Chapter 1

Introduction

This book is an exposition of the POP-11 language. In this introductory chapter we discuss the requirements which a computer language should meet, the view of computation presented to the user by the POP-11 language, how it meets the requirements, what its advantages are relative to other languages, and in what directions it should develop.

POP-11 was developed from POP-2[1]; since we will need to have some discussion of the development of the language we shall use the term POP to refer to POP-2 and to what might be called its *nearly parthenogenetic* descendents such as POP-11 and WonderPop.

A computer language is a formalism by which a *user* creates a *computer program*. A language can thus be regarded as providing a user a *view* of a computer. A language should meet the following requirements:

- A language should be *convenient* to use. A user should be able to express his requirements in a form that is reasonably succinct, and, where possible, in a form that is consonant with existing, non-computer-oriented forms, e.g. English and mathematics. The language should lend itself to a rapid comprehension by the human eye and brain. Some measure of user-extensibility of the language should be provided.

  Language designers should bear in mind Occam's Razor — *Non sunt multiplicanda entia praeter necessitatem.* While it is desirable to provide a wealth of capabilities, these should be built on a minimal set of basic constructs.

Compared with the succinct style of mathematics[1], computer languages tend to be verbose. Some of this verbosity is perhaps inevitable, since programs have the complexity of engineered structures, rather than the simplicity of the single aspects of engineering for which relevant mathematics can be developed. Much however arises from language weaknesses, from the typographic impoverishment[2].

of traditional program presentation, restricted as it is to the *typescript paradigm*, and the limited context sensitivity of programmning languages.

- A language should aid the production of *reliable programs*, that is to say it should prevent the user from making certain errors, and it should support programming disciplines that make others unlikely. One aspect of this is the provision of the capability of writing programs as *modules* with restricted, and carefully specified dependencies.

- A language should be *conveniently powerful*: Any serious language is of course capable of expressing any computation, but not necessarily in a convenient or efficient form. In particular, any truly general-purpose language should support that class of computations that can be regarded as *symbolic*, e.g. natural language processing or mathematical reasoning. We justifiy this requirement by the fact that almost all computing of any sort is nowadays dependent on the symbolic computations performed by a compiler, or at least an assembler — the days of prestidigitation at the hand-keys are past. Thus flexible support for various aspects of compilation should be part of the language capability, as providing a view of the code-creation capabilities of the machine.

- A language should be *acceptably efficient*. Since any language effectively constrains which programs, out of all possible programs, a user may create, any language can be expected to degrade the performance of a computer, although if we consider instead the programs that a user might be prepared to write in a given language, we may well find that performance is enhanced. However a language in which it is not possible conveniently to express a given algorithm so that its complexity class with respect to a standard model of computation is maintained should be regarded as *unacceptably inefficient*.

[1] Thoughout this book, the word "mathematics" is used in the modern sense, of a formalism in which propositions are subject to a process of *proof*. Mathematics is thus not necessarily about entities constructed out of numbers.

[2] This typographical impoverishment is even compounded in languages like Common Lisp, and the antique technology of the single case teletyp preserved, by the convention of converting, by default, lower case input to upper case

- A language and its implementation should *support the straightforward development of programs*. In particular, the most common operations required in developing a program should be capable of being done incrementally *in constant time*, so that the task of creating a large program does not become unbearably tedious.

These requirements are in some measure opposed, although developments in Computer Science are making it easier to implement languages with good all-round performance. For example, convenience, power and safety are to some extent opposed to efficiency.

POP-11 addresses the requirements in a way that can be summarised as:

- POP-11 supports symbolic processing in an idiom that is close to that understood by the great majority of computer scientists: the syntax is stylistically close to that of ADA, Pascal and C, as opposed to the minimalist syntax of LISP, while the semantic constructs are far less foreign than those of Prolog. This encourages users to perceive a continuum between symbolic and non-symbolic processing, facilitating essential cross-fertilisation between parts of Computer Science that are often perceived as being disjoint. Thus POP-11 is *powerful and convenient*.

- POP-11 supports, but does not mandate, the functional paradigm in programming. That is to say, it is possible to write POP-11 procedures which accurately represent mathematical functions over appropriate domains. More than any pure functional language, POP-11 provides support for the use of the functional paradigm in a way that offers competitive performance. It does this by providing structured access to the machine memory especially through *properties* and *vectors* which can be wrapped in functional clothes using its efficient *closure* mechanism based on *partial application*.

  Functional programming facilitates good software engineering by employing concepts whose formal properties are well-understood, and amenable to formal verification, and is the primary means by which POP-11 combines *reliability with power*.

- The incremental compiler, developed originally for the POP system, provides for the *constant time* modification of programs, as compared with the $O(n)$ time associated with conventional linkers. This supports the interactive development of POP-11 programs, forms the basis of the implementations of Common Lisp, Prolog and ML in the POPLOG system, and contains features especially intended for the support of Prolog.

  This kind of power, pioneered by the POP language, is not available in any direct way in conventional language systems, which always require some awkward interaction with the operating system, and do not normally support dynamic linking of code.

1.1 Engineering Advanced Software

For many important purposes a computer program needs to be regarded as an engineered artefact, that is to say an entity aspects of whose behaviour can be predicted in advance of its performance in the field. Programming languages have an important role to play in support of good software engineering, provided that there is some *preferably formal* basis for predicting that their behaviour will be satisfactory.

In any area of design, appropriate design methodologies depend not only upon the physical and logical laws underlying the subject of the design, but also upon how ambitious the artefact to be designed is. An architect designing a building will be at pains to assure his customer that whatsoever rains may beat upon that building, whatsoever winds may blow, it will stand. As a basis of this assurance, and as a defence in any malpractice suit subsequent to his assurance being ill-founded, he will rely upon codes of practice. An aircraft designer specifies an artefact subject to the same physical laws of mechanics and aerodynamics as the architect. However, since flying rather than standing is what his artefact is destined for he will employ different codes of practice, different design methodology, in which for example the properties of materials are much more tightly controlled.

Likewise, a more sophisticated computer program will need a more sophisticated language to develop it. This is especially true when a significant component of symbolic processing is to be embodied.

1.1.1 Data-Structures

Many of the basic operations of a computer, e.g. adding, subtracting, multiplying and dividing can be regarded as implementing mathematical functions. It is thus not surprising that programming languages should be proposed which aim to provide an extensible range of mathematical functions. Such languages would be amenable to simpler formal analysis, and provide a ready way of implementing more advanced mathematical theories (e.g. matrix multiplication as opposed to multiplication of integers).

However the memory of a computer does not present a simple functional model to a user. We may specify memory using a function *mem* of two arguments, applied thus: $mem(a, s)$, where a is an integer address, and s is a *machine state*. a is an integer $0 \leq a < a_{max}$, where a_{max} is the memory size of the machine. The essential difference between the addition function and the memory function is that in the case of performing the evaluation of $x + y$ the programmer has direct control of both arguments, whereas in the case of the evaluation of $mem(a, s)$ he only has direct control over the a argument. The s argument, although determined by the user's program[3] , depends upon the input data in a way that makes reasoning about the behaviour of the machine very difficult[4]

Access to memory is fast — typically it is modelled as taking constant time, although $O(\log(a_{max}))$ or $O(a_{max}^{1/3})$ is perhaps more accurate, the first value being based on the number of stages required to decode the address, and the second on the physical space required to hold a_{max} words of memory.

Designing a time-efficient algorithm for a given computation can often be achieved by avoiding the repetition of certain sub-computations. This can sometimes be done simply by re-arranging the order of computation, e.g. the evaluation of the polynomial $a_n x^n + a_{n-1} x^{n-1} \ldots a_0$ can be performed using $O(n)$ multiplications by employing the recurrence relation.

$$a_n x^n + a_{n-1} x^{n-1} \ldots a_0 = x(a_n x^{n-1} + a_{n-1} x^{n-2} \ldots a_1) + a_0$$

This can then be massaged into a non-recursive form, placing no additional requirements on memory. More often, however, repeating a sub-computation can only be avoided by making use of memory to store the result of the first instance of a given sub-computation for future reference, particularly when the laws underlying the computation are weaker than the commutative ring laws used in the above example.

Thus a (possibly sophisticated) use of memory is the key to achieving acceptable time-efficiency of many programs. But we have observed that memory is state-dependent, and is thus hard to reason about. This can have deleterious effects on the reliability of programs. In order to mitigate this, all computer languages structure the data held in memory in order to ease the analysis (formal or informal) of the behaviour of programs.

[3]At least if we speak of the state of a suitable virtual machine

[4]There is an interesting analogy with the *frame problem* which arises in reasoning about the ordinary world[4].

The Functional Paradigm

In general, the *data-structures* provided by programming languages retain the dependence on machine-state that is present in 'raw' memory, and thus some of the problems of reliability are still present. However there is an approach, due to John Mc.Carthy, which allows all properties of data-structures to be state-independent. In this approach, as generalised for POP and other languages, a user has available *constructor functions*. When a constructor function c is applied to multiple arguments, $c(x_1, x_2, \ldots x_m)$, the *storage allocator* provides a unique address a, and stores $x_1, \ldots x_m$ in memory locations $a + \delta \ldots a + \delta + m - 1$ where δ is an increment allowing some *housekeeping information* to be stored [5]. In subsequent constructor applications, none of the addresses $a \ldots a + \delta + m - 1$ will be returned by the allocator, nor is the programmer provided with a means of changing their contents. This provides a means of representing mathematical objects that are more complicated than a single number in a purely functional, or state-independent way.

A simple allocator would use memory only once, moving a *free pointer* on at each constructor function application. However this is not a practical proposition for most applications, and systems usually embody a *garbage collector* to recover the memory space occupied by objects which have been constructed but can no longer be accessed by the programmer.

The arguments of the constructor functions, encapsulated in the data-structure created by the constructor function, are accessed by *selector functions*.

1.1.2 The Modified Functional Paradigm

POP-11 (in common with LISP and ML) provides a modified functional paradigm. This is motivated by two considerations:

- Sometimes it is necessary to have an object which supercedes a given object, but which is almost the same. If the object concerned is large, it is more efficient to modify the appropriate location in memory, thus *updating* the object, than to construct a new

[5] In many LISP implementations $\delta = 0$ because of extensive use of *tag bits* associated with the pointer. POP-11 makes restricted use of tags, which has advantages in 32-bit architectures with current memory growth

object. Thus the data-structure that an editor uses to store the text a user is operating on may well be of this nature.

- The connection between the program and the world outside the computer (e.g. the user of the computer) is made through fixed locations in memory, which have to be updated. Or, to put it another way, users want a state-dependent protocol in their transactions with the machine.

Thus POP-11 provides a compromise. Store is administered by an allocator that fully supports the functional paradigm, but the data-structures so created may be updated at the user's discretion, as may be variables, using an assignment statement. This means that the user has a choice of paradigm. A large amount of a given program may be written functionally, but with some low-level operations being written non-functionally, and the operations necessary to provide the interaction with the external environment also written using data-structure updating.

The availability of the functional paradigm makes it much easier to write sophisticated software, even in circumstances in which it might seem *a priori* to be inappropriate. While the conventional VED editor of POPLOG is written using a buffer which is updated at every change to the document, the advanced typesetting Pantechnicon system is written using the functional paradigm, so that the edit buffer is *reconstructed* at every key stroke that changes the buffer. Even large structures can be efficiently reconstructed if they are held in an appropriate tree form.

Updating data-structures in POP-11 is accomplished in a uniform way, by defining *updater procedures* for the corresponding selectors. Thus *hd* is the selector which returns the first member of a list whilst $updater(hd)$ is the corresponding updater procedure. A special syntactic form is provided for applying these updaters: if s is a selector function then $v \to s(x)$ applies the corresponding updater (i.e. performs $updater(s)(v, x)$).

The modified functional paradigm can also be expressed in terms of the common properties that an object in such a language must have. In [?] this was expressed as an *Object's Charter of Rights*[6][7] .

[6] The term used in the original description of POP-2 was *item*
[7] The term "first class object" was used rather earlier

Anything which can be the value of a variable is an object. *All objects have certain fundamental rights.*

1. All objects can be the actual parameters of functions
2. All objects can be returned as results of functions
3. All objects can be the subject of assignment statements
4. All objects can be tested to see if they are *identical*.

1.1.3 Structure Identity in the Modified Function Paradigm

There are two natural notions of the equality of objects O_1 and O_2 , represented as data-structures in store, namely

1. two objects may be equal if their components are equal. In POP-11 this is tested by $O_1 = O_2$
2. two objects may be equal if they have the same address in memory, i.e. they are identically equal. This is tested by $O_1 == O_2$

It is clear that $O_1 == O_2 \Rightarrow O_1 = O_2$. The converse is not necessarily true. These operation have different time-complexity, $O_1 == O_2$ takes constant time, $O_1 = O_2$ is $O(n)$ in the size of the smaller object.

1.2 Stack Based Languages

The main storage allocation mechanism of POP-11 makes no assumption about any relationship between the order in which data-structures are obtained from the allocation mechanism and the order in which they are garbage-collected. In general, there is no relation. However many languages, which are descendents of ALGOL 60, require the order of freeing of memory for re-use to be the inverse of the original order of allocation. This is simpler and more efficient to implement than the fully general garbage collected mechanism that POP-11 has available[8].

The use of stacks in programming languages arose because they provide a simple way of evaluating mathematical expressions. Indeed, a stack provides a fully general way of implementing a functional language, *but at the cost of copying (possibly large) data-structures up and down the stack*, as indeed is done in the *call by value* paradigm in the ALGOL 60. The allocation method used in POP-11 avoids this cost because data-structures are referred to indirectly by an address or *pointer*.

In stack-based languages, it is conventional to use one stack in which space allocated for user data-structures is intermixed with procedure activation records. The effect of this is that *the lifetime of a data-structure is contained in the lifetime of the procedure in which it was created*. That is to say that a procedure P cannot return a data-structure that it has created to the procedure, P_1 say, that called P. This is acceptable for many numerical algorithms where it is trivial to predict what size of data-structure will be required to hold the results of a procedure in advance of calling the procedure. E.g. if you want to take the cross-product of two 3-vectors you know that you will need a 3-vector to hold the result, so you can allocate the space before you call the cross-product procedure.

It is characteristic of symbolic processing that there is no simple relationship between the size of the arguments to a procedure and the size of the result. This holds whether the symbolic operation be the differentiation or integration of a mathematical expression or the parsing of a sentence. This is why John McCarthy developed the LISP language, and it is why stack-based languages have never been popular for symbolic processing.

1.2.1 Extended Stack Based Languages

Given the limitations of stack-based languages it is not surprising that they should be extended. The usual approach is to provide for *allocation* of data-structures off a *heap*, that is an area of store distinct from that administered by the stack. In the C language for ex-

[8]However, if space available is increased by a constant factor generally the time required will also be increased by a (different) constant factor

ample, the *malloc* procedure serves to allocate data off the heap. However implementations of these languages seldom preserve enough information to enable a garbage collector to be implemented, so that freeing of storage for subsequent re-use has to be done by an explicit procedure call. This can lead to bizarre program errors which are very hard to find, arising from two distinct data-structures sharing the same store. Thus these languages cannot be regarded as fully supporting a functional paradigm.

1.2.2 Stacks in POP-11

Since stack allocation is cheap to administer, almost all languages make some use of stacks. The normal flow of control during procedure call and return is naturally handled by a stack mechanism. There are some exceptions to this rule — in an implementation of the Prolog language the execution of a predicate may terminate with the possibility of subsequent resumption, so that the normal stack-discipline cannot be applied to the activation records for the procedures which represent the definitions of Prolog predicates in an efficient implementation of that language.

Likewise any implementation of a language that supports multiple processes cannot be handled by the provision of a single stack.

For these reasons POP-11 makes use of the stack mechanism to support procedure call and return. Unusually, it makes use of two stacks working in opposition — an argument passing stack and a stack which holds procedure activation records. The argument passing stack provides a simple, explicit method of access to the arguments of a procedure, which is the basis of an efficient implementation of variadic functions, as described later. It also supports the efficient implementation of the significant range of algorithms that require direct access to a stack.

1.3 Abstraction

If you want to buy a new refrigerator you will specify only the details that are relevant to you — the capacity say and limits on height, depth and breadth. That is to say, you have abstracted the relevant details.

In the design of a complex program, abstraction should be used, so that the program can be specified in a modular manner, leaving the issue of details to the implementers of the modules. Many language designers regarded the *procedure* or *block* as a sufficient capability for modularisation, and indeed these constructs do provide important capabilities which are adequate for many purposes.

However some capabilities are best provided by a collection of procedures and possibly data-structures which share some common, hidden, attributes. The concept of *section* was introduced in POP-2 as a way of meeting this difficulty[9].

POP-11 supports abstraction:

- A collection of procedures to operate on the abstract objects can be specified.
- Internal details of the operation of a set of procedures to operate on such objects can be hidden from the outside by using the *section* mechanism.

One of the more attractive features that has been incorporated in programming languages in recent years has been *pattern based definition and call of procedures.* This has been incorporated in languages such as Prolog and ML. It is not a standard facility of POP-11, although it is easily implemented as a macro. However it does have pitfalls when it comes to supporting abstraction. Consider the ML:

```
datatype Vec = vec of (real*real*real)
fun add_vec(vec(x1,y1,z1),vec(x2,y2,z2)) = vec(x1+x2,y1+y2,z1+z2);
```

ML requires *vec* to be a concrete data-type, which prevents it being abstracted as a constructor function.

One feature of POP-11 that can assist in the support of abstraction over pattern matching is the *destructor function.* This is the inverse function to the constructor function. I.e. if c

[9] This concept, when introducted into LISP, is known as a package

is an constructor function and d is a destructor function then $d(c(x_1 \ldots x_n)) = x_1 \ldots x_n$ and $c(d(X)) = x$, where $=$ means elementwise equality.

Pattern matching can be supported by providing 3 procedures for each data-type, a constructor, a recogniser and a destructor. It is a straightforward exercise to translate add_vec into the following POP-11:

```
define add_vec(v1,v2);
  if isvec(v1) then destvec(v1) -> z1 -> y1 -> x1;
    if isvec(v2) then destvec(v2) -> z2 -> y2 -> x2;
    consvec(x1+x2,y1+y2,z1+z2)
    else mishap('vector needed',[^v2])
    endif
  else mishap('vector needed', [^v1])
  endif
enddefine;
```

1.4 Data-structures,Arrays, Memo functions and Variadic Functions

We have said that an efficient algorithm makes use of the memory of a computer to avoid repeating a computation. In the functional paradigm, this mostly means that there are defined *selector functions* [1] which execute in near constant time. These selector functions, together with constructor, destructor and recogniser functions provide the main implementation of an object class in POP-11 (for ways in which common properties of the class can be clustered, see Chapter **??**.

However, not all uses of memory can be handled by selector functions. The most elementary use of memory is simply to bind a variable, as when $vec(x/sqrt(a^2+b^2), y/sqrt(a^2+b^2))$ is rendered in POP-11 as $lvarsd \;=\; sqrt(a^2+b^2);\; vec(x/d, y/d)$.

Non-local uses will require some function.

Where data-objects are pre-defined, there is no scope for providing selector functions. Thus for example a function defined over a range of integers cannot be implemented by adding a new field to the integer data-structure, because there is no such structure for (short) integers. And anyway, such an addition would require re-linking of the POP system. This capability is provided instead by *arrays*. In POP-11, an array is essentially a procedure defined over ranges of integers, and implemented by associating access code with a section of memory.

In POP, a more general kind of use of memory was introduced by Michie[3]. If a function is *memoised*, it is modified so that before it is evaluated in the ordinary way, its value when applied to the given arguments is first looked up using a *property function*. It is easy to write a macro *memoise*

```
memoise f;
```

1.5 The POP-11 notation

POP-11 supports symbolic processing in an idiom that is close to that understood by the great majority of computer scientists: the syntax is stylistically close to that of ADA, Pascal and C, as opposed to the minimalist syntax of LISP, while the semantic constructs are far less foreign than those of Prolog. This encourages users to perceive a continuum between symbolic and non-symbolic processing, whereas most of the symbolic processing languages used for Artificial Intelligence have the appearance of being radically different from conventional languages. This gulf has hindered cultural diffusion within Computer Science: for example the incremental compilation techniques developed for POP-2 (the precursor for POP-11) over 20 years ago, and similar techniques subsequently employed for LISP, still have not found their way into standard computer science text-books, which insist on presenting the generation of code as a static process occurring during a compile-time which has no overlap with run-time.

Other symbolic processing languages used for Artificial Intelligence have the appearance of being radically different from conventional languages. This gulf has hindered cultural diffusion within Computer Science: for example the incremental compilation techniques developed for POP-2 (the precursor for POP-11) over 20 years ago, and similar techniques subsequently employed for LISP, still have not found their way into standard computer science text-books, which insist on presenting the generation of code as a static process occurring during a compile-time which has no overlap with run-time.

LISP is an interesting case. The heavily bracketed notation is derived from that of the λ-calculus. This notation serves a particular mathematical purpose very well — namely to provide a formal apparatus for reasoning about functions. E.g. one may wish to show that a particular set of rules for simplifying λ calculus expressions is *confluent*, i.e. you always get the same result no matter in which order you apply your rules.

LISP also serves to emphasise the functional view of programming. Every expression is explicitly written as a function application. Writing $(+\ x\ y)$ serves to emphasise that the same process is occurring as when you write $(sin\ x)$ or $(fred\ x\ y)$. So far an unimpeachable didactic case exists for LISP.

However the human engineering of LISP is poor when the aim is *not* to reason about the fundamentals of computation but to specify a complex computation. Human mathematicians depend very heavily on having a notation that is well human-engineered. Consider the expression

Discuss case confusion.

1.6 Lazy Languages

POP-11 is an *eager language*, that is, the arguments of a function are evaluated whether or not they will be needed. The conditional form and certain boolean forms are exceptions to this rule.

There is considerable interest in the literature in *lazy* or *non-strict* languages. These only evaluate the arguments of a function when absolutely necessary. Call by name in Algol 60 had something of this nature. They offer theoretical advantages, since they correspond to .... reduction in the λ-calculus, and practical advantages, particularly in the support of infinite structures, e.g. infinite sequences. The addition function is *strict*, so that $x + y$ always has its arguments evaluated. Constructors are treated as non-strict, e.g. the arguments of

$cons(l, L)$ will only be evaluated when a selector function is applied to the result.

Lazy languages, by only evaluating arguments as and when they are actually needed might appear to offer efficiencies, and indeed they do by some absolute measure, for some computations which are non-terminating when evaluated eagerly, do terminate when evaluated lazily.

More mundane considerations, alas, make laziness less desirable. Procrastination, as we all know, is a thief of time. Lazy evaluation involves the computer amassing upon the heap records of thoughts for the morrow which are a great burden upon the storage allocator. This can be mitigated by *strictness analysis*, by which the requirements of strict functions are propagated throughout a program.

POP-11 provides some of the capabilities of lazy languages within its essentially eager framework. In particular, POP-11 lists are designed to be capable of representing infinite sequences of objects. This can be useful in writing parsers, which can be made pure functions on lists of tokens.

Below are shown lazy and eager representations of matrices.

```
vars n_rows = newassoc([]),                ;;; Maps from matrix to the
     n_cols = newassoc([]);                ;;; number of rows and cols.

define new_lazy_matrix(m,n,A) -> A;        ;;; A lazy matrix is simply
  m -> n_rows(A);                          ;;; a function with associated
  n -> n_cols(A);                          ;;; dimensions.
enddefine;




define new_eager_matrix(m,n,A) -> A1;      ;;; An eager matrix is an array
  lvars m,n,A,A1 = newarray([%1,m,1,n%],A);
enddefine;

define n_rows(A);
  A.boundslist.tl.hd;
```

```
enddefine;

define n_cols(A);
  A.boundslist.tl.tl.tl.hd
enddefine;



define mult_mat(A,B);
  lvars m = n_cols(A);
  new_matrix(n_rows(A),
             n_cols(B),
             procedure(i,k) -> s;
             lvars i j k s=0;
               for j from 1 to m do
               A(i,j)*B(j,k) + s -> s;
               endfor
             endprocedure)
enddefine;
```

1.7 The development of POP-11

Since this book deals in detail with POP-11, it is best to explain the development of the language by reference to its current state.

The major structural features of the language were developed by 1968, including the functional treatment of arrays, dynamic lists, the incremental compiler, records and record-classes.

Subsequent development of POP-11 fall into the following categories:

1. provision of additional language features
2. providing user-accessibility to facilities that existed in the system but were not 'seen' by the user

3. providing a more complete 'view' of computer and operating system.

involved

- A move to using upper and lower case characters, distinguishing between the two (c.f. LISP). This was also provided on the DEC-10 dialect.
- The provision of a pattern matcher.

Lisp family - much in common with Pop Readability Clean treatment of procedures as objects (like Scheme) A short list of features only in Pop (described in more detail later): stack, VM, processes, partial application, matcher, dlocal, compiler routines, layered saved images, POPC??....

The original POP-2 code-generator was capable of emitting a very restricted set of instructions. This was partly motivated by the fact that it ran in a time-sharing system in which a user would crash not just his own process but the entire machine. This inhibited portability of the language, since an implementation on a new machine would require a sizable kernel of primitive procedures to be written.

Part of the original vision for POP-2 had been as an implementation language for other languages. Thus the paper [5] contains a discussion of the development of a logic-based language. POP-2 provided good support for the initial stages of this work. Work by Boyer and Moore explored structure sharing in logic — an essential component of any practical logic-based language. Burstall and Darlington and others worked on developing the Hope language, based on recursion equations. However the further development of POP-2 based systems was hindered by (a) lack of portability and (b) the inability of the system to generate good quality code.

This problem was attacked in the development of POPLOG. In this, a *system dialect* of POP was developed which supports extended code generation. Thus system portability is greatly enhanced.

Intended for the education of the sons of gentlemen. Research was administered by civil servants to whom publication appears to be an unnatural vice.

1.7.1 Comparison with LISP

LISP was one of the most significant influences in the design of POP-2.

- In POP-11 and SCHEME the binding of procedures to procedure names uses the same mechanism as the binding of other objects to variable names.
-
-

Bibliography

[1] Burstall, R.M. and Popplestone, R.J., [1968] The POP-2 Reference Manual, Machine Intelligence 2, pp. 205-46, eds Dale,E. and Michie,D. Oliver and Boyd, Edinburgh, Scotland.

[2] Burstall, R.M., Collins, J.S. and Popplestone, R.J., 1971, Programming in POP-2, Edinburgh University Press, Edinburgh.

[3] Michie,D. 1968 "Memo functions and Machine Learning, *Nature, 218* 19-22.

[4] Mc Carthy, J. and Hayes, P.J. [1969] Some Philosophical Problems from the Standpoint of Artificial Intelligence, in Machine Intelligence 4 (eds Meltzer B. and Michie, D), Edinburgh University Press.

[5] Popplestone,R.J. 1968 "The Design Philosophy of POP-2" in *Machine Intelligence 3*, ed. D.Michie.

Chapter 2

In which we learn All about Procedures

"Well," said Owl, "the customary procedure in such cases is as follows."

"What does Crustimoney Proseedcake mean?" said Pooh. "For I am a Bear of Very Little Brain, and long words Bother me."

"It means the Thing to Do."

"As long as it means that, I don't mind," said Pooh humbly.

Copyright R.J.Popplestone and the University of Sussex, 1988.

Somewhere we should have a discussion of 'call by value' and copying. NOTE - Cog App should look at dlocal, active.

From johng

In fact

define foo(); lvars x, y; define fred(); -¿ y -¿ x; lvars x, y; ...

DOES give an error, i.e. when lvars are used non-locally and then redeclared as local. The problem arises when the preceding non-local use references permanent variables; at the moment, the VM compiler doesn't check for this case (I agree it probably should, but it means maintaining a list of all permanent identifiers referred so far in the current procedure).

From aarons

I have just found this in an old email message from Robin Popplestone. Robin said: —— ¿I had trouble with the following (machine generated) definition, which ¿worked OK with vars. ¿ ¿define fred ; -¿ y -¿ x ; lvars x y ; + (x , y) enddefine ; —— My feeling is that this should not work.

Currently Pop-11 treats the first occurrences of x and y as non-local non-lexical, declaring them if necessary.

Wouldn't it be better if it gave an error?

Compare define fred; lvars y = (), x = (); +(x,y) enddefine;

which does what Robin probably wanted. Aaron

2.1 What are procedures?

Recall that any behavior of a POP system is achieved by procedures. In POP a procedure is a kind of data-object which can be "called" in order to make it exhibit behavior. For example

```
sqrt(0.5) =>
```

calls the procedure *sqrt* to extract the square-root of the number 0.5 [1]

There are a number of synonyms for calling a procedure. We may speak of *applying, calling, running, invoking, obeying, executing* or *activating* a procedure. We shall confine ourselves to referring to the terms "calling" and "applying", "running" and "executing". In a POP context "calling" and "applying" are synonymous, but they do have a different background, with different connotations. We will mostly use "calling" when we are concentrating on the mechanics of what happens when we get a procedure to do something. Mathematicians speak of "applying" a mathematical function to its arguments. In many cases it is useful to think of a POP procedure as implementing a mathematical function, and to speak of applying it to its arguments. The essential difference between a function and a procedure is that a function always produces the same result when it is applied to the same arguments, whereas a procedure does not necessarily do so. In addition, a mathematical function always produces a result when it is applied to any arguments in its *domain*.

In mathematics, if we apply a function f to an argument x obtaining a value y, we say that y is, or is equal to, $f(x)$, and write $y = f(x)$. In computing we tend to speak of a procedure as *returning* or *producing* a *result*.

The idea of a *partial function* which does not always produce a result is much employed in the theory of computation, since if a formalism is sufficiently powerful to define many "interesting" functions, it is a real problem to know whether a particular definition expressed in that formalism does indeed define a function, or only defines a partial function.

In POP the addition procedure + is a partial function, since $x + y$ always gives the same result if x and y have the same values, whereas the procedure $oneof$ for choosing a member of a list randomly is not a partial function since $oneof$([1 2 3]) can yield a different result on different occasions. The procedure + is only a partial function, since it may fail to perform addition on numbers which are too big.[2] A full theoretical treatment of procedures

[1] In fact it is possible to associate behavior with any POP object, using the *class_apply* field of the data-key. But this behavior is always accomplished by a procedure associated with the object class, as described in Chapter 3.13.

[2] Note that POPLOG POP-11 provides arbitrary precision integer arithmetic — however even with this capability it is still possible for the procedure + to fail to produce a result, since the system may run out of memory.

has to take into account the state of the machine that is executing the procedure, and is consequently much more complicated than a applicative theory. Those familiar with digital electronic circuits will recognise a close analogy here. Circuits composed of logic gates in which there is no feedback from the output of one gate to an input of a preceding gate can be analysed using Boolean algebra, whereas circuits with such feed-back can have internal states, and cannot be analysed using Boolean algebra alone. However, just because Boolean algebra is not applicable to the complete analysis of every circuit does not mean that it is useless for analysing any part of any circuit, and likewise an applicative understanding of procedures is a valuable way of analysing the behavior of all of some programs and of parts of others.

Many mathematical functions take several arguments. Addition, for example takes 2. We can regard this however as mapping from a *tuple of arguments* to a result, that is to say, the *domain of such a function is a Cartesian Product*. It is usual in mathematics to suppose that the *arity* of a function is known and constant in any particular theory - sin for example only ever takes one argument. It is useful computationally to extend this idea to allow functions which are *variadic* — i.e. they may take a tuple of unspecified size as argument. POP in particular does allow variadic functions, and also vari-result functions which may produce tuples of arbitrary size as result.[3].

A POP procedure has a standard written form, which is a sequence of ASCII characters. This is translated by the POP *compiler* into Virtual Machine Code (VMCODE), which may be further translated into the actual code of the computer on which POP is implemented. The serious POP user does need an understanding of the Virtual Machine, but this is a sufficient model for his understanding — details of the actual machine code are not important for users who are not concerned with implementing a POP system. We will speak of a procedure as *running* when it has been called and its VMCODE is currently being executed. We will use the term *executing* for the process of executing VMCODE.

Every POP system comes with many *built-in* procedures. For example *hd* and *tl*, *sin* and *cos* are built in procedures. Other procedures are available in the library, are defined in POP-11 and *autoloaded* as needed, so that they appear to be built-in. The library mechanism is described in 14.[4]

[3] The POP treatment of such functions does however differ significantly from that of LISP

[4] In fact, in POPLOG, most of the built-in procedures are also written in the system-dialect of POP, but their text is not available to the user in the way that library procedures are.

The running of a procedure can be regarded as having three phases

- *Entry* to the procedure: this serves to set up the computer's memory before:
- *Executing the body of the procedure*: this is when the main computation is done, whether specified by the user or as part of a built-in procedure.
- *Exiting* from the procedure: this usually sets up the computers memory to permit the computer to carry on doing what it was doing before the procedure was called.

Immediate execution of POP text outside of an explicit procedure definition, either as typed in commands, or from a file, is accomplished by treating POP text up to a semicolon (;) or printarrow ($\Rightarrow$) as a kind of anonymous procedure definition which is called as soon as the terminating word is encountered, and then thrown away.

When we speak of *compile time* we mean any time during which the POP compiler, which is simply a built-in procedure, is running. In conventional languages like Pascal, compiling is an entirely separate activity from running a program, but in POP the two are closely interleaved.

Some other interactive languages require no compilation at all. Basic is perhaps the best known example. The text of a Basic program is *interpreted* by the computer running Basic.[5]

Some programming languages allow the user only to define procedures which are partial functions; these are called *applicative languages*. The language ML [**?**] is an example of such a language. It is possible to use POP and LISP as applicative languages by never updating data-objects and by only making one assignment to any variable. This makes for programs which are less efficient, but does not seriously restrict the programmer except for applications in which complex interactions with the outside world are taking place.

POP and LISP can return any data-object as the result of a procedure. This is made practicable by the *garbage collector*, a concept first developed for LISP. The garbage collector is a built-in procedure which discovers data-objects that the user can no longer access, and collects up the storage that they occupy for re-use. It is not usually necessary for the POP

[5]It is possible to obtain improved performance in Basic by compiling each line of Basic before it is executed the first time, and continuing to use the compiled version until the text is edited by the human user.

user to call the garbage collector explicitly — it will be called as necessary by any procedure which creates a data-object. Languages like Pascal which do not have a garbage collector restrict the user to being only able to write applicatively those procedures which return a restricted range of data objects as result. In particular it is difficult to express symbolic computation in such languages in the applicative form which makes it (relatively) easy to know that the program is correct.

As we shall see later, it is much easier to reason about the behavior of programs written applicatively because the usual mathematical rule that *things which are equal can be freely substituted for each other.*

POP was strongly influenced by mathematics during its design — it is perhaps accurate to characterise POP as a blend of mathematical and computational ideas, whereas LISP is a purely mathematical core upon which has been grafted a great deal of computation. Some of the procedure names of POP (*apply*, *partapply*) are derived, via LISP, from the mathematical ideas, although they are not restricted to working with purely applicative programming.

During our systematic treatment of procedures in this chapter we address ourselves to the following questions:

- What is the written form of procedure definitions?
- How can we begin to reason about procedures?
- What actually happens when we run a procedure?
- A procedure is a data-object — what procedures can we use to access its fields?
- How can we write procedures which create other procedures?

2.2 The form and role of a procedure definition

A procedure definition can be best thought of as a special kind of an initialised variable declaration, and thus it does two main things; it declares an identifier (the name of the procedure) and assigns it a value (a data-object of type procedure). Thus for example:

```
define sumsq(x,y);
x**2 + y**2
enddefine;
```

defines a procedure which takes two *arguments* x and y and returns the sum of their squares. The name "*sumsq*" is defined as an ordinary variable, a procedure data-object is created and assigned to the variable. [6] The original text that defined a POP procedure object is not embodied in the procedure object when a procedure definition is processed (in most of the machines in which POP runs, it would be too expensive to keep it around except on disc). [7] It is however important for a procedure object to have a name, so that the user can recognise it if he prints it out. Hence a procedure definition takes an additional third action, it updates a field of the procedure object called the *pdprops* field to be the name of the procedure.

To illustrate that a procedure definition does indeed create a variable which has a procedure object as value, try typing in the definition above, and try:

```
sumsq =>
** <procedure sumsq>
```

That is, the value of a procedure variable can be printed in the same way as the value of any other variable. There is an exception to this rule in the case of certain procedures variables such as operations and macros — see 2.7.1 and 2.26.1. The default way of printing a procedure object is as above, i.e. the word "procedure" followed by the name of the procedure all enclosed in angle brackets.

Built in procedures are likewise the values of variables, although the variable concerned is usually *protected* to prevent it being accidentally changed (see 5.3.7).

[6] POP-2 departed from LISP in making the binding from procedure-name to procedure object be the same as the binding from variable to data-object. The POP convention was subsequently adopted for the SCHEME dialect of LISP, but COMMON-LISP preserves the original convention. Arguably the POP/SCHEME convention comes closer to treating procedures as "first class citizens".

[7] In a later chapter **??** we shall show how it is possible to to provide the user with the capability of seeing the text which gave rise to a particular procedure.

```
sin =>
** <procedure sin>
```

Just as we do not have to make every list we create be the value of a variable, there is a way of creating an "anonymous" procedure-object without an associated procedure valued variable, for example:

```
procedure(x,y); x**2 + y**2 endprocedure
```

creates the procedure object which performs the sum-of-squares operation defined above. Indeed the following is almost equivalent to the procedure definition at the beginning of this section

```
vars sumsq = procedure(x,y); x**2 + y**2 endprocedure;
```

You will notice that there is a difference if you try printing out the value of *sumsq*.

The general forms of the named procedures created with *define* and the anonymous procedures created with *procedure* are the same apart from a short sequence which names the variable and may associate certain special properties with it (e.g. that it is an operation). The specification of the syntax of a POP-11 procedure definition in BNF is:

```
<definition> = define {<defn_spec>} <args>{<rtarrow> <results>}{<with_specs>} ;
                     <expression_sequence>
               enddefine

<procedure>  = procedure <args><results>{<with_specs>} ;
                     <expression_sequence>
               endprocedure

<rtarrow>    = '->'
```

The $< defn\_spec >$ is usually absent. However it can be used to specify that a procedure identifier should behave as an infix operator such as + (see section 2.7.1) or to modify the language syntax using a *macro* (see section 2.26.1) or *syntax* (see section 2.27). definition, or to define what happens when a procedure occurs as the destination of an assignment statement (see section 2.10). The syntax of $< defn\_spec >$ is given below. The *procedure* option is used to restrict the identifier to being procedure-valued — it must not be confused with the anonymous procedure form.

```
<defn_spec> = {updaterof} | <identprops>
<identprops> =   macro | syntax | syntax <precedence>
               | procedure | <precedence>
<precedence> = <decimal>
```

As in the above examples, $< args >$ is a specification of the variables which are the *arguments* (or *formal parameters* or *input variables* of the procedure. Usually they provide a pattern for how the procedure should be called. For example the *sumsq* procedure defined above can be called by:

```
sumsq(3,4) =>
** 25
sqrt(sumsq(2+1,4)) =>
** 5.0
```

The essential idea is that the variables in $< args >$ are given values determined by the call of the procedure (in the above example $x = 3$ and $y = 4$ in both cases), and the $< expression\_sequence >$ is then *executed* to determine the result of the procedure. However for some purposes it is necessary to have a more detailed understanding than this, and what actually happens when a procedure is called is explained in section 2.5. In particular, we explain how the system avoids confusing the values of variables which have the same name in different procedures. Occasionally you may need to write POP procedures in which the arguments in the procedure heading do not match the call.

The formal syntax of the $< args >$ is as follows:

```
<args> = (<word> { {,} word}*)  | <word> {{,}word}*}
```

This is pretty free. Parentheses, if present, must match, and the words may be separated by commas. It is better to use both parentheses and commas if you are defining a normal procedure, since this indicates the pattern of how it should be called.

Note that the word "*procedure*" immediately preceding a word $< word >$ in an $< args >$ statement is used to restrict the variable $< word >$ to being of type procedure, and *not* to declare an argument called *procedure*.

The $< results >$ may not be present (more precisely they may be an empty sequence). They serve (a) to make it clear what the result of a complicated procedure is and (b) to allow a procedure conveniently to return more than one result. They are described in section 2.5.

The $< with\_spec >$ is rather seldom used. It allows the user to modify certain aspects of the procedure object, and is described in section 2.21.1.

The $< expression\_sequence >$ is what does the actual work of the procedure, and is often referred to as the *body* of the procedure. Unless the procedure is trivial, it will call other procedures — either built-in procedures or user-defined ones — to accomplish its purpose. E.g. the *sumsq* procedure defined above, the $< expression\_sequence >$ is $x**2 + y**2$ which calls the procedures + and ** (twice each) to perform addition and exponentiation. In many cases constructions will be used which will choose alternative procedures to call, or repeat sequences of procedure calls. We shall explain the possible forms of a statement sequence in section 2.12.

2.3 The ordinary syntax for calling procedures

A normal procedure identifier is simply an identifier that has a procedure as its value. See 5.3.5 for how to find out about the properties of an identifier using *identprops*. A procedure is called by the syntax:

```
<word>({<expression>}{,<expression>}*)
```

here < *word* > is the name of the procedure identifier.

Other forms of procedure calling are discussed in section 2.7

2.4 An applicative model of procedure calling

We can regard the calling of *sumsq* as simply *substituting* the actual arguments 3 and 4 for the formal arguments x and y, in the expression sequence $x**2+y**2$ thus obtaining $3**2+4**2$. In some sense, this expression then *evaluates* to 25, via applications of $+$ and $**$. Being able to understand the process of calling a user-defined procedure as being a process of substitution is the essence of applicative programming, and underlies the simplicity of reasoning about applicative programs.

In a later chapter we will develop objects called *terms* which allow us to treat procedure call as substitution, and reason about procedures. This opens up a number of possibilities — that of being able to *prove* that a procedure is correct, and being able to craft a complex procedure from a number of simpler standard ones without sacrificing efficiency. [8]

[8]It has to be admitted that LISP provides a more direct treatment of the applicative model than POP does, since LISP function definitions are lists with a very direct representation of the applicative structure. During the design of POP-2, Rod Burstall did advocate that POP should have a defined *abstract syntax* which would provide a similar capability. However it seemed difficult to incorporate variadic functions into this framework, and these functions were needed for some important capabilities, such as defining *newarray*. In addition, we needed to get the language implemented on what was a tiny machine by today's standards, and an abstract syntax representation of every procedure would have taken more room, and would have had to be interpreted. The Prolog language provides an abstract syntax, which is interpreted in some implementations, but compiled (and decompiled!) in others (CHECK). In this book, we will explore the idea of an abstract syntax for POP in **??**

2.5 But procedures actually operate off the stack

POP-11 uses a portion of its memory known as the *user stack*, abbreviated usually as the *stack*, for procedures to store information temporarily when communicating with other procedures. Besides the user stack, there is also a *call-stack* used by the procedure calling mechanism mostly to support the implementation of local variables. The use of this is described in section 2.6.1. Detailed knowledge of this second stack is not required for most applications of POP.

The stack is so called because, like a stack of plates, you can add things on the top, and you can take things off the top. The thing you take off will always be the *last* thing that was put on. Such a structure is sometimes called a *Last In First Out, or LIFO, structure*. Usually computer scientists speak of *pushing* data-objects on a stack — the image is that of a spring-loaded stack of plates. The action of taking items off the stack is referred to as *popping* — no pun intended. So procedures communicate information with one another as follows. If procedure P_1 wants to execute procedure P_2 giving it certain arguments, it pushes those arguments on the stack and then just calls P_2, leaving it to P_2 to take them off, as needed.

If when P_2 has finished it has any information for P_1, it should push the information on the stack, and P_1 can pop it off as needed. The syntactic forms of POP are intended to express these stack actions in a form that aids an understanding of the process in applicative terms, and indeed it is possible to show that an applicative language can be correctly implemented using stack operations [?].

We will illustrate the stack operations in discussing the operation of POP by drawing our stack *horizontally*, with the top to the right. Thus

1	2	3

is the result of pushing the numbers 1, 2 and 3 on the stack. Sometimes we will label a stack box with the name of the variable that has been pushed in it, if we have not stated what its value is, or the result of a computation, such as $x + 3$. The empty stack is displayed as

Recall our old friend *sumsq*. We will rewrite it in a slightly different form:

```
define sumsq(x,y) -> r;
    x*x + y*y -> r;
enddefine;
```

The *top* line of the procedure definition says a number of things:

- This is a definition of a procedure, whose name is "*sumsq*".
- It has three *local* variables, called x, y and r. Local variables are in some sense private[9] to the procedure, so that they do not get in the way of other variables of the same name. Indeed, when a procedure calls itself recursively, we have to keep distinct versions of local variables for different calls of the same procedure. Local variables are discussed in section 2.6.1. The first action that is taken when a procedure is called is to ensure that local variables do not interact with other variables of the same name, and the last action before control is returned from *sumsq* to the procedure that called it is in some sense to restore the local-variable accessing mechanism to its state before the procedure was called.
- Two of the local variables x and y are also *arguments*, so when *sumsq* runs, there should be two (or more) data-objects on the user stack. (There might be more, because other procedures have stored information which is to be used later.) The top two data-objects will be removed from the stack, the top one being assigned to y and the next one to x (compare the order in the procedure heading).
- When the procedure finishes running, but before the local variables are restored, the value of the variable r, whatever it is, should be left on the stack. (N.B. the procedure heading does not state that anything will be assigned to r. That has to be done inside the procedure body)

All of the above is implied in just the first line of the procedure definition. Look at the procedure heading again and see how it gives all that information. The rest of the procedure says how the values given to x and y are to be used to work out what number to assign to r, so that it can be produced as the result at the end.

[9] We are not using private in a special technical sense here, as it is used in ADA.

Now suppose you ask *sumsq* to find the sum of the squares of 3 and 4, and assign the result to z;

```
sumsq(3,4) -> z;
```

Notice how this has the same general *format* as the procedure heading in the definition

```
define sumsq(x,y) -> r;
```

The statement $sumsq(3,4) \to z$; translates into the following:

1. push 3 on the stack $\boxed{3}$
2. push 4 on the stack $\boxed{3}\boxed{4}$
3. call the procedure *sumsq* $\boxed{25}$
4. remove the top item from the stack and store it in the variable z,

We shall see in Chapter 16 that each of the operations 1–4 above is one VMCODE instruction.

If you then type $z \Rightarrow$ this translates into “Push the value of z on the stack, then run the *print arrow* procedure”. The *print arrow* procedure simply prints out two asterisks, and then all the items on the stack.

As implied in step 4 above, the assignment arrow $\to$ is used to cause the top of the stack to be moved to somewhere else, usually to a variable. The $\to$ should not be confused with $\Rightarrow$

2.5.1 Nested Procedure Calls

In general, if you call a procedure, by typing its name, then various expressions between parentheses, then this means push the values of all the expressions (i.e. the things denoted by the expressions) onto the stack, then call the procedure. E.g. if $fred$ is a procedure that adds three numbers then:

```
fred(10, sumsq(3,4), 99)
```

Means:

- push 10 on the stack $\boxed{10}$
- push the result of sumsq(3,4) on the stack $\boxed{10}\boxed{25}$
- push 99 on the stack $\boxed{10}\boxed{25}\boxed{99}$
- call the procedure called $fred$ $\boxed{134}$

Getting the result of $sumsq$(3,4) itself can be expanded in similar fashion, so the expression $fred(x, sumsq(3, 4), 99)$ translates to

- push the value of x on the stack $\boxed{\text{x}}$
- push 3 on the stack $\boxed{\text{x}}\boxed{3}$
- push 4 on the stack $\boxed{\text{x}}\boxed{3}\boxed{4}$
- call the procedure $sumsq$ $\boxed{\text{x}}\boxed{25}$
- push 99 on the stack $\boxed{\text{x}}\boxed{25}\boxed{99}$

- call the procedure $fred$ $\boxed{134}$

In general, any word or other data-object or expression terminating with parentheses immediately preceding an opening parenthesis will be treated as follows:

- Push the values of all of the expressions enclosed within the parentheses on the stack
- Call the object as though it were a procedure.

Usually the object will be a word which is the name of a variable whose value is a procedure-object, as in the example above. It is however possible to associate procedural behavior with any data-object, as described in chapter 3.13. For example if L is a list and i is a number, then $L(i)$ is the ith member of L.

In particular, a procedure call may produce a procedure-object as result which may itself be applied.

```
vars F_trig = [^sin ^cos ^tan];
  F_trig(2)(0.1) =>
```

Translates to

- Push 0.1 on the stack. $\boxed{0.1}$
- Push 2 on the stack. $\boxed{0.1}\boxed{2}$
- Call F_trig $\boxed{0.1}\boxed{\text{cos}}$
- Call *apply* — a special procedure for calling the procedure object on the top of the stack. $\boxed{0.995004}$

When a list is called as a procedure it replaces the number which is the head of the stack by the list-member indexed by that number. The effect of the above sequence is to evaluate $cos(0.1)$, where the angle is measured in radians (assuming $popradians = true$).

2.5.2 Procedures which act just on the user stack

The following procedures allow you to perform various useful manipulations on the user stack.

$identfn() \to ()$
The identity procedure does absolutely nothing when called and so leaves the user stack untouched. This might appear to be a useless procedure, but it is *useful* when passed as a procedure argument to other procedures, e.g. *applist* described in Chapter **??**.

$erase(O) \to ()$
This procedure removes the top item from the user stack. This can also be achieved by an isolated assignment, $; \to ;$.

$erasenum(O_1, O_2, \ldots, O_n, n)$
This procedure removes the top n items from the user stack.

$dup(O) \to O \to O$
This procedure duplicates the top item on the user stack.

There are some procedures which tell you something about the state of the user stack as a whole, or allow you to manipulate it as a whole:

$stacklength() \to n$
This procedure returns the number of items on the user stack.

$setstacklength(n)$
This procedure sets the user stack length to n. If the current stacklength m is greater than n, then $m - n$ items are erased, otherwise $n - m$ nils ([]s) are pushed on (the reason for using

[] is that this procedure is principally used by the Lisp compiler).

clearstack()
This procedure clears all items from the user stack.

$subscr\_stack(n) \rightarrow O$
$O \rightarrow subscr\_stack(n)$

Return or updates the n-th element on the user stack, where the element on top of the stack (the most recently pushed) has subscript 1.

Finally note the existence of this constant:

popstackmark
The value of this constant is a stackmark record (whose conventional use is by the POP-11 list constructor to mark a position on the stack for *sysconslist* – see **??**). This item is the only stackmark record available to the user (although others are used inside the system); it prints as $< popstackmark >$.

2.5.3 A comparison between procedure calling in POP and other languages

Readers who have an understanding of other languages, or of computer architectures should understand that the POP stack contains data-objects represented by one machine word each. This is usually 32 bits, or 4 bytes.

The procedure call mechanism in many languages is supported by a single stack. This is possible because the number of arguments to and results of a procedure are known at compile time. POP allows you to write procedures in which this is only determined at run-time, and indeed there are some built in procedures like this, for example *consword* (see 8.1).

Lisp also has a capability of passing a variable number of arguments to a procedure and returning a variable number of results. The Lisp model for doing this is to pass them as a list. In the Symbolics implementation of Common Lisp, this list is actually built by pushing the arguments on one of the machine stacks. The stacked arguments can then be made to appear as a list by using the technique of "CDR coding", which allows a sequence of pointers in store to be regarded as a list if they are suitably flagged. This technique however is contentious, because this "list" is very volatile — it only exists for the duration of the procedure call.

A detailed discussion of the POP implementation is given in Chapter 16.

2.6 What happens inside a procedure when it is called

The following is a sufficient model for understanding what happens when a procedure is called from the point of view of a POP user. For implementation it may be possible to make use of certain shortcuts where a procedure that is normally constant, such as +, is being called.

- The procedure prepares the memory locations that will hold its local variables, that is any variables declared between the $define$ and $enddefine$ or *procedure* and *endprocedure*. The nature of the preparation depends on whether the variables are *dynamically* or *lexically* local, as described in section 2.6.1 Arguments and results are treated as variables declared in the procedure.
- The actual values of the arguments are taken off the stack and assigned to the argument variables. The top of the stack is assigned to the last argument variable, the top-but-one of the stack to the penultimate argument variable etc.
- The VMCODE derived from the $< expression\_sequence >$ that forms the body of the procedure is executed.
- When this has finished, either by a *return*, described in section 2.12.4 or by reaching the end of the $< expression\_sequence >$, the result variables are placed on the stack, in inverse order to that in which they were declared.

- The procedure restores the memory locations that have held the local variables, and returns control to the procedure which called it.

There are certain exceptions to this behavior. A procedure can be called to update a data-object, and what happens in this case is described in section 2.10. A procedure may return control to a procedure other than that which immediately called it. This behavior is described in section 2.24, and in Chapter 12.

Thus *entering* a procedure means the behavior necessary to protect the local variables and give the argument-variables their values, and *exiting* from a procedure to mean the behavior necesary to restore the state of the local variables and return the results as appropriate. A *normal exit* is that accomplished by a *return* statement, or by completing the execution of the procedure body. An *abnormal exit* is one accomplished by any other means.

2.6.1 Local variables and expressions

As we have indicated above, a procedure needs *local variables* to have somewhere "private" to keep data-objects away from accidental interference by another procedure. There are two basic strategies for accomplishing this:

- The value of the variable can be saved on entry to the procedure and restored on exit. In this case, the variable is said to be *dynamically local*. [10]
- For the duration of the procedure execution, the value of the variable can be held in a place in store unique to that call of the procedure. In this case the variable is said to be *lexically local*.

It is important to realise that these categories refer to the manner in which a variable is local to a procedure, and not to the variable itself. In particular, a variable that is lexically

[10] Special variables in Common Lisp are dynamic in this sense.

local to a procedure P_1 can be dynamically local to a procedure P_2 defined within the body of P_1. [11]

A variable is said to be a *lexical variable* if it is lexically local in some procedure. Otherwise it is said to be a *permanent variable*. Further information about how variables are stored and accessed is to be found in Chapter 5 and Chapter 16. Note that it is important to distinguish between different variables with the same name — essentially a variable is one location in memory that stores a value associated with a word, which is the name of the variable. One word can be associated with many different locations at different times.

For historic reasons, arguments and results of procedures are dynamically local by default. You are urged to use lexical locality if it is available in your implementation, since it is more efficient and avoids some name-clashes which are still possible with dynamic variables.

Another reason for preferring lexical locality is to do with *processes*, described in Chapter 12. You will need processes if you want to have a number of procedures each of which may have to stop running because it needs some data, typically from an external source. When this happens, the process can be *suspended* until a later occasion in which data is available, when it may be *resumed*. Process swapping is more efficient if lexical variables are used in the procedures used in the processes, since the *state* of a process is neatly encapsulated in the user and save stacks. [12]

If you want to write code that will make use of lexical locality if available but default to dynamic locality if not, you can define suitable macros (see 2.26.1) which will allow you use the forms for declaring lexical locality while actually implementing dynamic locality. If you do this, you should be careful to ensure that your programs will work with either kind of variable — there are real differences between them.

[11] The return-address of the procedure call is also saved on the call-stack, although it has to be saved as a (pointer to a procedure data-object) + offset, since the garbage collector cannot accept pointers to the middle of a data-object. Whether is can be properly regarded as dynamically local will depend on the computer hardware. However POP users do not have direct access to it.

[12] When we designed POP-2 we considered lexical variables as an option. We chose dynamic variables because: (a) We were working on the Elliot 4100 series machines in which the single index register doubled as as stack pointer. Now efficient implementation of lexical locality requires that an index register be used as a base address to access them. So we couldn't both have a user stack and lexical locals and compact, efficient (by AI language standards) machine code. (b) It is easier to provide information about the values of the local variables of a procedure after a mishap (or error as we called it then) if dynamic locals are used, since the actual values are in the locations named by the variables).

Dynamic Locality

In POP-11 expressions as well as variables can be dynamically local. The syntax word *dlocal* provides the most general form to introduce dynamic locality, that is to specify expressions (including variables) whose values are to be saved on entry to a procedure and restored on exit. Thus any assignments to the variable or expression while the procedure is running do not affect the value the variable has when the procedure has returned control to its caller. The restoration involves an assignment operation, so the only expressions that can be dynamically local are those that can be assigned to. Section 2.10 tells you about assignment to expressions.

The term *exit* refers both to the normal return from a procedure and also to the abnormal exits described in section 2.24, and those associated with the suspension of a process, described in Chapter 12. Full details of the mechanisms involved are given in ??.

The simplest form of a dynamic local declaration is the ordinary *vars* declaration when used within the body of a procedure. The syntax is given in Chapter 5. In AlphaPop this is the only form currently available. It has the effect of *(re)declaring* a permanent variable, and should only be used if you are sure that the variable should be permanent. *dlocal*, described below, is more flexible, and should be used if available. You can in any event permit its simpler uses by defining it as a macro if not available.

The syntax of a *dlocal* declaration is:

```
<declaration>     =  <variable_class> <varspec>* ;
<dlocal_decl>     =  dlocal <dloc_varspec>*;
<dloc_varspec>    =    <word> {,}
                    |  <word>  = <expression>,
                    |  nonactive <varspec>
                    |  nonactive ( <varspec>*){,}
                    |  {<n>}%<expression>%
                    |  {<n>}%<expression 1> = <expression 2>%
                    |  {<n>}%{<expression 1>}, <expression 2>%
```

A *dlocal* declaration must be made within a procedure body, and it may specify that the value of any of the following be saved and restored:

1. a permanent variable.
2. a lexical variable.
3. an active variable, as described in section 2.28
4. an arbitrary expression, sandwiched between % and %.
5. an expression with a *multiplicity* indicated by a prefixed integer $< n >$.
6. two expressions, separated by a comma, one to be saved and one to be restored. The first expression is optional.

The multiplicity of an expression is the number of results it is expected to produce. This must be known and constant, and the expression must be capable of being updated with the same number of results. The default multiplicity for expressions is 1. The multiplicity for variables is always known to the compiler, and is 1 except for active variables. You can give a multiplicity for variables using the $< n > \% \ldots \%$ construction, but is almost always inappropriate to do so.

The declaration may also specify an initial value to be assigned to the expression when the procedure is run.

An example of a *dlocal* declaration is the following

```
dlocal
  var1, active_var1,        ;;; ordinary or active identifiers
  var2 = x+4,               ;;; initialise var2 with <expression>
  nonactive active_var2,    ;;; nonactive value of active identifier
  active_var3=(3,4,5),      ;;; does:    3,4,5 -> active_var3
  %hd(L)%,                  ;;; expression to be saved and restored
  %hd(L2)%= 4,              ;;; L2 is initialised (after saving)
  3 %explode(L3)%,          ;;; Multiplicity 3, so list L3
                            ;;; must have 3 members
  2 %explode(L4)%=(1+x,2+y),;;; does:  1+x, 2+y -> dest(L4) on entry
```

```
3 %hd(L5),hd(L6)%,        ;;; different expression run on exit
0 %,[]->tl(L7)%;          ;;; only an exit action
```

Thus, in the case when only one expression is given between % and % then the updater of its main procedure or operator is run when the procedure exits. So

```
dlocal 3 %f(x,g(y))%=(a,b,c);
```

Means

1. On entry, evaluate $f(x, g(y))$ and save its 3 results as sv_1, sv_2, sv_3, say. These sv_i are held in the save stack, so they are in effect anonymous lexicals of the procedure.
2. Where the declaration occurs, do $a, b, c \rightarrow f(x, g(y))$.
3. On exit do: $sv_1, sv_2, sv_3 \rightarrow f(x, g(y))$

The procedure *test* below saves and restores the value of the expression $hd(tl(List))$, where *List* is a global identifier.

```
vars List =[1 2 3];

define test;
    dlocal %hd(tl(List))%;
    [original List ^List]=>
    5 -> hd(tl(List));
    [updated List ^List] =>
enddefine;

test();
** [original List [1 2 3]]
** [updated List [1 5 3]]
```

But the value of $hd(tl(List))$ has been restored on exit:

```
list =>
** [1 2 3]
```

It would also be restored if the procedure call terminated abnormally, e.g. as a result of a *mishap*.

Since there are various ways in which the execution of a procedure may terminate, including process suspension as described in Chapter 12 and *exitto* as described in section 2.24, it is important to be able to determine what has happened. To this end the active variable *dlocal_context* is provided. It has an integer value, the integer defining the context. Moreover the active variable *dlocal_process* points to the process undergoing suspension or resumption.

So procedures run on entry or exit can use *dlocal_context* and *dlocal_process* to determine what actions to perform.

When and how to use dynamic locality

If you have the option of using lexical variables, then you should only use dynamically locality when you want to change the state of some variable or data-object that has meaning outside the current procedure, just for the duration of that procedure. The most common cases will be that of permanent system variables like *cucharout*, described in Chapter **??**, which are used to switch the behavior of the system. If, for example, you want to direct output to an output stream *charout*_1, defined locally, you should say:

```
dlocal cucharout = charout_1;
```

Declaring *cucharout* as *lexical* (q.v.) would be wrong — you would simply create a new lexical variable, and *that* would be initialised, leaving unchanged the memory location that the system uses to determine how it is to output characters.

The difference between *vars* and *dlocal*

This section may be skipped on first reading, and depends on you having read Chapter 5. Prior to the introduction of *dlocal* into POP, only permanent variables (i.e. those declared with *vars*) could be made dynamically local to a procedure, and this had to be done with a *vars* statement inside the procedure. However, this was unsatisfactory inasmuch as a *vars* statement, being committed to declaring a permanent variable with the *identprops* as specified within the *vars* statement may alter the *identprops* of a previous permanent declaration. In other words, *vars* statements inside procedures have the appearance of making the *identprops* of variables local to the procedure, when in fact they do not.

dlocal however enables both permanent and lexical variables to be made dynamically local to a procedure (without any redeclaration), and thus means that the use of *vars* statements inside procedures can be avoided altogether; all permanent variables can be declared outside of procedures, and then made procedure-local with *dlocal*, e.g.

```
 vars honey;

   define Pooh();
       dlocal honey;
       ...
   enddefine;
```

Note that this way of doing things makes permanent and lexical variables (q.v.) interchangeable, except insofar as lexical variables can be accessed only in the file in which they are declared.

So the above example could just as well have used *lvars honey*; instead of *vars honey*;. No procedure invoked by *Pooh* would then be able to access *honey* unless declared in the same file.

If a procedure P_1 calls another procedure P_2, then P_2 can make use of the dynamic local variables of P_1. Consider the following example:

```
define fred(x);
```

```
  joe()
enddefine;

define joe();
  x=>
enddefine;

vars x = "bunny";
x=>
** bunny
fred("cat");
** cat
```

Thus the argument x of $fred$ which is a dynamic local variable of $fred$ is visible to joe. Readers familiar with a language like Pascal will know that this behavior could only be obtained in such a language if definition of the procedure joe were contained within the body of the definition of the procedure $fred$, that is to say, the occurrence of the variable x in joe is within the lexical scope of the variable x in $fred$.

This kind of use of dynamic variables should be treated with circumspection, but it can be useful, especially if one has a procedure which, in some sense, sets up an "environment" of dynamic locals in which other procedures can operate.

Lexical Variables

A variable is said to be *lexical* if it is lexically local to some procedure, that is to say it is declared by a variable declaration beginning with the word $lvars$ or $dlvars$. The full syntax is given in Chapter 5. Lexical variables differ from dynamic variables in that it is impossible to refer to them by name outside of the text of the procedure in which they are declared. For the duration of the procedure in which they occur a lexical declaration of a word will override any non-lexical declaration for the same word.

Using the *dlvars* form can be more efficient than $lvars$ in cases in which the variables so declared are used non-locally in a lexically embedded procedure. This is described in Chapter 16.5.

For reasons of compatibility with earlier POP systems, argument and result variables are taken by the compiler to be dynamically local, unless they are explicitly and separately declared as lexically local. This lexical declaration must immediately follow the procedure header, and it is recommended that you declare all arguments and results in this way, e.g.:

```
define Pooh(x, y, z) -> P;
    lvars x a y b c d procedure P;
    ...
enddefine;
```

POP lexical variables correspond to the variables of Scheme and Common-Lisp. They differ from the variables of Pascal and Ada in that they may be referred to in closures, and so are more complicated to implement — they may continue to exist after a return from their procedure.

So if we run our $fred - joe$ example using lexical locality we obtain:

```
define fred(x);
  lvars x;
  joe()
enddefine;

define joe();
  x=>
enddefine;

vars x = "bunny";
x=>
** bunny
fred("cat");
** bunny
```

That is, the variable x in joe is outside the scope of x in $fred$.

On modern computers it will be possible to access some lexically scoped variables faster than dynamically scoped variables, and the instruction to do so will take less memory. This will certainly be true if a lexical is *fully local.* [13] A lexical variable v is said to be fully local if

- v is not referenced by any procedure which lexically nested within the procedure in which v is declared.
- v is not referred to using the *ident* construction 5.3.

In particular, it may be possible to hold the value of some lexicals in the machine-registers — ie. within the Central Processing Unit (CPU or mill) of the computer rather than main memory. In POPLOG the first *two* lexical variables to be declared in a procedure are normally allocated to registers, this allows a user to determine which variables should be in the registers (i.e. x and a in the example above).

In order to aid you in making the above recommended declarations, there is a variable provided in the POPLOG system. This variable, *pop_args_warning* has a default value of $false$, but if you set it to true then any procedure whose formal parameters or output locals are not explicitly declared as vars or lvars will produce a warning message, e.g.

```
true -> pop_args_warning;

define Pooh(x,y) -> z;
    . . . .
enddefine;

;;; y DEFAULTED TO VARS IN PROCEDURE Pooh
;;; x DEFAULTED TO VARS IN PROCEDURE Pooh
;;; z DEFAULTED TO VARS IN PROCEDURE Pooh
```

If you write one procedure within the body of another, then the inner procedure can access the lexical variables of the outer. This kind of layout is, of course, standard for

[13] *Fully local* lexical variables can be efficiently implemented since they can be allocated either to registers or to cells in the procedure stack frame (giving faster access and compacter code). For more details on the implementation see 16.

Pascal or Ada. In general it has a disadvantage in POP in that the procedures so enclosed are not accessible outside the enclosing procedure, so they cannot be called independently, e.g. for debugging. However we shall see that there is an important use of this nesting of procedures for the creation of closures. The control of the scope of non-lexical variables is better accomplished using sections, described in chapter 13

Restrictions on the use of lexical locals

The following restrictions can be expected on the use of lexical local variables:

- Their values cannot be accessed during a break, e.g. if *popready* is called. (see Chapter **??**).
- Their values cannot be accessed or updated via calls of *valof*.
- Lexical variables cannot be used as settable pattern variables (i.e. those that begin with "?" and "??") with *matches* and procedures that call *matches*, for example *present*, *lookup*, *remove*, *foreach*. See **??**. However lexical identifiers *can* occur after ↑ and ↑↑ in lists

A reminder about things you can do with variables

Both lexical and dynamic variable can be used in initialised declarations, as described in 5. For example you can say

```
lvars x = y+3, L = [];
```

which is equivalent to

```
lvars x, L; y+3 -> x; [] -> L;
```

Care is needed in using this construction, since if you put a semicolon instead of a comma, you get a different meaning:

```
lvars x = y+3; L = [];
```

will put the boolean result of evaluating $L = [\,];$ on the stack. Note that the initialisation is done at the place in the procedure body where the declaration occurs, whereas the actions required to make the variable local, i.e. saving and restoring in the case of dynamic locality, are done on entry and exit to the procedure.

The construction *ident* $<$ *word* $>$ available in POPLOG can be used to access a data-object in which the actual value of the variable is kept. It is described in **??**

A procedure can specify that one or more of its local variables parameters is to be a variable of type *procedure*, thus

```
define applyto (list, procedure P);
    vars x;
    for x in list do P(x) endfor
enddefine;
```

would define a procedure equivalent to the system procedure *applist*. It would be more efficient and would reduce the risk of a clash of identifiers if lexical identifiers were used locally:

```
define applyto (list, P);
    lvars x,list,procedure P;
    for x in list do P(x) endfor
enddefine;
```

2.7 Other syntactic forms for calling procedures

2.7.1 Operators

Some procedures have names which are defined as *operators*. Examples are $+$, $*$, $=$, $<>$, $::$. They are most commonly placed between their arguments, and do not need parentheses to indicate that they are to be called. Operators that are intended to be placed between their arguments are informally referred to as *infix* operators.

So $x * y$ translates as

- push the value of x on the stack $\boxed{\text{x}}$
- push the value of y on the stack $\boxed{\text{x}}\boxed{\text{y}}$
- call the procedure $*$ $\boxed{\text{x*y}}$

You can define new operators by making use of the $< defn\_spec >$ option in the procedure definition syntax. Restricted for operator definitions, this is:

```
<defn_spec> = {updaterof} | <identprops1>
<identprops1> = <precedence>
<precedence> =  <decimal>
```

Thus $< defn\_spec >$ can be a numerical $< precedence >$ and it is this that signals that an operator is being defined. The $< precedence >$ is a number between -12.7 and 12.7. Only one digit after the point is significant. It is used, as described below, to determine which operator applies to which arguments when there is more than one operator in an expression. E.g. $x * 2 + y$ is interpreted as $(x * 2) + y$ and not $x * (2 + y)$ because $*$ has a precedence that is less than the precedence of $+$.

The recommended syntax for the definition of an infix operation with two arguments and no result variables is:

```
<definition> =  define <precedence> <argument> <word> <argument>;
                <expression_sequence> enddefine;
<argument> = <word>
```

Note that this is syntactically included in the procedure definition syntax given earlier, but *the second of the three words is the actual name of the procedure being defined.*

Thus infix operators are defined in POP-11 by using the word "$define$" followed by an integer specifying the precedence. For example, we might wish to define an infix operator $+++$ which takes the average of two numbers. The two permitted and equivalent formats are:

```
define 4 x +++ y;
(x+y)/2
enddefine;
```

and

```
define 4 +++(x,y);
(x+y)/2
enddefine;
```

Operators can take more than two arguments, although this is rare. In this case the format using parentheses must be used in the definition, though parentheses are not required in the call, e.g.

```
define 5 list5(p,q,r,s,t);
    [^p ^q ^r ^s ^t]
```

```
enddefine;

1, 2, 3, 4 list5 5 =>
** [1 2 3 4 5]
```

Result variables are specified as for normal procedures, for example:

```
define 4 x foo y -> z;
```

The *precedence* of an operator is used to determine whether one operator is called before another. The rule is simple — an operator moves to the right until it encounters an operator of a higher precedence, when it is called. This re-arranging of the textual order is done at compile time. [14] Apart from the various kinds of brackets, syntax words like ";", ",", "→" behave as if they have a high precedence, with "→" having a lower precedence that the others. The syntactic properties of brackets are discussed later.

Thus $x * 3 + y * 5$; translates to

- push the value of x on the stack $\boxed{\text{x}}$
- push 3 on the stack $\boxed{\text{x}}\boxed{\text{3}}$
- call the procedure $*$ (since $+$ has a higher precedence) $\boxed{\text{x*3}}$
- push the value of y on the stack $\boxed{\text{x*3}}\boxed{\text{y}}$
- push 5 on the stack $\boxed{\text{x*3}}\boxed{\text{y}}\boxed{\text{5}}$

[14] Thus POP has a rather cruder syntactic scheme for dealing with operators than Prolog, where the user can specify that operators are *infix, prefix* or *postfix*, and restrict possible parsings according to accord with these specifications. In fact, there are a variety of equivalent constructions in POP, e.g. $x+y$; $x, +y$; $+(x, y)$, $x, y, +$. In practise relatively few are used, and relatively little trouble is caused by this lack of structure in the language.

- call the procedure $*$ (since the syntax word ";" has a higher precedence than $*$) $\boxed{\text{x*3}}\boxed{\text{y*5}}$

- call the procedure $+$ (since the syntax word ";" has a higher precedence than $+$) $\boxed{\text{x*3+y*5}}$

The *absolute value* of the precedence is used in comparing different operators. The sign of the precedence is used to control the *association* of the operator, that is what happens when the same operator occurs in an expression (at the same level). A positive precedence means "associate to the left", that is to say the instance of the operator that occurs to the right is done first. Thus, since we defined $+++$ with positive precedence, $x+++y+++z$ means $(x+++y)+++z$ and *not* $x+++(y+++z)$.

```
    2 +++ 3 +++ 4 =>
** 3.25
    (2 +++ 3) +++ 4 =>
** 3.25
    2 +++ (3 +++ 4) =>
** 2.75
```

Some kinds of bracket serve to make the program text enclosed within behave syntactically as an independent unit of low precedence. Thus $x*(y+z)$; translates as

- push the value of x on the stack $\boxed{x}$

- push the value of y on the stack $\boxed{x}\boxed{\text{y}}$ (the call of $*$ is postponed because (y+z) has a low precedence, and we are now dealing with the text $y+z$ isolated within the parentheses)

- push the value of z on the stack $\boxed{x}\boxed{\text{y}}\boxed{\text{z}}$

- call the procedure $+$ (a closing parenthesis behaves as a operator of high precedence) $\boxed{x}\boxed{\text{y+z}}$

- call the procedure $*$ (The word ";" has a high precedence). $\boxed{x * (y + z)}$

The *if* ... *endif* brackets, described in section 2.12.3 also work this way, so that such a conditional expression can act in the right way e.g. you can say

```
x + if y=3 then 2 else 1 endif -> z
```

and the value of x is added to 2 or to 1 depending on whether y is or is not equal to 3.

It is possible to find out if a word has been declared an an operator by making use of the *identprops* procedure described in Chapter 5.3.5.

There is one special case of an operation. In certain circumstances, the minus sign is interpreted as denoting a unary operation, that is a procedure that takes just one argument off the stack. This interpretation occurs when the minus sign immediately follows a syntax word or an operation. E.g. in $x + -y$ and $(-x + 3)$ the $-$ is interpreted as unary. In that case a procedure *negate* is called. So the first sequence translates as:

- push the value of x on the stack $\boxed{\text{x}}$
- push the value of y on the stack $\boxed{\text{x}}\boxed{\text{y}}$
- call the procedure *negate* $\boxed{\text{x}}\boxed{\text{-y}}$
- call the procedure + $\boxed{\text{x - y}}$

However, if the minus sign immediately preceds a number with no intervening space, the two will be combined by the itemiser **??** into one negative number, so that $x + -1$; is translated as

- Push the value of x on the stack. $\boxed{\text{x}}$

- Push -1 on the stack. | x | -1 |
- Call the procedure $+$ | x-1 |

A list of the built-in operators and their precedences is to be found in Appendix **??**

An operation identifier O may be called with any number of arguments, in any of the following forms

O	no arguments — a nonary operator
$O\ a_1$	one argument — a prefixed unary operator
$a_1\ O$	one argument — a postfixed unary operator
$a_1\ O\ a_2$	two arguments — a binary operator
$a_1, a_2, ..., a_{n-1}\ O\ a_n$	n arguments
$O(a_1, a_2, ...a_n)$	n arguments

Be careful with nonary operators. A call of a nonary operator looks just like a variable access.

Normally, any mention of an operator in an expression causes it to be called. This can be a cause of stack underflow mishaps (see section 2.8) Calling can be suppressed by placing the word *nonop* immediately before the operator:

```
nonop +++ =>
** <procedure +++>
```

Using *nonop* a value may be assigned to the identifier. E.g.

```
conspair -> nonop +++;

3 +++ 4 =>
** [3|4]
```

Nonop can also be used to pass an operation as a parameter to a procedure that needs a procedure as argument. E.g.

```
vars A = newarray([1 10 1 10], nonop *);
```

creates an array which is a multiplication table. See Chapter 10 for a description of *newarray*.

Only a procedure may be assigned to an operation identifier so there need be no run-time checking that $+++$ has a procedure value, as there is with ordinary procedure names not declared with *vars procedure*.

See appendix **??** for information about built-in infix operators.

Boolean Operators

There are two boolean operations *and* and *or* which appear syntactically as ordinary operators. Thus you can say

```
if x=1 and y=3 then 23 else 25 endif
```

A common inefficiency is to use these operations unnecessarily in a conditional. For example:

```
define eligible(person);
  if age(person)>65 and income(person)<10000 then true
  else false
  endif
enddefine
```

can be much better rendered

```
define eligable(person);
  age(person)>65 and income(person)<10000
enddefine;
```

The words *and* and *or* can be thought of for most purposes as ordinary operators, but *they are not*. The expression p *and* q is evaluated as follows. Firstly p is pushed on the stack. If it is $false$ then the whole conjunction p *and* q is $false$, that is the evaluation of q is skipped. If it is not $false$ then it is popped off the stack, and the value of q is pushed on the stack.

p *or* q is evaluated in a similar manner. The value of p is pushed on the stack and if it is $false$ it is popped off and the value of q is pushed on. Otherwise q is not evaluated, and the value of the disjunction is the value of p.

The fact they are not ordinary operators is important in two ways

1. Both "arguments" of *and* and *or* are not necessarily evaluated. This makes for greater efficiency, and also allows some recursive definitions that would not otherwise work. For example:

   ```
   define member(x,L);
     L/=[] and ( x = hd(L) or member(x,tl(L)))
   enddefine;
   ```

2. *and* and *or* cannot be passed as parameters to a procedure using *nonop*, or rather they should not, because they are *syntax procedures* and can cause very peculiar behavior if called out of context.

2.7.2 Calling any procedure in post-fix form

If you put a full-stop before any word, or indeed any procedure object or anything that can act as a procedure by virtue of *class_apply*, this is equivalent to procedure call. Thus

```
x.tl.hd
```

is equivalent to

```
hd(tl(x))
```

This allows you to access data-objects in the manner of Pascal. [15] It is recommended that you only use this form for procedures that take one argument and return one result, since any other use can be difficult to understand.

2.8 Mishaps and the Stack

There is a very common mishap message associated with the stack

```
MISHAP - STACK EMPTY (MISSING ARGUMENT? MISSING RESULT?)
```

this happens when a procedure, or the assignment arrow, is trying to take something off the stack when there is nothing left on the stack. [16] Suppose you gave the command $sumsq(3) \rightarrow z$; This would push 3 on the stack, then call *sumsq*. *sumsq* would pop the number 3 off the top of the stack and assign it to y. It would then try to pop the top of the stack and assign it to x and find nothing left. You would then get a stack error, with a message as above.

[15] The precursor of POP-2, POP-1, was written using the reverse Polish form, subsequently popularised in Forth and PostScript. This was a convenient form to write data-object access, since in effect you can follow the pointers in the same order as the procedures occur in the text, and was consequently preserved as an option in POP-2, with the full-stop prefixing an identifier to indicate application. Independently the form was used in PL/1, and subsequently in Pascal

[16] Stack errors do represent one of the more difficult features of POP for the novice (and experienced) programmer. They are the penalty one has to pay for the succinct constructions available for building variable length data objects via the stack, and being able to write procedure like *newarray* which return as result procedures which can take a number of arguments not known at compile time.

