POP-2 REFERENCE MANUAL

BY
R. M. BURSTALL
AND
R. J. POPPLESTONE
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

CONTENTS

1. INTRODUCTION

1.1. Aims

The following are the main design objectives for the POP-2 language:

(*i*) The language should allow convenient manipulation of a variety of data structures and give powerful facilities for defining new functions over them.

(*ii*) The language should be suitable for taking advantage of on-line use at a console, i.e. it should allow immediate execution of statements and should have a sufficiently simple syntax to avoid frequent typing errors.

(*iii*) A compiler and operating system should be easy to write and should not occupy much storage.

(*iv*) The elementary features of the language should be easy to learn and use.

(*v*) 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:

(*vi*) Fast arithmetical facilities on integer and real numbers.

(*vii*) Fast subscripting of arrays.

(*viii*) A wide variety of elegant syntactic forms.

Naturally whether (*iii*) or (*vi*) and (*vii*) 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:

(*i*) Variables (cf. ALGOL but no types at compile time).

(*ii*) Constants (cf. ALGOL numeric and string constants, LISP atoms and list constants).

(*iii*) Expressions and statements (cf. ALGOL).

(*iv*) Assignment (cf. ALGOL, also CPL left-hand functions).

(*v*) Conditionals, jumps and labels (cf. ALGOL but restrictions on jumps and labels).

(*vi*) Functions (cf. ALGOL procedures but no call by name, cf. CPL and ISWIM for full manipulation of functions).

(*vii*) Arrays (cf. ALGOL; cf. CPL for full manipulation of arrays).

(*viii*) Records (cf. COBOL, PL/1, Wirth-Hoare ALGOL records, CPL nodes).

(*ix*) Words (cf. LISP atoms).

(*x*) Lists (cf. LISP, IPL-V).

(*xi*) Macros.

(*xii*) Use of compiler during running (cf. LISP, TRAC, FORMULA ALGOL).

(*xiii*) Immediate execution (cf. JOSS, TRAC).

Notes:

LISP: LISP 1.5

CPL: See Barron, D. W., *et al.* 1964. The main features of CPL, *Computer J.*, **6**, 134-43.

CPL reference manual. Edited C. Strachey (privately circulated).

Wirth-Hoare ALGOL: See Wirth, N., and Hoare, C. A. R. 1966. A contribution to the development of Algol, *Communs Assn Comput. Mach.*, **9**, 413-32.

TRAC: See Mooers, C. N. 1966. TRAC, a procedure describing language for the reactive typewriter, *Communs Assn Comput. Mach.*, **9**, 215-24.

ISWIM: See Landin, P. J. 1966. The next 700 programming languages, *Communs Assn Comput. Mach.*, **9**, 157-166.

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", 500, [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]
routine 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, $n \geqslant 1$;
? means that a class may occur n times, $n=0$ or 1;
*? means that a class may occur n times, $n \geqslant 0$,

e.g. 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 **heavy type** without any change of meaning. It will be conventional however to use **heavy type** letters for syntax words, i.e. those identifiers such as **function, then, end** and **cancel** which have a special meaning for the POP-2 compiler and which characterise certain syntactic forms.

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

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, heavy type or not is regarded as the same character in the reference language.

The symbols → and ⇒ used in this paper should be read as a typographical abbreviation for the pairs of characters $-$ > (minus greater than) and = > (equals greater than) respectively.

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 $d1, d2, \ldots, dm$ and $r1, r2, \ldots, rn$ are all sets of items. Then $d1, d2, \ldots, dm \Rightarrow r1, r2, \ldots, rn$ is the set of all functions whose domain is $d1, d2, \ldots, dm$ and range $r1, r2, \ldots, rn$, i.e. with arguments which are m-tuples in $d1 \times d2 \times \ldots \times dm$ and with results which are n-tuples in $r1 \times r2 \ldots \times rn$. We express the fact that a function f is a member of this set of functions by

$$f \in d1, d2, \ldots, dm \Rightarrow r1, r2, \ldots, rn$$

Some examples will make this clear.

add ∈ *integer, integer* ⇒ *integer*
divrem ∈ *integer, integer* ⇒ *integer, integer*

where *divrem* is 'divide with remainder', e.g. *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 ∈ ... ⇒ ...

and

g ∈ ... ⇒ ...

.....

and

h ∈ ... ⇒ ...

