PROGRAMMING IN POP-2

R. M. Burstall
J. S. Collins
R. J. Popplestone

at the University Press
Edinburgh

Printed in Great Britain by
Unwin Brothers Limited
The Gresham Press, Old Woking, Surrey.

ISBN 0 85224 197 6

CONTENTS

Preface vii

Part One
A quick guide to the main features of POP-2 1
R. M. Burstall & R. J. Popplestone

Part Two
A primer of POP-2 programming 3
R. M. Burstall & J. S. Collins

1. Introduction 3
2. Simple arithmetic 4
3. Statements, declarations, variables 7
4. The stack 9
5. Function declarations 11
6. Conditionals 14
7. Labels and goto statements 16
8. Printing results 19
9. Words 19
10. Lists, list processing 20
11. Lambda expressions 24
12. Recursion 25
13. Defining new operations 27
14. More about lists 28
15. Records 33
16. Example of record processing 36
17. Arrays 39
18. Strips 40
19. Partial application 42
20. Input and output facilities 47
21. Cancel and sections 52
22. Macros and popval 54
23. Jumpout 56
24. Useful standard functions 57

Appendix. Answers to exercises 59

Part Three
POP-2 reference manual 77
R. M. Burstall & R. J. Popplestone

1. Introduction 77
   1.1 Aims 77
   1.2 Main features 77
   1.3 Examples 78
   1.4 Notation for syntactic description 79
   1.5 Notation for functions 80

2. Items 81
   2.1. Simple and compound items 81
   2.2 Integers 82
   2.3 Reals 82
   2.4 Truth values 82
   2.5 Undefined 83
   2.6 Terminator 83

3. Variables 83
  3.1 Identifiers 83
  3.2 Declaration and initialization 84
  3.3 Cancellation 85
  3.4 Sections 85

4. Functions 87
  4.1 Definition of functions 87
  4.2 Application of functions 88
  4.3 Nonlocal variables 89
  4.4 Partial application 90
  4.5 Doublets 91
  4.6 Arithmetic operations 92

5. Expressions and Statements 93
  5.1 Expressions 93
  5.2 Precedence 95
  5.3 Statements and imperatives 96
  5.4 Labels and goto statements 96
  5.5 Assignment 97
  5.6 Comments 98

6. Conditionals 99
  6.1 Conditional expressions 99
  6.2 Conjunctions and disjunctions 100

7. Data Structures 100
  7.1 Functions of data structures 100
  7.2 Records 103
  7.3 Strips 104
  7.4 Garbage collection 105

8. Standard Structures 105
  8.1 References 105
  8.2 Pairs 106
  8.3 Lists 106
  8.4 Full strips and character strips 108
  8.5 Arrays 109
  8.6 Words 110
  8.7 Functions 110

9. Input and Output 111
  9.1 Input 111
  9.2 Output 113

10. Machine Code 114

11. Modes of Evaluation 114
  11.1 Immediate evaluation 114
  11.2 Macros 115
  11.3 Evaluation of program text 116

Appendix 1. Standard POP-2 character set 117

Appendix 2. Optional functions 117

Appendix 3. Changes to the reference manual which affect the language, made since the previous edition 120

83
83
84
85
85

87
87
88
89
90
91
92

93
93
95
96
96
97
98

99
99
100

100
100
103
104
105

105
105
106
106
108
109
110
110

111
111
113

114

114
114
115
116

117

117

h affect the 120
ion

Part Four: Program Library 123

LIB ALLSORT	A program which sorts lists of items into the order specified by the user.	124
LIB ASSOC	A package for associating pairs of items to construct and handle association sets.	127
LIB CALL AND EXPLAIN	A package for monitoring POP-2 functions to allow conversational tutoring in the use of POP-2 programs.	133
LIB DCOMPILE	A simple line editor for use with the disc.	138
LIB DEBUG	A powerful debugging aid for POP-2 programs.	141
LIB EASYFILE	A simple but comprehensive disc filing system.	145
LIB EQUATIONS	A teaching aid which provides a simple facility for checking the manipulation of algebraic expressions.	160
LIB FOR	Provides FOR statements (DO loops) in POP-2.	164
LIB FOURS	Plays a game of three-dimensional noughts and crosses with the user.	168
LIB FULL MEMOFNS	A program generalizing the memo function facility so as to handle arguments and results grouped into sets.	175
LIB GRAPH TRAVERSER	A heuristic problem-solving algorithm.	184
LIB INDEX	A program to sort index references into alphabetical order.	192
LIB INVTRIG	Contains Arcsin, Arctan and Arccos.	196
LIB KALAH	A program which plays the game of Kalah with the user.	198
LIB MATRIX	A package which gives matrix handling facilities.	204
LIB MEMOFNS	A program which improves the run-time speed of POP-2 programs.	209
LIB NEW STRUCTURES	A program to provide non-numerical arrays and pseudo-records.	215
LIB PLOT	Allows users to plot, or tabulate, functions over given ranges.	221
LIB POPEDIT	A file-editing program.	227
LIB POPSTATS	A conversational, online, statistical package.	234

LIB PROOF CHECKER	A program which enables a person to develop proofs of predicate calculus theorems by the Resolution method.	248
LIB QUIZZING MACHINE	A conversational self tutorial program which quizzes the user in selected subjects.	261
LIB RANDOM	A pseudo-random number generator.	272
LIB RANDPACK	A package for generating pseudo-random numbers in several standard distributions.	274
LIB SETS	A collection of useful list-processing functions.	276
Note added in proof.	A language extension: generalized jumps and back-tracking	279

Bibliography 283

Index to the primer 285

Indexes to reference manual 287
- Technical terms 287
- Syntax definitions 288
- Standard functions and variables 288
- Optional functions 289
- Syntax words 290

rson to lculus ethod. 248

pro- n 261

erator. 272

do- andard 274

essing 276

l jumps 279

283

285

287

287

288

288

289

290

PREFACE

POP-2 is a programming language designed by R. M. Burstall and R. J. Popplestone and based on R. J. Popplestone's POP-1 (1968). POP-2 differs from most programming languages because it is designed for non-numerical as well as numerical applications. In addition, POP-2 is a conversational language allowing the user to communicate with his program and vice versa while it is running. This conversational property enables POP-2 to be used as a rather powerful calculating machine as well as a conventional computing system. A combination of these two modes produces a tool which can be matched to the particular problem being solved.

The first part of this publication is a quick guide to the aims and features of POP-2.

The Primer of POP-2 Programming, forming the second part, acquaints the reader with the POP-2 language and its terminology. Not all details of the language are described; enough is described, however, to provide the reader with a solid foundation in the language so that further questions about it can be directed to the Reference Manual. Although no previous experience of programming languages is required on the part of the reader, any such experience will be an advantage.

The Reference Manual, forming the third part, is a precise definition of the POP-2 language. Although the Primer can be read in its entirety before looking at the Reference Manual, it is recommended that the two parts are read in conjunction with each other. Thus, having read a section of the Primer on, say, conditionals, the corresponding section of the Reference Manual should be studied.

The final part consists of descriptions and listings of programs from the software library, mostly written at Edinburgh, and tested on the ICL 4100 system there. It illustrates the scope of the language and displays a number of programming tools and techniques. Examples are DEBUG—a debugging and tracing aid—and EASYFILE, which provides a disc filing system for programs and data.

The Reference Manual was originally published in *Machine Intelligence 2* (Edinburgh University Press 1968) and then republished with a brief introduction in *POP-2 Papers* by Oliver and Boyd (later distributed by Edinburgh University Press).

Since the reference manual was completed two years ago a number of errors, ambiguities, and omissions have come to light, and further experience of using the language has shown the need for a few minor changes and two additions of some significance. The first of these additions is *jumpout,* a facility which allows immediate exit from execution of one or more nested function bodies (see also the note on p. 279 on an extension giving generalized jumps and back-tracking). The second is the ability to break the program down into *sections,* somewhat analogous to ALGOL blocks, thus preventing clashes of identifiers.

The significant changes are listed in Appendix 3 of the Reference Manual. Many existing POP-2 programs should run unchanged and the remainder will need only a few simple alterations.

The 'Introduction to POP-2' which originally accompanied the reference manual, was brief and covered only the main points of the language. It has now been rewritten and greatly expanded to appear here as 'A primer of POP-2 programming'.

Acknowledgements

The suggestions and criticism of many members of the Department of Machine Intelligence and Perception at Edinburgh University, especially Mr Ray Dunn, and of others outside it, especially Mr Michael Healy and Mr Michael Woodger and those who have implemented POP-2 systems, are gratefully acknowledged. Technical consultations with Dr Michael Foster and Dr David Park about storage control were valuable. Thanks are due to Mrs Pat Ambler who has helped in the editing of this book, to Mr Bruce Anderson who has provided answers to the exercises in the primer and made many useful comments, and to Miss Eleanor Kerse who typed the new manuscript through numerous drafts.

POP-2 has been implemented on the ICL 4100 by Messrs Robin Popplestone, Ray Dunn, and David Pullin, on the ICL 1900 by Mr John Scott, and on the ICL System-4 and IBM 360 by Mr John Barnes and Mr Rod Steel (using a machine independent version of the compiler written in POP-2). The library of programs has been built up by many hands and organized by Mr Ray Dunn and Mr Robert Owen. The work has been part of a machine intelligence project, directed by Professor Donald Michie, whose guidance and encouragement have been invaluable, and supported by Edinburgh University, the Science Research Council, and the Medical Research Council.

We would like to thank Edinburgh University Press for their painstaking work on a very technical book, especially Dr Helen Muirhead, whose care and patience never failed us.