This is an example of a missing argument. Note that the 'DOING' line of the mishap message will tell you that *sumsq* was running at the time the stack was found to be empty. Try it, and look at the mishap message carefully.

It may also be the case that a procedure fails to produce a result when one is expected. For example you might write:

```
define sumsq(x,y);
    x * x + y * y -> x;
enddefine;
```

If you type $sumsq(3, 4)$, x and y get the values 3 and 4 respectively (via the stack). Then the result of the expression $x * x + y * y$ is assigned to x (which is local to *sumsq* so x will not have that value when sumsq is finished). At this stage, nothing is left on the stack. Moreover, the heading of the definition does not specify that x is an *output local* variable for *sumsq*, so the value of x is not left on the stack. Nor is anything else left on the stack. So if you type

```
sumsq(3,4) -> z;
```

sumsq runs, using up the 3 and the 4, then finishes with nothing on the stack. But the assignment arrow $\rightarrow$ tries to take something off the stack to give to z, and at this point finds the stack empty. The machine prints another stack-empty mishap message. This time the problem is the missing *result* from sumsq, and the 'DOING' line of the mishap message will not mention *sumsq*. This is because *sumsq* has finished running when the error is detected. This can sometimes make it difficult to track down errors due to procedures failing to return a result. Usually you can get clues by looking at which procedures *were* running at the time.

You are advised to try the examples above, looking carefully at the error messages.

As we mentioned in the section on operators, you will sometimes get the stack-empty mishap as a result of applying an operator without providing it with arguments:

```
+ =>
```

```
;;; MISHAP - STE: stack EMPTY (missing argument? missing result?)
;;; DOING    :  mishap + compile
```

2.9 Specifying the results of a procedure

As we have stated, a POP procedure leaves its results on the stack. This means that it can produce as many results as it likes, and the number of results is not necessarily determined at compile-time. There are two ways in which results can be pushed on the stack:

1. The procedure can have *result variables* specified in the header, using the $< results >$ syntax.
2. The procedure may push results on the stack in its body.

The formal syntax for the declaration of result variables, which are sometimes known as *output locals*, is:

```
<results> = {-> <word>}*
```

The variables are pushed on the stack in the inverse order to that in which they occur in the declaration. This means that the declaration serves as a kind of pattern for using the result variables. For example if we have a procedure

```
define stats(L) -> x_bar -> sigma;
  lvars x_bar = average(L), sigma = stdev(L);
enddefine;
```

This can be called by

```
stats(L1) -> x_bar1 -> sigma1;
```

It is good practice to declare result variables for a procedure except for the following cases.

- The procedure is very short, and it is obvious what the result is. This is especially so if only built-in procedures are used in the body.

- The procedure produces a variable number of results. This in itself can be an undesirable thing to do, since a simple applicative model of the behavior of the procedure is lost. On the other hand, the stack does make a very efficient short-term storage mechanism, since no garbage-collection overhead **??** is generated by placing a variable number of entities on the stack, whereas constructing a variable size data-object such as a list does generate garbage.

Thus, for example, the following is acceptable style

```
define sumsq(x,y);
    x * x + y * y
enddefine;
```

since it is manifest that this procedure produces one result, and it could be considered pedantic to give it a result variable.

2.10 We can define procedures which update data-objects

A procedure for accessing a data object has two independent tasks to perform. One is to extract the contents of the appropriate field ; the other is to change the contents of the fields. Both these capabilities are treated thoroughly in Chapter 3.7. Thus:

```
hd(x) -> v;      ;;; get hd of x and assign it to v
v -> hd(x);      ;;; set the value of v to be the hd of x
```

The procedure *hd* is referred to as a *selector* for the list data-type, but in the second use above it is acting as an *updater*. What actually happens in this case is that instead of calling the procedure *hd* the system calls another procedure object held in a special *updater* field of the *hd* procedure object (see section 2.10). Indeed, whenever a procedure is apparently called when it is immediately associated with the → symbol, its *updater* is called instead.

When a procedure call follows an assignment arrow (i.e. →) the updater part of the procedure is called, so that instead of executing:

```
v -> f(x);
```

we could execute:

```
updater(f)(v,x);
```

Note that in a command such as $23 \rightarrow hd(tl(x))$ the updater of the *hd* procedure is called, but the *tl* procedure is called normally.

Updaters take their arguments off the stack (see 2.5) in the same way as other procedures. Moreover, items occuring before the assignment arrow can be merely additional arguments for the updater. So

```
x,y -> f(p,q,r)
```

is equivalent to

```
-> f(x,y,p,q,r)
```

and to:

```
x,y,p,q,r -> f();
```

If we call the updater of f, g,

```
updater(f) -> g;
```

then the above are also equivalent to:

```
g(x,y,p,q,r)
```

However, you should write calls to updaters in a way which enhances the clarity of your program, which means that what follows the assignment arrow should have the same form as you would use if you were selecting from the data-object rather than updating it.

Most procedures, of course, have no updater. If we define a new data object accessing procedure it is sensible to give it an updater, for example:

```
define second(list);
    hd(tl(list))
enddefine;

define setsecond(value,list);
    value -> hd(tl(list))
enddefine;

setsecond -> updater(second);
```

```
vars l; [a b c d] -> l;
```

```
second(l) =>
** b

"e" -> second(l);
l =>
** [a e c d]
```

A more convenient syntax for declaring updaters uses the $< defn\_spec >$ syntax — you will recall that a procedure definition had the form $define < defn\_spec >$ ..., which can be *define updaterof*...

```
<defn_spec> = {updaterof} | <identprops>
```

for example:

```
define updaterof second(value, list);
    value -> hd(tl(list))
enddefine;
```

The form *define updaterof* sets the *pdprops* of the *updater* to be the name of the procedure, as described in section 2.21.1.

Every POP-11 procedure has an updater field. However, for most procedures the default updater is merely a procedure which produces an error.

```
define test(x);
    x * x
enddefine;

test(3) =>
** 9
```

```
9 -> test(3);

;;; MISHAP - EXECUTING NON-EXISTENT UPDATER
;;; INVOLVING:  <procedure test>
;;; DOING    :  compile nextitem popval compile
```

Thus if a procedure has not been given an updater, and your program attempts to run its updater, an error message will result.

Since every procedure can have an updater, the use of updaters is not restricted to procedures which access fields of data-objects. Updaters can be used to associate two procedures which do related computations. For instance if procedure F does some elaborate computation to solve a problem, the updater of F could then be used to store a solution to be found by F by direct lookup (but see 11.5 for ways of combining these two ideas).

2.10.1 Calling the updater of an operation

If an operation identifier is used on the right of an assignment, its updater will be called. If there is no updater, an error occurs:

```
99 -> 3 +++ 4;
;;; MISHAP - EXECUTING NON-EXISTENT UPDATER
;;; INVOLVING:  <procedure +++>
```

When you have a nested operator and procedure calls associated with an assignment arrow, what happens? The best way of understanding this is to consider the arrow as an operator of high precedence (just below that of semicolon). The assignment arrow causes the call of the updater only of that procedure immediately below it in the application structure.

```
3+99 -> 3*x +++ 4^y;
```

can be bracketed as

```
(3+99) -> ((3*x) +++ (4^y));
```

and so the updater of $+++$ is called.

2.11 Procedure Identifiers

In normal use, a procedure definition sets up the procedure name as the name of an ordinary variable whose values just happens to be a procedure. Since the procedure definition sets up a *variable*, the value can be changed by assignment, which is exactly what happens if you recompile the procedure definition.

We have already seen that if we define an *operator* the variable created, as well as having different syntactic properties, is restricted to having procedure objects as its value. The procedure header-line can specify the type of the identifier naming the procedure. There are several main types of procedure identifier, ordinary, operation, macro, syntax, and active. In addition the identifier may be lexical or non-lexical, variable or constant, global or not.

2.11.1 Making a procedure name a procedure identifier

If *Pooh* is an ordinary identifier whose value is a procedure, then an instruction to run *Pooh*, for instance in another procedure definition, compiles into machine code operations that include a check to ensure that the value really is a procedure, in case something else has been assigned to the identifier. This will detect errors like this.

```
define Pooh(list);
    hd(tl(list))
enddefine;

define test(list);
    Pooh(list)
enddefine;
```

```
test([a b c]) =>
** b

999 -> Pooh;

test([a b c])=>
;;; MISHAP - ENP: EXECUTING NON-PROCEDURE
;;; INVOLVING:  999
;;; DOING    :  mishap C test compile
```

This *run-time* procedure check will slow programs down. In order to avoid it, users can declare certain identifiers to be of type procedure, so that the check is done only when a value is assigned to the identifier, not when the procdure is run. Such a declaration can be made as part of a variable declaration as specified in Chapter 5, which will commonly be placed at the beginning of a file, or it can be combined with the procedure definition, using the syntax of section 2.2. Thus you can say:

```
vars procedure Pooh;
```

or, if you have several procedures:

```
vars procedure(Pooh, Piglet);
```

Alternatively you can say:

```
define procedure Pooh(list);
    hd(tl(list))
enddefine;

define test(list);
    Pooh(list)
```

```
enddefine;

test([a b c]) =>
** b
```

So far as before. But now

```
999 -> Pooh;
    ;;; MISHAP - ASSIGNING NON-PROCEDURE TO PROCEDURE IDENTIFIER
    ;;; INVOLVING:  999 Pooh
    ;;; DOING    :  mishap compile
```

$popdefineprocedure$
As we have seen, in the default state of the POP system, procedure definitions do not create variables that are restricted to be of type procedure. However this can be changed. If the variable $popdefineprocedure$ is not $false$ then any procedure definition will create procedure variables that are restricted to being of type procedure.

Constants and lexical identifiers may also be declared to be of type procedure, as in

```
define constant procedure Pooh(...)
define lvars procedure Pooh(...)
define lconstant procedure Pooh(...)
```

2.11.2 Making procedure identifiers constant and/or lexical

It is possible for procedure identifiers to be constant, and for them to be lexical variables. These options will mostly be useful when you are modifying a working program to put it in a *library*, since they generally provide a saving in time and space but make debugging harder.

Making a procedure name a constant

popdefineconstant
If the variable *popdefineconstant* is not *false* then the procedure identifier created by a procedure definition will be a constant and not a variable. It is not usually a good idea to define constant procedures when you are first developing a program, since if you redefine them during debugging of a program you will not affect any calls that you previously made to them — so to make the redefined procedure take effect you will have to recompile all of the procedures that called it. Moreover, it will not be possible to *trace* the procedure However, using constant procedures will save an indirection in the procedure call, with a slight speed advantage.

Examples of how to define the procedure *Pooh* to be a constant are given below:

```
constant Pooh;
lconstant Pooh;
define constant Pooh;
define constant procedure Pooh;
define lconstant Pooh;
```

Making the procedure name a lexical identifier

By default, a procedure definition creates a *permanent* variable. However, procedure variables can themselves be *lexical*. This has two possible uses:

1. Declaring a procedure as a lexical variable in a file but outside of any other procedure declaration in effect makes the procedure inaccessible outside of the file. This means that there is no danger of a name-clash with procedures defined in other places. Thus auxiliary procedures, which do not need to be referenced outside of a library program, can have their names made unavailable outside. This provides a similar capability to that provided by *sections* (see Chapter 13) but is simpler to do, and uses less store.

   To indicate that a lexical identifier is required, follow *define* with one of

```
lvars       -   a lexical variable: may be re-assigned
lconstant   -   a lexical constant: may not be re-assigned
```

E.g.

```
define lvars proc(a,b) -> c;
define lconstant proc(a,b);
```

2. A procedure variable may be lexical because it is actually intended to *vary* in use. For example, in Chapter **??** we see that $maplist(L_1, P) \rightarrow L_2$ takes a list and a procedure and produces a new list. The procedure parameter of *maplist* should be declared to be a lexical.

```
define maplist(L,P);
  lvars L P;
    if null(L) then [] else
    P(L.hd)::maplist(L.tl,P)
    endif
enddefine;
```

Making P and L to be lexical avoids possible name clashes between non-local variables of the procedure provided as actual parameter to *maplist*.

2.11.3 Making the procedure name global to sections

As described in Chapter 13, *sections* provide a way of controlling the scope of permanent variables. If it is required that the procedure should be acessible in all *sections* below the one in which it is being defined, or to which it is exported, then the name should be declared as a *global* identifier. Hence it must *not* be *lexical*. E.g.

```
define global Pooh(a,b)
```

Constants and lexical identifiers may also be declared to be of type procedure, as in

```
define constant procedure Pooh(...)
define lvars procedure Pooh(...)
define lconstant procedure Pooh(...)
```

Infix operations, macro names, syntax words, and active identifiers, are automatically of type procedure. If the variable $popdefineprocedure$ is not $false$, then all procedure names are automatically declared to be of type "procedure".

Once an identifier has been declared to be of type procedure it cannot be re-declared not to be, e.g. using $vars$. This is because non-checking invocations of its value may already have been compiled.

2.12 The Body of a Procedure

In this section we consider how the *body* of a procedure is written.

```
<statement_sequence> = {<statement> <statement_sep>}
<statement_sep>      = ; | '=>' | '==>'
<statement>          = <expression_sequence>
<expression_sequence>= <expression>|<expression>,<expression_sequence>
<expression>         = <lambda> |
                       <literal> |
                       <expression> <operator> <expression>|
                       <syntax_operator_form>  |
                       <syntax_form>
```

The $< syntax_form >$ has the generic form

```
<syntax_opener><body>
```

2.12.1 Assignment

Ordinary assignment using $\rightarrow$ has been treated earlier. There is also a form:

```
<expression 1> ->> <expression 2>
```

which evaluates $< expression1 >$ and transfers the value to $< expression2 >$ without popping it off the stack. This is particularly useful with those POP predicates that return either $false$ or some non-false value that has some significance. Another use is illustrated by

```
    while (P_rep() ->> c) /= termin do ...
```

where the result of the character repeater procedure P_rep is tested for $termin$, which would indicate that the end-of-file has been reached, but the character is assigned to the variable c.

$\rightarrow;$
The assignment arrow appearing immediately before a semicolon takes the top item off the stack, and is equivalent to $erase()$, as described in section 2.5.2.

2.12.2 Forms which start with a syntax word

These have the form:

```
<syntax_form> = <syntax_opener> <body>
```

where $< syntax_opener >$ is a word whose *identprops* is "*syntax*", and whose value is a procedure. $< body >$ is dependent on the particular syntax opener, and can be one of the forms specified in the following table:

Syntax Form	Where described.
$< assignment >$	2.12.1
$< conditional >$	2.12.3
$< for\_iteration >$	2.15.1
$< while\_iteration >$	2.14
$< until\_iteration >$	2.14
$< repeat\_iteration >$	2.15
$< goto >$	2.20.1
$< go\_on >$	??
$< quit >$	2.19.2
$< next >$	2.20
$< return >$	2.12.4
$< quoted\_word >$	??
$< var\_prefix >$	?? ?? 2.28 2.7.1 ??
$< const\_construct >$	??
$< declaration >$	5??
$< section\_decl >$	13
$< definition >$	2
$< procedure >$	2

2.12.3 Conditional Forms

A *conditional expression* is one in which certain sub-expressions, the *conditions* are used to determine which of certain other sub-expressions are executed. For example:

```
if is_haycorns(food) then
    give_to(food,"Piglet")
else
    give_to(food,"Pooh")
endif
```

When this statement is executed, the POP11 system first executes the condition that is $is\_haycorns(food)$; if this evaluates to $false$ then $give\_to("Pooh")$ is executed, otherwise $give\_to("Piglet")$ is executed. The full syntax of allowable conditional expressions is quite complex, and is given below:

```
<conditional> =
    if     <condition> then <conditional_body  endif          |
    unless <condition> then <conditional_body> endunless

<conditional_body> =
     <expression_sequence>
     {<else_term><condition>then<expression_sequence>}*
     {else <expression_sequence>}

<else_term> = elseif | elseunless
<condition> = <expression>
```

Note the following about this syntax.

- If a conditional statement starts with *if* it must end with *endif*. If it starts with *unless* it must end with *endunless*. Either form may include *elseif*, *elseunless*, or *else* clauses.
- There can be as many *elseif* and *elseunless* clauses as needed, and they can be freely intermingled.
- There may be a final *else* clause.
- A $< condition >$ is any expression, which should normally return one value.

The execution of an *if* conditional is as follows: The $< condition >$ should, when executed, produce a result. If the result is other than $false$ then the *expression_sequence* following the *then* is executed, and after that, the $< expression\_sequence >$ which follows the *endif* (this may be a null $< expression\_sequence >$, in which case the procedure returns control to its caller).

If the result of the $<$ *condition* $>$ is *false*, then execution commences after the next *elseif*, *elseunless*, *else*, or *endif*. Execution following an *elseif* is the same as that following an *if* — the $<$ *condition* $>$ is executed and used to choose whether to carry on execution immediately after the *then* which follows the $<$ *condition* $>$, or whether to carry on after the $<$ *endif* $>$ which terminates the whole conditional expression. Note however that *elseif* can occur in a condition that begins with *unless*, so that it is possible in this case to resume execution after an *endunless*. Execution after an *elseunless* is the same as that following an *unless*, discussed below. The expression sequence following an *else* is executed, and then that following the *endif* or *endunless* that goes with the *else*.

Conditionals which begin with *unless* behave in the opposite way to those that begin with *if*, that is if the $<$ *condition* $>$ evaluates to *false* execution proceeds after the *then*, and if it evaluates to anything other than *false* execution proceeds after the next *elseif*, *elseunless*, *else* or *endunless*.

The course of execution of conditionals described above can of course be changed if an expression which causes a *return* from the current procedure, or a *quitting* from the current loop, or a *goto* is encountered.

Note particularly that if the the result of executing a $<$ *condition* $>$ is anything other than *false*, then it is treated as if it were *true*, e.g.:

```
if 3 then "three" endif =>
** three
```

An *elseif* clause of the form

```
elseif not (<condition>) then....
```

e.g.

```
elseif not(list matches[??x ison ??y]) then...
```

can be expressed as

```
elseunless <condition> then....
```

Note that *not* is a procedure and needs parentheses around its argument.

It is permissible, but unusual, for an $< expression\_sequence >$ to contain no code at all, for example:

```
define prlist(list);
    if  list == [] then
    else
        hd(list) =>
        prlist(tl(list));
    endif
enddefine
```

It is sometimes clearer to use *unless* in this case.

```
unless <condition> then <conditional-body> endunless
```

is equivalent to:

```
if not(<condition>) then <conditional-body> endif
```

A conditional expression behaves like any other expression, and in particular can be nested within another conditional, either in the $< condition >$ or in the $< conditional - body >$. For example

```
if if x=2 then true elseif x=4 then true else false endif then 4
else 3
endif
```

Although this is much better expressed as

```
 if x=2 or x=4 then 4 else 3 endif;
```

2.12.4 Returning control to the calling procedure

As we have stated earlier 2.6, when the $< expression\_sequence >$ that forms the body of a procedure P_2 has been executed, after a certain "tidying up", execution of the procedure P_1 (say) which called P_2 is resumed, immediately after the call of P_2. It is sometimes convenient to be able to accomplish this return of control to the calling procedure somewhere in the middle of the body of P_2. To this end, a special *return* construct is provided. There are two syntactic forms:

```
return
return(<expression\_sequence>)
```

In either case, the return from the procedure is accomplished in exactly the same way as if the execution of the body of the procedure had been accomplished in the usual way, that is the output locals are put on the stack and the state of the local variables of P_2 is restored to that which pertained prior to the call of P_2.

The second form of the *return* construct is really just a notational convenience, and is equivalent to:

```
<expression\_sequence> return
```

It is conventionally used to emphasise what values are actually returned by the procedure. If there are output locals, they may be assigned values in this $<$ *expression_sequence* $>$, and if the result of the procedure is just to be pushed on the stack, then it should also be done here. An example of the use of *return* is given below. The procedure *char_ident* is supposed to recognise a character that can form part of an identifier in a simple language:

```
define char_ident(c);
  if     c>= `a` and c=<`z` then
    return(true)
  endif;
  if c>= `A` and c=<`Z` then
    return(true)
  endif
  false
enddefine;
```

In many cases, including this one, the use of *return* is not justified. A neater construct would be

```
define char_ident(c);
  if     c>=`a` and c=<`z` then true
  elseif c>=`A` and c=<`Z` then true
  else false
  endif
enddefine
```

Look at section 2.7.1 and see if you can see how to write the above definition without using if at all — it is neater still.

2.12.5 Iteration

Iteration is the repetition of an $<$ *expression_sequence* $>$ a number of times in succession — with mathematical perversity we include in this the possibility of *zero* or *one* repetition.

We also use the more mechanical term *looping* for iteration. Iteration is often the most convenient way of performing a computation that depends upon a data-object whose length we do not know in advance. For example, we saw in Chapter ?? that we could add up the members of a list using a recursive definition:

```
define sum(L);
  if L=[] then 0 else L.hd + sum(L.tl)
  endif
enddefine;
```

or an iterative definition:

```
define sum(L) -> s;
  lvars l,L,s=0;
    for l in L do l+s -> s
    endfor
enddefine;
```

Many people will find the second definition more intuitive, although a mastery of the techniques of recursive programming is necessary for many applications of POP.

The most primitive kind of iterative construct is *goto* $< label >$ which allows you to specify that execution should immediately be recommenced at some new place in a procedure (or even outside it). This new place must be labelled by $< label >$:. It should almost never be used by human programmers, although it has a place in POP code that is being automatically generated by a procedure. Instead, POP provides a range of iterative constructs, all of which have a distinct bracketing, so that it is clear to the reader *what sequence of expressions is being repeated.*

The most general of these are the *while* and *until* constructs, which allow iteration to take place depending on some arbitrary $< condition >$ being satisfied. Where iteration is taking place over a finite sequence of numbers, or a list, it is more convenient to use the for construction.

It is sometimes convenient to be able to stop the execution of an iteration immediately. For example suppose we are forming the product of a list of numbers. If we encounter a zero in this list, we may stop the iteration, since we know that the product *must* be zero. To this end, constructions are provided to *quit* an iteration.

Finally, some constructs are provided to allow a programmer to specify that the execution $<$ *expression_sequence* $>$ should be started again immediately, provided that the $<$ *condition* $>$ holds.

2.13 However you may not need to have an explicit iteration

An alternative to using the iterative constructs described in this chapter is to make use of the procedures such as *sysrepeat*, *lit*, *maplist*, *applist*, *mapdata*, *appdata* described in Chapter **??**. These procedure typically take as arguments a structure and a procedure and apply the procedure systematically to the fields of the structure.

2.14 The while and until iterative constructs

The form of a *while* iteration is:

```
<while_iteration> =  while <condition> do <expression_sequence>
                     endwhile
```

To execute this, POP evaluates the $<$ *condition* $>$ and if it is true executes the $<$ *expression_sequence* $>$, then goes back to test the condition again. This iteration continues until the condition is false, for example

```
    10 -> n;
```

```
while n > 0 do
    ppr(n);
    n - 1 -> n
endwhile;
```

will print out:

```
10 9 8 7 6 5 4 3 2 1
```

The statement

```
<until_iteration> =  until <condition> do <expression_sequence>
                     enduntil
```

is equivalent to:

```
while not(<condition>) do <expression_sequence> endwhile
```

It evaluates the condition, and if it is $< false >$ executes the $< expression\_sequence >$. It then goes back to test the condition again. The iteration continues until the condition is not $< false >$. For example

```
10 -> n;
until n <= 0 do
    ppr(n)
    n - 1 -> n
enduntil;
```

will print out

```
10 9 8 7 6 5 4 3 2 1
```

2.15 Simple iteration using *repeat*

The word *repeat* is one way of building a loop in POP-11. The statement

```
<repeat_iteration> =  repeat {<expression> times} <expression_sequence>
                      endrepeat;
```

is executed as follows: If the < *expression* > is present it is evaluated and should give a number n (if not a *mishap* occurs). The < *expression_sequence* > is then performed the appropriate n times. If the < *expression* > is not present, then the sequence is repeated indefinitely (i.e. until it is interrupted by a *quit* or by you typing an interrupt character or power failure...).

For example:

```
repeat 4 times pr(".") endrepeat;
```

will print four dots. Indefinite repetition is illustrated by:

```
repeat
    [isnt this boring] =>
endrepeat;
```

2.15.1 Iteration through a sequence

The iteration constructs based on for all have the basic form

```
<for_iteration> = for <for_seq_spec> do <expression_sequence>
                  endfor
```

The $< for\_seq\_spec >$ serves two purposes: it determines whether the

$$< expression\_sequence >$$

will be evaluated, and binds some variables, which would normally occur in it. It should be noted that *in no case* does for declare a variable — declarations must be done separately[17] The permissable forms are:

```
for <forvars> in <lists> do <expression_sequence> endfor

for <forvars> on <lists> do <expression_sequence> endfor

for <variable> in <object> using_subscriptor <procedure_expr> do
    <expression_sequence>
endfor

for <variable> {from <number>} {by <number>} to <number> do
<expression_sequence>
endfor

for <expression_sequence> step <expression_sequence>
                          till <condition> do
<expression_sequence>
endfor
```

These different forms will be discussed separately below.

[17] In retrospect this seems to be a mistake — more recent languages like Ada do make for loops declare the iteration variables, and this, while it prohibits access to the values of the variables outside the loop, which can be useful in determining how a premature quitting from the loop may have occurred, is likely to catch certain errors which can arise in POP

Iteration through lists

To complete the syntactic specification of the $< for_iteration >$, in the case of an iteration through lists, we need:

```
<forvars> = <variable>{<variable>}*
<lists>   = <expression_sequence>
```

The $< forvars >$ should normally be distinct. We will denote the jth such variable by v_j. The $< lists >$ should evaluate to as many lists as there are $< forvars >$. We will denote the jth such list as L_j. The construct iterates through the lists in parallel. That is to say, at the ith iteration, each variable v_j is bound to the ith member of the corresponding list L_j, i.e. $L_j(i) \rightarrow v_j$. Iteration terminates when $i > min(length(L_j))$.

```
for x in [a b c] do x => endfor;
** a
** b
** c
```

Use of the multiple variable form to iterate over elements of several lists at once is illustrated by:

```
for x y z in [ a b c d], [ e f g], [ h i j k] do
    [^x ^y ^z] =>
endfor;
```

```
** [a e h]
** [b f i]
** [c g j]
```

Note that the loop terminates when the end of the shortest list has been reached.

The iterative construct:

```
for <forvars> on <lists> do <expression_sequence> endfor
```

should be used when access to the list cells that contain the data-objects which form the members of the lists is required — perhaps because it is necessary actually to update the list itself. Thus it differs from the construction $for < forvars > \ in \ < lists >$ only in that the variables are bound to the list-cells containing the members. E.g.

```
for x on [a b c] do x => endfor;
** [a b c]
** [b c]
** [c]
```

In other words, at iteration i, v_j is bound to $tl^i(L_j)$, where the exponentiation denotes repeated application of tl.

A case where we would need to use the *on* version is the following procedure which updates a list of numbers to be a list of the squares of the numbers:

```
define dest_square_list(L);
  lvars L_i,L;
    for L_i on L do
    lvars l = hd(L_i); l*l -> hd(L_i);
    endfor
enddefine;

/*example
vars L_num = [ 1 2 3];
dest_square_list(L_num);
L_num =>
** [1 4 9]
*/
```

An example of the use of the multiple variable form to iterate through several lists at once is:

```
for x y z on [ a b c d], [ e f g], [ h i j k] do
    [^x ^y ^z] =>
endfor;
```

```
** [[a b c d] [e f g] [h i j k]]
** [[b c d] [f g] [i j k]]
** [[c d] [g] [j k]]
```

As with "in" the loop terminates when the shortest list is exhausted.

2.16 Iteration through data-objects other than lists

If you need to iterate through any data-object other than a list, you need to specify to POP how it is to access the fields of the data-object. The iterative construct provided for this purpose is:

```
for <variable> in <expression> using_subscriptor <procedure_expr> do
    <expression_sequence>
endfor
```

Here $< expression >$ should evaluate to a single data-object Obj, and $< procedure\_expr >$ should evaluate to a procedure P_{sub} which subscripts the object, i.e. $P_{sub}(i, Obj)$ is defined for $1 =< i =< length(Obj)$. The $< expression\_sequence >$ is iterated $length(Obj)$ times, binding the $< variable > v = P_{sub}(i)$ at the ith iteration.

```
for item in {[a] vector [of words] and [lists]}
```

```
using_subscriptor subscrv
do item=>
endfor;

** [a]
** vector
** [of words]
** and
** [lists]
```

2.17 Iteration over a sequence of numbers

The iterative construct:

```
for <variable> {from <expression 1>}
               {by   <expression 2>}
                to   <expression 3>
do
                     <expression_sequence>
endfor
```

binds the $< variable >$ to the result of evaluating $< expression1 >$. If $< expression2 >$ is positive it then performs $< expression\_sequence >$; $< expression2 >$ is then re-evaluated and added to the $< variable >$. If it is less than or equal to $< expression3 >$ the actions are performed again. The addition, testing and execution of actions is repeated until the variable exceeds the value of the third expression.

If $< expression2 >$ was negative (before the loop was entered) then the termination test uses $<$ instead of $>$.

The *by* and $from$ portions may both be omitted; in either case, the missing value is assumed to be 1.

2.18 The most general for construction

The iterative construct:

```
for <action 1> step <action 2> till
<condition> do <expression_sequence>
endfor;
```

Here $< action1 >$ and $< action2 >$ are just $< expression\_sequence >$'s. POP first executes $< action1 >$. Next $< condition >$ is checked to make sure it is $false$. If it is $false$, the $< expression\_sequence >$ is repeatedly executed, each execution being followed by the execution of $< action2 >$ and the checking of the condition. This continues until the $< condition >$ no longer evaluates to $false$, when the execution of the for loop is terminated. Thus:

```
for "steve" -> person step father(person) -> person
    till person = "adam" do
        person =>
endfor
```

would print out:

```
** steve
** frank
** tom
.....
** cain
```

2.19 Equivalence of the iterative constructs

Notice that the first three forms of the *for* command could all be accomplished using the last, thus:

```
for <variable> in <expression> do <expression_sequence>
endfor
```

is equivalent to:

```
for <expression> -> <temporary_variable>
    step tl(<temporary_variable>) -> <temporary_variable>
    till <temporary_variable> = []
do
    hd(<temporary_variable>) -> <variable>;
    <expression_sequence>
endfor
```

and:

```
for <variable> on <expression> do <expression_sequence>
endfor;
```

is equivalent to:

```
for <expression> -> <variable>
    step tl(<variable>) -> <variable>
    till <variable> = []
do
    <expression_sequence>
endfor
```

and:

```
for <variable> from <expression1>
               by   <expression2>
               to   <expression3>
               do
                    <expression_sequence>
endfor
```

is equivalent (when $< expression2 >$ is positive) to:

```
for <expression1> -> <variable>
    step <variable> + <expression2> -> <variable>
    till <variable> > <expression_3>
do
    <expression_sequence>
endfor
```

Finally, the for ... *step* format can be re-written using *until*, thus:

```
<action1>;
until <condition> do
    <expression_sequence>;
    <action2>
enduntil
```

2.19.1 Iteratively stacking up values

The *for* construction can be used to push or pop a variable number of items on the stack. For example:

```
for x from 1 to 10 do x endfor;
```

leaves the integers 1 to 10 on the stack. A *for* loop, like any POP-11 code, can be enclosed in decorated list or vector brackets. An example is:

```
[% for x from 1 to 10 do x endfor %] =>
** [1 2 3 4 5 6 7 8 9 10]
```

The *while*, *until* and *repeat* constructs can be used similarly. This technique is often useful when combined with variadic procedures like *consword*, described in **??**.

2.19.2 Interrupting an iteration

As we have mentioned above, it is possible to stop the execution of a POP iteration by using a *quitting* construct. The most basic is *quitloop* which is equivalent to *goto* < *label* >, where the < *label* > is placed just after the terminating bracket (*endwhile*, *endfor*, *enduntil* or *endrepeat*) of the iterative construct.

For example:

```
define Pi_list(L)->x;
  lvars l, L, x=1;
```

```
    for l in L do
      if l=0 then 0->x; quitloop
      endif;
    x*l->x;
    endfor
enddefine;

/*example
Pi_list([2 3 4]) =>
** 24
Pi_list([2 0 4]) =>
** 0
```

Of course, we could have replaced the *quitloop* construction by

```
return(0->x)
```

We can also quit a number of loops at once using this construction:

```
  quitloop(<integer>)
```

where $< integer >$ must be a positive integer ($= n$ say), and *not* an expression, causes POP to quit the n loops which enclose the expression. N.B. *quitloop* is not a procedure and cannot be called by a procedure inside the loop — it must be used inside the loop body itself.

The statement:

```
    quitif(<expression>)
```

translates to:

```
if <expression> then quitloop endif;
```

The parentheses surrounding the expression are essential.

To quit the $< integer >$th enclosing loop do:

```
quitif(<expression>)(<integer>)
```

There is a *quitunless* form which is analogous to the $quitif$ form.

2.20 Causing the next iteration to happen immediately

It is possible to get POP to begin a new iteration immediately by using the construct:

nextloop($<$ *integer* $>$)
nextloop

If the $<$ *integer* $>$ is absent, then control jumps to just before the syntax word closing the loop, for example *endwhile* in a *while* loop, and *endfor* in a *for* loop. This causes the enclosing loop to be restarted.

If an $<$ *integer* $>$, n say, is present, control is transferred to just before the syntax word closing the n'th enclosing loop.

N.B: *nextloop* is not a procedure and cannot be called by a procedure inside the loop: it must be used in the loop body itself. Moreover, the integer cannot be represented by a variable or an expression evaluating to an integer. If you need a 'computed jump', this effect can be achieved using *go_on*.

There are also conditional forms:

```
nextif(<condition>){(<integer>)}
```

is equivalent to

```
  if <condition> then nextloop{(<integer>)} endif
```

and

```
nextunless(<condition>){(<integer>)}
```

is equivalent to

```
  unless <condition> then nextloop{(<integer>)} endunless
```

2.20.1 Arbitrary transfers of control

It is possible to cause POP to 'transfer control' to almost any place in the current procedure, or any procedure which has called it. When POP encounters the $goto < label >$ statement statement below, it will not execute the statement following, but will execute the statement which has been 'labelled' using the following syntax:

```
<statement> = <label>:<statement> | <label>:*<statement>
```

For most purposes the form of label which uses just a colon is appropriate. Certain non-local jumps are made more efficient by using the second form, as described in Chapter ??.

A *goto* statement in POP-11 transfers control to the instruction labelled by the following word. For example:

```
define laugh();
    l: ppr("ho");
    goto l
enddefine;
```

```
laugh();
ho ho ho ho ho ho ho ho ho ........
```

Here l : labels the following instruction and *goto* l transfers control to the labelled point. Labels in POP-11 are represented by any legal identifier followed by a colon, e.g.

```
    loop:
    l3:
    +++ :
```

Naturally, space is needed if the label consists of *sign* characters — see Chapter 19.

The label referred to in a *goto* statement may be either a local label (somewhere in the current procedure) or a non-local one (somewhere in a lexically-enclosing procedure). A non-local *goto* is like doing an *exitto* to the target procedure, as described in section 2.24, followed by a local jump to the label.

The use of labels is hardly ever necessary. If you have to use labels to write a program this may be because you don't really understand the problem the program is solving, or you haven't thought hard enough about how to express the program clearly. Occasionally *goto* is useful, e.g. as a way of representing a *finite state machine*, but even then there may be better ways of doing it.

Non-local *goto* is sometimes useful for terminating a search deep in recursion. For details see Chapter 16 describing the implementation of non-local jumps.

A *go_on* statement in POP-11 allows you to 'switch' control to one of several labels depending on an integer value an expression. The form is

```
<go_on_expr> = go_on <expression> to <labseq> { else <word>};
<labseq>     = word {word}*
```

$< labseq >$ is thus a sequence of words (w_i) which should be labels. $< expression >$ should evaluate to produce an integer i from 1 to n. Control is transferred to the label w_i. If i is not between 1 and n inclusive, then $< word >$, which should be a label, is jumped to if present, otherwise a mishap results. A mishap will also result if $< expression >$ doesn't produce an integer.

All the labels specified, including $< elselab >$, may be either local to the current procedure or non-local, that is they are somewhere in a lexically-enclosing procedure.

2.21 Procedures as Data Objects

It is recommended that you read Chapter 3.7 before you read this section, which is placed in this chapter to provide a complete description of procedures.

procedure_key
Like any other object, a procedure has a *datakey*, $< keyprocedure >$. This key is the value of the constant *procedure_key*.

The main bulk of information contained in a procedure object is the code which defines the behavior of the procedure. This may be actual *native machine code* i.e. the instructions that the computer hardware executes, or it may be *POP Virtual Machine Code* which is *interpreted* by an interpreter program. POPLOG systems normally use *native machine code* and current AlphaPop interprets VM code.

2.21.1 The fields of a procedure

Apart from the code of a procedure P it has a number of other fields. These are:

$updater(P_1) \rightarrow P_2$
is another procedure which is called instead of P when P occurs as the immediate object of an assignment statement. The statement $\rightarrow P(a_1, a_2, ...a_n)$ causes the arguments $a_1...a_n$ to be evaluated in the usual way, but instead of P being called, its updater is. The procedure *updater* itself has an updater, so that the statement $P_{ud} \rightarrow updater(P)$; can be used to change the updater of the procedure P to be the procedure P_{ud}. The default value for the updater of a procedure is $false$, and attempting to use the updater of a procedure with false as an updater will cause a mishap. Note also that an updater will usually have one more argument than the procedure it is the updater of.

From: Aaron Sloman ¡aaronsDate: Sun, 11 Nov 90 13:17:35 GMT Cc: pop-forum@lb.hp.CO.UK

Robin's message to popforum said: ¿ ¿ Should pdnargs have an updater?

It does in the latest version V13.9 (shortly V14).

The HELP file is out of date and should be corrected. REF procedure now states

pdnargs(P) -¿ N [procedure] N -¿ pdnargs(P) Returns or updates the number of arguments N of the procedure P, where N is an integer in the range 0 - 254. When P is a proper procedure, the default value of N is the number of formal arguments given when the procedure was constructed. A closure, on the other hand, is initially set up so that until an explicit value is assigned to it with the updater of -pdnargs-, the value returned is pdnargs(pdpart(P)) - datalength(P)

i.e. the number of arguments of its -pdpart- minus the number of frozen values. For either a procedure or a closure, assigning an explicit value to its -pdnargs- causes that value to be returned by -pdnargs- thereafter.

Aaron

$pdnargs(P) \rightarrow n$
This procedure returns the number of arguments of the procedure P. If P a procedure defined using the $define$ or *procedure* syntax, $pdnargs(P)$ is initialised to the number of formal arguments given in its definition; if P is a closure created by partial application (see section 2.25.1 on closures):

$$pdnargs(P) = pdnargs(pdpart(P)) - datalength(P)$$

i.e. the number of arguments of its *pdpart* minus the number of frozen values. You can assign to $pdnargs(P)$ — this can be useful if you are making some special use of the stack. Note that variadic procedures will have a value of $pdnargs$ which doesn't truly indicate what they do to the stack.

pdprops$(P) \rightarrow O$
The object O that this procedure returns can be any data object. It is initialised by the $define$ syntax to be the word which is the name of the procedure; other procedure constructions initialise it to $false$. It can be updated to be any data object. *pdprops* can be used to attach various additional data to a procedure, although it is usually better to use a *property* procedure for this purpose (see chapter 11) since this is less likely to cause interaction between diferent uses. The default printing procedure *sys_syspr* will print a procedure P with *pdprops*$(p) = w_P$ as $< procedure\ w_P >$.

In creating a procedure, the user can specify the *pdprops* and $pdnargs$ fields of the procedure data object to be other than the default values by using the construction

```
<with_spec> ::- with_props <word> {<with_spec>} |
                with_nargs <integer> {<with_spec>}
```

2.21.2 Predicates on Procedures

The following predicates are available for recognising procedures, and distinguishing different kinds of procedures. Closures are described in section 2.25.1, arrays in Chapter 10.

$isprocedure(Obj) = true$ iff Obj is a procedure (i.e. a proper procedure or a closure), $false$ if not.

$isclosure(Obj) = true$ iff Obj is a closure, $false$ if not.

$isarray(Obj) = true$ iff Obj is an array, $false$ if not.

2.22 Generic Dataobject Procedures on Procedures and Closures

The generic dataobject procedures described in Chapter 3.7 (*datalength*, *appdata*, *explode*, $fill$, etc, and others defined in terms of those) are all applicable to closures (see section 2.25.1) but not to proper procedures. They treat a closure as if it were a vector of its $frozvals$, e.g. if a closure P_{close} has n frozen values, then $datalength(P_{close}) = n$.

copy may be applied to proper procedures and closures, providing they are user constructed ones, i.e. not part of the system. Note that, as usual with *copy*, only the top-level object is copied. Thus if you copy a closure the *pdpart* and the $frozvals$ remain identical with those of original.

The concatenation operator <> is also applicable to all kinds of procedures (see the chapter on data-objects for its action on other data types). Thus $P_1 <> P_2 \rightarrow P_3$; constructs a (proper) procedure which is the functional composition of its arguments P_1 and P_2, i.e. P_3 is a procedure which will call P_1 followed by P_2. P_3 is thus equivalent to

```
procedure();
```

```
    P_1();
    P_2()
endprocedure;
```

If P_2 has an updater, then the updater of P_3 will be a procedure which calls P_1 and then calls the updater of P_2, i.e.

```
procedure();
    P_1();
    -> P_2()
endprocedure;
```

If P_2 has no updater then neither will P_3.

2.23 Procedures which tell you how a procedure was called

A procedure can discover how it itself was called. This is particularly useful in developing debugging tools — it allows you to define a new procedure for handling mishaps which allows the user to discover how the mishap occurred.

All of the procedures described in this section examine the *call-stack* to find the details of how a particular procedure call occurred. The call-stack is organised into *frames* each of which corresponds to one procedure call. The frame is created on entry to the procedure, during the process of setting up its local variables, and is freed on return from the procedure.

$caller(n)$ is the n'th procedure up the calling chain from the current procedure. I.e. $caller(0)$ is the current procedure, and $caller(n)$ is the procedure which called $caller(n-1)$. $caller(n) = false$ if there are less then n procedures in the calling chain above the current one. For example:

```
define fred(x);
  joe(x);
enddefine;

define joe(x);
lvars i;
  for i from 0 to x do caller(i) =>
  endfor
enddefine;

fred(4);
** <procedure joe>
** <procedure fred>
** <procedure>
** <procedure sysEXECUTE>
** <procedure pop11_exec_stmnt_seq_to>
```

You may find procedures on the end of the calling chain that you did not expect to! What you are seeing in the last 3 lines of the above output is the sub-procedures of the POP compiler which were involved compiling the expression $fred(4)$. Note that the standard *prmishap* procedure suppresses the printing of some of the procedures on the calling chain.

A procedure *iscaller* is provided to assist in interrogating the call stack, e.g. to find out how many recursive calls of a given procedure have been made. $iscaller(P, m)$ is the caller number of P in the calling chain, starting from $caller(m)$, and is $false$ if P is not in the calling chain. $iscaller(P) = iscaller(P, 0)$.

E.g. a procedure to count the number of currently active calls of a given procedure P:

```
define count_calls(p) -> count;
    lvars p, count = 0, n = 0;
    while iscaller(p, n) ->> n do
        count + 1 -> count;
        n + 1 -> n          ;;; skips this call of p
    endwhile
enddefine;
```

syscallers() is a list of all procedures currently in the calling chain, starting with caller(2) inside *syscallers* (i.e. the caller of the procedure calling *syscallers*). It is defined as:

```
define syscallers() -> L;
    lvars L, n = 2, p;
    [%  while caller(n) ->> p do
            p;
            n + 1 -> n
        endwhile
    %] -> L
enddefine;
```

We now consider a procedure which allows us to access dynamic local variables in the current call context. Recall that if w is a word, then $valof(w)$ is the current value of the variable named by w 5.3. $caller\_valof(w, P_{caller})$ is the $valof$ of the word w as it would be in the environment of the currently-active procedure specified by P_{caller}. The argument P_{caller} may be either

- An actual procedure or a caller number as input to *caller*
- $false$, meaning that the value outside all procedure calls is denoted.

This procedure has an updater. [N.B. in POPLOG, as of September 1988, this procedure does not deal with active variables].

The procedure call $set\_global\_valof(Obj, w)$ uses $caller\_valof$ to assign Obj to be the valof w in the context of every currently active procedure for which the identifier associated with w is a dynamic local.

$callstacklength(n)$ is the length of the call stack at the stack frame for the n-th caller of the current procedure. [18]

[18] The procedures which refer to the length of either stack give the length in terms of POP data objects, which in all current implementations occupy 32 bits each

pop_callstack_lim This (active) variable holds an integer specifying the maximum length to which the call stack may expand when this value is exceeded, the mishap 'RLE: RECURSION LEVEL EXCEEDED' results. Its default value is 90000, but for very deeply nested recursive programs you may need to assign it a larger value.

2.24 Non-Standard ways of calling procedures

On occasions it is desirable to be able to return from the call of a procedure in a non-standard way. The simplest case in which this occurs is the procedure *setpop* which is called by default when a mishap occurs, and which returns from all the procedures that the user has called without executing any further code (and which clears the stack). Sometimes it is desirable to have a more controlled exit than this (for example to carry on in a defined way after a mishap), and the procedures defined in this section, apart from *apply*, provide for this need.

The procedure call $apply(P)$ calls the procedure P. If P has known arguments, it is better to write $apply(a_1, a_2...a_n, P)$, but since this is equivalent to $P(a_1, a_2, ...a_n)$ you will hardly want to write it. Most uses of *apply* are now obsolete — early versions of POP did not allow syntactic constructions such as $hd(L)(2)$, and required you to write $apply(2, hd(L))$ instead. However an implicit call of imply is generated whenever a computed procedure-object is applied, as in $hd(L)(2)$. However you might on occasion want to supply *apply* as an argument to a procedure. *apply* has an updater, with the obvious meaning that $\rightarrow apply(P)$ calls the updater of P.

The procedure call $chain(P)$ returns from the current procedure, restoring the environment of its caller, and then calls the procedure P. Thus P is effectively 'chained' onto the current procedure.

The procedure call $chainfrom(target, P)$ returns back up the calling chain until immediately inside a call of the $target$ procedure specified by $target$, exits from this call, and then calls the procedure P. The argument $target$ may be either an actual procedure (in which case exiting terminates on reaching a call of the given procedure), or a call stack length as returned by *callstacklength* (in which exiting terminates when the call stack length is $\leq target$).

The procedure call $chainto(target, P)$ returns back up the calling chain until immediately inside a call of the $target$ procedure specified by $target$, and then calls the procedure P. The value of $target$ is as for $chainfrom$, i.e. an explicit procedure or a call stack length.

The following demonstrates the use of *caller* and *chainto* to modify the system error procedure, so that when the user attempts to do certain arithmetic operations on items which are not numbers, the result is a "formal combination" of the items, rather than an error. A list of the operations which are to be formally combined is contained in $simp\_symbolic\_fns$.

```
vars old_prmishap = prmishap;

define simp_prmishap(Message,Culprits);

lvars c f, n = 3, name;

    while not( pdprops(caller(n)->>f) ->> name) do n+1 -> n
    endwhile;
    unless member(name, simp_symbolic_fns) then
    return(old_prmishap(Message,Culprits))
    endunless;

 prolog_maketerm(
        for c in Culprits do c.pr; 3.sp;
          if c.isundef then c.undefword else c
          endif
        endfor,
      pdprops(f),
      length(Culprits));
 chainto(f,identfn);
enddefine;

simp_prmishap -> prmishap;
```

The procedure calls $exitfrom(target)$ and $exitto(target)$ are equivalent to $chainfrom(target, identfn)$ and $chainto(target, identfn)$ respectively.

$jumpout(P, n) \rightarrow P_{jumpout}$
This procedure creates a new procedure $P_{jumpout}$ which when applied will cause a return from all procedures up to and including the procedure that called $jumpout$ in the first place. Before effecting the return, $P_{jumpout}$ first calls the given procedure P (with no arguments), and this is expected to return n results. If the user stack length excluding those top n items is then greater than it was at the time of the $jumpout$ call, a sufficient number of items above the top n are removed to reset it to that value. I.e. when the exit is effected, the user stack should be in its state at the time of the $jumpout$, but with the n results added. $jumpout$ is a library procedure defined using the chaining operations defined above. [19].

There are two library procedures $catch$ and $throw$ which are used to provide an exit to a place defined by a pattern. Their calls are:

$$catch(P, P_{caught}, O_{catch})$$
$$throw(O_{throw})$$

These two procedure work in conjunction: $catch$ 'catches' calls of $throw$. $catch$ applies the procedure P, which make then take further arguments off the stack, inside an environment which retains the values of the P_{caught} and $pattern_{catch}$ arguments for later use with a call of $throw$ occurring inside P or inside procedures that it calls, etc. Such a call of $throw$ then 'throws' the argument O_{throw} to the most recent call of $catch$ that will catch it, that is, it repeatedly exits through all procedures upto the next call of $catch$, until it reaches a call of $catch$ for which

$$O_{throw} \ matches \ O_{catch}$$

is true. When this happens, the P_{caught} argument to the $catch$ call is then applied if it is a procedure, or simply returned as result from that $catch$ otherwise. A mishap results if there is no call of $catch$ whose O_{catch} matches O_{throw}.

An application of $catch$ and $throw$ is discussed in Chapter 25 on the implementation of Prolog.

[19] $jumpout$ was a feature of POP-2 inspired by Landin's J operator [?]

2.25 How procedures can make other procedures

One of the crucial ideas of modern computing is that due to Von Neumann that *program and data can occupy the same storage*. Thus at the lowest level of computing these two are the same. This identification is often lost in high level languages — a Pascal program cannot create a Pascal procedure which it itself procedes to call. It can of course generate a text file which is a procedure to be called. That is Pascal can be used to write compilers, but one of the essential steps in executing the compiled program has to be done by the operating system, namely the *linking, loading and execution of the compiled program*.

LISP originally maintained the dual nature of program and data by making use of an interpreter to execute LISP programs which are simply lists — essentially the same as POP lists, which are indeed modelled on LISP lists. However, in order to be able to obtain greater speed, it is common to compile LISP into machine code.

In POP there are 3 ways in which a procedure can generate other procedures. These are

1. A procedure can generate text, either as a file, or as a string which is converted into a repeater procedure. It then compiles this text using the procedure *compile*.
2. A procedure can generate a list, which is then compiled by the procedure *popval*.
3. A procedure can be in some sense 'specialised' by having some of its variables bound. This is called creating a *closure*.

Of these (1) and (2) do not need any special discussion in this chapter. They differ in that (1) generates program text at a fine grain of characters, whereas (2) generates program text as items, or *tokens* as they are called in the literature on compilers [**?**]ompilers. We shall use them in subsequent chapters. We discuss (3) below.

2.25.1 Closures

We create a *closure* P_{close} from a procedure P by binding some of its variables so that they have determined values. There are two ways in which this can be done in POP.

- We can bind some of parameters of a procedure, by using a process referred to as *partial application* [20].
- We can bind the non-local lexical variables of a procedure P_2 which are local to an enclosing procedure P_1 by the simple act of calling P_1

The second way of creating closures corresponds to that of CPL [21], Scheme and Common LISP, and generally results in clear, but less efficient, code [22].

The usual construction for partial application is

```
<expression>(% <expression>{,<expression>}* %)
```

The first $< expression >$ must evaluate to a procedure-object, with say m arguments. The result is a *closure* which is the procedure-object with its last n arguments bound (or *frozen*) to be the n objects left on the stack by the execution of the expressions between the (% and %) parentheses. For example:

```
nonop + (% 1 %)
```

is equivalent to $procedure(x);\ 1 + x\ endprocedure.$

A closure can be thought of as a kind of procedure object which pushes the frozen arguments on the stack and then immediately calls the procedure from which the closure was constructed. In practice it differs in that some of the overhead of calling it as a separate procedure can be avoided, since there is no need to return to the closure after the procedure has been called. In addition, as described below, the frozen arguments can be accessed.

[20] This has always been the term used in POP since it was introduced in POP-2, although *partial calling* might be a little less contentious from the mathematcial point of view

[21] Combined Programming Language, or possible Christopher's Programming Language — a precursor of POP and C

[22] Thus it is possible to eliminate free variables from procedure bodies by using partial application. In the functional programming literature this is known as the technique of using *supercombinators* [?]

For example, you can define the *combinator* C by

```
define C(c);
  identfn(%c%)
enddefine
/*example C
vars two = C(2);
two() =>
*/
```

It is possible to access the procedure-object from which a closure was created, and its frozen values: $pdpart(P_{close})$ is the procedure part of the closure P_{close}, i.e. the procedure on which the closure P_{close} was constructed. Note that P_{close} can be a proper procedure, in which case $false$ is returned. *pdpart* can be used in the update sense.

$frozval(n, P_{close})$ is the n-th frozen value of the closure P_{close}. $frozval$ can be used in the update sense.

$partapply(P, L) \rightarrow P_{close}$ Another way of creating a closure is to use the procedure *partapply*. This constructs and returns a closure whose *pdpart* is the procedure P, and whose frozen values are the elements of the list L. This is equivalent to

```
P(%x,y,z,...%)  where  L = [%x,y,z,...%]
```

If P has an updater P_{ud}, then the updater of the constructed closure will be $partapply(P_{ud}, L)$

Closures are very useful, but this is not manifest at first sight! You will find examples of how they can be used in the chapter on Lists **??** and the chapter on properties 11.

The most perspicuous technique for generating a closure is to make use of lexical variables non-locally. For example, we could define a procedure *addn* by:

```
define addn(n);
```

```
  lvars n;
  procedure m; m+n
  endprocedure
enddefine;

/*example
vars add23 = addn(23);
add23(2) =>
** 25
*/
```

A more complicated example is a procedure that makes a content repeater for a vector (a procedure which will return the next content in the vector each time it is invoked).

```
    define vectorin(vector) -> P;
    lvars vector, P,
        i  = 1,
        n  = datalength(vector),
        P  = procedure;             ;;; The result of vectorin
               if i > n then        ;;; Have we exhausted vec?
               termin               ;;; Yes - return termin
               else                 ;;; No -
               subscrv(i, vector);  ;;; return next item
               i + 1 -> i;          ;;; increment index i
               endif;
             endprocedure;
    enddefine;

/*example vectorin
    vars rep;
    vectorin({1 2 3}) -> rep;
    rep() =>
    ** 1
    rep() =>
    ** 2
    rep() =>
    ** 3
```

```
    rep() =>
    ** <termin>
*/
```

2.26 Modifying the syntax of POP — Macros and Syntax procedures

You can create new syntactic forms in POP by using macros and syntax procedures. Both of these are essentially procedures that are called as soon as their names are encountered by the POP compiler, rather than being called when the procedure being defined is run. These capabilities should be used with caution, since they may give rise to programs that, while they seem elegant to the creator, may be hard to understand for a reader who does not appreciate that they are being used.

2.26.1 Macros

A macro definition looks just like a normal procedure definition, except that the word *macro* occurs after the $define$. The recommended syntax for a macro procedure is:

```
<def_macro>  =  define macro <word 1> {<macargs>};
                <statement_sequence>
                enddefine;
<macargs>    =  <word> | <word><macargs>
```

Notice that the arguments are *not* separated by commas or put in parentheses. Result variables (if any) are specified as for normal procedures though it is unusual for macros to have result variables. The word $< word1 >$ will be declared as a macro variable. A procedure-object is created to be the value of the variable, but as soon as the POP compiler reads its name it immediately calls the procedure, rather than taking its normal action of planting VMCODE to call the procedure, or push it on the stack depending on context. [23] This procedure may read things from the input item-stream, perhaps using *itemread* (see Chapter ??), and the results of the procedure are inserted into the input stream in place of the macro name and items read.

For example:

```
define macro swap;
    vars x y;
    itemread() -> x;
    itemread() -> y;
    x, ",", y, "->", x, "->", y;
enddefine;
```

Creates a macro *swap* which can be called thus:

```
swap a b;
```

which is equivalent to

```
a, b -> a -> b;
```

Any POP11 word may be used as a macro name provided it has not already been declared as an ordinary variable or procedure name. Once used as a macro a word cannot later be used as an ordinary variable without being explicitly reintroduced by a *vars* statement or being cancelled.

Words can be declared to be macros by using the prefix *macro* in a global variable declaration. If a word declared as a macro name turns out not to have a procedure as its

[23] Thus POP macros differ from LISP macros in that the latter take as argument a the list in which they occur.

associated value, it is treated just like a procedure that always produces that value as its result. That is, when the compiler hits the word, it just adds the value to the front of the input stream. For instance:

```
vars macro pi; 3.1416 -> nonmac pi;
```

is a slightly more concise way of getting the effect of:

```
define macro pi;
    3.1416
enddefine;
```

This is different from:

```
vars pi; 3.1416 -> pi;
```

The difference is that if you have a program with *pi* defined as this macro, the compiler will effectively substitute the number 3.1416 for every occurrence of the word *pi* in the program. So the procedure:

```
define circle_area(r);
    pi * r**2
enddefine;
```

would be saying "To find the area of a circle, multiply 3.1416 with the square of the radius". If *pi* was an ordinary variable, it would say instead "to find the area of a circle, multiply the value of the variable *pi* with the square of the radius". In this case there would be nothing to stop the value of *pi* changing between calls of *circle_area*. So the difference is that in the macro case the looking-up of the value is done while the program is being compiled, and not while the program is being run.

If the value associated with a macro is a list, the compiler puts all the elements of the list individually on the front of the input stream, rather than just considering the list as a single item. For instance, each time the macro

```
vars macro debug;
[if debugging then database ==> endif;] -> nonmac debug;
```

is used, a conditional statement will be inserted into the program. This statement has the effect that, when the program is running, if the variable *debugging* has a non-*false* value then the *database* (see **??**) will be printed out at this point.

Macros created with *define* can have formal parameters. These are set by calling *itemread* the appropriate number of times before invoking the macro. Thus

```
define macro swap x y;
    x, ",", y, "->", x, "->", y;
enddefine;
```

is equivalent to the definition give earlier.

Note that the *itemread* procedure *expands macros*, that is to say, any macro names that are encountered by *itemread* will have the corresponding procedures executed, or values substituted. So the parameter mechanism should not be used for macros which may themselves read in macros, but don't want to *expand* them. Instead, use *readitem*, which just reads text items off *proglist* without expanding macro items.

POP11 macros can be traced. e.g. *tracePooh*; will change *Pooh* so that when it runs it shows the text is is creating. This is described in chapter 4.

A detailed account of macro expansion is given in Chapter 15.2.2. If you want to avail yourself of the full capibilities of the POP system in defining macros, you should read Chapter 15. An extended macro definition is given in Chapter **??**.

Warning - if you are using code planting procedures described in Chapter 16 you should use syntax procedures and not macros, or code may be planted in places you didn't intend.

2.27 Syntax Procedures

The recommended form for the header-line of a syntax procedure is:

```
define syntax <name>;
```

Syntax procedures should have neither arguments nor results. They achieve their effects by calling compiler routines to plant VMCODE instructions. Like macros, syntax procedures are called at compile time. System syntax words, such as if , $define$, *until* correspond to syntax procedures.

```
define syntax Pooh;
    <expression_sequence>
enddefine;

define syntax 5 Pooh;
    <expression_sequence>
enddefine;
```

Each of these defines "*Pooh*" as a syntax word, the latter as a syntax operator of precedence 5.

When syntax words have a precedence this is used in parsing the input stream during compilation. E.g ″(″ is a syntax word of precdence -1, ″ →″ a syntax word of precedence 11. See 5.3.5 for a the meaning of these values. While syntax words and macros can do rather similar things, there is an especial advantage in using syntax words with precedence in that you will get more informative error messages.

When a syntax identifier is read by the compiler, the procedure it names is run immediately. Normally this will call code-planting procedures of the kinds defined in 16 to compile an expression, or expression sequence, or perform declarations, etc. $ObjREAD$ does not run syntax procedures itself. Closing brackets can be declared as syntax identifiers, in order that they may trigger syntax checking.

For an example of the definition of a new syntax word for looping over items in the POP-11 database, see Chapter 16, where a complete definition of the $foreach$ construction is given.

2.28 Active variables

2.28.1 Description

Active variables, which strictly speaking should be called active identifiers, are those whose names are used as if they were ordinary identifiers, but which in fact are associated with procedures.

The base procedure is run when the identifier is used normally and the updater is run when the identifier occurs on the right of "→".

This makes it possible to associate side-effects with the process of accessing or updating the identifier. E.g. the updater can call the error handler if the wrong type of object is assigned. Alternatively the procedures can keep track of the number of accesses.

An active identifier has a multiplicity specifying how many results it produces when accessed, and how many items have to be supplied when it is updated. Active identifiers can be made local to a procedure with *dlocal*, as described in section **??**. In this case, the multiplicity is used to determine storage requirements for the saved values.

Active variables can be declared with a variable declaration, whose syntax is given in Chapter 5, or a procedure definition, whose syntax is given in section **??** can be used to define an active variable. The word *active*, possibly followed by a $< multiplicity >$ is the indication that an active variable is being defined. If no multiplicity is specified, it is taken to be 1.

For example:

```
vars active av1;
vars active:3 av2;
```

declares two active variables, $av1$ with multiplicity 1, and $av2$ with multiplicity 3.

Let us now define an active variable $av3$ to store three values, and to count the number of accesses in the permanent variable acc_av3 and the number of updates in the permanent variable upd_av3. The three values will be stored in an inaccessible vector held in a lexical identifier vec_av3.

```
vars acc_av3 = 0, upd_av3 = 0;

lconstant L_av3 = [1 2 3];

define active:3 av3;
    acc_av3 + 1 -> acc_av3;         ;;; Increment access count
    explode(L_av3)                  ;;; Return all members of L_av3.
enddefine;

define updaterof active:3 av3(x1,x2,x3);
    lvars x1,x2,x3;
    x1,x2,x3 -> explode(L_av3);     ;;; Update all members of L_av3
    upd_av3 + 1 -> upd_av3;         ;;; Increment update count
enddefine;

av3 =>
** 1 2 3

4,5,6 -> av3;
```

```
av3 =>
** 4 5 6

av3 + 4 -> av3;

av3 =>
** 4 5 10
```

For a description of *explode*, see 3.11.

How many times has *av*3 been accessed?

```
acc_av3 =>
** 4
```

and updated?

```
upd_av3 =>
    ** 2
```

In this example the values are stored in a list. The active variable mechanism does not presuppose this. For example the values might be obtained from a generator function, or read in from a file. Values given to the updater might be output to a device by the updater. In that case the active variable would function as a stream. All that is required is that for an active variable of multiplicity n, the base procedure produces n results and the updater takes n arguments. There need not be any relationship between what they do.

2.28.2 Accessing the nonactive value

The syntax word "*nonactive*", analogous to "*nonop*", "*nonsyntax*" can be used immediately before an active identifier to suppress its invocation, e.g. to discover the real value, as opposed to its active value:

```
nonactive av3=>
** <procedure av3>
```

2.28.3 Making an active identifier local: dlocal

If an active identifier is to be used as local to a procedure it *must* be declared local using "*dlocal*". E.g

```
define test;
    dlocal av3=(99,100,101);
    av3
enddefine;
test() =>
** 99 100 101

av3 =>
** 4 5 10
```

vars should not be used to declare an identifier as dynamically local. This is because *vars av*3; would re-declare "*av*3" as an ordinary non-active identifier, and prevent it working properly as an active identifier thereafter.

2.29 System active identifiers

Examples of active identifiers in POPLOG are: *current_directory*, *current_section*, *popdeverr*, *popdevin*.

2.29.1 Finding the multiplicity and identprops of active identifiers

$isactive(W) \rightarrow n$
$isactive(W) \rightarrow false$

This procedure returns the multiplicity of an active identifier, or $false$ if it not active, e.g.

```
    isactive("av3") =>
    ** 3
```

```
isactive("acc_av3") =>
    ** <false>
```

Note that the *identprops* (see Chapter 5.3.5) of an active identifier do not reveal its activeness, although other attributed will be indicated. For further information about active variables see Chapter **??**.

2.30 History of the open stack

The following two sections are written by R.J.Popplestone.

The POP stack goes back to POP-1 and its predecessors. These were reverse polish languages, much in the style of Forth[**?**], but with list processing capabilities. When Rod Burstall and I designed POP-2, there were significant arguments in favour of hiding the explicit stack from the user. For example it is then possible to determine the applicative structure of a program at compile time, and provide more error checking. There are two arguments in favor of an open stack:

- It serves as a useful temporary data object e.g.

```
[%for i from 1 to n do i endfor%]
```

- It readily allows one to handle variable numbers of arguments to a procedure, and also to write procedures which return procedures as results which have variable numbers of arguments (e.g. *newarray*).

The first argument is not really clinching, but the second one seemed very important to us. LISP, which comes from an interpretive tradition, is implemented on a LISP machine using stacks, and has to make of CDR coding to provide a way of passing variable numbers of arguments without a big store turnover.

Subsequent language development offers ways forward. The ML language [?] has a tuple type which means that the arguments of a function are a distinct data-type in their own right, and are not confused with lists.

2.31 The history of the POP procedure

POP-11 differs considerably from POP-2 in the syntactic form of procedures, but conceptually the main difference is the introduction of lexical variables. Other modifications include the introduction of user-definable syntax procedures, the direct generation of VMCODE by users, and the addition of a *pdnargs* field to a procedure record. The following example of POP-2 code is taken from [?]

```
FUNCTION FACT N;
  IF N=0 THEN 1 ELSE N*FACT(N-1) CLOSE;
END;
```

The decision to make lower-case available, and be the case used for keywords, was taken in the early 1970's. Procedures with typed arguments (integers and reals) were embodied in WonderPop, developed in Edinburgh University in the 1970's. The syntax was considerably revised by Sussex University to obtain POP-11. They it was who introduced the systematic matching brackets (*if..endif* etc) and the current form of writing result variables. Both these innovations were intended to reduce the error rate among their students.

See also

2.32 Exercises

Try defining the combinators C K S ...

Chapter 3

In which we learn how to use Data Objects

NOTES

Equality does not, however, follow the usual mathematical axioms, e.g. $cons(2,3) \neq cons(2,3)$, where *cons* is the LISP list-cell constructor. The impracticability of providing a computed version of mathematical equality is clear if one considers tha case of functions. Nevertheless, care is taken in the definition of the language to make sure that the meaning of the equality is clearly defined. It corresponds to the LISP EQ, or the concept of equal address in mechanistic terms.

The introductory discussion of simple and compound is unsatisfactory (despite rewriting).
The idea of *pair* is introduced too baldly.
zero indexing for LISP vectors

Introduce boxes to represent data structures.

It is a Useful Pot to Keep Things In. — Winnie the Pooh

3.1 Introduction

In this chapter we will consider POP data objects in general in a systematic way. We have already met a number of different kinds of data-object, e.g. *numbers*, *lists*, *words*. We will see how these fit into a general framework, and how you can define new types of data object within this framework.

When talking about data-objects, we will use the terms "data-type" and "data-class". We are using these terms as follows:

- We use the term "data-type" to mean a set of POP objects together with procedures for recognising them and operating on them. Thus what constitutes a data-type is really a matter of convention. It is open to users to build their own conventions about data-types. This includes the possibility of developing formal conventions for describing type-hierarchies, for example in the manner of the Common Lisp Object System [?].
- The term "data-class" is much more restrictive. Associated with each POP object O is another object called its *key* which is used to provide a number of capabilities. For example the procedure to print out an object O is held in its key. A data-class is a set of POP data-objects *all of which have the same key*. If O is an object, then $datakey(O)$ is its key. To see the data-key for integers, try typing $datakey(23)$ =>.

In POP, numbers, which are described in Chapter 6, form a data-type, consisting of several classes, of which *integers* are one. Likewise POP lists are a data-type, built out of the classe of *pairs*, and the class which consists of [ ].

POP11 has a number of built in primitive data classes including *integers, integers, words, strings, booleans, procedures, vectors, processes*. There are some one-element classes. The class which consists of [ ] (the empty list), and the class which consists of *termin* are one-element classes.

From the user's point of view, POP data objects can be thought of as falling into to two broad categories

- Simple objects which do not have any sub-object e.g.(short) integers.

- Compound objects which do have sub-objects.

This categorisation is something of an oversimplification — the real distinction between simple and compound objects is explained in section 3.7.

The sub-objects of compound objects are referred to as *components* or $fields$. These are accessed by POP procedures, referred to as *selector* or *access* procedures. For example, the procedures to access the two components of a pair are $front$ and *back* — not *hd* and *tl* which are for lists (see **??**). The selector procedures can be used to *update* the component of the record that they access. E.g. $3 \rightarrow front(Pair)$ will give the $front$ of the pair record held in the variable $Pair$ a new value of 3.

3.2 Identity and assignment

In POP there is an infix operator == which is used to determine whether two objects are *identically equal*. If O_1 and O_2 are two objects then $O_1 == O_2$ implies the ordinary equality $O_1 = O_2$. In implementation terms, if $O_1 == O_2$ then O_1 and O_2 are the same bit pattern [1]

The following facts are true about identical equality:

- If V_1 and V_2 are (non-active) POP variables, then immediately after the assignment statement $V_1 \rightarrow V_2$, the statement $V_1 == V_2$ will always evaluate to $true$, and will continue to evaluate to $true$ at least until either of the variables is again assigned to. In implementation terms this means that assignment to a variable in POP only transfers the bit-pattern object identifier from one location to another[2].
- If (V_i) is a sequence of POP variables, and $(V_{local,i})$ are the input-local variables of a procedure P, and if P is called by $P(V_1, V_2, \ldots, V_i, \ldots)$, then within the body of P,

[1]The concept of identical equality is the same as that of the *eq* function in LISP. The concept is not present in Prolog, which makes that language rather further removed from the real machine than either LISP or POP.

[2]This differs from other languages, in which copying of the fields of a data-object may take place. The POP model of assignment to a variable is the same as that of LISP

$V_i == V_{local,i}$ evaluates to $true$, provided neither has been assigned to since the call. In terms of other programming languages, this states that simple objects are passed by value, compound ones by reference.

- We shall see in sections 3.4.1 and 3.4.2 how strict equality is affected by assignment to fields of members of record or vector classes.

3.3 Fixed and variable size objects

Classes of compound objects themselves fall into three categories,

1. Those for which all members of the class have a fixed number of fields, and each occupies a fixed amount of the computer's memory in standard implementations. These are called $records$. For example *pairs* and $references$ are records. There is a macro $recordclass$ described in 3.4.1 which allows a user to create his own class of records.

2. Those which for which the number of fields can vary between one member of a class and another, and for which the fields are accessed by an integer *subscript*, sometimes called an *index*. These are called $vectors$. For example strings belong to a class of vectors. There is a macro *vectorclass*, described in section 3.4.2, which allows a user to create his own class of vectors.

3. Other objects which occupy a variable amount of memory but whose fields are not accessed, or not all accessed, by an integer subscript. These include *words* and *procedures*.

3.3.1 Procedures associated with records

Records are built by *constructor procedures*. E.g. the constructor for pairs is $conspair$. Each constructor takes as many arguments as there are components in the record, and creates a new record which has components as specified by the arguments. E.g. $conspair(2, 3)$ makes a pair whose $front$ is 2 and whose $back$ is 3.

Every record-class also has an associated *destructor* procedure. This is perhaps a misnomer — the record itself is not at all changed by the destructor procedure, but all its components are put on the user stack (see Chapter 2.5.2) in such an order that an immediate call of the constructor function would make a copy of the original record. Thus the constructor and destructor are inverse functions.

There is also a *recogniser procedure* associated with a record-class, which returns *true* only when applied to an argument which is a member of the class.

3.3.2 Procedures associated with vectors

There are two ways of making a member of a vector class.

1. The *initialisation procedure* P_{init} may be applied to an integer n to give a vector $P_{init}(n)$, which has n elements, each of which is a standard value.

2. The *constructor procedure* P_{cons} is a variadic procedure. $P_{cons}(O_1, O_2, \ldots O_n, n)$ is a vector of n elements, of which the first is O_1, the second is O_2, etc.

The *destructor procedure* for a vector class is the inverse of the constructor procedure, it takes a vector and pushes its components on the stack, and finally pushes the number of components.

The components of a vector class can be accessed individually by subscriptor procedures. $P_{subscr}(i, \mathbf{v})$ returns the ith component of $\mathbf{v}$, supposing that $1 \leq i \leq \mathit{length}(\mathbf{v})$. Fast subscripting procedures are available, which, at the peril of your POP system's life, you may employ instead.

Finally, vector classes have a recogniser procedure in the same way that record classes do.

3.3.3 The data-key

We have stated above that each data class has a *key* associated with it. The key is a record that contains information common to all objects in the class, including procedures associated with the class. For example each class has a procedure to print members of that class. In addition there is a *word* associated with each class, called the *dataword*.

The main data-types that POP objects are considered as belonging to are given in the table below. The standard data-classes are tabulated in section 3.7.

array	A kind of procedure, allowing data to be accessed by multiple numeric indices e.g. A(i,j,k)
boolean	A data-class containing the truth values *true* or *false*
device	This data-class contains devices for input and output
key	Contains information common to all members of a data-class
list	A derived data-type built out of linked pairs
nil	The unique item [ ] in the variable *nil*, — the empty list.
number	The various kinds of numbers are detailed in Chapter6
pair	A record with two fields, used mostly to make lists
procedure	Procedures,including *closures*, *properties*, *arrays*
process	These store the saved state of a computation
property	Generalised association tables masquerading as procedures
reference	These are one-element records
section	Structures with information for the compiler about the *scope of variables*
string	Strings, i.e. vectors of characters.
termin	The unique item in *termin*, used to indicate the end of a file
vector	A vector is an object whose components are accessed by a single integer index.
word	A word is an object that can be the name of a variable.

3.4 Declaring your own class of data-objects

You youself can create new classes of data-object. The most convenient way of doing this is to use the macros *recordclass* and *vectorclass*.

3.4.1 Declaring your own records

The macro *recordclass* allows you to declare your own class of records. The syntax of its use is as follows:

```
<recordclassdef>
             =  recordclass {<identspec>} <classname> <fields>;
<classname>  =  <word>
<fields>     =  {<fieldname>{:<fieldspec>}{,}}*
<fieldname>  =  <word>
<fieldspec>  =  "full" | <n>
```

recordclass is a library macro — if you want to understand the built-in POP capabilities that it uses, turn to section 3.13. It automatically defines constructor, destructor, and recogniser procedures for the new class, a variable containing its *key*, and selector/updater procedures for each field.

The $< classname >$ specifies the dataword of the class. The name of the constructor procedure is obtained by appending to the word "*cons*" the $< classname >$, the name of the destructor procedure is similarly obtained by appending to the word "*dest*", and the name of the recogniser is obtained by appending to the word "*is*".

The access procedures all have names which are the $< fieldname >$ of each field. The optional $< fieldspec >$ specifies the type of object that may be stored in the given field. The possibilities are

- The word "*full*". This is also the default if there is no $< fieldspec >$. It indicates that the value of the field can be any POP item.
- An integer $< n >$. This indicates that the field can only have a value which is an integer of a maximum size specified by $< n >$. For the full meaning of $< n >$ see section 3.13.7.

Note that the $< field >$'s may be separated by commas.

The optional $< identspec >$ specifies the status of the identifiers created by *recordclass*. One of *constant* and *vars* may be used, to indicate that the identifers should or should not be made constant, and one of *procedure* or 0 may be used, to indicate their *identtype*. If neither *constant* nor *vars* is specified, the identifier status is defaulted from the variable *popdefineconstant*; if neither *procedure* nor 0 is specified, the identifier type is defaulted from the variable *popdefineprocedure*.

For example, the statement

```
recordclass point colour xof:16 yof:16 ;
```

creates procedures

```
conspoint, destpoint, ispoint, colour, xof, yof
```

and declares the variable *point_key* with the new key as its value. The *colour* field of a point record may be any object, but the *xof* and *yof* fields must be integers in the range 0 to $2^{16} - 1$.

Recompiling a *recordclass* declaration

The subscriptor, destructor and recogniser procedures for a record class only accept records for which the key is *identically equal* to the key with which the members of the class were created. The macro *recordclass* is defined so that if you re-compile a file containing a call of *recordclass* it will not construct a new key, unless the specification of the class has changed.. More precisely, when a *recordclass* declaration is executed the value of the word $< name >$ *_key* is examined and if the value is a key and its specification is the same as the specification in the declaration then a new key will not be constructed, although the procedures associated with the key (selectors, constructors etc.) will be reassigned to the variables that should contain them.

This behavior ensures that, if the declaration has not changed, then the old constructors, selectors, etc. will continue to work on objects already created. If for any reason you do wish to create a new key, then you should assign some non-key object (e.g. undef) to the key name. E.g., having compiled the code given above:

```
undef -> point_key;
```

will force the construction of a new key. Any previously constructed records using the old constructor will not work with the new field selectors/updaters and will not be recognised as instances of the new data-class.

3.4.2 Declaring your own class of vectors

Just as you can declare your own class of records with *recordclass*, so also you can declare your own class of vectors with *vectorclass*, using the syntax:

```
<vectorclassdef>
        =  vectorclass {<identspec>} <classname> <field>;
<field>  =  <fieldname>{:<fieldspec>}*
```

vectorclass automatically defines initialisation, constructor, destructor, subscriptor, fast subscriptor and recogniser procedures for the new class, and a variable containing its key.

The $<$ *classname* $>$ specifies the dataword of the class.

The optional $<$ *fieldspec* $>$ specifies the type of elements of this class of vectors, and has exactly the same form and meaning as that for *recordclass*, described in section 3.4.1and 3.13.7. Likewise the optional $<$ *identspec* $>$ specifies the status of the identifiers created by *vectorclass*, according to the conventions defined in 3.4.1. Again, the names of the procedures are obtained by appending the class name to the words "*init*", "*cons*", "*dest*", "*is*", "*subscr*" and "*fast_subscr*".

For example the statement

```
vectorclass short 16;
```

creates procedures

```
initshort, consshort, destshort, isshort
subscrshort, fast_subscrshort,
```

and declares the variable *short_key* with the new key as its value. Vectors of type *short* can only contain integers in the range 0 to $2^{16} - 1$.

Recompiling a *vectorclass* declaration

As in the case of recompiling a *recordclass* definition, discussed above, the macro *vectorclass* is defined so that if you re-compile a file containing a call of *vectorclass* it will not construct a new key unless the specification has been changed.