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 are called *Items*. They are divided into two distinct classes: *Compound* items, which are represented by addresses and *Simple* items which are directly represented by bitstrings which do not contain addresses (these bit strings are normally of fixed length for a given implementation, being a single machine word). The address representing a compound item points to a bit string whose length may vary from item to item. This bit string may contain other items. The areas of store immediately pointed to by two different compound items do not overlap.

The following standard function recognises compound items:

iscompnd ∈ *item* ⇒ *truthvalue*

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

isinteger, isreal all ∈ *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 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 following functions on integers are standard:

intadd, intsub, intmult, all ∈ *integer, integer* ⇒ *integer*
// ∈ integer, integer ⇒ *integer, integer*
intplus, intminus all ∈ *integer* ⇒ *integer.*
intsign ∈ *integer* ⇒ *integer*
intgr, intle, intgreq, intleeq all ∈ *integer, integer* ⇒ *truthvalue*

intadd, intsub, intmult and *intdiv* are the usual add, subtract and multiply. *//* is divide with remainder and produces a quotient and a remainder (if $a//b$ is (q, r), then $q*b+r=a$ and $0 \leqslant r < b$). It is an operation of precedence 4.

intplus carries an integer into itself and *intminus* complements an integer. *intsign* produces -1, 0, or $+1$ according to the sign of the integer. The remaining four functions are the relations 'greater than', 'less than', 'greater than or equal to' and 'less than or equal to'.

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

Integers may also be treated as bit-strings (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).

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 following functions on reals are standard:

realadd, realsub, realmult, realdiv all ∈ *real, real* ⇒ *real*
realplus, realminus all ∈ *real* ⇒ *real*
realsign ∈ *real* ⇒ *integer*
realgr, realle, realgreq, realleeq all ∈ *real, real* ⇒ *truthvalue*

These are the usual add, subtract, multiply and divide on reals. *realplus* carries a real into itself and *realminus* complements a real. *realsign* produces -1, 0 or $+1$ according to the sign of the real. The remaining four functions are the relations 'greater than', 'less than', 'greater than or equal to' and 'less than or equal to'.

There are also operations to convert a real to the nearest integer and to convert an integer to real:

intof ∈ *real* ⇒ *integer*
realof ∈ *integer* ⇒ *real*

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

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 propositiona 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 the word *"termin"* as its value on entry to the POP-2 system (*see* section 8.6 'Words'). It may be used as the first argument of a variadic function (*see* section 4.2 'Application of functions') or to mark the end of an input file (*see* section 9.1 'Input').

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 restricted to a certain range of values and it may be given special syntactic properties by being given a precedence (*see* section 5.2 'Precedence').

The syntax of identifiers is:

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

Example:

⟨*identifier*⟩*e.g.* ::=*x* | *y*99 | *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 initialisation

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 *Initialisation* 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 initialisation may also specify that the identifier is restricted to take only functions as values. This is not necessary but may make the implementation more efficient. A declaration or initialisation may also specify that the identifier is an *Operation,* i.e. it is restricted to take functions as its values and is given a precedence. This restriction is associated with the unique name (see below) produced by the declaration or the initialisation.

The syntax of declarations is:

⟨*declaration*⟩::=**vars** ⟨*declaration list element* *⟩
⟨*declaration list element*⟩::=⟨*identifier*⟩ | ⟨*restriction*⟩
⟨*restriction*⟩::=⟨*restrictor*⟩⟨*identifier*⟩ | ⟨*restrictor*⟩(⟨*identifier* *⟩)
⟨*restrictor*⟩::=**function** | **operation** ⟨*integer*⟩

Example:

⟨*declaration*⟩*e.g.*::=**vars** *x y* | **vars** *x y* **function**(*f g*) **operation** 7 ==

A declaration or initialisation 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 initialisation of the identifier.

The scope of a global declaration starts at the declaration and continues until the identifier is cancelled.

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 initialisation is the body of the function in which it occurs.

Each declaration or initialisation gives rise to a unique mark and this mark is associated with all occurrences of any identifier introduced by the declaration or initialisation within the scope of the declaration or initialisation. An identifier together with its unique mark is called a *Unique name.*

Thus an identifier which occurs in more than one declaration or initialisation corresponds to more than one unique name.