R. M. Burstall (Editor)

Department of
ersity, especially
ichael Healy and
POP-2 systems,
vith Dr Michael
valuable. Thanks
ng of this book,
e exercises in
ss Eleanor
drafts.

rs Robin Popple-
Ir John Scott, and
nd Mr Rod Steel
vritten in POP-2).
ds and organized
een part of a
nald Michie,
e, and supported
, and the Medical

their painstaking
rhead, whose

PART 1. A QUICK GUIDE TO THE MAIN FEATURES OF POP-2

R. M. BURSTALL AND R. J. POPPLESTONE

SUMMARY

POP-2 is a new computer language. Conceptual affinities can be traced to
1. John McCarthy's LISP (1962), from which it takes ideas for handling non-numerical objects of computation (lists).
2. Christopher Strachey's CPL (1963) and Peter Landin's ISWIM (1966), which foreshadow the aim of making a programming language into a notation with full mathematical generality, akin to algebra.
3. Cliff Shaw's JOSS (1964), which it resembles in its 'conversational' facilities.
4. Robin Popplestone's POP-1 (1968) of which POP-2 represents a rationalized and greatly-extended development.

These ingredients have produced a powerful but compact language for non-numerical programming. POP-2 was designed for implementation on a medium-sized machine with a modest investment in system programming. Because the language had to be stripped down to the level of the basic mathematical principles of programming, it is unrestrictive and open-ended.

The main distinctive features of POP-2 are
1. The syntax is very simple but the programmer has some freedom to extend it.
2. The programmer can create a wide variety of data structures: words, arrays, strings, lists, and records. A 'garbage collector' automatically controls storage for him.
3. Functions can be used in the same manner as in mathematics or logic, for example, as arguments or results of other functions, with no unfortunate restrictions on free variables.
4. The novel device of 'partial application' allows one to fix the value of one or more parameters of the function. This has a surprising multiplicity of uses, for example, to generalize the notion of an array to non-numerical subscripts and to disguise the distinction between stored values and computed values.
5. Another technique, 'dynamic lists', enables a physical device like a paper tape reader to be treated as if it were an ordinary list.
6. The programmer can call for immediate execution of statements at any time, giving facilities for conversational use and rapid debugging of complex programs.
7. The facility for immediate execution together with the variety of data structures available makes POP-2 suitable for use as the control language of a time-sharing system, enabling the user to effect filing, editing, compilation, and execution.
8. In the context of the widespread shortage of system programmers, a crucial feature is the open-endedness of the language. Work normally done in machine code by highly-skilled system programmers can be done in POP-2 itself.
9. POP-2 is compact and easy to implement. On the ICL 4100, for example, the whole system for compiling, running, and time sharing occupies only 22K of core (24-bit words). The effort needed to construct the complete system was less than 5 man-years. A machine-independent POP-2 in POP-2 compiler has been written.

A V A I L A B I L I T Y

POP-2 compilers are now available for the ICL 4100, ICL 1900, ICL System 4, and IBM Systems 360 series of machines. A PDP-10 compiler is being written. Multi-POP/4130 is a single-language system for a 64-K machine with disc, serving 8 simultaneous users. The other three implementations in their present form provide POP-2 programming in single-user mode, time-shared with batch operations

These systems are available to academic or research bodies from the Department of Machine Intelligence and Perception, and through arrangements with the National Research and Development Corporati from Conversational Software Ltd. CSL will also contract, on suitable terms, to develop extended versions of the present systems, and also new POP systems for other machine ranges. Enquiries may be addressed to

POP-2 Enquiries
Department of Machine Intelligence and Perception
Forrest Hill
Edinburgh EH1 2QL.

or to

Conversational Software Ltd
Hope Park Square
Edinburgh EH8 9NW

PART 2. A PRIMER OF POP-2 PROGRAMMING

R. M. BURSTALL AND J. S. COLLINS

1. INTRODUCTION

Two important features of the POP-2 programming language distinguishing it from many other languages are its inherent ability to be used in an on-line mode and the fact that it is not restricted to numerical manipulations.

One way of using a computer to solve a problem is to specify the problem, write a program to solve the problem, keypunch the program and have it executed by the computer. This method of using a computer assumes that: the problem is well defined; an accurate program is available; the user is a perfect typist. In some cases, such as routine data processing, these conditions are met fairly easily. In many cases, however, such as for computing research or any other research problem, or when the user is not an experienced programmer, these conditions cannot easily be met. It is then necessary for a *dialogue* to take place between the user and the machine. In this dialogue, the computer is asked to perform some computation. Having studied the results, the user requests another computation. The dialogue continues in a series of steps, each of which depends on what has happened up to that point.

In this situation, the user must be able to request the computer to carry out tasks without having to specify them all at the start of the session. POP-2 is a language designed for this kind of use. A fundamental property of the language is the ease with which the language can be extended in *ad hoc* directions.

Using a calculating machine is clearly an on-line activity with alternate action by the user and the machine. A notable deficiency of the calculating machine is that it can only execute one step at a time, so that the user is forced to interact with the machine even when he knows what the next step will be. Extending the POP-2 language is like adding extra keys to a calculating machine and attaching to them a meaning defined in terms of existing operations.

NON-NUMERICAL COMPUTING

Most programming languages fall into the class of either commercial or scientific programming languages. COBOL, a well-known example of a commercial programming language, facilitates the writing of programs to manipulate large quantities of information, such as payroll files. ALGOL and FORTRAN are well-known examples of scientific programming languages. Both are particularly suitable for engineering-type calculations where the bulk of the computing is simply arithmetic. The more recently introduced PL/1 attempts to meet both commercial and scientific requirements. None of these languages, however, is really suitable for writing programs to play chess, prove theorems, or carry out other complex non-numerical activities. This deficiency has long been felt and list-processing languages such as LISP, or text-processing languages such as COMIT have been developed for this type of application. Both LISP and COMIT are classed as non-numerical languages.

It is not easy to class POP-2 with the above languages. It has the basic numerical capability of ALGOL, the list-processing capability of LISP, and elementary record-handling facilities similar to those of COBOL. POP-2 is not, however, just a mixture of these languages; it is essentially a simple language which includes the fundamental concepts of FORTRAN, ALGOL, LISP, and COBOL, and has the ability to add new features in a natural way. For example, it is easy to write a matrix-processing package in POP-2 which enables the user to write arithmetic expressions of any complexity involving matrices.

To get a quick idea of the flavour of the language it may be helpful to look at the example of POP-2 program text in section 1.3 of the Reference Manual (see Part 3) and at some of the programs in the Program Library (see Part 4).

USING A POP-2 SYSTEM

The Reference Manual defines fully the POP-2 language. It does not, however, deal with problems like correcting typing errors, punching POP-2 programs, or getting permission to log into a POP-2 system; these are likely to vary from one installation to another. Each implementation of POP-2 for a particular computer system is described in a *functional specification* for the particular implementation. There are, therefore, as many functional specifications as there are different computer systems for which POP-2 has been implemented. Before any of the examples or exercises in this book can be tried out, the appropriate POP-2 functional specification must be consulted. It will describe what peripheral devices are available, how to log into the system, how the user is charged, and all such details.

A feature of most POP-2 systems is the *console,* through which communication takes place between the user and his program. This is usually a teleprinter, which allows the user to type his requests on the keyboard and the computer to print the results. Some POP-2 systems use another input-output mode, such as punched cards, as the main means of communication, but we will talk here as if a console were being used.

2. SIMPLE ARITHMETIC

The simplest use of a POP-2 system is as a rather high-powered calculating machine. Having logged into the system, the system is ready to execute any POP-2 statement we type. If we type the statement 2 + 2 => the answer ** 4 appears almost immediately. The *print arrow* sign => indicates that we wish the value of the preceding arithmetic expression to be printed. All results printed by this sign are preceded by the double asterisk.

The rules for writing numbers are very free. They are fully defined in sections 2.2 and 2.3 of the Reference Manual. Briefly, numbers may be integers or reals. Integers are written without a decimal point as a sequence of digits. Reals are written with a decimal point with at least one digit after the decimal point.

For example,
2.13   .2845   4.0
are legal reals but
4.
is not allowed.

Reals may have an *exponent part* consisting of the symbol $_{10}$ followed by a positive or negative integer. The integer is a power of ten by which the number is scaled.

For example,
$2.13_{10}1$ $.213_{10}2$ $213.0_{10}{-1}$ 21.3
all represent the same number.

ARITHMETIC EXPRESSIONS

The usual arithmetic operations + — * (for multiply) and / (for divide) are available. Arithmetic expressions involving these operations are evaluated following the usual rules of arithmetic; multiplication and division are carried out before addition and subtraction.

Thus if we type the statement

```
12.0 + 2.5 * 3.16 — 4
```

on the POP-2 console, the result

```
**15.9
```

is printed because the multiplication operation is carried out first. Notice that reals and integers can be mixed in arithmetic expressions.

Two further arithmetic operations are provided. The exponential operation ↑ enables a value to be raised to a power. For example, the arithmetic expression

```
(—2.5) ↑ 2
```

means minus two point five squared and produces the value 6.25. Similarly 4.0 ↑ 0.5 has the value 2.0.

An alternative division operation which may be used only between two integers is provided. This is written as //. This integer division operation produces two results; the quotient and the remainder.

Thus if the statement

```
25 // 3=>
```

is typed on the console, the results

```
**1 , 8
```

are printed, because 3 goes into 25 eight times with remainder one. Notice that the print arrow is able to print a sequence of results as well as just a single result.

PRECEDENCE AND PARENTHESES

Each of the operations described in the previous section has a *precedence* associated with it. A precedence is a number which determines the order in which the operations are applied. Both + and — have a precedence of 5. *, /, and // however have a precedence of 4, indicating that these operations are applied before addition and subtraction. The precedence table for arithmetic operations is as shown below.

Operation	Precedence
↑	3
* / //	4
+ —	5

Using this table, it can be seen that the result of typing the statement

```
3 — 2.5 ↑ 2 * 1.5 / 3 =>.
```

on the POP-2 console will be

```
**—0.125
```

The order of evaluation is as follows:

Original expression		3 — 2.5 ↑ 2 * 1.5 / 3
Apply operations of precedence	3	3 — 6.25 * 1.5 / 3
" " " "	4	3 — 3.125
" " " "	5	—0.125

In the case of operators of equal precedence they are applied from left to right, thus 6/2 * 5 = 15, not 0.6.

The precedence associated with each operation defines an order of evaluation of an expression. If another order is required, parentheses can be used in the conventional way.

For example, the statement
(3 — 2.5)↑2 * 1.5 / 3 =>
produces the result
**0.125

The rules of precedence apply to each expression within a pair of parentheses.

