TEACH PRIMER – AN OVERVIEW OF POP-11

(Second Edition)

Aaron Sloman

School of Computer Science, The University of Birmingham

With help from the Poplog development team

Updated For Poplog V15.01, Jan 1996

Minimal Model of Pop-11 added before Chapter 2 on 23 Oct 1997

Contents

0.1 INTRODUCTION TO SECOND EDITION (Jan 1996)

The language Pop-11 changed significantly with the release of Poplog Version 15.0 in late 1995. In particular the default for undeclared input and output variables in a procedure changed from dynamic scoping to lexical scoping. Thus it is no longer necessary to use ”lvars” to declare such variables explicitly. In addition the use of ”vars” within a procedure to simultaneously declare a variable as global (”permanent”) and make its value dynamically scoped, generated a warning message. After complaints from educational users those warning messages were withdrawn in Poplog Version 15.01. This primer is geared to V15.01, though users of earlier versions should still find it helpful.

There were several other changes, which are described in the HELP NEWS file available on Poplog systems, and in the release notes provided with Poplog Version 15.0. Notable among these are the introduction of a socket library, new options for spawning sub-processes, and other facilities for interacting with external libraries and the operating system.

In addition Version 15 saw the first release of Poplog for the DEC Alpha Architecture, both under VMS and under Digital Unix (previously known as OSF/1). There was also a port of Poplog to run on Intel-based PCs under the free Linux operating system, and a reduced version of Linux Poplog was made freely available via ftp, so that for the first time an up to date version of Pop-11 is available free of charge. There is a charge for the unrestricted Linux Poplog. All versions of Poplog are available from Integral Solutions Ltd, at the address given in the Acknowledgements section.

It is also worth mentioning that Poplog Version 14.5 provided the first official release of both the Objectclass Object Oriented Programming extension to Pop-11 and the Graphical Objects library (GO), both in the ”proto” directory

```
$usepop/pop/lib/proto
```

Pop-11 is still being extended, and a future version will provide a new data-type, a ”matchvar” which provides extended pattern matching facilities integrated with the Pop-11 equality operator ”=”.

This edition of the primer has had a chapter added on how to define a general problem solver, and many other minor changes. It also seemed a good idea to start by illustrating Pop-11 before going on to more general matters, so that the reader who is new to Pop-11 has some idea of what the language is like.

0.2 FOR NEW USERS: A TASTE OF POP-11

Pop-11 is a rich and complex language with many subtle features. However it has a very easily learnt subset, which is very close to what can be found in many other programming languages.

Readers who have access to a Poplog system and who wish to get a feel for the langauge should obtain the ”TEACH GSTART” file from the Birmingham University ftp site

```
ftp://ftp.cs.bham.ac.uk/pub/dist/poplog/teach/gstart
```

This provides an introduction to Pop-11 through its graphical facilities (which do not work in the first release of the FREE version of Linux Poplog V15.0, but will work in later versions. It should work in all other versions of Poplog.)

For those who are not able to run Pop-11 here are some illustrations of what would happen if you could.

The symbol ”=>”, known as the ”print arrow” is used to print ”results” left on the stack by preceding Pop-11 expressions. Lines starting ”**” are printed out by the print, arrow. In a few cases, the output overflows to the following lines. Those cases should be clear. Everything else is input. ";;;" precedes an end of line comment, ignored by the compiler.

The Pop-11 compiler is incremental (like most Lisp, Basic, and Prolog systems) which means that you can give a succession of commands as illustrated below. Each command is compiled to machine code then run. (It is not interpreted.) The commands may either be typed directly to the compiler or given to the compiler through the Poplog editor VED, or some other editor, e.g. Emacs.

Examples follow.

```
;;; Put some integers, decimals, a word, a string and a list on the
;;; Pop-11 stack and print them out.
1, 2, 3, 99.9, 105.0035, "cat", 'a string', [a list] =>
** 1 2 3 99.9 105.0035 cat a string [a list]

;;; Declare three variables, one initialised, and do some calculations:
vars radius = 10, circumference, area;
2*pi*radius -> circumference;
pi*(radius**2) -> area;
[The circumference is ^circumference and the area is ^area] =>
** [The circumference is 62.8319 and the area is 314.159]

;;; Define a procedure to create a palindrome from a list

define palindrome(list) -> result;
    ;;; This procedure takes one input and returns one result
    ;;; take in a list, append the reverse of the list
    list <> rev(list) -> result;
enddefine;

;;; test it
palindrome([a b c d e]) =>
** [a b c d e e d c b a]

palindrome([[the cat][sat on][the mat]])=>
** [[the cat] [sat on] [the mat] [the mat] [sat on] [the cat]]

;;; Use the Pop-11 pattern matcher to define a procedure to
;;; report associations expressed as "key value" pairs in a list
;;; of associations.

;;; set up an association list
vars person =
    [name fred age 30 job butcher wife sally kids [sue tom]];

define association(item, list) -> result;
    ;;; If something follows item in list, return it,
    ;;; otherwise return false. Declare "found" as a pattern variable;
    vars found;
    if list matches [== ^item ?found ==] then
```

```
        found -> result
    else
        false -> result
    endif
enddefine;

;;; Test it

association("name", person) =>
** fred

association("wife", person) =>
** sally

association("kids", person) =>
** [sue tom]

;;; And, of course, a recursive definition of factorial, showing
;;; the "functional" programming style, without output variables
define fact(x);
    if x == 0 then 1
    else
        fact(x - 1) * x
    endif
enddefine;

;;; test it
fact(6) =>
** 720

;;; Pop-11 has unlimited precision integers. A 'small' example:
fact(200) =>
** 788657867364790503552363213932185062295135977687173263294742533244359
   449963403342920304284011984623904177212138919638830257642790242637105
   061926624952829931113462857270763317237396988943922445621451664240254
   033291864131227428294853277524242407573903240321257405579568660226031
   904170324062351700858796178922222789623703897374720000000000000000000
   000000000000000000000000000000

;;; Check that the results are correct (under a second on a Sun IPX)
fact(1000)/fact(999)=>
** 1000

;;; And complex numbers, with a real part and an imaginary part
vars i = sqrt( -1 );
i =>
** 0.0_+:1.0

i * i =>
```

```
** -1.0_+:0.0

;;; cross type equality works:
i * i = -1 =>
** <true>

;;; And ratios, with numerator and denominator

66/48 =>
** 11_/8
```

Those examples illustrate only a small subset of the language. The rest of this primer introduces some of the history of Pop-11 and many additional features of the language. But several of the more advanced features, including mechanisms for interfacing to the operating system, or for interacting with other languages, including Prolog and other Poplog languages, are not included, though pointers are given to the online documentation for those features in the Poplog system.

0.3 ACKNOWLEDGEMENTS

”POPLOG” is a trade mark of the University of Sussex.

This Introduction to Pop-11 uses material produced over many years by colleagues in the School of Cognitive and Computing Sciences at Sussex University, including Steve Hardy, Max Clowes and John Gibson in the 1970s, and since then by colleagues at Sussex, Integral Solutions Ltd, The University of Massachusetts at Amherst, and Hewlett Packard Laboratories, including especially:

Harry Barrow, Ros Barrett, Graham Brown, Julian Clinton, Jonathan Cunningham, Chris Dollin, Ben du Boulay, Rob Duncan, Roger Evans, John Gibson, James Goodlet, Tom Khabaza, Rudi Lutz, Steve Knight, Clark Morton, Chris Mellish, Jon Meyer. Simon Nichols, Robin Popplestone, Ben Rabau, Allan Ramsay, Ian Rogers, Ben Rubinstein, Mark Rubinstein, Colin Shearer, Mike Sharples, Chris Slymon, Robert Smith, Chris Thornton, John Williams, David Young.

The chief architect of Poplog and Pop-11 was John Gibson, at Sussex University, who designed and implemented the core mechanisms on which everything else depends including the compilers, store management, and interfaces to external languages and the operating system (Unix and VMS).

The language Pop-11 would not have existed, but for some excellent ideas of Robin Popplestone in the late 60s, which provided the basis for the family of ”Pop” languages.

During the 1990s some of the development of Poplog was done by the main commercial distributors Integral Solutions Ltd (ISL), working in collaboration with Sussex University. ISL was bought in December 1998 by SPSS after which they stopped distributing Poplog as a commercial product.

Instead, in July 1999, Version 15.53 of Poplog became available free of charge with full system sources at the Free Poplog site in Birmingham University, including all the poplog languages Pop-11, Prolog, Common Lisp, Standard ML, and a version of Scheme. See

http://www.cs.bham.ac.uk/research/poplog/freepoplog.html

Distribution information referring to ISL in this primer is out of date. After ISL was bought by SPSS they stopped distributing Poplog and Pop-11 (though they still used it in their Clementine data-mining system). For more information about the change see

http://www.cs.bham.ac.uk/research/poplog/poplog.info.html or

The Alphapop subset of Pop-11 for the Apple Mac is no longer available. It was developed by

```
Cognitive Applications Ltd
4 Sillwood Terrace
Brighton BN1 2LR,
England

Phone +44 273 821600
```

Poplog is currently being ported to Apple Mac and OSX.

0.4 POPLOG INFORMATION AND FTP SITES

The main distribution site for poplog is

http://www.cs.bham.ac.uk/research/poplog/freepoplog.html

There is a sourceforge project to port the full version of poplog to Windows and bring the packaging up to date

http://www.cs.bham.ac.uk/research/poplog/openpoplog.html

0.5 The Birmingham Poplog directory

A lot of AI and Cognitive Science teaching material and utility libraries, including a rule-based programming system, and the SIM_AGENT toolkit are available from the Birmingham University Poplog directory:

```
http://www.cs.bham.ac.uk/research/poplog/freepoplog.html
http://www.cs.bham.ac.uk/research/poplog/poplog.info.html
http://www.cs.bham.ac.uk/research/poplog/comp.lang.pop.faq.html
http://www.cs.bham.ac.uk/research/poplog/README
```

The ”teach/” subdirectory includes several files that improve on the original Poplog versions (some of which the author produced before he left Sussex for Birmingham). The rclib/ subdirectory includes many extensions to the graphical tools in Pop-11, e.g. new tools for building graphical control panels, more flexible than Propsheet.

There are also html, postscript and plain text versions of this primer.

0.6 PREFACE TO THE FIRST EDITION (Sept 1994)

(This preface was slightly revised in Jan 1996)

Pop-11 is the core language of the Sussex University Poplog system, which also contains Prolog (the logic programming language) Common Lisp, and Standard ML.

This Primer for Pop-11 is based on the "TEACH PRIMER" Poplog file which I originally produced while at Sussex University in 1984. It was distributed with Poplog and provided part of the original text for the book on Pop-11 by Barrett, Ramsay and Sloman, published in 1985. In Poplog V15.0 it was replaced by an earlier version of this Primer.

During 1993 and 1994 the primer was re-written and extended, making it consistent with new versions of Poplog Pop-11 (Version 14.2, Feb 1993, Version 14.5, mid 1994). The second edition was revised for Poplog Version 15.0, released Autumn 1995.

This Primer is not totally compatible with older versions of Poplog Pop-11. The vast majority of the examples will work with older versions of Poplog Pop-11 (from about Version 13), though it is likely that most versions are no longer executable owing to operating system changes.

Because the Primer is intended primarily for use by people who have an executable version of Poplog Pop-11, there are frequent references to online Poplog files, which supplement this text, including TEACH files, which contain many tutorial examples, HELP files which contain useful summaries, and REF files, which provide more complete and systematic documentation.

The versatile Poplog editor VED provides a very convenient tool for browsing these files, many of which contain code examples, which can be run in the editor. Similarly examples in the online version of this Primer can be run in VED, and the results checked against those given in the text. For readers lacking Poplog this primer should nevertheless give a useful, though incomplete, introduction to Pop-11. For Emacs users, accessing Poplog online information is not quite so convenient. The Poplog contrib directory includes some facilities for customising Emacs for Poplog users. It is hoped to add further improved Emacs facilities at the Birmingham Poplog FTP site.

There are also now many facilities in Poplog Pop-11 concerned with the X Window System interface, supporting graphics and graphical and menu driven interaction. This document does not describe those facilities in any detail.

Poplog now contains an (optional) X-based menu-driven interface, and there are many utility programs for building interfaces and other things. The editor, VED (and its multi-window version, XVED) can be extended by the user, as it is just a collection of Pop-11 programs, and provides a general purpose, terminal independent, user interface for screen-based programs.

The X facilities in Poplog provide a basis for more sophisticated types of interaction using graphics, menus, etc. See TEACH RC_GRAPHIC and TEACH PROPSHEET for some examples.

Besides Pop-11, the core language, Poplog includes versions of Lisp, Prolog and Standrard ML. This is made possible by the fact that Pop-11 includes facilities for defining new syntactic forms that are directly compiled to machine code, unlike other languages that would require the use of macros, or an interpreter.

The version of Lisp in Poplog is based on the original definition of Common Lisp, in the book by G.L. Steele, Common Lisp The Language, first edition. It does not at present (1992) contain all the features of some more widely used Lisp systems and in particular does not (in Poplog version 15.0) contain an interpreter, though it does, like the other Poplog languages, provide an incremental compiler. For information on the latest version of Lisp in Poplog see the HELP LISPNEWS file. (The Lisp in Version 15.0 of Poplog has been made more compatible with Steele's book.)

Pop-11 is similar in many respects to Lisp (though some people find it more readable) and, like Lisp, it provides a fully fledged research and development language which can be used both interactively and in batch mode.

Common Lisp has features Pop-11 does not have (e.g. an interpreter, in some implementations of Lisp), and Pop-11 has some features that Common Lisp does not have, e.g. a pattern matcher and the compilation tools. The ”section” mechanism of Pop-11 works rather differently from the ”package” mechanism in Common Lisp, though both are useful for structuring large programs. An important difference as regards programming style is that Pop-11 has a boolean data-type so that FALSE and the empty list are distinct objects.

Pop-11 also has several experimental object-oriented extensions available as library programs. The most sophisticated OOP extension, which has been recently adopted as standard for Poplog, is known as ”Objectclass” and is close in spirit to the Common Lisp Object System (CLOS). Objectclass was mainly designed and implemented by Steve Knight, at Hewlett Packard research laboratories, Bristol.

Pop-11 is the core of Poplog in that the other languages are implemented in terms of it. For users of Poplog it is worth knowing about Pop-11, even if they mainly use Prolog, Lisp or ML.

Knowledge of a subset of Pop-11 will make it possible to take full advantage of facilities for extending and tailoring the editor VED to suit your own requirements. Knowing Pop-11 will also facilitate making full use of the X interface, though in principle this could be done via libraries specific to the different languages.

Prolog is probably the most popular sub-language of Poplog, and for many types of programs it provides powerful and elegant facilities. However, for certain kinds of programs Pop-11 is more convenient, and more efficient, than Prolog; e.g. programs which do a great deal of arithmetic, string or vector manipulation, deterministic list processing, or rapidly changing global data-structures.

This Pop-11 Primer is illustrative rather than definitive: A more complete and rigorous specification of the Pop-11 language is provided in the Poplog REF files. There is a British Computer Society working group designing a new standard for Pop-11.

This Primer will be found easier to follow by those who already have some programming experience in a conventional programming language (e.g. Pascal or C). Such readers may find some of the explanations regarding procedures, variables, arguments, results, loops, stacks, etc. redundant. However in Pop-11 there are sufficient differences from more common languages that there will be points of interest in these sections. Some sections may be found difficult on a first reading and should be skimmed at first, especially by novice programmers.

0.7 Advantages of Pop-11

Should anyone use Pop-11 as a programming language rather than the better known languages like Basic, Fortran, Pascal, C or C++ ? The answer is that it depends on what you need the language for. In some cases the other languages are clearly better. For example C runs on more machines, and programs written in C will often run faster and require less space than the same program written in Pop-11.

However, Pop-11 is a richer, more advanced language which enables more ambitious programs to be written with less effort. For particularly ambitious projects, where the problem is very complex and exact nature of the problem is not well understood in advance, a language like Pop-11 is particularly useful for the following reasons:

(a) It provides a very rich set of facilities, with a varied range of data-types and control facilities, that can cope with a wide range of types of applications (including those requiring object oriented programming facilities).

(b) Its syntax is extendable so that different sub-languages suited to different sub-problems can easily be implemented within it and integrated with Pop-11 itself.

(c) The Poplog version of Pop-11 comes in an incremental interactive development environment which can enormously speed up the process of development, debugging and testing compared with most non-AI programming languages. For example the incremental compiler allows a new procedure to be compiled into a system that already contains many megabytes of compiled code, in a fraction of a second. That is because the compiling and linking are all done within the system: the system does not have to be re-built.

(d) For certain applications (e.g. like a word processor or other interactive tool) the development can be unending in that some users will want to be able to produce their own extensions to the original application. Such software is never complete. I call it ”very soft ware”. Because Pop-11 uses an incremental compiler which is part of the run-time system, it is possible for an application implemented in Pop-11 to support indefinite extension and tailoring by end-users. For example the Poplog editor VED is implemented in Pop-11 and many users have used that fact to implement their own extensions for a variety of different purposes.

(e) It is available on a variety of Unix platforms and on VAX and Alpha VMS systems, and provides a common interface to the file system and many system calls across these platforms. It can be run on PCs under the Linux operating system (A port to Windows NT is in progress.)

