluations of `@label` fail. The number of results that a co-expression `e` has produced is given by `*e`.

The general form of the activation expression is

expr1 `@` *expr2*

which activates *expr2* and transmits the result of *expr1* to it. This form of activation can be used to return a result to the co-expression that activated the current one.

A co-expression is a value like any other value in Icon and can be passed as an argument to a procedure, returned from a procedure. and so forth. A co-expression can survive the call of the procedure in which it is created.

If the argument of a create expression contains identifiers that are local to the procedure in which the create occurs, copies of these local identifiers are included in the co-expression with the values they have at the time the create expression is evaluated. These copied identifiers subsequently are independent of the local identifiers in the procedure. Consider, for example,

```
procedure labgen(tag)
   local i, j
   ...
   i := 10
   j := 20
   e := create (tag || (i to j) || ":")
   ...
   i := j
   if i > 15 then return e
   ...
end
```

The expression

```
labels := labgen("X")
```

assigns to labels a co-expression that is equivalent to evaluating

```
create ("X" || (10 to 20) || ":")
```

The fact that `i` is changed after the co-expression was assigned to `e`, but before `e` returns, does not affect the co-expression, since it contains copies of `i` and `j` at the time it was created. Subsequent changes to the values of `i` or `j` do not affect the co-expression.

A copy of a co-expression `e` is produced by the *refresh* operation, `^e`. When a refreshed copy of a co-expression is made, its expression is reset to its initial state, and the values of any local identifiers in it are reset to the values they had when the co-expression was created. For example,

```
newlabels := ^labels
```

assigns to `newlabels` a co-expression that is capable of producing the same results as labels, regardless of whether or not labels has been activated.

The value of the keyword `&main` is the co-expression for the call of `main()` that initiates program execution.

2.1.12 Diagnostic Facilities

String Images. The function `type(x)` only produces the string name of the type of `x`, but the function `image(x)` produces a string that shows the value of `x`. For strings and csets, the value is shown with surrounding quotation marks in the fashion of program literals. For example,

```
write(image("Hi there!"))
```

writes `"Hi there!"`, while

```
write(image('aeiou'))
```

writes `'aeiou'`.

For structures, the type name and size are given. For example,

```
write(image([]))
```

writes `list(0)`.

Various forms are used for other types of data, using type names where necessary so that different types of values are distinguishable.

Tracing. If the value of the keyword `&trace` is nonzero, a trace message is produced whenever a procedure is called, returns, fails, suspends, or is resumed. Trace messages are written to standard error output. The value of `&trace` is decremented for every trace message. Tracing stops if the value of `&trace` becomes zero, which is its initial value. Suppose that the following program is contained in the file fibstr.icn:

```
procedure main()
   &trace := -1
   fibstr(3)
end

procedure fibstr(i)
   if i = 1 then return "a"
   else if i = 2 then return "b"
   else return fibstr(i -1) || fibstr(i -2)
end
```

The trace output of this program is

```
fibstr.icn: 3      | fibstr(3)
fibstr.icn: 9      | | fibstr(2)
fibstr.icn: 8      | | fibstr returned "b"
fibstr .icn: 9     | | fibstr(1)
fibstr.icn: 7      | | fibstr returned "b"
fibstr.icn: 9      | fibstr returned "ba"
fibstr.icn: 4     main failed
```

In addition to the indentation corresponding to the level of procedure call, the value of the keyword `&level` also is the current level of call.

Displaying Identifier Values. The function `display(i,f)` writes a list of all identifiers and their values for `i` levels of procedure calls, starting at the current level. If `i` is omitted, the default is `&level`, while if `f` is omitted, the list is written to standard error output. The format of the listing produced by display is illustrated by the following program:

```
procedure main()
   log := open("grade.log", "w")
   while write(log, check(read0))
end

procedure check(value)
   static count
   initial count := 0
   if numeric(value) then {
      count +:= 1
      return value
      }
   else {
      display()
      stop("nonnumeric value")
```

```
      }
end
```

Suppose that the tenth line of input is the nonnumeric string `"3.a"`. Then the output of `display()` is

```
check local identifiers:
   value = "3.a"
   count = 9
main local identifiers:
   log = file(grade.log)
global identifiers:
   main = procedure main
   check = procedure check
   open = function open
   write = function write
   read = function read
   numeric = function numeric
   display = function display
   stop = function stop
```

Error Messages. If an error is encountered during program execution, a message is written to standard error output and execution is terminated. For example, if the tenth line of a program contained in the file check.icn is

```
i +:= "x"
```

evaluation of this expression produces the error message

```
Run-time error 102 at line 10 in check.icn
numeric expected
offending value: "x"
```

2.2 Language Features and the Implementation

Even a cursory consideration of Icon reveals that some of its features present implementation problems and require approaches that are different from ones used in more conventional languages. In the case of a language of the size and complexity of Icon, it is important to place different aspects of the implementation in perspective and to identify specific problems.

Values and Variables. The absence of type declarations in Icon has far-reaching implications. Since any variable may have a value of any type and the type may change from time to time during program execution, there must be a way of representing values uniformly. This is a significant challenge in a language with a wide variety of types ranging from integers to co-expressions. Heterogeneous structures follow as a natural consequence of the lack of type declarations.

In one sense, the absence of type declarations simplifies the implementation: there is not much that can be done about types during program translation (compilation), and some of the work that is normally performed by conventional compilers can be avoided. The problems do not go away, however--they just move to another part of the implementation, since run-time type checking is required. Automatic type conversion according to context goes hand-in-hand with type checking.

Storage Management. Since strings and structures are created during program execution, rather than being declared, the space for them must be allocated as needed at run time. This implies, in turn, some mechanism for reclaiming space that has been allocated but

which is no longer needed--"garbage collection." These issues are complicated by the diversity of types and sizes of objects, the lack of any inherent size limitations, and the possibility of pointer loops in circular structures.

Strings. Independent of storage-management considerations, strings require special attention in the implementation. The emphasis of Icon is on string processing, and it is necessary to be able to process large amounts of string data sufficiently. Strings may be very long and many operations produce substrings of other strings. The repertoire of string analysis and string synthesis functions is large. All this adds up to the need for a well-designed and coherent mechanism for handling strings.

Structures. Icon's unusual structures, with sophisticated access mechanisms, also pose problems. In particular, structures that can change in size and can grow without limit require different implementation approaches than static structures of fixed size and organization.

The flexibility of positional, stack, and queue access mechanisms for lists requires compromises to balance efficient access for different uses. Sets of values with arbitrary types, combined with a range of set operations, pose non-trivial implementation problems. Tables are similar to sets, but require additional attention because of the implicit way that elements are added.

Procedures and Functions. Since procedures and functions are values, they must be represented as data objects. More significantly, the meaning of a function call cannot, in general, be determined when a program is translated. The expression write(s) may write a string or it may do something else, depending on whether or not write still has its initial value. Such meanings must, instead, be determined at run time.

Polymorphic Operations. Although the meanings of operations cannot be changed during program execution in the way that the meanings of calls can, several operations perform different computations depending on the types of their operands. Thus, x[i] may subscript a string, a list, or a table.

The meanings of some operations also depend on whether they occur in an assignment or a dereferencing context. For example, if s has a string value, assignment to s [i] is an abbreviation for a concatenation followed by an assignment to s, while if s[i] occurs in a context where its value is needed, it is simply a substring operation. Moreover, the context cannot, in general, be determined at translation time.

The way subscripting operations are specified in Icon offers considerable convenience to the programmer at the expense of considerable problems for the implementer.

Expression Evaluation. Generators and goal-directed evaluation present obvious implementation problems. There is a large body of knowledge about the implementation of expression evaluation for conventional languages in which expressions always produce a single result, but there is comparatively little knowledge about implementing expressions that produce results in sequence.

While there are languages in which expressions can produce more than one result, this capability is limited to specific contexts, such as pattern matching, or to specific control structures or data objects.

In Icon, generators and goal-directed evaluation are general and pervasive and apply to all evaluation contexts and to all types of data. Consequently, their implementation requires

a fresh approach. The mechanism also has to handle the use of failure to drive control structures and must support novel control structures, such as alternation and limitation. Efficiency is a serious concern, since whatever mechanism is used to implement generators is also used in conventional computational situations in which only one result is needed.

String Scanning. String scanning is comparatively simple. The subject and position--"state variables"--have to be saved at the beginning of string scanning and restored when it is completed. Actual string analysis and matching follow trivially from generators and goal-directed evaluation.

Co-Expressions. Co-expressions, which are only relevant because of the expression-evaluation mechanism of Icon, introduce a whole new set of complexities. Without co-expressions, the results that a generator can produce are limited to its site in the program. Control backtracking is limited syntactically, and its scope can be determined during program translation. With co-expressions, a generator in a state of suspension can be activated at any place and time during program execution.

RETROSPECTIVE: Icon has a number of unusual features that are designed to facilitate programming, and it has an extensive repertoire of string and structure operations. One of Icon's notable characteristics is the freedom from translation-time constraints and the ability to specify and change the meanings of operations at run time. This run-time flexibility is valuable to the programmer, but it places substantial burdens on the implementation---and also makes it interesting.

At the top level, there is the question of how actually to carry out some of the more sophisticated operations. Then there are questions of efficiency, both in execution speed and storage utilization. There are endless possibilities for alternative approaches and refinements.

It is worth noting that many aspects of the implementation are relatively independent of each other and can be approached separately. Operations on strings and structures are largely disjoint and can, except for general considerations of the representation of values and storage management, be treated as independent problems.

The independence of expression evaluation from other implementation considerations is even clearer. Without generators and goal-directed evaluation, Icon would be a fairly conventional high-level string and structure processing language, albeit one with interesting implementation problems. On the other hand, generators and goal-directed evaluation are not dependent in any significant way on string and structure data types. Generators, goal-directed evaluation, and related control structures could just as well be incorporated in a programming language emphasizing numerical computation. The implementation problems related to expression evaluation in the two contexts are largely the same.

While untyped variables and automatic storage management have pervasive effects on the overall implementation of Icon, there are several aspects of Icon that are separable from the rest of the language and its implementation. Any specific data structure, string scanning, or co-expressions could be eliminated from the language without significantly affecting the rest of the implementation. Similarly, new data structures and new access mechanisms could be added without requiring significant modifications to the balance of the implementation.

EXERCISES

2.1 What is the outcome of the following expression if the file f contains a line consisting of "end", or if it does not?

```
while line := read(f) do
     if line == "end" then break
     else write(line)
```

2.2 What does

```
write("hello" | "howdy")
```

write?

2.3 What is the result of evaluating the following expression:

```
1(1 to 3) > 10
```

2.4 Explain the rationale for dereferencing of variables when a procedure call returns.

2.5 Give an example of a situation in which it cannot be determined until run time whether a string subscripting expression is used in an assignment or a dereferencing context.

Chapter 3: Organization of the Implementation

PERSPECTIVE: Many factors influence the implementation of a programming language. The properties of the language itself, of course, are of paramount importance. Beyond this, goals, resources, and many other factors may affect the nature of an implementation in significant and subtle ways.

In the case of the implementation of Icon described here, several unusual factors deserve mention. To begin with, Icon's origins were in a research project, and its implementation was designed not only to make the language available for use but also to support further language development. The language itself was less well defined and more subject to modification than is usually the case with an implementation. Therefore, flexibility and ease of modification were important implementation goals.

Although the implementation was not a commercial enterprise, neither was it a toy or a system intended only for a few '"friendly users." It was designed to be complete, robust, easy to maintain, and sufficiently efficient to be useful for real applications in its problem domain.

Experience with earlier implementations of SNOBOL4, SL5, and the Ratfor implementation of Icon also influenced the implementation that is described here. They provided a repertoire of proven techniques and a philosophy of approach to the implementation of a programming language that has novel features.

The computing environment also played a major role. The implementation started on a PDP-11/70 running under UNIX. The UNIX environment (Ritchie and Thompson 1978), with its extensive range of tools for program development, influenced several aspects of the implementation in a direct way. C (Kernighan and Ritchie 1978) is the natural language for writing such an implementation under UNIX, and its use for the majority of Icon had pervasive effects, which are described throughout this book. Tools, such as the Yacc parser-generator (Johnson 1975), influenced the approach to the translation portion of the implementation.

Since the initial work was done on a PDP-11/70, with a user address space of only l28K bytes (combined instruction and data spaces), the size of the implementation was a significant concern. In particular, while the Ratfor implementation of Icon fit comfortably on computers with large address spaces, such as the DEC-10, CDC Cyber, and IBM 370, this implementation was much too large to fit on a PDP-11/70.

3.1 The Icon Virtual Machine

The implementation of Icon is organized around a virtual machine (Newey, Poole, and Waite 1972; Griswold 1977). Virtual machines, sometimes called abstract machines, serve as software design tools for implementations in which the operations of a language do not fit a particular computer architecture or where portability is a consideration and the attributes of several real computer architectures can be abstracted in a single common model. The expectation for most virtual machine models is that a translation will be performed to map the virtual machine operations onto a specific real machine. A virtual machine also provides a basis for developing an operational definition of a programming language in which details can be worked out in concrete terms.

During the design and development phases of an implementation, a virtual machine serves as an idealized model that is free of the details and idiosyncrasies of any real machine. The virtual machine can be designed in such a way that treatment of specific, machine-dependent details can be deferred until it is necessary to translate the implementation of the virtual machine to a real one.

Icon's virtual machine only goes so far. Unlike the SNOBOL4 virtual machine (Griswold 1972), it is incomplete and characterizes only the expression-evaluation mechanism of Icon and computations on Icon data. It does not, *per se,* include a model for the organization of memory. There are many aspects of the Icon run-time system, such as type checking, storage allocation and garbage collection, that are not represented in the virtual machine. Instead Icon's virtual machine serves more as a guide and a tool for organizing the implementation than it does as a rigid structure that dominates the implementation.

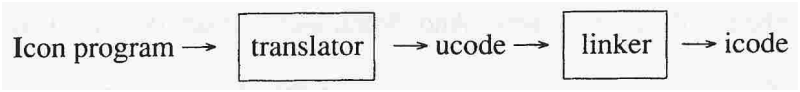
3.2 **Components of the Implementation**

There are three major components of the virtual machine implementation of Icon: a translator, a linker, and a run-time system. The translator and linker are combined to form a single executable program, but they remain logically independent.

The translator plays the role of a compiler for the Icon virtual machine. It analyzes source programs and converts them to virtual machine instructions. The output of the translator is called *ucode.* Ucode is represented as ASCII which is helpful in debugging the implementation.

The linker combines one or more ucode files into a single program for the virtual machine. This allows programs to be written and translated in a number of modules, and it is particularly useful for giving users access to pretranslated libraries of Icon procedures. The output of the linker, called *icode,* is in binary format for compactness and ease of processing by the virtual machine. Ucode and icode instructions are essentially the same, differing mainly in their format.

Translating and linking are done in two phases:



These phases can be performed separately. If only the first phase is performed, the result is ucode, which can be saved and linked at another time.

The run-time system consists of an interpreter for icode and a library of support routines to carry out the various operations that may occur when an Icon program is executed. The interpreter serves, conceptually, as a software realization of the Icon virtual machine. It decodes icode instructions and their operands and carries out the corresponding operations.

It is worth noting that the organization of the Icon system does not depend in any essential way on the use of an interpreter. In fact, in the early versions of this implementation, the linker produced assembly-language code for the target machine. That code then was assembled and loaded with the run-time library. On the surface, the generation of machine code for a specific target machine rather than for a virtual machine corresponds to the conventional compilation approach. However, this is somewhat of an

illusion, since the machine code consists largely of calls to run-time library routines corresponding to virtual machine instructions. Execution of machine code in such an implementation therefore differs only slightly from interpretation, in which instruction decoding is done in software rather than in hardware. The difference in speed in the case of Icon is relatively minor.

An interpreter offers a number of advantages over the generation of machine code that offset the small loss of efficiency. The main advantage is that the interpreter gets into execution very quickly, since it does not require a loading phase to resolve assembly-language references to library routines. Icode files also are much smaller than the executable binary files produced by a loader, since the run-time library does not need to be included in them. Instead, only one sharable copy of the run-time system needs to be resident in memory when Icon is executing.

3.3 The Translator

The translator that produces ucode is relatively conventional. It is written entirely in C and is independent of the architecture of the target machine on which Icon runs. Ucode is portable from one target machine to another.

The translator consists of a lexical analyzer, a parser, a code generator, and a few support routines. The lexical analyzer converts a source-language program into a stream of tokens that are provided to the parser as they are needed. The parser generates abstract syntax trees on a per-procedure basis. These abstract syntax trees are in turn processed by the code generator to produce ucode. The parser is generated automatically by Yacc from a grammatical specification. Since the translator is relatively conventional and the techniques that it uses are described in detail elsewhere (Aho, Lam, Sethi, and Ullman 2006), it is not discussed here.

There is one aspect of lexical analysis that deserves mention. The body of an Icon procedure consists of a series of expressions that are separated by semicolons. However, these semicolons usually do not need to be provided explicitly, as illustrated by examples in Chapter 2. Instead, the lexical analyzer performs semicolon insertion. If a line of a program ends with a token that is legal for ending an expression, and if the next line begins with a token that is legal for beginning an expression, the lexical analyzer generates a semicolon token between the lines. For example, the two lines

```
i := j + 3
write(i)
```

are equivalent to

```
i := j + 3;
write(i)
```

since an integer literal is legal at the end of an expression and an identifier is legal at the beginning of an expression.

If an expression spans two lines, the place to divide it is at a token that is not legal at the end of a line. For example,

```
s1 := s2 ||
   s3
```

is equivalent to

```
s1 := s2 || s3
```

No semicolon is inserted, since | | is not legal at the end of an expression.

3.4 The Linker

The linker reads ucode files and writes icode files. An icode file consists of an executable header that loads the run-time system, descriptive information about the file, operation codes and operands, and data specific to the program. The linker, like the translator, is written entirely in C. While conversion of ucode to icode is largely a matter of reformatting, the linker performs two other functions.

3.4.1 Scope Resolution

The scope of an undeclared identifier in a procedure depends on global declarations (explicit or implicit) in the program in which the procedure occurs. Since the translator in general operates on only one module of a program, it cannot resolve the scope of undeclared identifiers, because not all global scope information is contained in any one module. The linker, on the other hand, processes all the modules of a program, and hence it has the task of resolving the scope of undeclared identifiers.

An identifier may be global for several reasons:

- As the result of an explicit global declaration.
- As the name in a record declaration.
- As the name in a procedure declaration.
- As the name of a built-in function.

If an identifier with no local declaration falls into one of these categories, it is global. Otherwise it is local.

3.4.2 Construction of Run-Time Structures

A number of aspects of a source-language Icon program are represented at run time by various data structures. These structures are described in detail in subsequent chapters. They include procedure blocks, strings, and blocks for cset and real literals that appear in the program.

This data is represented in ucode in a machine-independent fashion. The linker converts this information into binary images that are dependent on the architecture of the target computer.

3.5 The Run-Time System

Most of the interesting aspects of the implementation of Icon reside in its run-time system. This run-time system is written mostly in C, although there are a few lines of assembly-language code for checking for arithmetic overflow and for co-expressions. The C portion is mostly machine-independent and portable, although some machine-specific code is needed for some idiosyncratic computer architectures and to interface some operating-system environments.

There are two main reasons for concentrating the implementation in the run-time system:

- Some features of Icon do not lend themselves to translation directly into executable code for the target machine, since there is no direct image for them in the target-machine architecture. The target machine code necessary to carry out these operations

therefore is too large to place in line; instead, it is placed in library routines that are called from in-line code. Such features range from operations on structures to string scanning.

- Operations that cannot be determined at translation time must be done at run time. Such operations range from type checking to storage allocation and garbage collection.

The run-time system is logically divided into four main parts: initialization and termination routines, the interpreter, library routines called by the interpreter, and support routines called by library routines.

Initialization and Termination Routines. The initialization routine sets up regions in which objects created at run time are allocated. It also initializes some structures that are used during program execution. Once these tasks are completed, control is transferred to the Icon interpreter.

When a program terminates, either normally or because of an error, termination routines flush output buffers and return control to the operating system.

The Interpreter. The interpreter analyzes icode instructions and their operands and performs corresponding operations. The interpreter is relatively simple, since most complex operations are performed by library routines. The interpreter itself is described in Chapter 8.

Library Routines. Library routines are divided into three categories, depending on the way they are called by the interpreter: routines for Icon operators, routines for Icon built-in functions, and routines for complicated virtual machine instructions.

The meanings of operators are known to the translator and linker, and hence they can be called directly. On the other hand, the meanings of functions cannot be determined until they are executed, and hence they are called indirectly.

Support Routines. Support routines include storage allocation and garbage collection, as well as type checking and conversion. Such routines typically are called by library routines, although some are called by other support routines.

RETROSPECTIVE: Superficially, the implementation of Icon appears to be conventional. An Icon program is translated and linked to produce an executable binary file. The translator and linker *are* conventional, except that they generate code and data structures for a virtual machine instead of for a specific computer.

The run-time system dominates the implementation and plays a much larger role than is played by run-time systems in conventional implementations. This run-time system is the focus of the remainder of this book.

EXERCISES

3.1 Explain why there is only a comparatively small difference in execution times between a version of Icon that generates assembly-language code and one that generates virtual machine code that is interpreted.

3.2 List all the tokens in the Icon grammar that are legal as the beginning of an expression and as the end of an expression. Are there any tokens that are legal as both? As neither?

3.3 Is a semicolon inserted by the lexical analyzer between the following two program lines?

```
s1 := s2
      || s3
```

3.4 Is it possible for semicolon insertion to introduce syntactic errors into a program that would be syntactically correct without semicolon insertion?

Chapter 4: Values and Variables

PERSPECTIVE: No feature of the Icon programming language has a greater impact on the implementation than untyped variables-variables that have no specific type associated with them. This feature originated in Icon's predecessors as a result of a desire for simplicity and flexibility.

The absence of type declarations reduces the amount that a programmer has to learn and remember. It also makes programs shorter and (perhaps) easier to write. The flexibility comes mainly from the support for heterogeneous aggregates. A list, for example, can contain a mixture of strings, integers, records, and other lists. There are numerous examples of Icon programs in which this flexibility leads to programming styles that are concise and simple. Similarly, "generic" procedures, whose arguments can be of any type, often are useful, especially for modeling experimental language features.

While these facilities can be provided in other ways, such as by C's union construct, Icon provides them by the *absence* of features, which fits with the philosophy of making it easy to write good programs rather than hard to write bad ones.

The other side of the coin is that the lack of type declarations for variables makes it impossible for the translator to detect most type errors and defers type checking until the program is executed. Thus, a check that can be done only once at translation time in a language with a strong compile-time type system must be done repeatedly during program execution in Icon. Furthermore, just as the Icon translator cannot detect most type errors, a person who is writing or reading an Icon program does not have type declarations to help clarify the intent of the program.

Icon also converts arguments to the expected type where possible. This feature is, nevertheless, separable from type checking; Icon could have the latter without the former. However, type checking and conversion are naturally intertwined in the implementation.

As far as the implementation is concerned, untyped variables simplify the translator and complicate the run-time system. There is little the translator can do about types. Many operations are polymorphic, taking arguments of different types and sometimes performing significantly different computations, depending on those types. Many types are convertible to others. Since procedures are data values and may change meaning during program execution, there is nothing the translator can know about them. For this reason, the translator does not attempt any type checking or generate any code for type checking or conversion. All such code resides in the run-time routines for the functions and operations themselves.

There is a more subtle way in which untyped variables influence the implementation. Since any variable can have any type of value at any time, and can have different types of values at different times, all values must be the same size. Furthermore, Icon's rich repertoire of data types includes values of arbitrary size-lists, tables, procedures, and so on.

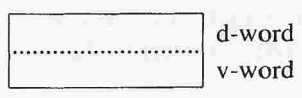
The solution to this problem is the concept of a *descriptor,* which either contains the data for the value, if it is small enough, or else contains a pointer to the data if it is too large to fit into a descriptor. The trick, then, is to design descriptors for all of Icon's data types,

balancing considerations of size, ease of type testing, and efficiency of accessing the actual data.

4.1 Descriptors

Since every Icon value is represented by a descriptor, it is important that descriptors be as small as possible. On the other hand, a descriptor must contain enough information to determine the type of the value that it represents and to locate the actual data. Although values of some types cannot possibly fit into any fixed-size space, it is desirable for frequently used, fixed-sized values, such as integers, to be stored in their descriptors. This allows values of these types to be accessed directly and avoids the need to provide storage elsewhere for such values.

If Icon were designed to run on only one kind of computer, the size and layout of the descriptor could be tailored to the architecture of the computer. Since the implementation is designed to run on a wide range of computer architectures, Icon takes an approach similar to that of C. Its descriptor is composed of "'words," which are closely related to the concept of a word on the computer on which Icon is implemented. One word is not large enough for a descriptor that must contain both type information and an integer or a pointer. Therefore, a descriptor consists of two words, which are designated as the *d-word* and the v-*word,* indicating that the former contains descriptive information, while the latter contains the value



The dotted line between the two words of a descriptor is provided for readability. A descriptor is merely two words, and the fact that these two words constitute a descriptor is a matter of context.

The v-word of a descriptor may contain either a value, such as an integer, or a pointer to other data. In C terms. the v-word may contain a variety of types, including both ints and pointers. On many computers, C ints and C pointers are the same size. For some computers, however, C compilers have a memory-model in which integers are smaller than pointers, which must allow access to a large amount of memory. In this situation, the C `long` or `long long` type are the same size as C pointers. There are computers with many different word sizes, but the main considerations in the implementation of Icon are the accommodation of computers with 32- and 64-bit words and the large-memory model, in which pointers are larger than integers. In the large-memory model, a v-word must accommodate the largest of the types.

The d-words of descriptors contain a type code (a small integer) in their least significant bits and flags in their most significant bits. There are twelve type codes that correspond to source-language data types:

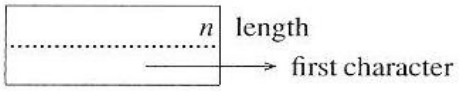
data type	*type code*
null	null
integer	integer or long
real number	real
cset	cset
file	file

procedure	proc
list	list
set	set
table	table
record	record
co-expression	coexpr

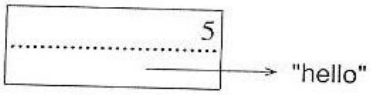
Other type codes exist for internal objects, which are on a par with source-language objects, from an implementation viewpoint, but which are not visible at the source-language level. The actual values of these codes are not important, and they are indicated in diagrams by their type code names.

4.1.1 Strings

There is no type code for strings. They have a special representation in which the d-word contains the length of the string (the number of characters in it) and the v-word points to the first character in the string:



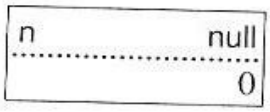
String descriptors are called *qualifiers.* In order to make qualifiers more intelligible in the diagrams that follow, a pointer to a string is followed by the string in quotation marks rather than by an address. For example, the qualifier for `"hello"` is depicted as



In order to distinguish qualifiers from other descriptors with type codes that might be the same as a string length, all descriptors that are not qualifiers have an n flag in the most significant bit of the d-word. The d-words of qualifiers do not have this n flag, and string lengths are restricted to prevent their overflow into this flag position, the most significant bit of a 32- or 64-bit dword.

4.1.2 The Null Value

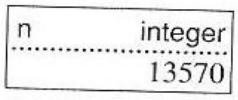
A descriptor for the null value has the form



As explained previously, the n flag occurs in this and all other descriptors that are not qualifiers so that strings can be easily and unambiguously distinguished from all other kinds of values. The value in the v-word could be any constant value, but zero is useful and easily identified---and suggests "null."

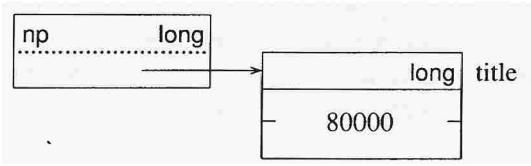
4.1.3 Integers

Icon supports word-size integers at least 32-bits in size. Such integers therefore are typically C longs, depending on the computer architecture. As long as it fits, the value of an Icon integer is stored in the v-word of its descriptor. For example, the integer 13570 is represented by



Note that the n flag distinguishes this descriptor from a string whose first character might be at the address 13570 and whose length might have the same value as the type code for integer.

An Icon integer that fits in the v-word is stored there. An integer that is too large to fit into a word is stored in a data structure that is pointed to by the v-word, as illustrated in the next section. The two representations of integers are distinguished by different internal type codes: integer for integers that are contained in the v-words of their descriptors and lrgint for integers that are contained in blocks pointed to by the v-words of their descriptors. Thus, there are two internal types for one source-language data type.



The p flag in the descriptor indicates that the v-word contains a pointer to a block.

Blocks of some other types, such as record blocks, vary in size from value to value, but any one block is fixed in size and never grows or shrinks. If the type code in the title does not determine the size of the block, the second word in the block contains its size in bytes. In the diagrams that follow, the sizes of blocks are given for computers with 32-bit words. The diagrams would be slightly different for computers with 16-bit words.

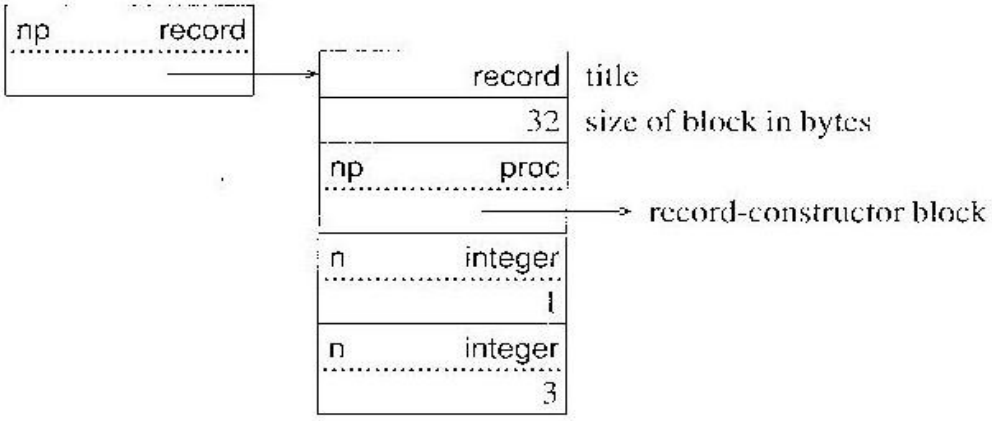
Records, which differ in size depending on how many fields they have, are examples of blocks that contain their sizes. For example, given the record declaration

```
record complex(r, i)
```

and

```
point := complex(1, 3)
```

the value of point is



The record-constructor block contains information that is needed to resolve field references.

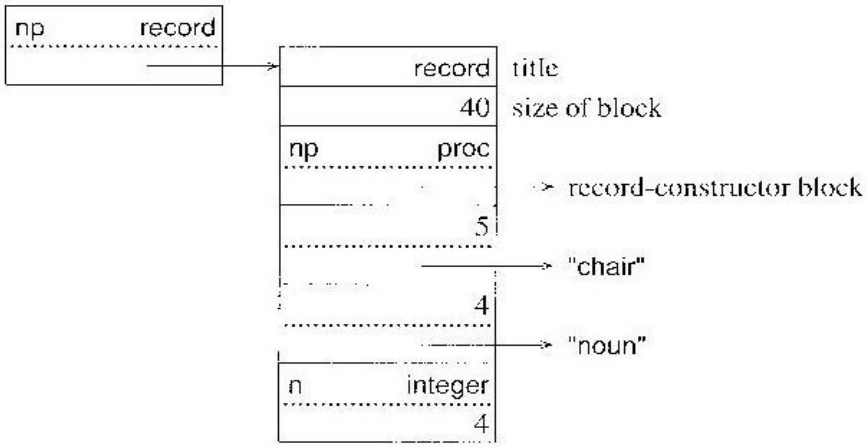
On the other hand, with the declaration

```
record term(value, code, count)
```

and

```
word := term("chair", "noun", 4)
```

the value of word is:



As illustrated by these examples, blocks may contain descriptors as well as non-descriptor data. Non-descriptor data comes first in the block, followed by any descriptors, as illustrated by the preceding figure. The location of the first descriptor in a block is constant for all blocks of a given type, which facilitates garbage collection.

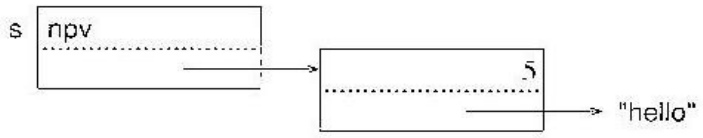
Blocks for the remaining types are described in subsequent chapters.

4.3 Variables

Variables are represented by descriptors, just as values are. This representation allows values and variables to be treated uniformly in terms of storage and access. Variables for identifiers point to descriptors for the corresponding values. Variables always point to descriptors for values, never to other variables. For example, if

```
s := "hello"
```

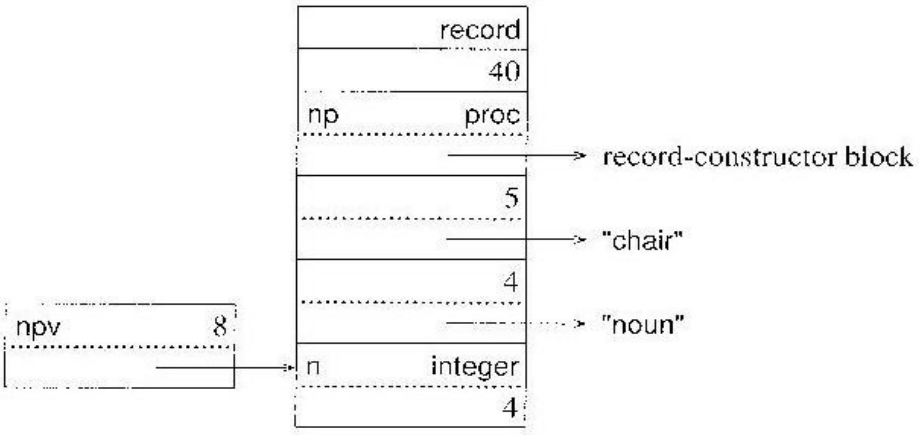
then a variable for `s` has the form



The v flag distinguishes descriptors for variables from descriptors for values.

The values of local identifiers are kept on a stack, while the values of global and static identifiers are located at fixed places in memory. Variables that point to the values of identifiers are created by icode instructions that correspond to the use of the identifiers in the program.

Some variables, such as record field references, are computed. A variable that references a value in a data structure points directly to the descriptor for the value. The least-significant bits of the d-word for such a variable contain the offset, in *words,* of the value descriptor from the top of the block in which the value is contained. This offset is used by the garbage collector. The use of words, rather than bytes, allows larger offsets, which is important for computers with 16-bit words. For example, the variable word.count for the record given in the preceding section is

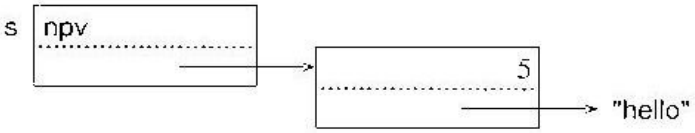


The variable points directly to the value rather than to the title of the block so that access to the value is more efficient. Note that the variable `word.count` cannot be determined at translation time, since the type of word is not known then and different record types could have count fields in different positions.

4.3.1 Operations on Variables

There are two fundamentally different contexts in which a variable can be used: *dereferencing* and *assignment.*

Suppose, as shown previously, that the value of the identifier s is the string "hello". Then a variable descriptor that points to the value of s and the corresponding value descriptor for "hello" have the following relationship:

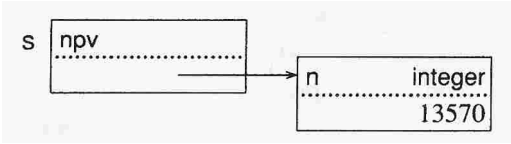


In an expression such as `write(s)`, `s` is dereferenced by fetching the descriptor pointed to by the v-word of the variable.

In the case of assignment, as in

```
s := 13570
```

the value descriptor pointed to by the v-word of the variable descriptor changed:



These operations on variables correspond to indirect load and store instructions of a typical computer.

4.3.2 Trapped Variables

Icon has several variables with special properties that complicate assignment and dereferencing. Consider, for example, the keyword `&trace`. Its value must always be an integer. Consequently, in an assignment such as

```
&trace := expr
```

the value produced by *expr* must be checked to be sure that it is an integer. If it is not, an attempt is made to convert it to an integer, so that in

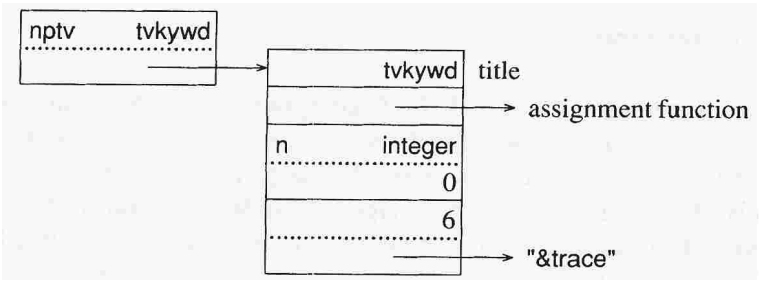
```
&trace := "1"
```

the value assigned to `&trace` is the integer `1`, not the string `"1"`.

There are four keyword variables that require special processing for assignment: `&trace`, `&random`, `&subject`, and `&pos`. The keyword `&random` is treated in essentially the same way that `&trace` is. Assignment to `&subject` requires a string value and has the side effect of assigning the value `1` to `&pos`. Assignment to `&pos` is even more complicated: not only must the value assigned be an integer, but if it is not positive, it must also be converted to the positive equivalent with respect to the length of `&subject`. In any event, if the value in the assignment to `&pos` is not in the range of `&subject`, the assignment fails. Dereferencing these keywords, on the other hand, requires no special processing.

A naive way to handle assignment to these keywords is to check every variable during assignment to see whether it is one of the four that requires special processing. This would place a significant computational burden on every assignment. Instead, Icon divides variables into two classes: *ordinary* and *trapped.* Ordinary variables point to their values as illustrated previously and require no special processing. Trapped variables, so called because their processing is "trapped," are distinguished from ordinary variables by a t flag. Thus, assignment only has to check a single flag to separate the majority of variables from those that require special processing.

A trapped-variable descriptor for a keyword points to a block that contains the value of the keyword, its string name, and a pointer to a C function that is called when assignment to the keyword is made. For example, the trapped variable for `&trace` is:



It is worth noting that the more conventional approach to handling the problem of assignment to keywords is to compile special code if a keyword occurs an assignment context. It is not always possible, however, to determine the context in which a variable is used in Icon. Consider a procedure of the form

```
procedure diagnose(s)
   return &trace
end
```

The semantics of Icon dictate that the result returned in this case should be a variable, not just its value, so that it is possible to write an expression such as

```
diagnose(s) := 10
```

which has the effect of assigning the value 10 to `&trace`.

The translator has no way of knowing that an assignment to the call diagnose(s) is equivalent to an assignment to `&trace`. In fact, the translator cannot even determine that the value of diagnose will be a function when the previous assignment is performed, much less that it will be the procedure given earlier.

Thus, the trapped-variable mechanism provides a way to handle uniformly all the situations in which such a keyword can be used.

4.4 Descriptors and Blocks in C

Descriptors and blocks of data are described and depicted abstractly in the previous sections of this chapter. In order to understand the implementation of some aspects of Icon, it is helpful to examine the C code that actually defines and manipulates data.

The following sections illustrate typical C declarations for the structures used in the implementation of Icon. Some of the terminology and operations that appear frequently in the C code are included as well. Other operations are introduced in subsequent chapters. as they are needed.

4.4.1 Descriptors

As mentioned in Sec. 4.1, for C compilers in which ints and pointers are the same size, the size of a word is the size of an int, while if pointers are larger than ints, the size of a word is the size of a long, or a long long. The difference between these models of memory is handled by typedefs under the control of conditional compilation. Two constants that characterize the sizes are defined: IntBits and WordBits. These sizes are used to select appropriate definitions for signed and unsigned words. The fact that on some 64-bit C compilers a `long` is only 32 bits, while on others it is 64 bits, complicates matters. The symbol `LongLongWord` indicates this situation.

```
#if IntBits != WordBits
   #ifdef LongLongWord
      typedef long word;
      typedef unsigned long uword;
   #else
      typedef long word;
      typedef unsigned long uword;
   #endif
#else
typedef int word;
typedef unsigned int uword;
#endif
```

A descriptor is declared as a structure:

```
struct descrip {   /* descriptor */
   word dword;     /*    type field */
   union {
      word integr; /*    integer value */
#ifdef DescriptorDouble
      double realval;
#endif
      char *sptr;  /*    pointer to character string */
      union block *bptr;      /*    pointer to a block */
      struct descrip *dptr;   /*    pointer to a descriptor */
   } vword;
};
```

The v-word of a descriptor is a union that reflects its various uses: an integer, a pointer to a string, a pointer to a block, or a pointer to another descriptor (in the case of a variable).

4.4.2 Blocks

Each block type has a structure declaration. For example. the declaration for record blocks is

```
struct b_record {          /* record block */
   word title;                /*    T_Record */
   word blksize;              /*    size of block */
   struct descrip recdesc; /* record constructor descriptor */
   struct descrip fields[1];/*     fields */
};
```

Blocks for records vary in size, depending on the number of fields declared for the record type. The size of 1 in

```
struct descrip fields[1];
```

is provided to satisfy the C compiler. Actual blocks for records are constructed at run time in a region that is managed by Icon's storage allocator. Such blocks conform to the previous declaration, but the number of fields varies. The declaration provides a means of accessing portions of such blocks from C.

The declaration for keyword trapped-variable blocks is

```
struct b_tvkywd {       /* keyword trapped variable block */
   word title;          /*    T Tvkywd */
   int (*putval) ();    /*    assignment function */
   struct descrip kyval;      /*    keyword value */
   struct descrip kyname;/*   keyword name */
};
```

Note that the title fields of `b_record` and `b_tvkywd` contain type codes, as indicated in previous diagrams. The second field of `b_record` is a size as mentioned previously, but `b_tvkywd` has no size field, since all keyword trapped-variable blocks are the same size, which therefore can be determined from their type.

The block union given in the declaration of `struct descrip` consists of a union of all block types:

```
union block {
   struct b_real realblk;
   struct b_cset cset;
   struct b_file file;
   struct b_proc proc;
   struct b_list list;
   struct b_lelem lelem;
   struct b_table table;
   struct b_telem telem;
   struct b_set set;
   struct b_selem selem;
   struct b_record record;
   struct b_tvsubs tvsubs;
   struct b_tvtbl tvtbl;
   struct b_refresh refresh;
   struct b_coexpr coexpr;
   struct b_externl externl;
   struct b_slots slots;
   struct b_bignum bignumblk;
   };
```

Note that there are several kinds of blocks in addition to those that correspond to source-language data types.

4.4.3 Defined Constants

The type codes are defined symbolically:

```
#define T_Null          0
#define T_Integer       1
#define T_Lrgint        2
#define T_Real          3
#define T_Cset          4
#define T_File          5
#define T_Proc          6
#define T_Record        7
#define T_List          8
#define T_Lelem         9
#define T_Set           10
#define T_Selem         11
#define T_Table         12
#define T Telem         13
#define T_Tvtbl         14
#define T_Slots         15
#define T_Tvsubs        16
#define T_Refresh       17
#define T_Coexpr        18
#define T_External      19
#define T_Kywdint       20
#define T_Kywdpos       21
#define T_Kywdsubj      22
```

The type codes in diagrams are abbreviated, as indicated by previous examples.

The defined constants for d-word flags are

```
n       F_Nqual
p       F_Ptr
v       F_Var
t       F_Tvar
```

The values of these flags depend on the word size of the computer.

The d-words of descriptors are defined in terms of flags and type codes:

```
#define D_Null          (T_Null | F_Nqual)
#define D_Integer       (T_Integer | F_Nqual)
#define D_Long          (T_Long | F_Ptr | F_Nqual)
#define D_Real          (T_Real | F_Ptr | F_Nqual)
#define D_Cset          (T_Cset | F_Ptr | F_Nqual)
#define D_File          (T_File | F_Ptr | F_Nqual)
#define D_Proc          (T_Proc | F_Ptr | F_Nqual)
#define D_List          (T_List | F_Ptr | F_Nqual)
#define D_Table         (T_Table | F_Ptr | F_Nqual)
#define D_Set           (T_Set | F_Ptr | F_Nqual)
#define D_Selem         (T_Selem | F_Ptr | F_Nqual)
#define D_Record        (T_Record | F_Ptr | F_Nqual)
#define D_Telem         (T_Telem | F_Ptr | F_Nqual)
#define D_Lelem         (T_Lelem | F_Ptr | F_Nqual)
#define D_Tvsubs        (T_Tvsubs | D_Tvar)
#define D_Tvtbl         (T Tvtbl | D_Tvar)
#define D_Tvkywd        (T_Tvkywd | D_Tvar)
#define D_Coexpr        (T_Coexpr | F_Ptr | F_Nqual)
#define D_Refresh       (T_Refresh | F_Ptr | F_Nqual)
#define D_Var           (F_Var | F _Nqual | F _Ptr)
#define D_Tvar          (D_Var | F_Tvar)
```

As indicated previously, flags, type codes, and d-words are distinguished by the prefixes `F_`, `T_`, and `D_`, respectively.

4.4.4 RTL Coding

Since the optimizing compiler was introduced in versions 8 and 9 of Icon, the routines for the run-time system use an extended C syntax called RTL (for Run-Time Language) that encodes the type information for arguments and results. Some of these are illustrated by the RTL function for the Icon operator `*x`, which produces the size of `x`:

```
operator{1} * size(x)
abstract { return integer }
type_case x of {
   string: inline { return C_integer StrLen(x); }
   list: inline { return C_integer BlkD(x,List)->size; }
   table: inline { return C_integer BlkD(x,Table)->size; }
   set: inline { return C_integer BlkD(x,Set)->size; }
   cset: inline {
      register word i = BlkD(x,Cset)->size;
      if (i < 0) i = cssize(&x);
      return C_integer i;
      }
   ...
   default: {
         /*
```

```
            * Try to convert it to a string.
            */
           if !cnv:tmp_string(x) then
              runerr(112, x);    /* no notion of size */
           inline {
              return C_integer StrLen(x);
              }
           }
      }
end
```

`operator` is an RTL construct that performs several operations. One of these operations is to provide a C function declaration. Since the function is called by the interpreter, the header is somewhat different from what it would be if `size` were called directly. The details are described in Chapter 8.

The arguments of the Icon operation are referred to via named descriptors, such as `x`. The result that is produced is also a descriptor.

RTL extends C's `return` statement to include type information, with which the d-word of the return value is set to `D_lnteger`, since the returned value is a `C_integer`. Next, the `type_case` selects different branches of code depending on the type of x. In the generated code there is a test to determine if descriptor `x` holds a qualifier. `Qual()` is a macro that is defined as

```
#define Qual(d)    (!((d).dword & F_Nqual))
```

If `x` is a qualifier, its length is placed in the v-word of the return value descriptor, using the macros IntVal and StrLen, which are defined as

```
#define IntVal(d)  ((d).vword.integr)
#define StrLen(d)  ((d).dword)
```

If `x` is not a qualifier, then the size depends on the type. The macro `Type()` isolates the type code

```
#define Type(d)    ((d).dword & TypeMask)
```

where the value of `TypeMask` is 63, providing considerable room for additions to Icon's internal types.

For most Icon types that are represented by blocks, their source-language size is contained in their `size` field. The macro `BlkLoc()` accesses a pointer in the v-field of a descriptor and is defined as

```
#define BlkLoc(d) ((d).vword.bptr)
```

A more specialized macro `BlkD()` wraps uses of `BlkLoc()` and subsequent union member access, allowing descriptor-block consistency to be verified at run-time if desired.

If the type is not one of those given, the final task is an attempt to convert `x` to a string. The RTL expression `cnv:tmp_string()` does this, using local temporary buffer. The value of `x` is changed accordingly. A fixed-sized buffer can be used, since there is a limit to the size of a string that can be obtained by converting other types. This limit is 256, which is reached only for conversion of `&cset`. The conversion may fail, as for `*&null`, which is signaled by the return value 0 from `cnv:tmp_string()`. In this case, program execution is terminated with a run-time: error message, using `runerr()`. If the conversion is successful, the size is placed in the v-word of the result, as is the case if `x` was a qualifier originally.

RETROSPECTIVE: Descriptors provide a uniform way of representing Icon values and variables. Since descriptors for all types of data are the same size, there are no problems with assigning different types of values to a variable---they all fit.

The importance of strings is reflected in the separation of descriptors into two classes---qualifiers and nonqualifiers---by the n flag. The advantages of the qualifier representation for strings are discussed in Chapter 5.

It is comparatively easy to add a new type to Icon. A new type code is needed to distinguish it from other types. If the possible values of the new type are small enough to fit into the v-word, as is the case for integers, no other data is needed. For example, the value of a character data type could be contained in its descriptor. For types that have values that are too large to fit into a v-word, pointers to blocks containing the data are placed in the v-words instead. Lists, sets, and tables are examples of data types that are represented this way. See Chapters 6 and 7.

EXERCISES

4.1 Give examples of Icon programs in which heterogeneous aggregates are used in significant ways.

4.2 Design a system of type declarations for Icon so that the translator could do type checking. Give special consideration to aggregates, especially those that may change in size during program execution. Do this from two perspectives: (a) changing the semantics of Icon as little as possible, and (b) maximizing ,the type checking that can be done by the translator at the expense of flexibility in programming.

4.3 Suppose that functions in Icon were not first-class values and that their meanings were bound at translation time. How much could the translator do in the way of error checking?

4.4 Compile a list of all Icon functions and operators. Are there any that do not require argument type checking? Are there any that require type checking but not conversion? Identify those that are polymorphic. For the polymorphic ones, identify the different kinds of computations that are performed depending on the types of the arguments.

4.5 Compose a table of all type checks and conversions that are required for Icon functions and operators.

4.6 To what extent would the implementation of Icon be simplified if automatic type conversion were not supported? How would this affect the programmer?

4.7 Why is it desirable for string qualifiers not to have flags and for all other kinds of descriptors to have flags indicating they are not qualifiers, rather than the other way around?

4.8 Is the n flag that distinguishes string qualifiers from all other descriptors really necessary? If not, explain how to distinguish the different types of descriptors without this flag.

4.9 On computers with extremely limited address space, two-word descriptors may be impractically large. Describe how one-word descriptors might be designed, discuss how various types might be represented, and describe the ramifications for storage utilization and execution speed.

4.10 Identify the diagrams in this chapter that would be different if they were drawn for a computer with 16-bit words. Indicate the differences.

4.11 There is nothing in the nature of keywords that requires them to be processed in a special way for assignment but not for dereferencing. Invent a new keyword that is a variable that requires processing when it is dereferenced. Show how to generalize the keyword trapped-variable mechanism to handle such cases.

4.12 List all the syntactically distinct cases in which the translator can determine whether a keyword variable is used in an assignment or dereferencing context.

4.13 What would be gained if special code were compiled for those cases in which the context for keyword variables could be determined?

Chapter 5: Strings and Csets

PERSPECTIVE: Several aspects of strings as a language feature in Icon have a strong influence on how they are handled by the implementation. First of all, strings are the most frequently used type of data in the majority of Icon programs. The number of different strings and the total amount of string data often are large. Therefore, it is important to be able to store and access strings efficiently.

Icon has many operations on strings---nearly fifty of them. Some operations, such as determining the size of a string, are performed frequently. The efficiency of these operations is an important issue and influences, to a considerable extent, how strings are represented.

Icon strings may be very long. Although some limitation on the maximum length of a string may be acceptable as a compromise with the architecture of the computer on which Icon is implemented (and hence considerations of efficiency), this maximum must be so large as to be irrelevant for most Icon programs.

String lengths are determined dynamically during program execution, instead of being specified statically in declarations. Much of the advantage of string processing in Icon over other programming languages comes from the automatic management of storage for strings.

Any of the 256 8-bit ASCII characters can appear in an Icon string. Even the "null" character is allowed.

Several operations in Icon return substrings of other strings. Substrings tend to occur frequently, especially in programs that analyze (as opposed to synthesize) strings.

Strings in Icon are atomic---there are no operations in Icon that change the characters in existing strings. This aspect of Icon is not obvious; in fact, there are operations that appear to change the characters in strings. The atomic nature of string operations in Icon simplifies its implementation considerably. For example, assignment of a string value to a variable need not (and does not) copy the string.

The order in which characters appear is an essential aspect of strings. There are many situations in Icon, however, where several characters have the same status but where their order is irrelevant. For example, the concepts of vowels and punctuation marks depend on set membership but not on order. Csets are provided for such situations. Interestingly, many computations can be performed using csets that have nothing to do with the characters themselves (Griswold and Griswold 1983, pp. 181-191).

5.1 Strings

5.1.1 Representation of Strings

Although it may appear natural for the characters of a string to be stored in consecutive bytes, this has not always been so. On earlier computer architectures without byte addressing and character operations, some string-manipulation languages represented strings by linked lists of words, each word containing a single character. Such a representation seems bizarre for modem computer architectures and obviously consumes a very large amount of memory-an intolerable amount for a language like Icon.

The C programming language represents strings (really arrays of characters) by successive bytes in memory, using a zero (null) byte to indicate the end of a string. Consequently, the end of a string can be determined from the string itself, without any external information. On the other hand, determining the length of a string, if it is not already known, requires indexing through it, incrementing a counter until a null byte is found. Furthermore, and very important for a language like Icon, substrings (except terminal ones) cannot occur within strings, since every C string must end with a null byte.

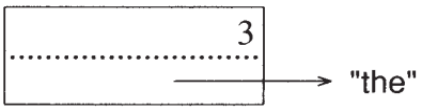
Since any character can occur in an Icon string, it is not possible to use C's null-termination approach to mark ends of strings. Therefore, there is no way to detect the end of a string from the string itself, and there must be some external way to determine where a string ends. This consideration provides the motivation for the qualifier representation described in the last chapter. The qualifier provides information, external to the string itself, that delimits the string by the address of its first character and its length. Such a representation makes the computation of substrings fast and simple---and, of course, determining the length of a string is fast and independent of its length.

Note that C-style strings serve perfectly well as Icon-style strings; the null byte at the end of a C-style string can be ignored by Icon. This allows strings produced by C functions to be used by Icon. The converse is not true; in order for an Icon string to be used by C, a copy must be made with a null byte appended at the end.

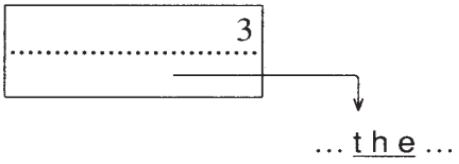
Some strings are compiled into the run-time system and others, such as strings that appear as literals in a program, are contained in icode files that are loaded into memory when program execution begins. During program execution, Icon strings may be stored in work areas (usually referred to as "buffers"). Most newly created strings, however, are allocated in a common string region.

As source-language operations construct new strings, their characters are appended to the end of those already in the string region. The amount of space allocated in the string region typically increases during program execution until the region is full, at which point it is compacted by garbage collection, squeezing out characters that are no longer needed. See Chapter 11 for details.

In the previous chapter, the string to which a qualifier points is depicted by an arrow followed by the string. For example, the string "the" is represented by the qualifier



The pointer to "the" is just a notational convenience. A more accurate representation is



The actual value of the v-word might be 0x569a (hexadecimal), where the character t is at memory location 0x569a, the character h is at location 0x569b, and the character e is at location 0x569c.

5.1.2 Concatenation

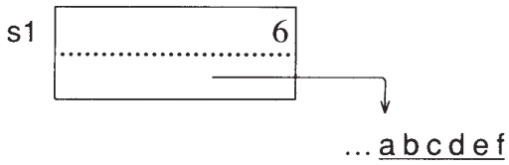
In an expression such as

```
s := "hello"
```

the string "hello" is contained in data provided as part of the icode file, and a qualifier for it is assigned to s; no string is constructed. Some operations that produce strings require the allocation of new strings. Concatenation is a typical example:

```
s1 := "ab" || "cdef"
```

In this expression, the concatenation operation allocates space for six characters, copies the two strings into this space, and produces a qualifier for the result:



This qualifier then becomes the value of `s1`.

There are important optimizations in concatenation. If the first argument in a concatenation is the last string in the string region, the second argument is simply appended to the end of the string region. Thus, operations of the form

```
s := s || expr
```

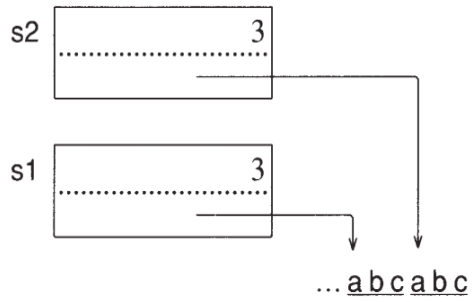
perform less allocation than operations of the form

```
s := expr || s
```

Similarly, if the strings being concatenated are already adjacent, no concatenation need be performed. Except for these optimizations, no string construction operation attempts to use another instance of a string that may exist somewhere else in the string region. As a result,

```
s1 := "ab" || "c"
s2 := "a" || "bc"
```

produce two distinct strings:



The RTL code for the concatenation operation is

```
   operator{1} || cater(x,y)
   if !cnv:string(x) then
      runerr(103, x)
   if !cnv:string(y) then
      runerr(103, y)

   abstract {
      return string
      }
   body {
      CURTSTATE();

      /*
       *  Optimization 1:  The strings to be concatenated are
       *  already adjacent in memory; no allocation is required.
       */
      if (StrLoc(x) + StrLen(x) == StrLoc(y)) {
         StrLoc(result) = StrLoc(x);
         StrLen(result) = StrLen(x) + StrLen(y);
         return result;
         }
      else if ((StrLoc(x) + StrLen(x) == strfree)
      && (DiffPtrs(strend,strfree) > StrLen(y))) {
         /*
          * Optimization 2: The end of x is at the end of the
string space.
          *  Hence, x was the last string allocated and need
          *  not be re-allocated. y is appended to the string
          *  space and the result is pointed to the start of x.
          */
       result = x;
       /*
        * Append y to the end of the string space.
        */
       Protect(alcstr(StrLoc(y),StrLen(y)), runerr(0));
       /*
        *  Set the length of the result and return.
        */
       StrLen(result) = StrLen(x) + StrLen(y);
       return result;
          }

      /*
       * Otherwise, allocate space for x and y, and copy them
       *  to the end of the string space.
       */
      Protect(StrLoc(result) = alcstr(NULL, StrLen(x) +
                                       StrLen(y)), runerr(0));
      memcpy(StrLoc(result), StrLoc(x), StrLen(x));
      memcpy(StrLoc(result) + StrLen(x), StrLoc(y), StrLen(y));

      /*
       *  Set the length of the result and return.
       */
      StrLen(result) = StrLen(x) + StrLen(y);
      return result;
      }
end
```

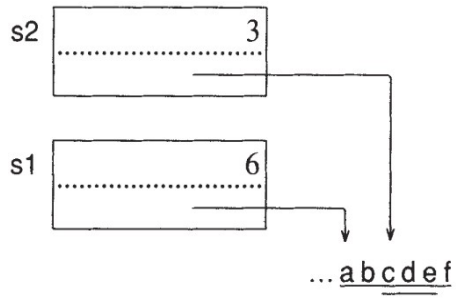
The function `strreq(n)` assures that there are at least `n` bytes available in the allocated string region. See Chapter 11 for details. The function `alcstr(s, n)` allocates `n` characters and copies `s` to that space. The global variable `strfree` points to the beginning of the free space at the end of the allocated string region.

5.1.3 Substrings

Many string operations do not require the allocation of a new string but only produce new qualifiers. For example, if the value of `s1` is `"abcdef"`, the substring formed by

```
s2 := s1[3:6]
```

does not allocate a new string but only produces a qualifier that points to a substring of `s1`:



In order for Icon string values to be represented in memory by substrings, it is essential that there be no Icon operation that changes the characters inside a string. As mentioned earlier, this is the case, although it is not obvious from a cursory examination of the language. C, on the other hand, allows the characters in a string to be changed. The difference is that C considers a string to be an array of characters and allows assignment to the elements of the array, while Icon considers a string to be an indivisible atomic object. It makes no more sense in Icon to try to change a character in a string than it does to try to change a digit in an integer. Thus, if

```
i := j
```

and

```
j := j + 1
```

the value of `i` does not change as a result of the subsequent assignment to `j`. So it is with strings in Icon.

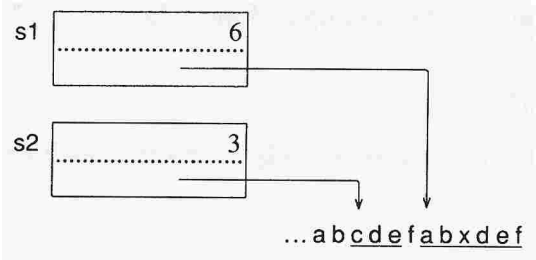
Admittedly, there are operations in Icon that *appear* to change the characters in a string. For example,

```
s1[3] := "x"
```

gives the appearance of changing the third character in `s1` to `"x"`. However, this expression is simply shorthand for

```
s1 := s1[1:3] || "x" || s1[4:0]
```

A new string is created by concatenation and a new qualifier for it is assigned to `s1`, as shown by



Of course, the length of the string may be increased or decreased by assignment to a substring, as in

```
s1[3] := "xxx"
s1 [2:5] := ""
```

5.1.4 Assignment to Subscripted Strings

Expressions such as `x[i]` and `x[i:j]` represent a particular challenge in the implementation of Icon. In the first place, the translator cannot determine the type of x. In the case of `x[i]`, there are four basic types that `x` may legitimately have: string, list, table, and record. Of course, any type that can be converted to a string is legitimate also. Unfortunately, the *nature* of the operation, not just the details of its implementation, depends on the type. For strings,

```
s1 [3] := s2
```

replaces the third character of `s1` by `s2` and is equivalent to concatenation, as described previously. For lists, tables, and records,

```
x[3] := y
```

changes the third *element* of `x` to `y`–quite a different matter (see Exercise 5.5).

This problem is pervasive in Icon and only needs to be noted in passing here. The more serious problem is that even if the subscripted variable is a string, the subscripting expression has different meanings, depending on the context in which it appears.

If `s` is a variable, then `s[i]` and `s[i:j]` also are variables. In a dereferencing context, such as

```
write(s[2:5])
```

the result produced by `s[2:5]` is simply a substring of `s`, and the subscripting expression produces the appropriate qualifier.

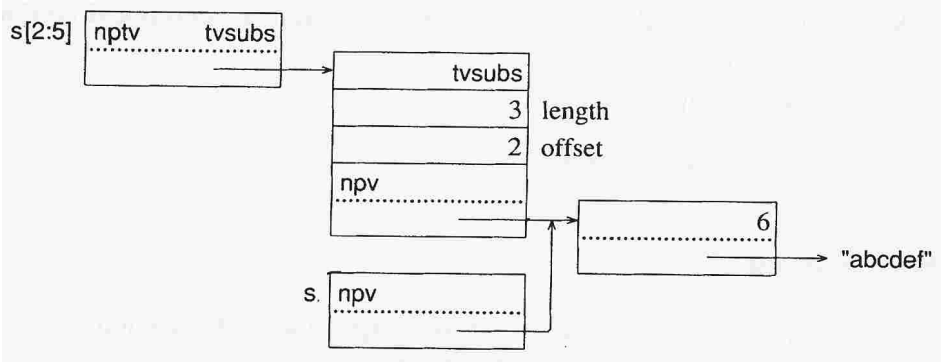
Assignment to a subscripted string, as in

```
s[2:5] := "xxx"
```

is not at all what it appears to be superficially. Instead, as already noted, it shorthand for an assignment to `s`:

```
s := s[1] || "xxx" || s[6:0]
```

If the translator could determine whether a subscripting expression is used in dereferencing or assignment context, it could produce different code for the two cases. As mentioned in Sec. 4.3.2, however, the translator cannot always make this determination. Consequently, trapped variables are used for subscripted string much in the way they are used for keywords. For example, if the value of `s` is `"abcdef"`, the result of evaluating the subscripting expression `s[2:5]` is a *substring trapped variable* that has the form

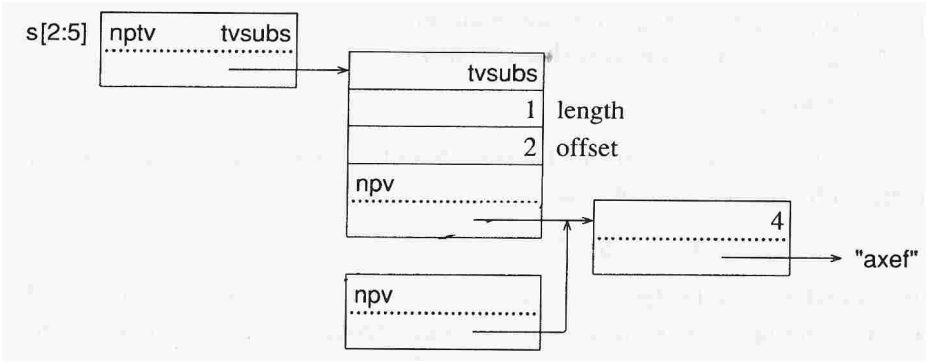


Note that both the variable for `s` and the variable in the substring trapped-variable block point to the same value. This makes it possible for assignment to the substring trapped variable to change the value of `s`.

The length and offset of the substring provide the necessary information either to produce a qualifier for the substring, in case the subscripting expression is dereferenced, or to construct a new string in case an assignment is made to the subscripting expression. For example, after an assignment such as

```
s[2:5] := "x"
```

the situation is



Note that the value of `s` has changed. The length of the subscripted portion of the string has been changed to correspond to the length of the string assigned to it. This reflects the fact that subscripting identifies the portions of the string before and after the subscripted portion (`"a"` and `"ef"`, in this case). In the case of a multiple assignment to a subscripted string, only the original subscripted portion is changed. Thus, in

```
(s[2:5] := "x") := "yyyyy"
```

the final value of `s` is `"ayyyyyef"`.

5.1.5 Mapping

String mapping is interesting in its own right, and the RTL function that implements it illustrates several aspects of string processing:

```
function{1} map(s1,s2,s3)
   /*
    * s1 must be a string; s2 and s3 default to (string
    * conversions of) &ucase and &lcase, respectively.
    */
   if !cnv:string(s1) then
      runerr(103,s1)
```

```
   ...
   abstract {
      return string
      }
   body {
      register int i;
      register word slen;
      register char *str1, *str2, *str3;
#ifndef Concurrent
      static char maptab[256];
#endif                                  /* Concurrent */
      CURTSTATE();
      /*
       * Default is here, conversion only if cached maptab fails
       */
      if (is:null(s2))
         s2 = ucase;
      /*
       * Short-cut conversions of &lcase and &ucase.
       */
      else {
       struct descrip _k_lcase_, _k_ucase_;
       Klcase(&_k_lcase_);
       Kucase(&_k_ucase_);
       if (s2.dword == D_Cset) {
          if (BlkLoc(s2) == BlkLoc(_k_lcase_)) {
             s2 = lcase;
             }
          else if (BlkLoc(s2) == BlkLoc(_k_ucase_)) {
             s2 = ucase;
             }
          }
       }

      if (is:null(s3))
         s3 = lcase;
      /*
       * Short-cut conversions of &lcase and &ucase.
       */
      else {
       struct descrip _k_lcase_, _k_ucase_;
       Klcase(&_k_lcase_);
       Kucase(&_k_ucase_);
       if (s3.dword == D_Cset) {
          if (BlkLoc(s3) == BlkLoc(_k_lcase_)) {
             s3 = lcase;
             }
          else if (BlkLoc(s3) == BlkLoc(_k_ucase_)) {
             s3 = ucase;
             }
          }
       }
#endif                                  /* !COMPILER */

      /*
       * If s2 and s3 are the same as for the last call of map,
       *  the current values in maptab can be used. Otherwise,
the
```

```
         *  mapping information must be recomputed.
         */
        if (!EqlDesc(maps2,s2) || !EqlDesc(maps3,s3)) {
           maps2 = s2;
           maps3 = s3;

#if !COMPILER
           if (!cnv:string(s2,s2))
              runerr(103,s2);
           if (!cnv:string(s3,s3))
              runerr(103,s3);
#endif                                  /* !COMPILER */
           /*
            * s2 and s3 must be of the same length
            */
           if (StrLen(s2) != StrLen(s3))
              runerr(208);

           /*
            * The array maptab is used to perform the mapping.
First,
            *  maptab[i] is initialized with i for i from 0 to 255.
            *  Then, for each character in s2, the position in
maptab
            *  corresponding to the value of the character is
assigned
            *  the value of the character in s3 that is in the same
            *  position as the character from s2.
            */
           str2 = StrLoc(s2);
           str3 = StrLoc(s3);
           for (i = 0; i <= 255; i++)
              maptab[i] = i;
           for (slen = 0; slen < StrLen(s2); slen++)
              maptab[str2[slen]&0377] = str3[slen];
           }

        slen = StrLen(s1);

        if (slen == 0) {
         return emptystr;
         }
        else if (slen == 1) {
         char c = maptab[*(StrLoc(s1)) & 0xFF];
         return string(1, (char *)&allchars[FromAscii(c) & 0xFF]);
         }

        /*
         * The result is a string the size of s1; create the result
         *  string, but specify no value for it.
         */
        StrLen(result) = slen;
        Protect(StrLoc(result) = alcstr(NULL, slen), runerr(0));
        str1 = StrLoc(s1);
        str2 = StrLoc(result);

        /*
         * Run through the string, using values in maptab to do the
```

```
         *  mapping.
         */
        while (slen-- > 0)
           *str2++ = maptab[(*str1++)&0377];

        return result;
        }
end
```

The mapping is done using the character array `maptab`. This array is set up by first assigning every possible character to its own position in `maptab` and then replacing the characters at positions corresponding to characters in `s2` by the corresponding characters in `s3`. Note that if a character occurs more than once in `s2`, its last (rightmost) correspondence with a character in `s3` applies.

To avoid rebuilding `maptab` unnecessarily, this step is bypassed if `map()` is called with the same values of `s2` and `s3` as in the previous call. The global variables `maps2` and `maps3` are used to hold these "cached" values. The macro `EqlDesc(d1,d2)` tests the equivalence of the descriptors `d1` and `d2`.

The function `map()` is an example of a function that defaults null-valued arguments. Omitted arguments are supplied as null values. The defaults for `s2` and `s3` are `&ucase` and `&lcase`, respectively. Consequently,

```
map(s)
```

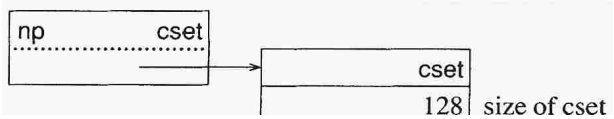
is equivalent to

```
map(s, &ucase, &lcase)
```

The macro `ChkNull(d)` tests whether or not `d` is null. The values of `&ucase` and `&lcase` are in the global constants `ucase` and `lcase`.

5.2 Csets

Since Icon uses 8-bit characters, regardless of the computer on which it is implemented, there are 256 different characters that can occur in csets. A cset block consists of the usual title containing the cset type code followed by a word that contains the number of characters in the cset. Next, there are words containing a total of 256 bits. Each bit represents one character, with a bit value of 1 indicating that the character is present in the cset and a bit value of 0 indicating it is absent. An example is the value of the keyword `&ascii`:



The first 128 bits are 1, since these are the bits that correspond to those in ASCII character set.

The C structure for a cset block is

```
struct b_cset {                     /* cset block */
   word title;                      /*    T_Cset */
   word size;                       /*    size of cset */
   unsigned int bits [CsetSize];    /*    array of bits */
};
```

where CsetSize is the number of words required to make up a total of 256 bits. CsetSize is 8 on a computer with 32-bit words and 4 on a computer with 64-bit words.

Cset operations are comparatively straightforward. The characters in a cset are represented by a bit vector that is divided into words to accommodate conventional computer architectures. For example, the C code for cset complementation is

```
operator{1} ~ compl(x)
   /*
    * x must be a cset.
    */
   if !cnv:tmp_cset(x) then
      runerr(104, x)

   abstract {
      return cset
      }
   body {
      register int i;
      struct b_cset *cp, *cpx;

      /*
       * Allocate a new cset and then copy each cset word from
       *  x into the new cset words, complementing each bit.
       */
      Protect(cp = alccset(), runerr(0));
      /* must come after alccset() since BlkLoc(x) could move */
      cpx = (struct b_cset *)BlkLoc(x);
      for (i = 0; i < CsetSize; i++)
          cp->bits[i] = ~cpx->bits[i];
      return cset(cp);
      }
end
```

RETROSPECTIVE: The central role of strings in Icon and the nature of the operations performed on them leads to a representation of string data that is distinct from other data. The qualifier representation is particularly important in providing direct access to string length and in allowing the construction of substrings without the allocation of additional storage. The penalty paid is that a separate test must be performed to distinguish strings from all other kinds of values.

The ability to assign to subscripted strings causes serious implementation problems. The trapped-variable mechanism provides a solution, but it does so at considerable expense in the complexity of code in the run-time system as well as storage allocation for trapped-variable blocks. This expense is incurred even if assignment is not made to a subscripted string.

EXERCISES

5.1 What are the ramifications of Icon's use of the 256-bit ASCII character set, regardless of the "native" character set of the computer on which Icon is implemented?

5.2 Catalog all the operations on strings in Icon and point out any that might cause special implementation problems. Indicate the aspects of strings and string operations in Icon that are the most important in terms of memory requirements and processing speed.

5.3 List all the operations in Icon that require the allocation of space for the construction of strings.

5.4 It has been suggested that it would be worth trying to avoid duplicate allocation of the same string by searching the string region for a newly created string to see if it already exists before allocating the space for it. Evaluate this proposal.

5.5 Consider the following four expressions:

```
s1 [i] := s2
s1 [i+:1] := s2
a1 [i] := a2
a1 [i+:1] := a2
```

where `s1` and `s2` have string values and `a1` and `a2` have list values. Describe the essential differences between the string and list cases. Explain why these differences indicate flaws in language design. Suggest an alternative.

5.6 The substring trapped-variable concept has the advantage of making it possible to handle all the contexts in which string-subscripting expressions can occur. It is expensive, however, in terms of storage utilization. Analyze the impact of this feature on the performance of "typical" Icon programs.

5.7 Since the contexts in which most subscripting expressions occur can be determined, describe how to handle these without using trapped variables.

5.8 If a subscripting expression is applied to a result that is not a variable, it is erroneous to use such an expression in an assignment context. In what situations can the translator detect this error? Are there any situations in which a subscripting expression is applied to a variable but in which the expression cannot be used in an assignment context?

5.9 There are some potential advantages to unifying the keyword and substring trapped-variable mechanisms into a single mechanism in which all trapped variables would have pointers to functions for dereferencing and assignment. What are the disadvantages of such a unification?

5.10 Presumably, it is unlikely for a programmer to have a constructive need for the polymorphic aspect of subscripting expressions. Or is it? If it is unlikely, provide a supporting argument. On the other hand, if there are situations in which this capability is useful, describe them and give examples.

5.11 In some uses of `map(s1, s2, s3)`, `s1` and `s2` remain fixed while `s3` varies (Griswold 1980b). Devise a heuristic that takes advantage of such usage.

Chapter 6: Lists

PERSPECTIVE:Most programming languages support some form of vector or array data type in which elements can be referenced by position. Icon's list data type fills this need, but it differs from similar types in many languages in that Icon lists are constructed during program execution instead of being declared during compilation. Therefore, the size of a list may not be known until run time.

Icon's lists are data objects. They can be assigned to variables and passed as arguments to functions. They are not copied when this is done; in fact, a value of type list is simply a descriptor that points to the structure that contains the list elements. These aspects of lists are shared by several other Icon data types and do not add anything new to the implementation. The attribute of lists that presents the most challenging implementation problem is their ability to grow and shrink by the use of stack and queue access mechanisms.

Lists present different faces to the programmer, depending on how they are used. They may be static vectors referenced by position or they may be dynamic changing stacks or queues. It might seem that having a data structure with such apparently discordant access mechanisms would be awkward and undesirable. In practice, Icon's lists provide a remarkably flexible mechanism for dealing with many common programming problems. The two ways of manipulating lists are rarely intermixed. When both aspects are needed, they usually are needed at different times. For example, the number of elements needed in a list often is not known when the list is created. Such a list can be created with no elements, and the elements can be pushed onto it as they are produced. Once such a list has been constructed, it may be accessed by position with no further change in its size.

6.1 Structures for Lists

The fusion of vector, stack, and queue organizations is reflected in the implementation of Icon by relatively complicated structures that are designed to provide a reasonable compromise between the conflicting requirements of the different access mechanisms.

A list consists of a fixed-size *list-header block,* which contains the usual title, the current size of the list (the number of elements in it), and descriptors that point to the first and last blocks on a doubly-linked chain of *list-element blocks* that contain the actual list elements. List-element blocks vary in size.

A list-element block contains the usual title, the size of the block in bytes three words used to determine the locations of elements in the list-element block and descriptors that point to the next and previous list-element blocks, if any. A null pointer (in Unicon, a pointer back to the list header block) indicates the absence of a pointer to another list-element block. Following this data, there are slots for elements. Slots always contain valid descriptors, even if they are not used to hold list elements.

The structure declarations for list-header blocks and list-element blocks are

```
struct b_list {    /* list-header block */
   word title;     /*    T_List */
   word size;      /*    current list size */
   word id;        /*    identification number */
   union block *listhead; /* first list-element block */
```

```
      union block *listtail; /* last list-element block */
   };
   struct b_lelem {   /* list-element block */
      word title;     /*    T_Lelem */
      word blksize;   /*    size of block */
      word nslots;    /*    total number of slots */
      word first;     /*    index of first used slot */
      word nused;     /*    number of used slots */
      union block *listprev;  /* previous list-element block */
      union block *listnext;  /* next list-element block */
      struct descrip lslots[1]; /* array of slots */
   };
```

When a list is created, either by

```
   list(n, x)
```

or by

```
   [x1 ,x2, ..., xn]
```

there is only one list-element block. Other list-element blocks may be added to the chain as the result of pushs or puts.

List-element blocks have a minimum number of slots. This allows some expansion room for adding elements to lists, such as the empty list, that are small initially. The minimum number of slots is given by MinListSlots, which normally is eight. In the examples that follow, the value of MinListSlots is assumed to be four in order to keep the diagrams to a manageable size.

The code for the list function is

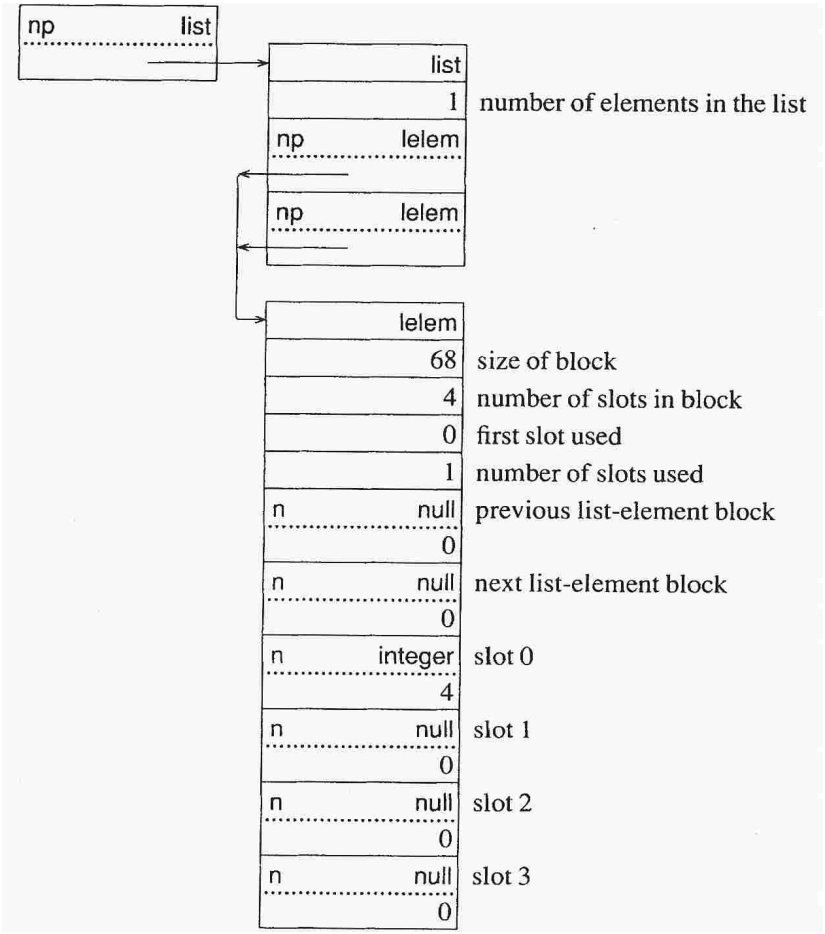
```
function{1} list(n, x)
   if is:set(n) then {
      abstract {
         return new list(store[type(n).set_elem])
            }
      body {
         struct descrip d;
         cnv_list(&n, &d); /* can't fail, we know n is a set */
         return d;
         }
      }
   else {
      if !def:C_integer(n, 0L) then
       runerr(101, n)

   abstract {
      return new list(type(x))
      }

   body {
      tended struct b_list *hp;
      register word i, size;
      word nslots;
      register struct b_lelem *bp; /* doesnt need to be tended */

      nslots = size = n;

      /*
       * Ensure that the size is positive and that the
```

```
         *  list-element block has at least MinListSlots slots.
         */
        if (size < 0) {
           irunerr(205, n);
           errorfail;
           }
        if (nslots == 0)
           nslots = MinListSlots;
        /*
         * Allocate the list-header block and a list-element block.
         *  nslots is the number of slots in the list-element
         *  block while size is the number of elements in the list.
         */
        Protect(hp = alclist_raw(size, nslots), runerr(0));
        bp = (struct b_lelem *)hp->listhead;

        /*
         * Initialize each slot.
         */
        for (i = 0; i < size; i++)
           bp->lslots[i] = x;

        Desc_EVValD(hp, E_Lcreate, D_List);

        /*
         * Return the new list.
         */
        return list(hp);
        }
     }
end
```

The data structures produced for a list are illustrated by the result of evaluating

```
a := list(1, 4)
```

which produces a one-element list containing the value 4:



Data Structures for `list(1,4)`

Note that there is only one list-element block and that the slot indexing in the block is zero-based. Unused slots contain null values that are logically inaccessible.

6.2 Queue and Stack Access

Elements in a list-element block are stored as a doubly-linked circular queue. If an element is added to the end of the list `a`, as in

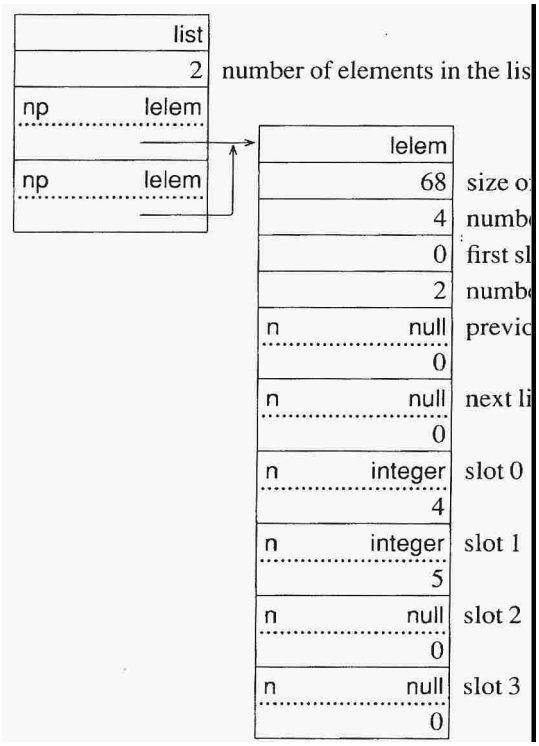
```
put(a, 5)
```

the elements of the list are 4 and 5. The value is added to the "'end" of the last list-element block, assuming there is an unused slot (as there is in this case). The code in `put()` to do this is

```
/*
 * Point hp to the list-header block and bp to the last
 * list-element block.
 */
hp = (struct b_list *)BlkLoc(x);
bp = (struct b_lelem *) hp->listtail;
/*
 * If the last list-element block is full, allocate a new
 * list-element block, make it the first list-element block,
 * and make it the next block of the former last list-element
 * block.
 */
```

```
      if (bp->nused >= bp->nslots) {
         /*
          * Set i to the size of block to allocate.
          */
         i = hp->size / two;
         if (i < MinListSlots)
            i = MinListSlots;
#ifdef MaxListSlots
         if (i > MaxListSlots)
            i = MaxListSlots;
#endif                              /* MaxListSlots */
         /*
          * Allocate a new list element block.  If the block
          *  can't be allocated, try smaller blocks.
          */
         while ((bp = alclstb(i, (word)0, (word)0)) == NULL) {
            i /= 4;
            if (i < MinListSlots)
             runerr(0);
            }

         hp->listtail->lelem.listnext = (union block *) bp;
         bp->listprev = hp->listtail;
         hp->listtail = (union block *) bp;
         }
   /*
    * Set i to position of new last element and assign val to
    * that element.
    */
   i = bp->first + bp->nused;
   if (i >= bp->nslots)
      i -= bp->nslots;
      bp->lslots[i] = dp[val];

      /*
       * Adjust block usage count and current list size.
       */
      bp->nused++;
      hp->size++;
      }

      /*
       * Return the list.
       */
      return x;
      }
end
```

The effect on the list-header block and list-element block is:

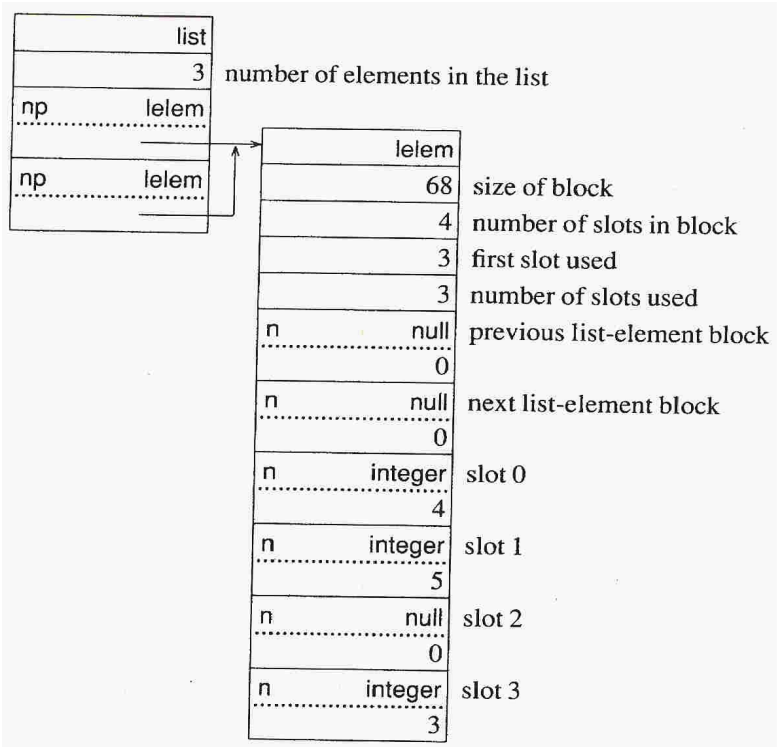


Note that the increase in the number of elements in the header block and in the number of slots used in the list-element block.

If an element is added to the beginning of a list, as in

```
push(a,3)
```

the elements of the list are 3, 4, and 5. The new element is put at the "'beginning" of the first list-element block. The result is



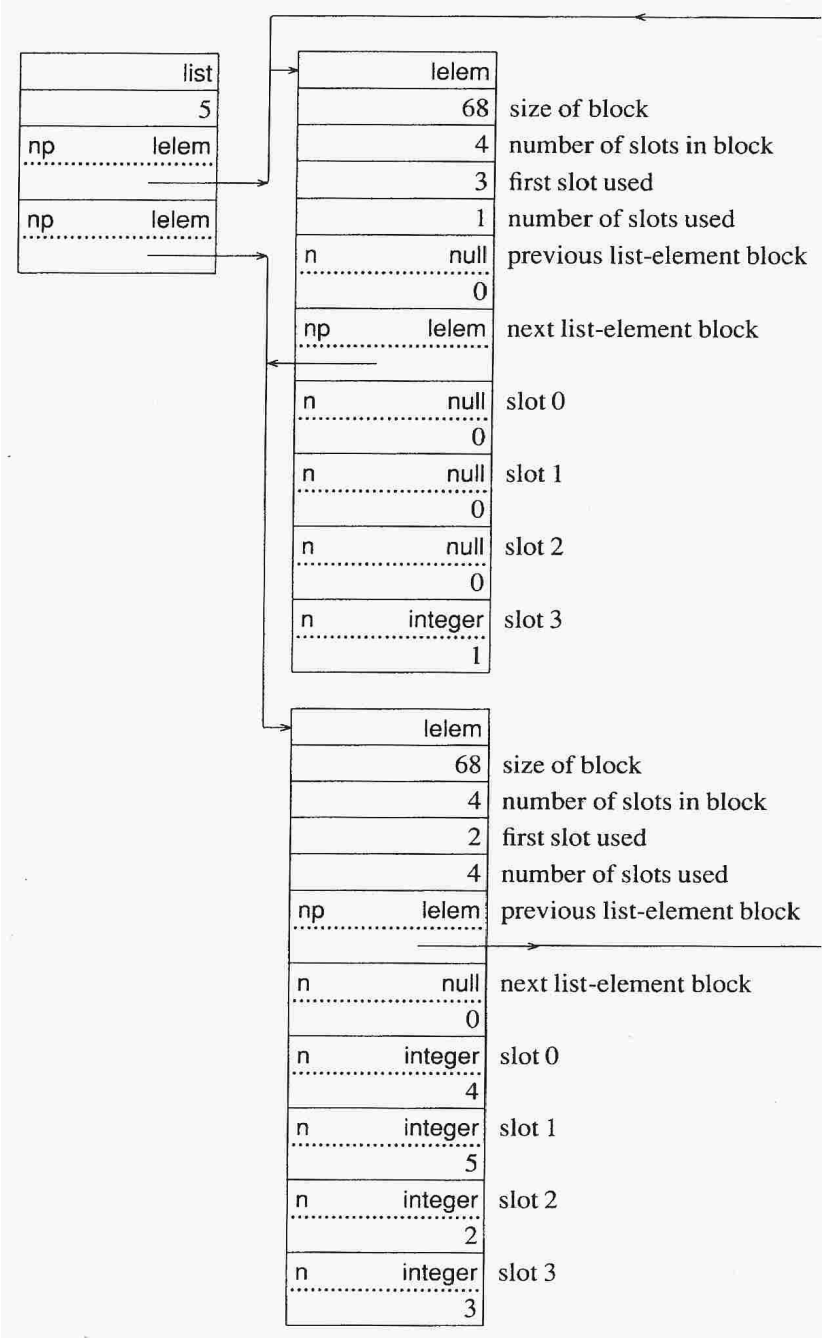
The List Element-Block after a push

Note that the "beginning," which is before the first physical slot in the list-element block, is the last physical slot. The locations of elements that are in a list-element block are determined by the three integers at the head of the list element block. "Removal" of an element by a pop, get, or pull does not shorten the list-element block or overwrite the element; the element merely becomes inaccessible.