3.5 Standard Full Vectors

There is a class of vectors built-in to POP systems called standard full vectors, abbreviated to SF vectors. Being full vectors, they can have any object as component. A constant SF vector has a syntactic form very like that of a list, but with curly brackets rather than square ones. E.g.

```
{1 {2 3} ^L}
```

is a SF vector, with first component the integer 1, second component the SF vector {23} and third component the value of the variable L.

vector_key [constant] This constant holds the key object for standard full vectors. Keys are described in section 3.13.

3.5.1 Predicates on S.F. Vectors

$isvector(O) \rightarrow b$
This procedure returns $true$ if O is a standard full vector, $false$ if not.

3.5.2 Constructing New S.F. Vectors

$consvector(O_1, O_2, \ldots, O_n, n) \rightarrow \mathbf{v}$
This procedure constructs and returns a standard full vector of size n. Thus the top object of the stack is used to specify the size of the vector $\mathbf{v}$, and the n objects below the top are used for the components of $\mathbf{v}$, in such a way that $\mathbf{v}(i) = O_i$. n must be a (short) integer.

$initv(n) \rightarrow \mathbf{v}$
This procedure constructs and returns a standard full vector $\mathbf{v}$of length n whose elements are all the word $"undef"$.

$sysvecons(O_1, O_2, \ldots, O_n, m) \rightarrow \mathbf{v}$
This procedure constructs and returns a standard full vector $\mathbf{v}$ containing all the items on the stack *except* the last m, i.e. it performs consvector(n) where $n = stacklength() - m$. This procedure is used by the POP-11 vector constructor, which saves the stacklength m before compiling a vector constructor expression, and then calls $sysvecons(m)$ after compiling it, thus producing a vector of all the items in the expression.

3.6 Accessing S.F. Vector Elements

$destvector(\mathbf{v}) \rightarrow n \rightarrow O_n \ldots \rightarrow O_2 \rightarrow O_1$
This is the destructor procedure for the standard full vector **v**, i.e. it puts all its elements on the stack, together with its length.

```
destvector({A B C D}) =>
** A B C D 4
```

$subscrv(n, \mathbf{v}) \rightarrow O$
$O \rightarrow subscrv(n, \mathbf{v})$

This procedure returns or updates the n-th element of the standard full vector **v**. Since *subscrv* is the *class_apply* of standard full vectors (see section 3.13), this can also be called as $\mathbf{v}(n) \rightarrow O$ and updated using $O \rightarrow \mathbf{v}(n)$

3.6.1 Applying generic object manipulation procedures to S.F.vectors

The generic object manipulation procedures described in section 3.11 (*datalength*, *appdata*, *explode*, *fill*, *copy*, etc) are all applicable to standard full vectors, as are the generic vector procedures (*initvectorclass*, *move_subvector*, *sysanyvecons*, etc) described in section 3.12.

3.7 Representation of POP objects in a machine

This section explains how POP data-objects are usually represented in a real computer. So you will need to have some knowledge of machine architectures to read this.

Because *word* in POP means a data-object which can be the name of a variable, e.g. $''x''$, as described in Chapter 8.1, we will use the term "machine-word" to mean that unit of storage which is commonly referred to as a "word" in talking about machine architecture. For most POP implementations this will be 32 bits. The Symbolics LISP machine uses a 40 bit word, and is the only likely exception to this rule.

In existing implementations of POP, a compound object is represented as a contiguous block of the computer's memory which will contain *fields* for the sub-objects. A field may contain a *pointer* to a compound sub-object, or the bit-pattern which constitutes a simple object. It is possible to restrict fields to containing only simple sub-objects, as we saw in section 3.4.1.

Thus the conventional implementation of POP compound objects is to have them be pointers to blocks of store in a Von-Neumann architecture. There is no obligation to implement them in this way, and indeed in some parallel architectures it would be better to regard them simply as *object identifiers* — unique bit patterns to which the component accessing procedures can be applied. An example of a system where objects are treated thus is X-windows system[**?**], where the window objects are simply passed between processes as unique identifiers.

In many programming language systems, the types of data objects are not encoded explicitly within the objects, but are merely implicit in the way in which they are processed; that is, objects are not identifiable at run-time, although they may have detailed descriptions at compile-time.

In POP, however (as in most AI programming languages), data objects carry around with them an explicit representation of type — or to be more accurate data class — which all procedures in the system can use in deciding how to process a given object. This is achieved by representing all objects to be manipulated as encoded bit patterns, one machine-word in size which identify themselves as either

1. directly containing the actual data ('simple' objects), or
2. as pointers to data objects ('compound' objects).

The latter pointer types are then further distinguished by the *key* field in the data objects to which they point (as explained in 3.13). Simple objects also have keys, but this is achieved by testing for them in the *datakey* procedure. [3] [4]

For efficiency of implementation it is important to give some thought to the location of the key field within the record. The most important aspect of this is that *procedures are data-objects* and it is important to be able to call them with a machine-code subroutine call instruction. On many machines this will not admit an offset, so that this means that the pointer to procedure objects should not point to the machine word that holds the key pointer, but to another word. In POPLOG the convention is to have the key pointer be held in the second word of the object record. In the case of procedures, the first word will hold a jump around the fixed set of fields (*pdprops*, *updater* etc.) that begin the procedure.

While POP compound data-objects are, in current implementations, pointers to store, it is important to bear in mind that *these pointers may change as a result of garbage collection*. The garbage collector reclaims store that is no longer in use. This could leave a lot of 'holes' which might be the wrong size to hold new data-items, so the garbage collecter relocates objects so that they are contiguous. Currently this is done at every collection.

Simple and compound objects are distinguished by the procedure *iscompound*. Note that some apparently simple objects, such as *ddecimals* have to be implemented as compound objects, since they take up too much space in memory to be simple items. Thus *ddecimals* require 64 bits of data.

Note that the above scheme does not apply to integer or floating-point data held in *packed* vectors and records (e.g. strings). The POP system knows the types of the fields of these objects by virtue of the class description held in the *datakey* of the objects. Thus the fields

[3] The idea of a *key* field is due to Dr Foster[?], who used it in the ABSYS constraint propagation system. The original plans (c.1967) for the implementation of POP-2 involved keeping a "zoo" in which data-objects would be held in "cages". Each cage would hold only one type of object, so the type of an object could be determined from the address. This plan was rejected in favor of Foster's key-cells because the latter was much simpler to implement. Later, the WonderPop system made use of cages.

[4] Thus POP implementations may use only a minimal set of tag bits, namely sufficient to distinguish between simple and compound items, and to distinguish between classes of simple items. Usually POP will need to distinguish between *false*, short integers and short decimals (floats), and compound items. Thus 2 tag bits often suffice. This is in contrast to most LISP implementations, which use a rich set of tag bits (typically 8). The AlphaPop implementation makes use of a richer set of tag bits. One advantage of a rich set of tag bits is that many type tests do not require a memory access. The disadvantage of using many tag bits in a 32-bit machine is that the virtual address space is restricted to say 24 bits, which is starting to look small.

can be represented as they would be in non-AI languages, i.e. as a sequence of bits which characterise the value of each datum without any type indication. However, extraction or insertion of these field values necessitates conversion to and from POP representation, this being performed automatically by the corresponding access and update procedures. These packed representations are therefore particularly valuable in communicating with external procedures — those written in another language like C or Fortran.

Below is a complete list of the data types currently available in POPLOG, together with their identifying names as given by the procedure *dataword*. Note that only integers and single-length floating point are simple objects: all others are compound.

Dataword	Type	Described In
biginteger	Arbitrary-precision integer	6
boolean	The unique objects *true* and *false*	3.8
complex	Complex number	6
ddecimal	Double-length Floating Point	6
decimal	Single-length Floating Point (simple)	6
device	I/O Device	??
ident	Identifier	5.3
integer	Small integer (simple)	6
intvec	Signed Integer vector	??
key	Class Key	3.13
nil	The unique object [] in *nil*	??
pair	Pair (and lists)	??
procedure	Procedure – in general, and closures	2
	– property procedures	11.3
	– array procedures	10
process	Process	??
prologterm	Prolog term	??
prologvar	Prolog variable	??
ratio	Ratio of two integers	6
ref	Reference	3.8
section	Section	13
stackmark	The object $<$ *popstackmark* $>$	7.4
string	String	9
termin	The unique object $<$ *termin* $>$ in *termin*	20
undef	Undef record — used to initialise variables	5.3
vector	Full vector	??
word	Word	9

The types *descriptor* and *external_procedure* may also be present when using external procedures (see **??**).

There is no special data type for characters, as these are simply 8-bit integers[5]. Character strings are merely *packed* vectors with a field-size of 8.

conskey (described in section 3.13) can be used to create a new record class or vector class e.g. bit-vectors. 3.4.2 and 3.4.1 describe two library macros providing convenient methods of calling *conskey* to create a new vector class (e.g. bit-vectors) or a new record class.

For each built-in data-type there is a global permanent variable whose name starts with the *dataword* and ends with *_key*, and whose value is the key for that type, e.g. *integer_key*, *ratio_key*, *biginteger_key*, *device_key* etc.

3.7.1 The Heap

In the conventional implementation of POP, space for data-objects is allocated in an area of store known as "the heap". However the POP system, which is created by the linking loader from object code generated mostly by the compiler for the POP system dialect, but partly by the assembler for the object machine, and partly by the C compiler, will contain many of the built in objects, for example the procedure *sin*. These objects are *not* in the heap, and so their space cannot be reclaimed by the garbage collector. Some of them may be sharable between different users of POP.

3.8 Standard record-classes not detailed in other chapters

Most of the built-in data-classes of POP are described in chapters dedicated to them, or to data-types built out of them. However there are a few data-classes which do not warrant a

[5]This is in distinction to Common Lisp, where characters are a distinct data-type. The use of multiple fonts for typography in POP is treated in Chapter **??**

separate chapter, and they are described in this section.

3.8.1 References

A reference, a member of a class whose dataword is "*ref*", is a record containing a single field, its *cont*, short for contents. This may be any POP object.

$isref(O) \to b$
This procedure returns $true$ if O is a reference, $false$ if not.

$consref(O) \to Ref$
This procedure constructs and returns a reference Ref for which $cont(Ref) = O$.

$cont(Ref) \to O$
$O \to cont(Ref)$

This procedure returns or updates the contents of the reference Ref.

ref_key
This constant holds the key object for references (see 3.13).

3.8.2 Booleans

The two objects $< true >$ and $< false >$ are the sole members of the data-class of *booleans*, whose *dataword* is "*boolean*".

Note that any condition testing in POP regards any object other than $< false >$ as being $true$. So that for example

```
if 1 then 2 else 3
endif
```

evaluates to 2. Most comparison tests will yield $true$ or $false$. However certain ordering procedures, such as $alphabefore$, described in Chapter 9 produce 1 as result when two objects being compared are *equal*. This is for the benefit of sorting procedures, which need to determine, given two objects O_1 and O_2, whether O_1 should precede O_2, or follow it, or whether they should be combined because they are equal.

$true$ [constant] The value of this constant is the boolean $< true >$.

$false$ [constant] The value of this constant is the boolean $< false >$.

$isboolean(O) \rightarrow b$
This procedure returns $true$ if O is one of the two boolean objects, it returns $false$ otherwise.

$not(O) \rightarrow b$
This procedure complements the truth-value of any object O. That is $not(false) = true$, and for any $O \neq false$, $not(O) = false$.

boolean_key
This constant holds the key object for booleans (see section 3.13).

3.9 Byte-Accessible Objects

Certain procedures (*move_bytes* and *set_bytes* in this chapter, *sysread*, *syswrite*, *sys_io_control*, etc, in **??** and *external_apply* in **??**) require *byte-accessible* objects as arguments. A object is byte-accessible if it does not contain any full fields following its key field, which is located in the second 'machine -word' of the object. Full fields are those declared with $< field\_spec >$ $"full"$. They thus may contain pointers to other POP objects, which cannot be straightforwardly written out to backing store, and are not readily interpretable by procedures written in languages not consistent with the POP system.

Byte accessible objects are strings, intvecs, user-defined non-full vectors, and user-defined records that, if they contain any full fields at all, contain only one such field as the first in the record.

Such objects can participate in 'byte oriented operations' as performed by the procedures mentioned above. In all these cases these procedures operate on bytes of the object with the convention that the first byte of the object begins at the first byte of the third word of the object, that is to say, the word following the key field. In the case of a string, intvec, or user-defined non-full vector, this is the location of the first element of the string/vector; in the case of a user-defined record it will be whatever field starts there (see 3.13).

3.10 Procedures which operate on many or all data-objects

There are a number of procedures which operate on all data objects, or which operate on several kinds of data-object class. The procedures given below provide various functions applicable to many objects (including user defined ones) although, with the exception of = etc, not to numbers. See also 21 for generic printing procedures.

3.10.1 Predicates on Objects

$issimple(O) \to b$
This procedure returns $true$ if O is a simple object (i.e. a simple integer or a single-float decimal), $false$ otherwise.

$iscompound(O) \rightarrow b$
This procedure returns $true$ if O is compound data object, $false$ if O is simple. Simple and compound are explained earlier in this chapter.

$isinheap(O) \rightarrow b$
For a compound object O_1, this procedure returns $true$ if O_1 is in the working heap, i.e. is not an object in in the system area, $false$ otherwise. For a brief description of the heap, see section 3.7.1.

$isrecordclass(O) \rightarrow K$
This procedure returns $datakey(O)$ if O is a record-type data object (e.g. O is a $reference$, $pair$, or a record type constructed with $conskey$). Otherwise it returns $false$.

$isvectorclass(O) \rightarrow K$
This procedure returns $datakey(O)$ provided that O is a vector-type data object (e.g. $string$, $intvec$, full vector, or a vector type constructed with $conskey$). Otherwise it returns $false$.

There are compound objects for which neither $isrecordclass$ nor $isvectorclass$ returns true. These are 'special' objects, e.g procedures, keys, etc., and form the third category described in 3.3.

3.10.2 Information About Objects

Much information about an object is available through its key, as described in section 3.13. The following procedures make use of the object-key to provide useful information about objects.

$datasize(O) \rightarrow n$
This procedure returns the number of machine-words occupied by the object O in memory. If the object is simple, i.e. an integer or decimal, 0 is returned.

3.10.3 Comparison Procedures

$O_1 == O_2 \to b$
This operator returns $true$ if O_1 and O_2 are absolutely identical (i.e. pointers to the same object or identical simple numbers), otherwise it returns $false$.

$O_1 / == O_2 \to b$
This is an operator returns $b = not(O_1 == O_2)$.

$O_1 = O_2 \to b$
This operator compares O_1 and O_2 by doing

$$class\_= (datakey(O_2))(O_1, O_2)$$

The default value in any key K for its $class\_=$ is $sys\_=$ (see below).

$O_1 / = O_2 \to b$
This operator returns $b = not(O_1 = O_2)$.

$sys\_= (O_1, O_2) \to b$
If O_1 and O_2 are objects, then this procedure compares O_1 and O_2, recursively element by element and returns $true$ if they are the same, $false$ otherwise. The sub-elements of two objects of the same class and length are compared by calling $=$ on them, which will thus use the $class\_= procedure$ for their class. If O_1 and O_2 are numbers of any kind then the result will depend on their mathematical equality/inequality – see the description of $=$ and $/=$ in 6. If K is any key object, then by default:

$$class\_= (K) = sys\_=$$

3.10.4 Structure Concatenation

There are two generic procedures which concatenate like data-objects. Chapter 21 contains a description of the operations >< and $sys\_$ >< in 21 for concatenating the printed representation of objects which may be unlike.

$O_1 <> O_2 \to O_3$
Concatenates O_1 and O_2, which must be of the same type, the result also being of that type; permissible types are strings, any kind of vector, lists and words. E.g.

```
[1 2 3] <> [4 5 6] =>
** [1 2 3 4 5 6]
{a b c} <> {d e f}  =>
** {a b c d e f}
'a ' <> 'string' =>
** 'a string'
"word1" <> "word2"  =>
** "word1word2"
```

This operator also composes two procedures, so that $P_1 <> P_2$ is a new procedure which first runs P_1 and then runs P_2, as described in chapter 2.22.

$O_1\ nc\_ <> O_2 \to O_3$
This operator is identical to <>, except that when O_1 and O_2 are lists, the second list O_2 is joined onto the end of the O_1, without copying O_1 (which is then the result). E.g.

```
[1 2 3] -> L;
L nc_<> [4 5 6] =>
** [1 2 3 4 5 6]
L =>
** [1 2 3 4 5 6]
```

3.11 Generic Dataobject Procedures

These procedures can be applied to most kinds of 'dataobjects', that is, compound objects which can be considered to have independent 'components' or 'elements' (this essentially includes everything except numbers, ordinary procedures, and special objects like $true$, $false$, $termin$, [ ], etc). The action of these procedures on individual data types is described in the sections dealing with the individual types.

$datalength(O) \rightarrow n$
$length(O) \rightarrow n$
These two procedures give the length of the object O_1, i.e. the number of elements in it. The only difference between *length* and *datalength* is that the former applied to a list returns the length of the list, whereas the latter would return 2 for the length of the first pair.

$appdata(O_1, P)$
Applies the procedure P in turn to each element of the object O_1.

$mapdata(O_1, P) \rightarrow O_2$
This procedure applies the procedure P in turn to each element of the object O_1, and uses $fill$ to fill a copy O_2 of O_1 with the resultant values. It is defined as

```
appdata(O_1, P);       ;;; get result values
if isword(object) then
    ;;; can't use fill with words
    consword(datalength(O_1)) -> O_2
else
    fill(copy(O_1)) -> O_2
endif
```

$ncmapdata(O, P) \rightarrow O$
This is the same as *mapdata*, but does not copy its argument (and therefore cannot work

on words). It is defined as

$$fill(appdata(O,P),O)$$

$copy(O_1) \to O_2$
This procedure returns a copy O_2 of the object O_1, in which sub-objects are not copied except where they form an 'essential' part of the outer object – see the chapters on individual data types. Note in particular that *copy* applied to a list will only copy the initial pair. As described in Chapter **??**, you should use *copylist* to copy a list at the 'top-level', i.e. following the *tl*'s.

$copydata(O_1) \to O_2$
This procedure copies its argument, and recursively copies its components, whereas *copy* merely copies the top level of a data object. There is an error check for one-level circularity.

$explode(O) \to O_n ... \to O_2 \to O_1$
$O_1, O_2, ..., O_n \to explode(O)$
This procedure puts the n elements of the object O on the stack. Apart from the case in which O is a list, this is the same as

$$appdata(O, identfn)$$

If O is a list, *explode* puts the elements of the list O on the stack, i.e. it is equivalent to:

$$dl(O)$$

In both cases, the updater does the opposite, i.e. given a object O, fills its n elements with objects from the stack.

$datalist(O) \rightarrow L$
L is a list of the elements of the object O. This is equivalent to

$$[\%explode(O)\%]$$

$fill(O_1, O_2, ..., O_n, O) \rightarrow O$
Given an n-element object O, this procedure fills it with n objects from the stack, and also returns O as result. The only difference between this procedure and the updater of *explode* is that the latter treats a *pair* as a *list*, whereas *fill* would treat it as a *pair*, i.e. a 2-element object.

$allbutfirst(n, O_1) \rightarrow O_2$
$allbutlast(n, O_1) \rightarrow O_2$
These procedures both take a (simple) integer n and a object O_1, which may be a list, a word, or any vector-class object and which must have at least n elements. They both return a object of the same kind with either the first n elements (*allbutfirst*) or last n elements (*allbutlast*) removed.

$last(O_1) \rightarrow O_2$
$O_1 \rightarrow last(O_2)$ Where O_1 is as for *allbutfirst* and *allbutlast*, returns or updates the last element of O_1 (which must have at least one element).

3.12 Generic procedures which operate on vectors or on byte-accessible objects

This section describes a number of generic procedures, some of which operate on all vectors, and some of which operate on all byte-accessible objects.

$initvectorclass(n, O_{init}, key_{\mathbf{v}}) \rightarrow \mathbf{v}$
The purpose of this procedure is to enable new vectors to be initialised with a given object, thus providing an extension of the capabilities of the normal vector $init-$ procedures initialise each element to a fixed value. It creates and returns a new vector of the class specified by the key object $key_{\mathbf{v}}$, using its *class_init* procedure, described in section 3.13. The object O_{init} must be a suitable component for the vector being constructed, and for all i, $1 \leq i \leq n$

$$length(\mathbf{v}) = n,\ \mathbf{v}(i) = O_{init}$$

$sysanyvecons(O_1, O_2, ..., O_n, m, cons_{\mathbf{v}}) \rightarrow \mathbf{v}$
Given a vector-class constructor procedure $cons_{\mathbf{v}}$ (like *consvector*, *consstring*, etc), this procedure constructs and returns a vector $\mathbf{v}$ of that class containing all the objects on the stack *except* the last m, i.e. it performs $cons_{\mathbf{v}}(n)$ where n is $stacklength() - m$. This procedure is used by the POP-11 *cons_with* vector constructor, which saves the stacklength m before compiling a constructor expression, and then calls

$$sysanyvecons(m, cons_{\mathbf{v}})$$

after compiling it, thus producing a vector of all the objects in the expression. The objects O_1, ..., O_n must of course be suitable for the kind of vector being constructed.

$move\_subvector(i_{src}, \mathbf{v}_{src}, i_{dst}, \mathbf{v}_{dst}, n)$
$\mathbf{v}_{src}$ and $\mathbf{v}_{dst}$ must be two vector-type objects. This procedure copies the n components of $\mathbf{v}_{src}$ starting at subscript i_{src} to the components of $\mathbf{v}_{dst}$ starting at subscript i_{dst} The destination vector $\mathbf{v}_{dst}$ must be of an appropriate size.

$move\_bytes(i_{src}, O_{byte,src}, i_{dst}, O_{byte,dst}, n)$
$O_{byte,src}$ and $O_{byte,dst}$ must be two *byte accessible* objects, as defined above. This procedure copies the n bytes of $O_{byte,src}$ starting at byte i_{src} to the bytes of $O_{byte,dst}$ starting at byte i_{dst}.

$set\_bytes(c, i, O_{byte}, n)$

This procedure sets the n bytes of the byte accessible object O_{byte} starting at byte number i to have the value c, which must be an integer from 0 to 255 inclusive.

3.13 The keys of data-objects

Every object in the POP system has in it a field containing a pointer to a *key*-object. This key-object identifies the class of the object. E.g. *vectors* contain a pointer to the *vector key*, *procedures* a pointer to the *procedure key*, and so on. Keys, being themselves objects, are identified by a key (the *key key*). So for every object class in the POP system, there is an identifying key object. For completeness, there are also key objects for the simple integer and decimal data types.

As well as identifying the class of any object, keys also serve as a means of holding various kinds of information about the class, e.g. which procedure is used to print objects in the class, which procedure is to be used by the operation = in comparing them, and which procedures are used to manipulate fields in the object.

The latter is, in particular, the means of providing procedures to manipulate new classes of object defined by the user. The process of defining a new class of object consists of simply defining a new key for the class, with the procedure *conskey*, described below. The appropriate procedures to construct and manipulate the new objects are then available from the key via the *class_* procedures described below.

Two distinct types of new object class can be constructed: record-type and vector-type. Recall that a record is a object containing a fixed number of distinct and possibly different fields, whereas a vector consists of a variable number of similar fields. For example a pair is a record, whereas vectors and strings are vectors. The built-in classes in the system also include other types which do not fall into these categories and which cannot be user-defined, e.g. keys, procedures, processes, etc.

The facilities given by *conskey* are more readily packaged in the two macros *recordclass* and *vectorclass*, as described in sections 3.4.1 and 3.4.2.

The following constant is provided: *key_key*

This constant holds the key object for key objects themselves.

3.13.1 Accessing Keys of Items

Note that for each built-in data-type there is a global permanent variable whose name starts with the dataword and ends with *_key*, and whose value is the key for that type, e.g. *integer_key*, *ratio_key*, *biginteger_key*, *device_key* etc.

$datakey(O) \rightarrow K$
This procedure returns the key of the class of which the object O is a representative — this includes simple data types. See 3.13.

$dataword(O) \rightarrow W$
This procedure returns the dataword of the object O.

$datakey(O) \rightarrow K$
This procedure returns the key K corresponding to the class of the object O.

$key\_of\_dataword(W) \rightarrow K$
Given a word W which is a dataword this procedure returns the corresponding K. This procedure is a property (see Chapter 11) defined in the library. It is initialised for the built-in data-classes, and updated by *recordclass* and *vectorclass*, but not by *conskey* itself.

3.13.2 Predicates on Keys

$iskey(O) \rightarrow b$
This procedure returns $true$ if O is a key, $false$ otherwise.

3.13.3 General Key Fields

$class\_= (K) \rightarrow P_{=}$
$P_{=} \rightarrow class\_= (K)$
For any *class_key* K, this procedure returns or updates the procedure used by $=$ in comparing objects of the class represented by K, the default value for any key being $sys\_=$. See section 3.7.

$class\_apply(K) \rightarrow P_{apply}$
$P_{apply} \rightarrow class\_apply(K)$
This procedure returns the 'apply procedure' for the class key K. If an object O of the class is applied as if it were a procedure, what happens is that P_{apply} is called instead, with O as argument. E.g. if the variable O contains a object of class K then the call $O()$ will turn into $P_{apply}(O)$. Similarily, if the updater of the object is called, then the updater of P_{apply} is called, i.e. $\rightarrow O()$ will result in $\rightarrow P_{apply}(O)$. The updater of *class_apply* assigns the procedure P_{apply} to be the apply procedure for the class key K. This mechanism will *not* change what happens when an actual procedure is applied, i.e. assigning to the *class_apply* of the procedure key has no effect.

WARNING: *class_apply* is the means by which array-type indexed access on lists, words, vectors and strings is implemented, i.e. the facility to use $O(n)$ for $subscrX(n, O)$. In general, the *class_apply* for these data types is a procedure like

```
procedure(object);
  if stacklength() /== 0 and isinteger(dup()) then
  subscrX(object)
  else
  mishap(object, 1, 'EXECUTING NON-PROCEDURE')
  endif
endprocedure;
```

and similarily for the updaters. Since this facility is used *widely* in both system and library procedures, you can expect trouble if you redefine *class_apply* for any of these data types.

$class\_dataword(key) \to W$
This procedure returns the dataword of the class key key.

$class\_hash(K) \to P_{hash}$
$P_{hash} \to class\_hash(K)$
For any class key key returns or updates the hashing procedure for objects of class key, called by $syshash$. Hashing is mostly used in implementing properties, as described in Chapter 11. The hashing procedure takes an object of the class key and returns an integer determined by the object, subject to the constraints that if for two objects O_1 and O_2, $O_1 = O_2$ then P_{hash} should have the property that $P_{hash}(O_1) = P_{hash}(O_2)$ value by the hashing procedure. The default hashing procedures are described in **??**.

$class\_print(K) \to P_{print}$
$P_{print} \to class\_print(K)$
For any class key K, P_{print} is the procedure used by $syspr$ to print an object in that class, as described in Chapter 21 The default value for any key is sys_syspr.

$class\_recognise(K) \to P_{recognise}$
This procedure returns the recogniser procedure of the class key K, i.e. a procedure which returns $true$ when applied to a member of the class, $false$ when applied to anything else.

$class\_spec(K) \to Spec$
This procedure returns the specification of the class key K — for a vector-type class this is a single field specifier, for a record-type class it is a list of them. The field specifiers are described along with the specification of $conskey$, to be found below. For any other type $Spec = false$.

3.13.4 Record & Vector-type Key Fields

$class\_cons(K) \to P_{cons}$
For a record or vector-type class key K, this procedure returns the constructor procedure for that class. For a record-type class, this procedure takes n arguments, where n is the the number of fields in the record; for a vector-type class, it takes an argument n and constructs a vector of length n with n objects taken off the stack. Examples of such procedures are *conspair* for pairs, *consvector* for S.F. vectors, and *consstring* for strings.

$class\_dest(K) \to P_{dest}$
For a record or vector-type class key K, this procedure returns the destructor procedure for that class. Examples of such procedures are *destpair* for pairs, *destvector* for S.F. vectors and *deststring* for strings.

3.13.5 Record-only Key Fields

$class\_access(n, K_R) \to P_{access}$
For a record-type class key K_R, this procedure returns the access procedure for the n-th field of that record class, i.e. the procedure that returns or updates the n-th field. Since *class_access* is the *class_apply* of keys, this may also be called as

$$K_R(n) \to P_{access}$$

$class\_datasize(K_R) \to n$

For a record-type class key K_R, this procedure returns the length in words of the record.

3.13.6 Vector-only Key Fields

$class\_init(K_{\mathbf{v}}) \to P_{init}$
For a vector-type class key $K_{\mathbf{v}}$, this procedure returns the initialiser procedure for that class, i.e. a procedure that takes an integer n and constructs a new vector of length n. For S.F. vectors, this is the procedure *initv*, for strings the procedure *inits*. Full vectors have their components initialised to $undef$. Non-full vectors have their components initialises to 0. See also *initvectorclass* in section 3.12, which initialises a vector for a given class key, but with a specified initialising value.

$class\_subscr(K_{\mathbf{v}}) \to P_{subscr}$
For a vector-type class key $K_{\mathbf{v}}$, this procedure returns the subcripting procedure P_{subscr} for that class. Thus

$$P_{subscr}(n, \mathbf{v}) \to O_n$$
$$O_n \to P_{subscr}(N, \mathbf{v})$$

return or update the n-th element of a vector in the class. Examples of such subscripting procedures are *subscrv* for S.F.vectors, $subscrs$ for strings. Note that since the *class_subscr* procedure of a vectorclass is by default its *class_apply*, a vector of the class can also be subscripted by

$$\mathbf{v}(n) \to O_n$$
$$O_n \to \mathbf{v}(n)$$

$class\_fast\_subscr(K_{\mathbf{v}}) \to P_{subscr}$
This procedure returns a subscripting procedure P_{subscr} in the same way as $class\_subscr$, but the P_{subscr} is a fast subscriptor procedure, which does not check the type of object it is being applied to, and can therefore return an illegal object which will cause the POP system to fail at the next garbage collection, or before if the updater is used.

3.13.7 Constructing New Keys

This section describes how to construct new key-objects, and hence new data-classes. Note however that this is more conveniently done by the following two macros: $recordclass$

$vectorclass$

These are described in section 3.4.1 and 3.4.2.

$conskey(W, Spec) \to K_{\mathbf{v}}$

$conskey(W, L_{spec}) \to K$

This procedure constructs and returns a key K for a new record or vector- type class of objects, with dataword W and specification as given by $Spec$ or L_{spec}. For a vector-type class, $Spec$ is a single field specifier, all components of a vector having the same specification. For a record-type class, L_{spec} is a list of field specifiers, where the n-th specifier in the list refers to the n-th field in the record, i.e. the record has $length(L_{spec})$ fields.

A field specifier can be one of the following:

- The word "$full$"
  This means that the field can hold any POP object whatsoever.

- a positive or negative integer n. This specifies a field that can contain integers only: the absolute value of n specifies how many binary bits the field occupies and must be between 1 and 32 inclusive in current implementations, i.e. $1 \leq abs(n) \leq 32$. If n is positive then the field can contain only positive integers, in which case the range of the integers $iis0 \leq i < 2^n$. If n is negative then the field can contain positive or negative integers, in which case the range for i is

$$-2^{n-1} \leq i < 2^{n-1}$$

- the word "$decimal$" or "$ddecimal$"

This means that the field can contain POP *decimal* — a floating point number. For "*decimal*", the field is 32 bits and can therefore hold a single-floating type datum; for "*ddecimal*" it is 64 bits and can hold a double-floating datum. Any non-complex number, including integers and ratios, can be assigned into such a field, conversion and/or rounding being done where necessary so that the value is always stored in floating point format. Output from the field is done according to the same rules as for arithmetic results, the production of a *decimal* or *ddecimal* being dependent on *popdprecision*, as described in Chapter **??**.

Notes:

1. A full field occupies 1 word, that is to say 32 bits in all current implementations.

2. An integer field of more bits than a POP simple integer (29 bits unsigned or 30 bits signed in current implementations) can produce a biginteger when accessed, if the value overflows a simple integer. Similarily, a biginteger within the range allowed can be assigned into such a field.

3. In a record, fields are allocated space in the order specified, treating the record as a sequence of bits, and starting at bit 0. However, the following points should be noted:

   - The key pointer occupies the second word of any object. Therefore, if a field would start in the the first word and finish in the second it is instead started at the third. Thus e.g. the *spec* [8 16 32 8] occupies 1 more word than [8 16 8 32].
   - A "full" field will always be started at the next machine-word boundary.
   - A *decimal* or *ddecimal* field will always be started at least at the next byte boundary and, depending on the implementation, probably at a higher one (e.g machine-word boundary).
   - The length of a record is always rounded up to an exact number of machine-words.

3.13.8 Examples

The following defines a new user vector-type class whose components are 16-bit positive integers:

```
conskey("short", 16) -> short_key;
class_cons(short_key) -> cons_short;
cons_short(32,64,128,3) -> a_short;
a_short =>
** <short 32 64 128>
```

whereas the next example defines a new user record-type class with 4 different fields, representing a person's name, address, age and a field to indicate sex (0=male, 1=female):

```
conskey("person", [full full 8 1]) -> person_key;
class_cons(person_key) -> cons_person;
;;; get the access procedures for each field
person_key(1) -> person_name;
person_key(2) -> person_address;
person_key(3) -> person_age;
person_key(4) -> person_sex;

cons_person('Fred Bloggs', 'No fixed abode', 99, 0) -> fred;
person_name(fred) =>
** Fred Bloggs
person_age(fred) =>
** 99
```

Using *recordclass*, an equivalent set of procedures could have been generated.

```
recordclass person
            person_name:full
            person_address:full
            person_age:8
            person_sex:1;
```

Note however that the constructor procedure would be *consperson* in this case.

3.14 Exercises

Extend the program given in Chapter **??** for describing a lists, words and numbers as English noun-phrases to describe any POP object as an English noun-phrase.

3.15 The development of data-representation in POP

In POP-2 the *key* record was not visible to the user, and the *dataword* was the only mechanism provided to allow the user to associate together information common to all members of the class. One of the big improvements made in POP-11 is to give the user access to the *key* record itself. Thus in POP-11 the main role of the *dataword* is to provide a name for the class — by default it is printed out when a vector or record is printed.

Most of the fields of the key record have come with POP-11. The most useful from the users point of view are the *class_print* field which allows a user to specify how to print objects, and the *class_apply* field which allows us to write $ss(i)$ to access the ith element of a string ss, rather than $subscrs(i, ss)$.

Chapter 4

How we deal with things going wrong.

NOTE How about time and profile???

4.1 Introduction

This chapter deals in detail with the facilities provided in POP to help you discover mistakes you have made in your programming. It is of course better not to make mistakes in the first place, and it is imporatant to develop a disciplined approach that will minimise the number of mistakes you make, and also make it easy to discover those which you do make easily. Such discipline will in general make it easier for other people to follow your work, or, for that matter, for you to understand it when you come back to it a year later.

There are two basic topics covered in the chapter, the *tracing* of procedures, and the *mishap* mechanism.

When you are developing a program it is often useful to follow the course of execution by *tracing* procedures, that is to say, for certain selected procedures, displaying on your terminal, or in a window, information about when a selected procedure is called, what its arguments are, when it returns and what its results are.

POP-11 provides a procedure tracing mechanism that can either be used unchanged for simple tracing, or tailored by users to provide more elaborate options.

Facilities are provided for specifying that certain procedures are to be traced, or that they should be no longer traced, that all tracing should be disabled or enabled, for re-directing trace printing to a specified file, and for re-defining some of the utilities used for tracing. In the case of POPLOG, tracing is integrated with the VED editor.

Section **??** explains the simple default facilities, involving *trace untrace* and the global variable *tracing*. Section 4.3 explains the mechanism of tracing in sufficient detail to allow you to modify it for your own purposes.

4.2 Summary of basic tracing facilities

Tracing of procedures is controlled overall by the following global variable:

tracing

The default value of this variable is *true*. Setting it to *false* switches off all trace printing of traced procedures. The use of *trace* or *untrace* sets it *true* again.

Tracing and untracing of procedures is accomplished using the following syntax:

```
      trace <names_of_procedures> ;
      untrace <names_of_procedures>;
 <names_of_procedures> = {<word>{,}}*
```

Commas are optional in *trace* and *untrace* commands. In both cases, if no procedure names are given, a distinctly different action occurs:

The command *trace*; is equivalent to $true \rightarrow tracing$;, and restarts the printing of all traced procedures.

The command *untrace*; is equivalent to $false \rightarrow tracing$;, and stops the printing of all traced procedures, although their trace printing will begin again when *tracing* ceases to be $false$.

$trace < names\_of\_procedures >$
This construction is used to alter procedures so that they print out information indicating when they begin execution and when they end execution. *trace*ing a procedure changes it so that when called, it does “before” printing, then it runs, then it does “after” printing.

The “before” printing includes the name of the procedure and the arguments to the procedure (if any), and “after” printing includes the results of the procedure (if any). Both “before” and “after” printing indicate the total depth of currently active traced procedure calls by printing a row of “!” symbols, followed by “¿” before, and “¡” after the procedure has run.

Trace printing of a procedure *Pooh* with three arguments and two results has a format like:

```
!!!!!> Pooh arg1 arg2 arg3
    --- trace printing of procedures called by Pooh ---
!!!!!< Pooh result1 result2
```

The 5 exclamation marks indicate that when *Pooh* was called there were 5 traced procedures higher up the calling chain.

4.2.1 Tracing a recursive procedure

Tracing is useful to indicate intermediate arguments and results. An example follows.

```
define factorial(n);
  if n=0 then 1 else n*factorial(n-1)
  endif
enddefine;
```

```
    factorial(3) =>
    ** 6

    trace factorial;
```

This causes nothing to be printed out - it merely changes the procedure so that *it* does the printing when run.

```
factorial(3) =>
>factorial 3
!>factorial 2
!!>factorial 1
!!!>factorial 0
!!!<factorial 1
!!<factorial 1
!<factorial 2
<factorial 6
** 6
```

The $'>'$ symbol indicates the start, and $'<'$ indicates the end of a procedure activation. The exclamation marks indicate depth of procedure calls.

$untrace < procedurenames >$;
To stop trace printing of one or more procedures, use *untrace*, e.g.:

```
    untrace P1 P3;
```

which will set the two named procedures back to normal.

You can switch off *all* trace printing by doing

```
untrace;
```

This is equivalent to $false \rightarrow tracing$;

Previously traced procedures remain *marked* as traceable.

These uses of *trace* and *untrace* without arguments have their effects immediately, and so they are not suitable for use inside a procedure definition. Instead use an assignment to *tracing*.

untraceall;
untraces all previously traced procedures.

All other uses of *trace* and *untrace* (i.e. with named procedures), will do $true \rightarrow tracing$, so restarting tracing.

If instead of *true*, 1 is assigned to *tracing* this will allow syntax procedures to be traced. If the value of *tracing* is *true*, attempting to *trace* a syntax procedure will cause an error.

4.2.2 Local tracing

trace and *untrace* can be used locally within a procedure definition to trace or *untrace* a procedure defined locally. This enables tracing to be dependent on calling environment.

```
define Pooh;
    vars addup;     ;;; make addup local
```

```
        trace addup;    ;;; tracing will occur only when Pooh is active
        ....
    enddefine;
```

4.2.3 Re-defining traced procedures

Definitions of procedures that have been traced, or their updaters, can be edited and recompiled. The new procedure, or the new updater, will be traced if the original one was. So re-tracing is uncessary.

4.2.4 Tracing system procedures

System procedures, e.g. *hd* can be traced in the usual way, e.g.

```
trace hd;
```

However, this will only be effective for calls of *hd* from your own or from library procedures which are compiled *after* this use of *trace*. So re-compile if necessary. System identifiers given to *trace* are no longer protected against redefinition by the user[1].

4.2.5 Tracing of updaters

If a traced procedure has an *updater*, the *updater* will automatically be traced. The trace printing will then include *updater* before the name.

For example:

[1] See Chapter 5.3.7 for a description of how to change protections of variables

```
trace hd tl;
vars list=[a b c d e];
"C" -> hd(tl(tl(list)));
>tl [a b c d e]              ;;; entering innermost call of tl
<tl [b c d e]                ;;; leaving tl
>tl [b c d e]                ;;; entering outer call of tl
<tl [c d e]                  ;;; leaving tl
>updater hd C [c d e]        ;;; entering updater of hd
<updater hd                  ;;; finished

list =>
** [a b C d e]
```

4.3 What you need to know to modify the tracing capability

We begin with two variables that are used by the standard tracing procedures.

poptraceindent
This global dynamic variable, initially 0, is incremented, in *systrace*, which is described below, whenever a traced procedure is running. Declared local in *systrace*, it is used by *systrace_pr* to count the number of '!'s to be printed.

cuchartrace
The default value of this global variable is *false*. If a character consumer is assigned to it, then trace printing done by *systrace_pr* will go through the consumer. Other printing is unchanged. This enables *trace* output to go to a special file, or to a window such as a VED window in POPLOG.

The procedures and variables defined below enable more sophisticted and varied tracing facilities to be defined by the user. All calls of traced procedures or their *updater*s go through the system procedure *systrace* or its *updater*. It increments *poptraceindent* and then in turn calls a user definable procedure *systrace_proc*. This has a default that does "before" trace printing then runs the procedure then does "after" trace printing. It does the printing by invoking another user definable procedure *systrace_pr*.

$systrace(P, W, P_{trace}, b_{updater})$

Closures of this procedure and its updater are used to replace procedures when they are traced. It is partially applied to four arguments by *trace*:

1. P is the original procedure to be traced
2. W is the name of P
3. The word "*systrace_proc*", the default value of which is defined below.
4. $b_{updater}$ is true iff P is the *updater* of the traced procedure.

If tracing is $false$, *systrace* just applies P. If $true$, *systrace* can be thought of thus:

```
define systrace(P, W, P_trace, Updater);
    lvars Args W P P_trace Updater;
    vars poptraceindent;
    poptraceindent + 1 -> poptraceindent;
    recursive_valof(P_trace) ->P_trace;
    P_trace(P,W, Updater)
enddefine;
```

systrace has an updater which takes the same arguments, and simply does

$$systrace(updater(P), name, P_{trace}, true);$$

If a procedure to be traced has an *updater*, then *trace* will make a closure of *systrace* to replace the procedure, and a closure of its updater will be the new updater.

So alterations to the original updater will affect tracing. The system procedures *sysPASSIGN* and *sysUPASSIGN* that handle redefined procedures or their updaters and which are described in Chapter 16, know how to deal with closures of *systrace* so that redefining a traced procedure or its updater has the expected effect.

$systrace\_proc(P, W, b_{updater})$
The word "*systrace_proc*" is the third argument of *systrace*. Its *valof*, i.e. *systrace_proc* is user assignable, and does the actual trace printing and running of the procedure. The arguments are:

- P is the original procedure being traced
- W is the name of the procedure
- $b_{updater}$ is a boolean, *true* iff $updater(P)$ is being traced

This procedure checks that there are enough arguments on the stack to correspond to the *pdnargs* [2] of P. It stores the arguments in a vector, so that they can be printed by *systrace_pr* then puts them back on the stack for P, which it then runs. If the stacklength is increased then again a vector containing the new stack items is used to print out results during "after" trace printing.

A simplified version of this procedure is:

```
define vars procedure systrace_proc(P, W, Updater);
    lvars Args P W Updater n;
    ;;; pdnargs can give funny values for closures of variadic pdrs
    max(0, pdnargs(P)) -> n;
    if stacklength() < n then
        mishap(stacklength(), 'TOO FEW ARGUMENTS FOR ' sys_>< W);
    endif;
    consvector(n)  -> Args;              ;;; save arguments in a vector
    stacklength() -> n;
    systrace_pr(W, `>`, Args, Updater);
```

[2] see Chapter ??

```
        explode(Args);                    ;;; put arguments back for P
        P();
        stacklength() - n -> n;          ;;; find out how many results
        consvector(max(0,n)) -> Args;
        systrace_pr(W, `<`, Args, Updater);
        explode(Args);
    enddefine;
```

$systrace\_pr(W, b, \mathbf{v}_{args}, b_{updater})$

This user assignable procedure is called by *systrace_proc* to do the trace printing. The arguments are:

1. W is the name of the procedure being traced
2. b is $true$ on entry $false$ on exit
3. $\mathbf{v}_{args}$ is a vector of arguments or results from the stack
4. $b_{updater}$ is $true$ iff the *updater* is being traced

Note that the vector argument may be re-used, so it should not be stored anywhere or updated, unless copied.

For example, to redefine this so that instead of using "!" to indicate depth it uses a level number, do something like:

```
    define strace_pr(W, Before, Args, Updater);
        lvars W, Before, Args, Updater;
        vars procedure cucharout, tracing = false;
        if isprocedure(cuchartrace) then
            cuchartrace -> cucharout    ;;; for redirecting trace output
        endif;
```

```
        pr('[Level:'); prnum(poptraceindent,2,0); spr("]");
        spr(if Before then '=>>' else '<<=' endif);
        if Updater then spr("updater") endif;
        spr(W);
        appdata(Args, spr);
        cucharout(`\n`);
    enddefine;
\end{verbatiim}

The trace printing for $factorial$ defined above will now look very
different.
\begin{verbatim}
    factorial(3) =>

    [Level: 1] =>> factorial 3
    [Level: 2] =>> factorial 2
    [Level: 3] =>> factorial 1
    [Level: 4] =>> factorial 0
    [Level: 4] <<= factorial 1
    [Level: 3] <<= factorial 1
    [Level: 2] <<= factorial 2
    [Level: 1] <<= factorial 6
    ** 6
```

This sort of format can be more readable when recursion gets very deep.

Another option would be to store infoormation associated with each procedure name, or each procedure, in a property, and then to re*define systrace_pr* to extract the information from the property and modify its actions accordingly. For example it would be possible to call popready either before or after the procedure runs, and to make the items on the stack available for inspection during the break. In addition, instead of simply printing out stack contents they could be associated with argument names stored in the property.

Yet another option is to redefine this procedure so that instead of printing information it builds a record of procedure entries and exits with arguments and results for use by a program that needs to know something about its own history.

Additional options and facilities may be added later, on the basis of feedback from users.

4.4 Warning: use of abnormal exits

If you use *chain*, *exitfrom*, *exitto*, *catch*, or the process mechanisms[3] this can mean that exits from traced procedures are not shown. The introduction of active variables will make it possible for this and other limitations to be overcome.

4.5 What happens when POP finds something wrong

In this section we consider the *mishap* mechanism. Firstly, in section 4.5.1, we consider the *mishap* procedure, which is the standard way of signalling that something is wrong, and which can, and should, be used by users themselves. Secondly, in section **??**, we consider how the mishap messages are actually delivered to the user, and how you might modify this to change the messages, or even decide that a no mishap has occurred at all.

4.5.1 Signalling a Mishap

$mishap(O_1, O_2, \ldots, O_n, n, \mathbf{s})$
$mishap(\mathbf{s}, L)$

This procedure is called by the system whenever some error condition is detected. It passes the given arguments to the variable procedure *prmishap* to print a mishap message, the standard value of which is *sysprmishap*. These procedures are specified in section **??**.

After calling *prmishap*, *mishap* calls the procedure in the variable *interrupt*, and, if that call returns, it then calls *setpop*. Before calling *interrupt* it also sets pop_exit_ok to false

[3] See Chapters**??** and 2.24

if the standard input is not a terminal — since the default value of *interrupt* under these circumstances is *sysexit*, this guarantees that a mishap will result in an error status being returned to the operating system. See Chapter **??** for further details.

Note that *mishap* locally sets the value of *prmishap* to be *sysprmishap* while the call of whatever procedure was in *prmishap* is in progress; this ensures that a another mishap occurring in that procedure will cause *sysprmishap* to be called.

4.5.2 Printing Mishaps

$sysprmishap(O_1, O_2, \ldots, O_n, n, \mathbf{s})$
$sysprmishap(\mathbf{s}, L)$

This procedure prints a mishap message, using *cucharerr* to output characters. The arguments to this procedure are either a number of 'culprits' n and a message string **s**, in which case there should be n items on the stack, or a message string **s** and a list of culprits L. The full format of the message is

```
MISHAP - <STRING>
INVOLVING:  <ITEM_1> <ITEM_2> ... <ITEM_N>
     FILE:  <popfilename>  LINE NUMBER <poplinenum>
COMPILING:  <names of procedures being compiled>>
    DOING:  <pdprops of procedures currently executing>
```

Note that the INVOLVING line is present only if there are 1 or more culprits, the FILE line is present only when *popfilename* is true, and the COMPILING line is present only when the POP VM is not at execute level.

This procedure is the standard value of the variable *prmishap*.

popfilename
If this variable is not *false*, *sysprmishap* assumes it contains the name of the file currently being compiled, and includes the line

```
FILE: <popfilename>  LINE NUMBER: <poplinenum>
```

in the mishap message. This variable is set locally by *compile* (and other POPLOG compilers where appropriate) to the name of the file being compiled.

poplinenum
This is assumed by *sysprmishap* to contain the line number within the current file being compiled. It is set locally to 1 by *compile* (and other POPLOG compilers where appropriate) when opening an input file with *discin*; the variable is then incremented by the *discin* character repeater each time a *newline* is read, as described in Chapter 20.

popsyscall
This controls which procedures currently executing are included in the DOING list of mishap messages produced by *sysprmishap*.

If it is *false* (the default), only procedures whose *pdprops* are not *false* and which are not ‘uninteresting’ are printed (where an ‘uninteresting’ procedure is one whose *pdprops* are not "*mishap*", "*setpop*", or a word being with '*sys*', '*pop*11_' or '*ved*').

If it is not *false* then all procedures are printed. If the value is addiionally an integer, then system procedures whose *pdprops* are *false* are printed as their hexadecimal addresses.

pop_mishap_doing_lim
This controls the number of procedures printed in the DOING list of mishap messages produced by *sysprmishap*. If *false* (the default), all currently executing procedures are included; otherwise it should be a positive integer N, specifying that only the most recent N callers are too be shon.

popmishaps
Each time it produces a mishap message *sysprmishap* puts in this variable a summary of the message printed (as list of items); this can be disabled by assigning *false* to *popmishaps*. (Default value []).

$prmishap(O_1, O_2, \ldots, O_n, n, \mathbf{s})$
$prmishap(\mathbf{s}, L)$

he procedure in this variable is called by *mishap* to print a mishap message: it takes the same arguments as *sysprmishap*, which is its default value.

4.5.3 Printing Warnings

$sysprwarning(W)$
Prints the warning message

```
;;; DECLARING VARIABLE <W>
```

and, if *popwarnings* is a list, adds W to that list. This procedure is the standard value of the variable *prwarning*.

popwarnings
If not *faalse*, *sysprwarning* uses this variable to build up a list of names of identifiers that have been declared automatically by the POP VM compiler through *sysdeclare* (which calls *prwarning*). Building of this list can thus be disable by assigning *false* to *popwarnings*; its default value is [ ].

$prwarning(W)$
The procedure in this variable is called by the POP VM compiler procedure *sysdeclare* to

print a warning message about an undeclared variable (see REF *VMCODE). Its standard value is *sysprwarning*.

warning($\mathbf{s}, L$)
This procedure prints a similiar message to to *sysprmishap*, but headed 'WARNING' instead of 'MISHAP'.

Chapter 5

In which we learn Important Facts about Variables

NOTE
GET THE SYNTAX OF DECLARATIONS RIGHT
I dont understand isglobal - ‘wrapping in a pair’.

Variables and procedures are highly interdependent concepts, especially in POP. In this chapter we explain the forms of variable declarations, and about the data-structures that are associated with variables. How variables behave in procedures is covered in Chapter 2. How variables are treated by the Virtual Machine is is explained in Chapter 16. For an introduction to POP variables, see Chapter 2.

5.1 Words identifiers and variables

On the close inspection that we shall give the POP variable in this chapter, it turns out to be quite a complicated concept. This should not be too surprising, since a number of kinds of data-object are involved that in a conventional language would be distributed between the compiler and run-time system.

In fact there are are three kinds of entity involved, *words*, *identifiers* and *variables proper*. The first two of these are POP objects, variables proper are not.

1. A *word* is a data-object of a class described in detail in Chapter 8.1. A word constant is enclosed in double quotes e.g. "+", "*enddo*", "x". The standard POP syntax for words is explained in Chapter 19. If we make the declaration *vars* x; we say that the word "x" is the *name* of the variable. Words have the property that if $W_1 = W_2$ then $W_1 == W_2$, that is to say, words which have the same sequence of characters are identical. In this they differ from strings, which are described in Chapter 9.

2. An *identifier* is a data-object which is normally a component of a word, and which contains information about how the word is currently acting as a variable, provided that it *is*. Thus information about whether a given word is a *macro* or an operator like + (see Chapter 2.26.1 and 2.7.1) is contained in the identifier associated with a word. This association is subject to change — a word can has a different associated identifier within a *section* (see Chapter 13) to that which it has outside.

3. A *variable* is a location in memory where a value associated with the name is stored. In the case of a so-called *permanent variable*, the variable is permanently associated with the *identifier* — indeed in current implementations of POP it is a field in the in the identifier. In the case of *lexical variables* the value is a field in a frame on the call-stack.

The *variable* associated with an identifier is the only entity that needs to exist when the procedures in which the identifier is referred to run.[1]

Computing terminology generally confuses words, identifiers and variables, and it is at

[1] In conventional languages the word and identifier records have been associated with compilers, whereas variables exist when the compiled program is run. However, in order to provide more supportive environments, in which variables can be accessed at run time, it is becoming customary to preserve the compiler information in a form that is available at run time. It is, of course, by no means universal practice to make the distinction between words and identifiers, although some apparatus has to be provided within a compiler to distinguish between different uses of the same name. E.g. two procedures may both have distinct variables called "x".

times convenient to do so in this book. Fortunately it is normally clear from the context which is meant. See 5.3

5.2 The syntax of variable declarations

A variable declaration can occur either inside or outside of a procedure body. The $< args >$ and $< results >$ of a procedure are special cases of a variable declaration, treated in Chapter 2.2. All of the effects of a variable declaration can be accomplished by procedure calls, as detailed later in this chapter. However, unless you are implementing some other kind of language within POP, you will usually use the syntax described in this section to accomplish variable declarations. Apart from the arguments and results of a procedure, the general form of a variable declaration is:

```
<declaration>    =  <variable_class> <varspec>* ;
                  | <dlocal_decl>
<variable_class> =  constant | dlvars | lvars
                  | vars | global constant | global vars
                  | active {<multiplicity>} {<identprops>}

<varspec>        =    <word> {,}
                  |  <word>  = <expression>,
                  |  <identprops> <varspec>
                  |  <identprops> ( <varspec>*){,}

<identprops>     =   macro | syntax | syntax <precedence>
                  | procedure | <precedence>
<precedence> = <decimal>
<multiplicity> = :<n>
```

Note that *dlocal* can also be used to specify that an expression should be saved and restored on procedure entry and exit, as described in Chapter 2.6.1. Note also that only 0 and *procedure* are allowed for the $< identprops >$ of *lvars* and *dlvars* inside a procedure.

A variable declaration accomplishes the following for a set of *words* that are referred to in the declaration:

- It establishes an association between a word and an identifier.
- It specifies how the value of the variable shall be stored. This includes *constant* declarations, in which case there is nothing to "vary" — the actual value associated with the word is bound into any procedure that refers to it.
- It may specify that the variable is restricted to have certain values.
- It may specify certain syntactic properties of the word.
- It may specify that a variable is to be "dynamically local", that is that it is to be saved on entry to a procedure definition and restored on exit.

5.2.1 What kind of identifier is created by a declaration?

Most of the declarations listed above will create a new identifier record associated with each word W declared in a $< varspec >$, with exceptions explained below:

- The *vars* and *constant* declarations, together with their *global* counterparts create a new *permanent identifier record* associated with the word W provided that no permanent identifier exists for the word in the current *section*. Sections are described in Chapter 13 Permanent identifiers may cease to exist only when they have been cancelled (see section 5.3.7), although they can be used for local variables by virtue of the save-and-restore mechanism described in Chapter **??**. If a global identifier does exist, these declarations may change its *identprops*, as specified below.
- The *lvars*, *dlvars* and *lconstant* declarations create a new *lexical identifier record* associated with W. This association exists only during the compilation of the innermost procedure in which the declaration occurs, or during the compilation of the file in which the declaration occurs if it occurs outside of any procedure definition. Note that in the latter case, an identifier record is in fact created to hold the variable value, but it becomes anonymous as soon as the file is compiled.

- The *dlocal* declaration may occur only in a procedure, and specifies that an existing variable, named W say, be subject to the save-and-restore mechanism. If W has not been declared as a variable elsewhere, it will be declared by the usual default mechanism, and a warning message printed. Otherwise *no identifier is created by a dlocal declaration.*

- Active variables, declared by the *active* declaration, generalise the notion of a variable with an associated value by allowing the actual value slot in an identifier record to contain not the associated value itself, but rather a procedure that will return that value when called. Thus, when an identifier is declared active, attempting to access its value will cause the procedure found in the value slot to be executed and its result returned; similarily, attempting to assign to the variable will run the updater of that procedure with the new value passed as its argument.

  Moreover, the mechanism is generalised still further by allowing the procedure associated with the variable and its updater to have not just one result/argument, but any fixed number of them: this number is called the *multiplicity* of the active variable. An access to an active variable of multiplicity n therefore produces n results, and a similar number must be given when assigning to it.

  Active variables are treated in detail in Chapter 2.28

5.2.2 Constants

Constant declarations can only occur outside of any procedure body. It is recommended that a constant should be initialised when it is created — e.g.

```
constant penelope = 'the wife of Odysseus';
```

Once initialised, a constant cannot be assigned to, and may be compiled literally into procedures. Uninitialised constants can be assigned to *once*, to specify their value. Constants created by *lconstant* have a scope which is the file in which they are declared — they cannot be referred to outside of that file.

5.2.3 Global variables and constants

The meaning of *global* is explained in Chapter 13.6.

5.2.4 What properties are specified by *identprops*?

The $<$ *identprops* $>$ occurring in a variable declaration can affect both what values a word W being declared may have, and its syntactic properties. You should see section 5.3.5 for a brief explanation of what these mean — a detailed exposition of the various options is to be found in Chapter 2. Note that the $<$ *identprops* $>$ only apply to the variable name that follows immediately, unless a number of variable names occur in parentheses. Thus, to declare a lot of procedures variables you need to say, e.g.:

```
vars procedure( Pooh Piglet Penelope Paris);
```

5.3 Identifier records

Identifier records are data-objects that represent program variables and constants. In general, they contain the following information:

- A field for holding the value of the identifier — its *idval*.
- Type information that restricts what objects can be assigned to the identifier — its *identtype*.
- A flag saying whether the identifier is an active variable or not, and if so, a field containing its multiplicity, as described in Chapter 2.28
- The syntactic properties of the identifier for the POP-11 compiler, which tell the latter how to interpret an occurrence of the identifier name in a program — *identprops*.
- An indication of whether the identifier is lexical or permanent, as treated below.

Although identifier records can be manipulated in their own right, program variables and constants are always specified to compilers, and to the POP Virtual Machine, by name, i.e. by word records which have identifiers associated with them (see 8.1).

5.3.1 Mapping from words to Identifiers

We have said above that identifiers are of two kinds : permanent and lexical. A lexical identifier only exists while the file or procedure that defines it is being compiled; permanent identifiers, on the other hand, have indefinite scope and exist until they are cancelled.

For this reason, the association from words to identifiers is maintained differently for the two kinds. For lexical identifiers, the POP VM holds an association list (see 11) which specifies the correspondence of words to identifier records, entries being deleted from this list when the scopes of identifiers terminate. For permanent identifiers, the association is achieved by making an actual field in the word record point directly to the identifier. It is not the case, however, that any given word record is constrained always to point to the same permanent identifier; as described in Chapter 13 the same word can be associated with different permanent identifiers in different program *sections*.

5.3.2 Procedures which can apply to words and identifiers

Because of the duality between words and identifiers, some of the procedures described below can take either an identifier or a word as argument However, since lexical declarations exist only at compile-time, these procedures, when given a word argument, will extract only a *permanent* identifier attached to the word. The only way of accessing a lexical identifier associated with a word is via *sys_current_ident*, described in 16.

5.3.3 Predicates On Identifiers

$isident(O) \to W_{kind}$
$isident(O) \to false$

This procedure returns $false$ if O is not an identifier record. Otherwise, W_{kind} indicates the restrictions on the identifier:

$"perm"$	permanent identifier
$"lex"$	'real' lexical identifier
$"lextoken"$	'token' lexical identifier

The difference between $"lex"$ and $"lextoken"$ is that a $"lex"$ is a real identifier record whose $idval$ is its actual run-time value, whereas a $"lextoken"$ is record being used by the POP VM to represent a procedure-local lexical variable, and which will not neccesarily play any part in the final compiled code for the procedure.

$isactive(W) \to n_{mult}$
$isactive(I) \to n_{mult}$

Given an identifier I or permanent identifier extracted from the word W, this procedure returns a $true$ value if the identifier is declared active, or $false$ if not. The true result returned is the active multiplicity of the identifier, i.e. an integer in the range $0 - 255$.

The following predicates test extra status information for an identifier or for a permanent identifier extracted from a word. In each case, if a procedure is applied to a word W, then the identifier I associated with W is extracted.

$isassignable(W) \to b$
$isassignable(I) \to b$

$isassignable$ returns $true$ if I (or $identof(W)$) can have its value updated (with $idval$ etc, or a program assignment statement), $false$ if not.

$isconstant(W) \to b$
$isconstant(I) \to b$

$isconstant$ returns $true$ if I (or $identof(W)$) has been declared as a constant, $false$ if not. Note that $isassignable$ will still be $true$ for a constant that has been declared but not yet assigned to.

$isprotected(W) \to b$
$isprotected(I) \to b$

$isprotected$ returns $true$ if I (or $identof(W)$) has been declared as a protected identifier, $false$ if not.

$isglobal(W) \to b$
$isglobal(I) \to b$

$isglobal$ returns $true$ if I (or $identof(W)$) is declared as a global permanent identifier. That is, if I is an identifier which will be automatically imported into daughter sections below its level. Otherwise it returns $false$.

Note that by default, the above four procedures all treat active identifiers as variables, i.e. $isassignable$ will always return $true$ and the other two will always return $false$. The status of the nonactive value can be got by wrapping I in a 'nonactive pair', i.e.

$$conspair(I,''nonactive'');$$

$isdlocal(W, P) \to b$
$isdlocal(I, P) \to b$

Given a word W or identifier I and a procedure P, this returns $true$ if W or I is a dynamic local variable of P, $false$ if not.

5.3.4 Constructing Identifiers

$consident(O_{idprops}, b_{const}, W_{kind}) \to I$
This procedure gives a way to construct identifier records directly, without the associated declaration of a word. The $O_{idprops}$ argument specifies the identprops and/or activeness of the identifier: permissible values are as for $sysSYNTAX$ (see Chapter 16). The boolean b_{const} argument should be $true$ if a constant identifier is desired, $false$ otherwise. Note that, unlike $sysSYNTAX$, this is em not affected by the value of $popconstants$. W_{kind} specifies the kind of identifier that I will be; permissible values are the words $''perm''$, $''lex''$ and $''lextoken''$.

5.3.5 Accessing Information About Identifiers

See also the $is\_$ predicates above for other properties of identifiers.

$identprops(W) \to O_{idprops}$
$identprops(I) \to O_{idprops}$

This procedure returns the identprops (macro/POP-11 syntax properties) of the identifier I (or of $I = identof(W)$). This can take the following values:

Value	Chapter	Notes
"*undef*"		I is a word not declared as a permanent identifier.
0		Identifier with no type restriction or special syntactic properties.
"*procedure*"		Procedure-type identifier
n	2.7.1	POP-11 operation of precedence n, where n is an *integer* or *decimal* in the range -12.7 to 12.7 .
"*macro*"	2.26.1	POP-11 macro.
"*syntax*"	2.27	POP-11 syntax word.
"*syntax n*"	2.27	POP-11 syntax operator of precedence n, range as above.

Note that "*syntax n*" is not a normal itemisable word in POP-11, but is produced using *consword*, e.g. *consword*('*active**s*' ><n) — see Chapter 8.1 about *consword*.

identtype(W) → *Type*
identtype(I) → *Type*

This procedure returns the data type of the identifier I, or permanent identifier I extracted from the word W. Possible values are

"*undef*"	Not declared as a permanent identifier
0	Untyped, i.e. may hold anything.
"*procedure*"	May hold procedures only.

full_identprops(I) → $L_{idprops}$
full_identprops(W) → $L_{idprops}$

This procedure returns a list of all the declaration keywords for the identifier (or permanent identifier extracted from the word) I, or "*undef*" if I is a word not declared as a permanent identifier. The list has the form

```
[<prot> <glob> <const/var> <type> <idprops>]
```

where

< *prot* >	"*protected*" for a protected identifier, empty otherwise;
< *glob* >	"*global*" for a global permanent identifier, empty otherwise;
< *const/var* >	"*constant*" for a constant (preceded by "*assignable*" if the identifier can still be assigned to), or "*vars*" for a variable;
< *type* >	"*procedure*" for a procedure-type identifier, empty otherwise;
< *idprops* >	the identifier's *identprops* (except that 0 for an ordinary untyped identifier is omitted, and "*syntax n*" for a syntax operator is returned as "*syntax*" followed by the number n).

5.3.6 Manipulating Values of Identifiers

For more information on dynamic local variables of procedures see Chapter **??**

$identof(W) \to I$

This procedure returns the permanent identifier currently attached to W. If W is undefined as a permanent identifier, then an attempt is made to autoload the word from the system library — if this fails the message

```
'DECLARING VARIABLE <W>'
```

is printed, and an identifier with *identprops* 0 and value an *undef* record is attached to W and returned.

ident

This syntax word compiles code to push an identifier object onto the stack. Its usage is

```
ident <name>
```

where $< name >$ is a word declared as any kind of identifier; the effect is to push onto the stack the identifier associated with the word at the time of compilation. This syntax word operates by calling $sysIDENT$ for the given word — see 16 for more details.

$idval(I) \to O$
$O \to idval(I)$

This procedure returns or updates the value cell of the identifier I. In update mode, O must be valid for the *identtype* of I. When I is an active variable, *idval* runs the nonactive procedure value of I, or its updater in update mode. The number of items produced as results by the procedure, or taken as arguments by its updater, will then be the multiplicity n of the variable.

$nonactive\_idval(I) \to O$
$O \to nonactive\_idval(I)$

This procedure returns or updates the value cell of the identifier I, regardless of whether the identifier is active or not.

$valof(W) \to O$
$O \to valof(W)$

This procedure is functionally equivalent to

$$idval(identof(W)) \to O$$

and

$$O \to idval(identof(W))$$

in update mode.

$nonactive\_valof(W) \to O$
$O \to nonactive\_valof(W)$

This procedure is the same as $valof$, but using $nonactive\_idval$.

$caller\_valof(W, P_{caller}) \to O$
$O \to caller\_valof(W, P_{caller})$

This procedure returns or updates the $valof$ of the word W as it would be in the environment of the currently-active procedure specified by P_{caller}. The argument P_{caller} may be either

- An actual procedure or a caller number as input to the *caller* described in Chapter 2.
- $false$, meaning that the value outside of all dynamic localisations (i.e. outside all procedure calls) is accessed/updated.

N.B. In present implementations, this procedure does not deal with active variables.

$set\_global\_valof(\mathrm{O}, W)$
This procedure uses $caller\_valof$ to assigns O to be the valof W in the context of every currently active procedure for which the identifier associated with W is a dynamic local.

$recursive\_valof(W) \to O$
Recursively applies $valof$ to W while O is a word, and returns the result — that is, $recursive\_valof$ calls $valof$ with W as argument: if the result O is a word, then $recursive\_valof$ is called with O as argument; if it is not a word, O itself is returned.

5.3.7 Manipulating Attachment of Permanent Identifiers to Words

The following procedures allow you in effect to make a variable have more than one name, to modify the procection flag of an identifier and to make a variable anonymous.

$syssynonym(W_1, W_2)$
Attaches to the word W_1 whatever permanent identifier is currently attached to the word W_2, so that they both refer to the same identifier. W_2 is automatically declared if there is no permanent identifier currently associated with it.

$sysprotect(W)$
This procedure *protects* the permanent identifier associated with W — that is, it stops W being redeclared, i.e. given a new identifier, and disallows the compilation of any assignment to the identifier.

$sysunprotect(W)$
This procedure removes the protection against assignment of the permanent identifier associated with W. If *sysprotect* has been used to protect W, $sysunprotect(W)$ allows the word W to be redeclared, that is to say given a new identifier, and permits compilation of assignment to the identifier.

$syscancel(W)$
This procedure cancels any permanent identifier currently attached to the word W, that is to say it makes the identifier be no longer attached to W, although it may continue to exist if it is referred to otherwise, e.g. in a procedure definition. Since "currently attached" refers to the state of affairs in the current *section*, the cancellation does not affect the attachment of the word to the identifier in other sections (except for imported identifiers).

Once a word is cancelled it can be used afresh with no thought to its use in the past. There is a library macro *cancel* which provides a more convenient syntax.

The actual behavior of *syscancel* is complex, and you should probably not use it. Cancellation was initially provided in POP-2 before *sections* were devised, and most of its usages can be better accomplished by *sections*.

5.4 Undef Records

When a non-lexical variable is first declared it is given an initial value which is an *undef record* and which contains only one field, its *undefword*. Undef records are also used for other purposes. The *undefword* may be a word or *false*; a new permanent identifier is initialised to a newly-constructed undef record whose *undefword* is the name of the identifier, whereas new global lexical identifiers are initialised to a fixed undef record containing *false* (this prints as $< undef >$ as opposed to $< undef\ Pooh >$ for one containing the word $''Pooh''$). The only exception to this is a procedure-type identifier: because one of these must always have a procedure value, it is initialised to a closure of a system error procedure, partially applied to the actual *undef record* for the identifier, where the error procedure will produce an appropriate mishap if an attempt is made to apply it. For consistency these 'undef' closures are also recognised by *isundef* and *undefword*.

$isundef(O) \rightarrow b$
This procedure returns *true* if O is an undef record or an 'undef' closure, *false* if not.

$consundef(W) \rightarrow Undef$
$consundef(false) \rightarrow Undef$

This procedure creates a new undef record $Undef$ for which $undefword(Undef) = W$.

$undefword(Undef) \rightarrow W$
$undefword(Undef) \rightarrow false$

If $Undef$ is an undef record, this procedure returns its single component. If it is closure of *mishap*, the *undef* record is first extracted from the frozen values of the closure.

5.5 Constants

ident_key
undef_key

These constants hold the key structures for identifiers and undef records (see 3.13).

Chapter 6

Numbers in POP

NOTE
some arithmetic quote from Pooh.

NOTE - I must find out how to set a tilde as and not above the following character. The sections on random numbers and on extracting mantissa of floats need attention.

6.1 Introduction

This chapter provides a complete description of the operations on numbers available in POPLOG POP-11. AlphaPop provides a subset of these, as described in section **??**. This chapter does not describe in detail the forms in which numbers are input — for that you should turn to Chapter 19.

Representing numbers in a computer presents considerable problems. Even the simplest kind of numbers, the natural numbers 0,1,2,3..., form an infinite set, and a computer is a finite machine! Worse still are the real numbers, which include numbers like the square root of 2 or π, since you cannot represent these by a finite sequence of digits. You can only approximate them by a finite sequence of digits, and the errors introduced in approximations have a nasty habit of building up until you have nonsense.

No existing computer language can deal with these problems automatically. The subject of Numerical Analysis is concerned with how you can do computations with approximate representations of numbers and arrive at an answer which is approximate, but for which you know *how* approximate. To take an elementary example, you cannot write down the value of π in a finite number of digits, but you can say for sure that $3.1415 < \pi < 3.1416$.

POPLOG POP provides you with a useful tool-kit to deal with these problems — it consists essentially of ways of representing numbers that have been found useful by various researchers.[1] The *data-classes* which are the basis of this tool-kit are the following:

- *Small integers.* These are integers which can be represented in a fixed number of binary digits. The number of digits depends on the implementation, but is usually 30, so a small integer is less in magnitude than 2^{30}. Thus, apart from 2 missing binary digits, which are used by POP to help it identify data-class — see Chapter 3.13NOTE ???, small integers are very similar to integers provided in languages such as Pascal and C. However, if you try to make an integer bigger than can be represented as a small integer, you do not get an error, instead you will form:
- *Big integers.* These are integers which are too big to be small integers. For example 2^{100} will be a big integer. To check out the big-integer capabilities of your system, try defining the factorial function, and asking it to compute 1000! (factorial 1000). The only limit on the size of a big integer in POP is the amount of memory your computer has!
- *Rationals.* A rational number is a fraction, that is a ratio of two integers (big or small). You may be surprised to find them provided — after all you probably gave up working with fractions at high-school when you learnt about decimals. However they are important in computational algebra (that is using a computer to do algebra, as described in Chapter **??**) because they can represent exactly numbers like 2/3 which cannot be exactly represented in a finite number of decimal digits (or binary digits, which is more to the point). Programs for computer algebra have to have statements

[1] The POPLOG numeric capabilities are derived from those of Common Lisp. Many of the capabilities embodied are derived from the needs of mathematical reasoning systems like Macsyma.

like "replace $x * 0$ by 0". If you are using approximate representations of numbers, then such a statement has to be rendered as "replace $x * \epsilon$ by 0, when ϵ is sufficiently small", and such statements can cause many problems — either you choose ϵ to be too small, in which case simplifications are left undone which ought to have been done, or you choose it too large, in which case expressions appear to be equal which ought not to appear equal.

This problem can be avoided by using rational numbers, and representing irrational numbers symbolically. E.g. you would have the *word* "*pi*" rather than any approximation to π in your algebraic expressions. Physical constants like Planck's constant, h, would be treated similarly. Only when you have massaged your algebraic expressions into something near the desired form will you substitute approximate values for these symbolic constants — assuming you want a numeric answer to compare with experimental results or as the thickness of a beam.

Rational numbers are created whenever you try to divide one integer i_1 by another i_2 using the / operator, where the result of the division cannot be exactly expressed as an integer.

There is a danger for users unfamiliar with POP created by the fact that there are two options available for printing out rational numbers. If you do

```
4/6=>
```

POP may respond:

```
** 2_/3
```

or you may get

```
** 0.666667
```

depending on the setting of the variable *pop_pr_ratios*. As explained later, rational numbers are normally reduced to their lowest terms. Now rationals are more expensive in space and computing with them takes more time than using pop *decimals* (the floating point numbers described below), and if you try to implement some of the standard iterative methods (e.g. Newton-Raphson) you may find that you are inadvertently using rational numbers, and generating ratios with huge numerators and denominators. For this reason, you should always set *pop_pr_ratios* to be *true*, except when you specifically want to see the decimal equivalent of a rational number, so that you know when you are dealing with rationals.

- *Short floating point numbers.* These are known in POP as *decimals*. The term is somewhat misleading, since only the printed representation is, by default, decimal. They provide an approximate representation of real numbers, including rationals. Floating point numbers are usually input in the syntactic form

  ```
  <digits>.<digits>
  ```

  e.g. 2.0. Apart from the coercion procedures described in section 6.12, any operation that creates a floating point number will create a short floating point number if the variable *popdprecision* is $false$. Otherwise a long floating point number (*ddecimal*) is created. Short floating point numbers are simple items (see Chapter 3.7), and offer considerable savings in space and time over long floating point numbers, which are compound items. If you want to be sure that a procedure is using floating point numbers and not rationals, you can use the *number_coerce* procedure described in section 6.12. An alternative that will often be available to you is to use floating point constants, and rely on "floating point contagion" — see 6.7.

  Floating point numbers, being approximations, do not exactly obey all of the laws of algebra. Floating point addition and multiplication are *commutative*, that is $d_1 + d_2 = d_2 + d_1$ and $d_1 * d_2 = d_2 * d_1$, but they are not *associative*, that is $x + (y + z)$, is not necessarily equal to $(x + y) + z$. This becomes painfully obvious if x and y are of large and equal magnitude and opposite sign, whereas z is small. Try, for example:

  ```
  vars  x = 10**20,
        y = -x,
        z=0.1,
        popdprecision = false;
  x+(y-z) =>
  ** -2004090000000.0
  ```

  You will be somewhat disturbed by how inaccurate the result is! This result was obtained using $decimals$, and the effect can be palliated by using $ddecimals$. However you will be able to construct an example in which even *ddecimals* gives a manifestly inaccurate result.

- *Long floating point numbers* These are known in POP as $ddecimals$. and provide a more precise representation of real numbers than do short floating point numbers.

- *Complex Numbers* To follow this chapter completely you will need to know something about *complex numbers*. If you are not familiar with complex numbers, you can still use the chapter for reference, since all of the numbers you know about are special cases of complex numbers, and almost all the procedures defined in this chapter will work on them.

The most important property of the complex numbers is that they are *algebraically closed*, that is to say, that any polynomial equation in one variable of degree n has exactly n roots (provided you count multiple roots correctly).

In the text, we will refer to a number which may be a complex number by z, z_1, $z_2, \ldots$. The main difference you will find if you are not used to complex numbers is that because they are algebraically closed, some operations which you would expect to give an error will give an answer! Try

```
vars iii = sqrt(-1);
iii =>
** 0.0_+:1.0
```

A complex number in POP is essentially a record, as described in Chapter3.7, containing two real numbers. If the second one (known as the *imaginary part*) is zero, then the complex number is identified with the real number which is the first component. Try typing $iii * iii \Rightarrow$.

Details of how you write numbers in program text, or for data input, are not found in this chapter, but in Chapter 19.

6.2 How the numeric data-classes represent numbers in POP

Classically, mathematical understanding of numbers has reached its most complete development in *complex numbers*, which are an algebraically closed *field*[?]. Thus a general purpose computational system should aim to provide a representation of the complex numbers. Other number systems, the Natural Numbers, the Integers, the Rationals and the Reals can be *embedded* in the complex numbers in a standard way. For example we can identify the integer 1 with the complex number $1 + 0i$

As we stated in section 6.1, most real numbers cannot be exactly represented as finite objects. However integers and rational numbers can. This makes it advantageous for us to have two major kinds of representation:

1. An exact representation of rational numbers, incuding integers. Within this representation, the procedures representing the operations on the field of rationals are partial functions which produce exact results. Apart from dividing by zero (which is never allowed in any field) these procedures are partial functions only in that they may fail to produce a result because insufficient memory is available.

2. An approximate representation of real numbers in floating point format. In POP-11 these are *decimals* and *ddecimals*. In this representation, while certain integers and rationals can be represented exactly, most procedures are approximate, and algebraic laws only hold approximately. The *decimals* are "single-precision" floating point, while the *ddecimals* are "double-precision".

