# PART 3. POP-2 REFERENCE MANUAL

R. M. BURSTALL AND R. J. POPPLESTONE

## 1. INTRODUCTION

### 1.1 AIMS

The following are the main design objectives for the POP-2 language:
(a) The language should allow convenient manipulation of a variety of data structures and give powerful facilities for defining new functions over them.
(b) The language should be suitable for taking advantage of on-line use at a console, that is, it should allow immediate execution of statements and should have a sufficiently simple syntax to avoid frequent typing errors.
(c) A compiler and operating system should be easy to write and should not occupy much storage.
(d) The elementary features of the language should be easy to learn and use.
(e) The language should be sufficiently self-consistent and economical in structure to allow it to incorporate new facilities when extensions are desired.

In attaining these objectives certain other desirable features of programming languages had to be relegated to secondary importance:
(f) Fast arithmetical facilities on integer and real numbers.
(g) Fast subscripting of arrays.
(h) A wide variety of elegant syntactic forms.

Naturally whether (c), or (f), or (g) are attained is to a considerable extent a matter of implementation.

### 1.2 MAIN FEATURES

The following main features are provided. Roughly analogous features of some other programming languages are mentioned in brackets as a guide:
(a) Variables (cf. ALGOL but no types at compile time)
(b) Constants (cf. ALGOL numeric and string constants, LISP atoms and list constants)
(c) Expressions and statements (cf. ALGOL)
(d) Assignment (cf. ALGOL, also CPL left-hand functions)
(e) Conditionals, jumps, and labels (cf. ALGOL)
(f) Functions (cf. ALGOL procedures but no call by name, cf. CPL and ISWIM for full manipulation of functions)
(g) Arrays (cf. ALGOL; also CPL for full manipulation of arrays)
(h) Records (cf. COBOL, PL/1, Wirth-Hoare ALGOL records, CPL nodes)
(i) Words (cf. LISP atoms)
(j) Lists (cf. LISP, IPL-V)
(k) Macros
(l) Use of compiler during running (cf. LISP, TRAC)
(m) Immediate execution (cf. JOSS, TRAC).

Notes:
LISP: See McCarthy (1960)
CPL: See Barron et al (1963) and Strachey (1967)
TRAC: See Mooers (1966)

ISWIM: See Landin (1966)
JOSS: See Marks (1967)
Wirth-Hoare ALGOL: See Wirth and Hoare (1966)

## 1.3 EXAMPLES

The following is an example of POP-2 program text. The sign => (not to be confused with that used in section 1.5 'Notation for functions') prints out some results on a newline prefixed with two asterisks. These results are included in the text below, as they would appear if the program were run on-line at a console.

```
comment arithmetic;
12.0+2.5*(1.5+2.5)=>
**22.0
vars a b sum;
2*2->a; 3*a->b; a*a+b*b->sum; sum=>
**160
function sumsq x y;
    x*x+y*y
end;
sumsq(a, b)+1=>
**161
function fact n; vars p;
    1->p;
loop: if n=0 then p else n*p->p; n-1->n; goto loop close
end;
fact(fact(3))=>
**720

comment arrays;
vars a i j;
10->i; 20->j;
newarray([% 1, i, 1, j %], sumsq)->a;
a(2, 3)=>
**13
10->a(2, 3); a(2, 3)=>
**10
function arraysum a1 a2 m n;
    newarray([% 1, m, 1, n %], lambda i j; a1(i, j)+a2(i, j) end)
end;
arraysum(a, a, 10, 20)->a; a(2, 3)=>
**20

comment lists;
vars u;
1->i; 2->j;
[% i, i+j, "dog", "cat" %]->u; u=>
**[1 3 dog cat]
cons ("pig", u)=>
**[pig 1 3 dog cat]
function append x y;
    if null(y) then [% x %] else cons(hd(y), append(x, tl(y))) close
end;
append(4, [% 1, i+1, 3 %])=>
**[1 2 3 4]
```

```
comment records;
vars consper destper forename surname male p1 p2;
recordfns('person",[0 0 1])->male->surname->forename
                              ->destper->consper;
consper("jane", "jones",false)->p1;consper("sam", "smith",true)->p2;
surname(p1)=>
**jones
datalist(p1)=>
**[jane jones 0]
function marry x y;
    if male(x) and not(male(y)) then surname(x)->surname(y) close
end;
marry(p2,p1);datalist(p1)=>
**[jane smith 0]
```

## 1.4 NOTATION FOR SYNTACTIC DESCRIPTION

We use the BNF (Backus-Naur Form) notation as used in the ALGOL report:
::= indicates a syntax definition;
⟨⟩ are used to enclose the name of a syntax class;
| denotes disjunction (union of syntax classes).

Concatenation denotes concatenation of any elements of two syntax classes.

We also use a convenient extension of this notation due to R. A. Brooker:

* means that a class may occur $n$ times, $n \geq 1$;
? means that a class may occur $n$ times, $n=0$ or $1$;
*? means that a class may occur $n$ times, $n \geq 0$.

For example, the definitions
⟨*astring*⟩ ::= ⟨*a*⟩ ⟨*astring*⟩ | ⟨*a*⟩
⟨*bstring*⟩ ::= ⟨*b*⟩ ⟨*astring*⟩
⟨*cstring*⟩ ::= ⟨*c*⟩ ⟨*astring*⟩ | ⟨*c*⟩

may be replaced by
⟨*bstring*⟩ ::= ⟨*b*⟩ ⟨*a**⟩
⟨*cstring*⟩ ::= ⟨*c*⟩ ⟨*a* ?*⟩

The characters ⟨, ⟩ and * are used in the POP-2 reference language but no confusion should arise.

When we wish to give examples of a syntax class we use the symbol 'e.g.::=', for example,

⟨*bstring*⟩ e.g.::= ⟨*b*⟩ ⟨*a*⟩ | ⟨*b*⟩ ⟨*a*⟩ ⟨*a*⟩ ⟨*a*⟩

The character set of the POP-2 reference language is as follows.
⟨*letter*⟩ ::= *a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|*n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*
⟨*digit*⟩ ::= *0*|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*
⟨*sign*⟩ ::= +|—|*|/|$|&|=|<|>|:|£|↑
⟨*separator*⟩ ::= , |;
⟨*period*⟩ ::= .
⟨*sub ten*⟩ ::= $_{10}$
⟨*bracket*⟩ ::= (|)|[|]
⟨*bracket decorator*⟩ ::= %
⟨*quote*⟩::= "
⟨*string quote*⟩ ::= '|`

Letters may be written in lower case, upper case, or bold type without any change of meaning. It will be conventional however to use bold

type letters for syntax words, that is, those identifiers, such as **function, then, end,** and **cancel,** which have a special meaning for the POP-2 compiler and which characterize certain syntactic forms.

Spaces, tabulate and new lines terminate identifiers, integers, reals, and words but otherwise they are ignored except in string constants.

A distinction is made between the reference language used in this document and a number of possible hardware languages used by particular computer implementations of POP-2. Each character in the reference language should be represented by a distinct character or sequence of characters in the hardware language. A particular letter, whether upper case, lower case, in bold type, or not, is regarded as the same character in the reference language.

The word 'list' is sometimes used in a syntactic sense with its ordinary English meaning, for example, in 'formal parameter list' or 'external list'. At other times it is used in a technical sense to refer to a kind of data structure (see section 8.3 'Lists'). The distinction will be clear from the context.

## 1.5 NOTATION FOR FUNCTIONS

It is convenient to have a notation to specify the domain and range of functions. We will consider functions having several arguments (or possibly none) and producing several results (or possibly none), the notion of functions with more than one result being an extension of normal mathematical usage (see section 4.2 'Application of functions'). We introduce a special symbol '=>' which is not to be confused with any identifier in the POP-2 language.

Suppose $d_1, d_2, \ldots, d_m$ and $r_1, r_2, \ldots, r_n$ are all sets of items. Then $d_1, d_2, \ldots, d_m => r_1, r_2, \ldots, r_n$ is the set of all functions whose domain is $d_1, d_2, \ldots, d_m$ and range $r_1, r_2, \ldots, r_n$, that is, with arguments which are $m$-tuples in $d_1 \times d_2 \times \ldots \times d_m$ and with results which are $n$-tuples in $r_1 \times r_2 \ldots \times r_n$. We express the fact that a function $f$ is a member of this set of functions by

$f \in d_1, d_2, \ldots, d_m => r_1, r_2, \ldots, r_n$

Some examples will make this clear.

*add ∈ integer, integer => integer*

(The usual mathematical convention is *add: INTEGERS×INTEGERS–> INTEGERS.)*

*divrem ∈ integer, integer => integer, integer*

where *divrem* is 'divide with remainder', for example, *divrem(7, 3)=2, 1* and *divrem(14, 4)=3, 2*

*roundup ∈ real => integer*
*prime ∈ integer => truthvalue*

If the function has no results we use an empty pair of parentheses, thus:
*printout ∈ integer => ()*

The arguments or results may themselves be functions
*differentiate ∈ (real=>real) => (real=>real)*

Where we wish to discuss a number of functions all having the same domain and range it is convenient to abbreviate thus:

*f, g, ..., h all ∈ ... => ...*

for $f \in \ldots \Rightarrow \ldots$
and $g \in \ldots \Rightarrow \ldots$
. . . . . .
and $h \in \ldots \Rightarrow \ldots$

Some functions do not have a fixed number of arguments and some do not have a fixed number of results (see section 4.2, 'Application of functions'). In such cases we may write, for example,

*f ∈ integer => real, integer, . . . , integer*

for the domain or range, meaning that a real and a variable number of integers are the results.

## 2. ITEMS

### 2.1 SIMPLE AND COMPOUND ITEMS

The objects on which one can operate and which one can obtain as results are called *Items*.* They are divided into two distinct classes: *Simple* items and *Compound* items.

Two kinds of simple item are distinguished: integers and reals. The following standard functions recognize them:

*isinteger, isreal all ∈ item => truthvalue*

Truthvalues (true and false) are represented by integers, and bitstrings (patterns of 0s and 1s susceptible to logical operations) are represented by positive integers, and although special functions are provided for operating on truthvalues and bitstrings, integers used in this way are in no way distinguishable from other integers.

All other items are compound, so called because they have components which can be selected and updated. They consist of records, strips, and functions, and include the particular kinds of records known as references, pairs, lists, and words, and the particular kinds of functions known as closure functions and arrays.

Although it is a matter of implementation and hence outside the scope of this manual, it may help the reader to think of a simple item as being represented by a pattern of information which does not contain the representation of any other item, and a compound item as being represented by an address pointing to an area of store which may contain the representations of other items (their addresses). In these terms compound items are equal just if they are represented by the same address; otherwise their addresses point to distinct non-overlapping areas of store, although of course these areas of store may contain copies of the same addresses or bitstrings. The formal properties of simple and compound objects, especially their behaviour with respect to equality, assignment, and updating, are motivated by this point of view.

The following standard function recognizes compound items:
*iscompnd ∈ item => truthvalue*

The standard function = (an operation of precedence 7) is used to represent equality of items. For integers and reals it has the usual

---

* Each new term is introduced in italic type with an initial capital letter.

meaning. Its meaning for compound items is given in section 7.1 'Functions of data structures'.

*= ∈ item, item => truthvalue*

## 2.2 INTEGERS

Integers are simple items. They may be positive, negative, or zero. The size of the largest and smallest integers allowed depends on the implementation.

The syntax of integers is:
⟨*integer*⟩ ::= ⟨*octal integer*⟩ | ⟨*binary integer*⟩ | ⟨*decimal integer*⟩
⟨*octal integer*⟩ ::= *8:*⟨*octal digit**⟩
⟨*binary integer*⟩ ::= *2:*⟨*binary digit**⟩
⟨*decimal integer*⟩ ::= ⟨*digit**⟩
⟨*octal digit*⟩ ::= *0|1|2|3|4|5|6|7*
⟨*binary digit*⟩::= *0|1*

Example:
⟨*integer*⟩*e.g.* ::= *8:777|2:101|0|6559*

This syntax does not allow negative integers. These are obtained using the operation — (minus or negate) defined in section 4.6 'Arithmetic operations'. Thus one may write —1 but syntactically it is an expression (see section 5.1 'Expressions').

For standard functions to manipulate integers, see section 4.6 'Arithmetic operations'.

Positive integers may also be treated as *Bitstrings* (the length depending on the implementation) and the following functions are standard:

*logand, logor, logshift all ∈ integer, integer => integer*
*lognot ∈ integer => integer*

*logand* and *logor* are the usual bit by bit 'and' and 'inclusive or'; *logshift* causes the first integer to be shifted left by the number of places given in the second, unless the second integer is negative when shifting to the right takes place (all new bits to fill up the end are zero in each case). The correspondence between binary and decimal is the natural one, for example, *2:101=5*. The binary equivalent of a negative integer is not defined; implementations may or may not provide one.