(f) It has a number of unusual features that facilitate certain sorts of programming. These features include the use of an ”open stack” for passing arguments and results, which makes it very easy, for example, to write procedures that take variable numbers of arguments or produce variable numbers of results, partial application, which supports the construction of very efficient ”closures” and memo-functions, dynamic lists, which implement a form of lazy evaluation, the process mechanism which supports the modelling of concurrent systems, the code-planting procedures that support the rapid development of new immediately portable languages, the choice of dynamic or lexical scoping of variables (a feature shared with Common Lisp), and the ”dynamic local expression” facility ”dlocal”, which automatic switching of environments in different contexts, including ”exit actions” for procedures. A feature that Pop-11 shares with Scheme and many functional languages is that procedures (functions) are ‘first class’ objects. That is they can not only be run, but can also be treated like all other data-types, assigned to variables, stored in data structures, and manipulated in various ways, including combining them with other data to form new procedures at run time (sometimes referred to as ‘closures’. This capability makes possible some very elegant forms of programs, such as are described (using Scheme) in the well known book ‘Structure and Interpretation of Computer Programs’ by Abelson and Sussman.

(g) The automatic store management system provides garbage collection of structures (including compiled procedures, device records, arrays, etc.) that are no longer needed by the running program. This both reduces the task of the programmer in working out which structures are no longer needed, and also prevents erroneous deallocation of memory that is still in use. Thus both wasteful ”leaking” of memory and obscure bugs due to deallocation errors are avoided. The garbage collector in Poplog Pop-11 is particularly fast, so that people using it for interactive purposes will often not even notice when garbage collections occur. (This depends on the size of process, the loading on the machine, and whether there is enough memory to prevent paging.)

(h) The interface to the X window system is particularly sophisticated, allowing externally developed X widget sets to be linked in dynamically, and providing full support for interaction between Pop-11 and X, via callbacks. Also the ‘destroy property’ mechanism in Pop-11 allows special actions to be associated with objects that are to be run if ever those objects become garbage. That means that a window on the screen can be automatically removed when an object is no longer needed, without the programmer having to write code to check whether the object is still needed.

(i) There are facilities for compiling stand-alone applications without the full development environment. This uses the tool POPC that is also used for building Poplog itself. For information about this see the HELP POPC file.

(j) There is an ”autoloadable library” mechanism based on search lists that makes it convenient for groups of users or individual users to have libraries that extend the facilities in Pop-11 and which are automatically compiled when needed. This is particularly useful for groups of programmers working together on a project and for students doing the same course.

(k) Pop-11 allows frequently used libraries to be pre-compiled in a ”saved image” which allows rapid startup and can be shared between different users. These saved images can even be mapped into shared memory if they contain no writable elements. Further saved images can be layered. In fact the language extensions for Prolog, Common Lisp and ML are implemented as shareable saved images. Users can them build additional saved images on top of those. (Note: Saved images cannot be built in the FREE version of Linux Poplog.)

These features make Pop-11 particularly appropriate for many of the complex problems of Artificial Intelligence, Cognitive Science or Human-machine interaction where it is difficult to be sure in advance what the nature of the problem is because our understanding of human beings is too limited and people vary too much, and where the modelling work requires rich knowledge stores, powerful inference mechanisms, rapid construction of complex temporary structures (e.g. during planning, or visual perception), and complex interactions between concurrent systems.

The closest comparable language with similar characteristics is Common Lisp, though many people (not all) find the Pascal-like syntax of Pop-11 easier to learn than Lisp's very terse syntax using very few syntax words. For the same reason some people find Pop-11 programs more maintainable. On several occasions I have met commercial programmers who were used to other languages, like Pascal, Fortran or C, and who had tried to learn Lisp and disliked its syntax immensely. By contrast when they tried to learn Pop-11 they found that it offered them a smooth transition from familiar programming constructs to more sophisticated AI programming. After that they found Lisp easier to learn.

Not everyone agrees on which is easier to learn or use: so it is good that both Pop-11 and Lisp should be available.

Other AI languages such as Prolog share some of the features of Pop-11 and Lisp, including automatic garbage collection and rapid incremental development and testing, though the logic based style of Prolog can for certain problems be more clumsy than the corresponding functional or imperative style of Lisp or Pop-11. However Prolog is excellent for applications where the logical semantics, the unifier and the built in backtracking mechanisms are needed, and often allows very economical and powerful solutions to complex problems.

One advantage of Poplog Pop-11 is that its code-planting mechanisms, support implementation of incremental compilers for the other AI languages so that applications requiring a mixed language style can be supported easily. These mechanisms were used to implement the other Poplog languages, namely Prolog, Lisp and Standard ML.

A user of one of these languages can use the interface to Pop-11 to access additional features of Poplog, or for sub-problems where a different style of language is more appropriate. Thus a Prolog programmer can use calls to Pop-11 for interaction with the X window system and graphics.

In some cases it is possible to develop a complex application in Pop-11, or another rapid-prototyping language, as part of the process of finding out exactly what the problems are and trying out alternative solutions in the quickest possible way, and then re-implement the final version in a different language that enables programs to run faster or use more space, or be more easily ported to a variety of machines and operating systems. For example, I once had to produce a package to control the direction of printing of a daisy-wheel printer in order to minimise print head movements. I first developed and tested the program in Pop-11, and then rewrote it in C. The whole process would have taken far longer had I had to use C throughout. Similarly some AI researchers develop low level procedures for analysing images or speech using Pop-11 and then rewrite them in C when they have stablised.

Like LISP, Pop-11 can be used either as a main programming language or as an efficient systems language on which to build higher level tools, such as a logic programming language or an expert-system shell.

Poplog allows programs in Pop-11 and Prolog to be combined, where a mixed style of programming is desirable. Programs written in conventional languages (e.g. Pascal, Fortran, or C) may be linked into Poplog. For details see the online documentation files HELP EXTERNAL and REF EXTERNAL, and other files referred to therein.

For sophisticated programmers, the online file REF PROLOG explains the interface between Prolog and Pop-11. For others, HELP PLOGINPOP describes some utilities for mixing the languages and REF SUBSYSTEM describes a more general mechanism.

0.8 Pop-11 as a teaching language

For teaching purposes, Pop-11 enables students to explore a wide range of programming styles (imperative, functional, object oriented, logical, pattern-based, rule-based, event driven, concurrent). Having learnt the various styles supported by Pop-11, they can use the same environment to branch out and learn the other Poplog languages, and learn how to design and implement new languages, which can then be tested in a rich supporting environment.

Moreover, the online documentation and program libraries provide a framework in which students can explore concepts and techniques at a pace that is matched to their own abilities. For absolute beginners the existence of very powerful built in facilities such as automatic garbage collection, syntax for list structures, a pattern matcher and the database library, all contribute to an environment in which interesting and challenging projects can be tackled at a much earlier stage than is possible with most languages.

The interface to X also makes it possible quickly to add powerful graphical capabilities and menu-driven interactions. These features also make it relatively easy for teachers to produce powerful demonstration packages with which students can interact in order to gain a good understanding of a variety of concepts and techniques. The new ”HIP” system (Hypermedia in Poplog), available from Integral Solutions Ltd extends these capabilities, by providing tools for developing multi-media applications, including images and sound.

Already there are powerful teaching libraries developed at the Universities of Sussex, Reading, Leeds, Oxford, Birmingham, and possibly elsewhere. It is expected that these will grow.

In particular the ‘incremental compiler’ and integrated screen editor VED which is part of the Poplog Pop-11 system can not only speed up program development and testing time, but also provide a convenient interface to sophisticated online documentation and browsing facilities, supporting different libraries for different groups of students.

In some ways Pop-11 in Poplog is similar to the best LISP systems, though many LISP systems seem to require far more memory. The richer, more redundant, syntax seems to make Pop-11 a more suitable first teaching language than LISP. However, this is an issue on which endless disagreement is to be expected, since there will always be those who think that a simpler syntax is preferable for students. My own view is that the syntactic requirements for ease of compilation and use by a computer are totally different from the requirements for easy learning and use by humans. This is why natural languages, with all their complexity, irregularity, and redundancy, are better suited to the human brain than most existing programming languages. Increasingly we should develop languages whose syntax is suited to the task and to human users rather than to designers of compilers or machines.

The fact that Pop-11 is not as widely used as other languages can discourage some teachers from using it. However, I would argue that for many learners the most important thing is that they develop their understanding of computing concepts and learn how to think about complex information manipulation processes in as friendly and supportive an environment as possible. After that they can use the ideas better in more primitive languages, such as C or C++ than if they had started only with those languages. For a short course intended to train commercial programmers, for limited tasks, Pop-11 would not be suitable. For university level instruction, where the concepts are more important than the syntax, and where there is scope for a great deal of student directed learning and exploration of new ideas, Pop-11 in the Poplog environment is extremely suitable, including those students who will subsequently not be writing code but who need to have a deep understanding of what computing systems are and what they can and cannot easily be made to do. This would apply to managers, computer journalists, systems analysts, people concerned with marketing software or supporting users, and so on. Program writers are a relatively small subset of those involved in the production, distribution, selection and use of good software systems, a point that is not always understood by teachers.

0.9 Disadvantages of Pop-11

Arguments about the relative merits of different languages can get very heated and are often extremely subjective. However, just as there are some objectively testable advantages to Pop-11 there are also some disadvantages:

(a) It is not available on PCs, except on relatively powerful PCs running the Linux operating system (a version of Unix). A port of Poplog to Windows NT port has been started but by January 1996 had not been completed. (For up to date information, please contact Integral Solutions Ltd., address above.) The is a Mac implementation of of Pop-11, known as Alphapop, previously available from Cognitive Applications Ltd, does not run under recent versions of the Mac operating system.

(b) Because of its size and the supported machine types, Pop-11 is not recommended for small embedded systems. However, it can be used to implement cross compilers for embedded systems.

(c) The complex, user-extendable, syntax makes it extremely difficult to automate the analysis of Pop-11 programs. For applications where mathematical analysis of software is a requirement, e.g. flight control systems, Pop-11, would not be suitable, though it could be used for rapidly developing prototypes prior to final implementation, and it could support expert-systems acting alongside the critical software, e.g. to analyse data or make heuristic suggestions that would not be adopted unless checked by a more rigorous system, or a person.

(d) Because of the long history of its development and our inability in the early days (e.g. 1970s and early 1980s) to anticipate some of the extensions that would be required several years later, there were some unfortunate choices of identifer names that are hard to repair without annoying existing users, and several aspects of the syntax of Pop-11 that are messy. Up to Poplog version 15, the default for procedure formal parameters and output variables was, unfortunately, to declare them as "vars" not "lvars" (lexical variables). This has now been remedied, though a special compile_mode is available to restore earlier semantics.

(e) The undisciplined use of the open stack in Pop-11 can lead to obscure run-time errors, even though it is often very useful.

(f) Some users find the redundant syntax of Pop-11 too verbose, especially users with a strong mathematical bent, many of whom prefer the elegance and economy of Lisp, supported by a powerful editor.

(g) The language is not very widely used, though there are pockets of enthusiasm in various countries, both among academics and among commercial users, and the main distributor, Integral Solutions Ltd has managed to continue growing despite the very severe recession in recent years.

(h) Like Common Lisp, Pop-11 is a very rich and complex language. Becoming fluent in the use of all of its capabilities can take a long time (e.g. a year or more), though many of its users have found the learning well worth while.

(i) Because Pop-11 has so many features it is possible for beginners inadvertently to trip over more advanced features and then be mystified and discouraged. This can be a disadvantage, though in a good teaching environment it is also a source of important new learning.

(j) Because Pop-11 is compiled (for speed) rather than interpreted, it is not possible to produce such sophisticated run-time debugging tools as for those versions of Lisp that have an interpreter. However, this disadvantage has recently been reduced considerably by the development of new source-level debugging tools, by Simon Nichols and Robert Duncan, documented in the online HELP DEBUGGER file.

0.10 A brief history of Pop-11

Pop-11 is derived from POP-2, a language originally invented at Edinburgh University for research in Artificial Intelligence. POP-2 was described in

```
PROGRAMMING IN POP-2
R. M. Burstall, J. S. Collins and R. J. Popplestone
Edinburgh University Press, 1971
```

Additional information about the history and philosophy of the POP family of languages can be found in

```
POP-11 Comes of Age: The Advancement of an AI Programming Language
ed J.A.D.W. Anderson,
Ellis Horwood,  1989.
```

This includes a long paper by the author of this Primer on the development of Pop-11 at Sussex University.

The original version of POP ran on an Elliot 4130 computer and is now obsolete. A later version was implemented on the DEC-10 computer running the TOPS-10 operating system in Edinburgh. Julian Davies, also at Edinburgh at that time, implemented an improved version, called POP-10, in the early seventies. Yet another version for the DEC-10 called WPOP (WonderPOP), was implemented by Robert Rae, with some help from Allan Ramsay, and became available in the late seventies. It was also transferred to the TOPS-20 operating system, and was quite widely used for AI research for a while.

Steve Hardy implemented the first version of Pop-11 in 1975. This was a small system which ran on PDP-11/40 computers under the UNIX operating system. It did not include all of POP-2, but had some extra features instead, designed to make it easier to use, especially for teaching purposes. In particular, the pattern matcher, the autoloadable library mechanism and a large collection of help and teaching files made it especially useful for educational purposes, and several Universities and one school in the UK (Marlborough College) used it for some time in the late 70s and early 80s. This version of Pop-11 is now obsolete, and probably unobtainable, though for a while the a version for PDP-11 computers running UNIX version 7 was available from Nottingham University Psychology Department.

The second version of Pop-11, available only since October 1981, and mostly implemented by John Gibson, is much larger and has far more facilities. It includes all the features of POP-2 and many more. However, it requires a computer with a large, 'flat' address space, well over a megabyte in size. We correctly predicted in the early 1980s that as the price of hardware fell, it would soon be possible for computers in the home, in the office, and in schools to run programs of the size of the Pop-11 system, and larger, providing a very powerful tool for program development, or for teaching computing. Unfortunately, the horrors of MSDOS and the PC environment preventing porting, as we lacked the resources to do that.

The version of Pop-11 described here, which is available only as part of Poplog, runs on VAX computers under VMS and a variety of different computers running versions of the Unix operating system, e.g. Sun(SPARC), HP, Silicon Graphics, Sequent Symmetry, DECStation, VAX-ULtrix, MIPS, and PC running Unix.

0.11 The Poplog editor VED

Poplog contains a powerful screen editor VED, similar in some respects to EMACS, an editor developed at MIT. Compared with most editors VED considerably reduces the effort involved in developing and testing programs, mainly because it is so closely linked in to the Poplog system and its online documentation files. Emacs can be set up to be almost as useful, though it is not so closely integrated with Poplog.

Besides being useful for developing and testing software, VED can also be used for writing documentation. It includes special facilities for producing online documentation (like the online version of this file). Moreover, it can easily be used for text manipulation. I wrote a fairly simple VED program to transform the ascii version of this file into Latex, to facilitate production of a more readable printed version.

(For more details on online documentation in VED, see HELP DOCUMENTATION)

Details of the use of VED vary according to which terminals are available, so this document will not include information about the editor. If you have access to a computer running Poplog you should ask for advice. There should be a 'TEACH file' available which will show you how to use the editor on the terminals which are in use at your site. You will almost certainly need a chart showing how VED's operations are mapped onto function keys available on your keyboard. At the University of Birmingham we also use the Pop-11 interface to the X window system to provide an extendable menu-driven interface to VED, available freely to Poplog users on request.

XVED is a version of VED that supports multiple windows, one for each file in the editor, and additional facilities for driving the editor using menus and the mouse. (See HELP XVED)

Early versions of VED were often criticised for not including a regular expression matcher, available in most other Unix editors (ed, vi, emacs). This has been remedied since Poplog Version 14.5

0.12 Interactive programming

An important feature of Pop-11 is its inherent ability to be used in an 'on-line' mode. This means that commands in the language can be given at any time: there is no division between a phase in which you specify a program which is to be compiled, and a phase in which your programs run. You can interleave additions to the program and commands to run the program, using the same language and the same running process for both. Thus you can define a procedure, then run it to test it, then define a new procedure using the first one, then run it, then define new procedures, etc. If you find a procedure needs to be changed you can edit it without leaving Pop-11 and the new version will immediately be available without re-compiling other procedures or re-linking your program. (Recompilation will be required if the procedure had been declared as a constant for efficiency.)

To make all this possible Pop-11 contains an 'incremental compiler' which is part of the system when programs are running. In most languages the compiler is a separate program which runs to transform your program file into a file called an object file, and then you have to link the object files together with any libraries used, and then finally you can start running the program. Pop-11 (like the other Poplog languages) makes this multi-stage process unnecessary. You can directly compile and run extensions to your program at any time, because the compiler remains a part of the running system.

You can also recompile a procedure, e.g. after extending it in some way. Except where procedure identifiers have been declared as constant, the newly compiled procedure will automatically be accessed by all previously compiled code, and the the space taken by the previous version of the procedure will be reclaimed automatically by the garbage collector.

The use of the incremental compiler enables development and testing to be far more rapid than with most non-AI languages, such as such as Pascal, Fortran, C and C++. Some languages with an interpreter offer a similar facility (e.g. BASIC, some versions of LISP). The advantage of an incremental compiler over an interpreter is that it produces machine code, so that programs run faster. The advantage of using an interpreter is that it reads in programs faster, and can provide more flexible development aids.

Some argue that the interactive method of developing programs is undisciplined. They assume that the best way to use a computer to solve a problem is to:

```
(a) specify the problem,
(b) devise an algorithm to solve the problem
(c) prepare a program embodying the algorithm
(d) compile the program
(e) run the program
(f) go back to b or c to fix 'bugs'
```

This method of using a computer is ideal when

```
(1) the problem is well defined
(2) the information structures required can be designed easily
(3) the algorithms for operating on them can be devised easily
(4) the developer never makes a programming mistake
(5) the developer is a perfect typist
(6) the process of development never leads to a modification
    of the problem
```

If the last three conditions are not met then a slow and expensive iteration of steps (c) to (f) must be performed. Often even the first three conditions are not met. The problem may be well defined but so complex that initial algorithms do not cover all cases, or include inconsistencies or other errors, so that development and testing involves a slow iteration through steps (b) to (f). Sometimes even the problem is not very well defined, for instance where requirements for a program can only be developed as a result of running a prototype in the intended environment. In that case the iteration even involves step (a).