Thus the statement
(3 — 2.5↑2) * 1.5 / 3 =>
produces the result
**—1.625

(The number of figures printed after the decimal point may vary from one computer implementation to another.) Parentheses may be nested to any depth, the expressions within inner parentheses being evaluated first.

STANDARD FUNCTIONS

A number of mathematical functions are available for the POP-2 user. The list may vary from one implementation to another, so the functional specification for the particular implementation should be consulted for the precise list. The usual list is:

sin	trigonometric	sine,	angle in radians
cos	"	cosine,	" " "
tan	"	tangent,	" " "
arctan	"	arc tangent	" " "
sqrt	square root		
log	natural logarithm		
exp	"	anti-logarithm	

In POP-2, the argument of a function is enclosed in parentheses after the name of the function. So √2 is written *sqrt* (2).

Thus the POP-2 statement
exp(*2* * *log*(*1.414*)) =>
produces the result
**2.0

The standard functions are listed at the end of the Reference Manual. So are some optional functions which an implementation of POP-2 on a particular computer may or may not provide (Appendix 2 of the Manual). Most implementations will provide a library of extra programs and functions with a simple method of compiling these, for example, *compile* (*library* (*[statistics]*)) might make available functions *mean, correlation, chisquare,* and so on, to do statistical tests. Part 4 'Program Library' gives descriptions and listings of many POP-2 library programs. They will be found to be of interest as examples of POP-2 programming. The contents of the library may vary from one implementation to another.

EXERCISES

Answers are given as an appendix to this primer.

1. The arithmetic features of POP-2 have been described in this section. These facilities enable a POP-2 console to be used in a calculating machine mode. What would you type on a POP-2 console to evaluate the following arithmetic expressions?

(a) $\dfrac{2.5 \times 2}{-1.5 \times 4}$ (b) $1 + 2(5 - 3)$ (c) $\sqrt{3^2 + 4^2}$

(d) $sin^2 0.13 + cos^2 0.13$ (e) $tan^{-1} 1.5$

2. What is the value of the following POP-2 expressions?
(a) $8/2 * 6$
(b) $1 + 2.0_{10}{-2} * 8$
(c) $7 * (sqrt(16) + 2)$

3. The following are not POP-2 expressions. Why not?
(a) $((2 * 3) + (4 - (6 + 3))$
(b) $sin\ 0.5$
(c) $1.25_{10}0.5$
(d) $6. + .5$

3. STATEMENTS, DECLARATIONS, AND VARIABLES

An *imperative* is a request to the POP-2 system to do something. An arithmetic expression followed by a print arrow is an imperative and requests that the expression be evaluated and the result printed. The imperative is actually carried out as soon as the print arrow is encountered. Thus, as well as being a printing operator, the print arrow is an imperative separator. The basic imperative separator is the semicolon and, in a sequence of imperatives, the individual imperatives must be separated from each other by a semicolon (or print arrow if appropriate).

There are two quite distinct types of imperative: the *declaration* and the *statement*. A declaration serves to introduce a new name or *identifier* by which some quantity will be known. The simplest kind of declaration introduces one or more *variables*. It consists of the word *vars* followed by the identifiers of the variables being introduced. For example, if we propose to use three variables called *x*, *y1* and *y2* then the declaration

vars *x y1 y2*

should be given. This declaration reserves space for the storage of three values. The three variables do not yet have any particular values. (Some POP-2 systems initialize them with values [*x.undef*], [*y1.undef*], and [*y2.undef*] respectively.) Identifiers can be made up of any group of letters and digits, beginning with a letter. Alternatively, they can be made up of any group of the following signs:

+ — * / $ & = <> : £ ↑
For example: ++ —*— :: ↑:↑

If more than eight characters are used the extra ones are ignored: thus *variable1* is the same identifier as *variable2*.

Some identifiers are reserved as the names of standard variables, mostly those whose values are standard functions such as *sin, cos,* and *sqrt.* Others are reserved for syntactic purposes such as =>, : , —>, **end**. These are called *syntax words.* Syntax words such as **end** are printed in bold face in this book to remind the reader that they are so used, but in the actual POP-2 text they are not distinguished from any other identifier. Attempts to use reserved identifiers, for example, by writing **vars** *end;* are illegal.

A statement is an imperative that causes some computation to take place. An expression followed by a print arrow is an example of a statement. Another type of statement is an *assignment,* which enables a new value to be assigned to a variable. The assignment

```
2 + 2 —> x
```

assigns the value 4 to the variable *x* replacing whatever was the previous value of *x*. Note that a semicolon must separate this statement from any statement or declaration preceding or following it. A typical imperative sequence might be:

```
vars x y1 y2;
2 + 2 —> x;
x + 1 —> y1;
y1 =>
**5
y1↑2 —> y2;
y2 =>
**25
```

Notice that, assuming this imperative sequence is typed on the console keyboard, the imperatives are executed one by one as they are typed.

It is quite all right to write

```
x + 1 —> x;
```

This means that the new value of *x* is to be the old value plus one.

If a variable is used without declaring it first, it will be automatically declared and a message will be printed to indicate that this has happened.

Note that only variables may appear on the right-hand side of an assignment (a later section will indicate how certain kinds of expression may appear in this position too). It is not correct to execute assignments such as

```
x —> 2;
or
5 —> x + y;
```

The first is wrong because 2 is a constant—not a variable. The second assignment is incorrect because *x* + *y* is an expression.

Note that it is illegal to attempt to assign a new value to a standard variable, for example, 2 —> *sin*;

The reader may have wondered when, if ever, it is necessary to put in spaces or newlines. There is no distinction between a space and a newline, or between these and any sequence of spaces and newlines. They all serve to separate sequences of characters which might otherwise be confused. For example, if we want to write the identifier *x*2 followed by the number 3 we must write *x*2 3 not *x*23 which would form a single identifier, but 3*x*2 would be a number followed by an

identifier since it could not form a single identifier. Similarly
/// —> *** is three identifiers whilst ///—>*** is one, but *x* —> *y* is the same as *x*—>*y*. In the case of real numbers spaces and newlines are not permitted in the middle of the number.

EXERCISES

1. Write assignments to exchange the value of two variables *x* and *y*. This can be done using a third variable.

2. What will be printed after typing in the following sequences of imperatives?

(a)
```
vars x1 x2;
3 —> x1; 5 —> x2;
x1 + x2 —> x2;
x1 + x2 =>
```

(b)
```
vars a b c
6 —> a; 7 —> b; a + b —> c;
c * a —> b;
a, b, c =>
```

3. Introduce a new temporary variable to write the following program more briefly and efficiently.

```
sqrt(sin(x + a)) * exp(sin(x + a)) =>
```

4. THE STACK

Consider the type of statement which consists of an arithmetic expression followed by a print arrow. This causes the arithmetic expression to be evaluated and the result to be printed on the console.

This process can be considered to take place in two stages:
(1) The arithmetic expression is evaluated.
(2) The result is printed.

In order that this can happen, the result obtained by evaluating the arithmetic expression must be left in some communication area for the print function to pick it up. This communication area is known as the *stack*. The stack is like a stack of cards on a table. When a new result is placed on the stack it becomes the new top of the stack and the first to be removed. The stack is therefore a *last-in-first-out* device.

Because the stack can accommodate more than one result, it is possible to write several expressions separated from each other by commas. For example,

```
2.5 + 3.6, 5 / 2, 2↑3
```
results in the three results 6.1, 2.5, and 8.0 being placed on the stack with 8.0 at the top.

The print arrow => does more than was implied in the previous section. It prints the entire stack, starting from the bottom, and empties it. Thus if the three expressions above were followed by a print arrow, the result

```
**6.1,2.5,8.0
```

would be printed and the stack would be left *empty*. Note that the top element of the stack is printed last. If we want to print just the top element instead of the whole stack we may write *pr*();.

We may write a whole sequence of statements which put numbers onto the stack or remove them from the stack. Thus the sequence of four statements

1;2;—> y;—> x;

puts 1 on the stack, then 2 on top of it, then removes 2 assigning it to *y*, then removes 1 assigning it to *x*, leaving the stack in its original state. We have now met the following kinds of statement

(a) expression =>
(b) expression;
(c) expression —> variable;
(d) —> variable;

In any of these the expression may be replaced by a sequence of expressions separated by commas, and in (c) or (d) the part —> *variable* may be replaced by a sequence of variables each preceded by an arrow. Thus instead of the sequence above we could write

1,2 —> y —> x;

CAUTION. Leaving numbers around on the stack and using them later is an easy way to make mistakes. Do not do it wantonly.

The standard function *stacklength* which has no arguments tells the number of items on the stack. Thus we can discover this by typing

pr(stacklength());

The standard function *setpop,* also of no arguments, clears the stack.

POP-2 functions take their parameters, if any, from the top of the stack, and leave their results, if any, on the stack. Thus the function *sin* removes the top item from the stack, computes its sine, and places this number on the stack. When we write

sin(0.143)

0.143 is loaded on the stack and function *sin* is called. If we write
sin( )

whatever is currently on top of the stack is used by the function *sin.* If this expression were executed with the stack empty, an error message would be printed indicating that the stack had underflowed.

Provided the number of parameters (arguments) put on the stack when a function is called is the same number as taken by the function, any numbers previously on the stack are unaffected by the transaction.

Note the difference between writing
sin
which loads the sin function itself onto the stack and
sin( )
which actually causes the *sin* function to be executed. An alternative way of writing the latter is
.sin
which has an identical effect.

There is a standard function *erase* which takes one parameter and produces no result. *Erase* simply removes one item from the stack. Thus if we type

erase(2 +2) =>

the expression in the parentheses will be evaluated but no result will be printed. A more useful example of *erase* is

erase (23//6) =>

which produces the result

```
**5
```

because the quotient, put on top of the stack by the integer division, is removed by the function *erase*.

EXERCISES

1. Assuming the functions *add* and *mult* replace the top two members of the stack with the sum and product respectively, what is left on the stack after executing the following?
2,3,4;add( ); mult( );

2. What is the effect of the following statements?
(a) *x,y —> x —> y;*
(b) *—> x —> y; x,y;*

3. Write a sequence of statements which exchange the first (top) and third items on the stack.

5. FUNCTION DECLARATIONS

Any POP-2 system will have a number of built-in standard functions such as square root. Facilities are provided to extend this basic set by defining new functions in terms of existing ones.