The generation of fresh unique names for identifiers can be suppressed by using the standard routines:

nonunique, unique all ∈ () ⇒()

If *nonunique* is applied, all declarations or initialisations of a given identifier until *unique* is applied will give rise to the same unique name. This may save storage space and can be used when no confusion is liable to occur.

To sum up:

A new identifier is introduced by introducing a fresh sequence of characters.

A new unique name is introduced by each declaration or initialisation (unless *nonunique* has been applied).

A new variable is introduced by each dynamic activation of a declaration or initialisation.

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 initialised and continues until exit from the body. During this extent the extent of any other variable with the same unique name 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 unique name during any evaluation. Other variables associated with the unique name are in abeyance.

More than one global declaration of the same identifier is not permitted unless a cancellation of it intervenes in the text.

Similarly a declaration of a local variable is not permitted if there is already a declaration of a local or initialisation 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. Certain standard variables are *Protected*, i.e. no assignment may be made to them.

3.3. Cancellation

A cancellation terminates the scope of any declaration of an identifier and removes the effect of any restrictions placed upon 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* ∗⟩

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 a *Formal Parameter List* which is a list of identifiers of formal variables, possibly an *output local list* which is a list of the 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 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 element*⟩::=⟨*ident*⟩ | ⟨*restriction*⟩
⟨*formal parameter list*⟩::=⟨*formal parameter list element* ∗?⟩
⟨*output local list element*⟩::=⟨*ident*⟩ | ⟨*restriction*⟩
⟨*output local list*⟩::= ⇒ ⟨*output local 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**

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*⟩⟨*formal parameter list*⟩; ⟨*function body*⟩ **end** → ⟨*identifier*⟩

This is so common that a special syntactic form is introduced which is equivalent to it:

⟨*function*⟩::=**function** | **routine**
⟨*function definition*⟩::=⟨*function*⟩⟨*identifier*⟩⟨*formal parameter list*⟩;
⟨*function body*⟩ **end**

The word **routine** is a synonym for **function**. It may be used for a function with no results.

If the identifier 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. The identifier may be an operation identifier.

Example:

⟨*function definition*⟩*e.g.*::=**function** *max x y*; **if** $x > y$ **then** *x* **else** *y* **close**
end
| **routine** *enter u v*; *cons*(*conspair*(*u*, *v*), *dict*)→
dict **end**
| **function** *order x y* ⇒ *u v*;
if $x > y$ **then** $x \rightarrow u$; $y \rightarrow v$
else $y \rightarrow u$; $x \rightarrow v$ **close**
end

4.2. Application of functions

An *n-Tuple* is an ordered sequence of *n* items ($n \geqslant 0$). An item is identical with the 1-tuple whose sole member is that item. An *n*-tuple and an *m*-tuple may be *Concatenated* to produce an $(n+m)$-tuple.

A function of *n* arguments (i.e. with *n* formal parameters, excluding frozen formals; *see* section 4.4 'Partial application'), may be *Applied* to an *n*-tuple, whose members are called the *Actual Parameters* of the function. Application of a function to its actual parameters produces an *m*-tuple, whose members are said to be the *Results* of the function. A function producing no results (i.e. an 0-tuple) is called a routine (*see* section 4.1 'Definition of functions').

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 is initialised to the corresponding actual parameter value, or if it is a frozen formal to the corresponding value in the frozen value list. A new variable corresponding to each local variable declaration in the function body but not in any interior function body is then created. 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. Any items which have been placed on the stack (*see* section 5.3 'Statements and imperatives') and were not there at entry are concatenated with the values of any *Output Local Variables* to 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 reinitialised from the frozen value list on the next entry to the function normally with the same values as last time; the frozen value list 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 (i.e. declared in the body) or formals (i.e. elements of the formal parameter list) 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 initialisation') or outside their extent. Mention of the identifier of such a nonlocal would refer to a quite different variable currently associated with that unique name. The difficulty can arise for recursive functions. Analogous trouble may arise if *nonunique* is used.

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 initialised by partial application to the value that the non-local would have taken. (Note that the frozen formals can be used in this way to give the equivalent of CPL fixed functions, *see CPL Reference Manual* privately circulated by C. Strachey, Programming Research Unit, Oxford University.) In cases where assignment to the nonlocal is desired a frozen formal can be used and initialised 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.

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 initialised 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 Value List.*

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.

More formally we say that a function f of m arguments may be partially applied to an n-tuple of actual parameters with $n \leqslant m$. 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 a frozen value list 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 value list.