If an element is added to a list and no more slots are available in the appropriate list-element block, a new list-element block is allocated and linked in. For example, following evaluation of

```
push(a.2)
push(a.1)
```

the list elements are 1,2,3,4, and 5. The resulting structures are



The Addition of a List-Element Block

As elements are removed from a list by pop (which is synonymous with get) or pull. The indices in the appropriate list-element block are adjusted. The code for pop is

```
int c_get(hp, res)
struct b_list *hp;
struct descrip *res;
{
   register word i;
   register struct b_lelem *bp;

   /*
    * Fail if the list is empty.
    */
   if (hp->size <= 0)
      return 0;

   /*
    * Point bp at the first list block.  If the first block has
    *  no elements in use, point bp at the next list block.
    */
   bp = (struct b_lelem *) hp->listhead;
   if (bp->nused <= 0) {
      bp = (struct b_lelem *) bp->listnext;
      hp->listhead = (union block *) bp;
      bp->listprev = NULL;
      }

   /*
    * Locate first element and assign it to result for return.
    */
   i = bp->first;
   *res = bp->lslots[i];

   /*
    * Set bp->first to new first element, or 0 if the block is
    *  now empty.  Decrement the usage count for the block and
    *  the size of the list.
    */
   if (++i >= bp->nslots)
      i = 0;
   bp->first = i;
   bp->nused--;
   hp->size--;
   return 1;
}
```

where the `c_get()` helper function is invoked from RTL as follows:

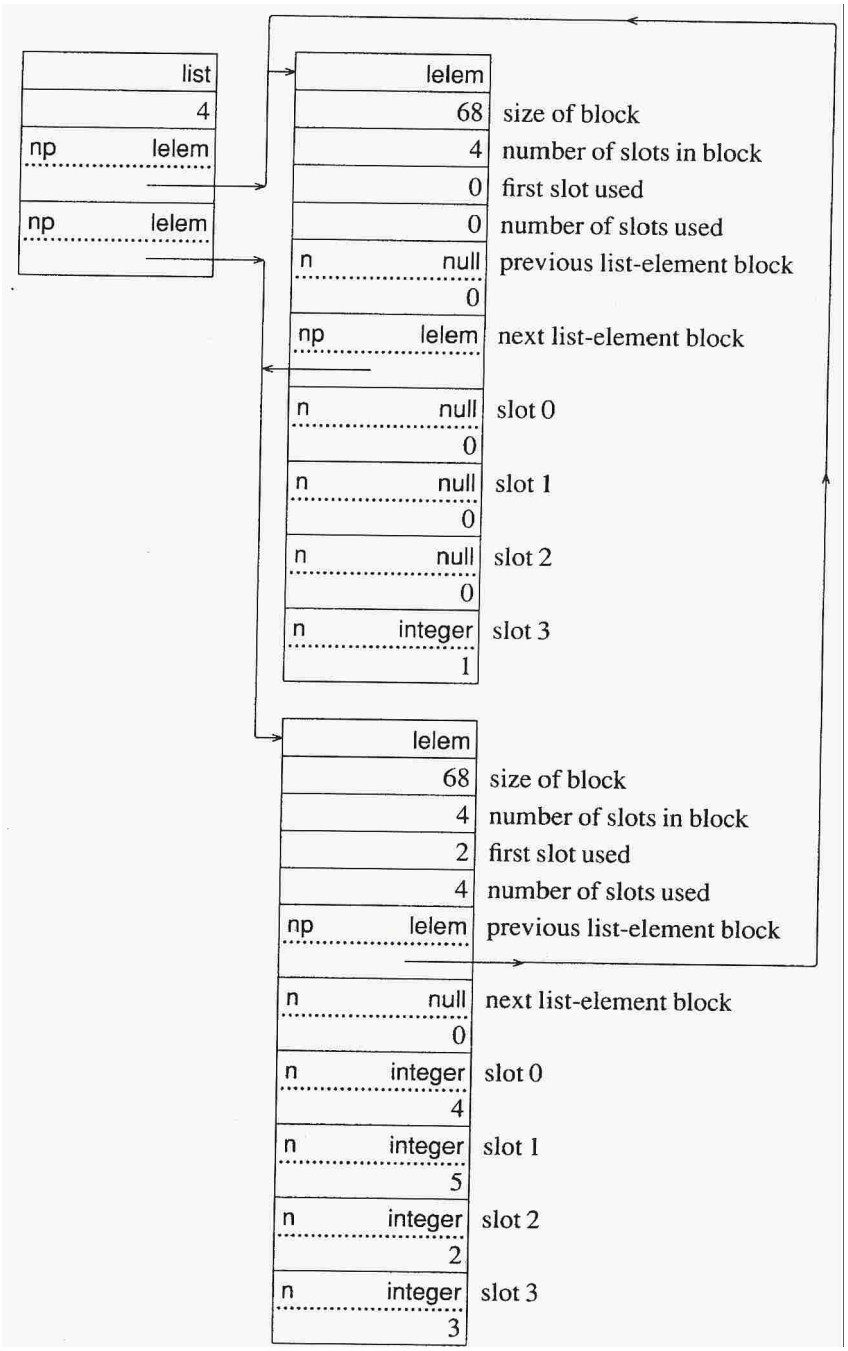
```
function{0,1} get_or_pop(x)
   if !is:list(x) then
      runerr(108, x)

   abstract {
      return store[type(x).lst_elem]
      }

   body {
      if (!c_get((struct b_list *)BlkLoc(x), &result)) fail;
      return result;
```

```
        }
end
```

Thus, as a result of

```
    pop(a)
```

the list elements are 2, 3, 4, and 5. The resulting structures are



The Result of Removing Elements from a List-Element Block

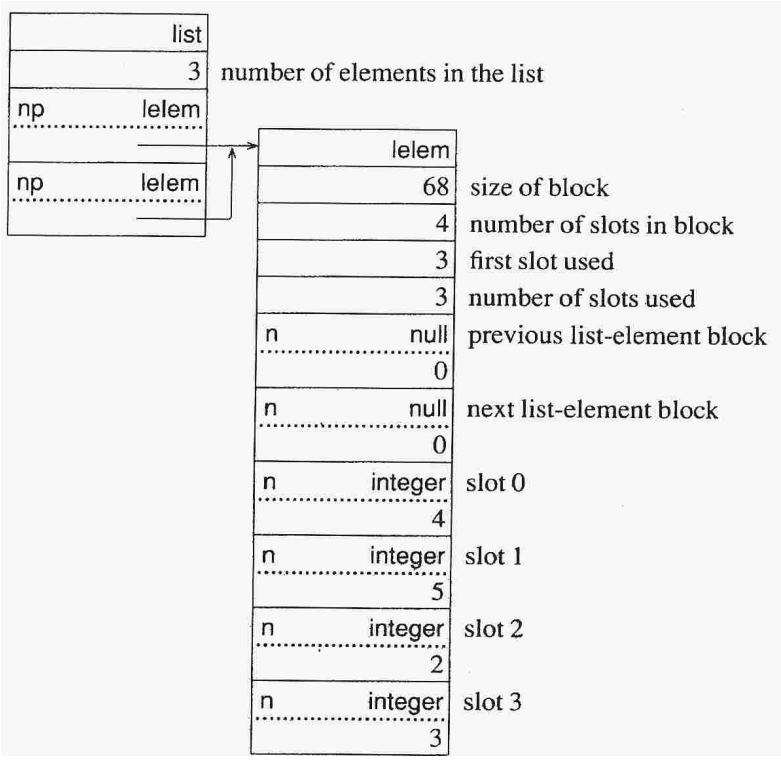
Note that the first list-element block is still linked in the chain, even though it no longer contains any elements that are logically accessible. A list-element block is not removed from the chain when it becomes empty. It is removed only when an element is removed from a list that already has an empty list-element block. Thus, there is always at least one list-element block on the chain, even if the list is empty. Aside from simplifying the

access to list-element blocks from the list-header block, this strategy avoids repeated allocation in the case that pop/push pairs occur at the boundary of two list-element blocks.

Continuing the previous example,

```
pop(a)
```

leaves the list elements 3, 4, and 5. The empty list-element block is removed from the chain:



Removal of an Empty List-Element Block

Note that the value 2 is still physically in the list-element block, although it is logically inaccessible.

6.3 Positional Access

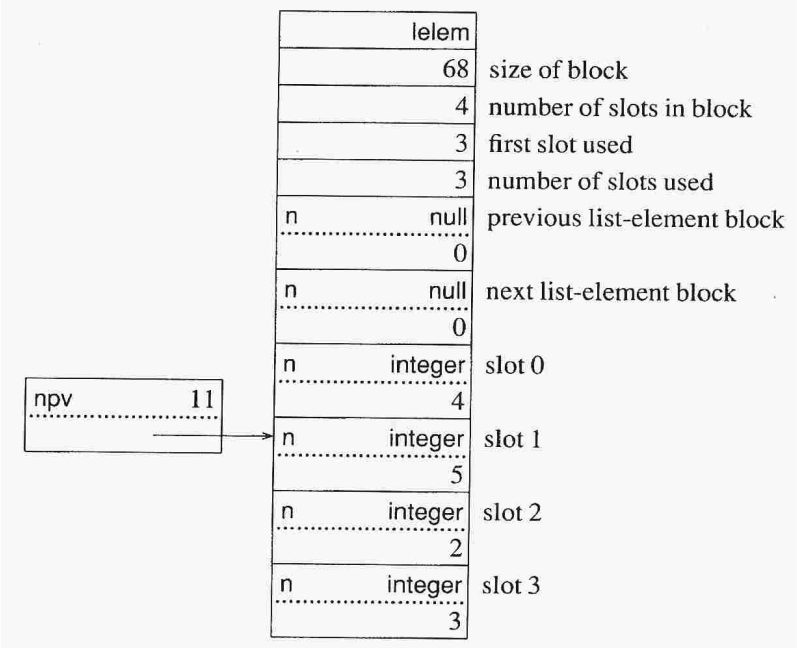
Positional reference of the form `a[i]` requires locating the correct list-element block. Out-of-range references can be determined by examining the list-header block. If the list has several list-element blocks, this involves linking through the list-element blocks, while keeping track of the count of elements in each block until the appropriate one is reached. The result of evaluating `a[i]` is a variable that points to the appropriate slot.

The portion of the subscripting code that handles lists is

```
type_case dx of {
   list: {
      abstract {
         return type(dx).lst_elem
         }
      /*
       * Make sure that y is a C integer.
       */
```

```
      if !cnv:C_integer(y) then {
      /*
       * If it isn't a C integer, but is a large integer,
       * fail on the out-of-range index.
       */
      if cnv : integer(y) then inline { fail; }
      runerr(101, y)
      }
      body {
         word i, j;
         register union block *bp; /* no need to be tended */
         struct b_list *lp;    /* doesn't need to be tended */

      /*
       * Make sure that subscript y is in range.
       */
         lp = (struct b_list *)BlkLoc(dx);
         i = cvpos((long)y, (long)lp->size);
         if (i == CvtFail || i > lp->size)
            fail;
         /*
          * Locate the list-element block containing the
          *  desired element.
          */
         bp = lp->listhead;
         j = 1;
      /*
       * y is in range, so bp can never be null here. If it
       * was, a memory violation would occur in the code that
       * follows, anyhow, so exiting the loop on a NULL bp
       * makes no sense.
       */
      while (i >= j + bp->lelem.nused) {
         j += bp->lelem.nused;
         bp = BlkLoc(bp->lelem.listnext);
         }

      /*
       * Locate desired element and return a pointer to it.
       */
      i += bp->lelem.first - j;
      if (i >= bp->lelem.nslots)
         i -= bp->lelem.nslots;
      return struct_var(&bp->lelem.lslots[i], bp);
      }
   }
```

For the preceding example, `a[3]` produces a variable that points to the descriptor for the value 5:



Referencing a List Element

Note the offset of eleven words in the d-word of the variable. This is present so that the title of the block to which the variable points can be located in case there is a garbage collection. See Chapter 11 for details.

RETROSPECTIVE: The structures used for implementing lists are relatively complicated, but they provide a reasonable compromise, both in the utilization of storage and access speed, that accommodates different access mechanisms.

Using a chain of list-element blocks allows lists to grow in size without limit. From the viewpoint of positional access, this amounts to segmentation. This segmentation only occurs, however, when elements are added to a list. The use of circular queues within list-element blocks allows elements to be removed and added without wasting space.

EXERCISES

6.1 Diagram the structures that result from the evaluation of the following expressions:

```
graph := ["a",,]
graph[2] := graph[3] := graph
```

6.2 How much space does an empty list occupy?

6.3 The portions of the structures for a list that are not occupied by elements of the list constitute overhead. Calculate the percentage of overhead in the following lists. Assume that the minimum number of slots in a list-element block is eight.

```
a := []
a := [1, 2]
a := [1, 2, 3, 4, 5]
a := list(100)
a := []; every put(a, 1 to 100)
```

How do these figures vary as a function of the minimum number of slots in a list-element block?

6.4 What are the implications of not "zeroing" list elements when they are logically removed by a pop, get, or pull?

6.5 When a list-element block is unlinked as the result of a pop, get, or pull, are the elements in it really inaccessible to the source program?

6.6 There is considerable overhead involved in the implementation of *lists* to support both positional access and stack and queue access mechanisms. Suppose the language were changed so that stack and queue access mechanisms applied only to lists that were initially empty. What would the likely impact be on existing Icon programs? How could the implementation take advantage of this change?

6.7 As elements are added to lists, more list-element blocks are added and they tend to become "fragmented." Is it feasible to reorganize such lists, combining the elements in many list-element blocks into one large block? If when and how could this be done?

6.8 A suggested alternative to maintaining a chain of list-element blocks is to allocate a larger block when space is needed and copy elements from the previous block into it. Criticize this proposal.

6.9 Suppose it were possible to insert elements in the middle of lists, rather than only at the ends. How might this feature be implemented?

Chapter 7: Sets and Tables

PERSPECTIVE: Sets and tables are data aggregates that are very useful for a number of common programming tasks. Nevertheless, few programming languages support these data types, with the notable exceptions of Sail (Reiser 1976) and SETL (Dewar, Schonberg, and Schwartz 1981). There are many reasons why these obviously useful data types are not found in most programming languages, but perceived implementation problems certainly rank high among them. If only for this reason, their implementation in Icon is worth studying.

Historically, tables in Icon were inherited from SNOBOL4 and SL5. Sets came later, as an extension to Icon, and were designed and implemented as a class project. Although sets were a late addition to Icon, they are simpler than tables. Nonetheless, they present many of the same implementation problems that tables do. Consequently, sets are considered here first.

Sets and the operations on them support the familiar mathematical concepts of finite sets: membership, the insertion and deletion of members, and the operations of union, intersection, and difference. What is interesting about a set in Icon is that it can contain members of any data type. This is certainly a case where heterogeneity significantly increases the usefulness of a data aggregate without adding to the difficulty of the implementation, *per se.*

The ability of a set to grow and shrink in size influences the implementation significantly. Efficient access to members of a set, which is needed for testing membership as well as the addition and deletion of members, is an important consideration, since sets can be arbitrarily large.

Tables have more structure than sets. Abstractly, a table is a set of pairs that represents a many-to-one relationship-a function. In this sense, the default value of a table provides an extension of the partial function represented by the entry and assigned value pairs to a complete function over all possible entry values. Programmers, however, tend to view tables in a more restricted way, using them to tabulate the attributes of a set of values of interest. In fact, before sets were added to Icon, tables were often used to simulate sets by associating a specific assigned value with membership.

7.1 Sets

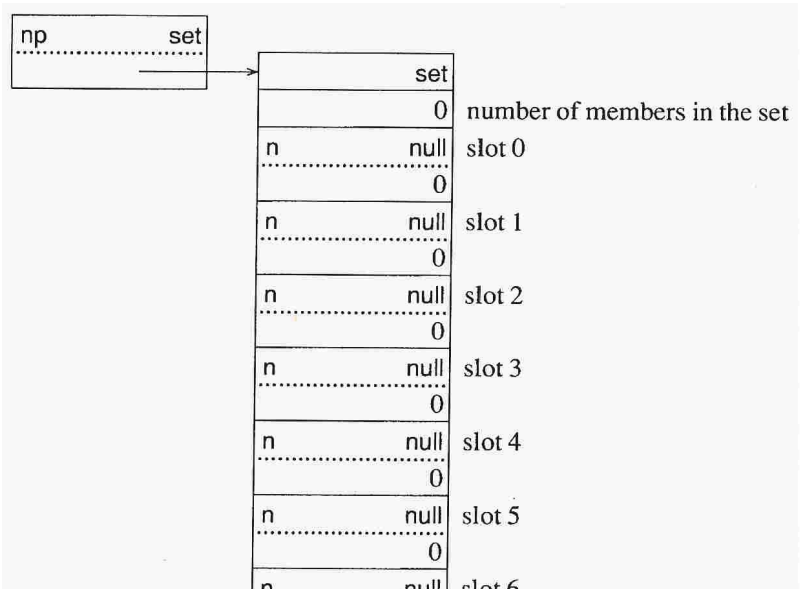
7.1.1 Data Organization for Sets

Hash lookup and linked lists are used to provide an efficient way of locating set members. For every set there is a set-header block that contains a word for the number of members in the set and slots that serve as heads for (possibly empty) linked lists of set-element blocks. The number of slots is an implementation parameter. There are thirty-seven slots in table-header blocks on computers with large address spaces but only thirteen slots on computers with small address spaces.

The structure for an empty set, produced by

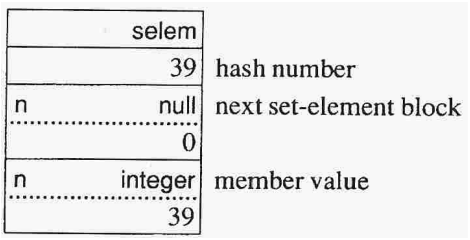
```
s := set([])
```

is



Each member of a set is contained in a separate set-element block. When a value is looked up in a set (for example, to add a new member), a hash number is computed from this value. The absolute value of the remainder resulting from dividing the hash number by the number of slots is used to select a slot.

Each set-element block contains a descriptor for its value, the corresponding hash number, and a pointer to the next set-element block, if any, on the linked list. For example, the set-element block for the integer 39 is:

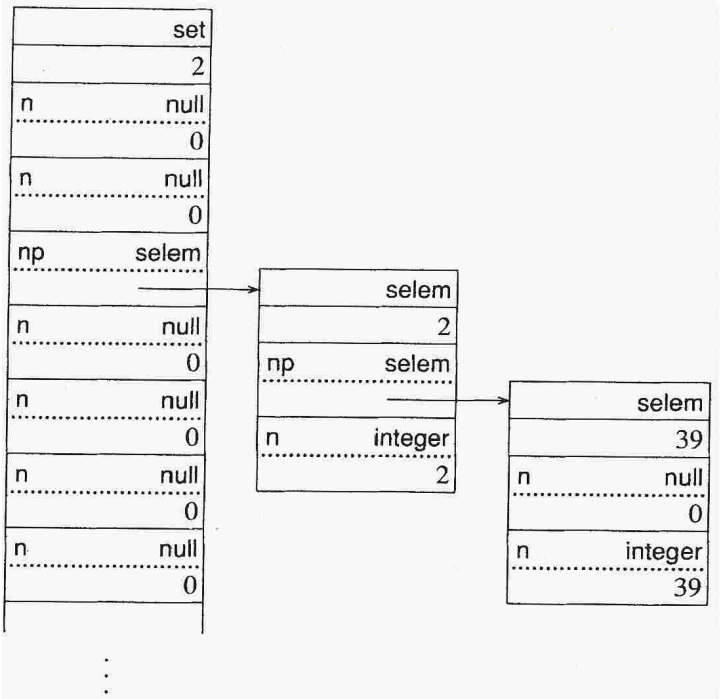


As illustrated by this figure, the hash number for an integer is just the value of the integer. This member goes in slot 2 on computers with large address spaces, since its remainder on division by the number of slots is two. Hash computation is discussed in detail in Sec. 7.3.

The structures for the set

```
s := set([39,2])
```

are



This example was chosen for illustration, since both 2 and 39 go in slot 2.

In searching the list, the hash number of the value being looked up is compared with the hash numbers in the set-element blocks. If a match is found, the value in the set-element block mayor may not be the same as the value being looked up, since collisions in the hash computation are unavoidable. Thus, if the hash numbers are the same, it is necessary to determine whether or not their values are equivalent. The comparison that is used is the same one that is used by the source-language operation `x === y`.

To improve the performance of the lookup process, the set-element blocks in each linked list are ordered by their hash numbers. When a linked list of set-element blocks is examined, the search stops if a hash number of an element on the list is greater than the hash number of the value being looked up.

If the value is not found and the lookup is being performed to insert a new member, a set-element block for the new member is created and linked into the list at that point. For example,

```
insert(s, -39)
```

inserts a set-element block for -39 at the head of the list in slot 2, since its hash value is -39. The word in the set-header block that contains the number of members is incremented to reflect the insertion.

7.1.2 Set Operations

The set operations of union, intersection, and difference all produce new sets and do not modify their arguments.

In the case of union, a copy of the larger set is made first to provide the basis for the union. This involves not only copying the set-header block but also all of its set-element blocks. These are linked together as in the original set, and no lookup is required. After this copy is made, each member of the set for the other argument is inserted in the copy, using the same technique that is used in insert. The larger set is copied, since copying

does not require lookup and the possible comparison of values that insertion does. The insertion of a member from the second set may take longer, however, since the linked lists in the copy may be longer.

In the case of intersection, a copy of the smaller argument set is made, omitting any of its members that are not in the larger set. As with union, this strategy is designed to minimize the number of lookups.

For the difference of two sets, a copy of the first argument set is made, adding only elements that are not in the second argument. This involves looking up all members in the first argument set in the second argument set.

7.2 Tables

7.2.1 Data Organization for Tables

The implementation of tables is similar to the implementation of sets, with a header block containing slots for elements ordered by hash numbers. A table-header block contains an extra descriptor for the default assigned value.

An empty table with the default assigned value 0 is produced by

```
t := table(0)
```

The structure of the table-header block is

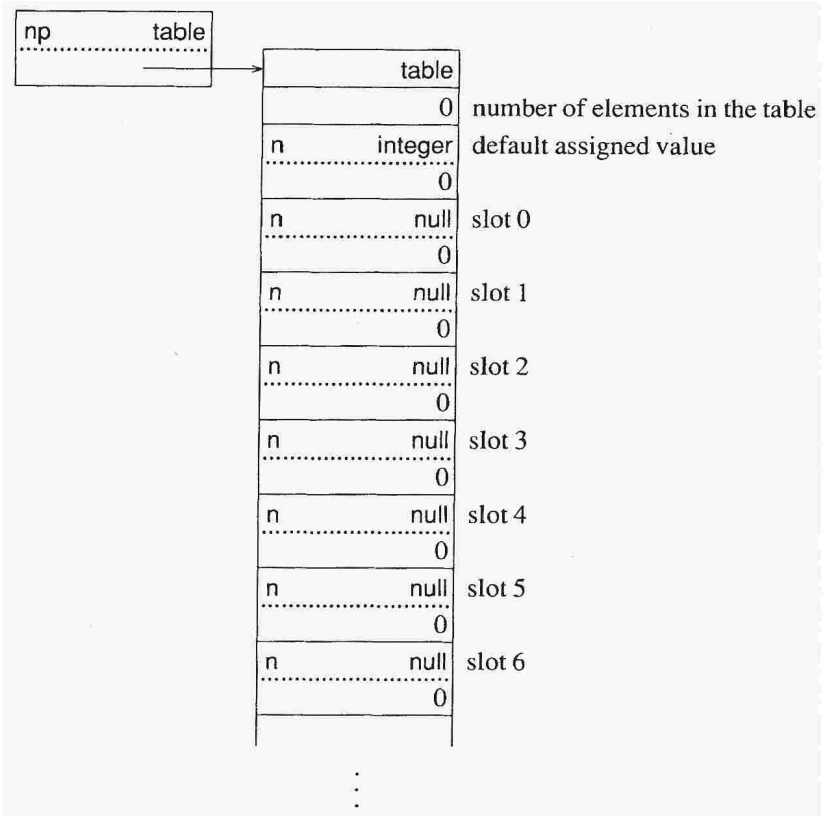
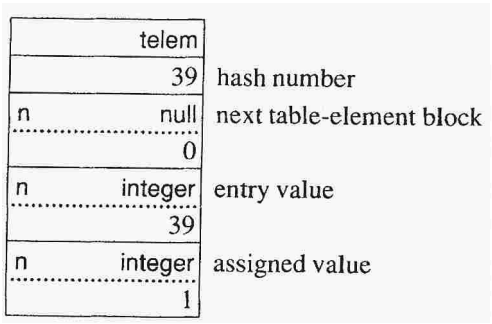


Table lookup is more complicated than set lookup, since table elements contain both an entry value and an assigned value. Furthermore, table elements can be referenced by variables. A new table element is created as a byproduct of assignment to a table reference with an entry value that is not in the table.

The result of evaluating an assignment expression such as

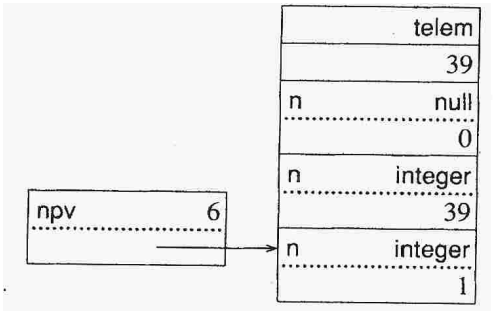
```
t[39] := 1
```

illustrates the structure of a table-element block:



In the case of a table reference such as `t[x]`, the hash number for the entry value x is used to select a slot, and the corresponding list is searched for a table-element block that contains the same entry value. As in the case of sets, comparison is first made using hash numbers; values are compared only if their hash numbers are the same.

If a table-element block with a matching entry value is found, a variable that points to the corresponding assigned value is produced. For example, if 39 is in t as illustrated previously, `t[39]` produces



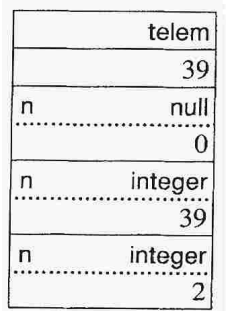
If this variable is dereferenced, as in

```
write(t[39])
```

the value 1 is written. On the other hand, if an assignment is made to this variable, as in

```
t[39] +:= 1
```

the assigned value in the table-element block is changed:

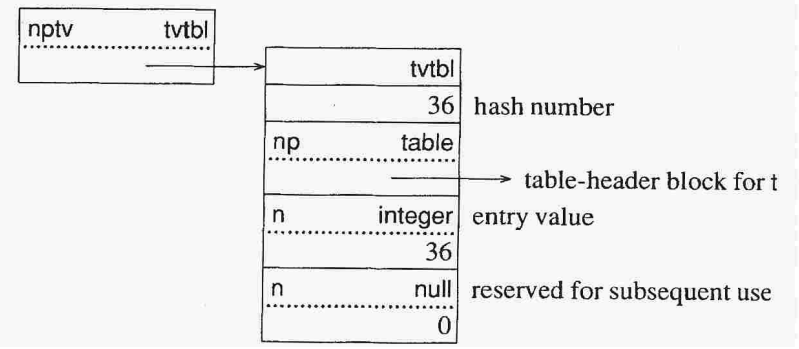


If a table element with a matching entry value is not found, the situation is very similar to that in a subscripted string: the operation to be performed depends on whether the table reference is used in a dereferencing or assignment context. In a dereferencing context, the

default value for the table is produced, while in an assignment context, a new element is added to the table.

The approach taken is similar to that for subscripted strings: a trapped variable is created. As with substring trapped variables, table-element trapped variables contain the information that is necessary to carry out the required computation for either dereferencing or assignment.

Suppose, for example, that the entry value `36` is not in the table `t`. Then `t[36]` produces the following result:



Note that the size of a table-element trapped-variable block is the same as the size of a table-element block. The last descriptor in the table-element trapped-variable block is reserved for subsequent use, as described below.

If this trapped variable is dereferenced, as in

```
write(t[36])
```

the default assigned value, 0, which is in the table-header block for `t`, is produced. Unfortunately, the situation is not always this simple. It is possible for elements to be inserted in a table between the time the table-element trapped-variable block is created and the time it is dereferenced. An example is

```
write(t[36] , t[36] := 2)
```

Since functions do not dereference their arguments until all the arguments have been evaluated, the result of dereferencing the first argument of write should be 2, not 0. In order to handle such cases, when a table-element trapped variable is dereferenced, its linked list in the table must be searched again to determine whether to return the assigned value of a newly inserted element or to return the default value.

If an assignment is made to the table reference, as in

```
t[36] +:= 1
```

the table-element trapped-variable block is converted to a table-element block with the assigned value stored in the reserved descriptor of the table-element trapped-variable block. The table-element block is then linked in the appropriate place. Note that the structures of table-element blocks and table-element trapped-variable blocks are the same, allowing this conversion without allocating a new table-element block.

It then is necessary to search the linked list for its slot again to determine the place to insert the table-element block. As in the case of dereferencing, elements may have been inserted in the table between the time the table-element trapped variable was created and the time a value is assigned to it. Normally, no matching entry is found, and the table-element trapped-variable block, transformed into a table-element block, is inserted with

the new assigned value. If a matching entry is found, its assigned value is simply changed, and the block is discarded.

Note that reference to a value that is not in a table requires only one computation of its hash value, but two lookups are required in the linked list of table-element blocks for its slot.

7.3 Hashing Functions

Ideally, a hash computation should produce a different result for every different value to which it is applied, and the distribution of the remainder on division by the number of slots should be uniform. Even approaching this ideal requires an impractical amount of computation and space. In practice, it is desirable to have a fast computation that produces few collisions.

The subject of hash computation has been studied extensively and there is a substantial body of knowledge concerning useful techniques (Knuth 1973, pp. 506-549). For example, it is known that the number of slots should be a prime that is not close to a power of two. This consideration motivated the choices of 37 and 13 for computers with large and small address spaces, respectively. In general, there is a trade-off between faster lookup, on the average, and more storage overhead.

In most situations in which hashing techniques are used, all the values for which hash computations are performed are strings. In Icon, however, any kind of value can be the member of a set or the entry value in a table. The hash computation must, therefore, apply to any type of value. The support routine for computing hash numbers is

```
uword hash(dp)
dptr dp;
   {
   register char *s;
   register uword i;
   register word j, n;
   register unsigned int *bitarr;
   double r;

   if (Qual(*dp)) {
   hashstring:
      /*
       * Compute the hash value for the string based on a scaled
       *  sum of its first ten characters, plus its length.
       */
      i = 0;
      s = StrLoc(*dp);
      j = n = StrLen(*dp);
      if (j > 10)     /* limit scan to first ten characters */
         j = 10;
      while (j-- > 0) {
         i += *s++ & 0xFF;  /* add unsigned version of char */
         i *= 37;           /* scale by a nice prime number */
         }
      i += n;               /* add (untruncated) string length */
      }

   else {
```

```
switch (Type(*dp)) {
   /*
    * The hash value of an integer is itself times eight
    *  times the golden ratio.  We do this calculation in
    *  fixed point.  We don't just use the integer itself,
    *  for that would give bad results with sets having
    *  entries that are multiples of a power of two.
    */
   case T_Integer:
      i = (13255 * (uword)IntVal(*dp)) >> 10;
      break;

   /*
    * The hash value of a bignum is based on its length and
    *  its most and least significant digits.
    */
 case T_Lrgint:
    {
    struct b_bignum *b = &BlkLoc(*dp)->bignumblk;

    i = ((b->lsd - b->msd) << 16) ^
      (b->digits[b->msd] << 8) ^ b->digits[b->lsd];
    }
    break;

   /*
    * The hash value of a real number is itself times a
    *  constant, converted to an unsigned integer.  The
    *  intent is to scramble the bits well, in the case of
    *  integral values, and to scale up fractional values
    *  so they don't all land in the same bin. The constant
    *  below is 32749 / 29, the quotient of two primes,
    *  and was observed to work well in empirical testing.
    */
   case T_Real:
      GetReal(dp,r);
      i = r * 1129.27586206896558;
      break;

   /*
    * The hash value of a cset is based on a convoluted
    *  combination of all its bits.
    */
   case T_Cset:
      i = 0;
      bitarr = BlkLoc(*dp)->cset.bits + CsetSize - 1;
      for (j = 0; j < CsetSize; j++) {
         i += *bitarr--;
         i *= 37;               /* better distribution */
         }
      i %= 1048583;        /* scramble the bits */
      break;

   /*
    * The hash value of a list, set, table, or record is
    *  its id, hashed like an integer.
    */
   case T_List:
```

```
            i = (13255 * BlkLoc(*dp)->list.id) >> 10;
            break;

         case T_Set:
            i = (13255 * BlkLoc(*dp)->set.id) >> 10;
            break;

         case T_Table:
            i = (13255 * BlkLoc(*dp)->table.id) >> 10;
            break;

         case T_Record:
            i = (13255 * BlkLoc(*dp)->record.id) >> 10;
            break;

      case T_Proc:
         dp = &(BlkLoc(*dp)->proc.pname);
         goto hashstring;

         default:
            /*
             * For other types, use the type code as the hash
             *  value.
             */
            i = Type(*dp);
            break;
         }
      }

   return i;
   }
```

To hash a string, its characters are added together as integers. At most ten characters are used, since strings can be very long and adding many characters does not improve the hashing sufficiently to justify the time spent in the computation. The maximum of ten is, however, *ad hoc.* To provide a measure of discrimination between strings with the same initial substring, the length of the string is added to the sum of the characters. This technique for hashing strings is not sophisticated, and others that produce better hashing results are known. However, the computation is simple, easy to write in C, and works well in practice.

For a numeric type, the hash value is derived from the number. In the case of a cset, the words containing the bits for the cset are combined using the exclusive-or operation.

The remaining data types pose an interesting problem. Hash computation must be based on attributes of a value that are invariant with time. Some types, such as files, have such attributes. On the other hand, there is no time-invariant attribute that distinguishes one list from another. The size of a list may change, the elements in it may change, and even its location in memory may change as the result of garbage collection. For a list, its only time-invariant attribute is its type.

This presents a dilemma–the type of such a value can be used as its hash number, but if that is done, all values of that type are in the same slot and have the same hash number. Lookup for these values degenerates to a linear search. The alternative is to add some time-invariant attribute, such as a serial number, to these values. Icon does this, at the cost of increasing the size of every such value.

RETROSPECTIVE: Few programming languages support sets or tables with Icon's generality. The implementation of sets and tables provides a clear focus on the generality of descriptors and the uniformity with which different kinds of data are treated in Icon.

Since sets and tables may be very large, efficient lookup is an important concern. The hashing and chaining technique used is only one of many possibilities. However, there must be a mechanism for determining the equivalence of values independent of the structure in which they are stored.

The fact that elements in tables are accessed by subscripting expressions introduces several complexities. In particular, the fact that the contents of the table that is subscripted may change between the time the subscripting expression is evaluated and the time it is dereferenced or assigned to introduces the necessity of two lookups for every table reference.

Hashing a variety of different types of data raises interesting issues. The hashing techniques used by Icon are not sophisticated and there is considerable room for improvement. The trade-offs involved are difficult to evaluate, however.

EXERCISES

7.1 Contrast sets and csets with respect to their implementation, their usefulness in programming, and the efficiency of operations on them.

7.2 Give an example of a situation in which the heterogeneity of sets is useful in programming.

7.3 How much space does an empty set occupy?

7.4 Diagram the structures resulting from the evaluation of the following expressions:

```
t := table()
t[t] := t
```

7.5 There are many sophisticated data structures that are designed to ensure efficient lookup in data aggregates like sets and tables (Gonnet 1984). Consider the importance of speed of lookup in sets and tables in Icon and the advantages that these more sophisticated data structures might supply.

7.6 Some of the more sophisticated data structures mentioned in the preceding exercise have been tried experimentally in Icon and either have introduced unexpected implementation problems or have not provided a significant improvement in performance. What are possible reasons for these disappointing results?

7.7 Icon goes to a lot of trouble to avoid adding table-element blocks to a table unless an assignment is made to them. Suppose a table-element block were simply added when a reference was made to an entry value that is not in the table.

- How would this simplify the implementation?
- What positive and negative consequences could this change have on the running speed and space required during program execution?
- Give examples of types of programs for which the change would have positive and negative effects on performance, respectively.

- Would this change be transparent to the Icon programmer, not counting possible time and space differences?

7.8 There is space in a table-element trapped-variable block to put the default value for the table. Why is this not done?

7.9 What is the consequence of evaluating the following expressions?

```
t := table(0)
t[37] := 2
write(t[37], t := table(1))
```

What would happen if the last line given previously were

```
write(t[37],t := list(100,3))
```

or

```
write(t[37], t := "hello")
```

7.10 Give examples of different strings that have the same hash numbers.

7.11 Design a method for hashing strings that produces a better distribution than the the current one.

7.12 What attribute of a table is time-invariant?

7.13 What kinds of symptoms might result from a hashing computation based on an attribute of a value that is not time-invariant?

Chapter 8: The Interpreter

PERSPECTIVE: The interpreter provides a software realization of Icon's virtual machine. This machine is stack-based. The basic units on which the Icon virtual machine operates are descriptors. The instructions for the virtual machine consist of operations that manipulate the stack, call C functions that carry out the built-in operations of Icon, and manage the flow of control. The Icon interpreter executes these virtual machine instructions. It consists of a loop in which a virtual machine instruction is fetched and control is transferred to a section of code to perform the corresponding operation.

8.1 Stack-Based Evaluation

Virtual machine instructions typically push and pop data on the interpreter stack. The interpreter stack, which is distinct from the stack used for calls of C functions, is an array of words. The variable `sp` points to the last word pushed on the interpreter stack. Pushing increments `int`, while popping decrements it. When the interpreter executes code that corresponds to a built-in operation in Icon, it pushes descriptors for the arguments on the interpreter stack and calls a C function corresponding to that operation with a pointer to the place on the interpreter stack where the arguments begin. A null descriptor is pushed first to serve as a "zeroth" argument (Arg0) that receives, by convention, the result of the computation and becomes the top descriptor on the stack when the C function returns. On a more conventional virtual machine, the result of the computation would be pushed on the stack, instead of being returned in an argument. The latter method is more convenient in Icon.

To illustrate this basic mechanism, consider the expression

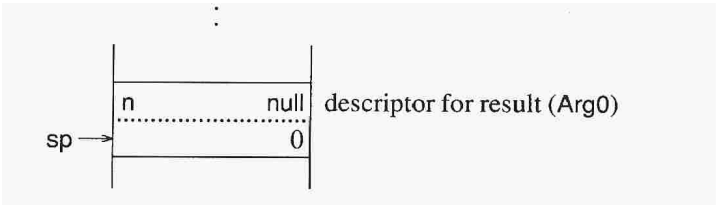
```
?10
```

which produces a randomly selected integer between 1 and 10, inclusive. The corresponding virtual machine instructions are

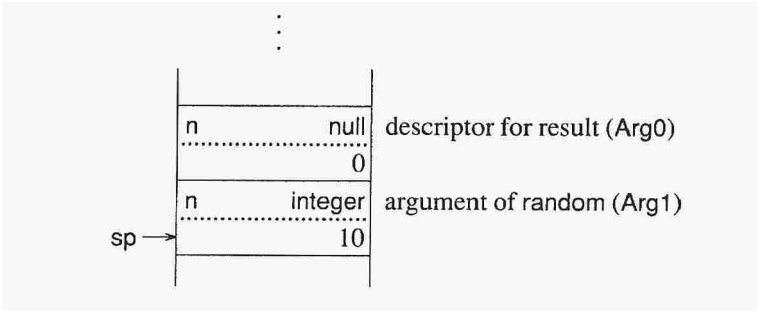
```
pnull         # push null descriptor for the result
int     10    # push descriptor for the integer 10
random        # compute random value
```

The instructions `pnull` and `int` operate directly on the stack. The instruction `random` calls a C function that computes random values.

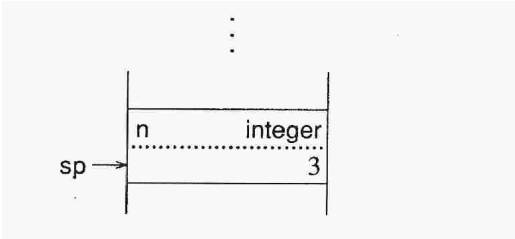
The `pnull` instruction pushes a null descriptor:



The `int` instruction pushes a descriptor for the integer 10:



Suppose that the C function for random computes 3. It replaces the null value of Arg0 by a descriptor for the integer 3. When it returns, `sp` is set to point to Arg0 and the situation is



8.2 Virtual Machine Instructions

The various aspects of expressions that appear in Icon source-language programs are reflected, directly or indirectly, in the instruction set for the Icon virtual machine. References to constants (literals) and identifiers have direct correspondences in the instruction set of the virtual machine. There is a virtual machine instruction for each source-language operator. This is possible, since the meaning of an operation is fixed and cannot be changed during program execution. The meaning of a function call, however, cannot be determined until it is evaluated, and there is a single virtual machine instruction for function invocation. The invocation of functions is described in detail in Chapter 10.

There are several virtual machine instructions related to control structures and the unique aspects of expression evaluation in Icon. These are discussed in the next two chapters. A complete list of virtual machine instructions is given in Appendix B.

8.2.1 Constants

Four kinds of data can be represented literally in Icon programs: integers, strings, csets, and real numbers. The four corresponding virtual machine instructions are

```
int   n     # integer n
str   n, a  # string of length n at address a
cset  a     # cset block at address a
real  a     # real block at address a
```

The values of integer literals appear as arguments of int instructions. In the case of strings, the two arguments give its length and the address of its first character.

The string itself is constructed by the linker and is loaded into memory from the icode file. For csets and real numbers, the linker constructs blocks, which are also loaded from

the icode file. These blocks are identical in format to blocks that are constructed during program execution.

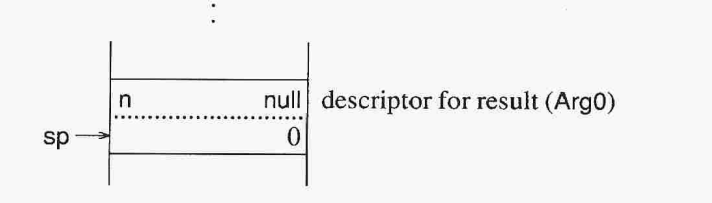
The virtual machine instructions `str`, `cset`, and `real` push appropriate descriptors to reference the data as it appears in the icode. For example, the virtual machine instructions for

```
?"aeiou"
```

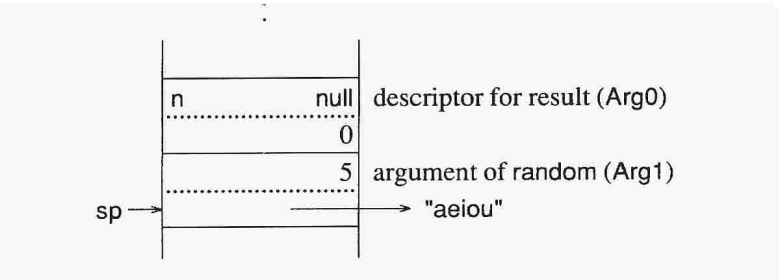
are

```
pnull
str     5, a
random
```

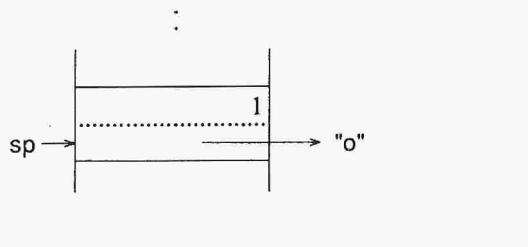
where `a` is the address of the string `"aeiou"`. The `pnull` instruction pushes a null descriptor as in the previous example:



The `str` instruction constructs a descriptor for the string `"aeiou"`:



If `random` produces the string `"o"`, this string replaces the null descriptor and the stack becomes



8.2.2 Identifiers

From the viewpoint of the interpreter, there are four kinds of identifiers: global identifiers, static identifiers, local identifiers, and arguments. The values of global and static identifiers are in arrays of descriptors at fixed locations in memory. The values of local identifiers and arguments, on the other hand, are kept on the stack as part of the infonnation associated with a procedure call.

The values of the arguments in the call of a procedure are pushed on the stack as the result of the evaluation of expressions prior to the invocation of the procedure. The initial null values for local identifiers are pushed on the stack when the procedure is called.

The portion of the stack between the arguments and local identifiers is fixed in size and contains information that is saved when a procedure is called. This information is described in Chapter 10.

There are four virtual machine instructions for constructing variable descriptors:

```
global n
static n
arg n
local n
```

Identifiers of each kind are numbered starting at zero. Consequently,

```
arg 0
```

pushes a variable descriptor for the first argument. In each case, the descriptor that is pushed on the stack is a variable that points to the descriptor for the value of the corresponding identifier.

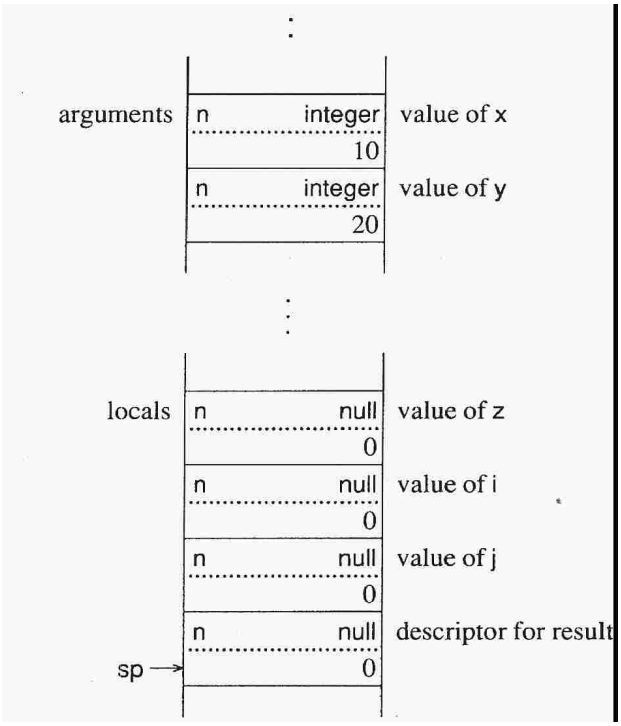
Consider the expression

```
j := 1
```

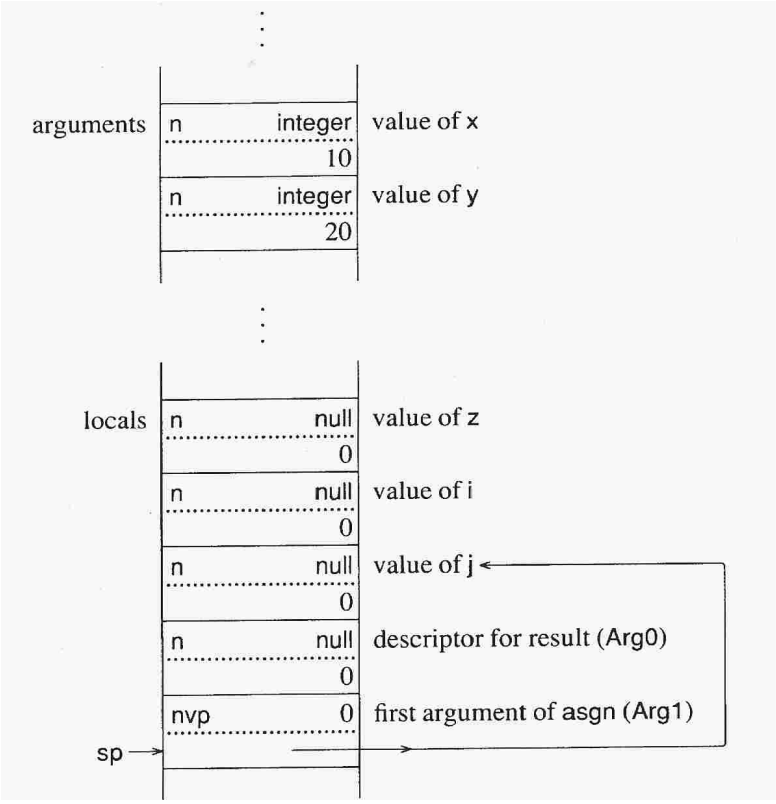
The corresponding virtual machine instructions are

```
pnull         # push null descriptor for the result
local   2     # push variable descriptor for j
int     1     # push descriptor for the integer 1
asgn          # perform assignment
```

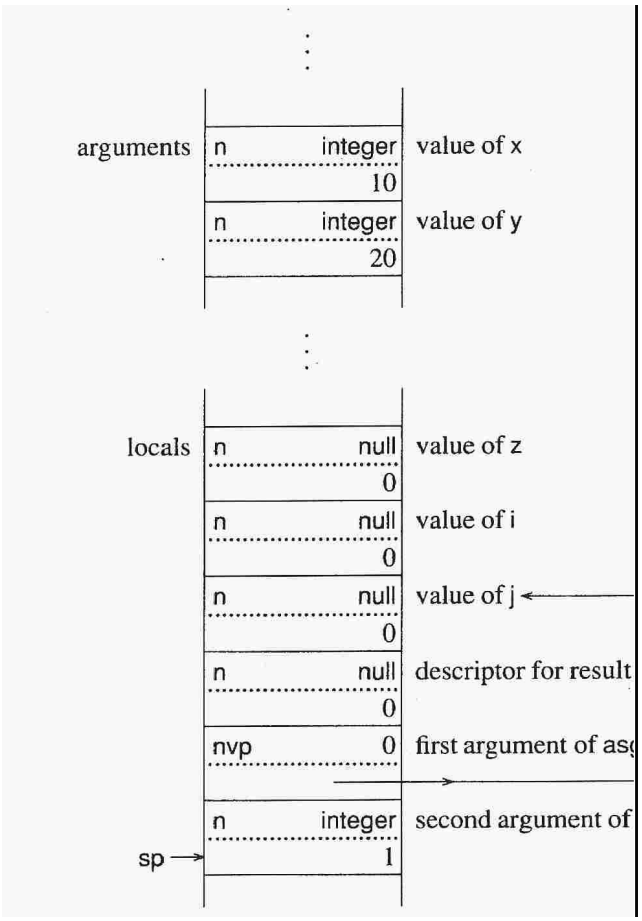
When these instructions are interpreted. the succession of stack states is



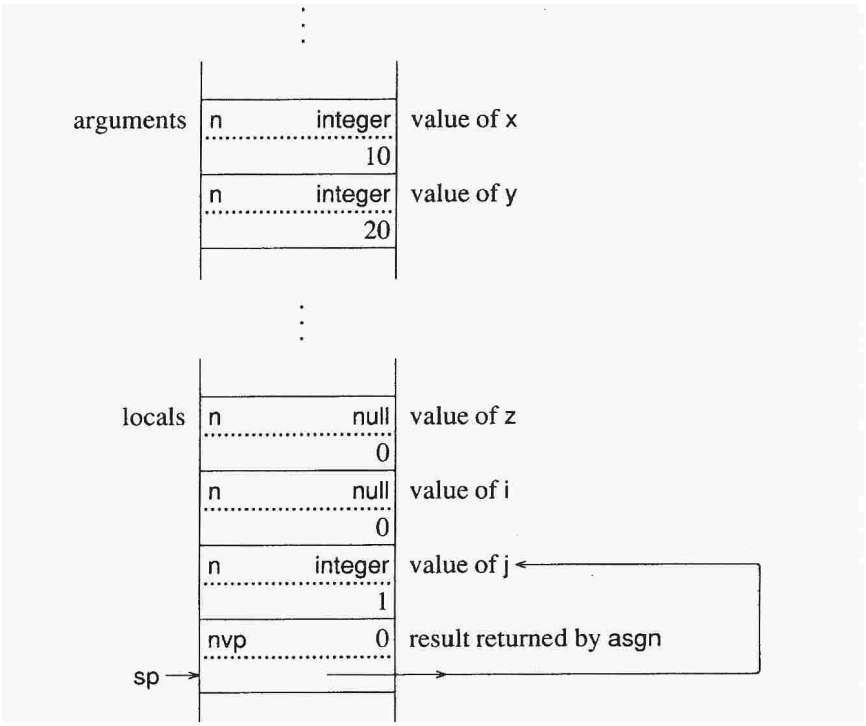
The Stack after `pnull`



The Stack after `local 2`



The Stack after `int 1`



The Stack after `asgn`

Note that `asgn` assigns the value of its second argument to `j` and overwrites Arg0 with a variable descriptor, which is left on the top of the stack.

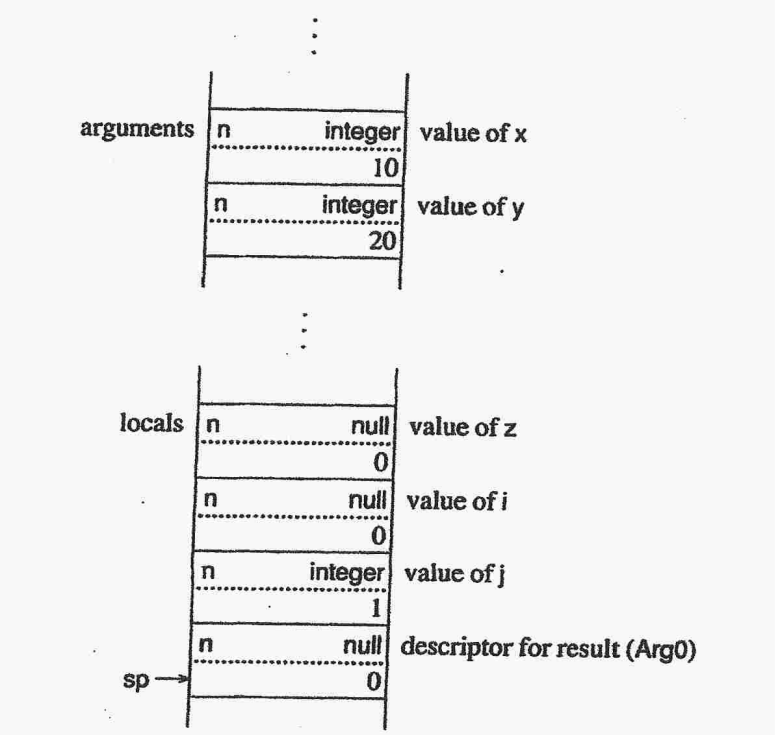
Similarly, the virtual machine instructions for

```
z := x
```

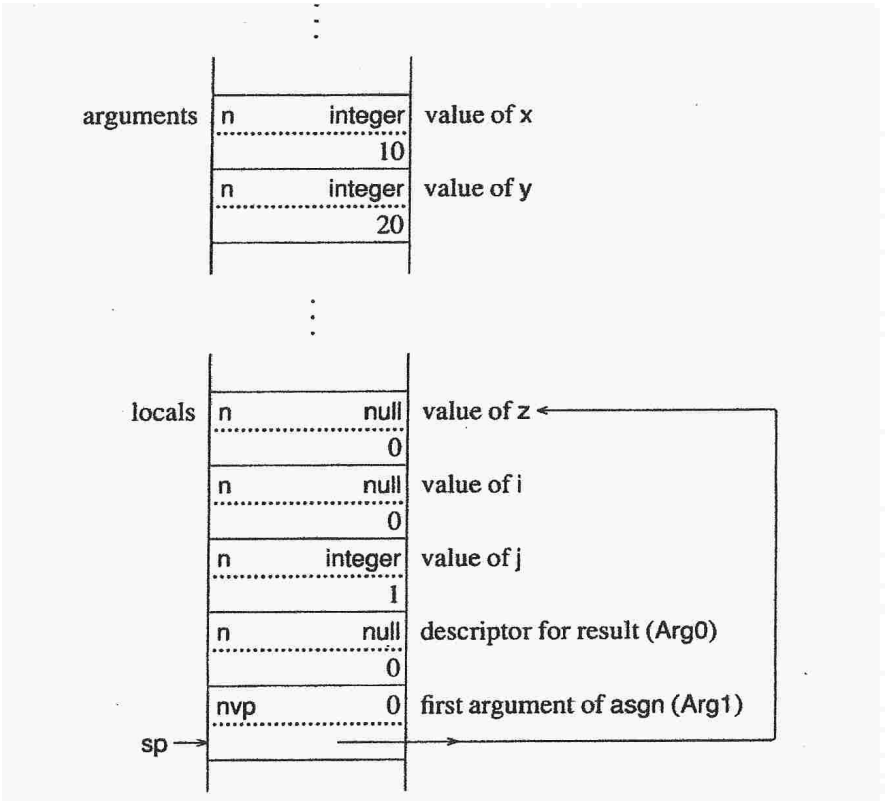
are

```
pnull
local 0
arg 0
asgn
```

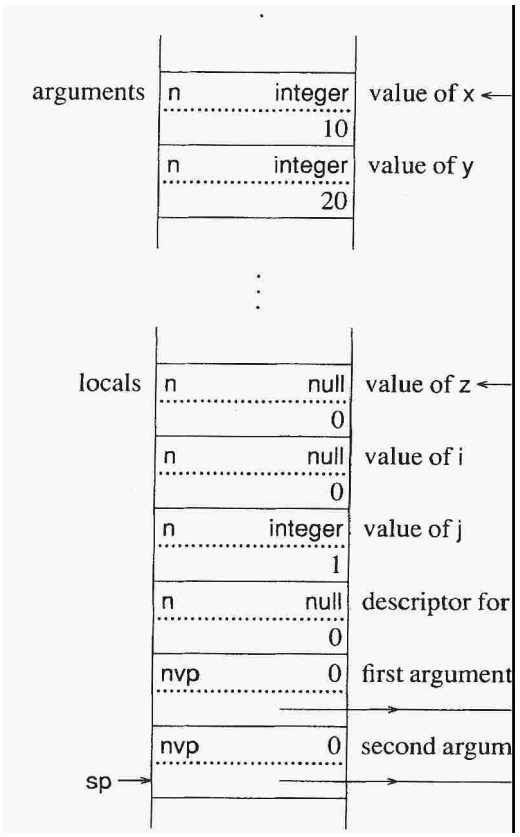
the states of the stack are



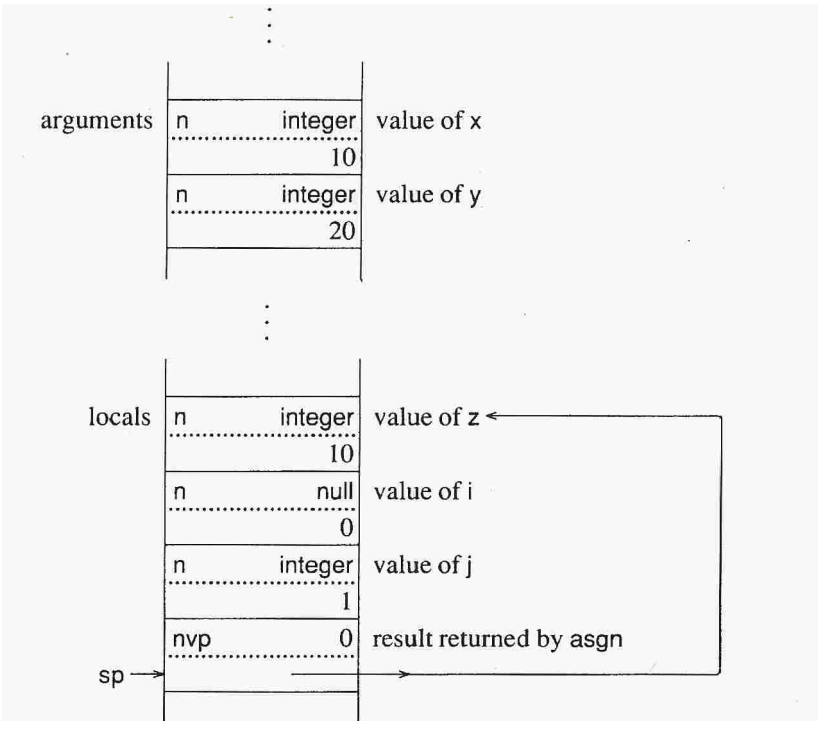
The Stack after `pnull`



The Stack after `local 0`



The Stack after `arg 0`



The Stack after `asgn`

8.3 **Operators**

There is a virtual machine instruction for each of the forty-two operators in Icon. The instructions random and asgn described previously are examples. Casting Icon operators as virtual machine instructions masks a considerable amount of complexity, since few Icon operators are simple. For example, although x + y appears to be a straightforward computation, it involves checking the types of x and y, converting them to numeric types if they are not already numeric, and terminating with an error message if this is not possible. If x and y are numeric or convertible to numeric, addition is performed. Even this is not simple, since the addition may be integer or floating-point, depending on the types of the arguments. For example, if x is an integer and y is a real number, the integer is converted to a real number. None of these computations is evident in the virtual machine instructions produced for this expression. which are

```
pnull
local x
local y
plus
```

In the instructions given previously, the indices that are used to access identifiers have been replaced by the names of the identifiers, which are assumed to be local. This convention is followed in subsequent virtual machine instructions fo ease of reading.

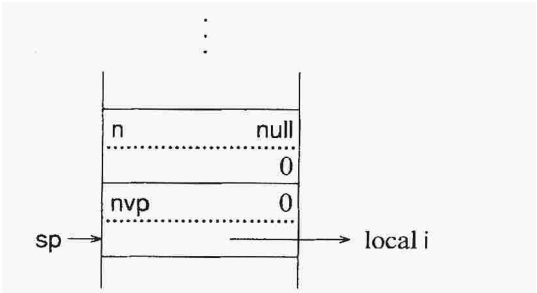
Augmented assignment operations do not have separate virtual machine instructions. Instead, the instruction `dup` first pushes a null descriptor and then pushes a duplicate of the descriptor that was previously on top of the stack. For example, the virtual machine instructions for

```
i +:= 1
```

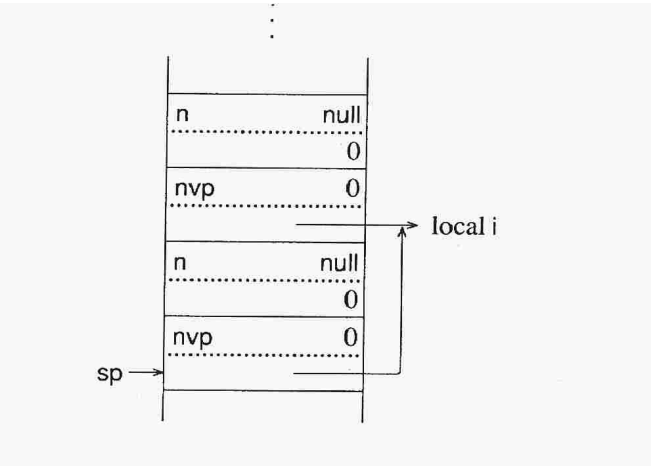
are

```
pnull
local i
dup
int 1
plus
asgn
```

The stack after the execution of `local` is



The execution of `dup` produces



The `dup` instruction simply takes the place of the `pnull` and second `local` instructions in the virtual machine instructions for

```
i := i + 1
```

which are

```
pnull
local i
pnull
local i
int 1
plus
asgn
```

In this case, only a single `local` instruction is avoided. If the variable to which the assignment is made is not just an identifier but, instead, a more complicated construction, as in

```
a[j] +:= 1
```

substantial computation may be saved by duplicating the result of the first argument expression instead of recomputing it.

8.2.4 Functions

While the meaning of an operation is fixed and can be translated into a specific virtual machine instruction, the meaning of a function call can change during program execution. The value of the function also can be computed. as in

```
(p[i])(x, y)
```

The general form of a call is

```
expr0(expr1, expr2, ..., exprn)
```

The corresponding virtual machine instructions are

```
code for expr0
code for expr1
code for expr2
code for exprn
invoke n
```

The `invoke` instruction is relatively complicated, since the value of *expr0* may be a procedure, an integer (for mutual evaluation), or even a value that is erroneous. Function invocation is discussed in detail in Chapter 10.

8.3 The Interpreter Proper

8.3. 1 The Interpreter Loop

The interpreter, which is called `interp()`, is basically simple in structure. It maintains a location in the icode (`ipc`) and begins by fetching the instruction pointed to by `ipc` and incrementing `ipc` to the next location. It then branches to a section of code for processing the virtual machine instruction that it fetched. The interpreter loop is

```
for (;;) {
   op = GetWord;
   switch (op) {
      case Op_Asgn:
      case Op_Plus:
      }
   }
```

where `GetWord` is a macro that is defined to be `(*ipc++)`.

Macros are used extensively in the interpreter to avoid repetitious coding and to make the interpreter easier to read. The coding is illustrated by the case clause for the instruction `plus`:

```
case Op_Plus:      /* e1 + e2 */
   Setup_Op(2);
   DerefArg(1);
   DerefArg(2);
   Call_Op;
   break;
```

`Setup_Op(n)` sets up a pointer to the address of Arg0 on the interpreter stack. The resulting code is

```
rargp = (dptr)(sp - 1) - n;
```

The value of `n` is the number of arguments on the stack.

`DerefArg(n)` dereferences argument `n`. If it is a variable, it is replaced by its value. Thus, dereferencing is done in place by changing descriptors on the interpreter stack.

`Call_Cond` calls the appropriate C function with a pointer to the interpreter stack as provided by `Setup_Op(n)`. The function itself is obtained by looking up `op` in an array of pointers to functions. The code produced by `Call_Cond` is (almost)

```
(*(optab(op]) )(rargp);
sp = (word * )rargp + 1:
```

Chapter 9: Expression Evaluation

PERSPECTIVE: The preceding chapter presents the essentials of the interpreter and expression evaluation as it might take place in a conventional programming language in which every expression produces exactly one result. For example, expressions such as

```
i := j
k := i + j
i +:= ?k
```

each produce a single result: they can neither fail nor can they produce sequences of results.

The one feature of Icon that distinguishes it most clearly from other programming languages is the capacity of its expression-evaluation mechanism to produce no result at all or to produce more than one result. From this capability come unconventional methods of controlling program flow, novel control structures, and goal-directed evaluation.

The generality of this expression-evaluation mechanism alone sets Icon apart from other programming languages. While generators, in one form or another, exist in a number of programming languages, such as IPL-V (Newell 1961), CLU (Liskov 1981), Alphard (Shaw 1981), and SETL (Dewar, Schonberg, and Schwartz 1981), such generators are limited to specific constructs, designated contexts, or restricted types of data. Languages with pattern-matching facilities, such as SNOBOL4 (Griswold, Poage, and Polonsky 1971), InterLisp (Teitelman 1974), and Prolog (Clocksin and Mellish 1981), generate alternative matches, but only within pattern matching.

Just as Icon's expression-evaluation mechanism distinguishes it from other programming languages, it is also one of the most interesting and challenging aspects of Icon's implementation. Its applicability in every context and to all kinds of data has a pervasive effect on the implementation.

9.1 Bounded Expressions

A clear understanding of the semantics of expression evaluation in Icon is necessary to understand the implementation. One of the most important concepts of expression evaluation in Icon is that of a *bounded expression,* within which backtracking can take place. However, once a bounded expression has produced a result, it cannot be resumed for another result. For example, in

```
write(i = find(s1,s2))
```

find may produce a result and may be resumed to produce another result if the comparison fails. On the other hand, in

```
write(i = find(s1, s2))
write(j = find(s1, s3))
```

the two lines constitute separate expressions. Once the evaluation of the expression on the first line is complete, it cannot be resumed. Likewise, the evaluation of the expression on the second line is not affected by whether the expression on the first line succeeds or fails. However, if the two lines are joined by a conjunction operation, as in

```
write(i = find(s1, s2)) &
write(i = find(s1, s3))
```

they are combined into a larger single expression and the expression on the second line is not evaluated if the expression on the first line fails. Similarly, if the expression on the first line succeeds, but the expression on the second line fails, the expression on the first line is resumed.

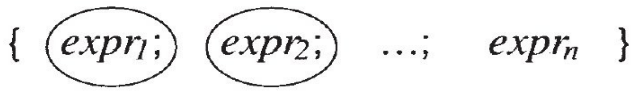
The reason for the difference in the two cases is obscured by the fact that the Icon translator automatically inserts a semicolon at the end of a line on which an expression is complete and for which a new expression begins on the next line.

Consequently, the first example is equivalent to

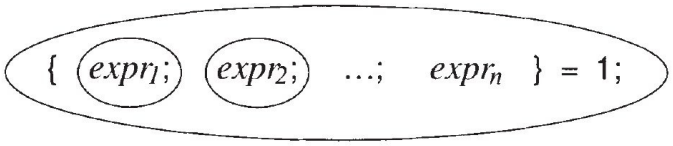
```
write(i = find(s1, s2));
write(i = find(s1 , s3))
```

The difference between the semicolon and the conjunction operator is substantial. A semicolon bounds an expression, while an operator binds its operands into a single expression.

Bounded expressions are enclosed in ovals in the following examples to make the extent of backtracking clear. A compound expression, for example, has the following bounded expressions:



Note that *exprn* is not, of itself, a bounded expression. However, it may be part of a larger bounded expression. as in

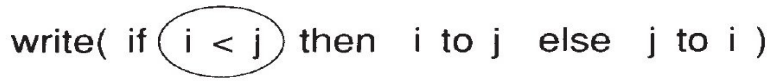


Here *exprn* is part of the bounded expression for the comparison operator. The entire enclosing bounded expression is a consequence of the final semicolon. In the absence of the context provided by this semicolon, the entire expression might be part of a larger enclosing bounded expression, and so on.

The separation of a procedure body into a number of bounded expressions, separated by semicolons (explicit or implicit) and other syntactic constructions, is very important. Otherwise, a procedure body would consist of a single expression, and failure of any component would propagate throughout the entire procedure body. Instead, control backtracking is limited in scope to abounded expression, as is the lifetime (and hence stack space) for temporary computations.

Bounded expressions are particularly important in control structures. For example, in the if-then-else control structure, the control expression is bounded but the other expressions are not:

As with the compound expression illustrated earlier, *expr2* or *exp13* (whichever is selected) may be the part of a larger bounded expression. An example is

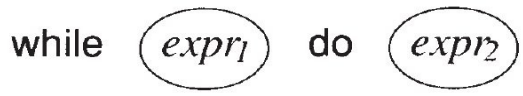


If the control expression were not a separate bounded expression, the failure of *expr2* or *exp13* would result in backtracking into it and the if-then-else expression would be equivalent to

```
(expr1 & expr2) | expr3
```

which is hardly what is meant by if-then-else.

In a while-do loop, the control expression and the expression in the do clause are both bounded:



The two bounded expressions ensure that the expressions are evaluated independently of each other and any surrounding context. For example, if *expr2* fails, there is no control backtracking into *expr,*

9.1.1 Expression Frames

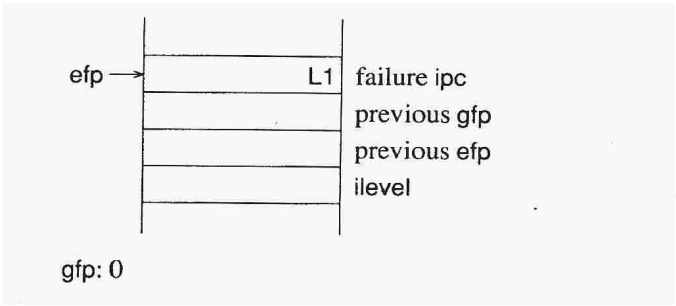
In the implementation of Icon, the scope of backtracking is delineated by *expression frames*. The virtual machine instruction

```
mark L1
```

starts an expression frame. If the subsequent expression fails, `ipc` is set to the location in the icode that corresponds to `L1`. The value of `ipc` for a label is relative to the location of the icode that is read in from the icode file. For simplicity in the description that follows, the value of `ipc` is referred to just by the name of the corresponding label.

The `mark` instruction pushes an *expression frame marker* onto the stack and sets the expression frame pointer, `efp`, to it. Thus, `efp` indicates the beginning of the current expression frame. There is also a generator frame pointer, `gfp`, which points to another kind of frame that is used to retain information when an expression suspends with a result and is capable of being resumed for another. Generator frames are described in Sec. 9.3. The mark instruction sets `gfp` to zero, indicating that there is no suspended generator in a new expression frame.

An expression frame marker consists of four words: the value `ipc` for the argument of mark (called the failure `ipc`), the previous `efp`, the previous `gfp`, and `ilevel`, which is related to suspended generators:



An expression frame marker is declared as a C structure:

```
struct ef_marker {      /* expression frame marker */
   word *ef_failure;          /*   failure ipc */
   struct ef_marker *ef_efp;  /*   efp */
   struct gf_marker *ef_gfp;  /*   gfp */
   word ef_ilevel;            /*   ilevel */
```

This structure is overlaid on the interpreter stack in order to reference its components. The code for the `mark` instruction is

```
case Op_Mark:      /* create expression frame marker */
   newefp = (struct ef_marker *)(sp + 1);
   opnd = GetWord;
   opnd += (word)ipc;
   newefp->ef_failure = (word *)opnd;
   newefp->ef_gfp = gfp;
   newefp->ef_efp = efp;
   newefp->ef_ilevel = ilevel;
   sp += Wsizeof(*efp);
   efp = newefp;
   gfp = 0;
   break;
```

The macro `Wsizeof(x)` produces the size of `x` in words.

An expression frame is removed by the virtual machine instruction

```
unmark
```

which restores the previous `efp` and `gfp` from the current expression frame marker and removes the current expression frame by setting `sp` to the word just above the frame marker.

The use of `mark` and `unmark` is illustrated by

```
if expr1 then expr2 else expr3
```

for which the virtual machine instructions are

```
      mark L1
      code for expr1
      unmark
      code for expr2
      goto L2
L1:
      code for expr3
L2:
```

The `mark` instruction creates an expression frame for the evaluation of *expr1*. If *expr1* produces a result, the `unmark` instruction is evaluated, removing the expression frame for *expr1,* along with the result produced by *expr1*. Evaluation then proceeds in *expr2*.

If *expr1* fails, control is transferred to the location in the icode corresponding to `L1` and the `unmark` instruction is not executed. In the absence of generators, failure also removes the current expression frame, as described in Sec. 9.2.

It is necessary to save the previous value of `efp` in a new expression marker, since expression frames may be nested. This occurs in interesting ways in some generative control structures, which are discussed in Sec. 9.4. Nested expression frames also occur as a result of evaluating compound expressions, such as

```
while expr1 do ifexpr2thenexpr2
```

9.2 Failure

The interesting aspects of implementing expression evaluation in Icon can be divided into two cases: without generators and with generators. The possibility of failure in the absence of generators is itself of interest, since it occurs in other programming languages, such as SNOBOL4. This section describes the handling of failure and assumes, for the moment, that there are no generators. The next section describes generators.

In the absence of generators, if failure occurs anywhere in an expression, the entire expression fails without any further evaluation. For example, in the expressions

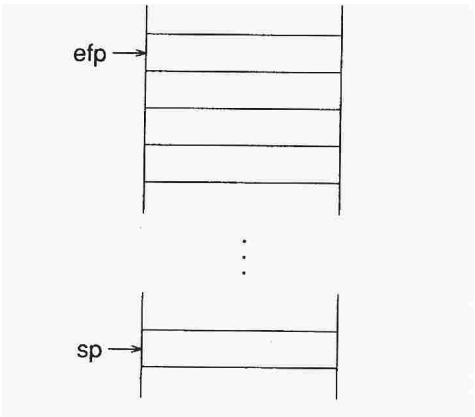
```
i := numeric(s)
line := read(f)
```

if `numeric(s)` fails in the first line, the assignment is not performed and evaluation continues immediately with the second line. In the implementation, this amounts to removing the current expression frame in which failure occurs and continuing with `ipc` set to the failure `ipc` from its expression frame marker.

The virtual machine instructions for the previous example are

```
   mark L1
   pnull
   local i
   global numeric
   local s
   invoke 1
   asgn
   unmark
L1:
   mark L2
   pnull
   local line
   global read
   local f
   invoke 1
   asgn
   unmark
L2:
```

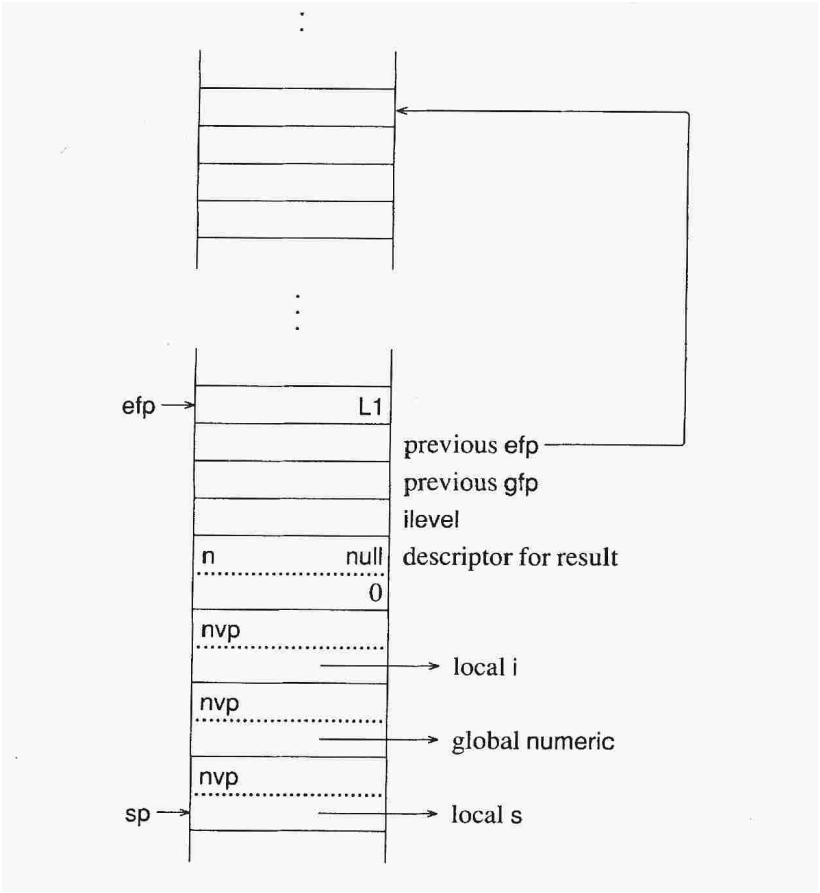
Prior to the evaluation of the expression on the first line, there is some expression frame on the stack:



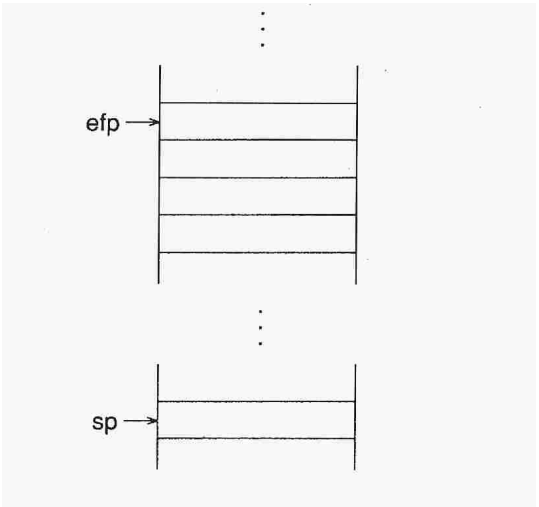
The instruction

```
mark L1
```

starts a new expression frame. The execution of subsequent virtual machine instructions pushes additional descriptors. The state of the stack when numeric is called by the `invoke` instruction is



If `numeric(s)` fails, `efp` and `sp` are reset, so that the stack is in the same state as it was prior to the evaluation of the expression on the first line:



Control is transferred to the location in the icode corresponding to `L1`, and the execution of

```
mark L2
```

starts a new expression frame by pushing a new expression frame marker onto the stack.

It is worth noting that failure causes only the current expression frame to be removed and changes `ipc` to the failure `ipc`. Any remaining virtual machine instructions in the current expression frame are bypassed; failure is simple and quick.

Failure can occur at three levels: directly from the virtual machine instruction `efail`, from a C function that implements an operator or function (as in the previous example), or from an Icon procedure.

When a conditional operator or function returns, it signals the interpreter, indicating whether it is producing a result or failing by using one of the RTL forms of return, `return` or `fail`. These RTL constructs simply produce return statements with different returned values.

The code in the interpreter for a conditional operation is illustrated by

```
case Op_Numlt:     /* e1 < e2 */
   Setup_Op(2);
   DerefArg(1);
   DerefArg(2) ;
   Call_Cond;
```

The macro `Call_Cond` is similar to `Call_Op` described in Sec. 8.3.1, but it tests the signal returned by the C function. If the signal corresponds to the production of a result, the break is executed and control is transferred to the beginning of the interpreter loop to fetch the next virtual machine instruction. On the other hand, if the signal corresponds to failure, control is transferred to the place in the interpreter that handles failure, `efail`.

An Icon procedure can fail in three ways: by evaluating the expression fail, by the failure of the argument of a return expression, or by flowing off the end of the procedure body. The virtual machine instructions generated for the three cases are similar. For example, the virtual machine instructions for

```
if i < j then fail else write(j)
```

are

```
   mark L1
   pnull
   local i
   local j
   numlt
   unmark
   pfail
L1:
   global write
   local j
   invoke 1
```

The virtual machine instruction `pfail` first returns from the current procedure call (see Sec. 10.3), and then transfers to `efail`.

9.3 Generators and Goal-Directed Evaluation

The capability of an expression not to produce a result is useful for controlling program flow and for bypassing unneeded computation, but generators add the real power and expressiveness to the expression-evaluation semantics of Icon. It should be no surprise that generators also present difficult implementation problems. There are several kinds of generators, including those for control structures, functions and operators, and

procedures. While the implementation of the different kinds of generators varies in detail, the same principles apply to all of them.

As far as using a result of an expression in further computation is concerned, there is no difference between an expression that simply produces a result and an expression that produces a result and is capable of being resumed to produce mother one. For example, in

```
i := numeric("2")
j := upto('aeiou', "Hello world")
```

the two assignment operations are carried out in the same way, even though `upto()` is a generator and `numeric()` is not.

Since such contexts cannot be determined, in general, prior to the time the expressions are evaluated, the implementation is designed so that the interpreter stack is the same, as far as enclosing expressions are concerned, whether an expression returns or suspends. For the previous example, the arguments to the assignment operation are in the same relative place in both cases.

On the other hand, if a generator that has suspended is resumed, it must be capable of continuing its computation and possibly producing another result. For this to be possible, both the generator's state and the state of the interpreter stack must be preserved. For example, in

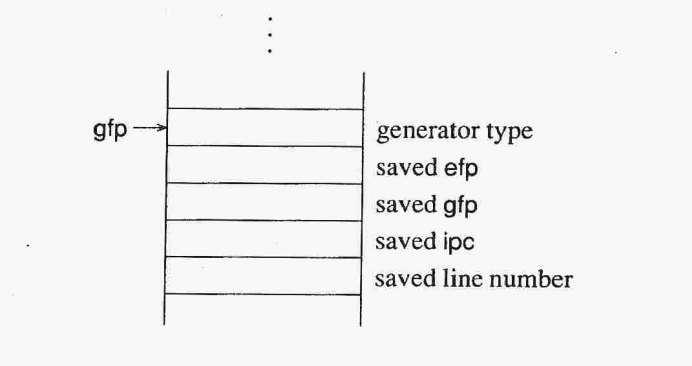
```
j := (i < upto('aeiou', "Hello world"))
```

when the function `upto()` suspends, both `i` and the result produced by `upto()` must be on the stack as arguments of the comparison operation. However, if the comparison operation fails and `upto()` is resumed, the arguments of `upto()` must be on the stack as they were when `upto()` suspended. To satisfy these requirements, when `upto()` suspends, a portion of the stack prior to the arguments for `upto()` is copied to the top of the stack and the result produced by upto is placed on the top of the stack. Thus, the portion of the stack required for the resumption of `upto()` is preserved and the arguments for the comparison are in the proper place.

Generator Frames. When an expression suspends, the state of the interpreter stack is preserved by creating a *generator frame* on the interpreter stack that contains a copy of the portion of the interpreter stack that is needed if the generator is resumed. A generator frame begins with a generator frame marker that contains information about the interpreter state that must be restored if the corresponding generator is resumed. There are three kinds of generator frames that are distinguished by different codes:

`G_Csusp` suspension from a C function
`G_Esusp` suspension from an alternation expression
`G_Psusp` suspension from a procedure

For the first two types of generators, the information saved in the generator frame marker includes the code for the type of the generator, the i-state variables `efp`, `gfp`, `ipc`, and the source-program line number at the time the generator frame is created:



The corresponding C structure is

```
struct gf_marker { /* generator frame marker */
   word gf_gentype;     /*   type */
   struct ef_marker *gf_efp;  /*   efp */
   struct gf_marker *gf_gfp;  /*   gfp */
   word *gf_ipc;   /*   ipc */
   word gf_line;   /*   line number */
};
```

Generators for procedure suspension contain, in addition, the i-state variable related to procedures. See Sec. 10.3.3.

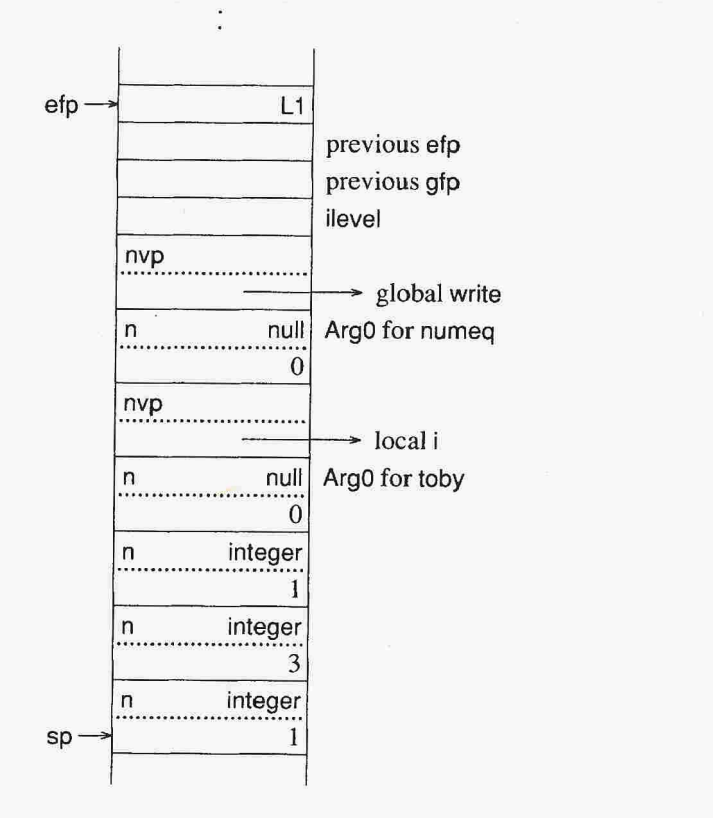
As an example, consider the expression

```
write(i = (1 to 3));
```

The virtual machine instructions for this expression are:

```
   mark L1
   global write
   pnull
   local i
   int 1
   int 3
   push 1  # default increment
   toby
   numeq
   invoke 1
   unmark
L1 :
```

The state of the stack after execution of the first seven instructions is



The code in the interpreter for calling a generative operator with n arguments is

```
rargp = (struct descrip *)(sp -1) -n;
signal = (*(optab[op]))(rargp);
goto C_rtn_term;
```

Note that `rargp` points to Arg0 and is the argument of the call to the C function for the operator.

The RTL function for `toby` is

```
operator{*} ... toby(from, to, by)
   /*
    * arguments must be integers.
    */
   if !cnv:C_integer(from) then
      runerr(101, from)
   if !cnv:C_integer(to) then
      runerr(101, to)
   if !cnv:C_integer(by) then
      runerr(101, by)
   abstract {
      return integer
      }
   inline {
      /*
       * by must not be zero.
       */
      if (by == 0) {
         irunerr(211, by);
         errorfail;
         }
      /*
```

```
       * Count up or down (depending on relationship of from and
       *  to) and suspend each value in sequence, failing when
       *  the limit has been exceeded.
       */
      if (by > 0)
         for ( ; from <= to; from += by) {
            suspend C_integer from;
            }
      else
         for ( ; from >= to; from += by) {
            suspend C_integer from;
            }
      fail;
      }
end
```

The RTL `operator` construct, which is similar to `function`, produces the C header

```
int Otoby(dptr r_args)
```

so that `toby` is called with a pointer to Arg0. Arguments with logical names Arg0, Arg1, and so forth are referred as `r_args[0]`, `r_args[1]`, and so on in the generated code.

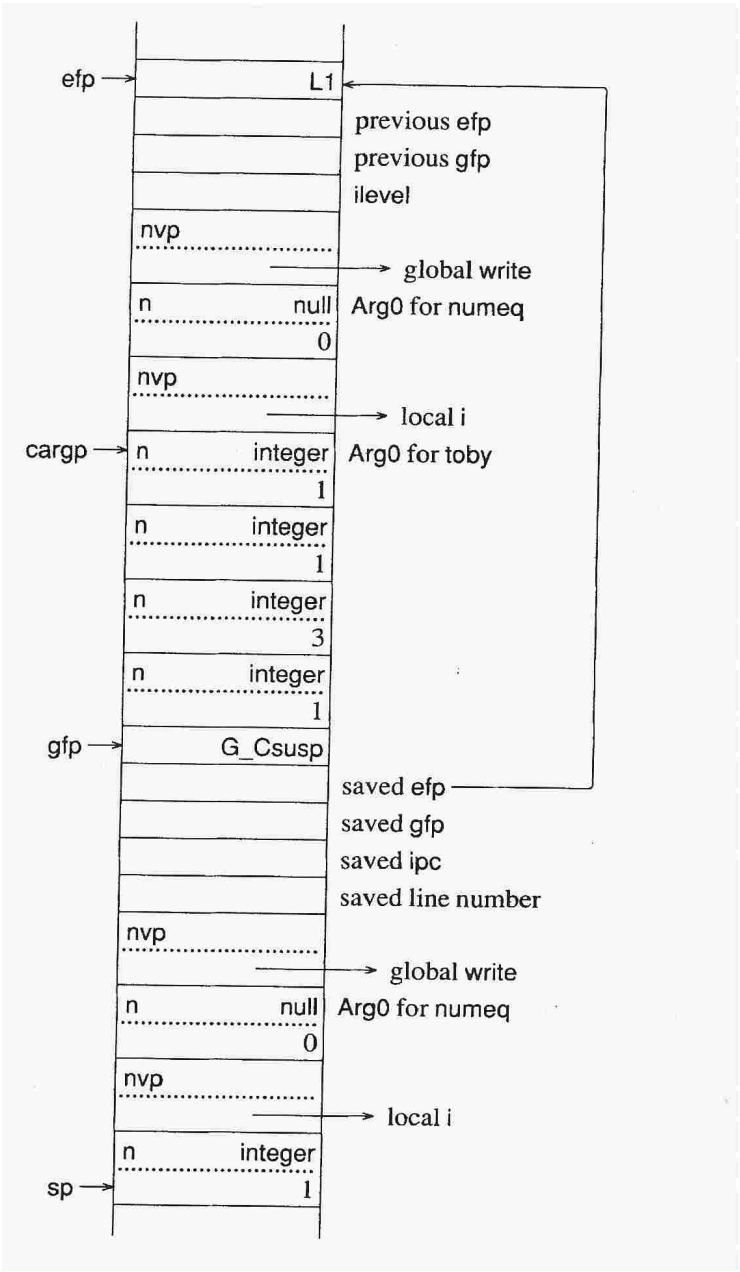
When `toby` is called, it replaces its Arg0 descriptor by a descriptor for the integer `from` and suspends by using the RTL `suspend` construct rather than `return`.

The `suspend` statement *calls* `interp()` instead of returning to it. This leaves the call of `toby` intact with its variables preserved and also transfers control to `interp()` so that the next virtual machine instruction can be interpreted. However, it is necessary to push a generator marker on the interpreter stack and copy a portion of the interpreter stack, so that interpretation can continue without changing the portion of the interpreter stack that `toby` needs in case it is resumed. This is accomplished by calling `interp()` with arguments that signal it to build a generator frame. The C code generated for `suspend` is

```
   if ((signal = interp(G_Osusp, r_args RTTCURTSTATARG)) !=
        A_Resume) {
      return signal;
      }
```

The argument `G_Osusp` in the call of `interp()` indicates that a generator frame for C function that implements an operator is needed. The argument `r_args` points to the location on the interpreter stack where Arg0 for the suspending C function is located. This location is the same as rargp in the call of `interp()` that called upto.

In this situation, `interp()` puts a generator frame marker on the interpreter stack and copies the portion of the interpreter stack from the last expression 01 generator frame marker through cargp onto the top of the interpreter stack:



The stack is exactly the same, as far as the execution of `numeq` is concerned, as it would have been if `toby` had simply returned. However, the arguments of `toby` (and the preceding arguments of `numeq`) are still intact, so that `toby` can be resumed. The generator frame is interposed between the two portions of the interpreter stack. The top of the stack corresponds to the evaluation of

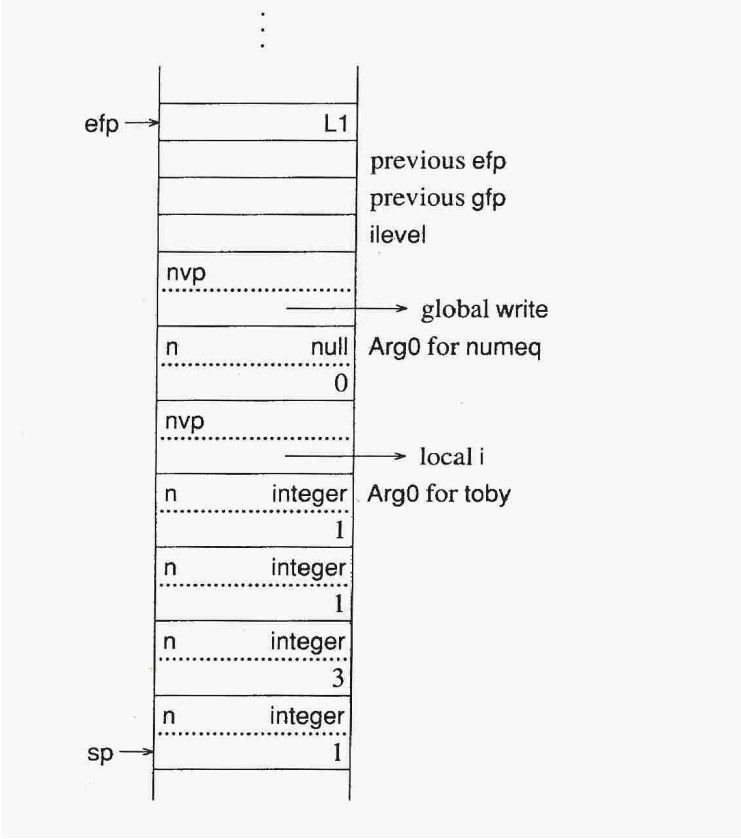
```
write(i = 1);
```

Resumption. Suppose the value of `i` in the previous example is 2. The comparison fails and control is transferred to `efail`, as it is in the case of all operations that fail. The code for `efail` is

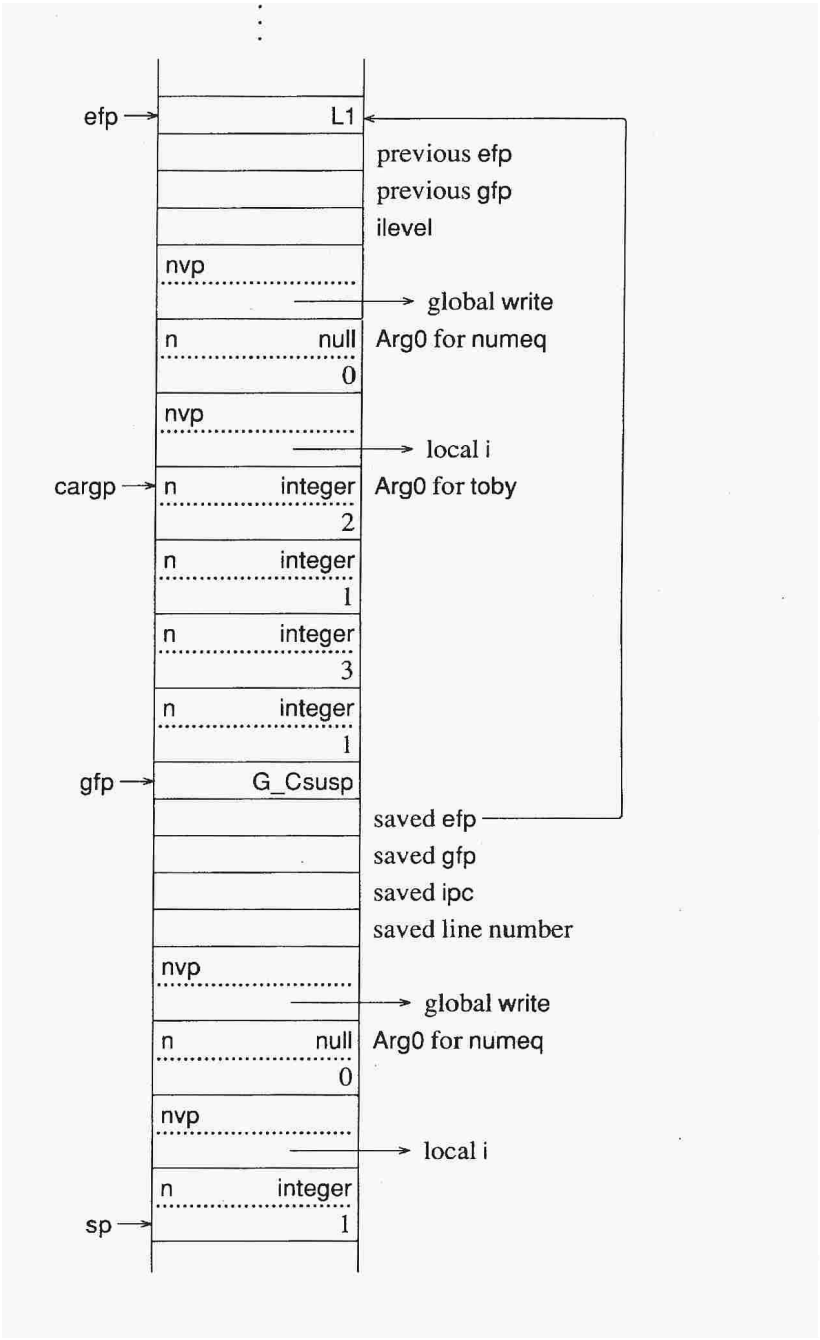
```
case Op_Efail:
efail:
   /*
    * Failure has occurred in the current expression frame.
    */
   if (gfp == 0) {
```

```
      /*
       * There are no suspended generators to resume. Remove
       * the current expression frame, restoring values.
       *
       * If the failure ipc is 0, propagate failure to the
       * enclosing frame by branching back to efail.
       * This happens, for example, in looping control
       * structures that fail when complete.
       */
      ipc = efp->ef_failure;
      gfp = efp->ef-9fp;
      sp = (word *)efp -1;
      efp = efp->ef_efp;
      if (ipc == 0)
         goto efail;
      break;
      }
   else {
      /*
       * There is a, generator that can be resumed. Make
       * the stack adjustments and then switch on the
       * type of the generator frame marker.
       */
      register struct gf_marker *resgfp = gfp;
      tvoe = resgfp->gf gentype;
      ipc = resgfp->gf_ipc;
      efp = resgfp->gf_efp;
      line = resgfp->gf_line;
      gfp = resgfp->gf_gfp;
      sp = (word * )resgfp -1;
      switch (type) {
      case G_Csusp: {
         --ilevel;
         return A_Resumption;
         break;
         }
      case G_Esusp:
         goto efail;
      case G_Psusp:
         break;
      }
   break;
   }
```

If there were no generator frame (if `gfp` were 0), the entire expression frame would be removed, and the expression would fail as described in Sec. 9.2. However, since there is a C_Susp generator frame, the stack is restored to the state it is in when `toby` suspended, and the values saved in the generator frame marker are *restored:*



All traces of the first execution of `numeq` have been removed from the stack. As shown by the code for `efail`, the call to `toby` is resumed by *returning* to it from `interp()` with the signal `A_Resumption`, which indicates another result is needed. When control is returned to `toby`, it changes its Arg0 descriptor to the integer 2 suspends again:



The interpreter stack is exactly as it was when `toby` suspended the first time, except that the integer 2 is on the stack in place of the integer 1. The top of the stack corresponds to the evaluation of

```
write(i = 2);
```

Since the value of `i` is 2, `numeq` succeeds. It copies the value of its second argument to its Arg0 descriptor and returns. The value 2 is written and the `unmark` instruction is executed, removing the entire expression frame from the stack.