The *function definition:*

```
function sumsq x y;
x↑2 + y↑2
end
```

defines a new function called *sumsq* whose value is the sum of the squares of two numbers. *x* and *y* are called *formal parameters*. When the function is called, for example, by writing *sumsq(3, 4)*, these formal parameters will be assigned the values of the corresponding actual *parameters*, or arguments, that is, 3 and 4, before the expression in the body of the function definition is evaluated. Thus evaluating the expression *sumsq*(3, 4) causes the *body* of *sumsq*, that is, *x*↑2 + *y*↑2 to be evaluated with *x* initialized (given an initial value) to 3, and *y* initialized to 4. The value 25.0 is left on the stack as a result.

Note that a function definition does not cause any calculation to be done; it simply creates a new function and assigns it as the value of a variable, in this case the variable *sumsq*. If the variable has not been previously declared, the function definition acts as a declaration of it. We must distinguish the act of defining a function from that of calling it, that is, applying it to some parameters, when the function is actually used to perform a calculation.

It is useful to know something about what takes place when a function is called. When the expression

sumsq(3, 4)

is evaluated, the values of the actual parameters 3 and 4 are placed on the stack. The piece of program associated with *sumsq* is then entered.

Because *sumsq* has two arguments, it takes two items off the stack and assigns them to *y* and *x*. (Note that *y* will be on top of the stack and will be the first to be removed.) The expression

x↑2 + y↑2

is then evaluated using these values of *x* and *y* and the result is left on the stack. The variables *x* and *y* belong to the function *sumsq* and have no connection with variables having the same name that might exist outside the function definition.

As a matter of fact we could achieve the same result as above without using this parameter mechanism defining *sumsq* in the following way:

```
function sumsq;
vars x y;
-> y; -> x          (take two numbers off the stack and assign them
x↑2 + y↑2            to y and to x)
end
```

This produces the same effect because the parameter mechanism is defined to work in this way. The parameter mechanism is simply a shorthand notation for the above.

Notice that the *body* of a function definition, that is, the text occurring after the names of the function and its parameters, may be simply an expression, or it may be a sequence of statements. In either case it may include some declarations. It may also include the print arrow =>, so that calling the function may cause some values to be printed. If used inside a function, the print arrow => prints and removes *only the top item* on the stack.

If we have defined a function *f* and want to change it we simply redefine it. For example:

```
function f x;x + 1 end;
function f x;x  + 2 end;
f(3) =>
**5
```

LOCAL VARIABLES

Consider the following piece of program:

```
vars a;2-> a;
function f x;vars a;
    x↑3 -> a;
    a + a
end;
f(3),a =>
```

The value printed for *f*(3) is clearly 54.0, that is, $3^3 + 3^3$, but what has happened to *a*? Is it now 27.0, the value it assumed in the calculation of *f*(3), or is it still 2, as it was originally? In fact the value printed will be 2, showing that the evaluation of *f*(3) has not affected the value of *a*.

This is because one *identifier* can name more than one *variable*, and each variable may have a different value. When we write an identifier the context determines which variable is named by that identifier. When we pass through the declaration of an identifier a new variable is associated with that identifier. Thus **vars** *a* in the first line of the example above creates a new variable called *a*, and the **vars** *a* in the second line also creates a new variable called *a* whenever the function *f* is applied to some argument, for example, in the evaluation of *f*(3). When *f*(3) has been evaluated this new variable is no longer required, thus a variable declared in the body of a function ceases to exist when that body has been evaluated. Such a variable is called a *local* variable. When an identifier occurs in a statement it always denotes the variable which has most recently been associated with that identifier, excluding any variables which have ceased to exist. For this purpose mentioning an identifier as a formal parameter, for instance, *x* in the example above, also acts as a declaration, and the variable it creates ceases to exist when the function body has been evaluated.

This convention enables us to use new identifiers freely to name formal parameters and local variables in a function body, knowing that what is done inside the function body cannot affect any variables outside which happen to have the same name.

Variables which are declared outside any function body (not formal parameters or local variables) are called *global* variables.

Just as the parameter list provides a convenient facility for declaring variables which are given values from the stack, so it is possible to declare local variables whose values are automatically placed on the stack when execution of the function is finished. Such a variable is called an *output local*. The function *sumsq* could be defined using an output local in the following way:

```
function sumsq x y => z;
x↑2 + y↑2 –> z
end
```

In this case, *z* is a local variable whose value is placed on the stack at the end of executing the function *sumsq*. Although the sign => is used to separate the parameters from the output locals, used in this context it has nothing to do with printing values. It is particularly convenient for functions which produce more than one result, for example,

```
function bothroots x => posroot negroot;
    sqrt(x) –> posroot; –posroot –> negroot
end;
bothroots(2) =>
**1.414,–1.414
```

If the results were merely left on the stack, instead of using output locals, the function would still work, but anyone reading its definition would have to look quite carefully to notice that it produces two results and to tell which comes first. Thus using output locals helps to make the program more readable, it is a matter of taste not necessity.

If we have already declared a variable as a local variable of a given function we may not declare it again as a local to the same function, unless the new declaration is inside some interior function.

Thus

```
function f x; vars a; vars a; . . . end;
```

is illegal, but

```
function f x; vars a;
    function g y; vars a; . . . end
    . . .
end;
```

is all right.

If similar declarations occur twice globally, that is, outside any function body, the second one is simply ignored.

EXERCISES

1. Declare a function *roots* which takes three parameters *a*, *b*, and *c*, and produces, as results, the roots of the quadratic equation

$ax^2 + bx + c = 0$

using the expressions

$$\frac{-b \pm \surd(b^2 - 4ac)}{2a}$$

If your first definition of *roots* does the same calculation twice, such as $\surd(b^2 - 4ac)$, rewrite it avoiding this duplication.

2. What is printed by the following program?

```
vars a b c; 1 -> a; 2 -> b; 3 -> c;
function f a => c; vars b;
      a * a -> b; b + b -> c
end;
f (a + b + c) + f (a + b + c) =>
```

3. Assume that the function *apply1ton* takes two arguments, the first an integer *n* and the other a function of one argument, and applies the function to all integers from 1 to *n*. For example,

```
function prsqrt n; sqrt(n) => end;
apply1ton(4, prsqrt);
**1.000
**1.414
**1.732
**2.000
```

How would you use this function to tabulate the value of the expression $(1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3)$ in the range 0 to 3.0 at intervals of 0.1? (The function itself is to be defined as an exercise in section 7.)

6. CONDITIONALS

It often occurs within a function definition that the particular result depends upon whether some condition is true or not. A *conditional* enables one of two possible courses of action to take place according to a condition.

Consider the function definition

```
function max x y;
if x > y then x else y close
end
```

The value of the expression *max*(*a*, *b*) is *a* or *b*, whichever is greater. The operation > is an operation which produces a *truth value.* The two possible truth values are *false* and *true.* They are represented by the numbers 0 and 1 respectively, but for convenience the standard variables *false* and *true* always have these values.

Other operations which produce truth values are

>=	greater than or equal
=<	less than or equal
>	greater than
<	less than
=	equal to.

All relation operations have a precedence of 7. Thus comparison between arithmetic expressions will always take place after all arithmetic operations have been carried out. For example:

```
10 > 3 =>
**1
50 =< 20 =>
**0
true =>
**1
```

A conditional is always terminated with the syntax word **close**. Any imperative sequence (including conditionals) can appear between the **then** and the **else** and between the **else** and the **close**.

Quite complicated conditions can be tested using the syntax words **and** and **or** to join a series of conditions. For example,

if $x > 3$ **and** $y =< 2$ **then** $x + y$ **else** $x - y$ **close**

will leave $x + y$ on the stack if x is greater than 3 and y is less than or equal to 2. Otherwise the value of $x - y$ is left on the stack.

Note that **and** and **or** are not operations. They are syntax words associated with conditionals which simplify testing complicated conditions. They may only appear after **if** or **elseif** (see below) and before **then**. The conditionals joined by these syntax words are evaluated from left to right according to the following table (if there is no **else** then '**else** statement' means the statement after **close**).

basic word following cond	value of condition	what is evaluated next
and	false	**else** statement
and	true	next condition
or	false	next condition
or	true	**then** statement
then	false	**else** statement
then	true	**then** statement

The reason why **and** and **or** are not operations like * and +, and may not be used freely to form expressions, is that in the expression $x > 3$ or $y \uparrow 10 > 2$, for example, if x is greater than 3 there is no point in calculating $y \uparrow 10$ and comparing it with 2. Analogous functions are provided which do evaluate both arguments and may be used to form parenthesized expressions (see section 24 'Some useful standard functions'). The function *not* reverses a truth value so that we may write

if *not*(p) **or** *not*$(x = 3)$ **then** . . . **close;**

A conditional can be used to select one of two possible imperative sequences or one of two possible expressions. In the first case, the conditional behaves like a statement and is usually called a *conditional statement*. In the second case it behaves like an expression, and is usually called a *conditional expression*.

Consider the following conditional expression:
if $x = 1$ **then** *2* **else**
if $x = 2$ **then** *4* **else**
if $x = 3$ **then** *3* **else** *−1* **close close close**

Its value is 2, 4, 3, or −1 depending on the value of x. This structure occurs so often in programming that it is convenient to avoid having to write many **closes** at the end. This is achieved using the syntax word **elseif**, which behaves exactly like **else** followed by **if** except that no corresponding **close** is required. Using **elseif**, the above conditional expression may be rewritten:

if $x = 1$ **then** *2*
elseif $x = 2$ **then** *4*
elseif $x = 3$ **then** *3*
else *−1* **close**

which only has one **if** requiring a corresponding **close**.

Sometimes there is no action to be taken if the condition is false. In this case the else part may be omitted, for example,

if *x* > *0* **then** *x* => **close;**

EXERCISES

1. Rewrite the function *roots* defined in the exercise in section 5 to
(a) produce the complex roots if $b^2 - 4ac < 0$
(b) work correctly if $a = 0$.