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 a particular frozen value list, 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 (*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 application. It is similar to that for ordinary application (*see* section 5.1 'Expressions').

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

The value of the variable currently associated with the identifier is partially applied to the concatenation of the expressions in the expression list. 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)⇒
```

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 quasi compound expression (i.e. as part of a destination of an 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

$$f \in d1, \ldots, dk ==\Rightarrow r$$

meaning that f has a selector s

$$s \in d1 \ldots, dk \Rightarrow r$$

and an update routine u

$$u \in r, d1, \ldots, dk \Rightarrow ()$$

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

In sections 2.2 'Integers' and 2.3 'Reals' a number of standard functions were introduced for performing arithmetic on integers and reals.

We say that an item is a *Number* if it is either a real or an integer. Arithmetic on numbers is performed by the following standard operations:

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
↑	3	exponent	real

These are defined in terms of *intadd*, *realadd*, etc. and *isreal*, *isint* and *realof*. +, − and * produce an integer result if both arguments are integer otherwise a real result.

5. EXPRESSIONS AND STATEMENTS

5.1. Expressions

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

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 list. 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 case where the number of arguments required by the function is not equal to the number of items obtained by evaluating the operands is dealt with in section 5.3 'Statements

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

An expression list is evaluated by evaluating the expressions of which it consists and concatenating the results. The order in which the evaluations are made is not defined. The order in which the results of evaluating the expressions are concatenated 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*⟩
⟨*parentheses*⟩::=(|)
⟨*compound expression*⟩::=⟨*non-operation identifier*⟩ (⟨*expression list*⟩)
|⟨*expression* ?⟩⟨*operation*⟩⟨*expression* ?⟩
|⟨*closed expression*?⟩⟨*dot operator**⟩
| ⟨*structure expression*⟩
⟨*closed expression*⟩::=⟨*simple expression*⟩ | ⟨*list expression*⟩
| ⟨*conditional expression*⟩
⟨*dot operator*⟩::= .⟨*non-operation identifier*⟩
⟨*structure expression*⟩::=⟨*partial application*⟩ | ⟨*list expression*⟩
⟨*expression list*⟩::=⟨*expression*?⟩ , ⟨*expression list*⟩ | ⟨*expression* ?⟩
⟨*expression*⟩::=⟨*simple expression*⟩ | ⟨*compound expression*⟩
| ⟨*conditional expression*⟩ | ⟨*imperative expression*⟩
| (⟨*expression list*⟩)

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*) | *a**(*b*+*c*) | *x.hd*
| *f*(% *x* %) | [% *x*, *x*+1, *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:

(*i*) ⟨*non-operation identifier*⟩ (⟨*expression list*⟩). Here the operator is the identifier and the operands are the expression list.

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

(*iii*) ⟨*closed expression* ?⟩. ⟨*non-operation identifier*⟩. This is equivalent to:
⟨*non-operation identifier*⟩ (⟨*closed expression* ?⟩) which is a special case of (*i*) above.

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

Note that there is no syntactic provision above for compound expressions whose operator is an expression other than an identifier.

5.2. Precedence

If a compound expression or quasi compound expression 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 analysed 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 7 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 which is restricted to having function values is purely a syntactic one.

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 but does not remove restriction of its values to functions. 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 machine code instruction or an expression list. It may be labelled.

An imperative is either a declaration or a statement.
The syntax is:

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

Example:

⟨*imperative sequence*⟩*e.g.*::=*loop*:$x-1\rightarrow x$; $f(x)\rightarrow y$; **if** $x>0$ **then goto** *loop*;
| $x+1\rightarrow y$; y; $u\rightarrow y$;$\rightarrow z$;

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*⟩)

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 the top of the stack or removed from the top of the stack. On entry to the POP-2 system the stack is empty. When a statement is evaluated any results produced are added to the top of the stack. The results of an imperative sequence are the items left on the stack when the sequence has been evaluated.

Evaluation of a statement which is a compound expression may affect the stack as follows. If the number of arguments required by the function obtained by evaluating the operator is not the same as the number of items produced by evaluating the operands, these items are loaded on to the stack in sequence. The function then takes its arguments off the stack, the last argument being the one which was on the top of the stack. Thus suppose that the function requires m arguments and the operands yield n items. If $m>n$ the first $m-n$ arguments are taken off the stack. If $m<n$ the first $n-m$ items produced by the operands are left on the stack. If $m=n$ the stack is not affected by evaluating the compound expression. Exactly analogous remarks apply to quasi compound expressions.

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 quasi compound expression it may not refer to a label outside that operand. The syntax is:

⟨*labelled statement*⟩::=⟨*label*⟩:⟨*statement* ?⟩
⟨*goto statement*⟩::=**goto** ⟨*label*⟩ | **return**
⟨*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 an identifier or sign 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.

Example:

loop: $x+1\rightarrow x$; $y*y\rightarrow y$;
if $x=0$ **then goto** *loop* **close.**

5.5. Assignment

An *Assignment* consists of a *Source*, which is an expression or sequence of expressions and a *Destination List*, which is a sequence of elements each of which is either an identifier or a *Quasi Compound Expression.*