Goal-Directed Evaluation. Goal-directed evaluation occurs when an expression fails and there are generator frames on the interpreter stack as the consequence of expressions that have suspended.

In the case of an expression such as

```
1 to upto(c, s)
```

`upto()` suspends first, followed by `toby`. These generator frames are linked together, with `gfp` pointing to the one for `toby`, which in turn contains a pointer to the one for `upto()`. In general, generator frames are linked together with `gfp` pointing to the one for the most recent suspension. This produces the last-in, first-out (depth-first) *order of* expression evaluation in Icon. Goal-directed evaluation occurs as a result of resuming a suspended expression when failure occurs in the surrounding expression frame.

Removing C Frames. Since C functions that suspend call the interpreter and the interpreter in turn calls C functions, expression evaluation typically results in a sequence of frames for calls on the C stack. When the evaluation of a bounded expression is complete, there may be frames on the C stack for generators, even though these generators no longer can be resumed.

In order to "unwind" the C stack in such cases, the i-state variable `ilevel` is used to keep track of the level of call of `interp()` by C functions. Whenever `interp()` is called, it increments `ilevel`. When an expression frame is created, the current value of `ilevel` is saved in it, as illustrated previously.

When the expression frame is about to be removed, if the current value of `ilevel` is greater than the value in the current expression frame, `ilevel` is decremented and the interpreter *returns* with a signal to the C function that called it to return rather than to produce another result. If the signal returned by `interp()` is `A_Resumption`, the C function continues execution, while for any other signal the C function returns.

Since C functions return to `interp()`, `interp()` always checks the signal returned to it to determine if it produced a result or if it is unwinding. If it is unwinding, `interp()` returns the unwinding signal instead of continuing evaluation of the current expression.

Consider again the expression

```
write(i = (1 to 3));
```

for which the virtua1 machine instructions are

```
   mark L1
   global write
   pnull
   local i
   int 1
   int 3
   push1   # default increment
   toby
   numeq
   invoke 1
   unmark
L1 :
```

When `toby` produces a result, it calls `interp()`. When the `unmark` instruction is executed, the C stack contains a frame for the call to `toby` and for its call to `interp()`. The code for `unmark` is

```
   case Op_Unmark:    /* remove expression frame */
      gfp = efp->ef-9fp;
      sp = (word *)efp -1;
      /*
       * Remove any suspended C generators.
       */
```

```
   Unmark_uw:
      if (efp->ef_ilevel < ilevel) {
         --ilevel;
         return A_Unmark_uw;
         }
      efp = efp->ef_efp;
      break;
```

Note that in this case Suspend gets the return code `A_Unmark_uw` and in turn returns `A_Unmark_uw` to `interp()`. The section of code in `interp()` that checks the signal that is returned from C functions is

```
   C_rtn_term:
      switch (signal) {
      case A_Failure:
         goto efail;
      case A_Unmark_uw:    /* unwind for unmark */
         goto Unmark uw;
      case A_Lsusp_uw:     /* unwind for Isusp */
         goto Lsusp_uw;
      case A_Eret_uw: /* unwind for eret */
         goto Eret_uw;
      case A_Pret_uw: /* unwind for pret */
         goto Pret_uw;
      case A_Pfail_uw:     /* unwind for pfail */
         goto Pfail_uw;
      }
      sp = (word * )rargp + 1;   /* set sp to result */
      continue;
   }
```

Thus, when `interp()` returns to a C function with an unwinding signal, there is a cascade of C returns until `ilevel` is the same as it was when the current expression frame was created. Note that there are several cases in addition to `unmark` where unwinding is necessary.

9.4 Generative Control Structures

In addition to functions and operators that may generate more than one result, there are several generative control structures at the level of virtual machine instructions.

9.4.1 Alternation

The virtual machine instructions for

expr₂ | *expr₃*

are

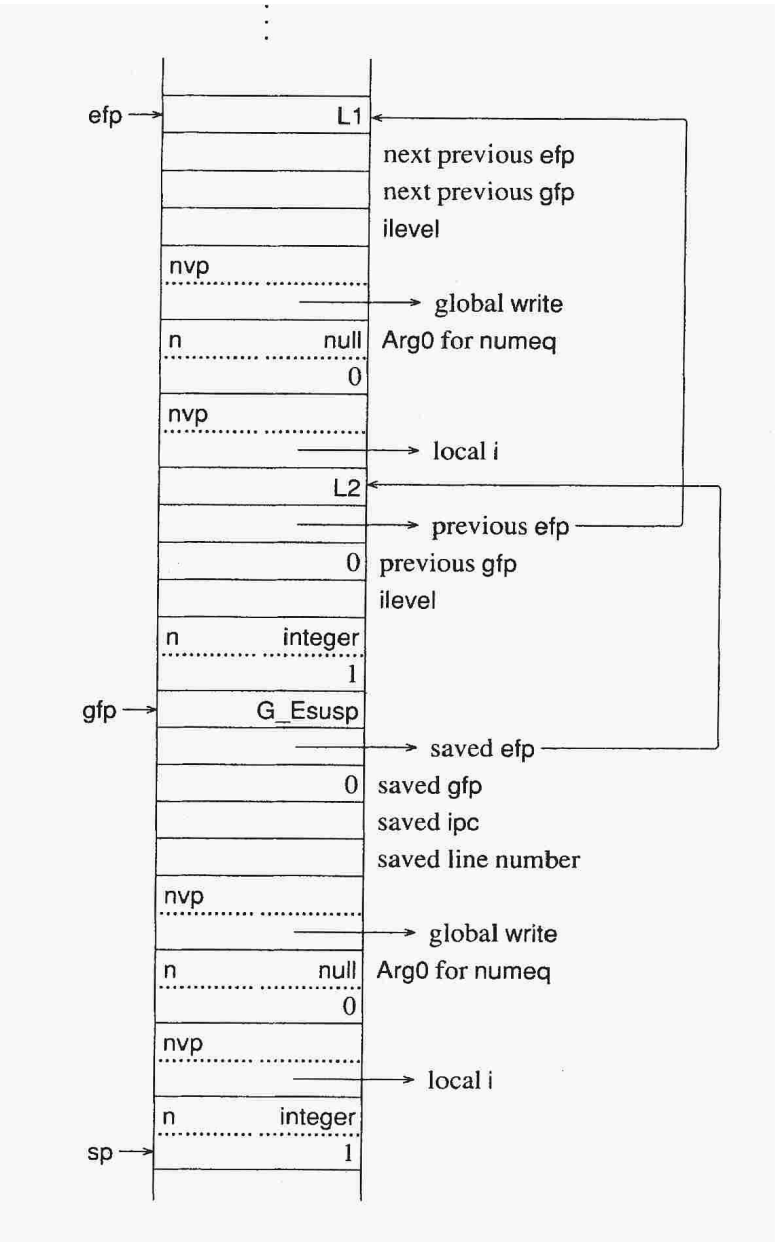
```
   mark L1
   code for expr2
   esusp
   goto L2
L1:
   code for expr3
L2:
```

The mark instruction creates an expression frame marker for alternation whose purpose is to preserve the failure `ipc` for `L1` in case the results for *expr3* are needed. If *expr2* produces a result, `esusp` creates a generator frame with the usual marker and then copies

the portion of the interpreter stack between the last expression or generator frame marker and the alternation marker to the top of the stack. It then pushes a copy of the result produced by *expr2*. This connects the result produced by *expr2* with the expression prior to the alternation control structure. Next, `esusp` sets `efp` to point to the expression frame marker prior to the alternation marker. For example, in the expression

```
write(i := 1 | 2)
```

the stack after the execution of `esusp` is



The top portion of the stack is the same as if *expr2* had produced a result in the absence of alternation. However, the generator frame marker pushed by `esusp` contains a pointer to the alternation marker.

If another result from *expr2* is needed, the generator frame left by `esusp` is removed, restoring the stack to its state when *expr2* produced a result. If *expr2* itself was a generator that suspended, it is resumed. Otherwise, control is transferred to `efail` and `ipc` is set to a value corresponding to `L1`, so that *expr3* is evaluated next.

9.4.2 Repeated Alternation

Alternation is the general model for generative control structures. Repeated alternation, `|`*expr,* is similar to alternation, and would be equivalent to

```
expr | expr | expr | ...
```

except for a special termination condition that causes repeated alternation to stop if *expr* does not produce a result. Without this termination condition, an expression such as

```
|upto(c, s)
```

would never return if `upto()` failed–expression evaluation would vanish into a "black hole." Expressions that produce results at one time but not at another also are useful. For example,

```
|read()
```

generates the lines from the standard input file. Because of the termination condition, this expression terminates when the end of the input file is reached. If it vanished into a "black hole," it could not be used safely.

If it were not for the termination condition, the virtual machine instructions for `|`*expr* would be

```
L1:
      mark L1
      code for expr
      esusp
```

The "black hole" here is evident---if *expr* fails, it is evaluated again and there is no way out.

The termination condition is handled by an instruction that changes the failure `ipc` in the current expression marker. The actual virtual machine instructions for `|`*expr* are

```
L1:
      mark0
      code for expr
      chfail    L1
      esusp
```

The virtual machine instruction `mark0` pushes an expression frame marker with a zero failure `ipc`. If a zero failure `ipc` is encountered during failure, as illustrated by the code for `efail` in Sec. 9.3, failure is transmitted to the enclosing expression. If *expr* produces a result, however, the `chfail` instruction is executed. It changes the failure `ipc` in the current expression marker to correspond to `L1`, so that if *expr* does not produce a result when it is resumed, execution starts at the location in the icode corresponding to `L1` again, causing another iteration of the alternation loop. It is important to realize that `chfail` only changes the failure `ipc` in the current expression marker on the stack. Subsequent execution of `mark0` creates a new expression frame whose marker has a zero failure `ipc`.

9.4.3 Limitation

In the limitation control structure,

```
expr1 \ expr2
```

the normal left-to-right evaluation of Icon is reversed and *expr2* is evaluated first. The virtual machine instructions are

```
      code for expr2
```

```
limit
code for expr1
lsusp
```

If *expr2* succeeds, its result is on the top of the stack. The limit instruction checks this result to be sure that it is legal---an integer greater than or equal to zero. If it is not an integer, an attempt is made to convert it to one. If the limit value is zero, limit fails. Otherwise, limit creates an expression frame marker with a zero failure `ipc` and execution continues, so that *expr1* is evaluated in its own expression frame. During the evaluation of *expr1,* the limit value is directly below its expression marker. For example. in

```
expr1 \ 10
```

the stack prior to the evaluation of *expr1* is

If *expr1* produces a result, `lsusp` is executed. The `lsusp` instruction is very similar to `esusp`. Before producing a generator frame, however, `lsusp` decrements the limit value. If it becomes zero, the expression frame for *expr1* is removed, the C stack is unwound, and the last value it produced is placed on the stack in place of the limit value. Otherwise, it copies the portion of the interpreter stack between the end of the last expression or generator frame marker and the limit value to the top of the stack. Note that no generator frame is needed.

9.5 Iteration

The difference between evaluation and resumption in a loop is illustrated by the virtual machine instructions for a conventional loop

```
while expr1 do expr2
```

and the iteration control structure

```
every expr1 do expr2
```

The instructions for `while-do` are

```
L1:
      mark0
      code for expr1
      unmark
      mark L1
      code for expr2
      unmark
      goto L1
```

If *expr1* fails, the entire expression fails and failure is transmitted to the enclosing expression frame because the failure `ipc` is zero. If *expr1* produces a result, *expr2* is evaluated in a separate expression frame. Whether *expr2* produces a result or not, its expression frame is removed and execution continues at the beginning of the loop.

The instructions for `every-do` are

```
      mark0
      code for expr1
      pop
      mark0
      code for expr2
      unmark
```

```
efail
```

If *expr1* fails, the entire expression fails and failure is transmitted to the enclosing expression frame as in the case of `while-do`. If *expr1* produces a result, it is discarded by the `pop` instruction, since this result is not used in any subsequent computation. The expression frame for *expr1* is not removed, however, and *expr2* is evaluated in its own expression frame within the expression frame for *expr1* (unlike the case for the while loop). If *expr2* produces a result, its expression frame is removed and `efail` is executed. If *expr2* fails, it transmits failure to the enclosing expression frame, which is the expression frame for *expr1*. If *expr2* produces a result, `efail` causes failure in the expression frame for *expr1*. Thus, the effect is the same, whether or not *expr2* produces a result. All results are produced simply by forcing failure.

If the expression frame for *expr1* contains a generator frame, which is the case if *expr1* suspended, evaluation is resumed accordingly, so that *expr1* can produce another result. If *expr1* simply produces a result instead of suspending, there is no generator frame, `efail` removes its expression frame, and failure is transmitted to the enclosing expression frame.

9.6 String Scanning

String scanning is one of the most useful operations in Icon. Its implementation, however, is comparatively simple. There is no special expression-evaluation mechanism associated with string scanning *per se;* all "pattern matching" follows naturally from goal-directed evaluation.

The string-scanning keywords, `&subject` and `&pos` must be handled properly, however. These keywords have global scope with respect to procedure invocation, but they are maintained in a stack-like fashion with respect to string-scanning expressions.

The expression

```
expr1 ? expr2
```

is a control structure, not an operation, since, by definition, the arguments for ar operation are evaluated before the operation is performed. This form of evaluation does not work for string scanning, since after *expr1* is evaluated, but before *expr2* is evaluated, the previous values of `&subject` and `&pos` must be saved and new ones established. Furthermore, when string scanning is finished, the old values of `&subject` and `&pos` must be restored. In addition, if string scanning succeeds, the values of these keywords at the time string scanning produces a result must be saved so that they can be restored if the string-scanning operation is resumed to produce another result.

The virtual machine instructions for

```
expr1 ? expr2
```

are

```
code for expr1
bscan
code for expr2
escan
```

If *expr1* succeeds, it leaves a result on the top of the stack. The `bscan` instruction assures that this value is a string, performing a conversion if necessary. Otherwise, the old values

of &subject and &pos are pushed on the stack, the value of &subject is set to the (possibly converted) one produced by *expr1,* and &pos is set to 1.

The bscan instruction then suspends. This is necessary in case *expr2* fails, so that bscan can get control again to perform data backtracking, restoring the previous values of &subject and &pos from the stack where they were saved.

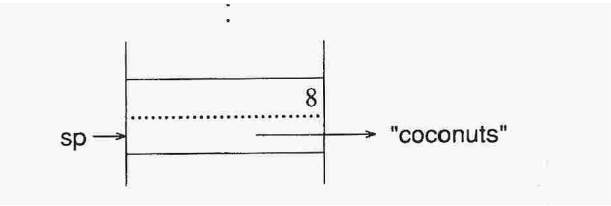
If *expr2* succeeds, the escan instruction copies the descriptor on the top of the stack to its Arg0 position, overwriting the result produced by *expr2.* It then exchanges the current values of &subject and &pos with those saved by bscan thus restoring the values of these keywords to their values prior to the scanning expression and at the same time saving the values they had at the time *expr2* produced a result. The escan instruction then suspends.

If escan is resumed, the values of &subject and &pos are restored from the stack, restoring the situation to what it was when *expr2* produced a result. The escan instruction then fails in order to force the resumption of any suspended generators left by *expr2.*

Suppose, for example, that the values of &subject and &pos are "the" and 2 respectively, when the following expression is evaluated:

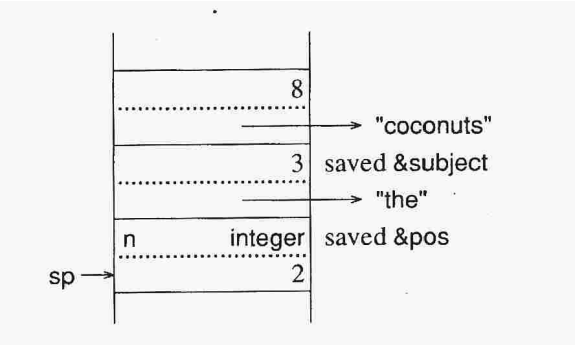
```
read(f) ? move(4)
```

Suppose read(f) produces the string "coconuts". The stack is



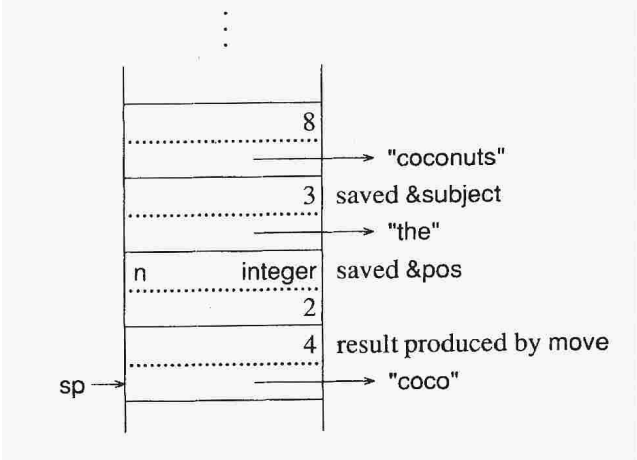

```
&subject: "the"
&pos: 2
```

The bscan instruction is executed, pushing the current values of &subject and &pos:



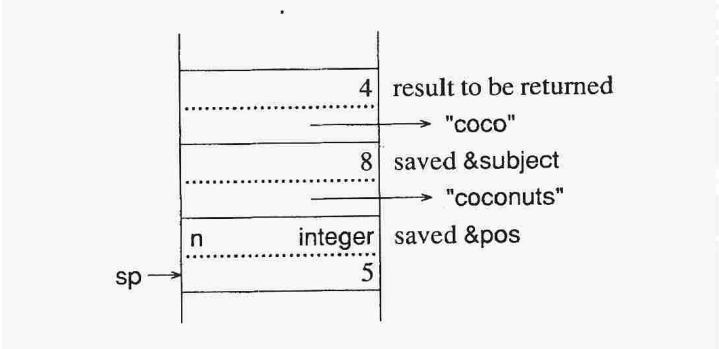

```
&subject: "the"
&pos: 2
```

The bscan instruction sets &subject to "coconuts" and &pos to 1. The bscan instruction then suspends and move(4) is evaluated. It suspends, and the top I



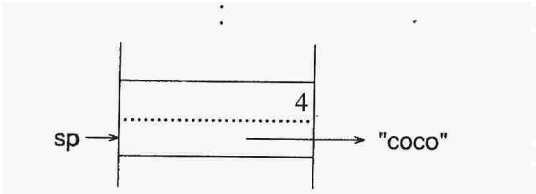

```
&subject: "coconuts"
&pos: 5
```

The `escan` instruction is executed next. It copies the descriptor on the top of the stack to replace the result produced by *expr2*. It then exchanges the current values of `&subject` and `&pos` with those on the stack:




```
&subject: "the"
&pos: 2
```

The `escan` instruction then suspends, building a generator frame. The result of *expr2* is placed on the top of the stack, becoming the result of the entire scanning expression.



Since `escan` suspends, the saved values of `&subject` and `&pos` are preserved in a generator frame on the stack until `escan` is resumed or until the current expression frame is removed.

RETROSPECTIVE: The implementation of expression evaluation in Icon focuses on the concept of an expression frame within which control backtracking can occur. Bounded expressions, for example, are represented on the stack by expression frames, which confine backtracking.

In the absence of generators, failure simply results in the removal of the current expression frame and transfer to a new location in the icode, bypassing instructions that otherwise would have been executed.

State information must be saved when a generator suspends, so that its evaluation can be resumed. This information is preserved in a generator frame within the current expression frame. Generator frames are linked together in a last-in, first-out fashion. Goal-directed evaluation is a natural consequence of resuming the most recently suspended generator when an expression fails, instead of simply removing the current expression frame.

String scanning involves saving and restoring the values of `&subject` and `&pos`. This is somewhat complicated, since scanning expressions can suspend and be resumed, but string scanning itself introduces nothing new into expression evaluation: generators and goal-directed evaluation provide "pattern matching."

EXERCISES

9.1 Circle all the bounded expressions in the following segments of code:

```
while line := read() do
   if *line = i then write(line)

if (i = find(s1,s2)) & (j = find(s1,s3)) then {
   write(i)
   write(j)
   }

line ? while write(move(1)) do
      move(1)
```

9.2 Describe the effect of nested generators and generators in mutual evaluation on the interpreter level.

9.3 Consider a hypothetical control structure called *exclusive alternation* that is the same as regular alternation, except that if the first argument produces least one result, the results from the second argument are not produced. Show the virtual machine instructions that should be generated for exclusive alternation.

9.4 The expression `read(f)` is an example of an expression that may produce result at one time and fail at another. This is possible because of a side effect of evaluating it---changing the position in the file f. Give an example of an expression that may fail at one time and produce a result at a subsequent time.

9.5 There are potential "black holes" in the expression-evaluation mechanism of Icon, despite the termination condition for repeated alternation. Give an example of one.

9.6 The expression frame marker produced by limit makes it easy to locate the limitation counter. Show how the counter could be located without the marker.

9.7 Suppose that the virtual machine instructions for

```
every expr1 do expr2
```

did not pop the result produced by *expr1*. What effect would this have?

9.8 The virtual machine instructions for

```
every expr
```

are

```
mark0
code for expr
pop
```

```
efail
```

so that failure causes *expr* to be resumed. The keyword `&fail` also fails, so that the virtual machine instructions for

```
expr & &fail
```

are

```
code for expr
efail
```

It is sometimes claimed that these two expressions are equivalent. If this were so, the shorter virtual machine instruction sequence for the second expression could be used for the first expression. Explain why the two expressions are not equivalent, in general, and give an example in which they are different.

9.9 Diagram the states of the stack for the example given in Sec. 9.6, showing all generator frames.

9.10 Show the successive stack states for the evaluation of the following expressions, assuming that the values of `&subject` and `&pos` are `"the"` and 2 respectively, and that `read()` produces `"coconuts"` in each case:

(a) read(f) ? move(10)
(b) (read(f) ? move(4)) ? move(2)
(c) read(f) ? (move(4) ? move(2))
(d) (read(f) ? move(4)) ? move(10)
(e) (read(f) ? move(4 | 6)) ? move(5)
(f) (read(f) ? move(4)) & (read(f) & move(10))

9.11 Write Icon procedures to emulate string scanning. *Hint:* consider the virtual machine instructions for

```
expr1 ? eXpr2
```

Chapter 10: Functions, Procedures, and Co-Expressions

PERSPECTIVE: The invocation of functions and procedures is central to the evaluation of expressions in most programming languages. In Icon, this activity has several aspects that complicate its implementation. Functions and procedures are data values that can be assigned to identifiers, passed as arguments, and so on. Consequently, the meaning of an invocation expression cannot be determined until it is evaluated. Functions and procedures can be called with more or fewer arguments than expected. Thus, there must be a provision for adjusting argument lists at run time. Since mutual evaluation has the same syntax as function and procedure invocation, run-time processing of such expressions is further complicated.

Co-expressions, which require separate stacks for their evaluation, add complexities and dependencies on computer architecture that are not found elsewhere in the implementation.

10.1 Invocation Expressions

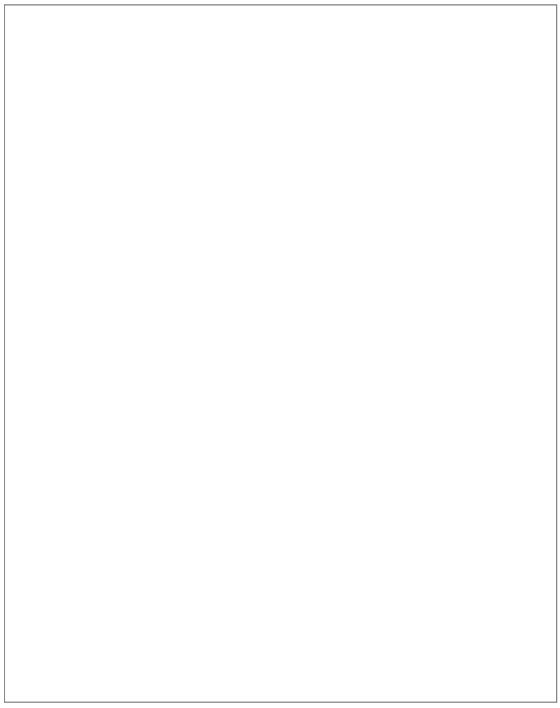
As mentioned in Sec. 8.2.4, the virtual machine code for an expression such as

```
expr0(expr1, expr2, ..., exprn)
```

is

```
code for expr0
code for expr1
code for expr2
...
code for exprn
invoke n
```

Consequently, the stack when the `invoke` instruction is executed is



The meaning of the expression, and hence the action taken by `invoke`, depends on the result produced by *expr0*. If the value of *expr0* is an integer or convertible to an integer, the invocation expression corresponds to mutual evaluation. If this integer is negative, it is converted to the corresponding positive value with respect to the number of arguments, If the value is between one and n, the corresponding descriptor is copied on top of the result of *expr0,* `sp` is set to this position, and `invoke` transfers control to the beginning of the interpretive loop. On the other hand, if the value is out of range, `invoke` fails. Note that the returned value overwrites the descriptor for *expr0,* whereas for operators a null-valued descriptor is pushed to receive the value.

If the value of *expr0* is a function or a procedure, the corresponding function or procedure must be invoked with the appropriate arguments. A function or procedure value is represented by a descriptor that points to a block that contains information about the function or procedure.

10.2 Procedure Blocks

Functions and procedures have similar blocks, and there is no source-language type distinction between them.

Blocks for Procedures. Blocks for procedures are constructed by the linker, using information provided by the translator. Such blocks are read in as part of the icode file when an Icon program is executed. The block for a procedure contains the usual title and size words, followed by six words that characterize the procedure:

(1)The icode location of the first virtual machine instruction for the procedure.

(2) The number of arguments expected by the procedure.

(3) The number of local identifiers in the procedure.

(4) The number of static identifiers in the procedure.

(5) The index in the static identifier array of the first static identifier in the procedure.

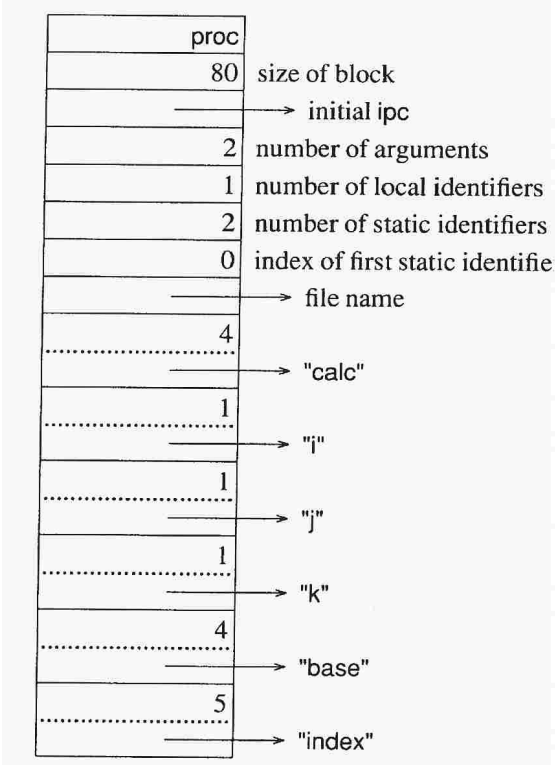
(6) A C string for the name of the file in which the procedure declaration occurred.

The remainder of the procedure block contains qualifiers: one for the string name of the procedure, then others for the string names of the arguments, local identifiers, and static identifiers, in that order.

For example, the procedure declaration

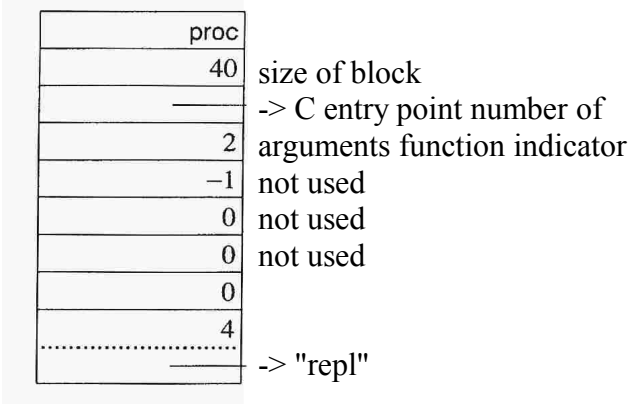
```
procedure calc(i,j)
local k
static base, index
        .
        .
        .
end
```

has the following procedure block:



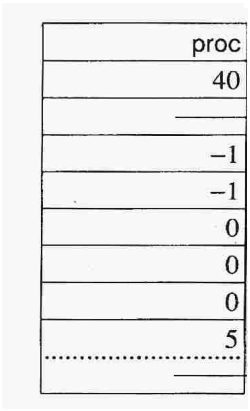
The 0 value for the index in the static identifier array indicates that base is the first static identifier in the program. The indices of static identifiers are zero-based and increase throughout a program as static declarations occur.

Blocks for Functions. Blocks for functions are created by the macro FncDcl that occurs at the beginning of every C function that implements an Icon function. Such blocks for functions are similar to those for procedures but are distinguished by the value -1 in the word that otherwise contains the number of local identifiers. The entry point is the entry point of the C routine for the function. The procedure block for `repl` is typical:



Note that there are no argument names.

Some functions, such as `write()`, allow an arbitrary number of arguments. This is indicated by the value -1 in place of the number of arguments:



10.3 Invocation

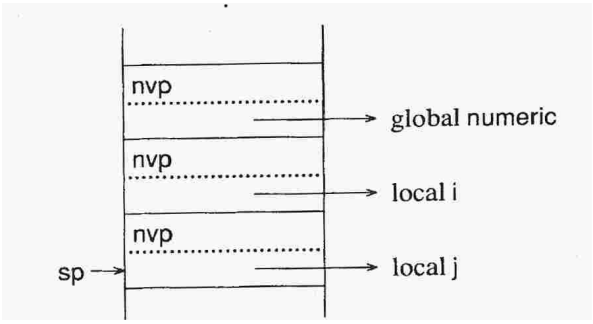
10.3.1 Argument Processing

Argument processing begins by dereferencing the arguments in place on the stack. If a fixed number of arguments is specified in the procedure block, this number is compared with the argument of `invoke`, which is the number of arguments on the stack.

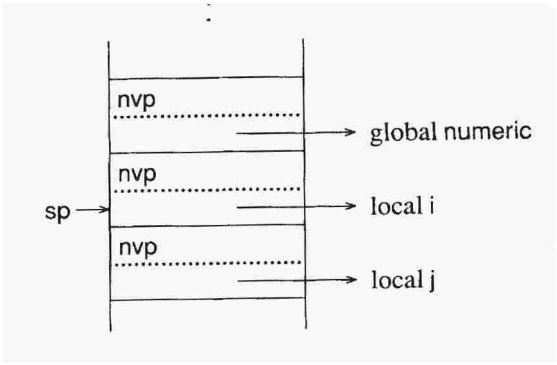
If there are too many arguments, `sp` is set to point to the last one expected. For example, the expression

```
numeric(i, j)
```

results in



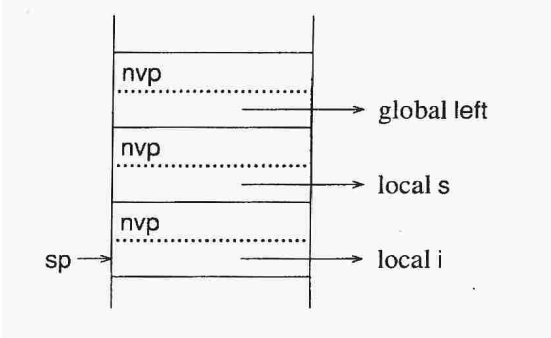
Since `numeric()` expects only one argument, `sp` is reset, effectively popping the second argument:



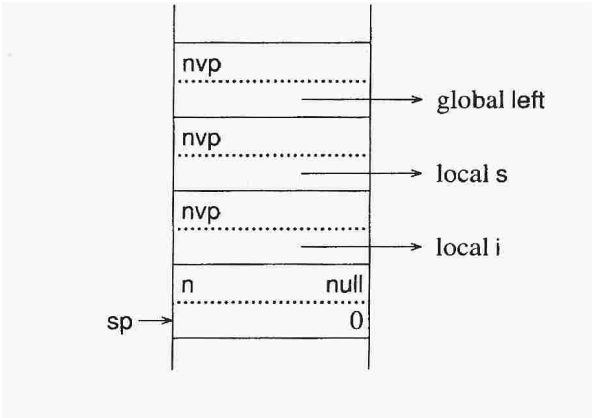
On the other hand, if there are not enough arguments, null-valued descriptors pushed to supply the missing arguments. For example, the expression

```
left(s, i)
```

results in



and a null value is pushed to provide the missing third argument:



10.3.2 Function Invocation

Function invocation involves calling a C function in a fashion that is very similar to evaluating an operator. In the case of an Icon function, the entry point of the corresponding C function is obtained from the procedure block rather than by indexing an array of function pointers corresponding to operator codes.

For an Icon function that has a fixed number of arguments, the C function is called with a single argument that is a pointer to the location of Arg0 on the interpreter stack. Note that Arg0 is the descriptor that points to the procedure block. For an Icon function that may be called with an arbitrary number of arguments, the C function is called with two arguments: the number of arguments on the stack and a pointer to Arg0.

Like an operator, a function may fail, return a result, or suspend. The coding protocol is the same as for operators. The function find is an example:

```
function{*} find(s1,s2,i,j)
   str_anal( s2, i, j )
   if !cnv:string(s1) then
      runerr(103,s1)

   body {
      register char *str1, *str2;
      C_integer s1_len, l, term;
```

```
      /*
       * Loop through s2[i:j] trying to find s1 at each point,
       * stopping when the remaining portion s2[i:j] is too short
       * to contain s1. Optimize me!
       */
      s1_len = StrLen(s1);
      term = cnv_j - s1_len;
      while (cnv_i <= term) {
         str1 = StrLoc(s1);
         str2 = StrLoc(s2) + cnv_i - 1;
         l    = s1_len;

         /*
          * Compare strings on a byte-wise basis; if the end is
          * reached before inequality is found, suspend with the
          * position of the string.
          */
         do {
            if (l-- <= 0) {
               suspend C_integer cnv_i;
               break;
               }
            } while (*str1++ == *str2++);
         cnv_i++;
         }
      fail;
      }
end
```

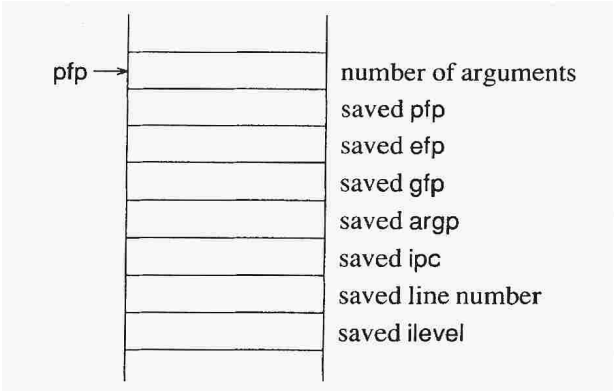
`str_anal()` is an RTL multi-line macro for performing the standard conversions and defaulting for string analysis functions. It takes as arguments the parameters for subject, beginning position, and ending position. It produces declarations for these 3 names prepended with `cnv_`. These variables will contain the converted versions of the arguments.

```
#begdef str_anal(s, i, j)
   declare {
      C_integer cnv_ ## i;
      C_integer cnv_ ## j;
      }

   abstract {
      return integer
      }

   if is:null(s) then {
      inline {
         s = k_subject;
         }
      if is:null(i) then inline {
         cnv_ ## i = k_pos;
         }
      }
   else {
      if !cnv:string(s) then
         runerr(103,s)
      if is:null(i) then inline {
```

```
         cnv_ ## i = 1;
         }
      }

   if !is:null(i) then
      if cnv:C_integer(i,cnv_ ## i) then inline {
         if ((cnv_ ## i = cvpos(cnv_ ## i, StrLen(s))) ==
              CvtFail)
            fail;
         }
      else
         runerr(101,i)

    if is:null(j) then inline {
       cnv_ ## j = StrLen(s) + 1;
       }
    else if cnv:C_integer(j,cnv_ ## j) then inline {
       if ((cnv_ ## j = cvpos(cnv_ ## j, StrLen(s))) == CvtFail)
          fail;
       if (cnv_ ## i > cnv_ ## j) {
          register C_integer tmp;
          tmp = cnv_ ## i;
          cnv_ ## i = cnv_ ## j;
          cnv_ ## j = tmp;
          }
       }
    else
       runerr(101,j)
#enddef
```

10.3.3 Procedure Invocation

In the case of procedure invocation, a *procedure frame* is pushed onto the interpreter stack to preserve information that may be changed during the execution of the procedure and that must be restored when the procedure returns. As for other types of frames, a procedure frame begins with a marker. A procedure frame marker consists of eight words:

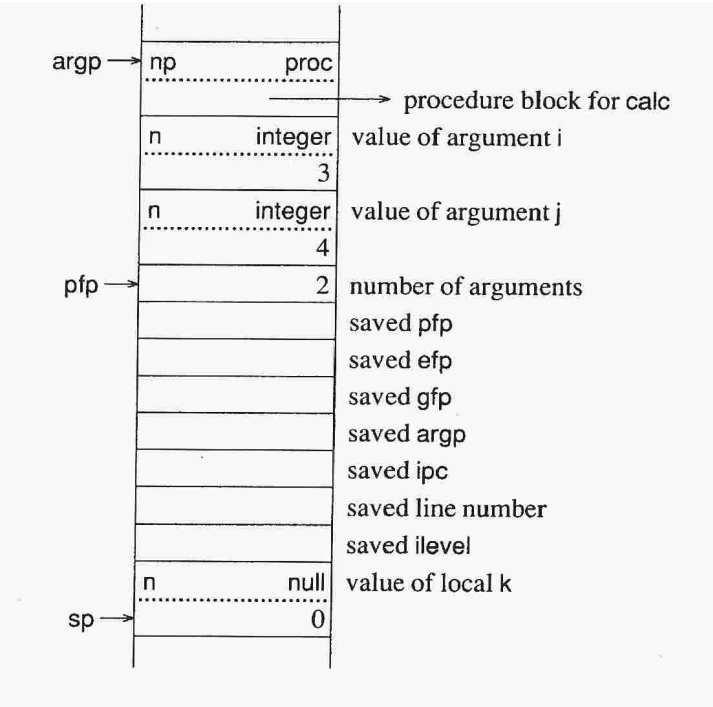


The current procedure frame is pointed to by `pfp`, and `argp` points to the place on the interpreter stack where the arguments begin, analogous to Arg0 for functions. The number of arguments, which can be computed from `pfp` and `argp`, is provided to make computations related to arguments more convenient.

After the procedure marker is constructed, a null-valued descriptor is pushed for each local identifier. For example, the call

```
calc(3,4)
```

for the procedure declaration given in Sec. 10.2 produces



Once the null values for the local identifiers are pushed, `ipc` is set to the entry point given in the procedure block and `efp` and `gfp` are set to zero. Execution then continues in the interpreter with the new `ipc`.

The three forms of return from a procedure are the same as those from a function and correspond to the source-language expressions

```
return e
fail
suspend e
```

The corresponding virtual machine instructions are `pret`, `pfail`, and `psusp`. For example, the virtual machine code for

```
return &null
```

is

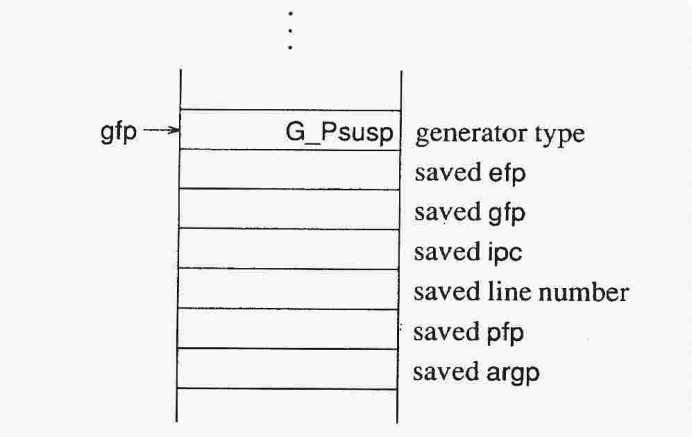
```
pnull
pret
```

In the case of `pret`, the result currently on the top of the interpreter stack is copied on top of the descriptor pointed to by `argp`. If this result is a variable that is on the stack (and hence local to the current procedure call), it is dereferenced in place. The C stack is unwound, since there may be suspended generators at the time of the return. The values saved in the procedure frame marker are restored, and execution continues in the interpreter with the restored `ipc`.

In the case of failure, the C stack is unwound as it is for `pret`, values arc restored from the procedure frame marker, and control is transferred to `efail`.

Procedure suspension is similar to other forms of suspension. The descriptor on the top of the interpreter stack is dereferenced, if necessary, and saved. A generator frame marker is

constructed on the interpreter stack to preserve values that may be needed if the procedure call is resumed. For procedure suspension, a generator frame marker contains two words in addition to those needed for other kinds of generator frame markers and has the form



After the generator frame marker is pushed, the portion of the stack between the last generator or expression frame marker before the call to this procedure and the word prior to `argp` is copied to the top of the stack. Finally, the saved descriptor, which is the result produced by the procedure, is pushed on the top of the stack. Execution then continues in the interpreter with the restored `ipc`.

10.4 Co-Expressions

Co-expressions add another dimension to expression evaluation in Icon. The important thing to understand about co-expressions is that Icon evaluation is always in *some* co-expression. Although it is not evident, the execution of an Icon program begins in a co-expression, namely the value of `&main`.

A co-expression requires both an interpreter stack and a C stack. In the co-expression for `&main`, the interpreter stack is statically allocated and the C stack is the one normally used for C execution–the "system stack" on some computers. The creation of a new co-expression produces a new interpreter stack and a new C stack, as well as space that is needed to save state information. When a co-expression is activated, the context for evaluation is changed to the stacks for the activated co-expression. When the activation of a co-expression produces a result, it in turn activates the co-expression that activated it, leaving the stacks from which the return occurred in a state of suspension. Thus, co-expression activation constitutes a simple context switch. In every co-expression, expression evaluation is in some state, possibly actively executing, possibly suspended, or possibly complete and unreachable.

The virtual machine instructions for

```
   create expr0
```

are

```
        goto L3
L1:
        pop
        mark L2
        code for expr0
        coret
        efail
```

```
L2:
      cofail
      goto L2
L3:
      create    L1
```

Control goes immediately to `L3`, where the instruction create constructs a co-expression block and returns a descriptor that points to it. This block contains space for i-state variables, space for the state of the C stack, an interpreter stack, and a C stack.

The code between `L1` and `L3` is not executed until the co-expression is activated. The pop instruction following `L1` discards the result transmitted to a co-expression on its first activation, since there is no expression waiting to receive the result of an initial activation. Next, an expression frame marker is created, and the code for *expr*$_0$ is executed. If *expr*$_0$ produces a result, `coret` is executed to return the result to the activating expression. If the co-expression is activated again, its execution continues with efail, which causes any suspended generators in the code for *expr*$_0$ to be resumed. If *expr*$_0$ fails, the expression frame is removed and `cofail` is executed. The `cofail` instruction is very similar to the `coret` instruction, except that it signals failure rather than producing a result. Note that if a co-expression that returns by means of `cofail` is activated again, the `cofail` instruction is executed in a loop.

A co-expression is activated by the expression

```
expr1 @ expr2
```

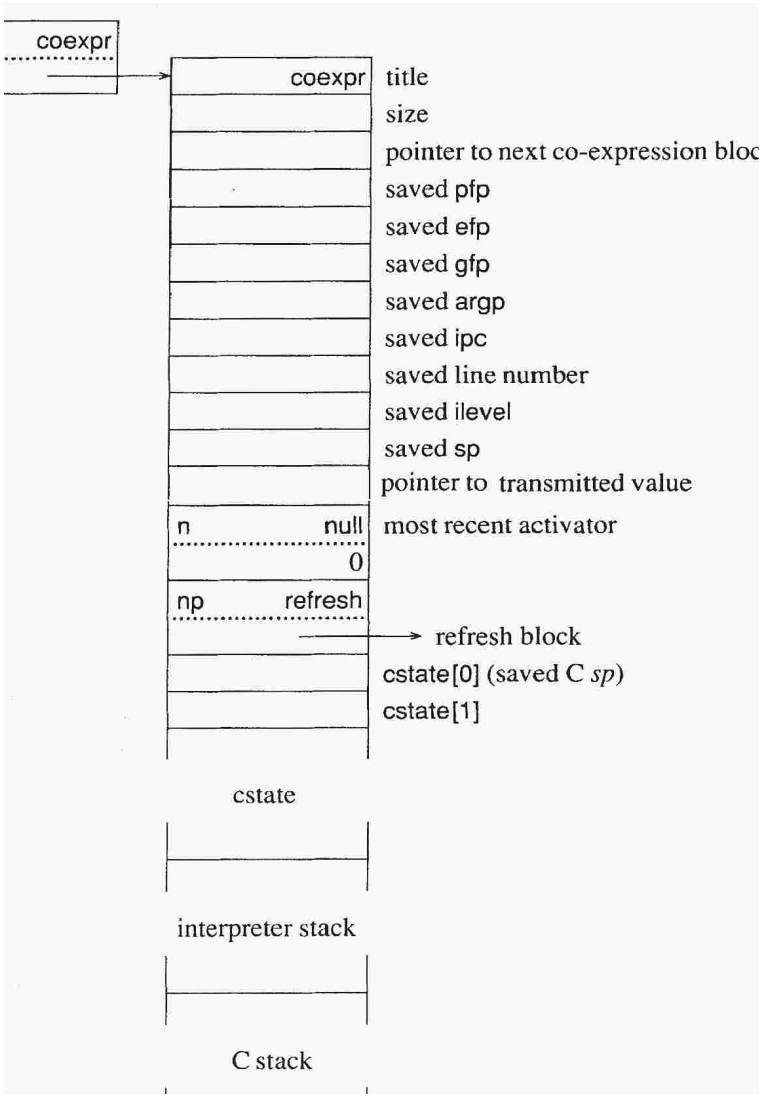
for which the virtual machine code is

```
code for expr1
code for expr2
coact
```

The more common form of activation, @*expr*$_0$, is just an abbreviation for `&null @` *expr*$_0$; a result is always transmitted, even if it is the null value.

The virtual machine code for *expr*$_1$ produces the descriptor for the result that is to be transmitted to the co-expression being activated. The `coact` instruction dereferences the result produced by *expr*$_2$, if necessary, and checks to make sure it is a co-expression. After setting up state information, `coact` transfers control to the new co-expression with `ipc` set to `L1`. Execution continues there. If `coret` is reached, control is restored to the activating co-expression. The instructions `coact` and `coret` are very similar. Each saves the current co-expression state, sets up the new co-expression state, and transfers control.

Co-Expression Blocks. There is quite a bit of information associated with a co-expression, and space is provided for it in a co-expression block:



The interpreter stack and C stack shown in this diagram are not to scale compared with the rest of the block. Both are comparatively large; the actual sizes depend on the address space of the target computer.

The first word of the block is the usual title. The next word contains the number of results the co-expression has produced-its "size." Then there is a pointer to the next co-expression block on a list that is maintained for garbage collection purposes. See Sec. 11.3.4. Following this pointer there are i-state variables: `pfp`, `efp`, `gfp`, `argp`, `ipc`, `sp`, the current program line number, and `ilevel`.

Then there is a descriptor for the transmitted result, followed by two more descriptors: one for the co-expression that activates this one and one for a *refresh* block that is needed if a copy of this co-expression block is needed. C state information is contained in an array of words, `cstate`, for registers and possibly other state information. The array `cstate` typically contains fifteen words for such information. The C *sp* is stored in `cstate[0]`. The use of the rest of `cstate` is machine-dependent.

Finally, there is an interpreter stack and a C stack. On a computer with a downward-growing C stack, such as the VAX, the base of the C stack is at the end of the co-

expression block and the interpreter and C stacks grow toward each other. On a computer with an upward-growing C stack, the C stack base follows the end of the interpreter stack.

Stack Initialization. When a co-expression is first activated, its interpreter stack must be in an appropriate state. This initialization is done when the co-expression block is created. A procedure frame, which is a copy of the procedure frame for the procedure in which the create instruction is executed, is placed on the new stack. It consists of the words from `argp` through the procedure frame marker and the descriptors for the local identifiers. The `efp` and `gfp` in the co-expression block are set to zero and the `ipc` is set to the value given in the argument to the `create` instruction (`L1`).

No C state is set up on the new C stack; this is handled when the co-expression is activated the first time. The initial null value for the activator indicates the absence of a valid C state.

Co-Expression Activation. As mentioned previously, `coact` and `coret` perform many similar functions–both save current state information, establish new state information, and activate another co-expression. The current i-state variables are saved in the current co-expression block, and new ones are established from the co-expression block for the co-expression being activated. Similar actions are taken for the C state. Since the C state is machine-dependent, the "context switch" for the C state is performed by a routine, called `coswitch()`, that contains assembly-language code.

The C state typically consists of registers that are used to address the C stack and registers that must be preserved across the call of a C function. On the VAX, for example, the C stack registers are *sp, ap, and fp.* Only the registers `r6` through `r11` must be saved for some C compilers, while other C compilers require that `r3` through `r11` be saved. Once the necessary registers are saved in the `cstate` array of the current co-expression, new values of these registers are established. If the co-expression being activated has been activated before, the C state is set up from its `cstate` array, and `coswitch()` returns to `interp()`. At this point, execution continues in the newly activated co-expression. Control is transferred to the beginning of the interpreter loop, and the next instruction (from the `ipc` for the co-expression) is fetched.

However, when a co-expression is activated for the first time, there are no register values to restore, since no C function has yet been called for the new co-expression. This is indicated, as mentioned previously, by a null activator, which is communicated to `coswitch()` by an integer argument. In this case, `coswitch()` sets up registers for the call of a C function and calls `interp()` to start the execution of the new co-expression. Such a call to `interp()` on the first activation of a co-expression corresponds to the call to `interp()` that starts program execution in the co-expression `&main` for the main procedure. There can never be a return from the call to `interp()` made in `coswitch()`, since program execution can only terminate normally by a return from the main procedure, in `&main`.

The function `coswitch()` is fastest if it is machine-dependent. The version for the x86 with the GCC compiler is an example:

```
coswitch:
      pushl %ebp
      movl %esp,%ebp
      movl 8(%ebp),%eax
      movl %esp,0(%eax)
      movl %ebp,4(%eax)
```

```
      movl 12(%ebp),%eax
      cmpl $0,16(%ebp)
      movl 0(%eax),%esp
      je .L2

      movl 4(%eax),%ebp
      jmp .L1

.L2:
      movl $0,%ebp
      pushl $0
      pushl $0
      call new_context
      pushl $.LC0
      call syserr
      addl $12,%esp

.L1:
      leave
      ret
```

If no assembler co-expression switch is available, modern platforms with POSIX threads can use Icon 9.5.1's supported form of co-expressions, which is more than 100x slower and approximately twice as many lines of C. The public interface in either case looks like:

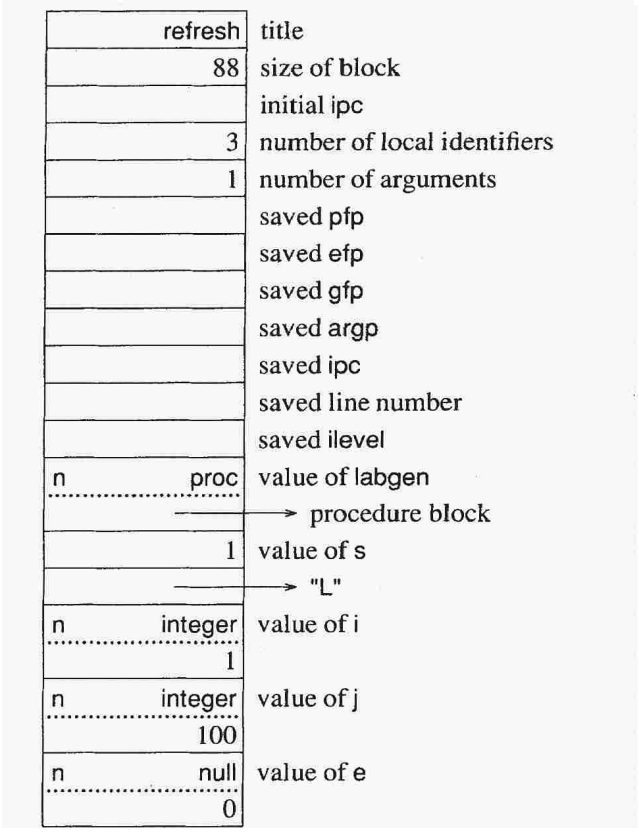
```
int coswitch(void *old_cs, void *new_cs, int first);
```

The variables `old_cs` and `new_cs` are pointers to the `cstate` arrays for the activating and activated co-expressions, respectively. The value of first is 0 if the co-expression is being activated for the first time. Note that in order to write `coswitch()` it is necessary to know how the first two arguments are accessed in assembly language. For the previous example, `old_cs` and `new_cs` are eight and twelve bytes from the *ebp* register, respectively.

Refreshing a Co-Expression. The operation ^*expr*$_0$ creates a copy of the co-expression produced by *expr*$_0$ with its state initialized to what it was when it was originally created. The refresh block for *expr*$_0$ contains the information necessary to reproduce the initial state. The refresh block contains the original `ipc` for the co-expression, the number of local identifiers for the procedure in which *expr*$_0$ was created, a copy of the procedure frame marker at the time of creation, and the values of the arguments and local identifiers at the time of creation. Consider, for example,

```
procedure labgen(s)
   local i, j, e
   i := 1
   j := 100
   e := create (s || (i to j) || ":")
end
```

For the call `labgen("L")`, the refresh block for `e` is



RETROSPECTIVE: Invocation expressions are more complicated to implement than operators, since the meaning of an invocation expression is not known until it is evaluated. Since functions and procedures are source-language values, the information associated with them is stored in blocks in the same manner as for other types.

The C code that implements Icon functions is written in the same fashion as the code for operators. Procedures have source-language analogs of the failure and suspension mechanisms used for implementing functions and operators. Procedure frames identify the portions of the interpreter stack associated with the procedures currently invoked.

A co-expression allows an expression to be evaluated outside its lexical site in the program by providing separate stacks for its evaluation. The possibility of multiple stacks in various states of evaluation introduces technical problems in the implementation, including a machine-dependent context switch.

EXERCISES

10.1 What happens if a call of a procedure or function contains an extra argument expression, but the evaluation of that expression fails?

10.2 Sometimes it is useful to be able to specify a function or procedure by means of its string name. Icon supports "string invocation," which allows the value of $expr_0$ in

$expr_0(expr_1, expr_2, \ldots, expr_n)$.

to be a string. Thus,

```
"write"(s)
```

produces the same result as

```
write(s)
```

Of course, such a string name is usually computed, as in

```
(read())(s)
```

Describe what is involved in implementing this aspect of invocation. Operators also may be invoked by their string names, as in

```
"+"(i,j)
```

What is needed in the implementation to support this facility? Can a control structure be invoked by a string name?

10.3 If the result returned by a procedure is a variable, it may need to be dereferenced. This is done in the code for `pret` and `psusp`. For example, if the result being returned is a local identifier, it must be replaced by its value What other kinds of variables must be dereferenced? Is there any difference in the dereferencing done by `pret` and `psusp`?

10.4 How is the structure of a co-expression block different on a computer with an upward-growing C stack compared to one with a downward-growing C stack? What is the difference between the two cases in terms of potential storage fragmentation?

Chapter 11: Storage Management

PERSPECTIVE: The implementation of storage management must accommodate a wide range of allocation requirements. At the same time, the implementation must provide generality and some compromise between "normal" programs and those that have unusual requirements. Although it is clearly sensible to satisfy the needs of most programs in an efficient manner, there is no way to define what is typical or to predict how programming style and applications may change. Indeed, the performance of the implementation may affect both programming style and applications.

Strings and blocks can be created during program execution at times that cannot be predicted, in general, from examination of the text of a program. The sizes of strings and of some types of blocks may vary and may be arbitrarily large, although practical considerations dictate some limits. There may be an arbitrary number of strings and blocks. The "lifetimes" during which they may be used are arbitrary and are unrelated, in general, to procedure calls and returns.

Different programs vary considerably in the number, type, and sizes of data objects that are created at run time. Some programs read in strings, transform them, and write them out without ever creating other types of objects. Other programs create and manipulate many lists, sets, and tables but use few strings. Relatively few programs use co-expressions, but there are applications in which large numbers of co-expressions are created.

Since a program can construct an arbitrary number of data objects of arbitrary sizes and lifetimes, some mechanism is needed to allow the reuse of space for "dead" objects that are no longer accessible to the program. Thus, in addition to a mechanism for allocating storage for objects at run time, there must be a storage-reclamation mechanism, which usually is called *garbage collection.* The methods used for allocation and garbage collection are interdependent. Simple and fast allocation methods usually require complex and time-consuming garbage-collection techniques, while more efficient garbage-collection techniques generally lead to more complex allocation techniques.

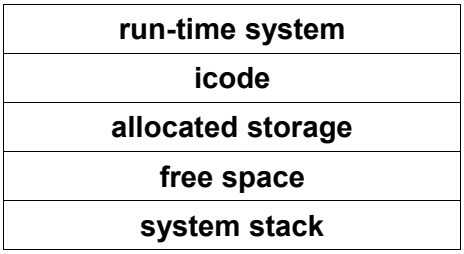
Storage management has influences that are far-reaching. In some programs, it may account for a major portion of execution time. The design of data structures, the layout of blocks, and the representation of strings are all influenced by storage-management considerations. For example, both a descriptor that points to a block and the first word of the block contain the same type code. This information is redundant as far as program execution is concerned, since blocks are accessed only via descriptors that point to them. The redundant type information is motivated by storage-management considerations. During garbage collection, it is necessary to access blocks directly, rather than through pointers from descriptors, and it must be possible to determine the type of a block from the block itself. Similarly, the size of a block is of no interest in performing language operations, but the size is needed during garbage collection. Blocks, therefore, carry some "overhead" for storage management. This overhead consists primarily of extra space, reflecting the fact that it takes more space to manage storage dynamically than would be needed if space were allocated statically. Balancing space overhead against efficiency in allocating and collecting objects is a complex task.

Such problems have plagued and intrigued implementors since the early days of LISP. Many ways have been devised to handle dynamic storage management, and some

techniques have been highly refined to meet specific requirements (Cohen 1981). In the case of Icon, there is *more* emphasis on storage management for strings than there is in a language, such as LISP, where lists predominate. Icon's storage-management system reflects previous experience with storage-management systems used in XPL (McKeeman, Horning, and Wortman 1970), SNOBOL4 (Hanson 1977), and the earlier Ratfor implementation of Icon (Hanson 1980). The result is, of course, somewhat idiosyncratic, but it provides an interesting case study of a real storage-management system.

11.1 Memory Layout

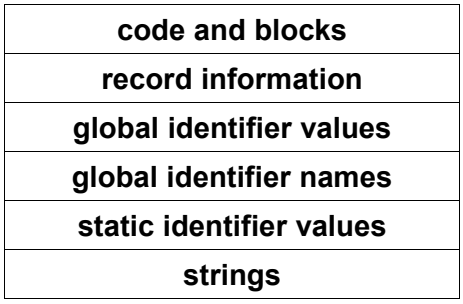
During the execution of an Icon program, memory is divided into several regions. The sizes and locations of these regions are somewhat dependent on computer architecture and the operating system used, but typically they have the following form:



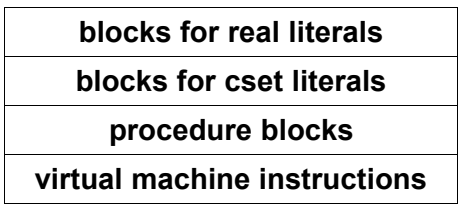
This diagram is not drawn to scale; some regions are much larger than others.

The Run-Time System. The run-time system contains the executable code for the interpreter, built-in operators and functions, support routines, and so forth. It also contains static storage for some Icon strings and blocks that appear in C functions. For example, the blocks for keyword trapped variables are statically allocated in the data area of the run-time system. Such blocks never move, but their contents may change. The size of the run-time system is machine-dependent.

The Icode Region. One of the first things done by the run-time system is to read in the icode file for the program that is to be executed. The data in the icode region, which is produced by the linker, is divided into a number of sections:



The first section contains virtual machine code, blocks for cset and real literals, and procedure blocks, on a per-procedure basis. Thus, the section of the icode region that contains code and blocks consists of segments of the following form for each procedure:



Record information for the entire program is in the second section of the icode region. Next, there is an array of descriptors for the values of the global identifiers in the program, followed by an array that contains qualifiers for the names of the global identifiers. These two arrays are parallel. The *ith* descriptor in the first array contains the value of the ith global identifier, and the ith descriptor in the second array contains a qualifier for its name.

Following the two arrays related to global identifiers is an array for the values of static identifiers. As mentioned in Sec. 2.1.10, static identifiers have global scope with respect to procedure invocation, but a static identifier is accessible only to the procedure in which it is declared.

Unlike cset and real blocks, which are compiled on a per-procedure basis, all strings in a program are pooled and are in a single section of the icode region that follows the array of static identifiers. A literal string that occurs more than once in a program occurs only once in the string section of the icode region.

Data in the icode region is never moved, although some components of it may change at run time. The size of the icode region depends primarily on the size of the corresponding source program. As a rule of thumb, an icode region is approximately twice as large as the corresponding Icon source-language file. An icode file for a short program might be 1,000 bytes, while one for a large program (by Icon standards) might be 20,000 bytes.

Allocated Storage. The space for data objects that are constructed at run time is provided in allocated storage regions. This portion of memory is divided into three parts:

The static region contains co-expression blocks. The remainder of the allocated storage region is divided into strings and blocks as shown. The string region contains only characters. The block region, on the other hand, contains pointers. This leads to a number of differences in allocation and garbage-collection techniques in different regions.

Data in the static region is never moved, but strings and blocks may be. Both the string and block regions may be moved if it is necessary to increase the size of the static region. Similarly, the block region may be moved in order to enlarge the string region.

The initial sizes of the allocated storage regions vary considerably from computer to computer, depending on the size of the user address space. On a computer with a small address space, such as the PDP-ll, Icon was implemented with region sizes as small as:

static region:	4,096 bytes	(2,048 words)
string region:	10,240 bytes	(5,120 words)
block region:	10,240 bytes	(5,120 words)
total:	24,576 bytes	(12,288 words)

On modern machines, initial sizes of 500,000 bytes or 2,000,000 bytes per region are common. Unicon allocates 1% of physical memory for each of the string and block region. The user may establish different initial sizes prior to program execution by using environment variables STRSIZE and BLKSIZE. As indicated previously, the sizes of

these regions are increased at run time if necessary, but there is no provision for decreasing the size of a region once it has grown to a given size.

Free Space and the System Stack. On computers with system stacks that grow downward, such as the VAX, the system stack grows toward the allocated storage region. Between the two regions is a region of free space into which the allocated storage region may grow upward. Excessive recursion in C may cause collision of the system stack and the allocated storage region. This is an unrecoverable condition, and the result is termination of program execution. Similarly, more space may be needed for allocated storage than is available. This also results in termination of program execution. In practice, the actual situation depends to a large extent on the size of the user address space, which is the total amount of memory that is available for all the regions shown previously. On a computer with a small user address space, such as the PDP-11, the amount of memory available for allocated storage is a limiting factor for some programs. Furthermore, collision of the allocated storage region and the system stack is a serious problem. On a computer that supports a large virtual memory, the size of the system stack is deliberately limited, since the the total amount of memory available is so large that runaway recursion would consume enormous resources before a collision occurred between the system stack and the allocated storage region.

11.2 Allocation

Storage allocation in Icon is designed to be fast and simple. Garbage collection is somewhat more complicated as a result. Part of the rationale for this approach is that most Icon programs do a considerable amount of allocation, but many programs never do a garbage collection. Hence, programs that do not garbage collect are not penalized by a strategy that makes garbage collection more efficient at the expense of making allocation less efficient. The other rationale for this approach is that the storage requirements of Icon do not readily lend themselves to more complex allocation strategies.

11.2.1 The Static Region

Data allocated in the static region is never moved, although it may be freed for reuse. Co-expression blocks are allocated in the static region, since their C stacks contain internal pointers that depend on both the computer and the C compiler and hence are difficult to relocate to another place in memory. Furthermore, since co-expression blocks are all the same size, it is economical and simple to free and reuse their space.

The C library routines `malloc()` and `free()` are used to allocate and free co-expression blocks in the static region. These routines maintain a list of blocks of free space. The routine `malloc()` finds a block of the requested size, dividing a larger block if necessary, and revises the free list accordingly. The routine `free()` returns the freed space to the free list, coalescing it with adjacent free blocks if possible. See Kernighan and Ritchie 1978 for a discussion of free-list allocation.

Icon contains its own version of these routines to assure that space is allocated in its own static region and to allow its overall memory region to be expanded without conflict with other users of `malloc()`. Thus, if a user extension to Icon or the operating system calls `malloc()`, Icon's own routine handles the request. This means that the static region may contain space allocated for data other than co-expression blocks, although this normally is not the case.

11.2.2 Blocks

For other kinds of blocks, Icon takes advantage of the fact that its own data can be relocated if necessary and uses a very simple allocation technique. The allocated region for blocks is divided into two parts:

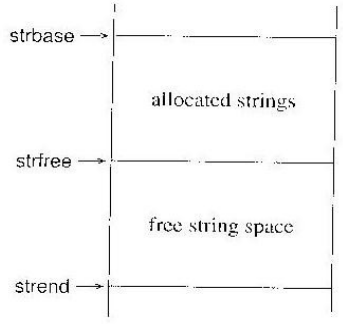
When there is a request for a block of *n* bytes, the free pointer, `blkfree`, is incremented by *n* and the previous value of the free pointer is returned as the location of the newly allocated block. This process is fast and free of the complexities of the free-list approach.

Note that this technique really amounts to a free list with only one block. The problem of reclaiming fragmented space on the free list is exchanged for the process of reclaiming unused blocks and rearranging the block region so that all the free space is in one contiguous portion of the block region. This is done during garbage collection.

11.2.3 Strings

There is even less justification for a free-list approach for allocating strings. A newly created string may be one character long or it may be thousands of characters long. Furthermore, while there is space in blocks that can be used to link together free storage, there is no such space in strings, and a free list would involve additional storage.

Instead, the string region is allocated in the same way that the block region is allocated:



As with the block region, a garbage collection is performed if there is not enough space in the string region to satisfy an allocation request.

11.3 Garbage Collection

Allocation is simple, but garbage collection is not. The primary purpose of garbage collection is to reclaim the space occupied by "dead" objects that are not needed for

subsequent program execution, so that this space can be reallocated. This means different things in different regions. In the static region, it means freeing dead co-expression blocks. In the string and block regions, it involves moving the space for dead objects from the allocated portion of the region to the free portion. This is considerably more complicated than adding a pointer to a free list. Since all free space must be in a single block in these regions, "live" objects must be moved to fill in the holes left by dead ones. This is done by compacting the allocated portion of these regions, relocating live objects toward the beginning of these regions and squeezing out dead objects. In turn, pointers to live objects have to be adjusted to correspond to their new locations. There are two phases in garbage collection:

- Location of live objects and all the pointers to them.
- Compaction of live objects and adjustment of the pointers to them.

"Garbage collection" is somewhat of a misnomer, since the process is oriented toward saving "non-garbage" objects; garbage disappears as a byproduct of this operation.

11.3.1 The Basis

The challenging problem for garbage collection is the location of objects that have to be saved, as well as all pointers to them. An object is dead, by definition, if it cannot be accessed by any future source-language computation. Conversely, by definition, an object is live if it can be accessed. Consequently, the important issue is the possibility of computational access. For example, it is always possible to access the value of `&subject`, and this value must be preserved by garbage collection. On the other hand, in

```
a := [1,2,3]
a := list(10)
```

after the execution of the second assignment, the first list assigned to a is inaccessible and can be collected.

It is essential to save any object that may be accessed, but there is no way, in general, to know if a specific object *will* be accessed. For example, a computational path may depend on factors that are external to the program, such as the value of data that is read from a file. It does comparatively little harm to save an object that might be accessed but, in fact, never is. Some storage is wasted, but it is likely to be reclaimed during a subsequent collection. It is a serious error, on the other hand, to discard an object that subsequently *is* accessed. In the first place, the former value of such an object usually is overwritten and hence is "garbage" if it is subsequently accessed. Furthermore, accessing such an object may overwrite another accessible object that now occupies the space for the former one. The effects may range from incorrect computational results to addressing violations. The sources of such errors also are hard to locate, since they may not be manifested until considerably later during execution and in a context that is unrelated to the real cause of the problem. Consequently, it is important to be conservative and to err, if at all, on the side of saving objects whose subsequent accessibility is questionable. Note that it is not only necessary to locate all accessible objects, but it is also necessary to locate all pointers to objects that may be relocated.

The location phase starts with a *basis* that consists of descriptors that point to objects that may be accessible and from which other objects may be accessed. For example, `&subject` is in the basis. The precise content of the basis is partly a consequence of

properties of the Icon language and partly a consequence of the way the run-time system is implemented. The basis consists of the following descriptors:

- `&main` (co-expression block for the initial call of main)
- current co-expression block
- values of global identifiers
- values of static identifiers
- `&subject`
- saved values of map arguments
- tended descriptors

The tended descriptors provide temporary storage for a run-time support routine in which a garbage collection may occur. See Sec. 12.2.2.

Not all objects that have to be saved are pointed to directly by the basis. The value of a local identifier on the interpreter stack may point to a list-header block that in turn points to a list-element block that contains elements pointing to strings and other blocks. Pointer chains also can be circular.

11.3.2 The Location Phase

For historical reasons, the location phase is sometimes called *marking*. This term refers to the common practice of setting an identifying bit in objects that have been located. Not all such processes actually change the objects that are located. The way that this is done in Icon depends on the region in which an object is located.

During the location phase, every descriptor in the basis is examined. A descriptor is of interest only if it is a qualifier or if its v-word contains a pointer (that is, if its d-word contains a p flag). For a pointer `dp` to a descriptor, the following checks are performed:

```
if (Qual(*dp))
   postqual(dp);
else if (Pointer(*dp))
   markblock(dp);
```

where the macro `Pointer(d)` tests the d-word of d for a p flag.

Strings. The routine `postqual()` first checks that the v-word of the qualifier points to a string in the allocated string region, since strings in other parts of memory are not of interest during garbage collection. If the string is in the allocated string region, a pointer to the qualifier is placed in an array:

```
void postqual(dptr dp)
{
...
if (InRange(strbase,StrLoc(*dp),strfree+1)) {
   *qualfree++ = dp;
   }
}
```

The array `quallist` is empty when garbage collection begins. Its size is checked before a pointer is added to it, and more space is obtained if it is needed although the code for doing that is not shown here. See Sec. 11.3.6.

The pointers that accumulate in `quallist` during the marking phase provide the information necessary to determine the portion of the allocated string region that is in use. In addition, these pointers point to all the qualifiers whose v-word must be adjusted when the strings they point to are moved during the compaction of string region. See Sec. 11.3.3.

Blocks. The location phase for blocks is more complicated than that for strings, since blocks can contain descriptors that point to strings as well as to other blocks. The objects that these descriptors point to must be processed also.

Unlike strings, in which a separate array is used to keep track of qualifier that have been located, no extra space is needed to keep track of descriptors that point to blocks. Instead, descriptors and the titles of the blocks they point to are modified temporarily.

The title of any block located in the allocated block region is changed to point to a *back chain* that contains all the descriptors that point to that block. The descriptors are linked through their v-words.

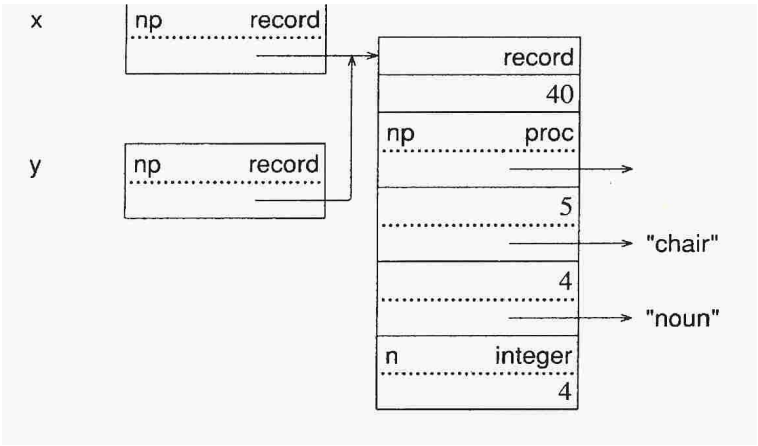
The following example illustrates the process. Suppose there is a record declaration

```
record term(value, code, count)
```

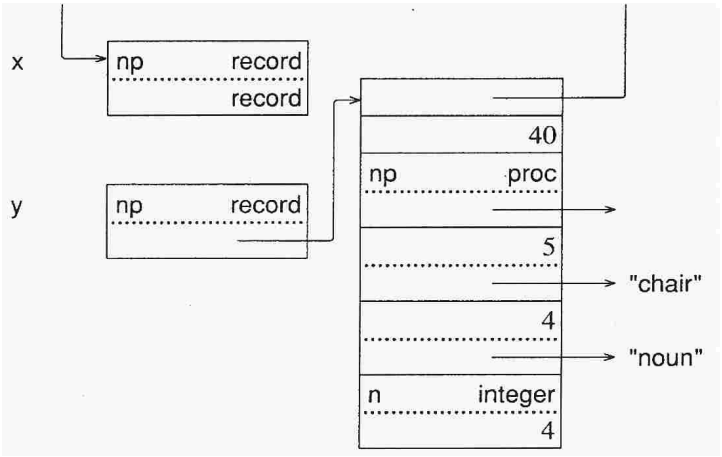
and that the following expressions are evaluated:

```
x := term("chair", "noun",4)
y := x
```

The values of x, y, and the block they point to are related as follows:



Suppose that the descriptor containing the value of x is processed during the location phase before the descriptor containing the value of y. This descriptor is identified as pointing to a block in the allocated block region by virtue of the p flag in its d-word and an address range check on the value of its v-word. The back chain is established by setting the title word of the block to point to the descriptor and setting the v-word of the descriptor to hold the previous contents of the title word. The result is



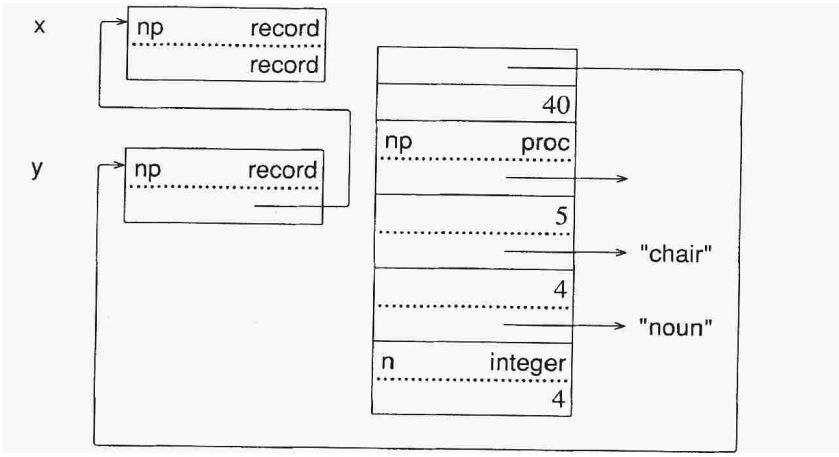
The title word of the block now points to the descriptor that previously pointed to the block. This change is reversible, and prior to the completion of the garbage collection process the previous relationship is restored. A crucial but somewhat subtle aspect of the change is that it is now possible to tell that the block has been marked. The numerical magnitude of the value of its title word is greater than that of any type code, since all descriptors in the run-time system are at memory locations whose addresses are larger than the largest type code.

The descriptors in the record block now are processed in the same way as descriptors in the basis. In order to do this, it is necessary to know where descriptors are located in the block. Since blocks in the allocated block region are organized so that all descriptors follow all non-descriptor data, it is only necessary to know where the first descriptor is and how large the block is. These values are determined using two arrays that have entries for each type code.

The first array, `bsizes`, provides the information that is needed to determine block sizes. There are three kinds of entries. An entry of -1 indicates a type for which there is no block or for which the blocks are not in the allocated block region. Examples are `T_Null` and `T_Coexpr`. An entry of 0 indicates that the size of the block follows the block title. This is the case for records. Any other entry is the actual size of the block in bytes. For example, the entry in `bsizes` for `T_List` is 24 on a 32-bit computer.

The second array, `firstd`, is used to determine the byte offset of the first descriptor in the block. As with `bsizes`, a value of -1 indicates a type for which there are no associated blocks in the allocated block region. A value of 0 indicates that there are no descriptors in the block. Examples are `T_Cset` and `T_Real`. For `T_Record`, the entry is 8 for 32-bit computers, indicating that the first descriptor is at an offset of 8 bytes (2 words) from the beginning of the block. See Sec. 4.2.

For the previous example, after the descriptors in the record block are processed, the location phase continues. When the descriptor that contains the value of y is processed, it is added to the back chain by again exchanging the contents of its v-word with the contents of the title of the block. As a result, the title of the block points to the descriptor for the value of y and its v-word points to the descriptor for the value of x:



Since the title of the block that y points to is marked, the descriptors in it are not processed. This prevents descriptors from being processed twice and also prevents the marking phase from looping in case there are pointer loops among blocks.

If a variable descriptor is encountered when processing descriptors whose d-words contain p flags, the value the variable points to belongs to one of the following categories:

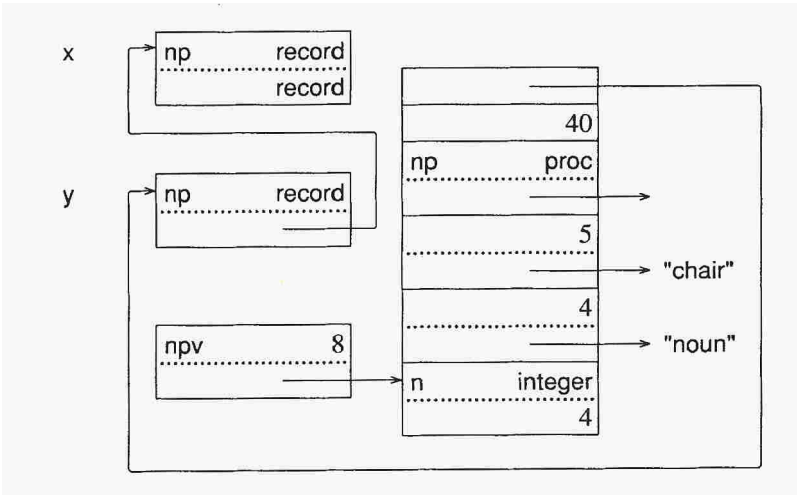
- trapped-variable block
- global or static identifier
- argument or local identifier
- descriptor in a structure

A trapped variable, indicated by a t flag in its v-word, points to a block and is processed like any other descriptor that points to a block. The values of global and static identifiers are in the basis and are processed separately. The values of arguments and local identifiers are on an interpreter stack and are processed when its co-expression block is processed. A variable descriptor that points to a descriptor in a structure points *within* a block, not to the title of a block. This is the only case in which the offset, which is contained in the least-significant portion of the d-word of a non-trapped-variable descriptor, is nonzero. Consequently, this offset is used to distinguish such variables from those in the second and third categories.

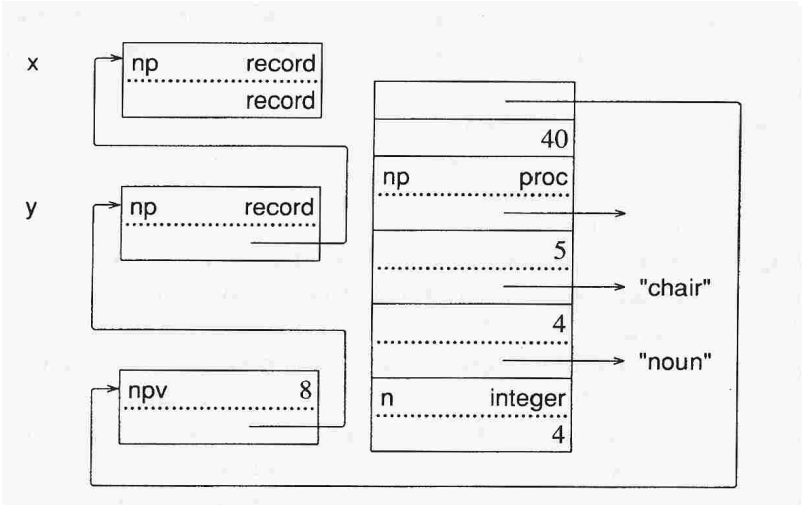
Continuing the previous example, suppose that a garbage collection is triggered by evaluation of the expression

```
x.count := read()
```

At the beginning of garbage collection, there is a variable descriptor for the field reference that points to the record block in addition to the descriptors for the values of x and y. If the values of x and y are processed first as described previously, the situation when the variable descriptor is encountered is



Note that the offset in the d-word of the variable descriptor is in words, not bytes. The offset, converted to bytes, is added to the v-word in the variable descriptor, and this descriptor is linked into the back chain.



When the location phase is complete, the title of each block in the allocated block region that must be saved points to a chain of all the descriptors that originally pointed to it. This provides the necessary information to adjust the v-words of these descriptors to account for the relocation of the block during the compaction phase. See Sec. 11.3.3.

If a descriptor that points to a co-expression block is encountered during the location phase, the title of the co-expression block is marked and the descriptors in the co-expression block are processed in a fashion similar to that for blocks in the allocated block region. Since co-expression blocks are never moved, it is not necessary to keep track of descriptors that point to them. To mark the title, it is sufficient to change it to a value that is larger than any type code.

The activator of the co-expression (if any) is processed like any other co-expression block. Similarly, the refresh block that is pointed to from the co-expression block must be processed like any other block. The rest of the descriptors associated with a co-expression are in its interpreter stack.

Here the situation is more complicated than it is with blocks in the allocated block region, since interpreter stacks contain frame markers in addition to descriptors. Despite this, all the descriptors, and only the descriptors, on an interpreter stack must be processed.

Interpreter stacks are processed by the routine `sweep()`, which starts at `sp` for the stack and works toward the stack base. Descriptors are processed until the next frame marker is encountered, at which point, depending on the type of the frame, the marker is skipped and new frame pointers are set up from it.

The routine for marking blocks is

```
static void markblock(dp)
dptr dp;
   {
   register dptr dp1;
   register char *block, *endblock;
   word type, fdesc;
   int numptr;
   register union block **ptr, **lastptr;

   if (Var(*dp)) {
       if (dp->dword & F_Typecode) {
          switch (Type(*dp)) {
             case T_Kywdint:
             case T_Kywdpos:
             case T_Kywdsubj:
                /*
                 * descriptor points to a keyword, not a block.
                 */
                return;
             }
          }
       else if (Offset(*dp) == 0) {
          /*
           * A simple variable not residing in a block.
           */
          return;
          }
      }

   /*
    * Get the block to which dp points.
    */
   block = (char *)BlkLoc(*dp);

   if (InRange(blkbase,block,blkfree)) {
      type = BlkType(block);
      if ((uword)type <= MaxType) {

         /*
          * The type is valid, which indicates that this block
          *  has not been marked.  Point endblock to the byte
          *  past the end of the block.
          */
         endblock = block + BlkSize(block);
         }

      /*
```

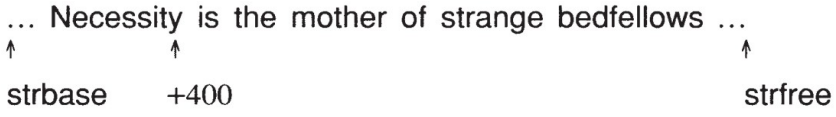
```
      * Add dp to the back chain for the block and point the
      *  block (via the type field) to dp.vword.
      */
     BlkLoc(*dp) = (union block *)type;
     BlkType(block) = (uword)&BlkLoc(*dp);

     if ((uword)type <= MaxType) {
        /*
         * The block was not marked; process pointers and
         *  descriptors within the block.
         */
        if ((fdesc = firstp[type]) > 0) {
           /*
            * The block contains pointers; mark each pointer.
            */
           ptr = (union block **)(block + fdesc);
           numptr = ptrno[type];
           if (numptr > 0)
              lastptr = ptr + numptr;
           else
              lastptr = (union block **)endblock;
           for (; ptr < lastptr; ptr++)
              if (*ptr != NULL)
                 markptr(ptr);
           }
        if ((fdesc = firstd[type]) > 0)
           /*
            * The block contains descriptors; mark each one.
            */
           for (dp1 = (dptr)(block + fdesc);
                (char *)dp1 < endblock; dp1++) {
              if (Qual(*dp1))
                 postqual(dp1);
              else if (Pointer(*dp1))
                 markblock(dp1);
              }
        }
     }

  else if ((unsigned int)BlkType(block) == T_Coexpr) {
     struct b_coexpr *cp;
     struct astkblk *abp;
     int i;
     struct descrip adesc;

     /*
      * dp points to a co-expression block that has not been
      *  marked.  Point the block to dp.  Sweep the interpreter
      *  stack in the block.  Then mark the block for the
      *  activating co-expression and the refresh block.
      */
     BlkType(block) = (uword)dp;
     sweep((struct b_coexpr *)block);

     /*
      * Mark the activators of this co-expression.   The
      *  activators are stored as a list of addresses, but
      *  markblock requires the address of a descriptor.  To
```

```
         *  accommodate markblock, the dummy descriptor adesc is
         *  filled in with each activator address in turn and then
         *  marked.  Since co-expressions and the descriptors that
         *  reference them don't participate in the back-chaining
         *  scheme, it's ok to reuse the descriptor in this manner.
         */
        cp = (struct b_coexpr *)block;
        adesc.dword = D_Coexpr;
        for (abp = cp->es_actstk; abp!=NULL; abp = abp->astk_nxt) {
           for (i = 1; i <= abp->nactivators; i++) {
              BlkLoc(adesc) =
                 (union block *)abp->arec[i-1].activator;
              markblock(&adesc);
              }
           }
        if(BlkLoc(cp->freshblk) != NULL)
           markblock(&((struct b_coexpr *)block)->freshblk);
        }
     else {
        /* … code for blocks found in other regions */
        }
     }
```

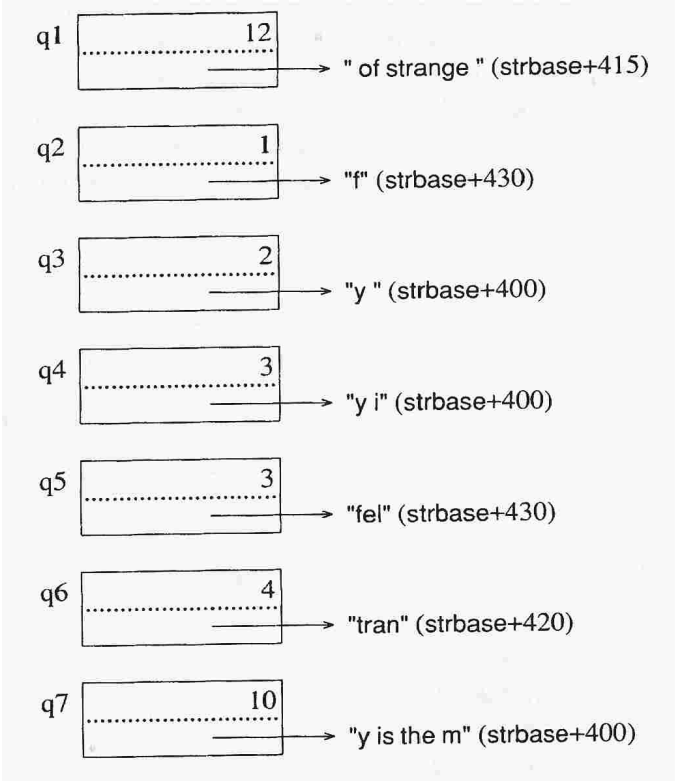
The macro `BlkType(cp)` produces the type code of the block pointed to by `cp`. The macro `BlkSize(cp)` uses the array `bsizes` to determine the size of the block pointed to by `cp`.

11.3.3 Pointer Adjustment and Compaction

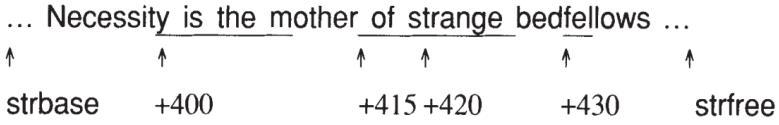
Strings. When the location phase is complete, quallist contains a list pointers to all the qualifiers whose v-words point to the allocated string region. For example, suppose that the allocated string region at the beginning of a garbage collection is



Suppose also that the following qualifiers reference the allocated string region:

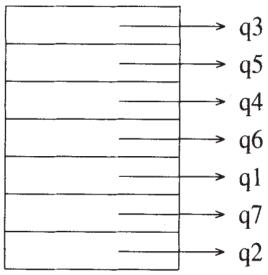


The pointers to the allocated string region are



Note that the qualifiers point to overlapping strings.

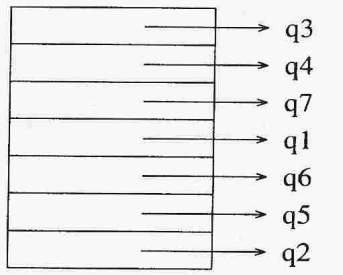
After the location phase, `quallist` might contain the following pointers:



The order of the pointers in `quallist` depends on the order in which the qualifiers are processed: there is no necessary relationship between the order of the pointers in `quallist` and the order of the pointers to the allocated string region.

At the beginning of the pointer-adjustment phase of garbage collection, the array `quallist` is sorted in non-decreasing order by the v-words in qualifiers that are pointed to from `quallist`. This allows the pointers to the allocated string region to be processed in non-decreasing order so that the portions of the allocated string region that must be saved and compacted can be determined.

Continuing the previous example, `quallist` becomes



The v-words of the qualifiers in the order of the pointers in `quallist` now are

strbase+400
strbase+400
strbase+400
strbase+415
strbase+420
strbase+430
strbase+430

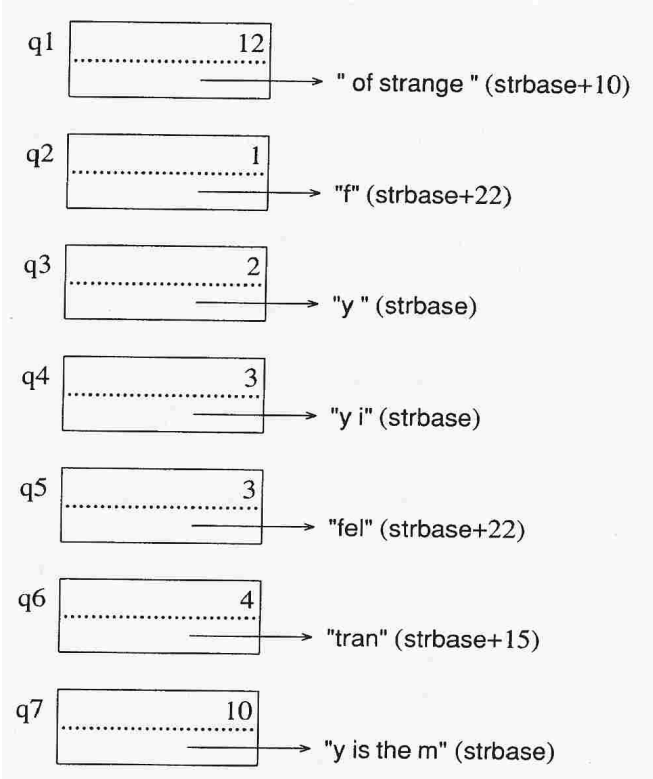
Since qualifiers may reference overlapping strings, care must be taken to identify contiguous "clumps" of characters that may be shared by qualifiers. The pointers in `quallist` are processed in order. Three pointers in the string region are maintained: dest, the next free destination for a clump of characters to be saved; source, the start of the next clump; and cend, the end character in the current clump.

When a qualifier that is pointed to from `quallist` is processed, the first question is whether its v-word addresses a character that is beyond the end of the current clump (since v-words are processed in numerical order, the address is either in the current clump or beyond the end of it). If it is in the current clump, cend is updated, provided the last character of the current qualifier is beyond cend. If it is not in the current clump, the clump is moved from source to dest. In either case, the v-word of the current qualifier is adjusted (dest -source is added to it).