2. Write a function to test whether a given number is less than 100 and divisible by 3, 4, or 5.

3. Tax is not levied on the first £150 of a man's income. It is levied at 10% on the next £250, at 25% on the next £200 and at 33% on the remainder. Write a function *tax* such that *tax* (*i*) is the tax levied on an income of £*i*.

4. Write a function which takes three parameters and puts them in ascending order of magnitude, for example,

```
f(1,-2,4) =>
** -2,1,4
```

7. LABELS AND GOTO STATEMENTS

In an imperative sequence, the statements are normally executed in the order in which they are written. The *goto statement* and its associated *label* enable statements to be executed in some other specified order. The destination of a goto statement is labelled with an identifier. The identifier (called a label) precedes the statement and is separated from it by a colon. After execution of a goto statement, the next statement to be executed is the one whose label is the destination of the goto statement.

Consider the following function definition:

```
function tab fun x step hi;
again: if x =< hi then fun(x) =>
x + step -> x; goto again close
end
```

This defines a function *tab* which tabulates the values of a function *fun* over a range from *x* to *hi* in steps of *step*. Executing the statement

```
tab(sqrt, 1, 1, 3)
```

causes the following results to be printed:

```
**1.0
**1.414
**1.732
```

Remember that if it is used in a function body => prints only the top item of the stack.

Goto statements and labels can only be used in a function definition, because labels are only appropriate if the labelled instruction is stored. Statements executed directly from the keyboard are not stored and cannot be labelled. Moreover a goto statement can only refer to a label in the same function definition as the goto statement itself. Clearly, to avoid ambiguity, no two statements may have the same label in a given function definition.

The goto statement which skips to the end of a function definition occurs so frequently that a special form is provided which requires no label. The statement

return

terminates execution of the function in which it occurs and returns to the program calling the function exactly as if the last statement had been executed. The function *tab* could have been defined using **return** as follows:

function *tab fun x step hi;*
again: **if** *x* > *hi* **then return close;**
fun(*x*) => *x* + *step* —> *x*; **goto** *again*
end;

We could have put a label, say *finish*, before **end** and put **goto** *finish* instead of **return.** It is just that **return** is a little neater.

Another syntax word **exit** is provided. The word **exit** is identical to the pair of syntax words **return** followed by **close.** This pair occurs together quite frequently, and can always be replaced by the single word **exit.**

As well as illustrating the goto statement, the definition of the function *tab* above has some other features worth commenting on. The formal parameter *fun* is used to denote a function. The particular function is specified when the function *tab* is called. This ability to use a variable whose value is a function is an important property of POP-2. It means that it is easy to define functions which operate on functions and, as we shall see later, produce functions as results.

A further point to note about the definition of *tab* is the use of a formal parameter *x* as a variable whose value is changed during execution of *tab*. As *x* is a local variable which has merely been given the *value* of the actual parameter (via the stack), changing *x* cannot change any variables in the calling program. This means that if we defined a function *increment* as

function *increment x*;*x* + *1* —> *x* **end**

and called it by executing the statement

increment(*y*)

this would have no effect on *y* because the parameter given to *increment* is simply the current value of *y*. The effect is simply to declare a local variable *x*, give it the value of *y*, add 1 to it, and then lose the value on exit from the function. To increment *y* we may define

function *increment;y* + *1* —> *y* **end**

and call it by *increment*( ).

Many loops, that is, sequences of instructions which may be executed repeatedly, start off with a conditional. For example, in computing *n* factorial

l: **if** *i* =< *n* **then** *i***p* —> *p*; *i*+*1*—>*i*; **goto** *l*
        close;

To save making up a label and putting a statement to go back to it, we may replace **if** by **loopif.**

loopif *i*=<*n* **then** *i***p*—>*p*; *i*+*1*—>*i*
close;

In general we can always replace **if** by **loopif**, even when **elseif** and **else** are being used. As soon as a condition succeeds we jump back to **if** (we think of **else** as being preceded by the condition *true*). Thus we have the equivalence

```
loopif ... then ...      loop:   if ... then ...; goto loop
elseif ... then ...          elseif ... then ...; goto loop
           else ...                     else ...; goto loop
close                        close
```

Here *loop* is any label which does not occur in any part of the same function definition. Of course this goes on for ever unless the dots contain some other **goto**, but so long as we leave out the **else** there is a way of stopping, since all the conditions may fail.

Counting on integers or real numbers is so common that a further abbreviation (in fact a standard 'macro', see section 22) is introduced to cope with the most common cases. Suppose *I* stands for any identifier and *M*, *K*, and *N* denote any identifiers or *unsigned* numbers. Then we may write

forall I M K N
to stand for
M—K—>I;
loopif (*I+K—>I;I=<N*) **then**

Thus *forall I M K N* means for all values of *I* from *M* up to *N* increasing in steps of *K*. The factorial loop above can be written simply as

```
forall i 1 1 n;
    i*p—>p
close;
```

If *K* is a real number we should take care over rounding errors.

forall x 0.0 0.1 1.01; ....
is safer than
forall x 0.0 0.1 1.00; ...

since 0.0 plus ten times 0.1 might come to, say, 1.0003 to within the computer's accuracy, and then the $x = 1.0$ value would not be done.

Remember that *forall* involves conditionals and **goto** so it can only be used inside a function body.

Straightforward loops can be done with *forall*. A more powerful and elaborate looping facility, the *FOR* facility, is provided in the Program Library (see Part 4).

We have not used **loopif** or **forall** in the answers to exercises of the Program Library since they were added during the revision of POP-2 and were not defined when this work was being done.

EXERCISES

1. The function *tab* used as an example in this section tabulates a function of one parameter over a specified range. Define a function *tab2* which tabulates a function of two parameters over specified ranges of the two parameters. Use *tab2* to print the products of all pairs of integers between 1 and 10. (For a way to get a proper tabular layout of the results see section 8 'Printing results'.)

2. The sequence

$$x_0 = 1$$
$$x_{k+1} = \tfrac{1}{2}\left(x_k + \frac{n}{x_k}\right)$$

gets closer and closer to the square root of N as k increases. Write a function *terms* such that *terms(n, epsilon)* is the value of k needed to compute the square root of n to within plus or minus epsilon. For example, if $n=9$ then $x_0=1, x_1=5, x_2=3.6, x_3=3.05$ and *terms*(9, 0.1) = 3.

3. Define the function *apply1ton* described in example 3 of section 5 'Function declarations'.

8. PRINTING RESULTS

So far we have used => to print results. This prints one or more results off the stack on a new line preceded by **.

Often we would like a different layout, and the following standard functions are provided (others are given in section 20).

pr(x) — this causes the value of x to be printed. Negative numbers are printed with a minus sign, positive ones preceded by a space.
prreal(r, m, n) — prints a real with m digits before the point and n digits after it.
nl(k) — this prints k new lines
sp(k) — this prints k spaces.

For example, to print the multiplication table:

```
function multab; vars i j p; 1 -> i;
    loopi: if i > 12 then exit;
           1 -> j; nl(1);
    loopj: if j > 12 then i + 1 -> i; goto loopi close;
           i * j -> p;
           sp(if p < 10 then 2 elseif p < 100 then 1 else 0 close);
           pr(p);
           j + 1 -> j; goto loopj
end;
```

```
multab():
    1  2  3   4   5   6   7   8   9  10  11  12

    2  4  6   8  10  12  14  16  18  20  22  24

    3  6  9  12  15  18  21  24  27  30  33  36

    etc.
```

Section 18 'Input and output facilities' gives further information about reading data and printing results, using the console or other input/output devices. Thus when we say above that something is printed we include the case where it is output to some other device, such as a disc file.

9. WORDS

So far, we have seen how a POP-2 variable can have a numerical value or a function value. Another important type of values is the *word*. A word, like an identifier, is made up of letters, digits, or signs. Only the first 8 characters are significant. Examples of words are

```
x    a5    pdp7    +    (
++    "    happy    birthday
```

The exact rules for constructing words are given in section 8.6 of the Reference Manual.

In order to assign a word to a variable, it is necessary to indicate that the word itself is meant, not the value of the variable designated by the word. For example, the assignment

```
cost —> x
```

assigns the *value* of the variable *cost* to the variable *x*. In order to assign the word *cost* it must be enclosed in quotes "and". In this case, the assignment would be

```
"cost" —> x
```

after which the value of *x* would be the word "*cost*" and the effect of printing *x* by typing

```
x =>
```

would be to print

```
**cost
```

If the variable *c* has value 100 then
pr("cost");pr("=");pr(c);
prints
cost= 100

We may test words for equality, for example, "*cat* " = "*cat*" has value *true.*

EXERCISES

1. What is printed by the following programs?

```
(a) function out n w;
          nl(2);pr(n);sp(1);pr(w);pr(", ");pr("please")
    end;
    out(20, "pounds");
    out(40, "dollars");
(b) function truthval p;
          if p then "true" else "false" close
    end;
    truthval(50>40) =>
```

2. The standard function *destword* takes as its parameter any word and leaves on the stack integer representations of the characters and the number of characters in the word, for example, *destword("CAT")* leaves i_c, i_a, i_t, 3 on the stack where i_c, i_a, and i_t are the integers corresponding to *c, a,* and *t*. Define a function *order* which takes two words and produces "*before*", "*same*", *or* "*after*", depending on whether the first letter of the first word is before, the same as, or after the first letter of the second word. Assume that the integer representations of the letters are in consecutive ascending order.

10. LISTS AND LIST PROCESSING

A *list* is simply an ordered sequence of items. A list of words or positive integers or positive reals can most easily be constructed by enclosing the words, integers, or reals in square brackets. For example, the assignment

```
[cat dog horse] —> x
```

makes *x* a list of three words. The value of *x* is the whole list and if *x* is printed by executing

```
x =>
```

the result

```
** [cat dog horse]
```

is printed.

Another example of a list formed in this way would be

```
[1 cat 2 dogs 3.1416 horses ***]
```

Note that although *cat* is used as a word, not a variable, there is no need to enclose it in quotes since the brackets serve the same purpose as quotes.