## 2.3 REALS

Reals are simple items. They may be positive, negative, or zero. The size of the largest and smallest reals allowed and the precision depends on the implementation.

The syntax of reals is as follows:

⟨*real*⟩*::=* ⟨*decimal integer?*⟩*.* ⟨*decimal integer*⟩ ⟨*exponent?*⟩
⟨*exponent*⟩*::=*$_{10}$*+*⟨*integer*⟩ |$_{10}$*—*⟨*integer*⟩ |$_{10}$⟨*integer*⟩

Example:

⟨*real*⟩*e.g.* ::= *.5|1.99|1.5*$_{10}$*-6*

For standard functions to manipulate reals (including an operation to produce negative reals) see section 4.6 'Arithmetic operations'.

## 2.4 TRUTH VALUES

The two items *True*, which is the integer 1, and *False*, which is the integer 0, are called *Truthvalues*.

On entry to the POP-2 system the standard variable *true* is set to 1 and the standard variable *false* is set to 0. The following standard functions on truthvalues are provided:

*booland, boolor all* ∈ *truthvalue, truthvalue => truthvalue*
*not* ∈ *truthvalue => truthvalue*

These are the usual functions 'and', 'inclusive or' and 'not' of propositional calculus.

## 2.5 UNDEFINED

The standard variable *undef* has the word *:"undef"* as its value on entry to the POP-2 system (see section 8.6 'Words'). The programmer may use it as the result of a function which fails to produce its normal result.

## 2.6 TERMINATOR

The standard variable *termin* has a certain item as its value on entry to the POP-2 system. This item is a special one which cannot be created by the programmer and is treated specially by the printing function *pr*. It may be used as the first argument of a variadic function (see section 4.2 'Application of functions') and is used to mark the end of an input file (see section 9.1 'Input') or an output file (see section 9.2 'Output') and as the last item produced by a repeater (see section 8.3 'Lists').

# 3. VARIABLES

## 3.1 IDENTIFIERS

An item may be the *Value* of a *Variable* (a variable is not itself an item). An *Identifier* is associated with the variable and this identifier is used to refer to it in a POP-2 program. A number of distinct variables may have the same identifier, but only one of them is *Currently associated* with it at a particular time in the evaluation process.

An identifier may be given special syntactic *Properties* by being made an *Operation* with a given precedence (see section 5.2 'Precedence') or by being made into a *Macro* (see section 11.2, 'Macros'). These properties cannot be changed without cancelling the identifier (see section 3.3 'Cancellation') or entering or leaving a section (see section 3.4 'Sections').

The syntax of identifiers is:

⟨*identifier*⟩ ::= ⟨*letter*⟩⟨*alphanumeric* * ?⟩ | ⟨*sign* *⟩
⟨*alphanumeric*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩

Example:

⟨*identifier*⟩ *e.g.* ::= *x* | *y99* | *alpha* | *u2a* | *+++* | */+* | *<>* | **$$**

Syntax words such as **then**, **end**, —> and *:* have special meanings and may not be used as identifiers. Only the first 8 characters are significant.

## 3.2 DECLARATION AND INITIALIZATION

A variable is either *Global*, *Local*, or *Formal*. A *Declaration* is used to introduce an identifier and associate it with a global or local variable. A *Local Declaration*, introducing a local variable, is a declaration which occurs in a function body. A *Global Declaration*, introducing a global variable, is one which does not.

An *Initialization* is used to introduce an identifier and associate it with a formal variable and give the variable an initial value. It is achieved by including the identifier in the formal parameter list of a function (see section 4.1 'Definition of functions').

A declaration or initialization may also specify that the identifier is an operation, that is, it is restricted to take functions as its values and is given a property called a precedence, or it may specify that it is a macro.

The syntax of declarations is:

⟨*declaration*⟩ ::= **vars** ⟨*declaration list element* * ?⟩
⟨*declaration list element*⟩ ::= ⟨*identifier*⟩ | ⟨*property specification*⟩
⟨*property specification*⟩ ::= ⟨*property*⟩ ⟨*identifier*⟩ | (⟨*property*⟩⟨*identifier* * ?⟩)
⟨*property*⟩ ::= **macro** | **operation** ⟨*integer*⟩

Example:
⟨*declaration*⟩ *e.g.* ::= **vars** *x y* | **vars** *x y* **operation** *7* == **macro** *help*

A declaration or initialization has a *Scope*, which is a piece of POP-2 text. An identifier may not be used to represent a variable outside the scope of a declaration or initialization of the identifier.

The scope of a global declaration starts at the declaration and continues either until the identifier is cancelled (see section 3.3 'Cancellation') or if it is introduced internally to a section of the program (see section 3.4 'Sections') until the end of that section.

The scope of a local declaration starts at the declaration and continues to the end of the innermost function body enclosing it.

The scope of an initialization is the body of the function in which it occurs.

To sum up: a new identifier is introduced by introducing a fresh sequence of characters, and a new variable is introduced by each dynamic activation of a declaration or initialization.

A variable has an *Extent* which is a sequence of evaluations of expressions and statements.

The extent of a global variable starts from its declaration and continues indefinitely.

The extent of a local or formal variable starts on entry to the body of the function in which it is declared or initialized and continues until exit from the body. During this extent, the extent of any other variable with the same identifier is temporarily interrupted. This is called a *Hole in the Extent* of the other variable. Its value is not altered but it cannot be accessed or changed by assignment. Thus there is only one variable *Currently Associated* with a particular identifier during any

evaluation. Other variables associated with the identifier are temporarily inaccessible.

More than one global declaration of the same identifier is permitted provided they give the same properties.

A declaration of a local variable is not permitted if there is already a declaration of a local, or initialization of a formal with the same identifier for the same function body.

A *Standard Variable* is a global variable which already has a value on entry to the POP-2 system. A *Standard Function* (or *Routine*) is one which is the value of a standard variable. Standard variables are normally *Protected*, that is, no assignment may be made to them, or to components of their values (exceptions are indicated).

To enable one to discover the properties of an identifier there is a standard function *identprops* which, given a word (see section 8.6 'Words'), gives the properties of the corresponding identifier, if any, or, if the word is a syntax word, indicates this. The identifier must be standard or have been declared (and available within the current section, if any) and not cancelled, otherwise *identprops* produces *undef* as a result.

*identprops* ∈ *word*=> *item*

The result is as follows:

| | | |
|---|---|---|
| syntax word | — | *"syntax"* |
| macro identifier | — | *"macro"* |
| operation identifier | — | precedence (an integer) |
| other identifier | — | 0 |
| none of these | — | *undef* |

Examples:

```
identprops ("end")=>
**syntax
function getsets;
    if not(identprops("nonmac sets")="macro")
                        then compile (discfile([sets section]))
    close
end;
```

## 3.3 CANCELLATION

A cancellation terminates the scope of any declaration of an identifier and removes any properties associated with the identifier. The cancellation must occur textually between the old declaration and any new declaration. It may not occur in a function body.

The syntax of cancellations is:

⟨ *cancellation* ⟩ ::=**cancel**⟨ *identifier* * ? ⟩

The identifier of a standard variable may be cancelled. Syntax words may also be cancelled and used thereafter as ordinary identifiers. They may appear in place of identifiers in the above syntactic definition (for obvious reasons semicolon is excluded and so is **nonmac**).

## 3.4 SECTIONS

To enable parts of a program to be written in a self-contained manner they can be made into *Sections*. A section may itself contain other

inner sections. With each section we may associate a section name and some *External Identifiers*. External identifiers may be used inside or outside the section. Any other identifiers declared inside the section are called *Internal Identifiers* and they are distinguished from the same identifiers used outside the section as if their names had been systematically changed, as if, for example, an extra character peculiar to the section had been added to the end of each such identifier. Thus the scope of internal identifiers is confined to the section and associating with them properties such as having a precedence or being a macro or cancelling them does not affect any identifier with the same name outside the section.

The external identifiers have not necessarily been declared on entry to the section and their appearance in the external identifier list acts as a declaration. They may be given a precedence. At the end of the section they are available for use outside it. If they have already been declared this new declaration must indicate the same properties as the old one. Otherwise it is ignored.

Other identifiers which have been declared (and not cancelled) in the text enclosing the section may be used inside it without redeclaration, but if an identifier is used inside it without having been declared previously a declaration is inserted automatically, external to all sections.

Syntax
⟨*external list*⟩ *::==>* ⟨*declaration list element * ?*⟩
⟨*section name*⟩ *::=* ⟨*identifier*⟩
⟨*section*⟩ *::=* **section** ⟨*section name ?*⟩
⟨*external list*⟩; ⟨*program ?*⟩ **endsection**

Example
**vars** *a b;2 —> a; 3 —> b;*
**section** *example=>f;*
**vars** *b; 4—> a; 5 —> b;*
**function** *f; a=> b =>* **end**
**endsection;**
*a=>*
***4*
*b=>*
***3*
*f();*
***4*
***5*

The occurrence of the section name after **section** has the effect of adding this identifier to the externals with the property of being a macro (see section 11.2 'Macros') and as if it had previously been declared as a macro, thus

**macro** ⟨*section name*⟩; *macresults*([**nonmac** ⟨*section name*⟩
⟨*identifiers of the externals*⟩ ]) **end**

Thus when the section name is subsequently mentioned it is as if all the externals had been mentioned. It may be used, for example, to cancel the externals when the functions or other items produced by the section are no longer required.

Example
```
section arith => f operation 2 g;
    function f x; x+1 end;
    operation 2 g x; x—1 end;
endsection;
f(3)=>
**4
g3=>
**2
cancel arith;      (i.e. cancel nonmac arith f g;)
```

After the cancel statement the identifiers *f* and *g* and their associated functions are no longer available, nor is the macro *arith*.

Note the loose but suggestive analogy between a section and a function definition (see section 4.1 'Definition of functions').

# 4. FUNCTIONS

## 4.1 DEFINITION OF FUNCTIONS

A *Function* is a compound item. Definition and application of functions are treated in this section and the next. Certain properties of a function regarded as a data structure are treated in section 8.7 'Functions'.

A function consists of some *Formal Parameters* which are identifiers of formal variables, possibly some *Output Locals* which are identifiers of output local variables (see section 4.2 'Application of functions') and a *Body* which is an imperative sequence (see section 5.3 'Statements and imperatives').

A function which produces no results (see section 4.2 'Application of functions') is sometimes called a *Routine*.

Functions may be referred to in the program by using a function constant, called a *Lambda Expression*, or they may be standard functions provided by the POP-2 system, or they may be created by partial application or by application of a standard function which produces a function as a result.

The syntactic representation of a function constant is:

⟨*formal parameter list*⟩ ::= ⟨*declaration list element* * ?⟩
⟨*output local list*⟩ ::==> ⟨*declaration list element* * ?⟩
⟨*function body*⟩ ::= ⟨*imperative sequence*⟩
⟨*lambda expression*⟩ ::= **lambda** ⟨*formal parameter list*⟩
⟨*output local list* ?⟩ ; ⟨*function body*⟩ **end**

Example
⟨*lambda expression*⟩ *e.g.* ::= **lambda** *x y; cons(x, cons(a, y))* **end**
| **lambda** *x; nl(1); print(x)* **end**
| **lambda** *x* => *y; x* * *x* —> *y* **end**

We very often wish to declare a variable and then assign a function to it. The syntactic form of this will be as follows:

**vars** ⟨*identifier*⟩;⟨*lambda expression*⟩—> ⟨*identifier*⟩;
or **vars macro** ⟨*identifier*⟩;⟨*lambda expression*⟩—> **nonmac** ⟨*identifier*⟩;
or **vars operation** ⟨*integer*⟩⟨*identifier*⟩; ⟨*lambda expression*⟩—> **nonop** ⟨*identifier*⟩;

These are so common that a special syntactic form is introduced which is equivalent to them:

⟨*function*⟩ ::= **function** | **macro** | **operation** ⟨*integer*⟩
⟨*function definition*⟩ ::= ⟨*function*⟩ ⟨*identifier*⟩⟨*formal parameter list*⟩ ⟨*output local list?*⟩; ⟨*function body*⟩ **end**

If an identifier after **function** has been previously declared at this level no new declaration is implied and the function definition is equivalent simply to an assignment of a lambda expression.

Example

⟨*function definition*⟩ *e.g.* ::= **function** *max x y;* **if** *x>y* **then** *x* **else** *y* **close**
**end**
| **operation** *2 enter u v; cons(conspair(u, v), dict) —> dict* **end**
| **function** *order x y =>u v;*
**if** *x>y* **then** *x—>u; y—>v*
**else** *y—>u; x—>v* **close**
**end**

Functions may also be viewed as data structures (see section 8.7 'Functions').