In the previous example, the allocated string region after collection is

```
y is the m of strange fel
↑                       ↑
strbase                 strfree
```

and the seven qualifiers that point to it are



The routine for compacting the allocated string region and adjusting pointers to it is

```
static void scollect(extra)
word extra;
   {
   register char *source, *dest;
   register dptr *qptr;
   char *cend;
   CURTSTATE();

   if (qualfree <= quallist) {
      /*
       * There are no accessible strings.  Thus, there are none
to
       *  collect and the whole string space is free.
       */
      strfree = strbase;
      return;
      }
   /*
    * Sort the pointers on quallist in ascending order of string
    *  locations.
    */
   qsort((char *)quallist, (int)(DiffPtrs((char *)qualfree,(char
*)quallist)) /
     sizeof(dptr *), sizeof(dptr), (QSortFncCast)qlcmp);
   /*
    * The string qualifiers are now ordered by starting location.
    */
   dest = strbase;
   source = cend = StrLoc(**quallist);

   /*
```

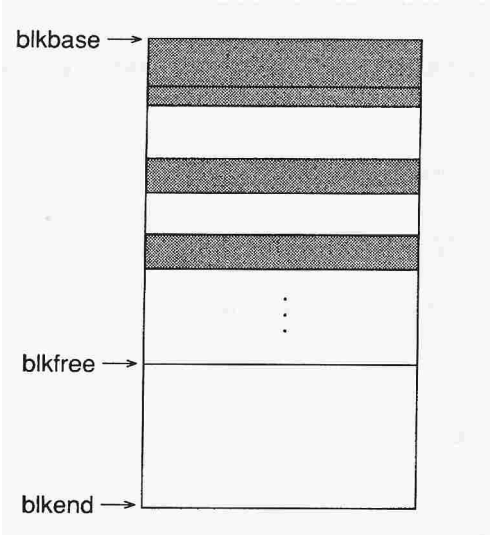
```
    * Loop through qualifiers for accessible strings.
    */
   for (qptr = quallist; qptr < qualfree; qptr++) {
      if (StrLoc(**qptr) > cend) {

         /*
          * qptr points to a qualifier for a string in the next
clump.
          *  The last clump is moved, and source and cend are set
for
          *  the next clump.
          */
         while (source < cend)
            *dest++ = *source++;
         source = cend = StrLoc(**qptr);
         }
      if ((StrLoc(**qptr) + StrLen(**qptr)) > cend)
         /*
          * qptr is a qualifier for a string in this clump;
extend
          *  the clump.
          */
         cend = StrLoc(**qptr) + StrLen(**qptr);
      /*
       * Relocate the string qualifier.
       */
      StrLoc(**qptr) = StrLoc(**qptr) + DiffPtrs(dest,source) +
(uword)extra;
      }

   /*
    * Move the last clump.
    */
   while (source < cend)
      *dest++ = *source++;
   strfree = dest;
   }
```

The argument extra provides an offset in case the string region is moved. See Sec. 11.3.5.

Sorting is done by the C library routine qsort, whose fourth argument is a routine that performs the comparison

```
static int qlcmp(dptr *q1, dptr *q2)
{
return (int)(DiffPtrs(StrLoc(**q1),Strloc(**q2)));
}
```

Blocks. After the location phase, some blocks in the allocated block region are marked and others are not. In the following typical situation, the horizontal lines delimit blocks, gray areas indicate marked blocks, and clear areas indicate unmarked blocks:



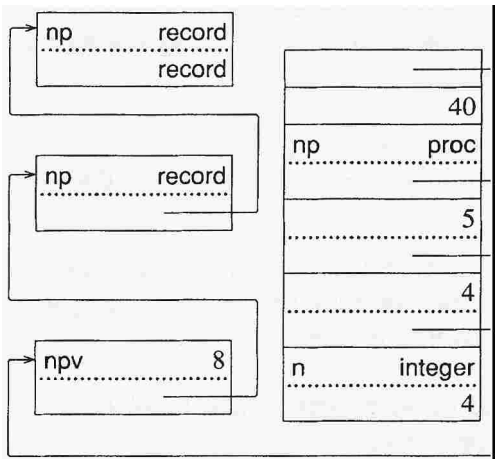
In the allocated block region, pointer adjustment and compaction are done in two linear passes over the region between `blkbase` and `blkfree`. In the first pass, two pointers are used, `dest` and `source`. `dest` points to where the next block will be after blocks are moved in the next pass, while `source` points to the next block to be processed. Both `dest` and source start at `blkbase`, pointing to the first allocated block.

During this pass, the title of each block pointed to by `source` is examined. If it is not marked (that is, if it is not larger than the maximum type code), `dest` is left unchanged and `source` is incremented by the size of the block to get to the title of the next block. Thus, unmarked blocks are skipped. The array `bsizes` is used, as before, to determine block sizes.

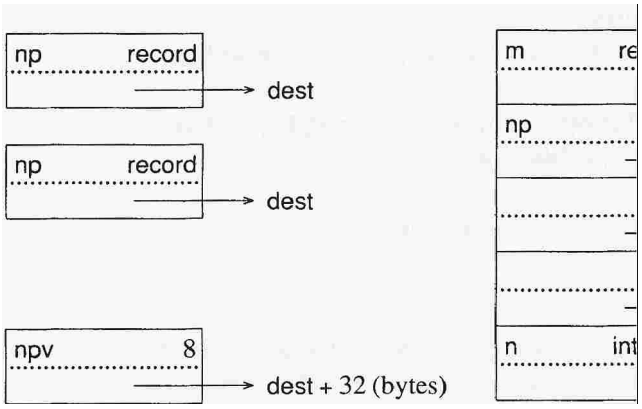
If the title of the block pointed to by `source` is marked, its back chain of descriptors is processed, changing their v-words to point to where `dest` points. In the case of a variable descriptor that is not a trapped-variable descriptor, the offset in its d-word is added to its v-word, so that it points to the appropriate relative position with respect to `dest`.

The last descriptor in the back chain is identified by the fact that its v-word contains a type code (a value smaller than any possible pointer to the allocated block region). This type code is restored to the title of the block before the v-word is changed to point to the destination. An m flag is set in the title to distinguish it as a marked block, since the former marking method no longer applies, but the compaction phase needs to determine which blocks are to be moved.

After the back chain has been processed, all descriptors that point to the block now point to where the block *will be* when it is moved during the compaction phase. The block itself is not moved at this time. This is illustrated by the example given previously, in which three descriptors point to a record block. After marking, the situation is



After processing the back chain, the situation is



Note that the v-words of the descriptors point to where the block *will be* after it is moved.

The routine for adjusting pointers to the allocated block region is

```
static void adjust(char *source, char *dest)
   {
   register union block **nxtptr, **tptr;

   /*
    * Loop through to the end of allocated block region, moving
    *  source to each block in turn and using the size of a block
    *  to find the next block.
    */
   while (source < blkfree) {
      if ((uword)(nxtptr = (union block **)BlkType(source)) >
           MaxType) {

         /*
          * The type field of source is a back pointer.  Traverse
          *  the chain of back pointers, changing each block
          *  location from source to dest.
          */
         while ((uword)nxtptr > MaxType) {
            tptr = nxtptr;
            nxtptr = (union block **) *nxtptr;
            *tptr = (union block *)dest;
            }
         BlkType(source) = (uword)nxtptr | F_Mark;
```

```
         dest += BlkSize(source);
         }
      source += BlkSize(source);
      }
   }
```

When the pointer-adjustment phase is complete, the blocks can be moved. At this time, all the block titles contain type codes, and those that are to be saved are marked by m flags. During the compaction phase, these pointers are used again to reference the destination and source of blocks to be moved.

If an unmarked block is encountered, source is incremented by the block skipping over the block. If a marked block is encountered, the m flag in its is removed and the block is copied to `dest`. Then `dest` and `source` are incremented by the size of the block.

When `blkfree` is reached, it is set to `dest`. At this point the allocated block region has been compacted. All saved blocks are before `blkfree`, and all free space is after it. The pointers that were adjusted now point to their blocks, and the relative situation is the same as it was before garbage collection.

The routine for compacting the allocated block region is

```
static void compact(source)
char *source;
   {
   register char *dest;
   register word size;

   /*
    * Start dest at source.
    */
   dest = source;

   /*
    * Loop through to end of allocated block space, moving source
    *  to each block in turn, using the size of a block to find
    *  the next block. If a block has been marked, it is copied
    *  to the location pointed to by dest and dest is pointed
    *  past the end of the block, which is the location to place
    *  the next saved block.  Marks are removed from the saved
    *  blocks.
    */
   while (source < blkfree) {
      size = BlkSize(source);
      if (BlkType(source) & F_Mark) {
         BlkType(source) &= ~F_Mark;
         if (source != dest)
            mvc((uword)size,source,dest);
         dest += size;
         }
      source += size;
      }

   /*
    * dest is the location of the next free block.  Now that
    *  compaction is complete, point blkfree to that location.
    */
   blkfree = dest;
```

```
}
```

The routine `mvc(n, source, dest)` moves n bytes from source to dest.

11.3.4 Collecting Co-Expression Blocks

After the location phase of garbage collection is complete, all the live co-expression blocks are marked, but the dead co-expression blocks are not. It is a simple matter to process the list of co-expression blocks, which are linked by pointers, calling free to deallocate dead ones and at the same time removing them from the list. For live co-expressions, the type code in the title is restored. The routine cofree that frees co-expression blocks is

```
static void cofree()
{
register struct b_coexpr **ep, *xep;

/*
 * Reset the type for &main.
 */
BlkLoc(k_main)->coexpr.titie = T_Coexpr;
/*
 * The co-expression blocks are linked together through their
 * nextstk fields, with stklist pointing to the head of the
 * list. The list is traversed and each stack that was not
 * marked is freed.
 */
ep = &stklist;
while (*ep != NULL) {
   if (BlkType(*ep) == T_Coexpr) {
      xep = *ep;
      *ep = (*ep)->nextstk;
      /*
       * Free the astkblks. There should always be one and it
       *  seems that it's not possible to have more than one,
       *  but nonetheless, the code provides for more than one
       */
      for (abp = xep->es_actstk; abp; ) {
         xabp = abp;
         abp = abp->astk_nxt;
         free((pointer)xabp);
         }
      coclean(xep->cstate);
      free((char *)xep);
      }
   else {
      BlkType(*ep) = T_Coexpr;
      ep = &(*ep)->nextstk;
      }
   }
}
```

11.3.5 Multiple Regions

Garbage collection may not produce enough free space in a region to satisfy the request that caused the garbage collection. In this case, the region for which the request was made is replaced with a new larger region. In addition, the allocated string and block regions are replaced if the amount of free space in them after garbage collection otherwise would

be less than a minimum value, which is called "breathing room." This attempts to avoid "thrashing" that might result from a garbage collection that leaves a small amount of free space, only to result in a subsequent garbage collection almost immediately.

The set of string and block regions for a program are managed as a linked list. Older, “tenured” regions are revisited, and garbage collected, prior to allocating new, larger regions. If an older region frees enough space, it is made the active region instead of allocating a new one.

11.3.6 Storage Requirements during Garbage Collection

Garbage collection itself takes some work space. Space for pointers to qualifiers is provided in `quallist`, while C stack space is needed for calls to routine that perform the various aspects of garbage collection, which are heavily recursive.

The space for `quallist` is obtained from the free space at the end of the allocated block region. The amount of space needed is proportional to the number of qualifiers whose v-words point to strings in the allocated string region and usually is comparatively small. Space for `quallist` is obtained in small increments

This is done in `postqual()`, for which the complete routine is

```
static void postqual(dptr dp)
   {
   char *newqual;

   if (InRange(strbase,StrLoc(*dp),strfree + 1)) {
      /*
       * The string is in the string space.  Add it to the string
       *  qualifier list, but before adding it, expand the string
       *  qualifier list if necessary.
       */
      if (qualfree >= equallist) {

         /* reallocate a qualifier list that's twice as large */
         newqual = realloc(quallist, 2 * qualsize);
         if (newqual) {
            quallist = (dptr *)newqual;
            qualfree = (dptr *)(newqual + qualsize);
            qualsize *= 2;
            equallist = (dptr *)(newqual + qualsize);
            }
         else {
            qualfail = 1;
            return;
            }
         }
      *qualfree++ = dp;
      }
   }
```

The amount of stack space required during garbage collection depends primarily on the depth of recursion in calls to `markblock()` and `markptr()`. Recursion in these functions corresponds to linked lists of pointers in allocated storage. It occurs where a descriptor in the static region or the allocated block region points to an as-yet unmarked block. C stack overflow may occur during garbage collection. This problem is particularly serious on computers with small address spaces for programs that use a large amount of

allocated data. The use of stack space during marking is minimized by testing descriptor v-words before calling `markblock()`, by using static storage for variables in `markblock()` that are not needed in recursive calls, and by incorporating the code for processing co-expression blocks in `markblock()`, rather than calling a separate routine.

11.4 Predictive Need

In most systems that manage allocated storage dynamically, garbage collections are triggered by allocation requests that cannot be satisfied by the amount of free storage that remains. In these systems, garbage collections occur during calls to allocation routines.

Whenever a garbage collection occurs, all potentially accessible data must be reachable from the basis, and any descriptors that are reachable from the basis must contain valid data. These requirements pose serious difficulties, since, in the normal course of computation, pointers to accessible objects may only exist in registers or on the C stack as C local variables that the garbage collector has no way of locating. Furthermore, descriptors that are being constructed may temporarily hold invalid data. While it is helpful to know that garbage collection can occur only during calls to allocation routines, allocation often is done in the midst of other computations. Assuring that all accessible data is reachable and that all reachable data is valid can be difficult and prone to error.

For these reasons, Icon uses a slightly different strategy, called "predictive need," for triggering garbage collections. Instead of garbage collection occurring as a byproduct of an allocation request, the amount of space needed is requested in advance. There is a routine, `reserve(Region,n)`, for reserving space in advance. This routine checks the specified region to assure the amount of free space needed is actually available. If it is not, it calls the garbage collector. The code for reserve() is conceptually

```
char *reserve(int r, uword n)
{
if (DiffPtrs(regions[r]->end,regions[r]->free < n)
   collect(r,n);
return regions[r]->free;
}
```

In practice, things are more complicated, as the current region may be changed or a new region may be allocated in order to satisfy the request.

The string allocator mainly ensures space is available and then updates the free pointer:

```
char *alcstr(char *s, word slen)
{
tended struct descrip ts;
register char *d;
char *ofree;

/*
 * Make sure there is enough room in the string space.
 */
if (DiffPtrs(strend,strfree) < slen) {
   StrLen(ts) = slen;
   StrLoc(ts) = s;
   if (!reserve(Strings, slen))
      return NULL;
   s = StrLoc(ts);
   }
```

```
strtotal += slen;
/*
 * Copy the string into the string space, saving a pointer to
 * its beginning. Note that s may be null, in which case the
 * space is still to be allocated but nothing is to be copied
 * into it.
 */
ofree = d = strfree;
if (s) {
   while (slen-- > 0)
   *d++ = *s++;
   }
else
   d += slen;
strfree = d;
return ofree;
}
```

If a garbage collection occurs, a parameter is passed to be sure that enough space is collected to satisfy any remaining allocation requests.

Since a predictive need request assures an adequate amount of space, no garbage collection can occur during the subsequent allocation request. The advantage of having a garbage collection occur during a predictive need request rather during an allocation request is that a safe time can be chosen for a possible garbage collection. The amount of space needed (or at least an upper bound on it) usually is known before the storage is actually needed. and when all valid data can be located from the basis.

A few lines from the implementation of the `image()` function, showing how string images are constructed, illustrates predictive need. The image will consist of a pair of double quotes, enclosing a representation of the string contents with special characters escaped. If `alcstr()` is called separately for the various components, they might be allocated non-contiguously. Reserving the maximum space needed ahead of time guarantees subsequent calls to `alcstr()` will be contiguous. The maximum needed for image would be `StrLen(source)*4+2`, done using a shift operator:

```
s = StrLoc(source);
len = StrLen(source);
Protect (reserve(Strings,(len << 2) + 2), return Error);
Protect(t = alcstr("\"", (word)(1)), return Error);
StrLoc(*dp2) = t;
StrLen(*dp2) = 1;
while (len-- > 0)
   StrLen(*dp2) += doimage(*s++, '"');
Protect(alcstr("\"", (word)(1)), return Error);
++StrLen(*dp2);
```

A disadvantage of predictive need is that the maximum amount of storage needed must be determined and care must be taken to make predictive need requests prior to allocation. These problems do not occur in storage-management systems where garbage collection is implicit in allocation.

RETROSPECTIVE: Storage management is one of the major concerns in the implementation of a run-time system in which space is allocated dynamically and automatically. Although

many programs never garbage collect at all, for those that do, the cost of garbage collection may be significant.

The requirements of storage management have a significant influence on the way that data is represented in Icon, particularly in blocks. Aspects of data representation that may appear arbitrary in the absence of considerations related to storage management have definite uses during garbage collection.

The garbage collector can only identify those pointers of which it is informed. Globals are informed by placing them in the basis set. Locals are informed by declaring them to be tended. Descriptors and block pointers within blocks are specified in tables, indexed by the block's type code, that describe the number and positional offset of all pointers within the block.

While it is possible to devise more economical methods of representing such data at the expense of complexity and loss of generality, any method of representing data for which space is allocated automatically has some overhead.

Garbage collection is most expensive when there are many live objects that must be saved. For programs in which allocated storage is used transiently and in which there are few live objects, garbage collection is fast.

EXERCISES

11.1 Since the first word of a block contains its type code, why is there also a type code in a descriptor that points to it?

11.2 Give an example of an Icon expression that changes the contents of a block that is allocated statically in the run-time system.

11.3 Give an example of an Icon expression that changes data in the icode region.

11.4 Why not combine global and static identifiers in a single array of descriptors in the icode region?

11.5 Why are the names of global identifiers needed?

11.6 Why is there no array for the names of static identifiers?

11.7 How long can a string be?

11.8 How many elements can a single list-element block hold?

11.9 List all the regions of memory in which the following Icon data objects can occur:

- strings
- descriptors
- co-expression blocks
- other blocks

11.10 List all the source-language operations in Icon that may cause the allocation of storage.

11.11 Give an example of an expression for which it cannot be determined from the expression itself whether or not it allocates storage.

11.12 List the block types for which block size may vary from one block to another.

11.13 List all the types of blocks that may occur in the allocated block region.

11.14 List all the types of blocks that may occur outside of the allocated block region.

11.15 Give an example of an Icon program in which the only access path to a live object during garbage collection is a variable that points to an element in a structure.

11.16 Give an example of an Icon program that constructs a circular pointer chain.

11.17 Explain how it can be assured that all blocks in the allocated block region are at addresses that are larger than the maximum type code.

11.18 Aside from the possibility of looping in the location phase of garbage collection, what are the possible consequences of processing the descriptors in a block more than once?

11.19 What would happen if there were more than one pointer on quallist to the *same* qualifier?

11.20 Because of the way that the Icon run-time system is written, blocks that are not in the allocated block region do not contain pointers to allocated objects. Consequently, the descriptors in such blocks do not have to be processed during garbage collection.

- What does this imply about access to such blocks?
- What changes would have to be made to the garbage collector if such blocks could contain pointers to allocated objects?

11.21 There is one exception to the statement in the preceding exercise that blocks that are not in the allocated data region do not contain pointers to allocated objects. Identify this exception and explain how it is handled during garbage collection.

11.22 In the allocated string region, pointer adjustment and compaction are done in one pass, while two passes are used in the allocated block region. Why are pointer adjustment and compaction not done in a single pass over the allocated block region?

11.23 What would be the effect of failing to remove the m flag from a block title during the compaction of the allocated block region?

11.24 If garbage collection cannot produce enough free space in the region for which the collection was triggered, program execution is terminated even if there is extra space in another region. Describe how to modify the garbage collector to avoid this problem.

11.25 Write a program that requires an arbitrarily large amount of space for quallist.

11.26 Write a program that causes an arbitrary amount of recursion in markblock during garbage collection.

11.27 Write a program that produces an arbitrarily large amount of data that must be saved by garbage collection, and observe the results.

11.28 Devise a more sophisticated method of preventing thrashing in allocation and garbage collection than the fixed breathing-room method.

11.29 There is no mechanism to reduce the size of an allocated region that may be expanded during one garbage collection, but which has an excessive amount of free space after another garbage collection. Describe how to implement such a mechanism.

11.30 Suppose that a garbage collection could occur during a call of any C routine from any other C routine. How would this complicate the way C routines are written?

11.31 What might happen if

- The amount of storage specified in a predictive need request were larger than the amount subsequently allocated?
- The amount of storage specified in a predictive need request were smaller than the amount subsequently allocated?

11.32 When a list-element block is unlinked as the result of a pop, get, or pull, can the space it occupies always be reclaimed by a garbage collection? What are the general considerations in answering questions such as these?

11.33 A variable that refers to a descriptor in a block points directly to the descriptor, with an offset in its d-word to the head of the block in which the descriptor resides. Could it be the other way around, with a variable pointing to the head of the block and an offset to the descriptor? If so, what are the advantages and disadvantages of the two methods?

11.34 Why does sweep process an interpreter stack from its sp to its base, rather than the other way around?

11.35 As mentioned in Sec. 11.3, all regions are collected, regardless of the region in which space is needed. Discuss the pros and cons of this approach.

11.36 Evaluate the cost of using two-word descriptors for all pointers to blocks, even when these pointers do not correspond to source-language values (as, for example, in the links among list-element blocks).

11.37 The need to garbage-collect blocks that are allocated during program execution significantly affects the structure and organization of such blocks. Suppose that garbage collection were never needed. How could the structure and organizations of blocks be revised to save space?

11.38 Discuss the pros and cons of having different regions for allocating blocks of different types.

11.39 Some expressions, such as

```
while write(read())
```

result in a substantial amount of "storage throughput," even though no space really needs to be allocated. Explain why this effect cannot be avoided in general and discuss its impact on program performance.

11.40 Physical memory is becoming less and less expensive, and more computer architectures and operating systems are providing larger user address spaces. Discuss how very large user address spaces might affect allocation and garbage-collection strategies.

Chapter 12: Run-Time Support Operations

PERSPECTIVE: Several features of Icon's run-time system cannot be compartmentalized as neatly as storage management but present significant implementation problems nonetheless. These features include type checking and conversion dereferencing and assignment, input and output, and diagnostic facilities.

12.1 Type Checking and Conversion

Type checking is relatively straightforward in Icon. If only one type is of interest a test of the d-word is sufficient, as in

```
if (Type(Arg1) != T_List)
   runerr(108, &Arg1);
```

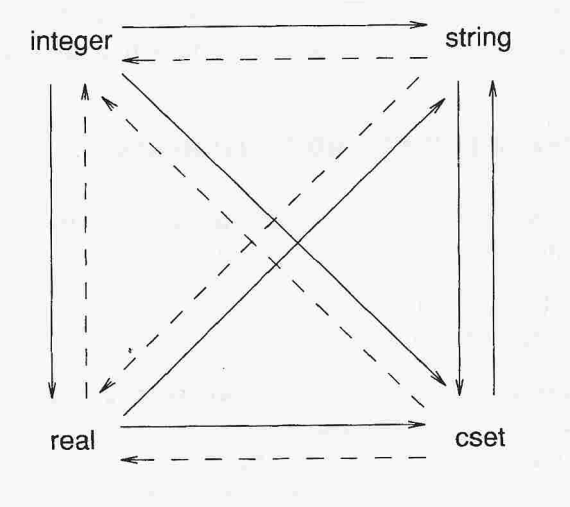
It is necessary to test the entire d-word, since a qualifier may have a length that is the same as a type code. The d-word test takes care of this, because all descriptors that are not qualifiers have n flags.

If different actions are needed for different types, a separate test is required for qualifiers, since there is no type code for strings. The RTL runtime language's `type_case` statement looks like a switch, but is really performing a selection according to type generally of the form:

```
if (is:string(Arg1))    /* string */
else switch (Type(Arg1) {
   case T_List:    /* list */
```

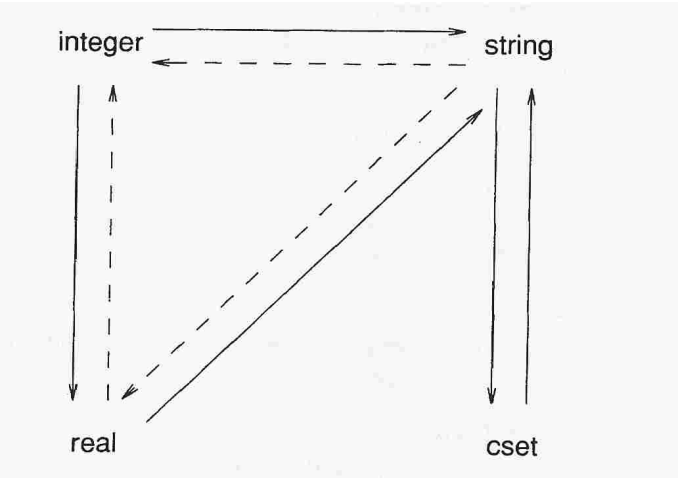
The real problems lie in type conversion, not type checking. At the source-language level, type conversion can occur explicitly, as a result of type-conversion functions, such as `string(x)`, or it may be implicit. Implicit type conversion occurs frequently in many kinds of computations. For example, numeric data may be read from files in the form of strings, converted to numbers in arithmetic computations, and then converted to strings that are written out. Many operations support this implicit type conversion, and they rely on type-conversion routines.

There are four types among which mutual conversion is supported: strings, csets, integers, and real numbers. The details of type conversion are part of the Icon language definition (Griswold and Griswold 1983). For example, when a cset is converted to a string, the characters of the resulting string are in lexical order. Some conversions are conditional and may succeed or fail, depending on the value being converted. For example, a real number can be converted to an integer only if its value is in the range of a C long. The conversions are illustrated in the following diagram, where dashed lines indicate conversions that are conditional:



Thus, of the twelve conversions, five are conditional.

Some kinds of conversions are "natural" and occur frequently in typical programs. Examples are string-to-integer conversion and integer-to-string conversion. Other conversions, such as cset-to-integer, are unlikely to occur in the normal course of computation. To reduce the number of conversion routines required, these unlikely conversions are done in two steps. For example, the conversion of a cset to an integer is done by first converting the cset to a string and then converting the string to an integer. The direct conversions are



Conversions are done by calling routines that convert values to expected types. These routines are

`cnv_cset`	convert to cset
`cnv_int`	convert to integer
`cnv_real`	convert to real
`cnv_str`	convert to string

Since these routines may be called with any type of value, all of them are conditional. For example, it is not possible to convert a list to a string. These routines return the value `CvtFail` to indicate the failure of a conversion. If conversion is successful, they return a value indicating the type of conversion.

Numerical computation introduces complications in addition to the types integer and real, since there is the concept of a numeric "type" that includes both integers and real numbers. This is represented explicitly by the Icon type-conversion function `numeric(x)`, which converts `x` to either integer or real, depending on the value of `x`. Its implementation illustrates RTL's extensions to the C language for type conversions. Rtt can generate special-case functions for each call to `numeric(x)`, based on what it knows about the type information at that call site, and skip the checks and conversions where they are not necessary.

```
function{0,1} numeric(n)
   if cnv:(exact)integer(n) then {
      abstract { return integer }
      inline   { return n; }
      }
   else if cnv:real(n) then {
      abstract { return real }
      inline   { return n; }
      }
   else {
      abstract { return empty_type }
      inline   { fail; }
      }
end
```

Numeric conversion also occurs implicitly in polymorphic operations such as

```
   n+m
```

which performs integer or real arithmetic, depending on the types of `n` and `m`. The RTL language was given an `arith_case` construct specifically to handle this issue. The general form of the arithmetic operators looks like:

```
   operator{1} icon_op func_name(x, y)
      declare {
         tended struct descrip lx, ly;
       C_integer irslt;
         }
     arith_case (x, y) of {
         C_integer: {
            abstract { return integer }
            inline {
               extern int over_flow;
               c_int_op(x,y);
               }
            }
         integer: { /* large integers only */
            abstract { return integer }
            inline {
               big_ ## c_int_op(x,y);
               }
            }
         C_double: {
            abstract { return real }
            inline {
               c_real_op(x, y);
               }
            }
         }
   end
```

Internally, a macro `GetReal()` is used to handle the real type, since some computers have restrictions on the alignment of doubles. Thus, `GetReal()` has different definitions depending on the target computer. The actual conversion of a string to a numeric value is done by `ston()`. Note that the conversion of a cset to a numeric value occurs by way of conversion to a string.

When types are not known at compile-time, RTL constructs such as `cnv:str(d)` are translated down to conversion routines in cnv.r. String conversion requires a buffer to construct the string. In a common special-case, this buffer is provided by the routine that is requesting string conversion for temporary use, avoiding the heap memory allocator. This is used, for example when a value is converted to a string in order to convert it to a number. See Sec. 4.4.4. The code for the "temporary string" version of string conversion is in a function `tmp_str()`:

```
static int tmp_str(char *sbuf, dptr s, dptr d)
   {
   type_case *s of {
      string:
         *d = *s;
      integer: {
         if (Type(*s) == T_Lrgint) {
            word slen, dlen;
            slen = (BlkLoc(*s)->bignumblk.lsd -
                    BlkLoc(*s)->bignumblk.msd +1);
            dlen=slen * NB * 0.3010299956639812; /* 1/log2(10) */
            bigtos(s,d);
            }
         else
            itos(IntVal(*s), d, sbuf);
         }
      real: {
         double res;
         GetReal(s, res);
         rtos(res, d, sbuf);
         }
      cset:
         cstos(BlkLoc(*s)->cset.bits, d, sbuf);
      default:
         return 0;
      }
   return 1;
   }
```

If a conversion is required, `itos()`, `rtos()`, or `cstos()` does the actual work, placing its result in sbuf and changing the descriptor pointed to by d accordingly. These routines return a 1 if the result is a string and a 0 otherwise. The monitoring facilities in Unicon can also report whether no conversion was needed (`E_Nconv`), a conversion was performed successfully (`E_Sconv`), or the conversion failed (`E_Fconv`).

If a converted string is in a buffer that is local to the calling routine, it must be copied into allocated storage, or it would be destroyed when that routine returns. The fully general version of `cnv_str()` is equivalent to the following function. Because this function is heavily called, in reality the body of `tmp_str()` is inlined in `cnv_str()`.

```
int cnv_str(dptr s, dptr d)
   {
   char sbuf[MaxCvtLen];
```

```
type_case *s of {
   string: {
      *d = *s;
      return 1;
      }
   default: {
      if (!tmp_str(sbuf, s, d)) return 0;
      }
   }
Protect(StrLoc(*d) =
           alcstr(StrLoc(*d), StrLen(*d)), fatalerr(0,NULL));
return 1;
}
```

12.2 Dereferencing and Assignment

If there were no trapped variables, dereferencing and assignment would be trivial. For example, the descriptor d is dereferenced by

```
d = *VarLoc(d)
```

where `VarLoc` references the v-word of d:

```
#define VarLoc(d) ((d).vword.dptr)
```

The dereferencing or assignment to a trapped variable, on the other hand, may involve a complicated computation. This computation reflects the meaning associated with the operation on the source-language expression that is represented in the implementation as a trapped variable. For example, as discussed previously, in

```
x[y] := z
```

the value of x may be a list, a string, a table, or a record. A subscripted list or record does not produce a trapped variable, but the other two cases do. For a string, the variable on the left side of the assignment is a substring trapped variable. For a table, the variable is a table-element trapped variable. In the first case, the assignment involves the concatenation of three strings and the assignment of the result to x. In the second case, it involves looking for y in the table. If there is a table element for y, its assigned value is changed to the value of z. Otherwise, the table-element trapped-variable block is converted to a table-element block with the assigned value, and the block is inserted in the appropriate chain.

12.2.1 Dereferencing

Dereferencing of other trapped variables involves computations of comparable complexity. Dereferencing is done in the interpreter loop for arguments of operators for which variables are not needed. For example, in

```
n+m
```

the identifiers n and m are dereferenced before the function for addition is called (See Sec. 8.3.1). On the other hand, in

```
s[i]
```

the identifier i is dereferenced, but s is not, since the subscripting routine needs the variable as well as its value.

The function invocation routine also dereferences variables before a function is called. Note that there is no function that requires an argument that is a variable. Suspension and

return from procedures also dereference local identifiers and arguments. Dereferencing occurs in a number of other places. For example, the function that handles subscripting must dereference the subscripted variable to determine what kind of result to produce.

The dereferencing routine begins as follows:

```
void deref(s, d)
dptr s, d;
   {
   /*
    * no allocation is done, so nothing need be tended.
    */
   register union block *bp, **ep;
   struct descrip v;
   int res;

   if (!is:variable(*s)) {
      *d = *s;
      }
```

If s does not point to a variable descriptor, the remaining code is skipped.

If s points to a variable that is not a trapped variable, dereferencing is simple:

```
         /*
          * An ordinary variable is being dereferenced.
          */
         *d = *(dptr)((word *)VarLoc(*s) + Offset(*s));
```

Otherwise, there are three types of trapped variables with a switch on the type:

```
   type_case *s of {
      tvsubs: {
         /*
          * A substring trapped variable is being dereferenced.
          *  Point bp to the trapped variable block and v to
          *  the string.
          */
         bp = BlkLoc(*s);
         deref(&bp->tvsubs.ssvar, &v);
         if (!is:string(v))
            fatalerr(103, &v);
         if (bp->tvsubs.sspos + bp->tvsubs.sslen - 1 > StrLen(v))
            fatalerr(205, NULL);
         /*
          * Make a descriptor for the substring by getting the
          *  length and pointing into the string.
          */
         StrLen(*d) = bp->tvsubs.sslen;
         StrLoc(*d) = StrLoc(v) + bp->tvsubs.sspos - 1;
        }

      tvtbl: {
         /*
          * Look up the element in the table.
          */
         bp = BlkLoc(*s);
         ep = memb(bp->tvtbl.clink,&bp->tvtbl.tref,
                   bp->tvtbl.hashnum,&res);
         if (res == 1)
```

```
            *d = (*ep)->telem.tval;   /* found; use value */
         else
            *d = bp->tvtbl.clink->table.defvalue;/*use default */
         }
```

A table-element trapped variable may point to a table-element trapped-variable block or to a table-element block. The second situation occurs if two table-element trapped variables point to the same table-element trapped-variable block and assignment to one of the variables converts the table-element trapped-variable block to a table-element block before the second variable is processed. See Sec. 7.2. In this case, the value of the trapped variable is in the table-element block. On the other hand, if the trapped variable points to a table-element trapped-variable block, it is necessary to look up the subscripting value in the table, since an assignment for it may have been made between the time the trapped variable was created and the time it was dereferenced. If it is in the table, the corresponding assigned value is returned. If it is not in the table, the default assigned value is returned.

The last case, keyword trapped variables, is almost the same as for simple variables. These variables impose special semantics on assignment, but not on dereferencing.

```
      kywdint:
      kywdpos:
      kywdsubj:
      kywdevent:
      kywdwin:
      kywdstr:
         *d = *VarLoc(*s);
```

12.2.2 Assignment

The values of global identifiers are established initially as a byproduct of reading the icode file into the icode region. When procedures are called, the values of arguments and local identifiers are on the interpreter stack. These operations associate values with variables, but assignment, unlike dereferencing, is explicit in the source program.

The macro `GeneralAsgn()` is used to perform all such operations. For example, the function for

```
   x := y
```

is

```
operator{0,1} := asgn(underef x, y)
   if !is:variable(x) then
      runerr(111, x)
   abstract {
      return type(x)
      }
   GeneralAsgn(x, y)
   inline {
      /*
       * The returned result is the variable to which assignment
       *  is being made.
       */
      return x;
      }
end
```

Note that assignment may fail. This can occur as the result of an out-of-range assignment to &pos and is indicated by an RTL `fail` statement from within `GeneralAsgn()`.

Like dereferencing, assignment is trivial for variables that are not trapped. The macro `GeneralAsgn()` begins as follows:

```
#begdef GeneralAsgn(x, y)

   type_case x of {
```

As for dereferencing, there are three types of trapped variables to be considered. Assignment to a substring trapped variable is rather complicated and deferred to a function `subs_asgn()`:

```
      tvsubs: {
        abstract {
           store[store[type(x).str_var]] = string
           }
        inline {
           if (subs_asgn(&x, (const dptr)&y) == Error)
              runerr(0);
           }
        }
```

The function subs_asgn():

```
int subs_asgn(dptr dest, const dptr src)
   {
   tended struct descrip deststr, srcstr, rsltstr;
   tended struct b_tvsubs *tvsub;

   char *s, *s2;
   word i, len;
   word prelen;/* length of portion of string before substring */
   word poststrt, postlen; /* start and length of portion of
                              string following substring */
   if (!cnv:tmp_string(*src, srcstr))
      ReturnErrVal(103, *src, Error);
   /*
    * Be sure that the variable in the trapped variable points
    *  to a string and that the string is big enough to contain
    *  the substring.
    */
   tvsub = (struct b_tvsubs *)BlkLoc(*dest);
   deref(&tvsub->ssvar, &deststr);
   if (!is:string(deststr))
      ReturnErrVal(103, deststr, Error);
   prelen = tvsub->sspos - 1;
   poststrt = prelen + tvsub->sslen;
   if (poststrt > StrLen(deststr))
      ReturnErrNum(205, Error);

   /*
    * Form the result string.
    *  Start by allocating space for the entire result.
    */
   len = prelen + StrLen(srcstr) + StrLen(deststr) - poststrt;
   Protect(s = alcstr(NULL, len), return Error);
   StrLoc(rsltstr) = s;
   StrLen(rsltstr) = len;
```

```
   /*
    * First, copy the portion of the substring string to
    *  the left of the substring into the string space.
    */
   s2 = StrLoc(deststr);
   for (i = 0; i < prelen; i++)
      *s++ = *s2++;
   /*
    * Copy the string to be assigned into the string space,
    *  effectively concatenating it.
    */
   s2 = StrLoc(srcstr);
   for (i = 0; i < StrLen(srcstr); i++)
      *s++ = *s2++;
   /*
    * Copy the portion of the substring to the right of the
    *  substring into the string space, completing the result.
    */
   s2 = StrLoc(deststr) + poststrt;
   postlen = StrLen(deststr) - poststrt;
   for (i = 0; i < postlen; i++)
      *s++ = *s2++;

   /*
    * Perform the assignment and update the trapped variable.
    */
   type_case tvsub->ssvar of {
      kywdevent: {
         *VarLoc(tvsub->ssvar) = rsltstr;
         }
      kywdstr: {
         *VarLoc(tvsub->ssvar) = rsltstr;
         }
      kywdsubj: {
         *VarLoc(tvsub->ssvar) = rsltstr;
         k_pos = 1;
         }
      tvtbl: {
         if (tvtbl_asgn(&tvsub->ssvar, (const dptr)&rsltstr) ==
               Error)
            return Error;
         }
      default: {
         Asgn(tvsub->ssvar, rsltstr);
         }
      }
   tvsub->sslen = StrLen(srcstr);
   return Succeeded;
   }
```

Table-element trapped variables are once again deferred in GeneralAsgn() to call to a function, tvtbl_asgn():

```
      tvtbl: {
        abstract {
           store[store[type(x).trpd_tbl].tbl_val] = type(y)
           }
        inline {
```

```
            if (tvtbl_asgn(&x, (const dptr)&y) == Error)
               runerr(0);
            }
         }
```

Table-element trapped variables have the same possibilities for assignment as for dereferencing. The processing is more complicated, since it may be necessary to convert a table-element trapped-variable block into a table-element block and link it into a chain. Parameters cannot be tended, so their information must be preserved in tended variables before anything is allocated.

```
int tvtbl_asgn(dptr dest, const dptr src)
   {
   tended struct b_tvtbl *bp;
   tended struct descrip tval;
   struct b_telem *te;
   union block **slot;
   struct b_table *tp;
   int res;

   /*
    * Allocate table element now (even if we may not need it)
    * because slot cannot be tended. Parameters have to be
    * preserved in tended variables first.
    */
   bp = (struct b_tvtbl *) BlkLoc(*dest);
   tval = *src;
   Protect(te = alctelem(), return Error);

   /*
    * First see if reference is in the table; if it is, just
    *  update the value.  Otherwise, allocate a new table entry.
    */
   slot = memb(bp->clink, &bp->tref, bp->hashnum, &res);

   if (res == 1) {
      /*
       * Do not need new te, just update existing entry.
       */
      deallocate((union block *) te);
      (*slot)->telem.tval = tval;
      }
   else {
      /*
       * Link te into table, fill in entry.
       */
      tp = (struct b_table *) bp->clink;
      tp->size++;

      te->clink = *slot;
      *slot = (union block *) te;

      te->hashnum = bp->hashnum;
      te->tref = bp->tref;
      te->tval = tval;

      if (TooCrowded(tp))  /* grow hash table if now too full */
         hgrow((union block *)tp);
```

```
   }
return Succeeded;
}
```

In the case of a keyword trapped variable, the semantic requirements of the keyword are expressed in the usual mixture of RTL and C, except that type information is guaranteed not to change. The code for `&subject` is typical:

```
kywdsubj: {
   /*
    * No side effect in the type realm = no abstract clause
    *  &subject is still a string and &pos is still an int.
    */
   if !cnv:string(y, *VarLoc(x)) then
      runerr(103, y);
   inline {
      k_pos = 1;
      }
   }
```

12.3 Input and Output

Icon supports only sequential file access. The run-time system uses C library routines to perform input and output, so the main implementation issues are those that relate to interfacing these routines.

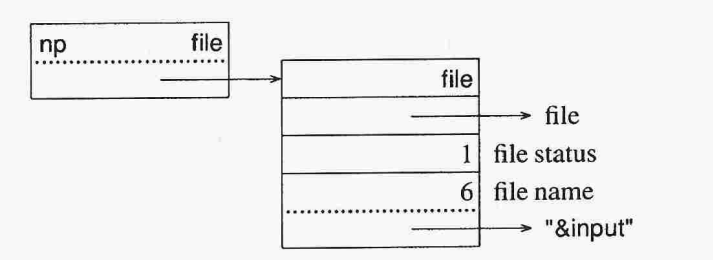
12.3.1 Files

A value of type file in Icon points to a block that contains the usual title word, a `FILE *` reference to the file, a status word, and the string name of the file. The file status values include

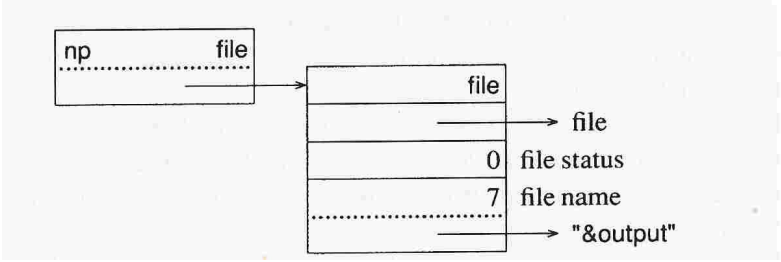
0	closed
1	open for reading
2	open for writing
4	open to create
8	open to append
16	open as a pipe

These decimal numbers correspond to bits in the status word.

For example, the value of `&input` is



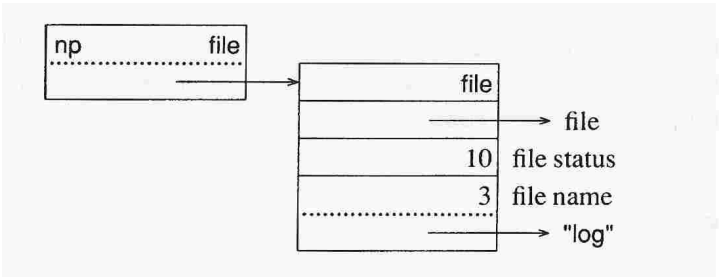
while the value of `&output` is



Another example is

```
out := open("log", "a")
```

for which the value of out is

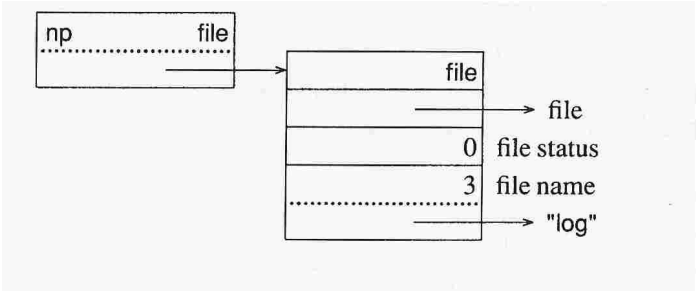


Note that the file status is 10, corresponding to being open for writing and appending.

Closing a file, as in

```
close(out)
```

merely changes its file status:



12.3.2 Reading and Writing Data

The function `read(f)` reads a line from the file `f`. In UNIX, a line is just a string of characters up to a newline character. There is no limit on the length of a line and the length of a line cannot be determined before it is read. On the other hand, there must be a place to store the line.

Characters are read into a buffer until a newline character is encountered or the buffer size (by default 512) is reached. A predictive need request is then made to assure that there is enough space in the allocated string region for the string, and the string is copied from the buffer into the string region. This is repeated as needed.

The function `reads(f,i)` reads `i` characters from the file `f`. These characters may include newline characters. There is no limit on `i` other than available memory. A predictive need request can be made to assure that there is enough space in the allocated string region. Characters are then read directly into the allocated string region without the use of an intervening buffer.

When strings are written, they are written directly from the allocated string region. There is no need to perform any allocation or to use an intermediate buffer.

Several strings can be concatenated on a file by

```
write(s1 , s2, ..., sn)
```

This avoids the internal allocation and concatenation that is required for

```
write(s1 || s2 || ...|| sn)
```

12.4 Diagnostic Facilities

Icon's diagnostic facilities consist of

- The function `image(x)`, which produces a string representation of the value of x.
- The function `display(f, i)`, which writes the names and values of identifiers in at most `i` levels of procedure call to the file f.
- Tracing of procedure calls, returns, resumptions, and suspensions.
- Run-time error termination messages.

Procedure tracing is done in the virtual machine instructions invoke, pret, pfail, and psusp. If the value of `&trace` is nonzero, it is decremented and an appropriate trace message is written to standard error output. See Sec. 2.1.12 for an example.

The function `display(f,i)` must locate the names and values of local identifiers and arguments. The names are in the procedure block for the current procedure, which is pointed to by the zeroth argument of the current procedure call. The values are on the interpreter stack as described in Sec. 10.3.3.

Run-time termination messages are produced by the C routine `runerr(n,dp)`, where `dp` is a pointer to the descriptor for the offending value. The value NULL is used for `dp` in cases where there is no offending value to print.

In all of these diagnostic situations, string representations of values are needed. The string representation for the "scalar" types string, cset, integer, and real is similar to what it is in the text of a source-language program. Long strings and csets are truncated to provide output that is easy to read. Other types present a variety of problems. For procedures, the type and procedure name are given.

A list, on the other hand, may be arbitrarily large and may contain values of any type, even lists. While the name may suffice for a procedure, often more information about a list is needed. As a compromise between information content and readability, only the first three and last three elements of a long list are included in its string representation. Since lists and other non-scalar types may be elements of lists, their representation as elements of a list is more restricted, with only the type and size being shown.

Since trace, display, and error output are written to files, the string representations can be written as they are determined, without regard for how long they are. The function `image(x)`, on the other hand, returns a string value, and space must be allocated for it. A more limited form of string representation is used for non-scalar values, since the space needed might otherwise be very large.

EXERCISES

12.1 It is possible to conceive of meaningful ways to convert *any* type of data in Icon to any other. For example, a procedure might be converted to a string that consists of the procedure declaration. How would such a general conversion feature affect the way that types are converted in the run-time system?

12.2 On computers with 16-bit words, Icon has two representations for integers internally (see Sec. 4.1.3). Describe how this complicates type conversion.

12.3 How would the addition of a new numeric type, such as complex numbers, affect type conversion?

12.4 How big would MaxCvtLen be if Icon had 512 different characters? 128? 64?

12.5 List all the source-language operations that perform assignment.

12.6 Assuming that x, y, z, and w all have string values, diagram the structures that are produced in the course of evaluating the following expressions:
x[y] := z
z := x[y]
x[y] := z[w]
x[y][z] := w
Repeat this exercise for the case where all the identifiers have tables as values.

12.7 Give an expression in which a table-element trapped variable points to a table-element block rather than to a table-element trapped-variable block.

12.8 Give an expression in which a table-element trapped variable points to a table-element trapped-variable block, but where there is a table-element block in the table with the same entry value.

12.9 Why are tended descriptors needed in assignment but not in dereferencing?

12.10 Show an expression in which, at the end of the case for assignment to a substring trapped variable, the variable to which the assignment is to be made is a trapped variable. Can such a trapped variable be of any of the three types?

12.11 Why is the string produced by `read(f)` not read directly into the allocated string region?

12.12 Are there any circumstances in which `write(x1, x2, ..., xn)` requires the allocation of storage?

12.13 Identify all the portions of blocks for source-language values that are necessary only for diagnostic output. How significant is the amount of space involved?

12.14 The use of trapped variables for keywords that require special processing for assignment suggests that a similar technique might be used for substring and table-element trapped variables. Evaluate this possibility.

Part II: An Optimizing Compiler for Icon

by Kenneth W. Walker

Preface to Part II

There are many optimizations that can be applied while translating Icon programs. These optimizations and the analyses needed to apply them are of interest for two reasons. First, Icon's unique combination of characteristics requires developing new techniques for implementing them. Second, these optimizations are useful in variety of languages and Icon can be used as a medium for extending the state of the art.

Many of these optimizations require detailed control of the generated code. Previous production implementations of the Icon programming language have been interpreters. The virtual machine code of an interpreter is seldom flexible enough to accommodate these optimizations and modifying the virtual machine to add the flexibility destroys the simplicity that justified using an interpreter in the first place. These optimizations can only reasonably be implemented in a compiler. In order to explore these optimizations for Icon programs, a compiler was developed. This part of the compendium describes the compiler and the optimizations it employs. It also describes a run-time system designed to support the analyses and optimizations.

Icon variables are untyped. The compiler contains a type inferencing system that determines what values variables and expression may take on during program execution. This system is effective in the presence of values with pointer semantics and of assignments to components of data structures.

The compiler stores intermediate results in temporary variables rather than on a stack. A simple and efficient algorithm was developed for determining the lifetimes of intermediate results in the presence of goal-directed evaluation. This allows an efficient allocation of temporary variables to intermediate results.

The compiler uses information from type inferencing and liveness analysis to simplify generated code. Performance measurements on a variety of Icon programs show these optimizations to be effective.

The optimizing compiler for Icon was developed by Ken Walker as part of his Ph.D. research, and this part of the Icon/Unicon Compendium is essentially a reprint of his dissertation, which also appeared as University of Arizona CS TR 91-16. Along with his consent, Ken kindly provided the original groff sources to his dissertation. Any typographical and formatting errors that remain are the fault of the editor.

Chapter 13: The Optimizing Compiler

Iconc is a practical and complete optimizing compiler for a unique and complex programming language. Part II describes the theory behind several parts of the compiler and describes the implementation of all interesting aspects of the compiler.

13.1 Motivation

The motivation for developing a compiler for the Icon programming language is to have a vehicle for exploring optimization techniques. Some performance improvements can be obtained by modifying the run-time system for the language, for example by implementing alternative data structures or storage management techniques. These improvements may apply to a broad class of programs and the techniques can reasonably be implemented in an interpreter system. However, other techniques, such as eliminating unnecessary type checking, apply to expressions within specific programs. The Icon interpreter described in Part I is based on a virtual machine with a relatively small instruction set of powerful operations. A small instruction set is easier to implement and maintain than a large one, and the power of many of the individual operations insures that the overhead of the decoding loop is not excessive. The disadvantage of this instruction set is that an Icon translator that generates code for the interpreter does not have enough flexibility to do many of the possible program-specific optimizations. It is possible to devise a set of more primitive virtual machine instructions that expose more opportunities for these optimizations. Increasingly primitive instruction sets provide increasingly more opportunities for optimizations. In the extreme, the instruction set for a computer (hardware interpreter) can be used and the translator becomes a compiler. A compiler was chosen for this research because it is a good vehicle for exploring program-specific optimizations and eliminates the overhead of a software interpreter which might otherwise become excessive.

13.2 Type Inferencing

Most Icon operations require operands with specific types. The types of the actual operands in an expression must be checked and possibly converted to the required types. However, Icon variables are untyped; in general, this checking cannot done at translation time. The Icon interpreter takes the simple approach to the problem and performs all of the type checking for an expression every time it is executed. For most programs, a *type inferencing system* can provide the information needed to do much of the checking at translation time, eliminating the need for these checks at run time. A type inferencing system determines the types that elements of a program (variables, expression, procedures, etc) can take on at run time. The Icon compiler contains an effective and practical type inferencing system, and implements code generation optimizations that make use of the information produced by the type inferencing system.

Two basic approaches have been taken when developing type inferencing schemes. Schemes based on unification [.Milner,smltlk type,unify.] construct type signatures for procedures; schemes based on global data flow analysis [.typinfer, typrcsv, flwanal, progflw.] propagate throughout a program the types variables may take on. One strength

of the unification approach is that it is effective at handling polymorphous procedures. Such schemes have properties that make them effective in implementing flexible compile-time type systems. Much of the research on them focuses on this fact. The primary purpose of the type inferencing system for the Icon compiler is to eliminate most of the run-time type checking rather than to report on type inconsistencies at compile time, so these properties have little impact on the choice of schemes used in the compiler. Type inferencing systems based on unification have a significant weakness. Procedure type-signatures do not describe side effects to global variables. Type inferencing schemes based on unification must make crude assumptions about the types of these variables.

Schemes based on global data flow analysis handle global variables effectively. Many Icon programs make significant use of global variables; this is a strong argument in favor of using this kind of type inferencing scheme for Icon. These schemes do a poor job of inferring types in the presence of polymorphous procedures. It is generally too expensive for them to compute the result type of a call in terms of the argument types of that specific call, so result types are computed based on the aggregate types from all calls. Poor type information only results if polymorphism is actually exploited within a program.

The primary use of polymorphous procedures is to implement abstract data types. Icon, on the other hand, has a rich set of built-in data types. While Icon programs make heavy use of these built-in data types and of Icon's polymorphous built-in operations, they seldom make use of user-written polymorphous procedures. While a type inferencing scheme based on global data flow analysis is not effective in inferring the precise behavior of polymorphous procedures, it is effective in utilizing the predetermined behavior of built-in polymorphous operations. These facts combined with the observation that Icon programs often make use of global variables indicate that global data flow analysis is the approach of choice for type inferencing in the Icon compiler.

Icon has several types of non-applicative data structures with pointer semantics. They all can be heterogeneous and can be combined to form arbitrary graphs. An effective type inferencing system must handle these data structures without losing too much information through crude assumptions. These composite data structures typically consist of a few basic elements used repeatedly and they logically have a recursive structure. A number of type inferencing systems handle recursion in applicative data structures [.analrcsv,prlgtyp,typrcsv.]; the system described here handles Icon data types that have pointer semantics and handles destructive assignment to components of data structures. Analyses have been developed to handle pointer semantics for problems such as allocation optimizations and determining pointer aliasing to improve other analyses. However, most of these analyses lose too much information on heterogeneous structures of unbounded depth (such as the mutually referencing syntax trees and symbol tables commonly found in a translator) to be effective type inferencing systems [.progflw,depptr.].

Work by Chase, Wegman, and Zadeck [.pntstr.] published subsequent to the original technical report on the Icon type inferencing system [.tr88-25.] presents a technique similar to the one used in this type inferencing system. They use a minimal language model to describe the use of the technique for pointer analysis. They speculate that the technique might be too slow for practical use and propose methods of improving the technique in the context of pointer analysis. Use of the prototype Icon type inferencing system described in the original technical report indicates that memory usage is more of a

problem than execution time. This problem is addressed in the implementation of type inferencing in the Icon compiler.

13.3 Liveness Analysis

Type checking optimizations can be viewed as forms of argument handling optimizations. Other argument handling optimizations are possible. For example, when it is safe to do so, it is more efficient to pass a variable argument by reference than to copy it to a separate location and pass a reference to that location (this particular opportunity for optimization arises because of implementation techniques borrowed from the Icon interpreter -- Icon values are larger than pointers and Icon parameter passing is built on top of C parameter passing). Such optimizations are not possible in a stack-based execution model; a temporary-variable model is needed and such a model is used by the Icon compiler. Icon's goal-directed evaluation can extend the lifetime of the intermediate values stored in temporary variables. Icon presents a unique problem in *liveness analysis*, which is the static determination of the lifetime of values in a program [ASU86, progflw.]. While this problem, like other liveness problems, can be solved with traditional techniques, it has enough structure that it can be solved without precomputing a flow graph or using expensive forms of data flow analysis.

The only previous implementation of Icon using a temporary-variable model is a partial implementation by Christopher [.tccompile.]. Christopher uses the fact that Icon programs contain many instances of bounded goal-directed evaluation to deduce limits for the lifetimes of intermediate values. However, this approach produces a very crude estimate for these lifetimes. While overestimating the lifetime of intermediate values results in a safe allocation of temporary variables to these values, a fine-grained liveness analysis results in the use of fewer temporary variables. The Icon compiler addresses this problem of fine-grained liveness analysis in the presence of goal-directed evaluation and addresses the problem of applying the information to temporary variable allocation.

13.4 Analyzing Goal-Directed Evaluation

Many kinds of analyses of Icon programs must deal with Icon's goal-directed evaluation and its unique control structures. These analyses include type inferencing, liveness analysis, and the control flow analyses in O'Bagy's prototype compiler [.tr88-31.]. Determining possible execution paths through an Icon program is more complicated than it is for programs written in more conventional languages. The implementation of the type inferencing system and liveness analysis here explore variations on the techniques presented by O'Bagy.

The Organization of Part II

Part II is logically divided into three subparts. Chapters 14 through 16 present the main ideas upon which the compiler is based, Chapters 17 through 22 describe the implementation of these ideas, and Chapter 23 presents performance measurements of compiled code.

Chapter 14 describes the code generated by the compiler. It explains how Icon data values, variables, and goal-directed evaluation are implemented, independent of the actual translation process. Chapter 15 presents a theoretical model of the type inferencing system used in the compiler. The model includes the important ideas of the type

inferencing system, while ignoring some purely pragmatic details. Chapter 16 explains the liveness analysis problem and presents the solution used in the compiler.

The Icon compiler is designed to be a production-quality system. The compiler system consists of the compiler itself and a run-time system. The fact that these two components are not entirely independent must be carefully considered in the design of such a production-quality system. Chapter 17 describes the system as a whole and how the interactions between the components are handled.

Chapter 18 presents the organization of the compiler itself. This chapter describes some parts of the compiler in detail, but defers major topics to other chapters. Chapter 19 builds on the model presented in Chapter 15 and describes the full type inferencing system used in the compiler and its implementation. Chapter 20 describes the translation techniques used to produce code from expressions that employ Icon's goal-directed evaluation scheme and its unique control structures. It also describes the allocation of temporary variables using the information produced by liveness analysis.

The code generator does no look-ahead and as a result it often produces code that is poor when taken in context of subsequent code. This problem is shared with most code generators as are some of the solutions used in this compiler. The unique code generation techniques required by Icon's goal-directed evaluation produce unusual variations of this problem and require some innovative solutions in addition to the standard ones. Chapter 21 describes the various techniques employed to handle this problem. Chapter 22 describes the optimizations that can be done using the results of type inferencing. These optimizations also make use of liveness information.

Chapter 23 demonstrates the effects of the various optimizations used in the compiler on the performance of specific kinds of expressions. It also presents measurements of the performance of compiled code for a variety of complete programs, comparing the performance to that of the Icon interpreter. In addition, the sizes of the executable code for the complete programs are presented. The conclusions, Chapter 24, summarize what has been done and lists some work that remains to be explored. Chapter 25 describes one successful project to improve the compiler and make it usable on larger programs.

Chapter 14: The Translation Model

Modern compilers seldom produce machine code directly. They translate a program into a form closer to machine code than the source language and depend on other tools to finish the translation. If the compiler produces an object module, it depends on a linker and a loader to produce executable code. If the compiler produces assembly language, it also depends on an assembler. A recent trend among compilers produced in research environments has been to produce C code [.cbook,ansi-c.], adding a C compiler to the list of tools required to finish the translation to machine code [.SR, Ramakrishnan, Bartlett 89, Yuasa,Stroustrup,yacc,lex.]. The Icon compiler takes this approach and generates C code.

There are several advantages to compiling a language into C. Low-level problems such as register allocation and the selection and optimization of machine instructions are handled by the C compiler. As long as these problems are outside the scope of the research addressed by the compiler, it is both reasonable and effective to allow another compiler to deal with them. In general, it is easier to generate code in a higher-level language, just as it is easier to program in a higher-level language. As long as the target language lies on a ``nearly direct path'' from the source language to machine code, this works well. C is closely matched to most modern machine architectures, so few tangential translations must be done in generating C code from Icon.

Another advantage of generating C code is that it greatly increases the portability of the compiler and facilitates cross-compilation. The popularity of C in recent years has resulted in production-quality C compilers for most systems. While the implementation of Icon in C contains some machine and system dependencies, C's conditional compilation, macro, and file inclusion facilities make these dependencies relatively easy to deal with when they arise. These facts make possible the development of a highly portable Icon compiler, allowing the compiler's effectiveness to be tested by Icon's large user community.

14.1 Data Representation

Because the target language is C, Icon data must be represented as C data. The careful representation of data and variables is important to the performance of an implementation of a high-level language such as Icon. In addition, information provided by type inferencing can be used to optimize these representations. However, such considerations are largely outside the scope of this current research. For this reason, the representations used in code produced by this compiler and the compiler's run-time system are largely unchanged from those of the Icon interpreter system described in Part I. The interpreter's run-time system is written in C. Therefore borrowing its data representations for the compiler system is simple. This choice of representation means that the run-time system for the compiler could be adapted directly from the run-time system for the interpreter, and it allowed the compiler development to concentrate on parts of the system addressed by this research. In addition, this choice of representation allows a meaningful comparison of the performance of compiled code to the performance of interpreted code.

An Icon value is represented by a two-word descriptor (see Section 4.1). The first word, the *d-word*, contains type information. In the case of a string value, the type is indicated by zero in a high-order bit in the d-word, and the length of a string is stored in low-order

bits of the d-word. All other types have a one in that bit and further type information elsewhere in the d-word. The *v-word* of a descriptor indicates the value. The v-word of the null value is zero, the v-word of an Icon integer is the corresponding C integer value, and v-words of other types are pointers to data. A descriptor is implemented with the following C structure:

```
struct descrip {
   word dword; /* type field */
   union {
      word integr; /* integer value */
      char sptr; /* pointer to character string */
      union block bptr; /* pointer to a block */
      dptr descptr; /* pointer to a descriptor */
      } vword;
   };
```

word is defined to be a C integer type (one that is at least 32-bits long), block is a union of structures implementing various data types, and dptr is a pointer to a descrip structure.

14.2 Intermediate Results

While the representation of data in the compiler is the same as in the interpreter, the method of storing the intermediate results of expression evaluation is not. Two basic approaches have been used in language implementations to store intermediate results. A stack-based approach is simple and dynamic. It requires no pre-analysis of expressions to allocate storage for the intermediate results, but the simple rigid protocol allows little room for optimization. For Icon there is an additional problem with a stack-based approach. Goal-directed evaluation extends the lifetime of some intermediate results, requiring that the top elements of the evaluation stack be copied at critical points in execution [see Part I, or UA tr88-31]. In spite of the need for this extra copying, most previous implementations of Icon have been implemented with an evaluation stack.

An alternative to using a stack is to pre-allocate a temporary variable for each intermediate result. In this model, operations take explicit locations as arguments. Therefore an operation can directly access program variables as arguments; there is no need to perform the extra operations of pushing addresses or values on a stack. In addition, the lifetime of a temporary variable is not determined by a rigid protocol. The compiler can assign an intermediate result to a temporary variable over an arbitrary portion of the program, eliminating the copying needed to preserve a value beyond the lifetime imposed by a stack-based approach. This compiler uses the temporary-variable model because it allows more opportunities to optimize parameter handling, a major goal of this research.

Icon's automatic storage management dictates the use of a garbage collector in the run-time system. When this garbage collector is invoked, it must be able to locate all values that may be used later in the program. In the interpreter system, intermediate values and local variables are stored on the same stack. The garbage collector sweeps this stack to locate values. In the compiler, a different approach is taken to insure that all necessary values are locatable. Arrays of descriptors are allocated contiguously along with a count of the number of descriptors in the array. The arrays are chained together. An array of descriptors may be local to a C function, or it may be allocated with the malloc library function. The garbage collector locates values by following the chain and scanning the descriptors in each array. These descriptors are referred to as *tended* descriptors.

14.3 Executable Code

Even more important than where intermediate results are stored is how they are computed. Some aspects of Icon expression evaluation are similar to those of many other languages, but others aspects are not. Goal-directed evaluation with backtracking poses a particular challenge when implementing Icon expression evaluation. The Icon interpreter is based on a virtual machine that includes backtracking, as are Prolog interpreters based on the Warren Abstract Machine [.wam.]. While details differ between the Icon and Prolog virtual machines, their implementation of control backtracking is based on the same abstract data structures and state variables. Such a virtual machine contains a stack of procedure frames, but the stack is maintained differently from that of a virtual machine that does not implement goal-directed evaluation.

The difference manifests itself when a procedure produces a result, but has alternate results that it can produce in the event of backtracking. When this occurs, the frame for the procedure remains on the stack after control returns to the caller of the procedure. This frame contains the information needed to produce the alternate results. The left stack in the following diagram shows that procedure f has called procedure g. The arrows on the left of the stack represent the *backtracking chain* of procedures that can produce alternate results. btp points to the head of the backtracking chain which currently starts further down in the stack. The arrows on the right represent the call chain of procedures. fp points to the frame of the currently executing procedure.

Suppose g produces the first of several possible results. Execution returns to f and g's frame is added to the backtracking chain. This is represented by the middle stack in the diagram. If f then calls h, its procedure frame is added to the top of the stack as shown in the right stack in the diagram.

If h produces a result and is not capable of producing more, execution returns to f and the stack again looks like the one in the middle of the diagram (the program pointer within f is different, of course). If h produces a result and is capable of producing more, execution returns to f, but h's frame remains on the stack and is added to the head backtracking chain, similar to what was done when g produced a result. If h produces no results, backtracking occurs. h's frame is removed from the stack, execution returns to the procedure g who's frame is at the head of the backtracking chain, and g's frame is removed from the head of the chain. The stack once again looks like left stack in the diagram and g proceeds to produce another result.

Traditional languages such as Pascal or C present high-level virtual machines that contain no notion of backtracking and have no need to perform low-level stack manipulations. Icon expressions with goal-directed evaluation cannot be translated directly into such languages. This is the fundamental problem that must be addressed when designing a compiler for Icon. O'Bagy presents an elegant solution to this problem in her dissertation

[.tr88-31.]. Her solution is used by this optimizing compiler as a basis for translating Icon expressions into C code. The rest of this section contains a brief explanation of the variation of her approach that is used in the compiler, while exploring useful ways of viewing the problem. O'Bagy's dissertation describes how control structures not covered in this discussion can be implemented using her model.

Formal semantics is one tool that can be used in understanding a language [.gordon denote,stoy.]. The added complexity caused by Icon's goal-directed evaluation is reflected in Gudeman's description of Icon using denotational semantics [.gudeman denotational.]. While conventional programming languages can be described using one continuation for each expression, Icon requires two continuations. One continuation for an expression embodies the rest of the program if the expression succeeds, while the other embodies the rest of the program if the expression fails.

The Icon compiler uses the notion of success continuations to implement goal-directed evaluation. However, these continuations violate some of the properties traditionally associated with continuations. A continuation in denotational semantics and in the language Scheme [.Abelson,[Rees 86].] is a function that never returns. However, the success continuations produced by the compiler implement backtracking by returning. In addition, these continuations implement the rest of the current bounded expression rather than the rest of the entire program. Note that unlike continuations in Scheme, these continuations are created at compile time, not at run time. Some Prolog compilers have been based on a similar continuation-passing technique [.Nilsson,Ramakrishnan.].

The C language is oriented toward an imperative style of programming. In order to produce efficient code, the Icon compiler should not generate an excessive number of function calls. Specifically, it should avoid creating continuations for every expression. A more operational view of Icon's semantics and of C's semantics can be useful in understanding how to accomplish this. An operation in Icon can succeed or fail. In the view of denotational semantics, the question of what will be done in each case must be answered, with the answers taking the form of functions. In an operational view, the questions can take the form of where to go in each case. The answers to these questions can be any type of transfer of control supported by the C language: execute the next sequential instruction, execute a function, return from a function, or go to a label.

Most operations in Icon are *monogenic*. That is, they produce exactly one result, like operations in conventional languages. For these operations, the compiler can generate code whose execution simply falls through into the code that implements the subsequent operation.

Conditional operations are more interesting. These operations either produce a single value or fail. If such an operation succeeds, execution can fall through into code implementing the subsequent operation. However, if the operation fails, execution must transfer elsewhere in the program. This is accomplished by branching to a *failure label*. If the code for the operation is put in-line, this is straightforward. However, if the operation (either a built-in operation or an Icon procedure) is implemented by a separate C function, the function must notify the caller whether it succeeded or failed and the caller must effect the appropriate transfer of control.

By convention, C functions produced by the compiler and those implementing the run-time routines each return a signal (this convention is violated in some special cases). A signal is an integer (and is unrelated to Unix signals). If one of these C functions needs to

return an Icon value, it does so through a pointer to a result location that is passed to it as an argument. Two standard signals are represented by the manifest constants A_Continue and A_Resume. A return (either an Icon return expression or the equivalent construct in a built-in operation) is implemented with code similar to

```
*result = operation result;
return A_Continue;
```

Failure is implemented with the code

```
return A_Resume;
```

The code implementing the call of an operation consists of both a C call and signal-handling code.

```
switch (operation(args, &result)) {
   case A_Continue: break;
   case A_Resume: goto failure label;
   }
```

This code clearly can be simplified. This form is general enough to handle the more complex signal handling that can arise during code generation. Simplifying signal handling code is described in Chapter 21.

Generators pose the real challenge in implementing Icon. A generator includes code that must be executed if subsequent failure occurs. In addition, a generator, in general, needs to retain state information between suspending and being resumed. As mentioned above, this is accomplished by calling a success continuation. The success continuation contains subsequent operations. If an operation in the continuation fails, an A_Resume signal is returned to the generator, which then executes the appropriate code. The generator retains state information in local variables. If the generator is implemented as a C function, a pointer to the continuation is passed to it. Therefore, a function implementing a generative operation need not know its success continuation until run time.

Consider the operation *i to j*. This operation can be implemented in Icon with a procedure like

```
procedure To(i, j)
   while i <= j do {
      suspend i
      i +:= 1
      }
    fail
end
```

It can be implemented by an analogous C function similar to the following (for simplicity, C ints are used here instead of Icon values).

```
int to(i, j, result, succ_cont)
int i, j;
int *result;
int (*succ_cont)();
   {
   int signal;

   while (i <= j) {
      *result = i;
      signal = (*succ_cont)();
      if (signal != A_Resume)
         return signal;
```

```
      ++i;
      }
   return A_Resume;
   }
```

There is no explicit failure label in this code, but it is possible to view the code as if an implicit failure label occurs before the ++i.

The Icon expression

```
every write(1 to 3)
```

can be compiled into the following code (for simplicity, the write function has been translated into printf and scoping issues for result have been ignored). Note that the every simply introduces failure.

```
switch (to(1, 3, &result, sc)) {
   /* standard signal-handling code */
   ...
   }
int sc() {
   printf("%d\n", result);
   return A_Resume;
   }
```

The final aspect of Icon expression evaluation that must be dealt with is that of bounded expressions. Once execution leaves a bounded expression, that expression cannot be resumed. At this point, the state of the computation with respect to backtracking looks as it did when execution entered the bounded expression. This means that, in generated code, where to go on failure (either by branching to an explicit failure label or by returning an A_Resume signal) must be the same. However, this failure action is only correct in the C function containing the start of the code for the bounded expression. If a function suspended by calling a success continuation, execution is no longer in that original C function. To accommodate this restoration of failure action, execution must return to that original function.

This is accomplished by setting up a *bounding label* in the original C function and allocating a signal that corresponds to the label. When the end of the bounded expression is reached, the signal for the bounding label is returned. When the signal reaches the function containing the label, it is converted into a goto. It can be determined statically which calls must convert which signals. Note that if the bounded expression ends in the original C function, the ``return signal'' is already in the context of the label. In this case, it is immediately transformed into a goto by the compiler, and there is no real signal handling.

Consider the Icon expression

```
move(1);
...
```

The move function suspends and the C function implementing it needs a success continuation. In this case, move is called in a bounded context, so the success continuation must return execution to the function that called move. The continuation makes use of the fact that, like the C function for to, the one for move only intercepts A_Resume signals and passes all other signals on to its caller.

This expression can be implemented with code similar to the following. There are two possible signals that might be returned. move itself might produce an A_Resume signal or

it might pass along the bounding signal from the success continuation. Note that for a compound expression, both the bounding label and the failure label are the same. In general, this is not true. In this context, the result of move(1) is discarded. The variable trashcan receives this value; it is never read.

```
      switch (move(1, &trashcan, sc)) {
         case 1:
            goto L1;
         case A_Resume:
            goto L1;
         }
   L1: /* bounding label & failure label */
      ...

   int sc() {
      return 1; /* bound signal */
      }
```

Calling Conventions

This discussion has touched on the subject of calling conventions for run-time routines. In Icon, it is, in general, impossible to know until run time what an invocation is invoking. This is handled in the compiler with a standard calling convention for the C functions implementing operations and procedures. This calling convention allows a C function to be called without knowing anything about the operation it implements.

A function conforming to the standard calling convention has four parameters. These parameters are, in order of appearance, the number of Icon arguments (a C int), a pointer to the beginning of an array of descriptors holding the Icon arguments, a pointer to the descriptor used as the Icon result location, and a success continuation to use for suspension. The function itself is responsible for any argument conversions including dereferencing, and for argument list adjustment. As explained above, the function returns an integer signal. The function is allowed to return the signals A_Resume, A_Continue, and any signals returned by the success continuation. It may ignore the success continuation if it does not suspend. The function may be passed a null continuation. This indicates that the function will not be resumed. In this case, suspend acts like a simple return, passing back the signal A_Continue (this is not shown in the examples). The outline of a standard-conforming function is

```
   int function-name(nargs, args, result, succ_cont)
   int nargs; dptr args; dptr result;
   continuation succ_cont;
      {
       ...
       }
```

continuation is defined to be a pointer to a function taking no arguments and returning an integer.

Later sections of this dissertation describe the code generation process in more detail and describe optimizations of various parts of the code including parameter passing, continuations, signal handling, and branching.

Chapter 15: The Type Inferencing Model

Three sections of this dissertation are devoted to type inferencing: two chapters and an appendix. This chapter develops a theoretical model of type inferencing for Icon. For simplicity, it ignores some features of the language. This chapter presents intuitive arguments for the correctness of the formal model. Chapter 19 describes the actual implementation of type inferencing in the Icon compiler. The implementation handles the full Icon language and, for pragmatic reasons, differs from the theoretical model in some details.

This chapter starts with the motivation for performing type inferencing. It then describes the concept of *abstract interpretation*. This concept is used as a tool in this chapter to develop a type inferencing system from Icon's semantics. This chapter gives an intuitive presentation of this development process before presenting the formal models of abstract semantics for Icon. The most abstract of the formal models is the type inferencing system.

15.1 Motivation

Variables in the Icon programming language are untyped. That is, a variable may take on values of different types as the execution of a program proceeds. In the following example, x contains a string after the read (if the read succeeds), but it is then assigned an integer or real, provided the string can be converted to a numeric type.

```
x := read()
if numeric(x) then x +:= 4
```

In general, it is impossible to know the type of an operator's operands at translation time, so some type checking must be done at run time. This type checking may result in type conversions, run-time errors, or the selection among polymorphous operations (for example, the selection of integer versus real addition). In the Icon interpreter system, all operators check all of their operands at run time. This incurs significant overhead.

Much of this run-time type checking is unnecessary. An examination of typical Icon programs reveals that the types of most variables remain consistent throughout execution (except for the initial null value) and that these types can often be determined by inspection. Consider

```
if x := read() then
   y := x || ";"
```

Clearly both operands of || are strings so no checking or conversion is needed.

The goal of a type inferencing system is to determine what types variables may take on during the execution of a program. It associates with each variable usage a set of the possible types of values that variable might have when execution reaches the usage. This set may be a conservative estimate (overestimate) of the actual set of possible types that a variable may take on because the actual set may not be computable, or because an analysis to compute the actual set may be too expensive. However, a good type inferencing system operating on realistic programs can determine the exact set of types for most operands and the majority of these sets in fact contain single types, which is the information needed to generate code without type checking. The Icon compiler has an effective type inferencing system based on data flow analysis techniques.

15.2 Abstract Interpretation

Data flow analysis can be viewed as a form of abstract interpretation [.absintrp.]. This can be particularly useful for understanding type inferencing. A ``concrete'' interpreter for a language implements the standard (operational) semantics of the language, producing a sequence of states, where a state consists of an execution point, bindings of program variables to values, and so forth. An abstract interpreter does not implement the semantics, but rather computes information related to the semantics. For example, an abstract interpretation may compute the sign of an arithmetic expression rather than its value. Often it computes a ``conservative'' estimate for the property of interest rather than computing exact information. Data flow analysis is simply a form of abstract interpretation that is guaranteed to terminate. This chapter presents a sequence of approximations to Icon semantics, culminating in one suitable for type inferencing.

Consider a simplified operational semantics for Icon, consisting only of program points (with the current execution point maintained in a program counter) and variable bindings (maintained in an environment). As an example of these semantics, consider the following program. Four program points are annotated with numbers using comments (there are numerous intermediate points not annotated).

```
procedure main()
local s, n

   # 1:
   s := read()
   # 2:
   every n := 1 to 2 do {
      # 3:
      write(s[n])
      }
   # 4:
end
```

If the program is executed with an input of abc, the following states are included in the execution sequence (only the annotated points are listed). States are expressed in the form *program point*: *environment*.

```
1: [s = null, n = null]
2: [s = "abc", n = null]
3: [s = "abc", n = 1]
3: [s = "abc", n = 2]
4: [s = "abc", n = 2]
```

It is customary to use the *collecting semantics* of a language as the first abstraction (approximation) to the standard semantics of the language. The collecting semantics of a program is defined in Cousot and Cousot [.absintrp.] (they use the term *static semantics*) to be an association between program points and the sets of environments that can occur at those points during all possible executions of the program.

Once again, consider the previous example. In general, the input to the program is unknown, so the read function is assumed to be capable of producing any string. Representing this general case, the set of environments (once again showing only variable bindings) that can occur at point 3 is

```
[s = "", n = 1],
[s = "", n = 2],
[s = "a", n = 1],
```

```
[s = "a", n = 2],
  ...
[s = "abcd", n = 1],
[s = "abcd", n = 2],
  ...
```

A type inferencing abstraction further approximates this information, producing an association between each variable and a type at each program point. The actual type system chosen for this abstraction must be based on the language and the use to which the information is put. The type system used here is based on Icon's run-time type system. For structure types, the system used retains more information than a simple use of Icon's type system would retain; this is explained in detail later. For atomic types, Icon's type system is used as is. For point 3 in the preceding example the associations between variables and types are

```
[s = string, n = integer]
```

The type inferencing system presented in this chapter is best understood as the culmination of a sequence of abstractions to the semantics of Icon, where each abstraction discards certain information. For example, the collecting semantics discards sequencing information among states; in the preceding program, collecting semantics determine that, at point 3, states may occur with n equal to 1 and with n equal to 2, but does not determine the order in which they must occur. This sequencing information is discarded because desired type information is a static property of the program.

The first abstraction beyond the collecting semantics discards dynamic control flow information for goal directed evaluation. The second abstraction collects, for each variable, the value associated with the variable in each environment. It discards information such as, ``x has the value 3 when y has the value 7", replacing it with ``x may have the value 3 sometime and y may have the value 7 sometime.". It effectively decouples associations between variables.

This second abstraction associates a set of values with a variable, but this set may be any of an infinite number of sets and it may contain an infinite number of values. In general, this precludes either a finite computation of the sets or a finite representation of them. The third abstraction defines a type system that has a finite representation. This abstraction discards information by increasing the set associated with a variable (that is, making the set less precise) until it matches a type. This third model can be implemented with standard iterative data flow analysis techniques.

This chapter assumes that an Icon program consists of a single procedure and that all invocations are to built-in functions. It also assumes that there are no co-expressions beyond the main co-expression. See Chapter 19 for information on how to extend the abstractions to multiple procedures and multiple co-expressions.

15.3 Collecting Semantics

The collecting semantics of an Icon program is defined in terms of a *flow graph* of the program. A flow graph is a directed graph used to represent the flow of control in a program. Nodes in the graph represent the executable primitives in the program. An edge exists from node **A** to node **B** if it is possible for execution to pass directly from the primitive represented by node **A** to the primitive represented by node **B**. Cousot and Cousot [.absintrp.] prove that the collecting semantics of a program can be represented as

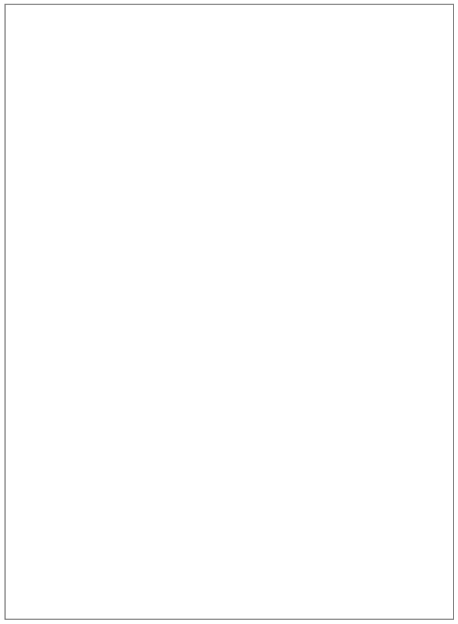
the least fixed point of a set of equations defined over the edges of the program's flow graph. These equations operate on sets of environments.

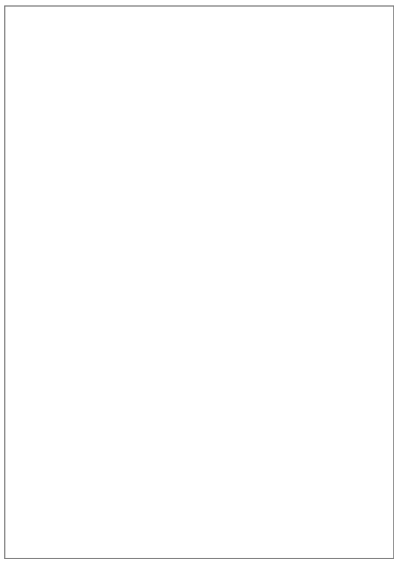
For an example of a flow graph, consider the Icon program

```
procedure main()
   every write(1 to 3)
end
```

The diagram below on the left shows the abstract syntax tree for this procedure, including the implicit fail at the end of the procedure. The invoke node in the syntax tree represents procedure invocation. Its first argument must evaluate to the procedure to be invoked; in this case the first argument is the global variable write. The rest of the arguments are used as the arguments to the procedure. pfail represents procedure failure (as opposed to expression failure within a procedure). Nodes corresponding to operations that produce values are numbered for purposes explained below.

A flow graph can be derived from the syntax tree. This is shown on the right.





The node labeled procedure main is the *start node* for the procedure; it performs any necessary initializations to establish the execution environment for the procedure. The edge from `invoke` to `to` is a resumption path induced by the control structure `every`. The path from `to` to `pfail` is the failure path for `to`. It is a forward execution path rather than a resumption path because the compound expression (indicated by ;) limits backtracking out of its left-hand sub-expression. Chapter 7 describes how to determine the edges of the flow graph for an Icon program.

Both the standard semantics and the abstract semantics must deal with the intermediate results of expression evaluation. A temporary-variable model is used because it is more convenient for this analysis than a stack model. This decision is unrelated to the use of a temporary-variable model in the compiler. This analysis uses a trivial assignment of temporary variables to intermediate results. Temporary variables are not reused. Each node that produces a result is assigned some temporary variable *ri* in the environment.

Assuming that temporary variables are assigned to the example according to the node numbering, the to operation has the effect of

```
r3 := r4 to r5
```

Expressions that represent alternate computations must be assigned the same temporary variable, as in the following example for the subexpression x := ("a" | "b"). The syntax tree below on the left and the and the flow graph are shown on the right.

The if and case control structures are handled similarly. In addition to temporary variables for intermediate results, some generators may need additional temporary variables to hold internal states during suspension. It is easy to devise a scheme to allocate them where they are needed; details are not presented here. The syntax tree is kept during abstract interpretation and used to determine the temporary variables associated with an operation and its operands.

The equations that determine the collecting semantics of the program are derived directly from the standard semantics of the language. The set of environments on an edge of the flow graph is related to the sets of environments on edges coming into the node at the head of this edge. This relationship is derived by applying the meaning of the node (in the standard semantics) to each of the incoming environments.

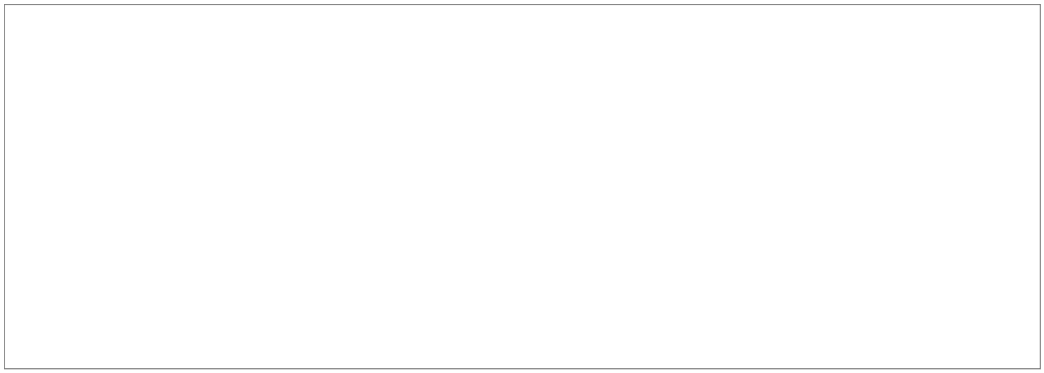
It requires a rather complex environment to capture the full operational semantics (and collecting semantics) of a language like Icon. For example, the environment needs to include a representation of the external file system. However, later abstractions only use the fact that the function read produces strings. This discussion assumes that it is possible to represent the file system in the environment, but does not give a representation. Other complexities of the environment are discussed later. For the moment, examples only show the bindings of variables to unstructured (atomic) values.

As an example of environments associated with the edges of a flow graph, consider the assignment at the end of the following code fragment. The comments in the if expression are assertions that are assumed to hold at those points in the example.

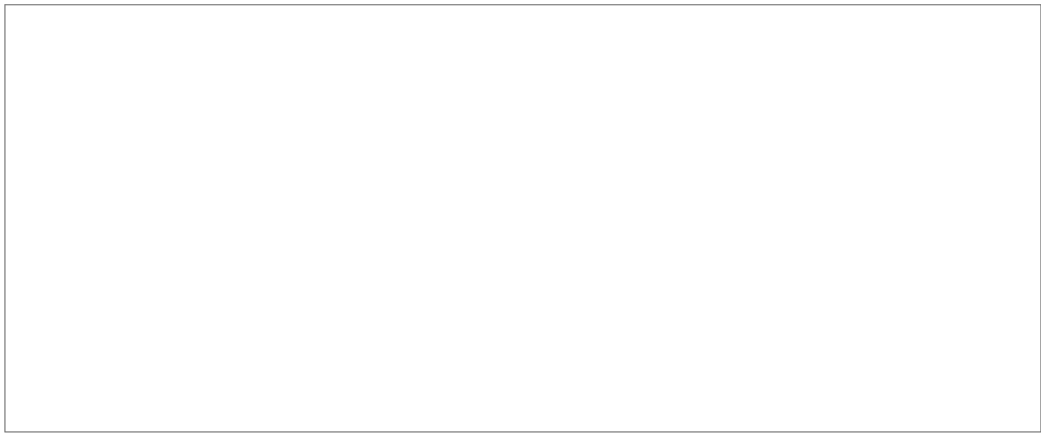
```
if x = 7 then {
   ...
   # x is 7 and y is 3
   }
```

```
else {
   ...
   # (x is null and y is 1) or (x is "abc" and y is 2)
   }
x := y + 2
```

Because of the preceding if expression, there are two paths reaching the assignment. The diagram below shows the flow graph and accompanying environments for the expression; the diagram ignores the fact that the assignment expression requires several primitive operations to implement.



For a conditional expression, an incoming environment is propagated to the path that it would cause execution to take in the standard semantics. This requires distinguishing the paths to be taken on failure (backtracking paths) from those to be taken on success. The following diagram shows an example of this.



In general there may be several possible backtracking paths. The environments in the standard and collecting semantics need to include a stack of current backtracking points and control flow information, and the flow graph needs instructions to maintain this stack. The Icon interpreter system described in Part I is an example of how this information can be maintained. However, the first abstraction to the collecting semantics eliminates the need for this information, so the information is not presented in detail here.

15.4 Model 1: Eliminating Control Flow Information

The first abstraction involves taking the union of the environments propagated along all the failure paths from a node in the collecting semantics and propagating that union along each of the failure paths in the new abstraction. This abstraction eliminates the stack of backtracking points from the environment.

A more formal definition for this model requires taking a closer look at Icon data values, especially those values with internal structure. In order to handle Icon data objects with pointer semantics, an environment needs more than variable bindings. This fact is important to type inferencing. The problem is handled by including two components in the environment. The first is the *store*, which maps variables to values. Variables include *named* variables, *temporary* variables, and *structure* variables. Named variables correspond to program identifiers. Temporary variables hold intermediate results as discussed above. Structure variables are elements of structures such as lists. Note that the sets of named variables and temporary variables are each finite (based on the assumption that a program consists of a single non-recursive procedure; as mentioned earlier, this assumption is removed in Chapter 19), but for some non-terminating programs, the set of structure variables may be infinite. *Program* variables include both named variables and structure variables but not temporary variables.

Values include atomic data values such as integers, csets, and strings. They also include *pointers* that reference objects with pointer semantics. In addition to the values just described, temporary variables may contain references to program variables. These *variable references* may be used by assignments to update the store or they may be dereferenced by other operations to obtain the values stored in the variables.

The second part of the environment is the *heap*. It maps pointers to the corresponding data objects (this differs from the heap in the Icon implementation in that that heap also contains some data objects that do not have pointer semantics). For simplicity, the only data type with pointer semantics included in this discussion is the list. A list is a partial mapping from integers to variables. Representing other data types with pointer semantics is straightforward; this is discussed in Chapter 19.

The first abstraction is called Model 1. The notations $\text{envir}_{[n]}$, $\text{store}_{[n]}$, and $\text{heap}_{[n]}$ refer to the sets of possible environments, stores, and heaps respectively in model n. For example, $\text{envir}_{[1]}$ is the set of possible environments in the first abstraction. In the following set of definitions, $X \times Y$ is the set of ordered pairs where the first value in the pair is from X and the second value is from Y. $X \rightarrow Y$ is the set of partial functions from X to Y. The definition of the set possible environments for model 1 is

$\text{envir}_{[1]} = \text{store}_{[1]} \times \text{heap}_{[1]}$
$\text{store}_{[1]} = \text{variables} \rightarrow \text{values}$
$\text{values} = \text{integers} \cup \text{strings} \cup ... \cup \text{pointers} \cup \text{variables}$
$\text{heap}_{[1]} = \text{pointers} \rightarrow \text{lists, where lists} = \text{integers} \rightarrow \text{variables}$

For example, the expression

```
a := ["abc"]
```

creates a list of one element whose value is the string abc and assigns the list to the variable a. Let p_1 be the pointer to the list and let v_1 be the (anonymous) variable within the list. The resulting environment, e $\in$ $\text{envir}_{[1]}$, might be

$e = (s,h)$, where $s \in \text{store}_{[1]}$, $h \in \text{heap}_{[1]}$
$s(a) = p_1$
$s(v_1) =$ "abc"
$h(p_1) = L_1$, where $L_1 \in$ lists
$L_1(1) = v_1$

If the statement

```
a[1] := "xyz"
```

is executed, the subscripting operation dereferences a producing p_1, then uses the heap to find L_1, which it applies to 1 to produce the result v_1. The only change in the environment at this point is to temporary variables that are not shown. The assignment then updates the store, producing

$e_1 = (s_1, h)$
$s_1(a) = p_1$
$s_1(v_1) =$ "xyz"

Assignment does not change the heap. On the other hand, the expression

```
put(a, "xyz")
```

adds the string xyz to the end of the list; if it is executed in the environment e, it alters the heap along with adding a new variable to the store.

$e_1 = (s_1, h_1)$
$s_1(a) = p_1$
$s_1(v_1) =$ "abc"
$s_1(v_2) =$ "xyz"
$h_1(p_1) = L_2$
$L_2(1) = v_1$
$L_2(2) = v_2$

If a formal model were developed for the collecting semantics, it would have an environment similar to the one in Model 1. However, it would need a third component with which to represent the backtracking stack.

15.5 Model 2: Decoupling Variables

The next approximation to Icon semantics, Model 2, takes all the values that a variable might have at a given program point and gathers them together. In general, a variable may have the same value in many environments, so this, in some sense, reduces the amount of space required to store the information (though the space may still be unbounded). The "cost" of this reduction of storage is that any information about relationship of values between variables is lost.

Model 2 is also defined in terms of environments, stores, and heaps, although they are different from those of Model 1. A store in Model 2 maps sets of variables to sets of values; each resulting set contains the values associated with the corresponding variables in environments in Model 1. Similarly, a heap in Model 2 maps sets of pointers to sets of lists; each of these sets contains the lists associated with the corresponding pointers in environments in Model 1. An environment in Model 2 contains a store and a heap, but unlike in Model 1, there is only one of these environments associated with each program point. The environment is constructed so that it effectively "contains" the environments in the set associated with the point in Model 1.

The definition of Model 2 is

$envir_{[2]} = store_{[2]} \times heap_{[2]}$
$store_{[2]} = 2^{variables} \rightarrow 2^{values}$
$heap_{[2]} = 2^{pointers} \rightarrow 2^{lists}$

In Model 1, operations produce elements from the set *values*. In Model 2, operations produce subsets of this set. It is in this model that read is taken to produce the set of all strings and that the existence of an external file system can be ignored.

Suppose a program point is annotated with the set containing the following two environments from Model 1.

$e_1, e_2 \in envir_{[1]}$
$e_1 = (s_1, h_1)$
$s_1(x) = 1$
$s_1(y) = p_1$
$h_1(p_1) = L_1$
$e_2 = (s_2, h_2)$
$s_2(x) = 2$
$s_2(y) = p_1$
$h_2(p_1) = L_2$

Under Model 2 the program point is annotated with the single environment $\hat{e} \in envir_{[2]}$, where

$\hat{e} = (\hat{s}, \hat{h})$
$\hat{s}(\{x\}) = \{1,2\}$
$\hat{s}(\{y\}) = \{p_1\}$
$\hat{s}(\{x, y\}) = \{1, 2, p_1\}$
$\hat{h}(\{p_1\}) = \{L_1, L_2\}$

Note that a store in Model 2 is distributive over union. That is,

$\hat{s}(X \cup Y) = \hat{s}(X) \cup \hat{s}(Y)$

so listing the result of $\hat{s}(\{x, y\})$ is redundant. A heap in Model 2 also is distributive over union.

In going to Model 2 information is lost. In the last example, the fact that x = 1 is paired with $p_1 = L_1$ and x = 2 is paired with $p_1 = L_2$ is not represented in Model 2.