A quasi compound expression has an operator which is an expression and some operands, i.e. a sequence of expressions (possibly an empty sequence). Note that a quasi compound expression is not an expression and cannot be evaluated alone to produce an item; it is merely a component of an assignment. It is syntactically the same as a compound expression but cannot be evaluated in isolation.

An assignment is evaluated as follows. First the source is evaluated to yield an n-tuple (where $n\geqslant k$, k being the number of destination elements). The last k elements of this n-tuple, 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:

(*i*) If the destination element is a variable the source item becomes the new value of that variable.

(*ii*) If the destination element is a quasi compound 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 to the concatenation of the source item and the values of the operands.

The syntax of quasi compound expression is given below. A further explanation of this syntax is given in section 5.2 'Precedence'.

⟨*quasi compound expression*⟩::=⟨*non-operation identifier*⟩ (⟨*expression list*⟩)

| ⟨*expression* ?⟩ ⟨*operation*⟩ ⟨*expression* ?⟩
| ⟨*closed expression* ?⟩
⟨*dot operator* * ?⟩

The syntax of assignments is:

⟨*assignment*⟩::=⟨*expression list*⟩ ⟨*destination* *⟩ | ⟨*function definition*⟩
| ⟨*macro definition*⟩
⟨*destination*⟩::= → ⟨*non-operation identifier*⟩ | → ⟨*quasi compound expression*⟩

Example:

⟨*assignment*⟩*e.g.*::=$x+1 \rightarrow y \mid u+v \rightarrow a(i,j) \mid x//y \rightarrow u \rightarrow v$

In the second example $a(i,j)$ is a quasi compound expression and the whole assignment is a euphemism for $a1(u+v, i, j)$ where $a1$ is the update routine of the doublet a.

Function definitions and macro definitions are special syntactic forms for assignments.

5.6. Comments

The word **comment** and all characters after it up to and including the next semicolon are ignored.

6. CONDITIONALS

6.1. Conditional expressions

A conditional expression is composed of three components which we will call the *Condition*, the *Consequent* and the *Alternative*. The condition is an expression with a single result, a conjunction or a disjunction (*see* section 6.2 'Conjunctions and disjunctions'). The consequent and the alternative are imperative sequences each having the same number of results. The method of evaluation of a conditional expression is as follows:

The condition is first evaluated. If its value is the truth value *true* then the consequent is evaluated and its value becomes the value of the expression. But if the value of the condition is the truth value *false* then the alternative is evaluated and its value becomes the value of the expression.

The alternative of a conditional expression may be omitted if it is an empty imperative sequence.

It will often happen that the alternative is itself a conditional expression. The syntax of conditionals is arranged to provide a compact notation to express this:

⟨*conditional body*⟩::=⟨*imperative sequence*⟩
⟨*conditional expression*⟩::=**if** ⟨*condition*⟩ **then** ⟨*conditional body*⟩
⟨*elseif clause* * ?⟩ ⟨*else clause* ?⟩ **close**

⟨*elseif clause*⟩::=**elseif** ⟨*condition*⟩ **then** ⟨*conditional body*⟩
⟨*else clause*⟩::=**else** ⟨*conditional body*⟩

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** $1\rightarrow y$ **close**

If there are no elseif clauses the conditional body is the consequent and the else clause is the alternative (which may be omitted). If there are elseif clauses then the first expression is the condition, the second is the consequent and the remainder is the alternative, and it is to be regarded as the conditional expression obtained by replacing the first **elseif** by **if** and inserting an extra **close** before the **close**, e.g.

if p **then** x **elseif** q **then** y **else** z **close**

is equivalent to

if p **then** x **elseif** q **then** y **else** z **close 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 has the truth value *false* the value of the conjunction has truth value *false*, otherwise the second expression is evaluated and the conjunction has a truth volue equal to that of the second component expression.

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 has the truth value *true* the value of the disjunction has truth value *true*, otherwise the second expression is evaluated and the disjunction has a truth value equal to that of the second component expression.

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.

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, i.e. 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, i.e. given a data structure it produces its components as results.

d ∈ *data structure* ⇒ *component*, ... , *component*

After applying the destructor to a structure, the structure is deleted.

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 '*E*1' is equal to the value of an expression '*E*2' then the expression

*E*1 = *E*2

has value *true*.

Equality means that the two compound items contain the same address, i.e. 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, i.e. it is

(*i*) reflexive ($x=x$);
(*ii*) symmetric (if $x=y$ then $y=x$); and
(*iii*) transitive (if $x=y$ and $y=z$ then $x=z$).

It has the following other properties:

(*iv*) The value of a formal parameter variable is equal to the corresponding actual parameter.

(*v*) If an item is assigned to a variable then the value of the variable is equal to that item.

(*vi*) 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.

(*vii*) The rules for equality of words are given in section 8.6 'Words'.

(*viii*) 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, $x1, \ldots, xi, \ldots, xk$ for items occurring as components, $s1, \ldots, si, \ldots, sk$ for selectors, $u1, \ldots, ui, \ldots, uk$ for update routines, c for a constructor and d for a destructor.

(*i*) $s1(a), \ldots, sk(a)$ is the same n-tuple as $d(a)$, i.e. they have equal elements.

(*ii*) $si(c(x1, \ldots, xi, \ldots, xk))=xi$ is true.

(*iii*) $c(s1(x), \ldots, sk(x))=x$ is always false, but the left-hand expression is equivalent to x.

(*iv*) After evaluating $ui(xi, a)$,

$si(a)=xi$ is true.

(*v*) After evaluating $ui(si(a), a)$, a is unchanged.

(*vi*) If $a=b$ then $ui(xi, a)$ is evaluated,

$si(b)=xi$ is true and $a=b$ is still true.