There are standard functions

*erase* ∈ *item* =>
*identfn* ∈ ( ) => ( )

defined by

**function** *erase x;* **end;**
**function** *identfn;* **end;**

## 4.2 APPLICATION OF FUNCTIONS

The *Stack* is an ordered sequence of items. The last item to be added to this sequence is said to be on *Top of the Stack*. Items can be added to, or removed from, the top of the stack. On entry to the POP-2 system the stack is empty. The standard function *stacklength* gives the number of items on the stack at the time when it is applied.

*stacklength* ∈ ( ) => *integer*

By an *n-Tuple* we mean an ordered sequence of $n$ items ($n \geq 0$). An item is identical with the 1-tuple whose sole member is that item. An $n$-tuple is said to be on top of the stack if it forms the last $n$ items of the stack.

A function of $n$ arguments (that is, with $n$ formal parameters, excluding frozen formals; see section 4.4 'Partial application'), may be *Applied* to an $n$-tuple of items on top of the stack, whose members are called the *Actual Parameters* of the function. Application of a function to its actual parameters produces an $m$-tuple of items on top of the stack for some $m$, whose members are said to be the *Results* of the function.

A function which does not take a fixed number of arguments is called *Variadic*. A function which does not produce a fixed number of results is called *Variresult*.

The application of a function to its actual parameters consists of the following sequence of events:

*Entry:* a new variable corresponding to each formal parameter from right to left is initialized to the corresponding actual parameter value removing it from the stack, or if it is a frozen formal to the corresponding frozen value. A new variable corresponding to each local variable declaration in the function body but not in any interior function body is then created. The value of this new variable is not defined. The variables previously associated with the identifiers of formal or local variables can no longer be referred to but their values are undisturbed.

*Running:* the function body is evaluated with the variables created on entry.

*Exit:* the values of any *Output Local Variables* are placed on the stack and these together with any items which may already have been placed on the stack during the running of the function form the results of the function. The variables created on entry are terminated and the variable associated with each identifier reverts to what it was on entry. There is no change in the values of variables which were previously associated with the formal or local variable identifiers and have now been reinstated. The values of formal and frozen variables are lost. The frozen formals will be reinitialized from the frozen values on the next entry to the function, normally with the same values as last time; the frozen values can be changed by using *frozval* (see section 8.7 'Functions').

## 4.3 NONLOCAL VARIABLES

Variables which occur in a function body and are not locals of that function body (that is, declared in the body) or formals (that is, elements of the formal parameter list) or output locals are called *Nonlocal* to the function. They may be globals or locals of some outer function body which textually encloses it. Care must be taken not to apply a function with nonlocals in a hole in the extent of some of its nonlocals (see section 3.2 'Declaration and initialization') when their values are temporarily inaccessible, or outside their extent. Mention of the identifier of such a nonlocal would refer to a quite different variable currently associated with that identifier. The difficulty can arise particularly for functions passed as parameters or produced as results.

To avoid such difficulties a frozen formal may be used instead of the nonlocal, provided that it is not desired to assign a new value to the nonlocal as a result of the call. The frozen formal can be initialized by partial application (see section 4.4 'Partial application') to the value that the nonlocal would have taken. (The frozen formals can be used in this way to 'bind' the values of the nonlocal variables to the function and make it usable in any context.)

Example

```
function val f; vars a; 0->a; f() end;
vars a b; 88->a; 99->b;
val(lambda;b end)=>
**99
val(lambda;a end)=>
**0
val(lambda a; a end (% a %))=>
**88
```

In general, to make **lambda** *x...z;* ... **end** safe if it uses *a*, ..., *c* as nonlocal variables, write instead **lambda** *x...z a...c;...* **end** (% *a*, ..., *c* %)

In cases where assignment to the nonlocal is desired a frozen formal can be used and initialized to take a reference (see section 8.1 'References') as value. The component of this reference can then be assigned to, and so long as the reference is made the value of some other exterior variable the value is accessible outside the function body.

Another way to avoid unwanted clashes of identifiers, which suffices in many cases, is to write a function such as *val* above in a separate section of the program (see section 3.4 'Sections') with *a* as an internal identifier.

Example

```
section=>val; function val f; vars a; 0—>a; f() end;
endsection;
vars a; 88—>a;
val(lambda; a end)=>
**88
```

## 4.4 PARTIAL APPLICATION

In section 4.2 'Application of functions' we explained the method of applying a function to its arguments. There is a process somewhat analogous to application called *Partial Application.* By this means some of the formal parameters of a function may be made into *Frozen Formals,* producing a new function with fewer arguments. The frozen formals are always initialized to a fixed value when the function is applied and do not require any corresponding actual parameters (see, however, section 8.7 'Functions' for means of altering this fixed value). In other words, the actual parameters corresponding to the frozen formals are supplied once and for all on partial application. The values of the frozen formals are called the *Frozen Values.*

For example, by partially applying the two-argument function 'multiply' to 2 we get a one-argument function to double a number, and by partially applying it to 3 we get a function to triple a number. These two functions can coexist, and in general one function can be used to generate any number of others by partial application.

*Note.* The mechanism may be clearer if we mention a convenient possible implementation. On the partial application of a function $f$ to some items a new function is created which, when it is applied, first places these items on the stack and then calls the function $f$.

More formally we say that a function $f$ with $m$ formals may be partially applied to an $n$-tuple of actual parameters with $n \leq m$. We assume, for the moment, that $f$ has no frozen formals. The partial application produces a new function $f'$ with $m-n$ ordinary formals corresponding to the first $m-n$ formals of $f$, and $n$ frozen formals corresponding to the last $n$ formals of $f$. The function $f'$ has frozen values consisting of the $n$ items supplied as actual parameters of the partial application.

If $f$ itself has some frozen formals already, say $k$ of them, then $f'$ will have $n+k$ frozen formals and $n+k$ corresponding items in its frozen values.

The standard function *partapply* takes a function as its first argument and a list as its second argument, and partially applies the function to the elements of the list.

*partapply* ∈*function, list* => *function*

Note that partial application constructs a new function with particular frozen values, it does not alter the original function in any way. A function which has been produced as the result of partial application is called a *Closure Function*. The frozen values of a closure function can be selected or updated as can the constituent function from which it was derived by partial application (see section 8.7 'Functions').

If a doublet (see section 4.5 'Doublets') is partially applied to one or more items it produces a new doublet. The selector of the new doublet is obtained by partially applying the selector of the original doublet to the given items. The update routine of the new doublet is obtained by partially applying the update routine of the original doublet to the given items.

A special syntactic form is also available for partial applications. It is similar to that for ordinary application (see section 5.1 'Expressions').

⟨*partial application bracket*⟩ ::= (% | %)
⟨*partial application*⟩ ::= ⟨*non-operation identifier*⟩
(% ⟨*expression sequence*⟩ %)
| ⟨*lambda expression*⟩ (% ⟨*expression sequence*⟩ %)

The value of the variable currently associated with the identifier is partially applied to the sequence of expressions in the expression sequence. Thus, for example,

```
vars c; cons(%[is a number]%) —> c;
c(1)=>
**[1 is a number]
c(2)=>
**[2 is a number]
```

```
function f x y z; .. etc. end;
f(%y1, z1%)—>f1; f1(x1)=>
```

Thus the statement
*f*(%2, 3%)—>*f1*;

would have the same effect as
**function** *f1 x*; *f*(*x*, 2, 3) **end**

except that in the latter case f1 would not be a closure and hence the doublets *frozval* and *fnpart* could not be used to select or update its components (see section 8.7 'Functions').

The reader may note the analogy between frozen formals and initializable *own* variables, to use ALGOL terminology. They are what is called the environment of a closure in Landin (1964).

## 4.5 DOUBLETS

When dealing with data structures, functions called Selectors are defined which may be applied to a structure to produce its components (see section 7.1 'Functions of data structures'). To each selector there corresponds an Update Routine which alters the value of the component in the structure to a given new value.

Any function may have an update routine associated with it. This will normally only be done for selector functions. The function is then called a *Doublet*. When a function is created using a lambda expression its associated update routine is not defined. An update routine may be associated with it by using the doublet *updater* (see section 8.7 'Functions').

When a variable whose value is a doublet is used as the operator of a compound expression the selector function of the doublet is applied. But when such a variable is used as the operator of a destination expression (that is, as part of a destination of an assignment; see section 5.5 'Assignment') the update routine is applied.

It is convenient to extend our notation for functions (see section 1.5 'Notation for functions') using the new symbol '==>' to express concisely the domain and range of the selector and update routines of a doublet. Thus if $f$ is a doublet we write

$f \in d_1, \ldots, d_k ==> r$
meaning that $f$ has a selector $s$
$s \in d_1, \ldots, d_k ==> r$
and an update routine $u$
$u \in r, d_1, \ldots, d_k ==> ()$

Example
The standard function *hd* used in list processing (see section 8.3 'Lists') is a doublet.

```
vars l;[1 2 3 4] -> l;hd(l)=>
**1
5->hd(l);l=>
**[5 2 3 4]
hd(l)=>
**5
function second l;hd(tl(l)) end;
lambda x l;x->hd(tl(l)) end-> updater(second);
second(l)=>
**2
6-> second(l);l=>
**[5 6 3 4]
```

## 4.6 ARITHMETIC OPERATIONS

We say that an item is a *Number* if it is either a real or an integer. Arithmetic on numbers is performed by the standard functions, which are operations, shown in the table.

| *Operation* | *Precedence* | *Explanation* | *Result* |
|---|---|---|---|
| < | 7 | less than | truthvalue |
| > | 7 | greater than | truthvalue |
| =< | 7 | less than or equal | truthvalue |
| >= | 7 | greater than or equal | truthvalue |
| + | 5 | add | real or integer |
| – | 5 | subtract | real or integer |
| * | 4 | multiply | real or integer |
| / | 4 | divide | real |
| // | 4 | divide with remainder | integer, integer |
| ↑ | 3 | to the power | real |

+, —, and * produce an integer result if both arguments are integer, otherwise a real result. // is divide with remainder and, given two integers, it produces a remainder and a quotient. If $a$ // $b$ is $(r, q)$ then $q*b+r=a$ and $|r| < |b|$ and $r*a \geq 0$. For example, *10 // 3—> q—> r;* leaves $q$ equal to *3* and $r$ equal to *1*.

After a semicolon, comma, (, (%, [%, **if, loopif, and, or, then, else** or **elseif,** — is interpreted as an operation of one argument which negates an integer or a real, similarly + is there interpreted as an identity operation having no effect on a real or integer.

There is a standard function which produces —1, 0, or +1 according to the sign of a number

*sign ∈ integer or real => integer*

There are standard functions to convert a real to the nearest integer less than or equal to it and to convert an integer to a real

*intof ∈ real => integer*
*realof ∈ integer => real*

The equality operation =, of precedence 7, has already been defined in section 2.1 'Simple and compound items'. An integer is never equal to a real.

# 5. EXPRESSIONS AND STATEMENTS

## 5.1 EXPRESSIONS

An *Expression* is either a simple expression, a compound expression, a conditional expression, an imperative expression (see section 5.3 'Statements and imperatives'), or a parenthesized expression sequence.

A *Simple Expression* is either an identifier or a *Constant*, a constant being an integer, a real, or a structure constant. If the simple expression is an identifier then its value is the value of the variable currently associated with that identifier. If it is a constant then its value is the item denoted by the constant. A Structure Constant is either a lambda expression, which is dealt with in section 4.1 'Definition of functions' and in section 8.7 'Functions', a word constant, a string constant, or a list constant, all of which are dealt with in section 8 'Standard structures'.

A *Compound Expression* has an *Operator* which is an expression and some *Operands* which are an expression sequence. The value of a compound expression is found by evaluating the operands and evaluating the operator, whose value should be a function (see section 4.5 'Doublets' for the case where the operator is a doublet). The sequence in which these evaluations are carried out is not defined. The function obtained from the operator is then applied to the $n$-tuple obtained by evaluating the operands. The results of this application are the value of the expression. Thus the value of the expression is an $n$-tuple, with $n=0$ if the function is a routine.

Evaluation of a compound expression affects the stack as follows. The operand expressions, when evaluated, leave their results on the stack, so does the operator expression. The top item of the stack is then immediately removed and applied, taking as arguments the items

remaining on top of the stack. If the number of arguments required is greater than the number resulting from evaluation of the operands, one or more items which were on top before the evaluation will be used as well; if it is less, then some of these results will not be used and will remain on the stack. The results of the application, if any, are left on the stack.

Evaluation of conditional expressions is described in section 6.1 'Conditional expressions', and that of imperative expressions in section 5.3 'Statements and imperatives'.