A given rational number r will have a unique representation in any of the types above. The first type (integers and rationals) will give an exact representation, and the floating point representation should be the nearest member of the data-class to r, with a suitable convention for tie breaking.

A given real number x which is not a rational will have no unique representation as a rational, since rationals can have numerator and denominator of arbitrary size, and so any rational approximation can be bettered. It will have a unique, but inexact, representation as a *decimal* or *ddecimal*. [2]

The POP procedures which implement irrational and trancendental functions will usually produce the best *decimal* approximation to the real result of applying the mathematical function, but they will not usually produce the best *ddecimal* approximation, because they use standard double-precision floating point computation, which has the same precision as *ddecimal* numbers, and cannot be expected to be accurate to the least significant digit.

[2]It might be argued that both in the case of floats and rationals, there is an implementation dependent nearest member of the class, since the biggest integer that can be constructed is determined by the machine word size. Trying to find the nearest rational to a real is not a practical computational aim. Most of the time, POP will find the nearest short-float to a given real (expressed,say, as a trancendental function applied to a float), although it will fail to find the nearest long-float, since the computation is done to the same precision as the result.

6.3 Other possible number representations

In this section we discuss other ways in which numbers might be represented, and in which they could indeed be represented in POP, if you care to define the appropriate data-types, and write the appropriate procedures.

6.3.1 Tracking floating point errors

Suppose d_1 and d_2 are two floating point numbers. Then if, for the purposes of this discussion, we use $+_{fp}$ to mean floating point addition, whereas $+$ is used to mean mathematical addition:

$$|(d_1 + d_2) - (d_1 +_{fp} d_2)| < k * max(|d_1|, |d_2|)$$

That is to say, the error in floating point addition is less than a constant k times the maximum of their absolute values. Here the value k will depend upon the floating point precision being used — i.e. there will be a different value of k for *decimal* operations and for *ddecimal* operations. Some computers have been built[**?**] which record the precision of the result of floating point operations along with the result, and it should be clear to you how you could construct a record-class of such numbers and provide procedures to operate upon them.

6.3.2 Representing Algebraic Numbers

Algebraic numbers are complex numbers which are solutions to a polynomial equation with rational (or equivalently integer) coefficients. Thus, for example, the square-roots of rationals are algebraic numbers. It is possible to show that π is not an algebraic number, and therefore the classic problem of squaring the circle is insoluble — this arises because, given a ruler and compass and rational numbers, you can only create a subset of the algebraic numbers by the classic Greek geometric constructs.

One way to represent an algebraic number is by its defining equation. However, since such an equation has, in general, more than one root, you need to specify which root you mean. In the case of *real* algebraic numbers, this can be done by specifying an interval in which the root must lie. Fortunately, given any algebraic equation, it is possible to find intervals bounded by rationals in which there is only one root of the equation, so that an algebraic number can be uniquely represented by an equation and a rational interval. How to add, subtract, multiply and divide such numbers is described in Loos[?].

Such algebraic numbers have applications in representing geometric shapes computationally. The shape of many mechanically engineered objects can be characterised as *semi-algebraic sets*, that is to say subsets of Euclidean space bounded by surfaces which have algebraic equations. If you use floating point computations to treat such shapes you can get into trouble — questions that ought to give the same answer may give a qualitatively different one. For example the question "how many times does this curve intesect this surface" might give inconsistent answers if we had different characterisations of the same curve. These problems can be avoided by using a proper treatment of algebraic numbers.

6.3.3 Representations of the reals

The real numbers are classified as *algebraic* or *trancendental*. Real numbers have to be defined as infinite entities. Naively, we might try to define reals as infinite sequences of digits, but this poses some problems — for example we can only generate such sequences left-to-right, but carries take place right-to-left, and can begin arbitarily far to the right.

The best known mathematical treatment of the reals is the Dedekind embedding of them in Set Theory. In this embedding *each* real number is an infinite set of rationals. More intuitively appealing constructions have reals be equivalence classes of infinite sequences of rationals which satisfy some convergence criterion.

6.4 A Comparison with Common LISP

POPLOG POP-11 provides the same data types and associated functions as those specified by the Common LISP standard. This includes integers, bigintegers, *decimals*, *ddecimals*, ratios and complex numbers, although LISP uses a different nomenclature. There are some minor differences in the procedures provided, but the POPLOG capabilities essentially include all of the LISP ones described in [?].

6.5 Note on Terminology

In this chapter the term *integer* means either a simple integer or a biginteger, except where explicitly qualified as one or the other; *integral* means the same in the nominal sense, or *integer-valued* as an adjective. Similarily, *real* means *non-complex*, or *non-complex number* — it does *not* carry the meaning of *floating-point number* used by some programming languages.

6.6 Fast integer operations

All POP arithmetic operations described in this chapter check the types of their arguments at run time in order to determine what to do. This gives great flexibility in writing generic procedures, but can mean that efficiency is sacrificed. As a partial solution fast, non-checking integer operations are provided. These are described in **??**. Further information concerning efficiency is available in **??**.

6.6.1 Computation on Rationals

We have stated that rational numbers have a unique representation: integers, bigintegers and ratios are mutually disjoint in terms of the numbers they represent, and that no two ratios represent the same number unless they have the same numerator and denominator. In other words, a ratio is always reduced to its lowest common terms, by dividing numerator and denominator by their greatest common divisor; if this makes the denominator equal to 1, the result is the integer numerator. This is called the rule of *rational canonicalisation*. Any integral result from a computation will always produce a simple integer, rather than a biginteger, if it can be so represented.

6.7 Procedures that operate on numbers

With a few exceptions, numerical procedures are *generic*, that is, will accept any kind of numbers as arguments. The procedures that operate on numbers can be classed as:

1. *Algebraic procedures.* These are procedures which implement the basic operations of the *field* of complex numbers, and those directly derived from them. The central ones are $+, -, *, /$. When applied to arguments that are all rational, they will produce rational results.
2. *Irrational procedures.* These are procedures which correspond to functions which map rationals onto irrationals at infinitely many places, e.g. *sqrt*. They include procedures like *sin* which correspond to mathematical functions which produce trancendental results when applied to some rational numbers.
3. *Access procedures.* These are used to access fields of the records that make up many number classes, e.g. *denominator* is used to access the 'top line' of a ratio. They will call *mishap* if applied to arguments of an inappropriate number-class.

In executing procedures of type (1) above, when or one or more of the arguments is floating-point, all arguments are converted to double-length floating-point and the computation performed with double-float arithmetic. This is called the rule of *floating-point contagion.* The result of the computation is returned by the procedures as a floating-point number which may be short or long depending on the value of the variable *popdprecision* as described below. Procedures of type (2) above always perform the computation in double-float arithmetic, and return a result short or long depending on *popdprecision*.

6.8 Complex Numbers

Complex numbers always have both real and imaginary parts of the same representation class, and so may similarily be sub-divided into rational complex, single-float-complex and double-float-complex. Aside from those irrational functions that can produce a complex result from a real argument (*sqrt* applied to a negative real, for example), the only way of constructing complex numbers is with the operators $+:$ and $-:$, which are read as *'plus i times'* and *'minus i times'*). These obey the same rules as for real arithmetic: with both arguments rational or rational-complex the result is a rational-complex; if either argument is floating or float-complex, the result is a float-complex whose format depends on *popdprecision*.

The only rider to this is the rule of *rational-complex canonicalisation*, which prevents the production of a rational-complex number with a 0 imaginary part. Instead, the result is just the real part of the number. Note that this does not apply to float-complex numbers, which can have a 0.0 imaginary part.

Most numerical procedures allow complex or mixed real and complex arguments. In a similar way to the rational/float distinction for the real case, computations are done in real arithmetic producing a real result if all arguments are real, or otherwise all arguments are converted to complex and the operation performed in complex to give a complex result. This called the rule of *complex contagion*.

6.9 Conventions for Formal Parameters of Procedures

As is the practice in this book, we have attempted to use one letter to indicate object type. Subscripts distinguish between distinct variables of the same type, and can be either numeric or descriptive.

O	any POP object
b	a boolean, *true* or *false*
d, d_1, d_2	a floating point number, *decimal* or *ddecimal*
r	A rational number
z, z_1, $z_2\ldots$	any number, including complex
x, x_1, $x_2,\ldots$, y	any number except complex
n	a short integer
i_1, $i_2\ldots$	any integer, short or big
	Use of i is avoided to prevent confusion with $\sqrt{-1}$
θ	a real number, where an angle would be expected.

6.10 Predicates Relating to Numbers

The following procedures are provided to recognise number classes and types.

$isinteger(O) \to b$
This procedure returns $true$ if O is a simple integer, $false$ otherwise.

$isbiginteger(O) \to b$
This procedure returns $true$ if O is a biginteger, $false$ otherwise.

$isintegral(O) \to b$
This procedure returns $true$ if O is a simple integer or a biginteger, $false$ otherwise.

$isratio(O) \to b$
This procedure returns $true$ if O is a ratio, $false$ otherwise.

$isrational(O) \to b$
This procedure returns $true$ if O is a simple integer, a biginteger or a ratio, and $false$ otherwise.

$isdecimal(O) \to b$
This procedure returns $true$ if O is a $decimal$ or a $ddecimal$ — i.e. a floating point number, $false$ otherwise.

$issdecimal(O) \to b$
This procedure returns $true$ if O is a single length decimal — i.e. a single-precision floating point number, $false$ otherwise.

$isddecimal(O) \rightarrow b$
This procedure returns $true$ if O is a $ddecimal$ — i.e. a double precision floating point number, $false$ otherwise.

$isreal(O) \rightarrow b$
This procedure returns $true$ if O is any number except a complex, $false$ otherwise.

$iscomplex(O) \rightarrow b$
This procedure returns $true$ if O is a complex number, $false$ otherwise.

$isnumber(O) \rightarrow b$
This procedure returns $true$ if O is any kind of number, $false$ otherwise.

6.11 Comparisons on Numbers

$z_1 = z_2 \rightarrow b$
$z_1 / = z_2 \rightarrow b$

On numbers, these operators compare their arguments for mathematical equality and inequality respectively. Different types of rationals (integers, bigintegers and ratios) can never represent the same number and so can never be equal; comparisions between floating-point (*decimals* and *ddecimals*) and between floating-point and rationals first convert both arguments to double float. The same rules apply to the comparison of the real and imaginary parts of a complex numbers. Note that a real number compared with a complex number will be equal only if the complex number is a float-complex with 0.0 imaginary part (since the imaginary part of a rational complex must always be non-zero).

$x_1 < x_2 \rightarrow b$
$x_1 <= x_2 \rightarrow b$

$x_1 > x_2 \to b$

$x_1 >= x_2 \to b$

These operators compare x_1 and x_2 and return b indicating whether x_1 is respectively less than, less than or equal, greater than, and greater than or equal to x_2. Comparisons between different number types are performed as for = and /=. Both arguments must be real numbers, in the POP sense, that is non-complex numbers.

$max(x_1, x_2) \to x_{max}$
$min(x_1, x_2) \to x_{min}$

max returns the greatest of its two arguments and *min* the least. I.e. $x_{max} \geq x_1$ and $x_{max} \geq x_2$, $x_{max} = x_1$ *or* $x_{max} = x_2$. Similarly $\leq$ is used for *min*. Both arguments must be real numbers.

$z_1 == \#z_2 \to b$
This procedure returns *true* if z_1 and z_2 are identical (i.e. ==), or numbers of the same representational type and numeric value. *Decimals* and *ddecimals* are *not* considered to be of the same type. Two complex numbers are ==# if their real parts are ==# and their imaginary parts are ==#

6.12 Variables Controlling Number Representation

popdprecision [variable] This value of this variable controls the production of results from floating-point computations, in combination with the types of the arguments supplied to the relevant procedure. In the following, *decimal* includes *decimal-complex* and *ddecimal* includes *ddecimal-complex* (when a complex floating-point operation is involved):

value	effect
$false$	Single-float decimal results are always produced.
the word "*ddecimal*"	A *ddecimal* result is produced only if one or other (or the only) argument was *ddecimal*. (This is the behaviour specified by Common LISP.)
any other	Same as the previous case, except that a *ddecimal* result is also produced when neither argument is a single-float decimal, i.e. all argument(s) are *ddecimal* or rational.

Note that the default value of *popdprecision* is $false$.

pop_reduce_ratios [variable] It was stated above that a ratio result is always reduced to its lowest common terms, and therefore to an integral result if the denominator becomes 1. However, in situations where a rational computation is being performed that involves a number of intermediate results, the continual reduction of intermediate values can be rather time-consuming; this boolean variable is therefore provided to enable reduction to be turned off by setting it to $false$. Although unreduced ratios will give correct results in computation, comparisons on them may not do so. E.g. $2/2 = 1$ will evaluate to $false$. Thus this facility must be used *carefully*: *pop_reduce_ratios* should only be set $false$ inside a procedure that has it as a dynamic local, and you should always ensure that the variable is returned to $true$ before producing the final result of a computation.

$number\_coerce(z_1, z_{patt}) \rightarrow z_2$

This procedure produces a number z_2 which is the number z_1 converted to the representation class (i.e. rational, single-float decimal or double-float *ddecimal*) of the number z_{patt}, as follows:

- No new number is constructed unless necessary; thus if the class of z_1 already matches that of z_{patt}, z_1 is returned unchanged.

- Otherwise, conversion from one class to another proceeds, as described below.

Conversion from rational form to floating-point form, or from one float format to the other, takes place in the obvious way. For conversion from floating-point to rational, the float is assumed to be completely accurate, and a mathematically equal rational number is returned.

If z_1 is complex, then z_2 is the complex number got by applying *number_coerce* recursively to the real and imaginary parts of z_1, i.e.

$$number\_coerce(realpart(z_1), z_{patt}) + : number\_coerce(imagpart(z_1), z_{patt}) \rightarrow z_2$$

If the second argument z_{patt} is itself complex, then z_1 is coerced to a complex number of the same representation class, i.e. the result is computed as

$$realpart(z_{patt}) \rightarrow z_{patt};$$
$$number\_coerce(realpart(z_1), z_{patt}) + : number\_coerce(imagpart(z_1), z_{patt}) \rightarrow z_2$$

where *imagpart* will fill in an appropriate zero value for the imaginary part if z_1 is real.

6.13 Arithmetic Operations

$z_1 + z_2 \rightarrow z_3$
$z_1 - z_2 \rightarrow z_3$

$z_1 * z_2 \rightarrow z_3$

$z_1 / z_2 \rightarrow z_3$

These operators respectively add, subtract, multiply and divide their arguments, which may be any numbers. The type of the result depends on the rules of floating-point and complex contagion as described above. In particular, note that dividing one integer by another produces a ratio when the result is not exact.

- $z_1 \rightarrow z_2$

As a prefix operator, $-$ is equivalent to negate(z_1).

$z_1 // z_2 \rightarrow z_{quot} \rightarrow z_{rem}$

z_1 *div* $z_2 \rightarrow z_{quot}$

z_1 *rem* $z_2 \rightarrow z_{rem}$

The two results returned by the operator // are defined by

$$intof(z_1/z_2) \rightarrow z_{quot} \;\; and \;\; z_1 - (z_{quot} * z_2) \rightarrow z_{rem}$$

where the arguments may be any numbers, including complex. The operator *div* returns just the quotient z_{quot} as defined above, and the operator *rem* returns just the remainter z_{rem}.

x_1 *mod* $x_2 \rightarrow x_{mod}$

This procedure returns x_1 modulo x_2, where both numbers must be real. This is defined as

```
define 2 x1 mod x2 -> x_mod;
  lvars x1, x2, x_mod,
        x_rem = x1 rem x2;
     if     (x2 > 0 and x_rem < 0)
         or (x2 < 0 and x_rem >= 0)
     then
         x_rem+x2 -> x_mod
     else
         x_rem -> x_mod
     endif
```

```
enddefine;
```

Thus the result of *mod* always has the same sign as the divisor.

$intof(z) \to i$
For z real, *intof* truncates its argument to an integer, i.e. it returns the integer of the same sign as z and with the largest magnitude such that $abs(i) \leq abs(z)$. For a complex number, the result is the integral complex number obtained by applying *intof* to its parts, i.e.

$$intof(realpart(z)) + : intof(imagpart(z))$$

$fracof(z) \to z_{frac}$
The fractional part of z, defined as $z - intof(z)$.

$round(z) \to i_1$
For z real, this procedure rounds z to an integer by taking $intof(z + 1/2)$ if z is positive, or $intof(z - 1/2)$ otherwise. If z is complex, the result is the Gaussian integer:

$$round(realpart(z)) + : round(imagpart(z))$$

$abs(z) \to x$
This procedure returns the absolute value of z, which (except for complex) will always be a number of the same type. For any complex z, the result will be a floating-point real, computed as

$$sqrt(realpart(z)^2 + imagpart(z)^2)$$

$negate(z_1) \to z_2$
This procedure returns the negation of z.

$sign(z) \to z_{sgn}$
For z real, $sign$ returns -1, 0 or 1 of the same type as z, depending on whether z is negative, zero or positive. If z is complex, the result is a floating-point complex number such that

$$abs(z_{sgn}) = 1.0,\ phase(z_{sgn}) = phase(z)$$

6.14 Rational & Integer Specific Operations

$checkinteger(O, i_{lo}, i_{hi}) \to b$
This procedure checks whether O is an integer within the range specified by lower bound i_{lo} and upper bound i_{hi} (inclusive). Either or both bounds may be $false$ to indicate no upper or lower limit. If all conditions are satisfied the procedure returns with no action, otherwise a mishap occurs.

$gcd\_n(i_1, i_2, ..., i_n, n) \to i_{gcd}$
Computes the greatest common divisor of the all the n integers i_1, i_2, ..., i_n, where the number n itself (a simple integer ≥ 0) appears as the rightmost argument. If $n = 0$, then $i_{gcd} = 0$; if $n = 1$, then $i_{gcd} = i_1$.

$lcm\_n(i_1, i_2, ..., i_n, n) \to i_{lcm}$
Computes the least common multiple of the all the n integers i_1, i_2, ..., i_n, where the number n itself (a simple integer ≥ 0) appears as the rightmost argument. If $n = 0$, then $i_{lcm} = 1$; if $n = 1$, then $i_{lcm} = i_1$.

$destratio(r) \to i_{denom} \to i_{num}$

$numerator(r) \rightarrow i_{num}$

$denominator(r) \rightarrow i_{denom}$

These procedures return the numerator and denominator parts of a rational number, either together (*destratio*), or separately (*numerator* and *denominator*). When r is integral, then $i_{num} = r$, and $i_{denom} = 1$.

6.15 Complex Specific Operations

$z_1 + : z_2 \rightarrow z_3$
$z_1 - : z_2 \rightarrow z_3$

These two operators are the basic way of creating complex numbers. Effectively, they both multiply their second argument by i (the square root of -1), and then either add the result to $(+:)$ or subtract the result from $(-:)$ the first argument.

$+ : z_1 \rightarrow z_2$
$- : z_1 \rightarrow z_2$

As prefix operators, $+:$ and $-:$ are equivalent to $unary\_+ : (z_1)$ and $unary\_- : (z_1)$ respectively.

$unary\_+ : (z_1) \rightarrow z_2$
$unary\_- : (z_1) \rightarrow z_2$

Single-argument versions of $+:$ and $-:$, which multiply their argument by i and $-i$ respectively.

$conjugate(z_1) \rightarrow z_2$
This procedure returns the complex conjugate of its argument. The conjugate of a real

number is itself, while for a complex number it is

$$realpart(z_1) - : imagpart(z_1)$$

i.e. a complex number with the same *realpart*, but negated *imagpart*.

$destcomplex(z) \rightarrow y \rightarrow x$
$realpart(z) \rightarrow x$

$imagpart(z) \rightarrow y$

These procedures return the real and imaginary parts of a complex number, either together ($destcomplex$), or separately ($realpart$ and $imagpart$). When z is real, then $x = z$, and a zero of the same type as z is returned for y.

6.16 Functions over the Complex Numbers

The procedures in this section, as well as most of those in the following section on trigonometric procedures, compute functions whose definitions on the complex plane necessitate choices of branch cuts and principal values. See the section *Branch Cuts, Principal Values and Boundary Conditions* in Chapter 12 of Steele[**?**] for details of these. $sqrt(z_1) \rightarrow z_2$

This procedure returns the principal square root of z_1. This will be a real positive floating-point number if z_1 is real and non-negative, and a float-complex otherwise.

$log(z_1) \rightarrow z_2$
This procedure returns the natural (base e) logarithm of z_1, which must not be a zero of any kind. If z_1 is real and non-negative, the result is a real floating-point number. Otherwise, it is the float-complex number

$$log(abs(z_1)) + : phase(z_1)$$

$log10(z_1) \rightarrow z_2$
This procedure returns the base 10 logarithm of z_1, which must not be a zero of any kind. Defined as

$$log(z_1)/log(10)$$

$exp(z_1) \rightarrow z_2$
This procedure returns e raised to the power z_1, where e is the base of natural logarithms. The result is a floating-point number if the argument is real, or a float-complex otherwise.

$z_1 * *z_2 \rightarrow z_3$
This procedure returns z_1 raised to the power z_2, where either argument may be any numbers (except that z_1 must not be zero if z_2 is zero of any type other than integer 0). If z_2 is an integer, the computation is performed by successively multiplying powers of z_1; thus if z_1 is rational, the result will be exact. If z_2 is the integer 0, the result is always a 1 of the same type as z_1. Otherwise, if z_2 is not an integer, the result is computed as

$$exp(z_2 * log(z_1))$$

6.17 Trigonometric Functions

All the procedures in this section take an angle as argument, or return one as a result. In both cases, the units of the angle (radians or degrees) are controlled by the boolean variable *popradians*. This applies equally when the angle is complex.

$popradians$
This boolean variable specifies whether the angle arguments or results for trigonometric procedures are in radians ($true$) or degrees ($false$). Note that the default value is $false$, implying angles in degrees.

$phase(z_1) \to \theta$
This procedure returns the complex phase angle of z_1 as a floating-point quantity. This will be in the range

$$-pi < \theta \leq pi(radians)$$

$$-180 < \theta \leq 180(degrees)$$

If z_1 is real, then $\theta = 0.0$. This procedure is defined as $arctan2(destcomplex(z_1))$

$cis(\theta) \to z$
This procedure returns the float-complex number $z = cos(\theta) + : sin(\theta)$ The name cis stands for $cos + isin$. Note that this is the same as $exp(+ : \theta)$

$sin(z_1) \to z_2$
$cos(z_1) \to z_2$

$tan(z_1) \to z_2$

These procedures compute the sine, cosine and tangent of z_1. The result is a floating-point number, or a float-complex if z_1 is complex.

$arcsin(z) \to z_2$
$arccos(z) \to z_2$

$arctan(z) \to z_2$

These procedures compute the arcsine, arccosine and arctangent of z. For z complex, the result is a float-complex. For z real, it is a real float, except in the case of $arcsin$ and $arccos$ when $abs(z) > 1$. For $arctan$, it is an error if $z = +1$ or -1.

$arctan2(x, y) \rightarrow \theta$
Computes the arctangent of y/x, but using the signs of the two numbers to derive quadrant information. The result is a floating-point number in the range

$$-\pi < \theta \leq \pi \ \ (popradians = true)$$

$$-180 < \theta \leq 180 \ \ (popradians = false)$$

When $x = 0$ and $y = 0$ the result is defined (arbitrarily) to be 0.0.

$sinh(z_1) \rightarrow z_2$
$cosh(z_1) \rightarrow z_2$

$tanh(z_1) \rightarrow z_2$

These procedures compute the hyperbolic sine, hyperbolic cosine and hyperbolic tangent of z_1. The result is a floating-point number, or a float-complex if z_1 is complex.

$arcsinh(z_1) \rightarrow z_2$
$arccosh(z_1) \rightarrow z_2$

$arctanh(z_1) \rightarrow z_2$

These procedures compute the hyperbolic arcsine, hyperbolic arccosine and hyperbolic arctangent of z_1. For z_1 complex, the result is a float-complex. For z_1 real, the result will be a real float, except in the following cases:

$arccosh(z_1)$	$z_1 < 1$	Complex result
$arctanh(z_1)$	$abs(z_1) > 1$	Complex result
$arctanh(z_1)$	$z_1 = 1$ or $z_1 = -1$	*mishap* called

pi
This constant is the best *ddecimal* approximation to π.

6.18 Bitwise/Logical Integer Operations

These procedures enable integers to be manipulated as bit patterns representing two's-complement values, where bit position n has weight 2^n, i.e. bits are numbered from 0 upwards. Note that, conceptually, at any rate, the sign bit of an integer is extended indefinitely to the left. Thus everywhere above its most significant bit, a positive integer has 0 bits and a negative integer has 1 bits. The procedures can be used for integers of any size.

$i_1 \&\& i_2 \rightarrow i_3$
The result of this operation is the bitwise logical "and" of the integers i_1 and i_2, i.e. there is a 1 in the result for each bit position for which there is a 1 in both i_1 and i_2.

$i_1 \&\& \sim\sim i_2 \rightarrow i_3$
The result of this operation is the bitwise logical "and" of i_1 and the bitwise logical complement of i_2, i.e. there is a 1 in the result for each bit position for which there is a 1 in i_1 and a 0 in i_2. The operation is the same as $i_1 \&\& (\sim\sim i_2)$ and is useful for clearing those bits of i_1 which are set in i_2).

$i_1 || i_2 \rightarrow i_3$
The result of this operation is the bitwise logical "inclusive or" of i_1 and i_2, i.e. there is a 1 in the result for each bit position for which there is a 1 in either i_1 or i_2.

$i_1 ||/\& i_2 \rightarrow i_3$
The result of this operation is the bitwise logical "exclusive or" of i_1 and i_2, i.e. there is a 1 in the result for each bit position for which there is a 1 in either i_1 or i_2 but not in both.

$\sim\sim i_1 \rightarrow i_2$
This operator produces the bitwise logical complement of the integer i_1, i.e. there is a 1 in the result for each bit position for which i_1 has 0. It is always true that $\sim\sim i = -(i+1)$

$i_1 << n \rightarrow i_2$
This operator produces the bit pattern of i_1 shifted left by n positions; a negative value for n produces a right shift. n must be a simple integer.

$i_1 >> n \rightarrow i_2$
This operator returns the bit pattern of i_1 shifted right by (simple integer) n positions; a negative value for n implies a left shift.

i_1 &&/ = _0 $i_2 \rightarrow b$
i_1 && = _0 $i_2 \rightarrow b$

These two operators are equivalent to the boolean expressions

$$i_1 \&\& i_2 / == 0$$
$$i_1 \&\& i_2 == 0$$

but are more efficient since they avoid producing intermediate results.

$testbit(i_1, n) \rightarrow b$
$b \rightarrow testbit(i_1, n) \rightarrow i_{new}$

This procedure and its updater enable the testing and setting or clearing of the bit at position

n in the integer i_1. The base procedure returns the state of bit n as a boolean value, $true$ for 1 and $false$ for 0. The updater, which is somewhat unusual for an updater in that it returns a result, produces a new integer i_{new} which is i_1 with bit n set to 1 or cleared to 0 as specified by the input b argument, which may in fact be any item, $false$ meaning 0 and anything else meaning 1.

$integer\_leastbit(i_1) \rightarrow n$
This procedure returns the bit position n of the least-significant bit set in the integer i_1. Equivalently, n is the highest power of 2 by which i_1 divides exactly.

$integer\_length(i_1) \rightarrow n$
This procedure returns the length in bits of i_1 as a two's-complement integer. That is, n is the smallest integer such that

$$i_1 < (1 << n), \ \mathbf{if}\ i_1 \geq 0$$

$$i_1 \geq (-1 << n), \ \mathbf{if}\ i_1 < 0$$

Put another way: if i_1 is non-negative then the representation of i_1 as an unsigned integer requires a field of at least n bits; alternatively, a minimum of $n+1$ bits are required to represent i_1 as a signed integer, regardless of its sign.

$integer\_bitcount(i_1) \rightarrow n$
This procedure counts the number of 1 or 0 bits in the two's-complement representation of i_1. If i_1 is non-negative, n is the number of 1 bits; if i_1 is negative, it is the number of 0 bits. Note that, owing to the sign extension, there are an infinite number of 0 bits in a non-negative integer or 1 bits in a negative integer. It is always the case that

$$integer\_bitcount(i_1) = integer\_bitcount(-(i_1 + 1))$$

$integer\_field(n, i_1) \to P_{access}$

This procedure is used to create accessing/updating procedures for sub-bitfields within integers, and provides a more convenient (and more efficient) way of manipulating such fields than by masking and shifting with the operators && and >>, etc. Given a specification of a bitfield in terms of its width in bits n, and lowest bit position i_1 (both simple integers), it returns a procedure P_{access}. When this is applied to any integer it extracts the binary value represented by bits i_1 to $i_1 + n - 1$ within the integer, shifting it right by i_1 bits to align it at bit 0. That is,

$$P_{access}(i_1) \to i_{val}$$

If the argument $n > 0$, the field is taken to contain unsigned values only — i.e. values ≥ 0. If $n < 0$, then the field is signed, and upon extraction, the highest bit of the field is extended as the sign bit of the resulting value. C.f. the corresponding convention used by *conskey* described in Chapter 3.13. The updater of P_{access}, on the other hand, takes a field value and an integer, and returns a new integer in which the contents of the field bits are replaced by the given value, but which has the same bits everywhere else. That is:

$$i_{val} \to P_{access}(i_1) \to i_{new}$$

Note that the updater makes no check on the range of i_{val}; bits 0 to $n - 1$ of i_{val} are masked out, shifted up, and inserted into the field position and it is therefore irrelevant in this context whether the field is signed or unsigned. A further feature is that P_{access} and its updater can be made to merely mask out or mask in the field bits, without shifting the value down or up. That is, upon extraction the field value is left aligned at bit i_1 in the result; the input value for updating is assumed to be similarily aligned. This is achieved by supplying a second argument of $false$ to P_{access} or its updater, i.e.

$$P_{access}(i_1, false) \to i_{unshifted}$$
$$i_{unshifted} \to P_{access}(i_1, false) \to i_{new}$$

Note also that in this case, extraction of a signed field will *not* cause it to be sign-extended.

6.19 Random Numbers

POP provides two built in procedures, $random0$ and $random$, to generate pseudo random numbers. These can be used in any program that needs to choose numbers "at random", as it were by the throw of a die.

Random numbers are generated by these procedures by making use of the variable $ranseed$, which is used as the "seed" for the generation. Each random number generated uses one or more successive seed values, depending on the type of the argument to $random0$ and $random$. The algorithm used to generate each successive seed value is

```
ranseed fi_* 524269 fi_+ 32749 -> ranseed
```

which, since all current implementations use 30 bits for a simple integer, produces a result modulo 2^{30}. The This algorithm has a verified cycle length of 2^{30}, i.e. it will produce every possible combination of 30 bits. The fast arithmetic procedures fi_+ and fi_* are specified in Chapter refFASTPROCS.

$random0(n) \rightarrow n_{random}$
$random0(x) \rightarrow x_{random}$

Given a strictly-positive integer or floating-point argument, this procedure generates a random number of the same type, in the range

$$0 \leq n_{random} < n$$

$$0 \leq x_{random} < x$$

where the distribution will be approximately uniform. The value of the variable $ranseed$ is used as the seed for the generation process, and is replaced with a new seed value afterwards.

$random(n) \rightarrow n_{random}$
$random(x) \rightarrow x_{random}$

This procedure is the same as $random0$, except that whenever the latter would return 0 or 0.0, the original argument n or x is returned instead. Hence the range of the result is $1 \leq n_{random} \leq n$ for integer arguments, or $0 < x_{random} \leq x$ for floating point arguments.

$ranseed$
This variable is used to hold the next seed for generation of random numbers by $random0$ or $random$, both of which side-effect it. If set to $false$, it will be re-initialised to a simple integer the next time either procedure is called. The procedure sys_real_time is used for this initialisation, so you can expect a program that sets $false \rightarrow ranseed$ to behave differently on each occasion it is executed. Apart from this, the value of $ranseed$ must always be a simple integer.

6.20 Floating-Point Utilities

The procedures in this section provide the means to manipulate floating-point numbers, and make possible the writing of machine-independent floating-point software. Note that none of these procedures are affected by the value of $popdprecision$.

$float\_decode(d, i_{mantissa}) \rightarrow d_{sgn} \rightarrow i_{expo} \rightarrow n_{mantissa}$
$float\_decode(d, false) \rightarrow d_{sgn} \rightarrow i_{expo} \rightarrow d_{mantissa}$

This procedure takes a floating-point number and splits it into its component parts, i.e. sign, exponent and mantissa.

- d_{sgn} represents the sign, being a 1.0 or -1.0 of the same type as the argument, and with the same sign. Note that if $d = 0.0$ in the procedure call, $d_{sgn} = 1.0$
- Denoting the radix of the floating-point representation by i_b (see pop_float_radix below), i_{expo} is the integer power of i_b by which $d_{mantissa}$ must be multiplied to regain the magnitude of the original number.

- The mantissa is the absolute value of the number d with $i_b^{i_{expo}}$ divided out, and can be returned in one of two ways: If the $i_{mantissa}$ argument is $false$, then $d_{mantissa}$ is returned as a float of the same type, in the range

  $$1/i_b \leq d_{mantissa} < 1$$

  Otherwise, $n_{mantissa}$ is is returned as an integer, scaled by

  $$i_b^{float\_precision(d)}$$

  that is, so that the least significant digit in the representation of d is scaled to unity in the result. If $p = float\_precision(d)$ we then have

  $$i_b^{p-1} \leq n_{mantissa} < i_b^p$$

  Thus whether the mantissa is returned as a float or an integer, it will always be the case that

  $$n_{mantissa} i_b^{i_{expo}} = abs(d)$$
  $$d_{mantissa} i_b^{i_{expo}} = abs(d)$$

  Note that this holds also when $d = 0.0$, since in this case $i_{expo} = 0$, or $d_{mantissa} = 0.0$ depending on $i_{mantissa}$.

$float\_scale(d_1, i_{expo}) \rightarrow d_2$
This provides a more efficient way of scaling a float by a power of the floating-point radix i_b than by using *nonop* $**$ and avoids any intermediate overflow or underflow that could occur with the latter. It returns

$$d_1 * i_b^{i_{expo}}$$

as a floating-point of the same type. If the final result overflows or underflows, i.e. the absolute value of the exponent is too large for the representation, then $false$ is returned. This procedure can be used in conjunction with $float\_sign$ to put back together a floating-point number decomposed with $float\_decode$. That is, after

$$float\_decode(d, false) \to d_{sgn} \to i_{expo} \to n_{mantissa};$$

one can use

$$float\_sign(d_{sgn}, float\_scale(n_{mantissa}, i_{expo}))$$

to retrieve the original number.

$float\_sign(d_{sgn}, d_1) \to d_2$
This procedure returns a floating-point number d_2 of the same type and absolute value as d_1, but which has the sign of the float d_{sgn}. The argument d_1 may also be $false$. In this case, d_2 is returned as a 1.0 or -1.0 of the same type and sign as d_{sgn}.

$float\_digits(d) \to n$
This procedure returns, as an integer, the number of radix-i_b digits represented in the floating-point format of the argument. I.e. n has only two possible values, one for *decimals* and one for *ddecimals*. In all current POP implementations, $i_b = 2$ and n is around 22 for *decimals*, 53–56 for *ddecimals*. **??**

$float\_precision(d) \to n$
Same as $float\_digits$, except that the number of significant radix-b digits in the argument is returned. Since POP floating-point *decimals* and *ddecimals* are always normalised, this will in fact be identical to $float\_digits(d)$, with the single exception that $float\_precision(0.0) = 0$ for either float type.

6.21 Numeric constants

This section describes constants that define the ranges of numbers available in various representation classes.

float_parameters
This is a library; to use any of the constants it defines, you must load it explicitly with

```
lib float_parameters;
```

It defines the following parameter constants, all of whose values relate to the particular implementation of POPLOG in use:

pop_most_positive_decimal The greatest positive value representable in single-length decimal format.

pop_least_positive_decimal The smallest positive (non-zero) value representable in single-length decimal format.

pop_least_negative_decimal The least negative value (i.e. closest to zero) representable in single-length decimal format.

pop_most_negative_decimal The most negative value (i.e. closest to negative infinity) representable in decimal format.

pop_most_positive_ddecimal
pop_least_positive_ddecimal
pop_least_negative_ddecimal
pop_most_negative_ddecimal
Same as the above for double-length ddecimal format.

pop_plus_epsiloi_decimal

pop_plus_epsiloi_ddecimal
These are the smallest positive numbers in each format which when added to 1.0 of the same format produce a value not equal to 1.0. I.e. for each format, the smallest positive ϵ such that

$$number\_coerce(1, \epsilon) + \epsilon / = 1.0$$

is true. N.B. For *ddecimal* format, this depends on *popdprecision* not being $false$.

pop_minus_epsiloi_decimal
pop_minus_epsiloi_ddecimal

Same as before, but subtracting from 1.0 instead of adding, i.e. for each format the smallest positive e such that

$$number\_coerce(1, e) - e / = 1.0$$

is $true$.

$pop\_float\_parameters$
This is a full vector that holds all the floating-point constants given under -float_parameters- above, and which the latter uses to define each value as a separate constant. You are not advised to use this directly (since its format may change); always access the values via -float_parameters-.

pop_float_radix
This integer constant is the radix of the floating-point representation (= 2 in all current POPLOG implementations).

int_parameters
This is a library; to use any of the constants it defines, you must load it explicitly with

```
lib int_parameters;
```

It defines the following parameter constants, all of whose values relate to the particular implementation of POPLOG in use:

pop_max_int
The largest (i.e. most positive) integer that can be represented as a simple (short) *integer*.

pop_min_int
The smallest (i.e. most negative) integer value that can be represented as a simple (short) *integer*.

6.22 Number Keys

Each numeric data class has an associated constant which holds its key (see 3.13. These are as follows.

integer_key
The key for simple integers. *biginteger_key*

The key for big integers. *ratio_key*

The key for ratios. *decimal_key*

The key for short floating point numbers (*decimals*). *ddecimal_key*

The key for long floating point numbers (*ddecimals*) *complex_key*

The key for complex numbers.

6.23 Restrictions in AlphaPop

AlphaPop provides only integers, *decimals* and *ddecimals*.

Exercise - try defining a number representation in which the error is tracked.

Chapter 7

Lists and Pairs

NOTE
I do not understand the difference between islink and ispair.

Page 293 of the draft Pop book asks about this. Here is the answer

define global constant islink(x); lvars x; ispair(x) and not(null(x)) enddefine;

Whether it is of any use I don't know. Aaron

7.1 Introduction

In earlier chapters we have made use of lists of POP objects. The purpose of this chapter is to describe POP lists in detail, including how they are built up out of *pairs*, and how you can make infinite non-repeating lists. Subsequent chapters will describe the POP-11 matcher, which allows you to recognise lists which have a particular pattern very easily, and how to exploit the infinite lists [1] that POP provides.

A pair is a record which contains two arbitrary POP data objects, called the $front$ and the *back* of the pair: pair records may be used in their own right for any purposes.

The most frequent use of pairs, however, is to represent lists of objects; a list is represented in POP as a chain of pairs, where the $front$ of each pair in the chain is used to hold the next element of the list, and the *back* to hold the continuation of the chain, which may be either another pair or the special object [] at the end of the list. The object [] is called the *empty list* and is the value of the constant nil. It is the only list which is not built out of pairs.

The two parts of a pair, considered as a list, are known respectively as the *head* and the $tail$. Thus, whereas the head of a list can be any object, the tail must always be another list. Referring to our discussion in Chapter 3.7, lists form a data-type in POP, but is not a data-class. On the other hand, pairs form a data-class.

Mathematicians often deal with infinite sequences of entities. POP provides a useful representation of such sequences, called a *dynamic list*. A dynamic list is one where the final pair which holds an actual member of the list contains in its *back* not [] but yet another pair, this pair having in its *back* a procedure. Whenever an attempt is made to access the head or tail of this special pair the procedure is called with the expectation that it will produce as result either the next element of the list or the object $termin$ to indicate that there are no more objects left in the list. The result so produced is then added to the end of the list by putting it in the $front$ of the special pair and constructing another special pair to go in its $back$. Thus the old special pair becomes the last proper pair of the list. [2] A pair with a $back$ which is a a procedure is therefore a valid list, as yet unexpanded, and which will be expanded by the application of list procedures like hd, tl etc; the $front$ of such a pair will contain $true$ until the procedure in its *back* produces $termin$, at which time the $front$ becomes $false$. This indicates that the list is now $null$, and any application of hd, tl etc will result in $mishap$ being called.

You can create dynamic lists out of data-files. Such lists can be lists of characters, or

[1] These are also known as streams.[?]

[2] The idea for dynamic lists came originally from Peter Landin, who advocated what he called 'streams' to us (Rod Burstall and Robin Popplestone) when we were defining POP-2. (How about Algol60?) The first implementation of this idea was in fact in POP-2 [2]. The implementation described here, and that used in is essentially the same as that employed in POP-2. Other implementations are described in [?].

more commonly of objects (called *tokens* by compiler writers). In particular this provides a good way of representing input from a keyboard, which is infinite as far as the poor computer knows.

Thus a list is either

- The object [ ], called the empty list.
- A pair whose *back* is [ ], or another pair.
- A pair whose $front$ is a boolean, and whose *back* is a procedure, i.e. a dynamic list.

7.2 Predicates on Pairs and Lists

The following procedures are provided to allow you to recognise lists and pairs, and distinguish between different kinds of list.

$atom(O) \rightarrow b$
This procedure returns $true$ if O is not a pair, $false$ otherwise. $atom(O)$ is equivalent to $not(ispair(O))$.

$isdynamic(O) \rightarrow P$
$isdynamic(O) \rightarrow false$

This procedure returns the generator procedure P underlying a dynamic list, or $false$ if O is not a dynamic list.

$ispair(O) \rightarrow b$
This procedure returns $true$ if O is a pair, $false$ otherwise.

$islist(O) \rightarrow b$

This procedure returns $true$ if O is a list, $false$ otherwise.

$islink(O) \rightarrow b$

This procedure returns $true$ if O is a non-null pair.

$null(L) \rightarrow b$

This procedure returns $true$ if the list L is empty, $false$ otherwise. L is empty if it is either

- [ ]
- A pair with $front$ $false$ and $back$ a procedure, i.e a dynamic list whose procedure has returned $termin$. Note that applying $null$ to an unexpanded dynamic list pair causes it to be expanded.

$member(O, L) \rightarrow b$

The default value of this procedure variable is a procedure which returns $true$ if O is an element of the list L, otherwise $false$, equality being determined with the operator "=".

$lmember(O, L) \rightarrow L_{sub}$

If O is an element of the list L, this procedure returns the trailing portion of L starting with that element, otherwise $false$. Equality is determined by using the operator "==". E.g.

```
lmember(2, [1 5 4 6 2 3 7 9]) =>
** [2 3 7 9]
```

The main use of this procedure is to allow you to find an object in a list, and then modify the list destructively by assigning to the $front$ of the pair so found.

7.3 Pair Constructor and Access Procedures

The following procedures allow you to construct new pairs, and to access their components.

$back(Pair) \to O$
$O \to back(Pair)$

This procedure returns or updates the $back$ of the pair $Pair$.

$conspair(O_1, O_2) \to Pair$
This procedure constructs and returns a pair whose $front$ is O_1 and whose $back$ is O_2.

$destpair(Pair) \to O_{back} \to O_{front}$
This procedure returns two results, the $back$ and the $front$ of the pair $Pair$.

$front(Pair) \to O$
$O \to front(Pair)$

This procedure returns or updates the $front$ of the pair Pair.

$recursive\_front(Pair) \to O$
This procedure does, in effect $Pair \to O$, and then iterates, doing $front(O) \to O$, until O is *not* a pair. It is used by *syspr* in printing the names of procedures.

7.4 Constructing Lists

If we want to build lists, the following procedures, and a constant are provided to enable us to do so:

nil
The value of this constant is the unique object [ ], the empty list.

$conslist(O_1, O_2, \ldots, O_n, n) \rightarrow L$
This procedure returns a list L constructed from the n objects on the stack below the top, which must be a (simple) integer.

$cons(O, L_1) \rightarrow L_2$
$O :: L_1 \rightarrow L_2$

These both construct and return a list whose head is the object O and whose tail is the list L_1. The operator form is often notationally more convenient.

$pdtolist(P) \rightarrow L$
This procedure constructs and returns a dynamic list from the procedure P, i.e. a pair whose $front$ is $true$ and whose $back$ is the procedure P. P should produce exactly one object each time it is called, and $termin$ (the value of the constant $termin$) as the last object, if there is one. For example, we can construct a list of all the integers by

```
vars count = 0;

define nextint;
  count; count+1->count;
enddefine;

vars L_integer = pdtolist(nextint);
```

```
/* example
L_integer =>            ;;; The list is unexpanded, so prints thus
** [...]
L_integer.hd =>         ;;; The first member is 0, and this causes expansion
** 0
L_integer=>             ;;; so the list now prints as:
** [0 ...]
L_integer.tl.tl.tl.hd=>
** 3
L_integer=>
** [0 1 2 3 ...]
```

$expandlist(L_1) \rightarrow L_2$

If this procedure terminates, then $L_2 == L_1$. That is, either $L_1 == L_2 == []$ or they start with the identical pair. If L_1 is a dynamic list then *expandlist* will repeatedly apply the procedure which terminates the chain of pairs, thus *expanding* L_1. This results in L_1 becoming a non-dynamic, or *static* list. This procedure will loop forever if the list is of infinite length.

$sysconslist(popstackmark, O_1, O_2 \ldots O_n) \rightarrow L$

This procedure constructs a list, L, of all the elements on the user stack up to the unique object $<$ *popstackmark* $>$, an object which is the value of the constant *popstackmark*. [3] POP-11 list constructors use this, i.e.,

```
[% 1, 2, 3, 4 %]
```

is equivalent to

```
sysconslist(popstackmark, 1, 2, 3, 4)
```

[3]This construction was invented by R.M.Burstall, and incorporated in POP-1.

$allbutfirst(n, L) \to L_{sub}$

This procedure takes a (simple) integer n and a list L, and returns the sublist consisting of all its elements except the first n. L_{sub} will be an actual sublist of L — i.e. it will consist of a set of pair identical with some of those of L.

$allbutlast(n, L) \to L_{sub}$

This procedure takes a (simple) integer n and a list L, and returns the sublist consisting of all its elements except the last n. L_{sub} is a newly-constructed list, containing no pairs identical with those of L.

7.5 List Access Procedures

The following procedures are provided to access elements of a list.

$dest(L) \to L_{tl} \to O_{hd}$

This procedure returns two results, the tail and the head of the list L, which must not be *null*.

$hd(L) \to O_{hd}$
$O_{hd} \to hd(L)$

This procedure returns or updates the head of the list L, which must not be *null*.

$tl(L) \to L_{tl}$
$L_{tl} \to tl(L)$

This procedure returns or updates the tail of the list L, which must not be *null*.

The operations *CAR* and *CDR* are provided for LISPERS who find these names appealing (!). They are in fact derived from assembly code mnemonics of an IBM antique.

$car(L) \rightarrow O_{hd}$
$O_{hd} \rightarrow car(L)$

This procedure returns or updates the head of the static list L. $car([\,]) = [\,]$. The updater of *car* calls *mishap* if applied to a null static list.

$cdr(L) \rightarrow L_{tl}$
$L_{tl} \rightarrow cdr(L)$

This procedure returns or updates the tail of the static list L. $cdr([\,]) = [\,]$. The updater of *cdr* calls *mishap* is applied to a null static list.

$subscrl(n, L) \rightarrow O$
$O \rightarrow subscrl(n, L)$

This procedure returns or updates the n-th element of the list L (where the first element is has subscript 1). Because this procedure is the *class_apply* procedure of pairs, described in 3.13, this can also be used in the form

$$L(n) \rightarrow O$$

$$O \rightarrow L(n)$$

$last(L) \rightarrow O$
$O \rightarrow last(L)$

This procedure returns or updates the last element of the list L, which must not be null.

$lastpair(L) \rightarrow Pair$
$Pair \rightarrow lastpair(L)$

This procedure returns or updates the last Pair of the list L. L must not be null.

$destlist(L) \rightarrow n \rightarrow O_n \ldots \rightarrow O_2 \rightarrow O_1$
This procedure puts each element of L onto the stack, and returns the number of elements.

$dl(L) \rightarrow O_n \ldots \rightarrow O_2 \rightarrow O_1$
$O_1, O_2, \ldots, O_n \rightarrow dl(L)$

This procedure puts the n elements of the list L on the stack, e.g.

```
dl([1 2 3 4])
```

is equivalent to the POP command 1, 2, 3, 4;. The updater fills L with the top n objects from the stack.

$explode(L) \rightarrow O_n \ldots \rightarrow O_2 \rightarrow O_1$
$O_1, O_2, \ldots, O_n \rightarrow explode(L)$
Applied to a list, *explode* is the same as *dl*.

$oneof(L) \rightarrow O$
This procedure returns a randomly chosen element of L. It uses the POP random number generator *random*, described in Chapter **??**.

7.6 Other List Utilities

There are a number of procedures provided which perform useful operations on lists, such as applying a procedure to each member of a list, sorting it, deleting members of it.

$applist(L, P)$
$\rightarrow applist(L, P)$

his procedure applies the procedure P to each element of the list L. The updater applies the updater of P to each element of L backwards, i.e. $\rightarrow applist(L, P)$ is the same as

$$applist(rev(L), updater(P))$$

$copylist(L_1) \rightarrow L_2$
This procedure returns a copy of the list, L_1, i.e. L_2 is a list in which all the pairs are copies of those in L_1. *copylist* does not copy elements of L_1 which are themselves lists. The procedure *copytree* (q.v.) provides recursive copying.

$copytree(L_1) \rightarrow L_2$
This procedure makes a list, L_2, which is a copy of L_1. Any elements of L_1 which are themselves lists are recursively copied.

$delete(O, L_1, n) \rightarrow L_2$
$delete(O, L_1) \rightarrow L_2$

This procedure takes a list L_1 and produces a new list L_2 with the object O omitted from the original list. The optional argument n, if present, specifies that only n occurrences of O should be omitted.

$ncdelete(O, L, P_{eq}) \rightarrow L$
This procedure returns a list which is the given list with every object which is equivalent to the given O deleted. Items are defined as equivalent according the the equivalence test defined by procedure P_{eq}. The procedure re-uses the pairs which make up the original list, i.e. it is destructive.

$flatten(L_1) \rightarrow L_2$
This procedure converts a tree L_1 into a "flat" list L_2, which contains all the leaves of the tree, but has no sublists.

$length(L) \to n$
$length(O) \to n$

This procedure behaves identically to *listlength* on lists, but if its argument is not a list it then applies *datalength* to it (see Chapter **??**).

$listlength(L) \to n$
This procedure gives the number of elements n in the list L.

$maplist(L_1, P) \to L_2$
$L_2 \to maplist(L_1, P)$

This procedure applies the procedure P to each element of the list L_1, and returns a list of all results produced in so doing. This is equivalent to

$$[\%applist(L_1, P)\%]$$

The action of the updater is

$$dl(L_2) \to applist(L_1, P)$$

$ncmaplist(L, P) \to L$
This procedure applies procedure P to every element of its first argument, replacing that element with the result of calling P. *ncmaplist* is destructive in that it re-uses the pairs which make up its first argument.

$rev(L_1) \to L_2$
This procedure returns a new list which is the list L_1 with its members in reverse order.

$ncrev(L) \to L$
This procedure destructively reverses the order of the elements of L, i.e. it re-uses the pairs of its argument.

$setfrontlist(O, L_1) \to L_2$
This procedure returns L_2 formed by moving the O to the $front$ of L_1, or adding the O if not already present.

$shuffle(L_1) \to L_2$
This procedure returns L_2, a copy of L_1 with the elements randomly re-ordered. It uses $oneof$.

$sort(L_1) \to L_2$
This procedure returns a list of sorted objects. If the argument list contains numbers only, then the operation $<$ (less than) is used to decide where to put things in the sorted list. If it contains words or strings, then the procedure $alphabefore$ is used. If the list contains both words and numbers then a mishap will result.

$syssort(L, P) \to L$
$syssort(L, b, P) \to L$

The first argument is a list, the last argument is a procedure which takes two objects and returns a boolean result (e.g., $nonop\ <$ for numbers or $alphabefore$ for string and words, etc). The objects in the list are compared using the procedure and the result is a list with elements sorted in accordance with the procedure. If the optional boolean argument is $false$, then the sorting is destructive, i.e. the pairs which make up L are re-linked in the new order. $syssort$ uses a merge sort algorithm.

$flatlistify(L_1) \to L_2$
$flatlistify(\mathbf{v}) \to L_2$

Given a list L or a vector $\mathbf{v}$, made of lists and/or vectors embedded arbitrarily, $flatlistify$

will return a list, L_2, which contains all the words needed to create the list if given to *popval* (see **??**).

7.7 Constants

The following constants are provided:

pair_key
nil_key

These constants hold the key structures for pairs and the unique object [ ] (see Chapter 3.13).

Chapter 8

In which we find out what are words and what are not

NOTE check quotation. *"I think it is 'nasturtiums' said Piglet. "No", said Pooh, "'Masturshalums"'.*

8.1 Overview

A *word* in POP is an object that represents a particular sequence of characters (the characters of the word) as a 'meaningful' entity. POP maintains a *dictionary* of word records and, whenever a word record is to be constructed for a particular sequence of characters with *consword*, the dictionary is searched to see if it already contains a word made up of that sequence. If so, that is returned — otherwise a new record is constructed and entered in the dictionary.

Strings, discussed in Chapter 9, are simply vectors of characters, and should be used when it is not desired to associate any particular *semantic* information with a sequence of characters. For example, if you want to print out a message to a user of your program you will use a string. On the other hand, if you wanted to construct a *property* mapping from, say, English words to French words, you would probably construct a POP word for each word of the language, or at least each one referenced in a given session with the user.

Although they can be employed for other purposes, the principal use of words is as names of identifiers (i.e. variables, constants, etc) in a program. These uses are described in detail in Chapter 5. In particular, you can see how to use the $valof$ procedure to find the value of a variable currently associated with a word.

The POP dictionary of words serves as the starting point for garbage collection. The very right of a POP item to exist depends upon its being referenced directly or indirectly from a variable mentioned in the dictionary. Thus the garbage collector preserves all words which are the names of variables, and all objects that are values of these variables, and components of these objects, and components of components of these objects, etc.. Note however that the garbage collector **??** *removes* words from the dictionary which are not referenced elsewhere, and which are *not* the names of permanent variables.

In addition to the word-specific procedures, described in this chapter, some string manipulation procedures, described in 9, can also be used on words. The general procedures that operate on POP objects described in Chapter 3.7 also provide useful capabilities for operating on words. Thus, for example, words can be concatenated, using $<>$.

8.2 Predicates on Words

There are some special predicates which operate on words, as described below. In addition the predicates on strings described in **??**, also apply to words. In particular, $alphabefore$ defines a lexicographical ordering on words.

$isword(O) \to b$
This procedure returns $true$ if O is a word, $false$ if not.

8.3 Constructing words and accessing their characters

This section describes how you can construct words, out of characters, and how you can access the characters of existing words.

$consword(c_1, c_2, \ldots, c_n, n) \rightarrow W$
$consword(\mathbf{s}) \rightarrow W$

This procedure returns a word W. In the first form of call, the n characters on the user-stack below the integer n, which must be on the top of the stack, form the characters of W, with the top-most character being the last in the word.

In the second form of call, the characters of the string **s** form the characters of W. The word is taken from the dictionary if it is in there, or a new word constructed and entered in the dictionary if not.

$subword(i, n, W_1) \rightarrow W_2$
This procedure returns the word W_2 whose characters are the n characters of the word W_1 starting from its i-th character. W_1 may also be a string, but the result is still a word. If you want a string result, then see the procedure *substring* in Chapter **??**.

$identof(W) \rightarrow I$
$valof(W) \rightarrow O$

$O \rightarrow valof(W)$

These procedures, which return the identifier attached to a word and, if it is permanent, its value, are described in Chapter 5. Note that the identifier is not treated as a component of a word by the *destword* procedure:

$destword(W) \rightarrow n \rightarrow c_n \rightarrow \ldots \rightarrow c_2 \rightarrow c_1$
This procedure puts all the characters of the word W on the stack, together with its length

n. In other words, it is the inverse of $consword(n)$. E.g.

```
destword("abcd") =>
** 97 98 99 100 4
```

$subscrw(n, W) \rightarrow c$
This procedure returns the n-th character c of the word W. It does *not* have an updater. Since $subscrw$ is the *class_apply* of words (see 3.13), this can also be called as $W(n) \rightarrow c$.

POP, following LISP, provides a procedure to allow a program to create new words. This is mostly useful in symbolic mathematics, where for example, you may wish to create new symbols during a Skolemisation procedure **??**, or in applying a general law to a particular situation, perhaps in changing the variable of integration.

$gensym(W_{root}) \rightarrow W$
$n \rightarrow gensym(W_{root})$
This procedure takes a 'root' word and returns a new word with an integer suffix appended. Each call of $gensym$ on the same word will increment the integer suffix. The updater of $gensym$ can be used to reset the counter for a particular word. You can reset the counter for all root words that $gensym$ has used by using the procedure $cleargensymproperty$.

$appgensymproperty(P)$
This procedure applies the procedure P to each active $gensym$ root word and its current integer counter, i.e. it evaluates $P(W_{root}, N)$.

$cleargensymproperty()$
This procedure resets the integer counter to 1 for all active $gensym$ root words.

$sysnvariable() \rightarrow W$
This procedure generates a unique, but unprintable, word. It is used to create new permanent identifiers for macros and syntax procedures.

8.4 Dictionary Procedures

To allow you to interrogate the dictionary of words, to find what words are in it, or whether words with certain substrings are in it, etc., the following procedures are provided.

$appdic(P)$
This procedure applies the procedure P to each word currently in the dictionary.

$mapdic(P) \rightarrow L$
This procedure applies procedure P to each word in the dictionary, and returns a list of any results produced.

$dic\_distrib()$
This procedure supplies information on the structure of the dictionary. The dictionary has 1024 slots - if a slot is empty a period character, and otherwise an integer representing the number of words resident in the slot, is printed.

$countwords() \rightarrow n$
This procedure returns the number of words in the dictionary.

$wordswith(O) \rightarrow L$
Given a word or a string as argument, this procedure returns an alphabetically sorted list of words from the system dictionary which contain the argument as a substring or subword.

$syscancelword(W)$
This procedure *cancels* the word W, i.e. removes it from the dictionary.

8.5 Constants associated with words

undef
This constant contains the word "*undef*". It should contain an undef record (see 5.3), but for historical reasons does not.

word_key
This constant holds the key structure for words (see 3.13).

Chapter 9

In which we string together characters

9.1 Overview

A POP-11 *string* is a *vector* of *characters*, where a character is an integer in the range 0–255, thus comprising one *byte* of information. Being a POP vector, as described in Chapter 3.7, a string is accessed by subscripting with an integer subscript, which ranges from 1 upwards to the length of the string.

String creation and manipulation procedures available in POP are listed below; note that some string procedures are also applicable to *words*. A string is a particular built-in instance of the general class of vectors which can be constructed using *conskey* or *vectorclass*. Those sections of Chapter 3.7 which treat vectors provide the information you will need.

9.2 Predicates on Characters

The following predicates allow us to distinguish between different types of character:

$isuppercode(O) \rightarrow b$
This procedure returns $true$ if O is the ASCII character code for an upper case letter, i.e. an integer in the range 65–90, or $false$ otherwise.

$islowercode(O) \rightarrow b$
This procedure returns $true$ if O is the ASCII character code for a lower case letter, i.e. an integer in the range 97–122, or $false$ otherwise.

$isalphacode(O) \rightarrow b$
This procedure returns $true$ if O is the ASCII character code for a alphabetic character, i.e. an integer in the ranges 65–90, 97–122, or $false$ otherwise.

$isnumbercode(O) \rightarrow b$
This procedure returns $true$ if O is the ASCII character code for a digit, i.e. an integer in the range 48–57, or $false$ otherwise.

9.3 Locating Characters in Strings

The following procedures allow us to search a string for the occurrence of a given character.

$locchar(c, n, \mathbf{s}) \rightarrow m$
$locchar(c, n, W) \rightarrow m$

This procedure searches the string **s** for the character c, starting the search at the n-th character of **s**. It returns the subscript m at which c was found if it was, or $false$ otherwise. If a word argument W is given the action is the same.

$locchar\_back(c, n, \mathbf{s}) \rightarrow m$
$locchar\_back(c, n, W) \rightarrow m$

This procedure is the same as *locchar*, except that the search is performed *backwards* starting from the n-th character.

$skipchar(c, n, \mathbf{s}) \rightarrow m$
$skipchar(c, n, W) \rightarrow m$

This procedure searches the string **s** for any character *other* than c, starting at the n-th character. It returns the subscript m at which such a character was found or $false$ if every character from the n-th onwards was a c. If a word argument W is given, the action is the same.

$skipchar\_back(c, n, \mathbf{s}) \rightarrow m$
$skipchar\_back(c, n, W) \rightarrow m$

This procedure is the same as *skipchar*, except that the search is performed *backwards* starting from the n-th character.

$strmember(c, \mathbf{s}) \rightarrow m$
$strmember(c, W) \rightarrow m$

This is equivalent to $locchar(c, 1, \mathbf{s})$. It returns the subscript m at which c first occurs, or $false$.

9.4 Predicates on Strings

Note that all the procedures in this section, except *isstring*, will accept words in place of any of their string arguments.

$isstring(O) \rightarrow b$
Returns *true* if O is a string, $false$ if not.

$issubstring(\mathbf{s}_{sub}, n, \mathbf{s}) \rightarrow m$
$issubstring(\mathbf{s}_{sub}, n, W) \rightarrow m$

This procedure searches the string $\mathbf{s}$, starting from its n-th character, for a substring equal to the string $\mathbf{s}_{sub}$ and, if found, returns the subscript m of $\mathbf{s}$ at which the matching substring begins; otherwise it returns $false$. The action is the same for a word W

$issubstring\_lim(\mathbf{s}_{sub}, n, i_{start}, i_{end}, \mathbf{s}) \rightarrow m$
$issubstring\_lim(W_{sub}, n, i_{start}, i_{end}, \mathbf{s}) \rightarrow m$

$issubstring\_lim(\mathbf{s}_{sub}, n, i_{start}, i_{end}, W) \rightarrow m$

$issubstring\_lim(W_{sub}, n, i_{start}, i_{end}, W) \rightarrow m$

This procedure is the same $issubstring$, but the match is constrained to start on or before the subscript i_{start}, and to end on or before the subscript i_{end}. The i_{start} or i_{end} constraints may be disabled by supplying $false$ for either argument, e.g.

$$issubstring\_lim(\mathbf{s}_{sub}, n, false, false, \mathbf{s})$$

is just the same as $issubstring$. The action is the same if either or both arguments are words.

$isstartstring(\mathbf{s}_{sub}, \mathbf{s}) \rightarrow m$
$isstartstring(W_{sub}, \mathbf{s}) \rightarrow m$

$isstartstring(\mathbf{s}_{sub}, W) \rightarrow m$

$isstartstring(W_{sub}, W) \rightarrow m$

If the string $\mathbf{s}$ starts with the substring $\mathbf{s}_{sub}$ then the procedure returns subscript 1, otherwise $false$. The action is the same if either or both arguments are words.

$isendstring(\mathbf{s}_{sub}, \mathbf{s}) \rightarrow m$
$isendstring(W_{sub}, \mathbf{s}) \rightarrow m$

$isendstring(\mathbf{s}_{sub}, W) \rightarrow m$

$isendstring(W_{sub}, W) \rightarrow m$

If the string $\mathbf{s}$ ends with the substring $\mathbf{s}_{sub}$, then this procedure returns the subscript m of $\mathbf{s}_{sub}$ in $\mathbf{s}$, otherwise $false$. The action is the same if either or both arguments are words.

$hassubstring(\mathbf{s}, \mathbf{s}_{sub}) \rightarrow m$
$hassubstring(W, \mathbf{s}_{sub}) \rightarrow m$

$hassubstring(\mathbf{s}, W_{sub}) \rightarrow m$

$hassubstring(W, W_{sub}) \rightarrow m$

The evaluation of $hassubstring(O, O_{sub})$ is equivalent to that of $issubstring(O_{sub}, 1, O)$,

$hasendstring(\mathbf{s}, \mathbf{s}_{sub}) \rightarrow m$
$hassubstring(W, \mathbf{s}_{sub}) \rightarrow m$

$hassubstring(\mathbf{s}, W_{sub}) \rightarrow m$

$hassubstring(W, W_{sub}) \rightarrow m$

The evaluation of $hasendstring(O, O_{sub})$ is equivalent to that of $isendstring(O_{sub}, O)$.

$hasstartstring(\mathbf{s}, \mathbf{s}_{sub}) \rightarrow m$
$hasstartstring(W, \mathbf{s}_{sub}) \rightarrow m$

$hasstartstring(\mathbf{s}, W_{sub}) \rightarrow m$

$hasstartstring(W, W_{sub}) \rightarrow m$

The evaluation of $hasstartstring(O, O_{sub})$ is equivalent to that of $isstartstring(O_{sub}, O)$.

$issubitem(\mathbf{s}_{sub}, n, \mathbf{s}) \rightarrow m$
$issubitem(W_{sub}, n, \mathbf{s}) \rightarrow m$

$issubitem(\mathbf{s}_{sub}, n, W) \rightarrow m$

$issubitem(W_{sub}, n, W) \rightarrow m$

This procedure is exactly like *issubstring* (see above) except that the match only succeeds if the matching substring is a distinct item in the string **s**. 'Distinct item' means roughly that the substring $\mathbf{s}_{sub}$ is not embedded in other characters in **s**; for example

```
issubitem('DEFINE', 1, 'ENDDEFINE')
```

returns *false*. It uses the rules of the POPLOG editor VED for deciding item beginnings and ends, not those of the POP-11 itemiser. The action is the same if either or both arguments are words.

$alphabefore(\mathbf{s}_1, \mathbf{s}_2) \rightarrow b$
$alphabefore(W_1, \mathbf{s}_2) \rightarrow b$

$alphabefore(\mathbf{s}_1, W_2) \rightarrow b$

$alphabefore(W_1, W_2) \rightarrow b$

This procedure returns *true* if the first argument is alphabetically before the second, or *false* if the first is alphabetically after the second; 1 is returned if character sequences of the objects are equal. This is also known as *lexicographical ordering*.

9.5 Constructing Strings

$consstring(c_1, c_2, \ldots, c_n, n) \rightarrow \mathbf{s}$
This procedure returns a string **s** constructed from the n objects on the stack below n, which must be *characters*. The topmost character on the stack is the last in the string, etc. So $\mathbf{s}_i = c_i$.

$inits(n) \rightarrow \mathbf{s}$
This procedure returns a newly created string $\mathbf{s}$ of length n containing all zero characters. In ASCII terms, these are known as NULL characters. If you want to initialise with any other object, see *initvectorclass* in 3.7.

$substring(i_{start}, n, \mathbf{s}) \rightarrow \mathbf{s}_{sub}$
$\mathbf{s}_{sub} \rightarrow substring(i_{start}, n, \mathbf{s})$
This procedure returns a string $\mathbf{s}_{sub}$ consisting of the n characters of the string $\mathbf{s}$ starting from the character at subscript i_{start}. $\mathbf{s}$ may also be a word, but the result is still a string. If you want a *word* result, see Chapter 8.3.

The updater copies the first n characters of the string $\mathbf{s}_{sub}$ into the string $\mathbf{s}$ starting at subscript i_{start}. In this case $\mathbf{s}_{sub}$ may also be a word, but not $\mathbf{s}$.

$lowertoupper(O_1) \rightarrow O_2$
If O_1 is a string, word or character then the procedure returns a new item of the same type with any ASCII code for lowercase characters converted to their uppercase equivalent. Otherwise the procedure just returns O_1.

$uppertolower(O_1) \rightarrow O_2$
If O_1 is a string, word or character then the procedure returns a new item of the same type with any ASCII code for uppercase characters converted to their lowercase equivalent. Otherwise the procedure just returns O_1.

9.6 Accessing String Characters

$deststring(\mathbf{s}) \rightarrow n \rightarrow c_n \ldots \rightarrow c_2 \rightarrow c_1$

This procedure puts all the characters of the string $\mathbf{s}$ on the stack, together with its length. In other words, it is the inverse of *consstring*. E.g.

```
deststring('abcd') =>
** 97 98 99 100 4
```

$subscrs(n, \mathbf{s}) \rightarrow c$
$c \rightarrow subscrs(\mathrm{n}, \mathbf{s})$
This procedure returns or updates the n-th character c of the string **s**. Since *subscrs* is also the *class_apply* of a string (see 3.13), this may also be called as:

$$\mathbf{s}(n) \rightarrow c$$
$$c \rightarrow \mathbf{s}(n)$$

9.7 Generic Datastructure/Vector Procedures on Strings

The generic datastructure procedures described in Chapter 3.7 (*datalength*, *appdata*, *explode*, *fill*, *copy*, etc) are all applicable to strings, as are the generic vector procedures (*initvectorclass*, *move_subvector*, *sysanyvecons*, etc) also described in that chapter.

9.8 Other procedures and constants

$strnumber(\mathbf{s}) \rightarrow n$
If the characters of the string **s** form a valid number according to the lexical syntax rules given in 19 then that number is returned, otherwise *false*. E.g.

```
strnumber('123') =>
** 123
```

stringin($\mathbf{s}$) $\rightarrow P_{c\_rep}$

This procedure returns a character repeater for the string **s**, i.e. a procedure which each time it is called produces the next character from the string, and *termin* when the string is exhausted (see also 20).

string_key

This constant holds the key structure for strings (see Chapter 3.13)

Chapter 10

Arrays in POP

10.1 Overview

Arrays in POP are procedures which permit access to data associated with them. An array is applied to integer arguments, also referred to as *indices* or *subscripts*. A given array is always applied to a fixed number of arguments, commonly referred to as the *dimension* of the array. Thus an n-dimensional array requires n integers as indices to access or update its components. Two dimensional arrays are often used to represent matrices, and also to represent visual images for work in Computer Vision.

Because computer memory cannot directly represent multiple dimensions, the elements of an array actually have to be stored in an underlying 1-dimensional structure (e.g. a vector), each position in which corresponds to a particular sequence of n subscript values. The task of the array procedure is therefore to convert a given set of n-dimensional subscripts into the appropriate 1-dimensional subscript for accessing this structure.

POP arrays are constructed by the procedure *newanyarray*, described in section 10.4. A simpler version of this procedure is *newarray*, described in section 10.4 Such a procedure for an n-dimensional array takes n arguments, and includes within it the *arrayvector*, that is, the underlying 1-dimensional structure in which the elements of the array are in reality stored. Despite its name this need not actually be a vector. When called, e.g.

$$A(i_1, i_2, \ldots, i_n)$$

the array procedure performs the computation of the 1-dimensional subscript from the n values given, and supplies this to the appropriate access procedure for the *arrayvector* structure. This access procedure then returns or updates the specified element.

An array may have any number of elements in a given dimension, subscripted by any suitable range of integers. That is, an array with say, 30 in a particular dimension is not restricted to using subscripts 1 to 30, but can use 0 to 29, 5 to 34, −10 to 19, etc. An array's dimensions are specified to *newanyarray* by supplying a list called its *boundslist*, which is a list of the minimum and maximum subscripts in each dimension. It is of length $2n$ for an n-dimensional array. When called, array procedures call *mishap* if any of their subscript arguments is not in the appropriate range.

When using an array in general, it is not necessary to know how its elements are arranged in the *arrayvector*; however, this does need to be known when the latter needs to be processed separately from the array. While in principle many different schemes are possible, POP (like most systems) provides just two: storing 'by row' or 'by column'. Since the terms 'row' and 'column' only make sense for 2-dimensional arrays, we shall not attempt to define them, but simply say that 'by row' means the elements are stored with subscripts increasing in significance from left to right, and 'by column' with subscripts increasing in significance from right to left. That is, letting A be a 3-dimensional array with subscripts 1 – 3 in each dimension, the order of the $3 * 3 * 3 = 27$ elements in its *arrayvector* with the two different schemes would be as follows:

```
arrayvector
 subscript        by row        by column

    1           A(1, 1, 1)      A(1, 1, 1)
    2           A(2, 1, 1)      A(1, 1, 2)
    3           A(3, 1, 1)      A(1, 1, 3)
    4           A(1, 2, 1)      A(1, 2, 1)
    5           A(2, 2, 1)      A(1, 2, 2)
    6           A(3, 2, 1)      A(1, 2, 3)
    7           A(1, 3, 1)      A(1, 3, 1)
    8           A(2, 3, 1)      A(1, 3, 2)
```

```
9           A(3, 3, 1)      A(1, 3, 3)
10          A(1, 1, 2)      A(2, 1, 1)
...           ...              ...
26          A(2, 3, 3)      A(3, 3, 2)
27          A(3, 3, 3)      A(3, 3, 3)
```

Note that, as mentioned above, the *arrayvector* of an array is not limited to being an actual vector-class structure. The arguments to *newanyarray* allow the specification of any 'virtual' vector, in terms of an 'vector init' procedure and a 'subscriptor' procedure; *newanyarray* then calls the 'init' procedure with an appropriate length argument to construct the *arrayvector* initially, and the array procedure then uses the 'subscriptor' to store and retrieve elements.

Note also that two or more arrays can be made to share the same *arrayvector* structure, either in whole or in part. For example, one array can be a sub-array of another.

While arrays are limited to storing associations between sets of integers and arbitrary values, POP also provides mechanisms for storing associations between arbitrary objects — see Chapter **??**. See also Chapter 2 for procedures applicable to arrays as procedures in general.

10.2 Predicates On Arrays

As mentioned above, arrays are procedures: thus $isprocedure(A) = true$ for any array A.

$isarray(A) \rightarrow A$
$isarray(O) \rightarrow false$
This procedure returns a non-$false$ result if O is either an array itself, or is a closure of one through any number of sub-closures, i.e. $pdpart(O)$ is examined recursively until a non-closure is found, and then this is tested for being an array. See Chapter **??** for a definition of closures. If O is an array procedure, or one is found inside a nest of closures, then that is returned, otherwise $false$. This enables closures of arrays to be recognised as arrays, e.g.

if A is 3-dimensional array then $isarray$ will return A when applied to the 2-dimensional array $A(\%2\%)$.

$isarray\_by\_row(A) \rightarrow b$
This procedure returns $true$ if the array A is stored by row, $false$ if stored by column.

10.3 Obtaining Array Parameters

$boundslist(A) \rightarrow L_{bounds}$
Given an array A, this procedure returns its list of minimum and maximum subscripts in each dimension.

$arrayvector(A) \rightarrow \mathbf{v}_A$
This procedure returns the 1-dimensional vector used to hold the elements of the array A.

$arrayvector\_bounds(A) \rightarrow i_{min} \rightarrow i_{max}$
This procedure returns the minimum and maximum subscripts used by the array A on $arrayvector(A)$. Generally, i_{min} will be 1 and i_{max} will be the number of elements in the array. However, $newanyarray$ can be used to create arrays which are mapped into subvectors of their $arrayvector$.

10.4 Constructing New Arrays

$newanyarray$ is the basic procedure for constructing arrays. For historical and other reasons, the arguments to this procedure are somewhat complicated, in that it can take a number of optional arguments in various combinations. Essentially though, the pieces of information it requires to construct a new array are quite straightforward, viz

1. A list of $2n$ minimum and maximum subscripts for each of the n dimensions (the *boundslist*). This is used to derive both the number of dimensions, and the size in each dimension, and hence the total number of elements in the array.

2. A specification for the *arrayvector* to hold the elements of the array, and a subscriptor procedure with which to access and update it. These two can be specified in a number of different ways, either as separate arguments or by a single argument which implies values for both together.

Other optional pieces of information can specify whether the elements are to be stored by row or by column, an initialisation for each array element, and (when the new array is to be a sub-array of an existing one), the starting offset of the new array within the existing *arrayvector*. Thus:

$newanyarray(L_{bounds}, O_{init}, O_{spec}, P_{subscr}, i_{offset}, b_{by\_row}) \rightarrow A$
constructs and returns a new n-dimensional array procedure A, where $n \geq 1$. Its arguments are as follows:

1. L_{bounds} A list of $2n$ integers whose elements are alternately the minimum and maximum subscript in each dimension, i.e.

$$[\%i_1, j_1, i_2, j_2, \ldots, i_n, j_n\%]$$

This argument is always required.

O_{init} This argument is optional, and specifies an initialisation for each array element: its interpretation depends on whether it is a procedure or not. If a procedure, it is assumed to take n subscript arguments and to return a (possibly different) initialising value for each position in the array. *newanyarray* will then initialise the array by applying the procedure in turn to every combinations of subscripts, i.e.

$$O_{init}(i_1, i_2, \ldots, i_n) \rightarrow A(i_1, i_2, \ldots, i_n)$$

Otherwise, if O_{init} is not a procedure, every element of the array is simply initialised to that value. However, to distinguish it from L_{bounds}, this argument must not be list — if you wish to initialise the elements to some list, you must use a procedure returning the list, as above.

O_{spec} This argument is always required, and specifies the *arrayvector* structure to be used to store the elements of the array. It must supply either an existing structure, or a procedure to construct a new one, that is to say:

(a) *either* an actual vector-class structure, e.g. a full- vector, string, etc;

(b) *or* an array procedure whose *arrayvector* is to be used;

(c) *or* a 'vector init' procedure P. This will be called as

$$P(n) \rightarrow \mathbf{v}_{array}$$

where n is the number of elements in the array;

(d) *or* a vector-class key whose *class_init* procedure is to be used, as described in Chapter3.13.

All cases except (c) also implicitly supply a subscriptor procedure for the structure, making the next argument (P_{subscr}) optional; for case (c), P_{subscr} must be present. If an existing structure is specified with (a) or (b), it must of course be large enough to accomodate the new array elements. Note that if O_{init} is omitted, the initial values of the array elements will depend upon the O_{spec} argument. I.e. for cases (c) and (d) in which a new structure is created they will be whatever the constructor procedure initialises them to. For an existing structure they will have their current values.

2. P_{subscr} A subscriptor procedure for accessing and updating elements of the *arrayvector*, i.e. a procedure with updater of the form

$$P_{subscr}(i, \mathbf{v}_{array}) \rightarrow O$$
$$O \rightarrow P_{subscr}(i, \mathbf{v}_{array})$$

This argument may always be supplied, but is essential only when O_{spec} specifies a 'vector init' procedure; in all other cases, i.e. an existing array, an existing vector-class structure or vector-class key, the subscriptor procedure derived from that is used if P_{subscr} is omitted.