There is a standard operation, written <>, which joins two lists together. Thus the expression

```
x <> [donkey cow]
```

produces a list like *x* but with 2 new items on the end. If this is now printed, the result would be

```
** [cat dog horse donkey cow]
```

The value of the variable *x* is not changed.

An item on a list may itself be a list. The list
[[*cat dog*] [*horse donkey*]]
is a list of two items. The first item of the list is the list [*cat dog*]. The second item is the list [*horse donkey*].

There are two standard functions for accessing the items of a list. The function *hd* has as value the first item, or head, of the given list. Thus if *x* is the list [*a b c d*] the value of the expression

```
hd(x)
```

is the word "*a*". The function *tl* has as value the *tail* of the given list. The tail of a list is the list with the head removed. The value of the expression

```
tl(x)
```

is the list [*b c d*]. Thus the functions *hd* and *tl* can be used together to access any item on a list. The first item of the list *x* is *hd*(*x*), the second item is *hd*(*tl*(*x*)), the third item is *hd*(*tl*(*tl*(*x*))), and so on.

The tail of a list of one item is the word "*nil*", which represents the empty list. The standard variable *nil* has the word "*nil*" as its value. Any attempt to use the functions *hd* and *tl* on anything other than a list will result in an error. Thus an attempt to extract the third item from a list of two items by writing

```
[a b] -> x;
hd(tl (tl (x))) =>
```

results in an attempt to evaluate the expression

```
hd(nil)
```

which produces an error message because the word *nil* is not a list.

There is a standard function *null* whose value is *true* for an empty list and *false* otherwise. It is very frequently used to test for the end of a list. Consider, for example, a possible definition of a function *length1*, the length of a list.

```
function length1 x; vars n;
0 -> n;
l1: if null(x) then n exit;
n + 1 -> n; tl(x) -> x; goto l1
end;
```

One way of constructing a list is to use square brackets. A more fundamental function is *cons* (short for construct) also written as ::, an operation of precedence 2. The expression *cons*(*a*,*b*), or *a* :: *b*, constructs a list whose head is *a* and whose tail is *b*. Thus

```
"cat" :: nil -> x;
```

makes *x* a list of one item—the word *cat*. A list of several items could be constructed using *cons* as follows:

```
"a" :: ("b" :: ("c" :: nil)) -> x
```

which creates exactly the same list as

```
[a b c] -> x
```

The latter is a shorthand notation for the first.

The function *cons* (i.e., ::) puts an item in front of a list. Let us define a function to put an item at the end of a list. This is

```
function append x xl; xl <> (x::nil) end;
```

Here *x* is an item and *xl* a list (we will make a habit of using identifiers such as *xl* and *yl* for lists), *x*::*nil* is the list whose only element is *x*, and the operation <> joins the list *xl* to the list *x*::*nil*. Thus

```
append(4, [1 2 3]) =>
** [1 2 3 4]
```

The reader may find the distinction between :: and <> and between *x* and *x*::*nil* a little puzzling. He may find the following picture helpful. Items are coloured beads and a list is a string with beads on it. The empty list *nil* is a string with no beads on it. If *x* is a bead and *xl* is a string of beads then *x*::*xl* puts an extra bead on the front of the string. If *xl* is a string of beads and *yl* is another string then *xl* <> *yl* ties the end of the first string to the beginning of the second. Thus if *x* is a bead *xl* <> *x* would be nonsense, since you cannot tie a bead onto the end of a piece of string, but *xl* <> (*x*::*nil*) is all right.

As a matter of fact this explanation is not strictly accurate. What happens if we do the following?

```
    [2 3 4] -> x;
    1::x -> y;
    x =>
```

The answer should be [2 3 4] since it would be inconvenient if the second statement upset the result of doing the first. To ensure this *1*::*x* does not put the bead 1 onto the string called *x* but rather ties a fresh piece of string with the bead 1 on it in front of the string *x*, placing the beginning of this fresh piece in *y*. Similarly <> uses fresh string to copy its first argument so that the operation does not affect the value of other variables. The details will be described fully later on.

The following function tests whether an item occurs in a list.

```
function member x xl;
loop: if  null(xl) then false
          elseif hd(xl) = x then true
          else tl(xl) —> xl; goto loop
      close
end;
member(1, [2 1 5]) =>
**1
member("Joe", [Fred Alf Bert]) =>
**0
```

The following pair of functions manipulate 'association lists', for example,
[dog chien cat chat pig cochon]

```
function assoc x xyl => y; vars x1;
loop: if  null(xyl) then undef —> y
          else hd(xyl) —> x1; tl(xyl) —> xyl;
             if x1 = x then hd(xyl) —> y
                       else tl(xyl) —> xyl; goto loop
             close
      close
end;
```

Note: *undef* is a standard variable with value "*undef*" meaning undefined.

```
function makeassoc x y xyl => xyl1;
    x::(y::xyl) —> xyl1
end;
```

Now we can use these in the following way

```
        [dog chien cat chat pig cochon] —> dict;
        assoc  ("cat", dict) =>
        ** chat
        makeassoc ("hen", "poule", dict) —> dict;
        assoc ("hen", dict) =>
        ** poule
```

Suppose that we wish to obtain a new list, each member of which is derived from the corresponding member of some given list by applying a function to it. We could define a function *maplist* such that, for example, *maplist* ([*1 2 3 4*], *sqrt*) *is* [*1.00 1.41 1.73 2.00*]

```
function maplist xl f => yl; nil —> yl;
loop: if not(null(xl)) then append(f(hd(xl)), yl) —> yl;
                            tl(xl) —> xl; goto loop
                       close
end;
```

In fact *maplist* is a standard function.

EXERCISES

1. Given a function *p* which produces a truth value as its result, write a function *exists* such that *exists*(*xl, p*) is *true* just if *p* produces *true* for some element of *xl*.
2. Write a function *delete* such that *delete*(*x, xl*) is a list similar to *xl* but with any items equal to *x* on it deleted.

3. An association list *price* associates a price in pence with each of a number of articles. Write a function which will take a list of articles purchased and work out the total price (use *assoc*).

4. You are given a list, each of whose elements is an association list describing a known criminal thus

> [[*name jones hair sandy eyes brown height 65*]
> [*name crippen hair none eyes green height 61*]. . . .]

Write a function which takes a specification of a wanted man, for example,

> [*hair grey eyes brown height 60*],

and produces a list of the names of known criminals who might correspond to the description.

5. An association list is given which associates with each town a list of other towns which can be reached from it by a direct flight. Write a function to produce a list of all the towns which can be reached from a given one with not more than 1 change. Now write one for not more than *n* changes. (*Hint.* A function to remove repeated elements from a list would be useful, for example, *prune* ([1 2 3 2 5 3]) = [1 2 3 5].)

11. LAMBDA EXPRESSIONS

It was mentioned in the section on functions that variables can have functions as values, as well as the more obvious types of values such as numbers or truth values. A function definition is therefore a kind of assignment in which a function value is assigned to a variable. It is possible in POP-2 to write function constants just like we can write numerical constants (for example, 0, 3.15) or truth values (*true, false*). A function constant is called a *lambda expression* and is simply a way of defining a function and leaving the definition on the stack. The function *sumsq* defined earlier in the conventional way could have been defined as follows:

vars *sumsq*;
lambda *x y*; *x*↑*2* + *y*↑*2* **end** —> *sumsq*;

Thus the basic word **lambda** is very similar to the basic word **function** except that no function name is included. A lambda expression is an 'anonymous' function.

Lambda expressions are useful in a variety of circumstances. A frequently-occurring situation is illustrated by the following. We wish to use the *tab* function defined above to tabulate the values of *x*↑3 between 1 and 10 in steps of 0. 5.

We cannot write

tab(*x*↑*3*, *1*, *0. 5*, *10*);

because *tab* assumes the value of the first parameter is a function, whereas the result of evaluating the expression *x*↑3 is a number. Executing the above statement would therefore cause an error message.

We could, however, write

function *cube x*; *x*↑*3* **end;**
tab(*cube*, *1*, *0. 5*, *10*);

and this would work correctly but it is simpler to write

tab(**lambda** *x*; *x*↑*3* **end,** *1*, *0. 5*, *10*);

and the effect is identical except that no *cube* function remains after execution of the statement.

EXERCISES

1. How would you use *tab* to tabulate the values of the expression $x^2 - 2x - 1$ for integers from 0 to 100.

2. What is the value of *x* after execution of the following?

```
vars x k g;
lambda x; x * x end -> k;
lambda f; f(2) end -> g;
g(k) -> x;
```

12. RECURSION

A function may call itself during its execution. The POP-2 system automatically provides a distinct set of local variables when this happens.

Consider two possible ways of defining a function for computing the factorial of a number; first, an *iterative* definition using a goto statement

```
function fact n => p;
1 -> p;
loop: if n > 1 then p * n -> p; n - 1 -> n;
goto loop close
end
```

second, a *recursive* definition in which the function itself is called from within

```
function fact n;
if n > 1 then n * fact(n - 1) else 1 close
end
```

An obvious difference between these two definitions is that the second, recursive definition is much simpler to write. However, a more important difference is that execution of the recursively-defined function involves much more storage space. In fact, the whole arithmetic expression

$n * n{-}1 * n{-}2 * \ldots\ldots * 2 * 1$

is set up before it is evaluated, whereas in the first case, the result is accumulated factor by factor.

Where a choice exists between an iterative and a recursive definition, the former is usually preferable on grounds of efficiency. Often, however, the recursive definition will be briefer and more perspicuous, particularly in handling complex data structures.