Just as read is extended to produce a set of values, so are all other operations. These "extended" operations are then used to set up the equations whose solution formally defines Model 2. This extension is straightforward. For example, the result of applying a unary operator to a set is the set obtained by applying the operator to each of the elements in the operand. The result of applying a binary operator to two sets is the set obtained by applying the operator to all pairs of elements from the two operands. Operations with more operands are treated similarly. For example

$$\begin{aligned}\{1, 3, 5\} + \{2, 4\} &= \{1 + 2, 1 + 4, 3 + 2, 3 + 4, 5 + 2, 5 + 4\} \\ &= \{3, 5, 5, 7, 7, 9\} \\ &= \{3, 5, 7, 9\}\end{aligned}$$

The loss of information mentioned above affects the calculation of environments in Model 2. Suppose the addition in the last example is from

```
z := x + y
```

and that Model 1 has the following three environments at the point before the calculation

```
[x = 1, y = 2, z = 0]
[x = 3, y = 2, z = 0]
[x = 5, y = 4, z = 0]
```

After the calculation the three environments will be

```
[x = 1, y = 2, z = 3]
[x = 3, y = 2, z = 5]
[x = 5, y = 4, z = 9]
```

If these latter three environments are translated into an environment of Model 2, the result is

```
[x = {1, 3, 5}, y = {2, 4}, z = {3, 5, 9}]
```

However, when doing the computation using the semantics of + in Model 2, the value for z is {3, 5, 7, 9}. The solution to the equations in Model 2 overestimates (that is, gives a conservative estimate for) the values obtained by computing a solution using Model 1 and translating it into the domain of Model 2.

Consider the following code with respect to the semantics of assignment in Model 2. (Assume that the code is executed once, so only one list is created.)

```
x := [10, 20]
i := if read() then 1 else 2
x[i] := 30
```

After the first two assignments, the store maps x to a set containing one pointer and maps i to a set containing 1 and 2. The third assignment is not as straightforward. Its left operand evaluates to two variables; the most that can be said about one of these variables after the assignment is that it might have been assigned 30. If (s, h) is the environment after the third assignment then

$s(\{x\}) = \{ p_1 \}$
$s(\{i\}) = \{1, 2\}$
$s(\{v_1\}) = \{10, 30\}$
$s(\{v_2\}) = \{20, 30\}$

$h(\{p_1\}) = \{L_1\}$

$L_1(1) = v_1$
$L_1(2) = v_2$

Clearly all assignments could be treated as *weak updates* [.pntstr.], where a weak update is an update that may or may not take place. However, this would involve discarding too much information; assignments would only add to the values associated with variables and not replace the values. Therefore assignments where the left hand side evaluates to a set containing a single variable are treated as special cases. These are implemented as *strong updates*.

15.6 Model 3: A Finite Type System

The environments in Model 2 can contain infinite amounts of information, as in the program

```
x := 1
repeat x +:= 1
```

where the set of values associated with x in the loop consists of all the counting numbers. Because equations in Model 2 can involve arbitrary arithmetic, no algorithm can find the least fixed point of an arbitrary set of these equations.

The final step is to impose a finitely representable type system on values. A type is a (possibly infinite) set of values. The type system presented here includes three classifications of basic types. The first classification consists of the Icon types without pointer semantics: integers, strings, csets, etc. The second classification groups pointers together according to the lexical point of their creation. This is similar to the method used to handle recursive data structures in Jones and Muchnick [.analrcsv.]. Consider the code

```
every insert(x, [1 to 5])
```

If this code is executed once, five lists are created, but they are all created at the same point in the program, so they all belong to the same type. The intuition behind this choice of types is that structures created at the same point in a program are likely to have components of the same type, while structures created at different points in a program may have components of different types.

The third classification of basic types handles variable references. Each named variable and temporary variable is given a type to itself. Therefore, if a is a named variable, {a} is a type. Structure variables are grouped into types according to the program point where the pointer to the structure is created. This is not necessarily the point where the variable is created; in the following code, a pointer to a list is created at one program point, but variables are added to the list at different points

```
x := []
push(x, 1)
push(x ,2)
```

References to these variables are grouped into a type associated with the program point for [], not the point for the corresponding push.

If a program contains k non-structure variables and there are n locations where pointers can be created, then the basic types for the program are integer, string, ..., P_1, ..., P_n, V_1, ..., V_n, $\{v_1\}$, ..., $\{v_k\}$ where P_i is the pointer type created at location i, V_i is the variable type associated with P_i, and v_i is a named variable or a temporary variable. Because programs are lexically finite they each have a finite number of basic types. The set of all types for a program is the smallest set that is closed under union and contains the empty set along with the basic types:

types = {{}, integers, strings,..., (integers ∪ strings),..., (integers ∪ strings ∪ ... ∪ $\{v_k\}$)}

Model 3 replaces the arbitrary sets of values of Model 2 by types. This replacement reduces the precision of the information, but allows for a finite representation and allows the information to be computed in finite time.

In Model 3, both the store and the heap map types to types. This store is referred to as the *type store*. The domain of type store is *variable types*, that is, those types whose only values are variable references. Similarly, the domain of the heap is *pointer types*. Its range is the set types containing only structure variables. A set of values from Model 2 is converted to a type in Model 3 by mapping that set to the smallest type containing it. For example, the set

```
{1, 4, 5, "23", "0"}
```

is mapped to

```
integer ∪ string
```

The definition of $envir_{[3]}$ is

```
envir[3] = store[3] × heap[3]
store[3] = variable-types → types
heap[3] = pointer-types → structure-variable-types
types ⊆ 2^values
variable-types ⊆ types
structure-variable-types ⊆ variable-types
pointer-types ⊆ types
```

There is exactly one variable type for each pointer type in this model. The heap simply consists of this one-to-one mapping; the heap is of the form

$h(P_i) = V_i$

This mapping is invariant over a given program. Therefore, the type equations for a program can be defined over $store_{[3]}$ rather than $envir_{[3]}$ with the heap embedded within the type equations.

Suppose an environment from Model 2 is

$e \in envir_{[2]}$
$e = (s, h)$

$s(\{a\}) = \{ p_1 , p_2\}$
$s(\{v_1\}) = \{1, 2\}$
$s(\{v_2\}) = \{1\}$
$s(\{v_3\}) = \{12.03\}$

$h(\{p_1\}) = \{L_1, L_2\}$
$h(\{p_2\}) = \{L_3\}$

$L_1(1) = v_1$

$L_2(1) = v_1$
$L_2(2) = v_2$

$L_3(1) = v_3$

Suppose the pointers p_1 and p_2 are both created at program point 1. Then the associated pointer type is P_1 and the associated variable type is V_1. The corresponding environment in Model 3 is

$\hat{e} \in envir_{[3]}$
$\hat{e} = (\hat{s},\hat{h})$

$\hat{s}(\{a\}) = P_1$
$\hat{s}(V_1) = integer \cup real$

$\hat{h}(P_1) = V_1$

The collecting semantics of a program establishes a set of (possibly) recursive equations between the sets of environments on the edges of the program's flow graph. The collecting semantics of the program is the least fixed point of these equations in which the set on the edge entering the start state contains all possible initial environments. Similarly, type inferencing establishes a set of recursive equations between the type stores on the edges of the flow graph. The least fixed point of these type inferencing equations is computable using iterative methods. This is discussed in Chapter 19. The fact that these equations have solutions is due to the fact that the equations in the collecting semantics have a solution and the fact the each abstraction maintains the “structure” of the problem, simply discarding some details.

Chapter 19 also extends type inferencing to handle the entire Icon language. Chapter 22 uses the information from type inferencing to optimize the generated code.

Chapter 16: Liveness Analysis of Intermediate Values

The maintenance of intermediate values during expression evaluation in the Icon programming language is more complicated than it is for conventional languages such as C and Pascal. O'Bagy explains this in her dissertation [.tr88-31.]:

> "Generators prolong the lifetime of temporary values. For example, in
>
> i = find(s1,s2)
>
> the operands of the comparison operation cannot be discarded when find produces its result. If find is resumed, the comparison is performed again with subsequent results from find(s1,s2), and the left operand must still be available."

In some implementation models, it is equally important that the operands of find still be available if that function is resumed (this depends on whether the operand locations are used during resumption or whether all needed values are saved in the local state of the function).

As noted in Chapter 14, a stack-based model handles the lifetime problem dynamically. However, a temporary-variable model like the one used in this compiler requires knowledge at compile-time of the lifetime of intermediate values. In a straightforward implementation of conventional languages, liveness analysis of intermediate values is trivial: an intermediate value is computed in one place in the generated code, is used in one place, and is live in the contiguous region between the computation and the use. In such languages, determining the lifetime of intermediate values only becomes complicated when certain optimizations are performed, such as code motion and common subexpression elimination across basic blocks [.dragonbk,progflw.]. This is not true in Icon. In the presence of goal-directed evaluation, the lifetime of an intermediate value can extend beyond the point of use. Even in a straightforward implementation, liveness analysis is not trivial.

In its most general form, needed in the presence of the optimizations mentioned above, liveness analysis requires iterative methods. However, goal-directed evaluation imposes enough structure on the liveness problem that, at least in the absence of optimizations, iterative methods are not needed to solve it. This chapter presents a simple and accurate method for computing liveness information for intermediate values in Icon. The analysis is formalized in an attribute grammar.

16.1 Implicit Loops

Goal-directed evaluation extends the lifetime of intermediate values by creating implicit loops within an expression. In O'Bagy's example, the start of the loop is the generator find and the end of the loop is the comparison that may fail. An intermediate value may be used within such a loop, but if its value is computed before the loop is entered, it is not recomputed on each iteration and the temporary variable must not be reused until the loop is exited.

The following fragment of C code contains a loop and is therefore analogous to code generated for goal-directed evaluation. It is used to demonstrate the liveness information

needed by a temporary variable allocator. In the example, *v1* through *v4* represent intermediate values that must be assigned to program variables.

```
v1 = f1();
while (--v1) {
   v2 = f2();
   v3 = v1 + v2;
   f3(v3);
   }
v4 = 8;
```

Separate variables must be allocated for v1 and *v2* because they are both needed for the addition. Here, x is chosen for *v1* and y is chosen for *v2*.

```
x = f1();
while (--x) {
   y = f2();
   v3 = x + y;
   f3(v3);
   }
v4 = 8;
```

x cannot be used to hold *v3*, because x is needed in subsequent iterations of the loop. Its lifetime must extend through the end of the loop. y, on the other hand, can be used because it is recomputed in subsequent iterations. Either variable may be used to hold *v4*.

```
x = f1();
while (--x) {
   y = f2();
   y = x + y;
   f3(y);
   }
x = 8;
```

Before temporary variables can be allocated, the extent of the loops created by goal-directed evaluation must be estimated. Suppose O'Bagy's example

```
i = find(s1, s2)
```

appears in the following context

```
procedure p(s1, s2, i)
   if i = find(s1, s2) then return i + *s1
   fail
end
```

The simplest and most pessimistic analysis assumes that a loop can appear anywhere within the procedure, requiring the conclusion that an intermediate value in the expression may live to the end of the procedure. Christopher's simple analysis [.tccompile.] notices that the expression appears within the control clause of an if expression. This is a bounded context; implicit loops cannot extend beyond the end of the control clause. His allocation scheme reuses, in subsequent expressions, temporary variables used in this control clause. However, it does not determine when temporary variables can be reused within the control clause itself.

The analysis presented here locates the operations within the expression that can fail and those that can generate results. It uses this information to accurately determine the loops within the expression and the intermediate values whose lifetimes are extended by those loops.

16.2 Liveness Analysis

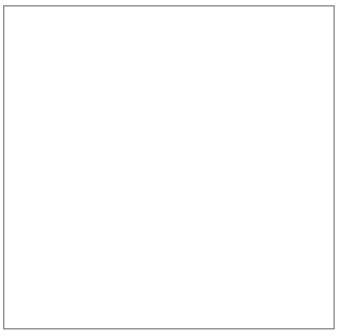
It is instructive to look at a specific example where intermediate values must be retained beyond (in a lexical sense) the point of their use. The following expression employs goal-directed evaluation to conditionally write sentences in the data structure x to an output file. Suppose f is either a file or null. If f is a file, the sentences are written to it; if f is null, the sentences are not written.

```
every write(\f, !x, ".")
```

In order to avoid the complications of control structures at this point in the discussion, the following equivalent expression is used in the analysis:

```
write(\f, !x, ".") & &fail
```

This expression can be converted into a sequence of primitive operations producing intermediate values (*v1*, *v2*, ...). This is shown in diagram. For convenience, the operations are expressed in Icon, except that the assignments do not dereference their right-hand operands.



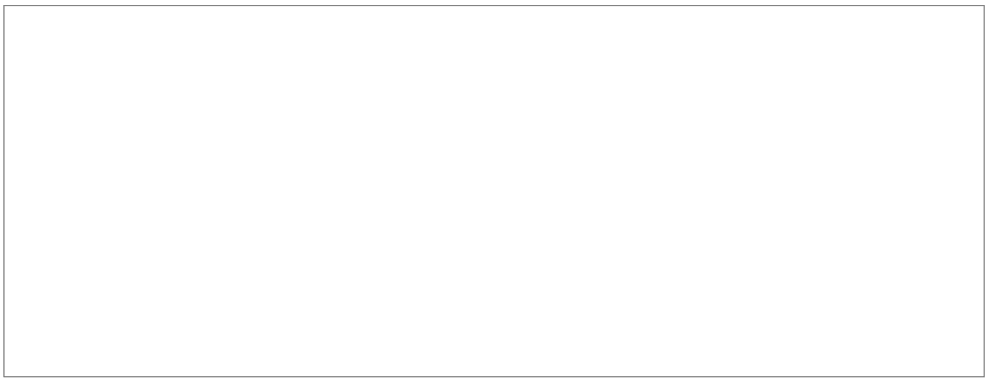
Whether or not the program variables and constants are actually placed in temporary variables depends on the machine model, implementation conventions, and what optimizations are performed. Clearly a temporary variable is not needed for &fail. However, temporary variables are needed if the subexpressions are more complex; intermediate values are shown for all subexpressions for explanatory purposes.

When &fail is executed, the ! operation is resumed. This creates an implicit loop from the ! to &fail, as shown by the arrow in the above diagram. The question is: What intermediate values must be retained up to &fail? A more instructive way to phrase the question is: After &fail is executed, what intermediate values could be reused without being recomputed? From the sequence of primitive operations, it is clear that the reused values include *v1* and *v3*, and, if the element generation operator, !, references its argument after resumption, then the reused values include *v4*. *v2* is not used within the loop, *v5* and *v6* are recomputed within the loop, and *v7* and *v8* are not used. The lines in the diagram to the left of the code indicate the lifetime of the intermediate values. The dotted portion of each line represents the region of the lifetime beyond what would exist in the absence of backtracking.

Liveness information could be computed by making the implicit loops explicit then performing a standard liveness analysis in the form of a global data flow analysis. That is unnecessarily expensive. There is enough structure in this particular liveness problem that it can be solved during the simple analysis required to locate the implicit loops caused by goal-directed evaluation.

Several concepts are needed to describe analyses involving execution order within Icon expressions. *Forward execution order* is the order in which operations would be executed at run time in the absence of goal-directed evaluation and explicit loops. Goal-directed evaluation involves both failure and the resumption of suspended generators. The control clause of an if-then-else expression may fail, but instead of resuming a suspending generator, it causes the else clause to be executed. This failure results in forward execution order. Forward execution order imposes a partial ordering on operations. It produces no ordering between the then and the else clauses of an if expression. *Backtracking order* is the reverse of forward execution order. This is due to the LIFO resumption of suspended generators. The backward flow of control caused by looping control structures does not contribute to this liveness analysis (intermediate results used within a looping control structure are also computed within the loop), but is dealt with in later chapters. The every control structure is generally viewed as a looping control structure. However, it simply introduces failure. Looping only occurs when it is used with a generative control clause, in which case the looping is treated the same as goal-directed evaluation.

A notation that emphasizes intermediate values, subexpressions, and execution order is helpful for understanding how liveness is computed. Both postfix notation and syntax trees are inadequate. A postfix notation is good for showing execution order, but tends to obscure subexpressions. The syntax tree of an expression shows subexpressions, but execution order must be expressed in terms of a tree walk. In both representations, intermediate values are implicit. For this discussion, an intermediate representation is used. A subexpression is represented as a list of explicit intermediate values followed by the operation that uses them, all enclosed in ovals. Below each intermediate value is the subexpression that computes it. This representation is referred to as a *postfix tree*. The postfix tree for the example above is:



In this notation, the forward execution order of operations (which includes constants and references to program variables) is left-to-right and the backtracking order is right-to-left. In this example, the backtracking order is &fail, invoke, ".", !, x, \, f, and write.

As explained above, the use of an intermediate value must appear in an implicit loop for the value to have an extended lifetime. Two events are needed to create such a loop. First, an operation must fail, initiating backtracking. Second, an operation must be resumed, causing execution to proceed forward again. This analysis computes the maximum lifetime of intermediate values in the expression, so it only needs to compute the rightmost operation (within a bounded expression) that can fail. This represents the end

of the farthest reaching loop. Once execution proceeds beyond this point, no intermediate value can be reused.

The intermediate values of a subexpression are used at the end of the subexpression. For example, invoke uses the intermediate values *v1*, *v3*, *v5*, and *v6*; the following figure shows these intermediate results and the operation in isolation.

In order for these uses to be in a loop, backtracking must be initiated from outside; that is, beyond the subexpression (in the example, only &fail and & are beyond the subexpression).

In addition, for an intermediate value to have an extended lifetime, the beginning of the loop must start after the intermediate value is computed. Two conditions may create the beginning of a loop. First, the operation itself may be resumed. In this case, execution continues forward within the operation. It may reuse any of its operands and none of them are recomputed. The operation does not have to actually generate more results. For example, reversible swap (the operator <->) can be resumed to reuse both of its operands, but it does not generate another result. Whether an operation actually reuses its operands on resumption depends on its implementation. In the Icon compiler, operations implemented with a C function using the standard calling conventions always use copies of operands on resumption, but implementations tailored to a particular use often reference operand locations on resumption. Liveness analysis is presented here as if all operations reuse their operands on resumption. In the actual implementation, liveness analysis computes a separate lifetime for values used internally by operations and the code generator decides whether this lifetime applies to operands. This internal lifetime may also be used when allocating tended descriptors for variables declared local to the in-line code for an operation. The behavior of the temporary-variable model presented in this dissertation can be compared with one developed by Nilsen and Martinek [.martinek.]; it also relies on the liveness analysis described in this chapter.

The second way to create the beginning of a loop is for a subexpression to generate results. Execution continues forward again and any intermediate values to the left of the generative subexpression may be reused without being recomputed. Remember, backtracking is initiated from outside the expression. Suppose an expression that can fail is associated with *v6*, in the previous figure. This creates a loop with the generator associated with *v5*. However, this particular loop does not include invoke and does not contribute to the reuse of *v1* or *v3*.

A resumable operation and generative subexpressions are all *resumption points* within an expression. A simple rule can be used to determine which intermediate values of an expression have extended lifetimes: If the expression can be resumed, the intermediate values with extended lifetimes consist of those to the left of the rightmost resumption point of the expression. This rule refers to the ``top level" intermediate values. The rule must be applied recursively to subexpressions to determine the lifetime of lower level intermediate values.

It sometimes may be necessary to make conservative estimates of what can fail and of resumption points (for liveness analysis, it is conservative to overestimate what can fail or be resumed). For example, invocation may or may not be resumable, depending on what

is being invoked and, in general, it cannot be known until run time what is being invoked (for the purposes of this example analysis, it is assumed that the variable write is not changed anywhere in the program).

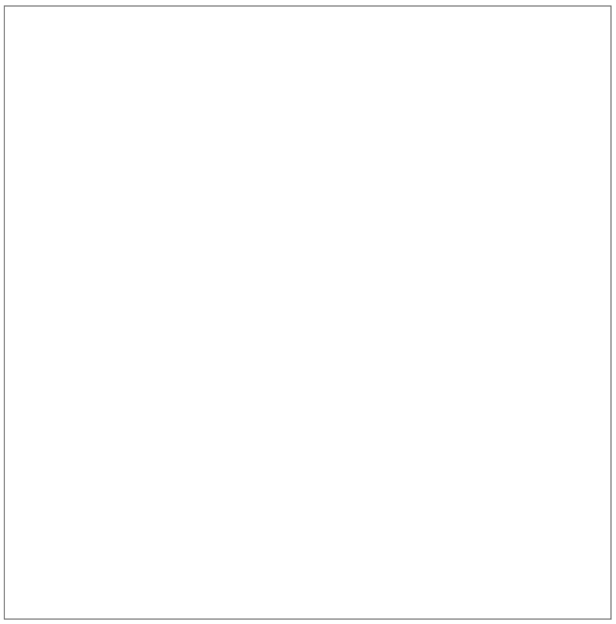
In the example, the rightmost operation that can fail is &fail. Resumption points are ! and the subexpressions corresponding to the intermediate values *v5* and *v7*.

Once the resumption points have been identified, the rule for determining extended lifetimes can be applied. If there are no resumption points in an expression, no intermediate values in that expression can be reused. Applying this rule to the postfix tree above yields *v1*, *v3*, and *v4* as the intermediate values that have extended lifetimes.

Similar techniques can be used for liveness analysis of Prolog programs, where goal-directed evaluation also creates implicit loops. One difference is that a Prolog clause is a linear sequence of calls. It does not need to be ``linearized'' by construction a postfix tree. Another difference is that all intermediate values in Prolog programs are stored in explicit variables. A Prolog variable has a lifetime that extends to the right of its last use if an implicit loops starts after the variable's first use and ends after the variable's last use.

16.3 An Attribute Grammar

To cast this approach as an attribute grammar, an expression should be thought of in terms of an abstract syntax tree. The transformation from a postfix tree to a syntax tree is trivial. It is accomplished by deleting the explicit intermediate values. A syntax tree for the example is:



Several interpretations can be given to a node in a syntax tree. A node can be viewed as representing either an operation, an entire subexpression, or an intermediate value.

This analysis associates four attributes with each node (this ignores attributes needed to handle break expressions). The goal of the analysis is to produce the lifetime attribute. The other three attributes are used to propagate information needed to compute the lifetime.

- resumer is either the rightmost operation (represented as a node) that can initiate backtracking into the subexpression or it is null if the subexpression cannot be resumed.
- failer is related to resumer. It is the rightmost operation that can initiate backtracking that can continue past the subexpression. It is the same as resumer, unless the subexpression itself contains the rightmost operation that can fail.
- gen is a boolean attribute. It is true if the subexpression can generate multiple results if resumed.
- lifetime is the operation beyond which the intermediate value is no longer needed. It is either the parent node, the resumer of the parent node, or null. The lifetime is the parent node if the value is never reused after execution leaves the parent operation. The lifetime is the resumer of the parent if the parent operation or a generative sibling to the right can be resumed. A lifetime of null is used to indicate that the intermediate value is never used. For example, the value of the control clause of an if expression is never used.

Attribute computations are associated with productions in the grammar. The attribute computations for failer and gen are always for the non-terminal on the left-hand side of the production. These values are then used at the parent production; they are effectively passed up the syntax tree. The computations for resumer and lifetime are always for the attributes of non-terminals on the right-hand side of the production. resumer is then used at the productions defining these non-terminals; it is effectively passed down the syntax tree. lifetime is usually saved just for the code generator, but it is sometimes used by child nodes.

16.4 Primary Expressions

Variables, literals, and keywords are primary expressions. They have no subexpressions, so their productions contain no computations for resumer or lifetime. The attribute computations for a literal follow. A literal itself cannot fail, so backtracking only passes beyond it if the backtracking was initiated before (to the right of) it. A literal cannot generate multiple results.

```
expr ::= literal {
   expr.failer := expr.resumer
   expr.gen := false
   }
```

Another example of a primary expression is the keyword &fail. Execution cannot continue past &fail, so it must be the rightmost operation within its bounded expression that can fail. A pre-existing attribute, node, is assumed to exist for every symbol in the grammar. It is the node in the syntax tree that corresponds to the symbol.

```
expr ::= &fail {
   expr.failer := expr.node
   expr.gen := false
   }
```

16.5 Operations with Subexpressions

Addition provides an example of the attribute computations involving subexpressions. The following diagram shows how resumer, failer, and gen information would be passed through the postfix tree.

This information would then be used to compute lifetime information for *v1* and *v2*. The next figure shows how the attribute information is actually passed through the syntax tree.

The lifetime attributes are computed for the roots of the subtrees for $expr_1$ and $expr_2$.

The details of the attribute computations depend, in part, on the characteristics of the individual operation. Addition does not fail, so the rightmost resumer, if there is one, of $expr_2$ is the rightmost resumer of the entire expression. The rightmost resumer of $expr_1$ is the rightmost operation that can initiate backtracking that continues past $expr_2$. Addition does not suspend, so the lifetime of the value produced by $expr_2$ only extends through the operation (that is, it always is recomputed in the presence of goal-directed evaluation). If $expr_2$ is a generator, then the result of $expr_1$ must be retained for as long as $expr_2$ might be resumed. Otherwise, it need only be retained until the addition is performed. $expr_1$ is the first thing executed in the expression, so its failer is the failer for the entire expression. The expression is a generator if either $expr_1$ or $expr_2$ is a generator (note that the operation | is logical *or*, not Icon's alternation control structure):

```
expr ::= expr1 + expr2 {
   expr2.resumer := expr.resumer
   expr2.lifetime := expr.node
   expr1.resumer := expr2.failer
   if expr2.gen & (expr.resumer ≠ null) then
      expr1.lifetime := expr.resumer
   else
      expr1.lifetime := expr.node
```

```
      expr.failer := expr1.failer
      expr.gen := (expr1.gen | expr2.gen)
      }
```

/expr provides an example of an operation that can fail. If there is no rightmost resumer of the entire expression, it is the rightmost resumer of the operand. The lifetime of the operand is simply the operation, by the same argument used for $expr_2$ of addition. The computation of failer is also analogous to that of addition. The expression is a generator if the operand is a generator:

```
   expr ::= /expr1 {
      if expr.resumer = null then
         expr1.resumer := expr.node
      else
         expr1.resumer := expr.resumer
      expr1.lifetime := expr.node
      expr.failer := expr1.failer
      expr.gen := expr1.gen
      }
```

!expr differs from /expr in that it can generate multiple results. If it can be resumed, the result of the operand must be retained through the rightmost resumer:

```
   expr ::= !expr1 {
      if expr.resumer = null then {
         expr1.resumer := expr.node
         expr1.lifetime := expr.node
         }
      else {
         expr1.resumer := expr.resumer
         expr1.lifetime := expr.resumer
         }
      expr.failer := expr1.failer
      expr.gen := true
      }
```

16.6 Control Structures

Other operations follow the general pattern of the ones presented above. Control structures, on the other hand, require unique attribute computations. In particular, several control structures bound subexpressions, limiting backtracking. For example, not bounds its argument and discards the value. If it has no resumer, then it is the rightmost operation that can fail. The expression is not a generator:

```
   expr ::= not expr1 {
      expr1.resumer := null
      expr1.lifetime := null
      if expr.resumer = null then
         expr.failer := expr.node
      else
         expr.failer := expr.resumer
      expr.gen := false
      }
```

$expr_1$; $expr_2$ bounds $expr_1$ and discards the result. Because the result of $expr_2$ is the result of the entire expression, the code generator makes their result locations synonymous. This

is reflected in the lifetime computations. Indeed, all the attributes of $expr_2$ and those of the expression as a whole are the same:

```
expr ::= expr1 ; expr2 {
   expr1.resumer := null
   expr1.lifetime := null
   expr2.resumer := expr.resumer
   expr2.lifetime := expr.lifetime
   expr.failer := expr2.failer
   expr.gen := expr2.gen
   }
```

A reasonable implementation of alternation places the result of each subexpression into the same location: the location associated with the expression as a whole. This is reflected in the lifetime computations. The resumer of the entire expression is also the resumer of each subexpression. Backtracking out of the entire expression occurs when backtracking out of $expr_2$ occurs. This expression is a generator:

```
expr ::= expr1 | expr2 {
   expr2.resumer:= expr.resumer
   expr2.lifetime := expr.lifetime
   expr1.resumer := expr.resumer
   expr1.lifetime := expr.lifetime
   expr.failer := expr2.failer
   expr.gen := true
   }
```

The first operand of an if expression is bounded and its result is discarded. The other two operands are treated similar to those of alternation. Because there are two independent execution paths, the rightmost resumer may not be well-defined. However, it is always conservative to treat the resumer as if it is farther right than it really is; this just means that an intermediate value is kept around longer than needed. If there is no resumer beyond the if expression, but at least one of the branches can fail, the failure is treated as if it came from the end of the if expression (represented by the node for the expression). Because backtracking out of an if expression is rare, this inaccuracy is of little practical consequence. The if expression is a generator if either branch is a generator:

```
expr ::= if expr1 then expr2 else expr3 {
   expr3.resumer := expr.resumer
   expr3.lifetime := expr.lifetime
   expr2.resumer := expr.resumer
   expr2.lifetime := expr.lifetime
   expr1.resumer := null expr1.lifetime := null
   if expr.resumer = null & (expr1.failer null | expr2.failer
null) then
      expr.failer := expr.node
   else
      expr.failer = expr.resumer
   expr.gen := (expr2.gen | expr3.gen)
   }
```

The do clause of every is bounded and its result discarded. The control clause is always resumed at the end of the loop and can never be resumed by anything else. The value of the control clause is discarded. Ignoring break expressions, the loop always fails:

```
expr ::= every expr1 do expr2 {
   expr2.resumer := null
   expr2.lifetime := null
   expr1.resumer := expr.node
```

```
expr1.lifetime := null
expr.failer := expr.node
expr.gen := false
}
```

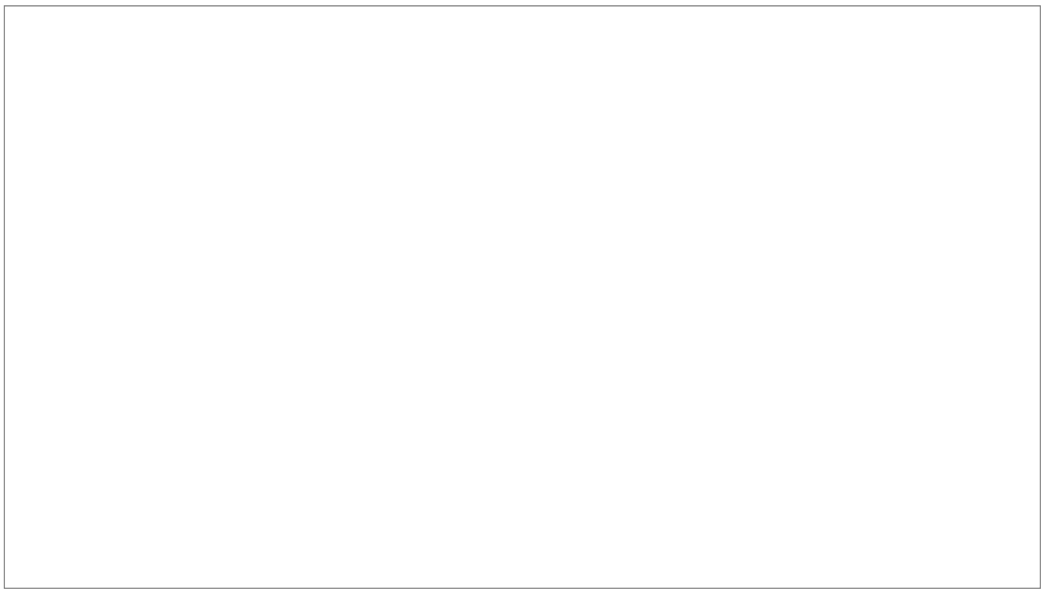
Handling break expressions requires some stack-like attributes. These are similar to the ones used in the control flow analyses described in O'Bagy's dissertation [.tr88-31.] and the ones used to construct flow graphs in the original technical report on type inferencing.

The attributes presented here can be computed with one walk of the syntax tree. At a node, subtrees are processed in reverse execution order: first the resumer and lifetime attributes of a subtree are computed, then the subtree is walked. Next the failer and gen attributes for the node itself are computed, and the walk moves back up to the parent node.

Chapter 17: Overview of the Compiler

17.1 Components of the Compiler

The Icon compiler is divided into two components: a run-time system and the compiler itself. This organization is illustrated below. In the diagram, labeled boxes represent programs, other text (some of it delimited by braces) represents files, and arrows represent data flow.



The run-time system appears above the dotted line and the compiler itself appears below the line. The programs shown with the run-time system are executed once when the run-time system is installed or updated. They build a data base, rt.db, and an object-code library, rt.a, for use by the compiler. The general definition of the term ``data base'' is used here: a collection of related data. rt.db is stored as a text file. It is accessed and manipulated in internal tables by the programs rtt and iconc. The rtt program is specific to the Icon compiler system and is described below. The C compiler and the library maintenance program are those native to the system on which the Icon compiler is being used. The format of the object-code library is dictated by the linker used with the C compiler. The file rt.h contains C definitions shared by the routines in the run-time system and code generated by the compiler.

The diagram of the compiler itself reflects the fact that the Icon compiler uses a C compiler and linker as a back end. However, iconc automatically invokes these programs, so the Icon programmer sees a single tool that takes Icon source as input and produces an executable file.

17.2 The Run-time System

As with the run-time system for the interpreter, the run-time system for the compiler implements Icon's operations. However, the compiler has needs beyond those of the interpreter. In the interpreter's run-time system, all operations are implemented as C

functions conforming to certain conventions. The interpreter uses the same implementation of an operation for all uses of the operation. While this approach results in acceptable performance for many Icon programs, the purpose of an optimizing compiler is to obtain better performance. A major goal in the development of iconc is to use information from type inferencing to tailor the parameter passing and parameter type conversions of an operation to particular uses of the operation and to place code in line where appropriate. The compiler needs a mechanism to support this tailored operation invocation. In addition, the compiler needs information about the properties of operations for use in performing type inferencing and other analyses.

In addition to supporting the analyses and optimizations of iconc, there are several other major goals in the design of the compiler's run-time system. These include

- Specification of all information about an operation in one place.
- Use of one coding to produce both general and tailored implementation of an operation.
- Use of the pre-existing run-time system as a basis for the new one.

Most of the design goals for the run-time system would best be served by a special-purpose language in which to implement the run-time system. Such a language would allow the properties of an operation needed by various analyses to be either explicitly coded or easily inferred from parts of the code used to produce an implementation of the operation. The language would also allow easy recognition and manipulation of parts of the code that need to be tailored to individual uses of an operation. In addition, the language would provide support for features of Icon such as its data types and goal-directed evaluation.

While a special-purpose language is consistent with most design goals, it is not consistent with using the interpreter's run-time system written in C as a basis for that of the compiler. A special-purpose language also has the problem that it requires a large effort to implement. These conflicting goals are resolved with a language that is a compromise between an ideal special-purpose implementation language and C. The core of the language is C, but the language contains special features for specifying those aspects of a run-time operation that must be dealt with by the compiler. This language is called the *implementation language* for the Icon compiler's run-time system. Because the implementation language is designed around C, much of the detailed code for implementing an operation can be borrowed from the interpreter system with only minor changes. The important facets of the implementation language are discussed here. A full description of the language can be found in the reference manual for the language [.ipd79.]. The core material from this reference manual is included as Appendix A of this dissertation.

17.3 The Implementation Language

The implementation language is used to describe the operators, keywords, and built-in functions of Icon. In addition to these operations, the run-time system contains routines to support other features of Icon such as general invocation, co-expression activation, and storage management. These other routines are written in C.

The program rtt takes as input files containing operations coded in the implementation language and translates the operations into pure C. rtt also builds the data base, rt.db, with

information about the operations. In addition to operations written in the implementation language, rtt input may contain C functions. These C functions can use several of the extensions available to the detailed C code in the operations. These extensions are translated into ordinary C code. While not critical to the goals of the run-time system design, the ability to use these extensions in otherwise ordinary C functions is very convenient.

The definition of an operation is composed of three layers. The outer layer brackets the code for the operation. It consists of a header at the beginning of the code and the reserved word end at the end of the code. The header may be preceded by an optional description of the operation in the form of a string literal; this description is used only for documentation. The second layer consists of type checking and type conversion code. Type checking code may be nested. The inner layer is the detailed C code, abstract type computations, and code to handle run-time errors. An abstract type computation describes the possible side-effects and result types of the operation in a form that type inferencing can use. This feature is needed because it is sometimes impractically difficult to deduce this information from the C code. The code to handle run-time errors is exposed; that is, it is not buried within the detailed C code. Because of this, type inferencing can easily determine conditions under which an operation terminates without either producing a value or failing. (A further reason for exposing this code is explained in the implementation language reference manual in the section on scoping.)

An operation header starts with one of the three reserved words operator, function, or keyword. The header contains a description of the operation's *result sequence*, that is, how many results it can produce. This includes both the minimum and maximum number of results, with indicating an unbounded number. In addition, it indicates, by a trailing +, when an operation can be resumed to perform a side-effect after it has produced its last result. This information is somewhat more detailed than can easily be inferred by looking at the returns, suspends, and fails in the detailed C code. The information is put in the data base for use by the analysis phases of iconc.

An operation header also includes an identifier. This provides the name for built-in functions and keywords. For all types of operations, rtt uses the identifier to construct the names of the C functions that implement the operations. The headers for operators also include an operator symbol. The parser for iconc is not required to use this symbol for the syntax of the operation, but for most operations it does so; list creation, [...], is an example of an exception. The headers for built-in functions and operators include a parameter list. The list provides names for the parameters and indicates whether dereferenced and/or undereferenced versions of the corresponding argument are needed by the operation. It also indicates whether the operation takes a variable number of arguments.

The following are five examples of operation headers.

```
function{0,1+} move(i)
function{} bal(c1,c2,c3,s,i,j)
operator{1} [...] llist(elems[n])
operator{0,1} / null(underef x -> dx)
keyword{3} regions
```

move is a function that takes one argument. It may produce zero or one result and may be resumed to produce a side effect after its last result. bal is a function that takes six arguments. It produces an arbitrary number of results. The list-creation operator is given the symbol [...] (which may be used for string invocation, if string invocation is enabled)

and is given the name llist. It takes an arbitrary number of arguments with elems being the array of arguments and n being the number of arguments. List creation always produces one result. The / operator is given the name null. It takes one argument, but both undereferenced and dereferenced versions are needed by the operation. It produces either zero or one result. &regions is a keyword that produces three results.

Type checking and type conversion constructs consist of an if-then construct, an if-then-else construct, a type_case construct that selects code to execute based on the type of an argument, and a len_case construct that selects code to execute based on the number of arguments in the variable part of a variable-length argument list. The conditions in the if-then and if-then-else constructs are composed of operations that check the types of arguments or attempt to convert arguments to specific types.

A type check starts with `is:'. This is followed by the name of a type and an argument in parentheses. For example, the then clause of the following if is executed if x is a list.

```
if is:list(x) then ...
```

A type conversion is similar to a type check, but starts with `cnv:'. For example, the following code attempts to convert s to a string. If the conversion succeeds, the then clause of the if is executed.

```
if cnv:string(s) then ...
```

There are forms of conversion that convert a null value into a specified default value, forms that put a converted value in a location other than the parameter, and forms that convert Icon values into certain types of C values. The later type of conversion is convenient because the detailed code is expressed in C. In addition, exposing conversions back and forth between Icon and C values leaves open the possibility of future optimizations to eliminate unnecessary conversions to Icon values. The control clause of an if may also use limited forms of expressions involving boolean operators. The full syntax and semantics of conversions are described in the implementation language reference manual.

Detailed code is expressed in a slightly extended version of C and is introduced by one of two constructs. The first is

```
inline { extended C }
```

This indicates that it is reasonable for the Icon compiler to put the detailed code in line. The second construct is

```
body { extended C }
```

This indicates that the detailed code is too large to be put in line and should only appear as part of a C function in the run-time library. The person who codes the operation is free to decide which pieces of detailed code are suitable to in-lining and which are not. The decision is easily changed, so an operation can be fine tuned after viewing the code produced by the compiler.

One extension to C is the ability to declare variables that are tended with respect to garbage collection. Another extension is the ability to use some constructs of the implementation language, such as type conversions, within the C code. An important extension is the inclusion of return, suspend, and fail statements that are analogous to their Icon counterparts. This extension, combined with the operation headers, makes the coding of run-time routines independent of the C calling conventions used in the compiler system. The return and suspend statements have forms that convert C values to Icon

values, providing inverses to conversions in the type checking code of the implementation language.

This mechanism is more than is necessary for those keywords that are simple constants. For keywords that are string, cset, integer, or real constants there is a special form of definition. The Icon compiler treats keywords coded with these definitions as manifest constants. When future versions of the Icon compiler implement constant folding, that optimization will be automatically applied to these keywords.

17.4 Standard and Tailored Operation Implementations

For every operation that it translates, except keywords, rtt creates a C function conforming to the standard calling conventions of the compiler system. With the help of the C compiler and library maintenance routine, rtt puts an object module for that function in the compiler system's run-time library. This function is suitable for invocation through a procedure block. It is used with unoptimized invocations.

rtt creates an entry in the data base for every operation it translates, including keywords. This entry contains the code for the operation. The code is stored in the data base in a form that is easier to parse and process than the original source, and the body statements are replaced by calls to C functions. These C functions are in the run-time library and implement the code from the body statement. The calling conventions for these functions are tailored to the needs of the code and do not, in general, conform to the standard calling conventions of the compiler system.

When the compiler can determine that a particular operation is being invoked, it locates the operation in the data base and applies information from type inferencing to simplify or eliminate the code in the operation that performs type checking and conversions on arguments. These simplifications will eliminate detailed code that will never be executed in this invocation of the operation. The compiler can attempt the simplification because the type checking code is in the data base in a form that is easily dealt with. If enough simplification is possible, a tailored version of the operation is generated in line. This tailored version includes the simplified type checking code, if there is any left. For detailed code that has not been eliminated by the simplification, the tailored version also includes the C code from inline statements and includes calls to the functions that implement the code in body statements. The process of producing tailored versions of built-in operations is described in more detail in a later chapter.

Ideally, when unique types can be inferred for the operands of an operation, the compiler should either produce a small piece of type-specific in-line C code or produce a call to a type-specific C function implementing the operation. It is possible to code operations in the implementation language such that the compiler can do this. In addition, this is the natural way to code most operations. For the few exceptions, there are reasonable compromises between ideal generated code and elegant coding in the implementation language. This demonstrates that the design and implementation of the run-time system and its communication with the compiler is successful.

Chapter 18: Organization of Iconc

The Icon compiler, iconc, takes as input the source code for an Icon program and, with the help of a C compiler and linker, produces an executable file. The source code may be contained in several files, but iconc does not support separate compilation. It processes an entire program at once. This requirement simplifies several of the analyses, while allowing them to compute more accurate information. Without the entire program being available, the effects of procedures in other files is unknown. In fact, it is not possible to distinguish built-in functions from missing procedures. Type inferencing would be particularly weakened. It would have to assume that any call to an undeclared variable could have any side effect on global variables (including procedure variables) and on any data structure reachable through global variables or parameters.

18.1 Compiler Phases

Iconc is organized into a number of phases. These are illustrated in the diagram on the following page.

The initialization phase includes reading a data base of information about run-time routines into internal tables. This information is used in many of the other phases.

The source analysis phase consists of a lexical analyzer and parser. These are adapted from those used in the interpreter system. The parser generates abstract syntax trees and symbol tables for all procedures before subsequent phases are invoked. The symbol resolution phase determines the scope of variables that are not declared in the procedures where they are used. This resolution can only be done completely after all source files for the program are read. If a variable does not have a local declaration, the compiler checks to see whether the variable is declared global (possibly as a procedure or record constructor) in one of the source files. If not, the compiler checks to see whether the variable name matches that of a built-in function. If the name is still not resolved, it is considered to be a local variable.

18.2 Naive Optimizations

Naive optimizations involve invocation and assignment. These optimizations are done before type inferencing to aid that analysis. Certain ``debugging features'' of Icon such as the variable function interfere with these optimizations. By default, these features are disabled. If the user of iconc requests the debugging features, these optimizations are bypassed. While these optimizations are being done, information is gathered about whether procedures suspend, return, or fail. This information is used in several places in the compiler.

The invocation optimization replaces general invocation by a direct form of invocation to a procedure, a built-in function, or a record constructor. This optimization involves modifying nodes in the syntax tree. It only applies to invocations where the expression being invoked is a global variable initialized to a value in one of the three classes of procedure values. First, the Icon program is analyzed to determine which variables of this type appear only as the immediate operands of invocations. No such variable is ever assigned to, so it retains its initial value throughout the program (a more exact analysis could be done to determine the variables that are not assigned to, but this would seldom yield better results in real Icon programs because these programs seldom do anything with

procedure values other that invoke them). This means that all invocations of these variables can be replaced by direct invocations. In addition, the variables themselves can be discarded as they are no longer referenced.

The invocation optimization improves the speed of type inferencing in two ways, although it does nothing to improve the accuracy of the information produced. Performing type inferencing on direct invocations is faster than performing it on general invocations. In addition, type inferencing has fewer variables to handle, which also speeds it up.

The invocation optimization does improve code generated by the compiler. In theory, the optimization could be done better after type inferencing using the information from that analysis, but in practice this would seldom produce better results. On most real Icon programs, this optimization using the naive analysis replaces all general invocations with direct ones.

As noted in Chapter 3, it is important for type inferencing to distinguish strong updates from weak updates. The data base contains a general description of assignment, but it would be difficult for a type inferencing system to use the description in recognizing that a simple assignment or an augmented assignment to a named variable is a strong update. It is much easier to change general assignments where the left hand side is a named variable to a special assignment and have type inferencing know that the special assignment is a strong update. Special-casing assignment to named variables is also important for code generation. General optimizations to run-time routines are not adequate to produce the desired code for these assignments. The optimizations to assignment are described in Chapter 22.

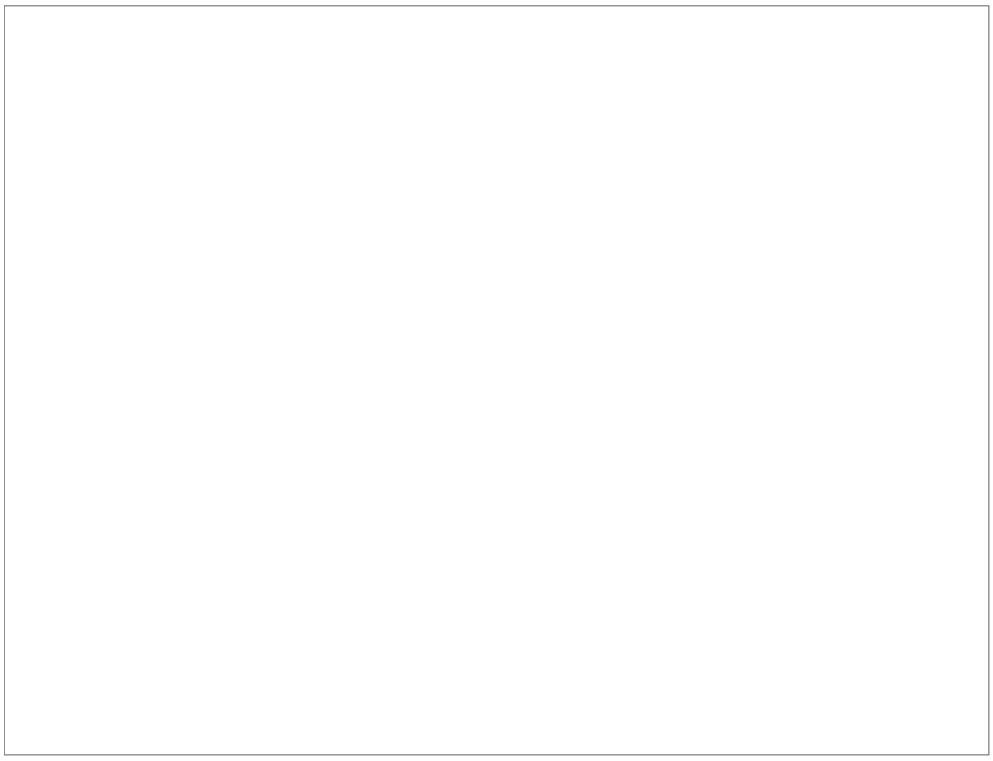
The details of type inferencing are described in other chapters. Producing code for the C main function, global variables, constants, and record constructors is straightforward. C code is written to two files for organizational purposes; it allows definitions and code to be written in parallel.

18.3 Code Generation for Procedures

Producing code for procedures involves several sub-phases. The sub-phases are liveness analysis, basic code generation, fix-up and peephole optimization, and output. During this phase of code generation, procedures are processed one at at time.

These sub-phases are described in later chapters. The code fix-up phase and peephole optimization are performed during the same pass over the internal representation of the C code. Some clean-up from peephole optimization is performed when the code is written. The logical organization of the compiler places the fix-up phase as a pass in code generation with peephole optimization being a separate phase. The organization of this dissertation reflects the logical organization of the compiler rather than its physical organization.

The physical organization of this phase is shown in the following diagram.



Chapter 19: The Implementation of Type Inferencing

Chapter 15 develops a theoretical type inferencing model for Icon, called Model 3. The purpose of that chapter is to explain the relationship between type inferencing and the semantics of Icon; for simplicity, some features of the language along with certain questions of practical importance are ignored in that chapter. This chapter describes the implementation of the type inferencing system used in the Icon compiler. The implementation is based on Model 3. This chapter describes solutions to those issues that must be addressed in developing a complete practical type inferencing system from Model 3. The issues include:

- the representation of types and stores
- the development of a type system for the full Icon language
- the handing of procedure calls and co-expression activation
- the determination of edges in the flow graph
- the computation of a fixed point for type information

In addition, performance of the abstract interpretation must be considered. This includes both speed and memory usage.

19.1 The Representation of Types and Stores

A type consists of a set of basic types (technically, it is a union of basic types, but the constituents of the basic types are not explicitly represented). The operations needed for these sets are: add a basic type to a set, form the union of two sets, form the intersection of two sets, test for membership in a set, and generate members of a subrange of basic types (for example, generate all members that are list types). A bit vector is used for the set representation, with a basic type represented by an integer index into the vector. The required operations are simple and efficient to implement using this representation. Unless the sets are large and sparse, this representation is also space efficient. While the sets of types are often sparse, for typical programs, they are not large.

A store is implemented as an array of pointers to types. A mapping is established from variable references to indexes in the store. In the type inferencing model, Model 3, presented in Chapter 3, there is one kind of store that contains all variables. In the actual implementation, temporary variables need not be kept in this store. The purpose of this store is to propagate a change to a variable to the program points affected by the change. A temporary variable is set in one place in the program and used in one place; there is nothing to determine dynamically. It is both effective and efficient to store the type of a temporary variable in the corresponding node of the syntax tree.

Another level of abstraction can be developed that requires much less memory than Model 3, but it must be modified to produce good results. This abstraction abandons the practice of a store for every edge in the flow graph. Instead it has a single store that is a merger of all the stores in Model 3 (the type associated with a variable in a merged store is the union of the types obtained for that variable from each store being merged). For variables that are truly of one type throughout execution, this abstraction works well.

Named variables do not have this property. They have an initial null value and usually are assigned a value of another type during execution. Because assignments to named variables are treated as strong updates, Model 3 can often deduce that a variable does not contain the null type at specific points in the flow graph.

For structure variables this further abstraction does work well in practice. These variables are initialized to the empty type (that is, no instances of these variables exist at the start of program execution), so the problem of the initial null type does not occur. Sometimes instances of these variables are created with the null type and later changed. However, the fact that assignments to these variables must be treated as weak updates means that type inferencing must assume that these variables can always retain their earlier type after an assignment. Propagating type information about structure variables through the syntax tree does not help much in these circumstances. While it is possible to construct example programs where Model 3 works better for structure variables than this further abstraction, experiments with prototype type inferencing systems indicate that the original system seldom gives better information for real programs [.tr88-25.].

Type inferencing in the compiler is implemented with two kinds of stores: local stores that are associated with edges in the flow graph and contain named variables (both local and global variables) and a global store that contains structure variables (in the implementation, the global store is actually broken up by structure-variable type into several arrays).

19.2 A Full Type System

Model 3 from Chapter 3 includes no structure type other than lists, nor does it consider how to handle types for procedure and co-expression values to allow effective type inferencing in their presence. This section develops a complete and effective type system for Icon.

Most of the structure types of Icon are assigned several subtypes in the type inferencing system. As explained for lists in Chapter 3, these subtypes are associated with the program points where values of the type are created. The exception to this approach is records. One type is created per record declaration. While it is possible to assign a subtype to each use of a record constructor, in practice a given kind of record usually is used consistently with respect to the types of its fields throughout a program. The extra subtypes would require more storage while seldom improving the resulting type information.

For efficiency, the size of the bit vectors representing types and the size of the stores need to remain fixed during abstract interpretation. This means that all subtypes must be determined as part of the initialization of the type inferencing system. In order to avoid excessive storage usage, it is important to avoid creating many subtypes for program points where structures are not created. The invocation optimization described in Chapter 6 helps determine where structures can and cannot be created by determining for most invocations what operation is used. The type inferencing system determines what structures an operation can create by examining the abstract type computations associated with the operation in the data base. A new clause in an abstract type computation indicates that a structure can be created. The following example is the abstract type computation from the built-in function list. It indicates the the function creates and returns a new list with elements whose type is the same as that of the parameter x (the second parameter).

```
abstract {
   return new list(type(x))
   }
```

This is the clause as written by the programmer developing the run-time library; it is translated into an internal form for storage in the data base.

Invocation optimizations may not identify the operation associated with some invocations. The initialization phase of type inferencing skips these invocations. Type inferencing may later discover that one of these invocations can create a structure. Each structure type is given one subtype that is used for all of these later discoveries. While it is possible for there to be several of these creation points representing logically distinct subtypes, this seldom occurs in practice. If it does happen, type inferencing produces a correct, but less precise, result.

The type system contains representations for all run-time values that must be modeled in the abstract interpretation. These run-time values can be divided into three categories, each of which is a superset of the previous category:

- the first-class Icon values
- the intermediate values of expression evaluation
- the values used internally by Icon operations

Just as these categories appear in different places during the execution of an Icon program, the corresponding types appear in different places during abstract interpretation. If certain types cannot appear as the result of a particular type computation, it is not necessary to have elements in the bit vector produced by the computation to represent those types. It is particularly important to minimize the memory used for stores associated with edges of the flow graph (this is discussed more in the last section of this chapter). These stores contain only the types of the smallest set listed above: the first-class values.

Types (or subtypes) are allocated bit vector indexes. The first-class types may appear as the result of any of the three classes of computation. They are allocated indexes at the front of the bit vectors. If they are the only types that can result from an abstract computation, the bit vector for the result has no elements beyond that of the last first-class types. The first-class types are:

- null
- string
- cset
- integer
- real
- file
- list subtypes
- set subtypes
- table subtypes
- record subtypes

- procedure subtypes
- co-expression subtypes

The list subtypes are allocated with

- one for the argument to the main procedure
- one for each easily recognized creation point
- one representing all other lists

The set subtypes are allocated with

- one for each easily recognized creation point
- one representing all other sets

The table subtypes are allocated with

- one for each easily recognized creation point
- one representing all other tables

The record subtypes are allocated with one for each record declaration. The procedure subtypes are allocated with

- one for each procedure
- one for each record constructor
- one for each built-in function
- one representing operators available for string invocation

Note that procedure subtypes are allocated after most procedure and function values have been eliminated by invocation optimizations (the procedures and functions are still there, they are just not Icon values). Therefore, few of these subtypes are actually allocated. The co-expression subtypes are allocated with

- one for the main co-expression
- one for each create expression

The bit vectors used to hold the intermediate results of performing abstract interpretation on expressions must be able to represent the basic types plus the variable reference types. Variable reference types are allocated bit vector indexes following those of the basic types. The bit vectors for intermediate results are just long enough to contain these two classifications of types. The variable reference types are

- integer keyword variable types
- &pos
- &subject
- substring trapped variable types
- table-element trapped variable types

- list-element reference types
- table assigned-value reference types
- field reference types
- global variable reference types
- local variable reference types

&random and &trace behave the same way from the perspective of the type inferencing system, so they are grouped under one type as integer keyword variables. The fact that &pos can cause assignment to fail is reflected in the type inferencing system, so it is given a separate type. &subject is the only string keyword variable so it is in a type by itself.

Substring trapped variables and table-element trapped variables are ``hidden'' structures in the implementation of Icon. They appear as intermediate results, but are only indirectly observable in the semantics of Icon. In order to reflect these semantics in the type inferencing system, there are substring trapped variable types and table-element trapped variable types. These are structure types similar to sets, but are variable reference types rather than first-class types. The substring trapped variable types are allocated with

- one for each easily recognized creation point
- one representing all other substring trapped variables

The table-element trapped variable types are allocated with

- one for each easily recognized creation point
- one representing all other table-element trapped variables

List elements, table assigned-values, and record fields are all variables that can appear as the intermediate results of expression evaluation. The type system has corresponding variable reference types to represent them. The list-element reference types are allocated with one for each list types. The table assigned-value reference types are allocated with one for each table type. The field reference types are allocated with one for each record field declaration.

Identifiers are variables and are reflected in the type system. The global variable reference types are allocated with

- one for each global variable (except those eliminated by invocation optimizations).
- one for each static variable

The local variable reference types are allocated with one for each local variable, but the bit vector indexes and store indexes are reused between procedures. The next section describes why this reuse is possible.

Icon's operators use a number of internal values that never ``escape'' as intermediate results. Some of these internal values are reflected in the type system in order to describe the semantics of the operations in the abstract interpretation. The full set of types that can be used to express these semantics are presented in the syntax of the abstract type

computations of the run-time implementation language; see Appendix A. In addition to the types of intermediate results, these types include

- set-element reference types
- table key reference types
- table default value reference types
- references to the fields within substring trapped variables that reference variables
- references to fields within table-element trapped variables that reference tables

These types are allocated bit vector indexes at the end of the type system. The only bit vectors large enough to contain them are the temporary bit vectors used during interpretation of the abstract type computations of built-in operations.

Set elements, table keys, and table default values do not appear as variable references in the results of expression evaluation. However, it is necessary to refer to them in order to describe the effects of certain Icon operations. For this reason, they are included in the type system. The set-element reference types are allocated with one for each set type. The table key reference types are allocated with one for each table type. The table default value reference types are allocated with one for each table type.

Substring trapped variable types contain references to the variable types being trapped and table-element trapped variable types contain references to the table types containing the element being trapped. These references are fields within these trapped variable types. There is one field reference type for each trapped variable type.

19.3 Procedure Calls and Co-Expression Activations

A type inferencing system for the full Icon language must handle procedures and co-expressions. As noted above, each procedure and each create expression is given its own type. This allows the type inferencing system to accurately determine which procedures are invoked and what co-expressions might be activated by a particular expression.

The standard semantics for procedures and co-expressions can be implemented using stacks of procedure activation frames, with one stack per co-expression. The first frame, on every stack except that of the main co-expression, is a copy of the frame for the procedure that created the co-expression. The local variables in this frame are used for evaluating the code of the co-expression. The type inferencing system uses a trivial abstraction of these procedure frame stacks, while capturing the possible transfers of control induced by procedure calls and co-expression activations.

The type inferencing system, in effect, uses an environment that has one frame per procedure, where that frame is a summary of all frames for the procedure that could occur in a corresponding environment of an implementation of the standard semantics. The frame is simply a portion of the store that contains local variables. Because no other procedure can alter a local variable, it is unnecessary to pass the types of local variables into procedure calls. If the called procedure returns control via a return, suspend, or fail, the types are unchanged, so they can be passed directly across the call. This allows the type inferencing system to keep only the local variables of a procedure in the stores on the edges of the flow graph for the procedure, rather than keeping the local variables of all procedures. Global variables must be passed into and out of procedure calls. Because

static variables may be altered in recursive calls, they must also be passed into and out of procedure calls.

A flow graph for an entire program is constructed from the flow graphs for its individual procedures and co-expressions. An edge is added from every invocation of a procedure to the head of that procedure and edges are added from every return, suspend, and fail back to the invocation. In addition, edges must be added from an invocation of a procedure to all the suspends in the procedure to represent resumption. When it is not possible to predetermine which procedure is being invoked, edges are effectively added from the invocation to all procedures whose invocation cannot be ruled out based on the naive invocation optimizations. Edges are effectively added between all co-expressions and all activations, and between all activations. Information is propagated along an edge when type inferencing deduces that the corresponding procedure call or co-expression activation might actually occur. The representation of edges in the flow graph is discussed below.

Type inferencing must reflect the initializations performed when a procedure is invoked. Local variables are initialized to the null value. On the first call to the procedure, static variables are also initialized to the null value. The initialization code for the standard semantics is similar to

```
initialize locals
if (first_call) {
   initialize statics
   user initialization code
   }
```

In type inferencing, the variables are initialized to the null type and the condition on the if is ignored. Type inferencing simply knows that the first-call code is executed sometimes and not others. Before entering the main procedure, global variables are set to the null type and all static variables are set to the empty type. In some sense, the empty type represents an impossible execution path. Type inferencing sees paths in the flow graph from the start of the program to the body of a procedure that never pass through the initialization code. However, static variables have an empty type along this path and no operation on them is valid. Invalid operations contribute nothing to type information.

19.4 The Flow Graph and Type Computations

In order to determine the types of variables at the points where they are used, type inferencing must compute the contents of the stores along edges in the flow graph. Permanently allocating the store on each edge can use a large amount of memory. The usage is

```
M = E (G + S + L) T / 8
```

where

```
M = total memory, expressed in bytes, used by stores
E = the number of edges in the program flow graph
G = the number of global variables in the program
S = the number of static variables in the program
L = the maximum number of local variables in any procedure
T = the number of types in the type system
```

Large programs with many structure creation points can produce thousands of edges, dozens of variables, and hundreds of types, requiring megabytes of memory to represent the stores.

The code generation phase of the compiler just needs the (possibly dereferenced) types of operands, not the stores. If dereferenced types are kept at the expressions where they are needed, it is not necessary to keep a store with each edge of the flow graph.

Consider a section of straight-line code with no backtracking. Abstract interpretation can be performed on the graph starting with the initial store at the initial node and proceeding in execution order. At each node, the store on the edge entering the node is used to dereference variables and to compute the next store if there are side effects. Once the computations at a node are done, the store on the edge entering the node is no longer needed. If updates are done carefully, they can be done in-place, so that the same memory can be used for both the store entering a node and the one leaving it.

In the case of branching control paths (as in a case expression), abstract interpretation must proceed along one path at a time. The store at the start the branching of paths must be saved for use with each path. However, it need only be saved until the last path is interpreted. At that point, the memory for the store can be reused. When paths join, the stores along each path must be merged. The merging can be done as each path is completed; the store from the path can then be reused in interpreting other paths. When all paths have been interpreted, the merged store becomes the current store for the node at the join point.

The start of a loop is a point where control paths join. Unlike abstract interpretation for the joining of simple branching paths, abstract interpretation for the joining of back edges is not just a matter of interpreting all paths leading to the join point before proceeding. The edge leaving the start of the loop is itself on a path leading to the start of the loop. Circular dependencies among stores are handled by repeatedly performing the abstract interpretation until a fixed point is reached. In this type inferencing system, abstract interpretation is performed in iterations, with each node in the flow graph visited once per iteration. The nodes are visited in execution order. For back edges, the store from the previous iteration is used in the interpretation. The stores on these edges must be kept throughout the interpretation. These stores are initialized to map all variables to the empty type. This represents the fact that the abstract interpretation has not yet proven that execution can reach the corresponding edge.

The type inferencing system never explicitly represents the edges of the flow graph in a data structure. Icon is a structured programming language. Many edges are implicit in a tree walk performed in forward execution order -- the order in which type inferencing is performed. The location of back edges must be predetermined in order to allocate stores for them, but the edges themselves are effectively recomputed as part of the abstract interpretation.

There are two kinds of back edges. The back edges caused by looping control structures can be trivially deduced from the syntax tree. A store for such an edge is kept in the node for the control structure. Other back edges are induced by goal-directed evaluation. These edges are determined with the same techniques used in liveness analysis. A store for such an edge is kept in the node of the suspending operation that forms the start of the loop. Because the node can be the start of several nested loops, this store is actually the merged store for the stores that theoretically exist on each back edge.

At any point in abstract interpretation, three stores are of interest. The *current store* is the store entering the node on which abstract interpretation is currently being performed. It is created by merging the stores on the incoming edges. The *success store* is the store

representing the state of computations when the operation succeeds. It is usually created by modifying the current store. The *failure store* is the store representing the state of computations when the operation fails.

In the presence of a suspended operation, the failure store is the store kept at the node for that operation. A new failure store is established whenever a resumable operation is encountered. This works because abstract interpretation is performed in forward execution order and resumption is LIFO. Control structures, such as if-then-else, with branching and joining paths of execution, cause difficulties because there may be more than one possible suspended operation when execution leaves the control structure. This results in more than one failure store during abstract interpretation. Rather than keeping multiple failure stores when such a control structure has operations that suspend on multiple paths, type inferencing pretends that the control structure ends with an operation that does nothing other than suspend and then fail. It allocates a store for this operation in the node for the control structure. When later operations that fail are encountered, this store is updated. The failure of this imaginary operation is the only failure seen by paths created by the control structure and the information needed to update the failure stores for these paths is that in the store for this imaginary operation. This works because the imaginary operation just passes along failure without modifying the store.

In the case where a control structure transforms failure into forward execution, as in the first subexpression of a compound expression, the failure store is allocated (with empty types) when the control structure is encountered and deallocated when it is no longer needed. If no failure can occur, no failure store need be allocated. The lack of possible failure is noted while the location of back edges is being computed during the initialization of type inferencing. Because a failure store may be updated at several operations that can fail, these are weak updates. Typically, a failure store is updated by merging the current store into it.

The interprocedural flow graph described earlier in this chapter has edges between invocations and returns, suspends, and fails. Type inferencing does not maintain separate stores for these theoretical edges. Instead it maintains three stores per procedure that are mergers of stores on several edges. One store is the merger of all stores entering the procedure because of invocation; this store contains parameter types in addition to the types of global and static variables. Another store is the merger of all stores entering the procedure because of resumption. The third store is the merger of all stores leaving the procedure because of returns, suspends, and fails. There is also a result type associated with the procedure. It is updated when abstract interpretation encounters returns and suspends.

Two stores are associated with each co-expression. One is the merger of all stores entering the co-expression and the other is the merger of all stores leaving the co-expression. Execution can not only leave through an activation operator, it can also re-enter through the activation. The store entering the activation is a merger of the stores entering all co-expressions in which the activation can occur. Because a procedure containing an activation may be called from several co-expressions, it is necessary to keep track of those co-expressions. A set of co-expressions is associated with each procedure for this purpose. Each co-expression also contains a type for the values transmitted to it. The result type of an activation includes the result types for all co-expressions that might be activated and the types of all values that can be transmitted to a co-expression that the activation might be executed in.

When type inferencing encounters the invocation of a built-in operation, it performs abstract interpretation on the representation of the operation in the data base. It interprets the type-checking code to see what paths might be taken through the operation. The interpretation uses the abstract type computations and ignores the detailed C code when determining the side effects and result type of the operation. Because the code at this level of detail contains no loops, it is not necessary to save stores internal to operations. An operation is re-interpreted at each invocation. This allows type inferencing to produce good results for polymorphous operations. At this level, the code for an operation is simple enough that the cost of re-interpretation is not prohibitive. All side effects within these operations are treated as weak updates; the only strong updates recognized by type inferencing are the optimized assignments to named variables (see Chapter 6).

The abstract semantics of control structures are hard-coded within the type inferencing system. The system combines all the elements described in this chapter to perform the abstract interpretation. A global flag is set any time an update changes type information that is used in the next iteration of abstract interpretation. The flag is cleared between iterations. If the flag is not set during an iteration, a fixed point has been reached and the interpretation halts.

Chapter 20: Code Generation

This chapter describes the code generation process. The examples of generated code presented here are produced by the compiler, but some cosmetic changes have been made to enhance readability.outer function for the procedure. This is the function that is seen as implementing the procedure. In addition to the outer function, there may be several functions for success continuations that are used to implement generative expressions.

The outer function of a procedure must have features that support the semantics of an Icon call, just as a function implementing a run-time operation does. In general, a procedure must have a procedure block at run time. This procedure block references the outer function. All functions referenced through a procedure block must conform to the compiler system's standard calling conventions. However, invocation optimizations usually eliminate the need for procedure variables and their associated procedure blocks. When this happens, the calling conventions for the outer function can be tailored to the needs of the procedure.

As explained in Chapter 2, the standard calling convention requires four parameters: the number of arguments, a pointer to the beginning of an array of descriptors holding the arguments, a pointer to a result location, and a success continuation to use for suspension. The function itself is responsible for dereferencing and argument list adjustment. In a tailored calling convention for an outer function of a procedure, any dereferencing and argument list adjustment is done at the call site. This includes creating an Icon list for the end of a variable-sized argument list. The compiler produces code to do this that is optimized to the particular call. An example of an optimization is eliminating dereferencing when type inferencing determines that an argument cannot be a variable reference.

The number of arguments is never needed in these tailored calling conventions because the number is fixed for the procedure. Arguments are still passed via a pointer to an array of descriptors, but if there are no arguments, no pointer is needed. If the procedure returns no value, no result location is needed. If the procedure does not suspend, no success continuation is needed.

In addition to providing a calling interface for the rest of the program, the outer function must provide local variables for use by the code generated for the procedure. These variables, along with several other items, are located in a procedure frame. An Icon procedure frame is implemented as a C structure embedded within the frame of its outer C function (that is, as a local struct definition). Code within the outer function can access the procedure frame directly. However, continuations must use a pointer to the frame. A global C variable, pfp, points to the frame of the currently executing procedure. For efficiency, continuations load this pointer into a local register variable. The frame for a main procedure might have the following declaration.

```
struct PF00_main {
   struct p_frame old_pfp;
   dptr old_argp;
   dptr rslt;
   continuation succ_cont;
   struct {
      struct tend_desc *previous;
```

```
      int num;
      struct descrip d[5];
      } tend;
   };
```

with the definition

```
struct PF00_main frame;
```

in the procedure's outer function. A procedure frame always contains the following five items: a pointer to the frame of the caller, a pointer to the argument list of the caller, a pointer to the result location of this call, a pointer to the success continuation of this call, and an array of tended descriptors for this procedure. It may also contain C integer variables, C double variables, and string and cset buffers for use in converting values. If debugging is enabled, additional information is stored in the frame. The structure p_frame is a generic procedure frame containing a single tended descriptor. It is used to define the pointer old_pfp because the caller can be any procedure.

The argument pointer, result location, and success continuation of the call must be available to the success continuations of the procedure. A global C variable, argp, points the argument list for the current call. This current argument list pointer could have been put in the procedure frame, but it is desirable to have quick access to it. Quick access to the result location and the success continuation of the call is less important, so they are accessed indirectly through the procedure frame.

The array of descriptors is linked onto the chain used by the garbage collector to locate tended descriptors. These descriptors are used for Icon variables local to the procedure and for temporary variables that hold intermediate results. If the function is responsible for dereferencing and argument list adjustment (that is, if it does not have a tailored calling convention), the modified argument list is constructed in a section of these descriptors.

The final thing provided by the outer function is a *control environment* in which code generation starts. In particular, it provides the bounding environment for the body of the procedure and the implicit failure at the end of the procedure. The following C function is the tailored outer function for a procedure named p. The procedure has arguments and returns a result. However, it does not suspend, so it needs no success continuation.

```
static int P01_p(args, rslt)
dptr args;
dptr rslt;
{
   struct PF01_p frame;
   register int signal;
   int i;
   frame.old_pfp = pfp;
   pfp = (struct p_frame )&frame;
   frame.old_argp = argp;
   frame.rslt = rslt;
   frame.succ_cont = NULL;

   for (i = 0; i < 3; ++i)
      frame.tend.d[i].dword = D_Null;
   argp = args;
   frame.tend.num = 3;
   frame.tend.previous = tend;
   tend = (struct tend_desc )&frame.tend;
```

```
      translation of the body of procedure p

   L10: /* bound */
   L4: /* proc fail */
      tend = frame.tend.previous;
      pfp = frame.old_pfp;
      argp = frame.old_argp;
      return A_Resume;
   L8: /* proc return */
      tend = frame.tend.previous;
      pfp = frame.old_pfp;
      argp = frame.old_argp;
      return A_Continue;
      }
```

The initialization code reflects the fact that this function has three tended descriptors to use for local variables and intermediate results. L10 is both the bounding label and the failure label for the body of the procedure. Code to handle procedure failure and return (except for setting the result value) is at the end of the outer function. As with bounding labels, the labels for these pieces of code have associated signals. If a procedure fail or return occurs in a success continuation, the continuation returns the corresponding signal which is propagated to the outer function where it is converted into a goto. The code for procedure failure is located after the body of the procedure, automatically implementing the implicit failure at the end of the procedure.

20.1 Translating Icon Expressions

Icon's goal-directed evaluation makes the implementation of control flow an important issue during code generation. Code for an expression is generated while walking the expression's syntax tree in forward execution order. During code generation there is always a *current failure action*. This action is either ``branch to a label'' or ``return a signal''. When the translation of a procedure starts, the failure action is to branch to the bounding label of the procedure body. The action is changed when generators are encountered or while control structures that use failure are being translated.

The allocation of temporary variables to intermediate results is discussed in more detail later. However, some aspects of it will be addressed before presenting examples of generated code. The result location of a subexpression may be determined when the parent operation is encountered on the way down the syntax tree. This is usually a temporary variable, but does not have to be. If no location has been assigned by the time the code generator needs to use it, a temporary variable is allocated for it. This temporary variable is used in the code for the parent operation.

The code generation process is illustrated below with examples that use a number of control structures and operations. Code generation for other features of the language is similar.

Consider the process of translating the following Icon expression:

```
      return if a = (1 | 2) then "yes" else "no"
```

When this expression is encountered, there is some current failure action, perhaps a branch to a bounding label. The return expression produces no value, so whether a result location has been assigned to it is of no consequence. If the argument of a return fails, the procedure fails. To handle this possibility, the current failure action is set to branch to the

label for procedure failure before translating the argument (in this example, that action is not used). The code for the argument is then generated with its result location set to the result location of the procedure itself. Finally the result location is dereferenced and control is transferred to the procedure return label. The dereferencing function, deref, takes two arguments: a pointer to a source descriptor and a pointer to a destination descriptor.

```
    code for the if expression
    deref(rslt, rslt);
    goto L7 /* proc return */;
```

The control clause of the if expression must be bounded. The code implementing the then clause must be generated following the bounding label for the control clause. A label must also be set up for the else clause with a branch to this label used as the failure action for the control clause. Note that the result location of each branch is the result location of the if expression which is in turn the result location of the procedure. Because neither branch of the if expression contains operations that suspend, the two control paths can be brought together with branch to a label.

```
    code for control clause
L4: /* bound */
    rslt->vword.sptr = "yes";
    rslt->dword = 3;
    goto L6 /* end if */;
L5: /* else */
    rslt->vword.sptr = "no";
    rslt->dword = 2;
L6: /* end if */
```

Using a branch and a label to bring together the two control paths of the if expression is an optimization. If the then or the else clauses contain operations that suspend, the general continuation model must be used. In this model, the code following the if expression is put in a success continuation, which is then called at the end of both the code for the then clause and the code for the else clause.

Next consider the translation of the control clause. The numeric comparison operator takes two operands. In this translation, the standard calling conventions are used for the library routine implementing the operator. Therefore, the operands must be in an array of descriptors. This array is allocated as a sub-array of the tended descriptors for the procedure. In this example, tended location 0 is occupied by the local variable, a. Tended locations 1 and 2 are free to be allocated as the arguments to the comparison operator. The code for the first operand simply builds a variable reference.

```
frame.tend.d[1].dword = D_Var;
frame.tend.d[1].vword.descptr = &frame.tend.d[0] /* a */;
```

However, the second operand is alternation. This is a generator and requires a success continuation. In this example, the continuation is given the name P02_main (the Icon expression is part of the main procedure). The continuation contains the invocation of the run-time function implementing the comparison operator and the end of the bounded expression for the control clause of the if. The function O0o_numeq implements the comparison operator. The if expression discards the operator's result. This is accomplished by using the variable trashcan as the result location for the call. The compiler knows that this operation does not suspend, so it passes a null continuation to the function. The end of the bounded expression consists of a transfer of control to the bounding label. This is accomplished by returning a signal. The continuation is

```
   static int P02_main()
   {
   register struct PF00_main *rpfp;

   rpfp = (struct PF00_main *)pfp;
   switch (O0o_numeq(2, &(rpfp->tend.d[1]), &trashcan,
(continuation)NULL))
      {
      case A_Continue:
         break;
      case A_Resume:
         return A_Resume;
      }
    return 4; /* bound */
    }
```

Each alternative of the alternation must compute the value of its subexpression and call the success continuation. The failure action for the first alternative is to branch to the second alternative. The failure action of the second alternative is the failure action of the entire alternation expression. In this example, the failure action is to branch to the else label of the if expression. In each alterative, a bounding signal from the continuation must be converted into a branch to the bounding label. Note that this bounding signal indicates that the control expression succeeded.

```
   frame.tend.d[2].dword = D_Integer;
   frame.tend.d[2].vword.integr = 1;
   switch (P02_main()) {
      case A_Resume:
         goto L2 /* alt */;
      case 4 /* bound */:
         goto L4 /* bound */;
      }
   L2: /* alt */
      frame.tend.d[2].dword = D_Integer;
      frame.tend.d[2].vword.integr = 2;
      switch (P02_main()) {
         case A_Resume:
            goto L5 /* else */;
         case 4 /* bound */:
            goto L4 /* bound */;
         }
```

The code for the entire return expression is obtained by putting together all the pieces. The result is the following code (the code for P02_main is not repeated).

```
   frame.tend.d[1].dword = D_Var;
   frame.tend.d[1].vword.descptr = &frame.tend.d[0] /* a */;
   frame.tend.d[2].dword = D_Integer;
   frame.tend.d[2].vword.integr = 1;
   switch (P02_main()) {
      case A_Resume:
         goto L2 /* alt */;
      case 4 /* bound */:
         goto L4 /* bound */;
      }
   L2: /* alt */
      frame.tend.d[2].dword = D_Integer;
      frame.tend.d[2].vword.integr = 2;
      switch (P02_main()) {
```

```
         case A_Resume:
            goto L5 /* else */;
         case 4 /* bound */:
            goto L4 /* bound */;
         }
   L4: /* bound */
      rslt->vword.sptr = yes;
      rslt->dword = 3;
      goto L6 /* end if */;

   L5: /* else */
      rslt->vword.sptr = no;
      rslt->dword = 2;
   L6: /* end if */
      deref(rslt, rslt);
      goto L7 /* proc return */;
```

20.2 Signal Handling

In order to produce signal handling code, the code generator must know what signals may be returned from a call. These signals may be either directly produced by the operation (or procedure) being called or they may originate from a success continuation. Note that either the operation or the continuation may be missing from a call, but not both. The signals produced directly by an operation are A_Resume, A_Continue, and A_FallThru (this last signal is only used internally within in-line code).

The signals produced by a success continuation belong to one of three categories: A_Resume, signals corresponding to labels within the procedure the continuation belongs to, and signals corresponding to labels in procedures farther down in the call chain. The last category only occurs when the procedure suspends. The success continuation for the procedure call may return a signal belonging to the calling procedure. This is demonstrated in the following example (the generated code has been ``cleaned-up'' a little to make it easier to follow). The Icon program being translated is

```
procedure main()
   write(p())
end
procedure p()
   suspend 1 to 10
end
```

The generative procedure p is called in a bounded context. The code generated for the call is

```
switch (P01_p(&frame.tend.d[0], P05_main)) {
   case 7 /* bound */:
      goto L7 /* bound */;
   case A_Resume:
      goto L7 /* bound */;
   }
L7: /* bound */
```

This call uses the following success continuation. The continuation writes the result of the call to p then signals the end of the bounded expression.

```
static int P05_main() {
   register struct PF00_main *rpfp;

   rpfp = (struct PF00_main )pfp;
```

```
      F0c_write(1, &rpfp->tend.d[0], &trashcan,
(continuation)NULL);
      return 7; /* bound */
   }
```

The to operator in procedure p needs a success continuation that implements procedure suspension. Suspension is implemented by switching to the old procedure frame pointer and old argument pointer, then calling the success continuation for the call to p. The success continuation is accessed with the expression rpfp->succ_cont. In this example, the continuation will only be the function P05_main. The suspend must check the signal returned by the procedure call's success continuation. However, the code generator does not try to determine exactly what signals might be returned by a continuation belonging to another procedure. Such a continuation may return an A_Resume signal or a signal belonging to some procedure farther down in the call chain. In this example, bounding signal 7 will be returned and it belongs to main.

If the call's success continuation returns A_Resume, the procedure frame pointer and argument pointer must be restored, and the current failure action must be executed. In this case, that action is to return an A_Resume signal to the to operator. If the call's success continuation returns any other signal, that signal must be propagated back through the procedure call. The following function is the success continuation for the to operator.

```
static int P03_p()
{
register int signal;
register struct PF01_p *rpfp;

rpfp = (struct PF01_p *)pfp;
deref(rpfp->rslt, rpfp->rslt);
pfp = rpfp->old_pfp;
argp = rpfp->old_argp;

signal = (*rpfp->succ_cont)();
if (signal != A_Resume) {
   return signal;
   }
pfp = (struct p_frame *)rpfp;
argp = NULL;
return A_Resume;
}
```

The following code implements the call to the to operator. The signal handling code associated with the call must pass along any signal from the procedure call's success continuation. These signals are recognized by the fact that the procedure frame for the calling procedure is still in effect. At this point, the signal is propagated out of the procedure p. Because the procedure frame is about to be removed from the C stack, the descriptors it contains must be removed from the tended list.

```
frame.tend.d[0].dword = D_Integer;
frame.tend.d[0].vword.integr = 1;
frame.tend.d[1].dword = D_Integer;
frame.tend.d[1].vword.integr = 10;
signal = O0k_to(2, &frame.tend.d[0], rslt, P03_p);
if (pfp != (struct p_frame )&frame) {
   tend = frame.tend.previous;
   return signal;
   }
```

```
switch (signal) {
   case A_Resume:
      goto L2 /* bound */;
   }
L2: /* bound */
```

So far, this discussion has not addressed the question of how the code generator determines what signals might be returned from a call. Because code is generated in execution order, a call involving a success continuation is generated before the code in the continuation is generated. This makes it difficult to know what signals might originate from the success continuation. This problem exists for direct calls to a success continuation and for calls to an operation that uses a success continuation.

The problem is solved by doing code generation in two parts. The first part produces incomplete signal handling code. At this time, code to handle the signals produced directly by an operation is generated. The second part of code generation is a fix-up pass that completes the signal handling code by determining what signals might be produced by success continuations.

The code generator constructs a call graph of the continuations for a procedure. Some of these calls are indirect calls to a continuation through an operation. However, the only effect of an operation on signals returned by a continuation is to intercept A_Resume signals. All other signals are just passed along. This is true even if the operation is a procedure. This call graph of continuations does not contain the procedure call graph nor does it contain continuations from other procedures.

Forward execution order imposes a partial order on continuations. A continuation only calls continuations strictly greater in forward execution order than itself. Therefore the continuation call graph is a DAG.

The fix-up pass is done with a bottom-up walk of the continuation call DAG. This pass determines what signals are returned by each continuation in the DAG. While processing a continuation, the fix-up pass examines each continuation call in that continuation. At the point it processes a call, it has determined what signals might be returned by the called continuation. It uses this information to complete the signal handling code associated with the call and to determine what signals might be passed along to continuations higher up the DAG. If a continuation contains code for a suspend, the fix-up pass notes that the continuation may return a *foreign* signal belonging to another procedure call. As explained above, foreign signals are handled by special code that checks the procedure frame pointer.

20.3 Temporary Variable Allocation

The code generator uses the liveness information for an intermediate value when allocating a temporary variable to hold the value. As explained in Chapter 4, this information consists of the furthest program point, represented as a node in the syntax tree, through which the intermediate value must be retained. When a temporary variable is allocated to a value, that variable is placed on a *deallocation list* associated with the node beyond which its value is not needed. When the code generator passes a node, all the temporary variables on the node's deallocation list are deallocated.

The code generator maintains a *status array* for temporary variables while it is processing a procedure. The array contains one element per temporary variable. This array is

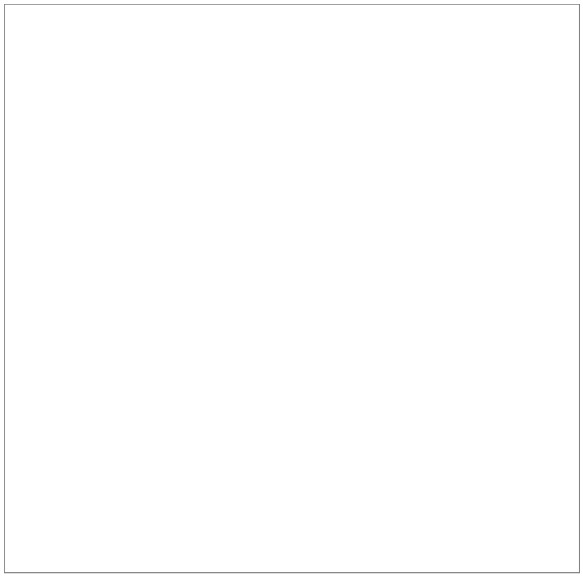
expandable, allowing a procedure to use an arbitrary number of temporary variables. In a simple allocation scheme, the status of a temporary variable is either *free* or *in-use*. The entry for a temporary variable is initially marked free, it is marked in-use when the variable is allocated, and it is marked free again when the variable is deallocated.

The simple scheme works well when temporary variables are allocated independently. It does not work well when arrays of contiguous temporary variables are allocated. This occurs when temporary variables are allocated to the arguments of a procedure invocation or any invocation conforming to the standard calling conventions; under these circumstances, the argument list is implemented as an array. All of the contiguous temporary variables must be reserved before the first one is used, even though many operations may be performed before the last one is needed. Rather than mark a temporary variable in-use before it actually is, the compiler uses the program point where the temporary variable will be used to mark the temporary variable's entry in the status array as *reserved*. A contiguous array of temporary variables are marked reserved at the same time, with each having a different reservation point. A reserved temporary variable may be allocated to other intermediate values as long as it will be deallocated before the reservation point. In this scheme, an entry in a deallocation list must include the previous status of the temporary variable as it might be a reserved status.

The compiler allocates a contiguous subarray of temporary variables for the arguments of an invocation when it encounters the invocation on the way down the syntax tree during its tree walk. It uses a first-fit algorithm to find a large enough subarray that does not have a conflicting allocation. Consider the problem of allocating temporary variables to the expression

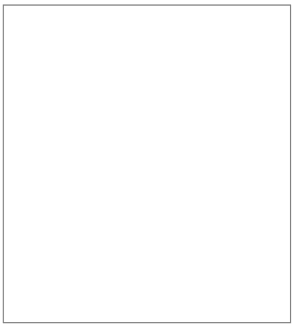
```
f1(f2(f3(x, f4())), y)
```

where f1 can fail and f4 is a generator. The syntax tree for this expression is shown below. Note that invocation nodes show the operation as part of the node label and not as the first operand to general invocation. This reflects the direct invocation optimization that is usually performed on invocations. Each node in the graph is given a numeric label. These labels increase in value in forward execution order.



The following figure shows the operations in forward execution order with lines on the left side of the diagram showing the lifetime of intermediate values. This represents the

output of the liveness analysis phase of the compiler. Because f4 can be resumed by f1, the value of the expression x has a lifetime that extends to the invocation of f1. The extended portion of the lifetime is indicated with a dotted line.

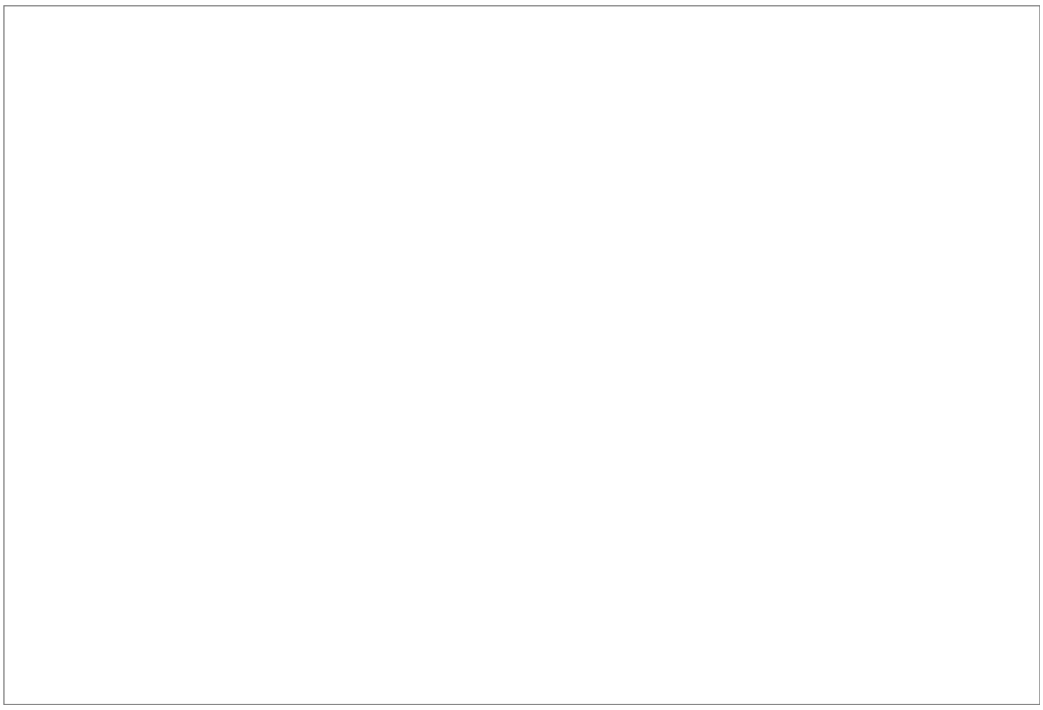


The following series of diagrams illustrate the process of allocating intermediate values. Each diagram includes an annotated syntax tree and a status array for temporary variables. An arrow in the tree shows the current location of the tree walk. A deallocation list is located near the upper right of each node. An element in the list consists of a temporary variable number and the status with which to restore the variable's entry in the status array. If a temporary variable has been allocated to an intermediate value, the variable's number appears near the lower right of the corresponding node.

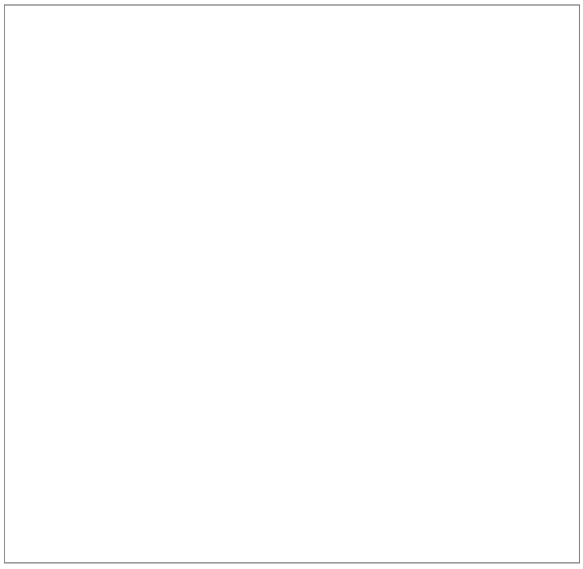
The status array is shown with four elements. The elements are initialized to *F* which indicates that the temporary variables are free. A reserved temporary variable is indicated by placing the node number of the reservation point in the corresponding element. When a temporary variable is actually in use, the corresponding element is set to *I*.

Temporary variables are reserved while walking down the syntax tree. The tree illustrated below on the left shows the state of allocation after temporary variables have been allocated for the operands of f1. Two contiguous variables are needed. All variables are free, so the first-fit algorithm allocates variables 0 and 1. The status array is updated to indicate that these variables are reserved for nodes *4* and *5* respectively, and the nodes are annotated with these variable numbers. The lifetime information in the previous figure indicates that these variables should be deallocated after f1 is executed, so the deallocation array for node *6* is updated.

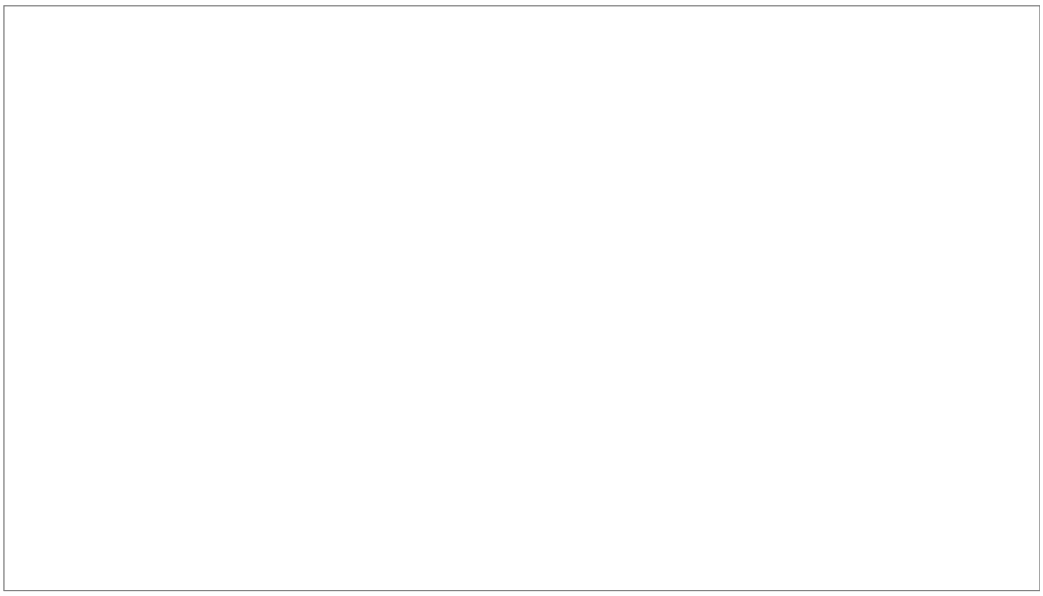
The next step is the allocation of a temporary variable to the operand of f2. The intermediate value has a lifetime extending from node *3* to node *4*. This conflicts with the allocation of variable 0, but not the allocation of variable 1. Therefore, variable 1 is allocated to node *3* and the deallocation list for node *4* is updated. This is illustrated in the tree on the right:



The final allocation requires a contiguous pair of variables for nodes *1* and *2*. The value from node *1* has a lifetime that extends to node *6*, and the value from node *2* has a lifetime that extends to node *3*. The current allocations for variables 0 and 1 conflict with the lifetime of the intermediate value of node *1*, so the variables 2 and 3 are used in this allocation. This is illustrated in the tree:



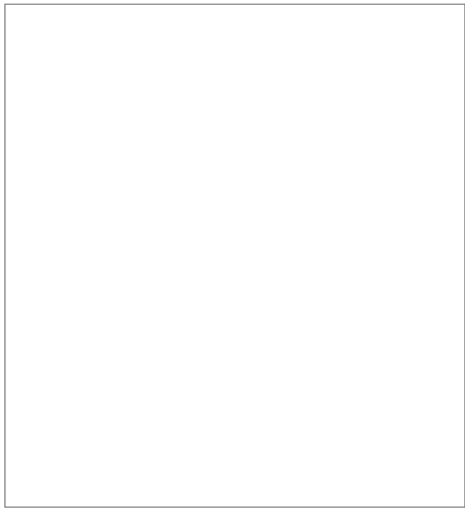
The remaining actions of the allocator in this example mark temporary variables in-use when the code generator uses them and restore previous allocated statuses when temporary variables are deallocated. This is done in the six steps illustrated in the following diagram. The annotations on the graph do not change. Only the node of interest is shown for each step. These steps are performed in node-number order.