3. i_{offset} This is an optional integer argument specifying the starting offset of the new array's elements within an existing array vector got from O_{spec}. This argument is *illegal* in all cases where a new *arrayvector* has to be constructed. Note that this is an offset, not a subscript, i.e. 0 means start at the first element (the default), 1 at the second, etc. The existing structure must be large enough accomodate all the array elements starting at the given offset.

4. b_{by_row} This is an optional argument specifying whether the array elements are to be arrayed by row or by column: if supplied, it *must* be a boolean, $true$ meaning by row or $false$ meaning by column. If omitted, the value of $poparray\_by\_row$ is used instead.

$poparray\_by\_row$
This boolean variable controls the order in which the elements of an array produced by $newanyarray$ are stored in its underlying vector, and supplies the default value for the b_{by_row} argument (q.v.).

$newarray(L_{bounds}, O_{init}) \rightarrow A_{full}$
This procedure provides a simpler interface to $newanyarray$ for constructing arrays of full items, and is just $newanyarray(\%vector\_key\%)$ I.e. it uses standard full vectors to store the array elements (see **??**).

$newsarray(L_{bounds}, O_{init}) \rightarrow A_{char}$
This procedure is the same as $newarray$, but uses strings (see Chapter 9) rather than full vectors, i.e.

$$newsarray = newanyarray(\%string\_key\%)$$

10.5 Generic Datastructure Procedures on Arrays

The generic datastructure procedures described in 3.7 ($datalength$, $appdata$, $explode$, $fill$, and others defined in terms of those) can all be applied to arrays: they treat an array as the set of its $arrayvector$ elements between the minimum and maximum subscripts given by the $arrayvector\_bounds$. Thus, for example, if

$$arrayvector\_bounds(A) \rightarrow i_{min} \rightarrow i_{max}$$

then $datalength(A)$ will be

$$i_{max} - i_{min} + 1$$

Similarily, $appdata(A, P)$ will apply the procedure P to the value of each *arrayvector* element from i_{min} to i_{max} inclusive, and so on.

The procedure *copy* when applied to an array copies both the array procedure and its *arrayvector*, so that the copy is completely independent of the original.

$arrayscan(L_{bounds}, P)$
Given a L_{bounds} of array minimum and maximum subscripts, as supplied to *newanyarray*, etc, this procedure applies the procedure P every *list* of subsripts in the range defined by L_{bounds}. These lists are ordered 'by column', i.e. with the last subscript varying fastest.

```
arrayscan([1 4 1 3],pr);
[1 1][1 2][1 3][2 1][2 2][2 3][3 1][3 2][3 3][4 1][4 2][4 3]
```

10.6 Examples of Arrays

The following statement:

```
        vars procedure
            multab = newarray([1 12 1 12], nonop *);
```

will create and assign to *multab* a 12 x 12 2-dimensional array each of whose elements can be a full POP item, and where each element subscripted by i, j is initialised to $i * j$. In other words, *multab* is a multiplication table:

```
m(4, 3) =>
** 12
```

To turn an existing string of "noughts-and-crosses" (tic-tac-toe) into an array whose entries are the characters O, X and *space*:

```
vars procedure
    oxo = newanyarray([1 3 1 3], '0 X X00 X');
```

We can

```
cucharout(oxo(1, 3));
0
```

If, in the previous example, we make the array by row instead of by column, the mapping between subscripts and positions in the string is reversed:

```
vars procedure
    oxo2 = newanyarray([1 3 1 3], '0 X X00 X', false);

cucharout(oxo2(1, 3));
X
```

10.7 Sparse arrays

Where most of the values of an array are identical. the array is said to be *sparse*. Typically the identical elements will be 0. Big sparse arrays represent a considerable wastage of store if they are implemented with newanyarray — at least if vectors are used to hold the elements. Sparse arrays can be implemented as *properties*, and this is described in Chapter **??**.

10.8 History of POP arrays

Arrays in POP-11 are in essence very similar to those defined in the POP-2 language. They owe their form to the procedural (or functional) approach to data-objects advocated by pioneers such as McCarthy, Landin, Strachey. The main improvements made by POP-11 have been to make the array vector explicitly available, to allow the user to control the storage by row or by column. Sparse arrays, derived from properties, are new in POP-11.

Chapter 11

Properties and Memo Functions

NOTES If you supply a false hash procedure to newanyproperty, I assume that only short integers form their own key.

In this chapter we discuss techniques available in POP to allow you to create procedures which use stored associations between one set of objects and another. For example a geography program might want to associate capital cities with nations, a natural language translator might want to associate words in French with words in English, and a compiler writer might want to map from identifiers to the type of the identifier. In none of these cases is it possible to predict by an algorithm what the association will be in the way that it is possible to use an algorithm to compute the *sin* of a number. There may, of course be some regularities in the association, as there are for example between English and French, where a native speaker in one language can usually know what technical terms in the other mean. And indeed, in natural language processing it is common to have a dictionary supplemented by algorithms which support the morphology of the language — that 'know' how plurals and tenses are formed from noun and verb roots. This chapter treats the implementation of 'rote' knowledge, and tells you what 'hooks' are provided to hang procedural 'rules' upon this 'rote'.

In treating these associations as procedures, we shall be dealing with the case in which the direction of the mapping is known (e.g. English to French). For a discussion of associations in which the direction is not known in advance, see the Chapter **??** on Relational Databases. Likewise, in Prolog, [**?**], many predicates can be regarded as imposing no direction. Underneath the apparent lack of directionality in these systems, there may well be a directionality imposed perhaps by how the data is stored. Some measure of *indexing* is common with Prolog implementations, as it is with Relational Databases, and the implementation of these systems may allow you to exploit this, often at the cost of restricting the choice of how you say things.

However, just as it is worth organising traditional foreign language dictionaries so that they support mapping in one particular direction, where there is a data-set of any size it will often be essential to provide a uni-directional mapping from one attribute of the members of the set to another. It may of course be desirable to provide more than one such mapping, depending on the patterns of use of the set.

Some such POP procedures, for example *assoc*, provide easy-to-use capabilities which are quite space-efficient, but provide poor time efficiency — $O(n)$ in the number of entries. Others, such as *newanyproperty* are more complicated to use but provide good time-efficiency by making use of the hashing procedures described in Chapter **??**.

11.1 Simple associations

$assoc(L) \rightarrow P_{assoc}$

This procedure creates a procedure P_{assoc} encoding an association table. The argument list supplies initial associations in the form of a list of two element lists. P_{assoc} when given an object will return its associated value, or $false$ if there isn't one, i.e.

$$P_{assoc}(O) \rightarrow O_{val}$$

For example.

```
vars capital = assoc([
[France Paris]
[USA Washington]
[UK London]                ]);
```

creates a procedure, *capital*, which we can use thus:

```
capital("France") =>
** Paris
```

The procedure *capital* has an updater, so we can extend the association:

```
"Vienna" -> capital("Austria");
capital("Austria")=>
** Vienna
```

For anything but very tiny association tables, you should use properties rather than this procedure, as described in Section 11.3.

appassoc(P_{assoc}, P)
This procedure applies procedure P to each entry in the association, P_{assoc}, which must have been created by *assoc*. P_{assoc} is applied as

$$P_{assoc}(O_{arg}, O_{val})$$

for each O_{arg} in the table.

The procedure *newassoc* is described in section 11.3.2. It provides a simple interface to the fast hash-coded properties available in POP.

11.2 Writing association procedures

The *assoc* procedure is built in to POP, but it is instructive to consider how we could implement such an association function, since *assoc* might not exactly serve our purpose. The first step is to construct a procedure to look for an object in a list:-

```
define lookfor(O,L);     ;;; Find an object in a list L.
  lvars O p L;
    for p in L do         ;;; Iterate through pairs p in  L
      if p.hd = O then    ;;; until we find one whose head is  O
      return(p.tl.hd)     ;;; return the associated value.
      endif
    endfor;
  return(false)           ;;; If no pair has O as head,
enddefine;                ;;; return false.
```

We can test this

```
vars Capitals = [[France Paris] [USA Washington]];
lookfor("France", Capitals) =>
** Paris
lookfor("Mexico", Capitals) =>

** <false>
```

Recall that a closure of one procedure is simply a procedure which is obtained from the first procedure by fixing the values of some of its variables. Clearly an association procedure simply needs to make a closure of $lookfor$.

```
define assoc1(L);      ;;; Produce an association procedure embodying L
```

```
  lookfor(%L%)         ;;; Result is the lookfor procedure
enddefine;             ;;; with association list frozen to be L.
```

Alternatively, we can use lexical variables to create this closure by moving the body of the lookfor procedure into assoc:

```
define assoc2(L);
  lvars L;
  procedure(O);        ;;; This procedure is the same as the lookfor
  lvars O p;           ;;; procedure above, and is returned as result
    for p in L do        ;;; of assoc2, with L frozen.
      if p.hd = O then
      return(p.tl.hd)
      endif
    endfor;
  return(false)
  endprocedure
enddefine;
```

This *assoc* function is deficient compared with the one provided by the system in that it does not allow the user to assign new values — we cannot do $"Vienna" \rightarrow capital("Austria")$. In order to accomplish this we would need to define a procedure which updates the association list, and this is a little complicated because we may have started off with a null association list in the first place — the null list cannot be changed! There are two solutions to this difficulty.

- In order to allow us to (destructively) change the association list in all circumstances we *cons* on an extra object to the head of this list. $lookfor$ will then simply skip over this, while the updating procedure will assign to its tail.
- Since POP allows us to do “surgery on closures” with the procedures *pdpart* and $frozval$, we could make the updating of an association procedure be accomplished by replacing the frozen value of the procedure itself.

Adopting the first strategy, we can define an updating version of $assoc2$.

```
define assoc3(L)->P;                    ;;; Create an updatable property
  lvars L,                              ;;;
        L1 = [assoc ^^L],               ;;; Cons on an extra element to
                                        ;;; allow updating of the list.
        P  =
              procedure(O);
              lvars O p;
                for p in L1.tl do      ;;; Skip the extra element and
                  if p.hd = O then     ;;; iterate until a pair whose head
                  return(p.tl.hd)      ;;; is O is found, returning the
                  endif                ;;; associated value.
                endfor;
              return(false)            ;;; If O is not found, the
              endprocedure,            ;;; result is false.

        P_update =                     ;;; This procedure does the updating
              procedure(v,O);          ;;; of the association.
              lvars O p;
                for p in L1.tl do      ;;; If we have found O,
                  if p.hd = O then     ;;; update the pair to have the
                  return(v -> p.tl.hd) ;;; new value, and return.
                  endif
                endfor;
              [^O^v] :: L1.tl -> L1.tl;;;; Otherwise change the association
              endprocedure;            ;;; list to include new object-value
                                      ;;; association
  P_update -> P.updater;              ;;; Make the updating procedure be
enddefine;                            ;;; the updater of P.
```

We can check this new version out, both adding new values by updating, and changing old ones:

```
vars capital = assoc3([]);
capital("France")=>
** $false$
"paris" -> capital("France");
capital("France")=>
```

```
** paris
"Madrid" -> capital("Spain");
capital("France")=>
** Paris
capital( "Spain") =>
** Madrid
"Rome" -> capital("Italy"); ;;;(etc.)
```

11.3 Properties

A POP property is a procedure that efficiently maps objects, to other objects . In the computing literature the objects that are mapped by a property are often called *keys* and what they are mapped to are called *values*. However, to avoid confusion with the *key* objects described in Chapter 3.13, we shall avoid this usage.

One way of creating properties is to use the procedure *newassoc*, described in section 11.3.2: this is called using a similar format to that used for *assoc*. However, POP provides other procedures, of which the most general is *newanyproperty*, which allow you much more flexibility in choosing what is meant by 'equality' in comparing entries, in what to do when an entry does not exist and in trading space against time.

Thus a POP property is a procedure which when called with some object O_{arg} as argument will return the object O_{val} associated with O_{arg}, if there is one, or a user-supplied default value, if there is not. Associations between objects may be set up by calling the same procedure in *update* mode, or in some cases, by supplying a list of initial associations.

Each property contains a set of *entries*, each of which associates a particular O_{arg} with a particular O_{val}. POP-11 provides two procedures for creating properties: *newproperty*, which is described in section **??** and deals with simple cases using a default mapping rule, and *newanyproperty* offering more flexibility.

Properties are organised around a table of 'buckets', each of which contains a list of entries; which bucket an entry is put in is determined by a hashing algorithm. Hashing procedures are treated in Chapter **??**. The number of buckets is user-specifiable: a larger number of buckets will decrease the access time for entry, since the lists in each bucket will be shorter, but at the cost of using more space.

Hashing procedures should use some invariant feature of the object O_{arg} to calculate a number that, appropriately scaled, indicates the bucket in which an entry for O_{arg} is stored. If the hashing procedure produces a wide variety of results for different keys, most entries are stored in unique locations, and thus very little 'blind' searching is needed when entries are retrieved. This is why, provided more than a small data set is stored, properties are faster than procedures produced with *assoc*.

There are two types of property, *permanent* and *temporary*. The only difference between them involves the interaction of the property with the garbage collector, which is described in Chapter 17.4. Suppose we have a property P which has an entry associating an object O_{val} (the value) with an object O_{arg} (the argument), and suppose that there are no other references remaining to the argument O_{arg} anywhere else in the POP system. Then, if a property entry is simply scanned during a garbage collection like any other record, the presence of the argument O_{arg} in the property entry will prevent it from being garbage collected. Indeed if P is a *permanent* property this is exactly what happens, so that both O_{arg} and O_{val} are preserved. In such a case the only way in which the entry for O_{arg} could then be accessed would be using the procedure *appproperty*, described in section 11.3.3.

However, a temporary property is treated specially by the garbage collector. The arguments of temporary property entries are never considered it in deciding which records are garbage. Thus by itself the presence of an object O_{arg} as the argument of a temporary property entry will not stop its garbage collection, or the garbage collection of the value object for O_{arg} if nothing else refers to that also. Should this happen then not only will O_{arg} be garbaged but also the entry for O_{arg} will be deleted from the property.

Thus permanent properties may be used for retaining permanent tables of information, while temporary properties may be used for hanging ad hoc pieces of information onto arbitrary records which will get garbage collected along with those records.

Using the procedure *newanyproperty* users may specify their own hashing algorithm: this is particularly useful if you wish to associate the same data with objects that are not always identical (as tested with ==) but merely structurally equal (as tested by =). Users may also create "expandable" properties, the tables of which will automatically enlarge by a given factor whenever the table passes a given threshold value. Expandable properties will never decrease in size even if the number of entries in the property drops below the given threshold.

newanyproperty also allows users to create "active" properties. When an object is not found in an active property then, instead of returning the default value, the object and the property is passed to a user supplied procedure to return the result.

A facility for constructing 'multi-dimensional' properties is also provided by the 'sparse array' mechanism, described in section **??**.

11.3.1 Predicates on Properties

As mentioned above, properties are procedures: thus *isprocedure* is true for any property.

$isproperty(O) \to b$
This procedure returns $true$ if O is a property, $false$ if not.

11.3.2 Simple ways of constructing properties

All of the procedures described below make use of *newanyproperty*, which is described in section 11.4.

$newassoc(L_{assoc}) \to P_{prop}$
This is the simplest interface to *newanyproperty*. It constructs permanent properties with default value $false$. It is defined as:

$$newproperty(L_{assoc}, 20, false, true) \to P_{prop}$$

where L_{assoc} is as for *newproperty*.

$newproperty(L_{assoc}, n, O_{default}, b_{perm}) \rightarrow P_{prop}$
This procedure returns a new property P_{prop}. The arguments are:

- L_{assoc} An initial list of associations, where each association is a 2-element list of the form $[< argument >< value >]$.
- n The table size, i.e. the number of buckets to be used.
- $O_{default}$ The default value to be produced when there is no entry for a given argument.
- $b_{perm} = true$ for a permanent property, $b_{perm} = false$ for a temporary property.

  Note that assigning the default value $O_{default}$ to an argument object actually removes the entry for that object if there was one.

$newmapping(L_{assoc}, n, O_{default}, b_{expand}) \rightarrow P_{prop}$
This procedure provides a simpler interface to *newanyproperty* for constructing properties that match argument objects on the basis of the standard structure equality operator[1] = and that use the standard procedure *syshash*, described in section **??**, to hash objects into buckets. The argument n is the initial table size.

If $b_{expand} = true$ it specifies that the property should have an k_{expand} of 1 and a t_{expand} of n. *newmapping* is thus defined as

```
define newmapping(L_assoc,n,O_default,b_expand) -> P_prop;
     newanyproperty(L_assoc,
                    n,
                    if b_expand then
                      1, n
                    else
                      false, false
```

[1]See Chapter 3.7.

```
                    endif,
                    syshash,  nonop =, false,
                    O_default, false
                   ) -> P_prop
enddefine;
```

11.3.3 Manipulating Properties

The procedures in this section allow you to treat a property as a whole, either to process all of its entries, or delete them.

appproperty(P_{prop}, P)
This applies the procedure P to each entry in the property P_{prop}. For each entry, P is given two arguments, i.e

$$P(O_{arg}, O_{val})$$

Note that P is effectively applied to a temporary copy of P_{prop} so that any alterations made by the procedure P to P_{prop} itself during the execution of *appproperty* will have no effect on the entries to which P is applied. Thus P can safely delete entries or add new ones.

Note also that since the organisation of property entries depends on a hashing algorithm, and also on the order in which entries are added, etc, the order in which entries are given to P is indeterminate. Moreover, it may change between garbage collections.

$fast\_appproperty(P_{prop}, P)$
This applies the procedure P to each entry in the property P_{prop}. For each entry, P is given two arguments, i.e.

$$P(O_{arg}, O_{val})$$

It works just like *appproperty* except that property table is not copied first. This means that $fast\_appproperty$ should not be used if the procedure is going to change any of the entries in the property.

$clearproperty(P_{prop})$
This removes all entries from the property P_{prop}.

$property\_default(P_{prop}) \to O$
This procedure returns the default value for the property P_{prop}.

The following example uses *newproperty* to create a property relating people's names to their ages:

```
vars procedure
    ageof = newproperty([[fred 23][mary 18][bill 99]], 10, 0,
                            true);

ageof("bill")=>
** 99
ageof("mary")=>
** 18
ageof("susan")=>
** 0                ;;; produces default value

"ageless" -> ageof("susan");
ageof("susan")=>
** ageless
0 -> ageof("bill"); ;;; removes entry for bill

define printage(person,age);
    pr(person), pr('\t'), pr(age), pr(newline);
```

```
enddefine;

appproperty(ageof, printage);

fred    23
mary    18
susan   ageless
```

11.4 The most general property generator

The procedure that provides all of the facilities discussed in the introduction to properties is *newanyproperty*, which takes nine arguments, summarized in the table below:

Argument	Meaning
L	The initial association, as in *assoc*
n	The initial size of the hash table
k_{expand}	How much to expand the hash table if needed
t_{expand}	If the property has more entries than this, it will be expanded.
P_{hash}	The procedure to be used for computing hash values.
P_{eq}	The procedure to be used to decide when two keys are equal
$b_{garbage}$	Indicates whether to garbage collect objects only occurring in P
$O_{default}$	These two arguments are used to indicate
$P_{default}$	what to do when a new key-object is encountered

1. The list L is a list of initial argument/value associations, as with *newproperty*. For example:

   ```
   newanyproperty([[one 1][two 2][three 3]], ...) -> prop;
   ```

   creates a property, *prop*, that maps the word *"one"* to the number 1, and so on. It is equivalent to:

   ```
   newanyproperty([], ...) -> prop;
   ```

```
1 -> prop("one");
2 -> prop("two");
3 -> prop("three");
```

2. The integer n allows you to specify the approximate size of the property table. The figure given, which should be a positive integer greater than one, does not affect how many entries you can store in the table, but can affect the speed with which they can be retrieved. In general, the larger n is, the faster entries can be found. A rough guide might be that properties are at their most efficient if they are about 75% full. However, if lots of properties are made with too much spare capacity the wasted space can lead to excessive paging by the computer Operating System, slowing programs down. So users will need to experiment to find appropriate sizes.

3. The expansion factor k_{expand}, which can be an *integer* or $false$, together with the t_{expand} argument, allows you to create properties which are efficient in space and time despite the fact that their size is not known in advance. If k_{expand} is a positive (short) integer, an "expandable" property is created, which automatically grows bigger when a certain number of entries has been added, determined by:

4. the expansion threshold t_{expand}

   When expansion takes place, k_{expand}, which should be a positive integer, indicates how much bigger the property should get, determined by the left shift operation[2]:

$$\begin{aligned} n_{new} &= n_{old} << t_{expand} \\ &= 2^{t_{expand}} n_{old} \end{aligned}$$

   Here n_{old} is the number of buckets in the property before expansion, and n_{new} is the number after. Thus an k_{expand} of 1 means that the property will expand to twice its original size. If this argument is $false$, the property size is fixed whatever the value supplied to the t_{expand} argument. If the t_{expand} is $false$ then the property will first expand when n objects have been added to the property.

   After expansion, the t_{expand} is increased in proportion to the property's new size.

5. The argument P_{hash}: If this argument is $false$, entry locations are computed from the O_{arg}'s address in memory, except for (short) integers, which are used as their own hash code.

   The advantage of this is that hashing is very fast; the disadvantage is that properties must be rehashed after a garbage collection since the address of a O_{arg} may have

[2]See Chapter 6.18

changed, and that argument objects must be identically equal, as determined by ==, to be considered as the same by the property procedure. Note that rehashing takes place automatically. This is the method used by properties created using *newproperty*.

P_{hash} should be a procedure that takes one argument, O_{arg}, and returns one result. The result can be any POP object which is then automatically mapped into a location in the table. The simplest method is to return an integer or a simple decimal number. A useful general-purpose hash function is *syshash*, described in Chapter **??**.

If the hashing procedure relies on the absolute address of the key then it is necessary to rehash the property table after a garbage collection. In this case it is necessary to set the argument $gcflag = true$ (q.v.). If the procedure uses the address of the key and $gcflag \neq true$ then the property is very unlikely to return the correct associated object for a key after the first garbage collection — since the address of an object can change after a garbage collection.

If the hashing procedure does not return a number or simple decimal, then it is a POP-11 data structure and its address will be used to determine a location in the table. Hence it is important in this case to set $gcflag = true$. Note that unless the property is required to produce a random result, the P_{hash} must always return an identical object for equivalent keys. For example it would be a mistake for the procedure to return a string unless it is always the same identical string.

6. The argument P_{eq} is used to check that a given O_{arg} matches the the argument part of an entry in the table.

   The default procedure, used if the argument is $false$, is the identity procedure ==. This is the procedure used by *newproperty*. For implementation reasons it is not possible to specify an P_{eq} if no P_{hash} has been specified.

7. The argument $b_{garbage}$ serves two purposes. As mentioned above, if a P_{hash} is provided then this argument is a flag which specifies if the property table needs to be rehashed after a garbage collection, i.e. when the O_{arg}'s are hashed on the basis of their absolute address.

   If no P_{hash} is provided then this argument serves the same purpose as the final argument to *newproperty*, described in **??**. That is it specifies if the property is to be a *permanent* property. Items in a temporary property which are not referenced from any other identifier are removed from the property on garbage collection.

8. The argument $O_{default}$ This operates in conjunction with:

9. The argument $P_{default}$

If an entry cannot be found for O_{arg} when looking up the property table, then if the $P_{default}$ is $false$ the $O_{default}$, which can be any POP object, is returned. If however the $P_{default}$ is a procedure then it is applied to O_{arg} and the property and the result returned.

On updating an entry in a property, if you update O_{arg} to be associated with $O_{default}$ then the effect is to remove the entry for O_{arg} from the property table, regardless of the value of $P_{default}$.

The fact that $P_{default}$ is given the property as argument makes for considerable flexibility. For example the new argument-value pair can be entered into the property. This is particularly useful if the default procedure takes a long time to run, and the same argument-value pairs keep on coming up, as is the case, for example, in algebraic simplification. This will be discussed some more in section 11.5 on Memo Functions.

Below is an example that uses *newanyproperty* to create a property relating peoples' nick-names to their full-names, where an active default procedure *nochange* is used to map nick-names with no explicit full-name to the name itself:

```
define nochange(object, property) -> object;
    lvars object, property;
enddefine;

vars procedure
    alias = newanyproperty([
                            [Pooh 'Winnie the Pooh']
                            [Tricky 'Richard Nixon']],
                           8, false, false, false, false,
                           true, false, nochange);

alias("Pooh") =>
** Winnie the Pooh
alias("Snoopy") =>
** Snoopy
```

11.5 Memo Functions and newanyproperty

So far we have regarded properties as repositories of "rote knowledge" — a mapping that is arbitary, and cannot be expressed as a procedure except as a large conditional. However properties can also be used as a means of recording what the result of applying a procedure to an argument is, and thus avoiding recomputing the procedure. This is an idea analogous to cashing values in a high-speed store of a computer. In its procedural form it was first proposed by D.Michie [?] and implemented in POP-2 by Michie and Chambers.

Using properties we can quite easily implement a simple form of memo-functions. The procedure we provide below is called by $newmemo(P, n)$, where P is the procedure being memoized, and n is a count of the maximum number of agrument-value pairs to be kept. After this is exceeded, the property is cleared, and memoizing re-starts. This is usually necessary in some form if extensive use is being made of the memoized procedure, in order to prevent store being clogged up — one of us remembers watching a SUN work-station getting slower and slower as its store was clogged up with various results of a memoized algebraic simplifier. Putting in a cut-off immediately improved performance. Various elaborations of this scheme can be imagined. For example you might keep usage counts of the arguments in the property, and only purge those that were little referred to, as is done in cashing schemes for virtual memory management. However, since the accounting has to be done in software, the cost in time of these elaborations might not be worth while in many applications.

```
define auxmemo(O1,Prop,P,n,ref_i)->O2;
  lvars O1,O2,Prop,P,n,i,ref_i;
  ref_i.cont-1 ->>i -> ref_i.cont;
    if i= 0 then n -> ref_i.cont;
    clearproperty(Prop);
    endif;
  P(O1) -> O2;
  O2 -> Prop(O1);
enddefine;

define newmemo(P,n);
  newanyproperty([],n,false,false,syshash,nonop=,false,undef,
  auxmemo(%P,n,consref(n)%));
enddefine;
```

```
define fact(n);
  if n=0 then 1 else n*fact(n-1)
  endif
enddefine;

newmemo(fact,20) -> fact;   ;;; Now make fact into a memo-function.
trace fact;

fact(2) =>       ;;; This executes normally.
>fact 2
!>fact 1
!!>fact 0
!!<fact 1
!<fact 1
<fact 2
** 2
fact(4) =>
>fact 4
!>fact 3
!!>fact 2         ;;; At this point we know the answer -- no need to
!!<fact 2         ;;; recurse further.
!<fact 6
<fact 24
** 24
```

11.6 Sparse Arrays

It is sometimes necessary to use large arrays in which many of the elements will all have some default value, and only relatively few will have a 'significant' value, i.e. different from the default. Represented as ordinary arrays, such *sparse* arrays are very wasteful of space, since the *arrayvector* must allow storage cells for each element corresponding to a given combination of subscripts, and yet many or most of these will just repeat the default value.

Rather than using an vector-type structure to hold the elements, POP sparse arrays are therefore implemented as multi-dimensional properties, that is, they use a tree of properties to associate each set of subscripts to a value. This means that only the 'significant' elements take up space, and moreover, the 'subscripts' are not restricted to being integers, but can be any POP objects. Although this approach can save space, it does however mean that accessing sparse array elements is slower than for ordinary arrays.

A sparse array can also have an 'active default' — i.e. a procedure that is run to decide what the default value of an element should be when it has not been assigned a value explicitly; for certain applications this can save even more space.

$newanysparse(n, O_{default}) \rightarrow A_{sparse}$
$newanysparse(n, P_{default}, apply) \rightarrow A_{sparse}$

$newanysparse(L, O_{default}) \rightarrow A_{sparse}$

$newanysparse(L, P_{default}, apply) \rightarrow A_{sparse}$

This constructs and returns a new n-dimensional sparse array procedure (where $n \geq 1$). This can be called as

$$A_{sparse}(i_1, i_2, \ldots, i_n) \rightarrow O$$

$$O \rightarrow A_{sparse}(i_1, i_2, \ldots, i_n)$$

to access or update the element associated with a given set of n 'subscripts', which can be any objects at all. Note that subscript equality means identity, that is to say the $==$ operation, is used and not $=$. The first argument specifies the number of dimensions n, either directly as an integer or as an n-element list L. In the latter case, the elements of L are integers giving the table sizes to be used for the properties employed for each dimension; in the former case, when n is simply an integer, the table size in each dimension defaults to 20. Choosing the table size for a property is discussed in section **??**.

Note also that subscripts are dealt with from right to left, so that putting 'larger' dimensions to the left of 'smaller' ones will increase efficiency.

The remaining argument(s) specify the default value for elements of the array: in the first form of the call, the object $O_{default}$ is the fixed default value for every element. The second form specifies an 'active default' procedure $P_{default}$, and to distinguish this from the first form, in which $O_{default}$ could be a procedure, the last argument must be the procedure *apply*. $P_{default}$ is expected to be a procedure which takes n 'subscript' arguments and returns a default value, i.e.

$$P_{default}(i_1, i_2, \ldots, i_N) \rightarrow O_{val}$$

This is comparable to a procedure specified as the O_{init} argument to *newanyarray*.

$newsparse(L) \rightarrow A_{sparse}$
$newsparse(n) \rightarrow A_{sparse}$

This is a simpler version of *newanysparse* which has the word "*undef*" as a fixed $O_{default}$, i.e. it is the same as

```
newanysparse(%"undef"%)
```

11.6.1 Examples of Sparse Arrays

The first example is an sparse array representing points in 3-D space, where each element in the array is a list of points to which that point is connected to; each point is defaults to being connected just to itself:

```
        define here(x, y, z);
            lvars x, y, z;
```

```
        [^x ^ y ^z]
    enddefine;
```

Note — doing this for real you would be better to define a procedure, *cons3vec* say, to create the vectors.

```
    vars procedure
        connected = newanysparse(3, here, apply);
```

In this array, each cell contains a list of the points it is directly connected to:

```
    connected(1, 2, 3) =>
    ** [[1 2 3]]
    connected(8, 9, 10) =>
    ** [[8 9 10]]
```

Some explicit connectivity can then be added:

```
    [8 9 10] :: connected(1, 2, 3) -> connected(1, 2, 3);

    connected(1,2,3) =>
    ** [[8 9 10] [1 2 3]]
```

The second example creates a sparse array in which the 3 'dimensions' are English words, foreign languages, and modalities, and where each element is the translation of the English word into the corresponding language and modality, the default value of $false$ indicating that no translation is known:

```
    vars procedure
        dictionary = newanysparse(3, false);
```

```
"bonjour" -> dictionary("hello", "french", "polite");
"ola" -> dictionary("hello", "spanish", "familiar");

dictionary("hello", "french", "polite") =>
** bonjour
dictionary("hello", "french", "vulgar") =>
** $false$
```

Try, as an exercise, creating these property procedures with $assoc1$ and $assoc2$ and $assoc3$ and extracting their $frozval$ and $pdpart$ components. Write a procedure which produces association procedures using strategy (2).

Chapter 12

Processes

12.1 Overview

You will need processes if you want to have a number of procedures each of which may have to stop running because it needs some data, typically from an external source. When this happens, the process can be *suspended* until a later occasion in which data is available, and it is *resumed*.

For example, let us consider trying to simulate the behavior of an automatic railway. Suppose you are restricted to using procedures in writing your program. You will want to write a procedure called *track*, say, which will simulate the behavior of the track. This will do things like read the sensors associated with particular blocks of track, and set the signals controlling access to each block according to whether or not there is a train on that block, thus enforcing what is known as "absolute block working". For each train, you will want to call a procedure called *train*, which will read the signals, and determine at what speed it should go.

There is a severe problem in writing a simulator thus — the only ways a procedure can hand over control to another procedure is (a) to *return* to a calling procedure (b) to *call* the other procedure. Now if a procedure returns, it loses information about what it was doing. It could of course store such information in a data-structure, and interrogate the data-structure when it is restarted. However this would lead to an awkward style of programming, whereby each procedure would be structured as a big conditional or *go_on* switch, to allow it to carry on where it had left off when it did its *return*. Thus, to write a simulation procedure naturally, it should *call* other parts of the system to which it belongs. However this would mean that *track* would call *train* and *train* would call track, so that a these procedure would never actually get to execute the program after the call.

To remedy this problem, POP-11 provides the idea of a *process*. This essentially is a data-object which encapsulates the state of a procedure-call (including any sub-procedures) at a time when control is handed back. For example, in our railway simulation, we might divide simulated time into 1 second slices, and require our procedures to hand back control

```
define runtrain(train);
  lvars
    train
    block  = block_train(train), ;;; The block of track the train is on
    v_comm = signal(block),      ;;; The speed the train is allowed to go.
    v_act  = v_train(train),     ;;; The speed the train is actually going.
    s      = s_block(train);     ;;; The distance along the current block

    if v_act>v_comm then         ;;; Are we going too fast?
    apply_brakes(train)          ;;; if so - slow down
    else apply_power(train)      ;;; otherwise speed up
    endif;

  s+v_act*dt -> s;               ;;; Add on how far we have gone in 1 sec.
    if s>s_max(block) then       ;;; If we have exited the current block
    next_block(block) -> block;   ;;; then find the next block and
    block ->  block_train(train);;;; make the train be in that block and
    0 -> s_block(train);         ;;; make the train be at the beginning
    endif;

enddefine;
```

Before control theorists jump on me to point out this “bang-bang” approach with a sample time of 1 second would give an uncomfortable ride and waste energy, I hasten to add that the point I am trying to make is computational, and not control-theoretic.

If a train were as simple as above, we could indeed use a procedural approach. We would need to construct a scheduler, which took a list of train records, and applied *runtrain* to each of them. Suppose however that a train needs to stop at a station. This could be controlled from the track (as you would with a model railway). Thus the commanded velocity would be steadily reduced to zero as the train approached and entered the station, and would be held at zero while the train was to remain in the station, and be increased again to move the train away from the station. Such a scheme is not likely to be used in actual practice — the train needs to make some decisions. For example the doors must all be closed before the train can start again. So we may wish to enlarge the *runtrain* procedure:

```
define runtrain(train);
  lvars
    train;

define runcontrol();
    if v_act>v_comm then         ;;; Are we going too fast?
    apply_brakes(train)          ;;; if so - slow down
    else apply_power(train)      ;;; otherwise speed up
    endif;
enddefine;


  repeat forever
  lvars
    block  = block_train(train), ;;; The block of track the train is on
    v_comm = signal(block),      ;;; The speed the train is allowed to go.
    v_act  = v_train(train),     ;;; The speed the train is actually going.
    s      = s_block(train);     ;;; The distance along the current block

    if is_station(block) then
    lvars u = v_max(block),
          s1 = s_stop(block);
      until v_act = 0 do         ;;; Maintain the velocity profile to come
      u^2s - 2*a_brake*s -> v_comm; ;;; to a stop in a distance determined
```

```
      runcontrol();              ;;; by the maximum permitted speed for
                                 ;;; the block
      suspend(0);
      enduntil;
    open_doors();
    t + t_station -> t_start;
      until t>t_start do
      suspend(0)
      enduntil;
    close_doors();
    suspend(0);
      until closed_doors() do
      open_doors();
      suspend(0);
      close_doors();
      suspend(0);
      enduntil;
    endif;

  endrepeat;
enddefine;
```

Firstly, we have got rid of the section of the procedure which dealt with simulating the physical behavior of the train. The reason for doing this is that *we should construct our simulation in such a way that modules representing physical behavior can be unplugged and replaced by the actual mechanism itself*. One module is software that will run in an actual train, the other is software that simulates the train.

A process in POP is a data-object that records the state of execution of a piece of POP program. The information stored in a process record comprises the sequence of procedure calls (stack-frames) that the process is currently inside, including the values of local variables of those procedures, and the state of the user stack.

A process is constructed initially in two ways:

1. from a procedure with *consproc*, in which case on running it for the first time with

runproc (or *resume*), the procedure is called in the normal way;

2. from part of the currently active procedure calls with *consprocto*, in which case execution will re-commence from the call to *consprocto* when the process is run.

Thereafter, the process may suspend itself at any time, e.g. by using *suspend*. This causes the current state of execution, i.e. all procedure calls upto *runproc* and the user stack, to be stored in the original process record. The process is then *swapped out*. The process may then be re-activated with *runproc* and will continue execution immediately following the *suspend* call, after the stored state of execution has been reinstated.

A process can also cause itself to be swapped out by calling *resume* to resume another process in its place (see below). There are also versions of *suspend* and *resume* (*ksuspend* and *kresume*) which *kill* the current process. This means that the process' state is swapped out but not stored, the process record being marked as *dead*. The process cannot then be run again. A process is also killed if a normal procedure exit to *runproc* is made.

Note that a process always has its own user stack, which is separate from the stack of any other process or from the normal stack when not running inside a process. Thus all arguments passed into or out of a process have to be explicitly declared in calls of *runproc*, *suspend*, etc. The exception to this is when a process does a normal procedure exit to *runproc*, or exits through *runproc* with *chain*: in this case *all* values on the process' stack are passed up as results. Thus if you leave stuff on the stack in a process which you do not want passed back on normal or chained exit, use *clearstack* to clear the stack first.

A further facility allows the suspension of *any* active process. An active process is one currently in the calling chain, i.e. that is running the current process, or running the one running that, etc. The procedures *suspend*, *resume*, *ksuspend* and *kresume* can all take an optional process argument $\bar{P}_{sus}$ specifying the active process to be (k)suspended. All processes up to and including $\bar{P}_{sus}$ are either killed (*ksuspend* and *kresume*), or suspended in such a way that on running or resuming $\bar{P}_{sus}$ the whole process chain is reactivated, and control returns from the original call (*suspend* and *resume*). A process chain like this is also constructed by *consprocto* if the calling sequence to the target procedure includes one or more processes; when the constructed process is run it will reactivate the whole chain.

12.2 Predicates on Processes

The following procedures permit you to find out if a data-object is a process and what its state is.

$isprocess(O) \to b$
This procedure returns $true$ if O is a process, $false$ if not.

$isliveprocess(\bar{P}) \to b$
This procedure returns a $true$ result if the process $\bar{P}$ is alive, and $false$ if dead. In the first case, the result is $\bar{P}$ itself if $\bar{P}$ is also currently active, and $true$ otherwise. Thus the expression

$$isliveprocess(\bar{P}) == \bar{P}$$

is true if $\bar{P}$ is the current active process.

$deadproc(\bar{P}) \to b$
This procedure returns $true$ if the process $\bar{P}$ is dead, $false$ if not. This procedure is now in the autoloadable library, having been superseceded by $isliveprocess$.

12.3 Constructing Processes

$consproc(n, P, b_{kill}) \to \bar{P}$
$consproc(n, P) \to \bar{P}$
The result of this procedure is a process constructed on the procedure P. The process has its initial user stack set to contain n items passed from the current stack. b_{kill} is an (optional)

boolean argument specifying whether the process should be killed if it is abnormally exited from (i.e. chained out of) while executing; *true* means kill, *false* means keep alive (in which case the state of the process will remain as it was at its last *suspend*). If not specified, this argument defaults to *true*.

consprocto$(n, P, b_{kill}) \rightarrow \bar{P}$
consprocto$(n, P) \rightarrow \bar{P}$ This procedure constructs a process $\bar{P}$ from the current calling sequence upto and including the the most recent call of the procedure P (it is an error if there is no current call of P). Any processes running below the call of P are suspended first. These will be reactivated when the process $\bar{P}$ is run. The user stack of $\bar{P}$ is then set to contain n items passed from the current stack, i.e. the stack as it is *after* suspending any intervening processes. To allow computation, in that environment, of the number of items to be passed, the argument n may also be a procedure which returns the number, i.e. $n()$ is evaluated after suspending intervening processes. The argument b_{kill} (and its default) are as for *consproc*.

copy$(\bar{P}_1) \rightarrow \bar{P}_2$
On a process, *copy* returns an exact copy of its argument. Note that: (1) any running of the original process has no effect on the copy, and (2) if the process being copied is the process currently running, then the state copied is the state at the time of the last *runproc*, *resume*, etc, em not the current state.

saveproc$() \rightarrow \bar{P}$
This procedure is used to save the state of the current process. It returns a process record $\bar{P}$ which when rerun will exit from the call of *saveproc* with *false* as result instead of the process record. Explicitly, the current process state is stored in a new process record, with *false* added to the saved user stack; this process record is returned as result.

12.4 Running, Suspending and Resuming

pop_current_process [protected variable] Always contains the current process, or *false* if no processes are active. You can thus use

$$pop\_current\_process == \bar{P}$$

to test if $\bar{P}$ is the current process.

$runproc(n, \bar{P})$
This runs the process $\bar{P}$ as a subprocess of the current process (or from outside any process), passing n items from the current stack to the process' stack. If *runproc* is used inside a process, the calling process is not 'swapped out'. All calls of the outer process remain in the calling chain, *but* the outer process' user stack is saved (somewhere) so the called process still runs with its own stack. Since *runproc* is the *class_apply* of processes (see 3.13), this can also be called as

$\bar{P}(n)$

$suspend(n)$
$suspend(n, \bar{P}_{sus})$

This suspends the current process, or all processes upto and including $\bar{P}_{sus}$ and returns from the call of *runproc* which ran the suspended process. n items are passed as results from the current process stack.

$resume(n, \bar{P}_{res})$
$resume(n, \bar{P}_{sus}, \bar{P}_{res})$

This runs the process $\bar{P}_{res}$ after swapping out the current process, or all processes upto and including $\bar{P}_{sus}$. n items are passed from the current user stack to the new process' stack. Thus the new process runs within the same call of *runproc* as did the old one, and suspending the new one will return from that call of *runproc*.

$ksuspend(n)$
$ksuspend(n, \bar{P}_{sus})$

This kills the current process, or all processes upto and including $\bar{P}_{sus}$, and returns from $runproc$, passing n items back.

$kresume(n, \bar{P}_{res})$
$kresume(n, \bar{P}_{sus}, \bar{P}_{res})$
This kills the current process, or all processes upto and including $\bar{P}_{sus}$, and then resumes the process $\bar{P}_{res}$, passing n items to the new process' stack.

12.5 Miscellaneous

process_key
This constant holds the key object for processes (see 3.13).

Chapter 13

Sections

NOTES

I need to write the historical and comparative note properly.

The example at the end of the chapter, while explanatory, is not really a good example of how sections should be used.

13.1 Overview

In any programming language which is to be used for creating complex programs there is a need to permit different entities to have the same name. To some extent this is provided by local variables in POP - two procedures can both have a local variable x without any risk of unwanted confusion between them, particularly if the variable is lexical. Of course sometimes you do want a local variable of one procedure to be a non-local variable of another. However the capabilities provided by local variables are not enough to satisfy all of the needs for isolation. It may also be desirable to isolate global variables, particularly those that hold procedures. Suppose you want to write a collection of procedures that provide a capability to be used by other people. This might be perhaps a graphics *section*, or a matrix *section*.

You will specify this capability to the user by saying that you provide certain procedures, and perhaps some variables. However, when you implement the capability, as well as defining the "published" procedures, you will define some auxiliary ones, which will not be available to the user. Indeed, perhaps they should not be available to the user, because you may wish to keep the option of changing the code in the section radically while still preserving the same external interface, for example for portability from one machine to another. It is to allow people to provide this kind of capability that sections are provided.

A section is a portion of POP program of the form:

```
<section> = section <pathname> <imports> => <exports>;
            <expression_seq>
            endsection
```

Sections in effect construct a "naming tree" very like the directory structures in operating systems, and the $<$ *pathname* $>$ specifies a path in that tree to a node in the tree in which the variables local to the section "roost". The $<$ *imports* $>$ and $<$ *exports* $>$ are sequences of variables which are referred to in the section, and which are to be the same as variables outside the section. $<$ *imports* $>$ are variables that are declared outside the section but used inside, $<$ *exports* $>$ are variables that are declared inside the section, but may be used outside.

13.2 Comparisons with other languages

Sections were added to POP-2 in the first revision (CHECK). They correspond to modules in .... and packages in LISP.

13.3 How Sections work

As described in 5.3, a declaration of a permanent program identifier, i.e. a variable or a constant, results in the attachment of a identifier record to the corresponding word, this record maintaining the *idval* (or $valof$) and *identprops* of the program identifier. Program *sections* provide a means whereby this attachment can be made on a localised basis, i.e. the same word can be associated with different permanent identifiers in different sections of a program. Note that the section mechanism does *not* apply to lexically-scoped identifiers, i.e. those declared with *lvars* or *lconstant*. For the rest of this chapter $identifier$ will mean *permanent identifier*.

We have said that the motivation for sections is that in writing a part of a large program (or in writing a library program or system which other people are going to use), it may be convenient to use particular words to name private identifiers on the basis of mnemonic significance. But at the same time these private identifiers should not conflict or interfere with identifiers of the same name either in other parts of the program, or that users of the library program or system will employ. This applies particularly to frequently used variable names, e.g. x. For example, the mere use of a variable name in a library routine which a user has loaded will prevent a warning message being issued for that variable when employed by a user who has forgotten to declare it as local to a procedure.

Another useful aspect of sections is that they can be cancelled. Cancelling a section simultaneously cancels all the permanent identifiers local to that section, implying that the words used to reference them, if not used elsewhere, will be garbage collected.

POP-11 sections are analagous to directories in a tree-structured file system, where the identifiers play the role of files. Just as files in different directories may have the same name, so identifiers in different sections may also; just as directories may have sub-directories, so sections may have sub-sections. Just as there is the concept of 'current directory', so there is the concept of 'current section', and the user may make any node in the section tree the current section at any time — changing sections in this way involves the system in manipulating the identifier fields in all word records currently in the dictionary.

The ability to *import* and *export* identifiers provides a facility not (usually) found in file systems. Importing an identifier named x into a sub-section b of a section a means that references to x in b refer to the identifier associated with x in a; similarily, exporting an identifier y from b up to a means that references to y in a refer to the identifier as declared in b. Thus the former allows references to identifiers already declared in sections above the current one, while the latter allows new identifiers to be declared in such sections from within the current.

The section tree is constructed of section records, each section record containing information about the identifiers local to that section, as well as a list of section records for sub-sections of that section. The root node of this tree is the section record in the constant *pop_section*, which represents the 'top-level' of the POP system, and all other sections can be reached by working downwards from this. Procedures are provided to enable the user to effect this and other manipulations on sections and identifiers at run-time (see below), as well as syntactic constructs for use at compile-time.

13.4 Pathnames

As with directories, sections are identified by name; a *pathname* syntax is provided to enable reference to identifiers within sections in a manner similar to Unix's file pathnames. The word "\$−" is used to separate parts of the pathname, rather than "/" as in Unix. Thus, for example:

```
$-tom$-dick$-harry
```

refers to the identifier *harry* in subsection *dick* of section *tom* (*tom* being a subsection of *pop_section*), while

```
$-foo
```

refers to the top-level identifier *foo*. Again analagously to Unix, a pathname not beginning with '\$-' is taken to be relative to the current section, e.g.

```
bill$-ben
```

means the identifier *ben* in the subsection *bill* of the current section.

13.5 The standard syntactic form of sections

The *section* construct is the principal way of using sections when compiling programs. It has the form

```
section <pathname> <imports> => <exports> ;
    <expression_seq>
endsection
```

where both $< imports >$ and $< exports >$ are optional sequences of words, the $'' =>''$ being omitted if there are no exports. $< pathname >$ is a pathname as described above, but in *this* context it refers to a section, not an identifier. So

```
section $-tom$-dick$-harry; ... endsection
```

specifies subsection *harry* of subsection *dick* of subsection *tom* of *pop_section*, etc. In addition, the name of *pop_section* is $< blank >$ (it is!), so that omitting the pathname references the top-level, thus

```
section; ... endsection
```

There must be no imports or exports in this case because they don't make sense at top-level.

The effect of *section* is to save the current section, make the named section the current, and then continue compiling until *endsection* is encountered, at which point the previous current section is restored. After entering the named section, *section_import* is called on each word specified by $< imports >$, and *section_export* on each specified by $< exports >$, as described in section 13.7 below.

13.6 Global Identifiers

Certain permanent identifiers, e.g. those in the system and those in the autoloadable library, are normally required to be accessible in all sections, and thus to be imported into sections without any explicit declaration to that effect. To this end, an identifier can be declared as *global*, meaning that it should be considered an automatic import into any sub-section of a section where it is accessible. This can be done either at run-time with *sysGLOBAL* (see Chapter 16), or at compile-time with a *vars* or *constant* statement prefixed by *global*, e.g.

```
global vars x, y, z;
global constant a, b, c;
```

The keyword *global* can also appear in a *define* statement after the word *define*, but before any *vars* or *constant*. An example is:

```
define global x(); ... enddefine;
define global constant y(); ... enddefine;
```

13.7 Section Procedures

These procedures allow the manipulation of sections by the user. The syntactic constructs described above are in fact implemented in terms of these. At system startup time the current (and only) section is *pop_section*, which represents the 'normal' top-level of *pop*. New sections are then created by use of *section_subsect* as described below.

On entering a section (by assigning to *current_section*), all non-imported words have their identifiers set to *undef*, with the following exceptions:

- All system words.
- All words having system identifiers associated with them.
- All words having associated with them identifiers marked as global. This actually subsumes (2), since all system identifiers are so marked.

After this, words having section-local identifiers are set appropriately. The process of entering a section in general involves 'unwinding' all sections up to top-level, and then recursively entering all sections from there down to the given one, although in certain cases this process can be "optimised".

Whenever a new identifier is declared (i.e. with *vars*, *constant* etc), this identifier is made local to the current section, unless the identifier name has been declared as an export (see below). There is currently no facility for creating new identifiers in any section other than the current. Redeclarations of existing identifiers merely alter the information in the existing identifier record, and so do not change their section status in any way.

13.8 Predicates on Sections

To recognise section records the following is provided:

$issection(O) \rightarrow b$
Returns *true* if O is a section, *false* otherwise.

13.9 Creating/Manipulating Sections

The following procedures can be used to make and modify section records.

$section\_subsect(W, Sect, b_{create}) \rightarrow Sect_{sub}$
$section\_subsect(Sect) \rightarrow L_{sub\_sect}$

In the first form of the call, given a section $Sect$, this returns the subsection called W of $Sect$, where W is a word. If no such subsection currently exists, then

1. If b_{create} is true, a new subsection of $Sect$ called W is created and returned;
2. If b_{create} is $false$, the mishap NONEXISTENT SECTION occurs.

In the second form, given a section $Sect$, returns a list of all subsections of $Sect$.

$section\_supersect(Sect) \rightarrow Sect_{super}$
Given a section $Sect$, returns the section $Sect_{super}$ of which $Sect$ is a subsection, or $false$ if $Sect$ is the top-level section *pop_section*.

$section\_cancel(Sect)$
$section\_cancel(Sect, b_{zap})$
This cancels the section $Sect$, i.e. breaks the link to $Sect$ from its supersection. $Sect$ must not be top-level, although it is quite alright for $Sect$ to be the $current$ section.

If the optional boolean argument b_{zap} is $true$, the *pdprops* of all procedures held in local identifiers of $Sect$ are set to $false$, providing that they are user procedures whose current *pdprops* is the word with which the identifier is associated — if not the *pdprops* are left untouched.

section_cancel also recursively applies itself, with the same value for b_{zap}, to any subsections of $Sect$, cancelling them too.

$section\_name(Sect) \to W$
This procedure returns the name of $Sect$, or $false$ if $Sect$ is *pop_section*. Note that the name does not include the full pathname of $Sect$.

13.10 Standard Sections

pop_section
The value of this constant is the top-level section record, the root node of the section tree.

$pop\_default\_section$
This variable holds the default section to return to on doing a *setpop* (or when $vederror$ is called inside VED). In other words, they both do the assignment

```
pop_default_section -> current_section;
```

The initial value of this variable is *pop_section*.

$current\_section$
This (active) variable holds the current section; its initial value is *pop_section*.

13.11 Importing/Exporting Identifiers

$section\_export(word)$
This procedure declares the word $word$ to be an export of the current section (which must not be top-level), meaning that whenever $word$ is subsequently declared, the identifier attached

to it is made local to the section above, or the section above that if it is exported from there , etc. At the same time, *word* is made an import to the current section; thus the identifier actually 'rises' to the highest level section it is *not* exported from, while at the same time 'sinking' down from there to the current section through all intermediate sections. If *word* currently has a local identifier associated with it, this 'rises' as described above and ceases to be local; if it has an associated imported identifier, then exporting has no effect unless *word* is cancelled and redeclared, in which case the redeclaration is exported.

$section\_import(word)$
Declares the word *word* to be an import of the current section (which must not be top-level), thus making available the identifier associated with *word* in the super-section of $Sect$. If *word* already has a local identifier associated with it, this is *cancelled*. *word* is automatically declared in the super-section if it is undefined there.

13.12 Other operations on sections, and constants

$sys\_read\_path(W_{first}, b_{use\_itemread}, true) \rightarrow Sect$
$sys\_read\_path(W_{first}, b_{use\_itemread}, false) \rightarrow W$

This procedure reads a section/identifier pathname from the current input stream (i.e. from *proglist*). If the second argument is $true$, the pathname is interpreted as a section name, and the appropriate section record is returned; if $b_{needsect}$ is $false$, the pathname is interpreted as referring to an identifier, and the appropriate $word\_identifier$ (see below) is the result. It is assumed that the pathname begins with the word W_{first}, which has already been read from *proglist*; successive items of the pathname are then read with *itemread* if $b_{use\_itemread}$ is true, or *readitem* otherwise. If the first item has not been read then the W_{first} argument should be obtained by *readitem*() or *itemread*(), etc.

$word\_identifier(W, Sect, b_{see\_imports}) \rightarrow W_{id}$
This procedure effectively enables the user to gain access to identifier records, although indirectly through word records. Given a word W and a section $Sect$, it returns a unique word W_{id} which has associated with it the identifier associated with W in the section $Sect$, or

returns $false$ if there is no associated identifier. The word W_{id} is not entered in the dictionary, and so does not participate in the section mechanism; thus the identifier associated with it is always guaranteed to be that associated with W in $Sect$ In other words W_{id} is a 'symbolic' representation of that particular identifier record.

What is meant by 'the identifier associated with W in $Sect$' further depends on the value of the boolean argument $b_{see_imports}$. If this is true, then imported identifiers are taken into account, i.e. the state of W as it would be if $Sect$ were the current section is considered; if $false$, only identifiers strictly local to $Sect$ are relevant. In either case, the characters of the word W_{id} are the full section pathname of its identifier, except that top-level identifiers are not prefixed by '$-'.

section_key
This constant holds the key structure for sections (see Chapter 3.13).

13.13 Examples

We will now illustrate the mechanics of sections by some examples. Suppose we declare *list* as a constant:

```
constant list = [this is a list of words];
```

Normally, the following procedure definition will produce a mishap, because *list* has already been declared as a constant (and so cannot be a procedure local):

```
define count(list);
    if list == [] then
```

```
            0
        else
            hd(list) + count(tl(list))
        endif;
    enddefine;
```

However, if we put this definition inside a section called *various* (if we want to make sure it's a subsection of top-level we should call it $-various)

```
    section various;

    define count(list);
        if list == [] then
            0
        else
            hd(list) + count(tl(list))
        endif;
    enddefine;

    endsection;
```

this is fine, because the local variable *list* used in *count* now has nothing whatever to do with the previous definition of *list* as a constant — the word "*list*" is associated with different identifiers inside and outside of section *various*. Outside of this section, the identifier which is the local of *count* is inaccessible by the name "*list*", but could be accessed as $''various\$ - list''$.

On the other hand, this applies also the identifier *count*, i.e. it cannot be accessed outside of *various*, and this is probably not what we want. So if *count* is required to be used outside of *various*, we can make it an export of the section:

```
    section various => count;
```

```
define count(list); ... enddefine;
endsection;
```

Thus the definition of *count* is treated as if it were made outside of *various*, while at the same time *list* remains internal to the section.

Now on entering a section, it is *not* the case that any identifiers defined outside are automatically accessible inside. This is generally what we want, i.e that these external identifiers should not conflict with ones which are internal to the section. So in the example above, the constant *list* is not available inside *various*, and so does not conflict with the local of *count*. If we have another constant, *vector* say, and we try to access it inside section *various*, it will be undefined:

```
constant vector = {1 2 3 4 5 6 7 8 9};
section various;
vector=>
;;; DECLARING VARIABLE vector
** <undef vector>
endsection;
```

However, we can either make *vector* an explicit import of section *various*

```
section various vector;
vector=>
** {1 2 3 4 5 6 7 8 9}
endsection;
```

or, if we know that we are going to want to import it into any section, we could have declared it as global in the first place:

```
global constant vector = {1 2 3 4 5 6 7 8 9};
section various;
vector=>
** {1 2 3 4 5 6 7 8 9}
endsection;
```

Note that, in the definition of *count* above, the system procedures +, ==, *hd* and *tl* were accessible inside *various* by virtue of being declared global, as all system identifiers are.

Chapter 14

In which we learn how to use and extend the POP library

14.1 Note on Directory Names

In this file, directories names are specified in Unix format, e.g.

```
'$popautolib'  '$poplocal/local/lib'
```

etc, where the leading $ specifies an environment variable name at the beginning. For VMS, the environment variables just translate to logical names, i.e.

```
        'popautolib:'  'poplocal:[local.lib]'
```

etc.

14.2 Library Search & Compilation

$syslibcompile(Name, L_{dir}) \rightarrow \mathbf{s}_{dir}$
$syslibcompile(Name) \rightarrow \mathbf{s}_{dir}$

This is the main procedure for compiling POP-11 source files from libraries: it searches sequentially through the directories given by L_{dir} for the file specified by $Name$, the first such file found being compiled (with *compile*), and the relevant directory returned as the result. If the file is not found in any of the directories, the result is $false$. $Name$ is a string or a word, which gives the name of a library file (and must not include any directory pathname). L_{dir} is a list of pathnames (strings) specifying the directories in which to search, and if omitted, defaults to *popliblist* (see below). Each directory in turn is tested for a file called $Name.p$ (or $Name$ if this already contains the extension '.p'). To enable hooks to be added to this mechanism (e.g. for automatic definition of autoloaded identifiers), L_{dir} may also contain procedures mixed in with the pathname strings. If a procedure P is encountered in the search of L_{dir}, it is called with $Name$ as argument, and is expected to return a result reflecting whether it 'found' $Name$ or not. That is,

```
P(Name) -> O;
```

where $O = false$ means the search of L_{dir} should continue, and a true value means return from $syslibcompile$ with result $result$.

$prautoloadwarn(Name)$
This variable procedure is called by $syslibcompile$ when it finds a library file specified by $Name$, immediately before compiling it. (Note that this only happens when $Name$ is found in an actual directory, not when 'found' by a procedure in L_{dir}.) The default value is $erase$, which means that autoloading etc is normally done silently; $sysprautoloadwarn$ is available to assign to this variable when notification is required.

$sysprautoloadwarn(Name)$
This procedure prints the line

```
;;; AUTOLOADING <Name>
```

on the standard output, i.e. through *cucharout*.

popliblist
The list of autoloadable library directories, used as the default search list by *syslibcompile*. Its initial value will be something like:

```
['$poplocalauto' '$popautolib'  '$popvedlib' ldots]
```

etc.

14.3 Autoloading

sys_autoload$(W) \rightarrow \mathbf{s}_{dir}$
This procedure is called by parts of the system that require to check that an autoloadable definition of a permanent identifier called W is loaded (e.g. see the description of macro expansion in 15.2, and *sysdeclare* in 16). It first checks to see if W is already declared as a permanent identifier, and returns *true* if so. Otherwise, it calls

```
syslibcompile(W) -> {\bf s}_{dir}
```

to try to autoload a file $' < W > .p'$ from one of the directories in *popliblist*, the result of that procedure being returned. It should normally be the case that an autoloaded file will declare W as a permanent identifier; note that during the call of *syslibcompile*, W is marked specially, so that a call of *sys_autoload* on the same word during compilation of the file will return $false$ rather than cause a recursive call of *syslibcompile*. In addition, an abnormal exit from *sys_autoload* during compilation will remove any declaration that W may have acquired, thus ensuring that W remains undeclared after a compilation mishap.

poplibdir [variable] Contains a string naming the main POP-11 autoloadable library directory. No longer used by the system, since it is always assumed to be '$*popautolib*' — its default value.

14.4 Loading Library Files Explicitly

loadlib(*Name*)
This procedure calls *syslibcompile* with *popuseslist* as the list of directories to search (see below), i.e.

```
syslibcompile(Name, popuseslist)
```

(and calls *mishap* if the library file was not found).

lib
This is a macro for calling *loadlib*, used as *lib* < *Name* > where *Name* is an identifier name (or filename, extending upto a ";" or newline). It substitutes itself on *proglist* with code to print the message

```
;;; LOADING LIB <Name>
```

and call *loadlib*(*Name*).

uses
A macro for loading a library file for a given identifier name that may or may not already be loaded. It reads a single word *Name* from proglist, i.e.

```
uses <Name>
```

will test if $Name$ is already declared as a permanent identifier, and if not, will call *loadlib*($Name$).

popuseslist
This holds the search list of directories for use with *loadlib*, *lib* and *uses*. Its default value is created when any one of those is first invoked, and will be something like

```
[^^popliblist '$poplocal/local/lib'
                    ^popliblibdir ^popdatalib]
```

popliblibdir
This holds a string naming the main library directory for non-autoloadable files. Its value is inserted into the list *popuseslist* when one of *lib*, *loadlib*, or *uses* is first invoked. (It can be assigned to, but this will have no effect if done after *popuseslist* is constructed). (Default value '$*popliblib*').

*popdatalib*variable This holds a string naming the directory used for storing information used as data by other programs. Also inserted into *popuseslist* when that is first used. (Default value '$*popdatalib*').

14.5 Miscellaneous

sysstring
sysstringlen

sysstring is a string of length *sysstringlen*, which libraries and other programs can use as a character buffer to give to routines such as *sysread* and *syswrite* this is set up in $lib/; sysstring$.

popdisk
In VMS, this is the *disk* containing the POPLOG system. Not used by system procedures. Its default value is $'\$usepop'$. In Unix, the directory.

Chapter 15

In which we achieve such intimacy with the POP compiler that it will do odd jobs for us.

NOTES What about the obsolete forms? especially sysnvariable.

To date we have treated the POP compiler as a monolithic procedure which will convert a sequence of characters into VM code for execution. In this chapter we discover that it is indeed a coordinated bundle of procedures, many of which are available to you to use for reading and compiling whatsoever little chunks of POP as you deem meet to the occasion. Having achieved mastery over the material in this chapter and its successor, which deals with the Virtual Machine, you will be readily able to embellish POP with new syntactic constructs of your own designing. An example of such, namely the definition of the $foreach$ construct, is to be found in Chapter 16.8.

15.1 The anatomy of the POP-11 Compiler

The POP-11 compiler consists of a set of procedures corresponding to the various syntactic constructs in the language, as specified by the syntax diagrams given in Appendix **??**. These divide into two classes:

1. those which begin with a syntax or syntax-operator word, W (say). An example of such a word is $"define"$. These each have an associated procedure, which is called by the compiler whenever it encounters them in an appropriate context. Their modus operandi is determined by the specific associated procedure, although it in its turn may call procedures which are treated below; their effect is described in the text of this book that treats them, mostly in Chapter 2. This chapter is only concerned with how the compiler recognises them and decides to call the associated procedure. Such a word is recognised by the compiler from the value of $identprops(W)$ as described in Chapter5.3.5. Macros can be regarded as a user or library defined extension of this class.
2. those which are not associated with a particular syntax word, for example the $< expression >$ construct.

The next section 15.2 treats *proglist*, which is the list of objects (or *tokens* to use the language of the compiler literature) which the POP compiler operates from.

Subsequent sections deal with the second class of syntactic construct, which are: $< compilation\_stream >$, $< statement\_sequence >$, $< statement >$, $< expression\_sequence >$ and $< expression >$. They deal only with those specific constructs that have generally-useful utility procedures associated with them.

15.2 In which we meet the list of objects that the compiler works with

A standard requirement for compilers and other utilities in POP is the ability to read source text in itemised (or 'tokenised') form, i.e. to have an input character stream *lexically* analysed into objects such as words, numbers, strings, etc. This process can be performed, for example, by the item repeater procedures constructed by *incharitem* from character repeaters, which are treated in Chapters 20 and 19.

For programs which wish to process such an input object stream, it is particularly convenient to do so via a dynamic list constructed from the item repeater [1]. An item can then be obtained simply by taking the head of the list, and saved there until the next item is required, at which point the list can be tailed. Another advantage of this method is that extra items can be inserted into the stream when necessary, by concatenation onto the front of the list, as is the case for macro expansion, described below.

Because many parts of programs may require access to the item stream currently in use, it is also useful to define a standard location where the input list can be found; for this reason, POP provides the global permanent variable *proglist*. The list (dynamic or otherwise) in this variable is used as the input item stream by many system modules, including the POP-11 compiler, as described later in this chapter.

Although *proglist* can be processed directly, it is conventional, and usually more convenient, to take input items from it using the item-reading procedures described below. Of these procedures, *itemread* and *nextitem* also provide a macro expansion facility.

15.2.1 Reading Items from *proglist*

Let us now consider the forms in which access to the input object stream is available.

proglist
This is the input list of items (possibly dynamic) used by *readitem*, *nextread*, *itemread* and *nextitem*, etc.

poplastitem
This contains the last item read from *proglist* with *readitem* or *itemread*.

[1] Dynamic lists are treated in Chapter **??**

$readitem() \rightarrow O$

This procedure removes the next item from $proglist$ and returns it. If $proglist$ is $null$, then $termin$ is returned. The item returned is saved in $poplastitem$.

$nextreaditem() \rightarrow O$

This procedure returns the next item from $proglist$, but without removing it. Thus another call of $nextreaditem$ will return the same item again. $termin$ is returned if $proglist$ is $null$.

$readtill(O_{end}) \rightarrow O_{end} \ldots \rightarrow O_2 \rightarrow O_1$
$readtill(L_{end}) \rightarrow O_{end} \ldots \rightarrow O_2 \rightarrow O_1$

This procedure reads items with $readitem$ until an item either equal to O_{end} (first form) or a member of L_{end} (second form) is encountered. All items read, including the last, are returned.

$readline() \rightarrow L$

This procedure reads items from $proglist$ (using $readitem$) until a $newline$ is encountered. This is done by setting $popnewline$ true, so that $newline$ is returned as a word. The result is a list containing all the items read, excluding the newline.

pop_readline_prompt

The item in this variable, whose default value is $'?\backslash s'$, is locally assigned to $popprompt$ during $readline$. See Chapter **??** for possible values of $popprompt$.

$requestline(bfs_{prompt}) \rightarrow L$

This procedure is the same as $readline$, but with *pop_readline_prompt* locally set to bfs_{prompt}.

$rdstringto(O_{end}) \rightarrow bfs$
$rdstringto(L_{end}) \rightarrow bfs$

This procedure reads items with $readitem$ until an item either equal to O_{end} (first form) or a member of L_{end} (second form) is encountered. The result is the string formed by concatenating together the printed representations of all the items read, *excluding* the last. The procedure *sys_* ><, described in Chapter 21 is used to perform the concatenation.

$readstringline() \rightarrow bfs$
This procedure reads a string of characters bfs up to, but not including, a newline or $termin$, in the following way:

- If *proglist* is an unexpanded dynamic list, then *nextchar*(*readitem*) is used to get the characters until *newline* or *termin* is encountered (see Chapter 19). Leading spaces are ignored; a backslash character may be used to escape a following newline, backslash, or leading space when this is to be included in the string.
- Otherwise, if *proglist* actually contains some items already, the result is

  ```
  rdstringto([^newline ^termin])
  ```

  (with *popnewline* set true).

15.2.2 Reading Items with Macro Expansion

Chapter 2.26.1 dealt with the POP *macro* concept from a user's point of view. The compiler recognises a *macro* as a word W which is currently declared as a lexical or permanent identifier with $identprops(W) ='' \; macro''$. (For information on *identprops* see Chapters 8.1 5.3). When such a word is read from *proglist* using the procedures *itemread* and *nextitem*, it undergoes macro expansion, that is, the word is replaced in *proglist* by a sequence of items derived from the value of the identifier. The item actually returned by the call of *itemread* or *nextitem* is then the first of these, unless this is again a macro, in which case expansion proceeds recursively until the first item is not a macro. Thus a macro can substitute itself in the input stream with any desired items.

The effect of a macro however can only be "down stream", that is to say it cannot affect any text which preceded it. Thus the syntax extension achieved by macros is always to introduce constructs which begin with a fixed word, the macro name.

The sequence of items into which a macro is expanded depends on its identifier value, which may be either a procedure, a list, or any other item (except an undef record):

- For a procedure, the expansion is all the items left on the user stack by calling it (i.e. the topmost item on the stack is the last in the sequence). The procedure is called with the next N items from *proglist* (read with *itemread*) as its arguments, where N is its *pdnargs*.
- For a list, the expansion is all the elements of the list (possibly none).
- For any other item, the expansion is that item. A mishap will result if the item is an undef record (on the assumption that this means the macro's value is undefined).

Expansion is actually performed by collecting the items in a list L, and then adding them to *proglist* with

$L <> proglist \rightarrow proglist;$

after removal of the macro word itself and any arguments. Thus the original list pairs constituting *proglist* are *not* updated, which can be important in some contexts, as described in *proglist_macro_pair* below.

Note that before checking for a macro, *nextitem* and *itemread* will attempt to autoload any word which is not currently declared as an identifier by calling *sys_autoload*, which is described in Chapter 14. This permits autoloading of (permanent) macros, and is also the mechanism by which autoloading takes place in POP-11. When unwanted, this can be turned off by locally assigning [] to *popliblist*.

If a macro word is required to be read from *proglist* without macro expansion, it must be preceded by the word "*nonmac*", as described in section 15.2.3 below.

$itemread() \rightarrow O$
After expanding macros (and possibly autoloading undefined words), this procedure returns and removes the next non-macro item from *proglist*. *termin* is returned if *proglist* is *null*. The item returned is saved in *poplastitem*.

nextitem() → *O*
As for *itemread*, but does not remove the returned object from *proglist*. Thus another call of *nextitem* will return the same object again.

proglist_macro_pair
As described above, macro expansion always creates new list pairs to add to *proglist*, and does not update the original ones; this variable provides a means for macro procedures to actually update the original *proglist* if they wish to do so. When a macro procedure is applied, *proglist_macro_pair* holds the list pair on *proglist* that actually contained the macro word. In the case in which the macro has no arguments, the next pair on *proglist* will be the *tl* of this. Thus for example, a macro producing a single object can, as well as returning it as a result, also assign it directly to *hd*(*proglist_macro_pair*) (alternatively, it can do the latter and then assign *proglist_macro_pair* to *proglist*, returning no result). Of particularly relevance in this context are macros that read characters from the input stream using *nextchar* (see Chapter 19), which can fail to work properly with certain compilers (e.g. POP-11) that expect to be able to read sections of *proglist* more than once. The reason for this is that on a second or subsequent reading the original *proglist* still contains the macro word, but no longer has available the characters required by the macro, since it has been itemised ahead of that point. Macros of this type should therefore use *proglist_macro_pair* to replace themselves directly, so that re-reading of the object stream will function correctly.

15.2.3 Proglist-Related Built-in Macros

nonmac
This specifies that the next word on *proglist* is not to be treated as a macro, so that when read with *nextitem* or *itemread* the word will not be expanded. E.g. if *proglist* is

```
[nonmac Pooh ...]
```

where "*Pooh*" is a macro, then *itemread*() will evaluate to "*Pooh*". On the other hand, *readitem*() would evaluate to "*nonmac*".

#_ <
> _#

#_ < implements 'on the fly' macros by allowing the evaluation of a sequence of POP-11 statements in the input stream; its usage is

```
#_< statement-sequence >_#
```

The statement sequence up to > _# is compiled and evaluated, and any objects left on the stack in so doing are spliced back onto *proglist* in place of the whole construct. For example, if *proglist* is

```
[ #_< 3 + 4 * 5 >_# ...]
```

then *itemread*() will evaluate to 23. Note that > _# is simply a macro that produces *termin*. Thus #_ < can compile the statement sequence with *pop11_comp_stmnt_seq_to*(*termin*) (see section 15.5).

#_IF
#_ELSEIF

#_ELSE

#_ENDIF

These macros enable the conditional reading or skipping of objects in *proglist*. Their usage is

```
#_IF     expression
    item-sequence
#_ELSEIF expression
```

```
    item-sequence
#_ELSEIF expression
    item-sequence
#_ELSE   expression
    item-sequence
#_ENDIF
```

where the *#_ELSEIF* and *#_ELSE* clauses are optional, and where each POP-11 expression is terminated by a newline as for *readline* (qv).

Starting from the *#_IF*, each expression occurring after a *#_IF* or *#_ELSEIF* is evaluated until one is found that returns a non-*false* result, object sequences following a false result being skipped without macro expansion. The sequence following the true result, or that following a *#_ELSE* if no expression evaluated to true, is then 'allowed through' in the sense that the current call of *itemread* etc and subsequent calls will return these objects (if there is no true clause or *#_ELSE* then whatever object follows the *#_ENDIF* will result).

These macros may be nested to any depth, i.e. any of the object sequences can contain further *#_IF* constructs.

15.3 The compiler procedures

All compiler procedures take source code input objects from the list *proglist*, using the procedure *itemread* described in section 15.2, and emit POP Virtual Machine code directly. No parse tree is built.

The interface procedures *compile* and *popval* set up *proglist* initially to compile code either from a character stream or from an existing list, and then commence compilation by calling *pop11_comp_stream* to compile the input stream.

compile(Stream)
This procedure compiles POP-11 program text from the character source *Stream*. Permis-

sible values of *Stream* are:

- A character repeater procedure.
- An argument valid for *discin*, i.e. a filename string or word, or a device record.

Unless given a character repeater procedure directly, *compile* first translates its argument into a character repeater using *discin* as described in chapter 20. Note that, if given a word as a filename, *discin* will append $'.p'$ to it. It then assigns the repeater locally to *cucharin*, and sets up *proglist* locally with

$$pdtolist(incharitem(cucharin)) \rightarrow proglist;$$

It then calls $pop11\_comp\_stream$.

$cucharin() \rightarrow c$
This contains the character repeater procedure initially set up in *proglist* by *compile*. Changing this variable will affect nothing, since after *proglist* is set up compiler procedures do not reference *cucharin* again.

$trycompile(Filename) \rightarrow b$
If the file specified by *Filename* exists, *compile* is called with it as argument, and, provided no mishaps occur during the compilation, *true* is returned. *false* is returned if the file doesn't exist.

$popval(L)$
This procedure compiles the sequence of objects held in the list L: this is defined simply as

```
define popval(proglist);
    dlocal proglist;
```

```
                    pop11_comp_stream()
                enddefine;
```

For example:

```
vars L = [23 + 4 =>];
popval(L);
**27
```

popval has two main uses:

- For compiling POP code that has been synthesised by a procedure. See for example the Chapter **??** on the *let* macro. A more efficient alternative in such cases is to call the VM code planting procedures directly, as described in Chapter 16.

- For delaying the evaluation of a macro. For example you may say in the middle of a procedure body:

  ```
  define Pooh(adventure);
  popval([lib Heffalump;]);
  enddefine;
  ```

  This has the effect that the library procedure *Heffalump* will only be loaded when the procedure *Pooh* is called, and not when it is compiled. This is particularly useful when using *syssave* – *sysrestore* to develop large programming systems, as described in Chapter **??**. You can use *sysrestore* to restore a basic core of your system, which may include many externally loaded procedures, and then load the current version of files under development by making use of a sequence such as

  ```
    syssave('Winnie_the_Pooh.psv')->;   ;;; Save the image, ignore result.
    popval([lib Heffalump]);            ;;; Load library files either after
                                        ;;; save or after restore.
  ```

15.4 Stream Compilation

pop11_comp_stream()
This procedure sets up a new execute level compilation of POP-11 code, and then compiles *proglist* until it becomes *null*. Aside from initialising certain global variables locally, this procedure is in essence:

```
define pop11_comp_stream();
    sysCOMPILE(termin, pop11_exec_stmnt_seq_to) ->
enddefine;
```

that is, call *pop11_exec_stmnt_seq_to*(*termin*) to compile and execute statements inside a new invocation of the VM compiler.

popclosebracket
This contains the closing 'bracket', i.e. syntax word, for the current syntax construct being compiled. It can also be a list of such words when the construct has a choice of closers. Syntax procedures normally set this variable locally to an appropriate value before commencing compilation of a construct, and then call *pop11_need_nextitem* with that value to check for the closer at the end.

15.5 Statement Sequence Compilation

A < *statement* > is an expression sequence, i.e. zero or more expressions separated by commas. A < *statement_sequence* > consists of zero or more such < *statement* >s, separated by semicolons. Note that the standard syntax operators " =>" and " ==>" also act as statement separators by replacing themselves on *proglist* with semicolons.

Expression and statement sequences are largely interchangeable in POP-11, except in a few places (such as procedure call arguments) where the latter are not allowed. At execute level, the end of each statement indicates where execution is to take place.

$pop11\_comp\_stmnt\_seq()$

This procedure compiles and plants VM code for a sequence of statements separated by semicolons.

$pop11\_comp\_stmnt\_seq\_to(W_{closer}) \rightarrow O_{found}$
This procedure is the same as $pop11\_comp\_stmnt\_seq$ except that during compilation of the sequence, $popclosebracket$ is set locally to be the argument W_{closer}, and after compilation

$$pop11\_need\_nextitem(W_closer) \rightarrow O_{found}$$

is executed to check the next object on $proglist$ and produce the result. $pop11\_need\_nextitem$ is described in section 15.8.

$pop11\_exec\_stmnt\_seq\_to(W_{closer}) \rightarrow O_{found}$
This procedure behaves in the same way as $pop11\_comp\_stmnt\_seq\_to$, save that it executes the VM code planted at the end of each statement by calling $sysEXECUTE$, described in Chapter 16.3.1.

15.6 Expression Sequence Compilation

$pop11\_comp\_expr\_seq()$
This procedure compiles and plants VM code for a sequence of expressions separated by commas. A 'MISSING SEPARATOR' mishap will occur if the next object on $proglist$ after the sequence does not match the current value of $popclosebracket$ and is an expression opener. An expression opener is anything except $termin$ or a non-procedure syntax word, i.e. a word whose $identprops$ are "$syntax$" and whose value is not a procedure.