Whilst conditions (1) to (6) can be met for some cases (such as routine data processing) they cannot be met for complex problems, especially problems involving ”user-friendly” interfaces for people, since it is very difficult to find out what people will find friendly. Asking them is no good, since people cannot tell in advance what they will find awkward or convenient. Thus a system that allows a prototype to be built and then rapidly modified in order to try out variants on real users, can lead to far better designs than those that do not support incremental development.

Incremental development can also be particularly helpful for beginners who do not have the experience to understand problems in advance of doing the design. So the use of testable and modifiable prototypes can be a powerful learning tool.

The use of an incremental compiler (or interpreter) with an integrated editor and other program development aids can enormously speed up the process of producing and testing initial draft versions of programs. Since each new portion of program can be tested quickly before the next portion is added, without a slow process of re-linking, an environment like that of Poplog can encourage more thorough testing, leading to more reliable programs, as well as saving programmer effort. This is why many software developers now favour ‘Rapid Prototyping’ for solving complex and ill-defined problems.

A consequence of using an incremental compiler is that there is not really any such thing as a program in Pop-11. In many languages a program consists of a set of procedure or function definitions (explained below) together with some commands to the computer, which make use of those definitions; and the whole program has to be specified completely before any commands can be obeyed. In an interactive language like Pop-11, you can go on indefinitely interleaving running commands and defining new procedures or modifying old ones, usually working entirely within an editor that communicates with the compiler. The commands and procedure definitions may either be typed in at the terminal or read in from a previously constructed file.

Pop-11 does not require you to separate your data and your programs. Data-structures such as lists, strings, vectors and arrays can be created with their contents in the same files as the procedures that use them. For example the pattern matching facilities in Pop-11 use lists as patterns. These pattern lists can be embedded in procedure definitions, using expressions like

```
if sentence matches [the ??nounphrase1 ?verb the ??nounphrase2] then
```

As we shall see, this can make it easy to write down complex instructions and also makes it possible to mix factual assertions with other kinds of programming constructs, e.g. using commands like:

```
add([boat at ^place])
```

to store information in a Pop-11 database that other commands can then manipulate.

0.13 Declarative and procedural languages

In some languages (e.g. Prolog) there are also ASSERTIONS, which can be used to give the computer information, e.g. the information

```
tom is the father of dick or, in Prolog:

father(tom,dick)
```

In some languages it is possible to store generalisations, or inference rules, like

```
a father is a male parent, or
x is the father of y if x is a parent of y and x is male,
```

or, in Prolog:

```
father(X,Y):-parent(X,Y),male(X).
```

Such languages also allow the construction of QUESTIONS, such as

```
who is the father of joe?
```

or, in Prolog:

```
?- father(X,joe).
```

These are examples of the 'declarative' style of programming.

Pop-11 is not based primarily on declarative programming, but there is a subset, called the 'database package' which provides something like assertions and questions, and uses procedures like 'add' mentioned above and others called remove, lookup, present, allpresent, foreach and forevery, all described below. (In the Birmingham Poprulebase package the database is re-implemented in a way that provides more efficient indexing.)

This Pop-11 database package is built on more primitive facilities, described in the next few chapters. A Prolog-like extension to the database was implemented by Steve Hardy in the library described in the online HELP SUPER file. A more sophisticated system, designed for building expert systems or problem solving programs based on "condition-action" rules can be found in the "newpsys" library (pronounced "newpeesys"), described in the online file HELP NEWPSYS.

Note: NEWPSYS has been made obsolete by the POPRULEBASE library available from the Birmingham Poplog FTP directory.

Finally, the Prolog subsystem of Poplog is available to Pop-11 programmers when needed for a richer style of declarative programming.

0.14 The need for procedures

Some advocates of declarative languages suggest that logic programming should be used for all tasks. The idea is that one should be able to state WHAT the problem is and leave it to the computer to figure out HOW to solve it. Thus all one needs is assertions general facts, and questions, and not imperatives saying what to do when.

It is very natural, however, for people to think in terms of instructions as well as in terms of assertions and inference rules, and that is one reason why we have provided both in Poplog. For example, if someone asks you the way to the station, you will not usually succeed in communicating if you give lots of facts about where the station is located. Instead you will probably give some instructions:

```
go down that road until you come to the post office,
then turn left and.....etc
```

Of course, if the questioner knows the town very well, it may suffice for him to be told:

```
the station is two blocks north of the post office.
```

In order to make use of that information he will have to find some way of translating that into a plan for getting from where he is to the post office. The plan will contain a set of instructions about what to do when. Similarly, if you want the computer to do something other than tell you the answers to factual questions you often need to be able to give it instructions.

Even if all you are interested in is storing information and getting answers to questions relating to the information, someone must first tell the computer how to store information, and how to respond to questions. Thus instructions of some sort are required as the basis even for a purely declarative system.

0.15 Monitors

Another thing missing from Pop-11 (and many other languages) is the concept of a 'monitor' or 'demon': a program which waits until some condition becomes true, and then immediately takes control and carries out its instructions. Thus you cannot say in Pop-11 something like

```
'if ever the value of X becomes 99 then print out a warning'.
```

It is possible, using advanced facilities in Pop-11, to extend the language to allow such things. In particular Pop-11's interface to the X window system allows such monitors, or demons, know as "event handlers" to be attached to graphical windows or control panels. It is possible to attach a Pop-11 "call-back" procedure to a screen object. The procedure is then run whenever the user performs an action involving that object with the mouse, e.g. pointing at the object and clicking a mouse button.

There are also a few special purpose monitors built into Pop-11. For instance there is a procedure which is run whenever an error occurs, and the user can define that to take appropriate action. Also there is a user-definable procedure called 'interrupt' which is run whenever the user types an interrupt character at the terminal (usually CTRL-C). Another is the user-definable procedure pop_after_gc which runs whenever an automatic garbage collection has occurred.

Further, it is possible to attach a monitor to a variable in Pop-11 by declaring the variable as "active". This means that a procedure can be associated with it which is run whenever the value is accessed, and another procedure which is run whenever the value is updated. For more information see the online file HELP ACTIVE_VARIABLES

Another feature that makes it possible to monitor events is the fact that access to data-structures goes via procedures rather than via compiled offsets. This means that by re-defining the procedures for accessing particular data-structures one can insert "traps" that are activated whenever the contents are examined, or changed. The user-definable "methods" supported by the Objectclass package also provide monitor-like facilities.

0.16 TEACH files and the VED editor

This document is not intended for absolute beginners working without any help. Having a working Pop-11 system and the VED editor makes it much easier to learn the language by trying out the examples and varying them.

Many beginners do best by working on mini-projects at a computer terminal. After a few weeks of practical experience, such people may find this document useful for revision purposes, and as a way of learning more about Pop-11 more quickly than by working through interactive 'TEACH files'. In case the reader has access to a full Poplog system, brief information about the online files is presented here.

TEACH files are read using the editor, invoked by the TEACH command. Usually the first command is

```
teach teach
```

which introduces the use of VED reading teach files.

Since the editor allows two (or on some terminals more) files to be visible at once on the screen it is often convenient to learn about programming by having a teach file and a user file visible at the same time. The teach files give information, examples of programs, and suggestions for practical exercises.

The following interactive 'TEACH' files, provided with Poplog introduce the use of the editor VED, and its role in developing programs.

```
TEACH TEACH
TEACH VED
TEACH VEDPOP
TEACH MARK
TEACH LMR    (=Load Marked Range)
```

For an absolute beginner, with no experience of programming, the following TEACH files provide a succession of mini-projects which introduce both AI concepts and Pop-11 programming techniques:

```
TEACH RIVER
TEACH RESPOND
TEACH RIVER2
TEACH RIVERCHAT
```

More general introductions to Pop-11 facilities are provided by additional teach files, many of whose contents overlap considerably with this primer. An overview of currently available TEACH files is provided by

```
TEACH TEACHFILES
```

There are usually additional files available from Universities that use Poplog for teaching, e.g. Sussex and Birmingham. E.g. teach files from Birmingham are available via ftp at

ftp://ftp.cs.bham.ac.uk/pub/dist/poplog/teach

0.17 The HELP command

The VED 'help' command invokes files which more compact descriptions of Pop-11 facilities, without lengthy tutorial introductions. The following gives an overview of help files:

```
HELP HELPFILES
```

There are many hundreds of help files giving information about error messages, system utilities, editor procedures, system concepts, syntax, data structures, arithmetic, the pattern matcher, library facilities (e.g. expert system tools), Poplog process initialisation, how to tailor the system, etc. There are also 'news' files summarising the main recent developments.

For the Prolog and Lisp subsystems there are also collections of help files, though not as many.

0.18 The REF command

The VED ‘ref’ command may be used to access Poplog files which give a definitive description of the Pop-11 language, and its main system utilities. These files are definitely not intended for beginners, and many of them define quite sophisticated concepts quite tersely. The file called

```
REF REFFILES
```

gives an overview of the main REF files for Pop-11, VED and the external interface. It is summarised at the end of this primer. There are additional REF files available for various sub-systems, for instance REF files for the X interface facilities, REF files for Objectclass, and so on.

The rest of this introduction is a summary of the core facilities in Pop-11. Many references will be given to online documentation which describes more advanced features, or which give more detailed information for experienced programmers. A full description of Pop-11 would require a document several times the size of this one.

We start in the next chapter by describing in some detail an example which illustrates how to use Pop-11 as a list-processing language with a fairly conventional syntax. Later chapters provide more general definitions of the language and some of its most useful facilities.

1 CHAPTER.1: INTRODUCTION — THE ROOMS EXAMPLE

This chapter presents a simple Pop-11 program worked out in some detail in order to give a feel for some of the structures in the language, including the definition of procedures, and the use of lists, conditionals and looping constructs.

Not all the details will be explained fully in this chapter. More general and complete explanations are given in later chapters. Ideally the reader should try out the example on a computer running Poplog, but, for readers with some programming experience, it should be intelligible without that. If some of the examples use a notation which is hard to understand, reading later chapters should make things clearer.

1.1 POP-11 facilities illustrated in this chapter

This chapter will introduce a number of concepts, including these:

o the difference between ”declarations” which define something for future use and ”imperatives” that instruct some action

o declaring variables, both global and local

o defining named procedures that perform complex actions built out of simpler actions.

o accessing values of variables and changing them

o constructings lists to store information about the world

o various operations on lists, including comparing them, searching them and building new lists

o ”atomic” data items which can be stored in lists or assigned to variables, namely words, numbers and text strings

o arithmetic operations, e.g. used to compute areas and volumes from dimensions of rooms

o various forms of syntax for giving the computer instructions, including

```
- running a procedure with some data and getting a result back
- giving a sequence of explicit instructions to make up a complex
  action
- using a conditional expression which does a test before acting
- a "loop" instruction which repeats some action until a "stopping
  condition" is reached
```

o The ‘lexical’ rules that define how Pop-11 breaks up program text into separate items.

o the difference between syntax words, like ”vars”, ”define”, ”if”, ”->”, ”for”, ”then”, ”do”, ”enddefine”, and variables which store data.

o the difference between procedures that print something out for the user to see, and procedures that produce a result internally to be assigned to variables or used by other procedures.

o the difference between ‘built in’ procedures that are part of the Pop-11 language, such as

```
- the printing procedure "pr", and
- the arithmetical operations "+" (addition) "*" (multiplication)
```

and the ‘user-defined’ procedures created out of the built in ones, such as the procedure display_data, defined below.

o The use of comments in program code, starting with ";;;"

o "Declaring variable" warning messages

o Error messages printed out when a running program finds a mistake

o Incremental development and testing of a package of related procedures.

If you are an experienced programmer the concepts will be familiar, though the syntax may be new. For novice programmers the concepts and the syntax will be new. Later chapters will introduce concepts that will be new even for many experienced programmers.

1.2 The ROOMS DATABASE Example

Suppose you have the dimensions of a set of five rooms in a building whose names are "room1", "room2", ... "room5".

For each room you have measured the length, the breadth and the height.

Suppose you want to be able to answer a variety of questions, for instance

o questions about individual named rooms, e.g. questions about the perimeter, the total floor area, or total volume.

o questions about all the rooms

o questions about all the rooms that satisfy a certain condition (e.g. area greater than some given number).

Those are fairly easy problems to solve in many programming languages. We'll show how to do them simply as a way of introducing the concepts and techniques of Pop-11.

1.3 Preliminary analysis

First you have to find a way to represent the initial information, and then you have to define procedures for operating on that information in order to answer the questions.

There are many ways you can express this sort of information. Below you'll see how to give the room dimensions to Pop-11 in the form of a list of lists. The procedures for answering the questions then have to use operations on lists.

The procedures to be defined in this introduction will each have a header line, which indicates the name of each procedure, the 'arguments' required by the procedure (i.e. data for it to operate on), and whether the procedure produces a 'result' for use by other procedures. The latter may have an arrow "->" in the header followed by a variable called an 'output variable'. An example is this procedure heading, which will be found below.

```
define perim(len, breadth) -> total;
```

It defines a procedure called "perim" which takes in two arguments (the value of the variable len and the value of the variable breadth) and then it returns one result, the value of the variable total. Some procedures do not have "->" and a following output variable (result variable).

Some of the procedures will use other procedures. For instance, display_data will use display_room, and display_room will use the procedures perim, area, volume. All these procedures will be defined below. If you have access to Poplog with an online version of this document you will be able to mark and load the definitions and test out the examples.

The procedures to be defined are summarised thus:

```
display_data(list_of_lists)
    Given a list of lists of information about rooms print out
    general facts about each one

display_room(list)
    Given a list containing information about just one room
    print out information about its name, its perimeter, its
    area and its volume.

perim(len, breadth) -> total
area(len, breadth) -> total
volume(len, breadth, height) -> total
    The above three take in numbers corresponding to measurements
    of a particular room, compute a new number, and return it
    as a result. (They leave it on the Pop-11 stack, instead of
    printing it out.)

findroom(name, list_of_lists) -> data
    Given the name of a room and list of information about all rooms
    find the data about the named room and return it as a result.

find_and_show(name, list_of_lists)
    Like the previous procedure, but print out the information
    instead of returning it as a result.

find_and_show_all(namelist, list_of_lists)
    Do the same not just for one named room but for the rooms
    whose names happen to be in the list namelist
```

Representing room dimensions using a list of lists

A major design decision for any program is (a) which information has to be represented in the program and (b) how it should be represented.

We assume that for each room we have four items of information, namely its name, in the form of a Pop-11 word, and three numbers representing its length its breadth and its height.

We shall represent individual rooms with a four element list, thus:

```
     [room2 16 11 8]
i.e.
    [<name> <length> <breadth> <height>]
```

Note that unlike Prolog and some other languages, Pop-11 does not always require items in a list expression to be separated by commas. Later we'll see contexts where commas are needed.

Then we can represent all the rooms by means of a list of four element lists of the form shown above. Note that we are using position in the list to indicate the difference between length, breadth and height. We could have used a more verbose representation, such as

```
     [room2 [length 16] [breadth 11] [height 8]]
```

```
or
     [room2 [breadth 11] [height 8] [length 16]]
```

This freer format (called an ‘association list’) would have allowed the information to be presented in a more flexible form, with associations stored in any order, but would take up more space and would require more complex processing, so for now we’ll use the simpler representation.

This is how you might give Pop-11 information about the set of rooms. First we use the word ”vars” to tell Pop-11 to declare a ”global” variable, called ”rooms”. We sometimes use the more accurate word ”identifier” rather than ”variable”. But for now we’ll ignore the difference.

```
global vars rooms;
```

Then we construct a list of lists, giving for each room its length, breadth and height in feet (or whatever units we have chosen to work with). Finally we use the assignment arrow ”->” to ask Pop-11 to ”assign” the list to the variable ”rooms”.

```
[[room1 14 12 8]
 [room2 16 11 8]
 [room3 15 11 8]
 [room4 10  9 9]
 [room5 21 11 9]] -> rooms;
```

In Pop-11 it is possible to combine the assignment and the declaration into an ‘initialised variable declaration’, thus:

```
global vars rooms =
[[room1 14 12 8]
 [room2 16 11 8]
 [room3 15 11 8]
 [room4 10  9 9]
 [room5 21 11 9]];
```

Note that Pop-11 instructions can go over several lines, and a semi-colon ”;” is used to terminate most declarations and most imperative instructions.

The above creates a very simple ”database” of information about five rooms in the form of a list of lists of words and numbers.

The occurrence of a pair of square brackets [....] tells Pop-11 that a list is to be constructed. Nesting the brackets tells Pop-11 to make a list of lists, as in this example.

If you are using VED you could mark that example, starting from the ”vars” line down to the second semi-colon, then give the ”load marked range” (LMR) command. (See TEACH LMR). Nothing visible will happen when you have done that. But it will enable you to do the examples that come later. You can check that it has been compiled by giving the command to print the value of the variable ”rooms” thus:

```
rooms ==>
```

This should produce the following printout:

```
** [[room1 14 12 8]
```

```
    [room2 16 11 8]
    [room3 15 11 8]
    [room4 10 9 9]
    [room5 21 11 9]]
```

The two asterisks are printed by the Pop-11 print arrows "==>" and "=>", the former being used to produce neater formatting when printing long lists or other structures. "==>" prints only one (possibly complex) item at a time, unlike "=>" which can all the items that have been left on the Pop-11 "user stack", described below, and which does not format lists. This is how "=>" would print out the list

```
rooms ==>
** [[room1 14 12 8] [room2 16 11 8] [room3 15 11 8] [room4 10 9 9]
    [room5 21 11 9]]
```

1.3.1 Lexical rules for reading programs

Pop-11 (like any other textual programming language) has some built in "lexical analysis" rules, which specify how to analyse a sequence of characters into "text items" or "lexical items", from which a program is constructed. Normally the text items are words, numbers or strings. In this example we have:

```
words like "room1", "room2" etc.,
words that are made of a single square bracket, "[" or "]",
```

and

```
numbers, 10, 12, 8 etc.
```

The lexical analysis rules of Pop-11 state that a word starting with a letter may include a number, so that "room1" is accepted as a single word, not a word and a number, whereas "1room" would be split into a number and a word, i.e. 1 and "room". Many expressions typed without spaces will be broken into separate text items. For example:

```
x1+33             will be broken into:  x1  +  3

member(x, list)   will be broken into: member  (  x ,  list )
```

A later chapter will give more on lexical rules in Pop-11.