As an example of the use of recursive functions in list processing let us define a function to produce the list of all items on a given list which are greater than 100. We use *x* for an item and *xl* (i.e., *x*-list) for a list.

```
function gr100 xl; vars x;
    if null (xl) then nil
        else hd (xl) -> x;
            if x > 100 then x :: gr100(tl(xl))
                        else gr100(tl(xl))
            close
    close
end;
gr100([90 101 85 106 107]) =>
** [101 106 107]
```

More generally if *p* is any property, that is, a function producing a truth value

```
function sublist xl p; vars x;
    if null(xl) then nil
       else hd(xl) -> x;
          if p(x) then x::sublist(tl(xl), p)
                 else sublist(tl(xl), p)
          close
   close
end;

function big x; x > 100
end;
sublist([90 101 85 106 107], big) =>
** [101 106 107]
```

Since we often want both the head and tail of a list the function *dest* is provided. It produces both the head and the tail. Thus we may write

```
function sublist xl p; vars x;
    if null(xl) then nil
       else dest(xl) -> xl -> x;
          if p(x) then x::sublist(xl, p)
                 else sublist(xl, p)
          close
    close
end;
```

Another example is a function to test whether an item occurs in a list.

```
function member x1 xl; vars x;
    if null(xl) then false
       else dest(xl) -> xl -> x;
          if x = x1 or member(x1, xl) then true else false close
       close
end;
```

EXERCISES

1. Write a recursive definition of the function *hcf* to determine the highest common factor of two integers. Also write an iterative definition of the same function. Which function is more efficient
(a) in terms of storage requirement
(b) in terms of running time?

2. Define the function *maplist* recursively (it was defined with a loop in the section on lists). You had better give it another name such as *maplist2* since *maplist* is standard.

3. What is the output of the following program?

```
function itlist xl y g;
    if null(xl) then y
       else g(hd(xl), itlist(tl(xl), y, g))
    close
end;
function add x y; x+y end;
itlist([1 2 3 4], 0, add) =>
itlist([1 2 3 4], nil, append) =>
```

4. Write recursive functions for exercises 1 and 2 of section 10 (p.23).

13. DEFINING NEW OPERATIONS

Having defined a function such as *sumsq* with the function definition

```
function sumsq x y;
x↑2 + y↑2
end
```

we can evaluate expressions involving the function such as

```
3 + sumsq(sumsq(4, 5 *2),3) =>
```

It is often convenient, however, to use an operation rather than an ordinary identifier to denote a function. This is standard practice in the case of arithmetic operations where it is much simpler to write

```
a + b + c + d
```

than to write

```
add(add(add(a, b), c), d)
```

where *add* is a function for adding a pair of integers. The only difference between an ordinary identifier denoting a function and an operation is that the latter has a *precedence* which can affect the order of evaluation of the expression, and hence it may be written between its agreements without any parentheses.

We can declare new operations called, say, ++ with precedence 5 and ** with precedence 3 by executing the declaration

```
vars operation 5 ++ operation 3 **;
```

and write 6 ++ 8 ** 10, meaning 6 ++ (8 ** 10). It is usual, though not necessary, to use identifiers made up of signs rather than letters and digits when naming operations. This convention helps the (human) reader parse an expression.

Having declared an operation, an assignment is used to assign a function value to it. It is not possible to write

```
sumsq —> ++
```

because we do not wish to perform the operation ++, only assign a value to it. To make an operation behave like an ordinary identifier, we place the word **nonop** before it. Thus the assignment

```
sumsq —> nonop ++
```

makes ++ into an operation for adding the square of the two expressions surrounding it. Alternatively we could simply write **operation** 5 ++ instead of **function sumsq** in the function definition.

Another use for **nonop** is when we wish to pass a function denoted by an operation variable to another function as a parameter. For example, if — — denotes a function of one argument, we could write

```
tab(nonop — —, 1, 1, 100)
```

but not

```
tab(— —, 1, 1, 100)
```

The latter would apply — — once before applying *tab* instead of applying it 100 times inside *tab*;

By associating a precedence with an identifier we can dispense with some parentheses in expressions containing that identifier. Another way of avoiding parentheses is to use the dot notation. Instead of writing *f*(*x*) we write *x*.*f*, instead of *sin*(*cos*(*x*)) + *cos*(*sin*(*x*)) we write *x*. *cos*. *sin* + *x*. *sin*. *cos*. This is allowed when the argument of the function is denoted by an identifier or a constant, or is itself a dot expression.

EXERCISE

POP-2 does not have a standard *not equals* operation whose value is *true* if the arguments are not equal and *false* otherwise. Define a suitable operation written /= with the same precedence as the = operation, i.e. 7.

14. MORE ABOUT LISTS

LISTS WHOSE ELEMENTS ARE LISTS

Lists may have other lists as their elements, for example,

[[1 2 3] 2 [1[2 3] 4]]

This has 3 elements, the first a list, the second a number, and the third a list of 3 elements, one of them itself a list. The following function will count how many numbers there are in such a list of lists, 8 in the one above.

```
function lengthll l;
     if null(l) then 0
         elseif islist(hd(l)) then lengthll(hd(l)) + lengthll(tl(l))
         else 1 + lengthll(tl(l))
end;
```

Note that the function *islist* recognizes lists.

Consider the following function to read a list from the keyboard. The standard function *itemread* reads one word or positive number from the keyboard, and *append* (*y*,*x*) appends the item *y* to the end of the list *x*.

```
function listread;
     vars x y;
     itemread() -> x;
     if x = "[" then nil -> y;
                   loop: listread() -> x;
                          if x = "]" then y
                                       else append(x,y) -> y
                                       goto loop
                          close
                 else x
     close
end;
```

Note that a recursive definition is necessary here in order to allow lists to contain lists to any complexity.

Thus *listread*() -> *x*;
[*1* [*2 3*]]
has the same effect as
[*1* [*2 3*]] -> *x*;

DECORATED LIST BRACKETS

The list brackets described above provide a convenient notation for writing list structures consisting entirely of words or positive numbers.

Alternative *decorated brackets* [% and %] are provided for use when the individual elements of the list structure are obtained by evaluating expressions. Within decorated brackets, the expressions are separated from each other by commas. For example, the list

[%*x*, *3* + *4*, "*x*"%]

is a list of three items: the value of the variable *x*, 7, and the word "*x*". Decorated brackets can be used to create list structures of any complexity. They must be used to create lists with negative numbers since negative numbers are expressions. For example,

[% *–3.5*, *7.0*, *–2.6* %]

has a value the list of three numbers –3.5, 7.0, and –2.6.

UPDATING LISTS

Given a list *x*, say [*a b c*], it is possible in POP-2 to use an assignment to change part of the list. By executing the assignment

"*d*" –> *hd*(*x*)

the list *x* becomes [*d b c*].
The function *tl* can also appear on the right-hand side of an assignment. The statement

tl(*tl*(*x*)) –> *tl*(*x*)

results in the middle item of the list being deleted and *x* becomes [*d c*]. Also, if *x* is [*a b c*], the statement

d –> *hd*(*tl*(*x*))

gives *x* the value [*a d c*].

Functions like *hd* and *tl* which can be used on either side of an assignment are called *doublets.* They actually consist of two functions, one of which is chosen for use depending upon which side of the assignment the function is called. There is a difference between the two functions of a doublet because on the left-hand side of an assignment the function must produce a value, but on the right-hand side a value must be used to change some structure. These two component functions of a doublet are called the 'selector' and the 'updater' respectively. Thus the function *sqrt* is not a doublet because there is no reasonable interpretation of the assignment

3 –> *sqrt*(*2*);

The POP-2 user may define doublets. Consider, for example, a function *element* to get the *n*th element from a list *x*. The definition of *element* is

function *element n x;*
if *n* = *1* **then** *hd*(*x*) **else**
element(*n–1*, *tl*(*x*)) **close**
end;

Thus if *y* is the list [*a b c*] then *element* (*2*, *y*) is *b*. Because *element* has not been defined as a doublet, we cannot write

"*z*" –> *element* (*2*, *y*);

even though there is a very reasonable interpretation of such an assignment. That is, to replace the second element of the list *y* with the word *z*. In order to make *element* into a doublet with this meaning when used on the right-hand side of an assignment, an auxiliary function, say *changeelement* must be defined. A suitable function is

function *changeelement a n x;*
if *n = 1* **then** *a —> hd(x)* **else**
changeelement(a, n—1, tl(x)) **close**
end;

Notice that *changeelement* has an extra formal parameter *a* before the other two formal parameters *n* and *x* which were used in the definition of *element.* The extra formal parameter represents the value to be assigned. Having defined *changeelement,* it can itself be used to update a list. The statement

changeelement("z", 2, x)

replaces the second item of the list *x* with the word *z*; the effect required of *element* on the right-hand side of an assignment. To make *element* into a doublet, the assignment

changeelement —> updater (element)

is executed using a standard function *updater.* This assignment puts the function *changeelement* in the place that *element* goes to when called on the right-hand side of an assignment. Now we can execute the statement

"z" —> element (2, x)

Oddly enough, the standard function *updater* is itself a doublet. It acts on functions and can be used to get at their update part. Thus *updater (element) = changeelement* would now be *true.*

An important point to note about doublets, is that the updater function of a doublet is called only if the function is the main function on the right-hand side of an assignment. Thus in the assignment

hd(tl(x)) —> hd(tl(y))

which replaces the second item of list *y* with the second item of list *x*, only the function *hd* on the right-hand side uses its updater function rather than its selector. The function *tl* on the right-hand side is used in its normal sense.

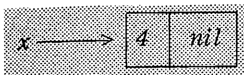
STATIC AND DYNAMIC LISTS

No mention has so far been made of the structure of lists as they appear in the memory of the machine. Lists have two representations, *static* and *dynamic.* The functions which operate on lists described so far work equally well with either static or dynamic lists, or even combinations of the two types.

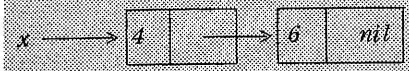
The list brackets (both plain and decorated) and the *cons* operator :: all generate static lists.

An element of a static list is called a *pair.* A pair contains two values. One is the head and the other the tail. The pair is normally represented in the computer by two adjacent memory cells and to designate a specific pair it suffices to pass the 'address' or serial number in memory of the first of these cells. Thus if we write *4::nil—> x* two adjacent memory cells are reserved and *4* and *nil* are placed in them. The address of the first of these cells is placed on the stack and then removed and placed as the value of *x*.

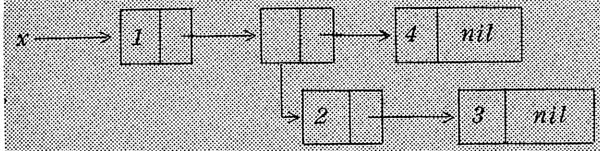
We may represent the situation thus, using an arrow to show that *x* contains the address of a pair.