$pop11\_comp\_expr\_seq\_to(W_{closer}) \rightarrow O_{found}$
This procedure is the same as $pop11\_comp\_stmnt\_seq$ except that during compilation of the sequence, $popclosebracket$ is set locally to be the argument W_{closer}, and after compilation

$$pop11\_need\_nextitem(W_{closer}) \rightarrow O_found$$

is evaluated to check the next object on $proglist$ and produce the result. See section 15.8 for a description of $pop11\_need\_nextitem$.

15.7 Expression Compilation

The procedure $pop11\_comp\_prec\_expr$ is the heart of the POP-11 compiler: it compiles a single expression containing operators up to a specified maximum precedence. Since an understanding of this procedure is helpful in writing new syntax or syntax operator constructs, its operation is described in greater detail in the section **??** below.

$pop11\_comp\_prec\_expr(i_{prec}, b_{update}) \rightarrow O_{next}$
This procedure compiles and plants VM code for a single expression, the extent of which is determined as follows:

- If the first object is not an expression opener, or is a syntax word equal to $popclosebracket$, then the expression is empty.
- Otherwise, the expression has the generic form

  ```
  <object> <operator expr> <operator expr> ...
  ```

  where each operator expression is either a syntax operator followed by an appropriate code body, or an ordinary operator followed by a sub-expression. The expression is terminated by meeting either a non-operator, or an operator whose internally-represented precedence (see below) is greater than or equal to the argument i_{prec}.

The i_{prec} argument is a value specifying the limit for operator precedences in this expression; for efficiency reasons, it is supplied not in the normal $identprops$ format, but in the form in which precedences are actually represented internally. If x_{prec} is a normal *identprops* precedence, then the corresponding i_{prec} value is the positive integer given by

$$i_{prec} = intof(abs(x_{prec}) * 20) + (if\ x_{prec} > 0\ then\ 1\ else\ 0\ endif)$$

E.g. an x_{prec} of 4 will give a i_{prec} of 81, whereas -4 would give 80. Since an *identprops* precedence can range between -12.7 and 12.7, the normal range for i_{prec} is 2–255; any value greater than 255 is guaranteed to include all operators in an expression.

The b_{update} argument is a boolean which if true causes the expression to be compiled in update mode rather than normal evaluate mode.

The result of the procedure is the object following the expression. Note that this remains as the next object on *proglist*.

pop_expr_prec
pop_expr_update

These two variables are dynamic locals of *pop_comp_prec_expr*, and hold the values of the i_{prec} and b_{update} arguments for the current call of that procedure.

pop_expr_inst
pop_expr_item

These two variables are also dynamic locals of *pop_comp_prec_expr*, and together constitute a buffer for planting VM code. Their use is described in section 15.10 below.

$pop11\_comp\_expr()$
This procedure compiles and plants VM code for a single expression with unlimited operator precedence. Its behavior is the same as: $pop11\_comp\_prec\_expr(256, false)$

$pop11\_comp\_expr\_to(W_{closer}) \rightarrow O_{found}$
This procedure is the same as $pop11\_comp\_expr$ except that during compilation of the expression, $popclosebracket$ is set locally to be the argument W_{closer}, and after compilation

$$pop11\_need\_nextitem(W_{closer}) \rightarrow O_{found}$$

is executed to check the next object on $proglist$ and produce the result.

15.8 Utility Procedures to Test/Check for Input Items

$pop11\_need\_nextitem(O) \rightarrow O_{found}$
This procedure checks the next object from $proglist$ to be (a) a member of O if O is a list, or (b) equal to O otherwise. If neither condition holds, the mishap

```
MSW: MISPLACED SYNTAX WORD
```

or

```
MEI: MISPLACED EXPRESSION ITEM
```

results, depending on whether the next object is an expression opener or not. Here the INVOLVING list of the mishap is

```
FOUND <nextitem> READING TO <O/hd(O)>
```

Otherwise, the next object is removed from $proglist$ and returned. This procedure uses $nextitem$ to examine the next object, so macros are expanded, autoloading can occur, etc.

$pop11\_try\_nextitem(O) \rightarrow O_{found}$
This procedure works in the same way as $pop11\_need\_nextitem$, except that $false$ is returned if O (or a member of O) is not the next object. If it is, then the next object is removed from $proglist$ and returned as result.

$pop11\_try\_nextreaditem(O) \rightarrow O_{found}$
This procedure is the same as $pop11\_try\_nextitem$, except that $nextreaditem$ is used to get the next object, so that autoloading or macro expansion can't occur.

15.9 Procedures Associated With Individual Syntactic Constructs

$pop11\_comp\_constructor(W_{closer}) \rightarrow b$
popconstruct

This procedure compiles and plants code for a list or vector constructor, i.e. $[\ldots]$ or $\{\ldots\}$, after the opening bracket has been read, and compiles to the closing bracket W_{closer}, which should therefore be either "]" or "}".

In general, code is planted to construct the list or vector at run-time, so that a new one will be produced every time the code is run; however, if the variable *popconstruct* is true, then constructor expressions not containing any of the 'evaluate' keywords "%", " ↑" or " ↑↑" will be compiled as constants, i.e. actually constructed at compile-time. Note that inside a procedure this means that any updating of the components of the structure will permanently change it.

The result b is $true$ if a constant structure was produced, $false$ if not.

$pop11\_comp\_declaration(P_{declare})$
This procedure is used by the $constant$, $vars$, $lvars$, $dlvars$ and $lconstant$ constructs to read declaration/initialisation statements for identifiers, as specified in Chapter 5 and Appendix

??.

The argument $P_{declare}$ is a declaration procedure of the form $P_{declare}(W, O_{idprops})$ (e.g. $sysVARS$), which for each word appearing in the declaration is called on the word and its specified $O_{idprops}$, the latter defaulting to 0 when not specified.

If an identifier declaration is followed by an initialisation expression then this is compiled, and, except in the case where $P_{declare}$ is $sysLCONSTANT$, the code $sysPOP(WORD)$ is planted (and executed immediately if at execute level). For $sysLCONSTANT$, the expression is evaluated immediately and the result O assigned with $sysPASSIGN(O, W)$.

pop_args_warning
This boolean variable determines whether the *procedure* and *define* statements warn about formal argument and result identifiers that are not locally declared, i.e. that do not appear in an identifier declaration or *dlocal* statement immediately following the procedure header. Such identifiers are automatically made dlocal permanent identifiers; if *pop_args_warning* is *false*, this is done silently, otherwise the warning message

```
<WORD> DEFAULTED TO VARS IN <PROCEDURE_NAME>
```

is printed.

$pop11\_define\_declare(W_{id}, P_{global}, P_{decl}, O_{idprops})$
The procedure in this variable (which is dynamically local to *compile*) is called by the *define* statement to declare as an identifier the name of the procedure being defined, when such a declaration is needed. This is in all cases *except* the following:

- *dlocal* is specified, i.e. '*define dlocal* ...'. In this case the identifier is assumed to be declared already.
- *updaterof* is specified without any declaration keywords, the name is already declared as a lexical or permanent identifier, and, for a nested define, the identifier is 'local' to the enclosing procedure, i.e. an *lvars* of it, or a *dlocal* of it.

The arguments are:

W_{id} : The name of the identifier to be declared.

P_{global} : The procedure *sysGLOBAL* if *global* was present in the header and $false$ otherwise.

P_{decl} : The *sys_* declaration procedure for a declaration keyword if present (e.g. $sysVARS$ for $vars$, $sysCONSTANT$, $sysLVARS$, etc), or $false$ if none.

$O_{idprops}$: A value for the *identprops* of the name, i.e. the *identprops* keyword specified, or $false$ if none.

The default procedure in this variable, which makes use of the variables $popdefineconstant$ and $popdefineprocedure$, is shown below:

```
define vars pop11_define_declare(W_id, P_global, P_decl,
                                              idprops);
    lvars P_global, P_decl, idprops, W_id;

    unless P_decl then
        ;;; no explicit declaration given
        if isprocedure(popdefineconstant) then
            ;;; use that procedure to declare the identifier
            popdefineconstant
        elseif popdefineconstant and popexecute then
            ;;; declare as constant at execute level
            sysCONSTANT
        else
            ;;; declare as variable
            sysVARS
        endif -> P_decl
    endunless;

    unless idprops then
        ;;; no explicit identprops given -- declare as
```

```
        ;;; procedure-type if $popdefineprocedure$ true
        if popdefineprocedure then "procedure" else 0 endif
                    -> idprops
    endunless;

    if P_global
    and P_decl /== sysVARS and P_decl /== sysCONSTANT
    then
        mishap(0, 'IDS: INCORRECT DEFINE SYNTAX (not
                        vars or constant after global')
    endif;

    P_decl(W_id, idprops);

    if P_global then P_global(W_id) endif
enddefine;
```

$popdefineconstant$
$popdefineprocedure$

These two variables (both dynamically local to *compile*) are used only by the standard procedure for $pop11\_define\_declare$ shown above. As can be seen from that procedure, $popdefineconstant$ controls the default constant/variable declaration for the identifier being defined when no explicit declaration is specified, while $popdefineprocedure$ controls whether the identifier is declared procedure-type or not.

$pop11\_define\_props(W_{id}, W_P, b_{upd}) \to PROPS$
The procedure in this variable is called by the $define$ statement to return the default *pdprops* value for the procedure being defined. Note that this is the *default* value, and will be overridden by an explicit $with\_props$ declaration.

The arguments are:

W_{id} : The name of the identifier the procedure resides in.

W_P : The name of the procedure, i.e. the W_{id} with any section pathname removed.

b_{upd}: $true$ if "$updaterof$" was present, $false$ if not.

The default value for this procedure is

```
define vars pop11_define_props(W_id, p_name, upd) -> props;
    lvars W_id, p_name, upd, props;
    ;;; default is to return procedure name as props
    p_name -> props
enddefine;
```

$pop11\_loop\_start(Lab)$
This procedure adds the label Lab to the list of loop iteration labels referenced by $nextloop$, $nextif$ and $nextunless$. Lab will normally be a label produced by $sysNEW\_LABEL$.

$pop11\_loop\_end(Lab)$
This procedure adds the label Lab to the list of loop end labels referenced by $quitloop$, $quitif$ and $quitunless$.

$pop11\_forloop\_test(n, i, i_{inc}) \to b$
This procedure returns the result of testing i to be greater than or less than n, depending on whether i_{inc} is positive or negative repectively. It is used by the for statement for testing the end condition in an iteration over i, where i_{inc} is the positive or negative increment and n is the limit value.

15.10 More On Expression Compilation

This section describes the procedure $pop11\_comp\_prec\_expr$ in more detail, and should be read in conjunction with section 15.7 above.

Essentially, the procedure has two phases: phase 1 reads and examines the first object of the expression, while phase 2 loops around absorbing operator expressions until an operator is found whose precedence is too high to form part of the current expression (or the next object is not an operator). Phase 1 is responsible for recognising syntax words and calling their associated procedures, while phase 2 does the same for syntax operators.

Although the procedure emits POP VM code directly (i.e. without constructing an intervening parse tree), this is effected via a 1-instruction 'buffer' to allow possible re-interpretation of a preceding instruction. The buffer is represented by the two variables *pop_expr_inst* and *pop_expr_item*, which contain respectively an instruction-planting procedure, such as $sysPUSH$, and an argument value to be given to it. Two 'dummy' code-planting procedures, $pop11\_EMPTY$ and $pop11\_FLUSHED$, are used in this context to indicate that the buffer is empty, and also to distinguish whether this is because no code has yet been compiled ($pop11\_EMPTY$), or because code has been planted but flushed ($pop11\_FLUSHED$); in these cases the value of *pop_expr_item* is irrelevant. Note that both *pop_expr_inst* and *pop_expr_item* are dynamically local to *pop11_comp_prec_expr*, so that nested expressions have no interaction with each other in respect of buffered instructions.

Phase 1 thus commences by setting *pop_expr_inst* to $pop11\_EMPTY$. If O is the next object on *proglist* and is one of the following, then O is removed from *proglist* and the corresponding actions performed:

- O is a non-operator syntax word with procedure value P, not equal to *popclosebracket*:

```
P();                            ;;; Call syntax procedure
if pop_expr_inst == pop11_EMPTY then
    pop11_FLUSHED -> pop_expr_inst  ;;; see below
endif;
```

- O is any other non-syntax, non-operator word:

```
O -> pop_expr_item;
sysPUSH -> pop_expr_inst;
```

- O is any non-word except *termin*:

```
O -> pop_expr_item;
sysPUSHQ -> pop_expr_inst;
```

Phase 2, to which all other cases go directly without changing the next object, then loops while the next object O is an operator word whose internally-represented precedence OP_PREC is less than the current expression precedence given by *pop_expr_prec* (see the discussion of internal precedence values in section 15.7). For each such operator, the word is removed from *proglist* and the following actions taken:

- O is a syntax operator word with procedure value P:

```
P();                        ;;; Call syntax op procedure
if pop_expr_inst == pop11_EMPTY then
    pop11_FLUSHED -> pop_expr_inst  ;;; see below
endif;
```

- O is an ordinary operator word:

```
pop_expr_inst(pop_expr_item);       ;;; flush buffer
$O$ -> pop_expr_item;
sysCALL -> pop_expr_inst;
pop11_comp_prec_expr(OP_PREC || 1, false);
```

Note that for an ordinary operator word, the precedence passed to the recursive call of *pop11_comp_prec_expr* for the following sub-expression is the operator's precedence with bit 0 set. This value corresponds to the absolute value of its signed *identprops* precedence, and causes the sub-expression to include operators of negative (but not positive) *identprops* precedence of the same absolute value, thus making operators with negative precedence associate to the right rather than to the left.

Note also that after calling either syntax procedures in phase 1 or syntax operator procedures in phase 2, *pop_expr_inst* is changed to *pop11_FLUSHED* if it remains *pop11_EMPTY*. This ensures that a subsequent syntax operator can use a test for *pop11_EMPTY* to determine whether it occurred at the beginning of an expression, without forcing every syntax procedure to make this change.

pop11_comp_prec_expr finishes by flushing the instruction buffer in a mode specified by the value of the b_{update} argument stored in *pop_expr_update*; for normal evaluate mode ($false$) this is just

```
pop_expr_inst(pop_expr_item);
```

Update mode (*true*) is interpreted as meaning that the final instruction-planting procedure of the expression should have an updater, which will plant appropriate update-mode code; the absence of one is taken mean that the compiled code was illegal in an update context, and results in the mishap 'IMPERMISSIBLE UPDATE EXPRESSION'. Otherwise, the buffer is flushed by applying the updater:

```
updater(pop_expr_inst)(pop_expr_item);
```

So that they work correctly with this scheme, all update-mode POP VM instructions are the updaters of the corresponding normal-mode procedures, e.g. $sysPOP$ is the updater of $sysPUSH$, $sysUCALL$ of $sysCALL$, etc.

$pop11\_EMPTY(O_{dummy})$
$\rightarrow pop11\_EMPTY(O_{dummy})$

$pop11\_FLUSHED(O_{dummy})$

$\rightarrow pop11\_FLUSHED(O_{dummy})$

Both these procedures do nothing but erase their dummy argument. This means that whenever one of them is the value of *pop_expr_inst*, it is always safe to do *pop_expr_inst*(*pop_expr_item*) to flush the instruction buffer. Their updaters are different, however: on the basis that an arbitrary piece of already-flushed code has no update-mode interpretation, $\rightarrow pop11\_FLUSHED$ produces the mishap 'IMPERMISSIBLE UPDATE EXPRESSION'. On the other hand, $\rightarrow pop11\_EMPTY$ does allow that an empty expression can be meaningful in this context: if the (necessarily not $false$) value of *pop_expr_update* is in fact a procedure, then this is run without arguments, otherwise no action is taken. This facility is used, for example, by the $\rightarrow$ and $->>$ syntax operators to interpret an empty assignment destination as a stack erase, by calling *pop11_comp_prec_expr* to compile the following expression with an b_{update} argument of $sysERASE(\%0\%)$.

15.11 Example Definitions of Syntax/Syntax Operator Words

This section gives two simple examples of defining new syntactic constructs. The first is the syntax word ", for pushing a quoted word:

```
define syntax ";
    unless isword(readitem() ->> pop_expr_item) then
        mishap(pop_expr_item, 1, 'IQW: INCORRECT QUOTED WORD')
    endunless;
    pop11_need_nextitem(""") -> ;
    sysPUSHQ -> pop_expr_inst
enddefine;
```

The second is a simplified version of the "(" syntax operator, illustrating how push instructions, etc generated in phase 1 of *pop11_comp_prec_expr* are re-interpreted in phase 2 as procedure calls. It also demonstrates how a syntax operator, by testing for *pop11_EMPTY*, can behave differently depending on whether it opens an expression or not:

```
define syntax -1 (;
    dlocal popclosebracket = ")";

    if pop_expr_inst == pop11_EMPTY then
        ;;; starts expression -- just compile parenthesised stmnt seq
        pop11_comp_stmnt_seq();
        pop11_FLUSHED -> pop_expr_inst
    elseif pop_expr_inst == sysPUSH then
        ;;; re-interpret push as call of identifier
        pop11_comp_expr_seq();
        sysCALL -> pop_expr_inst
    elseif pop_expr_inst == sysPUSHQ then
        ;;; re-interpret as call of quoted structure
        pop11_comp_expr_seq();
        sysCALLQ -> pop_expr_inst
    else
```

```
        ;;; call of whatever's on the stack
        pop_expr_inst(pop_expr_item);           ;;; flush it
        sysPOP(sysNEW_LVAR() ->> pop_expr_item); ;;; save in temp lvar
        pop11_comp_expr_seq();
        sysCALL -> pop_expr_inst
    endif;
    ;;; check for closing bracket
    pop11_need_nextitem(")") ->
enddefine;
```

15.12 Obsolete Forms

The following table lists names for the compiler procedures which are now obsolete, but may be found in library programs.

```
sysloop          pop11_loop_start
systxcomp        pop11_comp_expr_to
systxsqcomp      pop11_comp_stmnt_seq_to
sysnlabel        sysNEW_LABEL
sysnvariable      ----
```

Chapter 16

The Virtual Machine

NOTES
How about calling it the PVM = POP VM.
pop_optimise - seems a bit makeshift
pop_as_mode - what do we say about this?

lib foreach uses a lot of obsolete procedure names. I have listed them in a table against their new counterparts, but am concerned about the status of sysnvariable. In general this library file looks a bit out of date.

16.1 Introduction to the VM

This Chapter describes the POP virtual machine (VM) in detail. As well as being used for the implementation of POP-11, the VM has also been used, within POPLOG, as the basis of implementations of Common Lisp[?], Prolog[?] and ML[?]. As a consequence, the perspective of the chapter encompasses the implementation of various languages.

You will recall from previous references to the VM, especially in Chapter 2 that POP programs are translated into operations on a stack: these are the operations of the VM. Subsequently the VM operations may be translated into actual machine code — this is done in the POPLOG system. Translating into machine code could be simple were the target machine to provide suitable operations on a stack. However few machines can be optimally programmed using just stacking operations, and so the process of producing machine code to correspond to a particular sequence of VM instructions will involve a certain amount of optimisation. How VM code is translated into machine code is *not* treated in this chapter.

The standard way of creating VM code is to make use of a set of POP procedures. Some of these procedures *plant* an instruction to perform an operation on the stack. For example $sysPUSH(W)$ plants code to push the value of the identifier associated with the word W on the stack, and $sysCALL(W)$ plants code to execute the value of the identifier associated with the word W, $sysPUSHQ(O)$ plants code to push any object O on the stack. Thus the POP sequence $f(x,2) \rightarrow z$; is translated into VM code by executing:

```
sysPUSH("x");
sysPUSHQ(2);
sysCALL("f");
sysPOP("z");
```

Note that this sequence of procedure calls is made by the POP compiler, and results in the creation of code, which will then be obeyed when the procedure of which the POP sequence $f(x,2) \rightarrow z$ is part is called.

You can make the POPLOG system print out the code it is generating in this case by loading *lib showcode*; and setting the variable *pop_show_code* to *true*. You will get a print out for the above sequence as follows:

```
    PUSH    x
    PUSHQ   2
    CALL    f
    POP     z
```

As well as procedures like those above which tell the VM to plant code to execute a POP expression sequence, there are procedures which are used to tell the VM to start compiling a new procedure, or to finish up the procedure it has been compiling, or to allocate space for variables. If you simply want to use the VM for writing macros or syntax procedures you may not need to know all of these.

These procedures are described in section 16.3, under the following headings:

- 16.3.1 Compiler Control Instructions and Variables: these are procedures which tell the VM to start compiling a new procedure, finish compiling the current one, etc. The variables affect certain aspects of the operation of the compiler.
- 16.3.2 Identifier Declarations: these procedures are used to declare variables, both permanent and lexical, and to give them the attributes discussed in Chapter 5 and Chapter 2.6.1.
- 16.3.3 Compile-Time Assignment: this treats the creation of procedure identifiers and their initialisation, as required by a procedure definition. (See Chapter**??**).
- 16.3.4 Dynamic Local Expressions: this is used in providing the kind of local variables and expressions discussed in Chapter 2.6.1.
- 16.3.5 User Stack Manipulation: these are procedures which plant the code which pushes variables and constants on the stack, pops them off, etc..
- 16.3.6 Procedure Calling: these are procedures which plant the code to call procedures and their $updaters$, including constant procedures, and procedure expressions.
- 16.3.7 Labels and Jump Instructions: these are procedures which plant code used in implementing iterations, conditionals and *goto* instructions, which are described in Chapter 2.12.
- 16.3.8 Accessing/Updating Data Objects: these are procedures which plant *in-line, non type-checking* code to access fields of data-objects.

The chapter ends with a sections which describe important aspects of the operation of the VM in more detail, and with an example of its use.

- 16.5: this section describes in general how lexically scoped variables will actually be implemented in the final output code.

- 16.6: this section discusses the implemention of jumps out of the procedure in which the jump command occurred.

- More On Dynamic Local Expressions: this section discusses the how expressions which are saved on entry to and restored on exit from a procedure are treated, especially with reference to abnormal exits from a procedure.

- An example of the use of the VM code planting instructions: this section contains the POP implementation of the $foreach$ syntax command.

- A history of the POP VM: this section describes how the POP VM has evolved since the 1960's

In order to implement the Prolog language, the POP VM has been extended. These extensions are described in Chapter 26.6.

16.2 What the VM assembler actually does.

Compilation of user procedures in POP is effected by constructing a *list* of POP Virtual Machine (VM) instructions from the source code, using the code-planting procedures described in section 16.3. When this list is complete for a single user procedure, it is assembled machine code in a POP procedure record. This happens whenever $sysENDPROCEDURE$ is called. Languages other than POP, which are implemented using the POP VM are treated in the same way. For example, the POPLOG Prolog implementation creates a procedure record for each arity of a given predicate name.

The procedures which plant each type of VM instruction are given in section 16.3. Compilers which use these procedures should always commence code-planting for a new file or input stream by calling $sysCOMPILE$, described in section 16.3.1. Note that planting of VM code can be 'turned off' by setting the variable *pop_syntax_only* to $true$.

16.3 The VM procedures available to the user

This section describes all of the procedures the user has available to plant VM code, and tell it to create procedures and declare variables.

16.3.1 Compiler Control Instructions and Variables

$sysCOMPILE(P_{compile})$
This procedure should be called when compilation of a new file or input stream is to begin. Its action is to re-initialise the VM context to execute level, setting *popexecute* (see section 16.3.1) to $true$, and to apply the procedure $P_{compile}$, which should be the user's compilation routine, i.e. the thing which will actually compile code and plant VM instructions. In general $P_{compile}$ will be a closure which has the file frozen into it (see 2.25.1). The virtual machine compiler maintains a number of global variables which record the current code-planting context. These include the variable *popexecute*, the stack of procedures being constructed, the lexical variable context, etc. The procedure $sysCOMPILE$ has all these variables as locals. By applying $P_{compile}$ through $sysCOMPILE$, nested compilation streams are thus properly distinguished, and exiting through the call of $sysCOMPILE$ will recover the previous context.

$sysEXECUTE()$
This procedure executes any instructions currently planted at execute level, i.e. when not inside any procedure and *popexecute* is $true$. Recall that a call of $sysCOMPILE$ establishes a new execute level. N.B. If there are any lexical constants which are still waiting for an assignment with $sysPASSIGN$, the code will not actually be executed until these assignments are completed.

$sysPROCEDURE(O_{props}, n_{args})$
This procedure starts code generation for a new procedure. It stacks up the code list for any procedure currently being compiled, to be restored after a $sysENDPROCEDURE$. The variable *popexecute* is set $false$. When the procedure record is produced, it will have the

object O_{props} as its *pdprops* and the integer n_{args} as its $pdnargs$.

$sysENDPROCEDURE() \to P'$

This procedure causes the procedure for which code is currently being generated to be produced as a procedure object P', with *pdprops* and $pdnargs$ as specified by the arguments to $sysPROCEDURE$. P' is returned as result, and the code list stacked up by the last $sysPROCEDURE$ is restored to be the current codelist. *popexecute* is restored to its value before the previous $sysPROCEDURE$.

Note that P' will be an actual executable procedure record *only* in the case where the procedure does not use any non-top-level lexical variables non-locally, and does not employ any non-local jumps. Otherwise it will be a ‘procedure compilation object’ which must be passed to $sysPASSIGN$, $sysPUSHQ$ or $sysCALLQ$, detailed later in this section.

$sysEXEC\_OPTION\_COMPILE(P_{compile})$

This procedure runs the compiler procedure $P_{compile}$, of no arguments, with option of executing the code planted: I.e. it does $P_{compile}() \to O$, and if $O \neq false$, any code which $P_{compile}$ planted is detached from the procedure being currently compiled and immediately executed.

popexecute

This boolean variable is $true$ when the current invocation of the VM through $sysCOMPILE$ is at execute level, and $false$ when code is being planted inside a procedure.

pop_syntax_only

If this variable is $true$ planting of VM code is suppressed. That is to say, code planting procedures do nothing, and $sysENDPROCEDURE$ merely returns $identfn$. Declarations, e.g. as performed by $sysSYNTAX$, are still effective, although these too can be turned off by making *pop_syntax_only* have an integer as its value. Not surprisingly, the default value of this variable is $false$.

pop_vm_flags

Flag bits in this integer variable control certain aspects of the virtual machine's operation. These are defined in

$$UNIX : pop/lib/include/vm\_flags.ph$$
$$VMS : [pop.lib.include]vm\_flags.ph.$$

pop_optimise

When $true$, this boolean variable causes the VM compiler to make extra optimisations in the code for compiled procedures. This will slow down compilation a little. This variable will eventually be replaced by a set of compiler optimisation options: for now, set it $true$ when you want your code to be as fast as possible Note that this variable is local to $sysCOMPILE$, and so has to be set $true$ inside each call of that procedure.

pop_pas_mode

This variable is used by the system compiler POPAS. For system use only.

16.3.2 Identifier Declarations

$sysSYNTAX(W, O_{idprops}, b_{const})$

This procedure declares the word W to be a permanent identifier with $identprops(W) = O_{idprops}$ in the current section. If the boolean b_{const} is $true$, the word is made a constant, otherwise not. However the treatment of constants is affected by the variable *popconstants* described in section 16.3.2. Permissible values for $O_{idprops}$ are those given in Chapter 5.3.5, for the result of the procedure *identprops*, except that $"undef"$ is not allowed.

To declare an active variable, $O_{idprops}$ may also be a pair whose front is any of the above identprops values, and whose back is the active multiplicity. For an ordinary untyped variable ($identprops(W) = 0$), the multiplicity may be an integer in the range 0–255; for any other $identprops(W)$, it must be 1. The b_{const} argument in this case refers to the constancy or otherwise of the *nonactive* procedure value of the identifier — its active value(s) are always variable.

If W was previously undefined, its $valof$ is initialised to a newly-created *undef record* whose $undefword$ is W.

$sysGLOBAL(W)$
This procedure marks the permanent identifier currently associated with W as global, that is, to be automatically imported into any subsection of a section in which it is accessible. Note that the term *global* in this sense is slightly misleading since so marking an identifier does *not* imply that it is exported to top or any other level, merely that it will 'sink' down the section tree from the highest level at which it is accessible. See Chapter 13.

$popconstants$
If this boolean variable is $true$, as it is by default, then $sysSYNTAX$ actually produces permanent constants, but if $false$ then all constant declarations are treated as variables. Note that this variable does *not* affect lexical constants.

$sysVARS(W, O_{idprops})$
This procedure is used by the POP-11 $vars$ statement, this procedure at execute level is the same as $sysSYNTAX(W, O_{idprops}, false)$.

When not at execute level, it should produce a *mishap*. However, for compatibility with the old-style use of $vars$ statements inside POP-11 procedures to simultaneously declare and dynamically localise a permanent variable, its effect is roughly similar to

$$sysSYNTAX(W, O_{idprops}, false);$$
$$sysLOCAL(W)$$

although it is not guaranteed to work properly in all situations particularly with active variables. You are not advised to use it in this way: if you wish to declare and/or dynamically localise a permanent variable, use $sysSYNTAX$ and/or $sysLOCAL$.

$sysCONSTANT(W, O_{idprops})$
Used by the POP-11 *constant* statement, the call of this procedure has the same effect as

$$sysSYNTAX(W, O_{idprops}, true)$$

but produces the mishap 'CONSTANT DECLARATION INSIDE PROCEDURE' if not at execute level.

$sysLVARS(W, O_{idprops})$
$sysDLVARS(W, O_{idprops})$

These procedures declare the word W as a lexically-scoped variable, either local to the current compilation stream if at execute level, or local to the procedure for which code is currently being planted if not. For the duration of the compilation stream/procedure, including all enclosed procedures, any permanent declaration for W is overidden. The $sysCOMPILE$ or $sysENDPROCEDURE$ which terminates the scope cancels the declaration and no further reference to the variable is possible. W must not have been declared as a dynamic local of the current procedure. The $O_{idprops}$ argument is the same as for $sysSYNTAX$, but with the restriction that procedure-local syntax, syntax operator and macro declarations are not allowed, i.e. these are permissible only at execute level. This is because procedure local variables cannot take compile-time values, and therefore don't make sense as syntax words or macros. The restriction does not apply to sysLCONSTANT. Top-level lexical variables, that is those declared at execute level, are initialised to a fixed *undef record* whose $undefword$ is $false$, and which prints as $< undef >$. Procedure-local variables, on the other hand, are *not* initialised: on entry to a procedure, their values are undefined.

Full lexical scoping is supported; you can reference W anywhere inside the current procedure or non-locally inside a nested one. See the discussion below in section 16.5, which also explains the difference between $lvars$ and $dlvars$

The flag bit $VM\_MIX\_NONLOCAL\_AND\_LOCAL\_LEX$ in *pop_vm_flags* controls whether a procedure is allowed to first access a lexical identifier W while it is non-local, and then redeclare it as local. If set then this allowed, and subsequent references to W will get the local value; otherwise, attempting to do this will cause a mishap.

$sysDLVARS$ is identical to $sysLVARS$, except that the most general mechanism for implementing non-local access to the lexical variable W is disabled. See section 16.5.

$sysLCONSTANT(W, O_{idprops})$
This procedure declares the word W to be a lexically-scoped constant, either local to the current compilation stream if at execute level, or local to the procedure for which code is currently being planted if not. The scope of a lexical constant is the same as for a lexical variable; the $O_{idprops}$ argument is as for $sysSYNTAX$, with no restrictions.

A lexical constant is assigned a value with $sysPASSIGN$. This must happen somewhere within its scope, but need not be done before referencing it, perhaps in a call of $sysPUSH$, $sysCALL$, etc.

Note that, in contrast to permanent constants, it is perfectly permissible to have nested definitions for lexical constants, although at each level only one definition of each lexical constant is allowed.

$sysNEW\_LVAR() \rightarrow W$
This procedure is used to generate temporary working local variables. W is a word, declared as a lexical local of the current procedure, which is guaranteed to be different from any other used so far in that procedure *or* in any enclosing procedures.

pop_new_lvar_list
This is a dynamic list that generates the words produced by $sysNEW\_LVAR$. Making it dynamically local to a procedure that calls $sysNEW\_LVAR$ and plants code will mean that on exit from the procedure, any temporary variables generated will be freed for re-use.

$sys\_current\_ident(W) \rightarrow Id$
Given a word W,this procedure returns the (lexical/permanent) identifier record currently associated with W, or $false$ if W is not declared. The identifier returned is the compile-time record, i.e. for procedure-local lexical variables it will be a *lextoken* identifier and not necessarily the same as the run-time identifier that would be produced by $sysIDENT$. For further information on identifiers, see Chapter 5.3.

$sysdeclare(W)$
Those code-planting procedures which take as argument a word declared as an identifier (e.g.

sysPUSH) will call this procedure if given an undeclared W. *sysdeclare* is then expected to declare W appropriately, otherwise a mishap results. The default value of this procedure is approximately as follows:

```
define vars sysdeclare(word);
   lvars word;
   sys_autoload(word) -> ;
   if identprops(word) == "undef" then
       sysSYNTAX(word, 0, false);
       prwarning(word)
   endif
enddefine;
```

I.e. first try to autoload W as described in Chapter 14.3, then if this doesn't declare W as a permanent identifier, declare W as a permanent variable and call the *prwarning* procedure, described in Chapter 4.5.

To make *sysdeclare* is user assignable, it *sysunprotect* should be used as described in Chapter 5.3.7.

16.3.3 Compile-Time Assignment

$sysPASSIGN(P, W)$
$\rightarrow sysPASSIGN(P, W)$

This procedure 'assigns' P to be the value of the identifier represented by the word W, where P is a procedure or a 'compilation object' produced by *sysENDPROCEDURE*. The assignment is done in the following way:

1. If W is declared as a lexical constant, then P is simply assigned to it. In this case only, P may be any object at all, not just a procedure. W must not have been assigned a value previously.

2. If *popexecute* is $false$, i.e. if a procedure is currently being compiled, then code is planted to pop the procedure into the variable at run-time, i.e.

$$sysPUSHQ(P), sysPOP(W);$$

3. Otherwise, P is simply assigned directly to the value of W, unless the current value of W is a procedure P_2, in which case:
   (a) If P_2 has an *updater*, and P has not, then the *updater* of P_2 is assigned to be the *updater* of P;
   (b) If P_2 is a closure of *systrace*, i.e. P_2 is a traced procedure, then the appropriate field of the closure is updated with P, thus ensuring that the procedure continues to be traced.

The *updater* of this procedure is $sysUPASSIGN$; note that in both $sysPASSIGN$ and $sysUPASSIGN$ the current value of the identifier is accessed/updated using $idval$, meaning that if W is an active variable the value will be got or sent through calls of its *nonactive* procedure value. To reference the *nonactive* value directly, the second argument to these procedures may be a '*nonactive* pair', as described under $sysPUSH$ below.

$sysUPASSIGN(P, W)$

This procedure 'assigns' P to be the *updater* of the value of the identifier associated with W, as follows:

1. If W is declared as a lexical constant, then it must have had a procedure, or a 'compilation object' resulting from $sysENDPROCEDURE$, already assigned to it with $sysPASSIGN$. P is simply made the *updater* of this.

2. If W is not a lexical constant, then, unless *popexecute* is $true$, the following code is generated:

```
sysPUSHQ(P), sysPUSH(W);
sysUCALL("updater");
```

Note that this will only produce a proper local *updater* if W is local to the current procedure, and has had a local procedure assigned to it — otherwise the *updater* will be changed non-locally!

Otherwise, if the current value of W is a procedure P_2 then P is assigned to the *updater* of P_2, unless P_2 is traced (i.e. a closure of *systrace*), in which case P replaces the previously traced *updater* of P_2. If the current value of W is not a procedure, then a procedure which will produce the mishap 'ONLY UPDATER DEFINED' when applied is made its value, and P is made the *updater* of this.

In both *sysPASSIGN* and its *updater* the current value of the identifier is accessed/updated using *idval*, meaning that if W is an active variable the value will be got or sent through calls of its *nonactive* procedure value. To reference the *nonactive* value directly, the second argument to these procedures may be a '*nonactive* pair', as described under *sysPUSH* below.

16.3.4 Dynamic Local Expressions

$sysLOCAL(P_{code})$
$sysLOCAL(W)$

This procedure enables the value, or values, produced by an arbitrary expression to be made dynamically local to the procedure for which code is currently being planted, as described in Chapter 2.6.1. 'Dynamically local' in this context means that the value(s) of the expression will be saved on each entry to the procedure and the saved value(s) restored on each exit. This applies in all contexts, including abnormal exits from procedures and the suspension and resumption of processes created with *consproc*. The expression must have an access part and an update part, the two parts being represented respectively by the base and *updater* of the code-planting procedure argument P_{code}. That is, $P_{code}()$ is assumed to plant VM code to access the value(s) and push them on the stack, while $\rightarrow P_{code}()$ is assumed to plant code to update them by popping an equal number of objects off the stack. The number m of values produced/updated is called the *multiplicity* of the expression, and is specified by the *pdprops* of P_{code} (an integer in the range 0–255). Making the values dynamically local is thus achieved by incorporating the access code and update code at appropriate points in the procedure, and by allocating m local lvars in which to save the values. Effectively this can be thought of as

$$< accesscode > \rightarrow save_m \rightarrow save_{m-1} \rightarrow \ldots \rightarrow save_1$$

on entry to the procedure, and

$$save_1, save_2, \ldots, save_m \rightarrow < updatecode >$$

on exit. This is treated in more detail in section 2.6.1 below.

Calling $sysLOCAL$ with a word argument W (or '*nonactive* pair') is functionally identical to

$$sysLOCAL(sysPUSH(\%W\%))$$

with the *pdprops* of the $sysPUSH$ closure set to 1 (or, for an active variable, to its multiplicity). That is, the expression is just the value of the identifier currently associated with W. However, since the dynamic localising of simple non-active identifiers is handled specially, $sysLOCAL(W)$ produces much more efficient code than the above.

dlocal_context
dlocal_process

These are two *active* variables which may only be used directly inside the code of a dynamic local expression. They are described in section 16.3.4.

16.3.5 User Stack Manipulation

$sysPUSH(W)$
$\rightarrow sysPUSH(W)$

This procedure plants code to push the value of the (lexical/permanent) identifier associated with the word W onto the stack. Note that this is optimised to a $sysPUSHQ$ for the value of an initialised constant.

If W is an active variable then this instruction is equivalent to $sysCALL(W)$ for the *nonactive* value of W. To access the *nonactive* value of an active variable, the argument to this procedure may alternatively be a '*nonactive* pair', i.e. a pair of the form

```
conspair(W, "nonactive")
```

The *updater* of this procedure is $sysPOP$.

$sysPOP(W)$
This procedure plants code to pop the top object from the stack and store it in the value of the identifier associated with the word W. If W is an active variable then this instruction is equivalent to $sysUCALL(W)$ for the *nonactive* value of W. To update the *nonactive* value of an active variable, the argument to this procedure may alternatively be a *nonactive* pair, as above.

$sysIDENT(W)$
This procedure plants code to push the run-time identifier record for W onto the stack. You should see the note under *sys_current_ident* in section 16.3.2 and also section 16.5.

$sysPUSHQ(O)$
This procedure plants code to push the object O onto the stack.

$sysPUSHS(O_{dummy})$
This procedure plants code to push the top object onto the stack onto the stack again, i.e duplicate it. This procedure is given a dummy argument O_{dummy}.

$sysERASE(O_{dummy})$
This procedure plants code to erase the top object on the stack. This procedure is given a dummy argument O_{dummy}.

$sysSWAP(n, m)$
This procedure plants code to swap the objects at the n-th and m-th positions on the user stack. The indexing of the stack for this purpose starts with the top object on the stack having index 1.

16.3.6 Procedure Calling

$sysCALL(W)$
$\rightarrow sysCALL(W)$

This procedure plants code to execute the value of the (lexical/permanent) identifier associated with the word W. Note that this is optimised to a $sysCALLQ$ for the value of an initialised constant. If W is an active variable then this instruction is equivalent to $sysCALL(W)$ on the *nonactive* value of W, followed by $sysCALLS$. To call the *nonactive* value of an active variable, the argument to this procedure may be a *nonactive* pair, as for $sysPUSH$. The *updater* of this procedure is $sysUCALL$.

$sysUCALL(W)$
This procedure plants code to execute the *updater* of the value of identifier. It is optimised to $\rightarrow sysCALLQ$ for an initialised constant. If W is an active variable then this instruction is equivalent to $sysCALL(W)$ on the *nonactive* value, followed by $sysUCALLS$. To *updater*-call the *nonactive* value of an active variable W, the argument to this procedure may be a *nonactive* pair as for $sysPUSH$.

$sysCALLQ(O)$
$\rightarrow sysCALLQ(O)$

This procedure plants code to execute the object O. The *updater* of this procedure is $sysUCALLQ$.

$sysUCALLQ(O)$
This procedure plants code to execute the *updater* of O, or execute O in update mode if it is a non-procedure — i.e. call the *updater* of its *class_apply*, etc.

$sysCALLS(O_{dummy})$
$\rightarrow sysCALLS(O_{dummy})$

This procedure plants code to pop and execute the object on top of the stack; the procedure takes a dummy argument O_{dummy}. The *updater* of this procedure is $sysUCALLS$.

$sysUCALLS(O_{dummy})$
This procedure plants code to execute the *updater* of the object, or execute the object in update mode, etc; the procedure takes a dummy argument O_{dummy}.

16.3.7 Labels and Jump Instructions

A label is an object used for referencing a position within the stream of VM code, and which can be specified as the target of jump instructions like *goto*, *ifso*, *and*, etc; a label is attached at a given position in the code by calling $sysLABEL$ (or $sysDLABEL$) at that point.

A jump instruction may reference a label anywhere in the procedure for which code is currently being planted (a local jump), or anywhere in an outer, enclosing procedure (a non-local jump). In the latter case, the effect is similar to a call of *exitto* for the target procedure occurring at the point of the jump, followed by a local jump to the label. The implementation of non-local jumps is described in section 16.6.

Labels come in two kinds, symbolic and absolute. Symbolic labels are chosen by the user, and are usually words, although they can in fact be any POP objects except pairs; this type of label is lexically-scoped, meaning that a jump to a label Lab will reference the innermost occurence of Lab in the whole nest of procedures currently being compiled. Equality of labels is tested with ==. Absolute labels, on the other hand, are unique objects (pairs) generated by $sysNEW\_LABEL$, and are not scoped at all. Jump instructions that refer to an absolute label go to wherever that label is planted, be it in the current procedure or an enclosing one.

$sysNEW\_LABEL() \rightarrow Lab$

This procedure generates a new absolute label Lab. This can be planted with $sysLABEL$ or $sysDLABEL$ at any point in the code of a procedure, and referenced as the target of a jump instruction.

$sysLABEL(Lab)$
$sysDLABEL(Lab)$

This procedure sets the label Lab to reference the next instruction planted, i.e. that instruction will be the target of a *goto*, $ifso$, $ifnot$, *and*, *or* or *go_on* instruction using lab, as described above. $sysDLABEL$ is a special variant of $sysLABEL$ that allows optimisation of non-local jumps – see section 16.6 below.

$sysGOTO(Lab)$

This procedure plants a instruction to jump unconditionally to the label lab, as e.g. required by the POP *goto* command.

$sysIFSO(Lab)$

This procedure plants an $ifso$ instruction: this will jump to the label Lab if the top object on the stack is not $false$, removing it from the stack whether the jump is taken or not. See $sysLABEL$ above for a description of labels.

$sysIFNOT(Lab)$

This procedure plants an $ifnot$ instruction which will jump to the label Lab if the top object on the stack is $false$, removing it from the stack whether the jump is taken or not.

$sysAND(Lab)$
This procedure plants an *and* instruction which will

- jump to the label Lab if the top object on the stack is $false$, leaving the object on the stack;
- remove the object from the stack otherwise.

$sysOR(Lab)$
This procedure plants an *or* instruction which will

- go to the label Lab if the top object on the stack is not $false$, leaving the object on the stack;
- remove the object from the stack otherwise.

$sysGO\_ON(L_{lab}, Lab_{else})$
This procedure plants a *go_on* instruction which will

- Take an integer i off the stack;
- If $1 \leq i \leq length(L_{lab})$, then jump to the i-th label in the label list L_{lab};
- If i is not within range and the *else* label Lab_{else} is $false$ then a mishap results, otherwise a jump to Lab_{else} is taken.

A mishap will also result if i is not an integer.

16.3.8 Accessing/Updating Data objects

The VM provides two instructions for referencing a specific field of an object of specific type. These are:

$sysFIELD\_VAL(i_{field}, O_{spec})$
This instruction assumes that, at run-time, there is an object of type O_{spec} on top of the user stack. O_{spec} may be any record or vector type key object or key specification. $sysFIELD\_VAL$ plants inline, non type-checking code to push the component of the object specified by i_{field} onto the user stack. E.g. the sequence of instructions

```
sysPUSH("list");
sysFIELD_VAL(2, [full full])
```

is equivalent to applying $fast\_back$ to $list$.

$sysUFIELD\_VAL(i_{field}, O_{spec})$
This instruction assumes that, at run-time, there is an object of type O_{spec} on top of the user stack. O_{spec} may be a record or vector type key object or key specification. $sysUFIELD\_VAL$ plants inline, non type-checking code to assign the object second from the top of the stack into the field of the object specified by i_{field}. E.g. the sequence of instructions

```
sysPUSH("object");
sysPUSH("list");
sysUFIELD_VAL(2, [full full]);
```

is equivalent to applying the *updater* of $fast\_back$ to $object$ and $list$.

16.4 Implementation of Procedures

Although there are different types of procedures, all can be structured in such a way that they can be executed by the same protocol, namely 'extract the address of the executable code and transfer control to it'. In addition, all procedures have an *updater* field, which may possibly hold another procedure, to be run when the base procedure is executed in update mode, and a *pdprops* field, which usually contains the procedure name.

This said, there are two basic types of procedure: 'proper' procedures and closures. A proper procedure is one that maintains an environment of local variables, which may be *lexical* or *dynamic* or both, and that creates a stack frame on the call stack. As well as containing local variable values, the stack frame holds the address of the procedure of which this is an invocation — the 'owner' of the frame, and the return address into the caller procedure. Most proper procedures are created by the POP Virtual Machine from code planted by compilers.

A closure is an object which combines a procedure (its *pdpart*) with zero or more argument values (its $frozvals$). The executable code inside a closure loads the frozen values onto the user stack and then executes the *pdpart* procedure, which may in its turn be another closure or a proper procedure. Unlike a proper procedure, a closure does not maintain local variables or a call stack frame. Closures are created by *partapply*.

In addition, special forms of both types of procedure are created by various system facilities. *newanyarray* for example, creates proper procedures that are distinguishable as arrays (see 10), and *newproperty* and *newanyproperty* create closures that represent properties (see Chapter 11.3).

16.5 Implementation of Lexical Scoping

This section describes the way in which lexically-scoped variables declared with $sysLVARS$ will actually be implemented in the final output code. Top-level lexical variables, i.e. those declared at execute level, simply translate to unique permanent identifiers, which become anonymous when the file in which they occur has been compiled. Thus, while being lexical in scope, they are stored in the same way as permanent variables, as discussed in Chapter 5.

On the other hand, a lexical variable W declared inside a procedure P falls into one of three categories, depending on its usage:

- type 1

  Used strictly locally, i.e. W is not referenced non-locally inside any procedure nested within P, and is not pushed as an identifier with $sysIDENT$.

- type 2

  Used non-locally inside a nested procedure P_2, but does not cross a 'push' boundary. By this we mean that neither P_2 nor any other procedure that encloses it inside P is pushed as a data object (see below). Also not pushed as an identifier with $sysIDENT$.

- type 3

  Used non-locally inside a nested procedure, and does cross a 'push' boundary, or directly pushed as an identifier with $sysIDENT$.

A procedure is 'pushed as a data object' if the object returned by $sysENDPROCEDURE$ is given to $sysPUSHQ$, or $sysPASSIGN$'ed to any identifier other than a lexical constant. If assigned to a lexical constant, it is also 'pushed' by any $sysPUSH$ for that constant. In POP-11 terms, this means that the only kind of nested $define$ that does not count directly as a 'push' is $define\ lconstant \ldots$.

Thus for example, in the following POP-11 procedure, the lexical variable w is of type 1, x is of type 2, and y and z are of type 3:

```
define P();
  lvars w x y z;
  w =>                        ;;; w only used locally

    define lconstant P_2();
      x =>                    ;;; x used non-locally
    enddefine;
  P_2();                        ;;; P_2 is called but not pushed
```

```
    define lconstant R();
      define lconstant S();
        y =>                  ;;; y used non-locally
      enddefine;
      S();                    ;;; S is called but not pushed
    enddefine;
  R -> w;                     ;;; but R is pushed

    define vars T();          ;;; T is pushed by being "vars"
      z =>                    ;;; and it uses z non-locally
    enddefine;

enddefine;
```

The different variable types are implemented by the following mechanisms, given in order of decreasing efficiency:

- The first n type-1 variables, in order of declaration, are allocated to machine registers. The number n is implementation dependent, but in all current systems is 2.
- The rest of the type-1's are allocated to stack frame cells.
- Type-2 variables are allocated to unique dynamically local identifiers.
- Type-3's require the run-time construction of identifier records in the heap to hold their values, since a 'pushed' procedure that uses them non-locally (or the identifiers themselves if pushed with $sysIDENT$) can be passed either

  1. up and right out of the current lexical environment, or
  2. down into another invocation of that same environment.

  They use stack frame cells initialised to hold identifiers, these being created on entry to the home procedure and passed down as hidden extra arguments to nested procedures that use them non-locally. Whenever such a procedure is 'pushed', a closure of the base procedure with the run-time identifier arguments is created and in addition, an auxilary variable is generated to hold the closure, ensuring that it is only produced once for each invocation of the environment. This variable itself may come out as any of the three types, depending on whether it is used non-locally or across a push boundary.

In fact, providing neither (a) nor (b) actually happens for a type-3, a unique dynamic variable (as for type-2 variables) would suffice; the VM compiler is unable to recognise this, but the user may be able to in many situations. For this reason the $sysDLVARS$ declaration is provided: this is identical to $sysLVARS$, except that type-3 $sysDLVARS$ are forced to be type-2's.

A typical use of this in POP-11 would be something like the following procedure to count the number of characters printed for an object:

```
define countchars(x) -> n;
    lvars x; dlvars n;

    define vars cucharout();
        -> ;        ;;; erase the character
        n+1 -> n    ;;; increment non-local count
    enddefine;

    0 -> n;
    pr(x)
enddefine;
```

On the other hand, it may be known that an individual 'push' for a particular procedure (or a push of an identifier with $sysIDENT$) does not necessitate the use of type-3 variables. E.g. in

```
applist([1 2 3 4], P)
```

where P is a procedure that uses non-locals, (1) or (2) cannot happen because *applist* is a system procedure that can never do either with P.

Where this sort of information is known to the compiler writer, the flag

$$VM\_DISCOUNT\_LEX\_PUSHES$$

in pop_vm_flags can be employed. While set, $sysPUSH$, $sysPUSHQ$ and $sysIDENT$ instructions that would otherwise cause variables to be marked as type-3 will not do so.

However, any other $sysPUSH$ or $sysPUSHQ$ on the relevant procedure while this flag is *not* set will cause the 'damage' to be done, irrevocably.

In fact, the VM compiler *does* recognise this situation when the pushed procedure is the last argument to one of a small number of system procedures: currently, these are just the 'app' procedures *applist*, *maplist*, *appdata* and *appproperty*. So, the particular example above will not in itself cause the production of type-3 variables.

16.6 Implementation of Non-Local Jumps

A non-local jump from a procedure P_2 to a label in a lexically-enclosing procedure P closely parallels a non-local access to a lexical variable; if we think of the label as being a variable, and the jump instruction as an access to it, then what matters from the point of view of how the jump is implemented is whether the label is 'type-3' or not — that is, whether it crosses a push boundary.

In the case where it does not, the nested procedure P_2 can only be called directly from within P or some other directly-called sub-procedure of P. This means that there cannot be any other intervening call of P between the call of P_2 and the target stack frame; to unwind the call stack to the correct call of P it is therefore sufficient to do $exitto(P)$. Moreover, it can safely be assumed that the target call of P is still extant, obviating the need to run up the call stack to check this first.

When the jump does cross a push boundary this is not sufficient, because P_2 can now have been passed down into other invocations of P (or indeed right out of P altogether), and there there could be any number of intervening calls of P to be erased before reaching the target. A means of identifying different invocations of P is therefore required, and the system does this as follows:

For each procedure P that can be the target of a non-local jump across a push boundary, a unique dynamic variable is generated, and made a local of P; on entry to P, the value of this variable is set to a sequentially-generated integer, which will be unique for each call of P. The identifying integer is then passed down, like a type-3 $lvar$, as an extra, hidden argument to pushed sub-procedures that jump to P, enabling them to identify which call of P to exit to, by inspecting the current value of the dynamic variable. In addition, this type of jump has first to inspect the saved values of the variable on the call stack to check that the target call is still extant. It is therefore slightly slower than the first type, and, as with a type-3 *lvar*, pushing a sub-procedure that does such a jump requires the creation of a closure.

Again as with *lvars*, the user may know that a pushed procedure doing a non-local jump will in practice require only the simple case, i.e. the target will always be extant, and there will be no intervening calls of the same procedure. Analogously to *sysDLVARS*, the VM provides *sysDLABEL* to specify this: non-local jumps to a label planted with *sysDLABEL*, rather than *sysLABEL*, will always use the simple mechanism, and pushing sub-procedures doing the jumps will not create closures. In POP-11, you can use the label syntax

```
<label>:*
```

when *sysDLABEL* is appropriate.

16.7 More On Dynamic Local Expressions

As described under *sysLOCAL*, expression values are made dynamically local to a procedure by incorporating the access code and update code for the expression at appropriate points, and by the allocation of local *lvars* in which to save the values. This section describes the process in detail.

The access code for an expression is actually incorporated at three places in a procedure, and the update code at four places. As previously mentioned, the normal entry code for the procedure contains

$$< accesscode > \rightarrow save_m \rightarrow save_{m-1} \rightarrow \ldots \rightarrow save_1;$$

Here $save_1, \ldots, save_m$ are the local *lvars* allocated to the expression.

The normal exit code contains

$$save_1, save_2, \ldots, save_m \rightarrow < updatecode >;$$

However, separate code sections are used for abnormal exit, (i.e. when the procedure is exited with *chain*, *chainfrom*, *exitfrom*, etc.), and for swapping in and out of the procedure when it forms part of a *consproc* process being resumed or suspended. For abnormal exit, the expression code used is the same as for normal exit, i.e.

$$save_1, save_2, \ldots, save_m \rightarrow< updatecode >;$$

The procedures *uspend* and *resume*, on the other hand, are a little different: in these cases dynamic local values must be exchanged with their saved values. For process suspension, the current value is swapped with the saved value in the procedure stack frame *before* that stack frame is saved in the process record; for resumption the opposite happens, i.e. the values are swapped around *after* restoring the stack frame from the process, thus putting everything back as it was before the process was suspended.

The code generated for each of these contexts is therefore an interleaving of the code for entry and exit, i.e.

$$< accesscode >, save_1, save_2, \ldots, save_m \rightarrow< updatecode >, \rightarrow save_m \rightarrow save_{m-1} \rightarrow \ldots \rightarrow save_1;$$

for *suspend*, and

```
           save_1, save_2, \ldots, save_m,
               <access code> \rightarrow save_m \rightarrow
 save_{m-1} \rightarrow \ldots \rightarrow save_1,
                                     \rightarrow <update code>;
```

for *resume*.

To enable the access or update code to determine in which context it is being called, an integer context value (1–4) is available via the special active variable *dlocal_context*. This variable may be referenced *only* in local expression code, and nowhere else. The contexts and their applicablity are as follows:

value	context	applies to
1	normal entry	access
	normal exit	update
2	abnormal exit	update
3	$suspend$	access
		update
4	$resume$	access
		update

In the case of *suspend* and *resume*, the current process record is also available to the code from the active variable *dlocal_process*, with similar restrictions on its use. The value this returns is defined only for contexts 3 and 4, and is *undefined* for the other two. Using *dlocal_process* enables the access or update code to manipulate the process undergoing suspension or resumption in any desired way. Note that owing to *consprocto*, which uses the *suspend* and *resume* mechanisms to create a process from an existing section of the call stack, the process given by *dlocal_process* is not necessarily the same as *pop_current_process*.

Having dealt with the code generated for individual local expressions, we now discuss the overall handling of a complete set of n expressions in a given procedure, declared with *sysLOCAL* in the order 1 to n:

Basically, expressions are executed in declaration order (1 upto n) for 'ingoing' contexts (normal entry and *resume*), and in the reverse order (n down to 1) for 'outgoing' contexts (normal/abnormal exit and *suspend*). However, a prime constraint is that the restore/update code for any expression must not be run unless the corresponding access/save code has been executed first, since the latter must produce the values to be given to the former; because a procedure can be interrupted during the execution of the access code for a particular expression (or indeed, before any of the access code sections have been started), and because such an interrupt can lead to an abnormal exit or a process *suspend*, it is incumbent on the procedure to provide a mechanism that prevents this constraint being violated.

To this end, the procedure maintains a counter, or index, of expressions run. The index is initialised to 0 immediately on entry and before interrupts can be serviced; thereafter, for each ingoing context, the index is set to i after completing code for the i-th expression, and for each outgoing context to $j-1$ before starting code for the j-th expression. E.g., for normal entry and exit:

```
0 -> index;     ;;; before interrupts

normal entry                 normal exit


<save expr 1>        LabN:  N-1 -> index;
1 -> index;                 <restore expr N>
<save expr 2>               .
2 -> index;                 .
.                           .
.                    Lab2:  1 -> index;
.                           <restore expr 2>
<save expr N>        Lab1:  0 -> index;
N -> index;                 <restore expr 1>
                     Lab0:
```

The abnormal exit code is then similar to that for normal exit, except that it switches to Lab_i, where i is the value of the index; this prevents expressions being restored that have not been saved. Notice also that it prevents a second or subsequent attempt to chain out of the procedure from restoring expressions that have already been restored.

The procedures *suspend* and *resume* are rather more complicated: a *suspend* at a time when the index has value i runs the swap code for expressions i down to 1, reducing the index actually saved in the stack frame to 0, but the corresponding *resume* that will follow it must undo only what the *suspend* did and no more, that is, run the swap code from 1 upto, and finishing at, i. A *suspend* therefore starts by saving the current index in another location: this is then the limit value for a subsequent *resume*. An additional complication here is that both the *suspend* code and the *resume* code can themselves be interrupted by a recursive *suspend*-and-*resume* under certain conditions, and so the code has to be re-entrant to cope with this.

Finally, note that all local expression code is run *inside* the normal local-identifier environment of a procedure, i.e. expression entry code follows the creation of local *lvars* and the saving of (non-active) dynamic local identifiers, and expression exit code precedes the unwinding of this environment. POP-11 users should also note that access code for *dlocal* expressions is run *before* popping procedure formal arguments from the stack, and that update code is run *after* pushing formal result variables, etc.

I've done a lot of work to remove any assumption that the tag bits live in any particular place, i.e. tests are always done in terms of an object being compound or simple (where simple is further sub-divided into simple1 (integer) and simple2 (decimal). There is therefore no direct reference to tag bits anywhere in the *sysPOP* code. (Though doubtless porting it to a word-addressing machine would show up things I've missed in that respect.)

The first word of a procedure record contains a pointer to the start of its code. Apart from anything else, this allows the code of a procedure to be separate from the procedure record per se, although this isn't currently used. A procedure-type variable call thus goes indirect through the first word. Calls of constant procedures have an offset compiled into the code.

16.8 An example of the use of the VM

In this section we will go through a complete example of a POPLOG library file, namely *lib foreach*. This allows you to take a specified action for each member of the POP database which *matches* a certain pattern. The database is described in Chapter **??**. The syntactic form is:

```
foreach <pattern> do <expression_sequence>
endforeach
```

For example, we might say:

```
foreach [brother ?x Elizabeth] do x=>
endforeach
```

which would print out every brother of Elizabeth, assuming her family were entered in the data-base in that form.

The UNIX version of the library file is given. The file is given as it occurs in the POPLOG system, except that we have broken it up to intersperse more description than exists in the program text, and have commented in some line numbers.

It begins with a standard comment, which gives:

- The file-name with a path starting at the environment variable $*usepop*, which is the root of the POP system.
- The purpose of the file, which is to define a *foreach* construction
- The author(s).
- What documentation is available in the POPLOG on-line documentation system.
- Related files, which define similar constructs, or define some of the data-objects used.

```
/*  --- Copyright University of Sussex 1986.  All rights reserved. ---
 |  File:           $usepop/master/C.all/lib/database/foreach.p
 |  Purpose:        performs action on all objects in
 |                  daabase matching pattern.
 |  Author:         A.Sloman S.Hardy 1982 (see revisions)
 |  Documentation:  HELP * FOREACH
 |  Related Files:  LIB * FOREVERY, * DATABASE
 */
```

Next, in the top-level, or root, *section*, the program is defined.

We begin by defining the procedure $trynext$. The *lconstant* construction is used to make the procedure $trynext$ unavailable to the user, except throught the $foreach$ construction. Note that this is *not* how this program would be developed. The *lconstant* construction is inconvenient during debugging because you can't call the procedure independently, or even $trace$ it. And the procedures $fast\_front$ and $fast\_back$ are dangerous, so they should replace $front$ and $back$ only when you are sure that the program is working correctly, and that no user-errors can cause them to be applied to non-pairs.

The procedure takes as input a list, PL whose first member is a pattern, and whose second member is a list of objects that are to be matched to the pattern. The procedure iterates through the list of objects until it finds one that matches the pattern. If it finds none, it returns $false$. If it does find an object that matches the pattern, it returns the object, and modifies the list PL, so that next time it is called it will start looking only at the part of the list of objects it has not yet examined.

```
section ;

;;;  TRYNEXT takes a possibilities list as argument
;;;  It finds the next matching element and returns it, updating
;;;  the list. If none is found it returns FALSE.

define lconstant procedure trynext(PL);
    lvars P, L, PL;
    fast_front(PL) -> P;
    fast_back(PL) -> L;
    until null(L) do
        if fast_front(L) matches P then
            fast_front(L) -> it;
            fast_back(L) -> fast_back(PL);
            return(true)
        endif;
        fast_back(L) -> L;
    enduntil;
    return(false);
enddefine;
```

Having defined the auxiliary procedure, we can now write the main syntax procedure that will be called when the word *foreach* is encountered by the compiler.

```
;;;  FOREACH pattern DO actions ENDFOREACH
;;;
;;;      VARS %X;
;;;      FETCH(pattern) -> %X;
;;;      WHILE TRYNEXT(%X) DO actions ENDWHILE
;;;
global vars syntax endforeach;


lconstant _temp=popconstruct;
true -> popconstruct;       ;;; make lists compile as constants


define global syntax foreach;                  ;;; fe1
    lvars Var Lab Endlab _x;                   ;;; fe2
    sysnvariable() -> Var;                     ;;; fe3
    sysnlabel() -> Lab;    sysloop(Lab);       ;;; fe4
    sysnlabel() -> Endlab; sysloopend(Endlab); ;;; fe5
    sysVARS(Var,0);                            ;;; fe6
    systxcomp([do then in]) -> _x;             ;;; fe7
    if _x == "in" then                         ;;; fe8
        erase(systxcomp([do then]));           ;;; fe9
        sysCALLQ(nonop ::)                     ;;; fe10
    else                                       ;;; fe11
        sysPUSH("database");                   ;;; fe12
        sysCALLQ(nonop ::)                     ;;; fe13
    endif;                                     ;;; fe14
    sysPOP(Var);                               ;;; fe15
    sysLABEL(Lab);                             ;;; fe16
    sysPUSH(Var);                              ;;; fe17
    sysCALLQ(trynext);                         ;;; fe18
    sysIFNOT(Endlab);                          ;;; fe19
    erase(systxsqcomp([endforeach close]));    ;;; fe20
    sysGOTO(Lab);                              ;;; fe21
    sysLABEL(Endlab);                          ;;; fe22
```

```
enddefine;                                         ;;; fe23

_temp -> popconstruct;

endsection;

/* --- Revision History ---------------------------------------
--- Aaron Sloman, Nov  7 1986 lvarsed, desectionised.
*/
```

This syntax procedure works as follows: $fe1$ declares $foreach$ to be a syntax procedure of global scope, i.e. it will be available in every section without having to be imported.

$fe2$ declares lexical variables, which are initialised below.

$fe3$ creates a new anonymous variable for the object code, and assigns it to Var. $sysnvariable$ in fact creates a word, whereas it would be more efficient to use $sysNEW\_LVAR$. This variable will be used to hold the 'list of possibilities'.

$fe4$ creates a label *Lab*, which will label the beginning of the loop we are going to compile, and, using *sysloop*, makes it the label that any *next* command in the iteration construct body will jump to.

$fe5$ creates a label *Endlab*, which will label the instruction after the end of the loop we are going to compile, and, using *sysloopend*, makes it the label that any *quit* command in the iteration construct body will jump to.

$fe6$ makes Var a dynamic local of the procedure in which the $foreach$ construct occurs.

$fe7$ compiles an expression, terminated by one of the words "*do*", "*then*" or "*in*". This expression will be the pattern. *then* is permitted for compatibility with POP2. It will be the top of the stack when the code generated by $fe7$ has just executed.

$fe8 - fe10$ treats the case $foreach < expr > in < expr > do$. It compiles code to generate the second $< expr >$, and then compiles a call to ::, which will form a list whose first member is the pattern and whose second member is a list of objects to match it against.

$fe11 - fe13$ treats the alternative, in which the pattern is consed onto the *database* to create the same sort of list.

$fe15$ stores the list in the anonymous variable held in Var.

$fe16$: Now we are ready to start the loop — we have planted code to generate the pattern and where we are going to look for it. So we plant the label that starts the loop.

$fe17 - 18$ plants code to do $trynext(< Var >)$.

$fe19$ plants code to check whether anything was found, and if not to jump to the label which we will place after the end of the loop.

$fe20$ compiles the body of the iteration, which may end with $endforeach$ or $close$ (for POP-2 compatibility).

$fe21$ plants a jump back to the beginning of the loop.

$fe22$ plants the label which marks the end of the loop.

The example use of $foreach$ given above produces the code below, which is given verbatim, except for being commented (by hand!). Note that while the loop variable prints as ↑↑↑, it does not in fact contain these characters.

```
VARS    ^^^ 0               ;;; Declare the loop variable, ^^^.
PUSH    popstackmark        ;;; Prepare to make a list from stacked
                            ;;; values
PUSHQ   brother             ;;; 1st stacked object is "brother"
PUSHQ   ?                   ;;; 2nd stacked object is "?"
PUSHQ   x                   ;;; 3rd stacked object is "x"
```

```
    PUSHQ   Elizabeth           ;;; 4th stacked object is "Elizabeth"
    CALL    sysconslist         ;;; Make a list of the stacked objects.
                                ;;; this is the pattern.
    PUSH    database            ;;; Push the POP database on the stack
    CALLQ   <procedure ::>      ;;; cons the pattern to the database
    POP     ^^^                 ;;; and store in the loop variable.
label_87                        ;;; The loop starts here
    PUSH    ^^^                 ;;; Push the loop variable
    CALLQ   <procedure trynext> ;;; call the trynext procedure
    IFNOT   label_88            ;;; jump out of loop if no match found
    PUSH    x                   ;;; now do  x =>
    PUSH    false
    CALL    sysprarrow
    GOTO    label_87            ;;; and go back for next iteration.
label_88
```

16.9 History of the POP Virtual Machine

(by R.J.Popplestone)

The POP VM is descended from a real machine, the Eliott 4100 series, designed in the 1960's by a team led by C.A.R Hoare. This machine provided the essentials — it had stacking and unstacking instructions, and indirect sub-routine calls.

```
   MVE  Address     Push the argument on the stack
   MVB  Address     Pop the stack to the argument
   JIL  Address     Jump to the contents of the address, saving program
                    counter.
```

In addition it had conditional and unconditional forward relative jumps, and an unconditional backward relative jump. These were very much appreciated, since they simplified the task of the garbage collector when it was moving code blocks around.

The 4100 architecture also provided a range of user-definable extracodes — which provided a way of extending the instruction set by doing a vectored jump depending on a field in the instruction. POP used three extracodes. The first had the effect of a JIL instruction, but checked that the object being jumped to was a procedure record (all pointers pointed to the word after the key). The second was used to call the *updater* of a procedure. The third provided a backward relative jump, but performed certain checks before it did so. The checks were for stack underflow/overflow, keyboard interrupt and time-slot expiry. This latter was required because the POP machine was used as a time sharing system.

The garbage collector was able to search procedure objects to find the addresses in them, both for a mark and sweep operation and for storage compaction. A minimal set of high order tag bits were used, to distinguish between integers, short (22 bit) floats and pointers. Pointers pointed to the word after the key, so that procedures could be entered by a JIL instruction, which did not provide a fixed offset.

The main limitation of the machine were the short address field of 15 bits, which limited variables to the first 32k words of store (96k bytes). There was only one index register, which doubled up as a stack pointer. This was one reason we did not incorporate lexical variables in the language, given our decision to keep the open stack.

Chapter 17

POPLOG system facilities

NOTES
colon in Unix as a separator??
See section 17.4 for account of garbage collector - is it correct?

This chapter is specific to the POPLOG system, and tells you important facts about how it relates to the operating system, and how it operates internally.

17.1 System Startup and Reset

When POPLOG starts up, its initial action depends on whether or not arguments have been supplied to the operating system command that invoked it.

If no arguments were supplied, then after setting up, the banner in *popheader* is printed (if standard input is from a terminal), the procedure *sysinitcomp* is called to compile any '*init.p*' files, and finally *setpop* is called to reset the system for the first time.

If on the other hand arguments are present, the first argument must specify either

1. A string of POP-11 source to compile. This is the case when the first argument begins with '"' on VMS and with ':' on Unix.
2. A saved image to restore. This is the case when the first argument begins with '/' on VMS or '−' on Unix. This may be followed by other layered saved images beginning with the same character. For filenames not including an explicit directory, the directories given by the environment variable/logical name '*popsavepath*' are searched (see below).
3. Anything else, interpreted as the name of a POP-11 source file to compile. For filenames not including an explicit directory, the directories given by the environment variable/logical name '*popcomppath*' are searched (see below).

In all three cases, a list of the remaining arguments is placed in *poparglist*, so that user programs can access them, and the specified action is performed; if this returns normally, then the system exits. Thus compiling a POP-11 string or file will just do that and no more; a successful restore of a saved image, on the other hand, will not return, but simply restore the system state as it was when the file was saved.

A directory search path mechanism is supported for filenames specified in (2)and (3). A search path is an environment variable/logical name whose translation is a sequence of directory names, separated from one another by ':' (colon) characters in Unix or '|' (vertical bar) characters in VMS, and in which an empty directory name is interpreted as the current directory. A filename that does not include an explicit directory is then searched for in the given sequence of directories.

For saved images, the search path is given by '*popsavepath*', whose standard value is

```
:$poplib:$poplocalbin:$popsavelib   (Unix)
|poplib:|poplocalbin:|popsavelib:   (VMS)
```

(i.e. search the current directory first, then '*poplib*', then '*poplocalbin*', etc). For POP-11 files, the search path is given by '*popcomppath*', with standard value

```
:$poplib:$popautolib:$popliblib   (Unix)
|poplib:|popautolib:|popliblib:   (VMS)
```

In Unix systems only, there is an additional startup mechanism which allows the name under which the POPLOG image was invoked to be used as an argument. When the system is invoked under the name X, it looks to see if there is an environment variable called $'pop\_X'$, and if so, makes the translation string of that variable the first argument, shifting up the other arguments if necessary. Thus for example, if one creates a link called $'mysave'$ to the POPLOG image $'\$popsys/pop11'$, and defines an environment variable $'pop_mysave = -myimage.psv'$, then running the command $'mysave'$ will have the same as effect as

```
pop11 -myimage.psv
```

For this reason, the directory $'\$popsys'$ in fact contains the POPLOG image linked under various names, with corresponding $'pop\_X'$ environment variables defined, thus enabling the system to be invoked as $'pop11'$, $'ved'$, $'prolog'$, etc.

poparglist
This contains a list of strings which aare the rguments with which the POPLOG system was invoked. This will always exclude the first argument, and, if a number of layer saved images are being restored, will exclude those too. *poparglist* may be the empty list [ ].

sysinitcomp()
This is called on normal (no argument) POPLOG startup to compile $'init.p'$ files. If it exists, $'\$popsys/init.p'$ ($'popsys : init.p'$ on VMS, etc) is first compiled; then if $'\$poplib/init.p'$ exists, that is commpiled, otherwise $'init.p'$ in the current directory if that exists.

setpop()
This procedure resets the POPLOG system: unwinds all procedure calls, restoring dynamic local values, and clears the user stack, etc. If this is the first setpop, indicated by

$$pop\_first\_setpop = true$$

or if the standard input is a terminal, then compilation is (re)started; otherwise *sysexit* is called to exit from the POPLOG system.

If compilation is restarted, *pop_first_setpop* is set *false*, and the variables *cucharout* and *cucharerr* are set to *charout* and *charerr* respectively, after which the variable procedure *popsetpop* is called (see below). If this returns normally, POP-11 compilation is restarted. When standard input is from a terminal, *'Setpop'* is first printed, and in any case *compile*(*charin*) is called.

pop_first_setpop
This is *true* before *setpop* is first called, and thereafter *false*.

popsetpop()
This is called by *setpop* before doing *compile*(*charin*) to restart compilation. By assigning to it, a user program can thus regain control when *setpop* is invoked. Its default value is *identfn*.

17.2 System Exit

sysexit()
This calls the variable procedure *popexit*, and if this exits normally, *sysexit* then exits from the POPLOG system, i.e. it closes all files and returns to the operating system. *sysexit* is called by *setpop* when it reads *termin* from the standard input.

popexit()
This holds a procedure to be run on exit from the system, i.e. when *sysexit* is called. Thus the procedure in *popexit* may prevent the system from actually exiting, e.g. by doing a *setpop*. Its default value is *identfn*.

pop_exit_ok
This controls the status value returned by *sysexit* to the operating system: if *false*, the standard error code (1 in Unix, *SS$_ABORT* in VMS) is returned, otherwise the standard success code (0 in Unix, *SS$_NORMAL* in VMS). The default value is *true*, but note that a call of *mishap* will set it *false* if the standard input is not a terminal. Since the default value of *interrupt* under these circumstances is *sysexit*, this guarantees that a mishap will result in an error status being returned.

17.3 Interrupts

See also *timer_interrupt*, described in Chapter **??**.

interrupt()
This procedure variable is called:

1. by the procedure *mishap* after it has printed a mishap message and before it calling *setpop*. Thus redefining *interrupt* can be used to alter the action taken after mishaps – see Chapter 4.5;

2. by a keyboard interrupt (usually, typing Ctrl-C).

   The default value of this variable is *setpop* if the standard input is a terminal, *sysexit* otherwise. Thus e.g. when the system is running with a command file as its standard input, a mishap will result in system exit by default.

pop_enable_interrupts
This (active) boolean variable controls whether interrupt checking is enabled (true) or disabled (false). When disabled, neither keyboard (*interrupt*) or timer (*timer_interrupt*) interrupts are serviced until this variable becomes true again. (Note that this doesn't affect the calling of *interrupt* by *mishap*.)

From Robin Popplestone Mon Dec 19 12:12:59 EST 1988

17.4 Store Management

Managing the computer's store for a POP system presents a challenge, which is posed primarily by the fact that the allocation of storage in which objects are to be held is indepentent of the calling of procedures. In languages descended from ALGOL 60, such as Pascal, the default way of allocating store for a data-object is to place it in the stack-frame associated with the call of the procedure in which the data-structure is created. However this means that the data-object goes out of existence when the procedure in which it was created exits. Consequently, if we want to create data-objects which can be *returned* by the procedure in which they are created, a good way to do this is to provide a *heap*, which is an area of store independent of the stack[1]. An object is in fact represented as a number of contiguous machine words on the heap.

There are various ways in which objects can be allocated off the heap. The simplest is to keep a pointer to the next free memory locations, so that allocation would look something like this:

```
define alloc(n) -> O;
  free_memory fi_+ compound_tag -> O;
  free_memory + n -> free_memory;
enddefine;
```

This procedure however would not create a valid POP object, which must have a key pointer, usually as its second machine word.

Clearly the heap will at some time become exhausted: at this point the garbage collector is called to *reclaim unused objects in the heap*. The garbage collector knows that objects cannot be used if there are no pointers to them anywhere in the POP system. What in fact it does is determin which objects *do* have pointers to them, mark them as such, and then recover the storage used by those objects which are not marked.

If we are using the simple allocation mechanism described above, there is still a problem — the objects which are discovered to be free by the garbage collector will not in general

[1]In fact in POP, every process has a stack which is held in the *heap*, so stacks are in effect just POP objects

be contiguous. Thus the garbage collector has to relocate all the used objects. This can be done in various ways — one simple way is to recreate the whole heap in a new area of memory. This can be done, at some cost in disk traffic, in a system like Unix which provides virtual memory.

The garbage collection of *temporary properties* differs from that of ordinary objects, and is discussed in Chapter 11.3.

The C language also provides a heap: objects can be allocated in C using the *malloc* procedure. When C and POP are used together, the heaps are kept distinct, since the POP heap must be structured in such a way that the garbage collector can recognise objects.

sys_lock_heap()
This tells the system to *lock* the heap at its current endpoint. The effect is that all structures in the heap before this point will automatically be treated as non-garbage, thus considerably reducing the amount of work needed to be done on them during a garbage collection. Structures created after the *sys_lock_heap* will be treated normally. It is sensible to call *sysgarbage* before *sys_lock_heap*. E.g. at the end of a file which compiles a lot of procedures which are not going to be edited, do

```
sysgarbage(); sys\_lock\_heap();
```

See also *sys_lock_system* below.

sys_unlock_heap()

This reverses the effect of *sys_lock_heap*, so that all structures previously locked in now participate fully in the next garbage collection.

sysgarbage()
This forces the system to do a garbage collection. Note that the time for such a user-invoked

garbage collection is not included in the automatic GC time total see *popgcratio* below.