Printing out the value of "rooms"

We have declared "rooms" as a GLOBAL variable. That is, it is not declared inside any procedure, unlike some of the variables to be introduced below. This means that it can be referred to at any time, and its value altered, or used, or printed out. E.g. we can give a command to print it out, using the 'pretty-print' arrow ==>, thus:

```
rooms ==>
```

If you are using VED you can get that command obeyed by marking the line and then giving the load marked range command (or pressing the LOADLINE key). The computer then prints out the value of the variable "rooms", something like this (where the two asterisks are produced by the print arrow "==>".

```
** [[room1 10 12 8]
    [room2 6 11 8]
    [room3 15 11 8]
    [room4 10 12 9]
    [room5 21 11 9]]
```

Defining procedure display_data

Let's start by defining a master procedure, called "display_data", which takes in a list of lists like that above, then for each room extracts the name and the three numbers representing the length, the breadth and the height, and then prints out the name followed by the perimeter, area and volume. So for room3 it might print out something like:

```
Room 3, length 15, breadth 11, height 8, ..... etc.
```

If we already had a procedure called "display_room" that took in a list like

```
[room3 15 11 8]
```

to print out the information regarding room3, then we could repeatedly use that procedure in display_data to print out information about each room.

Using Pop-11 we could express this as follows:

We might define define display_data as follows, using two 'local' variables 'list_of_lists' to hold the complete list of information, given as input to the procedure, and 'room' to refer to the data for each room in turn:

```
define display_data(list_of_lists);
    lvars room;
    pr('ROOM INFORMATION');
    pr(newline);
    ;;; Now print information about each room in the list
    for room in list_of_lists do
        display_room(room)
    endfor;
    pr('-----------------------------------------------');
    pr(newline);
enddefine;
```

This tells Pop-11 that a new procedure is to be defined.

1. The first line says that the procedure is to be called 'display_data', and that when it runs it requires as input one object which, within the procedure, will be called "list_of_lists". The object can be called anything else in other places: the name, or variable, list_of_lists is private, or 'local' to the procedure.

2. The second line declares the additional variable "room" as "lvars" (a special kind of local variable called a lexical variable, explained in a later chapter), The variable "room" will be used to refer in turn to the list of information about each room. Since Poplog version 15 the input variable "list_of_lists" in the procedure header will be automatically declared as "lvars". In earlier versions it would default to "vars" (explained below), unless explicitly declared otherwise.

3. The third line says that the string of characters 'ROOM INFORMATION' should be printed out on the terminal.

4. The fourth line says that a "newline" should be printed. That simply means that the next lot of printing will continue on a new line, instead of to the right of 'INFORMATION'.

5. The next line, starting ";;;" is a comment, which is ignored by Pop-11.

6. The next three lines give a Pop-11 looping (or iterative) instruction of the form:

```
        for <variable> in <list> do <actions> endfor
```

This tells Pop-11 that the <variable> should take a succession of values from the elements of the <list>. For each value it should perform the <actions> specified. When there are no more values, i.e. when it has got to the end of the <list>, it should continue with the instructions following "endfor".

In the definition above, this "for" instruction uses the local variable 'room' to refer to the first element of the list, then the second element of the list, then the third element, etc. It does not matter how long the list is each element will be referred to in turn, and the repetition (or "iteration" as it is sometimes called) will continue as long as necessary, till every room, i.e. every individual list in the larger list_of_lists has been dealt with by display_room.

Thus when you design this procedure you do not need to know how many rooms will be in the list.

The instruction in the middle of the "for loop"

```
            display_room(room)
```

states that the procedure called 'display_room' should be applied to whatever is referred to by 'room'.

But display_room has not yet been defined, and we shall have to define it later.

7. The next instruction tells Pop-11 to print a string made of a lot of hyphens.

8. Finally the word "enddefine" merely signals the end of the procedure definition.

Notice how the word "define" starts a procedure definition that ends with "enddefine" and the word "for" starts a loop instruction that ends with "endfor". There are many similar pairs of matching opening and closing brackets in Pop-11. Most languages have such things but they differ from one language to another in their details.

If you have access to Poplog you can "mark and load" the above procedure. It will complain that you are using display_room, which has not yet been defined. It will declare the variable automatically for you, and print out

```
    ;;; DECLARING VARIABLE display_room
```

You can try to run display_data nevertheless, giving the command:

```
    display_data(rooms);
```

Try to mark and do that command. It will start by printing out ROOM INFORMATION

Then it will try to run the "for loop" and you will get an error message starting

```
    ;;; MISHAP - enp: EXECUTING NON-PROCEDURE
    ;;; INVOLVING:  <undef display_room>
    ;;; ......
```

Later you will learn to read error messages. For now just note that this one is produced because you tried to get the procedure display_data run, but it failed because it used the procedure display_room which you have not yet defined.

You can give display_room a temporary definition, thus, to suppress the above message:

```
define display_room(room);
    ;;; Note "room" is implicitly declared as "lvars room;"
    ;;; simply print the list room, followed by a new line
    pr(room);
    pr(newline);

enddefine;
```

This is not very interesting, but it can be used to test the previous definition. Note that each line beginning ";;;" is a 'comment' and will be ignored by Pop-11.

If you are using VED, then first mark and load the above definition of display_data, then re-try:

```
display_data(rooms);
```

This time it should print out the following instead of the error message:

```
ROOM INFORMATION
[room1 14 12 8]
[room2 16 11 8]
[room3 15 11 8]
[room4 10 9 9]
[room5 21 11 9]
------------------------------------------------
```

Defining the subroutine display_room

We now have to define the procedure display_room to give more interesting information. Look back at the list of information assigned to the global variable 'rooms'. It was a list of lists. Each embedded list contained the name of a room and three numbers, from which we want to be able to compute things like perimeter and area of the room and print them out.

So we define display_room to cope with such a four-element list, as its input.

Using the built in printing procedure 'pr' and three procedures we shall define later, for computing perimeter, area and volume, we could re-define the procedure display_room like this:

```
define display_room(list);
    ;;; Print out information about the room given in list
    lvars list, room_name, room_length, room_width, room_height;
    list(1) -> room_name;
    list(2) -> room_length;
    list(3) -> room_width;
    list(4) -> room_height;
    pr('INFORMATION CONCERNING: '); pr(room_name);
    pr(newline);
    pr('  perimeter is: ');
    pr(perim(room_length, room_width));
    pr(newline);
    pr('  area is: ');
```

```
        pr(area(room_length, room_width));
        pr(newline);
        pr('  volume is: ');
        pr(volume(room_length, room_width, room_height));
        pr(newline);
    enddefine;
```

(Note: this could be defined considerably more compactly, using more powerful facilities, some of which will be explained below. The input variable "list" has been explicitly included in the lvars declaration list, though since Poplog Version 15 that is redundant.)

If you mark and load that procedure it will complain that you have yet to define perim, area, and volume, but it will automatically declare them for you

```
    ;;; DECLARING VARIABLE perim
    ;;; DECLARING VARIABLE area
    ;;; DECLARING VARIABLE volume
```

Let's look at the definition in detail. The first line says a new procedure is being defined called 'display_room'. When it runs it must be given one thing as input, which will be referred to as 'list' in the procedure. The second line, starting ";;;" is a comment, ignored by Pop-11.

The next line declares the input variable list, and four additional names of local variables to be used in the procedure:

```
        lvars list, room_name, room_length, room_width, room_height;
```

The next line is an imperative which says, take the first element in the object called 'list' and assign it to the variable 'room_name', which can then be used later in the procedure to refer to it:

```
        list(1) -> room_name;
```

Notice that the expression 'list(1)' can be used to refer to the first element of a list. Similarly 'list(2)' refers to the second element, and so on.

To avoid repeatedly having to extract the second element we use the assignment arrow to store the result in a variable 'room_length', which is used several times in the procedure. Similarly with the remaining items of information.

In a later chapter we will see that there are alternative methods of extracting selected elements of a list using the Pop-11 pattern matcher. In this case, since we want to get all the elements out of the list and assign them to four variables, we can use a still more more compact form based on the multiple assignment operation in Poplog, and the "explode" procedure which simultaneously returns all the elements of a list on the Pop-11 stack:

```
    explode(list) -> (room_name, room_length, room_width, room_height);
```

Try replacing that line with the four assignments in the definition, and recompile it.

Going back to the original definition of display_room, we see that Line 8 of the definition has two imperatives,

```
        pr('INFORMATION CONCERNING: '); pr(room_name);
```

which print information on the screen. The first instruction includes a string delimited by the single quote character ‘, also known as the string quote character in Pop-11. All the items in the string are printed by the procedure pr, exactly as they are given.

By contrast the next instruction uses the unquoted variable ”room_name”. so instead of printing ”room_name” it prints the VALUE of that variable, which, if our program is working, will be a word, e.g. ”room1” or ”room2”, etc., depending on which list is given as input to the procedure display_room.

The imperative:

```
pr(newline);
```

Causes subsequent printing to be started on a new line.

A line like the following simply prints out a string, which in this case also include several spaces:

```
    pr('  perimeter is: ');
```

The next imperative:

```
    pr(perim(room_length, room_width));
```

This is actually composed of two instructions. The first is

```
    perim(room_length, room_width)
```

gives the values of ”room_length” and ”room_width” to a procedure called ‘perim’ (yet to be defined, since it is not built in to Pop-11), and expects that procedure to produce one RESULT. That is followed by

```
    pr( ... );
```

which takes the result and prints it. E.g. if room_length has the value 16 and room_width has the value 12 then this should print out 44. Here we are using the expression

```
perim(room_length, room_width)
```

to refer to the number which is the perimeter of the room. So when we define the procedure called ”perim”, we have to ensure that it produces a result which is a number, i.e. something which will be referred to by this sort of expression, and printed.

The remainder of our procedure uses similar techniques, first using the procedure area, which has yet to be defined, then a procedure volume, also to be defined. Note that the line:

```
pr(volume(room_length, room_width, room_height));
```

implies that VOLUME will be given three inputs, not two like the others.

Defining the procedure perim

Here is how you might define perim to calculate the perimeter. It should take two numbers, add them, then multiply by 2 to get the total perimeter, which is then the result of the procedure.

```
define perim(len, breadth) -> total;
    lvars len, breadth, total;
    (len + breadth) * 2 -> total
enddefine;
```

If you are using VED, mark and load that procedure. (If you are using Poplog version 15 or later, the "lvars" line is not needed.)

The first line has two new features. First we are here defining a procedure with two input variables 'len' and 'breadth', not just one as before. (Note: we cannot use 'length' for the first argument, as that's already a name of a Pop-11 system procedure. If we used 'length' we'd get a mishap message. Hence 'len'). Secondly the procedure header ends with '-> total', which implies that this procedure is to produce one result. What that result will be will depend on what is assigned to the variable 'total' in the procedure.

The occurrence of '-> total' in the header is not itself an assignment. It is merely an indication of how the variable "total" is being used in the procedure. I.e. it is an 'output local' variable, whereas "len" and "breadth" are 'input locals'. Although the output local declaration looks like an assignment, it does not cause the procedure to assign anything to the variable. Rather it indicates that after the procedure is used there will be a 'result' available, and the user must take care to assign something to the variable "total" within the procedure, to be used as the result.

We can test the procedure thus, using "==>" to print out the result:

```
perim(5,3) ==>
** 16
```

(Mark and load that command. Try it with different numbers to make sure it always gives the right answer.).

Actually, "==>", the 'pretty print' arrow is not needed for printing out something as simple as a number. It is intended for printing more complex structures, like lists of lists. So in this case, we could use the simple print arrow "=>", thus:

```
perim(5,3) =>
** 16
```

Pop-11 has many built in procedures besides "+" and "*" which can be used to create either complex expressions or complex imperatives. More of them will be introduced later.

Defining the procedure area

Similarly we can define AREA, using the multiplication symbol "*":

```
define area(len, breadth) -> total;
    len * breadth -> total;
enddefine;

;;; and now test it:
```

```
area(18,12) =>
** 216
```

Prior to Poplog version 15 the procedure header should be followed by

```
    lvars len, breadth, total;
```

Defining volume:

```
define volume(len, breadth, height) -> total;
    len * breadth * height -> total;
enddefine;
```

Mark and load that, and test it:

```
volume(5,5,5) =>
** 125
```

Having defined all the required subsidiary procedures, we can now test display_room, remembering that it requires a list of information about one room:

```
display_room([room17 9 6 7]);
```

And this command causes the following to be printed out:

```
INFORMATION CONCERNING: room17
  perimeter is: 30
  area is: 54
  volume is: 378
```

We could arrange 'prettier' formatting, but for now we shall ignore that. We can now run the master program on the list rooms, which we created above. Just to check, we can print out the contents of the list:

```
rooms ==>
** [[room1 14 12 8]
    [room2 16 11 8]
    [room3 15 11 8]
    [room4 10 9 9]
    [room5 21 11 9]]
```

Where the asterisks are again produced by "==>". Then run the master procedure, giving it the list rooms as input:

```
display_data(rooms);
```

mark and load that, which produces all the following printout:

```
ROOM INFORMATION
INFORMATION CONCERNING: room1
  perimeter is: 52
  area is: 168
  volume is: 1344
INFORMATION CONCERNING: room2
  perimeter is: 54
  area is: 176
  volume is: 1408
INFORMATION CONCERNING: room3
  perimeter is: 52
  area is: 165
  volume is: 1320
INFORMATION CONCERNING: room4
  perimeter is: 38
  area is: 90
  volume is: 810
INFORMATION CONCERNING: room5
  perimeter is: 64
  area is: 231
  volume is: 2079
------------------------------------------------
```

Notice that only two of our procedures do any printing, namely display_data, and display_room. The procedures perim, area, and volume produce RESULTS which they do not print out themselves. Instead they leave their results on the Pop-11 stack, explained in a later chapter. Items left on the stack can then be used by other procedures.

In this case, the results are printed in display_room. They could have been used for other purposes. For instance, we could use area to define volume, by first computing the area, and then multiplying by the height:

```
define volume(len, breadth, height) -> total;
    area(len, breadth) * height -> total;
enddefine;
```

Here 'area(len, breadth)' produces a result, which is then multiplied by the value of 'height' to produce the volume. This version of volume should produce the same results as the previous version.

1.4 "Top down" and "Bottom up" design

The above example illustrated the use of "top down" design. We started with a specification and definition of the main procedure display_data, then defined the procedure used by it, display_room, then defined the lowest level procedures used by that one, i.e. area, perim, volume.

However, there are some things that were already defined as part of the Pop-11 language, in particular the print procedure "pr" and the arithmetic procedures for addition and multiplication: "+" and "*"

A design methodology that starts by defining the "low level" procedures then moves up towards more and more complex procedures using those already defined is called "bottom up design". Most real life software development uses a mixture of top down and bottom up design. Top down design is usually possible only when you start by having a very clear idea of the problem and how to solve it.

Whether you do your programming top down or bottom up, it is absolutely essential to be clear about the ”ontology” of your problem. This includes the following topics, not all of which will be relevant to every problem:

```
What kinds of objects are there?
What sorts of properties can the objects have?
What sorts of relationships can the objects have?
What sorts of events or processes can occur involving those objects?
What sorts of problems can arise involving those objects?
What algorithms, or procedures are required to solve those
    problems?
```

Which of these would be relevant to the rooms example so far?

1.5 Exercise on the ROOMS example

The rooms example illustrates features of Pop-11 which need to be explained in more detail later on. The following exercise should test your understanding so far.

1. Modify the program so that in addition to the information shown above, the dimensions of each room are printed out, before the perimeter. You will need to work out which procedure should be modified, and how to modify it. All the techniques required have already been demonstrated.

Some further questions for revision:

2. Explain what the ‘for....endfor’ form achieves.

3. What is a local variable? What is the difference between an input local and an output local? How are additional local variables declared? (More on this in later chapters.)

4. In the left half of the imperative:

```
    area(len, breadth) * height -> total;
```

there are five expressions denoting numbers. What are they? Hint: here are two of them:

```
breadth, area(len, breadth),
```

5. Explain the difference between ‘->’ in a procedure heading and in an assignment instruction. One of them causes a value to be copied from the Pop-11 stack to a variable. The other specifies that a value should be left on the stack when a procedure finishes. Which is which?

6. What are lexical rules for? How many separate items would the Pop-11 lexical rules find in the following line:

```
[room3 44hat 999+x ]
```

Hint: you can find out by printing it out thus:

```
    [room3 44hat 999+x ] =>
```

or using listlength, to count the items, thus:

```
    listlength([room3 44hat 999+x ]) =>
```

7. Suppose you wanted to be able to answer questions about which rooms were connected to which, and questions about which rooms you had to pass through to get from e.g. room1 to room4. (This could be essential information for an adventure-game program, or a route-planning program). How might you enrich the ontology of the problem? E.g. what additional objects, properties and relationships might be involved? How would you represent the additional information in the the "rooms database", i.e. the list of lists of information about rooms? (There are several different ways of doing this. Try to invent one.)