An expression sequence is evaluated by evaluating the expressions of which it consists and placing the results on the stack. The order in which the evaluations are made is not defined. The order in which the results of evaluating the expressions are used to form the $n$-tuple is the order in which the expressions occur.

The syntax of expressions is given below. There are a number of syntactic forms for compound expressions. A further explanation of the syntax is given in section 5.2 'Precedence'.

⟨*non-operation identifier*⟩ ::= ⟨*identifier*⟩ | **nonop** ⟨*operation*⟩
⟨*constant*⟩ ::= ⟨*integer*⟩ | ⟨*real*⟩ | ⟨*structure constant*⟩
⟨*structure constant*⟩ ::= ⟨*lambda expression*⟩ | ⟨*quoted word*⟩
| ⟨*string constant*⟩ | ⟨*list constant*⟩
⟨*simple expression*⟩ ::= ⟨*non-operation identifier*⟩ | ⟨*constant*⟩
⟨*operation*⟩ ::= ⟨*identifier*⟩
⟨*parenthesis*⟩ ::= ( | )
⟨*compound expression*⟩ ::= ⟨*non-operation identifier*⟩⟨*parenthesized expression **⟩ | ⟨*parenthesized expression*⟩ ⟨*parenthesized expression **⟩
| ⟨*expression ?*⟩ ⟨*operation*⟩ ⟨*expression ?*⟩
| ⟨*closed expression ?*⟩ ⟨*dot operator **⟩
| ⟨*parenthesized expression*⟩ ⟨*dot operator **⟩
| ⟨*structure expression*⟩
⟨*parenthesized expression*⟩ ::= (⟨*expression sequence*⟩) | ⟨*imperative expression*⟩
⟨*closed expression*⟩ ::= ⟨*simple expression*⟩ | ⟨*list expression*⟩
| ⟨*conditional expression*⟩
⟨*dot operator*⟩ ::= . ⟨*non-operation identifier*⟩ | . ⟨*parenthesized expression*⟩
⟨*structure expression*⟩ ::= ⟨*partial application*⟩ | ⟨*list expression*⟩
⟨*expression sequence*⟩ ::= ⟨*expression ?*⟩ , ⟨*expression sequence*⟩
| ⟨*expression ?*⟩
⟨*expression*⟩ ::= ⟨*simple expression*⟩ | ⟨*compound expression*⟩
| ⟨*conditional expression*⟩ | ⟨*parenthesized expression*⟩

Examples

⟨*simple expression*⟩ *e.g.* ::= *x* | **nonop** *+* | *3* | **lambda** *x;x+1* **end**
| *[3 5 9]*
⟨*operation*⟩ *e.g.* ::= *+* | **** | *adjoin*
⟨*compound expression*⟩ *e.g.* ::= *f(x+1, y)* | *(f fncomp g) (2)* | *a*(b+c)* | *f()*
| *x.hd* | *.f* | *f(%x%)* | *[%x, x+2%]*
⟨*expression*⟩ *e.g.* ::= *a* | *g(h(x+1))* | **if** *x=0* **then** *y* **else** *z* **close**
| *(x+1—>x; y+1—>y; x*y)*
| *(x, y+1, z—1)*

The various syntactic forms of compound expressions denote the operator and operands in the following way:

*(a)* ⟨*non-operation identifier*⟩ ⟨*parenthesized expression **⟩

If there is just one parenthesized expression its component expressions form the operands and the non-operation identifier is the operator. In general, the component expressions of the last parenthesized expression form the operands and the non-operation identifier, followed by all the parenthesized expressions except the last, treated as a single compound expression, forms the operator. Thus, for example, *f(g)(x)* is equivalent to *(f(g))(x)*, meaning apply *f* to *g* and apply the result to *x*.

*(b)* ⟨*parenthesized expression*⟩ ⟨*parenthesized expression **⟩

Similar to *(a)* but using the first parenthesized expression in place of the non-operation identifier.

*(c)* ⟨*expression ?*⟩ ⟨*operation*⟩ ⟨*expression ?*⟩. This is equivalent to: **nonop** ⟨*operation*⟩ (⟨*expression ?*⟩, ⟨*expression ?*⟩) which is a special case of *(a)* above.

*(d)* ⟨*closed expression ?*⟩ ⟨*dot operator **⟩

This could be rewritten
⟨*closed expression ?*⟩ ⟨*dot operator *?*⟩ . ⟨*non-operation identifer*⟩ | ⟨*closed expression ?*⟩ ⟨*dot operator *?*⟩ . ⟨*parenthesized expression*⟩

The non-operation identifier or parenthesized expression is the operator and the remainder is the operands. Thus, for example, *x.f.g* is equivalent to *g(f(x))*.
⟨*parenthesized expression*⟩ ⟨*dot operator **⟩ is analogous, with a parenthesized expression in place of the closed expression.

*(e)* ⟨*structure expression*⟩
This is equivalent to *(a)* above with a special identifier for the operand. The exact rules are given in section 4.4 'Partial application' and section 8.3 'Lists'.

## 5.2 PRECEDENCE

If a compound expression or destination expression (see section 5.5 'Assignment') is of the form

⟨*expression ?*⟩ ⟨*operation*⟩ ⟨*expression ?*⟩

the operator is the operation. In this case ambiguity might arise in the analysis of expressions such as

⟨*expression*⟩ ⟨*operation*⟩ ⟨*expression*⟩ ⟨*operation*⟩ ⟨*expression*⟩

which could be analyzed with association to the left or to the right. This ambiguity is resolved by the notion of precedence. A *Precedence* is a positive integer between 1 and 9 associated with an operation identifier. It is set by a declaration and can only be changed by cancellation. The operator of a sequence of expressions containing one or more operations is the operation of highest precedence, or if there is more than one operation of highest precedence the rightmost of these.

It must be made clear that the difference between an operation and any other identifier is purely a syntactic one, except for the restriction that its value must be a function.

It may be desired to use an operation in a context other than as the operator of a compound expression. If so it must be prefixed with the word **nonop** in which case it is treated syntactically like any other identifier. The use of **nonop** overrules the precedence of the identifier. This facility enables operations to appear as operands and enables assignment to operations.

Example
> has precedence *7*, + and – have precedence *5* and * has precedence *4*.
*5–x+2*y>1+2* is the same as *((5–x)+(2*y))>(1+2)*.

## 5.3 STATEMENTS AND IMPERATIVES

A statement is either an assignment, a **goto** statement, a comment, a machine code instruction, or an expression sequence. It may be labelled.

An imperative is either a declaration or a statement.

The syntax is:

⟨*statement*⟩ ::= ⟨*assignment*⟩ | ⟨*goto statement*⟩ | ⟨*comment*⟩
| ⟨*code instruction*⟩ | ⟨*expression sequence*⟩
| ⟨*labelled statement*⟩
⟨*imperative*⟩ ::= ⟨*declaration*⟩ | ⟨*statement*⟩
⟨*imperative sequence*⟩ ::= ⟨*imperative*⟩;⟨*imperative sequence*⟩
| ⟨*imperative ?*⟩

Example
⟨*imperative sequence*⟩ *e.g.* ::= *loop: x–1–>x; f(x)–>y;* **if** *x>0*
**then goto** *loop* **close**
| *x+1–>y; y; u–>y; –>z*

The evaluation of an *Imperative Sequence* consists of evaluating the statements in the sequence in which they occur, except when a goto statement occurs and the sequence continues at the point indicated by the goto statement.

An *Imperative Expression* may be formed from an imperative sequence.

The syntax is
⟨*imperative expression*⟩ ::= (⟨*imperative sequence*⟩)

When a statement is evaluated items may be removed from the top of the stack and any results produced are added to the top of the stack (see section 4.2 'Application of functions'). The results of an imperative sequence are the items left on the stack, if any, when the sequence has been evaluated.

## 5.4 LABELS AND GOTO STATEMENTS

A *Label* may be attached to a statement. Evaluation of a *Goto Statement* using that label causes the sequence of evaluation to be changed so that the labelled statement is evaluated next. A goto statement may not refer to a label outside the function body in which it occurs. If a goto statement occurs in an operand of a compound or destination expression it may not refer to a label outside that operand. The syntax is:

⟨*labelled statement*⟩ ::= ⟨*label*⟩:⟨*statement ?*⟩
⟨*goto statement*⟩ ::= **goto** ⟨*label*⟩ | **return**
| ⟨*expression ?*⟩ **switch** ⟨label*?⟩

⟨*label*⟩ ::= ⟨*identifier*⟩

The statement **return** causes transfer of control to the exit of the innermost current function body. There is a standard macro **exit** which is synonymous with **return close.**

If the expression before **switch** (or the top of the stack) has value *i*, this is equivalent to **goto** the *i*th label after **switch.**

If an identifier is used for a label it may not appear as an identifier associated with a variable in the text constituting that function body.

Goto statements and labelled statements may only occur inside a function body.

Note that a label is not an item.

The fact that a goto statement may not cause a jump to a label outside the function body in which it occurs may prove restrictive and so a special standard function *jumpout* is provided (a device due to Landin, 1966).

*jumpout* ∈ *function, integer* => *function*

If *f* is any function the statement
*jumpout(f, n)—>f1*

occurring in the body of a function *g* produces a function *f1* such that if *f1* is called any time before exit from the body of *g* it will behave exactly as the function *f*, but on exit from *f1* an immediate exit from the body of *g* will be precipitated. Furthermore the last *n* items on top of the stack after execution of *f* will be left on the stack, but all other items in excess of the number which were on the stack when *jumpout* was originally applied will be removed. If *n* is *undef* then calling *f1* has no effect on the stack. Output locals of *g* are not put on the stack. Thus

```
vars escape;
function try n;
    if n.isperfectsquare then escape(n) else try(n+1) close
end;
function g k;
    jumpout(sqrt,1)—>escape; try(k)
end;
g(21)=>
**5
```

Thus the effect is analogous to saying **return** in the body of *g*, but this can be done during execution of a subsidiary function and provision is made to pass out results. Note that the return point depends on the function body in which *jumpout* is applied.

## 5.5 ASSIGNMENT

An *Assignment* consists of a *Source*, which is an expression or sequence of expressions, and a *Destination Sequence*, which is a sequence of elements, each of which is either an identifier or a *Destination Expression*. It is used to remove one or more items from the stack, and to make them become the values of variables or to update a data structure with them.

A destination expression has an operator which is an expression and some operands, that is, a sequence of expressions (possibly an empty sequence). A destination expression is not one of the kinds of expression

defined in section 5.1 'Expressions', but it is very similar to a compound expression. It is distinguished by appearing on the right-hand side of an assignment and by having a special evaluation rule.

An assignment is evaluated as follows. First the source is evaluated to yield an $n$-tuple of items which are placed on the stack. The last $k$ elements on the stack, which we will call the *Source Items*, are then taken in sequence starting from the last and each source item is combined with the corresponding destination element (taken in sequence starting from the first) as follows:

1. If the destination element is a variable the top item of the stack becomes the new value of that variable.
2. If the destination element is a destination expression the operator and operands of this expression are evaluated. The value of the operator must be a doublet (see section 4.5 'Doublets') and its update routine is applied. (Contrast the evaluation of a compound expression where the select function is applied.)

Thus $E_0{-}{>}f(E_1, \ldots, E_k)$ has the same affect as *apply*$(E_0, E_1, \ldots, E_k,$ *updater*$(f))$.

The syntax of destination expressions is given below. A further explanation of this syntax is given in section 5.2 'Precedence'.

⟨*destination expression*⟩ ::= ⟨*non-operation identifier*⟩ ⟨*parenthesized expression* *⟩
| ⟨*parenthesized expression*⟩ ⟨*parenthesized expression* *⟩
| ⟨*expression ?*⟩ ⟨*operation*⟩ ⟨*expression ?*⟩
| ⟨*closed expression ?*⟩ ⟨*dot operator* *⟩
| ⟨*parenthesized expression*⟩ ⟨*dot operator* *⟩

The syntax of assignments is:

⟨*assignment*⟩ ::= ⟨*expression sequence*⟩ ⟨ *destination* *⟩
| ⟨*function definition*⟩
⟨*destination*⟩ ::= −>⟨*non-operation identifier*⟩ | −>⟨*destination expression*⟩

Example

⟨*assignment*⟩ *e.g.* ::= *x+1−>y* | *u+v−>a(i, j)* | *x//y−>u−>v*
| *x, y−>x−>y*

In the second example *a(i, j)* is a destination expression and the whole assignment is a euphemism for *a1(u+v, i, j)* where *a1* is the update routine of the doublet *a*. The last example exchanges the value of *x* and *y*.

Function definitions are special syntactic forms for assignments.