$pop\_after\_gc()$

The procedure in this variable is run after every garbage collection (default value $identfn$).

popgcratio

This controls the weighting of garbage collection time in determining memory allocation for the system. The decision as to whether expand or contract memory is made by considering the ratio of total CPU time to total garbage collection CPU time multiplied by *popgcratio*, and then expanding or contracting by an amount proportional to this ratio. The maximum permitted value for *popgcratio* is 64, default value is 25.

popgctime

This contains the total cpu time spent doing garbage collections, in 1/100ths second.

popgctrace

If this variable is true then a message is output after each garbage collection, giving the GC time and the heap memory allocation, in the form

```
GC-<code> TIME: <t> MEM: <u> used + <f> free + <s> stack = <m>
```

where $< code >$ is a 4-letter code indicating why the GC happened, $< t >$ is the time for the GC in 1/100ths sec, $< u >$ is the number of words used in the heap, $< f >$ the number free, $< s >$ the number taken by the userstack, and $< m >$ the total. Note that these values include structures locked in with *sys_lock_heap*. The procedure held in *cucharerr* is used for the output.

Also, if *popgctrace* is an integer, additional information about the Prolog store area is printed out, for example:

```
;;; GC-user TIME: 0.83, MEM: used 9729 + free 94719 + stack 0 = 104448
;;;            CALLSTACK: 293
;;;            PROLOG: trail 7 + free 2008 + contn 33 = 2048
```

This indicates that the collection took 0.83 seconds, and afterwards there were 9729 words used, and 94719 free, with a total of 104448. The number for STACK shows how much space is taken on the POP-11 argument stack. The third line shows how space is used on Prolog's variable trail and continuation stacks.

popmemlim
This integer specifies the maximum number of words to which the system should expand heap memory, including the user stack. If this is exceeded, the system will not expand further and a $'RUNOUTOFMEMORY'$ mishap will result. Its default value when just POP-11 is in use is 100000, but higher default values are used by the Prolog and Lisp systems.

Note that setting this to too high a value may mean that the system will expand heap memory to the point where there is insufficient extra available for the garbage collector to operate, resulting in the mishap $'CANNOTALLOCATEGCWORKSPACE'$, which is non-recoverable, and necessitates restarting the POPLOG system from scratch. If the default value is not large enough for your program, you should therefore experiment with increasing it gradually. There are other limits on the size of certain data-areas in POPLOG, see *pop_callstack_lim* in Chapter 2.23 and *pop_prolog_lim* in Chapter **??**.)

popmemused
This contains the number of words of memory in use at the last garbage collection.

17.5 Saving and Restoring the System State

$syssave(FILENAME)- > RESTORED$
This saves the total state of the POPLOG system in a file called $FILENAME$. The result

of this procedure is *false* on immediate return; on restoring the file with *sysrestore* (qv) the result will be *true*.

FILENAME may be either a word or a string: if a word, *'.psv'* is appended. Note that the state of external procedure loading is also saved — see Chapter **??** for details.

sysrestore(*FILENAME*)
sysrestore(*fILENAME*, *MODE*) This restores the POPLOG system to be in the state it was at the time the saved image specified by the filename *FILENAME* was created with *syssave* or *sys_lock_system*, with the exception of

- The standard files (*popdevin*, *popdevout*, *popdeverr*);
- *poparglist*;
- *poppid* and *popusername*.

etc, all of which remain as they were before the call of *sysrestore*. When this procedure exits, it will be just as if the call of *syssave* or *sys_lock_system* that created the file had returned, but with *true* as result instead of *false*.

By default, a mishap will result if either (1)*FILENAME* does not exist, or (2)the identification string from *FILENAME* does not match the current value of *pop_system_version*, i.e. *FILENAME* may only be restored into the same system environment in which it was created.

The optional boolean argument MODE can be used to prevent a mishap occurring in either of these cases. *false* means no mishap for case (1), *true* means no mishap for either (1)or (2). Instead of producing a mishap, *sysrestore* will simply return to its caller (with no result). *FILENAME* may be either a word or a string: if a word, *'.psv'* is appended.

Note (VMS): If *FILENAME* is a shareable image created by *sys_lock_system* the constant part of *FILENAME* will be shared between all users in the same group, unless *FILENAME* has been installed with *sys_install_image* to be shareable system-wide (i.e. there is a permanent system global section for it).

$pop\_after\_restore()$
The procedure in this variable is run by *sysrestore* immediately after a successful restore and before returning (default value $identfn$).

$sys\_lock\_system(FILENAME, SHARE, SYSTEM_ID) -> RESTORED$
This procedure makes possible the creation of 'layered' systems in POPLOG. It defines the current heap endpoint to be the boundary of a self-contained $'system'$ on top of which other layers will be built. The principal purpose of specifying such a boundary is to enable POPLOG to partition the current contents of the heap into areas of constant and non-constant data structures, and thereby:

1. create a saved image for the new system in which the area of constant structures can, at least in principle, be shared by all users of the saved image;
2. make it unecessary to for the constant area to be saved when creating further saved images on top of the new system, either with *syssave* or with another *sys_lock_system*.

The major consequence of (2)is that subsequent saved images can be restored only into the environment saved by (1), and em not into the basic POPLOG system, i.e. one saved image must be layered on top of another.

In detail, *sys_lock_system* operates as follows:

The heap is first locked at its current endpoint, as for *sys_lock_heap*. All the structures so locked in are then partitioned into 2 areas – constant and non-constant – which from then on are considered to be completely *outside* of the heap, constituting 'part of the system'. Thereafter this change *cannot* be undone (i.e. there is no corresponding unlocking procedure). The structures in the constant area become non-writeable, so that any subsequent attempt to update a field in one of them will cause an error. Currently, only procedure records and the strings pointed at by word records are made constant, everything else being assumed to be non-constant, This means, since word strings cannot be updated anyway, that the only fields that become non-writeable are the *pdprops*, *updater*, *pdpart* and $frozval$s of procedures and closures. To reflect the fact that the constant and non-constant areas have been removed from the heap, *popmemlim* is reduced by their total size, or to 50000, whichever is the greater.

Following the constant/non-constant partitioning, a 'special' call of *syssave* with filename argument $FILENAME$ creates a saved image to preserve the initial state of the new augmented system, the result of *syssave* being returned as the result of this *sys_lock_system* (i.e. $false$ after saving, *true* on restoring). The call is 'special' in the sense that it does save the constant area, whereas any subsequent *syssave* or *sys_lock_system* will not. As mentioned above, this means that subsequent saved images can be restored only into the new system environment, either without ever leaving it or after restoring the system saved image. To ensure that any attempt to do otherwise is trapped as an error, the variable *pop_system_version* holds an identification string for the current system, this being recorded on any saved image created therein. *sysrestore* will produce an error if requested to restore an image whose identification does not equal the current value of *pop_system_version* (which is set by *sys_lock_system* to be the $SYSTEM_ID$ argument string 'timestamped' by concatenating *sysdaytime*() onto the end). On restoring as saved image created with *sys_lock_system*, the boolean argument SHARE controls whether the memory into which the constant area is mapped will be shared by all users, true meaning shared, false meaning unshared. Currently, sharing is only possible in the VMS system (and then only between users in the same group unless *sys_install_image* is used, qv). It is not possible in Unix versions, although may become so in later versions of Berkeley 4.2. Note that the SHARE argument may still be true. In VMS, you are strongly advised em not to make an image shared unless it really needs to be, i.e. will be used concurrently by several users on a regular basis, since shared images consume VMS system resources.

sys_install_image$(FILENAME, INSTALL)$
This procedure is currently available only in VMS systems. In VMS, the constant part of a shareable saved iamge created by *sys_lock_system* is by default shared only between users in the same group; this procedure enables or disables its sharing by *all* users in the system. To operate, however, the process running this procedure must have the privileges

```
SYSGBL (create/delete system global section)
PRMGBL (create/delete permanent global section)
```

If the boolean argument INSTALL is true, then the image specified by $FILENAME$ is 'installed', i.e. made shareable system-wide. The blocks comprising the constant area are made a permanent system global section. If INSTALL is false, $FILENAME$ is deinstalled and will revert to group sharing. (An existing permanent system global section for $FILENAME$ is deleted.)

17.6 Incremental Save and Restore

syssaveincr($FILENAME$,
) procedure Saves, in a file called $FILENAME$, all structures that can traced from the item O, i.e. O, all structures that O references, that those structures reference, and so on. However, although references to any structure which is part of the system are saved, anything referenced BY that system structure will not be saved. Thus, for example, if *cucharout* has been given a particular value, this value will not be saved. To obviate this problem, if O is a list then the following happens: O and all structures traceable from it will be saved, but in addition, permanent identifiers associated with any system words occurring in O will also be saved. Thus every system word required to be saved must be mentioned explicitly in the list O. See *sysrestoreincr* for what happens when the file is restored. Example:

```
vars list = [1 2 3 4];
syssaveincr("save", "list");
```

saves the word "list", its identifier and the list it contains, in a file called *'save.psi'*. $FILENAME$ may be either a word or a string: if a word, *'.psi'* is appended. (Note: this procedure is defective in that it doesn't cope with sections and various other things.)

sysrestoreincr($FILENAME$)
sysrestoreincr($FILENAME, MODE$) Restores all the structures saved with *syssaveincr* in the file $FILENAME$. Any words that were in the dictionary at the time of saving are restored to the dictionary if not already there, associated permanent identifiers being set to their saved values. Any system words explicitly saved also have associated permanent identifiers restored. E.g. restoring the example given in -syssaveincr-:

```
sysrestoreincr("save");
list =>
** [1 2 3 4]
```

restores the file called *'save.psi'*, i.e. the word *"list"*, its identifier and the list it contains. $FILENAME$ may be either a word or a string: if a word, *'.psi'* is appended. The optional

MODE argument is the same as for *sysrestore*, qv.

17.7 System Identification

pop_internal_version
This is an integer defining the version of the POPLOG system in use, in the form:

$$(n_{major} * 1000) + n_{minor}$$

where n_{major} is the major version number and n_{minor} the minor version number within the former.

popversion
This contains an identification string giving the major version number and creation date of this POPLOG system, e.g.

```
'(Version 12.3 Thu Nov 20 20:54:11 GMT 1987)'
```

Note that the version number appearing in this string is derived as

```
(pop_internal_version div 1000)/10.0
```

pop_system_version
This contains the identification string for the current system and version, used by *syssave* and *sys_lock_system* to record the identity of the system in which a saved image is being created, and by *sysrestore* to check that identity when restoring. Only *sys_lock_system* alters this variable; in the base POPLOG system, it has the same value as *popversion*.

popheader

This is a banner string, normally printed out when POPLOG starts up in normal mode, i.e. when not invoked with arguments, and thereafter made $false$ to indicate that the banner has been printed. The string is constructed as

```
'Sussex POPLOG ' <> popversion
```

e.g.

```
'Sussex POPLOG (Version 12.3 Thu Nov 20 20:54:11 GMT 1986)'
```

sys_os_type

This is a list of words and numbers giving attributes of the operating system under which POPLOG is running. Current possible values are:

```
[vms 4.0]                ;;; VAXes
[unix bsd 4.2]           ;;; Sun Workstations and VAXes
[unix bsd 4.1]           ;;; VAXes
[unix att 5.2]           ;;; GEC Series 63
[unix att 5.0]           ;;; Hewlett-Packard Bobcat
[unix unisoft 5.0]       ;;; Bleasdale
```

Be careful about comparing floating-point version numbers in this list: rounding and other factors may mean that = will not return $true$ for numbers that appear to be equal. You should always compare by testing the absolute value of the difference between two numbe rs to be less than some small amount.

sys_processor_type

This is a list of words/numbers giving attributes of the machine on which POPLOG is running. Current possible values are:

```
[vax]
[68000]
[68010]
[68020]
[gec63]
```

pophost$(W) -> O_{attribute}$
This procedure is a property that supplies more complete information about the underlying system on which POPLOG is running than do *sys_os_type* and *sys_processor_type*. Currently, there are entries in this property for the following keywords (W):

```
"os"                      operating system name
"osversion"               operating system version
"machine"                 host machine
"machine"                 generic machine type (eg $'vax'$)
"systemname"              the local name of the machine
"memory"                  approximate amount of memory
"machineserialnumber"     serial number of machine
"site"                    where the machine is
"sitemailname"            electronic mail address for site
"fullsitename"            offical site address
```

All $O_{attribute}$ values are strings or numbers, generally in lowercase only, the default for entries being $false$. You should examine the library file that defines this property, by doing *showlib pophost* to find the format of the information you require before writing programs to use it.

Chapter 18

How POP tells the time

NOTE
Sort out $t_1 00$ - is it float or integer?

18.1 Date and Time Procedures

$sys\_real\_time() \rightarrow n_{secs}$
This procedure returns the time as an integral number of seconds since the standard Unix base date, i.e. 00:00 GMT on 1 Jan 1970 (you should not assume this value is a simple integer: it may very well be a biginteger).

$sys\_convert\_date(n_{secs}, LOCAL) \rightarrow bfs$
Given a time in seconds since 00:00 GMT 1 Jan 1970 (as returned by *sys_real_time*, or a file date returned by *sys_file_stat* etc), returns a date string in the standard operating system format, i.e.

```
'nnn mmm dd hh:mm:ss <timezone> yyyy'
```

in Unix systems (where nnn is the day name), and

```
'dd$mmm$yyyy hh:mm:ss'
```

in VMS systems. In VMS systems, the $LOCAL$ argument is ignored; in Unix systems it specifies whether n_{secs} is interpreted as local time ($true$) or GMT ($false$), and therefore affects the value of the $< timezone >$ substring (which depends upon the $timezone$ C library function — see $ctime(3)$ in the Unix Programmers Manual.)

$sysdaytime() \rightarrow bfs$
This procedure returns the current date and time as a string in the standard form. This procedure is just

$$sys\_convert\_date(sys\_real\_time(), true)$$

18.2 CPU Time

$systime() \rightarrow t_{100}$
This procedure returns the elapsed CPU time for this run of the POPLOG system, an integer number of hundreths of a second.

$timediff() \rightarrow t_{secs}$
This procedure returns the elapsed CPU time since the last call of $timediff$, as a floating-point number of seconds t_{secs} (the first call produces a meaningless result). This procedure uses $systime$, so the resolution of the result will be hundreths of a second.

18.3 Timer Procedures

The variables *pop_timeout* and *pop_timeout_secs* are also available for timing-out terminal read operations — see Chapter **??** for details.

$syssettimer(t_{100})$
$syssettimer(t_{100}, P)$

This sets a timer interrupt for t_{100} hundreths of a second. When this time has expired, the procedure in the variable *timer_interrupt* is called inside whatever procedure the system is currently executing. This is a one-off setting, i.e. if another timer setting is required, *syssettimer* must be called again.

The second form of the call simultaneously assigns a procedure to *timer_interrupt*, and is the same as

```
P -> timer_interrupt;
syssettimer(t_100);
```

For example, the following will cause "hello" to be printed every 5 seconds:

```
define vars timer_interrupt();
    "hello"=>
    syssettimer(500)
enddefine;

;;; start it off
syssettimer(500);
```

In Unix systems note that, although t_{100} is specified in hundreths of a second the actual timer value set can only be an integral number of seconds, because that's all the Unix *alarm* system call allows. The given value of t_{100} is therefore rounded up to the nearest integer number of seconds.

syscantimer()
Cancels any outstanding timer interrupts set by *syssettimer*.

timer_interrupt()
The procedure in this variable (default value *identfn*) is called when the timer set by *syssettimer* goes off. However, note that all interrupts, including the timer, can be disabled by assigning *false* to *pop_enable_interrupts*, as described in Chapter **??**.

syssleep(t_{100})
Suspends the current POPLOG process awaiting some form of interrupt, or until t_{100} hundreths of a second has expired, which ever is the sooner. This procedure is equivalent to

```
define syssleep(t_100);
     lvars t_100;
     dlocal timer_interrupt = identfn;
     syssettimer(t_100);
     syshibernate()
enddefine;
```

(and so in Unix, the value of t_{100} is rounded up to an integral number of seconds, as for *syssettimer*).

syshibernate()
Suspends the current POPLOG process awaiting some form of interrupt, e.g. Ctrl-C typed on the terminal, a timer interrupt set by *syssettimer*, etc. What happens then depends on the appropriate interrupt-handling procedure, i.e. *interrupt* for Ctrl-C, *timer_interrupt* for the timer, etc.

Chapter 19

How POP reads objects from the input

There is a type of POP procedure called an *itemiser*, which converts a stream of input characters into a stream of data-objects.[1] You can write itemisers for yourself, but there a capability built into the POP system to provide you with itemisers that embody the standard POP syntax. Moreover, it is possible to modify the behavior of these itemisers to some extent to match particular needs you may have. If you want to write your own itemiser this is discussed in Chapter **??**.

19.1 Character Classes

The itemiser procedure returned by *incharitem* (see below) takes a stream of input characters produced by a character repeater procedure and turns it into a stream of items for compilation, or any other use. To effect this process, each ASCII character value from $0-255$ has associated with it an integer defining the class of that character, the class of a character governing how it is treated.

[1] In the compiler literature this capability is called a "tokeniser".

The 12 pre-defined classes are described below. Note that the class names (and examples of them) are determined by the normal assignment of classes to characters, although by using *item_chartype* the user can assign any character to any desired class, either globally or for a particular item repeater (thus for example, the letter 'A' can made to behave as if it were a separator in class 5).

Class	Description
1	Alphabetic — the letters $a - z$, $A - Z$
2	Numeric — the numerals $0 - 9$
3	Signs — characters like $+$, $-$, #, $, & etc. characters in classes 10 and 11 (bracketed comment 1 & 2) will default to this class if not occurring in the context of such a comment.
4	Underscore, i.e. _
5	Separators— the characters . , ; " % and the brackets [,], {, }. Control characters are also included in this class (except for those in class 6), as are all characters 128-255
6	Spaces — the space, tab and newline characters
7	String quote — the character '. This appears on most terminals as $'$.
8	Character quote — the character `. This normally has a less "comet-like" appearance when displayed by a computer.
9	End of line comment character — the character ';' (but see below)
10	Bracketed comment or sign, 1st character — the character /
11	Bracketed comment or sign, 2nd character — the character *
12	Alphabeticiser — this is special class that forces the next character in the input stream to be of class alphabetic, i.e. class 1 — see below. \ (backslash) may be given this type by default in later versions of POPLOG.

New classes other than these can be defined with the procedure *item_newtype*, as described in section 19.5

19.2 Syntax of Items Produced by the Itemiser

The itemiser splits up a stream of characters into a stream of objects, each object being one of the following types:

$word$ (see 8.1) $string$ (see 9) $integer$ (or biginteger) $ratio\ floating-point$ (decimal or ddecimal) $complexnumber$ (see 6).

This is done according to the following rules:

19.2.1 Word

A word is represented by either

1. a sequence of alphabetic or numeric characters beginning with an alphabetic one, e.g. 'abc123', 'X45' ;
2. a sequence of sign characters, e.g. $'+'$, $'\&\$+'$;
3. a sequence of words produced by (1) or (2) joined by underscores, e.g. '$fast\_+'$, '$\_123\_++\_678'$
4. a single separator character, e.g. '$['$.
5. a sequence of characters in a new class created by $item\_newtype$.

19.2.2 String

A string is represented by any sequence of characters starting and ending with string quotes, e.g. $'abcdefgh12\&\&\&\&'$. If the characters of the string extend over more than one line, the newline character at the end of the line must be preceded by the character '\' (backslash), unless $pop\_longstrings$ is true, i.e. if $pop\_longstrings$ is $false$ then an unescaped newline causes a mishap. A newline can also be inserted as $'\backslash n'$. See Note (1) below.

19.2.3 Integer

An integer is represented by either:

1. A sequence of digits, optionally preceded by a minus sign '$-$', e.g. 12345, -789;
2. A number preceded by an integer and a colon, meaning that the number is to be taken to the base of the integer, e.g. 2:1101 represents 13 as a binary number. The integer base must be in the range $2-36$; if greater than 10, the letters A-Z (uppercase only) can be used in the number to represent digit values from 10 to 35, e.g. $16:1FFA$ represents 8186 as a hexadecimal number. If a minus sign is present, this may either follow the radix or precede it, e.g. $-8:77$ or $8:-77$ are both valid.
3. A character constant, giving the integer ASCII code for that character. This is any character preceded by (and optionally followed by) a character quote. E.g. 'a' gives the ASCII value for lowercase 'a', namely the integer 97. See Note(1) below.

Except in the character constant case, an integer may optionally be followed by the letter 'e' and a (signed or unsigned) integer to indicate an exponent specification, i.e.

```
<int-1>e<int-2>
```

will produce $n_1 b^{n_2}$ where n_1 is the integer corresponding to $< int-1 >$ and n_2 is that corresponding to $< int-2 >$ and b is the radix specified in $< int-1 >$. This may actually result in the production of a ratio rather than an integer, e.g.

```
2:110e5 = 2:110 * (2 ** 5) = 192
23e-2   = 23 * (10 ** -2)  = 23_/100
```

If the integer read in is too large to be represented as a *simple* object (see 3.7) then a biginteger is created. E.g.

```
isinteger(123456789) =>
** $true$
isinteger(123456789123456789) =>
** $false$
isbiginteger(123456789123456789) =>
** $true$
```

19.2.4 Floating-Point

A floating-point literal is a sequence of numeric characters containing a period, e.g. 12.347; as with integers, this can also be prefixed with a base, i.e. an integer followed by a colon. The whole number, including fractional places, is taken to this base. As with integers, an exponent specification may follow, but in this case any of the letters '*e*', '*s*' or '*d*' may be used. That is

```
<float>e<n> <float>e<n> <float>e<n>
```

all produce $x * (b^n)$, where x is the number specified by $< float >$, and b is the base of $< float >$. The difference between them is that e and d specify a double-float (*ddecimal*), whereas s results in a single-float (*decimal*). Thus

```
23.0e-2  = 23.0 * (10 ** -2)  = 0.23     (ddecimal)
2:11.1d5 = 2:11.1 * (2 ** 5)  = 112.0    (ddecimal)
56.2s+3  = 56.2 * (10 ** 3)   = 56200.0  (decimal)
```

19.2.5 Ratio

A ratio is two integers (numerator and denominator) joined by the character sequence '_/'. Thus 2_/3, −467_/123678 are ratios. If the numerator is preceded by a radix, then this radix applies also to the denominator; the denominator itself must not have a radix or preceding minus sign. Note that owing to the rule of rational canonicalisation the resulting object will actually be a *ratio* with the greatest common denominator divided out of numerator and denominator, or an *integer* if this would make the denominator equal to 1.

19.2.6 Complex Number

A complex number is any two of the above kinds of number (the real part and the imaginary part) joined by the character sequence '_+:' or '_-:', Thus 2_+ : 3, 1.2_+ : 8.9, 5_/4_− : 3_/2 are complex numbers.

The imaginary part must not have either a radix specification or a preceding minus sign; as with ratios, the radix of the first number (if any) carries over to the second, and the sign of the imaginary part is determined by the joining sequence, '_+:' or '_-:'. If an explicit radix is specified, then this must *precede* any minus sign on the real part. That is, 16 : −10_+ : 4 is correct, while −16 : 10_+ : 4 is not.

The two numbers may be of different types, although when either is a floating-point the actual result will have both parts coerced to the same type of float; in addition, when both parts are rational, the result will be a rational rather than a complex if the imaginary part is integer 0.

19.3 Operation of Character Classes

The itemiser reads characters and produces items from them according to the rules given above; all characters in the space class are ignored, and only serve to delineate item boundaries (but see Note 2). The effect of other classes not mentioned in the preceding rules, i.e. the comment classes and the alphabeticiser, are as follows:

19.3.1 Alphabeticiser - Class 12

An occurrence of a character of this class causes the next character read to be interpreted as though it belonged to the alphabetic class, regardless of its actual class. Assuming that \ belongs to this class, this means that for example

```
A\+B\-C   \&_\[\{\(   \12345
```

are all valid 5-character words. In addition, the following character is also interpreted as for the character following \ in strings and character constants (see Note (1) below), thus enabling non-printable characters to have class alphabetic, e.g.

```
\nA\^A\^Z\r
```

is a word consisting of the characters newline, A, Ctrl-A, Ctrl-Z and carriage return (ASCII 10, 65, 1, 26, 13).

19.3.2 End of line Comments - Class 9

A character in this class causes the rest of the current line upto a newline to be treated as a comment and ignored. Normally, this character is semicolon ';' and, *in this case only*, 3 adjacent semicolons are actually required for a comment. If a semicolon occurs by itself, or only adjacent to one other, then it is treated as a separator (class 5). This is due to the POP-11 compiler needing ';' for punctuation, and the fact that ';;;' has always been the POP-11 comment escape.

19.3.3 Comments - Classes 10 and 11

These two classes provide for comments which begin with a 2-character sequence like '/*' and end with the reversed sequence '*/', and which otherwise occupy any number of characters or lines in between. The start of such a comment is therefore recognised as a class 10 character immediately followed by a class 11 character, after which characters are read and discarded until the sequence class 11 followed by class 10 is encountered. During the reading of the comment another occurrence of class 10, class 11 is taken as a nested comment and so will correctly account for such nesting. For example (assuming / and * have classes 10 and 11 respectively):

```
    1 -> x; /* this is a comment */ 2 -> y;
/*  1 -> x; /* this is a comment */ 2 -> y; */
```

where in the second example the whole line has been commented out. Any occurrence of class 10 or 11 characters other than one immediately followed by the other will default to class 3, i.e. to the sign class.

19.4 Notes

(1) Non-printable characters (e.g. control characters) can be represented inside strings and character constants using the character '\' (backslash) combined with other characters, as follows:

```
\n  = newline           (ASCII 10)
\r  = carriage return   (ASCII 13)
\t  = tab               (ASCII 9)
\b  = backspace         (ASCII 8)
\s  = space             (ASCII 32)
```

'\' in conjunction with ' ↑' (up-arrow) followed by one of the characters

```
@  A-Z  [  \  ]  ^  _  ?
```

can be used to represent the control characters ASCII 0 - 31 and ASCII 127, i.e.

```
Seq     ASCII   Name

\^@       0     null
\^A       1     Ctrl-A
\^B       2     Ctrl-B
   ...
\^Z      26     Ctrl-Z
\^[      27     ESC
   ...
\^_      31
\^?     127     DEL
```

'\' can also be followed by '(' to signal an explicit integer value for a character, the integer being terminated by ')'. E.g.

```
'\(255)abc'
```

is a string containing the characters 255, 'a', 'b' and 'c'. The integer obeys the normal itemiser syntax, so can be radixed, etc. It must, of course, be ≥ 0 and ≤ 255.

As described above, these conventions also operate with the character following any alphabeticiser (class 12) character.

(2) The effect of the variable *popnewline* being true is to change the class of the newline character (ASCII 10) to be 5, i.e. a separator. Thus instead of being ignored as a space-type character, a newline will produce the word whose single character is a newline.

19.5 Associated Procedures

$incharitem(P_{char\_rep}) \rightarrow P_{itemrep}$
This returns an item repeater $P_{itemrep}$ constructed on the character repeater P_{char_rep}, i.e. $P_{itemrep}$ is a procedure which each time it is called returns the next item produced from the characters supplied by P_{char_rep}, or $termin$ when there are no more to come. $P_{itemrep}$ is initially set up to use the global character table; by use of *item_chartype* (see below) $P_{itemrep}$ can be made to use its own local table.

popnewline
If true, this boolean variable causes item repeaters produced by *incharitem* to change the class of the newline character (ASCII 10) to be 5, (i.e. a separator), so that instead of being ignored as a space-type character, a newline will produce the word whose single character is a newline. (Default value $false$)

pop_longstrings
A boolean variable controlling reading of quoted strings by *incharitem* item repeaters. If this is $false$ then quoted strings cannot contain a newline unless preceded by '\'. Otherwise strings can extend over several lines without the backslash at the end of each line. (Default value $false$)

$isincharitem(P_{itemrep}) \rightarrow P_{char\_rep}$
Given an item repeater produced by *incharitem*, or given *itemread* or *readitem*, returns the underlying character repeater being used to construct items, or $false$ if there isn't one. If the argument is *itemread* or *readitem* then *proglist* is examined to see if it is a dynamic list: if so, then *isincharitem* is applied to the generator procedure (see 15.2).

$item\_chartype(c) \rightarrow n$
$item\_chartype(c, P_{itemrep}) \rightarrow n$
$n \rightarrow item\_chartype(c)$
$n \rightarrow item\_chartype(c, P_{itemrep})$

The base procedure returns the integer class number n associated with the character whose ASCII code is c, either for the global character table (the first form) or for the item repeater $P_{itemrep}$ (the second form). The updater assigns the class number n to the character whose ASCII code is c, either for the global character table (the first form) or for the item repeater $P_{itemrep}$ only (the second form). Note that once an assignment has been done for a particular item repeater $P_{itemrep}$, it will no longer use the global table, so that subsequent changes to this will not be reflected in $P_{itemrep}$. On the other hand, changes to the global table *will* be reflected in all item repeaters which have not been locally changed. For both base and updater, the item repeater $P_{itemrep}$ (when supplied) may be either a procedure produced by *incharitem*, or one of the procedures *itemread* or *readitem*. In the latter case, the item repeater at the end of *proglist* is used.

$item\_newtype() \rightarrow n$

This returns an integer $n > 12$ representing a new class of characters that form words only with members of that class. The value returned can be given to *item_chartype* to assign any desired characters into the new class.

$nextchar(P_{itemrep}) \rightarrow c$

$O_{c_{or_s}} \rightarrow nextchar(P_{itemrep})$

This returns (and removes) the next character in the input stream for the item repeater $P_{itemrep}$ — this may or may not call the character repeater on which $P_{itemrep}$ is based, depending on whether there are any characters buffered inside $P_{itemrep}$. The updater adds character(s) back onto the front of the current input stream for the item repeater $P_{itemrep}$. If $O_{c_{or_s}}$ is an integer character, then this is added; otherwise it must be string, in which case all the characters of the string are added. $P_{itemrep}$ may take the same values as for *item_chartype*.

Chapter 20

Character Input and Output

NOTES What is ‘normal mode character output?’ is it a Unix term or what?
Say what streams are (if I know).

This chapter describes the POP capabilities for inputting and outputting a sequence of characters one at a time using procedures called character repeaters and character consumers.

So long as you are inputting from and outputting your terminal (or terminal emulator on a workstation) you may not need to know about character repeaters and consumers, but use the procedures described in Chapter 19 for the input of POP objects expressed in external form as *items*, or those described in Chapter 21 for the output of objects.

However you will need to make use of the character repeaters and consumers described in this chapter if you want your program to read input from some source other than the terminal, perhaps a disc-file, or to direct output to somewhere other than the terminal.

As well as the capabilities described in this chapter, there are also capabilities which use *device records* which are more closely related to the operating system, and which consequently are also more dependent upon the particular operating system POP is running under. In order to obtain the most detailed control of input and output you may need to make use of these more basic capabilities, which are described in Chapter **??**.

It is common to refer to a sequence of characters being input or being output as a *stream*, although somewhat different concepts go by this name. For example Abelson and Sussman [?] use the term for a construct very like POP dynamic lists. The term seems to have been used first by Landin [?].

A character repeater is a procedure which each time it is called returns the next character from an input stream; a character consumer takes a character as argument and writes it to an output stream. Character streams are terminated by the special item *termin* (the value of the constant *termin*); a character repeater will return *termin* when an input stream is exhausted, and giving *termin* to a character consumer will close its output stream.

These repeater and consumer procedures are used by many parts of the system, including the POP-11 compiler and printing utilities.

The input and output capabilities of POP are switchable at various levels, which arise from the development of POP and the operating system environments that it runs in. If you are using the character consumers to do output, it makes sense to switch output by making *cucharout* to be a *dlocal* variable of procedures that do output.

20.1 Standard Character Repeaters

$charin() \to c$
This is a character repeater which normally takes input from the keyboard of the terminal, or terminal emulator, that you use by default to communicate with POP.

In POPLOG running under Unix, it is a character repeater for the standard input channel of the Unix process in which POP is running. Except when in the 'interactive mode' of the VED editor, it reads characters from the device *popdevin*.

In POPLOG, when the VED editor is in interactive mode *charin* receives characters from the current VED input buffer. For details see the VED manual[?].

Also in POPLOG, it is possible to arrange for a *charin* call to 'time out'. For details see Chapter **??**.

poplastchar
This always contains the last character read by *charin*.

popprompt
The 'prompt string' is a sequence of characters that is output by the computer to indicate that it is ready to receive input, and this variable holds the string that will be output to prompt input if *charin* is used. Information about the use of other terminals is to be found in Chapter 23.3.

poplinenum
This integer variable is incremented whenever a *newline* character is read by *charin* or a character repeater created by *discin*. *poplinenum* is local to *compile* where it is initialised to 1: hence it records the number of newline characters read in since the beginning of the call of *compile*. It is used by *sysprmishap* in printing mishap messages, see Chapter 4.5.

poplogfile
This variable may contain a character consumer procedure or *false*, which is the default value. In the former case every character read by *charin*, or output by *charout* is also given to *poplogfile*. Prompts are included in this file. This can be used to record terminal interactions.

termin
termin_key

The value of *termin* is the unique item *termin*, used as an end-of-file marker for character/item stream I/O. It is also used in the implementation of dynamic lists as described in Chapter **??**). *termin_key* is a constant holding the key structure of the unique item *termin* (see Chapter 3.13).

20.2 Standard Character Consumers

charout(c)
This is the character consumer that is usually used to cause characters to appear on your terminal, although, if you want more control of what your terminal does, you may wish to use the 'raw mode' capability provided by the procedure *rawcharout*. *charerr*(c)

This is the character consumer used to print *mishap* messages.

In Unix POPLOG, the above are character consumers for the standard output and error output channels respectively of the Unix process in which POPLOG runs. They normally output characters via the devices *popdevout* and *popdeverr*, both of which are described in Chapter 23.7, and will thus cause characters to appear on your terminal, or terminal emulator.

There are two exceptions to this: Firstly if you are using the POPLOG VED editor interactively, characters will be put in a VED buffer, and so indirectly will usually both appear on your screen and be available for editing. Secondly, it is possible to redirect Unix output streams, as described in Chapter 23.7 and the Unix manual.

pop_charout_col
pop_charerr_col

These variables provide you with a way of knowing the horizontal location of the next character to be output by *charout* and *charerr* respectively, and thus provide assistance in controlling the layout of text produced by a program. In fact they contain integers representing the number of columns filled by *charout* and *charerr* respectively in their current lines of text. Normally *tab* characters count as 8 columns. Control characters, i.e. those < 32 count as filling no column. All other characters count as filling 1 column.

The values of these variables are reset to 0 when a *newline* character is output through the respective consumer.

In POPLOG, the VED editor provides a variable *vedindentstep* which is used to control the effect of *tab* characters.

Chapter **??** gives a much more sophisticated treatment of the layout of output using variable size fonts etc.

poplinemax
poplinewidth

When *poplinewidth* is an integer, these two variables together control the breaking of long lines by *charout* and *charerr*. If on outputting a character with *charout*, *pop_charout_col* is greater than or equal to *poplinewidth* and either the current character for output is a space or a tab *or* the column count is already $\geq$ *poplinemax*, then a *newline* and a *tab* will be inserted before printing that character. In other words, lines are broken at the next whitespace character after *poplinewidth* columns, or failing that after *poplinemax* columns. The treatment of the *charerr* procedure is the same.

If *poplinewidth* is not an integer, line breaks are not inserted. Its default value is 70.

cucharout(c)
cucharerr(c)

These procedure variables contain the current character consumers for standard output and error output. All printing utilities in POP use *cucharout* to produce output, and all error printing is done via *cucharerr*. The standard values of these variables are *charout* and *charerr* respectively.

cuchartrace
This contains the character consumer procedure used for tracing output, or *false* if *cucharout* is to be used instead. Assigning a character consumer to this variable enables trace printing to be separated from normal printout. For a discussion of the standard tracing capabilities of POP, see Chapter 4.

pop_buffer_charout

Setting this boolean variable to *false* causes output to a terminal to appear immediately, rather than being stored in a buffer until a *newline* or *termin* is output. A full description is given in Chapter 23.5.

20.3 Raw Mode Repeaters/Consumers

The term 'raw mode' is used in Unix to mean the setting of a terminal device handler [1] so that a program has direct access to the stream of characters coming from and going to the terminal. On workstations, like the SUN, terminal emulators are provided, which can make a window behave the same way as a conventional VDU terminal. Normally input from a terminal is 'cooked' by the operating system. For example text is stored up in a 'line buffer' until a *return* character is received, and only then handed on to a user program like POP. This permits the operating system to provide 'line editing' capabilities, so that, for example, you can use the *delete* key to rub out mistakes.

However many user programs offer more sophisticated editing facilities. Both the POPLOG VED editor and the EMACS editor are examples of this. These programs want their character input 'raw' so that they can decide what to do with each character as it is typed. Having received a character, they then immediately update the display that the user sees by outputting a 'raw' character or characters.

The procedures described below allow you to raw access to the terminal.

$rawcharin() \rightarrow c$

This repeater provides raw mode input. It reads characters from the device *popdevraw* which is described in Chapter **??**.

$rawcharout(c)$

[1] That is the part of the operating system program that handles communication to and from a terminal

This consumer provides raw mode output by writing characters to the device *popdevraw* as described in Chapter **??**.

$rawoutflush()$
This flushes *popdevraw*, i.e. writes out any outstanding characters in the buffer. It is the same as $sysflush$(*popdevraw*), as described in Chapter **??**.

charin_timeout$(t_{100}) \rightarrow c$
This procedure returns the first character read from the terminal with *rawcharin* during the time-span t_{100}, or $false$ if none were available. t_{100} is measured in hundredths of a second, but in Unix is rounded to the nearest whole second. It is implemented in terms of *pop_timeout_secs* and *pop_timeout*, which are described in Chapter **??**.

20.4 Creating New Repeaters/Consumers

discin$(Filename) \rightarrow P_{crep}$
This procedure returns a character repeater for the filename $Filename$, which may be a string or word. If it is a word then *pop_default_type* is appended to the name (see below). Note that, despite the name of this procedure, $Filename$ is not restricted to being a disk file; it may be any suitable operating system device. A mishap will result if the specified file cannot be opened. $Filename$ may also be a device record open for reading, in which case a character repeater for that device is returned.

discout$(Filename) \rightarrow P_{cc}$
This procedure returns a character consumer for the filename $Filename$. The conventions for the file name are the same as those described for for *discin* above. If $Filename$ is a disk file, then a new file is created, otherwise the file is simply opened; a *mishap* will result if the specified file cannot be created or opened.

$Filename$ may also be a device record [2] open for writing, in which case a character consumer for that device is returned.

$discappend(Filename) \to P_{cc}$
This opens an existing file, and returns a character consumer P_{cc} that will append characters to the file, that is to say it will actually update an existing disk file. The conventions for $Filename$ are the same as for $discin$.

$pop\_default\_type$
The string in this variable, whose default value $'.p'$, is used as the default file type/extension by $discin$, $discout$ and $discappend$, and is concatenated onto a $word$ argument given to these procedures, but not onto a $string$ argument.

$sysisprogfile(Filename) \to b$
Where $Filename$ is a string, $b = true$ if $Filename$ has the file type/extension given by $pop\_default\_type$, $false$ otherwise.

$stringin(\mathbf{s}) \to P_{crep}$
This procedure returns a character repeater for the string **s**, i.e. a procedure which each time it is called produces the next character from the string, and $termin$ when the string is exhausted.

$isclosed(O, b_{mishap}) \to b$
$isclosed(O) \to b$

If O is a device, then $b = true$ if the device is closed, $false$ if it is still open.

If O is a character repeater produced by $discin$ or $stringin$, $b = true$ if the repeater is exhausted, i.e. would return $termin$ if called. If characters are still available from O, then $b = false$.

[2] See Chapter **??**

If O is a character consumer produced by $discout$, $b = true$ if $termin$ has been given to the consumer.

The b_{mishap} argument is a boolean controlling what happens if O is not one of the above: if $false$, the procedure returns $undef$, otherwise a mishap occurs. The b_{mishap} argument defaults to $true$ if omitted.

From Robin Popplestone Mon Dec 19 10:29:49 EST 1988

Chapter 21

Printing out Objects

NOTES
I do not understand the explanation of $nprintf$ given in the online manual, and have, I think, corrected it.
Can $format\_print$ be explained by reference to the Common Lisp manual? I am somewhat reluctant to include another 400+ lines of text.

This chapter describes the capabilities in POP for printing out objects in a variety of formats. These capabilities are user-extendable, via the *class_print* mechanism. Additional capabilities, for example printing out numbers according to Roman conventions, are provided in POPLOG through the Common Lisp FORMAT function, which can be called from POP using $print\_format$.

The basic printing procedure in POPLOG is *sys_syspr*, which prints any object in its standard format. While this can be called directly if so desired, the system additionally provides the procedures *pr* and *syspr* as a two-stage mechanism for printing objects in a way that allows dynamic redefinition of the actual printing procedures used.

The mechanism is based on the convention that programs normally print objects using the variable procedure *pr*, which in the first place, can be redefined in any desired way. However, the default value of the variable *pr* is the procedure *syspr*, which prints an object by calling its *class_print* procedure with the object as argument [1]; thus in the second place, the printing of individual data classes can be altered by redefining their *class_print* procedures. Because the default *class_print* of any class is *sys_syspr*, i.e. printing in standard format, the normal sequence of events is therefore:

```
pr(O)
        ---> syspr(O)
                    ---> class_print(datakey(O))(O)
                                ---> sys_syspr(O)
```

It is important to note, however, that to enable the redefinition of printing procedures for given data classes to take effect at any level, *sys_syspr* always calls *pr* to print the sub-components of any data object, e.g. list, vector and record elements. Thus saying that *sys_syspr* 'prints any object in its standard format' is not strictly correct, since the printing of sub-components will depend on *pr*. A *completely* standard printing procedure would be

```
define pr_standard(O)
    lvars O;
    dlocal pr = sys_syspr;
    sys_syspr(O)
enddefine;
```

i.e. one that locally redefines *pr* to be *sys_syspr*.

Objects are actually printed by *sys_syspr* by passing each character in its printed representation to the standard character consumer *cucharout*. See Chapter 20 for a description of character stream I/O (including formatting of line output, etc).

Chapter 4.5 describes procedures to print *mishap* and *warning* messages.

[1] See Chapter 3.13

21.1 Standard Printing Procedures

sys_syspr(O)
This is the basic printing procedure in the system: it prints any object O in its standard format, calling *pr* to print the sub-components of data objects, e.g. list, vector, and record elements. Printing characters produced are passed to *cucharout*.

Certain aspects of the way *sys_syspr* prints things are controlled by the *'pop_pr_'* variables below.

pop_pr_quotes
This boolean variable, whose default value is *false*, determines how strings are printed by *sys_syspr*. If it is *true*, *sys_syspr* will print them enclosed in string quotes "'"; otherwise, strings are printed without any decoration.

pop_pr_radix
This variable, whose default value is 10, contains an integer controlling the base to which all kinds of numbers are printed by *sys_syspr*. Thus a value of 2 will cause them to be printed in binary, 16 in hexadecimal, etc. Allowable values are 2 – 36, the letters A – Z being used on output as numerals for digit values of 10 – 35.

pop_pr_ratios
This boolean variable, whose default value is *true*, controls the printing of ratios by *sys_syspr*. If it is *true*, ratios are printed as ratios in the form < *n num* > _/ < *n denom* > which can be input by the standard POP itemiser[2]. In this form < *n num* > is the integer numerator and < *n denom* > the integer denominator. Otherwise, ratios are printed as though they were floating-point numbers.

pop_pr_places

[2]See Chapter 19

The bottom 16 bits of the integer in this variable, whose default value is 6, specify the maximum number of fractional places to which floating-point numbers are printed by *sys_syspr*; numbers printed are rounded to this many places. A value of 0 causes them to be printed as integers.

Note that this is normally the *maximum* number of fractional places output, in the sense that trailing zeros in the fractional part are truncated. However, if *pop_pr_places* contains a non-zero value above the bottom 16 bits, this value is taken to be a padding character to be output in each place containing a trailing zero. E.g.

```
(`\s` << 16) || 6 -> pop_pr_places
```

will ensure that 6 places are always produced, with trailing zeros replaced by spaces.

pop_pr_exponent
This variable, whose default value is $false$, controls in which format *sys_syspr* prints floating-point numbers. If it is $false$, numbers are printed in normal format; if it is $true$, printing is in exponent format, i.e in the form

```
<n>.<digits f>e<sgn><digits e>
```

where $1 \leq n < pop\_pr\_radix$. There are a maximum of *pop_pr_places* fractional digits $< digits\ f >$ after the dot. $< sgn >$ is the sign of the exponent, whose magnitude is printed as $< digits\ e >$. The exponent is always printed in base 10.

The value of *pop_pr_exponent* can also be an integer, the bottom 16 bits of which specify the field width for the exponent $< digits\ e >$, which is then padded on the left to this width, the padding character being taken from the bits above the bottom 16 if this is non-zero, or defaulting to the character $'0'$ otherwise.

$syspr(O)$
This procedure does

$$class\_print(datakey(O))(O)$$

i.e. apply the *class_print* of the data class of O to O. The default *class_print* of every data class is *sys_syspr*, but this can be redefined as desired. See Chapter 3.13 for information about data-classes.

$pr(O)$
This variable procedure is used by all procedures in the system to print an object, so that any user-assigned value will take effect across all printing. Its default value is *syspr*.

$npr(O)$
This prints O followed by a newline, i.e.

$$pr(O),\ cucharout(`\backslash n`);$$

$spr(O)$
This prints O followed by a space, i.e.

$$pr(O), cucharout(`\backslash s`);$$

$ppr(O)$
If O is a list, this prints O 'flattened', i.e. with L and all its sublists without list brackets. If O is not a list, it does spr(O). It is defined as

```
if ispair(O) then
    applist(O, ppr)
else
    spr(O)
endif;
```

$sp(n)$
$tabs(n)$

$nl(n)$

These procedures respectively output n spaces (ASCII 32), tabs (ASCII 9) and newlines (ASCII 10) to *cucharout*.

$quote\_pr(O)$
This prints O using pr, surrounded by the appropriate POP-11 quote characters if O is a word or string, i.e. single quotes $'$ for strings, double quotes $''$ for words.

$printlength(O) \rightarrow n$
This procedure returns the number of characters that $pr(O)$ would output to *cucharout*.

$outcharitem(P_{cc}) \rightarrow P_{Oc}$
Given a character consumer procedure P_{cc}, such as *charout*, or one returned by *discout*, this procedure returns an object print consumer procedure P_{Oc}, i.e. a procedure which when given an object O will do $pr(O)$ with *cucharout* locally set to P_{cc}.

21.2 Printing Items off the Stack

$sysprarrow(b_{all})$
This is the procedure called by the POP-11 print arrow $\Rightarrow$. It prints the string in $pop\_ => \_flag$ (q.v.) and then either

- Prints and removes from the stack one object only, if b_{all} is $false$;
- Prints and clears from the stack all objects upto the stack length as it was on entry to *compile*, if b_{all} is true.

Each object is printed with *spr*, printing finishing with a *newline*, i.e. $cucharout(`\backslash n`)$.

$pretty(O)$
This procedure attempts to print objects in a more readable format than *sysprarrow*. It is the procedure used by the POP-11 pretty print arrow ==>. Basically, it prints an object starting with $pop\_ => \_flag$ and ending with a *newline*. However, when an object cannot be printed on a single line, this procedure uses indentation to indicate sub-objects appropriately.

$pop\_ => \_flag$
This variable contains the string to be printed by *sysprarrow* and *pretty* before printing things off the stack. Its default value is $' * * \backslash s'$.

21.3 Formatted Printing

$printf(O_n, \ldots, O_2, O_1, \mathbf{s})$
$printf(\mathbf{s}, L)$

This procedure provides formatted printing, where printable characters in the string **s** may be intermixed with field specifiers that cause the next O argument to be printed at that position.

A field specifier is the character '%' immediately followed by a selector character, which may (currently) be one of the following:

- p - any POPLOG object, printed with *pr*
- P - any POPLOG object, printed with *sys_syspr*
- s - a string printed recursively with *printf*
- c - an integer interpreted as an ASCII character code
- % - output a % character

The characters b, d, i and x are also meaningful, but reserved for system use.

The characters of **s** are scanned from left to right, printable characters being output with *cucharout*, and each field specifier encountered causing the next O argument to be printed as per the specifier; thus the i'th field specifier in the string selects the i'th argument object. In the first form of the call the arguments are taken off the stack one by one, and must therefore be stacked in *reverse* order, whereas in the second form the arguments are supplied in a list L. E.g.

```
printf('The sum of %p and %p is %p.\n', [65 66 131]);
printf(131, 66, 65, 'The sum of %p and %p is %p.\n');
```

both produce the line

```
The sum of 65 and 66 is 131.
```

Note that the first form is incompatible with contexts in which *cucharout* is redefined to leave characters on the stack, because in that case the characters get mixed up with the *printf* arguments. Thus the second form is the *preferred* one if you are defining a printing procedure of any generality. In particular the capability of 'printing into a string' described in section 21.4 will not work if you use the first form.

$nprintf(O_n, \ldots, O_2, O_1, \mathbf{s})$
$nprintf(\mathbf{s}, L)$

This behaves in the same way as *printf*, but followed by a *newline*, i.e.

$$printf(\mathbf{s}, L),\ cucharout(`\backslash n`);$$

$pr\_field(O, n, c_{lpad}, c_{rpad}, P_{pr})$
$pr\_field(O, n, c_{lpad}, c_{rpad})$

This prints O in a field of width n, using the procedure P_{pr} to print the object if this is supplied, or pr otherwise. The object can be left-justified, right-justified, or centred in the field, depending on the the values of c_{lpad} and c_{rpad}, both of which may be an integer ASCII character or $false$.

- If c_{lpad} is a character and c_{rpad} is $false$, the object is right-justified, by being left-padded to the field width with c_{lpad} (or left-truncated if too long).
- Alternatively, if c_{lpad} is $false$ and c_{rpad} is a character, the object is left-justified, by being right-padded to the field width with c_{rpad}, or right-truncated if it is too long.
- If both are characters, then the object is centred in the field, by being left-padded with c_{lpad} and right-padded with c_{rpad} as appropriate, or right-truncated if it is too long.

  Finally, if both are $false$, then c_{rpad} defaults to '$\backslash s$', i.e. the object is left-justified, padded on the right with spaces.

  $prnum(x, n_{int}, n_{frac})$
  This procedure takes any non-complex number x and prints it in floating-point format.

- n_{int} is an integer specifying the number of character positions that the integer part should occupy, including a minus sign if x is negative; this will be left-padded with spaces to the given width.
- n_{frac} specifies the number of positions the fractional part should occupy, including the fractional point; trailing zeros are printed to this width if necessary. If n_{frac} is 1, only the fractional point is printed, if 0 then x is printed as an integer.

This is a library procedure which uses *pr_field* and *pop_pr_places*.

$format\_print(\mathbf{s}, O_{struct})$
This procedure gives the formatted printing capabilities of the Common LISP function FORMAT. Users should refer to Steele [**?**].

21.4 Printing Into Strings/Character Codes

$sprintf(O_n, \ldots, O_2, O_1, \mathbf{s}) \rightarrow \mathbf{s}_{pr}$
$sprintf(\mathbf{s}, L) \rightarrow \mathbf{s}_{pr}$

This is the same as *printf*, described in section 21.3, save that the characters that the latter would print are instead returned as a string.

$O_1 >< O_2 \rightarrow \mathbf{s}_{pr}$
This produces, for any two objects O_1 and O_2, a string which is the concatenation of the printed representations of O_1 and O_2. E.g.

```
'abcd' >< 'efgh'   is   'abcdefgh'
"word" >< 'string' is   'wordstring'
[1 2 3] >< {a b c} is   '[1 2 3]{a b c}'
false >< true      is   '<false><true>'
```

The two objects are 'printed' by using *pr* and redefining *cucharout* to get the printing characters.

$O_1 sys\_ >< O_2 \rightarrow \mathbf{s}_{pr}$
This is the same as ><, but uses *sys_syspr* to 'print' the objects, locally setting all the *'pop_pr_'* variables, described in section 21.1, to their standard values.

$dest\_characters(O) \rightarrow c_n \ldots \rightarrow c_2 \rightarrow c_1$
This prints O with *sys_syspr* (with all the *'pop_pr_'* variables locally set to their standard values), leaving all the character codes on the user stack by redefining *cucharout* to be *identfn*.

21.5 Useful Printing Constants

space
tab

newline

These three constants contain words whose single characters are respectively a space (ASCII 32), a tab (ASCII 9), and a newline (ASCII 10).

Chapter 22

General Communication with the Unix Operating System

NOTES
Is the *popenvlist* re-created when a saved image is restored? see NOTE below.
Can POP be used as a *login* shell??
What is the Host machine in a file name?

This chapter described POP capabilities which allow you to communicate with the Unix [?] operating system. You will need to consult the Unix manual to achieve a full understanding of what is provided, but you may find below is a short glossary of the Unix terms used.

Note that references to the Unix manual are of the form $< function_name >$ ($< section >$), where $< function_name >$ is the name of a C function, which corresponds to a POP procedure. However $< section >$ is *not* an argument, but tells you which part of the Unix manual to look in.

- A *login name* is the name you, or any other user, type when you log in to the Unix system.

- Users are arranged into *groups* in the Unix system. Members of a group can share access to files which are not accessible to people outside the group.

- An *environment variable* is a variable of the Unix system, or more precisely a variable provided by the *shell* program which interprets the commands you give to Unix. For example, to run the POP system you have to give the environment variable *usepop* a value. This is usually done in a file of commands called $'.login'$ which is run when you log in to Unix[1]. Environment variables must not be confused with POP variables.

- A Unix pathname is a string which is the 'full name' of a file, that is it specifies how to start at the Unix root node, and 'climb' the Unix directory tree ending up at the file in question.

- A Unix process is a combination of program and data which is executed by the computer. There are many processes in a Unix system, and their execution is interleaved. This interleaving provides the time-sharing capability of Unix. There is at least one process associated with every user on the system. When you start up a POP system, it is a Unix process, or, in the case of the PWM window manager, two. A Unix process can create an almost identical copy of itself by executing a $fork$ operation. The original process is called the *parent*, and the new one is called the *child*. Both have associated Process Identifiers (PIDs), which are integers. Typically a child will 'decide' to become quite a different process. This it does by an *exec* operation: *exec* specifies that the program the child is obeying (which is initially the same as the parent's program) is to be replaced by a program whose name is an argument to the *exec* operation. When *you* type a command name to Unix, you cause the shell program that is interpreting what you type in to *exec* the program named in the command.

- The 'operations' like *exec* referred to above are accomplished by 'system calls' to Unix. These resemble a procedure call instruction, in that they cause a transfer of control with a return link being left, but they switch the computer into a special 'privileged mode' in which it can do all kinds of things that an ordinary user cannot do, such as write anywhere on the disk it likes. However, a system call can only transfer control into program that ordinary users cannot change, and only to appropriate addresses in that program, so that access to resources which are shared by many users is restricted in such a way that no user can interfere in a disastrous way with resources allocated to another. This, at least, is the intention, but the Unix system is not without loopholes in this cordon sanitaire.

[1]The directory listing program *ls* will not print the names of files beginning with $'.'$ unless you do *ls -a*.

- Parent and child processes can communicate in a variety of ways, which are described in Chapter **??**

These procedures described below perform miscellaneous functions within the POP system, including interfacing to Unix facilities not directly concerned with input/output, which are described in chapter **??**. Some of these are direct interfaces to Unix system calls, etc, while others are in form that can be made compatible across all POP implementations. [2]

22.1 Username and Environment Variable Processing

$sysgetpasswdentry(\mathbf{s}_{user}) \rightarrow \mathbf{v}_{user}$
$sysgetpasswdentry(n_{user}) \rightarrow \mathbf{v}_{user}$

$sysgetpasswdentry(n_{user}, O_{spec}) \rightarrow \mathbf{v}_{user}$

This procedure accesses information about a given user from the password file $'/etc/passwd'$, which is specified in $passwd(5)$ and $getpwent(3)$ in the Unix Programmers Manual. The first argument is either a user $login$ name (the string $\mathbf{s}_{user}$), or a user identification (the integer n_{user}); in all cases, $false$ is returned if the specified user cannot be found.

Without a second O_{spec} argument, the procedure returns a new standard full vector containing the broken-down fields from the password entry line, subscripted thus:

[2]There is a distinct difference in philosophy between POP and Common LISP here. LISP has opted for a uniform external interface. This has the virtue of making LISP code which runs in a LISP implementation under one operating system portable to a LISP implementation which runs under another operating system. This is achieved at the cost of giving the user a view of the operating system which does not match the standard view of that system. The format of file-names, for example, do not change as from LISP implementation to implementation. POP has opted for a rather more hybrid approach. There is a layer of communication which depends only on the operating system in so far as file name conventions differ, described in Chapter **??**. Below this is the layer, described in this Chapter, and in Chapter **??** in which the mapping to a particular operating system is rather specific. There is an advantage to this approach in that POP can be regarded as a way of communicating with Unix which corresponds quite closely with that employed by C. This means that it is straightforward to interpret how the Unix manuals are usable by POP programmers. In general, $'sys'$ is appended to the C name for a Unix procedure to obtain the corresponding POP name

1. A string which is the *login* name of the user, and contains no upper case characters.

2. A string which is the encrypted password of the user. It is encrypted so that if somebody else looks at the entry, he cannot tell what the password is. At least you may hope that that is the case. It is in fact hard to decrypt a well chosen password, but beware, the world abounds in inquisitive people, who exploit the fact that other people often make predictable choices of password. The inquisitive have been known to construct programs which try logging on to machines using selections of the obvious choices. Little jokes along the lines of 'There is a password but its name is secret.' may strike you as original, but they have been around a long time, and are current among the company of the inquisitive.

3. An integer which is user's identifier used by Unix for all its internal operations — for example the ownership of a file is stored in a directory data-structure in the form of this integer, and *not* by the *login* name.

4. An integer which is the identifier of the group to which the user belongs.

5. This is called $'pw\_quota'$, and is always 0 at present.

6. This is called $'pw\_comment'$, and is always an empty string.

7. This string contains data about the user. It begins with his real name, terminated by a comma, and can include his office, phone extension etc.

8. This string is the name of user's initial working directory, that is the directory which he finds himself in after *login* (assuming none of his initialisation files do a *cd* command or the equivalent).

9. This string is the name of a file which is the program to be used as the user's *login shell*, i.e. a program which will be started up to obey his instructions to the computer.

If specified, the O_{spec} argument selects variations on this, as follows:

- If O_{spec} is a standard full vector for which $length(O_{spec}) \geq 2$, then it is used instead of creating a new one. Its fields are updated in accordance with the above scheme, but restricted to the actual number of them, and it itself is returned as the result.

- If O_{spec} is a vector of length 1, containing an integer from 1 – 9, then only the field selected by that integer is returned.

- Finally, O_{spec} may be *true*, in which case the whole of the password entry line is returned as a string. This does not include fields 5 and 6, which aren't actually in the password file.

$sysgetusername(\mathbf{s}_{user}) \rightarrow \mathbf{s}_{name}$
$sysgetusername(n_{user}) \rightarrow \mathbf{s}_{name}$

This procedure uses $sysgetpasswdentry(\mathbf{s}_{user}, 7)$ (or the corresponding second form) to select the user's real name, and then returns the leading substring of this up to the first comma if such exists, or otherwise the whole string. This is assumed to be the part of the string which is the users real name.

$systranslate(\mathbf{s}) \rightarrow \mathbf{s}_{trans}$
$systranslate(\mathbf{s}) \rightarrow false$

This procedure provides a translation of the string **s** from a user *login* name or an environment variable name to a Unix pathname.

1. If the first character of **s** is ˜ (tilde) then the remaining substring of **s** is taken to be a user *login* name. The user's *login* directory is returned as a string (using *sysgetpasswdentry*), or $false$ if the user is non-existent. An empty user name (i.e. $\mathbf{s} =' \tilde{}'$) is interpreted as the current user, and returns $systranslate('HOME')$.

2. Otherwise, the whole string **s** (or **s**, excluding the first character if this is \$ (dollar)), is interpreted as an environment variable name. The strings in the list *popenvlist* are searched, and if one is found of the form $< \mathbf{s} >=< \mathbf{s}_{trans} >$ then the substring $< \mathbf{s}_{trans} >$ is returned, otherwise $false$.

   Note *popenvlist* is *not* recreated when a saved image is restored. (or is it)

popenvlist
This holds a list of the environment variable strings passed to the POP system on startup,

i.e. a list of strings of the form $' < name >=< value >'$ where $< name >$ is the name of an environment variable and $< value >$ is its value. *systranslate* uses this to translate an environment variable to its value.

popusername
This variable is initialised on POP startup to be the string got from $systranslate('USER')$ for Berkeley Unix, or $systranslate('LOGNAME')$ in Unix System V. Thus it contains the *login name* of the current user.

22.2 Filename Processing

In the procedures described in this section, $\mathbf{s}_f$ denotes a string which is intended to be the name of a file.

$sysfileok(\mathbf{s}_{file}) \rightarrow \mathbf{s}_{f\,trans}$
This procedure is called by most procedures in POP which take a file-name argument; it performs the following translations on filenames: First, if the filename $\mathbf{s}_{file}$ begins with the character or the character \$, then the substring from the that character up to the next character before a / (or to the end of the name if there isn't one), is replaced by *systranslate* on that substring. This means that environment variables and user names can be used at the beginning of filenames given to POP. E.g.

```
sysfileok('~johng/ref') =>
** /cog/johng/ref

sysfileok('$popautolib/nl.p') =>
** /poplog/pop/lib/auto/nl.p
```

In addition, in System V POPLOG, *sysfileok* truncates filenames longer than 14 characters. This is done in such a way that any file 'extension' or trailing hyphens on a 'back' file name are preserved, e.g.

```
sysfileok('$popautolib/cleargensymproperty.p-') =>
** /poplog/pop/lib/auto/cleargensym.p-
```

$\mathbf{s}_{dir}$ $dir\_><$ $\mathbf{s}_{f1} \rightarrow \mathbf{s}_{f2}$

This procedure concatenates a directory pathname $\mathbf{s}_{dir}$ onto a filename $\mathbf{s}_{f1}$. Both arguments may be either a word or a string; the result $\mathbf{s}_{f2}$ is their concatenation (using $sys\_><$), with a "/" added in between if $\mathbf{s}_{dir}$ doesn't already end with one and $\mathbf{s}_{f1}$ doesn't already start with one. E.g.

```
'$popautolib' dir_>< 'sort.p' =>
** $popautolib/sort.p
```

$sysfileparse(\mathbf{s}_{file}) \rightarrow \mathbf{v}$

Given a file name $\mathbf{s}_{file}$, this procedure translates it through $sysfileok$, and then returns a standard full vector of (possibly empty) strings containing the broken-down fields of the name at the following subscripts:

1. Host machine
2. Disk: this is provided for VMS compatibility, but in Unix is always an empty string.
3. Directory pathname: the path to the directory the file belongs to.
4. File name: this is the substring of the Unix file-name which comes before the first '.' in the string.
5. File type/extension: this is the substring of the Unix file-name which comes after the first dot. POPLOG systems, particularly the VED editor, use this to distinguish the language a file is written in, e.g. an extension of '*.pl*' is used to indicate a Prolog file.
6. Version: This is the substring which consists of zero or more trailing hyphens. E.g., if $\mathbf{s}_{file} =' Pooh.p--'$ then this field will be $'--'$.

$sysfilehost(\mathbf{s}_{file}) \rightarrow \mathbf{s}_{host}$
$sysfiledisk(\mathbf{s}_{file}) \rightarrow \mathbf{s}_{disk}$

$sysfiledir(\mathbf{s}_{file}) \rightarrow \mathbf{s}_{dir}$

$sysfilename(\mathbf{s}_{file}) \rightarrow \mathbf{s}_{name}$

$sysfiletype(\mathbf{s}_{file}) \rightarrow \mathbf{s}_{ext}$

$sysfileversion(\mathbf{s}_{file}) \rightarrow \mathbf{s}_{version}$

These procedures all call $sysfileparse$ on $\mathbf{s}_{file}$, and then return $\mathbf{v}(1)$ to $\mathbf{v}(6)$ respectively.

$sys\_file\_match(\mathbf{s}_{f\,spec}, \mathbf{s}_{def}, \mathbf{v}, b_{strip}) \rightarrow P_{f\_rep}$

This procedure can be used to find all filenames that match a given file specification $\mathbf{s}_{f\,spec}$. It returns a filename repeater, i.e. a procedure which each time it is called produces the next actual filename matching $\mathbf{s}_{f\,spec}$, or $termin$ if there are no more. The arguments are as follows:

$\mathbf{s}_{f\,spec}$ This is the file specification string. It is first run through $sysfileok$, which means that any initial $\$ < environment\_var >$ is translated, as is an initial $< username >$; it can then contain normal shell-type wildcard characters as follows:

```
            *           match 0 or more characters
            ?           match any single character
  [abcA-Z]    match any of a,b,c or A-Z, etc
  [^abcA-Z]   match any character but a,b,c or A-Z, etc
```

As usual, $'.'$ at the beginning of a name part must be matched explicitly.

In addition, there are two other special wildcards: the first is concerned with the POP convention of naming 'back' files by appending 1 or more trailing hyphens, described under *pop_file_versions* in Chapter 23.3.

matches the start position of 0 or more trailing '$-$' characters on the end of a filename.

Thus for example, $'*\#'$ would match only filenames with no trailing hyphens, whereas $'*\#--*'$ would match names with at least two trailing hyphens. The other special wildcard is $'\ldots'$ for a directory name, which means match all sub-directories recursively, e.g.

```
'/foo/.../baz/*.p'
```

would match all $*.p$ files in any directories called $'baz'$ anywhere in the tree from $/foo$ downwards.

$\mathbf{s}_{def}$ The file specification $\mathbf{s}_{f\,spec}$ is considered to have 3 parts, namely pathname, filename and version, the last being a # character plus anything following it; if present, the corresponding parts of the default specification string $\mathbf{s}_{def}$ are then used to fill in the missing parts of $\mathbf{s}_{f\,spec}$. For example, if $\mathbf{s}_{f\,spec}$ was $'*.ph'$ (having only the filename part), and $\mathbf{s}_{def}$ was $'\$usepop/\ldots/*.p\#'$ (having all three parts), then the actual specification used would be

```
'$usepop/.../*.ph#'
```

$\mathbf{v}$ This is a standard full vector or $false$. If it is a vector then $sys\_file\_stat(\mathbf{v})$ is evaluated with each filename $\mathbf{v}$ generated [3]. When a vector is specified, the repeater then returns a second result for each filename produced, i.e.

$$P_{f\_rep}() \to \mathbf{s}_{file} \to O_{stat}$$

which is the result that would be produced by sys_file_stat, i.e. the vector $\mathbf{v}$ or $false$. If $\mathbf{v}$ is $false$ then only the filename is returned, i.e.

[3] *sys_file_stat* in Chapter **??**

$$P_{f\_rep}() \rightarrow \mathbf{s}_{file}$$

b_{strip} When $b_{strip} = false$, the names returned by the repeater are the full names, with pathname included in each. With $b_{strip} = true$ however, the pathname is stripped from each individual name and returned in between the stream of filenames whenever the pathname changes. When this happens, the repeater returns $false$ for the filename and a second result for the pathname. Thus when b_{strip} is $true$, the repeater must be used in something like:

```
until (P_filename_rep() ->> FILENAME) == termin do
    if FILENAME then
        <process filename, -> STAT if $\popvector$ a vector>
    else
        -> PATHNAME;
        <process pathname>
    endif
enduntil;
```

$systmpfile(\mathbf{s}_{dir}, \mathbf{s}_{pre}, \mathbf{s}_{post}) \rightarrow \mathbf{s}_{file}$
This procedure generates a new unique file name in the given directory. If $\mathbf{s}_{dir}$ is $false$, it uses the default temporary directory $'/tmp'$; if $\mathbf{s}_{dir}$ is an empty string, it uses the current directory. $\mathbf{s}_{pre}$ should be a string which identifies the program which needs the temporary file, and $\mathbf{s}_{post}$ should be the extension required. An example is:

```
systmpfile('/foo', 'load', '.o') =>
** /foo/load6x21564.o
```

22.3 Directory Manipulation

current_directory
This active variable holds the current directory as a string: assigning to it changes the current

Unix directory of the process.

popdirectory
This variable is initialised on POPLOG startup to be the string got from

$$systranslate('HOME')$$

i.e. the current user's home directory.

$syslink(\mathbf{s}_{f,old}, \mathbf{s}_{f,new}) \rightarrow b$
This procedure creates a link named $\mathbf{s}_{f,new}$ to the file named $\mathbf{s}_{f,old}$, where $\mathbf{s}_{f,old}$ and $\mathbf{s}_{f,new}$ are both strings. If the call fails merely because $\mathbf{s}_{f,old}$ doesn't exist then $false$ is returned, but any other failure produces a *mishap*. *true* is returned if the link was successfully created.

$sysunlink(\mathbf{s}_{file}) \rightarrow b$
This procedure unlinks the directory entry for the file named $\mathbf{s}_{file}$, where $\mathbf{s}_{file}$ is a string. If the call fails merely because $\mathbf{s}_{file}$ doesn't exist then $false$ is returned, but any other failure produces a *mishap*. *true* is returned if $\mathbf{s}_{file}$ was successfully unlinked.

$sysdelete(\mathbf{s}_{file}) \rightarrow b$
This procedure deletes the file with filename string $\mathbf{s}_{file}$, returning *true* if successful, $false$ if not. This procedure does the opposite of *syscreate* in that it 'moves forward' any back file versions of $\mathbf{s}_{file}$ that exist, as described in $pop\_file\_versions$ in Chapter **??**.

22.4 Unix Processes

The procedures in this section allow you to create new Unix processes. They are thus conceptually similar to the POP processes described in Chapter **??** but are created and administered by the Unix operating system at the request of POP, whereas POP processes are treated entirely by POP. Unix processes use more machine resources to create, but offer you access to Unix capabilities that POP processes do not. In particular, one way in which you can use a 'foreign' program that is not written in POP, or in any of the POPLOG 'native' languages, is to have POP invoke it as a process, using $sysfork$ (or $sysvfork$) and *sysexec*, as described below. This can usually be done given just the user manual for the foreign program, since the POP system can then 'talk' to the foreign program just as if the POP system were a human user. This 'talking' is accomplished by switching the input and output channels of the child process that is going to become the foreign system, using the active variables *popdevin*, *popdevout* and *popdeverr* to reassign standard input and output as described in Chapter **??**.

Another way of accomplishing the same end can be to use the *external* capabilities of POP described in Chapter **??**. However to do this you will require detailed knowledge of the implementation of parts of the foreign program.

$sysfork() \to n_{PID}$
$sysfork() \to false$

This forks the current POP process, producing a new child process which is an exact copy of the current one. In the parent, the call of $sysfork$ returns with the child process identifier, n_{PID}; in the child it returns $false$. This is the *only* difference manifest between the parent and child.

$sysvfork() \to n_{PID}$
$sysvfork() \to false$

This procedure forks the current POP process, producing a new child process which 'borrows' the address space of the current one until a *sysexit* or *sysexecute* is performed; until this happens, only the open files of the two processes are different.

$sysvfork$ must therefore be used *only* in the situation where the purpose of forking is to immediately *sysexecute* another image in the child, possibly after redirecting the standard input and/or output etc. It is quicker than $sysfork$ because Unix doesn't have to copy the whole POP process. Using it in any other way will crash the system – in particular, the procedure that calls $sysvfork$ must not exit when running as the child. This is the same limitation placed on the use of the $vfork$ system call in C.

In the parent, the call of $sysvfork$ returns with the child n_{PID}; in the child it returns $false$.

$syswait() \rightarrow n_{PID}$
This procedure waits for the termination of child process(es) created by $sysfork$ or $sysvfork$, returning the Process IDentification number of a dead child.

Each call of *syswait* either returns the n_{PID} of a child that has already died, or waits for the death of one to do so. A *mishap* will result if there are no child processes, or none that have not already been waited for.

syswait can also return $false$, if an interrupt (e.g. Ctrl-C or timer) occurred while waiting for a death. The normal way of ensuring that a given child whose process identifier is n_{PID} has died is therefore a loop like

```
until syswait() = PID do enduntil
```

pop_status
This variable is set to the integer valued exit status of a child that has just been waited for with *syswait*. Since utilities like *sysobey*, etc also use *syswait* this variable will also contain the exit status of commands run with that procedure.

$sysexecute(\mathbf{s}_{file}, L_{arg}, L_{env})$
This does a Unix *execve* system call, i.e. runs the executable file named by the string $\mathbf{s}_{file}$ in place of the current POP image, passing it the strings in L_{arg} as arguments, and the strings in L_{env} as environment variables. If $L_{env} = false$ then it is replaced by the current value of *popenvlist* Note that all strings are null-terminated before being passed across and that the list L_{arg} begins with the zeroth argument, i.e., that usually used for the name of the program.

The combined use of *sysvfork* and *sysexecute* to start up a 'foreign' process is illustrated below in the case where *pipes* are used as the communication channels. These are first created by calling the procedure *syspipe*, described in Chapter 23.3. Next the *sysvfork* procedure is called. After this, there are two Unix processes, both running POP, indeed both running the code following the *sysvfork*. Both of these will start executing the conditional, but, unless you have a true parallel processor like the Sequent Symmetry, at different real times. The parent, which has the n_{PID} of its child, will execute the *return*. It will 'talk' to its child by making explicit use of the pipes. The child, for which $n_{PID} = false$, will switch the input and output so that the pipes become the standard input and output channels.

```
lvars n_PID, pipe_1_in, pipe_1_out, pipe_2_in, pipe_2_out;
syspipe(false) -> pipe_1_in -> pipe_1_out;   ;;; The parent to child pipe
syspipe(false) -> pipe_2_in -> pipe_2_out;   ;;; The child to parent pipe
sysvfork() -> n_PID;
    if n_PID then                          ;;; Am I the parent?
                                           ;;; Yes, close unneeded pipe ends.
      sysclose(pipe_1_out);                ;;; These ends of the pipe are
      sysclose(pipe_2_in);                 ;;; used by baby.
      return
    else                                   ;;; I am the child.
      pipe_1_in  -> popdevin;              ;;; Take input from parent
      pipe_2_out -> popdevout;             ;;; Send output to parent
      sysclose(pipe_1_in);                 ;;; These ends are used only by the
      syscload(pipe_2_out);                ;;; parent, so close them
      sysexec('foreign_program')           ;;; and become a changeling.
    endif;
```

poppid
This contains the Process Identifier of the current POP process.

22.5 Running Shell Commands

$sysobey(\mathbf{s})$
$sysobey(\mathbf{s}, c_{shell})$

$sysobey(\mathbf{s}_{file}, L_{arg})$

The first two forms obey the string **s** as a shell or cshell command, by forking a child process. The integer c_{shell} is an ASCII character code controlling which shell is used, as follows:

```
‘$‘     /bin/sh
‘%‘     /bin/csh
‘!‘     the value of systranslate('SHELL')
```

If c_{shell} is absent, */bin/sh* is assumed.

The third form forks a child, *sysexecute*s the file $\mathbf{s}_{file}$ with argument strings L_{arg}, and then *syswait*s for the child. The status return from the child process is placed in *pop_status* by *syswait* (qv).

$sysobeylist(L_{comm})$
Given a list of commands strings L_{comm}, this procedure runs

$$sysobey(\mathbf{s}, \text{‘!‘})$$

on each **s** in the list.

22.6 Signal Handling

$sys\_send\_signal(n_{PID}, n_{sig}) \rightarrow b$
This sends the signal number n_{sig} to the process whose Process Identification number is the integer n_{PID}, returning $true$ if successful. $false$ is returned if the specified process does not exist, or you don't have the privilege to send that signal to it.

$sys\_reset\_signal()$
This restores POP signal handling for those signals normally trapped by the system. It may need to be done after an external procedure has altered the signal-handling, which typically occurs when a complex system is externally loaded, which has its own ideas about how signals should be handled. Otherwise it is unlikely that you will need to do it.

From Robin Popplestone Thu Dec 8 10:28:18 EST 1988

Chapter 23

Unix-related input and output

NOTES rewrite *pop_file_versions* explanation.

This chapter describes those input and output (I/O) facilities of POPLOG running under Unix which provide the closest accessibility to Unix. It is thus complementary to Chapter 22 which deals with procedures that are less directly related to I/O, and describes the capabilities which underlie the character-stream procedures of Chapter 20.

All input and output in POPLOG is via *device records*. These are created for a file using the procedures *sysopen*, *syscreate*, described in section 23.3. The concept of 'file' in Unix has been generalised to include peripheral devices, which are to be found in the directory */dev*. These are used in a similar way to files, subject to the constraints imposed by the physical nature of the device. However there is a procedure, called *ioctl* in C and *sys_io_control* in POP, which is used to change the state of these peripherals. How you call it will depend upon the specific device. POP devices correspond to Unix *channels*.

Every Unix process has three 'standard channels', called the input, output and error channels which have the integer identifiers 0, 1 and 2. These are the channels by which that process communicates by default. When a process is created by a *fork* command, as discussed in Chapter 22, channels are inherited from the parent.

For communication between Unix processes the procedure *syspipe* can be used to create device records corresponding to a Unix *pipe*. POPLOG does not currently provide sockets, although a user can employ the external load capability of POPLOG.

As far as possible, the POP I/O facilities have been designed to provide procedure calls which match the Unix interface which is specified for the C language. However, in order to provide compatibility with the capabilities of the VMS system, the above procedures take an additional argument to the corresponding C functions which specifies the 'organisation' of the file.

Terminal devices, whether they be a real VDU or a terminal emulator in a window system, are rather complicated to deal with, because they offer a range of capabilities not all of which will be available on every terminal. POPLOG allows you to associate more than one device with a given terminal. Each such device maintains a full set of *terminal parameters*. These are attributes of the *driver* program which is associated with the terminal in Unix, and include information about whether the user program is to be given every character the user types as soon as he types it, or whether characters are to be held in a 'line buffer' and only relayed to the user program when a complete line has been input. Whenever such a device is used by a POPLOG program, POPLOG makes sure that the terminal parameters have the right value for that device, resetting them if necessary. In this way multiple devices with different characteristics can be made to work for the same underlying terminal, with automatic switching between different settings as appropriate. See *sys_io_control* in section 23.6 for more information.

Interfaces to other Unix system calls are provided where appropriate, wherever possible in a form that can be made compatible across all POPLOG implementations.

23.1 Predicates on Devices

$isdevice(O) \rightarrow b$
This procedure returns $true$ if O is a device, $false$ if not.

$systrmdev(Dev) \to b$
This procedure returns $true$ if the device Dev is a terminal, $false$ if not.

$isclosed(Dev) \to b$
$isclosed(P_{rep}) \to b$

If O is a device, this procedure returns $true$ if the device is closed, $false$ if it is still open. This procedure is also applicable to character repeaters produced by *discin*, as described in Chapter 20.

23.2 Device/File Information

$device\_open\_name(Dev) \to \mathbf{s}$
This procedure returns the *open name string* of the device Dev, i.e. the name with which the device was opened/created.