A possible answer: suppose that the rooms are all rectangular and aligned with the compass points. Then a room that is connected to room5 via its eastern wall could have the following information added to its list:

```
[east room5]
```

Try drawing a plan of a house, including a corridor and some rooms, and then see how you can represent the rooms, the corridor, and their connectivity. What about the external doors? Would the above representation of dimensions as length, breadth and height adequately show the configuration of each room, or which a different representation be better?

1.6 Searching for information in the "rooms" database

In our example so far, we have used all the information in the list rooms in a rather verbose fashion. If we wanted to find information about a particular room, e.g. room3, we could use our knowledge of the order of the items in the list rooms, and get the third element and give it to display_room, using the expression 'rooms(3)' to denote the third element of the list called 'rooms', thus:

```
display_room(rooms(3));

INFORMATION CONCERNING: room3
  perimeter is: 52
  area is: 165
  volume is: 1320
```

But if we do not know the order in which information is stored, then we can define a procedure to search in the list for information about the room required. We want a procedure called findroom which

```
a. takes two inputs:
        a room name and
        a list of data about all rooms
b. produces the list of data for the named room as its result
```

For example

```
findroom("room3", rooms)
```

should produce as its result, to be left on the Pop-11 stack, the list

```
[room3 15 11 8]
```

We can do this, using techniques already illustrated, by using a "for" loop, and examining each piece of room information in turn until we find one whose first element is the name of the required room. At that point we can stop the search, using the "return()" command. If we get to the end of the list without finding the relevant room, produce a "mishap" message.

```
define findroom(name, list_of_lists) -> data;

    ;;; search list_of_lists for a list starting with name

    for data in list_of_lists do
        if data(1) = name then
            return();       ;;; i.e. stop the procedure
        endif;
    endfor;
    ;;; produce a mishap message
    mishap('DATA NOT FOUND', [^name ^list_of_lists])
enddefine;
```

We have again used the format:

```
for <item> in <list> do <action> endfor
```

This time instead of the <action> containing a single simple instruction that is to be obeyed with every room it contains a complex construction, with a part that is to be applied only if a condition is to be satisfied. The <action> is a conditional imperative, of the form:

```
if <condition> then <action> endif
```

Later we'll meet alternative forms of conditionals. The action in this case is simply to 'return', i.e. to jump to the end of the procedure. Because 'data' is specified as an output local variable in the procedure header, the value of 'data' which made the condition

```
data(1) = name
```

true will be the result of the procedure. In other words the result will be the list of information about the room with the name specified.

We can test findroom as follows using the word-quote symbol '"' to say that the first input given to findroom is a word, not its value. We use the print arrow '=>' to print out the result produced by findroom and left on the stack.

```
findroom("room3", rooms) =>
** [room3 15 11 8]

findroom("room5", rooms) =>
** [room5 21 11 9]
```

Alternatively the output of findroom can be given as input to display_room, e.g.

```
display_room( findroom("room5", rooms) );
```

(Try that.)

And if we give it a room which can't be found we get a mishap message:

```
findroom("room17", rooms) =>
```

```
;;; MISHAP - DATA NOT FOUND
;;; INVOLVING:  room17 [[room1 14 12 8] [room2 16 11 8] [room3 15 11
     8] [room4 10 9 9] [room5 21 11 9]]
;;; FILE     :  /home/staff/aaron/primer   LINE NUMBER:  2310
;;; DOING    :  findroom .......
    ......
```

The message will contain more information, and will have a different file name and line number for you. The extra information can be ignored for now, though it may be relevant when debugging complex programs. This 'mishap' message is produced by the line in our procedure definition which invokes the built in procedure called "mishap" (which is actually re-definable by the user, if different error handling is required.)

At this stage it may be difficult to understand everything in the definition of the procedure findroom. After reading later chapters on list processing, conditionals, and looping constructs, it may be useful to look back at this chapter, which may then make more sense.

Findroom is sufficiently general to be given different lists of rooms on different occasions. We can use it to define a more complex procedure called 'find_and_show' to print out the data concerning a given room. This procedure is to be given the name of a room (a word) and the list of information about all rooms. It uses findroom to dig out the list of data about the particular room, and gives that to display_room for printing. Here is a possible definition of find_and_show:

```
define find_and_show(name, list_of_lists);
    display_room( findroom(name, list_of_lists) );
enddefine;
```

Notice that the two items given as input to find_and_show are simply handed on as input to findroom. The latter produces a result which is used as input for display_room. The same thing could be said less compactly, though perhaps more clearly, by using an extra local variable "room" in the procedure below, to hold the result produced by findroom.

```
define find_and_show(name, list_of_lists);
    lvars room;
    findroom(name, list_of_lists) -> room;
    display_room(room);
enddefine;
```

To test the procedure, we can type:

```
find_and_show("room2", rooms);
```

Which produces the following printout:

```
INFORMATION CONCERNING: room2
  perimeter is: 54
  area is: 176
  volume is: 1408
```

Note that find_and_show does not produce a result (it has no output local variable, and leaves nothing on the Pop-11 stack). Instead it does some printing via display_room.

Because find_and_show produces no result our test imperative instruction does not use the print arrow "=>" to print out a result. Instead the instruction is terminated with a semi-colon.

1.7 Finding several rooms

If we wanted to dig out information about not just one room, but about several, then we could define a procedure to take a list of the names of wanted rooms, as well as the master list, and search for each required room in turn, using find_and_show:

```
define find_and_show_all(namelist, list_of_lists);
    lvars name;
    for name in namelist do
        find_and_show(name, list_of_lists)
    endfor
enddefine;
```

We can test it, by giving it a list of two room names and our global list_of_lists:

```
find_and_show_all([room2 room4], rooms);
```

which prints out:

```
INFORMATION CONCERNING: room2
  perimeter is: 54
  area is: 176
  volume is: 1408
INFORMATION CONCERNING: room4
  perimeter is: 38
  area is: 90
  volume is: 810
```

You may already be able to see that we can define procedures which are much more flexible than this. In this example we always dig out the same sort of information, even though it may be about different rooms. We could define procedures which produce different sorts of information. E.g. given a width, find all the rooms with that width, or find all the rooms which have a given length and height, etc.

Later we shall see that use of the Pop-11 matcher would make this sort of flexibility much easier to achieve.

1.8 Exercises

Before attempting these exercises, some readers may find it helpful to read the next chapter.

1. Modify the above examples to define a procedure called rooms_with_length which instead of being given the name of a room is given a length, i.e. a number. It should print out the name of every room with that length.

You could model the definition of rooms_with_length on the definition of findroom. It would use a similar looping structure, but instead of testing whether the first element of each room is the required name, it would test whether the second element is the required length. And instead of stopping (using return) it would then print the name and continue round the loop.

2. The procedure find_and_show_all could be very inefficient. For each room name it initiates a new search down the list, using find_and_show. If the list of room data is very long, this could be very wasteful.

Try defining a new version of find_and_show_all which is essentially like findroom, but modified to take a list of names instead of just one name. Further it would not have a output local variable, since it would print relevant information instead of producing a result. Two further modifications of findroom will be needed. The call of mishap at the end should be removed, and the conditional

```
if data(1) = name then
    return();       ;;; i.e. stop the procedure
endif;
```

will need to be replaced by something equivalent to the following, (which is not Pop-11):

```
if data(1) is a member of the list of names then
    display_room(data)
endif;
```

There is a procedure called ‘member’ in the Poplog system which takes an item and a list and produces the result TRUE if the item is in the list, false otherwise. So

```
member("room4", [room2 room4 room17])
```

would be an expression denoting the object true, whereas

```
member("room5", [room2 room4 room17])
```

would denote the object false. You could use a conditional starting something like:

```
if member(data(1), namelist) then
```

3. If the procedure called ‘member’ did not already exist in Pop-11 you could define it using concepts already illustrated. Try defining a procedure called ‘iselement’ which takes an object and a list of objects, and searches down the list using ”=” to see if the given object is the same as an element of the list. If so stop searching and produce the result TRUE, using the imperatives:

```
    true -> result;
    return();
```

If you get to the end of the list without finding the given object then the last action would be

```
    false -> result;
```

The search process could use the ‘for ... endfor’ syntax. When tested your procedure should behave like this:

```
iselement("cat", [mouse cat dog]) =>
** <true>

iselement("pig", [mouse cat dog]) =>
** <false>

iselement(4, [a 1 b 2 c 3 d 4 e 5]) =>
** <true>

islement(99, []) =>
** <false>
```

A procedure like iselement that returns a boolean (i.e. true or false result) is called a ”predicate”. Pop-11 has many built in predicates for testing individual objects (e.g. isinteger, islist, isword, isstring, isprocedure, and many more) and also ‘binary’ predicates for testing relations between things, including the arithmetical relations ‘<’, ‘>’, member(item, list), issubstring(string1, string2), and many more. The most commonly used binary predicates are ”=” for testing equality of structures, and ”==” for testing strict identity of objects. Two structures with the same components can pass the ”=” test and fail the ”==” test, for example two strings with the same characters:

```
'the cat' = 'the cat' =>
** <true>
;;; the following returns false because they are two distinct strings
'the cat' == 'the cat' =>
** <false>
```

1.9 Type-less higher order procedures

This section is best omitted by beginners.

Notice that in Pop-11 (like Lisp) a procedure like member (or iselement) can be defined to take any sort of object and any sort of list of objects. In a ‘typed’ language like Pascal you would have to define one version of the procedure for lists of numbers, another for lists of words, another for lists of lists, etc. This kind of generality is one of the things that gives Pop-11 its power.

Another example is the library procedure syssort. This takes a list and an ordering predicate, and returns a copy of the list that has been sorted according to the ordering predicate. For example, this will sort a list of words or strings alphabetically:

```
syssort([the cat sat on the mat], alphabefore) =>
** [cat mat on sat the the]
```

One can define a predicate that orders two lists L1 and L2 according to whether the second item of L1 is alphabetically earlier than the second item of L2.

```
define second_first(list1, list2) -> boole;
    if list1(2) < list2(2) then true else false endif -> boole
enddefine;
```

or more concisely

```
define second_first(list1, list2);
    alphabefore( list1(2), list2(2) )
enddefine;
```

We can use this ordering predicate to sort a list of lists:

```
syssort([[a d] [c a] [e b] [f c]], second_first) =>
** [[c a] [e b] [f c] [a d]]
```

syssort is an example of a second order procedure, since one of its arguments is a procedure, which it uses internally. It is very general because it can be applied to many different combinations of types of lists and ordering procedures. syssort is a type-less procedure: it can only be applied to a list and a procedure, but the list can contain objects of any type, and the procedure can be of any type that is suitable for the elements of the list. It is even possible to have a list of elements of different types, provided that the ordering procedure imposes a suitable ordering on them.

This is a small example of the sort of power and modularity that comes from using a language that treats procedures as objects that can be given to other procedures, and which does not require all type information to be available when a procedure is defined. Higher order programming of this kind can make software development and maintenance more effective because the same procedure can be re-used in many different contexts.

It might be thought that this freedom from type restrictions makes Pop-11 an unsafe language: the answer is that the types are checked at run time instead of compile time. The second argument that is given to syssort will check that the elements of the list are of the right type. If the types are wrong, an error message is produced.

```
;;; This will make alphabefore complain because it gets a number
syssort([ 21 4 3 55 22], alphabefore) =>

;;; MISHAP - STRING NEEDED
;;; INVOLVING:  21
;;; DOING    :  alphabefore get_run syssort ....
```

Sometimes it is easier to produce bug-free programs if the type checking is postponed till run time than if very complex checking is done at compile time and then a compiled program is produced that is ASSUMED to be safe to run. In some cases the assumption may turn out wrong. Ruling that out can make a language extremely restrictive and inflexible to use for complex software.

In the next chapter we shall give a somewhat more formal introduction to Pop-11, defining in more general terms some of the constructs illustrated here.

First here's a simplified overview of the Pop-11 virtual machine.

1.10 POP-11 : A MINIMAL MODEL

Added 11 Oct 1997

Understanding Pop-11 requires understanding at least the syntax and semantics of Pop-11.

1.10.1 The syntax of Pop-11.

Chapter 2 of this Primer includes a lot of information about the syntax, including:

o The "lexical" syntax, i.e. how the stream of characters read into Pop-11 is broken up into "lexical items", or "text items", including: words, strings, and numbers. (See also HELP WORDS, HELP STRINGS, HELP NUMBERS).

o The "compositional" syntax, i.e. how the text items can be combined in various ways to form more complex programming constructs, e.g. variable declarations, procedure definitions, procedure invocations, list expressions, vector expressions, commands, comments, the use of *infix* operators like "+", "-", "<>", etc. (Experienced students can try looking at REF POPSYNTAX)

1.10.2 The semantics of Pop-11

Chapters 2 and 3 primer give a lot of information about the semantics of Pop-11 including:

o Some of the types of structures that can be created and manipulated by Pop-11 programs.

o Some of the types of processes that can be generated by running Pop-11 programs. (See also TEACH STACK).

o The differences between the "compile time" semantics (i.e. the processes that occur when a program is being read in and translated into machine instructions) and the "run time" semantics (i.e. the processes that occur when your program has already been compiled, and starts running).

These notes provide additional information about what happens when declarations are compiled and when procedures are running.

1.10.3 The Pop-11 "virtual machine"

When Pop-11 starts up space is allocated for it in the computer. Some of the space may be in the main memory of the computer (RAM), some in temporary disk files. Where the space is allocated may change while the program is running. (Some parts of the Pop-11 system include bits that your programs cannot alter, so their space can be shared with other users, saving memory in the machine.)

Pop-11 breaks up the space into different areas which have different functions. These are parts of the Pop-11 "virtual machine". When your program is compiled, and when it runs, it causes changes to occur in the virtual machine.

The actual virtual machine is quite complicated because it is designed to support a variety of types of languages (including Lisp, Prolog and ML), and also interactions with the operating system, the file system, the windowing system and remote machines. However, a subset depicted in the diagram is fairly easy to understand, consisting of six parts of the virtual machine:

o 1. The dictionary, containing *words* (some of which are variables, some Pop-11 syntax words, and some simply words used as symbols to be manipulated or printed out),

o 2. The set of "built-in" procedures,

o 3. The "heap" where new procedures and structures are created, including temporary ones.

o 4. The procedure-call stack (procedure-activation stack)

o 5. The user stack (sometimes just called "the stack")

o 6. Input and output channels for communicating with the terminal.

All of these can change, both while programs are being compiled and while they are running. However during the running of the program it is normally the last four which change.

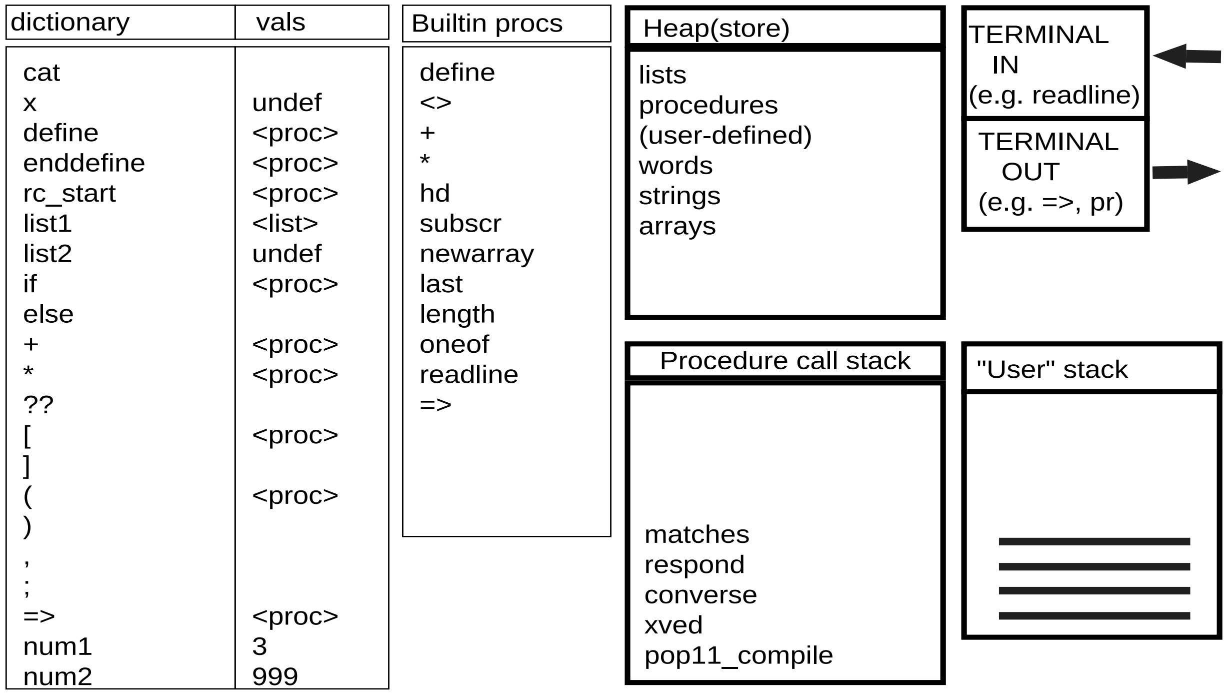
1.10.4 The dictionary

The dictionary includes (among other things):

o Syntax words known to Pop-11, including "define", "enddefine", "if", "for", "endfor", "vars", "(", ")", "[", "]", "=>", "->", ",", ";",

o Procedure names known to Pop-11, including "readline", "mishap", "ved_teach", "pr", "sqrt", "hd", "last", "random", and infix procedure names, like "+", "-", "=", "*", "<>",

A "MINIMAL" MODEL OF POP11



o Names of global variables defined in the Pop-11 system, or by your program.

o Other words which do not have any "meaning" for the program, but are used as symbols, e.g. in a list printed out by eliza, or a list of words typed in by the user in answer to a question. These words do not have a "value" cell associated with them, unlike variables, procedure names, and some of the syntax words. E.g. the list **[the cat]** contains two words which are put in the dictionary, but need not have any associated value (unless they are later used as variables).