(*vii*) From (*i*) and (*ii*) above $d(c(x1, \ldots, xk))$ is the same n-tuple as $x1, \ldots, xk$, i.e. they have equal elements.

If a and b are data structures and a is not equal to b and updating a com-

ponent 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 ∈ *integer, data structure* ⇒ *component*

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

u ∈ *component, integer, data structure* ⇒ ( )

Thus if *ui* is the *i*th update routine, $u(xi, i, a)$ has the same effect as $ui(xi, a)$.

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'.

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 characterise a new family of data structures. Suppose for example that we have a class of structures with two selectors, say *s*1 and *s*2, and components which are full items and members of the same class of structures. We can then define a new class of structures whose selectors are given by:

function *s*11 *a*; *s*1(*s*1(*a*)) **end**; **function** *s*12 *a*; *s*1(*s*2(*a*)) **end**;
function *s*21 *a*; *s*2(*s*1(*a*)) **end**; **function** *s*22 *a*; *s*2(*s*2(*a*)) **end**;

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

function *cc x*1 *x*2 *x*3 *x*4; *c*(*c*(*x*1, *x*2), *c*(*x*3, *x*4)) **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.

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 item. If all the items in the set are restricted

to be non-negative integers less than 2^n, the size is the integer n, otherwise if the component is a *Full Item* (i.e. 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. A word is associated with each record class.

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 word to be associated with the record class, an estimate of the number of records in the record class (this is purely to help in efficient implementation) and the specification of the record class. It produces the constructor, the destructor and the doublets for the record class. 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 ∈ *word*, *integer*, *specification* ⇒ *constructor*, *destructor*, *doublet*, ... , *doublet*

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

datalist ∈ *record* ⇒ *list*

There is a function *dataword* which given a record produces the word associated with its record class.

dataword ∈ *record* ⇒ *word*

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 three functions are extended to deal with records of that class.

The routine *enddata* may be given the word associated with a record class and removes all records in that class. It also adjusts the three functions just mentioned so that they no longer deal with that record class.

enddata ∈ *word* ⇒ ()

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. Al strips in a strip class must have the same component size but not necessarily the same number of components. 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 may be paired with the update routine to form a doublet. 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 word to be associated with the strip class, an estimate of the total number of all components of all strips in the strip class (this is purely to help in efficient implementation) and the component size of the strip class. It produces as results the initiator function and the doublet for the strip class:

stripfns ∈ *word, integer, 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 function which given a strip produces the word associated with its strip class. This is *dataword* (*see* section 7.2 'Records').

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

The functions *datalist, dataword* and *copy* and the routine *enddata* 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, e.g. because they are the value of a variable whose extent has not finished (*see* section 3.2 'Declaration and initialisation'). Any items which can no longer be referred to are destroyed, i.e. 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, e.g. to zero. In the case of identifiers the identifier can be cancelled (*see* section 3.3 'Cancellation').

To avoid too frequent garbage collection compound items can be deleted, i.e. returned to the storage control system, using the standard routine:

delitem ∈ *item* ⇒ ( )

When an item is deleted its components are not deleted.

After an item has been deleted it is no longer available and the onus is on the programmer not to use it. The implementation may not give an error message if he does use it, the value simply not being defined.

8. STANDARD STRUCTURES

8.1. References

There is a standard record class called *References*. These have one component which is a full item. The word associated with the class 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 e.g. 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 word 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 recognised by the standard function *atom*.

atom ∈ *item* ⇒ *truthvalue*

8.3. Lists

There is a standard data structure called a *Link* which is used to construct another data 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').

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 *true* and whose back is a function.

The standard function *null* recognises the null list

null ∈ *list* ⇒ *truthvalue*

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

A link is either:

(*i*) a pair whose front component is any item and whose back component is a list, or

(*ii*) a pair whose front component is *false* and whose back component is a function with no arguments and one result.

In case (*ii*) the function is one which when repeatedly applied produces a succession of items, not necessarily all the same, i.e. normally the function will side-effect itself. The last item produced should be the terminator. 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*, *list* ⇒ *link*
destructor: *dest* ∈ *list* ⇒ *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*; *cons*(*false*, *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 1; vars f x;
  if isfunc(back(1))
    then back(1)→f; f() → x;
      if x=termin then true→front(1)
        else x→front (l); conspair(false, f)→back(l)
      close; l
    else 1
  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; vars f;
  if isfunc(back(l)) then back(l)→f; (f(),l)
    else (front(l),back(l))
  close
end;
function null l;
  if l=nil then true
    elseif isfunc(back(l))
      then if hd(l) or null(solidified(l)) then true else false close
    else false
  close
end;
```