$device\_full\_name(Dev) \to \mathbf{s}$
This procedure returns the full name string of the device Dev, i.e. the open name with any environment variable components etc translated.

$device\_os\_channel(Dev) \to n$
This procedure returns the Unix integer file descriptor associated with the device Dev.

$sys\_file\_stat(File, \mathbf{v}) \to \mathbf{v}$
Where $File$ is either a string naming a file, or a device record for an open file, this procedure puts information about the file in the standard full vector **v**, returning **v** as the result. $false$ is returned if the file is nonexistent or is not a disk or tape file, or cannot be opened due to a protection violation. The information returned in each subscript position of the vector **v** is as follows:

1. Size of file in bytes
2. Last modified time (t_{mod})
3. Group id of owner
4. User id of owner
5. Mode flags
6. Number of links
7. Major/minor device
8. I-node number
9. Last accessed time (ATIME)
10. Last status change time (CTIME)

Further details are given under $stat(2)$ in the Unix Programmers Manual for further details.

Note that all the values are integers or bigintegers. If the length of $\mathbf{v}$ is less than 10, only the information that will fit in is given, e.g. if the length is 1, only the size is given, if the length is 2 then the size and the last modified time, etc.

$sysfilesize(File) \rightarrow n$
This procedure returns the size in bytes n of the file represented by the device record or file name string $File$ (which must be either a disk or tape file). This is the same as $\mathbf{v}(1)$ from sys_file_stat.

$sysmodtime(File) \rightarrow t_{mod}$
This procedure returns the last modified time of the file represented by the device record or file name string $File$, which must be either a disk or tape file. This is the same as $\mathbf{v}(2)$ from sys_file_stat.

23.3 Opening and Closing Devices

The procedures *sysopen*, *syscreate* and *syspipe* described below all take an argument O_{org} to specify the organisation of a file. This argument can currently take the following values:

- $false$
  For disk files and pipes, this value will optimise the device for single character input or output, otherwise there is no difference between $false$ and $true$. For terminals, this gives normal interactive line mode with prompts given by *popprompt*.
- the word *"line"* (or *"record"*)
  For all kinds of device, this value will mean that a *sysread* of n bytes from the device will only only read up to the next newline character, e.g. $sysread(Dev, BUFF, 512)$ will read the next line and return the number of characters read.
- $true$
  For a disk file or a pipe, the device is optimised for reads and writes of more than 1 byte at a time, otherwise there is no difference between $true$ and $false$. For terminals, the device is initially set up for 'rare' mode (i.e. *cbreak*, *−echo*, *−nl*, *−tabs*) and no prompt is output for read operations. See *sys_io_control* below.

Essentially then, use $false$ for character stream I/O on text files, *"line"* for line input on text files, and $true$ for block I/O on disk files or pipes and rare or raw mode on terminals.

Note that all the procedures below which take a $\mathbf{s}_{file}$ argument first translate the given name with $sysfileok$, which is described in Chapter 22.

$sysopen(\mathbf{s}_{file}, m_{access}, O_{org}) \rightarrow Dev$
This procedure returns a device record *Dev* for the filename, which as we noted above can include various peripherals in the */dev* directory, named by the string $\mathbf{s}_{file}$, opened for access mode m_{access} with organisation O_{org}. Permissible values of m_{access} are

0	Read only
1	Write only
2	Read and Write

Permissible values for O_{org} are as described above. If the file does not exist, and so cannot be opened, then $false$ is returned. If it cannot be opened for any other reason, including reference within ${}_{file}$ to a non-existent directory, a *mishap* occurs.

readable$(\mathbf{s}_{file}) \rightarrow Dev$
This procedure returns a device record *Dev* for $\mathbf{s}_{file}$, opened for reading with $false$ for the organisation argument O_{org}. If for any reason the file cannot be opened, $false$ is returned. I.e. this procedure will never cause a $mishap$.

syscreate$(\mathbf{s}_{file}, m_{access}, O_{org}) \rightarrow Dev$
This procedure returns a device record *Dev* for $\mathbf{s}_{file}$, created for access mode m_{access} with organisation O_{org}. The values of m_{access} and O_{org} are as for *sysopen*. A *mishap* results if for any reason the file cannot be created. This procedure uses the Unix *creat* system call, qv.

pop_file_mode
This integer variable supplies the access permissions mode given to the Unix *creat* system call by *syscreate*. See *chmod*(2) in the Unix Programmers Manual for possible values; the default value is 8:664, i.e. permission to read and write the file is given to both the *owner* and members of the *group* to which the file belongs. Any body else in the *world* can only read the file.

Note that this variable is only used by *syscreate* when creating a *new* file: if a file exists already, its access permissions are left unchanged.

pop_file_versions
This integer variable controls the creation of 'back' file versions by *syscreate*. POPLOG uses a convention that 'back' versions of a disk file are named by suffixing the original file name with one or more '$-$' characters. E.g. the most recent back file of $'foo'$ is $'foo-'$, the next most recent is $'foo--'$, and so on.

The action of *syscreate* is therefore to try to maintain *pop_file_versions* of a file. E.g. 2 means the original file plus 1 back version, 3 means the original plus 2 back versions, etc. This is done by 'moving back' all existing versions up to *pop_file_versions*, the oldest one being deleted. The operation of 'moving back' a file depends on whether the file has only 1 link, or more than 1: in the former case the file is simply renamed as the 'back' name, while in the latter case (to preserve the links), the file is copied to a new one of that name.

The procedure *sysdelete*, described in Chapter 22, also uses this variable to 'move forward' old versions when deleting a file. That is, if back versions of the file within the range *pop_file_versions* exist, then the above process is reversed to bring the back versions forward.

$syspipe(O_{org}) \to Dev_{in} \to Dev_{out}$

This creates a Unix pipe and returns input and output device records for it. Permissible values for O_{org} are as described above. Note that the 'in' and 'out' directions are from the point of view of the program using the pipe, and not the pipe itself — what comes *out* of a pipe is what goes *in* to a program. A worked example of the use of pipes is to be found in Chapter 22.4.

$sysclose(Dev)$

This closes the device Dev. If Dev was open for output device, then any buffered characters will be written to the physical peripheral. Note that two or more Unix processes can share a device, which one of them may close without closing it for others. Note also that POPLOG automatically closes all garbage collected device records. In addition, it closes all devices on system exit.

23.4 Reading from Devices

$sysread(Dev, i, O_{byte}, n) \to n_{read}$
$sysread(Dev, O_{byte}, n) \to n_{read}$

This reads up to n bytes from the device Dev into the structure O_{byte} starting at byte subscript i, and returns as result the actual number n_{read} of bytes read.

In general, n_{read} will be n for disk/tape, except possibly near or at end-of-file; for terminals and pipes it will be whatever is available, depending on the value of the O_{org} argument when the device was opened. A result of 0 bytes read indicates end-of-file. The structure O_{byte} must be *byte accessible*, a full explanation of which is given in Chapter 3.9. The bytes are read into the structure O_{byte} starting at byte subscript i, the first applicable byte of the structure having subscript 1, which, as explained in Chapter 3.9 is the first byte after the structure's key.

This means that if O_{byte} is a string or vector, the bytes are read in starting at the 1st component; if O_{byte} is a record the bytes will occupy fields occurring after the key. In both cases, the structure must be large enough to contain all bytes read.

The second form of the call with i omitted is the same as

$$sysread(Dev, 1, O_{byte}, n)$$

i.e. i defaults to 1.

$getc(Dev) \rightarrow c$
This uses *sysread* to read a single byte c from the device Dev, which must be open for reading. *termin* is returned at end of the file.

popprompt
The value of this variable determines the prompt characters output by *sysread* when reading from a terminal in normal line mode, i.e. when the device is opened with O_{org} argument $false$. It may be either an actual prompt string, or a procedure of no arguments returning one, i.e. $popprompt() \rightarrow \mathbf{s}$

$pop\_timeout() \rightarrow ()$
pop_timeout_secs

If a program tries to read from a terminal device it is possible that the human user of the terminal may fail to type anything. Normally the program will have to wait until he does type something, although of course Unix will suspend the Unix process in which POP is running and give another process the chance to use the computer's CPU. However it is possible to arrange for a *sysread* procedure call, which is how all read operations are done at the bottom level, to 'time out'.

Whether a time out occurs depends on the value of the variable *pop_timeout_secs*. If this has its default value of *false*, then no time-out occurs. Otherwise it must be an integer, and the time-out will occur after a number of seconds specified by this integer, if it is ≥ 0.

When the time-out occurs, the procedure which is the value of *pop_timeout* is called. Its default value is *identfn*, the 'do-nothing' procedure. If *pop_timeout* exits normally, then the *sysread* procedure will return the number of characters read (usually zero). However you can assign a procedure to *pop_timeout* which will cause some other action to occur. For example you might activate another POP process, thus providing time-sharing within POP, although this would only be desirable in specialised circumstances, such as when all the users needed to share a great deal of POP code, and perhaps collaborate closely in some way. The capabilities required for doing this are described in Chapter 12.

Exiting normally from this procedure inside *charin* or *rawcharin*[1] will cause the read to be re-tried if no characters have actually been read before the timeout. Exiting inside *sysread*[2] will cause the number of characters read before the timeout to be returned.

In Unix this capability depends on *syssettimer*, about which see Chapter 18.3, so that it may interfere with other procedures using *syssettimer*[3].

$sys\_input\_waiting(Dev) \rightarrow n$
For a readable 'interactive'-type device (i.e. a terminal or a pipe), this procedure returns an integer count n of the number of input characters currently available to be read on the given device, or *false* if none are available, in which case a read from the device would hang up waiting for input.

[1] See Chapter 20.1
[2] See section23.4
[3] This limitation does not apply to VMS.

For all other kinds of input device, which can't hang up on a read, it just returns the number of characters currently in POPLOG's input buffer for the device (which could be 0).

$sys\_clear\_input(Dev)$
For a readable 'interactive'-type device (terminal or pipe), this clears any input characters currently available to be read on the device, including those 'typed-ahead' on a terminal. If Dev is also writeable, any buffered output is written out first.

For all other kinds of input device, this procedure simply writes out any buffered output if the device is also writeable.

23.5 Writing to Devices

$syswrite(Dev, i, O_{byte}, n)$
$syswrite(Dev, O_{byte}, n)$

This writes n bytes from the structure O_{byte} to the device Dev, starting at byte i after the structure's key, or byte 1 if i is absent. Thus $syswrite$ is similar to $sysread$ except that bytes are written rather than read and that no result is returned. In addition the O_{byte} argument may be a word, in which case, bytes are written out from the sequence of characters held in the word record.

$pop\_buffer\_charout$
This boolean variable, whose default value is $true$, controls whether write operations to normal line mode terminal or pipe devices (i.e. those opened with the argument $O_{org} = false$) are buffered. If $true$, then characters are only actually written out when a control character, that is one such as $newline$ whose value is < 32, is encountered. Otherwise, all characters are written out immediately. Note that all other devices are buffered by default. The procedure $sysflush$, described below, can be used to force data to be written out.

$sysflush(Dev)$
For any writeable device Dev, this flushes (i.e. writes out) any bytes outstanding in POPLOG's buffer for that device. For a disk file, this operation doesn't necessarily guarantee that the current state of the file is actually reflected on disk, because of buffering by Unix.

$pop\_file\_write\_error(Dev)$
If a write error occurs while writing to a disk or tape device with $syswrite$ the device is immediately closed. Usually, with disk files, this is due to a full disk or exceeded quota. The device is then given as argument to this variable procedure, which should take appropriate action with the partly-written file, e.g. to delete it with $sysdelete(device\_full\_name(Dev))$. The default value of this variable is $erase$.

23.6 File Control Operations

$sysseek(Dev, i_{file}, m, true) \rightarrow j_{file}$
$sysseek(Dev, i_{file}, m)$

This procedure controls at which byte in the file the next read or write will operate, dependent on the integers i_{file} and m as follows:

$m = 0$	i_{file} is an absolute byte position within the file (1st byte = 0)
$m = 1$	i_{file} is a byte offset (possibly negative) relative to the current byte (i.e. the next one that would be read or written).
$m = 2$	i_{file} is a byte offset relative to the byte immediately after the last byte in the file.

With an optional 4th argument of $true$, $sysseek$ returns the absolute byte position j_{file} within the file after the seek.

$sys\_io\_control(Dev, n_{req}, O_{byte}) \to b$
$sys\_io\_control(Dev, n_{req}) \to b$

This procedure provides an interface to the Unix *ioctl* system call, and has essentially the same arguments as the latter. *Dev* is a device record, n_{req} is an integer specifying the desired operation to be performed, and O_{byte} is an (optional) byte-accessible structure argument through which data is passed or returned [4]. Except for those n_{req} values which set or get the characteristics of a terminal (described below), a call of *sys_io_control* translates into a straight *ioctl* system call for the file descriptor allocated to Dev, with n_{req} value as supplied and with O_{byte} passed as the address of its byte data (or 0 if O_{byte} is omitted). The result b is *true* if the call succeeds, *false* if not. For values of n_{req} which set or get the characteristics of a terminal, however, *sys_io_control* operates differently (i.e. the values TIOCGETP, TIOCSETP, TIOCSETN, TIOCGETC, TIOCSETC and for Berkeley Unix TIOCLGET, TIOCLSET, TIOCLBIC, TIOLBIS, TIOCGLTC and TIOCSLTC – see Unix Programmers Manual, section *tty*(4)). Every device representing a terminal in POPLOG maintains a full set of terminal parameters which are set on the terminal whenever that device is used (unless it is already set for that device). In this way multiple devices with different characteristics can be made to work for the same underlying terminal, with automatic switching between different settings as appropriate. *sys_io_control* therefore provides the means of controlling this setup, in two ways:

1. with a O_{byte} argument supplied, the selected device characteristics are set or got from O_{byte}. Therefore if new characteristics are set, these will take effect on the terminal only when the device is next used. Although this means that TIOCSETP or TIOCSETN are equivalent at the time of the call, use of one of these will be 'remembered' by the device so that any subsequent automatic switch to this device's parameters will use that code for the switch (TIOCSETP being the default).

2. with no O_{byte} argument, *all* the device characteristics are set or got from the terminal. In this case, all setting codes (TIOCSETP, TIOCSETN, TIOCSETC, TIOCLSET, TIOCSLTC, etc) are equivalent, as are all getting codes (TIOCGETP, TIOCGETC, TIOCLGET, TIOCGLTC), since they all transfer all the parameters. The only exception to this is that the setting code TIOCSETP will flush buffers, etc, whereas any other setting code will not.

With all these calls, the result b is *true*.

[4] See Chapter 3.9 for a definition of byte-accessible structures

sys_link_tty_params(L_{Dev})
This procedure is used to make two or more devices which represent the same actual terminal share terminal parameters, so that changing any one of them with *sys_io_control* automatically affects the other(s).

The argument L_{Dev} is a list of devices; for each set of devices in the list which represent the same actual terminal, all members of that set are changed permanently to use the parameter structure of the first-occurring member of the set. Non-terminal devices in the list are ignored.

set_process_entry_term()
If the standard input is a terminal, this resets the terminal characteristics to be what they were on entry to the POPLOG system. Otherwise it does nothing.

23.7 Standard Devices

popdevin
popdevout

popdeverr

These (active) variables hold device records for the standard input, output and error channels respectively of the process, i.e. Unix file descriptors 0, 1, and 2. The devices are opened initially with O_{org} argument $false$, giving normal mode line I/O for terminals, etc. Where any 2 or all 3 of these devices represent the same terminal, their terminal parameters are linked together with *sys_link_tty_params* (see above), which means that changing the characteristics of one automatically affects the other(s). These 3 devices are used by *charin*, *charout* and *charerr* respectively to perform character stream I/O on the standard channels [5]

Note that assigning a new device to one of these variables will redirect the standard

[5] See chapter 20 for a specification of these procedures.

channel for the process. This could be done e.g. after a *sysfork*, and Chapter 22.4 has an example of this.

poprawdevin
poprawdevout

When either the standard input or the standard output is a terminal, these (active) variables are set up initially to hold reading and writing devices respectively for that terminal, opened with O_{org} argument *true*, to give terminal input and output in 'raw' mode (as described above). The two devices have their terminal parameters linked by *sys_link_tty_params* (see above). If neither standard channel is a terminal, the value of both these variables is an undef record, $<$ *undefpoprawdevin* $>$ and $<$ *undefpoprawdevout* $>$. These will produce the *mishap* 'DEVICE NEEDED' if an attempt is made to use them for I/O. It is therefore advisable to test with *isdevice* first if your program might attempt to use them in this situation.

These are the devices used by *rawcharin* and *rawcharout* respectively, which procedures are specified in Chapter 20.

popdevraw
Prior to the introduction of *poprawdevin* and *poprawdevout*, this (active) variable contained a combined read/write device for 'raw mode' terminal I/O. It is now an autoloadable synonym for *poprawdevin*, but, to maintain upward compatibility, output operations (*syswrite* or *sysflush*) applied to *poprawdevin* will be redirected to to *poprawdevout*.

23.8 Miscellaneous

sysiomessage() $\rightarrow$ **s**
Whenever POPLOG system procedures do Unix system calls which result in an error return from Unix, they leave the Unix error number in an internal variable [6]. *sysiomessage* returns

[6] This corresponds to *errno* in C

an error message string for the value currently in this variable: this can therefore be used after after errors occurring in things like *sysopen*, *syscreate*, *sysread*, *syswrite* etc, to determine what caused the problem. The string has enclosing parentheses added so that it can be concatenated on the end of *mishap* messages.

device_key
This constant holds the key object for device records. Keys are explained in Chapter 3.13.

Chapter 24

Using procedures written in languages other than POP

NOTE
what is aka???
we need info from REF EXTERNAL
The *array_of_int* example - Fortran or Pascal - it is not clear whether resp. is intended

A mechanism for easily declaring, loading and unloading procedures compiled externally to Poplog.

24.1 Introduction

There are occasions when POP-11 is not suitable for a particular computation and the problem could be most easily solved by handing control over to another language and then waiting for the answer to come back. For instance, when many numerical calculations are needed (e.g. multiplying very large matrices, or convolving an image) the most convenient language might be Fortran, or C.

Facilities are provided in the POP-11 kernel for linking and calling such external procedures, but they are difficult to use. This autoloading library attempts to provide an interface for the built in routines, in such a way that the user does not need to know the details of the mechanism for handling external procedures.

The library file also provides a system which checks that actual parameters given to an external procedure conform to the formal parameter specification.

This facility was inspired by the library NAGCALL, by David Young. We are grateful to him for the help he gave in the design of the Fortran interface, and his general comments on the design of an interface for external routines.

Note. It is assumed that the reader is familiar with at least one non-Poplog language, such as C or Fortran. The reader should also have some experience with POP-11, with knowledge of the following data-structures: vectors, arrays, and strings, and be aware of the distinction between single and double precision decimals.

24.2 The syntax of external procedure declarations

The general syntax form for *external* operations is:

```
external <word:W> <word:T>
```

The type of operation performed is determined by the argument W, which should be one of the following:

declare	Introduces a block of language specific declarations.
load	invokes the system linker to fetch all external procedures previously declared. Introduces a block of object file-names.
unload	frees the memory previously occupied by external procedures

Each operation will be discussed in detail.

The word argument T denotes a *tag* which identifies a set of external procedures, and any valid POP-11 word may be used. Thus two sets of *declare* operations with the same tag are taken to refer to a single set of procedures. Also, a *load* operation can selectively load a single set of external procedures according to the tag used.

24.2.1 Declaring external procedures

The *declare* operation has two forms:

(a) language specific:

```
external declare <word:T> in <word:L> ;

;;; body

endexternal
```

The argument T is the tag, and is used to refer to the set of external procedures declared in the body. L is a word which names the language that the block specifies, for example "c" or "$fortran$".

A language-specific parser is then invoked. Its task is to read the declarations found in the body of the statement block, and create POP11 procedures which, when called, will run an external procedure.

(b) Raw import mode:

```
external declare <word:T> ;

;;; body

endexternal
```

The body in this case is a sequence of words which are the linkers names for external procedures. A POP-11 variable of the same name will be created to hold the raw external procedure at *load* time. The syntax

```
<word:P> = <word:S>
```

may also be used to import the linker symbol S to POP-11 variable P.

Here is an example, using C syntax, of importing a procedure which takes two floating point numbers and returns their product:

```
external declare mytag in c;

float multiply(x, y)
float x, y;
{}

endexternal
```

For non-C users, the declaration reads: *multiply* is a function which returns a floating point decimal, and takes two arguments, both of which are floating point decimals.

Notice the use of {} to indicate the end of a declaration. This is a feature of the C interface - each language system has its own separator.

When the above example is compiled, a POP-11 variable named *multiply* will be created. It's value will be a procedure, and the procedure, when executed will attempt to run an external procedure named *multiply*. Of course, the external procedure *multiply* does not exist until it has been linked in with *externalload*.

Once linked, the Pop procedure *multiply* can be called like any other POP-11 procedure. However, it will check that both its arguments are double precision floating point numbers. When it is satisfied with its arguments, it will apply the C routine *multiply*, and return the double precision decimal that C *multiply* returns.

The procedure *multiply* could also be loaded in raw mode using the following example:

```
external declare rawtag;

multiply = _multiply;

endexternal
```

After *loading*, the POP-11 variable *multiply* will hold an external procedure, which can be called using *external_apply* (see REF * EXTERNAL). Note the example is from a machine ruinning the Unix operating system - in *raw* mode it is the responsibility of the user to utilise any linker conventions on external label names.

24.2.2 Loading external procedures

The syntax of the *load* operation is:

```
external load <word:T> ;
```

```
;;; body

endexternal
```

The body in this case is a sequence of object files to link. The files can be specified in one of three ways:

1. as a word, in which case an operating system dependent suffix is appended. (in Unix, this is '.o', in VMS, '.obj').
2. as a string, in which case the filename is passed unchanged (this permits you to use non-standard suffixes, such as '.a')
3. as a library specification. This must be a string, but its form is operating system dependant. On Unix, an example might be '-lm' to indicate the maths library. On VMS, using 'NAG$LIBRARY:NAG/LIB' might be used to indicate the NAG graphics library.

Note that an *unload* operation is performed on the given tag before the load is executed, see below for details. See further below for a full example, including *external load*.

24.2.3 Unloading external procedures

Syntax:

```
external unload <word:T>
```

This operation frees the memory previously occupied by the external procedures known by the tag T. Attempting to run a procedure after it has been unloaded will result in a *mishap*.

Notice that POP-11 maintains a history list of *load* operations, and that *unload*ing will undo the effects of all loads back to the given tag. Thus if the following are loaded A, B,

C, D (in that order) then unloading B will also unload C and D automatically. (See REF * EXTERNAL for more details).

To examine the state of the external load system, do the following:

```
external_show();
```

24.3 Extensions to pop11

Many simple Pop objects correspond directly to primitive data-types used by external procedures. Examples are the integer, and decimals (floating point). Here is a table which summarises the main similarities:

POP-11	C	Fortran	Pascal
(integer)	char		char
integer	int	integer	integer
decimal	float	real	
ddecimal	double	double precision	real

Notice that POP-11 has no *char* data-type, but does have strings (ie packed arrays) of characters. Individual characters can be passed as integers, hence the entry in the table.

The (so-called) derived data-types, however, do not have direct equivalences in POP-11, however, in most cases they can be simulated using POP-11 datastructures.

Since scanning all the external procedures arguments to verify that they conform to the formal parameter specification is very time consuming, a variable is provided to disable this feature:

$$false \rightarrow external\_type\_check;$$

will disable *all* checking of arguments until *true* is assigned back to this variable.

Warning: Arguments that need special treatment on passing, eg. arrays, functions etc., require *external_type_check* to be set to *true* to facilitate this pre-processing.

On the subject of efficiency, it should be noted that, for implementation reasons, using pointers will slow down the calling process, as will passing an *ident* for call-by-reference. Used rarely, these should cause no noticable degredation of performance; em however, using arrays of pointers can make the calling sequence intolerably slow.

24.4 Arrays

The most common derived data-type is the array, and although POP-11 arrays are implemented in a non-conventional manner, the external load package is able to coerce the correct (?) behaviour, provided the following guidelines are observed. To build an array suitable for an external procedure to manipulate, a set of POP-11 procedures is provided, the names being designed for use with specific external languages.

```
Data        aka         C procedure         Fortran procedure

char                    consstring
short                   array_of_short
integer     int         array_of_int        array_of_integer
decimal     float       array_of_float      array_of_real
ddecimal    double      array_of_double     array_of_double
```

The procedures *array_of_...* take arguments like the POP-11 procedure *newarray*, described in chapter 10.4. That is, a *boundslist*, giving the lower and upper subscript of each dimension of the array. For details of *consstring* see Chapter 9, and see the section below on C related issues.

For example, if an external procedure expects the following:

```
C:          int i[10];
Pascal:     var i: array [0..9] of integer;
Fortran:    dimension i(10)
```

then the following would construct a suitable array for passing as parameter *i*:

```
vars i = array_of_int([0 9]);
```

or

```
vars i = array_of_integer([0 9]);
```

if using Fortran or Pascal.

Note that the POP-11 subscripts needn't be in the same range as the external procedure's ones — as long as the number of elements is the same, all will be fine. For example, a Pascal *array*[10..20] *of integer* is compatible with a POP-11 *array_of_integer*([105 115]) since both have exactly 11 elements.

Whenever possible, the facility will check that an array passed as argument is the same size as the one expected by the external procedure.

Note: The C convention that (e.g.) $int\ foo[99]$; and $int * foo$; both introduce arrays is not followed by this facility and the run-time checking will object to a relaxed attitude in this matter. Also the size of the array, for checking purposes, is set at *externaldeclare* time, not at run time when the external procedure is called.

24.5 Pointers

Both C and Pascal can expect pointers to data, these are constructed using the procedure *conspointer_to*. For example, an external procedure which expects data of the form:

```
C:          int *x;
Pascal:     var x: ^integer;
```

would be satisfied by passing the value of x, declared:

```
vars x = conspointer_to 7;
```

The de-referenced value can be then accessed by $x(1)$.

Arrays of pointers can also be constructed, although every member of the array must be initialised to a pointer before passing it to the external handler. It is also necessary to specify the type of the pointers in the array - that is, the type of the data which is pointed to. For example:

```
C:          int *x[5];
Pascal:     var x: array [0..4] ^integer;
```

might receive data declared:

```
vars    p = conspointer_to(0),
        x = array_of_pointer([0 4], p, [int]);
```

Notice the use of a constant pointer (i.e. the same pointer for each cell in the array).

When working with pointers, it might be useful to declare an operator for calling *conspointer_to*. In C, the operator might be called &:

```
define -3 & datum;
lvars datum;
    conspointer_to(datum);
enddefine;
```

The newly-declared & operator can now be used to build pointers as follows:

```
vars p = &0;
```

N.B. only pointers to data can be built in this way - not pointers to a variable's storage area, unlike the C statements:

```
int a, *b;
b = &a;
```

where values assigned to a can be retrieved by reference to $*b$ (that which b points to).

This may seem to be a drawback, since it appears impossible for an external procedure to affect the values of POP-11 variables (as per 'call-by-reference'). The facility, however, provides a means for call-by-reference, using the syntax *ident*.

As an example of call-by-reference, consider the following example. Suppose an external procedure wishes to halve the value of a variable, whose value is an integer, it might be defined thus:

```
C:
    void halve(x)
    int *x;
    {
        *x = *x / 2;
    }
```

Fortran:

```
    subroutine halve(x)
    integer x
```

```
x = x / 2
return
end
```

Pascal:

```
procedure halve(var x:integer);
begin
    x := x div 2
end;
```

Here is an example of a call to then routine *halve*, assuming it has been loaded.

```
vars a = 18;
halve(ident a);
a =>
** 9
```

It is possible to think of ident as constructing a pointer, or of passing the variable by reference.

24.6 Call by reference

As has been discussed, call-by-reference is implemented using the POP-11 syntax word *ident*. However, in Fortran (and other 'true' call-by- reference languages — not C or Pascal), all the arguments need to be references. This facility, however, takes care of this, and in the case of Fortran, the only c-b-r language that we have studied, it creates any references required, at run time. This means that a Fortran routine which takes a number and returns its square root *via a variable*, such as this:

```
subroutine myroot(n, r)
double precision n, r

r = sqrt(n)
return
end
```

can be called like this:

```
vars ans = 1.0;
myroot(9.0, ident ans);
ans =>
** 3.0
```

Notice that before passing an *ident*, the variable must contain an instance of the kind of data it is expected to return with.

The number 9.0 has been converted to a reference before being passed to the Fortran procedure.

24.7 Structures

Structure passing is not supported in the current version of the facility.

24.8 Details of C interface

The C language interface is the parser used by the *external declare* syntax for reading C language declarations.

The parser will accept any combination of standard C declarators, with the exception of the structure, union and enumerated types. It is possible to build arbitrary combinations of the derived types *pointer to*..., *array of* ... and *function returning*.... The primitive types accepted are:

```
char, int, short and long int, float, double
```

Note that both *short* and *long* integers are treated as *int*, since *short*s are converted to *int* during a procedure call, and on all machines supported, a *longint* is equivalent to an *int*.

No distinction is drawn at run time between a *double* and *float*.

The special symbols {} are used to separate function declarations.

The keyword *unsigned* is accepted and ignored.

Arrays passed as parameters may have their size fully, or partially specified.

The C operation *typedef* is supported, so it is possible to declare:

```
typedef char *string;

foo(x)
string x;
{}
```

which declares the single parameter of *foo* to be a pointer to an *int*.

When passing strings (i.e. arrays of *char*), it is the responsibility of the user to null terminate the string. For example:

```
vars s = 'This is a null terminated string\^@';
```

24.9 Details of fortran interface

The Fortran language interface is the parser used by the *externaldeclare* syntax for reading Fortran language declarations.

A Fortran language declaration is basically a Fortran function or subroutine with all its executable statements taken out. This usually means that you can directly transcribe the header of the Fortran code.

The Fortran type descriptors currently recognised are INTEGER, REAL, DOUBLE PRECISION, EXTERNAL

If a variable is typed as EXTERNAL then is is considered to be *special*: no type checking will be done on its value. A declaration is terminated by the keyword END.

E.g.

The Fortran function:

```
    DOUBLE PRECISION FUNCTION SQUARE(N)
    DOUBLE PRECISION N
    SQUARE = N * N
    END
```

would be declared thus:

```
external declare anothertag in fortran;

    DOUBLE PRECISION FUNCTION SQUARE(X)
    DOUBLE PRECISION X
    END

endexternal;
```

Some procedures for constructing arrays are also provided. They take a boundslist like the POP-11 procedure *newarray*. They make the sort of array you would expect them to:

```
array_of_integer, array_of_real, array_of_double
```

All Fortran arguments are passed as call-by-reference. The facility takes care of this by converting any non-pointer arguments to pointers at run time. (q.v. the ident mechanism for updating te contents of variables)

24.10 Writing other language interfaces

The library files *c_dec.p* and *fortran_dec.p* should be consulted for examples of the techniques used in writing an interface module for the external library.

Any interface module should be written in the section $-$*external*, where some extra identifiers that should prove useful can be found. These are summarised below. The module should export (to the top level section) a routine named $<$ *language* $>$ *_dec*, in the same way as the Fortran module defines a procedure named *fortran_dec*, and the C module *c_dec*.

This routine will be called (at compile time) by the syntax word *external*, and its job is to read up to the syntax word *endexternal*, removing it from the *proglist*.

As the *_dec* routine reads from proglist, it can make calls to the procedure

```
external_import(<word:S>, <word:P>, <vector:A>, <list:R>,
                <procedure:E>, <boolean:C>)
```

to *register* each external procedure that it wants imported.

The arguments are as follows:

S the linkers' name for the external procedure (it is the interface module's responsibility to prepend underscores, etc, as it feels appropriate).
P the name of the POP-11 variable which is to take a procedure which calls the external procedure, after checking the arguments.
A a vector of type specifications. Each entry corresponds to a formal parameter of the external procedure. see below.
R the type specification (see below) for the return value of the external procedure
E a procedure to print fatal argument type mismatches. It should expect three arguments, the offensive object, the desired type of the object, and the name of the procedure being called. The procedure should cause a *mishap*, or at least call *interrupt*.
C a flag indicating, when *true*, that the external procedure expects arguments call-by-reference.

The following is a useful identifiers in section $-external$, which is set when the library file *external.p* is compiled, so itmay be used in compile-time preprocessor statements (i.e. #_*IF* etc).

UNIX a constant, when *true*, then the current operating system is Unix 4.2 BSD. When *false*, VMS is indicated.

24.11 Type specifiers

When a type-specifier is called for, a list (non-dynamic!) should be supplied. It is used to verify that a given datum conforms to a formal parameter specification. The list should consist of the following words only:

```
pointer, array (followed by an integer, or false) or function,
```

and terminate in one of the following simple-types:

```
special, char, int, float or double
```

The simple numerical types can be followed by a range specification, ie two integers (or false). The *special* type is provided for passing a value with no checking being performed.

e.g.

```
[int]           expect an integer
[int 0 65535]   expect an integer in the range 0 - 65535 (this is
                the spec to simulate $unsigned short int$ in C)
[pointer int]   expect a pointer to an int, or a call-by-reference
                style ident
[pointer char]  expect a string
[array false int]
                expect an array of int, don't check its size
[array 100 int]
                expect an array of 100 integers total
[pointer pointer function int]
                expect a pointer to a pointer to a function (see
                below) which returns an int
[array pointer char]
                expect an array of strings
```

```
and so on.
```

When specifying the return value of a procedure, an extra type-specifier is allowed, namely [void], which simply indicates that the procedure returns no value.

24.12 Full example

The following Fortran program is compiled to produce an object file called *thresh.o* (Unix) or $THRESH.OBJ$ (VMS)

```
c
c This subroutine thresholds an array IMAGE with dimensions XSIZE
c and YSIZE with the threshold LIMIT.
c
      subroutine threshold(image, xsize, ysize, limit)
      integer limit,xsize,ysize
      integer image(xsize,ysize)

      do 10 j=1,ysize
      do 10 i=1,xsize
10    if (image(i,j) .lt. limit) image(i,j) = 0
      end
```

And the following C program is compiled to produce an object file *array.o* (Unix) or *ARRAY.OBJ* (VMS)

```
/* print the contents of a two dimensional integer array */

void prarr(array, xsize, ysize)
int array[], xsize, ysize;
{
    int i;

    for (i = 0; i < xsize * ysize; i++)
    {
        printf("%d ", array[i]);
        if ((i + 1) % xsize == 0)
            printf("\n");
    }
}
```

The following two external declaration blocks are used:

```
external declare myprog in fortran;
```

```
        subroutine threshold(image, xsize, ysize, limit)
        integer limit,xsize,ysize
        integer image(xsize,ysize)
        end

    endexternal;

    external declare myprog in c;

        void prarr(array, xsize, ysize)
        int array[], xsize, ysize;
        {}

    endexternal;
```

And both procedures can be linked in using:

```
    external load myprog;
        thresh
        array
    endexternal;
```

Notice that the *external* syntax chooses the suffix for the object files according to the operating system in use.

To test the routines, a two dimensional array of integers is needed:

```
    vars a = array_of_int([1 10 1 10],  procedure (i, j);
                                            random(9);
                                        endprocedure);
```

The C procedure can be used to print the contents of the array before, and after, it is thresholded using the Fortran routine:

```
prarr(a);
9 8 1 8 6 5 1 1 4 3
1 3 4 6 7 7 6 9 6 6
8 1 1 9 3 4 8 7 7 4
5 2 7 2 6 3 1 5 2 6
3 8 2 9 9 2 1 1 3 5
4 9 6 1 7 3 6 4 5 2
4 6 8 9 7 6 9 5 8 7
9 3 5 7 7 7 4 3 6 8
1 7 4 7 9 1 8 8 6 3
9 9 5 3 1 5 3 7 5 2

threshold(a, 10, 10, 5);
prarr(a);
9 8 0 8 6 5 0 0 0 0
0 0 0 6 7 7 6 9 6 6
8 0 0 9 0 0 8 7 7 0
5 0 7 0 6 0 0 5 0 6
0 8 0 9 9 0 0 0 0 5
0 9 6 0 7 0 6 0 5 0
0 6 8 9 7 6 9 5 8 7
9 0 5 7 7 7 0 0 6 8
0 7 0 7 9 0 8 8 6 0
9 9 5 0 0 5 0 7 5 0
```

The array has been thresholded with level=5, ie all values less than 5 have been set to zero.

Chapter 25

INTEGRATING PROLOG IN THE POPLOG ENVIRONMENT

NOTE Can we have some up-to-date timings?

This is a slightly expanded version of a paper with the same title that appeared in IJCAI-83 — the 8th International Joint Conference on AI, University of Karlsruhe 1983.

After this paper was written, lexically scoped identifiers were introduced into Poplog and changes were made to the Poplog virtual machine to support a more efficient implementation of compiled Prolog. (The timings given in the paper are out of date.)

25.1 Prolog within High Level Language Systems

There have been a number of projects involving implementing Prolog-like languages within LISP systems, notably the LOGLISP [Robinson and Sibert 82] and QLOG [Komorowski 82] systems. Since POP-11 is so similar to LISP, it is worthwhile stating some of our main aims for comparison:

- BANDWIDTH OF INTERFACE. LOGLISP and QLOG both incorporate mechanisms for calling LISP routines as "subroutines" from the logic language, as well as low bandwidth interfaces in the other direction. We aim to develop a model of how Prolog datastructures and control can mesh in with those of POP-11, so that, for instance, POP-11 programs can create backtracking points and control the generation of solutions by Prolog.
- SYMMETRY BETWEEN LANGUAGES. A multi-language programming environment should treat the languages it supports in a symmetrical way. Both LOGLISP and QLOG are clearly logic languages implemented in LISP, rather than the other way around. In the POPLOG system, we have achieved a symmetry by having Prolog an equal partner with POP-11, programs in both languages being compiled into instructions for the same virtual machine.
- COMPATIBILITY. Our aim is to provide a Prolog system that is compatible with an existing standard [Clocksin and Mellish 81], and which can be used without any knowledge of the other programming languages in POPLOG. In our aim for compatibility, we differ from both QLOG and LOGLISP, but especially from LOGLISP, as it is not our intention to investigate alternative ways of running logic programs.
- EFFICIENCY. Aiming for an efficient Prolog system, we have followed Warren [Warren 77], and have implemented only a compiler (not an interpreter, as in both QLOG and LOGLISP).

[Note: these timings were made using an early version of Poplog.]

25.2 Implementing Backtracking by Continuation Passing

Prolog is implemented using a technique called *continuation passing*. In this technique, procedures are given an additional argument, called a *continuation*. This continuation, which is a closure, describes whatever computation remains to be performed once the called procedure has finished *its* computation. Suppose, for example that we have a procedure *prog* which has just two steps: calling the subprocedure foo and then, when that has finished, calling the subprocedure baz. Were such a procedure to be written using explicit continuations, baz would be passed as an extra argument to foo since baz is the continuation for *foo*. Actually, *prog* itself would also have a continuation and this must be passed to *baz* as *its* continuation, thus:

```
define prog(continuation);
    foo(baz(%continuation%))
enddefine;
```

Thus, if we invoke *prog* we must give it explicit instructions, *continuation*, as to what is to be done when it has finished. *prog* invokes *foo*, giving *foo* as its continuation the procedure *baz* which has been partially applied to the original continuation since that is what is to be done when *baz* (now invoked by *foo* as its continuation) has finished its task.

Continuations have proved of some significance in studies on the semantics of programming languages [Strachey and Wadsworth 74] [Steele 76]. This apparently round about way of programming also has an enormous practical advantage for us — since procedures have explicit continuations there is no need for them to 'return' to their invoker. Conventionally, sub-procedures returning to their invokers means:

"I have finished — continue with the computation"

With explicit continuations we can assign a different meaning to a sub- procedure returning to its invoker, say:

"Sorry — I wasn't able to do what you wanted me to do"

prog accomplishes its task by first doing *foo* and then doing *baz*. The power of continuation programming is made clear if we define a new procedure *newprog*, thus:

"Try doing *foo* but if that doesn't work then try doing *baz*"

This is represented thus:

```
define newprog(continuation);
```

```
    foo(continuation);
    baz(continuation);
enddefine;
```

If we now invoke *newprog* (with a continuation) then it first calls *foo* (giving it the same continuation as itself). If *foo* is succesful then it will invoke the continuation. If not then it will return to *newprog* which then tries *baz*. If *baz* too fails (by returning) then *newprog* itself fails by returning to *its* invoker.

Now consider the following Prolog procedure:

```
happy(X) :- healthy(X), wise(X).
```

This says that X is *happy* if X is *healthy* and *wise*. If this is the only clause for *happy* then we may translate this to the following POP-11 procedure:

```
define happy(x, continuation);
    healthy(x, wise(%x, continuation%))
enddefine;
```

A call of this procedure can be interpreted as meaning:

"Check that X is happy and if so do the CONTINUATION"

This is accomplished by passing X to *healthy* but giving *healthy* a continuation which then passes X across to *wise*. Let us suppose that someone is *healthy* if they either *jog* or else *eat cabbage*, i.e.:

```
healthy(X) :- jogs(X).
healthy(X) :- eats(X, cabbage).
```

This can be translated as:

```
define healthy(x, continuation);
    jogs(x, continuation);
    eats(x, "cabbage", continuation);
enddefine;
```

Finally, let us assume that we know that *chris* and *jon* both *jog*. This can also be represented by a POP-11 procedure:

```
jogs(chris).
jogs(jon).
```

```
define jogs(x, continuation);
    if x = "chris" then continuation() endif;
    if x = "jon" then continuation() endif;
enddefine;
```

25.3 A Simple Prolog without Datastructures

The translation of *jogs* given in the last section does not cater for the case where X is unknown and we wish to *find* someone who *jogs*. In fact, we need to take account of the special features of Prolog variables. Prolog variables start off *uninstantiated* and can only be given a value once. In addition, two *uninstantiated* variables can be made to "share" which means that as soon as one of them obtains a value, the other one automatically obtains the same value. In the Prolog sub-system of *poplog*, this is dealt with by representing unknowns by single element data structures of the class *prologvar*, which are specified in section 26.4.

The CONTents of one of these *prologvar*s is initially *undef*. If a variable is instantiated to some value, this value is placed into the *prologvar* contents. If two variables "share", one *prologvar* cell is made to contain (a pointer to) the other. To find the "real" value of a sharing variable, it is then necessary to "dereference" it, i.e. to look for the contents of the "innermost" reference.

In the *jogs* example, instead of simply comparing X with the word *chris*, it is necessary to attempt to *unify* the data structure with the word*chris*. If we are trying to *find* somebody who jogs, X will be a reference with contents *undef*, whereas if we are trying to *check* whether some specific person jogs, it will be a word (such as *chris*).

Here is a simplified version of our unification procedure. *unify* operates by binding Prolog variables which have no value, i.e. by putting something other than *undef* into the reference. Once the unification is complete, *unify* performs the continuation, and if this returns (i.e. fails), *unify* undoes the changes it made to the datastructures and then itself returns. If the two structures cannot be unified, then *unify* returns without taking any action. Thus, calling the continuation means success (in Prolog terms) and returning means failure.

```
define unify(x,y,c);
   if x == y then
      c()
   elseif isref(x) and cont(x) /= "undef" then
      unify(cont(x),y,c)
   elseif isref(y) and cont(y) /= "undef" then
      unify(x,cont(y),c)
   elseif isref(x) then
      y -> cont(x);
      c();
      "undef" -> cont(x)
   elseif isref(y) then
      unify(y,x,c)
   endif
enddefine;
```

The procedure first sees if the two given datastructures, X and Y, are identical. If so, it immediately applies the continuation C. If the structures aren't identical, but either of X and Y is a variable that has become bound (a reference with contents not *undef*) then unification can use the value of that variable instead. In the case where X is an unbound variable (not the same as Y), *unify* binds it to Y (by setting the CONTents of X to Y) and calls the continuation. Once this has returned,*unify* unbinds the variable (by resetting its *contents* to *undef*) and then itself returns. This definition of unification does not deal with the case where X or Y is a Prolog complex term. The handling of Prolog datastructures is not significantly more complex.

Given the existence of the *unify* procedure, the correct definition of *jogs* is now simply:

```
define jogs(x,c);
   unify(x,"chris",c);
   unify(x,"jon",c)
enddefine;
```

25.4 Representing Prolog Datastructures

Most POPLOG datastructures are treated by Prolog as new classes of constants, the exceptions being those used to implement the standard Prolog datastructures (terms). A Prolog term has a fixed type (principal functor) and length (arity), and it is important that accessing a given component can be achieved in constant time. This means that terms are well represented by objects very like POP vectors.

List pairs in standard Prolog are simply instances of the general term, whereas in POPLOG, as in LISP, the pair is a special datastructure. For compatibility, we have actually implemented Prolog pairs as POPLOG pairs, although this is not visible to the Prolog user who does not wish to use the other POPLOG languages.

As an example of how complex datastructures are handled, here is the definition of the list concatenation predicate *append*, together with a "corresponding" POP-11 program:

```
append([],X,X).
append([L|M],Y,[L|N]) :- append(M,Y,N).

define append(x,y,z,c);
   vars l, m, n;
   unify(x,[],unify(%y,z,c%));
   consref("undef") -> l;
   consref("undef") -> m;
   consref("undef") -> n;
   unify(x,conspair(l,m),
```

```
      unify(%z,conspair(l,n),
         append(%m,y,n,c%)%))
enddefine;
```

This procedure attempts to unify the first argument, X, with an empty list and if successful unifies the second and third arguments, Y and Z, with each other and then applies the continuation C. If this returns (ie fails), it creates three new unbound Prolog variables, L, M and N. The first argument of *append* is unified with a pair made (by using the procedure *conspair*) from L and M. If X is already a pair, this should set L and M to its $front$ and *back* respectively. The third argument to*append* is unified with a pair made from L and N. This ensures that the first elements of X and Z are identical. Finally the recursive call of *append* is performed and if this is successful the original continuation C is performed!

25.5 More Complex Control Structures

So far, we have seen how passing continuations between procedures allows Prolog-style backtracking to be implemented in POPLOG. However, when a continuation-expecting procedure is called from one that is not provided with one, what continuation should it be given? In fact, there are a number of non-local control procedures in POPLOG that can be used, giving rise to a variety of ways of invoking Prolog programs.

First of all, consider the problem of calling the Prolog system as a "subroutine" from POP-11. We wish to present some query, and simply find out whether it can be satisfied, possibly finding out the values of relevant variables in the first solution. In this case, the final continuation to be executed needs to be something that will cause a procedure exit right back to where the first Prolog predicate was called. The procedures *throw* and *catch*[1], which are developments of facilities available in some LISP systems, enable this to be done. The facility can be packaged up in the form of a procedure $YESNO_CALL$.

```
define succeed();
   true
enddefine;
```

[1] These are treated in detail in Chapter 2.24

```
define dorun(proc);
   proc(throw(%"yesno"%));
   false
enddefine;

define yesno_call(p);
   catch(%dorun(%p%),succeed,"yesno"%)
enddefine;
```

The procedure *yesno_call* takes a continuation-expecting procedure (Prolog procedure) as its argument and produces a new procedure (a closure of *catch*) which always *returns*, leaving *true* or *false* on the stack (according to whether the final continuation is executed or not). *catch* is supplied with three arguments - a main procedure to call, a second procedure and a *pattern*. The first thing that *catch* does is to simply call the first procedure. If, during the execution of this procedure,*throw* is called with an argument that matches the pattern, control returns immediately to *catch*, which calls the second procedure and then returns. If *throw* is never called with an appropriate argument, *catch* just returns as soon as the first procedure does.

In this instance, the main procedure given to *catch* is a closure of *dorun*, which will call the Prolog procedure and, if that returns (ie fails), simply put the value *false* on the stack. The Prolog procedure is given a continuation such that, if it succeeds, it will perform a *throw* back to the original *catch*. The second *catch*argument, *succeed*, will then run, and will put the value *true* on the stack. In this example, the pattern used to 'link' the *throw* and the *catch* is simply the word "*yesno*".

throw and *catch* can be used to provide an implementation of the Prolog *cut* operator. Simple uses of the cut can be accomplished through the $YESNO_CALL$ procedure. For instance, the Prolog clauses:

```
tax_code(X,Y) :- employs(Z,X), !, employed_code(X,Y).
tax_code(X,Y) :- unemployed_code(X,Y).
```

could be translated into a POP-11 procedure as follows:

```
define tax_code(x,y,c);
   if yesno_call(employs)(consref("undef"),x) then
      employed_code(x,y,c)
   else
      unemployed_code(x,y,c)
   endif
enddefine;
```

A problem with this particular implementation of *cut* is that information about variables that exist previously and are instantiated within the $YESNO_CALL$ is lost. Hence certain variables will not be reset if backtracking subsequently takes place. One remedy for this would involve packaging up the actions depending on the truth value of the condition into an *expression continuation* [Strachey and Wadsworth 74]. In fact, our actual implementation solves the problem by secretly keeping reset information in a different way (see below).

Sometimes one would like to use the Prolog system as a *generator* of solutions to some problem. These solutions may need to be produced in a *lazy* fashion [Henderson and Morris 76] and we may wish to manipulate the generator in CONNIVER-like ways [Sussman and McDermott 72]. To do this, we need to exploit the POPLOG mechanisms for handling co-routining between multiple processes. To create a POPLOG process, we use the procedure *consproc*, which when given a procedure returns a process which when invoked with *runproc* will call the given procedure. *consproc* must also be given the arguments that will be needed by the procedure and a count of the number of arguments. A running process interrupts its execution by calling the procedure *suspend*. This causes the process which originally invoked it to restart. Suppose we had a Prolog predicate *legal_move*, which returned possible legal moves in some game in its one argument. We might want to produce a generator that produced these, one by one, as they were needed by some other program. The following POP-11 code would do this:

```
vars x;
consref("undef") -> x;

vars generator;
consproc(x,suspend(%x,1%),2,legal_move) -> generator;
```

consproc is being used here to make a new process involved with the calling of *legal_move*. *legal_move* is provided with two arguments, the normal Prolog argument X, which is to be instantiated to some move, and a continuation. The continuation will be invoked when the Prolog goal succeeds. In this case, it will *suspend* the execution of the process, leaving one result, X, on the stack. Thus to get the first possible legal move into a variable Y, we now write:

```
runproc(0,generator) -> y;
```

(the 0 specifies that no arguments are to be passed to the process). When we wish to obtain the second move, we call *runproc* with *generator* again. The process is now *woken up*, and it acts as if its call to*suspend* has simply returned like a normal procedure call. Within the continuation passing model, procedure return means failure. Hence the legal move generator will backtrack to find another solution. When it has succeeded, it will again *suspend* with X put on the stack.

In this example, the generator is returning its answers in the $reference$ created for the variable X. As Prolog backtracks, the contents of this reference will be reset to $undef$ and then set to the next solution. In order that Y keeps an unchanging record of the first solution, it must actually be given the $dereferenced$ version of the value returned by the generator.

25.6 The Actual Implementation (1983)

What we have presented so far is a model for how Prolog could be implemented within POPLOG. This is the model that we expect our users to have, and the system is expected to behave as if this is the way it is actually constructed. Given this basic framework, it is possible to make certain optimisations that are *invisible to the user*. This section mentions some of the more interesting optimisations that we have made.

1. The number of closures constructed, and the number of control frames grown, can be reduced by having compiled Prolog clauses make use of modified unification code, which always $returns$ and indicates success or failure with a boolean result. The disadvantage with this is that the responsibility for resetting Prolog variables on backtracking is

no longer taken by *unify*, but must be handled by extra procedure calls at each backtrack point. Moreover, there needs to be a globally accessible datastructure (*trail*) for holding variables to be reset on backtracking.

2. The number of datastructures created can be reduced by having the compiler generate special purpose *unification code* for structures mentioned in the heads of clauses, rather than code to create such structures and then invoke a general *unify* procedure. This is Warren's approach [Warren 77], and is one way of introducing "structure sharing" [Boyer and Moore 72].

3. The control frames for Prolog procedures can actually be discarded as soon as there are no more untried choices. The POP procedure *chain* allows the compiler to produce code to do this.*chain* simply provides an alternative way of calling a *poplog* procedure, which discards the current stack frame before invoking the new procedure. The explicit representation of continuations and the use of *chain* have a potential for allowing more space to be reclaimed than in normal "tail recursion optimising" schemes [Warren 80].

4. The representation of a Prolog procedure as a *single* POPLOG procedure is not always appropriate, especially when the use of the Prolog predicates *assert* and *retract* causes individual clauses to come and go rapidly. Our Prolog compiler can use an alternative representation, with each *clause* represented by a procedure, and can choose which representation to use.

25.7 Future Developments

There are many possible ways in which we can extend the POPLOG system to enhance mixed language programming further.

First of all, we can make more use of the screen editor interface and realise its great potential for debugging. There already exists a POPLOG implementation of the STRIPS problem solver [Fikes and Nilsson 71], which produces a continuous display of the changing goal tree using the facilities of the editor. It would be extremely valuable to have such a debugging aid for Prolog programs. [Note added by A.S. 1987. Prolog LIB CT and LIB Tracer does this.]

Secondly, we have hardly begun to explore the productive ways in which programs can use the facilities of the two languages. POPLOG is already being used for mixed language programming in natural language processing and vision, but many of the possibilities are unexplored, such as the possibilities for using Prolog in conjunction with the POPLOG "process" mechanism. We also need to further develop and refine the range of syntaxes available for accessing these facilities.

Finally, more work needs to be done on basic implementation. Some issues that we are considering are the space/time efficiency of various types of in-line unification code, and ways to minimise the *trailing* of variables.

25.8 Conclusions

The POPLOG system provides an integrated environment for developing genuinely MIXED LANGUAGE programs in POP-11 and Prolog. We believe that its most important features in this respect are as follows. Firstly, the POP-11 and Prolog compilers are just two of potentially many procedures which generate code for the underlying virtual machine. This means that the two languages are compatible at a low level, without there being the traditional asymmetry between a language and its implementation. Secondly, the continuation passing model provides a semantics for communication between these two languages which allows for far more than a simple "subroutine calling" interface. Finally, the control facilities available within POPLOG make it possible to implement a system which is faithful to the theoretical model, but which is nevertheless efficient.

25.9 Acknowledgements

We would like to thank John Gibson, the main implementer of POPLOG and the POP-11 compiler, as well as Aaron Sloman and Jon Cunningham for many useful discussions.

25.10 References

Boyer, R.S. and Moore, J.S., "The Sharing of Structure in Theorem Proving Programs", in Machine Intelligence 7, Edinburgh University Press, 1972.

Burstall, R.M., Collins, J.S. and Popplestone, R.J., PROGRAMMING IN POP-2, Department of Artificial Intelligence, University of Edinburgh, 1977.

Clocksin, W.F. and Mellish, C.S., "The UNIX Prolog System", Software Report 5, Department of Artificial Intelligence, University of Edinburgh, 1979.

Clocksin, W.F. and Mellish, C.S., "Programming in Prolog", Springer Verlag, 1981.

Fikes, R.E. and Nilsson, N.J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Intelligence 2, 1971.

Hardy, S., "Towards More Natural Programming Languages" Cognitive Studies Memo CSRP 006, University of Sussex, 1982a.

Hardy, S., "The POP Programming Environment", Cognitive Studies Memo CSRP 005, University of Sussex, 1982b.

Henderson, P. and Morris, J.H., "A Lazy Evaluator", Proceedings of the 3rd ACM Symposium on Principles of Programming Languages, 1976.

Hunter J.R.W., Mellish, C.S. and Owen, D., "A Heterogeneous Interactive Distributed Computing Environment for the Implementation of AI Programs", SERC grant application, School of Engineering and Applied Sciences, University of Sussex, 1982.

Komorowski, H.J., "QLOG - The Programming Environment for Prolog in LISP", in Clark, K.L. and Taernlund, S.-A., LOGIC PROGRAMMING, Academic Press, 1982.

Kowalski, R., "Logic as a Database Language", Department of Computing, Imperial

College, London, 1981.

Mellish, C.S. and Hardy, S., "Integrating Prolog in the POPLOG Environment", Cognitive Studies Memo CSRP 010, University of Sussex, 1982.

Pereira, L.M., Pereira, F. and Warren, D., "User's Guide to DECsystem-10 Prolog", Occasional Paper 15, Department of Artificial Intelligence, University of Edinburgh, 1979.

Robinson, J.A. and Sibert, E.E., "LOGLISP: An Alternative to Prolog" in MACHINE INTELLIGENCE 10, Ellis Horwood, 1982.

Steele, G.L., "LAMBDA: The Ultimate Declarative", Memo 379, Artificial Intelligence Lab, MIT, 1976.

Strachey, C. and Wadsworth, C.P., "Continuations: A Mathematical Semantics for Handling Full Jumps", Technical Monograph PRG-11, Programming Research Group, Oxford University, 1974.

Sussman, G.J. and McDermott, D.V., "The CONNIVER Reference Manual", Memo 203, AI Lab, MIT, 1972.

Swinson, P.S.G., "Prescriptive to Descriptive Programming: A Way Ahead for CAAD", in Taernlund, S.-A., Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980.

Warren, D.H.D., "Implementing Prolog", Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

Warren, D.H.D., "An Improved Prolog Implementation which Optimises Tail Recursion", in Taernlund, S.-A., Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980.

Chapter 26

The POPLOG Prolog system

A description of those datatypes and procedures built in to POPLOG which are intended for the support of Prolog.

26.1 The prolog memory area

In addition to the heap, the user stack and the call stack, POPLOG maintains an extra area of memory exclusively for the use of Prolog. This contains two extra Prolog stacks: the continuation stack and the trail. This area of memory, like the others, is dynamic, and so will be expanded as required; a maximum limit for its expansion is set by the variable *pop_prolog_lim*, and a trace of its size and usage can be got by assigning the value 1 to the variable *popgctrace*, described in Chapter **??**.

pop_prolog_lim
The integer value of this variable controls the maximum number of POPLOG words to which the system will expand the area used for the continuation stack and trail (default value 16000). When this limit is reached, the mishap 'PROLOG AREA OVERFLOW' occurs.

26.2 The Continuation Stack

The execution model used by POPLOG Prolog is that of "continuation passing", described fully in DOC * CONTINUATION. In this technique, procedures are given an additional argument called a continuation. The continuation is a closure which describes whatever computation needs to be performed once the called procedure has successfully finished its computation: the procedure invokes the continuation rather than returning.

The usefulness of this model is that it enables us to implement success and failure in terms of whether or not a procedure call returns. If the procedure successfully completes its computation, it invokes its continuation; if it fails, it returns.

Consider the following Prolog procedure:

```
p(X) :- q(X), r(X).
p(X) :- s(X).
```

In the continuation passing model, it could be represented (in theory) by the following POP-11 procedure, where C represents the continuation passed to p:

```
define p(X, C);
    q(X, r(% X, C%));
    s(X, C);
enddefine;
```

A call of this procedure can be interpreted as meaning:

"Prove $p(X)$ and if it is true, do the continuation C".

This is accomplished by passing to q both X and a continuation which, if q succeeds, passes X and the original continuation C across to r. If r succeeds, C will be invoked. If q or r fails, q will eventually return (i.e. backtracking will have occurred). In that case, X and C will be passed to s. If s succeeds it will call C; if it fails it will return, and p will fail.

The continuation passing model is the model of Prolog that users are expected to have, and POP-11/Prolog mixed-language code is normally written in terms of it (see *prolog_unifyc*, below). However, the real Prolog implementation does not pass continuation closures as arguments, but instead makes use of a dedicated stack — the continuation stack. Before a procedure is called, a number of items which form its continuation are pushed onto this stack. Essentially what happens is that for a procedure which is the compiled form of a clause with n terms in its body, $n-1$ sets of items are pushed onto the continuation stack. These correspond to the second up to the nth terms. Each set consists of a count of the number of items pushed (which is equal to one more than the number of arguments to be passed to the procedure) the procedure and any arguments. These are pushed in the following order (assuming M arguments):

```
count (m + 1)       <-- top of continuation stack
argument 1
...
argument m
procedure
```

After each set consisting of count, arguments and procedure has been pushed, the procedure corresponding to the first term in the body of the clause is called.

A POP-11 procedure which more closely represents the Prolog procedure above is thus:

```
define p(X);
    PLOG_SAVE;
    prolog_push_continuation(X, r, 2);
    q(X);
    PLOG_RESTORE;
    s(X);
    PLOG_RESTORE;
enddefine;
```

PLOG_SAVE and *PLOG_RESTORE* are hypothetical syntax words which correspond to the VM instructions *sysPLOG_SAVE* and *sysPLOG_RESTORE*. These instructions create a choice (backtrack) point (see below). *prolog_push_continuation* is a system procedure and is not for general use.

The continuation stack pointer is saved at each choice point and restored if, through backtracking, the choice point is returned to.

26.3 The Trail

The trail records all assignments made to Prolog variables. It too is a stack. Variables are represented by objects of type *prologvar* (see below in the section on datatypes), and whenever one of these is assigned to, it is pushed onto the trail. At each choice point, a pointer to the current top of the trail is saved. Whenever the Prolog system backtracks through a choice point, the old trail pointer is restored and any variables pushed onto the trail after that point are reset to be undefined.

Another effect of backtracking is to free *prologvar*s which are no longer accessible to active procedures. A free-list of *prologvar*s is maintained for this purpose: once backtracking has returned past the point at which a *prologvar* was first created, it is put back onto the free-list for later re-use. The garbage collector can short-circuit some of this variable collection: when it sweeps the trail, if it encounters a *prologvar* which has no outstanding choice points between the point of its creation and the point of its most recent instantiation, then it will collect that *prologvar* as garbage. This is safe, because if ever backtracking should return to the point where the variable was instantiated, it must also pass the point at which it was created and so free the variable for re-use. This means that the variable will never be rebound in the remainder of its lifetime, and so is effectively a constant. When the garbage collector collects the variable, it dereferences it first, and any reachable references to it are overwritten with its dereferenced value. This process saves time on backtracking and compacts the space used by the trail and by Prolog data structures in the heap.

$prolog\_barrier\_apply(P)$
Applies the procedure P in such a way as to isolate it from any outstanding invocations of Prolog procedures. This is done by placing "barriers" on the Prolog continuation stack and trail which serve to mark the beginning of the Prolog area accessible to P; any existing trailed variables or outstanding choice points become invisible to P. This is of importance when using Prolog in conjunction with the POPLOG process mechanism. If the situation arises where several live processes are all using Prolog procedures, the barriers serve to divide up the Prolog area correctly between them, marking those parts of the continuation stack

and trail which need to be saved and restored with each process. The standard entry points into Prolog (such as *prolog_compile* and *prolog_invoke*, defined when the Prolog system is loaded) call this procedure internally and so are safe for general use. However, whenever a process is created which calls lower-level Prolog operations explicitly (such as *prolog_assign*, *prolog_unify* etc.) then *prolog_barrier_apply* should be called as the first action when that process is run.

prolog_reset()

Reinitialises the Prolog memory area, and the *prologvar* free-list and variable numbering. This must be done whenever a Prolog computation terminates abnormally. The procedure *setpop* always runs *prolog_reset*.

26.4 Prolog datatypes

POPLOG provides two special datatypes for Prolog support.

The datatype *prologvar* is used to represent variables in Prolog terms. The structure of a *prologvar* is identical to that of a reference [1], with a single field containing the value of the variable (an uninstantiated *prologvar* has itself as its value). The class of *prologvar*s is special however, as extra support is needed for their efficient allocation and garbage collection and for management of the trail (see the description of the trail above). There is no direct access available to the contents of a *prologvar* except via *fast_cont*; in normal usage, prolog variables should only be manipulated by the higher-level procedures *prolog_deref*, *prolog_assign* and similar which take proper account of their special requirements.

The datatype PROLOGTERM represents general complex terms in Prolog. A prologterm is characterised by a functor (which is a word) and a set of arguments, where the number of arguments (the arity) is variable. Currently a prologterm is an ordinary vectorclass object whose *class_subscr* procedure *prolog_arg_nd* (see REF * KEYS), but this does not give proper access to the functor part of a term and will be changed for Version 14 of POPLOG.

These datatypes, together with the integers, are sufficient for the representation of all

[1] See Chapter 3.8.1

pure Prolog data. The existing Prolog system though is more liberal in its interpretation of data both for efficiency and to simplify the interface between Prolog and other POPLOG languages. Thus the following special cases are made:

– Prolog terms with functor "./2" are represented as POPLOG conspairs so that lists in Prolog are the same as lists in POP-11 (see REF * LS);

– Prolog atoms, which could be modelled as prologterms with arity 0, are in fact represented more simply as words;

– all other POPLOG datatypes are legal in Prolog, but are treated as atomic; i.e. they cannot be decomposed into parts. Like ordinary atoms, these atomic objects can be considered as terms with no arguments.

When handling Prolog data, it is important to keep in mind this special interpretation imposed by the Prolog system. In the following list of procedures, those which begin with "*prolog_*" respect that interpretation and so are recommended for general use; the remaining procedures operate on the concrete datatypes prologterm and *prologvar* and must be used with care.

An extra level of complexity is added when variables are included in terms. Data structures built by Prolog will typically contain chains of variables, and to obtain the true structure it is necessary to remove these chains by dereferencing. The basic procedure to use for this is *prolog_deref* which will follow a chain of *prologvar*s and return the common value to which they are all bound; *prolog_full_deref* will remove all the variable chains from a term. Standard procedures which access parts of Prolog terms are provided with dereferencing built in, as this is the generally more useful behaviour. Raw, non-dereferencing versions are also available however, and such procedures are indicated by the suffix *_nd* in their names.

$consprologterm(WORD, n) \rightarrow PROLOGTERM$

Constructs and returns a prologterm with functor WORD and arity n, the arguments being taken from the next n items on the stack.

$destprologterm(PROLOGTERM)$

Destructs PROLOGTERM, leaving its arguments, its functor and its arity on the stack (i.e. it does the opposite of $consprologterm$).

$initprologterm(n) \rightarrow PROLOGTERM$
Creates and returns a prologterm with functor $undef$ and arity n, and whose arguments are all initialised to 0.

$isprologterm(O) \rightarrow b$
This procedure returns $true$ if O is a prologterm, $false$ if not.

$isprologvar(O) \rightarrow b$
This procedure returns $true$ if O is a $prologvar$, $false$ if not.

$isprologvar(O) \rightarrow b$
Applies the procedure P to each dereferenced prolog argument of O in turn (cf. $prolog\_appargs\_nd$).

$prolog\_appargs\_nd(O, P)$
Applies the procedure P to each prolog argument of O, without dereferencing (cf. $prolog\_appargs$).

$prolog\_arg(n, TERM) \rightarrow O$
Dereferences and returns the nth argument of the pair or prologterm TERM (cf. $prolog\_arg\_nd$). n must be less than or equal to the arity of TERM.

$prolog\_arg\_nd(n, TERM) \rightarrow O$
$O \rightarrow prolog\_arg\_nd(n, TERM)$ This procedure returns or updates the nth argument of the pair or prologterm TERM without dereferencing (cf. $prolog\_arg$). n must be less than or equal to the arity of TERM.

$prolog\_args(O)$
Puts all the dereferenced prolog arguments of O onto the stack (cf. $prolog\_args\_nd$). A call of this procedure is equivalent to $prolog\_appargs(O, identfn)$

$prolog\_args\_nd(O)$
Puts all the prolog arguments of O onto the stack without dereferencing them (cf. $prolog\_args$). A call of this procedure is equivalent to $prolog\_appargs\_nd(O, identfn)$

$prolog\_arity(O) \rightarrow n$
This procedure returns the prolog arity n of O; if O is not a pair or a prologterm then n will be 0.

$prolog\_checkspec(O_1, O_2, n) \rightarrow b$
This procedure returns $true$ if O_1, interpreted as a prolog term, has functor O_2 and arity n; returns $false$ otherwise. If O_1 is a prologterm, then O_2 and n must be the real functor and arity of that term; if O_1 is a pair, then O_2 must be the word "." and n must be 2; for any other O_1, O_2 must be identically equal to O_1 and n must be 0.

$prolog\_complexterm(O) \rightarrow b$
This procedure returns $true$ if O is either a pair or a prologterm, $false$ if not.

$prolog\_deref(O_1) \rightarrow O_2$
If O_1 is a $prologvar$ it is dereferenced and its value returned; any other item is simply returned. Dereferencing does not extend to any sub-components of (the dereferenced value of) O_1 (cf. $prolog\_full\_deref$).

$prolog\_full\_deref(O_1) \rightarrow O_2$
This procedure returns a fully dereferenced version of O_1, such that the only $prologvar$s remaining in O_2 are uninstantiated ones. Any sub-components of O_1 which contain $prologvar$s will be copied during the dereferencing, but other, non-variable components may not be.

$prolog\_functor(O_1) \rightarrow O_2$
$WORD \rightarrow prolog\_functor(PROLOGTERM)$ When used as an accessor, returns the prolog functor of O_1. If O_1 is a prologterm, then its real functor (a word) is returned; if O_1 is a pair, then the word "." is returned; if O_1 is any other datatype, then the item itself is returned. When used as an updater, $prolog\_functor$ works only on prologterms; there are few circumstances in which changing the functor of a term can be justified.

$prolog\_maketerm(O_1, n) \rightarrow O_2$
Creates and returns a term with prolog functor O_1 and prolog arity n, the arguments being taken from the next n items on the stack. If n is 0 then O_1 is returned; if O_1 is the word "." and n is 2, then a pair is returned; otherwise O_1 must be a word and a prologterm is returned (cf. $consprologterm$).

$prolog\_newvar() \rightarrow PROLOGVAR$
Creates and returns a new, uninstantiated $prologvar$.

$prolog\_termspec(O_1) \rightarrow n \rightarrow O_2$
This procedure returns the prolog arity and the prolog functor of O_1. If O_1 is a prologterm then n and O_2 will be the real arity and functor of the term; if O_1 is a pair then n will be 2 and O_2 will be the word "."; otherwise n will be 0 and O_1 will be returned as O_2.

$prolog\_undefvar(O) \rightarrow b$
This procedure returns $true$ if O is an uninstantiated $prologvar$, $false$ if not. O is not dereferenced, so this may return $false$ where $isprologvar(prolog\_deref(O))$ would return $true$.

$prolog\_var\_number(PROLOGVAR) \rightarrow n$
$prolog\_var\_number(false)$ When applied to an uninstantiated $prologvar$, this returns some integer n. The value of n may vary between applications, but any two $prologvar$s which are sharing will always return the same n, i.e. this procedure defines an equivalence on $prologvar$s. (It is primarily used for generating the printing representations of variables.) When applied to the value $false$, the numbering is reset so that subsequent calls will resume

counting from 1. If applied to any other datatype, *false* is returned.

prologterm_key
The key for objects of type prologterm.

prologvar_key
The key for objects of type *prologvar*.

26.5 Variable instantiation and unification

These next procedures provide for the instantiation of *prologvar*s. This may be done explicitly via *prolog_assign* or implicitly through a call to the unifier. In either case, each assignment done both updates the value field of the *prologvar* and keeps a record of the assignment on the trail so that it may be undone if backtracking ever returns past that point. The bindings can only be undone, however, if a proper choice point has been established before the assignments are made. In fact, due to the trail compaction performed by the garbage collector (explained above), *prologvar*s assigned to before a choice point has been established (i.e. before the first *sysPLOG_SAVE* has been executed) are effectively constants; they will be elided at the next garbage collection and any references to them replaced with their current values. An example demonstrates this effect:

```
prolog_vars X;
prolog_assign(X, "cat");
X =>
** <prologvar cat>

sysgarbage();
X =>
** cat
```

$prolog\_assign(PROLOGVAR, O)$
Updates the value of PROLOGVAR to be O and pushes PROLOGVAR onto the trail for resetting on backtracking. No checking or dereferencing is done, so this procedure should only be used when the condition

```
prolog_undefvar(PROLOGVAR)
```

is $true$ (cf. $prolog\_assign\_check$).

$prolog\_assign\_check(PROLOGVAR, O)$
A checking version of $prolog\_assign$; this performs the same operations but will mishap unless the condition $prolog_undefvar(PROLOGVAR)$ is $true$.

$prolog\_unify(O_1, O_2) \rightarrow b$
This procedure returns $true$ if O_1 unifies with O_2, $false$ if not. If the unification succeeds, any $prologvar$s in the two items will be instantiated in such a way that the condition

$$prolog\_full\_deref(O_1) = prolog\_full\_deref(O_2)$$

is true. If the unification fails, some or all of the $prologvar$s in the two items may still be instantiated. In either case, instantiated variables will not be reset until the next $PLOG\_RESTORE$ is done (cf. $prolog\_unifyc$).

$prolog\_unifyc(O_1, O_2, P)$
Attempts the unification of O_1 and O_2, and if successful calls the procedure P. (P is the continuation for the unification - see above.) A proper choice point is created for the unification using $sysPLOG\_SAVE$ and $sysPLOG\_RESTORE$ so that if P is called, any $prologvar$s in O_1 and O_2 which are instantiated by the unification remain so for the duration of that call. Once P has returned, or if it is never invoked at all, any variable bindings created by the unification are reset before the call of $prolog\_unifyc$ itself returns. Thus a caller of this procedure will see no difference in O_1 and O_2 before and after the call.

26.6 Prolog VM instructions

Two distinct sets of Prolog VM instructions are provided. One works in conjunction with the continuation stack and trail to support the creation of choice points and backtracking; the other provides special purpose unification code for efficient head-matching of clauses.

26.7 VM instructions: backtracking

The instruction $sysPLOG\_SAVE$ creates a choice point. A call of this procedure causes the state of the Prolog procedure being compiled to be saved on entry to that procedure. The state is represented by three variables: the continuation stack pointer, the trail pointer and a pointer to the next free Prolog variable. A call of $sysPLOG\_RESTORE$ causes these state variables to be restored to their saved values, and thus implements backtracking to the previous choice point.

$sysPLOG\_SAVE()$
Marks the procedure currently being compiled as a Prolog procedure. This ensures that the Prolog state (continuation stack pointer, trail pointer and next$free$variable pointer) will be saved on each entry to the procedure, and that appropriate portions of the Prolog memory area will be saved if ever the procedure is included as part of a process record. Only one call of this instruction has any practical effect inside a procedure, and two calls cannot be made without an intervening $sysPLOG\_RESTORE$.

$sysPLOG\_RESTORE()$
Restores the state of the continuation stack and trail from the current saved values. The trail is unwound back to the saved position, and all the variables removed from it are reset to be undefined. The pointer to the next free variable is also restored. This procedure cannot be called without a preceding call to $sysPLOG\_SAVE$.

$sysPLOG\_RESTART()$

Re-saves the state of the continuation stack and trail, and the pointer to the next free Prolog variable. It is used when a second choice point is to be set up by a procedure, e.g. when a tail-recursive call is optimised to a backward jump.

26.8 VM instructions: unification

Some arguments to the Prolog VM unification instructions are QUALIFIED; that is, the interpretation of an argument (and therefore its value) depends on the value of another argument - the argument QUALIFIER. An argument qualifier may assume one of the following value: $true$, $false$, a word or an integer. The effect of these upon the argument which is qualified is as follows:

$true$ the item it qualifies denotes a word which is the name of a variable which needs to be pushed (using sysVAL);

$false$ the item it qualifies denotes a constant which needs to pushed (using sysPUSHQ);

a word which is the name of a selector procedure (e.g. $fast\_front$) which is applied to the item it qualifies;

an integer which is used to subscript the Prolog term which it qualifies.

More than one argument of an instruction may be qualified (there being one qualifier per qualified argument).

$sysPLOG\_ARG\_PUSH(O, O_QUALIFIER)$
Plants code to push (on to the user stack) either O, the value of the (lexical/permanent) identifier associated with the word O, or some component of the datastructure O, depending on the value of $O_QUALIFIER$ (see above).

$sysPLOG\_IFNOT\_ATOM(O, O_QUALIFIER, ATOM, ATOM_QUALIFIER, FAIL_LABEL)$
protected procedure variable Plants a

$LOG\_IF\_NOT\_ATOM$ instruction: this will

– dereference O (qualified by $O_QUALIFIER$) if it is a Prolog variable;

– unify O and ATOM (qualified by $ATOM_QUALIFIER$) using a restricted form of unification where ATOM must be an atom which does not require dereferencing;

– jump to the label $FAIL_LABEL$ if the unification fails.

$sysPLOG\_TERM\_SWITCH(O, O_QUALIFIER, TERM, VAR_LABEL, FAIL_LABEL)$ Plants a $PLOG\_TERM\_SWITCH$ instruction: this will

– dereference O (qualified by $O_QUALIFIER$) if it is a Prolog variable;

– jump to the label VAR_LABEL if O is an uninstantiated Prolog variable;

– compare the functor of TERM with the functor of O otherwise, and jump to FAIL LABEL if they are different.

26.9 Invoking prolog

The Prolog system may be invoked either in top-level mode, where each term read is executed as a goal, or in assert mode, where terms are interpreted as clauses to be added into the database.

prolog [macro variable] Switches subsystem from POP-11 to Prolog top-level, autoloading the Prolog system if necessary.

prolog_compile($STREAM$) [procedure variable] If the Prolog system is loaded, this compiles Prolog program text from the character source STREAM. Calling *prolog_compile* from Pop is the same as doing a *reconsult* from Prolog: clauses are added to the database rather than executed, and new procedure definitions supersede existing definitions of the same name. STREAM may be one of:

– A character repeater procedure;

– A string or word representing a filename (the filename "user" or 'user' is treated as a synonym for the character repeater *charin*);

– A device record.

The compiler does an initial "see" on the file it is given to make it the current input stream. *prolog_compile* will mishap if the Prolog system is not loaded.

26.10 Variable declarations

prolog_lvars [library macro variable]

Declares lexical variables initialised to uninstantiated *prologvar*s. See section **??**.

prolog_vars [library macro variable] Declares permanent variables initialised to uninstantiated *prologvar*s. See section **??**.

prolog_lvars is a macro (see HELP * MACRO) providing a syntax for constructing Prolog variables in POP-11. The POP-11 words to which the Prolog variables are assigned are declared as lexical variables (see HELP * LVARS).

The statement

```
prolog_lvars k, l;
```

is equivalent to

```
lvars k, l;
prolog_newvar() -> k;
prolog_newvar() -> l;
```

See the Prolog file HELP * TERMSINPOP for more information on *prolog_newvar*.

prolog_vars is a macro (see HELP * MACRO) providing a nice syntax for constructing Prolog variables in POP-11. The statement

```
prolog_vars k, l;
```

is equivalent to

```
vars k, l;
prolog_newvar() -> k;
prolog_newvar() -> l;
```

See the Prolog file HELP * TERMSINPOP for more information on $prolog_newvar$.