## 5.6 COMMENTS

A comment is a statement which is ignored by the compiler.
⟨*comment*⟩ ::= **comment** ⟨*any sequence of character groups other than ;*⟩

# 6. CONDITIONALS

## 6.1 CONDITIONAL EXPRESSIONS

A conditional expression consists of a sequence of one or more pairs of component expressions, each pair consisting of a *Condition* and a *Consequent.* It may be an ordinary conditional expression or a loop conditional expression.

The condition may be an expression, a conjunction or a disjunction (see section 6.2 'Conjunctions and disjunctions'). The consequent is an expression or statement. The method of evaluation of a conditional is as follows.

The first pair in the sequence is taken and its condition is evaluated; if the value is not the truth value *false* then its consequent is evaluated, otherwise the next pair in the sequence is taken, and so on, until either a condition produces a value other than *false* or the sequence is exhausted. In the case of an ordinary conditional expression, evaluation of the conditional expression terminates as soon as a consequent has been evaluated or the sequence has been exhausted. In the case of a loop conditional expression evaluation of the conditional expression restarts from the beginning as soon as a consequent has been evaluated, but if the sequence is exhausted without a consequent being evaluated the evaluation of the conditional expression terminates.

The syntax of conditional expressions provides a compact notation for the sequence of conditions and consequents with an abbreviation for the case where the final condition is *true*. It is:

⟨*consequent*⟩ ::= ⟨*imperative sequence*⟩
⟨*ordinary or loop if*⟩ ::= **if** | **loopif**
⟨*elseif clause*⟩ ::= **elseif** ⟨*condition*⟩ **then** ⟨*consequent*⟩
⟨*else clause*⟩ ::= **else** ⟨*consequent*⟩
⟨*conditional expression*⟩ ::=
    ⟨*ordinary or loop if*⟩ ⟨*condition*⟩ **then** ⟨*consequent*⟩
    ⟨*elseif clause * ?*⟩
    ⟨*else clause ?*⟩ **close**

The conditional expression is ordinary if it starts with **if** and it is a loop conditional if it starts with **loopif.**

The first condition and consequent occurs before and after **then** respectively. The remaining pairs of the sequence, if any, appear as elseif clauses and possibly as an else clause. The else clause has only a consequent and the missing condition is assumed to give the value true.

Example
⟨*conditional expression*⟩ *e.g.* ::=
    **if** *x>0* **and** *x<3* **then** *y*
        **elseif** *x>3* **then** *z*
        **else** *0*
    **close**
  | **if** *x=0* **then** *1—>y* **close**
  | **loopif** *n>0* **then** *r(n); n—1—>n* **close**

One may use **loopif** to define macros (see section 11.2 'Macros') to do various kinds of iteration. There is a standard macro *forall* which

allows the most common kind of iteration. If *I*, *M*, *K*, and *N* are any text items (see section 9.1 'Input'),

*forall I M K N*

expands to
*M–K–>I;* **loopif** *(I+K–>I;I=<N)* **then**

Example
*forall j m 2 n;j=>* **close**
has the same effect as
*m–2–>j;* **loopif** *(j+2–>j;j=<n)* **then;** *j=>* **close;**

## 6.2 CONJUNCTIONS AND DISJUNCTIONS

A *Conjunction* is composed of two component expressions each producing a single result. The method of evaluating a conjunction is to evaluate the first component expression and if its value is the truth value *false* the value of the conjunction is truth value *false*, otherwise the second expression is evaluated and the conjunction has value *false* if the second component expression has value *false* otherwise *true*.

A *Disjunction* is composed of two component expressions each producing a single result. The method of evaluating a disjunction is to evaluate the first component expression and if its value is not the truth value *false* the value of the disjunction has truth value *true*, otherwise the second expression is evaluated and the disjunction has value *false* if the second component expression has value *false* otherwise *true*.

A number of conjunctions and disjunctions can be combined to form a condition.

The syntax is
⟨*condition*⟩ ::= ⟨*expression*⟩ **and** ⟨*condition*⟩ | ⟨*expression*⟩ **or** ⟨*condition*⟩ | ⟨*expression*⟩

These three kinds of conditions are respectively a conjunction, a disjunction and an expression.

Thus **and** and **or** associate to the right.

⟨*condition*⟩ *e.g.* ::= *x<10* **and** *x>0* | *x>10* **or** *x<0* | *null(x)* | *b*
| *p(x)* **and** *q(x)* **or** *r(x)*

In the last example the following cases can occur ('—' means that the expression is not evaluated).

| *p(x)* | *q(x)* | *r(x)* | value of condition |
|---|---|---|---|
| *false* | — | — | *false* |
| *true* | *false* | *false* | *false* |
| *true* | *false* | *true* | *true* |
| *true* | *true* | — | *true* |

# 7. DATA STRUCTURES

## 7.1 FUNCTIONS OF DATA STRUCTURES

A *Data Structure* is a compound item which has other items as its *Components*. For each class of data structures there is a family of functions called the *Characteristic Functions* acting upon structures of

that class. These functions are a constructor, a destructor, selectors, and update routines. A given compound item may represent a number of different data structures by being used in association with more than one family of functions and hence having different components. The number of components may be zero.

Given values for its components it is possible to construct a data structure using a *Constructor* function, say *c*.

*c* ∈ *component*, . . . , *component* => *data structure*

It is possible to select the value of a component of a data structure. For each component there is a *Selector* function, say *si*.

*si* ∈ *data structure* => *component*

It is possible to update a component of a data structure, that is, to give it a new value. For each component there is an *Update Routine*, say *ui*:

*ui* ∈ *component*, *data structure* => ()

When a data structure is updated the old version is overwritten.

It is convenient to define another function called a *Destructor* function, say *d*, which is the inverse of the constructor, that is, given a data structure it produces its components as results.

*d* ∈ *data structure* => *component*, . . . , *component*

There is a relation called equality (see section 2.1 'Simple and compound items') which may hold between two compound items. It is denoted by the standard function = (an operation of precedence 7). This function is also defined for simple items with the usual meaning.

= ∈ *item*, *item* => *truthvalue*

Thus if the value of an expression '*E1*' is equal to the value of an expression '*E2*' then the expression

*E1=E2*

has value *true*.

Equality means that the two compound items contain the same address, that is, they point to the same area of store. If the items are not equal they point to entirely different areas of store. We say that a compound item is *Copied at the Top Level* if a new item is formed pointing to a new area of store which contains items equal to those of the given compound item. The new item and the previous one are not equal. They are, however, *Equivalent*.

Equivalent compound items are defined as items which are either equal or all of whose components are equivalent.

Updating an item alters a component item in the store area pointed to by the item but does not cause copying.

We will now give a more formal explanation of equality, but the model in terms of addresses and storage may be kept in mind.

Equality is an equivalence relation, that is, it is
(a) reflexive *(x=x)*;
(b) symmetric *(if x=y then y=x)*; and
(c) transitive *(if x=y and y=z then x=z)*.

It has the following other properties:
(d) The value of a formal parameter variable is equal to the corresponding actual parameter.
(e) If an item is assigned to a variable then the value of the variable is equal to that item.
(f) An item, other than a word or simple item, which is read in (see section 9.1 'Input') is not equal to any other item.
(g) The rules for equality of words are given in section 8.6 'Words'.
(h) Two integers or two reals are equal according to the usual rules of arithmetic. An integer is never equal to a real.

Items are equal only if their equality follows from the above properties.

We can now state some relationships between the various functions on data structures. We will use $a$ and $b$ for data structures, $x_1, \ldots, x_i, \ldots, x_k$ for items occurring as components, $s_1, \ldots, s_i, \ldots, s_k$ for selectors, $u_1, \ldots, u_i, \ldots, u_k$ for update routines, $c$ for a constructor, and $d$ for a destructor.

(a) $s_1(a), \ldots, s_k(a)$ is the same $n$-tuple as $d(a)$, that is, they have equal elements.
(b) $s_i(c(x_1, \ldots, x_i, \ldots, x_k))=x_i$ is true.
(c) $c(s_1(x), \ldots, s_k(x))=x$ is always false, but the left-hand expression is equivalent to $x$.
(d) After evaluating $u_i(x_i, a)$,
$s_i(a)=x_i$ is true.
(e) After evaluating $u_i(s_i(a), a)$, $a$ is unchanged.
(f) If $a=b$ and then $u_i(x_i, a)$ is evaluated, $s_i(b)=x_i$ is true and $a=b$ is still true.
(g) From (a) and (b) above $d(c(x_1, \ldots, x_k))$ is the same $n$-tuple as $x_1, \ldots, x_k$, that is, they have equal elements.

If $a$ and $b$ are data structures and $a$ is not equal to $b$ and updating a component of $a$ also updates some component of $b$, then $a$ and $b$ are said to *Share*.

When we wish to discuss a class of data structures which do not all have the same number of components (such as strips, see section 7.3 'Strips') it is convenient to define a *General Selector* function and a *General Update Routine*.

The general selector function, say $s$, has as arguments, an integer, $i$, and a data structure. It selects the $i$th component of the data structure.

$s \in integer, data\ structure \Rightarrow component$

Thus if $s_i$ is the $i$th selector, $s(i, a)=s_i(a)$.
Similarly for the general update routine, say $u$,

$u \in component, integer, data\ structure \Rightarrow ()$

Thus if $u_i$ is the $i$th update routine, $u(x_i, i, a)$ has the same effect as $u_i(x_i, a)$.

The programmer is able to create new kinds of data structures called records and strips (see section 7.2 'Records' and 7.3 'Strips'). He can also create functions by methods already described. He may be able to create other kinds of data structures using extra standard functions or machine code, but this depends on the implementation. Certain classes of records and strips are standard and these are described in section 8

'Standard structures'. Functions considered as data structures are also described there.

There is a standard doublet *dataword* which, given a data structure, produces or updates the item, normally a word, associated with the data structure class to which it belongs. It is also defined for integers and reals giving *"integer"* and *"real"* respectively. In the case of standard data structures, integers, and reals the dataword may not be updated.

*dataword* $\in$ *item* $\Longrightarrow$ *item*

There is a standard function
*samedata* $\in$ *item, item* $\Rightarrow$ *truthvalue*

which gives *true* if, and only if, the items belong to the same data structure class. It is also defined for integers and reals and distinguishes them from other classes.

There are a number of special expressions called 'structure expressions' used to construct these standard structures (see section 5.1 'Expressions').

Given a class or several classes of data structures with their associated functions it is possible to define functions which characterize a new family of structures. We will call these *Derived Structures*. Suppose, for example, that we have a class of structures with two selectors, say *s1* and *s2*, and components which are full items and members of the same class of structures. We can then define a new class of derived structures whose selectors are given by:

**function** *s11 a; s1(s1(a))* **end; function** *s12 a; s1(s2(a))* **end;**
**function** *s21 a; s2(s1(a))* **end; function** *s22 a; s2(s2(a))* **end;**

If *c* is the constructor of the first class we define the new constructor:

**function** *cc x1 x2 x3 x4; c(c(x1, x2), c(x3, x4))* **end;**

Note that if it is associated with two or more families of functions the same compound item can represent two or more structures, one of each class. However, for each class of compound items there is one *Primitive Data Structure Class* and other data structures are defined in terms of this primitive class. A primitive data structure does not share with any other primitive data structure. Examples of standard data structures which are not primitive are links and lists.

## 7.2 RECORDS

A *Record* is a compound item which is a member of a *Record Class*. The *Size* of a set of items is an integer or the word *"compnd"*. If all the items in the set are restricted to be non-negative integers less than $2^n$, the size is the integer $n$, if they are restricted to be compound items the size is the word *"compnd"*, otherwise if the component is a *Full Item* (that is, the set is not restricted) the size is the integer 0. For each component of a record there is a size associated with the set of possible values of that component. The *Specification of a Record* is the list of sizes associated with its components. A record class is a set of records which all have the same specification, and this is said to be the specification of the record class. Note that a record class is not an item. An item is associated with each record class. This is normally a word, although it may be any item.

A family of functions is associated with each record class to form a primitive class of data structures. This family comprises a set of selectors (∈ *record* => *component*) and a set of corresponding update routines (∈ *component*, *record* => ()), a constructor (∈ *component*, . . . , *component* => *record*) and a destructor (∈ *record* => *component*, . . . , *component*). Each selector function may be paired with the corresponding update routine to form a doublet (∈ *record* = => *component*). The standard function *recordfns* is used to create a new record class. It requires as arguments the item (normally a word) to be associated with the record class, and the specification of the record class. It produces the constructor, the destructor, and the doublets for the record class in the order in which they are given in the specification. The number of its results depends on the length of the specification list. Normally the programmer will immediately assign these resulting functions to variables.