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.

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 brackets*⟩::=[|]
⟨*list constant*⟩::=[⟨*list constant element* *?⟩]
⟨*list constant element*⟩::=⟨*list constant*⟩ | ⟨*character group*⟩

Example:

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

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

⟨*list expression brackets*⟩::=[% | %]
⟨*list expression*⟩::=[%⟨*expression list*⟩%]

Thus
[% %] is equivalent to *nil*
and [% ⟨*expression*⟩ %] is equivalent to *cons*(⟨*expression*⟩, *nil*)
and [% ⟨*expression*⟩, ⟨*expression list*⟩ %] is equivalent to
cons(⟨*expression*⟩, [%⟨*expression list*⟩%])

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

For convenience the following functions are standard:

next ∈ *list*⇒*item*, *list* (similar to *dest* but non-destructive)
::(a synonym for *cons* but an operation of precedence 2)
⟨⟩ ∈ *list*, *list*⇒ *list* (concatenates the lists, an operation of precedence 2).

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* ∈ *integer* → *full strip*
doublet: *subscr* ∈ *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* ∈ *integer* ⇒ *character strip*
doublet: *subscrc* ∈ *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 bracket*⟩::=' | `
⟨*string constant*⟩::='⟨*string constant element* * ?⟩`
⟨*string constant element*⟩::=⟨*string constant*⟩ | ⟨*any character except a string bracket*⟩

Example:

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

Spaces and newlines are significant in string constants.

There are functions to input and output character strings stored as character strips (*see* section 9.1 'Input and output'). The external format is as for string constants.

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 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.

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 two integers being alternately the lower and upper bounds for each subscript. The second parameter is a function used to initialise 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 obtained by partial application and is equivalent to

newanyarray (% *init*, *subscr* %).