When the Pop-11 compiler reads a declaration like the following,

```
vars list1, list2;
```

it adds the words "list1" "list2", to the dictionary, specifies (in the dictionary) that each word is a em variable, and associates with it a default *value*, i.e. an "undefined" value, which prints as `<undef list1>`, for example. Later an *assignment* using the format "-> list1", may assign something to the variable and that will change the contents of the value cell for that variable. The picture shows two user variables **num1** and **num2** which have been assigned numbers as values.

1.10.5 Built-in procedures

When Pop-11 starts up it has many built-in, previously compiled, procedures, e.g. all the syntax procedures used by the Pop-11 compiler, and procedures for creating and manipulating words, strings, arrays, lists, vectors, numbers, buffers in the editor, graphical windows, etc. These are in a "permanently allocated" portion of the Pop-11 virtual machine. I.e. you can't remove them.

1.10.6 The heap: for structures which may change

When your program creates a list, e.g. using expressions like these:

```
[the cat sat on the mat]

[perhaps in your fantasy we .^^.list each other]
```

as an Eliza program might do, then the list is assembled using locations in the heap (explained in TEACH WAL).

A "pointer" to the list is put on the user stack, i.e. an internal symbol with the address of the list (the place in the heap where it starts) is put on the stack. An assignment or a procedure requiring an input value, may remove it from the stack, while it remains in the heap. Later, if your program runs out of heap space, a special procedure called the "garbage collector" is automatically invoked. It may decide that nothing in your program can access that list any more (e.g. none of your variables has the list as its value, and none of the other lists that your program can still access includes the list). In that case the space will be cleared and can be re-used for something else.

(One source of inefficiency in programs is creating unnecessary temporary structures and giving the garbage collector too much to do.)

Besides lists, Pop-11 programs can create many other types of objects, some described in the Primer, Chapter 2. E.g. each Ved buffer is a "vector" of strings, stored in the heap.

1.10.7 The procedure call stack

When you have started up Pop-11 and the editor, Ved, there are already several procedures "active". First there are procedures that set up Pop-11, and find out what you want to do. If you specify that you want to run the editor, the setup procedures will invoke Ved procedures, and just wait until Ved finishes before they do anything else, such as close down Ved windows and terminate the Pop-11 process.

If you have created and compiled the conversational procedures specified in TEACH RESPOND, you can then run your procedure called *converse*, Ved will hand the instruction to Pop-11, which will start the procedure. Converse may then invoke other procedures, e.g. a procedure to find out the user's name and print a greeting, the *readline* procedure to read in a sentence typed by the user. It may then invoke the *respond* procedure to work out how to reply.

That procedure might call (i.e. invoke) the system procedure *matches* to decide what sort of sentence the user typed in.

At that point several procedures are all active, as shown (approximately) in the picture of the procedure call stack, where each procedure is waiting for the one above it (which it invoked) to finish. Only the one at the top is actually doing anything, and as soon as it has finished, it is removed from the top of the call stack and the one below it continues immediately with the next instruction. That may call yet another procedure, which will then go on the call stack.

Thus the contents of the call stack are constantly increasing and decreasing. For each active procedure (either running, or waiting to continue) the call stack needs information about which procedure it is, what its next instruction is, whether it has "local" variables (lvars) and if so what their current values are.

If you get a MISHAP, the "DOING" line tells you which procedures were in the procedure call stack (apart from "system" procedures which are not printed out).

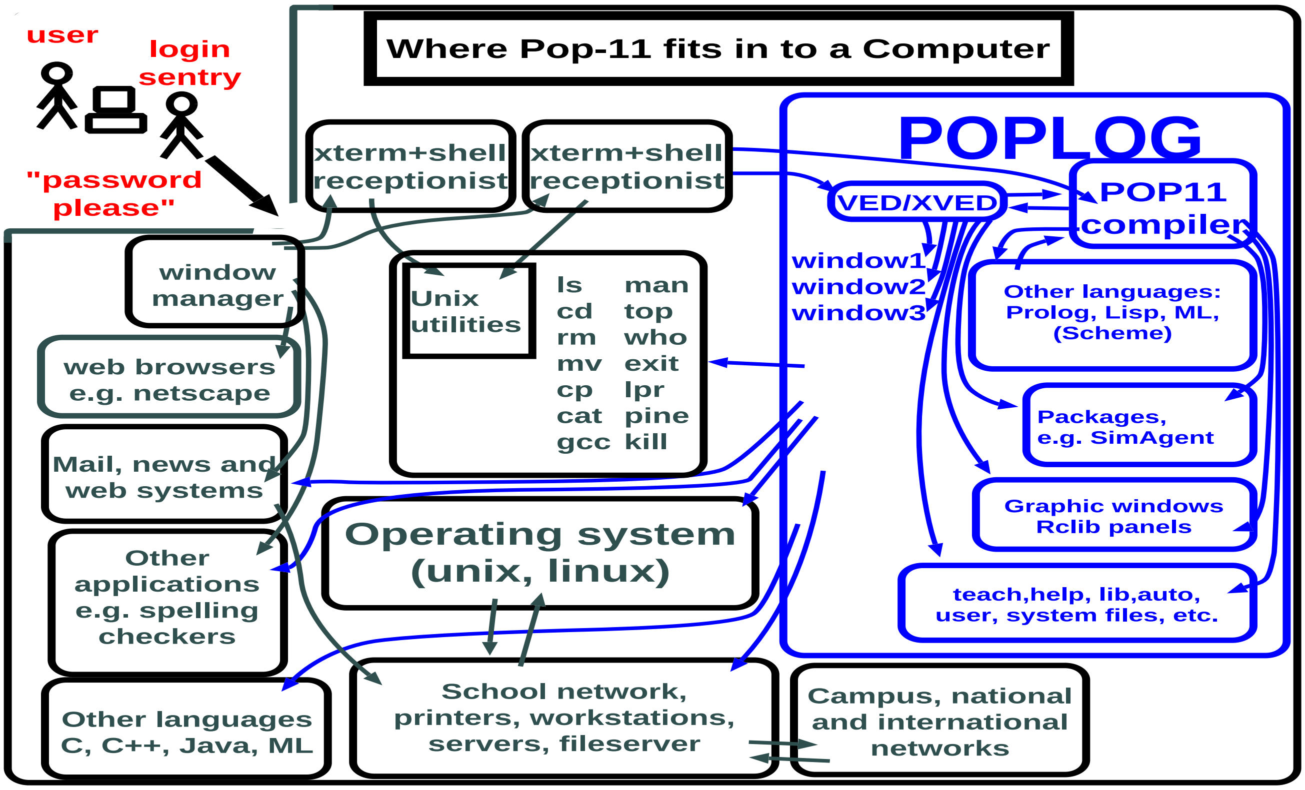


Figure 1: Diagram showing Poplog, Pop11 and the rest of the operating system

1.10.8 The user stack

If a procedure (e.g. *readline*) creates a list, the list is created in the heap, and a pointer to it (its address in the heap) is put on top of the stack. Then another procedure can get the last item constructed, by examining the top of the stack. E.g. if this command is run, then

```
readline() -> sentence;
```

the procedure *readline* pauses while the user types things. Pressing RETURN wakes up *readline*, so that it reads the characters from the terminal input channel (possibly via the editor), breaks them up into words, assembles them into a list in the heap, and puts its address on the stack. The assignment operation moves the address to the value slot currently associated with the variable ”sentence”.

The value slot is in the dictionary if ”sentence” is a global variable, declared with *vars* or in the current procedure activation record if it is a local variable declared with *lvars*.

”Arguments” can be passed to a procedure via the user stack. E.g. if *converse* includes the command,

```
respond(sentence) =>
```

that first of all puts the value of the variable *sentence* on the stack, then suspends *converse* and runs *respond*. The latter takes the item off the stack and then does things with it (e.g. if it's a list, the words in it may be used to decide how to reply). When finished *respond* puts another item on the stack, its result. Then the print arrow => procedure takes it off, prints it through the output channel (a character at a time), and then *converse* continues with its next instruction.

The chapter on procedures and the stack, Chapter 3, gives a lot more information about how procedures use the stack.

1.10.9 Input and output channels

These are used for reading in characters typed at the terminal, and for printing out characters, e.g. in the form of words, or numbers, or spaces, newlines, etc. Pop-11 can also read things in from files stored on the disc, or send information out to files on disk: Ved uses that capability to read your files and save them afterwards. So a more complete model would show additional input and output channels, and all the program and documentation libraries.

1.10.10 Poplog and the operating system

For some purposes, understanding how Poplog, Pop-11, and the Editor works requires knowledge of other parts of the operating system, which is usually some version of unix or linux. The following diagram gives an approximate impression of the relationships, though many details are omitted, such as the fact that Pop-11 may have to communicate with the operating systems memory management sub-system, or its filemanagement sub-system.

1.10.11 Conclusion

This section may make it easier for you to understand some of the remaining portions of the Primer. Much of the explanation here has been simplified. However, this model will enable you to do a lot of the programming required for an introductory course. The rest of the PRIMER provides a lot more information. Further information is in REF files, HELP files and TEACH files.

2 CHAPTER.2: INTRODUCTION TO THE SYNTAX AND SEMANTICS OF POP-11

A programming language has two main aspects: (A) the permitted sequences of symbols making up a program, and (B) their meanings. (A) is often often called the syntax and (B) the semantics of the language. However, the notion of "semantics" or "meaning" in this context is often ambiguous as sometimes people who talk about the meaning of a program are referring to what objects it creates and manipulates in the computer and sometimes they are referring to things in the world that the program models or represents.

For example, if a program builds a database of information about rooms using a list of lists (like the example in Chapter 1), then a portion of the program has one meaning insofar as it specifies which lists are constructed or examined in the computer, and a totally different meaning insofar as the information manipulated is about rooms and their measurements, i.e. things that have nothing to do with the computer.

2.1 Internal semantics and external semantics

It might be useful to call the first "internal semantics" and the second "external semantics". The internal semantics will be concerned with manipulation of symbolic structures in the machine. These, like the external programming language, will have a SYNTAX, i.e. there will be rules specifying which structures can be built and how they can be manipulated. And these internal structures may themselves have a semantics, insofar as they refer to things in the world.

In that case programs have a syntax and internal and external semantics. The internal structures are the internal semantics of the programs, but they too have a syntax and an external semantics.

Generally a programming language is defined in terms of its syntax and its internal semantics. It's up to the user to determine how to give it an external semantics, by applying the language to different sorts of problems.

2.2 Expressions denote objects. Imperatives denote actions

In a language like Pop-11 the meanings are of different sorts: in particular, EXPRESSIONS refer to OBJECTS, IMPERATIVES to ACTIONS. This is only a rough and ready distinction and many Pop-11 expressions both denote objects and specify actions that create those objects and perhaps also have other side-effects. For example, if the variable, (or identifier) x has the value 3 and the variable y has the value 4 then the expression

```
(x + y) * (y - x)
```

(externally) denotes the number 7. Internally it denotes a machine representation of 7. However it also can be thought of as an imperative which causes various things to happen in the machine. In particular it causes the value of x to be added to the value of y and the result saved on the Pop-11 "stack", then the value of x is subtracted from the value of y and the result saved on the stack. Then the two things on the stack are multiplied and the product left on the stack.

This dual aspect of expressions in Pop-11 as also having an imperative meaning is quite pervasive and is worth learning about as it will help with developing and debugging programs. In fact 'imperative' is the more basic concept in this sort of language.

Some further useful information about the imperative semantics of Pop-11 is found in the online TEACH STACK file, which gives a more detailed tutorial introduction to the Pop-11 "virtual" machine. Chapter 3, below, gives further information about the stack.

2.3 Compile time vs run time processes

There is another "imperative" aspect of programming language constructs. So far we've mentioned the internal actions that are produced when the commands are obeyed by the computer, i.e. at "run time". There is an earlier process that occurs when your instructions are read in by the Pop-11 system, e.g. from a file on the disk, or from the editor buffer, or from what you type at a terminal. Pop-11 has a "compiler" which reads the commands, analyses them and translates them into "low level" machine instructions which will later be obeyed by the computer. So the Pop-11 expressions cause additional processes to occur BEFORE the program is run. These processes include:

```
    o Lexical analysis - breaking the "input stream of characters" into
      separate symbols, namely words, numbers and strings.

    o Syntactic analysis - working out how to analyse complex sequences
      of symbols, like the analysis of (x + y)*(y - x) given above.

    o Code generation - i.e. production of the compiled procedure
      containing machine instructions.
```

This is a slight oversimplification, but it will suffice for most purposes. These processes happen at "compile time", i.e. when the program text is being read in, analysed, and translated into a machine code version. When the program is later obeyed and the machine code instructions executed, things happen at "run time". We could think of the internal semantics of the program as having two aspects: the compile time semantics, which determine the processes that translate source code into machine code, and the run time semantics which determine what happens later when the machine code instructions are executed.

Understanding the difference between compile time and run time is important because different things can go wrong at those times. E.g. at compile time you can get syntactic errors, like leaving out a closing bracket or a semi colon. At run time you get "semantic" errors, like trying to add a number to a string, or trying to examine the 15th element of a list that has only 14 elements. Moreover, in languages like Lisp and Pop-11 whose syntax users can extend by defining new so-called macros or syntax words, the processes that occur at compile time can be modified by users, and doing this requires fairly detailed knowledge of what happens at compile time. This primer will not go into such details. Experts who wish to know more can read the online documentation in REF PROGLIST, REF ITEMISE, REF POPCOMPILE, and HELP MACRO.

Some languages have interpreters instead of compilers, and they do something different for the last stage, i.e. they build a structure that does not contain machine instructions, but symbols that can later be interpreted by another program, the interpreter, which performs actions under the control of the symbols being interpreted. In Pop-11, and many widely used languages, a compiler is used and the machine itself (the CPU, or central processing unit) is the interpreter, rather than another program.

Understanding all those processes is not essential for understanding how to design and develop programs, but it can help you design programs that are more efficient, and it can help you understand what goes wrong when there are obscure errors, especially in a language like Pop-11 that does not have a fixed syntax, but allows you to extend it by defining new forms of expressions and imperatives.

2.4 How a programming language is specified: virtual machines

Defining a programming language involves specifying which sorts of objects there are, and what sorts of actions can be performed on them, and learning the syntax for creating or referring to objects and for specifying or performing actions. In the case of languages like Pop-11, Lisp and Prolog the components of the language (e.g.

words, syntactic forms, procedure definitions) are themselves among the objects that the language can be used to manipulate, which is not the case for a language like C, or Pascal.

The set of internal objects and the actions that can be performed on them is sometimes referred to as a ”virtual machine”. The reason is that the objects (e.g. words, lists, strings, numbers, arrays) are not actually physical objects that you can see if you open up the machine. They are abstract objects that only exist as a kind of ”interpretation” of the processes going on at a lower level, which usually consist of turning switches on or off and sending electrical signals through wires, etc. In fact different virtual machines, corresponding to different programming languages, can be made to run on the same physical machine, by having compilers or interpreters for the languages. Thus the Prolog virtual machine is very different from the Pop-11 virtual machine.

Just to add to the confusion, one virtual machine can be used as a basis for ”implementing” another. So the Pop-11 virtual machine can be (and has been) used to implement a Prolog virtual machine. There may be several layers of virtual machines all running at the same time when your program runs: but at the bottom layer there's always something physical. Perhaps that's also a good way to think about how the mind is related to the brain.

Some virtual machine actions are concerned entirely with processes within the computer, like re-ordering a list of names, or adding two numbers.

Others may be concerned with information flowing into or out of the computer, e.g. text read in from or printed out to the terminal, or a file. Most of this introduction is concerned with actions within the computer, though you will also have to know something about how to print results of programs and how to get files on the disk compiled, or read in by your programs.

Specifying the internal semantics of Pop-11 then involves specifying which sorts of objects, i.e. ”which data types” the Pop-11 virtual machine can construct and manipulate, and also which sorts of operations it can perform on various sorts of data. Specifying the syntax is a separate matter: languages with different forms of syntax might be able to operate on the same data types in the same ways. (For example there's a lot in common between the data types of Lisp and Pop-11, and the operations available, but their syntax is completely different.)

In the next section some of the most commonly used data-types are listed and examples shown of the syntactic forms that can be used to denote them. Some of them are basic data-types whose instances do not contain other objects. Others are complex data-types, and their instances contain other objects. An example of a complex object in the previous chapter was a list of lists of words and numbers, used as a database of information about rooms. Integers, like 0, 3, 99, -55 and decimals, like 1.414, 3.14159, are basic data-types and do not contain other objects as parts. (Chapter 5 introduces a distinction between ”simple” items that are represented directly by bit-patterns in the computer and ”compound” items that are represented indirectly by pointers, or addresses.)

2.5 Some Pop-11 data-types and their external forms

Objects referred to in Pop-11 include numbers, words, strings, lists, and others explained later. These are all things that can exist in the Pop-11 virtual machine. There are standard notations in Pop-11 for referring to such things. A few examples are presented here, to introduce ideas used in the following sections. A more comprehensive list is given later in this chapter.

Two sorts of numbers

The two most commonly used types of numbers are

o Integers:

```
E.g. 66,   99876789,   -66,     0
```

o Decimals:

```
E.g. 88.532,   1.2345e3 (= 1234.5),   123.4e-3 (=0.1234), 0.0
```

Other types of numbers are described below. Summary and tutorial information on programming with numbers in Pop-11 can be found in the files: HELP MATH, and TEACH ARITH

Words:

```
E.g. "cat",   "ninety",   "fast_back",   "+**+",    "[",    ")"
```

Strings:

```
E.g. 'cat',   'ninety',   '66',   'a string with spaces ++**@@@@'
```

Words are delimited by double quotes, strings by single quotes. Further differences are described later.