*recordfns* ∈ *item*, *specification* => *constructor*, *destructor*, *doublet*, . . . , *doublet*

Examples

```
recordfns("person", [0 7 1])->sexof->ageof->nameof->destper
                            ->consper;
```

There is a standard function which converts a record to a list of its components:

*datalist* ∈ *record* => *list*

The standard doublet *dataword* when given a record produces or updates the item associated with its record class (see section 7.1 'Functions of data structures').

The function *copy* copies a record at the top level.
*copy* ∈ *record* => *record*

The functions *datalist*, *dataword*, and *copy* are defined over records of any class, and whenever *recordfns* is used to create a new record class these four functions are extended to deal with records of that class.

## 7.3 STRIPS

A *Strip* is a compound item which is a member of a *Strip Class*. All components of a strip must have the same size (see section 7.2 'Records' for definition of size) which is called the *Component Size* of the strip. All strips in a strip class must have the same component size but not necessarily the same number of components. An item (normally a word) is associated with each strip class.

A family of functions is associated with each strip class to form a primitive class of data structures. This family includes a general selector function (∈ *integer*, *strip*=>*component*) and a general update routine (∈ *component*, *integer*, *strip*=>() ). The selector function is paired with the update routine to form a doublet. The components are numbered from 1 upwards. It also includes for each strip class an initiator function (∈ *integer*=>*strip*). This constructs a strip with the given number of components but the values of these components are not defined. The initiator may be used with the update function to define a constructor function for strips of the strip class.

The standard function *stripfns* is used to create a new strip class. It takes as arguments the item to be associated with the strip class and

the component size of the strip class. It produces as results the initiator function and the doublet for the strip class:

*stripfns* ∈ *item, size* => *initiator, doublet*

There is a standard function which converts a strip to a list of its components. This is *datalist* (see section 7.2 'Records'). There is a doublet which, given a strip, produces or updates the item associated with its strip class. This is *dataword* (see section 7.1 'Functions of data structures').

There is a function *datalength* which, given a strip, produces the number of components it has.

*datalength* ∈ *strip* => *integer*

The function *copy* copies a strip at the top level (see section 7.2 'Records').

The functions *datalist, dataword,* and *copy* act for strips just as for records.

## 7.4 GARBAGE COLLECTION

Storage for the construction of data structures is made available by a storage control system. This system must be able to make use of areas of store which have been used but are no longer required. This is achieved by a process known as *Garbage Collection* which is undertaken whenever the system runs short of store. This first of all discovers what items can still be referred to by the programmer, for example, because they are the value of a variable whose extent has not finished (see section 3.2 'Declaration and initialization'). Any items which can no longer be referred to are destroyed, that is, their storage area is returned to the system for use in constructing other items. Since he cannot refer to them, the programmer is not aware of this destruction.

If variables refer to compound items which are no longer in use, the garbage collector cannot recover the associated storage. The variable should be reset, for example, to zero. In the case of identifiers the identifier can be cancelled (see section 3.3 'Cancellation').

# 8. STANDARD STRUCTURES

## 8.1 REFERENCES

There is a standard record class called *References*. These have one component which is a full item. The item associated with the class as its *dataword* is *"ref"*. Thus before entry to the POP-2 system this class is created using *recordfns*, and the resulting functions are assigned to variables to give the following standard functions:

| | |
|---|---|
| constructor: | *consref* ∈ *item* => *reference* |
| destructor: | *destref* ∈ *reference* => *item* |
| doublet: | *cont* ∈ *reference* ==> *item* |

A reference may be used, for example, as an actual parameter of a function to enable the function to cause side effects by updating the reference.

## 8.2 PAIRS

There is a standard record class called *Pairs*. Records of this class have two components which are both full items. The item associated with the class is *"pair"*. Thus before entry to the POP-2 system this class is created using *recordfns*, and the resulting functions are assigned to variables to give the following standard functions:

constructor: *conspair ∈ item, item => pair*
destructor: *destpair ∈ pair => item, item*
doublets: *front, back all ∈ pair ==> item*

An *Atom* is an item which is not a pair. Atoms are recognized by the standard function *atom*.

*atom ∈ item => truthvalue*

## 8.3 LISTS

There is a standard derived structure called a *Link* which is used to construct another derived structure called a *List*. Lists in POP-2 include structures analogous to LISP Lists, but also structures which compute the elements dynamically (cf. P. J. Landin's 'streams').

Links are not primitive data structures. They are represented by pairs, but although, like pairs, they have two components the doublets to select and update these components are more complex than those for pairs, indeed they are defined below in terms of the doublets for pairs. Similarly lists, which may have any number of components, are defined in terms of links. Lists do not have their own doublets to select and update components, but they do have constructors, namely the list constants and list expressions defined below.

The word *"nil"* is used to represent the *Null List* and the standard variable *nil* takes this value on entry to the POP-2 system. The null list is also represented by a pair whose front is *false* and whose back is a function and by a pair whose front is *false* and whose back is a function of no arguments which produces the terminator when it is applied.

The standard function *null* recognizes the null list

*null ∈ list => truthvalue*

A list is either the null list or it is a link.

A link is either:

(a) a pair whose front component is any item and whose back component is a list; or
(b) a pair whose front component is not *false* and whose back component is a function with no arguments and one result.

Case (a) gives rise to the familiar notion of a list (cf. LISP lists). In case (b) the function is one which when repeatedly applied produces a succession of items, not necessarily all the same, that is, normally the function will side-effect itself. The last item produced should be the terminator. Such a function is called a *Repeater*. For example, this enables us to convert an input file to a list. Lists with this sort of link are dynamic and some or all of their elements are computed rather than stored statically.

The characteristic functions of a link are:

| | | |
|---|---|---|
| constructor: | *cons* ∈ *item, link* => *link* | |
| destructor: | *dest* ∈ *link* => *item, list* | |
| doublets: | *hd* ∈ *link* ==> *item* | (called the 'head') |
| | *tl* ∈ *link* ==> *item* | (called the 'tail') |

These functions are very similar to those for pairs, but in the case of a link of the second kind special precautions are taken to make sure that on applying the selector *tl* the front component is not lost but preserved in a pair. Thus, if *x* has a list as its value and *tl(x)* is evaluated, there is a side effect on *x*, but matters are so arranged that this side effect is not detectable using the list-processing functions. The function *cons* is the same as *conspair* and produces a link of the first kind. The following standard function produces a link of the second kind or the null list given a function of no arguments:

```
fntolist ∈ (() => item) => list
function fntolist f; conspair (true, f) end
```

The other characteristic functions are defined as follows:
First an auxiliary function (*not* standard) to convert the first link of a dynamic list to static form.

```
function solidified l; vars f x;
    if l=nil then nil exit;
    if front(l)=false and isfunc(back(l)) then nil exit;
    if not(isfunc(back(l))) then l exit;
    if not(front(l)=false) and isfunc(back(l)) then
        back(l)->f;f()->x;
        if x=termin then false->front(l); nil exit;
        if not(x=termin) then x->front(l); conspair(true, f)->back(l)
        exit
    close
end;
function hd l ; front(solidified(l)) end;
lambda i l ; i->front(solidified(l)) end->updater(hd);
function tl l; back(solidified(l)) end;
lambda i l; i->back(solidified(l)) end->updater(tl);
function dest l; hd(l), tl(l)  end;
function null l; solidified(l)=nil end;
```

The functions *fntolist, hd, tl, dest* and *null* may be implemented differently from the above definitions as long as the difference cannot be detected by using any of these functions and definition (b) above holds.

The function which recognizes links is

*islink* ∈ *item* => *truthvalue*

A list may have no components (if it is the null list) or one or more (if it is a link).

If it is a link its first component is the head component of the link and its remaining components are the components of the list which is the tail component of the link. Thus the characteristic functions for lists can be defined in terms of those for links.

The function

*islist* ∈ *item* => *truthvalue*

recognizes lists (including the null list).

There are two special syntactic forms for constructing lists. These are list constants and list expressions. List constants may have lists, integers, reals, words, or strings (see section 8.4 'Full strips and character strips') as components. The list is constructed at compile time.

*<list constant>::= [<list constant element *?>]*
*<list constant element>::= <list constant> | <character group>*

The character group should not be [ or ].

Example
*⟨list constant⟩e.g. ::= [1 2 DOG CAT] | [[1 2] [4 5] 6]*

List expressions are evaluated by evaluating a number of expressions at run time and constructing a list.

*<list expression>::= [%< expression sequence>%]*
Thus
*[% %]* is equivalent to *nil*, as is *[]*
and *[%⟨expression⟩%]* is equivalent to *cons(⟨expression⟩, nil)*
and *[%⟨expression⟩, ⟨expression sequence⟩%]* is equivalent to *cons (⟨expression⟩, [%⟨expression sequence⟩%])*

Example
*⟨list expression⟩e.g.::= [%x+1, [%x+2, x+3%], tl(y)%]*

Note that negative numbers are not allowed in list constants. To get a list with negative numbers a list expression must be used.

For convenience the following functions are standard:
*::* (a synonym for *cons* but an operation of precedence 2)
*<>* ϵ list, list=>list (concatenates the lists, copying the first one, an operation of precedence 2)
e.g. *[1 2] <> [3 4]* is *[1 2 3 4]*

*maplist* ϵ *list, (item => item) => list* (applies to each element of a given list a given function constructing a new list of the results.)
e.g. *maplist([1 4 9 16], sqrt)* is *[1.0 2.0 3.0 4.0]*

## 8.4 FULL STRIPS AND CHARACTER STRIPS

Two strip classes are standard.

The first is *Full Strips* with full items as components and associated word *"strip"*. The characteristic functions are:
initiator: *init* ϵ *integer => full strip*
doublet: *subscr* ϵ *integer, strip ==> item*

The second is *Character Strips* (also called *'Strings'*) (for characters, see section 8.6 'Words') with component size 6 and associated word *"cstrip"*. The characteristic functions are:
initiator: *initc* ϵ *integer => character strip*
doublet: *subscrc* ϵ *integer, strip ==> integer of size 6*

The components of a character strip may be any integers of size not more than 6, they need not necessarily be used to represent characters.

There is a structure constant to construct character strip constants at compile time.
*⟨string constant⟩ ::= '⟨string constant element *?⟩'*
*⟨string constant element⟩ ::= ⟨string constant⟩ | ⟨any character except a string quote⟩*

Example
⟨*string constant*⟩*e.g.*::= '...*rubbish. please type* 'sorry'\

Spaces and newlines are significant in string constants.

The function *pr* will output the characters of a string enclosing them in quotes, and the function *prstring* will output them without any quotes (see section 9.2 'Output').

## 8.5 ARRAYS

Arrays give a convenient method of accessing and updating structures indexed by integers. An array has components, which are items of a given size. Each component is associated with a sequence of integers called *Subscripts*. The number of subscripts is known as the number of *Dimensions* of the array. An array is a particular kind of doublet:

*array* ∈ *subscript*, . . . , *subscript* ==> *component*

This is in contrast to strips which have a general selector and a general update routine associated with a whole class of strips and take the actual strip referred to as a parameter. Arrays can be formed from strips (or from other data structures) by using partial application. The programmer is free to do this in any way he chooses but standard functions for creating arrays are provided. For example, to create a one-dimensional array *a* with components indexed from 1 to *n* we could write

*subscr*(% *init*(*n*) %)—>*a*;

There is a standard function to create a many-dimensional array of items of any size. Updating a component of this array does not affect any other component. This function is:

*newanyarray* ∈ *boundslist*, (*subscript*, . . . , *subscript* => *component*),
*strip initiator*, *strip doublet* => *array*

The array produced will normally be immediately assigned to a variable.

The boundslist is a list of integers, these integers being alternately the lower and upper bounds for each subscript. The second parameter is a function used to initialize the components of the array. It must produce the appropriate component for each combination of subscripts. The strip doublet and strip initiator are the characteristic functions of a strip class whose components are of the same size as that required for the array components.

There is also a standard function to create arrays of full items:

*newarray* ∈ *boundslist*, (*subscript*, . . . , *subscript*=> *component*)
=> *array*

This is equivalent to

*newanyarray*(%*inil*, *subscr*%)

The standard function

*boundslist* ∈ *array* => *boundslist*

produces a copy of the boundslist for any array produced by *newanyarray* or *newarray*; for any other function which is a closure it produces the first frozen value.