In general, the tree walk will alternate up and down the syntax tree. For example, if node *5* had children, the allocation status after the deallocation associated with node *4*,

is used to allocate temporary variables to those children. If this requires more than four temporary variables, the status array is extended with elements initialized to *F*.

This allocation algorithm is not guaranteed to produce an allocation that uses a minimal number of temporary variables. Indeed, a smaller allocation for the previous example is illustrated in the tree:



While the non-optimality of this algorithm is unlikely to have a measurable effect on the performance of any practical program, the problem of finding an efficient optimal solution is of theoretical interest. Classical results in the area of register allocation do not apply. It is possible to allocate a minimum number of registers from expression trees for conventional languages in polynomial time [.dragon.]. The algorithm to do this depends

on the fact that registers (temporary variables) are dead as soon as the value they contain is used. This is not true for Icon temporary variables.

The result of Prabhala and Sethi stating that register allocation is NP-complete even in the presence of an infinite supply of registers also does not apply [.prabhala subexp.]. Their complexity result derives from performing register allocation in the presence of common subexpression elimination (that is, from performing register allocation on expression DAGS rather than trees) on a 2-address-instruction machine with optimality measured as the minimum number of instructions needed to implement the program. Goal-directed evaluation imposes more structure on lifetimes than common subexpression elimination, the machine model used here is the C language, and optimality is being measure as the minimum number of temporary variables needed.

The Icon temporary variable allocation problem is different from the Prolog variable allocation problem. Prolog uses explicit variables whose lifetimes can have arbitrary overlaps even in the absence of goal-directed evaluation. The Prolog allocation problem is equivalent to the classical graph coloring problem which is NP-complete [.debray apr91, dragon.].

If the allocation of a subarray of temporary variables is delayed until the first one is actually needed in the generated code, an optimum allocation results for the preceding example. It is not obvious whether this is true for the general case of expression trees employing goal-directed evaluation. This problem is left for future work.

In addition to holding intermediate values, temporary variables are used as local tended variables within in-line code. This affects the pattern of allocations, but not the underlying allocation technique.

Chapter 21: Control Flow Optimizations

21.1 Naive Code Generation

Naive code generation does not consider the effects and needs of the immediately surrounding program. The result is often a poor use of the target language. Even sophisticated code generation schemes that consider the effects of relatively large pieces of the program still produce poor code at the boundaries between the pieces. This problem is typically solved by adding a *peephole optimizer* to the compiler to improve the generated code [.peep1,Wulf,Tanenbaum peephole,dragon.]. A peephole optimizer looks at several instructions that are adjacent (in terms of execution) and tries to replace the instructions by better, usually fewer, instructions. It typically analyzes a variety of properties of the instructions such as addressing modes and control flow.

The Icon compiler has a peephole optimizer that works on the internal form of the generated C code and deals only with control flow. The previous examples of generated code contain a number of instances of code where control flow can be handled much better. For example, it is possible to entirely eliminate the following code fragment generated for the example explaining procedure suspension.

```
   switch (signal) {
      case A_Resume:
         goto L2 /* bound */;
      }
    L2: /* bound */
```

This code is produced because the code generator does not take into account the fact that the bounding label happens to immediately follow the test.

21.2 Success Continuations

For the C code in the preceding example, it is quite possible that a C compiler would produce machine code that its own peephole optimizer could eliminate. However, it is unlikely that a C compiler would optimize naively generated success continuations. An earlier example of code generation produced the continuation:

```
   static int P02_main()
   {
   register struct PF00_main *rpfp;

   rpfp = (struct PF00_main *)pfp;
   switch (O0o_numeq(2, &(rpfp->tend.d[1]), &trashcan,
(continuation)NULL)) {
      case A_Continue:
         break;
      case A_Resume:
         return A_Resume;
      }
   return 4; /* bound */
   }
```

If the statement

```
   return 4; /* bound */
```

is brought into the switch statement, replacing the break, then P02_main consists of a simple operation call (a C call with associated associated signal handling code). This operation call is

```
   switch (O0o_numeq(2, &(rpfp->tend.d[1]), &trashcan,
(continuation)NULL)) {
      case A_Continue:
         return 4; /* bound */
      case A_Resume:
         return A_Resume;
      }
```

P02_main is called directly in two places in the following code.

```
   frame.tend.d[2].dword = D_Integer;
   frame.tend.d[2].vword.integr = 1;
   switch (P02_main()) {
      case A_Resume:
         goto L2 /* alt */;
      case 4 /* bound */:
         goto L4 /* bound */;
      }
   L2: /* alt */
      frame.tend.d[2].dword = D_Integer;
      frame.tend.d[2].vword.integr = 2;
      switch (P02_main()) {
         case A_Resume:
            goto L5 /* else */;
         case 4 /* bound */:
            goto L4 /* bound */;
         }
   L4: /* bound */
```

A direct call to a trivial function can reasonably be replaced by the body of that function. When this is done for a continuation, it is necessary to *compose* the signal handling code of the body of a continuation with that of the call. This is accomplished by replacing each return statement in the body with the action in the call corresponding to the signal returned. The following table illustrates the signal handling composition for the first call in the code. The resulting code checks the signal from O0o_numeq and performs the final action.

signal from O0o_numeq	signal from P02_main	final action
A_Continue A_Resume	4 A_Resume	goto L4; goto L2;

The result of in-lining P02_main is

```
   frame.tend.d[2].dword = D_Integer;
   frame.tend.d[2].vword.integr = 1;
   switch (O0o_numeq(2, &frame.tend.d[1], &trashcan,
(continuation)NULL)) {
      case A_Continue:
         goto L4 /* bound */;
      case A_Resume:
         goto L2 /* alt */;
     }
   L2: /* alt */
      frame.tend.d[2].dword = D_Integer;
```

```
      frame.tend.d[2].vword.integr = 2;
      switch (O0o_numeq(2, &frame.tend.d[1], &trashcan,
                        (continuation)NULL)) {
         case A_Continue:
            goto L4 /* bound */;
         case A_Resume:
            goto L5 /* else */;
      }
   L4: /* bound */
```

With a little more manipulation, the switch statements can be converted into if statements and the label L2 can be eliminated:

```
   frame.tend.d[2].dword = D_Integer;
   frame.tend.d[2].vword.integr = 1;
   if (O0o_numeq(2, &frame.tend.d[1], &trashcan,
                 (continuation)NULL) == A_Continue)
      goto L4 /* bound */;
   frame.tend.d[2].dword = D_Integer;
   frame.tend.d[2].vword.integr = 2;
   if (O0o_numeq(2, &frame.tend.d[1], &trashcan,
                 (continuation)NULL) == A_Resume)
      goto L5 /* else */;
   L4: /* bound */
```

The Icon compiler's peephole optimizer recognizes two kinds of trivial continuations. The kind illustrated in the previous example consists of a single call with associated signal handling. The other kind simply consists of a single return-signal statement. As in the above example, continuations do not usually meet this definition of triviality until control flow optimizations are performed on them. For this reason, the Icon compiler's peephole optimizer must perform some optimizations that could otherwise be left to the C compiler.

21.3 Iconc's Peephole Optimizer

The peephole optimizer performs the following optimizations:

- elimination of unreachable code
- elimination of gotos immediately preceding their destinations
- collapse of branch chains
- deletion of unused labels
- collapse of trivial call chains (that is, in-lining trivial continuations)
- deletion of unused continuations
- simplification of signal checking

Unreachable code follows a goto or a return, and it continues to the first referenced label or to the end of the function. This optimization may eliminate code that returns signals, thereby reducing the number of signals that must be handled by a continuation call. This provides another reason for performing this traditional optimization in the Icon compiler rather than letting the C compiler do it. This code is eliminated when the fix-up pass for signal handling is being performed. gotos immediately preceding their labels also are eliminated at this time.

Unused labels usually are eliminated when the code is written out, but they may be deleted as part of a segment of unreachable code. Unused continuations are simply not written out.

A branch chain is formed when the destination of a goto is another goto or a return. A break in a switch statement is treated as a goto. There may be several gotos in a chain. Each goto is replaced by the goto or return at the end of the chain. This may leave some labels unreferenced and may leave some of the intermediate gotos unreachable. Branch chains are collapsed during the fix-up pass.

Inter-function optimization is not traditionally considered a peephole optimization. This is because human beings seldom write trivial functions and most code generators do not produce continuations. The Icon compiler, however, uses calls to success continuations as freely as it uses gotos. Therefore collapsing trivial call chains is as important as collapsing branch chains.

There are two kinds of calls to trivial continuations: direct calls and indirect calls through an operation. A direct call always can be replaced by the body of the continuation using signal handling code that is a composition of that in the continuation and that of the call. If the continuation consists of just a return statement, this means that the call is replaced by the action associated with the returned signal: either another return statement or a goto statement. For continuations consisting of a call, the composition is more complicated, as demonstrated by the example given earlier in this chapter.

In the case of an indirect call through an operation, the continuation cannot be placed in line. However, there is an optimization that can be applied. Under some circumstances, the compiler produces a continuation that simply calls another continuation. For example, this occurs when compiling the Icon expression

```
every write(!x | end)
```

The compiler allocates a continuation for the alternation, then compiles the expression !x. The element generation operator suspends, so the compiler allocates a continuation for it and code generation proceeds in this continuation. However, the end of the first alternative has been reached so the only code for this continuation is a call to the continuation for the alternation. The continuation for the alternation contains the code for the invocation of write and for the end of the every control structure. The code for the first alternative is

```
   frame.tend.d[2].dword = D_Var;
   frame.tend.d[2].vword.descptr = &frame.tend.d[0] /* x */;

   switch (O0e_bang(1, &frame.tend.d[2], &frame.tend.d[1],
P02_main)) {
      case A_Resume:
         goto L1 /* alt */;
      }
    L1: /* alt */
```

The code for the two continuations are

```
   static int P02_main()
   {
      switch (P03_main()) {
         case A_Resume:
            return A_Resume;
         }
```

```
   }

static int P03_main()
{
register struct PF00_main *rpfp;

rpfp = (struct PF00_main *)pfp;
F0c_write(1, &rpfp->tend.d[1], &trashcan, (continuation)NULL);
return A_Resume;
}
```

The call to O0e_bang can be optimized by passing the continuation P03_main in place of P02_main.

The optimizations that collapse trivial call chains are performed during the fix-up pass for signal handling.

The final peephole optimization involves simplifying the signal handling code associated with a call. In general, signals are handled with a switch statement containing a case clause for each signal. The C compiler does not know that these signals are the only values that are ever tested by the switch statement, nor is the C compiler likely to notice that some cases simply pass along to the next function down in the call chain the signal that was received. The Icon compiler can use this information to optimize the signal handling beyond the level to which the C compiler is able to optimize it. The optimizer may replace the general form of the switch statement with a switch statement utilizing a default clause or with an if statement. In some cases, the optimizer completely eliminates signal checking. This optimization is done when the code is written.

Chapter 22: Optimizing Invocations

Several optimizations apply to the invocation of procedures and built-in operations. These include optimizations resulting from the application of information from type inferencing, optimizations resulting from the application of lifetime information to passing parameters and returning results, and optimizations involving the generation of in-line code. There are interactions between the optimizations in these three categories.

A primary motivation in developing the Icon compiler was to explore the optimizations that are possible using information from type inferencing. These optimizations involve eliminating type checking and type conversions where type inferencing indicates that they are not needed. Dereferencing is not normally viewed as a type conversion, because variable references are not first-class values in Icon. However, variable references occur as intermediate values and do appear in the type system used by the Icon compiler. Therefore, from the perspective of the compiler, dereferencing is a type conversion.

When a procedure or built-in operation is implemented as a C function conforming to the standard calling conventions of the compiler system, that function is responsible for performing any type checking and type conversions needed by the procedure or operation. For this reason, the checking and conversions can only be eliminated from tailored implementations.

22.1 Invocation of Procedures

As explained earlier, a procedure has one implementation: either a standard implementation or a tailored implementation. If the compiler decides to produce a tailored implementation, the caller of the procedure is responsible for dereferencing. When type inferencing determines that an operand is not a variable reference, no dereferencing code is generated. Suppose p is a procedure that takes one argument and always fails. If P01_p is the tailored C function implementing p, then it takes one argument: a pointer to a descriptor containing the dereferenced Icon argument. Without using type information, the call p(3) translates into

```
frame.tend.d[0].dword = D_Integer;
frame.tend.d[0].vword.integr = 3;
deref(&frame.tend.d[0], &frame.tend.d[0]);
P01_p(&frame.tend.d[0]);
```

With the use of type information, the call to deref is eliminated:

```
frame.tend.d[0].dword = D_Integer;
frame.tend.d[0].vword.integr = 3;
P01_p(&frame.tend.d[0]);
```

22.2 Invocation and In-lining of Built-in Operations

Icon's built-in operations present more opportunities for these optimizations than procedures, because they can contain type checking and conversions beyond dereferencing. Built-in operations are treated differently than procedures. Except for keywords, there is always a C function in the run-time library that implements the operation using the standard calling conventions. In addition, the compiler can create several tailored in-line versions of an operation from the information in the data base.

It is important to keep in mind that there are two levels of in-lining. An in-line version of an operation always involves the type checking and conversions of the operation (although they may be optimized away during the tailoring process). However, detailed code is placed in-line only if it is specified with an inline statement in the run-time system. If the detailed code is specified with a body statement, the ``in-line'' code is a function call to a routine in the run-time library. The difference can be seen by comparing the code produced by compiling the expression ~x to that produced by compiling the expression /x. The definition in the run-time implementation language of cset complement is

```
   operator{1} ~ compl(x)
      if !cnv:tmp_cset(x) then
         runerr(104, x)
      abstract { return cset }
      body {
      ...
      }
    end
```

The conversion to tmp_cset is a conversion to a cset that does not use space in the block region. Instead the cset is constructed in a temporary local buffer. The data base entry for the operation indicates that the argument must be dereferenced. The entry has a C translation of the type conversion code with a call to the support routine, cnv_tcset, to do the actual conversion. cnv_tcset takes three arguments: a buffer, a source descriptor, and a destination descriptor. The entry in the data base has a call to the function O160_compl in place of the body statement. This function takes as arguments the argument and the result location of the operation. The code generator ignores the abstract clause. The in-line code for ~x is

```
      frame.tend.d[3].dword = D_Var;
      frame.tend.d[3].vword.descptr = &frame.tend.d[0] /* x */;
      deref(&frame.tend.d[3], &frame.tend.d[3]);
      if (cnv_tcset(&(frame.cbuf[0]), &(frame.tend.d[3]),
&(frame.tend.d[3])))
         goto L1 /* then: compl */;
      err_msg(104, &(frame.tend.d[3]));
   L1: /* then: compl */
      O160_compl(&(frame.tend.d[3]) , &frame.tend.d[2]);
```

The following is the definition of the `/' operator. Note that both undereferenced and dereferenced versions of the argument are used.

```
operator{0,1} / null(underef x -> dx)
   abstract {
      return type(x)
      }
   if is:null(dx) then
      inline {
         return x;
         }
   else inline {
      fail;
      }
end
```

In this operation, all detailed code is specified with inline statements. The generated code for /x follows. Note that the order of the then and else clauses is reversed to simplify the test. L3 is the failure label of the expression. The return is implemented as an assignment

to the result location, frame.tend.d[2], with execution falling off the end of the in-line code.

```
      frame.tend.d[3].dword = D_Var;
      frame.tend.d[3].vword.descptr = &frame.tend.d[0] /* x */;
      deref(&frame.tend.d[3], &frame.tend.d[4]);
      if (frame.tend.d[4].dword == D_Null)
         goto L2 /* then: null */;
      goto L3 /* bound */;
   L2: /* then: null */
      frame.tend.d[2] = frame.tend.d[3];
```

If type inferencing determines a unique type for x in each of these expressions, the type checking is eliminated from the code. Suppose type inferencing determines that x can only be of type cset in the expression

a := ~x

If parameter passing and assignment optimizations (these are explained below) are combined with the elimination of type checking, the resulting code is

```
   O160_compl(&(frame.tend.d[0] /* x */), &frame.tend.d[1] /* a
*/);
```

The form of this translated code meets the goals of the compiler design for the invocation of a complicated operation: a simple call to a type-specific C function with minimum parameter passing. The implementation language for run-time operations requires that type conversions be specified in the control clause of an if statement. However, some conversions, such as converting a string to a cset, are guaranteed to succeed. If the code generator recognizes one of these conversions, it eliminates the if statement. The only code generated is the conversion and the code to be executed when the conversion succeeds. Suppose type inferencing determines that x in the preceding example can only be a string. Then the generated code for the example is

```
   frame.tend.d[2] = frame.tend.d[0] /* x */;
   cnv_tcset(&(frame.cbuf[0]), &(frame.tend.d[2]),
&(frame.tend.d[2]));
   O160_compl(&(frame.tend.d[2]) , &frame.tend.d[1] / a /);
```

22.3 Heuristic for Deciding to In-line

The in-line code for the operators shown so far in the section is relatively small. However, the untailored in-line code for operations like the element generation operator, !, is large. If tailoring the code does not produce a large reduction in size, it is better to generate a call to the C function in the run-time library that uses the standard calling conventions. A heuristic is needed for deciding when to use in-line code and when to call the standard C function.

A simple heuristic is to use in-line code only when all type checking and conversions can be eliminated. However, this precludes the generation of in-lining code in some important situations. The operator / is used to direct control flow. It should always be used with an operand whose type can vary at run time, and the generated code should always be in-lined. Consider the Icon expression

```
   if /x then x :=
```

The compiler applies parameter-passing optimizations to the sub-expression /x. It also eliminates the return value of the operator, because the value is discarded. An

implementation convention for operations allows the compiler to discard the expression that computes the return value. The convention requires that a return expression of an operation not contain user-visible side effects (storage allocation is an exception to the rule; it is visible, but the language makes no guarantees as to when it will occur). The code for /x is reduced to a simple type check. The code generated for the if expression is

```
      if ((frame.tend.d[0] /* x */).dword == D_Null)
         goto L2 /* bound */;
      goto L3 /* bound */;
   L2: /* bound */
      frame.tend.d[0] /* x */.vword.sptr = ;
      frame.tend.d[0] /* x */.dword = 0;
   L3: /* bound */
```

To accommodate expressions like those in the preceding example, the heuristic used in the compiler is to produce tailored in-line code when that code contains no more than one type check. Only conversions retaining their if statements are counted as a type checks. This simple heuristic produces reasonable code. Future work includes examining more sophisticated heuristics.

22.4 In-lining Success Continuations

Suspension in in-line code provides further opportunity for optimization. In general, suspension is implemented as a call to a success continuation. However, if there is only one call to the continuation, it is better not to put the code in a continuation. The code should be generated at the site of the suspension. Consider the expression

```
every p(1 to 10)
```

The implementation of the to operator is

```
operator{} ... to(from, to)
   /*
    * arguments must be integers.
    */
   if !cnv:C_integer(from) then
      runerr(101, from)
   if !cnv:C_integer(to) then
      runerr(101, to)
   abstract {
      return integer
      }
   inline {
      for ( ; from <= to; ++from) {
         suspend C_integer from;
         }
      fail;
      }
end
```

The arguments are known to be integers, so the tailored version consists of just the code in the inline statement. The for statement is converted to gotos and conditional gotos, so the control flow optimizer can handle it (this conversion is done by rtt before putting the code in the data base). The suspend is translated into code to set the result value and a failure label used for the code of the rest of the bounded expression. This code is generated before the label and consists of a call to the procedure p and the failure introduced by the every expression. The generated code follows. The failure for the every expression is translated into goto L4, where L4 is the failure label introduced by the

suspend. The control flow optimizer removes both the goto and the label. They are retained here to elucidate the code generation process.

```
      frame.tend.d[1].dword = D_Integer;
      frame.tend.d[1].vword.integr = 1;
      frame.tend.d[2].dword = D_Integer;
      frame.tend.d[2].vword.integr = 10;

   L1: /* within: to */
      if (!(frame.tend.d[1].vword.integr <=
frame.tend.d[2].vword.integr) )
         goto L2 /* bound */;
      frame.tend.d[0].vword.integr =
frame.tend.d[1].vword.integr;
      frame.tend.d[0].dword = D_Integer;
      P01_p(&frame.tend.d[0]);
      goto L4 /* end suspend: to */;
   L4: /* end suspend: to */
      ++frame.tend.d[1].vword.integr;
      goto L1 /* within: to */;
   L2: /* bound */
```

This is an example of a generator within an every expression being converted into an in-line loop. Except for the fact that descriptors are being used instead of C integers, this is nearly as good as the C code

```
   for (i = 1; i <= 10; ++i)
      p(i);
```

22.5 Parameter Passing Optimizations

As mentioned above, parameter-passing optimizations are used to improved the generated code. These optimizations involve eliminating unneeded argument computations and eliminating unnecessary copying. These optimizations are applied to tailored in-line code. They must take into account how a parameter is used and whether the corresponding argument value has an extended lifetime.

In some situations, a parameter is not used in the tailored code. There are two common circumstances in which this happens. One is for the first operand of conjunction. The other occurs with a polymorphous operation that has a type-specific optional parameter. If a different type is being operated on, the optional parameter is not referenced in the tailored code. If a tailored operation has an unreferenced parameter and the invocation has a corresponding argument expression, the compiler notes that the expression result is discarded. Earlier in this chapter there are examples of optimizations possible when expression results are discarded. If the corresponding argument is missing, the compiler refrains from supplying a null value for it. Consider the invocation

```
   insert(x, 3)
```

insert takes three arguments. If x is a table, the third argument is used as the entry value and must be supplied in the generated code. In the following generated code, the default value for the third argument is computed into frame.tend.d[2].dword:

```
   frame.tend.d[1].dword = D_Integer;
   frame.tend.d[1].vword.integr = 3;
   frame.tend.d[2].dword = D_Null;
   frame.tend.d[3] = frame.tend.d[0] /* x */;
   F1o0_insert(&(frame.tend.d[2]), &(frame.tend.d[1]),
&(frame.tend.d[3]),
```

```
      &trashcan);
```

Because F1o0_insert uses a tailored calling convention, its arguments can be in a different order from those of the Icon function. It appears that the argument expression x is computed in the wrong place in the execution order. However, this is not true; the expression is not computed at all. If it were, the result would be a variable reference. Instead, the assignment of the value in x to the temporary variable is a form of optimized dereferencing. Therefore, it must be done as part of the operation, not as part of the argument computations. This is explained below.

If the value of x in this expression is a set instead of a table, the entry value is not used. This is illustrated by the following code. Note that a different C function is called for a set than for a table; this is because a different body statement is selected.

```
   frame.tend.d[1].dword = D_Integer;
   frame.tend.d[1].vword.integr = 3;
   frame.tend.d[2] = frame.tend.d[0] /* x */;
   F1o1_insert(&(frame.tend.d[1]) , &(frame.tend.d[2]) ,
&trashcan);
```

In general, an operation must copy its argument to a new descriptor before using it. This is because an operation is allowed to modify the argument. Modification of the original argument location is not safe in the presence of goal-directed evaluation. The operation could be re-executed without recomputing the argument. Therefore, the original value must be available. This is demonstrated with the following expression.

```
   every p(0 to (1 to 3))
```

This is a double loop. The outer to expression is the inner loop, while the inner to expression is the outer loop. to modifies its first argument while counting. However, the first argument to the outer to has an extended lifetime due to the fact that the second argument is a generator. Therefore, this to operator must make a copy of its first argument. The generated code for this every expression is

```
      frame.tend.d[2].dword = D_Integer;
      frame.tend.d[2].vword.integr = 0;
      frame.tend.d[4].dword = D_Integer;
      frame.tend.d[4].vword.integr = 1;
      frame.tend.d[5].dword = D_Integer;
      frame.tend.d[5].vword.integr = 3;
   L1: /* within: to */
      if (!(frame.tend.d[4].vword.integr <=
frame.tend.d[5].vword.integr))
         goto L2 /* bound */;
      frame.tend.d[3].vword.integr =
frame.tend.d[4].vword.integr;
      frame.tend.d[3].dword = D_Integer;
      frame.tend.d[6] = frame.tend.d[2];
   L3: /* within: to */
      if (!(frame.tend.d[6].vword.integr <=
frame.tend.d[3].vword.integr))
         goto L4 /* end suspend: to */;
      frame.tend.d[1].vword.integr =
frame.tend.d[6].vword.integr;
      frame.tend.d[1].dword = D_Integer;
      P01_p(&frame.tend.d[1]);
      ++frame.tend.d[6].vword.integr;
      goto L3 /* within: to */;
   L4: /* end suspend: to */
```

```
      ++frame.tend.d[4].vword.integr;
      goto L1 /* within: to */;
   L2: /* bound */
```

The first argument to the outer to is copied with the statement

```
frame.tend.d[6] = frame.tend.d[2];
```

The copying of the other arguments has been eliminated because of two observations: the second argument of to is never modified and the first argument of the inner to (outer loop) is never reused without being recomputed. This second fact is determined while the lifetime information is being calculated. There is no generator occurring between the computation of the argument and the execution of the operator. Even if there were, it would only necessitate copying if the generator could be resumed after the operator started executing.

As noted above, another set of optimizations involves deferencing named variables. If an operation needs only the dereferenced value of an argument and type inferencing determines that the argument is a specific named variable (recall that each named variable is given a distinct variable reference type), the code generator does not need to generate code to compute the variable reference, because it knows what it is. That is, it does not need the value of the argument. If the argument is a simple identifier, no code at all is generated for the argument.

As shown in the code presented above for

```
insert(x, 3)
```

dereferencing can be implemented as simple assignment rather than a call to the deref function:

```
frame.tend.d[3] = frame.tend.d[0] /* x */;
```

In fact, unless certain conditions interfere, the variable can be used directly as the argument descriptor and no copying is needed. This is reflected in the code generated in a previous example:

```
if /x then ...
```

x is used directly in the in-line code for /:

```
if ((frame.tend.d[0] /* x */).dword == D_Null)
   goto L2 /* bound */;
```

This optimization cannot be performed if the operation modifies the argument, nor can it be performed if the variable's value might change while the operation is executing. Performing the optimization in the presence of the second condition would violate the semantics of argument dereferencing. The compiler does two simple tests to determine if the second condition might be true. If the operation has a side effect, the compiler assumes that the side-effect might involve the named variable. Side effects are explicitly coded in the abstract type computations of the operation. The second test is to see if the argument has an extended lifetime. The compiler assumes that the variable might be changed by another operation during the extended lifetime (that is, while the operation is suspended).

22.6 Assignment Optimizations

The final set of invocation optimizations involves assignments to named variables. These includes simple assignment and augmented assignments. Optimizing these assignments is important and optimizations are possible beyond those that can easily be done working

from the definition in the data base; assignments to named variables are treated as special cases. The optimizations are divided into the cases where the right-hand-side might produce a variable reference and those where it produces a simple Icon value.

There are two cases when the right-hand-side of the assignment evaluates to a variable reference. If the right-hand-side is a named variable, a dereferencing optimization can be used. Consider

```
s := s1
```

This Icon expression is translated into

```
frame.tend.d[0] /* s */ = frame.tend.d[1] /* s1 */;
```

This is the ideal translation of this expression. For other situations, the deref function must be used. For example the expression

```
s := ?x
```

is translated into

```
   if (O0f2_random(&(frame.tend.d[0] /* x */), &frame.tend.d[2])
== A_Resume)
      goto L1 /* bound */;
   deref(&frame.tend.d[2], &frame.tend.d[1] /* s */);
```

When the right-hand-side computes to a simple Icon value, the named variable on the left-hand-side can often be used directly as the result location of the operation. This occurs in the earlier example

```
a := ~x
```

which translates into

```
   O160_compl(&(frame.tend.d[0] /* x */), &frame.tend.d[1] /* a
*/);
```

This optimization is safe as long as setting the result location is the last thing the operation does. If the operation uses the result location as a work area and the variable were used as the result location, the operation might see the premature change to the variable. In this case, a separate result location must be allocated and the Icon assignment implemented as a C assignment. String concatenation is an example of an operation that uses its result location as a work area. The expression

```
s := s1 || s
```

is translated into

```
      if (StrLoc(frame.tend.d[1] /* s1 */) +
StrLen(frame.tend.d[1] /* s1 */)
         == strfree )
         goto L1 /* within: cater */;
      StrLoc(frame.tend.d[2]) = alcstr(StrLoc(frame.tend.d[1] /*
s1 */),
         StrLen(frame.tend.d[1] /* s1 */));
      StrLen(frame.tend.d[2]) = StrLen(frame.tend.d[1] /* s1 */);
      goto L2 /* within: cater */;
   L1: /* within: cater */
      frame.tend.d[2] = frame.tend.d[1] /* s1 */;
   L2: /* within: cater */
      alcstr(StrLoc(frame.tend.d[0] /* s */),
StrLen(frame.tend.d[0] /* s */));
      StrLen(frame.tend.d[2]) = StrLen(frame.tend.d[1] /* s1 */)
+
```

```
      StrLen(frame.tend.d[0] /* s */);

   frame.tend.d[0] /* s */ = frame.tend.d[2];
```

frame.tend.d[2] is the result location. If frame.tend.d[0] (the variable s) were used instead, the code would be wrong.

There are still some optimizations falling under the category covered by this chapter to be explored as future work. For example, as shown earlier,

```
a := ~x
```

is translated into

```
   frame.tend.d[2] = frame.tend.d[0] /* x */;
   cnv_tcset(&(frame.cbuf[0]), &(frame.tend.d[2]),
&(frame.tend.d[2]));
   O160_compl(&(frame.tend.d[2]) , &frame.tend.d[1] /* a */);
```

when x is a string. The assignment to frame.tend.d[2] can be combined with the conversion to produce the code

```
   cnv_tcset(&(frame.cbuf[0]), &(frame.tend.d[0] /* x */),
&(frame.tend.d[2]));
   O160_compl(&(frame.tend.d[2]) , &frame.tend.d[1] /* a */);
```

There is, of course, always room for improvement in code generation for specific cases. However, the optimizations in this chapter combine to produce good code for most expressions. This is reflected in the performance data presented in Chapter 23.

Chapter 23: Performance of Compiled Code

The performance of compiled code is affected by the various optimizations performed by the compiler. This chapter demonstrates the effects of these optimizations on the execution speed of Icon expressions. It also presents speed improvements and memory usage for compiled code versus interpreted code for a set of complete Icon programs. All timing results used in this chapter were obtained on a Sun 4/490 and are the average of the results from three runs.

23.1 Expression Optimizations

The effects of four categories of optimization are demonstrated. These are assignment optimizations, invocation optimizations, control flow optimizations, and optimizations using information from type inferencing. Expression timings for the first three categories were made using techniques described in the August 1990 issue of *The Icon Analyst* [.ianl1.]. The following program skeleton is used to construct the programs to perform these timings.

```
procedure main()
   local x, start, overhead, iters
   iters := 1000000
   start := &time
   every 1 to iters do {
      }
   overhead := &time - start
   x := 0
   start := &time
   every 1 to iters do {
      expression to be timed (may use x)
      }
   write(&time - start - overhead)
end
```

The timings are performed both with and without the desired optimizations, and the results are compared by computing the ratio of the time without optimization to the time with optimization.

The assignment optimizations are described in Chapter 10. The effect of the assignment optimizations on the expression

```
x := &null
```

is measured using the program outlined above. The analysis that produces the assignment optimization is disabled by enabling debugging features in the generated code. The only other effect this has on the assignment expression is to insert code to update the line number of the expression being executed. In this test, the line number code is removed before the C code is compiled, insuring that the assignment optimization is the only thing measured. The timing results for this test produce

Assignment Test
Time in Milliseconds Averaged over Three Runs

Unoptimized	Optimized	Ratio

1122	478	2.3

The tests were performed with type inferencing enabled. Therefore, even the "unoptimized" version of the assignment has the standard operation optimizations applied to it. This test demonstrates the importance of performing the special-case assignment optimizations.

The next category of optimization measured is invocation optimization. This results in the direct invocation of the C functions implementing operations, or in some cases results in the operations being generated in line. The execution time for the expression

```
tab(0)
```

is measured with and without invocation optimizations. As with the assignment optimizations, these optimizations are disabled by enabling debugging features. Once again the line number code is removed before the C code is compiled. These optimizations interact with the optimizations that use information from type inferencing. The measurements are made with type inferencing disabled. Therefore, no type checking simplifications are performed. Without the invocation optimizations, the generated code consists of an indirect invocation through the global variable tab. With the invocation optimizations, the generated code consists of type checking/conversion code for the argument to tab and a call to the function implementing the body statement of tab. The timing results for tab(0) produce

Invocation Test
Time in Milliseconds Averaged over Three Runs

Unoptimized	Optimized	Ratio
8394	4321	1.9

The third category of optimization is control flow optimization. As explained in Chapter 9, these optimizations only perform improvements that a C compiler will not perform when the code contains trivial call chains. One situation that produces trivial call chains is nested alternation. The execution time for the expression

```
every x := ixor(x, 1 | 2 | 3 | 4 | 5)
```

is measured with and without control flow optimizations. The timing results for this every loop produce

Control Flow Test
Time in Milliseconds Averaged over Three Runs

Unoptimized	Optimized	Ratio
6384	4184	1.5

The final category of optimization results from type inferencing. The speed improvements result from generating operations in line, eliminating type checking, and generating success continuations in line. Use of the to operation is a good example of where these optimizations can be applied. This is demonstrated by measuring the speed of an every loop using the to operation. The program that performs the measurement is

```
procedure main()
   local x, start
   start := &time
   every x := 1 to 5000000
   write(&time - start)
```

```
end
```

The timing results for this program produce

```
Type Inference Test
Time in Milliseconds Averaged over Three Runs

Unoptimized                    Optimized  Ratio

9233                           2721       3.3
```

Another approach to determining the effectiveness of type inferencing is to measure how small a set it deduces for the possible types of operands to operations. This indicates whether future work should concentrate on improving type inferencing itself or simply concentrate on using type information more effectively in code generation. A simple measure is used here: the percentage of operands for which type inferencing deduces a unique Icon type. Measurements are made for operands of all operators, except optimized assignment, and for operands of all built-in functions appearing in optimized invocations. For the most part, these are the operations where the code generator can use type information. Measurements were made for a set of 14 programs (described below). Unique operand types within each program range from 63 percent to 100 percent of all operands, with an overall figure for the tests suite of 80 percent (this is a straight unweighted figure obtained by considering all operands in the test suite without regard to what program they belong to); even a perfect type inferencing system will not deduce unique types for 100 percent of all operands, because not all operands have unique types. This suggests that an improved type inferencing system may benefit some programs, but will have only a small overall impact. Future work should give priority to making better use of the type information rather than to increasing the accuracy of type inferencing.

23.2 Program Execution Speed

It has been demonstrated that the compiler optimizations are effective at improving the kinds of expressions they are directed toward. The question remains: How fast is compiled code (with and without optimizations) for complete programs as compared to interpreted code for the same programs? For some expressions, optimizations may interact to create significant cumulative speed improvements. For example, the fully optimized code for the every loop in the previous example is 30 times faster than the interpreted code; the improvement of 3.3 from type inferencing contributes one factor in the total improvement. Other expressions may spend so much time in the run-time system (which is unaffected by compiler optimizations) that no measurable improvements are produced.

A set of 14 programs was selected mostly from contributions to the Icon program library [.tr90-7.] for testing the performance of the compiler. These programs were selected to represent a variety of applications and programming styles (an additional requirement is that they run long enough to obtain good timing results).

The following table shows the speed improvements for the compiled code as compared to interpreted code. The compiler and interpreter used for the measurements both implement Version 8 of Icon. The execution time used to compute the speed improvements is the cpu time measured using the Bourne shell's time command. The first column in the table shows the execution time under the interpreter. The second column is for compiled code with debugging features enabled and optimizations disabled. This code is still better than what would be obtained by just removing the interpreter loop, because intelligent code

generation is performed, especially for bounded expressions, and keywords are generated in line. The third column is for code with debugging features disabled and full optimization enabled.

Execution Time in Seconds Averaged over Three Runs

Program	Interpreter	Compiler Unoptimized	Compiler Optimized
cksol	49.9	33.5 (1.48)	22.5 (2.21)
concord	31.1	18.5 (1.68)	9.8 (3.17)
iidecode	60.3	34.0 (1.77)	12.9 (4.67)
iiencode	50.4	34.4 (1.46)	10.5 (4.80)
impress	44.6	24.8 (1.79)	14.0 (3.18)
list	43.1	24.5 (1.75)	13.6 (3.16)
memfiltr	60.8	34.3 (1.77)	15.3 (3.97)
mf	30.1	18.7 (1.60)	14.7 (2.04)
pssplit	64.0	39.0 (1.64)	26.6 (2.40)
roffcmds	32.9	18.1 (1.81)	12.0 (2.74)
sentence	34.3	23.9 (1.43)	16.2 (2.11)
spandex	36.8	23.3 (1.57)	14.7 (2.50)
textcnt	36.2	18.4 (1.96)	9.9 (3.65)
wrapper	27.3	15.9 (1.71)	9.4 (2.90)

The numbers in parentheses are speed-up factors obtained by dividing the interpreter execution time by the execution time of compiled code.

23.3 Code Size

One advantage the compiler has over the interpreter is that, unless a program is compiled with full string invocation enabled, the executable code for a program need not include the full run-time system. For systems with limited memory, this can be a significant advantage.

The sizes of executable code presented here are obtained from file sizes. All executable files have had debugging information stripped from them. The size of the executable code in the interpreter system is taken to be the size of the interpreter (278,528 bytes) plus the size of the icode for the program being measured (under Unix systems, the size of the executable header, 12,800 bytes for the Sun 4, is subtracted from the size of the icode file, because it is not present during interpretation). Measurements for the 14 test programs are:

Program Sizes in Bytes

Program	Interpreter	Compiler	Ratio
cksol	282,153	81,920	0.29
concord	284,416	90,112	0.31

iidecode	285,525	98,304	0.34
iiencode	283,567	81,920	0.28
impress	295,656	114,688	0.38
list	287,376	98,304	0.34
memfiltr	296,082	114,688	0.38
mf	282,739	81,920	0.28
pssplit	279,709	73,728	0.26
roffcmds	280,797	81,920	0.29
sentence	283,249	81,920	0.28
spandex	281,843	81,920	0.29
textcnt	280,397	73,728	0.26
wrapper	279,780	73,728	0.26

Other factors create differences in memory usage between the interpreter and compiled code. For example, the interpreter allocates a stack for expression evaluation. On the Sun 4, this stack is 40,000 bytes. The compiler, on the other hand, allocates work areas on a per-procedure basis and only allocates the maximum needed at any execution point within the procedure.

Chapter 24: Future Work on the Compiler

24.1 Summary

The underlying ideas used in type inferencing, liveness analysis, and temporary variable allocation were explored using prototype systems before work was started on the compiler described in this dissertation. The fundamental reasons for creating the compiler were to prove that these ideas could be incorporated into a complete and practical compiler for Icon, to explore optimizations that are possible using the information from type inferencing, and to determine how well those optimizations perform. The goal of proving the usefulness of ideas continues a long tradition in the Icon language project and in the SNOBOL language project before it.

The prototype type inferencing system demonstrates that a naive implementation uses too much memory; implementation techniques were developed for the compiler to greatly reduce this memory usage. As the design and implementation of the compiler progressed, other problems presented themselves, both large and small, and solutions were developed to solve them. These problems include how to elegantly produce code either with or without type checking, how to generate good code for simple assignments (a very important kind of expression in most Icon programs), how to generate code that uses the continuation-passing techniques chosen for the compilation model, and how to perform peephole optimizations in the presence of success continuations.

This dissertation describes the problems addressed by the Icon compiler and why they are important to the compiler, along with innovative solutions. It presents a complete set of techniques used to implement the optimizing compiler. Performance measurements demonstrate the improvements brought about by the various optimizations. They also demonstrate that, for most programs, compiled code runs much faster than interpreted code. Previous work has shown that simply eliminating the interpreter loop is not enough to produce large performance improvements [.tr88-31.]. Therefore, the measurements show that the set of techniques, in addition to being complete, is also effective.

24.2 Future Work

The Icon compiler builds upon and adds to a large body of work done previously by the Icon project. There are many problems and ideas relating to the implementation of Icon that remain to be explored in the future. Several are presented in earlier chapters. Others are described in the following list.

- The quality of type inferencing can be improved. For example, if

  ```
  x ||| y
  ```

  is successfully executed, both x and y must contain lists. The current version of type inferencing in the compiler does not use this information; it updates the store based on result types and side effects, but not based on the argument types that must exist for successful execution without run-time error termination. Another improvement is to extend the type system to include constants and thereby perform constant propagation automatically as part of type inferencing. The type system can also be extended to distinguish between values created in allocated

storage and those that are constant and do not reside in allocated storage. A descriptor that never contains values from allocated storage does not need to be reachable by garbage collection.

- In spite of large improvements in the storage requirements of type inferencing over the prototype system, this analysis requires large amounts of memory for some programs. A suggestion by John Kececioglu [.johnk.] is to explore the use of applicative data structures that share structure with their predecessors.

- Type inferencing provides information about values that do not need run-time type information associated with them. In the case of integers and reals, this information along with information from the data base about run-time operations can be used to perform computations on pure C values and to demote Icon descriptor variables to simple C integer and double variables. The current compiler makes little use of these opportunities for optimization. Numerous other optimizations using the information from type inferencing are possible beyond what is currently being done. One of them is to choose the representation of a data structure based on how the data structure is used.

- Translating constant expressions involving integer and real values into the corresponding C expressions would allow the C compiler to perform constant folding on them. For other Icon types, constant folding must be performed by the Icon compiler. This is particularly important for csets, but is not presently being done.

- O'Bagy's prototype compiler performs two kinds of control flow optimizations. It eliminates unnecessary bounding and demotes generators that can not be resumed. The code generation techniques used in this compiler combined with the peephole optimizer automatically eliminate unnecessary bounding. The peephole optimizer also automatically demotes generators that are placed in-line. Enhancements to the peephole optimizer could effect the demotion of generators that are not placed in-line.

- The compiler uses a simple heuristic to decide when to use the in-line version of an operation and when to call the function implementing the operation using the standard calling conventions. More sophisticated heuristics should be explored.

- Temporary variables can retain pointers into allocated storage beyond the time that those pointers are needed. This reduces the effectiveness of garbage collection. Because garbage collection does not know which temporary variables are active and which are not, it retains all values pointed to by temporary variables. This problem can be solved by assigning the null value to temporary variables that are no longer active. However, this incurs significant overhead. The trade off between assigning null values and the reduced effectiveness of garbage collection should be explored.

- The Icon compiler generates C code. If it generated assembly language code, it could make use of machine registers for state variables, such as the procedure frame pointer, and for holding intermediate results. This should result in a significant improvement in performance (at the cost of a less portable compiler and one that must deal with low-level details of code generation).

- Several of the analyses in the compiler rely on having the entire Icon program available. Separate compilation is very useful, but raises problems. On possible solution is to change the analyses to account for incomplete information. They could assume that undeclared variables can be either local or global and possibly initialized to a built-in function or unknown procedures, and that calls to unknown operations can fail, or return or suspend any value and perform any side-effect on any globally accessible variable. This would significantly reduce the effectiveness of some analyses. Another approach is to do incremental analyses, storing partial or tentative results in a data base. This is a much harder approach, but can produce results as good as compiling the program at one time.
- Enhancements to the compiler can be complemented with improvements to the run-time system. One area that can use further exploration is storage management.

Chapter 25: Optimizing the Icon Compiler

This chapter details a set of optimizations that were made to the Icon compiler by Anthony Jones in 1996. Several optimizations are implemented to the type inferencing system and the intermediate code generation with the goals of improving execution time of the generated executable and lower memory requirements.

25.1 Introduction

Compiler optimizations is a difficult but exciting subject. There are a wide variety of ways a compiler could be optimized. There are also different levels that optimizations may be performed on. For example, one level of optimization deals with the front end and intermediate code generation. Some examples of these optimizations include common subexpression elimination, copy propagation, dead-code elimination, constant folding, loop unrolling, and strength reduction.

Another level of optimiztion is machine specific, which might include efficient use of register assignments, using platform specific instructions that offer greater performance, or doing peephole transformations [ASU86]. However, the optimizations proposed for the Icon compiler are not platform specific because of the way the Icon compiler generates code. The Icon compiler's intermediate code is actually C. This means that Iconc translates Icon code into C code which then calls the native C compiler to finish the job.

Another way a compiler can be optimized is by improving the performance of the compiler itself and not the generated code. These optimizations include improving memory usage or making internal data structures more efficient.

The optimizations proposed for the Icon compiler deal exclusively with the front end and intermediate code stages of compilation and improving the performance of the compiler itself.

Specifically, one of the main motivations behind this project was to make the compiler more effective by improving the memory usage for the type inferencing system because the Icon compiler was running out of memory compiling medium-large length programs. The next concern was the intermediate code generation. An examination of the intermediate code provided many areas of improvement. Some of the optimizations possible were eliminating redundant Icon system calls, replacing Icon literals with C literals, and eliminating unnecessary logic in variable initialization blocks.

Areas Where Iconc Can Be Improved

The advantage of iconc's compiled code is that it is many times faster than interpreted code. Unfortunately, Iconc contains some major problems that prevents the compiler from being widely used. The next few sections describe the components of the existing compiler that were optimizated in this project, and each section details the reasons for improvement.

All optimizations were peformed on the Iconc source from Icon Version 9, and the optimized version of the compiler will be referred to as UTSA Iconc.

Type Inference

Variables in Icon are implicitly typed and do not require a declaration of a specific type cite{Walker91a}. All type conversions are implicit in assignments and computations cite{Walker92a}. In order to avoid type checks at run-time, the Icon compiler keeps track of the type of each variable and *infers* the types that each variable may hold. The language has all the ``normal'' types such as integers, floating point numbers, strings, and other common types, but it also has complex structure types such as character sets, lists, tables, and records. The type inferencing model allocates a unique type to each source location at which heterogeneous structure types such as lists or records are created. The Icon compiler represents all the possible types as a bit vector with each bit position representing a specific type. In the course of compiling a large program, the number of total types, and therefore the size of the bit vectors, can skyrocket.

Redundant Function Calls

Icon has a function named `Poll` which is called every so often to handle certain system events such as processing window system events. The current compiler does an inefficient job of placing these function calls in the generated code. Often there will be two calls to `Poll` one right after the other or a simple assignment between two calls. The objective of this part is to remove the redundancy and let a reasonable number of calls remain.

Constant Propagation

Simple literals appearing in Icon source code are assigned into the local variable descriptor table within a procedure. This descriptor table is an array of compilcated structures and pointers that incurs many memory references simply for a constant. It is doubtful that even the most robust C compilers would be able to recognize these values as constants and propagate them accordingly. The objective is to remove assignments of constants into the descriptor table and replace references to those descriptor locations with constant values.

Variable Initialization

At the beginning of every intermediate procedure there are several loops that initialize local variables and parameters. Sometimes these loops initialize only one or two variables. In certain situations the loop will not be executed at all, but the code for the loop is still generated, requiring a comparison when the program executes. The object of this part is to simplify the initialization loops and to remove loops that have no effect.

Changes to the Compiler Source

All the changes made to the Icon compiler in order to implement these optimizations were done with C compiler directives so that each optimization can be turned on or off during compilation.

All directives are included in `src/c/define.h`. The following code turns on all optimizations which are type, redundant functions, literal propagation, and loop optimizations respectively.

```
#ifndef OptimizeType
#define OptimizeType
#endif
```

```
#ifndef OptimizePoll
#define OptimizePoll
#endif

#ifndef OptimizeLit
#define OptimizeLit
#endif

#ifndef OptimizeLoop
#define OptimizeLoop
#define LoopThreshold  6
#endif
```

In the last directive, `LoopThreshold` is declared to have the value of 6. This constant is used in the loop unrolling optimizations and is present so that the user can control this value. It simply is a limit on the number of entries that may be unrolled in variable initialization loops.

25.2 Optimizing the Type Representation

The first area of optimization is the representation of types. Iconc maintains a structure for each variable that contains information about that variable, including a bit vector with each bit representing a particular type used in the program. When a bit vector is allocated it is one of three possible sizes. The first size is composed of first class types which are those built in types plus user defined types that are utilized. The second size consists of the first class types plus intermediate value types. Lastly, there is the number of total types in the database. The database refers to the collection of all builtin operations, their number of parameters and types, and the type for the return value.

Data was gathered from *Ctree*, a circular tree visualization tool. This program consists of ~500 lines of source code. The number of possible first class types is 209 which translates to a 28 byte bit vector. Note that bit vectors are allocated in multiples of a word (4 bytes). During the course of the compilation, 137,946 bit vectors are allocated. The number of first class types plus types for intermediate values is 1,012, resulting in a 128 byte bit vector, and 18,925 vectors of this size are allocated. Lastly, the number of database types is 1,369 types, using a 172 byte bit vector with only 121 allocations of this size. The total memory requirement for the bit vectors is 6.22 megabytes. This information is summarized in Figure 25-1.

Vector Type	*Number of Types*	*Number Allocated*	*Required Memory (MB)*
first class	209	137946	3.8
intermediate class	1012	18925	2.4
database class	1369	121	0.02

Figure 25-1: Bit Vector Sizes

Figure 25-2 is an example of what a bit vector from *Ctree* might look like. This example shows the division between the three type classes. Within the partition for first class types is bit 0 which represents an integer and bit 6 which is a real. Within the intermediate types partition, bit 209 represents an instance of a `cnode` record and an instance of a list variable, and bit 232 is an instance of a variable that is of the list type. Every instance of a

list or an aggregate type such as a record results in a new type that gets its own bit in the bit vector.

Lastly, within the database class are builtin operations. The functions for random number (`O0z7_random`) and subtraction (`O114_subc`) are assigned bits 1,012 and 1,368 respectively. The functions are builtin to the Icon compiler and are assigned their own types in the bit vector.

Figure 25-2: Sample Bit Vector

Additional tests were run on a 25,000 line Icon program called *Freedom in the Galaxy*, which was a semester long effort by Dr. Jeffery's Software Engineering class. The program has hundreds of variables, but in the process of the execution, Icon requires many intermediate variables which dramatically increases the number of bit vectors allocated during compilation. *Freedom in the Galaxy* has 12,591 different distinct types including builtin, intermediate, and database types. This is an example of a program that runs out of memory during compilation.

New Type Representation

The first order of business was to develop a new way to represent type information. The first idea was to utilize the pointers to a type vector. All type vectors are pointers to arrays of integers, and the initial plan was to change a type vector's pointer to be not aligned on a 4 byte boundary in the case that the type vector only represents a simple integer.

Unfortunately, it was discovered that several different locations referenced the same type vector, and any change to one would not be apparent to the other. The second plan which was actually adopted was to create a structure that could contain a packed representation or a pointer to a full length type vector. This allowed multiple variables to reference the same structure which would always be current since only the fields of a structure were being modified. The following structure is the new type vector.

```
struct typinfo {
   unsigned int *bits;
   unsigned int packed;
   };
```

The `bits` field is a pointer to an array of unsigned integers which hold the full type representation of the variable. The `packed` field serves two purposes. First, the lower 24 bits of the integer are reserved for the length of this type vector which corresponds to either the first class, intermediate class, or database class type. This information is required in case a full length vector needs to be allocated. Secondly, the upper 8 bits will contain the packed representation of the type vector. These bits are set by ORing the field with enumerated constants. Figure 25-3 lists the possible values of this field.

Type	Value	Description
NULL_T	1	Null type
REAL_T	2	Real type
INT_T	4	Integer type
CSET_T	8	C Set type

Type	Value	Description
STR_T	16	String type

Figure 25-3: Valid Packed Types

The `typinfo` structure and defined constants for builtin types were added to `csym.h`.

How Type Allocation Works

The new scheme for the type representation is actually rather straightforward. Similar to the old method, a call to `alloc_typ` is made that returns a new type vector. The old method simply returned a pointer of type `unsigned int` to a portion of memory of sufficient size to hold the requested number of types while the new method returns a pointer to `struct typinfo`. This structure contains a packed representation of the type information which holds the most frequently used types such as integers, reals, strings, C sets, and the null value. This requires only an integer which is four bytes. The size of each bit vector is also encoded in this integer as explained earlier. The structure also has the capacity to hold a pointer to a region of memory that can contain an entire type vector in the event that this type vector needs to represent more than the builtin five types. The entire structure occupies only eight bytes.

In reality, `alloc_typ` does not allocate `struct typinfo` structures one at a time. Because an enormous number of these structures are allocated during the compilation of a program, `alloc_typ` allocates a large number of these structures at once. Currently, these structures are allocated in blocks of 400,000. This is done to reduce the overhead that `malloc` requires when allocated blocks of memory. Every time memory is allocated `malloc` needs extra memory (usually around 2-4 bytes) for bookkeeping purposes. As you can see, over 800,000 bytes are saved by allocating this large block of structures. Additionally, `malloc` is generally slow so this change will improve upon compile time.

The five types that were chosen to be represented were integers, reals, strings, csets, and null. This is because the Icon compiler keeps a global variable for each one of these types that specifies which bit position it is kept in for all bit vectors. Other types such as lists or tables were not suitable because the compiler assigns them a unique type and bit position for each occurrence of the variable.

During normal execution, all requests for a type vector return the new type vector with the `packed` field initialized to zero. It is important to note that the null data type is distinct from having no type at all. Through the course of the compilation, the compiler will either call a function to set bits in the vector or check to see if a particular bit is set that corresponds to some type. When the compiler is checking for the presence of a type, the type structure is checked for either a compact or full representation. Once that is known, a simple mask is created to see if the requested type is present. However, the process becomes somewhat more complicated when the compiler requests that a bit is to be set. First, a check is made to determine whether the type structure contains the compact or full type vector. If the requested type is an integer, real, string, character sets or the null value and the type structure uses the compact vector, then the appropriate bit is set in the compact vector. On the other hand, if the requested type is not one of the special five types, a full length vector must be allocated, the compact types must be copied into it, and the new type must also be set. The last possible situation is if the full type vector already exists in the type structure which simply means the requested type can be set without any special actions or additional tests.

In order to accomplish this, several functions that manipulate the type vectors had to be changed to accomodate the new representation. The following sections detail the changes and/or reorganization that was made to the Icon compiler.

Reorganizing the Code

After analyzing the functions that manipulate type information, those functions that inspect, modify, or delete type bits were isolated. These functions required modification so that they could handle the packed type representation. In order to facilitate the understanding of these changes, these functions and macros that manipulate type vectors moved from `typinfer.c` to a new file called `types.c`. The following macros were modified or moved to `types.c`.

```
NumInts(n_bits)
ClrTyp (size, typ)
CpyTyp (nsize, src, dest)
MrgTyp (nsize, src, dest)
ChkMrgTyp(nsize, src, dest)
```

ClrTyp, CpyTyp, MrgTyp, and ChkMrgTyp were modified to handle the compact vectors while NumInts moved for the sake of consistency. The functionality of these macros has not changed.

The following functions were also modifed or moved to `types.c`.

```
struct typinfo *alloc_typ(unsigned int n_types);
novalue set_typ(struct typinfo *type, unsigned int bit);
novalue clr_typ(struct typinfo *type, unsigned int bit);
int has_type(struct typinfo *type, int typcd, int clear);
int other_type(struct typinfo *type, int typcd);
int bitset(struct typinfo *type, int bit);
int is_empty(struct typinfo *type);
novalue bitrange(int typcd, int *frst_bit, int *last_bit);
novalue typecd_bits(int typcd, struct type *type);
```

All the above functions required modification for the new type representation except for `bitrange` and `typecd`.

New Functions

The following functions were added to support the new type representation and were placed in `types.c`. A description of the purpose of each function is provided after the prototypes.

```
unsigned int *alloc_mem_type(int unsigned int n_ntypes)
```

Allocates an actual bit vector large enough to hold `n_types` number of bits. The pointer to the `unsigned int` array is returned.

```
novalue xfer_packed_types(struct typinfo *type)
```

Transfers the types in the packed representation to the full length bit vector in the same `struct typinfo` variable. It assumes that the `bits` field of the `struct typinfo` is valid. The transfer is done by finding the appropriate word in the array where a specific bit is supposed to be and creating a mask that is ANDed to that position in the array.

```
int xfer_packed_to_bits(struct typinfo *src, struct typinfo
*dest, int nsize)
```

Transfers the types in the packed representation from `src` to a full length bit vector, `dest`, of type `struct typinfo` upto a certain type (bit) in the vector represented by `nsize`.

```
novalue and_bits_to_packed(struct typinfo *src, struct typinfo
*dest, int nsize)
```

Performs a bitwise AND on two type vectors. Appropriate measures will be taken for both packed and full type representation.

```
unsigned int get_bit_vector(struct typinfo *src, int pos)
```

Builds a slice (selected word) of a full length bit vector from a compact type form.

```
novalue clr_packed(struct typinfo *src, int nsize)
```

Zeros out the bits of the packed representation.

```
novalue cpy_packed_to_packed(struct typinfo *src, struct typinfo
*dest, int nsize)
```

Copies the packed vector from one variable to the packed representation of another variable. That is, the source variable's types are copied into the destination if the type is within the first `nsize` types.

```
int mrg_packed_to_packed(struct typinfo *src, struct typinfo
*dest, int nsize)
```

Merges two packed vectors into one. This performs a logical AND of all types within the first `nsize` types.

Other Changes

Other significant changes had to accompany the switch over to the new type representation. All pointer variables of type `unsigned int` that referred to a type had to be changed to a pointer to type `struct typinfo`. This included changes in the following compiler source files: `cproto.h`, `csym.h`, `ctree.h`, and `typinfer.c`. Note that this also includes function parameters. Additionally, there were functions that had code embedded in them to manipulate the bits of a type vector manually. In these places, the code required reworking either to call the functions that encapsulated bit manipulations or rewriting in order to take advantage of the compact types. These functions are listed in the following list followed by a list of brief explanations of the modifications.

```
novalue typinfer(void)
```

Allocates a special variable with all the bits on. This required a call to `alloc_mem_type` in order to allocate a full length type vector. All the bits were then set to on.

```
struct store *alloc_stor(int store_sz, int n_types)
```

Allocates a store which includes type information. This required changing the `alloc` call to allocate `struct typinfo` instead of `unsigned int`.

```
struct symtyps *symtyps(int n_syms)
```

Allocates symbol tables. This also required changing the `alloc` call to allocate `struct typinfo` instead of `unsigned int`.

```
novalue typ_deref(struct typinfo *src, struct typinfo *dest, int
chk)
```

Before the type dereferencing is performed the `src` was merged with the `dest` parameters. This required checking for packed or full type vectors and handling them appropriately. Also, if the boundary between first class and intermediate types falls in the middle of a word, those intermediate types on the boundary word are zeroed out.

```
novalue abstr_typ(struct il_code *il, struct type *typ)
```

In one of the cases of a `switch` statement two type vectors are ANDed together. This requires placing a function call to `and_bits_to_packed` in place of the existing code.

```
int eval_cnv(int type_cd, int index, int def, int *cnv_flags)
```

This function determines if a type conversion on a type will succeed. To do this, a type vector is ANDed with several different bit masks. This required checking for packed or full bit vectors and handling them appropriately. In the case of a packed vector, the function `get_vector` is called to build a word with the appropriate type bits set if they fall in the selected word of the type vector.

```
struct argtyps *get_argtyp(void)
```

Allocates an argument list. This required changing the `alloc` call to allocate `struct typinfo` instead of `unsigned int`.

Results of Type Optimization

After the new type representation was implemented, tests were run again on *Ctree*. The results showed a dramatic decrease in the required amount of memory necessary for compilation. UTSA Iconc required one third the amount of memory of the old compiler. The program *Freedom in the Galaxy* even compiled under this optimization. Although *Freedom in the Galaxy* still needed a substantial amount of memory, the important fact is that it compiled. Section 25.4 provides detailed results of the memory usages for both *Ctree* and *Freedom in the Galaxy*.

25.3 Optimizing the Generated Code

The other area of optimization is the efficiency of the C code generated by the Icon compiler. The optimizations undertaken were to remove redundant calls to system functions, constant propagation, and variable initialization. These optimizations were obvious from a cursory examination of the C code generated. The goals of these optimizations are to make the intermediate code as small as possible and to speed up the resulting executable. First a brief summary of the internal representation of the C code is provided. This is necessary because most of these optimizations rely heavily on analyzing the internal C code. Following this, the individual optimizations are discussed in detail.

Intermediate Code Representation

This section briefly describes how the intermediate C code is represented and generated internally by the Icon compiler. The majority of the functions that generate this internal representation and print it to a file are contained in the following compiler source files: `ccode.c`, `codegen.c`, and `inline.c`.

How Code is Generated

Once the source code is parsed and evaluated, the intermediate C code needs to be generated and output to a file for compilation by the native C compiler. First the compiler

builds a syntax tree plus symbol tables and other necessary structures. Then, the header file is created. This includes standard definitions necessary for all Icon programs and structures and variables specific to the program being compiled. Next, the `proccode` function is called for each function in the tree. This outputs the function definition and variable in initialization code, and then steps through the syntax tree and creates C code to represent the body of the function. After all the code for the body of the current procedure is generated internally, the code is then written to the file.

The internal C code is represented through a C structure called `struct code` which is shown below.

```
struct code {
   int cd_id;
   struct code *next;
   struct code *prev;
   union  cd_fld  fld[1];
   };
```

The `cd_id` field is an identifier signaling what type of code is held in this structure. This field may be set to one of the following enumerated values. Each value corresponds to a type of code that can be written to the intermediate C code. The table in Figure 25-4 contains the enumerated name along with its integer value and a short description.

Code Type	*Value*	*Description*
`C_Null`	0	No code in this struct
`C_CallSig`	1	Call a signal (function)
C_RetSig	2	Return a signal
C_NamedVar	3	Reference a variable
C_Goto	4	Goto statement
C_Label	5	Label statement
C_Lit	6	Literal value
C_Resume	7	Resume signal
C_Continue	8	Continue signal
C_FallThru	9	Fall through signal
C_PFail	10	Procedure failure
C_PRet	11	Procedure return
C_PSusp	12	Procedure suspend
C_Break	13	Break out of signal handling switch
C_LBrack	14	Start of a new C block
C_RBrack	15	End of a C block
C_Create	16	Call create() for a create expression

Code Type	*Value*	*Description*
C_If	17	If statement
C_SrcLoc	18	Source file name
C_CdAry	19	Array of code pieces

Figure 25-4: Code Types

The `fld` field is important and is directly linked to what type of code the `struct code` is defined as. For example, if a `struct code` is defined as `C_If` then `fld[0]` is a pointer to another `struct code` that corresponds to the if portion of the statement, and `fld[1]` is another pointer to a `struct code` representing the then portion of the statement. In fact there are two macros for extracting each pointer. These macros plus macros for all the other code types are found in the `ccode.h` header file. However, there is one special case that requires some explanation. If the `cd_id` is `C_CdAry` then the `fld` element is an unspecified length of `cd_fld` unions. In this case all even indices into the array are tags describing the contents of the following array element. There is a special marker, `A_End`, that signifies the end of the array. Figure~\ref{fig:struct_code_assign_full} shows these field identifiers along with their corresponding fields for an assignment statement. It is important to note that only when the `cd_id` is `C_CdAry` will the field identifiers be present. Figure 25-5 gives the possible values for these tags.

Element Type	*Value*	*Description*
A_Str	0	Pointer to a string
A_ValLoc	1	Pointer to a struct val_loc
A_Intgr	2	Integer value
A_ProcCont	3	Procedure continuation
A_SBuf	4	String buffer
A_CBuf	5	Cset buffer
A_Ary	6	Pointer to a subarray of struct code structures
A_End	7	Marker for end of array

Figure 25-5: Element Types

For the most part the `C_CdAry` is used for miscellaneous code that is not covered by the other 19 code types. Most simple assignments fall into this category. The last two elements of a `struct code`, `next` and `prev`, are links to the next and previous `struct code` structures in the chain.

Figure 25-6: Literal

Redundant Function Calls

An example of this type of optimization are several function calls needed to handle certain run-time system activities in Icon that are included in the generated C code. For example, throughout the code Icon places a call to the function `Poll` which checks for pending events such as window redraws. In some cases there is a call to `Poll` followed by an assignment and another call to `Poll` which is far too frequent. The placement of these function calls can be analyzed to determine when they are necessary.

Analyzing Function Call Placement

The solution to this problem entails analyzing where the calls to `Poll` were being placed. The `Poll` function is inserted into the generated code by the function `setloc` which is located in the file `ccode.c` of the compiler source. The old method for determining when to insert a call to this function is somewhat confusing. Also `setloc` does more than insert these function calls so there was no change in the way it determined when to put a call in. Instead, a call to `analyze_poll` is made that determines if it is safe to remove the previous occurrence of the `Poll` function. To accomplish this, a global variable is kept, called `lastpoll`, which is a pointer of type `struct code`, and it is always assigned to the location of the last `Poll` function. Of course, initially `lastpoll` is `NULL`. The global variable is declared in `ccode.c`. The prototypes for the two new functions are as follows:

```
int analyze_poll(void)
```

This function analyzes the code between the last occurence of the `Poll` function and the current position. If there are no function calls (`C_CallSig`), return signals (`C_RetSig`), C code blocks (`C_LBrack` or `C_RBrack`), return calls (`C_PRet`), procedure suspends (`C_PSusp`), or break (`C_Break`), then the previous instance of `Poll` will be removed; otherwise, it will be left in place.

The reason why the above code types are restricted is because they all involve calling other functions. If it were known that these functions were short and did not call other functions, then the call to `Poll` could be removed without worry; however, this kind of detailed analysis is not performed and is inhibited by the fact that some of these functions represented by `C_CallSig` may be library functions and these are linked at C compile time.

Also, regardless of whether the previous instance of a call to `Poll` is removed the new call to `Poll` is added to the code list and the `lastpoll` variable is updated.

```
novalue remove_poll(void)
```

This function actually removes the call to `Poll` by setting the `cd_id` field in the `struct code` structure to `C_Null`. It is important to note that the `struct code` that represents the call to `Poll` is not physically deallocated from the list. It's `cd_id` field is simply set to `C_Null` because removing it introduces side effects which are either errors during C compilation or the misplacement of `goto` labels which affects the flow of execution and unpredictable results. This occurs because a `struct code` of type `C_Goto` may reference the removed node.

Icon Literals and Constant Propagation

Constant progagation was the second most difficult optimization to implement next to the new type representation because the Icon compiler generates a complex data structure that

contains Icon values, including literals. These Icon literals are assigned into this tended descriptor table even though these values are constants. There are several reasons to improve the representation of these constants.

First, by changing these complicated Icon literals to simple C literals, the resulting executable code will be smaller. Secondly, there is the issue of constant propagation. In many cases, an index into the descriptor table is passed to a function or assigned to a variable. The question that arises is whether the C compiler can detect that the descriptor table value being passed is a constant that can be propagated to all places where the descriptor table is used. For example, the following code fragment is fairly common:

```
r_frame.tend.d[4].dword = D_Integer;
r_frame.tend.d[4].vword.integr = 1;
irslt = sub(argp[0].vword.integr,
            r_frame.tend.d[4].vword.integr);
```

In this section of code, the structure `r_frame.tend.d[4].vword.integr` is assigned a value and then immediately used. This code can be simplified to:

```
irslt = sub(argp[0].vword.integr, 1);
```

Note that the assignment of the literal into the descriptor table may no longer be necessary; time savings on this initialization may be as great as the savings for the simplified reference.

Tended Descriptor Tables

Most functions contain a tended descriptor table. This is an array of descriptor structures which contain either an integer, pointer to a string, pointer to a block, or a pointer to another descriptor location. A named variable is assigned a specific index into the descriptor table while temporary variables are assigned an index, but other temporary variables can be assigned into the same cell many times over. Named variables are all those that are explicitly used in the Icon source code such as loop control variables, and temporary variables are constants values (regardless of type). For example, in the first Icon code example the value 2.4 is assigned its own location into the descriptor table. The same thing holds true for the second example. The string `"foo"` is assigned its own location. Because both these values are only literals in the Icon code, they are given temporary locations in the tended desciptor table that may be used over again.

```
if (x_val = 2.4) then
   do_something(x_val)
...
...
if (str_val == "foo") then
   do_something(str_val)
```

For example, if the constant `2.4` is not used after the second code fragment then `"foo"` may be assigned into the location previously occupied by `2.4`.

Analyzing Literal Assignments

Several new functions were introduced in order to analyze all constants and their use. Inside the function `proccode` before the internal C code is written to a file, a call to `analyze_literals` and `propagate_literals` is made which does the propagation. The `analyze_literals` function builds a table which contains information such as the scope of a descriptor entry, whether it is safe to propagate a literal, and the literal value. The table structure is given below.

```
struct lit_tbl {
int    modified;
int    index;
int    safe;
struct code    *initial;
struct code    *end;
struct val_loc *vloc;
struct centry  *csym;
struct lit_tbl *prev;
struct lit_tbl *next;
};
```

The field `modified` is a flag which can be set to one of the enumerated types in Figure 25-7.

Name	*Value*	*Description*
`NO_LIMIT`	0	Descriptor never changes
`LIMITED`	1	Descriptor value does change, propagate any type
`LIMITED_TO_INT`	2	Descriptor value does change, propagate only if integer
`NO_TOUCH`	3	Descriptor value should not be propagated

Figure 25-7: Modify Flags

The `NO_LIMIT` value refers to those descriptor locations that always contain the same constant. That is, no other value shares the same descriptor location, and it may be propagated freely without conflicts. The `LIMITED` value refers to those descriptor locations that are either reused at some point or are modified is some way. The value `LIMITED_TO_INT` is similar except that special care must be taken when propagating this constant. For example, a constant such as a string should not be propagated everywhere an interger may be propagated.

Lastly, the value `NO_TOUCH` refers to descriptor locations that should not be propagated. These descriptor locations often contain loop control variables which are marked as temporary but should under no circumstances be replaced with their initial values. For example, the first code fragment shows unoptimized code, and the second fragment is the same code but with constants propagated. Descriptor location 6 should not be touched because it serves as a loop control variable while the use of location 7 may be replaced with its constant value 10 even though the same location is assigned a new value later on after label `L9`.

```
   r_frame.tend.d[6].dword = D_Integer;
   r_frame.tend.d[6].vword.integr = 1;
   r_frame.tend.d[7].dword = D_Integer;
   r_frame.tend.d[7].vword.integr = 10;
L8:
   if (!(r_frame.tend.d[6].vword.integr <=
            r_frame.tend.d[7].vword.integr) )
     goto L9;
   ...
   ++r_frame.tend.d[6].vword.integr;
   goto L8;
L9:
   r_frame.tend.d[7].dword = D_Integer;
```

```
   r_frame.tend.d[7].vword.integr = 7;
```

```
   r_frame.tend.d[6].dword = D_Integer;
   r_frame.tend.d[6].vword.integr = 1;
L8:
   if (!(r_frame.tend.d[6].vword.integr$<=$ 10) )
      goto L9;
   ...
   ++r_frame.tend.d[6].vword.integr;
   goto L8;
L9:
   r_frame.tend.d[7].dword = D_Integer;
   r_frame.tend.d[7].vword.integr = 7;
```

The field `index` is the index into the descriptor table for each constant.

The field `safe` refers to whether or not it is safe to modify the `end` field. This field refers to the point in the intermediate code beyond which it is no longer safe to propagate this value. The `end` field is sometimes modified when inserting a new entry into the literal table. This is described in detail under the `tbl_add` function presented shortly.

The fields `initial` and `end` refer to the scope where it is safe to propagate the current literal between. If `end` is `NULL` then it is safe to propagate to the end of the function.

The fields `vloc` and `csym` are pointers to either a `struct val_loc` or a `struct centry` which contain the constant value of the current descriptor. The `struct centry` member points to the corresponding location in the global symbol table of constant values maintained by the Icon compiler.

The fields `prev` and `next` are necessary to make the table doubly linked.

Also, it should be noted that the number of entries in the literal table is fairly small. During compilation of *Ctree*, the largest literal table used contained 15 entries.

The analysis phase consists of stepping through the `struct code` chain for each function looking for each instance of a literal. Figure 25-8 shows how a literal is contained within a `struct code` structure. At this point, a new entry into the literal table is created that keeps track of where in the code the literal is assigned into the descriptor table and a pointer to the `struct centry` structure where the literal value is kept. This phase also attempts to find the point at which descriptor entries are assigned new values. Thus a scope is defined which the constant may only be propagated between.

Figure 25-8: Literal

Once the analysis is complete and the literal table is built then the function `propagate_literals` is called which goes through each entry in the literal table and examines the code beginning at the `initial` field until the `struct code` referenced by the `end` field is encountered. If a `struct code` is found that references the descriptor containing the current literal then that reference is replaced by the literal itself. Figure 25-6 illustrates a fragment of code that does an assignment, and Figure 25-9 shows the same fragment with the second descriptor replaced with its literal (assuming that descriptor location 8 was previously initialized to 27). It is important to note that only the `struct val_loc` on the right side of the equal sign will be replaced by its literal.

Figure 25-9: Assignment

New Functions

The following functions were created to support the contant propagation optimization. All these functions are placed in the compiler source file `ccode.c`. Each function used in the constant propagation is prototyped and described below.

```
struct lit_tbl *alc_tbl(void)
```

This function allocates a `struct lit_tbl` entry that contains information about a literal and its usage. It first checks a global pointer called `free_lit_tbl` to see if there are any free table structures that may be reused. If there are no free structures in this list then a new structure is allocated. Lastly, the fields are initialized to predefined values.

```
novalue free_tbl(void)
```

This function frees the memory used for the current table by attaching the current table to the list of free table structures (free_lit_tbl).

```
novalue tbl_add(struct lit_tbl *add)
```

This function adds a new struct lit_tbl structure into the current table. The insertion is to the end of the table plus it checks for the previous use of the descriptor location used in the element being added. For the previous use of the same element, that location's `end` pointer is set to the initial pointer of the element being added. In essence, this defines a scope for each descriptor location. Once `end` is set for the first time, it should not be changed later.

```
int substr(const char *str, const char *sub)
```

This function is used to scan strings for logical operators (==, !=, >=, <=, etc). If the string represented by `sub` is found in `str` then `TRUE` is returned. It is necessary to identify these operators so a string is not propagated as an operand to one of these operators which is not valid C syntax.

```
int instr(const char *str, int chr)
```

This function is used to determine if a string contains an assignment operator. This function will return `TRUE` if the string `str` contains any type of assignment (=, +=, -=, *=, /=, or %=).

```
novalue invalidate(struct val_loc *v, struct code *end, int code)
```

This function sets values for an element in the literal table. For all literal table entries that point to the `struct val_loc` represented by `v` the `end` field is set to `end` with the `modified` field set to `code`. `code` can be one of the following enumerated values: `NO_LIMIT`, `LIMITED`, `LIMITED_TO_INT`, or `NO_TOUCH`.

```
novalue analyze_literals(struct code *start, struct code *top,
int lvl)
```

This function steps through the `struct code` list for each function, building up a literal table, and analyzing the scope between which each literal can be safely propagated. It checks for loop control variables, when and if the value of a constant descriptor location changes, and checks to see if a descriptor location is passed by reference to any functions.

```
novalue propagate_literals(void)
```

This function steps through each entry in the literal table and begins to replace occurences of the descriptor location with the literal between the `struct code` structures from the `initial` field to the `end` field. The function `eval_code` is called to do the actual propagation.

```
int eval_code(struct code *cd, struct lit_tbl *cur)
```

This function first checks to see if the descriptor index of the code currently being examined matches that of the current literal table entry. If the current descriptor is accessed as an integer or a string then the descriptor is replaced with the literal value. Also, the `modified` is checked to see if there are any restrictions on replacement. The table in Figure 25-10 lists the restrictions for each possible value of `modified`.

Name	*Replacement Restrictions*
`NO_LIMIT`	Always replace
`LIMITED`	Always replace within `initial` and `end`
`LIMITED_TO_INT`	Only replace if used as int, also limited by scope
`NO_TOUCH`	Never replace

Figure 25-10: Replacement Policy

The actual replacement of a descriptor reference to a literal is accomplished by setting the current index into the `fld` array to the `A_Str` type and allocating a string where the literal is copied into. Figure 25-6 and Figure 25-9 illustrate an occurrence of this.

Variable Initialization

Another issue is the initialization of the descriptor tables in each C function that is generated by the Icon compiler. Many of the generated functions contain a loop that initializes all the entries of the local descriptor table to the null descriptor. This is rather cumbersome and generates a great deal of overhead.

Eliminating Dead Code

The first optimization to the variable initialization was to eliminate ``dead'' code, which is code that is never executed. In some cases the loops that initialize the descriptor tables resembled this:

```
for (i = 0; i < 0; ++i)
   r_frame.tend.d[i] = nulldesc;
```

This code is generated for Icon library functions in the function `outerfnc` located in `codegen.c`. There is a separate function that outputs similar code for user written functions which does check to see if the loop will ever execute. Both functions contain a variable `ntend` which hold the number of descriptor entries. A simple check for equality with zero was added.

Loop Unrolling

Every user function initializes all tended descriptor entries to the value of the null descriptor `nulldesc` at the beginning of the function. It is a simple one-line `for` loop similar to the following code fragment.

```
for (i = 0; i < 3; ++i)
   r_frame.tend.d[i] = nulldesc;
```

Also, upon examining the C code generated from several programs, the number of descriptor entries per procedure rarely exceeds ten. Because this is a relatively small number, these loops can be unrolled into a series of assignments, and the loop may be removed. The following code is the above loop unrolled.

```
r_frame.tend.d[0] = nulldesc;
r_frame.tend.d[1] = nulldesc;
r_frame.tend.d[2] = nulldesc;
```

While this will increase the size of the generated code, the loop overhead is eliminated. There is a limit placed on the number of loop iterations that will be unrolled which is defined in `define.h`. Currently, this value, `LoopThreshold`, is set to 6. Because this number and the number of descriptor table entries is small, the number of unrolled elements is reasonable, and the code size is not greatly affected.

The code that unrolls these loops is in the function `outerfnc` in the file `codegen.c`. Because this change is only several lines, the code that implements loop unrolling is included below.

```
#ifdef OptimizeLoop
if (ntend > 0)           /* Check for dead code */
   for (i=0; i < ntend ;i++)
      fprintf(codefile,
              "   r_frame.tend.d[%d] = nulldesc; \n", i);
#else
fprintf(codefile, "for (i=0; i < %d ;i++) \n", ntend);
fprintf(codefile, "   f_frame.tend.d[i] = nulldesc;\n;");
#endif
```

Results of Code Generation Optimizations

Several tests were run to determine whether the code generation optimizations were effective. These optimizations were performed in hopes of improving the execution speed of the compiled program, reducing the size of the intermediate code and the resulting executable, and the compilation time. A brief description of the results follows; however, a more detailed analysis of the optimizations is given in Chapter 4.

Overall, the optimizations improved the execution speed by a modest amount. The improvement is roughly between 6-8.25%. While this is not a great as was hoped, it still is an improvement. The code size of both the intermediate code and the generated executable are suprisingly smaller. The loop unrolling seemed to be offset by the constant progagation which eliminated unnecessary assigments and references. The size of the executables were reduced by approximately 4-8% for large programs, but there was no change in executable size for small programs (20 lines). The size of the generated C file was consistently around 3% smaller than before the optimizations.

Also, on average around half of all calls to `Poll` were removed, and in one case, two thirds were eliminated. The largest improvement was to compilation time. The optimizations improved compile time by 24-31% on large programs and 13% on small programs; however, it should be noted that all of the tests for this section were performed with all the optimizations on, including the type representation optimization.

25.4 Results

This chapter presents detailed information on the results of each optimization discussed in this paper. The first section discusses the improvements in memory usage resulting from the type representation optimizations while the second sections presents the results from removing redundant function calls, unrolling loops, removing dead code, and propagating literals.

Type Representation

Tests were performed on *Ctree* and *Freedom in the Galaxy* to determine the new memory requirements of UTSA Iconc. These tests were run with only the type representation optimizations and no other optimizations that were covered in Chapter 3. The results show a substantial decrease in the memory required to compile the program. For *Ctree*, there were 156,992 packed type structures allocated which is the total number of all type vectors from the first test. Once the packed structure was allocated only 11,653 needed an actual first class vector allocated. Of the intermediate class, only 5,172 full-size vectors needed to be allocated. However, all 121 of the database class variables needed the full sized vector. Overall, the total memory usage for type representation is 2.17 megabytes which is 35% of the memory required by the old type representation. The results are summarized in Figure 25-11.

Vector Type	Number of Types	Number Allocated	Required Memory (MB)
packed class	5	156992	1.25
first class	209	11653	0.3
intermediate class	1012	5172	0.6
database class	1369	121	0.02

Figure 25-11: Memory Usage (*Ctree*)

Unfortunately, the improvement in memory usage was not great enough for *Freedom in the Galaxy* to compile on the same machine that the tests on *Ctree* were run; however, the program did compile on a Sparc 10 with 128MB of memory with no one else logged on at the time. The Figure 25-12 contains the memory requirements for each of the classes of vectors.

Vector Type	Number of Types	Number Allocated	Required Memory (MB)
packed class	5	4294822	34.36
first class	1425	119468	21.5
intermediate class	8508	24349	25.91
database class	12591	7	0.01

Figure 25-12: Memory Usage (*Freedom in the Galaxy*)

Even with the optimization, *Freedom in the Galaxy* requires over 81 megabytes of memory for the type inferencing alone. Because Freedom in the Galaxy could not be compiled before the type optimization, there are no numbers to compare these with. However, considering that the type optimization reduce the memory requirements for Ctree by one third, then a good estimate for the memory requirements would be around 240 megabytes!

While the new type representation drastically reduces the amount of memory used during compilation, it still uses too much memory to be of use when compiling large programs on anything but an expensive workstation with a substantial amount of memory. However, UTSA Iconc still offers the user the advantage of compiled code, and the new type representation makes UTSA Iconc practical on many programs that could not be compiled because of memory requirements of the old Icon compiler.

Code Generation

This section details the results of the code generation optimizations in the area of execution speed, code size, and compilation time. These tests were run on several programs. The first program, *Beards*, generates production grammars, non-terminals, terminals, and epsilon sets from an input grammar. The second program, *Yhcheng*, is a line editor similar to `ed` which also has revision control capabilities. For the code size and compilation time tests, two other programs, *Ctree* and *Sphere*, were used for tests. *Beards*, *Yhcheng*, and *Ctree* are all large programs while *Sphere* is included because it is a very small program (less than 25 lines). All timings performed used the Unix or Linux `time` utility. Also note that these timings were performed with all optimizations turned on including the type representation optimization.

Execution Speed

Each program was run 10 times with sample input and averages were computed. Figure 25-13 summarizes the execution times for *Beards* and *Yhcheng*.

Program	*Version*	*User*	*System*	*Elapsed*
	Optimized	0.5	0.12	00:01.17
Beards	Unoptimized	0.52	0.13	00:01.27
	Improvement	4.97%	9.09%	8.25%
	Optimized	0.59	1.11	00:01.99
Yhcheng	Unoptimized	0.62	1.27	00:02.14
	Improvement	4.21%	12.91%	6.79%

Figure 25-13: Execution Times

Code Size

Tests were run on the same two programs to determine if there was an improvement in either the intermediate code size or the size of the resulting executable. Figure 25-14 displays the code sizes for *Beards*, *Yhcheng*, *Ctree*, and *Sphere*. The first three programs are large (500-1800 lines) while *Sphere* is small (20 lines).

Program	*Version*	*C File*	*H File*	*Executable*
	Optimized	246159	12967	204800
Beards	Unoptimized	252041	12967	212992
	Improvement	*2.33%*	*0.00%*	*3.85%*

Program	*Version*	*C File*	*H File*	*Executable*
	Optimized	554014	46168	294912
Yhcheng	Unoptimized	568118	46168	319488
	Improvement	*2.48%*	*0.00%*	*7.70%*
	Optimized	290536	61545	225280
Ctree	Unoptimized	298813	61545	237568
	Improvement	*2.77%*	*0.00%*	*5.17%*
	Optimized	82289	49755	159744
Sphere	Unoptimized	84972	49755	159744
	Improvement	*3.16%*	*0.00%*	*0.00%*

Figure 25-14: Code Sizes

Much of the reduction in code size can be attributed to the removal of redundant calls to `Poll`, and it is this reduction that offsets the loop unrolling. Improvements on *Beards*, *Yhcheng*, and *Sphere* show that almost one half of all calls to `Poll` were eliminated; however, *Ctree* shows almost a two thirds reduction. Figure 25-15 shows the number of calls to `Poll` for each program before and after the optimization.

Test Program	No. Before	No. After
Beards	810	481
Yhcheng	2144	1135
Ctree	745	293
Sphere	40	22

Figure 25-15: Number of Redundant Functions Removed

Compilation Time

Lastly, the compilation times for the sample programs are given. Each program was compiled five times with the results averaged. Again, results for the *Beards*, *Yhcheng*, *Ctree*, and *Sphere* are in Figure 25-16.

Program	*Version*	*User*	*System*	*Elapsed*
	Optimized	43.57	1.77	00:47.40
Beards	Unoptimized	60.93	1.65	01:02.93
	Improvement	*28.49%*	*-7.27%*	*24.68%*
	Optimized	116.97	2.76	02:04.14
Yhcheng	Unoptimized	163.37	2.86	02:49.71

Program	*Version*	*User*	*System*	*Elapsed*
	Improvement	*28.40%*	*3.50%*	*26.85%*
	Optimized	65.26	2.54	01:13.44
Ctree	Unoptimized	92.25	2.88	01:47.44
	Improvement	*29.26%*	*11.81%*	*31.65%*
	Optimized	11.98	1.83	00:16.36
Sphere	Unoptimized	13.62	2.22	00:18.85
	Improvement	*12.04%*	*17.57%*	*13.21%*

Figure 25-16: Compile Times

Analysis of Intermediate Code Optimizations

The gains in execution speed and code size were modest but not startling. For the most part, improvement was less than 10%. However, the results for compilation time are more promising. The speedup was between 24% and 31% for large programs, which is between a 15 and 45 second improvement.

The eliminated functions calls most likely have a negligible effect on execution speed but greatly contributed to the reduction in code size. For example, on a large program like *Yhcheng* which contained more than 18,600 lines of C code, approximately 450 redundant calls were removed. It was not expected that eliminating ``dead'' initialization loop would have much effect on execution speed. Constant propagation and loop unrolling probably accounted for the improved execution times. However, more of an improvement was expected from the constant propagation optimization. Two possible explanations could be that the native C compiler is able to reduce the complex structure lookup to its literal value or that the compiler has so much other baggage slowing down execution that the constant propagation improvement was not enough to make a great difference. The second explanation seems more likely.

The size of the intermediate code and executable code were also modestly improved. The elimination of redundant function calls offset the addition of code due to loop unrolling. Also, eliminating unnecessary initializations for literals that were propagated contributed to the smaller code sizes. It is important to note that as it is, the compiler generates an enormous amount of code for procedure continuations and suspensions so that 25-30% of the intermediate code are these functions and the rest is user code.

Lastly, the speed of compilation was a pleasant surprise; however, I do believe that this improvement is due to the type inferencing optimization because the current optimizations being discussed only add extra logic to improve the generated code. Another significant factor is that less memory is being used by the type inferencing system, which therefore causes less access to virtual memory. I should note that all the tests were run with that optimization on, and the improvement to type inferencing simplifies the type system in many ways. To determine if a specific bit is set, the old system had to create a mask and find the appropriate index into a long bit vector. The new system requires a single comparison in the case of the five builtin types.

Conclusion

All of the optimizations discussed in this chapter have been implemented. Some of the optimizations performed extremely well while others did not have much effect. The type representation change provided a substantial improvement on the memory usage required by the type inferencing system. As was stated early, the compiler still uses too much memory to be of much use to the average Icon programmer but is much better suited to offering the added speedup of compiled code when occasionally necessary.

The intermediate code optimizations were really just the tip of the iceberg of all the possible improvements to this area. The removal of redundant calls to system calls was a small improvement. Literal propagation was probably the most significant improvement along with loop unrolling. Further optimizations in this area are likely to yield the best improvements to performance.

Future Optimizations

After studying the generated code, several other optimizations were identified that may offer additional improvements to both the speed of execution and the size of the intermediate and executable code. The next few paragraphs describe additional optimizations and are organized in the order of the easiest to hardest to implement.

1. For the unrolled descriptor initializations change the indexing array to pointer arithmetic which is faster. For example the following code fragment is modified as follows:

```
r_frame.tend.d[0] = nulldesc;
r_frame.tend.d[1] = nulldesc;
```

```
register dptr p;
p = r_frame.tend.d;
(p++)->dword = nulldesc;
(p++)->dword = nulldesc;
```

2. Analyze the logic of loops and also unroll smaller ones. For example, the following loop appears at the beginning of most functions.

```
for (i = 0; i < r_nargs ; ++i)
   deref(&r_args[i], &r_frame.tend.d[i + 0]);
for(i = r_nargs; i < 2 ; ++i)
   r_frame.tend.d[i + 0] = nulldesc;
```

In this case `r_nargs` cannot be greater than two because it was earlier declared to have only two entries. It would be necessary to guarantee that `r_nargs` can never be more than two, but if it is certain that there are exactly two elements then we can write the initialization loop as follows:

```
if(r_nargs > 0) {
   deref(&r_args[0], &r_frame.tend.d[0]);
   if (r_nargs > 1)
      deref(&r_args[1], &r_frame.tend.d[1]);
   else
      tend.d[1].dword = D_Null;
   }
else
   tend.d[0].dword = D_Null;
```

This optimization could lead to a gain in execution speed. For example, if the unrolling is performed on descriptors with array sizes of one or two, approximately 40% of these loops would be unrolled.

3. An easy and obvious solution would be to simplify expressions like i + 0 which commonly occur. This will not improve execution time because the C compiler will catch this, but by removing it before writing the statement to the intermediate file, the compile time of the C compiler will be improved.

4. Another easy optimization would be to shorten variable names. This causes a penalty by having to write long names such as {\tt{r_frame.tend.d}} to file and then having the C compiler read it back in. This could be changed to {\tt{r_f.t.d}}. While this makes the intermediate C code hard to read, the intermediate code is not meant to be inspected by the user and will result in faster compilations.

5. For the initialization loops present in all functions, remove the initialization of the loop control variable when unnecessary. Consider the following loop:

```
for (i = 0; i < r_nargs ; ++i)
   deref(&r_args[i], &r_frame.tend.d[i + 0]);
for(i = r_nargs; i < 2 ; ++i)
   r_frame.tend.d[i + 0] = nulldesc;
```

The variable `i` in the second loop does not need to be initialized since it is already at the value that it is supposed to be for the second loop. The next fragment of code illustrates this change.

```
for (i = 0; i < r_nargs ; ++i)
   deref(&r_args[i], &r_frame.tend.d[i + 0]);
for( ; i < 2 ; ++i)
   r_frame.tend.d[i + 0] = nulldesc;
```

While this change is very easy, it is questionable whether this will provide noticeable improvement in execution except in large programs where these loops are very common.

6. Assignments of the `r_frame.tend.d` structures may be simplified. Consider the following assignment:

```
r_frame.tend.d[2] /* i */.vword.integr =
   r_frame.tend.d[4].vword.integr;
r_frame.tend.d[2] /* i */.dword = D_Integer;
```

This could be changed into a single assignment as follows:

```
   r_frame.tend.d[2] = r_frame.tend.d[4];
```

This optimization would require more work than the previously described ones. Each `struct val_loc` structure would have to be examined, including the context in which it is used in order to catch assignments such as this; however, these assignments are very common and could lead to substantial gains in execution speed.

7. Similarly, perform the same simplified descriptor assignment on global descriptor locations. A method needs to be created for changing global assignments such as:

```
globals[63] /* rnode */.dword = D_Integer;
globals[63] /* rnode */.vword.integr = 1;
```

into

```
globals[63] /* rnode */ = onedesc;
```

where `onedesc` is a single descriptor that already contains the values of the `dword` and `vword` being assigned. This could be performed by creating several constant decriptors for the common values such as 0 or 1. Like the previous optimization, this change will offer a smaller improvement to execution speed because global descriptor assignments occur much less frequently.

8. When a variable is dereferenced, it is often the case that the variable location is passed in for both parameters to the `deref` function. For example, in the following code example, `r_frame.tend.d[7]` is the variable being derefenced and the location where the dereferenced value is to be placed. This can be simplified by creating another version of `deref`, perhaps named `deref1`, that takes a single argument, dereferences it, and places the dereferenced value into the parameter location.

```
deref(&r_frame.tend.d[7], &r_frame.tend.d[7]);
```

9. Another issue is redundant constant initializations. Consider the following code:

```
r_frame.tend.d[8].dword = D_Integer;
r_frame.tend.d[8].vword.integr = 1;
if (!(argp[1] /* level */.vword.integr== 1) )
   goto L19 /* bound */;
r_frame.tend.d[8].dword = D_Integer;
r_frame.tend.d[8].vword.integr = 1;
```

The descriptor location 8 is assigned the value of 1 and then a conditional statement is performed which is followed by a possible `goto`. If the jump does not occur then the same descriptor location is assigned the same value over again. Clearly the second assignment is wasteful and needs to be eliminated. This would require fairly aggressive analysis of the intermediate code in order to catch these code sequences, but does offer the benefits of increased execution speed and smaller code size.

A more difficult optimization that offers a substantial reduction in the size of the intermediate and executable code deals with the initialization functions that set up frames. In the case of *Ctree*, over 30% of the generated C code consists of these functions. For example, in *Ctree* there are two functions named `P06i_set_value_Vtext` and `P03n_unset_Vbool_coupler` which are identical except for their frame structures, similarly defined as `PF06i_set_value_Vtext` and `PF03n_unset_Vbool_coupler`; however, these structures are identical. A possible solution would be to write out one copy of each unique frame structure along with its corresponding function that would initialize that frame. In addition to the reduction of code size this would result in faster compilations and faster loading of the resulting executable. This last optimizations is the most difficult and would require extensive changes; however, this optimization offers the best improvements in code size, execution time, and compile time.

Part III: The Implementation of Unicon

Chapter 26: The Unicon Translator

Unicon is implemented as a minimal addition to Icon. Most of its features are implemented by extending existing functions with new semantics, and by the addition of new functions and keywords with no new syntax. Originally, the object-oriented facilities were implemented as a preprocessor named Idol (Icon-drived object language). After several years of experience and evolution, Idol became a part of Unicon. Concurrently with this name change, thanks to Ray Pereda developing a version of Berkeley YACC for Icon and Unicon, the Unicon translator was substantially modified from a line-oriented preprocessor to a full parser that generates code by traversing the syntax tree. At this point it is still reasonable to call the Unicon translator a preprocessor, but it has many of the traits of a compiler.

26.1 Overview

The Unicon translator lives in uni/unicon/. In addition to many Unicon source files, it uses the external tools iyacc and merr to generate its parser and syntax error message tables, which depend on files unigram.y and meta.err, respectively. Unicon is written in Unicon, creating a bootstrapping problem. When building from sources, some of the .icn files can be translated by icont (the Icon translator, a C program). Those files that require Unicon itself in order to compile are included in precompiled object format (in .u files) in order to solve the bootstrapping problem.