Lists:

```
E.g. [a list of words]     [ [a list] [of lists] [66 77]]
```

Here is a list containing words, numbers and lists:

```
[name [joe bloggs] age 33 job [university teacher] sex male]
```

Note that inside a list expression you do not need to use the "quote" marks to refer to a word. Outside a list, if you wish merely to refer to the word, then use the double quote symbols, e.g.:

```
"elephant"
```

Without the context of quote marks or list or vector expressions, Pop-11 will take a word to refer to its value, assuming that it has been declared as a variable. Sometimes this will produce an error, because the word has no suitable value associated with it.

If matching pairs of square brackets in one of the examples above were replaced by braces, the result could be a vector containing words, numbers and vectors, i.e.

```
{name {joe bloggs} age 33 job {university teacher} sex male}
```

Lists are more flexible and general. Vectors are more compact. Later the differences between vectors and lists will be explained in more detail.

Pop-11 also provides several other data-types, described later, and allows the user to define new ones. Advanced programmers may find it helpful to look at the online REF DATA file, to get a more complete overview of the types of data in Pop-11 and some information about how they are represented in the machine.

2.6 Some Pop-11 actions

Associated with each type of data will be a collection of actions that can be applied to instances of that type. Some actions can be applied to several different types, e.g. printing.

The actions, or processes are produced by imperatives in Pop-11. Actions may create objects, compare them, search for them, store them inside other objects, print them out, etc.

There are many forms of imperatives for different purposes. For example, here is a command to calculate the sum of two numbers and print out the result:

```
99 + 66 =>
```

which prints out

```
** 165
```

Here is a command to concatenate two strings and print out the result:

```
'the cat sat ' >< 'on the mat' =>
```

which prints out

```
** the cat sat on the mat
```

(Pop-11 does not normally print the string quotes, but can be made to, by assigning true to the global variable pop_pr_quotes.)

Here is an imperative that declares the word "vec" to be the name of a new variable:

```
vars vec;
```

Here is an imperative that creates a vector of four elements and assigns it to "vec";

```
{a four element vector} -> vec;
```

Here is a command that accesses the third element of the vector and prints it out

```
vec(3) =>
```

which prints out

```
** element
```

The following command updates the third element of vec with the word "item", and the next one prints out the vector vec, which has been changed:

```
"item" -> vec(3);
vec =>
```

which prints

```
** {a four item vector}
```

2.6.1 Assignments in Pop-11 and other languages.

An assignment is a command to store an item in a variable (or identifier). Here is an assignment of a number to a variable.

```
66 -> num;
```

Read '->' as 'goes to'. Unlike most languages, assignments in Pop-11 go from left to right: first specify the object to be assigned, then say where it is to be stored. In C this would be written as

```
num = 66
```

In languages of the Algol family (e.g. Pascal) it would be

```
num := 66
```

In Lisp it would be

```
(setq num 66)
```

In some languages, e.g. Prolog, this sort of thing cannot be expressed.

In Pop-11 the assignment syntax using ">" can be used for more complex processes than updating a variable. In particular, it can also be used to invoke the "updater" of a procedure, as will be explained later.

2.7 Built in and user-defined procedures

Pop-11 has many different sorts of built in instructions, and new ones can be defined by the user. In Pop-11 these are called 'procedures'. In some other languages (e.g. LISP) they are called 'functions' rather than 'procedures', but the kinds of entities referred to are the same, even if their definitions look different.

(Several lists of built in procedures are provided in the Poplog REF files, grouped roughly according to their function, or according to the kinds of datastructures they operate on.)

Some languages distinguish functions from procedures. The former are sets of instructions that take some input and return a result, like the mathematical functions addition, multiplication. The word "procedure" would then be reserved for sets of instructions that do something but do not have this mathematical property. Pop-11 acknowledges that underlying both concepts is a single unifying notion of a set of instructions that can be obeyed, and simply uses the one word "procedure" to cover all cases. (Lisp uses the one word "function" to cover all the cases.)

A procedure is defined in Pop-11 by specifying a name, and instructions to be obeyed when the name is used in an imperative. Having defined a procedure you can then later use it. In Pop-11 lots more can be done with procedures, as explained in chapter 4.

There are different ways of putting instructions together to form procedures. One common form is just a sequence of things to be done one after another. Another form is a conditional which specifies that what is to be done may depend on the result of one or more tests. Another common form is a loop in which a set of instructions is obeyed repeatedly, until some terminating condition is reached.

The ability to use 'conditional instructions' is one of the things that make it possible for a computer to be intelligent, or appear to be intelligent. Conditionals provide flexibility in the use of instructions. This is because the program decides at run time which option to choose, rather than the programmer. However, the options available will normally have been anticipated by the programmer. In a language like Pop-11 that can create new programs at run time there is more scope for a conditional instruction whose options were not thought of by the programmer. (This is also true of Lisp and Prolog.)

The different forms of instructions available in Pop-11 will be illustrated and explained in this and later chapters: they are many and varied.

2.8 Errors and error messages

To err is human, and the machine will print out 'mishap' messages. Some of these report compile time errors. E.g. if you leave out the "+" between two numbers:

```
6 6 =>

;;; MISHAP - MSEP: MISSING SEPARATOR (eg semicolon)
;;; INVOLVING: 6 6
;;; DOING    : compile ...
```

The 'code' before the colon in the top line ('MSEP' in this case), is a key to a help file which will explain in more detail what this sort of error message is about. You can examine the relevant explanatory file with the this command in VED, or direct to Pop-11:

```
help msep
```

This will invoke the Pop-11 HELP facility which uses the screen editor VED and will print out an explanation of the missing separator type of error. It should only be used after you have learnt to use the screen editor.

2.8.1 Syntactic and semantic errors

Some mistakes are 'syntactic', or 'compile time' errors: you have not typed what Pop-11 regards as meaningful instructions. The 'MISSING SEPARATOR' error is an example. 'Semantic errors', or 'run time' errors occur when Pop-11 understand the instructions, but discovers in the course of obeying them that something is wrong, e.g.

```
"one" + "two" =>    ;;; Pop-11 doesn't understand English!

;;; MISHAP - NUMBER(S) NEEDED
;;; INVOLVING:  one two
;;; DOING    :  + compile
```

Pop-11 understood that it was being told to put two objects on the stack then add them and print out the result. There was no syntactic error. But when it tried to obey the instruction, then at run time it asked the procedure "+" to add the two items on the stack. This procedure (+) checked what it was given as inputs, and then invoked the error handler to print the message above message. Notice that the DOING line mentions that it was actually doing "+", as well as doing "compile". In the previous case it was only doing compile, so the error occurred at compile time.

In order to do anything Pop-11 has to 'compile' what you type in, i.e. translate your commands into machine instructions, which are then obeyed. So Pop-11 invoked compile, which invoked the "+" procedure to do something. Thus it was in the middle of doing both the procedure "+" and the (temporarily suspended) procedure "compile", when the error occurred. So:

```
Pop-11 runs compile
            compile calls +
                + calls the error procedure 'prmishap'
```

After the error it goes back to doing whatever it was doing before you typed the command which caused the trouble.

(Note for experienced programmers: there is an 'error-break' facility which enables the user to take control when an error or other interrupt occurs. This is done by re-defining either a procedure called PRMISHAP or a procedure called INTERRUPT, or both. The procedure POPREADY can be used for the latter.)

2.9 Comments in Pop-11 programs

You will find that inserting short explanatory passages (usually only a line) helps explain a piece of Pop-11 program. To facilitate that Pop-11 skips over the remainder of any line that contains three semi colons thus

```
;;; This is a comment
3 + 5 =>    ;;; Print sum of 3 and 5
** 8

/*
Longer comments, going over several lines can be enclosed
between the comment brackets "/*" and "*/", just like this
paragraph.
*/
```

All such text will be completely ignored by the Pop-11 compiler. We occasionally insert comments in one of these two formats in examples in this primer.

C programmers will recognize the extended comment syntax with opening and closing brackets "/*" and "*/". The difference is that in Pop-11 such comments can be nested, whereas they cannot in C. Thus in Pop-11 a comment can include a programming example which itself includes a comment.

2.10 Pop-11 Expressions: some examples

Before explaining how to define procedures, we need to explain some of the building blocks - expressions and imperatives, already mentioned.

We need expressions to tell Pop-11 which objects to manipulate. Among the objects we can manipulate are numbers, words and lists.

Here are some expressions in Pop-11 which denote numbers:

```
3
999
999 + 5
(999 + 5) - 666
```

Here is an imperative which contains an expression '999 + 5' and, using the so-called 'print arrow', tells Pop-11 to print out what the expression denotes

```
999 + 5 =>
```

That will make the computer print out:

```
** 1004
```

The two asterisks '**' are produced by the command '=>', and are usually an indication that what follows was printed by the computer.

Pop-11 can refer to very large numbers. Here is the number 35 raised to the power 75 and printed out (the result being too long to fit on one line):

```
35 ** 75 =>
** 638415358328838953519612090376318512663247375138619145584048066674
   345771037686059212745703916880302131175994873046875
```

Here is an expression denoting a list of words

```
[a list of five words]
```

Here is a list of three lists of words and numbers

```
[ [a dog 4 5] [a big cat] [66 silly old men 99] ]
```

If you want to refer to a list containing a single word in Pop-11 you can do it with the following sort of expression: [elephant]

This denotes a list containing the word "elephant". More precisely, it is an instruction to Pop-11 to create a list containing just the word "elephant". If you type the same thing again it will create another similar list, with the same word in it.

A program might use a list of words rather than a single word if, for example, it has to store someone's forenames, and you cannot tell in advance whether there will be only one forename, or several forenames. E.g.

```
[John Jones]
[Mary Sue Wilkins]
[Liberace]
```

Since a list can contain arbitrarily many words, using lists will be more flexible in that case. Later we shall see many examples of the use of lists to group things together.

To sum up: an expression is a piece of Pop-11 which refers to some object. It may be an object which always exists, e.g. the number 99, or it may an object created as a result of the use of the expression, e.g. a lists of numbers. It may be a simple, or atomic object, such as a number, or it may be a structured object which has other objects as components, e.g lists, vectors, words and strings.

In order to do something to any such object it is necessary to use an imperative which tells Pop-11 to perform some action.

2.11 Imperatives: some examples

2.11.1 Printing

We have already met the use of "=>" in an imperative to print something out. For example the following prints out a list of words:

```
[tom dick harry] =>
** [tom dick harry]
```

2.11.2 Assignment

Another very common form of imperative is an ‘assignment’, using the assignment arrow ‘->’. Here is how you assign a number to be the value of the variable ‘x’ and another to the value of y:

```
99 -> x;
99 + 5 -> y;
```

In both cases the left hand side is an expression which denotes a number. The whole thing says: ‘take the thing denoted by the expression on the left, and make the thing on the right (the variable x or y), refer to it from now on. Thus, in the second example y will refer to 104. This can be tested with another imperative:

```
y =>
 ** 104
```

Normally an imperative which does not end with the print arrow ”=>”, must end with a semi-colon, which is an ”imperative separator”. Later we’ll see that if an imperative is embedded in a larger structure it does not always need the semi-colon to terminate an imperative or expression.

A few more examples of assignments:

To give the variable x the value 33 do:

```
33 -> x;
```

to give the variable y twice the value of x do:

```
x + x -> y;
```

OR

```
(x + x) -> y;
```

Notice the difference between using the variable on the left and on the right of an assignment. On the left its existing value is USED. On the right a new value is SET, or assigned. So to assign the value of x to y do

```
x -> y;
```

To assign a list of numbers to list1 do something like:

```
[ 1 2 3 4 5 ]  -> list1;
```

To assign a list of words to list2:

```
[cat dog mouse elephant] -> list2;
```

To assign a list like list1 but in reverse order to x, do:

```
rev(list1)  -> x;
x  =>
** [5 4 3 2 1]
```

Although what rev does is intuitively clear, stating precisely what it does is slightly tricky: rev takes as input (off the Pop-11 stack) a list L1 and creates a new list L2 containing the same elements as L1, but in the opposite order. L2 is returned as the output of rev.

Notice that rev does NOT reverse the orignal list: that can remain totally unchanged, as can be tested:

```
list1 =>
```

2.11.3 Multiple assignment

Pop-11 allows you to assign to several variables at the same time, using a single assignment arrow. First we declare three variables, then we assign the numbers 111, 222, and 333 to them.

```
vars num1, num2, num3;
111, 222, 333 -> (num1, num2, num3);

num3 =>
** 333
```

(Multiple assignments did not work in the earliest versions of Pop-11).

2.11.4 Declarations of dynamic and lexical (static) variables.

Declarations may be used to introduce new variable names. E.g.

```
vars x, y;
```

This ‘declares’ to Pop-11 that you wish to use the names ‘x’ and ‘y’ as names for objects. It is a global and dynamic variable declaration. Any number of variables may be introduced in one declaration. The same syntax is used for introducing both global variables and variables that are local to a procedure, explained below.

Lexical variables are declared using a similar syntax, but using ”lvars” instead of ”vars”.

```
lvars num, lista, listb;
```

declares three lexical variables.

The difference between lexical and non-lexical (= dynamic) variables is rather subtle and will not be explained fully here (but see the Note below). The main difference is that a variable declared using ”vars” is dynamic and global and can be potentially accessed by any procedure, no matter whether it is defined in the same file as the ”vars” declaration or somewhere else, whereas a lexical variable has a more restricted ”scope”.

Such a variable can usually be accessed only by instructions in the procedure in which the variable was declared, or by procedures in the same file, or by procedures in the same compilation stream, as explained in the file HELP

LEXICAL. The word "lexical" is used to indicate that the scope of the variable is determined by the textual context in which it is declared.

For most purposes programs will run more quickly and be less liable to errors if the variables which are local to a procedure are declared as lexical using "lvars". (However, if variables are to be used in connection with the Pop-11 pattern matcher, explained in a later chapter, they may need to be declared using "vars", not "lvars".)

In some Poplog implementations some of the lvars will allocated to registers. E.g. on the VAX, the first two lvars in each procedure are treated as names of registers, for efficiency. On SPARC machines, the first seven lvars are held in registers. (These numbers may change.)

There are, in fact many different sorts of declarations, used for telling the Pop-11 compiler that a symbol is to have special properties, but they will not be discussed in detail in this primer.

For more information see these online Poplog files:

```
REF SYNTAX, REF POPSYNTAX, HELP VARS, HELP LEXICAL
```

For many purposes it does not matter whether you use "vars" or "lvars". However using "vars" rather than "lvars" for local variables can sometimes lead to obscure bugs. Also "lvars" variables are generally more efficient and allow some more powerful constructs to be used when procedures create new procedures while running. (Advanced programmers can get more information from the HELP LEXICAL and HELP LVARS files. The latter explains how so-called 'lexical closures' can be produced in Pop-11, as happens in functional languages such as Scheme, ML, Miranda and Haskell.)

Normally, unless you have a special reason for using "vars", such as wanting to use the Pop-11 pattern matcher (explained in TEACH MATCHES, and in Chapter 7, below), you should always use "lvars" for local variables.

For making changes to a Pop-11 global variable constrained to a particular procedure, the local declaration "dlocal" may be used, as explained in HELP DLOCAL. This is normally required only for advanced programming.

There is a slightly less convenient version of the matcher, described in HELP FMATCHES which works with lvars, and can be used by more experienced programmers, e.g. together with "sections". A more up to date version of this known as DOESMATCH, with associated libraries, is available from the Birmingham Poplog FTP site, along with documentation.

2.11.5 Declarations may contain initialisations.

It is often convenient to combine a declaration followed by an assignment into a single imperative. Thus, instead of writing the following:

```
vars list1, list2;
lvars num1, num2;

[a b c] -> list1;
list1 <> [d e f] -> list2;

33 -> num1;
num1 * 2 -> num2;
```

You can write the more compact, and perhaps clearer version:

```
vars
```

```
     list1 = [a b c],
     list2 = list1 <> [d e f];

lvars num1 = 33, num2 = num1 * 2;
```

Note that each initialisation consisting of the form

```
<variable> = <expression>
```

MUST be followed by a comma, unless it is the final one, in which case the semicolon ends the list of declarations.

WARNING: people who are used to using the symbol "=" as an assignment symbol may be confused into thinking that it can be used for assignments from right to left in Pop-11, like "=" in C, and ":=" in Pascal. This is not so. In Pop-11 "=" is used for initialising a variable ONLY in the scope of a variable declaration, such as "vars" or "lvars" or "dlocal", etc. In other contents the assignment arrow "->" should be used, as shown above.

Programmers who forget this point and write things like

```
z = x + y;
```

will in fact not assign a value to z but create an expression which does a comparison of the two values on the left and right of "=", which will result in the value TRUE or the value FALSE being produced. That value is left on the Pop-11 stack, as explained above.

2.11.6 Variables and constants

If you wish an identifier to be given a value once only, which is never changed thereafter, you can declare it using "constant" or "lconstant", the former being used for global constants, to be accessible everywhere, and latter being used for lexically scoped identifiers, usually accessible in only one file or one procedure. Constant declarations may also include initialisations.

Examples are

```
constant number_of_rooms = 66;

lconstant pattern = [ ?subject ?verb ?object];
```

2.11.7 Using mixed case and underscores in variables

Note: Pop-11 is a mixed-case language, so that each of the following:

```
list, LIST, List, lisT
```

is treated as different, and they can be used as names of different variables (though this is likely to cause confusion). Most system words use only lower case, though many of the identifiers concerned with the X window system include upper case letters.

Users may declare variables using upper or lower case. So:

```
lvars cat, Cat;
```

declares two different variables.