If a list *x* has 2 items, say, [4 6], 2 pairs are needed to represent it, thus

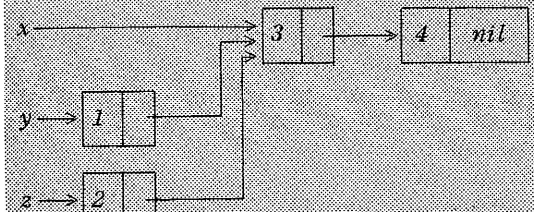


The second cell of the first pair contains the address of the second pair. Similarly [1 [2 3] 4] —> *x* gives rise to

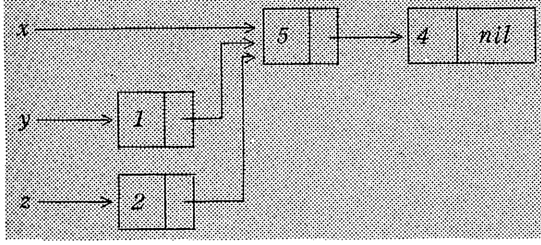


Two lists can *share* a common tail thus

```
[3 4] —> x;
1 ::x —> y;
2 ::x —> z;
```



```
5 —> hd(x);
```



We see that an assignment which updates a component of a list which is the value of one variable can alter the value of another variable.

Consider now the following piece of program

```
x =>
**[1 2 3]
y =>
**[1 2 3]
4 —> hd(y);
x =>
```

What will be printed, [1 2 3] or [4 2 3]? This depends on whether *x* and *y* share the same list [1 2 3], or have as their values distinct lists which happen to have the same elements, or indeed some intermediate situation such as sharing only the last two elements. We could find out by looking back over the preceding program, or more directly we can print *x*=*y*. This will be true just if *x* and *y* share the same list, not just copies with the same elements. Similarly *tl*(*x*) = *tl*(*y*) tests whether they share their last two elements.

How can we test the weaker proposition that the lists *x* and *y* have the same elements?

```
function equallist x y;
    if x = y or null(x) and null(y) then true
       elseif null(x) or null(y) then false
       elseif hd(x) = hd(y) and equallist(tl(x), tl(y)) then true
       else false
    close
end;
```

Thus [1 2 3] = [1 2 3] is *false* but *equallist* ([1 2 3], [1 2 3]) is *true*.

Equallist as defined assumes that the elements of the given lists are to be tested for strict equality, not merely list equality, and *equallist* ([1[2 3]], [1[2 3]]) is false. To make *equallist* test for element-wise equality throughout put **if** *atom*(*x*) **or** *atom*(*y*) **then** *x* = *y* **exit;** before **if** and to replace *hd*(*x*)=*hd*(*y*) by *equallist*(*hd*(*x*), *hd*(*y*)).

We will see later that the *pair* used to build static lists is just a special standard kind of *record.*

If the elements of a list are arbitrary and have no relation to each other, the static representation is very suitable. If, however, each element of a list is related to its predecessor by a well-defined rule, the list can be represented by this rule rather than by the actual elements. Thus the rule 'add 1' could be used to represent the infinite list 0, 1, 2. . . . . A dynamic list is a list represented by a rule in the form of a POP-2 function. The function must be a function of zero parameters and must always yield exactly one result. The function must be so written that each successive call generates the successive elements of the list. There is a built-in function called *fntolist* which converts such a function into a dynamic list.

Consider how the dynamic list, 0, 1, 2, and so on, might be constructed. The following will achieve it.

```
vars n;-1 -> n;
function suc;n + 1 -> n;n end;
fntolist(suc) -> y;
```

The above POP-2 text produces the dynamic list *y*, which behaves just like any static list except that very little storage space is required. For example, the function *element* defined above to extract the *n*th item from a list will work just as well with a dynamic list as with a static list. Thus

```
element(y,20) =>
```

produces the output

```
**19
```

because the twentieth item on the list *y* is the integer 19.

An important use of dynamic lists is to represent a stream of items read from an input device. For example, the function *itemread* described earlier, which reads one item from the keyboard, can be turned into a dynamic list,

```
fntolist(itemread) -> x
```

enabling any program that processes a list of items to work on items typed directly on the keyboard.

Dynamic lists provide a variant of the facility called a *stream* devised by Landin (1965).

EXERCISES

1. The function *makeassoc* was previously defined as

```
function makeassoc x y xyl => xyl1
    x :: (y ::xyl) -> xyl1
end;
```

Rewrite it so that if *x* is already on the list *xyl* it changes the associated value to *y* producing the altered list as a result.

2. What is the output of the following program?

```
[1 2] -> x;
x -> x.tl.tl;
x.hd,x.tl.hd,x.tl.tl.hd,x.tl.tl.tl.hd =>
```

3. Define a function *edit* which has as parameters three lists. The function should look for the second list within the first list and replace it with the third list. For example,

```
[jim is a son of a bitch and so is bob] -> xl;
edit (xl, [son of a bitch], [* * * *]) -> xl;
xl =>
** [jim is a * * * * and so is bob]
```

4. Write a function to produce as a dynamic list the prime numbers from 1 to *n*.

15. RECORDS

The pair described in the previous section is a special case of a *record*. A pair consists of two components called the *front* and the *back*. When a pair is used as an element in a static list, the functions *hd* and *tl* refer to the front and back of the pair respectively. A pair is created by the function *conspair*, which finds an area of memory and places two values in the front and back of the new pair. The function *cons* used in list processing is the same as *conspair*. *Conspair* is called the *constructor* for pair records. Just as a constructor takes the components of a record and produces a record containing the components, there is a complementary function, called a *destructor*, which takes a record and yields the components of the record as results. In the case of a pair, the destructor function is *destpair*, which takes a pair and produces the front and back components of the pair as results. Note, however, that in spite of its name, a destructor does not actually destroy the record; it merely extracts its components.

The pair can be used by the POP-2 programmer in many ways. It is not restricted to its use in list processing. A pair could be used to represent a complex number and functions and operators defined for handling pairs representing complex numbers. Thus 1::2 represents the number $1 + 2i$. The following might serve as a basis for complex number manipulation.

```
operation 6 +++ x y;
conspair(front(x) + front(y), back(x) + back(y))
end;
```

```
operation 6 --- x y;
conspair(front(x) - front(y), back(x) - back(y))
end;
operation 5 *** x y;
conspair(front(x) * front(y) - back(x) * back(y)
front(x) * back(y) + back(x) * front(y))
end;
operation 5 /// x y;
vars z; sqrt(front(y)↑2 + back(y)↑2) -> z;
conspair((front(x) * front(y) + back(x) * back(y))/z,
         (back(x) * front(y) - front(x) * back(y))/z)
end;
(-1)::2 -> u;
1::2 +++ 3::(-3) *** u -> v;
v.front, v.back =>
** 4,11
```

The record therefore enables a collection of quantities to be known by one name. This is useful not only for complex numbers but for a wide variety of situations, such as constructing list processing functions for lists with both forward and backward pointers or storing several items of information about an individual employee. The pair record cannot be used of course if more than two items of information are associated with the particular object.

POP-2 provides facilities for defining new records and functions for dealing with them. If the pair was not already defined it could be defined using the *recordfns* function

```
recordfns ("pair",[0 0]) -> back -> front -> destpair -> conspair;
```

The standard function *recordfns* takes two parameters: the name to be associated with the type of record being defined, and a list of integers. The number of integers in the list indicates the number of components the records are to have—in this case two. In this list, the integer zero indicates space for storing a POP-2 value just like any variable. Any value other than zero indicates the number of bits required to store the particular component. This enables more than one component to be stored in a single machine word. Instead of an integer we may have "*COMPND*", meaning the component must be a compound item, i.e., not a real or integer.

It is important to note that *recordfns* does not produce records; it produces functions for handling a new class of records. In the case of the pair, *recordfns* places on the stack the constructor function *conspair*, the destructor function *destpair*, the doublets for accessing the two components *front* and *back* (called select/update doublets because they allow us either to select out part of a record or to update that part of the record, giving it a new value). In order to use these functions they must be taken off the stack and assigned to variables. Note that with functions producing more than one result, the order of assignment is the reverse of the order in which results are placed on the stack.

Consider the problem of handling information about a collection of persons. Each person can be represented by a record with three components indicating the person's name, age, and sex. The name can be represented by a word, the age by an integer less than 128, and the sex by the integers 0 or 1. We can therefore set up functions for handling such records as follows:

```
recordfns ("person", [0 7 1]) -> male
-> age -> name -> destper -> consper;
```

This defines and names five new functions, and the following shows how they might be used. First *consper* can be used to construct a few records

```
consper("smith", 31, 1) -> p1;
consper("jones", 21, 0) -> p2;
consper("robinson", 93, 1) -> p3;
```

*p*1, *p*2, and *p*3 are now records of type *person*, and can be interrogated or updated by the doublets *name*, *age*, and *sex*.

```
name(p2) =>
**jones
function birthday p;
age (p) + 1 -> age (p)
end;
birthday(p3);
age(p3) =>
**94
function marry boy girl;
if male(boy) and not(male(girl)) then
name(boy) -> name(girl) else pr("shame") close
end;
marry(p1, p3);
shame
marry(p1, p2);
name(p2) =>
**smith
```

Sometimes we wish to test a record to see to which class it belongs. The standard function *dataword* produces the word associated with the class.

Thus

```
dataword(p1) =>
**person
```

The record facility of POP-2 permits the user to define new compound objects out of existing objects, thus extending the language to handle quantities associated with a class of problems. The objects could be represented instead by list structures, but the appropriate record structures usually take less storage space and the use of specially-named functions (select-update doublets) to access the components makes programming easier and clearer.

EXERCISES

1. A point can be represented by two real numbers. A triangle can be represented by three points. Define classes of records to represent points and triangles. Define a function *equilateral* which tests if a given triangle has sides of equal length.

2. A flight has a number, a starting place, a finishing place, a starting time, and a finishing time. Given a list of flights, write a function to get to a given place by a given time starting from a given place at a given time.