26.2 Lexical Analysis

Unicon's lexical analyzer is written by hand, in Unicon, using a lex-compatible interface. Some of its design is borrowed from the Icon lexical analyzer (which is handwritten C code). It would be interesting to replace Unicon's lexical analyzer with a machine generated lexical analyzer to reduce the amount of compiler source code we have to maintain. The lexical analyzer consists of a function yylex() located in unicon/uni/unicon/unilex.icn, about 500 lines of code.

Globals Comprising the Lex-compatible Public API

The global declarations that exist in order to provide a Lex-compatible API include:

```
$include "ytab_h.icn"                    # yacc's token categories
global yytext                            # lexeme
global yyin                              # source file we are
reading
global yytoken                           # token (a record)
global yylineno, yycolno, yyfilename     # source location
```

Character Categories

The lexical analyzer uses several csets for different character categories beyond the built-in ones:

```
global O, D, L, H, R, FS, IS, W, idchars

procedure init_csets()
   O  := '01234567'
   D  := &digits
   L  := ≤tters ++ '_'
```

```
   H  := &digits ++ 'abcdefABCDEF'
   R  := &digits ++ ≤tters
   FS := 'fFlL'
   IS := 'uUlL'
   W  := ' \t\v'
   idchars := L ++ D
end
```

The Token Type

The record type storing each token's information just bundles together the syntactic category (an integer), lexeme (a string), and location at which the token occurred. This is pretty minimalist.

```
record token(tok, s, line, column, filename)
```

Global Variables for Error Handling and Debugging

Several Remaining global variables are mainly used for error handling, and for debugging the lexical analyzer itself.

Reserved Words

Global reswords() creates and becomes a table holding the Unicon reserved words. For each word, a pair of integers [tokenflags, category] is kept. Language design note: tables in this language need a "literal" format.

```
procedure reswords()
static t
initial {
   t := table([Beginner+Ender, IDENT])

   t["abstract"] := [0, ABSTRACT]
   t["break"] := [Beginner+Ender, BREAK]
   t["by"] := [0, BY]
   t["case"] := [Beginner, CASE]
   t["class"] := [0, CLASS]
   t["create"] := [Beginner, CREATE]
   t["default"] := [Beginner, DEFAULT]
   t["do"] := [0, DO]
   t["else"] := [0, ELSE]
   t["end"] := [Beginner, END]
   t["every"] := [Beginner, EVERY]
   t["fail"] := [Beginner+Ender, FAIL]
   t["global"] := [0, GLOBAL]
   t["if"] := [Beginner, IF]
   t["import"] := [0, IMPORT]
   t["initial"] := [Beginner, iconINITIAL]
   t["initially"] := [Ender, INITIALLY]
   t["invocable"] := [0, INVOCABLE]
   t["link"] := [0, LINK]
   t["local"] := [Beginner, LOCAL]
   t["method"] := [0, METHOD]
   t["next"] := [Beginner+Ender, NEXT]
   t["not"] := [Beginner, NOT]
   t["of"] := [0, OF]
   t["package"] := [0, PACKAGE]
   t["procedure"] := [0, PROCEDURE]
   t["record"] := [0, RECORD]
```

```
   t["repeat"] := [Beginner, REPEAT]
   t["return"] := [Beginner+Ender, RETURN]
   t["static"] := [Beginner, STATIC]
   t["suspend"] := [Beginner+Ender, SUSPEND]
   t["then"] := [0, THEN]
   t["to"] := [0, TO]
   t["until"] := [Beginner, UNTIL]
   t["while"] := [Beginner, WHILE]
}
   return t
end
```

Lexical Analyzer Initialization and the Big Inhale

A function, yylex_reinit() is called the first time yylex() is called, along with each time the compiler moves to process a new file named on the command line. Along with initializing the public API variables, this function reads in the entire file, in a single global string variable, named "buffer". This allows extremely fast subsequent processing, which does not file I/O for each token, while avoiding complex buffering sometimes done to reduce file I/O costs in compilers.

This "big-inhale" model did not work well on original 128K PDP-11 UNIX computers, but works well in this century. At present, the code assumes Unicon source files are less than a megabyte -- a lazy programmer's error. Although Unicon programs are much shorter than C programs, an upper limit of 1MB is bound to be reached someday.

```
procedure yylex_reinit()
   yytext := ""
   yylineno := 0
   yycolno := 1
   lastchar := ""
   if type(yyin) == "file" then
      buffer := reads(yyin, 1000000)
   else
      buffer := yyin
   tokflags := 0
end
```

Semicolon Insertion

Icon and Unicon insert semicolons for you automatically. This is an easy lexical analyzer trick. The lexical analyzer requires one token of lookahead. Between each two tokens, it asks: was there a newline? If yes, was the token before the newline one that could conceivably be the end of an expression, and was the token at the start of the new line one that could conceivably start a new expression? If it would be legal to do so, it saves the new token and returns a semicolon instead.

This little procedure is entirely hidden from the regular lexical analyzer code by writing that regular code in a helper function yylex2(), and writing the semicolon insertion logic in a yylex() function that calls yylex2 when it needs a new token.

Initialization for the yylex() function shows the static variables used to implement the one token of lookahead. If the global variable buffer doesn't hold a string anymore, /buffer will succeed and it must be that we are at end-of-file and should return 0.

```
procedure yylex()
  static saved_tok, saved_yytext
  local rv, ender
```

```
  initial {
      if /buffer then
          yylex_reinit()
     }
   if /buffer then {
      if \debuglex then
         write("yylex() : 0")
      return 0
      }
```

If we inserted a semicolon last time we were called, the saved_tok will be the first token of the next line; we should return it.

```
  if \saved_tok then {
    rv := saved_tok
    saved_tok := vll
    yytext := saved_yytext
    yylval := yytoken := token(rv, yytext, yylineno, yycolno,
yyfilename)
    if \debuglex then
      write("yylex() : ",tokenstr(rv), "\t", image(yytext))
    return rv
  }
```

Otherwise, we should obtain the next token by calling yylex2(). We have to check for end of file, remember if the last token could end an expression, call yylex2(), and update buffer to be the smaller string remaining after the token.

```
  ender := iand(tokflags, Ender)
  tokflags := 0
  if *buffer=0 then {
      buffer := vll
      if \debuglex then
          write("yylex() : EOFX")
      return EOFX
     }
  buffer ? {
      if rv := yylex2() then {
          buffer := tab(0)
      }
      else {
         buffer := vll
         yytext := ""
         if \debuglex then
             write("yylex() : EOFX")
         return EOFX
      }
  }
```

After fetching a new token, we have to decide whether to insert a semicolon or not. This is based on global variable ender (whether the previous token could end an expression) and global variable tokflags (which holds both whether the current token could begin an expression, and whether a newline occurred between the last token and the current token. iand() is a bitwise AND, equivalen to C language & operator, used to pick bits out of a set of boolean flags encoded as bits within an integer.

```
  if ender~=0 & iand(tokflags, Beginner)~=0 & iand(tokflags,
Newline)~=0 then {
    saved_tok := rv
    saved_yytext := yytext
```

```
    yytext := ";"
    rv := SEMICOL
    }
```

Returning a token requires allocation of a token() record instance, which is stored in a global variable.

```
   yylval := yytoken := token(rv, yytext, yylineno, yycolno,
yyfilename)
   if \debuglex then
      write("yylex() : ", tokenstr(rv), "\t", image(yytext))
   return rv
end
```

The Real Lexical Analyzer Function, yylex2()

This function maintains a table of functions, calling a helper function depending on what the first character in the token is.

```
procedure yylex2()
static punc_table
initial {
   init_csets()
   reswords := reswords()
   punc_table := table(uni_error)
   punc_table["'"] := do_literal
   punc_table["\""] := do_literal
   punc_table["!"] := do_bang
   punc_table["%"] := do_mod
   punc_table["&"] := do_and
   punc_table["*"] := do_star
   punc_table["+"] := do_plus
   punc_table["-"] := do_minus
   punc_table["."] := do_dot
   punc_table["/"] := do_slash
   punc_table[":"] := do_colon
   punc_table["<"] := do_less
   punc_table["="] := do_equal
   punc_table[">"] := do_greater
   punc_table["?"] := do_qmark
   punc_table["@"] := do_at
   punc_table["\\"] := do_backslash
   punc_table["^"] := do_caret
   punc_table["|"] := do_or
   punc_table["~"] := do_tilde
   punc_table["("] := do_lparen
   punc_table[")"] := do_rparen
   punc_table["["] := do_lbrack
   punc_table["]"] := do_rbrack
   punc_table["{"] := do_lbrace
   punc_table["}"] := do_rbrace
   punc_table[","] := do_comma
   punc_table[";"] := do_semi
   punc_table["$"] := do_dollar
   every punc_table[!&digits] := do_digits
   every punc_table["_" | !≤tters] := do_letters
   }
```

The main lexical analyzer code strips comments and whitespace, and calls the function table for the first non-whitespace character it finds. Note support for #line directives, and the use of string scanning.

```
   yycolno +:= *yytext

   repeat {
       if pos(0) then fail
       if
           ="#" then {
               if ="line " then {
                   if yylineno := integer(tab(many(&digits)))
then {
                       =" \""
                       yyfilename := tab(find("\"")|0)
                   }
               }
               tab(find("\n") | 0)
               next
           }
       if ="\n" then {
           yylineno +:= 1
           yycolno := 1
           if tokflags < Newline then
               tokflags +:= Newline
           next
       }
       if tab(any(' ')) then { yycolno +:= 1; next }
       if tab(any('\v\^l')) then { next }
       if tab(any('\t')) then {
           yycolno +:= 1
           while (yycolno-1) % 8 ~= 0 do yycolno +:= 1
           next
       }

       yytext := move(1)
       return punc_table[yytext]()
   }
end
```

The functions in the punctuation table select integer codes and match the rest of the lexeme. do_comma() illustrates an unambiguous token selection, while do_plus() illustrates a more common case where the "+" character could start any of 5 different tokens depending on the character(s) that follow it. Tokens starting with "letters" are looked up in a reserved words table, which tells whether they are special, or just a variable name.

```
procedure do_comma()
   return COMMA
end

procedure do_plus()
   if yytext ||:= ":" then {
      if yytext ||:= "=" then { return AUGPLUS }
         return PCOLON
      }
   if yytext ||:= "+" then {
      if yytext ||:= ":=" then {return AUGUNION}
```

```
         return UNION
      }
   tokflags +:= Beginner
   return PLUS
end

procedure do_letters()
   yytext ||:= tab(many(idchars))
   x := reswords[yytext]
   tokflags +:= x[1]
   return x[2]
end
```

26.3 The Unicon Parser

Unicon's parser is written using a YACC grammar; a graduate student (Ray Pereda) modified Berkeley's public domain version of YACC (byacc) to generate Unicon code, following in the footsteps of someone who had earlier modified it to generate Java. The Unicon parser lives in uni/unicon/unigram.y in the source distribution (22kB, 700 lines, 119 terminals, 71 nonterminals). Unicon's YACC grammar was obtained by copying the Icon grammar, and adding Unicon syntax constructs. Prior to this time the object-oriented dialect of Icon was called Idol and really was a line-oriented preprocessor instead of a compiler.

The start symbol for the grammar is named `program`, and the semantic action code fragment for this nonterminal calls the rest of the compiler (semantic analysis and code generation) directly on the root of the syntax tree, rather than storing it in a global variable for the main() procedure to examine.

```
program : decls EOFX { Progend($1);} ;
```

Many context free grammar rules are recursive, with an empty production to terminate the recursion. The rule for declarations is typical:

```
decls   : { $$ := EmptyNode }
        | decls decl {
           if yynerrs = 0 then iwrites(&errout,".")
           $$ := node("decls", $1, $2)
           } ;
```

The "semantic action" (code fragment) for every production rule builds a syntax tree node and assigns it to $$ for the nonterminal left-hand side of the rule.

Another common grammar pattern is a production rule that has many different alternatives, such as the one for individual declarations:

```
decl    : record
        | proc
        | global
        | link
        | package
        | import
        | invocable
        | cl
        ;
```

For such "unary" productions, child's syntax tree node suffices for the parent, no new tree node is needed.

Some nonterminals mostly correspond to a specific sequence of terminals, as is the case for package references:

```
packageref : IDENT COLONCOLON IDENT { $$ := node("packageref",
$1,$2,$3) }
   | COLONCOLON IDENT { $$ := node("packageref", $1,$2) }
   ;
```

The lexical analyzer has already constructed a valid "leaf" for each terminal symbol, so if a production rule has only one terminal symbol in it, for a syntax tree we can simply use the leaf for that nonterminal (for a parse tree, we would need to allocate an extra unary internal node):

```
lnkfile : IDENT ;
        | STRINGLIT ;
```

The expressions (which comprise about half of the grammar) use a separate nonterminal for each level of precedence intead of YACC's declarations for handling precedence (%left, %right, etc). The Icon and Unicon grammars approach 20 levels of nonterminals. A typical rule looks like:

```
expr6   : expr7 ;
        | expr6 PLUS expr7 { $$ := node("Bplus", $1,$2,$3);} ;
        | expr6 DIFF expr7 { $$ := node("Bdiff", $1,$2,$3);} ;
        | expr6 UNION expr7 { $$ := node("Bunion", $1,$2,$3);} ;
        | expr6 MINUS expr7 { $$ := node("Bminus", $1,$2,$3);} ;
```

The "B" stands for "binary", to distinguish these operators from their unary brethren. The 20 levels of nonterminals approach is inherited from Icon and probably makes the parser larger than it has to be, but taking these nonterminals out doesn't seem to help much.

Syntax Error Handling

Icon employed a relatively clever approach to doing syntax error messages with YACC -- the parse state at the time of error is enough to do fairly good diagnoses. But, every time the grammar changed, the parse state numbers could change wildly. For Unicon I developed the Merr tool, which associates parse error example fragments with the corresponding diagnostic error message, and detects/infers the parse state for you, reducing the maintenance problem when changing the grammar. Merr also considers the current input token in deciding what error message to emit, making it fundamentally more precise than Icon's approach.

26.4 The Unicon Preprocessor

The Icon language originally did not include any preprocessor, but eventually, a simple one was introduced, with ability to include headers, define symbolic constants (macros without parameters), and handle conditional compilation (ifdef). The preprocessor implementation in Unicon was written by Bob Alexander, and came to Unicon by way of Jcon, an Icon-to-JVM translator. This preprocessor is written in a single 600+ line file, uni/unicon/preproce.icn.

The external public interface of the preprocessor is line-oriented, consisting of a generator preproc(filename, predefinedsyms) which suspends each line of the output, one after another. Its invocation from the main() procedure looks like:

```
   yyin := ""
   every yyin ||:= preprocessor(fName, uni_predefs) do
      yyin ||:= "\n"
```

Since the preprocessor outputs line-by-line, there is a mismatch between it and the lexical analyzer's big-inhale model. The preprocessor could be modified to fit better with the lexical analyzer or vice versa.

The preprocessor function takes the filename to read from, along with a table of predefined symbols which allows the preprocessor to respond to lines like

```
$ifdef _SQL
```

based on what libraries are available and how Unicon was built on a given platform.

The preprocessor() function itself starts each call off with initializations:

```
    static nonpunctuation
    initial {
        nonpunctuation := ≤&letters ++ &digits ++ ' \t\f\r'
    }

    preproc_new(fname,predefined_syms)
```

The initialization code opens fname, creates empty stacks to keep track of nested $ifdef's and $include's, initializes counters to 0 and so forth.

The preprocessor is line-oriented. For each line, it looks for a preprocessor directive, and if it does not find one, it just scans for symbols to replace and returns the line. The main loop looks like

```
   while line := preproc_read() do line ? {
      preproc_space()       # eat whitespace
      if ="#" & match("line")) | ="$" & any(nonpunctuation))
then {
         suspend preproc_scan_directive()
         }
      else {
         &pos := 1
         suspend preproc_scan_text()
         }
      }
```

The procedures preproc_scan_directive() and preproc_scan_text() work on special and ordinary lines, respectively. The line is not a parameter because it is held in the current string scanning environment. The preproc_scan_directive() starts by discardign whitespace and identifying the first word on the line (which must be a valid preprocessor directive). A case expression handles the various directives (define, undef, ifdef, etc.). Defined symbols are stored in a table. $ifdef and $ifndef are handled using a global variable preproc_if_state to track the boolean conditions. A count of $ifdef's is maintained, in order to handle matching endif's.

Include files are handled using a stack, but an additional set of filenames is kept to prevent infinite recursion when files include each other. When a new include directive is encountered it is checked against the preproc_include_set and if OK, it is opened. The including file (and its associated name, line, etc) are pushed onto a list named preproc_file_stack. It is possible to run out of open files under this model, although this is not easy under modern operating systems.

Include files are searched on an include file path, consisting of a list of directories given on an optional environment variable (LPATH) followed by a list of standard directories.

The standard directories are expected to be found relative to the location of the virtual machine binaries.

The procedure preproc_scan_text has the relatively simple job of replacing any symbols by their definitions within an ordinary source line. Since macros do not have parameters, it is vastly simpler than in a C preprocessor. The main challenges are to avoid macro substitutions when a symbol is in a comment or within quotes (string or cset literals). An additional issue is to handle multiline string literals, which occur in Icon when a string literal is not closed on a line, and instead the line ends with an underscore indicating that it is continued on the next line. Skipping over quoted text sounds simple, but is trickier than it looks. Escape characters mean you can't just look for the closing quote without considering what comes before it, and you can't just look at the preceding character since it might have been escaped, as in "\\". The code looks similar to:

```
repeat {
   while tab(upto('"\\')) do {
      case move(1) of {
         "\\": move(1)
         default: {
            break break
            }
         }
      }
   # ...
   if not match("_",,-1) then
      break
   ⊂&subject := preproc_read() | fail
   # ...
   }
```

The code in preproc_read() for reading a line does a regular Icon read(); end of file causes the preprocessor file_stack to be popped for the previous file's information. Performance has not been perceived as a significant problem, it it would be interesting to convert preproc_read() to use a big-inhale model to see if any statistical difference could be observed. When an include is encountered under a big-inhale, the saved state would contain the string of remaining file contents, instead of the open file value.

26.5 Semantic Analysis

The Unicon translator's semantic analysis is minimal, and revolves mainly around object-oriented features such as inheritance and package imports. Before we can look at those things, we need to look at the syntax tree structure.

In conventional YACC, a %union declaration is necessary to handle the varying types of objects on the value stack including the type used for syntax tree nodes, but iyacc has no need of this awkward mechanism: the value stack like all structure types can hold any type of value in each slot. Similarly, tree nodes can hold children of any type, potentially eliminating any awkwardness of mixing tokens and internal nodes. Of course, you do still have to check what kind of value you are working with.

Parse Tree Nodes

uni/unicon/tree.icn contains procedures to handle the syntax tree node data type, including both the following declaration and the yyprint() traversal function we'll be discussing in today's lecture.

```
record treenode(label, children)
```

holds one node worth of information. For convenience, a procedure node(label, kids[]) takes an arbitrary number of parameters and constructs the list of children for you. Leaves have a null children field.

"Code Generation" in the Unicon Translator

In a regular preprocessor, there is no code generation, there is a text-filter model in which the preprocessor writes out (modified) versions of the lines it reads in. In the Unicon translator, the code that is written out is produced by a traversal of the syntax tree. The same technique might be used by a "pretty printer". We will explore this aspect of the Unicon translator as the best available demonstration of working with Unicon syntax trees. Later on we will consider more "real" code generation in the virtual machine and the optimizing compiler.

Earlier we saw that the start symbol of the Unicon grammar had a semantic action that called a procedure Progend(). We will cover most of that procedure next week since it is all about object-orientation, but at the end Progend(), a call to yyprint() performs the tree traversal for code generation. A classic tree traversal pattern would look like:

```
procedure traverse(node)
   if node is an internal node {
      every child := ! node.children do traverse(child)
      generate code for this internal node (postfix)
      }
   else
      generate code for this leaf
end
```

The code generator traversal yyprint() is a lot more complicated than that, but fits the general pattern. The main work done at various nodes is to write some text to the output file, yyout. Most ordinary internal nodes are of type treenode as described above. But because there are several kinds of internal nodes and several kinds of leaves, the "if node is an internal node" is implemented as a case expression. Besides a regular treenode, the other kinds of internal nodes are objects of type declaration, class, and argument list. For regular treenodes, another case expression on the node's label field is used to determine what kind of code to generate, if any, besides visiting children and generating their code.

The default behavior for an internal node is to just visit the children, generating their code. For ordinary syntax constructs (if, while, etc.) this works great and a copy of the code is written out, token by token. But several exceptions occur, mainly for the pieces of Unicon syntax that extend Icon's repertoire. For example, packages and imports are not in Icon and require special treatment.

```
procedure yyprint(node)
   static lasttok
   case type(node) of {
      "treenode" : {
         case node.label of {
         "package": { } # handled by semantic analysis
         "import": { print_imports(node.children[2]) }
         # implement packages via name mangling
         "packageref": {
             if *node.children = 2 then
                 yyprint(node.children[2]) # ::ident
             else { # ident :: ident
```

```
                yyprint(node.children[1])
                writes(yyout, "__")
                outcol +:= ((* writes(yyout, node.children[3].s))
+ 2)
                }
            }
```

New syntax constructs such as procedure parameter defaults and type restrictions, and variable initializers, are other examples where the default traversal would output things illegal in Icon. They are implemented by skipping some of the children (assignment and value) in the regular pass, and adding extra code elsewhere, discussed below.

```
         "varlist2"|"stalist2": { yyprint(node.children[1]) }
         "varlist4"|"stalist4": {
            yyprint(node.children[1])
            yyprint(node.children[2])
            yyprint(node.children[3])
            }
```

Much of this special logic is orchestrated by the code for traversing a procedure; it can visit its arguments and variable declarations and apply special rules to them.

```
         "proc": {
            yyprint(node.children[1])
            every yyprint(node.children[2 to 3])
            if exists_statlists(node.children[3]) then {
               ini := node.children[4]
               yyprint("\ninitial {")
               if ini ~=== EmptyNode then { # append into
existing initial
                  yyprint(ini.children[2])
                  yyprint(";\n")
                  }
               yystalists(node.children[3])
               yyprint("\n}\n")
               }
            else
               every yyprint(node.children[4])
            (node.children[1].fields).coercions()
            yyvarlists(node.children[3])
            yyprint(node.children[5])
            yyprint(node.children[6])
            }
```

The default behavior of visiting one's children is very simple, as is the handling of other kinds of internal nodes, which are objects. For the objects, a method Write() is invoked.

```
         "error": fail
         default:
            every yyprint(!node.children)
         }
      "declaration__state" | "Class__state" | "argList__state":
         node.Write(yyout)
```

The outer case expression of yyprint() continues with various kinds of leaf (token) nodes. These mainly know how to write their lexemes out. But, a lot of effort is made to try to keep line and column number information consistent. Variables outline and outcol are maintained as each token is written out. Integers and string literals found in the syntax tree are written out as themselves. Since they have no attached lexical attributes, they are a bit suspect in terms of maintaining debugging consistency. It turns out the reason they

occur at all, and the reason they have no source lexical attributes, is that artificial syntax subtrees are generated to handle certain object-oriented constructs, and within those subtrees strings and integers may be placed, which do not correspond to anywhere in the source code.

```
      "integer": {
         writes(yyout, node); outcol +:= *string(node)
         }
      "string": {
         node ? {
            while writes(yyout, tab(find("\n")+1)) do {
               outline+:=1; outcol:=1;
               }
            node := tab(0)
            }
         writes(yyout, node); outcol +:= *node
         }
```

"Normally", tokens are written out at exactly the line and column they appear at in the source code. But a myriad of constructs may bump them around. If the output falls behind (in lines, or columns) extra whitespace can be inserted to stay in sync. If output gets ahead by lines, a #line directive can back it up, but if output gets ahead by columns, there is nothing much one can do, except make sure subsequent tokens don't accidentally get attached/concatenated onto earlier tokens. This occurs, for example, when the output code for an object-oriented construct in an expression is longer than the source expression, perhaps due to name mangling. Specific token combinations are checked, but the list here may be incomplete (possible BUG!). For source tokens, not only might the line and column change, the filename could be different as well.

```
      "token": {
         if outfilename ~== node.filename | outline > node.line
then {
            write(yyout,"\n#line ", node.line-1," \"",
node.filename,"\"")
            outline := node.line
            outcol := 1
            outfilename := node.filename
            }
         while outline < node.line do {
            write(yyout); outline +:= 1; outcol := 1
            }
         if outcol >= node.column then {
            # force space between idents and reserved words, and
other
            # deadly combinations (need to add some more)
            if ((\lasttok).tok = (IDENT|INTLIT|REALLIT) &
reswords[node.s][2]~=IDENT)|
                (((\lasttok).tok = NMLT) & (node.tok = MINUS)) |
                ((\lasttok).tok = node.tok = PLUS) |
                ((\lasttok).tok = node.tok = MINUS) |
                ((reswords[(\lasttok).s][2]~=IDENT) &
(node.tok=(IDENT|INTLIT|REALLIT)))|
                ((reswords[(\lasttok).s][2]~=IDENT) &
(reswords[node.s][2]~=IDENT))
                   then
               writes(yyout, " ")
            }
```

```
         else
            while outcol < node.column do { writes(yyout, " ");
outcol +:= 1 }
```

Most tokens' lexemes are finally written out by writing node.s:

```
         writes(yyout, node.s)
         outcol +:= *node.s
         lasttok := node
         }
      "null": { }
      default: write("its a ", type(node))
      }
end
```

Keywords

Besides the large set of interesting reserved words, Icon and Unicon have another set of predefined special words called *keywords*. These words are prefixed by an ampersand, for example, &subject holds the current "subject" string being examined by string scanning. A procedure Keyword(x1,x2) semantically checks that an identifier following a unary ampersand is one of the valid keyword names. The valid names are kept in a set data structure.

26.6 Object Oriented Facilities

Unicon features classes, packages, and a novel multiple inheritance mechanism. These items are implemented entirely within the Unicon translator. The Icon virtual machine thusfar has only the slightest of extensions for object-orientation, specifically, the dot operator has been extended to handle objects and method invocation.

The Unicon OOP facilities were originally prototyped as a semester class project in a "special topics" graduate course. Writing the prototype in a very high-level language like Icon, and developing it as a preprocessor with name mangling, allowed the initial class mechanism to be developed in a single evening, and a fairly full, usable system with working inheritance to be developed in the first weekend. By the end of the semester, the system was robust enough to write it in itself, and it was released to the public shortly afterwards as a package for Icon called "Idol". Many many improvements were made after this point, often at the suggestion of users.

An initial design goal was to make the absolute smallest additions to the language that were necessary to support object-orientation. Classes were viewed as a version of Icon's record data type, retaining its syntax for fields (member variables), but appending a set of associated procedures. Because records have no concept of public and private, neither did classes. Another graduate student criticized this lack of privacy, and for several versions, everything was made private unless an explicit public keyword was used. But eventually support for privacy was dropped on the grounds that it added no positive capabilities and was un-Iconish. The existence of classes with hundreds of "getter" and "setter" methods was considered a direct proof that "private" was idiotic in a rapid prototyping language.

The Code Generation Model for Classes

"unicon -E foo" will show you what code is generated for Unicon file foo.icn. If foo.icn contains classes, you can enjoy the code generation model and experiment to see what it does under various circumstances. As a first example, consider

```
class A(x,y)
   method m()
      write("hello")
   end
end
```

These five lines generate 25 lines for Icont to translate into virtual machine code. The first two lines are line directives showing from whence this source code originated:

```
#line 0 "/tmp/uni13804206"
#line 0 "a.icn"
```

Global declarations (including procedures) would be passed through the preprocessor pretty nearly intact, but for the class, we get a bunch of very different code. Methods are written out, with names mangled to a classname_methodname format.

```
procedure A_m(self)


#line 2 "a.icn"
     write("hello");
end
```

Two record types are defined, one for the class instances and one for the "methods vector", or "operation record". The methods vector is instantiated exactly once in a global variable in classname__oprec format.

```
record A__state(__s,__m,x,y)
record A__methods(m)
global A__oprec
```

The default constructor for a class takes fields as parameters and uses them directly for initialization purposes. The first time it is called, a methods vector is created. Instances are given a pointer to themselves in an __s field (mainly for historical reasons) and to the methods vector in an __m field. Current NMSU grad student Sumant Tambe did an independent study project to get rid of __s and __m with partial success, but his work is not finished or robust enough to be enabled by default.

```
procedure A(x,y)
local self,clone
initial {
  if /A__oprec then Ainitialize()
  }
  self := A__state(vll,A__oprec,x,y)
  self.__s := self
  return self
end

procedure Ainitialize()
  initial A__oprec := A__methods(A_m)
end
```

Symbols and Scope Resolution

One of the basic aspects of semantic analysis is: for each variable, where was it declared, so we can identify its address, etc. Unicon inherits from Icon the curious convenience that variables do not have to be declared: they are local by default. This feature is implemented by deferring the local vs. global decision until link time, so the Unicon translator has no local vs. global issues. Class variables, however, have to be identified, and looked up relative to the implicit "self" variable. A family of procedures in

uni/unicon/tree.icn with names starting "scopecheck" go through the syntax tree looking for such class variables. Like most tree traversals, this is a recursive process, and since local and parameter declarations override class variables, there are helper functions to walk through subtrees building mini-symbol tables such as local_vars in scopecheck_proc(node):

```
# Build local_vars from the params and local var expressions.
local_vars := set()
extract_identifiers(node.children[1].fields, local_vars)
extract_identifiers(node.children[3], local_vars)
```

Eventually, every identifier in every expression is checked against local_vars, and if not found there, against the class variables stored in a variable self_vars:

```
self_vars := set()
every insert(self_vars, c.foreachmethod().name)
every insert(self_vars, c.foreachfield())
every insert(self_vars, (!c.ifields).ident)
every insert(self_vars, (!c.imethods).ident)
```

For an IDENT node, the tests boil down to:

```
if node.tok = IDENT then {
   if not member(\local_vars, node.s) then {
      if member(\self_vars, node.s) then
         node.s := "self." || node.s
      else
         node.s := mangle_sym(node.s)
   }
}
```

Undeclared locals and globals are mangled to include the current package name if there is one.

Inheritance

Inheritance means: creating a class that is similar to an existing class. In object-oriented literature there is "abstract inheritance" in which a class supports all the same operations with the same signatures, and there is concrete inheritance in which actual code is shared. Early object-oriented languages supported only concrete inheritance, while more recent languages tend to discourage it. Unicon is not typed at compile time, so abstract inheritance is not a big deal. There are abstract methods, and classes whose every method is abstract, but the use of abstract is mainly for documentation: subclass authors must provide certain methods. Anyhow, the syntax of inheritance in Unicon is

```
class subclass : super1 : super2 : ... ( ...fields... )
```

The semantics of inheritance, and particularly of multiple inheritance, are interesting in Unicon; the implementation is relatively simple. An example of inheritance is given by class Class, from uni/unicon/idol.icn

```
class declaration(name,fields,tag,lptoken,rptoken)
   ...
end
...
class Class : declaration (supers,
                           methods,
                           text,
                           imethods,
                           ifields,
```

```
                                glob,
                                linkfile,
                                dir,
                                unmangled_name,
                                supers_node)
```

Unique perspective on inheritance in Unicon comes from the actual acquisition of inherited data fields and methods by the subclass. Some object-oriented languages do this inheritance "by aggregation", creating a copy of the superclass in the subclass. This is fine, but it makes "overriding" an anomaly, when overriding the parent with new/different behavior is entirely routine. Unicon instead inherits by the child looking for things in the parent (and the parent's parent, etc.) that they don't already have. In the above example, class declaration effectively appends 5 fields from class declaration onto the end of its field list. The generated code for instances looks like

```
record Class__state(__s,__m,
                    supers,methods,text,imethods,ifields,
                    glob,linkfile,dir,unmangled_name,supers_node,
                    name,fields,tag,lptoken,rptoken)
```

The inheritance semantics is called "closure based" because the process of looking for things to add from parent superclasses iterates until no new information can be added, after which the subclass is said to be closed on its parents. Other forms of closure appear frequently in CS.

Implementing Multiple Inheritance in Unicon

The actual code in the Unicon translator is, by analogy to transitive closure, looking for things to inherit via a depthfirst traversal of the inheritance graph. Multiple inheritance can be separated out into two portions:

1. a method transitive_closure() that finds all superclasses and provides a linearization of them, flattening the graph into a single ordered list of all superclasses

2. a method resolve() that walks the list and looks for classes and fields to add.

Method transitive_closure() is one of the cleaner demonstrations of why Unicon is a fun language in which to write complex algorithms. It is walking through a class graph, but by the way it is not recursive.

```
  method transitive_closure()
    count := supers.size()
    while count > 0 do {
        added := taque()
        every sc := supers.foreach() do {
          if /(super := classes.lookup(sc)) then
            halt("class/transitive_closure: _
                  couldn't find superclass ",sc)
          every supersuper := super.foreachsuper() do {
            if / self.supers.lookup(supersuper) &
                 /added.lookup(supersuper) then {
              added.insert(supersuper)
            }
          }
        }
        count := added.size()
        every self.supers.insert(added.foreach())
```

```
    }
  end
```

Now, given what I've said about Unicon providing a depthfirst inheritance hierarchy semantics, what is wrong with this picture? The code is stable and hasn't needed changes in several years, so I am not fishing for syntax bugs, or claiming that there is a bug. But there is something odd. A chocolate "peanut butter cup" is available in my office for the first correct description of the problem.

The method resolve() within class Class finds the inherited fields and methods from the linearized list of superclasses.

```
  #
  # resolve -- primary inheritance resolution utility
  #
  method resolve()
    #
    # these are lists of [class , ident] records
    #
    self.imethods := []
    self.ifields := []
    ipublics := []
    addedfields := table()
    addedmethods := table()
    every sc := supers.foreach() do {
        if /(superclass := classes.lookup(sc)) then
            halt("class/resolve: couldn't find superclass ",sc)
        every superclassfield := superclass.foreachfield() do {
            if /self.fields.lookup(superclassfield) &
               /addedfields[superclassfield] then {
                addedfields[superclassfield] := superclassfield
                put ( self.ifields ,
classident(sc,superclassfield) )
                if superclass.ispublic(superclassfield) then
                    put( ipublics, classident(sc,superclassfield)
)
            } else if \strict then {
                warn("class/resolve: '",sc,"' field
'",superclassfield,
                     "' is redeclared in subclass ",self.name)
            }
        }
        every superclassmethod :=
(superclass.foreachmethod()).name() do {
            if /self.methods.lookup(superclassmethod) &
               /addedmethods[superclassmethod] then {
                addedmethods[superclassmethod] :=
superclassmethod
                put ( self.imethods,
classident(sc,superclassmethod) )
            }
        }
        every public := (!ipublics) do {
            if public.Class == sc then
                put (self.imethods, classident(sc,public.ident))
        }
    }
  end
```

Class and Package Specifications

In the "old days" of Unicon's ancestor Idol, you could only inherit from a class that appeared in the same source file. Anything else poses a librarian's problem of identifying from what file to inherit. Java, for instances, takes a brute-force approach of one class per file.

Unicon generates in each source directory an NDBM database (named uniclass.dir and uniclass.pag) that includes a mapping from class name to: what file the class lives in, plus, what superclasses, fields, and methods appear in that class. From these specifications, "link" declarations are generated for superclasses within subclass modules, plus the subclass can perform inheritance resolution. The code to find a class specification is given in idol.icn's fetchspec(). A key fragment looks like

```
   if f := open(dir || "/" || env, "dr") then {
      if s := fetch(f, name) then {
         close(f)
         return db_entry(dir, s)
         }
      close(f)
      }
```

Unicon searches for "link" declarations in a particular order, given by the current directory followed by directories in an IPATH (Icode path, or perhaps Icon path) environment variable, followed by system library directories such as ipl/lib and uni/lib. This same list of directories is searched for inherited classes.

The string stored in uniclass.dir and returned from fetch() for class Class is:

```
idol.icn
class Class :
declaration(supers,methods,text,imethods,ifields,glob,linkfile,di
r,unmangled_name,supers_node)
ismethod
isfield
Read
ReadBody
has_initially
ispublic
foreachmethod
foreachsuper
foreachfield
isvarg
transitive_closure
writedecl
WriteSpec
writemethods
Write
resolve
end
```

Unicon's Progend() revisited

Having presented scope resolution, inheritance, and importing packages and inheriting classes from other files via the uniclass.dir NDBM files, we can finally show the complete semantic analysis in the Unicon compiler, prior to writing out the syntax tree as Icon code:

```
procedure Progend(x1)
```

```
   package_level_syms := set()
   package_level_class_syms := set()
   set_package_level_syms(x1)
   scopecheck_superclass_decs(x1)

   outline := 1
   outcol := 1
   #
   # export specifications for each class
   #
   native := set()
   every cl := classes.foreach_t() do {
      cl.WriteSpec()
      insert(native, cl)
      }
   #
   # import class specifications, transitively
   #
   repeat {
      added := 0
      every super := ((classes.foreach_t()).foreachsuper() | !
imports) do {
         if /classes.lookup(super) then {
            added := 1
            readspec(super)
            cl := classes.lookup(super)
            if /cl then halt("can't inherit class '",super,"'")
            iwrite("  inherits ", super, " from ", cl.linkfile)
            writelink(cl.dir, cl.linkfile)
            outline +:= 1
            }
        }
     if added = 0 then break
  }
  #
  # Compute the transitive closure of the superclass graph. Then
  # resolve inheritance for each class, and use it to apply
scoping rules.
  #
  every (classes.foreach_t()).transitive_closure()
  every (classes.foreach_t()).resolve()

  scopecheck_bodies(x1)

   if \thePackage then {
      every thePackage.insertsym(!package_level_syms)
      }

   #
   # generate output
   #
   yyprint(x1)
   write(yyout)
```

Other OOP Issues

The primary mechanisms for object-oriented programming that we have discussed so far include: classes, method invocation, inheritance. There were certainly a few parts we glossed over (like how a$super.m() is implemented.) The main way to look for additional issues we skipped is to read uni/unicon/idol.icn, which handles all the object-oriented features and comes from the original Idol preprocessor. Here are some thoughts from a scan of idol.icn:

- the preprocessor semi-parsed class and method headers in order to do inheritance. After the real (YACC-based) parser was added, I hoped to remove the parsing code, but it is retained in order to handle class specifications in the uniclass.dir NDBM files
- The classes in idol.icn correspond fairly directly to major syntax constructs; the compiler itself is object-oriented.
- Packages are a "virtual syntax construct": no explicit representation in the source, but stored in the uniclass.dir database
- There is a curious data structure, a tabular queue, or taque, that combines (hash) table lookup and preserves (lexical) ordering.
- Aggregation and delegation patterns are used a lot. A class is an aggregate of methods, fields, etc. and delegates a lot of its work to objects created for subparts of its overall syntax.

An Aside on Public Interfaces and Runtime Type Checking

Object-oriented facilities are usually discussed in the context of large complex applications where software engineering is an issue. We don't usually need OOP for 100 line programs, but for 10,000+ line programs it is often a big help.

Besides classes and packages, Unicon adds to Icon one additional syntax construct in support of this kind of program: type checking and coercion of parameters. Parameters and return values are the points at which type errors usually occur, during an integration phase in a large project where one person's code calls another. The type checking and coercion syntax was inspired by the type checks done by the Icon runtime system at the boundary where Icon program code calls the C code for a given function or operator.

One additional comment about types is that the lack of types in declarations for ordinary variables such as "local x" does not prevent the Icon compiler iconc from determining the exact types of well over 90% of uses at compile time using type inference. Type checking can generally be done at compile time even if variable declarations do not refer to types... as long as the type information is available across file and module boundaries.

Chapter 27: Portable 2D and 3D Graphics

Graphics facilities in Unicon Version 11 are a large component of the Unicon language. Version 11 introduces a powerful set of 3D facilities. This document describes the design and implementation internals of the 2D and 3D graphics facilities and their window system implementation. It is intended for persons extending the graphics facilities or porting Unicon to a new window system.This chapter is derived from Unicon Technical Report #5a, The Implementation of Graphics in Unicon Version 11, by Clint Jeffery and Naomi Martinez.

27.1 Window Systems and Platform-Independence

This chapter describes the internals of the implementation of Unicon's graphics and window system facilities. Much of the code is devoted to hiding specific features of C graphics interfaces that were deemed overly complex or not worth the coding effort they entail. Other implementation techniques are motivated by portability concerns. The graphics interface described below has been implemented to various levels of completeness on the X Window System, Microsoft Windows, OS/2 Presentation Manager, and Macintosh platforms. Most of this discussion is relevant also for Icon Version 9.4; Unicon's graphics facilities include minor improvements.

Relevant Source File Summary

This document assumes a familiarity with the general organization and layout of Unicon sources and the configuration and installation process. For more information on these topics, consult Icon Project Documents IPD 238 [TGJ96] and IPD 243 [TGJ98] for UNIX, and Appendix B of this document for MS Windows.

Unicon's window facilities consist of several source files, all in the runtime directory unless otherwise noted. They are discussed in more detail later in this document.

header files -- h/graphics.h contains structures and macros common across platforms. Each platform adds platform-specific elements to the common window structures defined in this file. In addition, each platform gets its own header file, currently these consist of X Windows (h/xwin.h), Microsoft Windows (h/mswin.h), OS/2 Presentation Manager (h/pmwin.h),and the Macintosh (h/mac.h). Every platform defines several common macros in the window-system specific header file in addition to its window system specific structures and macros. The common macros are used to insert platform-dependent pieces into platform-independent code.

Unicon functions -- fwindow.r contains the RTL (Run-Time Language) interface code used to define built-in graphics functions for the Unicon interpreter and compiler. For most functions, this consists of type checking and conversion code followed by calls to platform-dependent graphics functions. The platform dependent functions are described later in this document; fwindow.r is platform independent. You will generally modify it only if you are adding a new built-in function. For example, the Windows native functions are at the bottom of this file.

internal support routines -- rwindow.r, rwinrsc.r, rgfxsys.r and rwinsys.r are basically C files with some window system dependencies but mostly consisting of code that is used on all systems. For example, rwindow.r is almost 100 kilobytes of portable source code

related to Unicon's event model, attribute/value system, portable color names, GIF and JPEG image file support, palettes, fonts, patterns, spline curves and so forth.

window-system specific files -- Each window system gets its own source files for C code, included by the various r*.r files in the previous section. Currently these include rxwin.ri and rxrsc.ri for X Window; rmswin.ri for MS Windows; rpmwin.ri, rpmrsc.ri, and rpmgraph.ri for Presentation Manager; and rmac.ri for the Macintosh. Each platform will implement one or more such r*.ri files. In addition, common/xwindow.c contains so many X Window includes that it won't even compile under UNIX Sys V/386 R 3.2 if all of the Unicon includes are also present -- so its a .c file instead of a .r file.

tainted "regular" Unicon sources -- Many of the regular Unicon source files include code under #ifdef Graphics and/or one or more specific window system definitions such as #ifdef XWindows or #ifdef PresentationManager. The tainted files that typically have to be edited for a new window system include h/grttin.h, h/features.h, h/rexterns.h, h/rmacros.h, h/rproto.h, h/rstructs.h, and h/sys.h. Other files also contain Graphics code. This means that most of the system has to be recompiled with rtt and cc after Graphics is defined in h/define.h. You will also want to study the Graphics stuff in h/grttin.h since several profound macros are there. Also, any new types (such as structures) defined in your window system include files will need dummy declarations (of the form typedef int foo;) to be added there.

Under UNIX the window facilities are turned on at configuration time by typing

```
make X-Configure name=system
```

instead of the usual make Configure invocation. The X configuration modifies makefiles and defines the symbolic constant Graphics in h/define.h. If OpenGL libraries are detected, configuration enables them automatically. Similar but less automatic configuration handling is performed for other systems; for example, an alternate .bat file is used in place of os2.bat or turbo.bat.

Graphics #define-d symbols

The primary, window-system-independent defined symbol that turns on window facilities is simply Graphics. Underneath this parent #ifdef, the symbol XWindows is meant to mark all X Window code. Other window systems have a definition comparable to XWindows: for Microsoft Windows it is MSWindows, for OS/2 it is PresentationManager, and for the Macintosh, MacGraph. Turning on any of the platform specific graphics #define symbols turns on Graphics implicitly.

27.2 Structures Defined in graphics.h

The header file graphics.h defines a collection of C structures that contain pointers to other C structures from graphics.h as well as pointers into the window system library structures. The internals for the simplest Unicon window structure under X11 are depicted in Figure 1. The picture is slightly simpler under MS Windows, with no display state or related color or font management; on the other hand MS Windows maps the Unicon context onto a large set of resources, including pens, brushes, fonts and bitmaps.

Figure 1: Internal Structure of an Unicon Window Value

At the top, Unicon level, there is a simple structure called a *binding* that contains a pointer to a window state and a window context. Pointers to bindings are stored in the FILE * variable of the Unicon file structure, and most routines that deal with a window take a pointer to a binding as their first argument. Beneath this facade, several structures are accessed to perform operations on each window.

The window state holds the typical window information (size, text cursor location, an Unicon list of events waiting to be read) as well as a window system pointer to the actual window, a pointer to a backing pixmap (a "compatible device context" used to handle redraw requests), and a pointer to the display state.

The window context contains the current font, foreground, and background colors used in drawing/writing to the window. It also contains drawing style attributes such as the fill style. Contexts are separate from the window state so that they may be shared among windows. This is a big win, and Unicon programs tend to use it heavily, so in porting the window functions a central design issue must be the effective use of a comparable facility on other window systems, or emulating the context abstraction if necessary. Nevertheless, one might start out with Couple() and Clone() disabled and only allow one hardwired context associated with each window.

The display state contains whatever system resources (typically pointers or handles) that are shared among all the windows on a given display in the running program. For example, in X this includes the fonts, the colors, and a window system pointer for an internal Display structure required by all X library calls to denote the connection to the X server.

27.3 Platform Macros and Coding Conventions

Since the above structure is many layers deep and sometimes confusing, Unicon's graphics interface routines employ coding conventions to simplify things. In order to avoid many extra memory references in accessing fields in the multi-level structure, "standard" local variables are declared in most of the platform dependent interface routines in rxwin.ri and rmswin.ri. The macro STDLOCALS(w) declares local variables pointing to the most commonly used pieces of the window binding, and initializes them from the supplied argument; each window system header should define an appropriate STDLOCALS(w) macro. Under some window systems, such as MS Windows, STDLOCALS(w) allocates resources which must be freed before execution continues, in which case a corresponding FREE_STDLOCALS(w) macro is defined.

Some common standard locals are wc, ws, stdwin, and stdpix. While wc, and ws are pointers to structures copied from the window binding, stdwin, and stdpix are actual X (or MS Window) resources that are frequently supplied to the platform-dependent routines as arguments. Each window system will have its own standard locals. For example, MS Windows adds stddc and pixdc since it uses a device context concept not found in X11.

In much of the platform-dependent source code, the window system calls are performed twice. This is because most platforms including X, MS Windows, and PresentationManager do not remember the contents of windows when they are reduced to iconic size or obscured behind other windows. When the window is once again exposed, it is sent a message to redraw itself. Unicon hides this entirely, and remembers the

contents of the window explicitly in a window-sized bitmap of memory. The calling of platform graphics routines twice is so common that a set of macros is defined in xwin.h to facilitate it. The macros are named RENDER2 through RENDER6, and each of them takes an Xlib function and then some number of arguments to pass that function, and then calls that function twice, once on the window and once on the bitmap.

Platforms that provide backing store may avoid this duplicated effort. In practice however it seems most window systems have redraw events even if they implement retained windows (for example, MGR [Uhler88]).

27.4 Window Manipulation in rxwin.ri and rmswin.ri

The platform-dependent calls in the Unicon run-time system can be categorized into several major areas:

- Window creation and destruction
- Low-level event processing
- Low-level text output operations
- Window and context attribute manipulation

Window Creation and Destruction

A graphics window is created when the Unicon program calls open() with file attribute "g". The window opening sequence consists of a call to wopen() to allocate appropriate Unicon structures for the window and evaluate any initial window attributes given in additional arguments to open(). After these attributes have been evaluated, platform resources such as fonts and colors are allocated and and the window itself is instantiated. Under X, wopen() busy-waits until the window has received its first expose event, ensuring that no subsequent window operation takes place before the window has appeared onscreen.

A window is closed by a call to wclose(); this removes the on-screen window even if other bindings (Unicon window values) refer to it. A closed window remains in memory until all Unicon values that refer to it are closed. A call to unbind() removes a binding without necessarily closing the window.

Event Processing

The system software for each graphics platform has a huge number of different types of events. Unicon ignores most of them. Of the remainder, some are handled by the runtime system code in the .ri files implicitly, and some are explicitly passed on to the Unicon program.

Most native graphic systems require that applications be event-driven; they must be tightly I/O bound around the user's actions. The interaction between user and program must be handled at every instant by the program. Unicon, on the other hand, considers this event-driven model optional.

Making the event-driven model optional means that the Unicon interface must occasionally read and process events when the Unicon program itself is off in some other computation. In particular, keystrokes and mouse events must be stored until the user

requests them, but exposure events and resizes must be processed immediately. The Unicon interpreter pauses at regular intervals in between its virtual machine instructions (the Unicon compiler emits polling code in its generated C code, so window system facilities are supported by the compiler as well) and polls the system for events that must be processed; this technique fails when no virtual machine instructions are executing, such as during garbage collections or when blocked on file I/O.

On some platforms such as X, this probably could be done using the platform event queue manipulation routines. Instead, the Unicon window interface maintains its own keystroke and mouse event queue from which the Unicon functions obtain their events. This additional queue makes the implementation more portable. Various window systems probably do not support queue manipulation to the extent or in the same way that X does. It also provides the programmer with a higher level event processing abstraction which has proven useful.

Window resizing is partly handled by the interface. The old contents of the window are retained in their original positions, but the program is informed of the resize so it can handle the resize in a more reasonable manner. As has already been noted exposure events are hidden entirely via the use of a backing pixmap with identical contents for each window. The pixmap starts out the same size as the window. It is resized whenever the window grows beyond one of its dimensions. It could be reduced whenever the window shrinks, but then part of the window contents would be lost whenever the user accidentally made the window smaller than intended.

The platform-dependent modules also contains tables of type stringint. These tables are supported by routines that map strings for various attributes and values to native window system integer constants. Binary search is employed. This approach is a crude but effective way to provide symbolic access "built-in" to the language without requiring include files. Additional tables mapping strings to integers are found in the platform independent source files.

Resource Management

One of the most important tasks performed by platform-specific graphics functions is the management of resources, both the on-screen resources (windows) and the drawing context elements used by the window system in performing output.

Memory Management and r*rsc.ri Files

Memory management for internal window structures is independent of Unicon's standard memory management system. Xlib memory is allocated using malloc(2).

Most internal Unicon window structures could be allocated in Unicon's block region, but since they are acyclic and cannot contain any pointers to Unicon values, this would serve little purpose (Actually, it is probably the right thing to do, and will probably happen some day, but its just not in the cards right now unless you feel like messing with the garbage collector.). In addition when an incoming event is being processed it has to be matched up with the appropriate window state structure, so some of the window structures must be easily reached, not lost in the block region. The window interface structures are reference counted and freed when the reference count reaches 0.

Color Management

Managing colors under X Windows is painful. In particular, if the same color is allocated twice the color table entry is shared (which is good) and that entry may only be freed once (which is bad). For this reason, every color allocated by Unicon is remembered and duplicate requests are identified and freed only once. In the general case it is impossible to detect when a particular color is no longer being displayed, and so colors are only freed on window closure or when a window is cleared.

Font Management

Unicon supports a portable font name syntax. Since the available fonts on systems vary widely, "interesting" code has been written to support these portable names on various X servers. Each window system may need to include heuristics to pick an appropriate font in the font allocation routine in the window system's r*.ri file.

27.6 External Image Files and Formats

Reading and writing window contents to external files is accomplished by the routines loadimage() and dumpimage(), implemented in each platform's window system specific file, such as rxwin.ri. These routines take a window binding and a string filename and perform the I/O transfer. Presently, the file format is assumed to be indicated by the filename extension; this is likely to change. Ideally Unicon should tolerate different file formats more flexibly, inferring input file formats by reading the file header where possible, and running external conversion programs where appropriate.

GIF and JPEG files are self-identifying, so they are always recognized independent of name. They are checked in system-independent code before platform-specific image reading code is invoked.

27.7 Implementation of 3D Facilities

In order to implement the 3D facilities, the Unicon runtime system was modified to support 2D and 3D windows. The Unicon runtime system is written in a custom superset dialect of C called RTL [Walker94]. The 3D facilities use the existing 2D facilities code for window creation and destruction, as well as handling keyboard and mouse input.

3D Facilities Requirements

OpenGL 1.2 or later must be present on the system in order for Unicon's 3D graphics facilities to work. A check for this is performed in wopengl() which can be found in the file ropengl.ri. The requirement of OpenGL 1.2 is based on the fact that the function glTexBind(), which make the implementation of textures more efficient, is only available in OpenGL 1.2 and later.

Also needed for the Unicon 3D graphics facilities is a system that supports a true color visual with a depth buffer of 16 and a double buffer. The requirement of a depth buffer is a necessity to implement lighting. For lighting to work properly in OpenGL, a depth test must be performed, hence the need of a depth buffer. A double buffer is needed to implement the list structure that is used to redraw a window. More information can be found on redrawing of windows in section 7.3.

Files

Several existing files contain extensions to support the Unicon 3D graphics facilities under #ifdef Graphics3D, including data.r (new runtime error codes), fwindow.r (new 3D functions), rmemmgt.r (3D window display lists), rxwin.ri and rmswin.ri (modified wopen() and wmap() to support 3d mode), rwindow.r (new 3D attributes), and graphics.h (new 3D fields in canvas and context structures). Also a new file, ropengl.ri was added that contains the C helper functions for functions in fwindow.r, rxwin.ri, and rwindow.r.

Redrawing Windows

In the 2D graphics facilities, events that require the redrawing of a window are handled by using a pixmap. Instead of using a pixmap, for the Unicon 3D graphics facilities, a Unicon list of lists is created for each window opened in "gl" mode. This list of lists keeps track of all objects in a 3D graphics scene. This list is called funclist and is found in the wstate structure of a "gl" window. The individual lists of contain the function name and the parameters of that function. Also placed on the list are attributes that affect the scene. These include dim, linewidth, texcoord, texture, texmode, and fg. When a window receives an event that requires redrawing, the window is cleared, all attributes are reset to the defaults, and the Unicon list of lists is traversed to redraw every object in the scene.

There are some functions and attributes that are not placed in the list. Instead they much either modify the list or call the list to redraw the scene. The function EraseArea(), not only clears the screen but also clears the contents of the list. The attributes light0-light7, eye, eyeup, eyedir, and eyepos use the list to redraw the window with the new attributes. So if the position of a light changes, the new lighting calculations are preformed and the scene is redraw. Besides these functions and attributes, every function or attribute available in the 3D graphics facilities is placed on this list. In is important to note that functions and attributes that have no effect in the 3D graphics facilities are not placed in this list.

Textures

In OpenGL, textures can be one, two, or three-dimensional and are represented as multi-dimensional arrays. In the Unicon 3D graphics facilities all texture are 2D dimensional, and represented as three-dimensional arrays. This array keeps track of the position and red, green, and blue components of each pixel in the texture image. When a texture image is specified in a Unicon program, the texture is converted from the Unicon internal representation of the image to a three-dimensional array. For most cases, this does not take a long time, but as a texture image gets larger, the slower the application will run. Several measures have been taken in order to increase the efficiency of converting the texture image into an array. Since lighting and texturing are fairly expensive operations, especially if several lights are activated, these features are temporarily deactivated. Despite these efforts, converting a "g" window to a texture is still fairly expensive. Possible future work includes ways to speed up this process.

Instead of adding a texture to the list of lists as described in section 7.3, OpenGL's internal texture resources are used. OpenGL assigns to each texture a name. The names assigned to each texture in a Unicon scene are stored in texname[], which can be found in a "gl" window's context. To ensure that a texture name is not reused, a call to glGenTextures() made which produces unused texture names. When a texture is applied to the scene, only the index of the array texname[] is stored in the list. When the list is

traversed, a call to glBindTexture() is made which binds the texture to the subsequent objects in the scene. One problem of using this representation of textures is that it places an upper bound on the number of texture used. This is because glGenTextures() requires the number of texture names to generate. Also by using glBindTexture(), never deletes a texture from the texture resources, possibly using up all texture resources. Future work might look into when to delete a texture an ways to check when the texture resources have been used up.

Texture Coordinates

The primitives as mentioned in previous sections are cubes, tori, cylinders, disks, partial disks, points, lines, polygons, spheres, line segments, and filled polygons. Some of these primitives are drawn using different aspect of the OpenGL library, with some using the glu library. Points, lines, line segments, polygons, and filled polygons are drawing using glBegin(), glEnd(), and vertices in between the function calls. Cylinders, disks, partial disks, and spheres are implemented using the glu library. They are considered to be gluQuadrics objects. Finally cubes and tori are a composition of several polygons.

The texturing method used is influenced by the how the primitive is composed. For the primitives built using the OpenGL library, default texture coordinates are obtain much differently than those primitives built using the glu library. For those primitives built using glBegin() and glEnd(), glTexGen() is used to determine the default parameters when "texcoord=auto". In order to use this feature we must enable GEN_S and GEN_T with glEnable(). This generates texture coordinates for a 2D textures. The texture coordinates for a torus are obtained in the same ways.

Primitives built using the glu library, have texture coordinates associated with them. These texture coordinates can be obtained by calling gluQuadricTexture().The use of the glu texture coordinates verses the OpenGL coordinates, is due to the fact that the glu texture coordinate look nicer. In order to use these texture coordinates instead of the ones specified by OpenGL, it is necessary to disable GEN_S and GEN_T. After the object has been drawn, GEN_S and GEN_T are turned back on.

A cube uses default texture coordinates that map the texture onto each of the faces of a cube. In order to use these default coordinates, it is necessary to disable GEN_S and GEN_T, similar to glu objects.

27.8 Graphics Facilities Porting Reference

This section documents the window-system specific functions and macros that generally must be implemented in order to port Unicon's graphics facilities to a new window system. The list is compiled primarily by studying fwindow.r, rwindow.r, and the existing platforms.

A note on types: w is a window binding pointer (wbp), the top level Unicon "window" value. i is an integer, s is a string. wsp is the window state (a.k.a. canvas) pointer, and wcp is the window context pointer. A bool return value returns one of the C macro values Succeeded or Failed, instead of the usual C booleans 1 and 0.

Convert from radians into window system units. For example, under X these are 1/64 of a degree integer values, while under PresentationManager it converts to units of 1/65536 of a degree in a fixed-point format.

ARCHEIGHT(arc)

The height component of an XArc.

ARCWIDTH(arc)

The width component of an XArc.

ASCENT(w)

Returns the number of pixels above the baseline for the current font. Note that when Unicon writes text, the (x,y) coordinate gives the left edge of the character at its baseline; some window systems may need to translate our coordinates.

int blimage(w, x, y, width, height, ch, s, len)

Draws a bi-level (i.e. monochrome, 1-bit-per-pixel) image; used in DrawImage() which draws bitmap data stored in Unicon strings.

wcp clone_context(w)

Allocate a new context, cloning attributes from w's context.

COLTOX(w, i)

Return integer conversion from a 1-based text column to a pixel coordinate.

copyArea(w1, w2, x, y, width, height, x2, y2)

Copies a rectangular block of pixels from w1 to w2.

DESCENT(w)

Returns the number of pixels below the baseline for the current font.

DISPLAYHEIGHT(w)

Return w's display (screen) height in pixels.

DISPLAYWIDTH(w)

Return w's display width in pixels.

bool do_config(w, i)

Performs move/resize operations after one or more attributes have been evaluated. Config is a word with two flags: the one bit indicates a move, the two bit indicates a resize. The desired sizes are in the window state pointer, e.g. w->window->width.

drawarcs(w, thearcs, i)

Draw i arcs on w, given in an array of XArc structures. Define an appropriate XArc structure for your window system; it must include fields x, y and width and height fields accessible through macros ARCWIDTH() and ARCHEIGHT(). Also, a starting angle angle1 and arc extent angle2, assigned through macros ANGLE(), EXTENT(), and

FULLARC. This is currently a mess. Imitation of the X or PresentationManager code is in order.

---

drawlines(w, points, i)

Draw i-1 connected lines, connecting the dots given in points.

---

drawpoints(w, points, i)

Draw i points.

---

drawsegments(w, segs, i)

Draw i disconnected line segments; define an Xsegment structure appropriate do your window system, consisting of fields x1, y1, x2, y2. This type definition requirement should be cleaned up someday.

---

drawstring(w, x, y, s, s_len)

Draw string s at coordinate (x,y) on w. Note that y designates a baseline, not an upper-left corner, of the string.

---

drawrectangles(w, rectangles, i)

Draw i rectangles. Define an XRectangle structure appropriate to your window system.

---

int dumpimage(w, s, x, y, width, height)

Write an image of a rectangular area in w to file s. Returns Succeeded, Failed, or NoCvt if the platform doesn't support the requested format. Note that this is the "platform- dependent image writing function"; requests to write GIF or JPEG are handled outside of this function.

---

eraseArea(w, x, y, width, height)

Erase a rectangular area, that is, set it to the current background color. Compare with fillrectangles().

EXTENT(a)

Convert from radians into window system units, e.g. under PresentationManager it converts to units of 1/65536 of a circle and does some weird type conversion.

---

fillarcs(w, arcs, i)

Fill wedge-like arc sections (pie pieces). See drawarcs().

---

fillrectangles(w, rectangles, i)

Fill i rectangles. See drawrectangles().

---

fillpolygon(w, points, i)

Fill a polygon defined by i points. Connect first and last points if they are not the same.

FHEIGHT(w)

Returns the pixel height of the current font, hopefully ASCENT + DESCENT.

free_binding(w)

Free binding associated with w. This gets rid of a binding that refers to w, without necessarily closing the window itself (other bindings may point to that window).

free_context(wc)

Free window context wc.

free_mutable(w, i)

Free mutable color index i.

free_window(ws)

Free window canvas ws.

freecolor(w, s)

Free a color allocated on w's display.

FS_SOLID

Define this to be the window system's solid fill style symbol.

FS_STIPPLE

Define this to be the window system's stippled fill style symbol.

FULLARC

Window-system value for a complete (360 degree) circle or arc.

FWIDTH(w)

Returns the pixel width of the widest character in the current font.

wsp getactivewindow()

Return a window state pointer to an active window, blocking until a window is active. Probably will be generalized to include a non-blocking variant. Returns NULL if no windows are opened.

getbg(w, s)

Returns (writes into s) the current background color.

getcanvas(w, s)

Returns (writes into s) the current canvas state.

getdefault(w, s_prog, s_opt, s)

Get any window system defaults for a program named s_prog resource named s_opt, write result in s.

getdisplay(w, s)

Write a string to s with the current display name.

getdrawop(w, s)

Return current drawing operation, one of various logical combinations of source and destination bits.

---

getfg(w, s)

Returns (writes into s) the current foreground color.

---

getfntnam(w, s)

Returns (writes into s) the current font. This interface may get changed since a portable font naming mechanism is to be installed. Name is presently always prefixed by "font=" (pretty stupid, huh); must be an artifact of merging window system ports, will be changed.

---

geticonic(w, s)

Return current window iconic state in s, could "iconify" or whatever. Obsolete (subsumed by canvas attribute, getcanvas()).

---

geticonpos(w, s)

Return icon's position to s, an encoded "x,y" format string.

---

int getimstr(w, x, y, width, height, paltbl, data)

Gets an image as a string. Used in GIF code.

---

getlinestyle(w, s)

Return current line style, one of solid, dashed, or striped.

---

get_mutable_name(w, i)

Returns the string color name currently associated with a mutable color.

---

getpattern(w, s)

Return current fill pattern in s.

---

getpixel(w, x, y, long *rv)

Assign RGB value for pixel (x,y) into *rv.

---

getpixel_init(w, struct imgmem *imem)

Prepare to fetch pixel values from window, obtaining contents from server if necessary. This function does all the real work used by subsequent calls to getpixel().

---

getpointername(w, s)

Write mouse pointer appearance, by name, to s.

---

getpos(w)

Update the window state's posx and posy fields with the current window position.

---

getvisual(w, s)

Write a string to s that explains what type of display w is on, e.g. "visual=x,y,z", where x is a class, y is the bits per pixel, and z is number of colormap entries available. This X-specific anachronism is likely to go away.

HideCursor(wsp ws)

Hide the text cursor on window state ws.

ICONFILENAME(w)

Produce char * for window's icon image file name if there is one.

ICONLABEL(w)

Produce char * for icon's title if there is one.

---

isetbg(w, i)

Set background color to mutable color table entry i. Mutable colors are not available on all display types.

---

isetfg(w, i)

Set foreground color to mutable color table entry i. Mutable colors are not available on all display types.

ISICONIC(w)

Return 1 if the window is presently minimized/iconic, 0 otherwise.

ISFULLSCREEN(w)

Return 1 if the window is presently maximized/fullscreen, 0 otherwise.

ISNORMALWINDOW(w)

Return 1 if the window is neither minimized nor maximized, 0 otherwise.

LEADING(w)

Return current integer leading, the number of pixels from line to line.

LINEWIDTH(w)

Return current integer line width used during drawing.

---

lowerWindow(w)

Lower the window to the bottom of the stack.

---

mutable_color(w, dptr dp, i, C_integer *result)

Allocate a mutable color from color spec given by dp and i, placing result (a small negative integer) in *result.

---

nativecolor(w, s, r, g, b)

Interpret a platform-specific color name s (define appropriately for your window system). Under X, we can do this only if there is a window.

pollevent()

Poll for available events on all opened displays. This is where the interpreter calls the window system interface. Return a -1 on an error, otherwise return count of how long before it should be polled (400).

query_pointer(w, XPoint *xp)

Produce mouse pointer location relative to w.

query_rootpointer(XPoint *xp)

Produce mouse pointer location relative to root window on default screen.

raiseWindow(w)

Raise the window to the top of the stack.

bool readimage(w, s, x, y, int *status)

Read image from file s into w at (x,y). Status is 0 if everything was kosher, 1 if some colors weren't available but the image was read OK; if a major problem occurs it returns Failed. See loadimage() for the real action.

rebind(w, w2)

Assign w's context to that of w2.

RECHEIGHT(rec)

The height component of an XRectangle. Gets "fixed up" (converted) into a Y2 value if necessary, in window system specific code.

RECWIDTH(rec)

The width component of an XRectangle. Gets "fixed up" (converted) into a X2 value if necessary, in window system specific code.

RECX(rec)

The x component of an XRectangle.

RECY(rec)

The y component of an XRectangle.

ROWTOY(w, i)

Return integer conversion from a 1-based text row to a pixel coordinate.

SCREENDEPTH(w)

Returns the number of bits per pixel.

int setbg(w, s)

Set the context background color to s. Returns Succeeded or Failed.

setcanvas(w, s)

Set canvas state to s, make it "iconic", "hidden" or whatever. A canvas value extension such as fullscreen would go here. Changes in canvas state are tantamount to destroying the old window, creating a new window (with appropriate size and style) and adjusting the pixmap size correspondingly. Much of the associated logic, however, might be located in the event handlers for related window system events.

---

setclip(w)

Set (enable) clipping on w from its context.

---

setcursor(w, i)

Turn text cursor on or off. Text cursor is off (invisible) by default.

---

setdisplay(w, s)

Set the display to use for this window; fails if the window is already open somewhere.

---

setdrawop(w, s)

Set drawing operation to one of various logical combinations of source and destination bits.

---

int setfg(w, s)

Set the context foreground color to s. Returns Succeeded or Failed.

---

setfillstyle(w, s)

Set fill style to solid, masked, or textured.

---

bool setfont(w, char **s)

Set the context font to s. This function first attempts to use the portable font naming mechanism; it resorts to the system font mechanism if the name is not in portable syntax.

---

setgamma(w, gamma)

Set the context's gamma correction factor.

---

setgeometry(w, s)

Set the window's size and/or position.

---

setheight(w, i)

Set window height to i, whether or not window is open yet.

---

seticonicstate(w, s)

Set window iconic state to s, it could be "iconify" or whatever. Obsolete; setcanvas() is more important.

---

seticonimage(w, dptr d)

Set window icon to d. Could be string filename or existing pixmap (i.e. another window's contents). Pixmap assignment no longer possible, so one could simplify this to just take a string parameter.

---

seticonlabel(w, s)

Set icon's string title to s.

---

seticonpos(w, s)

Move icon's position to s, an encoded "x,y" format string.

---

setimage(w, s)

Set an initial image for the window from file s. Only valid during open().

---

setleading(w, i)

Set line spacing to i pixels from line to line. This includes font height and external leading, so i < fontheight means lines draw partly over preceding lines, i > fontheight means extra spacing.

---

setlinestyle(w, s)

Set line style to solid, dashed, or striped.

---

setlinewidth(w, i)

Set line width to i.

---

set_mutable(w, i, s)

Set mutable color index i to color s.

SetPattern(w, s, s_len)

Set fill pattern to bits given in s. Fill pattern is not used unless fillstyle attribute is changed to "patterned" or "opaquepatterned".

SetPatternBits(w, width, bits, nbits)

Set fill pattern to bits given in the array of integers named bits. Fill pattern is not used unless fillstyle attribute is changed to "patterned" or "opaquepatterned".

---

setpointer(w, s)

Set mouse pointer appearance to shape named s.

---

setpos(w, s)

Move window to s, a string encoded "(x,y)" thing.

---

setwidth(w, i)

Set window width to i, whether or not window is open yet.

---

setwindowlabel(w, s)

Set window's string title to s.

ShowCursor(wsp ws)

Show the text cursor on window state ws.

---

int strimage(w, x, y, width, height, e, s, len)

Draws a character-per-pixel image, used in DrawImage(). See blimage().

SysColor

Define this type to be the window system's RGB color structure.

TEXTWIDTH(w, s, s_len)

Returns the integer text width of s using w's current font.

---

toggle_fgbg(w)

Swap the foreground and background on w.

---

unsetclip(w)

Disable clipping on w from its context.

UpdateCursorPos(wsp ws, wcp wc)

Move the text cursor on window state ws and context wc.

---

walert(w, i)

Sounds an alert (beep). i is a volume; it can range between -100 and 100; 0 is normal.

---

warpPointer(w, x, y)

Warp the mouse location to (x,y).

---

wclose(w)

Closes window w. If there are other bindings that refer to the window, they are converted into pixmaps, i.e. the window disappears but the canvas is still there and can be written on and copied from.

---

wflush(w)

Flush output to window w; a no-op on some systems.

---

wgetq(w, dptr result)

Get an event from w's pending queue, put results in descriptor *res. Returns -1 for an error, 1 for success (should fix this).

WINDOWLABEL(w)

Produce char * for window's title if there is one.

FILE *wopen(s, struct b_list *lp, dptr attrs, i, int *err_index, is_3d)

Open window named s, with various attributes. This ought to be merged from various window system dependent files, but presently each one defines its own. Copy and modify from rxwin.ri or rmswin.ri. The return value is really a wbp, cast to a FILE *.

---

wputc(c, w)

Draw character c on window w, interpret newlines, carriage returns, tabs, deletes, backspaces, and the bell.

wsync(w)

Synchronize server and client (a no-op on some systems).

xdis(w, s, s_len)

Draw string s on window w, low-level.

XTOCOL(w, i)

Return integer conversion from a 0-based pixel coordinate to text column.

YTOROW(w, i)

Return integer conversion from a 0-based pixel coordinate to text row.

26.9 The X Implementation

The reference implementation of Unicon's graphics facilities is written in terms of Xlib, the lower-level X Window C interface [Nye88]. It does not use the X resource manager. The end result of these two facts is that the implementation is relatively visible: the semantics are expressed fairly directly in the source code. Although it is necessary to understand the semantics of the underlying X routines, hidden behavior has been minimized.

Unicon does not rely on the X Toolkit Intrinsics (Xt) or any higher level widget set such as Motif. This guarantees that Unicon will compile and run on any X11 platform. Unicon programs implement their own look and feel, which may or may not be consistent with the other applications on a given X workstation. The Unicon Program Library includes routines that implement user interface components with an appearance that is similar to Motif.

The X implementation employs the XPM X pixmap library if it is available; XPM is a proposed extension to Xlib for storing color images in external files [LeHors91]. XPM provides color facilities analogous to the built-in X black-and-white bitmap routines. In addition to the image formats native to each platform, Unicon also supports GIF and JPEG as portable image file formats.

26.10 The MS Windows Implementation

The Microsoft Windows implementation of Unicon is written using Win32, the lower-level 32-bit Windows API. It does not use the Microsoft Foundation Classes. This makes it easier to build with different C compilers, and easier to port to different Windows implementations, such as Windows CE.

Installing, Configuring, and Compiling the Source Code

Building Unicon for Windows Version 11.0 requires Mingw32 GCC 2.95.2. Newer versions of Windows GCC might be made to work, but thusfar have produced non-working executables. We hope to add Cygwin GCC support in the future. The sources may also build with modest revision under MS Visual C++ 2.0 or newer. I have built earlier versions with MSVC versions 2, 5, and 6. I encourage you to try building using other compilers, and send me your configuration files. You will need a robust Win32

platform to compile these sources; the build scripts and "make" process tend to fail on older versions of Windows.

1. Unpack the sources.

Unpack uni.zip in such a way that it preserves its subdirectory structure. Unzip.exe is recommended rather than WinZip. See Icon Project Document 243 [ipd243] for a picture of the directory hierarchy. In particular, there should be a BIN directory along with the SRC directory under the unicon/ directory.

2. Configure the sources.

Run "make W-Configure-GCC" (or "make W-Configure" under MSVC) to configure your sources to build wiconx and wicont, the Unicon virtual machine interpreter, and the Unicon bytecode compiler, with graphics facilities enabled.

3. Compile to make executables.

Run "make Unicon" to build the currently-configured binary set. It is worth discussing why I provide makefiles instead of a project file for use in the Visual C++ IDE. The reason is that the source files for the Unicon virtual machine interpreter (generically called iconx; wiconx.exe in this case) are written in an extended dialect of ANSI C called RTL [ipd261]. Files in this language have the extension .r instead of .c and .ri instead of .h. During compilation, a program called rtt (the run time translator) translates .r* files into .c files. If someone wants to show me how to insert this step into the Visual C++ IDE build process, I would be happy to use their IDE. You can write project files for the other C programs that make up the Unicon system, but most modifications to the language are changes to the interpreter.

Notes on the MS Windows internal functions

The functions documented here are those most likely to be involved in projects to add features to Windows Unicon.

---

handle_child(w, UINT msg, WPARAM wp, LPARAM lp)

This procedure handles messages from child window controls such as buttons. In many cases, this enqueues an event on the Unicon window.

---

int playmedia(w, char *s)

This crude function will call one of several multimedia functions depending on whether s is the name of a multimedia file (.wav, .mid, .rmi are supported) or an MCI command string.

---

int getselection(w, char *s)

Return the current contents of the clipboard text. The design of this and setselection() need to be broadened a bit to support images.

---

int setselection(w, char *s)

Set the clipboard text to s.

Chapter 28: Networking, Messaging and the POSIX Interface

Unicon's system interface is greatly enriched compared with Icon, primarily in that it treats Internet connections and Internet-based applications as ubiquitous, and extends the file type with appropriately high-level capabilities. Fundamental TCP and UDP connections are a breeze using the networking facilities, and common application-level protocols are supported via the messaging facilities (see also the X11 graphics facilities and the SQL/ODBC database facilities for examples where application-level networking is provided in Unicon). Portions of this chapter related to the messaging facilities were contributed by their author, Steve Lumos.

28.1 Networking Facilities

...

28.2 Messaging Facilities

The Transfer Protocol Library

All of the message facilities are handled by the transfer protocol library (libtp). This library provides an abstraction of the many different protocols (HTTP, SMTP, etc) into a clear and consistent API. Ease of adding support for new protocols and porting the entire library to new operating system interfaces were primary design goals. These goals are both accomplished by using the AT&T Labs discipline and method (DM) architecture described below.

Libtp Architecture

The key feature of the DM architecture is that it makes explicit two interfaces in the library: *disciplines* which hold system resources and define routines to acquire and manipulate them, and *methods* which define the higher-level algorithms used to access these resources. This model fits the problem of Internet transfer protocols nicely; the discipline abstracts the operating system interface to the network, and there is a method for each protocol that defines communication with a server only in terms of the discipline.

This architecture makes porting easy because you need only create a discipline for the new system, which means writing 9 functions. The only currently-existing discipline handling both the Berkeley Socket and WINSOCK APIs is only 400 lines long. Once a discipline exists, the new system immediately gains all of the supported protocols.

The Discipline

The discipline is a C structure whose members are pointers to functions:

```
typedef struct _tp_disc_s  Tpdisc_t;   /* discipline */

typedef int                (*Tpconnect_f)(char* host, u_short
port, Tpdisc_t* disc);
typedef int                (*Tpclose_f)(Tpdisc_t* disc);
typedef ssize_t  (*Tpread_f)(void* buf, size_t n, Tpdisc_t* disc);
```

```
typedef ssize_t (*Tpreadln_f)(void* buf, size_t n, Tpdisc_t*
disc);
typedef ssize_t (*Tpwrite_f)(void* buf, size_t n, Tpdisc_t*
disc);
typedef void*         (*Tpmem_f)(size_t n, Tpdisc_t* disc);
typedef int                (*Tpfree_f)(void* obj, Tpdisc_t*
disc);
typedef int                (*Tpexcept_f)(int type, void* obj,
Tpdisc_t* disc);
typedef Tpdisc_t* (*Tpnewdisc_f)(Tpdisc_t* disc);

struct _tpdisc_s
{
     Tpconnect_f     connectf;  /* establish a connection */
     Tpclose_f       closef;    /* close the connection */
     Tpread_f        readf;     /* read from the connection */
     Tpreadln_f readlnf;   /* read a line from the connection */
     Tpwrite_f       writef;    /* write to the connection */
     Tpmem_f              memf; /* allocate some memory */
     Tpfree_f        freef;     /* free memory */
     Tpexcept_f exceptf;   /* handle exception */
     Tpnewdisc_f     newdiscf;  /* deep copy a discipline */
     int                  type; /* (not used currently) */
};
```

These functions define a complete API for acquiring and manipulating all of the system resources needed by all of the methods and (it is hoped) any conceivable method. By convention, every discipline function takes a pointer to the current discipline as its last argument. (Every method function takes a library handle which contains a pointer to the current discipline, so the discipline functions are always available when needed.) The Tpdisc_t is an abstract discipline. In practice, a new discipline will extend Tpdisc_t by at minimum adding some system dependent data such as a Unix file descriptor or Windows SOCKET*. Here is the "Unix" discipline (it would be better called the socket discipline since it works for the Berkeley Socket API and WINSOCK on multiple systems):

```
struct _tpunixdisc_s
{
   Tpdisc_t tpdisc;
   int fd;
}
```

Exception Handling

The DM archtecture defines a very useful convention for exception handling. Exceptions are passed as integers to the exceptf function along with some exception-specific data. The function can do arbitrary processesing and then return {-1, 0, 1}, which instructs the library to retry the operation (1), return an error to the caller (-1), or take some default action (0). Libtp uses constants TP_TRYAGAIN, TP_RETURNERROR, and TP_DEFAULT.

Although not as powerful as languages with true exceptions, the DM exception handling definitely serves to make the code more readable. In the Unix discipline, exceptf is used to aggregate all of the many, sometimes transient errors that can occur in network programming. For example, the Unix discipline's readf function is:

```
ssize_t unixread(void* buf, size_t n, Tpdisc_t* tpdisc)
```

```
{
  Tpunixdisc_t* disc = (Tpunixdisc_t*)tpdisc;

  size_t  nleft;
  ssize_t nread;
  char*   ptr = buf;

  nleft = n;
  while (nleft > 0) {
    if ((nread = read(disc->fd, ptr, nleft)) <= 0) {
      int action = tpdisc->exceptf(TP_EREAD, &nread, tpdisc);
      if (action > 0) {
        nread = 0;
        continue;
      }
      else if (action == 0) {
        break;
      }
      else {
        return (-1);
      }
    }

    nleft -= nread;
    ptr += nread;
  }
  return (n – nleft);
}
```

The Unix read() system call can return a positive number, indicating the number of bytes read, a negative number, indicated error, or zero, if end-of-file is reached (or a network connection is closed by the remote host). We consider the latter two cases exceptional, and ask exceptf what we should do. An exceptf function is normally a large switch with one case for each exception. For TP_EREAD, it says:

```
   case TP_EREAD:
      if (errno == EINTR) {
         return TP_TRYAGAIN;
         }
      else {
         ssize_t nread = (*(ssize_t*)obj);
         if (nread == 0) { /* EOF */
            return TP_DEFAULT;
            }
         else {
            return TP_RETURNERROR;
            }
         }
```

This may not seem very revolutionary, after all the code that calls exceptf and branches on its result is just as long as the exception handler itself. We aren't even gaining much code-reuse over the conventional method, which wraps system calls in another function with names like Read(). The real win here lies in the ability of the caller to replace or extend exceptf at runtime. You may have noticed that there is no code above to output an error message, unixread() simply returns -1 on errors. In fact, *the* standard and expected way to output errors is to override exceptf. The wtrace example shown [XXX: at the end somewhere?] uses the following:

```
Tpexcept_f tpexcept;
Tpdisc_t disc;

int exception(int e, void* obj, Tpdist_t* disc)
{
   int rc = tpexcept(e, obj, disc);
   if (rc == TP_RETURNERROR) {
      if (errno != 0) {
         perror(url);
         }
      else {
         switch (e) {
            case TP_HOST:
               fputs(url, stderr);
               fputs(": Unknown host\n", stderr);
               break;
            default:
               fputs(url, stderr);
               fputs(": Error connecting\n", stderr);
               }
         }
      exit(1);
      }
   else {
      return rc;
      }
}
```

Then instead of the usual:

```
tp = tp_new(<uri>, <method>, TpdUnix);
```

wtrace copies TpdUnix, saves and replaces the default exception handler, and then uses the copied discipline:

```
disc = tp_newdisc(TpdUnix);
tpexcept = disc->exceptf;
disc->exceptf = exception;

tp = tp_new(<uri>, <method>, disc);
```

In the same way, wtrace also overrides all of the read and write functions to provide a trace log of HTTP communications.

Part IV: Appendixes

Appendix A: Data Structures

This appendix summarizes, for reference purposes, all descriptor and block lay

outs in Icon.

A.1 Descriptors

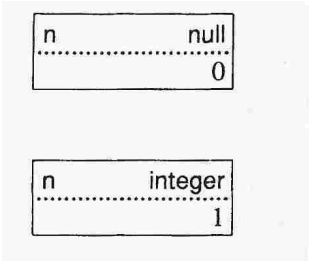
Descriptors consist of two words (normally C ints): a d-word and a v-word. The d-word contains flags in its most significant bits and small integers in its least significant bits. The v-word contains a value or a pointer. The flags are

n	nonqualifier
p	v-word contains a pointer
v	variable
t	trapped variable

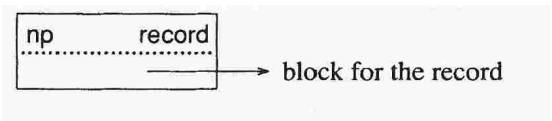
A.1.1 Values

There are three significantly different descriptor layouts for values. A qualifier for a string is distinguished from other descriptors by the lack of an n flag in its d-word, which contains only the length of the string. For example, a qualifier for the string "hello" is

The null value and integers have type codes in their d-words and are self-contained. Examples are:



For all other data types, a descriptor contains a type code in its d-word and a pointer to a block of data in its v-word. A record is typical:

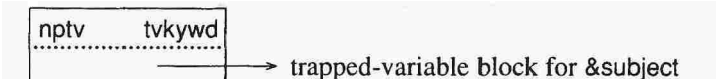


A.1.2 Variables

There are two formats for variable descriptors. The v-word of an ordinary variable points to the descriptor for the corresponding value:

If the variable points to a descriptor in a block, the offset is the number of *words* from the top of the block to the value descriptor. If the variable points to a descriptor that corresponds to an identifier, the offset is zero.

The descriptor for a trapped variable contain,s a type code for the kind of trapped variable in its d-word and a pointer to the block for the trapped variable in its v-word. The trapped variable for &subject is typical:

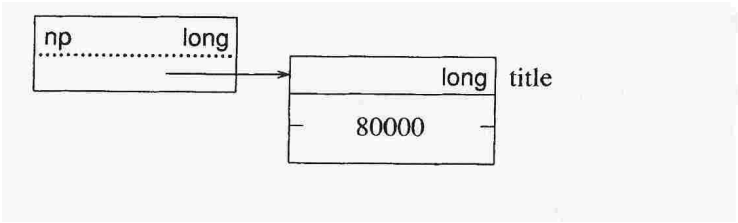


A.2 Blocks

With the exception of the null value, integers, and strings, the data for Icon values is kept in blocks. The first word of every block is a title that contains the type code for the corresponding data type. For blocks that vary in size for a particular type, the next word is the size of the block in bytes. The remaining words depend on the block type, except that all non-descriptor data precedes all descriptor data. With the exception of the long integer block, the diagrams that follow correspond to blocks for computers with 32-bit words.

A.2.1 Long Integers

On computers with 16-bit words, integers that are too large to fit in the d-word of a descriptor are stored in blocks. For example, the block for the integer 80,000 is

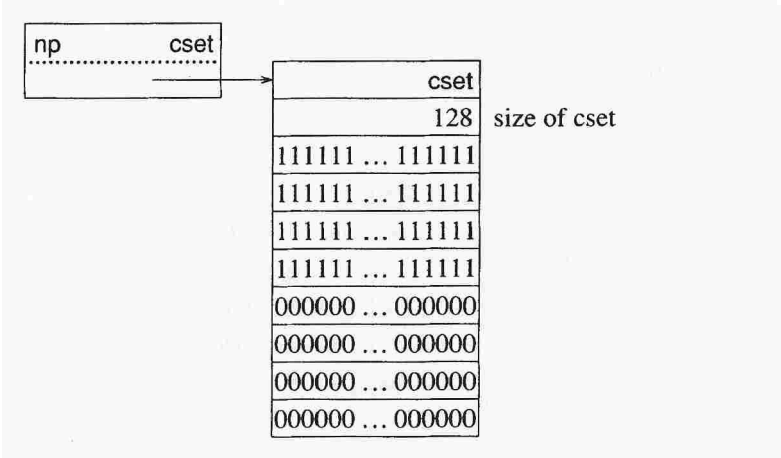


A.2.2 Real Numbers

Real numbers are represented by C doubles. For example, on computers with 32-bit words, the real number 1.0 is represented by

A.2.3 Csets

The block for a cset contains the usual type code, followed by a word that contains the number of characters in the cset. Words totaling 256 bits follow, with a one in a bit position indicating that the corresponding character is in the cset, and a zero indicating that it is not. For example, &ascii is

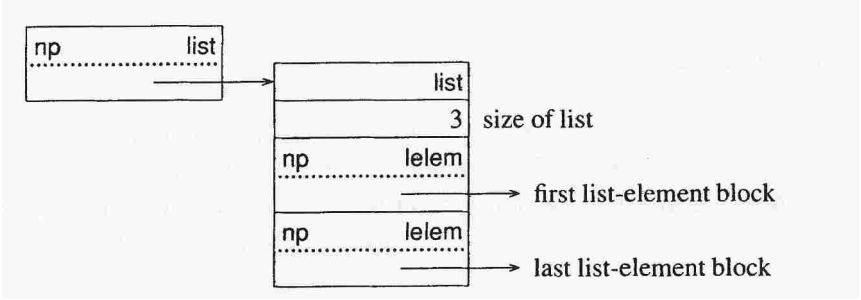


A.2.4 Lists

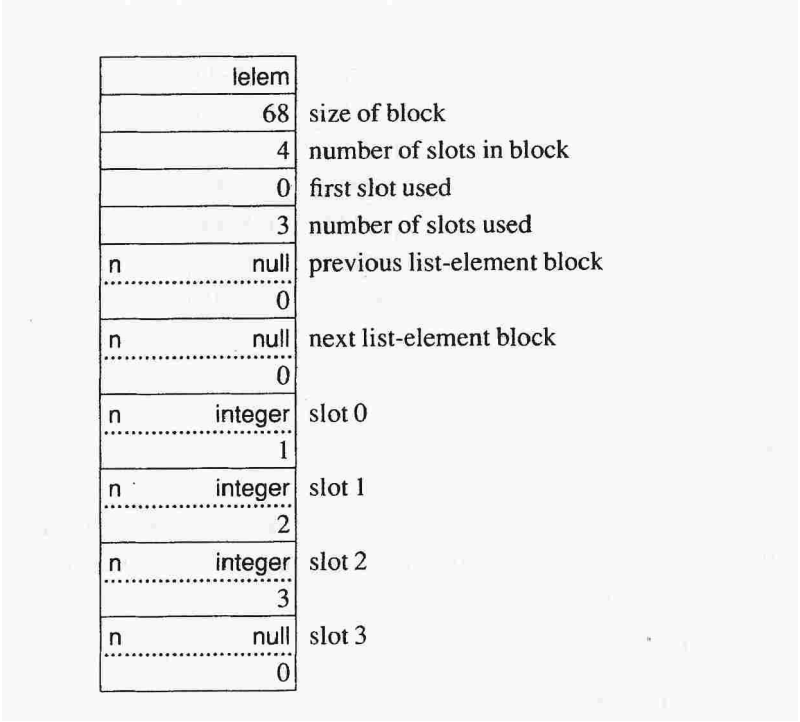
A list consists of a list-header block that points to a doubly-linked list of list-element blocks, in which the list elements are stored in circular queues. See Chapter 6 for details. An example is the list

```
[1,2,3]
```

which is represented as



Here there is only one list-element block:

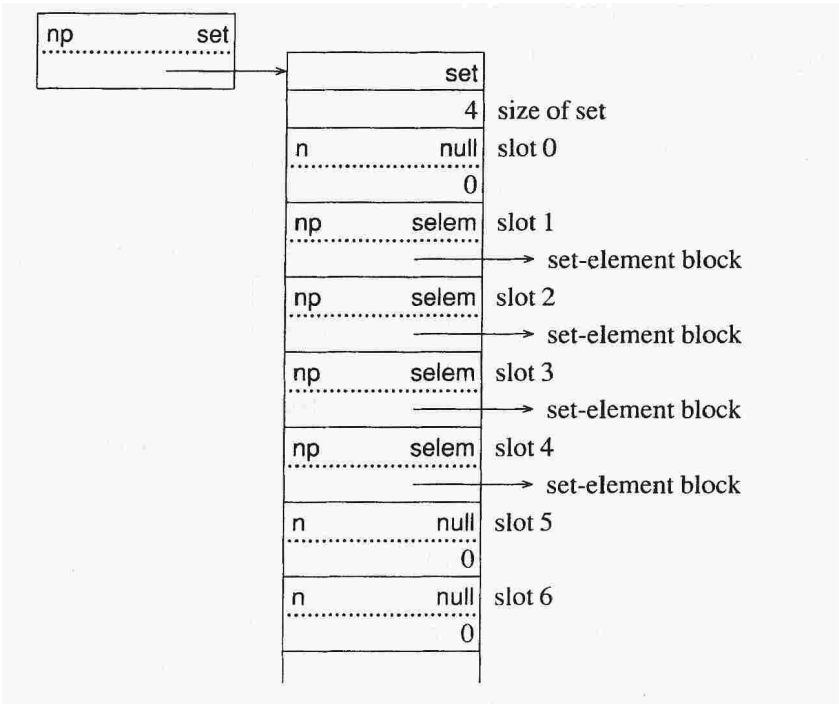


A.2.5 Sets

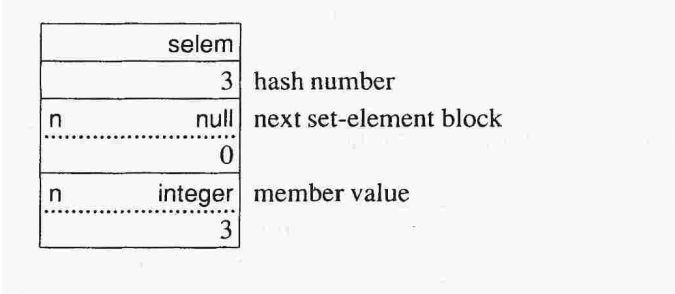
A set consists of a set-header block that contains slots for linked lists of set-element blocks. See Sec. 7.1 for details. An example is given by

```
set([1, 2, 3, 4])
```

which is represented as



The set-element block for the member 3 is

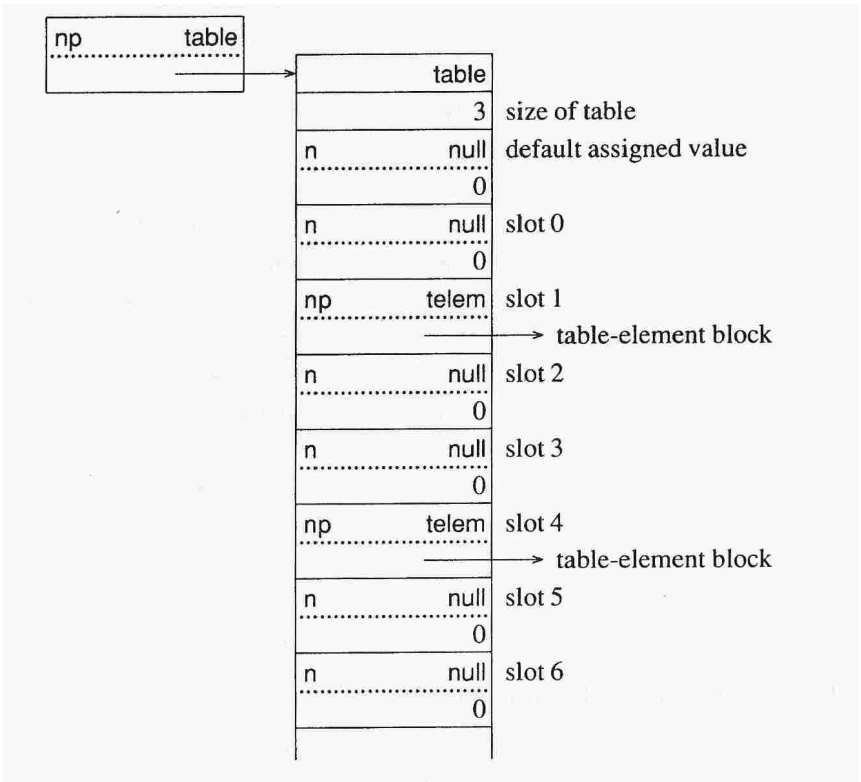


A.2.6 Tables

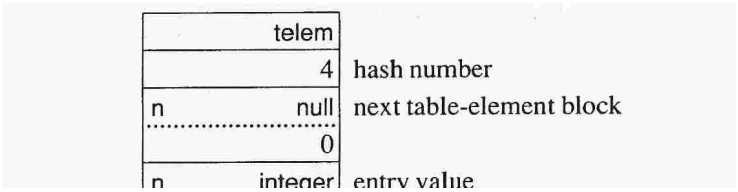
A table is similar to a set, except that a table-header block contains the default assigned value as well as slots for linked lists of table-element blocks. See Sec. 7.2 for details. An example is given by

```
t := table()
every t[1 | 4 | 7] := 1
```

The table t is represented as



The table-element block for the entry value 4 in the previous example is

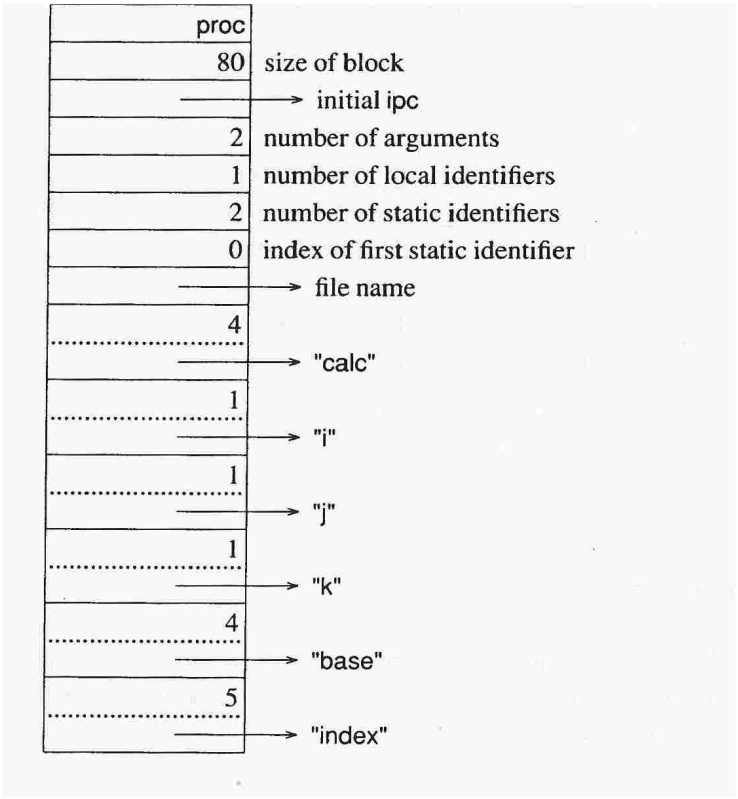


A.2.7 Procedures

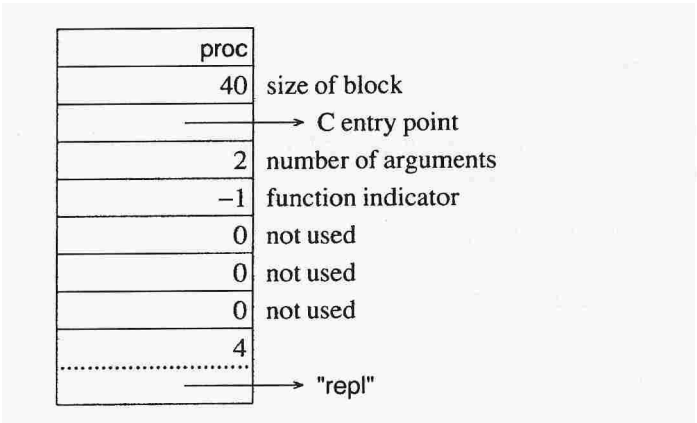
The procedure blocks for procedures and functions are similar. For a procedure declaration such as

```
procedure calc(i,j)
local k
static base, index
end
```

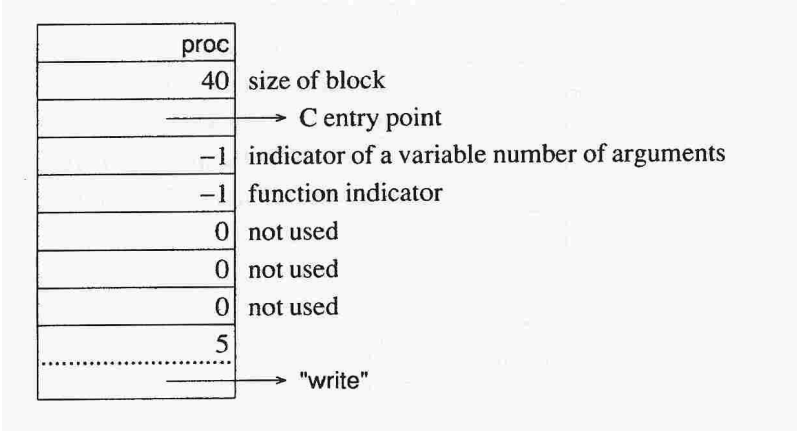
the procedure block is



In a procedure block for a function, there is a value of -1 in place of the number of dynamic locals. For example, the procedure block for repl is

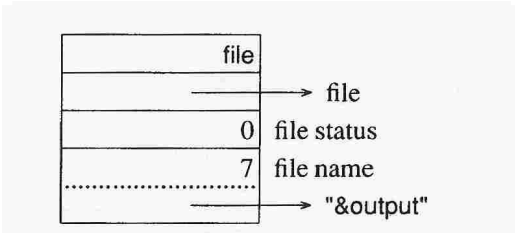


In the case of a function, such as write, which has a variable number of arguments, the number of arguments is given as -1:



A.2.8 Files

The block for a file contains a pointer to the corresponding file, a word containing the file status, and a qualifier for the name of the file. For example, the block for &output is



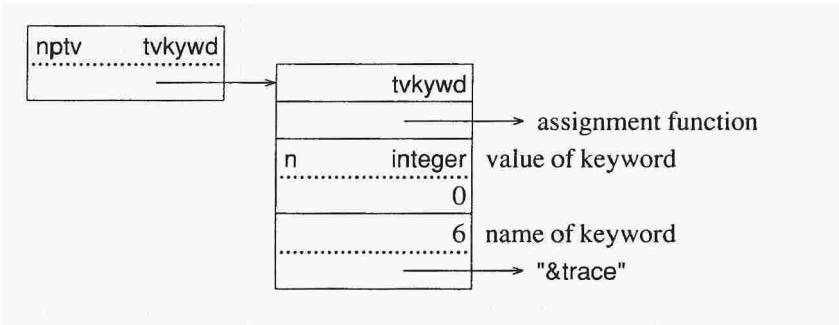
The file status values are

0	closed
1	open for reading
2	open for writing
4	open to create
8	open to append
16	open as a pipe

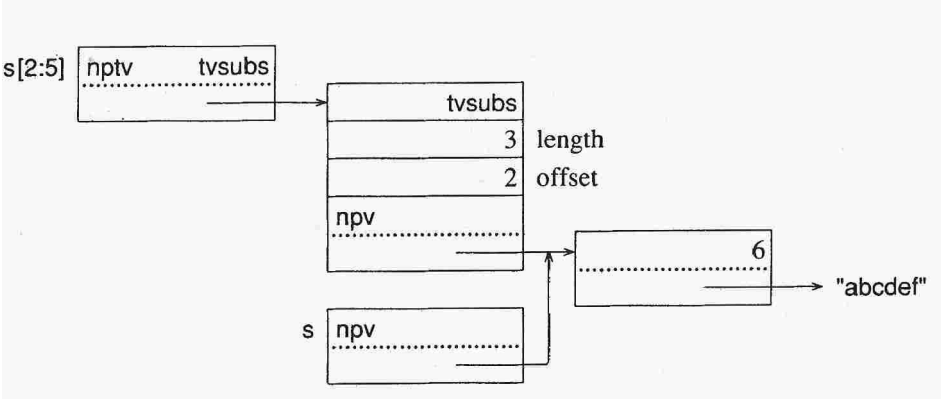
A.2.9 Trapped Variables

There are three kinds of trapped variables: keyword trapped variables, substring trapped variables, and table-element trapped variables. The corresponding blocks are tailored to the kind of trapped variable.