## 8.6 WORDS

There is a standard record class called *Words*. It has 8 components of size 6 called *Characters*, and a component called the *Meaning*, which may be used to associate any information desired with the word (cf. the use of *fnprops* for functions). The item associated with the record class is *"word"*. The standard functions characterizing words are:

constructor: *consword ∈ character, . . . , character, integer => word*
destructor: *destword ∈ word => character, . . . , character, integer*
selectors: *charword ∈ word, integer => character*
*meaning ∈ word ==> item*

Each character of the POP-2 character set corresponds to a unique integer. The correspondence rule is given in Appendix 1. Note that the constructor and destructor functions above are variadic and variresult respectively and work on a variable number of characters followed by that number as an integer. If there are less than 8 characters supplied to the constructor the remaining character components are not defined and they are not produced by the destructor or selector. The characters are numbered from one up from the left. The constructor does not take a *meaning* component as argument. The *meaning* of a word is *undef* unless the word has been updated to have a particular *meaning*. It is not possible to alter individual characters of a word once it has been created. The standard function *isword* recognizes words
*isword ∈ item => truthvalue*.
When *datalist* (see section 7.2 'Records') is applied to a word it produces a list of its characters.

Words may occur in the program as quoted words, that is, word constants, with the following syntax:

⟨*unquoted word*⟩ ::= ⟨*letter*⟩ ⟨*alphanumeric *?*⟩ | ⟨*sign **⟩
| ⟨*decorated bracket*⟩
| ⟨*separator*⟩ | ⟨*period*⟩ | ⟨*quote*⟩
⟨*decorated bracket*⟩ ::= *(* | *)* | *(%* | *%)*
| *[* | *]* | *[%* | *%]*
⟨*quoted word*⟩ ::= *"*⟨ *unquoted word*⟩*"*

Example
⟨*quoted word*⟩*e.g.* ::= *"big"* | *"++"* | *"%)"* | *""""*

Words may also occur as components of constant lists (see section 8.3 'Lists'). Only the first 8 characters are significant.

Words may also be read as data (see section 9.1 'Input'). Words which occur as constants or are read as data are *Standardized*, that is, if a word with the same characters already exists no new word is constructed and the compound item produced is the previously-existing word, but if no such word exists a new word is constructed with *undef* as its *meaning*. Words constructed using *consword* are also standardized. Thus two words are equal if and only if they have the same characters. In this they are unlike normal records.

## 8.7 FUNCTIONS

Functions are compound items and form a class of data structures. The item associated with the class is *"function"*. There is no constructor or destructor for functions. They can be constructed by the methods described in section 4.1 'Definition of functions'. There is a family of characteristic functions associated with the class of functions to form a class of data structures.

Functions have an accessible component which may be used to associate extra information with the function. It is accessed by the standard doublet

*fnprops* ∈ *function* ==> *item*

Functions have an update routine (see section 4.5 'Doublets'). For a function constructed by using **lambda** or **function** this has initially no defined value. This component may be selected or updated by using a standard doublet:

*updater* ∈ *function* ==> *routine*

It is intended that the updater of a function should normally be a routine for updating a component of a data structure, but any function is permissible as the updater. The updater of a function has its own *fnprops*, *frozval*, *fnpart*, and updater components.

Closure functions, that is, those constructed by partial application, have a doublet to select or update the values of their frozen formals.

*frozval* ∈ *integer*, *closure function* ==> *item*

The integer determines which of the frozen formal values is affected, counting from left to right from 1 upwards (if a closure function is obtained by successive partial applications only the formals frozen by the last one are counted). *datalist* produces a list of the frozen formals. There is also a doublet to select the function from which the closure function was constructed or replace it with another function.

*fnpart* ∈ *closure function* ==> *function*

The standard function = follows the usual rules for compound items when applied to functions, that is, equality is preserved over assignment, updating, and actual parameter/formal parameter correspondence, but each construction of a function produces a different one.

The following standard function recognizes functions:

*isfunc* ∈ *item* => *truthvalue*

# 9. INPUT AND OUTPUT

## 9.1 INPUT

Information which is input to the POP-2 system is organized into *Files*, each of which comes from a *Device*.

Before a file can be accessed it must be *Opened*. From then on it can be read one character at a time. Eventually it must be *Closed*.

The naming of files and devices depends on the operating system of the implementation. The names of files are lists and the names of devices may be any item. A device name may refer to more than one device.

There is a standard variresult function *popmess* used for communicating with the operating system, sending a message in the form of a list.

*popmess* ∈ *list* => *item*, . . . , *item*

This is used for various input and output purposes.

To open a file from a given device, *popmess* is used to produce a repeater function to read characters from it, that is, a function ∈ *()=>character*. Such a function is called a character repeater (see

section 8.3 'Lists'). The list supplied to *popmess* has a head which is an input device name and a tail which is a file name.

To close a file before reaching the end of it, *popmess* is used again. The list supplied to it has a head which is the word *"close"* and a tail which is a list of one element: the character reading function obtained when the file was opened. No result is produced by *popmess* in this case.

The sequence of characters making up a POP-2 text may be split up into *Character Groups*, each of which represents a *Text Item*. A text item is either an integer, a real, a word, or a string. It is represented by a character group, thus:

⟨*character group*⟩ ::= ⟨*integer*⟩ | ⟨*real*⟩ | ⟨*unquoted word*⟩
| ⟨*string constant*⟩

Character groups are followed by spaces or newlines where necessary to separate them from the following character group. That is, a character group should be terminated thus if the first character of the following character group could otherwise be construed as belonging to it, for example, 1 2 is different from 12 but 1 *x* is not different from 1*x*, which must in any case be construed as two character groups 1 and *x*.

There is a standard function to convert a function which produces a character whenever it is applied (character repeater) into a corresponding one which produces a text item whenever it is applied (item repeater).

*incharitem* ∈ *(() => character) => (() => text item)*

The item repeater produced by *incharitem* uses the character repeater. To produce a text item it needs to obtain either one or two characters ahead of those comprising the text item, storing them in a buffer. It always looks one character ahead except when the text item is the integer 2 or 8 and the next character is a colon or the text item is an integer and the next character is a period, when it looks two ahead.

The program is input on a standard file called the *Standard Input File* from a standard device called the *Standard Input Device*. There is a standard function to read characters from the *standard input* file:

*charin* ∈ *() => character*

If the halt code occurs on the input device, *charin* produces the terminator, which is considered here as if it were an extra character. All input character repeaters produced by *popmess* do the same. Item repeaters produced by *incharitem* convert the terminator into itself.

The program is compiled from the text item list which is the value of the standard variable *proglist*. Initially this has as value the list of text items from the standard input file. It is not protected and may be assigned to by the programmer who wishes to compile from a different source.

For convenience there is a standard function *itemread* producing the next item of the list which is the value of *proglist*. If, however, this list contains macro identifiers not preceded by *nonmac* these are applied by *itemread* as they are encountered and *itemread* is again applied to the resulting *proglist* (see section 11.2 'Macros').

*itemread* ∈ *() => text item*

There is also a standard function to compile from any character repeater

*compile* ∈ *character repeater => ()*

## 9.2 OUTPUT

Information which is output from the POP-2 system is organized into files, each of which is sent to a device (see section 9.1 'Input').

A routine with one parameter is called a *Consumer*, for example, if

*c* ∈ *character* => ()

we call *c* a character consumer.

To open an output device the standard function *popmess* (see section 9.1 'Input') is used to produce a character consumer to output characters to it. The list supplied to *popmess* has a head which is an output device name and a tail which is a file name. The device is closed by outputting the halt code or the terminator.

Compiler messages and results of computation are normally output on a standard file called the *Standard Output File* to a standard device called the *Standard Output Device*. There is a standard routine to output characters to the standard output file:

*charout* ∈ *character* => ()

There is a standard variable *cucharout* which contains the routine to output characters to the *Current Output File*. This contains initially the routine for the standard output file but it is not protected and a different consumer may be assigned to it if a different output file is to be made current.

There are standard routines to output spaces or newlines to the current output file:

*sp* ∈ *integer* => ()
*nl* ∈ *integer* => ()

There is a standard item consumer function which outputs any item to this file in some suitable format.

*pr* ∈ *item* => ()

Numbers are printed with a minus sign if they are negative and a space before them if they are positive. Lists are printed with brackets, words without quotes, strings with quotes. For other records and strips the dataword is printed but not the components.

The standard function *print* is defined by

**function** *print x; pr(x); x* **end;**

Applying *pr* to the terminator terminates the output file, by outputting the halt code.

There is a standard function

*prreal* ∈ *real, integer, integer* => ()

which prints a real number with the given numbers of places before and after the decimal point. If both integers are zero the number is printed in exponential form.

There is a standard function

*prstring* ∈ *item* => ()

which prints a string without enclosing it in string quotes.

There is a syntax word => which uses *pr* and causes the items on the stack starting at the bottom to be printed on a newline preceded by two asterisks. These items are removed from the stack. In a function

body only the top item of the stack is affected. This usage is not to be confused with the separator used in function definitions and sections, nor with the => used in this manual to show the type of functions. A semicolon is implied before and after so that immediate evaluation can occur (see section 11.1 'Immediate evaluation').

The standard function

*genout* ∈ *(text item => ()), (character => ()) => (text item => ())*

converts a given text item consumer to a similar one which outputs to a given character consumer. It is defined thus:

```
function genout t c;
    lambda x t cucharout; t(x);
            end (% t, c %)
end;
```

For example, if *disccharout* is a character consumer, we may write

```
genout(prreal, disccharout)->prreal2; prreal2(3.5, 2, 2);
   prreal2(6.7, 2, 2);
```

## 10. MACHINE CODE

It is possible to insert sections of machine code in an imperative sequence. The rules depend on the implementation. A code instruction is represented by the identifier *$* followed by any sequence of character groups which do not include';'

⟨*code instruction*⟩ ::= *$*⟨*any sequence of character groups other than;*⟩

## 11. MODES OF EVALUATION

### 11.1 IMMEDIATE EVALUATION

A POP-2 program consists of a sequence of imperatives and cancellations:

⟨*program element*⟩ ::= ⟨*imperative*⟩ | ⟨*cancellation*⟩ | ⟨*section*⟩
⟨*program*⟩ ::= ⟨*program element*⟩;⟨*program*⟩ | ⟨*program element*⟩;

The program elements are evaluated in sequence in the same way as an imperative sequence. Each program element is evaluated as soon as the terminal semicolon has been read by the compiler. The body of any function in the program element will be compiled and kept so that it may be evaluated when that function is applied. Note that labels, goto statements, and conditionals are not program elements and may occur only in function bodies.

If the compiler detects an error or an error occurs during evaluation the standard function

*errfun* ∈ *item, integer=>()*

is called with an item and an integer indicating the kind of error as arguments. The standard variable *errfun* is not protected and may be assigned to. It prints a message and applies *setpop* (see section 11.3 'Evaluation of program text').

If the error is merely use of an undeclared identifier however, *errfun* is not called, the identifier is automatically declared as a global (outside

all current sections), a warning message is output, and compilation proceeds.

## 11.2 MACROS

A *Macro* is an identifier whose value is a routine which is applied at compile time.

The definition of a macro routine is similar to that of any other routine except that **macro** is used instead of **function** and no formal parameters are allowed. A macro is applied whenever it is mentioned except in a list constant, in a quoted word, after **macro** in a function definition or declaration list element, or when it is preceded by the syntax word **nonmac** in which cases it is treated like an ordinary identifier or word. Thus whenever the syntax allows ⟨*identifier*⟩ it is understood to allow **nonmac** identifier. If **nonmac** is used before an identifier which is not a macro it is simply ignored. In a comment macros are ignored.

Although a macro has no parameters the function *itemread* (see section 9.1 'Input') may be used to read the text items following the macro identifier. There is a standard routine which, when applied in a macro body to a list of text items, concatenates these items to the right of the macro identifier in the program sequence of text items.

*macresults* ∈ *text item list* => ()

If it is applied more than once it concatenates to the right of the previously-inserted items. On exit from the macro the inserted text items are evaluated as program. An identifier used as a macro can be cancelled, for example, by writing

**cancel nonmac** *m;*

Note that *itemread* applies macros (see section 9.1 'Input'), but the functions produced by *incharitem* do not.

Examples

```
macro-->;vars x y;.itemread->x;.itemread->y;
    macresults([% "->", y, "->", x %])
end;
7//2-->r q;
```

This is the same as *7//2->q->r;*

```
macro help;macresults([' go and ask ray \ =>]) end;
help
** 'go and ask ray \
function f;macresults([' go and ask dave \ =>]) end;
    f->nonmac help;
help
** 'go and ask dave \
[help end nonmac] =>
**[help end nonmac]
```

The correspondence between a list of text items and POP-2 program is as follows. Syntax words, identifiers, and unquoted words are represented by corresponding words in the list. A quoted word is represented by a word with the word quote (consisting of the character quote) before and after it in the list. Integers and reals are represented by integers and reals. String constants are represented by strings.

## 11.3 EVALUATION OF PROGRAM TEXT

A standard function *popval* is provided which will evaluate a list of text items treating it as a POP-2 imperative sequence. The sequence is evaluated immediately. It may contain function definitions and assignments to current variables. Any declarations in it which are not in a function body are global. The list must terminate with the word **goon.** For the correspondence between a list of text items and POP-2 program, see section 11.2 'Macros'.