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*. The word associated with the record class is *"word"*. The standard functions characterising words are:

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

Each character of the POP-2 character set corresponds to a unique integer. The correspondence rule depends on the implementation. Note that the functions above are variadic 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 constructor does not take a meaning component as argument. The meaning of a word is *undefined* unless the word has been updated to have a particular meaning.

Words may occur in the program as quoted words, i.e. word constants, with the following syntax:

⟨*unquoted word*⟩::=⟨*letter*⟩ ⟨*alphanumeric* *?⟩ | ⟨*sign* *⟩
| ⟨*decorated bracket*⟩ | ⟨*bracket decorator*⟩
| ⟨*separator*⟩ | ⟨*period*⟩ | ⟨*exponent*⟩ | ⟨*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 *Standardised*, i.e. 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 standardised, but the update routines do not standardise.

8.7. Functions

Functions are compound items. There is no constructor or destructor for functions. They can be constructed by the methods described in section 4.1 'Definition of functions', and they can be deleted by the routine *delitem* (*see* section 7.4 'Garbage collection'). There is a family of characteristic functions associated with the class of functions to form a primitive 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** or **routine** this has initially no defined value. This component may be selected or updated by using a standard doublet:

updater ∈ *function*==⇒*routine*

Closure functions i.e. 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 the front (if a closure function is obtained by successive partial applications only the formals frozen by the last one are counted). 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, i.e. 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 recognises functions:

isfunc ∈ *item* ⇒ *truthvalue*

9. INPUT AND OUTPUT

9.1. Input

Information which is input to the POP-2 system is organised 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

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 function to read characters from it i.e. a function ∈ ()⇒*character*. 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 again used. The list supplied to it has a head which is the word *"close"* and a tail which is a list of one element: a 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 terminated by spaces or newlines where necessary to separate them from the following character group.

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

incharitem ∈ (() ⇒ *character*) ⇒ ((*l*) *text item*)

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 characterx from the *standard input* file:

charin ∈ () ⇒ *character*

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 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*.

itemread ∈ () ⇒ *text item*

It is defined thus:—

function *itemread*; *proglist* . *dest* → *proglist* **end**;

9.2. Output

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

To open an output device the standard function *popmess* (*see* Section 9.1 'Input') is used to produce a routine to deliver characters to it i.e. a routine ∈ *character* ⇒ (). The list supplied to *popmess* has a head which is an output device name and a tail which is a file name.

There is a standard function to convert a routine which delivers a sequence of characters to an output file into one which delivers a sequence of text items.

outcharitem ∈ (*character* ⇒ ( )) ⇒ (*text item* ⇒ ( ))

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 may be assigned to if a different output file is to be made current. An output file is closed by outputing the terminator.

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

sp ∈ *integer* ⇒ ( )
nl ∈ *integer* ⇒ ( )

There is a standard function which outputs any item to this file in some suitable format and produces that item unchanged as its result.

print ∈ *item* ⇒ *item*

There is a standard macro which uses *print* 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 macro is denoted by the POP-2

identifier ⇒ (not to be confused 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').

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 characters which do not include ';'

⟨*code instruction*⟩::=$⟨*any sequence of characters 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*⟩
⟨*program*⟩::=⟨*program elementt*⟩;⟨*program*⟩

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 and a space 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.

11.2. Macros

A *Macro* 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 **routine** and no formal parameters are allowed.

⟨*macro definition*⟩::=**macro** ⟨*identifier*⟩; ⟨*function body*⟩ **end**

A macro, like an operation, is applied whenever it is mentioned and does not need parentheses after it.

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.

Example:

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

This is the same as 7//2→r→q;

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* section 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. 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 evaluates the next program element. *setpop* may also be applied in a function body.

setpop ∈ () ⇒ ()

ACKNOWLEDGMENTS

This language is a development of R. J. Popplestone's 'POP-1' programming language (see the paper in this volume). The debt to the ALGOL, LISP, CPL and

ISWIM programming languages should be obvious. We are indebted to a number of people in this department and elsewhere for helpful discussion and criticism, to Dr David Park who contributed to discussion of the storage control scheme and to Mrs Margaret Pithie and Miss Eleanor Kerse who typed this report. Mr M. Healy kindly pointed out a number of errors and obscurities in a draft.

The work has been undertaken on a grant from the Science Research Council under the supervision of Dr D. Michie, whose encouragement has been invaluable.