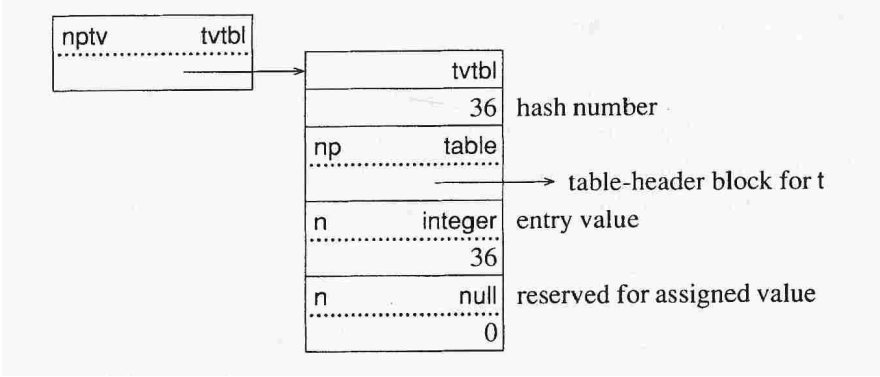
The value of &trace illustrates a typical keyword trapped variable:



A substring trapped variable contains the offset and length of the substring, as well as a variable that points to the qpalifier for the string. For example, if the value of s is "abcdef", the substring trapped-variable block for s [2:5] is

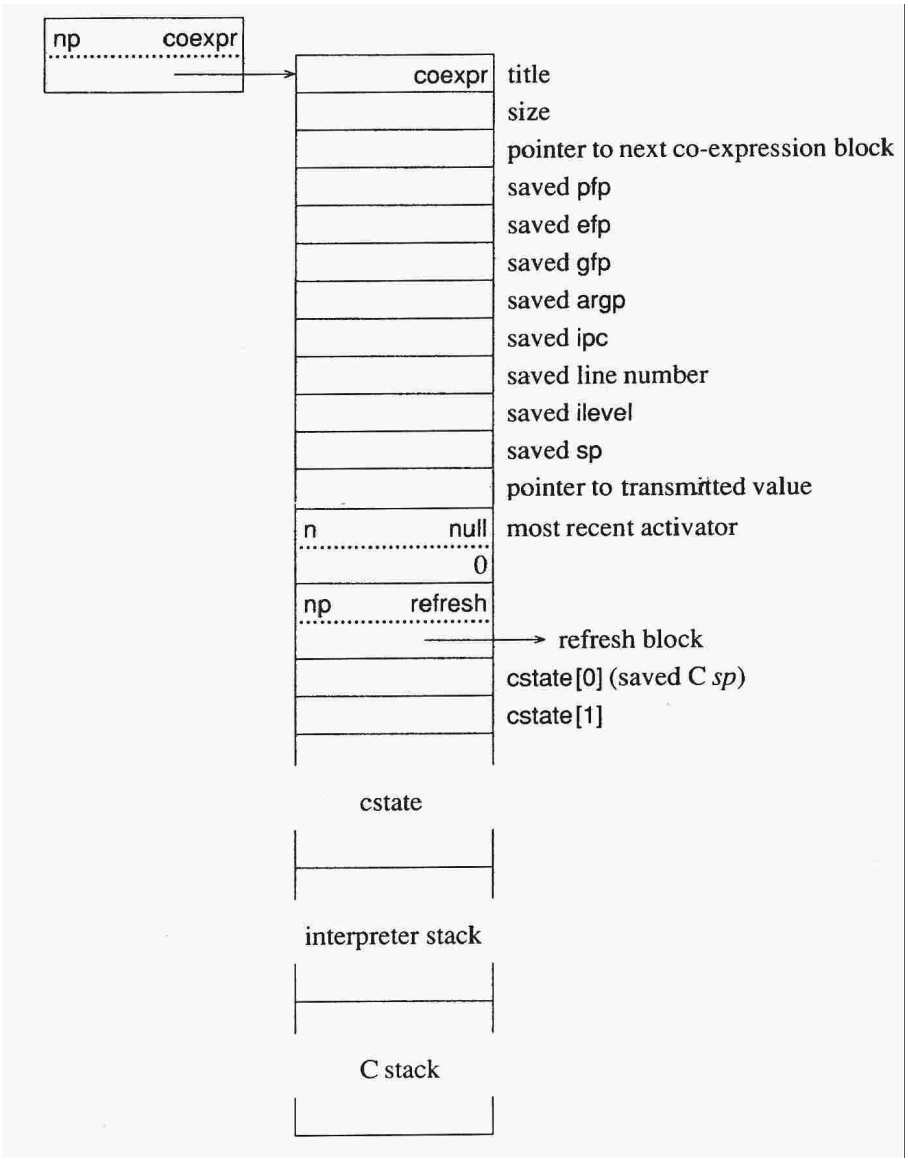


A table-element trapped-variable block contains a word for the hash number of the entry value, a pointer to the table, the entry value, and a descriptor reserved for the assigned value. For example, if t is a table, the table-element trapped-variable block for t[36] is



A.2.10 Co-Expressions

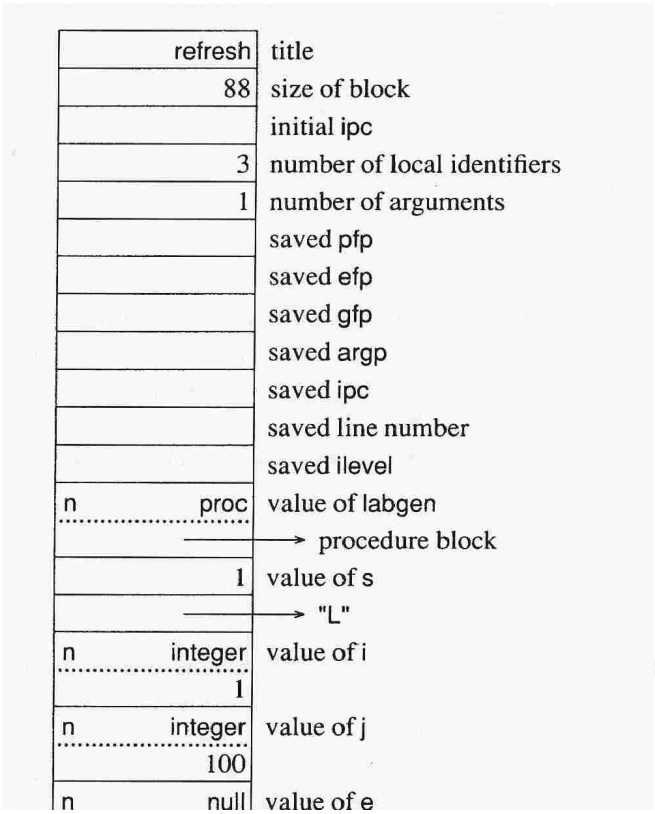
A co-expression block consists of heading information, an array of words for saving the C state, an interpreter stack, and a C stack:



The refresh block contains information derived from the procedure block for the procedure in which the co-expression was created. Consider, for example,

```
procedure labgen(s)
local i, j, e
i := 1
j := 100
e := create (s || (i to j) || ":")
...
end
```

For the call labgen("L"), the refresh block for e is


refresh
title
88
size of block
initial ipc
3
number of local identifiers
1
number of arguments
saved pfp
saved efp
saved gfp
saved argp
saved ipc
saved line number
saved ilevel
n proc
value of labgen
procedure block
1
value of s
"L"
n integer
value of i
1
n integer
value of j
100
n null
value of e
0

Appendix B: Virtual Machine Instructions

This appendix lists all the Icon virtual machine instructions. For instructions that correspond to source-language operators, only the corresponding operations are shown. Unless otherwise specified, references to the stack mean the interpreter stack.

arg n Push a variable descriptor pointing to argument n.

asgn *expr1 := exp12*

bang *lexpr*

bscan Push the current values of &subject and &pos. Convert the descriptor prior to these two descriptors into a string. If the conversion cannot be performed, terminate execution with an error message. Otherwise, assign it to &subject and assign 1 to &pos. Then suspend. If resumed, restore the former values of &subject and &pos and fail.

cat *expr1 || exp12*

ccase Push a copy of the descriptor just below the current expression frame.

chfail n Change the failure ipc in the current expression frame marker to n.

coact Save the current state information in the current co-expression block, restore state information from the co-expression block being activated, perform a context switch, and continue execution.

cofail Save the current state information in the current co-expression block, restore state information from the co-expression block being activated, perform a context switch, and continue execution with co-expression failure signal set.

compl *~expr*

caret Save the current state information in the current co-expression block, restore state information from the co-expression block being activated, perform a context switch, and continue execution with co-expression return signal set.

create Allocate a co-expression block and a refresh block. Copy the current procedure frame marker, argument values, and local identifier values into the refresh block. Place a procedure frame for the current procedure on the stack of the new co-expression block.

cset a Push a descriptor for the cset block at address a onto the stack.

diff *expr1 --expr2*

div *expr1 / expr2*

dup Push a null descriptor onto the stack and then push a copy of the descriptor that was previously on top of the stack.

efail If there is a generator frame in the current expression frame, resume its generator. Otherwise remove the current expression frame. If the ipc in its marker is nonzero, set ipc to it. If the failure ipc is zero, repeat efail.

eqv *expr1 === expr2*

eret Save the descriptor on the top of the stack. Unwind the C stack. Remove the current expression frame from the stack and push the saved descriptor.

escan Dereference the top descriptor on the stack if necessary. Copy it to the place on the stack prior to the saved values of &subject and &pos (see bscan). Exchange the current values of &subject and &pos with the saved values on the stack. Then suspend. If resumed, restore the values of &subject and &pos from the stack and fail.

esusp Create a generator frame containing a copy of the portion of the stack that is needed if the generator is resumed.

field n Replace the record descriptor on the top of the stack by a descriptor for field n of that record.

global n Push a variable descriptor pointing to global identifier n.

goto n Set ipc to n.

init n Change init instruction to goto.

int n Push a descriptor for the integer n.

inter *expr1 ** exp'2*

invoke n *exp1lJ(expr1, exp'2, ..., exprn)*

keywd n Push a descriptor for keyword n.

lconcat *expr1 ||| expr2*

lexeq *expr1 == expr2*

lexge *expr1 »= expr2*

lexgt *expr1 » expr2*

lexle *expr1 «= expr2*

lexlt *expr1 « expr2*

lexne *expr1 -== expr2*

limit Convert the descriptor on the top of the stack to an integer. If the conversion cannot be performed or if the result is negative, terminate execution with an error message. If it is zero, fail. Otherwise, create an expression frame with a zero failure ipc.

line n Set the current line number to n.

llist n *[expr1,expr2,...,exprn]*

local n Push a variable descriptor pointing to local identifier n.

lsusp Decrement the current limitation counter, which is immediately prior to the current expression frame on the stack. If the limit counter is nonzero, create a generator frame containing a copy of the portion of the interpreter stack that is needed if the generator is resumed. If the limitation counter is zero, unwind the C stack and remove the current expression frame from the stack.

mark Create an expression frame whose marker contains the failure ipc corresponding to the label n, the current efp, gfp, and ilevel.

mark0 Create an expression frame with a zero failure ipc.

minus *expr1 -expr2*

mod *expr1 % expr2*

mult *expr1 * expr2*

neg *-expr*

neqv *exprl -=== expr2*

nonnull *\expr*

null */expr*

number +*expr* .

numeq *expr1 = expr2*

numge *exprl >= expr2*

numgt *expr1 > expr2*

numle *exprl <= expr2*

numlt *exprl < expr2*

numne *exprl -= expr2*

pfail If &trace is nonzero, decrement it and produce a trace message. Unwind the C stack and remove the current procedure frame from the stack. Then fail.

plus *exprl + expr2*

pnull Push a null descriptor.

pop Pop the top descriptor.

power *exprl ~ expr2*

pret Dereference the descriptor on the top of the stack, if necessary, and copy it to the place where the descriptor for the procedure is. If &trace is nonzero; decrement it and produce a trace message. Unwind the C stack and remove the current procedure frame from the stack.

psusp Copy the descriptor on the top of the stack to the place where the descriptor for the procedure is, dereferencing it if necessary. Produce a trace message and decrement &trace if it is nonzero. Create a generator frame containing a copy of the portion of the stack that is needed if the procedure call is resumed.

push1 Push a descriptor for the integer 1.

pushn1 Push a descriptor for the integer -1.

quit Exit from the interpreter.

random *?expr*

rasgn *expr1 <- expr2*

real a Push a descriptor for the real number block at address a onto the stack.

refresh *~expr*

rswap *expr1 <-> expr2*

sdup Push a copy of the descriptor on the top of the stackr

sect *expr1 [expr2: expr3*]

size *expr

static n Push a variable descriptor pointing to static identifier n.

str n, a Push a descriptor for the string of length n at address a.

subsc *expr1[expr2]*

swap *expr1 :=: exp12*

tabmat *=exp'*

toby *expr1* to *expr2* by *expr3*

unions *expr1 ++ expr2*

unmark Remove the current expression frame from the stack and unwind the C stack.

value *.expr*

Appendix C: Virtual Machine Code

The virtual machine code that is generated for various kinds of Icon expression is listed below. The form of code given is icode, the output of the Icon linker cast in a readable format. The ucode produced' by the Icon translator, which serves as input to the Icon linker, is slightly different in some cases, since th linker performs some refinements.

C.1 Identifiers

As mentioned in Sec. 8.2.2, the four kinds of identifiers are distinguished by where their values are located. All are referred to by indices, which are zero based.

The values of global identifiers are kept in an array that is loaded from thl icode file and is at a fixed place in memory during program execution. By con vention, the zeroth global identifier contains the procedure descriptor for main The following instruction pushes a variable pointing to the value of main onto the interpreter stack:

main	global	0

Static identifiers are essentially global identifiers that are only known on a per-procedure basis. Like global identifiers, the values of static identifiers are in an array that is at a fixed location. Static identifiers are numbered starting at zero and continuing through the program. For example, if count is static identifier 10 the following instruction pushes a variable descriptor pointing to that static identifier onto the stack:

count	static	10

The space for the values of arguments and local identifiers is allocated on the stack when the procedure in which they occur is called. If x is argument zero and i is local zero for the current procedure, the following instructions push variable descriptors for them onto the stack:

x	arg	0

i	local	0

C.2 Literals

The virtual machine instruction generated for an integer literal pushes the integer onto the stack as an Icon descriptor. The value of the integer is the argument to the instruction:

100	int	100

The instruction generated for a string literal is similar to that for an integer literal, except that the address of the string and its length are given as arguments. The string itself is in a region of data produced by the linker and is loaded as part of the icode file:

"hello"	str	5,a1

The instruction generated for a real or cset literal has an argument that is the address of a data block for the corresponding value. Such blocks are in the data region generated by the linker:

100.2	real a2

'aeiou'	cset a3

C.3 Keywords

The instruction generated for most keywords results in a call to a C function that pushes a descriptor for the keyword onto the stack. The argument is an index that identifies the keyword. For example, &date is keyword 4:

&date	keywd 4

Some keywords correspond directly to virtual machine instructions. Examples are &null and &fail:

&null	pnull

&fail	efail

C.4 Operators

The code generated for a unary operator first pushes a null descriptor, then evaluates the code for the argument, and finally executes a virtual machine instruction that is specific to the operator:

*expr	pnull *code for expr* size

The code generated for a binary operator is the same as the code generated for a unary operator, except that there are two arguments:

expr1 + expr2	pnull *code for expr1* *code for expr2* plus

An augmented assignment operator uses the virtual machine instruction dup to duplicate the result produced by its first argument:

expr1 +:= expr2	pnull *code for expr1* dup *code for expr2* plus asgn

The difference between the code generated for left- and right-associative operators is illustrated by the following examples:

expr1 + expr2 + expr3	pnull pnull *code for expr1* *code for expr2* plus *code for expr3* plus

expr1 := expr2 := expr3	pnull *code for expr1* pnull *code for expr2* *code for expr3* asgn asgn

A subscripting expression is simply a binary operator with a distinguished syntax:

expr1 [expr2]	pnull *code for expr1* *code for expr2* subsc

A sectioning expression is a ternary operator:

expr1 [expr2 : expr3]	pnull *code for expr1* *code for expr2* *code for expr3* sect

Sectioning expressions with relative range specifications are simply abbreviations. The virtual machine instructions for them include the instructions for performing the necessary arithmetic:

expr1 [expr2 +: expr3]	pnull *code for expr1* *code for expr2* dup *code for expr3* plus sect

A to-by expression is another ternary operator with a distinguished syntax:

expr1 to expr2 by expr3	pnull *code for expr1* *code for expr2* *code for expr3* toby

If the by clause is omitted, an instruction that pushes a descriptor for the integer is supplied:

expr1 to expr2	pnull *code for expr1* *code for expr2* push1 toby

The code generated for an explicit list is similar to the code generated for an operator. The instruction that constructs the list has an argument that indicates the number of elements in the list:

[expr1, expr2, expr3]	pnull *code for expr1* *code for expr2* *code for expr3* llist

C.5 Calls

The code generated for a call also is similar to the code generated for an operator except that a null descriptor is not pushed (it is provided by the invoke instruction). The argument of invoke is the number of arguments present in the call, not counting the zeroth argument, whose value is the procedure or integer that is applied to the arguments:

expr0(expr1, expr2)	*code for expr0* *code for expr1* *code for expr2* invoke 2

In a mutual evaluation expression in which the zeroth argument of the "call" is omitted, the default value is -1, for which an instruction is provided:

(expr1, expr2, expr3)	pushn1 *code for expr1* *code for expr2* *code for expr3* invoke 3

C.6 Compound Expressions and Conjunction

The difference between a compound expression and a conjunction expression is illustrated by the following examples. Note that the code generated for conjunction is considerably simpler than that generated for a compound expression, since no separate expression frames are needed:

{expr1; expr2; expr3}	mark L1 *code for expr1* unmark L1: mark L2 *code for expr2* unmark L2: *code for expr3*

expr1 & expr2 & expr3	*code for expr1* pop *code for expr2* pop *code for expr3*

C.7 Selection Expressions

In the code generated for an if-then-else expression, the control expression bounded and has an expression frame of its own:

if expr1 then expr2 else expr3	mark L1 *code for expr1* unmark *code for expr2* goto L2 L1: *code for expr3* L2:

If the else clause is omitted, mark0 is used, so that if the control expression fails, this failure is transmitted to the enclosing expression frame:

if expr1 then expr2	mark0 *code for expr1* unmark *code for expr2*

The code generated for a case expression is relatively complicated. As for similar control structures, the control expression is bounded. The result it produces is placed on the top of the stack by the eret instruction, which saves the result of evaluating *expr1*, removes the current expression frame, and then push the saved result on the top of the stack. The ccase instruction pushes a null descriptor onto the stack and duplicates the descriptor just below the current efp on the top of the stack. This has the effect of providing a null descriptor and the first argument for the equivalence comparison operation performed by eqv. The second argument of eqv is provided by the code for the selector clause. The remainder of the code for a case clause removes the current expression frame marker. in case the comparison succeeds. and evaluates the selected expression:

case expr1 of {
  expr2 : expr3
  expr4 : expr5
  default: expr6
  }
	mark0
	code for expr1
	eret
	mark L2
	ccase
	code for expr2
	eqv
	unmark
	pop
	code for expr3
	goto L1
	L2:
	mark L3
	ccase
	code for expr4
	eqv
	unmark
	pop
	code for expr5
	goto L1
	L3:
	pop
	code for expr6
	L1:

C.8 Negation

The not control structure fails if its argument succeeds but produces the null value if its argument fails:

not expr	mark L1 *code for expr* unmark efail L1: pnull

C.9 Generative Control Structures

If the first argument of an alternation expression produces a result, esusp produces a generator frame for possible resumption and duplicates the surrounding expression frame on the top of the stack. The result of the first argument is then pushed on the top of the stack, so that it looks as if the first argument merely produced a result. The second argument is then bypassed. When the first argument does not produce a result, its expression frame is removed, leaving the second argument to be evaluated:

expr1 | expr2	mark L1 *code for expr1* esusp goto L2 L1: *code for expr2* L2:

Since alternation is treated as a binary operation, a succession of alternations produces the following code:

expr1 | expr2 | expr3	mark L1 *code for expr1* esusp goto L2 L1: mark L3 *code for expr2* esusp goto L2 L2: *code for expr3* L3:

Repeated alternation is complicated by the special treatment of the case in which its argument does not produce a result. If it does not produce a result, the failure is transmitted to the enclosing expression frame, since the failure ipc is 0. However, if it produces a result, the failure ipc is changed by chfail so that subsequent failure causes transfer to the beginning of the repeated alternation expression. The esusp instruction produces the same effect as that for regular alternation. Note that changing the failure ipc only affects the expression frame marker on the stack. When mark is executed again, a new expression frame marker with a failure ipc of 0 is created.

| expr	L1: mark0 *code for expr* chfail L1 esusp

In the limitation control structure, the normal left-to-right order of evaluation is reversed and the limiting expression is evaluated first. The limit instruction checks that the value is an integer and pushes it. It then creates an expression frame marker with a zero failure ipc. Thus, the limit is always one descriptor below the expression marker created by the subsequent mark instruction. The lsusp instruction is similar to the esusp instruction, except that it checks the limit. If the limit is zero, it fails instead of suspending. Otherwise, the limit is decremented:

expr1 \ expr2	*code for expr2* limit *code for expr1* lsusp

C.10 Loops

The code generated for a repeat loop assures that the expression frame is handled uniformly, regardless of the success or failure of the expression:

repeat expr	L1: mark L1 *code for expr* unmark goto L1

A while loop, on the other hand, transmits failure to the enclosing expression frame if its control expression fails. Note that both *expr1* and *expr2* are evaluated in separate expression frames:

while expr1 do expr2	L1: mark0 *code for expr1* unmark mark *code for expr2* goto L1

If the do clause is omitted, the generated code is similar to that for a repeat loop, except for the argument of mark:

while expr	L1: mark0 *code for expr* unmark goto L1

An until loop simply reverses the logic of a while loop:

until expr1 do expr2	L1: mark L2 *code for expr1* unmark efail L2: mark L1 *code for expr2* unmark goto L1

The every-do control structure differs from the while-do control structure in that when its control expression produces a result, its expression frame is not removed. Instead, the result is discarded by pop, and the do clause is evaluated in its own expression frame. The efail instruction forces the resumption of a suspended generator that may have been produced by an esusp instruction in the code for expr1:

every expr1 do expr2	mark0 *code for expr1* pop mark0 *code for expr2* unmark efail

Breaks from loops normally occur in the context of other expressions. In the following example, the break expression removes the expression frame corresponding to the repeat control structure, evaluates its argument expression, and then transfers to a point beyond the end of the loop:

repeat expr1 | break expr2	L1: mark L1 mark L3 *code for expr1* esusp goto L4 L3: unmark *code for expr2* goto L2 L4: unmark goto L1 L2:

Like break, next normally occurs in the context of other expressions. In the following example, next transfers control from a selection expression to the beginning of the loop:

while expr1 do if expr2 then next else expr3	L1: mark0 *code for expr1* unmark mark L1 mark L4 *code for expr2* unmark goto L2 L4: *code for expr3* L2: unmark goto L1

C.11 String Scanning

String scanning is a control structure, rather than an operator, since the values of &subject and &pos must be saved and new values established before the second argument is evaluated. This is accomplished by bscan. The instruction bscan saves the current values

of &subject and &pos and establishes their new values before *expr2* is evaluated. escan restores their values prior to the execution of bscan:

expr1 ? expr2	*code for expr1* bscan *code for expr2* escan

Augmented string scanning is similar to other augmented operations, but it differs in that the string scanning operation does not push a null value on the stack. The instruction sdup therefore is slightly different from dup, which is used in other augmented assignment operations:

expr1 ?:= expr2	pnull *code for expr1* sdup bscan *code for expr2* escan asgn

C.12 Procedure Returns

The code generated for a return expression consists of the pret instruction. However, it allows for failure of the argument of return, which is equivalent to fail:

return expr	mark L1 *code for expr* pret L1: pfail

fail	pfail

The code generated for the suspend expression is analogous to the code generated for alternation, except that the result is returned from the current procedure. The efail instruction causes subsequent results to be produced if the call is resumed:

suspend expr	mark0 *code for expr* psusp efail

C.13 Co-Expression Creation

The first instruction in the code generated for a create expression is a transfer around the code that is executed when the resulting co-expression is activated The create instruction constructs a descriptor that points to the co-expression whose code is at the label given in its argument and pushes this descriptor on the stack. When the co-expression is activated the first time, evaluation starts at the label stored in the co-expression. The result that is on the top of the stack is popped, since transmission of a result to the first activation of a co-expression is meaningless. If *expr* produces a result, coret returns that result to the

activating co-expression. If *expr* fails, cofail signals failure to the activating co-expression:

create expr	goto L3 L1: pop mark L2 *code for expr* coret efail L2: cofail goto L2 L3: create L1

Appendix D: Adding Functions and Data Types

Icon is designed so that new functions and data types can be added with com.

parative ease. Such additions require changes only to the run-time system; the translator and linker are not affected.

This appendix provides some guidelines for modifying the Icon run-time system and lists useful macro definitions and support routines. It is designed to be read in conjunction with the source code for the implementation. The material included here only touches on the possibilities. There is no substitute for actually implementing new features and spending time studying the more intricate parts of the Icon system.

D.1 File Organization

The Icon system is organized in a hierarchy. Under UNIX, the Icon hierarchy is rooted at v6 and is usually located at /usr/icon/v6. For other operating systems, Icon may be named differently. The v6 directory has several subdirectories that contain source code, test programs, documents, and so forth. The source code is in v6/src. There are five subdirectories in src:

h	common header files
icont	command processor
iconx	run-time system
link	linker
tran	translator

The subdirectory h holds header files that are included by files in the other subdirectories. The file h/rt.h is particularly important, since it contains most *0f* the definitions and declarations used in the run-time system.

The rest of the code related to the run-time system is in the subdirectory iconx. The first letters of files in this subdirectory indicate the nature of their contents. Files that begin with the letter f contain code for functions, while files that begin with o contain code for operators. Code related directly to the interpretive process is in files that begin with the letter i. "Library" routines for operations such as list construction that correspond to virtual machine instructions are in files that begin with the letter l. Finally, files that begin with the letter r hold run-time support routines.

Within each category, routines are grouped by functionality. For example, string construction functions such as map are in fstr.c, while storage allocation and garbage collection routines are in rmemmgt.c.

D.2 Adding Functions

There are several conventions and rules of protocol that must be followed in writing a new function. The situations that arise most frequently are covered in the following sections. The existing functions in f files in iconx provide many examples to supplement the information given here.

D.2.1 Function Declarations

A function begins with a call of the macro FncDcl(name, n), where name is the name of the function as it is called in a source-language program, and n is the number of arguments for the function. For example,

```
FncDcl(map,3)
```

appears at the beginning of the function map. This macro declares the procedure block for the function and provides the beginning of the declaration of a C function for the code that follows. The value of n appears in the procedure block and is used to assure that the number of arguments on the interpreter stack when the function is called is the same as the number of arguments that the function expects. See Sec. 10.3.

An X is prepended to the name given to avoid a collision with the names of other C routines in the run-time system. Thus, the C function that implements map is named Xmap. Although the Icon function map has three arguments, the corresponding C function has only one: cargp, which is a pointer to an array of descriptors on the interpreter stack. For example, FncDcl(map, 3) generates

```
Xmap(cargp)
register struct descrip *cargp;
```

Other macros are provided for referencing the descriptors: Arg0 is the descriptor into which the result of a function is placed before it returns, Arg1 is the first descriptor argument in the call of the function, Arg2 is the second descriptor argument, and so on. These macros conceptually refer to the arguments in a source-language call of the function. It is never necessary (or desirable) to refer to cargp directly.

Note that the descriptor at Arg0 initially points to the procedure block for the function (see Sec. 10.1). It is fair to assume that Arg1, Arg2, ..., Argi, where i arguments *are* specified in the declaration contain valid descriptors Nothing can be assumed about the nature of these descriptors, other than that they represent valid source-language values. Similarly, a function must place a valid descriptor in Arg0 before returning, overwriting the procedure descriptor.

The macros described previously allow functions to be written without worrying about the details of the interpreter stack. It is not important to know how these macros are actually defined; it is best to think of them in terms of the higher-level concepts they embody.

D.2.2 Returning from a Function

A function returns control to the interpreter by use of one of three macros, Return, Suspend, or Fail, depending on whether the function returns, suspends, or fails, respectively. Return and Fail return codes that the interpreter uses to differentiate between the two situations. Suspend returns control to the interpreter by calling it, as described in Sec. 9.3.

The use of Return is illustrated by the following trivial function that simply returns its argument:

```
FncDcl(idem, 1)
{
Arg0 = Arg1;
Return;
}
```

For example,

```
write(idem("hello"))
```

writes hello.

The use of Suspend and Fail is illustrated by the following function, which generates its first and second arguments in succession:

```
FncDcl(gen2,2)
{
Arg0 = Arg1;
Suspend;
Arg0 = Arg2;
Suspend;
Fail;
}
```

For example,

```
every write(gen2("hello", "there"))
```

writes

```
hello
there
```

As illustrated previously, Fail is used when there is not another result to produce. It is safe to assume that Arg0, Arg1, ...are intact when the function is resumed to produce another result.

Most functions have a fixed number of arguments. Only write, writes, and stop in the standard Icon repertoire can be called with an arbitrary number of arguments. For a function that can be called with an arbitrary number of arguments, an alternative declaration macro, FncDcIV(name), is used. When this macro is used, the function is called with two arguments: the number of arguments in the call and a pointer to the corresponding array of descriptors. For example, FncDcIV(write) generates

```
Xwrite(nargs, cargp)
int nargs;
register struct descrip cargp;
```

Within such a function, Arg0 refers to the return value as usual, but the arguments are referenced using the macro Arg(n). For example, a function that takes an arbitrary number of arguments and suspends with them as values in succession is

```
FncDcIV(gen)
{
register int n;
for (n = 1; n <= nargs; n++) {
Arg0 = Arg(n);
Suspend;
      }
Fail;
}
```

For example,

```
every write(gen("hello","there","!"))
```

writes

```
hello
there
!
```

Note the use of Fail at the end of the function; the omission of Fail would be an error, since returning by flowing off the end of the function would not provide the return code that the interpreter expects.

D.2.3 Type Checking and Conversion

Some functions need to perform different operations, depending on the types of their arguments. An example is type(x):

```
      FncDcl(type, 1)
      {
      if (Qual(Arg1)) (
StrLen(Arg0) = 6;
StrLoc(Arg0) = "string";
}
else {
      switch (Type(Arg1)) (
case T_Null:
StrLen(Arg0) = 4;
StrLoc(Arg0) = "null";
      break;
case T_Integer:
case T_Long:
StrLen(Arg0) = 7;
StrLoc(Arg0) = "integer";
break;
case T_Real:
StrLen(Arg0) = 4;
StrLoc(Arg0) = "real";
break;
      }
      }
      Return;
      }
```

As indicated by this function, the d-word serves to differentiate between types, except for strings, which require a separate test.

For most functions, arguments must be of a specific type. As described in Sec. 12.1, type conversion routines are used for this purpose. For example, the function tab(i) requires that i be an integer. It begins as follows:

```
FncDcl(tab. 1 )
{
register word i, j;
word t, oldpos;
long 11;
      /*
      * Arg1 must be an integer.
      */
if (cvint(&Arg1, &11) == CvtFail)
runerr(101, &Arg1);
```

Note that cvint is called with the addresses of Arg1 and 11. If the conversion is successful, the resulting integer is assigned to 11. As indicated by this example, it is the responsibility of a function to terminate execution by calling runerr if a required conversion cannot be made.

The routine cvstr, which converts values to strings, requires a buffer, which is supplied by the routine that calls it. See Sec. 4.4.4. This buffer must be large enough to hold the longest string that can be produced by the conversion of any value. This size is given by the defined constant MaxCvtLen. For example, the function to reverse a string begins as follows:

```
FncDcl(reverse.1 )
{
register char c. *floc. *lIoc;
register word slen;
char sbuf[MaxCvtLen];
extern char *alcstrO;
/*
* Make sure that Arg1 is a string.
*/
if (cvstr(&Arg1. sbuf) == CvtFail)
runerr(103. &Arg1);
```

The buffer is used only if a nonstring value is converted to a string. In this case,

Arg1 is changed to a qualifier whose v-word points to the converted string in sbuf. This string does not necessarily begin at the beginning of sbuf. In any event, after a successful call to cvstr, the argument is an appropriate qualifier, regardless of whether a conversion actually was performed.

D.2.4 Constructing New Descriptors

Some functions need to construct new descriptors to return in Arg0. Sometimes it is convenient to construct a descriptor by assignment to its d- and v-words. Various macros are provided to simplify these assignments. As given in the function type previously, StrLen and StrLoc can be used to construct a qualifier. For example, to return a qualifier for the string "integer", the following code suffices:

```
StrLen(Arg0) = 7;
StrLoc(Arg0) = "integer";
Return;
```

Here, the returned qualifier points to a statically allocated C string.

There also are macros and support routines for constructing certain kinds of descriptors. For example, the macro

```
Mkint(i, dp);
```

constructs an integer descriptor containing the integer i in the descriptor pointed to by dp. The definition of Mkint depends on the word size of the computer. On 32-bit computers, Mkint simply produces assignments to the d-word and v-word of descriptor pointed to by dp. On computers with 16-bit words, which have both T_Integer and T_Long forms of integers, Mkint produces a call to a support routine.

D.2.5 Default Values

Many functions specify default values for null-valued arguments. There are support routines for providing default values. For example,

```
defstr(Arg3, sbuf, &q);
```

changes Arg3 to the string given by the qualifier q in case Arg3 is null-valued. If Arg3 is not null-valued, however, its value is converted to a string, if possible, by defstr. If this is not possible, defstr terminates execution with an error message.

D.2.6 Storage Allocation

Functions that construct new data objects often need to allocate storage. Allocation is done in the allocated string region or the allocated block region, depending on the nature of the object. Support routines are provided to perform the actual allocation.

As mentioned in Sec. 11.4, predictive need requests *must* be made before storage is actually allocated. The functions strreq(i) and blkreq(i) request i bytes of storage in the allocated string and block regions, respectively.

Such a request generally should be made as soon as an upper bound on the amount of storage needed is known. It is not necessary to know the exact amount, but the amount requested must be at least as large as the amount that actually will be allocated. For example, the function reads(f. i) requests i bytes 0 string storage, although the string actually read may be shorter.

String Allocation. The function alcstr(s, i) copies i bytes starting at s into the allocated string region and returns a pointer to the beginning of the copy. Fo example, a function double(s) that produces the concatenation of s with itself is written as follows:

```
FncDcl(double. 1)
{
register int glen;
char sbuf[MaxCvtLen];
extern char *alcstrO;
if (cvstr(&Arg1. sbuf) == NULL)
runerr(103, &Arg1);
glen = StrLen(Arg1);
strreq(2 * glen);
StrLen(Arg0) = 2 * glen;
StrLoc(Arg0) = alcstr(StrLoc(Arg1). glen);
alcstr(StrLoc(Arg1), glen);
Return;
}
```

If the first argument of alcstr is NULL, instead of being a pointer to a string, the space is allocated and a pointer to the beginning of it is returned, but nothing is copied into the space. This allows a function to construct a string directly in the allocated string region.

If a string to be returned is in a buffer as a result of conversion from another type, care must be taken to copy this string into the allocated string region---otherwise the string in the buffer will be overwritten on subsequent calls. Copying such strings is illustrated by the function string(x) given in Sec. 12.1.

Block Allocation. The routine alcblk(i) allocates i bytes in the allocated block region and returns a pointer to the beginning of the block. The argument of alcblk must correspond to a whole number of words. There are run-time support routines for allocating various kinds of blocks. These routines, in turn, call alcblk. Such support routines generally fill in part of the block as well. For example, alccset(i) allocates a block for a cset, fills in the title and size words, and zeroes the bits for the cset:

```
struct b_cset *alccset(size)
int size;
```

```
{
register struct b_cset *blk;
register i;
extern union block *alcblk();
blk = (struct b_cset *)alcblk((word)sizeof(struct b_cset) ,
T_Cset); blk->size = size;
/*
* Zero the bit array.
*/
for (i = 0; i < CsetSize; i++)
blk->bits[i] = 0; return blk;
}
```

See Sec. D.5.5 for a complete list of block-allocation functions.

D.2.7 Storage Management Considerations

In addition to assuring that predictive need requests are made before storage is allocated, it is essential to assure that all descriptors contain valid data at any time a garbage collection may occur, that all descriptors are accessible to the garbage collector, and that all pointers to allocated data are in the v-words of descriptors.

Normally, all the descriptors that a function uses are on the interpreter stack and are referenced as Arg0, Arg1, Such descriptors are processed by the garbage collector. Occasionally, additional descriptors are needed for intermediate computations. If such descriptors contain pointers in their v-words, it is *not* correct to declare local descriptors, as in

```
FncDcl(mesh,2)
{
struct descrip d1, d2;
```

The problem with this approach is that d1 and d2 are on the C stack and the garbage collector has no way of knowing about them.

However, since all descriptors on the interpreter stack are accessible to the garbage collector, intermediate computations can be performed on descriptors on the interpreter stack. Extra descriptors for this purpose can be provided by increasing the number of arguments specified for the function. Thus,

```
FncDcl(mesh,4)
```

makes Arg3 and Arg4 available for intermediate computations. The initial values of Arg3 and Arg4 will be null because of argument adjustment performed by invoke unless mesh is called with extra arguments.

Garbage collection can occur only during a predictive need request. However, a predictive need request can occur between the time a function suspends and the time it is resumed to produce another result. Consequently, if a pointer is kept in a C variable in a loop that is producing results by suspending, the pointer may be invalid when the function is resumed. Instead, the pointer should be kept in the v-word of a descriptor that is accessible to the garbage collector.

D.2.8 Error Termination

An Icon program may terminate abnormally for two reasons: as the result of a source-language programming error (such as an invalid type in a function call), or as a result of

an error detected in the Icon system itself (such as a descriptor that should have been dereferenced but was not).

In case a source-language error is detected, execution is terminated by a call of the form

```
runerr(i, &d);
```

where i is an error message number and d is the descriptor for the offending value. If there is no specific offending value, the second argument is 0.

The array of error message numbers and corresponding messages is contained in iconx/imain.c. If there is no appropriate existing error message, a new one can be added, following the guidelines given in Appendix D of Griswold and Griswold 1983.

In theory, there should be no errors in the Icon system itself, but no large, complex software system is totally free of errors. Some situations are recognizable as being potential sources of problems in case data does not have the expected values. In such situations, especially during program development, it is advisable to insert calls of the function syserr, which terminates execution, indicating that an error was detected in the Icon system, and prints its argument as an indication of the nature of the error. It is traditional to use calls of the form

```
syserr("mesh: can't happen");
```

so that when, in fact, the "impossible" does happen, there is a reminder of human frailty. More informative messages are desirable, of course.

D.2.9 Header Files

If a new function is added to an existing f file in iconx, the necessary header files normally will be included automatically. If a new function is placed in a new file, that file should begin with

```
#include "../h/rt.h"
```

This header file includes three other header files:

../h/config.h	general configuration information
../h/cpuconf.h	definitions that depend on the computer word size
../h/memsize.h	definitions that depend on the computer address space

All of these files contain appropriate information for the local installation, and no changes in them should be needed.

In rare cases, it may be necessary to include other header files. For example, a function that deals directly with garbage collection might need to include iconx/gc.h.

D.2.10 Installing a New Function

Both the linker and the run-time system must know the names of all functions. This infonnation is provided in the header file h/fdefs.h.

In order to add a function, a line of the form

```
FncDef(name)
```

must be inserted in h/fdefs.h in proper alphabetical order.

Once this insertion is made, the Icon system must be recompiled to take into account the code for the new function. The steps involved in recompilation vary from system to

system. Information concerning recompilation is available in system-specific installation documents.

D.3 Adding Data Types

Adding a new data type is comparatively simple, although there are several places where changes need to be made. Failure to make all the required changes can produce mysterious bugs.

D.3.1 Type Codes

At present, type codes range from 0 to 18. Every type must have a distinct type code and corresponding definitions. These additions are made in h/rt.h. First, a T -definition is needed. For example, if a Boolean type is added, a definition such as

```
#define T  _Boolean   19
```

is needed. The value of MaxType, which immediately follows the type code definitions, must be increased to 19 accordingly. Failure to set MaxType to the maximum type code may result in program malfunction during garbage collection. See Sec. 11.3.2.

Next a 0- definition is needed for the d-word of the new type. For a Boolean type, this definition might be

```
#define D_Boolean     (T_Boolean I F_Nqual)
```

All nonstring types have the F _Nqual flag and their T- type code. Types whose v-words contain pointers also have the F _Ptr flag.

D.3.2 Structures

A value of a Boolean type such as the one suggested previously can be stored in the d-word of its descriptor. However, most types contain pointers to blocks in their v-words. In this case, a declaration of asttucture corresponding to the block must be added to h/rt.h. For example, a new rational number data type, with the type code T_Rational, might be represented by a block containing two descriptors, one for the numerator and one for the denominator. An appropriate structure declaration for such a block is

```
struct b_rational {
int title;
struct descrip numerator;
struct descrip denominator;
};
```

Since rational blocks are fixed in size, no size field is needed. However, a vector type with code T_Vector in which different vectors have different lengths needs a size field. The declaration for such a block might be

```
struct b_vector {
int title;
int blksize;
struct descrip velems[1];
};
```

As mentioned in Sec. 4.4.2, the size of one for the array of descriptors is needed to avoid problems with C compilers. In practice, this structure conceptually overlays the allocated block region, and the number of elements varies from block to block.

Any new structure declaration for a block must be added to the declaration union block in h/rt.h.

D.3.3 Information Needed for Storage Management

All pointers to allocated data must be contained in the v-words of descriptors, since this is the only way the garbage collector can locate them. Furthermore, all non-descriptor data must precede any descriptors in a block. The amount of non-descriptor data, and hence the location of the first descriptor in a block, must be the same for all blocks of a given type.

As described in Sec. 11.3.2, the garbage collector uses the array bsizes to determine the size of a block and the array firstd to determine the offset of the first descriptor in the block. These arrays are in iconx/rmemmgt.c. When a new data type is added, appropriate entries must be made in these arrays. Failure to do so may result in serious bugs that occur only in programs that perform garbage collection, and the symptoms may be mysterious.

There is an entry in bsizes for each type code. If the type has no block, the entry is -1. If the type has a block of constant size, the entry is the size of the block. Otherwise, the entry is 0, indicating that the size is in the second word of the block. Thus, the entry for T_Boolean would be -1, the entry for T_Rational would be sizeof(struct b_rational), and the size for T_Vector would be O.

There is a corresponding entry in firstd for each type code that gives the offset of the first descriptor in its corresponding block. If there is no block, the entry is -1. If the block contains no descriptors, the entry is O. For example, the entry for T_Boolean would be -1, the entry for T_Rational would be WordSize, and the entry for T_Vector would be 2*WordSize, where WordSize is a defined constant that is the number of bytes in a word.

A third array, blknames, provides string names for all block types. These names are only used for debugging, and an entry should be made in blknames for each new data type.

D.3.4 Changes to Existing Code

In addition to any functions that may be needed for operating on values of a new data type, there are several functions and operators that apply to all data types and which may, therefore, need to be changed for any new data type.

These are

*x	size of x (in iconx/omisc.c)
copy(x)	copy of x (in iconx/fmisc.c)
image(x)	string image of x (in iconx/fmisc.c)
type(x)	string name of type of x (in iconx/fmisc.c)

There is not a concept of size for all data types. For example, a Boolean value presumably does not have a size, but the size of a vector presumably is the number of elements it contains. The size of a rational number is problematical. Modifications to *x are easy; see Sec. 4.4.4.

There must be some provision for copying any value. For structures, such as vectors, physical copies should be made so that they are treated consonantly with other Icon structures. For other data types, the "copy" consists of simply returning the value and not

making a physically distinct copy. This should be done for data types, such as Boolean, for which there are only descriptors and no associated blocks. Whether or not a copy of a block for a rational value should be made is a more difficult decision and depends on how such values are treated conceptually, at the source-language level. It is, of course, easiest not to make a physical copy.

Some image must be provided for every value. This image should contain enough information to distinguish values of different types and, where possible, to provide some useful additional information about the specific value. The amount of detail that it is practical to provide in the image of a value is limited by the fact that the image is a string that must be placed in the allocated string region.

The type must be provided for all values and should consist of a simple string name. For example, if x is a Boolean value, type(x) should produce "boolean". The coding for type is trivial; see Sec. D.2.3.

There also are several run-time support routines that must be modified for any new type:

outimage	image for tracing (in iconx/rmisc.c)
order	order for sorting (in iconx/rcomp.c)
anycmp	comparison for sorting (in iconx/rcomp.c)
equiv	equivalence comparison (in iconx/rcomp.c)

The image produced for tracing purposes is similar to that produced by image and must be provided for all data types. However, outimage produces output and is not restricted to constructing a string in allocated storage. It therefore can be more elaborate and informative.

There must be some concept of sorting order for every Icon value. There are two aspects to sorting: the relative order of different data types and the ordering among values of the same type. The routine order produces an integer that corresponds to the order of the type. If the order of a type is important with respect to other types, this matter must be given some consideration. For example, a rational number probably belongs among the numeric types, which, in Icon, sort before structure types. On the other hand, it probably is not important whether vectors come before or after lists.

The routine anycmp compares two values; if they have the same order, as defined previously, anycmp determines which is the "smaller." For example, Boolean "false" might (or might not) come before "true," but some ordering between the two should be provided. On the other hand, order among vectors probably is not important (or well-defined), and they can be lumped with the other structures in anycmp, for which ordering is arbitrary. Sometimes ordering can be quite complicated; a correct ordering of rational numbers is nontrivial.

The routine equiv is used in situations, such as table subscripting and case expressions, to determine whether two values are equivalent in the Icon sense. Generally speaking, two structure values are considered to be equivalent if and only if they are identical. This comparison is included in equiv in a general way. For example, equiv need not be modified for vectors. Similarly, for data types that have no corresponding blocks, descriptor comparison suffices; equiv need not be modified for Boolean values either. However, determining the equivalence of numeric values, such as rational numbers, requires some thought.

D.4 DEFINED CONSTANTS AND MACROS

Defined constants and macros are used heavily in Icon to parameterize its code for different operating systems and computer architectures and to provide simple, high-level constructions for commonly occurring code sequences that otherwise would be complex and obscure.

These defined constants and macros should be used consistently when making additions to Icon instead of using *ad hoc* constructions. This improves portability, readability, and consistency.

Learning the meanings and appropriate use of the existing defined constants and macro definitions requires investment of time and energy. Once learned, however, coding is faster, simpler, and less prone to error.

D.4.1 Defined Constants

The following defined constants are used frequently in the run-time system. This list is by no means exhaustive; for specialized constants, see existing functions.

CsetSize	number of words needed for 256 bits
LogHuge	one plus the maximum base-lO exponent of a C double
LoglntSize	base-210garithm of number of bits in a C int
MaxCvtLen	length of the longest possible string obtained by conversion
MaxLong	largest C long
MaxShort	largest C short
MaxStrLen	longest possible string
MinListSlots	minimum number of slots in a list-element block
MinLong	smallest C long
MinShort	smallest C short
WordSize	number of bytes in a word

D.4.2 Macros

The following macros are used frequently in the run-time system. See h /rt.h and iconx/gc.h for the definitions, and see existing routines for examples of usages.

Arg(n)	nth argument to function
ArgType(n)	d-word of nth argument to function
ArgVal(n)	integer value of v-word of nth argument to function
BlkLoc(d)	pointer to block from v-word of d
BlkSize(cp)	size of block pointed to by cp
BlkType(cp)	type code of block pointed to by cp
ChkNull(d)	true if d is a null-valued descriptor
CsetOff(b)	offset in a word of cset bit b
CsetPtr(b, c)	address of word c containing cset bit b
DeRef(d)	dereference d
EqlDesc(d1, d2)	true if d1 and d2 are identical descriptors
GetReal(dp, r)	get real number into r from descriptor pointed to by dp
IntVal(d)	integer value of v-word of d
Max(i, j)	maximum of i and j
Min(i. j)	minimum of i and j
Mkint(i, dp)	make integer from i in descriptor pointed to by dp

Offset(d)	offset from d-word of variable descriptor d
Pointer(d)	true if v-word of d is a pointer
Qual(d)	true if d is a qualifier
Setb(b, c)	set bit b in cset c
SlotNum(i, j)	Slot for hash number i given j total slots
StrLen(q)	length of string referenced by q
StrLoc(q)	location of string referenced by q
Testb(b, c)	true if bit b in cset c is one
Tvar(d)	true if d is a trapped variable
TvarLoc(d)	pointer to trapped variable from v-word of d
Type(d)	type code in d-word of d
Var(d)	true if d is a variable descriptor
VarLoc(d)	pointer to value descriptor from v-word of d
Vsizeof(x)	size of structure x less variable array at end
Vwsizeof(x)	size of structure x in words less variable array at end
Wsizeof(x)	size of structure x in words

D.5 Support Routines

There are many support routines for performing tasks that occur frequently in the Icon run-time system. Most of these routines are in files in iconx that begin with the letter r. The uses of many of these support routines have been illustrated earlier; what follows is a catalog for reference.

D.5.1 Comparison

The following routines in iconx/rcomp.c perform comparisons:

anycmp(dp1, dp2) Compare the descriptors pointed to by dp1 and dp2 as Icon values in sorting order, returning a value greater than 0, 0, or less than 0 depending on whether the descriptor pointed to by dp1 is respectively greater than, equal to, or less than the descriptor pointed to by dp2.

equiv(dp1, dp2) Test for equivalence of descriptors pointed to by dp1 and dp2, returning 1 if equivalent and 0 other

Appendix E: Projects

Appendix F: Solutions to Selected Exercises

Appendix G: The RTL Run-Time Language

This appendix contains a description of the language used to implement the run-time operations of the Icon compiler system. Chapter 5 provides a description of the design goals of the implementation language and an introduction to it. Some of the design decisions for the language were motivated by optimizations planned for the future, such as constant folding of csets. The use of these features is presented as if the optimizations were implemented; this insures that the optimizations will be supported by the run-time system when they are implemented. This appendix is adapted from the reference manual for the language [.ipd79.].

The translator for the implementation language is the program rtt. An rtt input file may contain operation definitions written in the implementation language, along with C definitions and declarations. Rtt has a built-in C preprocessor based on the ANSI C Standard, but with extensions to support multi-line macros with embedded preprocessor directives [.ipd65.]. Rtt prepends a standard include file, grttin.h, on the front of every implementation language file it translates.

The first part of this appendix describes the operation definitions. C language documentation should be consulted for ordinary C grammar. The extensions to ordinary C grammar are described in the latter part of the appendix.

The grammar for the implementation language is presented in extended BNF notation. Terminal symbols are set in Helvetica. Non-terminals and meta-symbols are set in *Times-Italic*. In addition to the usual meta-symbols, *::=* for ``is defined as" and | for ``alternatives", brackets around a sequence of symbols indicates that the sequence is optional, braces around a sequence of symbols followed by an asterisk indicates that the sequence may be repeated zero or more times, and braces followed by a plus indicates that the enclosed sequence may be repeated one or more times.

G.1 Operation Documentation

An operation definition can be preceded by an optional description in the form of a C string literal.

```
documented-definition ::= [ C-string-literal ] operation-definition
```

The use of a C string allows an implementation file to be run through the C preprocessor without altering the description. The preprocessor concatenates adjacent string literals, allowing a multi-line description to be written using multiple strings. Alternatively, a multi-line description can be written using `' for line continuation. This description is stored in the operation data base where it can be extracted by documentation generation programs. These documentation generators produce formatted documentation for Icon programmers and for C programmers maintaining the Icon implementation. The documentation generators are responsible for inserting newline characters at reasonable points when printing the description.

G.2 Types of Operations

Rtt can be used to define the built-in functions, operators, and keywords of the Icon language. (Note that there are some Icon constructs that fall outside this implementation specification system. These include control structures such as string scanning and limitation, along with record constructors and field references.)

```
   operation-definition ::=
      function result-seq identifier ( [ param-list ] ) [ declare
] actions end |
      operator result-seq op identifier ( [ param-list ] )
[ declare ] actions end |
      keyword result-seq identifier actions end |
      keyword result-seq identifier const key-const end
   result-seq ::= { length , length [ + ] } |
                  { length [ + ] } |
                  { }
   length ::= integer | *
```

result-seq indicates the minimum and maximum length of the result sequence of an operation (the operation is treated as if it is used in a context where it produces all of its results). For example, addition always produces one result so its *result-seq* is {1, 1}. If the minimum and maximum are the same, only one number need be given, so the *result-seq* for addition can be coded as {1}. A conditional operation can produce either no results (that is, it can fail) or it can produce one result, so its *result-seq* is {0, 1}. A length of indicates unbounded, so the *result-seq* of ! is indicated by {0, }. An in the lower bound means the same thing as 0, so {0, } can be written as {, }, which simplifies to {}. A *result-seq* of {} indicates no result sequence. This is not the same as a zero-length result sequence, {0}; an operation with no result sequence does not even fail. exit is an example of such an operation.

A + following the length(s) in a *result-seq* indicates that the operation can be resumed to perform some side effect after producing its last result. All existing examples of such operations produce at most one result, performing a side effect in the process. The side effect on resumption is simply an undoing of the original side effect. An example of this is tab, which changes &pos as the side effect.

For functions and keywords, *identifier* is the name by which the operation is known within the Icon language (for keywords, *identifier* does not include the &). New functions and keywords can be added to the language by simply translating implementations for them. For operations, *op* is (usually) the symbol by which the operation is known within the Icon language and *identifier* is a descriptive name. It is possible to have more than one operation with the same *op* as long as they have different identifiers and take a different number of operands. In addition to translating the implementation for an operator, adding a new operator requires updating iconc's lexical analyzer and parser to know about the symbol (in reality, an *operator* definition may be used for operations with non-operator syntax, in which case any syntax may be used; iconc's code generator identifies the operation by the type of node put in the parse tree by a parser action). In all cases, the *identifier* is used to construct the name(s) of the C function(s) which implement the operation.

A *param-list* is a comma separated list of parameter declarations. Some operations, such as the write function, take a variable number of arguments. This is indicated by appending a pair of brackets enclosing an identifier to the last parameter declaration. This

last parameter is then an array containing the *tail* of the argument list, that is, those arguments not taken up by the preceding parameters. The identifier in brackets represents the length of the tail and has a type of C integer.

```
param-list ::= param { , param } [ [ identifier ] ]
```

Most operations need their arguments dereferenced. However, some operations, such as assignment, need undereferenced arguments and a few need both dereferenced and undereferenced versions of an argument. There are forms of parameter declarations to match each of these needs.

```
param ::= identifier |
             underef identifier |
             underef identifier -> identifier
```

A simple identifier indicates a dereferenced parameter. underef indicates an undereferenced parameter. In the third form of parameter declaration, the first identifier represents the undeferenced form of the argument and the second identifier represents the dereferenced form. This third form of declaration may not be used with the variable part of an argument list. These identifiers are of type *descriptor*. Descriptors are implemented as C structs. See Chapter 4 for a detailed explanation of descriptors. Examples of operation headers:

```
   detab(s,i,...) - replace tabs with spaces, with stops at
columns indicated.
   function{1} detab(s, i[n])
      actions
   end

   x <-> y -swap values of x and y.
   Reverses swap if resumed.
   operator{0,1+} <-> rswap(underef x -> dx, underef y -> dy)
      declare
      actions
   end

   &fail -just fail
   keyword{0} fail
      actions
   end
```

G.3 Declare Clause

Some operations need C declarations that are common to several actions. These can be declared within the declare clause.

```
declare ::= declare { C declarations }
```

These may include *tended* declarations, which are explained below in the section on extensions to C. If a declaration can be made local to a block of embedded C code, it is usually better to put it there than in a declare clause. This is explained below in the discussion of the body action.

Constant Keywords

Any keyword can be implemented using general *actions*. However, for constant keywords, iconc can sometimes produce more efficient code if it treats the keyword as a literal constant. Therefore, a special declaration is available for declaring keywords that

can be represented as Icon literals. The constant is introduced with the word const and can be one of four literal types.

```
      key-const ::= string-literal | cset-literal | integer-
literal | real-literal
```

When using this mechanism, it is important to be aware of the fact that rtt tokenizes these literals as C literals, not as Icon literals. The contents of string literals and character literals (used to represent cset literals) are not interpreted by rtt except for certain situations in string concatenation (see [.ipd65.]). They are simply stored, as is, in the data base. This means that literals with escape sequences can be used even when C and Icon would give them different interpretations. However, C does not recognize control escapes, so '^", which is a valid Icon literal, will result in an error message from rtt, because the second quote ends the literal, leaving the third quote dangling. Only decimal integer literals are allowed.

G.4 Actions

All operations other than constant keywords are implemented with general *actions*.

Actions fall into four categories: type checking and conversions, detail code expressed in extended C, abstract type computations, and error reporting.

```
   actions ::= { action }*
   action ::= checking-conversions |
              detail-code |
              abstract { type-computations } |
              runerr( msg_number [ , descriptor ] ) [ ; ]
             { actions }
```

Type Checking and Conversions

The type checking and conversions are

```
   checking-conversions ::= if type-check then action |
                            if type-check then action else action
|
                            type_case descriptor of { { type-
select }+ } |
                            len_case identifier of { { integer :
action }+ default : action }
   type-select ::= { type-name : }+ action |
                   default : action
```

These actions specify run-time operations. These operations could be performed in C, but specifying them in the implementation language gives the compiler information it can use to generate better code.

The if actions use the result of a *type-check* expression to select an action. The type_case action selects an action based on the type of a descriptor. If a type_case action contains a default clause, it must be last. *type-select* clauses must be mutually exclusive in their selection. The len_case action selects an action based on the length of the variable part of the argument list of the operation. The *identifier* in this action must be the one representing that length.

A *type-check* can succeed or fail. It is either an assertion of the type of a descriptor, a conversion of the type of a descriptor, or a logical expression involving *type-check*s. Only limited forms of logical expressions are supported.

```
   type-check ::= simple-check { && simple-check }* |
                  ! simple-check

   simple-check ::= is: type-name ( descriptor ) |
                    cnv: dest-type ( source [ , destination ] ) |
                    def: dest-type ( source , value [ ,
destination ] )

   dest-type ::= cset |
                 integer |
                 real |
                 string |
                 C_integer |
                 C_double |
                 C_string |
                 (exact)integer |
                 (exact)C_integer |
                 tmp_string |
                 tmp_cset
```

The is check succeeds if the value of the descriptor is in the type indicated by *type-name*. Conversions indicated by cnv are the conversions between the Icon types of cset, integer, real, and string. Conversions indicated by def are the same conversions with a default value to be used if the original value is null.

dest-type is the type to which to a value is to be converted, if possible. cset, integer, real, and string constitute a subset of *icon-type* which is in turn a subset of *type-name* (see below). C_integer, C_string, and C_double are conversions to internal C types that are easier to manipulate than descriptors. Each of these types corresponds to an Icon type. A conversion to an internal C type succeeds for the same values that a conversion to the corresponding Icon type succeeds. C_integer represents the C integer type used for integer values in the particular Icon implementation being compiled (typically, a 32-bit integer type). C-double represents the C double type. C-string represents a pointer to a null-terminated C character array. However, see below for a discussion of the destination for conversion to C_string. (exact) before integer or C_integer disallows conversions from reals or strings representing reals, that is, the conversion fails if the value being converted represents a real value.

Conversion to tmp_string is the same as conversion to string (the result is a descriptor), except that the string is only guaranteed to exist for the lifetime of the operation (the lifetime of a suspended operation extends until it can no longer be resumed). Conversion to tmp_string is generally less expensive than conversion to string and is never more expensive, but the resulting string must not be exported from the operation. tmp_cset is analogous to tmp_string.

The source of the conversion is the descriptor whose value is to be converted. If no destination is specified, the conversion is done ``in-place''. However, it may not actually be possible to do an argument conversion in the argument's original location, so the argument may be copied to another location as part of the conversion. Within the *scope* of the conversion, the parameter name refers to this new location. The scope of a conversion is usually only important for conversions to C types; the run-time system translator and

the Icon compiler try to keep the movement of descriptor parameters transparent (see below for more details). All elements of the variable part of an argument list must be descriptors. Therefore, when an element is converted to a C type, an explicit location must be given for the destination.

The destinations for conversions to cset, integer, real, string, (exact)integer, tmp_string, and tmp_cset must be descriptors. The destinations for conversions to C_integer, C_double, and (exact)C_integer must be the corresponding C types. However, the destination for conversion to C_string must be tended. If the destination is declared as ``tended char '', then the dword (string length) of the tended location will be set, but the operation will not have direct access to it. The variable will look like a ``char ''. Because the operation does not have access to the string length, it is not a good idea to change the pointer once it has been set by the conversion. If the destination is declared as a descriptor, the operation has access to both the pointer to the string and the string's length (which includes the terminating null character). If a parameter is converted to C_string and no explicit destination is given, the parameter will behave like a ``tended char '' within the scope of the conversion.

The second argument to the def conversion is the default value. The default value may be any C expression that evaluates to the correct type. These types are given in the following chart.

cset:	struct b_cset
integer:	C_integer
real:	double
string:	struct descrip
C_integer:	C_integer
C_double:	double
C_string:	char *
tmp_string:	struct descrip
tmp_cset:	struct b_cset
(exact)integer:	C_integer
(exact)C_integer:	C_integer

The numeric operators provide good examples of how conversions are used:

```
operator{1} / divide(x, y)
   if cnv:(exact)C_integer(x) && cnv:(exact)C_integer(y) then
      actions
   else {
      if !cnv:C_double(x) then
         runerr(102, x)
      if !cnv:C_double(y) then
```

```
            runerr(102, y)
         actions
         }
   end
```

Within the code indicated by *actions*, x and y refer to C values rather than to the Icon descriptors of the unconverted parameters.

The subject of any type check or type conversion must be an unmodified parameter. For example, once an in-place conversion has been applied to a parameter, another conversion may not be applied to the same parameter. This helps insure that type computations in iconc only involve the unmodified types of arguments, simplifying those computations. This restriction does not apply to type checking and conversions in C code.

Scope of Conversions

The following discussion is included mostly for completeness. The scope of conversions sounds complicated, but in practice problems seldom occur in code that ``looks reasonable''. If a problem does occur, the translator catches it. Normally, the intricacies of scope should be ignored and the person writing run-time routines should code conversions in a manner that seems natural.

An "in-place" conversion of a parameter can create a scope for the parameter name separate from the one introduced by the parameter list. This is because conversions to C types may require the converted value to be placed in a different location with a different type. The parameter name is then associated with this new location. The original scope of a parameter starts at the beginning of the operation's definition. The scope of a conversion starts at the conversion. A scope extends through all code that may be executed after the scope's beginning, up to a runerr or a conversion that hides the previous scope (because the type checking portion of the implementation language does not contain loops or arbitrary gotos, scope can easily be determined lexically).

The use of an in-place conversion in the first sub-expression of a conjunction, *cnv1 && cnv2*, has a potential for causing problems. In general, there is no way to know whether the first conversion will effectively be undone when the second conversion fails. If the first conversion is actually done in-place, the parameter name refers to the same location in both the success and failure scope of the conjunction, so the conversion is not undone. If the conversion is done into a separate location, the failure scope will refer to the original value, so the conversion will effectively be undone. Whether the conversion is actually done in-place depends on the context in which operation is used. However, conversion to C_integer and C_double always preserve the original value, so there is no potential problem using them as the first argument to a conjunction, nor is there any problem using a non-conversion test there. An example of this uncertainty:

```
   if cnv:string(s1) && cnv:string(s2) then {
      /* s1 and s2 both refer to converted values */
      }
   else { /* s2 refers to the original value. s1 may
             refer to either the original or the converted value
*/
      }
```

The translator issues a warning if there is a potential problem.

It is possible for scopes to overlap; this happens because scopes start within conditional actions. In rare instances, executable code using the name may appear within this overlapping scope, as in the following example, which resembles code that might be found in the definition of a string analysis function such as find.

```
if is:null(s) then {
   if !def:C_integer(i, k_pos) then
      runerr(101, i)
   }
else {
   if !def:C_integer(i, 1) then
      runerr(101, i)
   actions
```

Here, *actions* occurs within the scope of both conversions. Note that *actions* is not in the scope of the original parameter i. This is because that scope is ended in each branch of the outer if by the conversions and the runerrs.

If overlap does occur, the translator tries to insure that the same location is used for the name in each scope. The only situation when it cannot do this is when the type of the location is different in each scope, for instance, one is a C_integer and the other is a C_real. If a name is referenced when there is conflicting scope, the translator issues an error message.

Type Names

The *type-name*s represent types of Icon intermediate values, including variable references. These are the values that enter and leave an operation; ``types'' internal to data structures, such as list element blocks, are handled completely within the C code.

```
type-name ::= empty_type |
                 icon-type |
                 variable-ref
icon-type ::= null |
              string |
              cset |
              integer |
              real |
              file |
              list |
              set |
              table |
              record |
              procedure |
              co_expression
variable-ref ::= variable |
                 tvsubs |
                 tvtbl |
                 kywdint |
                 kywdpos |
                 kywdsubj
```

The *type-name*s are not limited to the first-class types of Icon's language definition. The *type-name*s that do not follow directly from Icon types need further explanation. empty_type is the type containing no values and is needed for conveying certain information to the type inferencing system, such as an unreachable state. For example, the

result type of stop is empty_type. It may also be used as the internal type of an empty structure. Contrast this with null, which consists of the null value.

Variable references are not first-class values in Icon; they cannot be assigned to variables. However, they do appear in the definition of Icon as arguments to assignments and as the subject of dereferencing. For example, the semantics of the expression

```
s[3] := s
```

can be described in terms of a substring trapped variable and a simple variable reference. For this reason, it is necessary to include these references in the type system of the implementation language. variable consists of all variable references. It contains five distinguished subtypes. tvsubs contains all substring trapped variables. tvtbl contains all table-element trapped variables. kywdint contains &random and &trace. kywdpos contains &pos. kywdsubj contains &subject.

Including C Code

As noted above, C declarations can be included in a declare clause. Embedded C code may reference these declarations as well as declarations global to the operation.

Executable C code can be included using one of two actions.

```
detail-code ::= body { extended-C } |
                inline { extended-C }
```

body and inline are similar to each other, except that inline indicates code that is reasonable for the compiler to put in-line when it can. body indicates that for the in-line version of the operation, this piece of C code should be put in a separate function in the link library and the body action should be replaced by a call to that function. Any parameters or variables from the declare clause needed by the function must be passed as arguments to the function. Therefore, it is more efficient to declare variables needed by a body action within that body than within the declare. However, the scope of these local variables is limited to the body action.

Most Icon keywords provide examples of operations that should be generated in-line. In the following example, nulldesc is a global variable of type descriptor. It is defined in the include files automatically included by rtt.

```
&null - the null value.
keyword{1} null
   abstract {
      return null
      }
   inline {
      return nulldesc;
      }
end
```

Error Reporting

```
runerr( msg_number [ , descriptor ] ) [ ; ]
```

runerr is translated into a call to the run-time error handling routine. Specifying this as a separate action rather than a C expression within a body or inline action gives the compiler additional information about the behavior of the operation. *msg_number* is the

number used to look up the error message in a run-time error table. If a descriptor is given, it is taken to be the offending value.

Abstract Type Computations

```
abstract { type-computations }
```

The behavior of an operation with respect to types is a simplification of the full semantics of the operation. For example, the semantics of the function image is to produce the string representing its operand; its behavior in the type realm is described as simply returning some string. In general, a good simplification of an operation is too complicated to be automatically produced from the operation's implementation (of course, it is always possible to conclude that an operation can produce any type and can have any side effect, but that is hardly useful). For this reason, the programmer must use the abstract action to specify *type-computation*s.

```
type-computations ::= { store [ type ] = type [ ; ] } [ return type [ ; ] ]
```

type-computations consist of side effects and a statement of the result type of the operation. There must be exactly one return *type* along any path from the start of the operation to C code containing a return, suspend, or fail.

A side effect is represented as an assignment to the *store*. The store is analogous to program memory. Program memory is made up of locations containing values. The store is made up of locations containing types. A type represents a set of values, though only certain such sets correspond to types for the purpose of abstract type computations. Types may be basic types such as all Icon integers, or they may be composite types such as all Icon integers combined with all Icon strings. The rules for specifying types are given below. A location in the store may correspond to one location in program memory, or it may correspond to several or even an unbounded number of locations in program memory. The contents of a location in the store can be thought of as a conservative (that is, possibly overestimated) summary of values that might appear in the corresponding location(s) in program memory at run time.

Program memory can be accessed through a pointer. Similarly, the store can be indexed by a pointer type, using an expression of the form store[*type]*, to get at a given location. An Icon global variable has a location in program memory, and a reference to such a variable in an Icon program is treated as a pointer to that location. Similarly, an Icon global variable has a location in the store and, during type inferencing, a reference to the variable is interpreted as a pointer type indexing that location in the store. Because types can be composite, indexing into the store with a pointer type may actually index several locations. Suppose we have the following side effect

```
store[ type1 ] = type2
```

Suppose during type inferencing *type1* evaluates to a composite pointer type consisting of the pointer types for several global variables, then all corresponding locations in the store will be updated. If the above side effect is coded in the assignment operator, this situation might result from an Icon expression such as

```
every (x | y) := &null
```

In this example, it is obvious that both variables are changed to the null type. However, type inferencing can only deduce that at least one variable in the set is changed. Thus, it must assume that each could either be changed or left as is. It is only when the left hand

side of the side effect represents a unique program variable that type inferencing knows that the variable cannot be left as is. In the current implementation of type inferencing, assignment to a single named variable is the only side effect where type inferencing recognizes that the side effect will definitely occur.

Indexing into the store with a non-pointer type corresponds to assigning to a non-variable. Such an assignment results in error termination. Type inferencing ignores any non-pointer components in the index type; they represent execution paths that don't continue and thus contribute nothing to the types of expressions.

A type in an abstract type computation is of the form

```
type ::= type-name |
            type ( variable ) |
            attrb-ref |
            new type-name ( type { , type } ) |
            store [ type ] |
            type ++ type |
            type ** type |
            ( type )
```

The *type(variable)* expression allows type computations to be expressed in terms of the type of an argument to an operation. This must be an unmodified argument. That is, the abstract type computation involving this expression must not be within the scope of a conversion. This restriction simplifies the computations needed to perform type inferencing.

This expression is useful in several contexts, including operations that deal with structure types. The type system for a program may have several sub-types for a structure type. The structure types are list, table, set, record, substring trapped variable, and table-element trapped variable. Each of these Icon types is a composite type within the type computations, rather than a basic type. Thus the type inferencing system may be able to determine a more accurate type for an argument than can be expressed with a *type-name*. For example, it is more accurate to use

```
if is:list(x) then
   abstract {
      return type(x)
      }
   actions
else
   runerr(108, x)
```

than it is to use

```
if is:list(x) then
   abstract {
      return list
      }
   actions
else
   runerr(108, x)
```

Structure values have internal ``structure''. Structure types also need an internal structure that summarizes the structure of the values they contain. This structure is implemented with type attributes. These attributes are referenced using dot notation:

```
attrb-ref ::= type . attrb-name
attrb-name ::= lst_elem |
```

```
set_elem |
key |
tbl_elem |
default |
all_fields |
str_var |
trpd_tbl
```

Just as values internal to structure values are stored in program memory, types internal to structure types are kept in the store. An attribute is a pointer type referencing a location in the store.

A list is made up of (unnamed) variables. The lst_elem attribute of a list type is a type representing all the variables contained in all the lists in the type. For example, part of the code for the bang operator is as follows, where dx is the dereferenced operand.

```
type_case dx of {
   list: {
      abstract {
         return type(dx).lst_elem
         }
      actions
      }
   ...
```

This code fragment indicates that, if the argument to bang is in a list type, bang returns some variable from some list in that type. In the type realm, bang returns a basic pointer type.

The set_elem attribute of a set type is similar. The locations of a set never ``escape'' as variables. That is, it is not possible to assign to an element of a set. This is reflected in the fact that a set_elem is always used as the index to the store and is never assigned to another location or returned from an operation. The case in the code from bang for sets is

```
set: {
   abstract {
      return store[type(dx).set_elem]
      }
   actions
   }
```

Tables types have three attributes. key references a location in the store containing the type of any possible key value in any table in the table type. tbl_elem references a location containing the type of any possible element in any table in the table type. default references a location containing the type of any possible default value for any table in the table type. Only tbl_elem corresponds to a variable in Icon. The others must appear as indexes into the store.

Record types are implemented with a location in the store for each field, but these locations cannot be accessed separately in the type computations of the implementation language. These are only needed separately during record creation and field reference, which are handled as special cases in the compiler. Each record type does have one attribute, all_fields, available to type computations. It is a composite type and includes the pointer types for each of the fields.

Substring trapped variables are implemented as structures. For this reason, they need structure types to describe them. The part of the structure of interest in type inferencing is the reference to the underlying variable. This is reflected in the one attribute of these

types, str_var. It is a reference to a location in the store containing the pointer types of the underlying the variables that are ``trapped''. str_var is only used as an index into the store; it is never exported from an operation.

Similarly table-element trapped variables need structure types to implement them. They have one attribute, trpd_tbl, referencing a location in the store containing the type of the underlying table. The key type is not kept separately in the trapped variable type; it must be immediately added to the table when a table-element trapped variable type is created. This pessimistically assumes that the key type will eventually be put in the table, but saves an attribute in the trapped variable for the key. trpd_tbl is only used as an index into the store; it is never exported from an operation.

The type computation, new, indicates that an invocation of the operation being implemented creates a new instance of a value in the specified structure type. For example, the implementation of the list function is

```
function{1} list(size, initial)
   abstract {
      return new list(type(initial))
      }
   actions
end
```

The type arguments to the new computation specify the initial values for the *attributes* of the structure. The table type is the only one that contains multiple attributes. (Note that record constructors are created during translation and are not specified via the implementation language.) Table attributes must be given in the order: key, tbl_elem, and default.

In the type system for a given program, a structure type is partitioned into several sub-types (these sub-types are only distinguished during type inferencing, not at run time). One of these sub-types is allocated for every easily recognized use of an operation that creates a new value for the structure type. Thus, the following Icon program has two list sub-types: one for each invocation of list.

```
procedure main()
   local x

   x := list(1, list(100))
end
```

Two operations are available for combining types. Union is denoted by the operator `++' and intersection is denoted by the operator `**'. Intersection has the higher precedence. These operations interpret types as sets of values. However, because types may be infinite, these sets are treated symbolically.

C Extensions

The C code included using the declare, body, and inline actions may contain several constructs beyond those of standard C. There are five categories of C extensions: access to interface variables, declarations, type conversions/type checks, signaling run-time errors, and return statements.

In addition to their use in the body of an operation, the conversions and checks, run-time error, and declaration extensions may be used in ordinary C functions that are put through the implementation language translator.

Interface Variables

Interface variables include parameters, the identifier for length of the variable part of an argument list, and the special variable result. Unconverted parameters, converted parameters with Icon types, and converted parameters with the internal types tmp_string and tmp_cset are descriptors and within the C code have the type struct descrip. Converted parameters with the internal type of C_integer have some signed integer type within the C code, but exactly which C integer type varies between systems. This type has been set up using a typedef in the automatically included include file so it is available for use in declarations in C code. Converted parameters with the internal type of C_double have the type double within the C code. Converted parameters of the type C_string have the type char . The length of the variable part of a argument list has the type *int* within the C code.

result is a special descriptor variable. Under some circumstances it is more efficient to construct a return value in this descriptor than to use other methods. See Section 5 of the implementation language reference manual for details.

Declarations

The extension to declarations consists of a new storage class specifier, tended (register is an example of an existing storage class specifier). Understanding its use requires some knowledge of Icon storage management. Only a brief description of storage management is given here; see the Icon implementation book for further details.

Icon values are represented by descriptors. A descriptor contains both type information and value information. For large values (everything other than integers and the null value) the descriptor only contains a pointer to the value, which resides elsewhere. When such a value is dynamically created, memory for it is allocated from one of several memory regions. Strings are allocated from the *string region*. All other relocatable values are allocated from the *block region*. The only non-relocatable values are co-expression stacks and co-expression activation blocks. On some systems non-relocatable values are allocated in the *static region*. On other systems there is no static region and these values are allocated using the C malloc function.

When a storage request is made to a region and there is not enough room in that region, a *garbage collection* occurs. All *reachable* values for each region are located. Values in the string and block regions are moved into a contiguous area at the bottom of the region, creating (hopefully) free space at the end of the region. Unreachable co-expression stacks and activator blocks are ``freed''. The garbage collector must be able to recognize and save all values that might be referenced after the garbage collection and it must be able to find and update all pointers to the relocated values. Operation arguments that contain pointers into one of these regions can always be found by garbage collection. The implementations of many operations need other descriptors or pointers into memory regions. The tended storage class identifies those descriptors and pointers that may have

live values when a garbage collection could occur (that is, when a memory allocation is performed).

A descriptor is implemented as a C struct named descrip, so an example of a tended descriptor declaration is

```
tended struct descrip d;
```

Blocks are also implemented as C structs. The following list illustrates the types of block pointers that may be tended.

```
tended struct b_real *bp;
tended struct b_cset *bp;
tended struct b_file *bp;
tended struct b_proc *bp;
tended struct b_list *bp;
tended struct b_lelem *bp;
tended struct b_table *bp;
tended struct b_telem *bp;
tended struct b_set *bp;
tended struct b_selem *bp;
tended struct b_record *bp;
tended struct b_tvkywd *bp;
tended struct b_tvsubs *bp;
tended struct b_tvtbl *bp;
tended struct b_refresh *bp;
tended struct b_coexpr *cp;
```

Alternatively, a union pointer can be used to tend a pointer to any kind of block.

```
tended union block *bp;
```

Character pointers may also be tended. However, garbage collection needs a length associated with a pointer into the string region. Unlike values in the block region, the strings themselves do not have a length stored with them. Garbage collection treats a tended character pointer as a zero-length string. These character pointers are almost always pointers into some string, so garbage collection effectively treats them as zero-length substrings of the strings. The string as a whole must be tended by some descriptor so that it is preserved. The purpose of tending a character pointer is to insure that the pointer is relocated with the string it points into. An example is

```
tended char *s1, *s2;
```

Tended arrays are not supported. tended may only be used with variables of local scope. tended and register are mutually exclusive. If no initial value is given, one is supplied that is consistent with garbage collection.

Type Conversions/Type Checks

Some conditional expressions have been added to C. These are based on type checks in the type specification part of the implementation language.

```
is: type-name ( source )
cnv: dest-type ( source , destination )
def: dest-type ( source , value , destination )
```

source must be an Icon value, that is, a descriptor. *destination* must be a variable whose type is consistent with the conversion. These type checks may appear anywhere a conditional expression is valid in a C program. Note that is, cvn, and def are reserved words to distinguish them from labels.

The type_case statement may be used in extended C. This statement has the same form as the corresponding action, but in this context, C code replaces the *action*s in the *type-select* clauses.

Signaling Run-time Errors

runerr is used for signaling run-time errors. It acts like a function but may take either 1 or 2 arguments. The first argument is the error number. If the error has an associated value, the second argument is a descriptor containing that value.

Return Statements

There are three statements for leaving the execution of an operation. These are analogous to the corresponding expressions in the Icon language.

```
ret-statments ::= return ret-value ; |
                  suspend ret-value ; |
                  fail ;
ret-value ::= descriptor |
              C_integer expression |
              C_double expression |
              C_string expression |
              descript-constructor
```

descriptor is an expression of type struct descrip. For example

```
{
   tended struct descrip dp;
   ...
   suspend dp;
   ...
}
```

Use of C_integer, C_double, or C_string to prefix an expression indicates that the expression evaluates to the indicated C type and not to a descriptor. When necessary, a descriptor is constructed from the result of the expression, but when possible the Icon compiler produces code that can use the raw C value (See Section 5 of the implementation language reference manual). As an example, the integer case in the divide operation is simply

```
inline {
   return C_integer x / y;
   }
```

Note that a returned C string must not be in a local (dynamic) character array; it must have a global lifetime.

A *descript-constructor* is an expression that explicitly converts a pointer into a descriptor. It is only valid in a return statement, because it builds the descriptor in the implicit location of the return value.

```
descript-constructor ::= string ( length , char-ptr ) |
                         cset ( block-ptr ) |
                         real ( block-ptr ) |
                         file ( block-ptr ) |
                         procedure ( block-ptr ) |
                         list ( block-ptr ) |
```

```
                                set ( block-ptr ) |
                                record ( block-ptr ) |
                                table ( block-ptr ) |
                                co_expression ( stack-ptr ) |
                                tvtbl ( block-ptr ) |
                                named_var ( descr-ptr ) |
                                struct_var ( descr-ptr , block-ptr )
|
                                substr ( descr-ptr , start , len ) |
                                kywdint ( descr-ptr ) |
                                kywdpos ( descr-ptr ) |
                                kywdsubj ( descr-ptr )
```

The arguments to string are the length of the string and the pointer to the start of the string. *block-ptr*s are pointers to blocks of the corresponding types. *stack-ptr* is a pointer to a co-expression stack. *descr-ptr* is a pointer to a descriptor. named_var is used to create a reference to a variable (descriptor) that is not in a block. struct_var is used to create a reference to a variable that is in a block. The Icon garbage collector works in terms of whole blocks. It cannot preserve just a single variable in the block, so the descriptor referencing a variable must contain enough information for the garbage collector to find the start of the block. That is what the *block-ptr* is for. substr creates a substring trapped variable for the given descriptor, starting point within the string, and length. kywdint, kywdpos, and kywdsubj create references to keyword variables.

Note that returning either C_double *expression* or substr(*descr-ptr, start, len*) may trigger a garbage collection.

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Text may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

References

[ASU86] Aho, Alfred, Sethi, Ravi, and Ullman, Jeffrey. Compilers, Principles Techniques and Tools. Addison-Wesley, 1986.

[Foley82] Foley, J.D; and A.Van Dam. Fundamentals of Interactive Computer Graphics. Reading, MA: Addison-Wesley Publishing Company, 1982.

[Griswold96] Griswold, Ralph E and Griswold, Madge T. The Icon Programming Language, Third Edition. San Jose, CA: Peer-To-Peer Communications, 1996.

[Griswold98] Griswold, Ralph E.; Jeffery, Clinton L.; and Townsend, Gregg M. Graphics Programming in Icon. San Jose, CA: Peer-To-Peer Communications, 1998.

[Griswold71] Griswold, Poage, and Polonsky . The SNOBOL 4 Programming Language, 2nd ed. Englewood Cliffs, N.J. Prentice-Hall, Inc. 1971.

[Jeffery99] Clinton L. Jeffery. Program Monitoring and Visualization: An Exploratory Approach. Springer-Verlag, New York, NY. 1999.

[Jeffery04] Jeffery, Clinton; Mohamed, Shamim; Pereda, Ray; and Parlett, Robert. Programming with Unicon. Draft manuscript from http://unicon.org

[LeH91] Arnaud LeHors. The X PixMap Format. Groupe Bull, Koala Project, INRIA, France, 1991.

[Nye88] Adrian Nye, editor. Xlib Reference Manual. O'Reilly & Associates, Inc., Sebastopol, California, 1988.

[OpenGL99] OpenGL Architecture Review Board; Woo, Mason; Neider, Jackie; Davis, Tom; Shreiner, Dave. OpenGL Programming Guide: the Official Guide to Learning OpenGL, Third Edition. Reading, MA: Addison-Wesley Publishing Company, 1999.

[OpenGL00] OpenGL Architecture Review Board; Shreiner, Dave. OpenGL Programming Guide: the Official Reference Document to OpenGL, Third Edition. Upper Saddle Reading, MA: Addison-Wesley Publishing Company, 2000.

[TGJ96] Gregg M. Townsend, Ralph E. Griswold, and Clinton L. Jeffery. Configuring the Source Code for Version 9 of Icon; Technical Report IPD238c, Department of Computer Science, University of Arizona, April 1996.
http://www.cs.arizona.edu/icon/docs/ipd238.htm.

[TGJ98] Gregg M. Townsend, Ralph E. Griswold, and Clinton L. Jeffery. Installing Version 9 of Icon on UNIX Platforms; Technical Report IPD243e, Department of Computer Science, University of Arizona, February 1998.
http://www.cs.arizona.edu/icon/docs/ipd243.htm.

[Uhl88] StephenA. Uhler. MGR --- C Language Application Interface. Technical report, Bell Communications Research, July 1988.

[Wal94] Kenneth Walker. The Run-Time Implementation Language for Icon; http://www.cs.arizona.edu/icon/ftp/doc/ipd261.pdf. Technical Report IPD261, Department of Computer Science, University of Arizona, June 1994.

[Walker94] Walker, Kenneth; The Run-Time Implementation Language for Icon. Technical Report from http://www.cs.arizona.edu/icon/

[Rees 86] Jonathan Rees, William Clinger. et al. Revised Report on the Algorithmic Language Scheme. SIGPLAN Notices, 21:12, December 1986.

[Bartlett 89] J. Bartlett. SCHEME->C a Portable Scheme-to-C Compiler. Research Report 89/1. DEC Western Research Laboratory, January 1989.

[Yuasa] T. Yuasa and M. Hagiya. Kyoto Common Lisp Report. Research Institute for Mathematical Sciences, Kyoto University

[SR] Gregory R. Andrews, Ronald A. Olsson et al. An Overview of the SR Language and Implementation. TOPLAS 10:1, January 1988, pp 51-86.

[Weiner] J.L. Weiner and S. Ramakrishnan. A Piggy-back Compiler for Prolog. Proceeding of the 1988 Conference on Programming Language Design and Implementation, SIGPLAN Notices 23:7, July 1988, pp. 288-295.

[Stroustrup 86] B. Stroustrup. The C++ Programming Language. Addison-Wesley, 1986.

[peephole] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. Using Peephole Optimization on Intermediate Code. TOPLAS 4:1, January 1982.

[Wulf] William A. Wulf, Richard. K. Johnsson, Charles. B. Weinstock, Steven. O. Hobbs, Charles. M. Geschke. The Design of an Optimizing Compiler. American Elsevier Pub. Co., New York, 1975.

[denote] M. J. C. Gordon. The Denotational Description of Programming Languages, An Introduction. Springer, 1979.

[Stoy] J. E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, 1977.

[ansi-c] American National Standard for Information Systems. Programming Language - C, ANSI X3.159-1989. American National Standards Institute, New York, 1990.

[Prabhala] Bhaskaram Prabhala and Ravi Sethi. Efficient Computation of Expressions with Common Subexpressions. Fifth Annual ACM Symposium on Principles of Programming Languages, pp. 222-230, January 1978.

[Nilsson] J\(o/rgen Fischer Nilsson. On the Compilation of a Domain-Based Prolog. Information Processing; Richard Edward Allison Mason ed., North-Holland, 1983, pp. 293-299.

[Martinek] John Martinek and Kelvin Nilsen. Code Generation for the Temporary-Variable Icon Virtual Machine. Technical Report 89-9, Department of Computer Science, Iowa State University, December 1989.

[pntstr] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. Proceeding of the 1990 Conference on Programming Language Design and Implementation, SIGPLAN Notices 25:6, June 1990, pp. 296-310.

[depptr] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence Analysis for Pointer Variables. Proceeding of the 1989 Conference on Programming Language Design and Implementation, SIGPLAN Notices 24:7, July 1989, pp. 28-40.

[smltlk type] Norihisa Suzuki. Inferring Types in Smalltalk. Eighth Annual ACM Symposium on Principles of Programming Languages, pp. 187-199, January 1981.

[Milner] Robin Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences. 17:3, December 1978, pp. 348-375.

[unify] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle. JACM, 12:1, January 1965, pp. 23-41.

[ianl1] Ralph E. Griswold and Madge T. Griswold. The Icon Analyst #1, August 1990.

[johnk] John Kececioglu. Private Communication. November 1990.

[debray apr91] Saumya K. Debray. Private Communication. April 1991.

[wam] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

Index