The result of the application of *popval* is the result of evaluating the imperative sequence.

*popval* ∈ *text item list* => *item*, . . . , *item*

Note that *popval* is used to evaluate an imperative sequence at run time and the list of text items may have been produced as the result of computation. It may temporarily affect the standard variable *proglist* (see 9.1 'Input').

Example

```
1—>a;popval([vars x;a+2—>x;x*x goon])=>
**9
```

The standard routine *setpop* may be applied in the imperative sequence. This restores the system to execute mode. The stack is cleared. The variable currently associated with any identifier is not altered, except that *cucharout* and *proglist* are restored to their standard values. After *setpop* has been applied the rest of the imperative sequence is ignored and all function bodies currently being evaluated are abandoned. The system then prints *"setpop"* and evaluates the next program element. *setpop* may also be applied in a function body.

*setpop* ∈ *()* => *()*

If the operating system of the implementation permits it, a program may be interrupted by a signal from the console. This has the effect of requiring a text to be input from the standard device, ending with the word **goon.** The effect is as if the statement *popval([. . . ]);* had been inserted before the next backward goto statement or function entry, where '. . . ' represents the text input, except that *cucharout* and *errfun* are reset to the standard values during the interrupt and reinstated at the end of the interrupt unless their values have been changed, and that the pre-existing stack is not available to the user during the interrupt.

## APPENDIX 1: STANDARD POP-2 CHARACTER SET

| Decimal | Octal | Character | Decimal | Octal | Character |
|---|---|---|---|---|---|
| 0 | 00 | 0 | 32 | 40 | |
| 1 | 01 | 1 | 33 | 41 | A |
| 2 | 02 | 2 | 34 | 42 | B |
| 3 | 03 | 3 | 35 | 43 | C |
| 4 | 04 | 4 | 36 | 44 | D |
| 5 | 05 | 5 | 37 | 45 | E |
| 6 | 06 | 6 | 38 | 46 | F |
| 7 | 07 | 7 | 39 | 47 | G |
| 8 | 10 | 8 | 40 | 50 | H |
| 9 | 11 | 9 | 41 | 51 | I |
| 10 | 12 | : | 42 | 52 | J |
| 11 | 13 | ; | 43 | 53 | K |
| 12 | 14 | < | 44 | 54 | L |
| 13 | 15 | = | 45 | 55 | M |
| 14 | 16 | > | 46 | 56 | N |
| 15 | 17 | 10 | 47 | 57 | O |
| 16 | 20 | space | 48 | 60 | P |
| 17 | 21 | newline | 49 | 61 | Q |
| 18 | 22 | " | 50 | 62 | R |
| 19 | 23 | HALT CODE | 51 | 63 | S |
| 20 | 24 | £ | 52 | 64 | T |
| 21 | 25 | % | 53 | 65 | U |
| 22 | 26 | & | 54 | 66 | V |
| 23 | 27 | ' | 55 | 67 | W |
| 24 | 30 | ( | 56 | 70 | X |
| 25 | 31 | ) | 57 | 71 | Y |
| 26 | 32 | * | 58 | 72 | Z |
| 27 | 33 | + | 59 | 73 | [ |
| 28 | 34 | , | 60 | 74 | $ |
| 29 | 35 | - | 61 | 75 | ] |
| 30 | 36 | . | 62 | 76 | ↑ |
| 31 | 37 | / | 63 | 77 | shift |

### NOTES

The halt code (decimal 19) is used by the system for file termination.

The shift character (decimal 63) may be used by the user as a special purpose marker, as no out-shift facilities are envisaged. If printed, this character will be output as a space.

The character set is based on the ISO standard one.

## APPENDIX 2: OPTIONAL FUNCTIONS

The table below gives identifiers and definitions for a number of functions which are not standard but optional. That is, implementations are not bound to include them but may do so, either as part of the system or in the program library. They are described here so that if they are provided by an implementation they may have the same identifier and definition as in other POP-2 implementations.

Two further suggestions are made for diagnostic purposes, again optional and not mandatory: (1) that the values of variables on declaration (not defined in the manual) should be a pair whose *front* is the

word corresponding to the identifier of the variable and whose *back* is *undef*, (2) that the *fnprops* of a function introduced by a function definition should be a similar pair but with the *back* equal to *nil*.

Example
**function** *sort l;* ... **end;**
*fnprops(sort)=>*
***[sort]*

*apply* — This applies a function. It is defined by
**function** *apply f;f()* **end;**
**lambda** *f;—> f()* **end** —> *updater(apply)*

*fncomp* — This gives the composition of two functions. It is defined by
**function** *fncomp f g;***lambda** *f g; .f.g* **end** *(% f,g %)* **end**
It is an operation of precedence 2. Thus *x.(f fncomp g)* is *x.f.g*

*valof* — A function which, given a word, produces the value of the word when it is used as an identifier or changes this value. It is defined by using *popval*.
*valof ∈ word => item*
e.g.
*5—>x; "x"—>a; valof(a)=>*
***5*
*3—>valof(a); x=>*
***3*

*cos* — A function which, given a number, produces the cosine of that number (in radians).
*cos ∈ number => real*

*sin* — A function which, given a number, produces the sine of that number (in radians).
*sin ∈ number => real*

*tan* — A function which, given a number, produces the tangent of that number (in radians).
*tan ∈ number => real*

*arctan* — A function which, given a number, produces the inverse tangent of that number (in radians).
*arctan ∈ number => real*

*log* — A function which, given a number, produces the natural logarithm of that number.
*log ∈ number => real*

*exp* — A function which, given a number, produces the exponential function of that number. $(exp(x)=e^x)$.
*exp ∈ number => real*

*sqrt* — A function which, given a number, produces the square root of that number.
*sqrt ∈ number => real*

*applist* — A function which, given a list and a function, applies the function to each member of the list. *applist(l,f)* applies the function *f* to each member of the list *l*.
*applist ∈ list,function => ()*
e.g.
*applist([1 2 3],pr);*
*1 2 3*

*appdata* — A function which, given a record or strip, applies a given function to each component of it, that is, *appdata(x,f)*

has the same effect as *applist(datalist(x),f)* but may be more efficient

*appdata ∈ record or strip,(item => ()) => ()*

*rev* — A function to reverse a list (at the top level). The original list is copied.

*rev ∈ list => list*

e.g.

```
rev([1 2 3])=>
***[3 2 1]
rev([1[1 2 3]])=>
**[[1 2 3]1]
```

*copylist* — A function to copy a list (*copy* just copies the first pair) e.g.

```
copylist([1 2 3])=>
**[1 2 3]
```

*length* — A function which, given a list, produces the number of items in that list (at the top level).

*length ∈ list => integer*

e.g.

```
length([1 2 3])=>
**3
length([1[1 2 3]])=>
**2
```

*/=* — Not equal. An operation of precedence 7.

```
/= ∈ item, item=>truth value
operation 7 / = x y; not(x=y) end
```

*equal* — A function to test the equality of two lists to all depths, by testing the equality of corresponding members of the two lists.

*equal ∈ list, list => truth value*

e.g.

```
equal([1[a]], 1::[[a]])=>
** 1,
equal([1 2], [2 1])=>
** 0,
```

*library* — A function which, given a list (the name of a file in the library), produces a character repeater to access that file.

*library ∈ list => character repeater*

e.g.

*library([sets])—>a; compile(a);*

compiles the library file *[sets]*.

*prbin* — A function which, given an item, prints it as a binary number of a given number of bits.

*prbin ∈ item, integer => ()*

*proct* — A function which, given an item, prints it as an octal number of a given number of octal digits.

*proct ∈ item, integer => ()*

*carryon* — A function which can be used to continue compilation of a file after an error.

*carryon(a)* causes the file whose character repeater is *a* to be searched for the word **end** and then compiled.

*carryon ∈ character repeater => ()*

*listread* — A function of no arguments which reads the next list from *proglist*.

*listread ∈ () => list*

*numberread* A function of no arguments which reads the next number (signed integer or signed real) from *proglist*.
*numberread* $\in$ *() => number*

*coreused* A variable whose value is the number of words of core currently in use by the program.

*poptime* A variable whose value is the amount of processor time which has been used by the program.

## APPENDIX 3: CHANGES TO THE REFERENCE MANUAL WHICH AFFECT THE LANGUAGE MADE SINCE THE PREVIOUS EDITION

2.2, 2.3 All standard functions with identifiers starting *int* or *real*, except *intof* and *realof* are abolished. The ordinary arithmetical operations, + etc., should be used instead of *intadd*, *realadd*, and so on. *intsign* and *realsign* are both replaced by the single function *sign*.

2.6 The value of the variable *termin* is no longer *"termin"* but a special word which may not be read or printed.

3.2 (a) Restriction of identifiers to functions is abolished except for operations. The word **function** is no longer to be used in declarations. The word **macro** may be.

(b) *unique* and *nonunique* abolished, all identifiers are now non-unique (but see 3.4 below for alternative facility).

(c) A standard function *identprops* is introduced to give the properties of identifiers.

3.4 *Sections* (inserted to give new facility which will enable identifier clashes to be avoided).

4.1 (a) The syntax mistakenly insisted on => after formal parameters even if there was no output local list. This is corrected. The output local list was mistakenly allowed only in lambda expressions and not in function definitions. This is corrected.

(b) The word **routine** is abolished. **function** may be used instead.

(c) **operation** ⟨*integer*⟩ may be used instead of **function** in function definitions.

4.2 A standard function *stacklength* $\in$ *() => integer* is introduced to give the number of items on the stack.

4.6 The results of // are now produced in the opposite order. Thus *m//n—>q—>r*.

5.1 Expressions with a compound operator are now allowed, for example *diff(sin)(0.5)* and *x.(f(g))*.

5.4 A standard function *jumpout* is introduced to allow one to jump out of a function body, interrupting its execution (since this function is irregular in its behaviour some existing systems may not implement it immediately). **switch** is provided.

5.5 Destination expressions with a compound operator are now allowed.

5.6 A comment is a statement and is made of text items, not of any characters.

6.1 (a) In a conditional the consequent was evaluated if the condition was *true*, now it is evaluated unless the condition is *false*.

(b) **if** may be replaced by **loopif** which causes a jump back as soon as a consequent has been evaluated.

(c) A standard macro *forall* is introduced to facilitate writing some simple loops.

7.1 (a) The number of components of a record or strip may be zero.
(b) A standard function *samedata* is introduced to test whether two items are of the same data class.

7.2 (a) The size of a component of a record or strip may be *"compnd"*, restricting it to compound items.
(b) *dataword* is a doublet and its value may be any item, instead of just a word. It is defined for functions, integers, and reals, as well as for records and strips.
(c) *enddata* has been discarded.
(d) *recordfns* now has only two arguments, not three.

7.3 (a) A standard function
*datalength* ∈ *record* => *integer*
is introduced to give the number of items in a strip.
(b) The components of a strip are numbered from 1 upwards (this was previously not specified).
(c) *stripfns* now has only two arguments, not three.

7.4 *delitem* has been discarded.

8.3 (a) The standard function *dest* now has no side-effect on a dynamic list and its definition has been altered to simply produce the head and the tail.
(b) The functions *solidified* and *null* have been redefined to deal correctly with dynamic null lists.
(c) *maplist* has been added as a standard function, for example,
*maplist([1 4 9 16], sqrt) = [1.0 2.0 3.0 4.0]*
(d) Standard link and list recognition functions, *islink* and *islist*, are introduced.

8.4 A standard function *prstring* is introduced which prints strings without any quotes.

8.5 A standard function *boundslist* is introduced to give the boundslist of an array.

8.6 (a) The destructor for a word was said to produce an item. This was a mistake. It produces only some characters and an integer.
(b) Instead of doublets for a word there is a single selector function *charword* (no updater)
*charword* ∈ *word, integer=>character*

9.2 (a) It is specified that the item repeater produced by *incharitem* has a buffer of either one or two characters.
(b) A function *prreal* is defined to print reals.
(c) A function *genout* is defined to make it easier to output to different devices.

10. The action to be taken on encountering an error is now specified. The standard function used is
*errfun* ∈ *integer, item => ()*

11.1 Programs may now include sections (new facility described in 3.4 above).

11.2(a) It is made clear that being a macro is a property of an identifier rather than of the associated routine.
(b) A macro identifier may be prefixed by the new syntax word **nonmac** which prevents it being activated as a macro on that occasion. This allows, for example, assignment of a routine to a macro identifier. Otherwise macros are expanded wherever they are encountered, except inside quotes or square brackets.

11.3 A facility for interrupting a program from the console is described. The effect is as if *popval([...])* had been inserted in the program at that point, where "..." is the text typed in.