Occasionally we use upper case in our text, to refer to lower case identifiers, e.g. IF and ENDIF in order to make the words stand out.

One issue on which opinions vary is whether to use capital letters or ”underscores” to form long identifier names, e.g. some people would call a procedure

```
transform_input_list
```

whereas others would prefer

```
TransformInputList
```

There is no right answer. Many Pop-11 and VED system identifiers use underscores, e.g. procedures used by the compiler, such as:

```
pop11_comp_constructor   pop11_comp_declaration
pop11_comp_expr_seq      pop11_comp_expr_seq_to
pop11_define_declare     pop11_define_props
pop11_exec_stmnt_seq_to  pop11_loop_end
pop11_loop_start         pop11_need_nextitem
```

On the other hand, many of the system identifiers concerned with the interface to the X window system use mixed case, following the conventions of the X system, e.g.

```
XpolAppContext       XptActionHookWrapper    XptActionList
XptCheckCacheType    XptCheckCallbackList    XptCheckKeySymTable
XtWindowOfObject     XtUnmapWidget           XtUnrealizeWidget
XpwSetFont           XpwTextWidth            XptCallbackList
```

and hundreds more!

2.11.8 Using undeclared variables

Variables may be used without being declared. This is equivalent to declaring them globally, except that Pop-11 prints out a warning message.

```
3 -> xx;
;;; DECLARING VARIABLE xx        (Printed by Pop-11)
```

If you attempt to declare a variable which is already part of the Pop-11 system vocabulary, for example:

```
vars define;
```

you will get a mishap message:

```
;;; MISHAP - IDW: ILLEGAL DECLARATION OF A WORD
```

NOTE on variables for experienced computer scientists:

Like many implementations of the language LISP, Pop-11 has two kinds of identifiers:

(a) Lexically scoped identifiers (declared using "lvars", "lconstant", and "dlvars", as explained in HELP LEXICAL, HELP LVARS), which can only be accessed within the smallest enclosing text block in which they are declared. Such a text block may be any of the following:

```
    o   a procedure
    o   a lexical block delimited by "lblock  ... endlblock"
    o   a file
    o   a "compilation stream" (See REF PROGLIST)
```

(b) Dynamically scoped or "permanent" identifiers (usually either declared globally using "vars" or "constant", or made local to a procedure using "vars" or "dlocal"). Once these have been declared, they can be accessed globally, that is by expressions using those variables in other procedures, other files, etc. (The section mechanism described later (and in HELP SECTIONS) can be used to restrict unwanted access to permanent, dynamically scoped, identifiers.)

Dynamically scoped variables can in principle be implemented using either "deep binding" or "shallow binding".

o Deep binding, as used in Interlisp, stores variables and their values in a list, often called "oblist". This has the form of a list of name-value pairs, so that access to the value of a variable requires searching along the list to find the "most recent" binding for the variable. Starting a new procedure P in which X is a local variable merely requires adding a pair of items to the list, X and its value. While P is active, accessing or updating the value of X will use that most recent occurrence, ignoring others further along the list, representing local variables of other currently active procedures higher up the calling chain. However if another procedure Q is invoked by P or by something it invokes, and Q uses X as local, then Q will add another name-value pair for X to the front of the list, thereby "shadowing" the pair added by P. When Q procedure finishes it removes all the name-value pairs corresponding to its own local variables from the list, so that the previous ones will again be found by searching along it. Similarly when P finishes it will remove its name-value pairs allowing older bindings for X to be accessed.

o Shallow binding, by contrast, always stores the value of a variable, e.g. X, in the same portion of memory. This means that when a procedure P in which X is a dynamic local variable starts up, the existing value of X has to be saved somewhere (sometimes referred to as the "auxilliary stack"), so that the value can be restored when P finishes execution. The saved values need to be stacked, since whenever a procedure using X as local finishes, the top of the stack for X is copied back to the standard location for values of X and the stack shortened by one.

Shallow binding makes accessing values much faster than deep binding since no searching along a list is required. However deep binding makes context switching easier since all that's needed is to add or remove an initial segment of oblist, requiring a few pointers to be manipulated, whereas shallow binding requires all the values of local variables to saved (by copying) when a context is entered, and copied back when the context is left. Pop-11, like some, but not all, Lisp systems, uses shallow binding for dynamic local variables.

Lexical variables are more efficient than both, and introduce fewer bugs due to unintended interactions between programs, but they can be less flexible in some ways. Older versions of Pop-11, like older versions of Lisp, had ONLY dynamic variables. So the older text books for Pop-11 and older TEACH files use only "vars", even in situations where "lvars" would be preferable. This Primer uses "lvars" by default, except where "vars" is needed, e.g. for pattern matcher variables, or global variables.

There are some people who dislike lexical scoping and use only "vars". There are others who think that only lexical scoping should be used, and who never use "vars" for local variables!

(End NOTE on variables for experienced computer scientists.)

2.12 Procedure definitions are initialised identifier declarations

The previous chapter gave several examples of procedure definitions, in connection with the "rooms" database. These are all examples of imperatives. Each of those imperatives implicitly declared a new global variable, the name of the procedure, and assigned a procedure to it.

As an example, here is the definition of a silly little procedure called 'silly' which takes one thing as input and then prints it out twice using the print-arrow:

```
define silly(item);
    item =>  item =>
enddefine;
```

This contains an implicit global declaration of 'silly' as the name of a variable, whose initial value is the procedure defined here. I.e. a procedure definition is like a declaration that includes an initialisation. Just as we can use a declaration to initialise a variable to refer to a list, as in

```
vars people = [mary tom suzy dick];
```

so we can use the form

```
vars <name> = procedure ..... endprocedure;
```

to initialise a variable to refer to a procedure. The full syntax for this will not be explained till later. The main thing for now is to point out that the above definition of "silly" could be re-written as follows, using the syntax "procedure ... endprocedure" to form an expression that refers to a procedure without naming it:

```
vars silly =
    procedure (item);
        item =>  item =>
    endprocedure;

;;; Now test it
silly([the cat]);
** [the cat]
** [the cat]
```

However if silly is defined this way it will not have a name as part of the procedure, so printing it will not give its name:

```
silly =>
** <procedure>
```

We can give it a name using the updater of "pdprops", thus

```
[silly] -> pdprops(silly);
silly =>
** <procedure silly>
```

Note that the pdprops can be a list in which case only the first item will be used as the name. Other information associated with the procedure can be stored in the list. If no further information is needed then the name can simply be a word:

```
"silly" -> pdprops(silly);
silly =>
** <procedure silly>
```

What the ”define silly(item); ... enddefine” syntax does is equivalent to the above initialised declaration plus storing the word as the name of the procedure. See also HELP PDPROPS, and REF PDPROPS

2.13 Global and local procedure definitions

Where a procedure is to be accessible only within another procedure, or only within a file, its name can be made into a lexical identifier, e.g.

```
define lvars silly(item);
    item =>  item =>
enddefine;
```

Or if there is no requirement ever to be able to change the value of silly to be another procedure, or anything else, then it can be defined as a lexical constant, as in:

```
define lconstant silly(item);
    item =>  item =>
enddefine;
```

Warning: If you define a procedure using lvars or lconstant like this, you will not be able to test it by typing in commands using it. You can invoke it only by calling other global procedures that call these procedures, and are compiled in the same ”lexical context”. E.g.

```
define lconstant silly(item);
    item =>  item =>
enddefine;

define test_silly(item);
    [^item is about to be given to silly] =>
    silly(item);
enddefine;
```

If, in the editor, you mark and load both of those at the same time, then test_silly can be used to run silly:

```
test_silly([hello there]);
** [[hello there] is about to be given to silly]
** [hello there]
** [hello there]
```

But if you redefine silly you will have to recompile test_silly also, as they have to be compiled in the same compilation stream for test_silly to access the lexically scoped silly.

2.14 Procedure calls in Pop-11

The definition of silly included two imperatives of the form ”item =>”. These printing instructions are not obeyed at compile time, when Pop-11 reads in the definition. Rather, a procedure is then created in which the instructions are stored for future use (in the same login session - if you leave Poplog and then restart it later, it will not remember your previously compiled procedures, unless you make use of the saved image mechanism, described in HELP SYSSAVE).

So we need a means of telling Pop-11 to actually obey the instructions after they have been compiled.

We also need to tell Pop-11 which object it should give to silly as the ‘item’ to be printed out twice. This is done by writing the name of the procedure, followed by parentheses specifying the inputs (arguments) of the procedure:

```
silly([mary had a little lamb]);
```

Given the above definition of the procedure silly, this would cause the following to be printed out:

```
** [mary had a little lamb]
** [mary had a little lamb]
```

Terminology varies. We may say the compiled procedure is ‘run’, ‘invoked’, ‘obeyed’, ‘executed’, or ‘applied’. These all mean roughly the same, though some may be more natural in certain contexts. For instance we say the procedure silly was ‘applied’ to the list

```
[mary had a little lamb]
```

Some procedures take no arguments (no inputs). They are run without being applied to anything, though they still need the parentheses after the procedure name to signify that running is required, and not just reference to the procedure itself. One such procedure is the system procedure sysdaytime, which produces a string of information about date and time. We can run it, with no inputs, and use the print arrow to print out the resulting string. Note the empty parentheses signifying that the procedure needs no inputs:

```
sysdaytime() =>
** Sun Feb 27 11:56:11 GMT 1994
```

Contrast what happens without the imperatives:

```
sysdaytime =>
** <procedure sysdaytime>
```

2.14.1 Imperatives end with separators

Notice that all the imperatives requesting that Pop-11 do something ended with either a semi-colon or a print-arrow. Pop-11 will not start executing the imperative until it finds the semi-colon or the print-arrow, since until then it is not sure whether the imperative is complete or some additional portion is to be added. This is because Pop-11 allows a single command to be spread over several lines. (Individual lines and whole commands can be as long as you like: newlines and spaces are equivalent in Pop-11).

2.15 Using infix operators

In the imperative

```
3 + 4 =>
```

the symbol ”+” acts as the name of a built-in procedure to be applied to the numbers 3 and 4. The symbol ”+” is called an ‘infix operator’ because you can run the procedure by writing the name between the inputs, although the following ”parenthesised prefix” syntax would work just as well:

```
+(3, 4) =>
** 7
```

Just as ”+” can be used for addition, so can the asterisk ”*” be used as an infix operator for multiplication. So:

```
3 * 5 =>
** 15
999 * 999 =>
** 998001
```

Pop-11 provides many built in procedures whose names function as infix operators. Other examples are the equality predicates ”=” (which tests for the same type of object with equal components), ”==” (which tests for the very same object), the STRING concatenator ”><”, the generic structure concatenator ”<>”, the list constructor ”::”, and other arithmetic operators for division ”/”, remainder ”rem”, subtraction ”-” and exponentiation ”**”. (That is not a complete list.)

Infix operators have a numerical precedence which determines how they should be grouped if combined in complex expressions, like ”x + y * z”, and this is explained in Chapter 4. Chapter 4 also shows how users can define their own infix operators.

2.16 Procedures which produce results

Just as procedure definitions can extend the range of imperatives in Pop-11, so can they also be used to extend the range of expressions. For example, here is a procedure which, when given two numbers, produces another number which is got by doubling the first two then adding.

```
define doublesum(num1, num2) -> total;
    num1 + num1  + num2 + num2 -> total;
enddefine;
```

This defines a procedure whose name is ”doublesum” and which has two input (lexical) local variables, namely num1 and num2, and one output (lexical) local variable, namely total.

There are several different formats for procedure definitions, but we have now seen the two most common – one which produces no results and one which produces a single result. The formats for these are as follows:

```
define <name> (<arguments>);
    <declarations of local variables>;
    <body of procedure>
enddefine;
```

```
define <name> (<arguments>) -> <name of result>;
    <declarations of local variables>;
    <body of procedure>
enddefine;
```

If the input and output variables are not declared explicitly, they will default to lexical variables, as if "lvars" had been used. Before Poplog version 15, the default was as if "vars" had been used.

(Experienced programmers can use a compile_mode declaration, described in the file HELP COMPILE_MODE, to control what happens to undeclared input and output locals.)

Given the above definition of doublesum, the expression

```
doublesum(2, 3)
```

invokes the procedure, applying it to the numbers 2 and 3. More precisely, it first puts the two numbers on the stack, and then invokes the procedure. The procedure takes whatever inputs it requires from the Pop-11 stack, and then when finished puts its results on the stack, in this case the result of evaluating the expression 2 + 2 + 3 + 3, which gives the number 10. We can say that the whole expression "doublesum(2,3)" 'denotes' the number 10.

Similarly

```
doublesum(4, 5)
```

is an expression which denotes the number 18. Did you notice that DOUBLESUM is effectively the same procedure as PERIM, defined in the rooms example in chapter 1, even though its name, and the names of the variables it uses, are different? Convince yourself of the equivalence by studying them closely. Although they are different internally, they take the same sorts of arguments, and produce the same sorts of results, computed in the same way.

Generally, if a procedure produces a result then its name can be used to form an expression, by supplying it with appropriate argument expressions.

The arithmetic procedures "+" and "*" also produce results, and so by applying them to arguments we can form expressions denoting numbers:

```
3 + 5

3 + 5 * 2
```

The latter denotes 13, since the multiplication is done before the addition, for reasons which will be explained later.

When we define a procedure we specify whether it needs to be given any input and whether it produces any 'results'. For example the procedure silly was defined so as to take one thing as its input. We say it takes one 'argument'. Doublesum was defined to take two arguments, and produce one result. Silly printed things out, using the print arrow, but that is not the same as producing a result.

2.16.1 Procedure output values and the stack

When a procedure produces a result, the result is available for use by other procedures in the computer. But if something is printed out on the screen, this can't be used by other procedures, since the computer cannot see what is on the screen. Instead, results are stored for internal use in a special portion of the computer memory called the ‘stack’. This will be explained in more detail later. When a result is produced it can be assigned to a variable, or used as input to another procedure.

```
vars x;
3 + 4 -> x;                    ;;; assign the result of + to x.

silly (3 + 4);                 ;;; result of + is input to silly.
```

The word ”output” can be ambiguous: referring either to something printed out, or to a result left by a procedure on the ‘stack’. (Later, in Chapter 3, we explain in more detail what the stack is and how it works.) When we talk about a procedure producing output the context should make clear which is intended: printing something out, or leaving a result on the stack. Students often confuse printing something out and leaving a result internally for another procedure to use, because both can be described using the word ”output”. Similarly the word ”input” is used sometimes to refer to information typed in from a terminal or read in from a file, and sometimes to refer to the arguments handed to a procedure by another procedure within the machine.

The arithmetic procedures + and * each take two arguments (two numbers) and produce a result, one number which is left on the stack. Neither directly reads anything from the terminal nor prints anything out to the terminal, though they can be made to do so, e.g. the following could be part of an interactive session with Pop-11

```
: itemread() + itemread() =>
: 22
: 33
** 55
```

Where ”itemread()” runs a procedure to read in an item from the terminal.

2.16.2 Procedures returning more than one result

In Pop-11 a procedure can return any number of results, 0, 1, 2 or more, or even a variable number. E.g. the procedure explode, which takes a list or some other structure containing several items and produces the contained items as results, produces different numbers of results.

```
vars list1 = [a b c], vect1 = {1 2 3 4 5};

explode(list1) =>
** a b c

explode(vect1) =>
** 1 2 3 4 5
```

The procedure ”dest” when given a list, returns exactly two items, the head of the list (the first item) and the tail of the list (a list of all other items).

```
dest(list1) =>
** a [b c]
```

The notation for defining a procedure that returns two results is a slight extension of the notation shown previously. E.g. suppose you wanted to define a procedure that when given two lists as input, list1 and list2, returned two lists as output, namely list1 concatenated with list2, and list2 concatenated with list2. You could define the procedure with two output local variables thus:

```
define join_two(list1, list2) -> (out1, out2);
    list1 <> list2 -> out1;
    list2 <> list1 -> out2;
enddefine;

;;; Now Test it
join_two([a b c], [1 2 3]) =>
** [a b c 1 2 3] [1 2 3 a b c]
```

The two outputs can be simultaneously assigned to two variables;

```
vars new1, new2;

join_two([a b c], [1 2 3]) -> (new1, new2);

new1 =>
** [a b c 1 2 3]
new2 =>
** [1 2 3 a b c]
```

The same sort of notation can be used for a procedure that produces three results.

2.17 Exercises

1. Define a procedure with one input variable and three output locals, that takes a three element list of numbers as input and produces as output the squares of those three numbers (using "x * x" to get the square of x).

2. Try changing the definition of join_two to use initialised declarations for two of its variables, so that it becomes more compact, using the format:

```
lvars
    out1 = .... ,
    out2 = .... ,
```

It will have only four declarations and no additional instructions, but it will still work.

2.18 Introduction to printing in Pop-11

2.18.1 Two forms of print-arrow

Pop-11 provides several means of printing information so that it goes to the terminal, into a file, into the editor buffer, or into a data-structure.

One of the most commonly used is the print-arrow, which we have seen previously. This comes in two versions.

```
=> (ordinary print arrow)
```

This prints everything 'on the stack', preceded by '**', unless called from inside a procedure, in which case it prints out only the top item of the stack, preceded by '**'. The stack is explained below. Roughly think of => as printing out the 'unused' results of previously executed procedures.

```
==> (pretty-print arrow)
```

This prints out only ONE object. If the object is a list or a vector, then if printing it out would take more than one line, the object is printed in a special format using indentation to make its structure clearer.

Examples:

```
33 + 66 =>
    ;;; print sum of 33 and 66, preceded by two asterisks.
** 99

234 + 22  >  33 * 66  =>
    ;;; is 234 + 22 bigger than 33 times 66?
** <false>

X + 5 < Y =>
    ;;; is the sum of X and 5 less than Y?
    ;;; X and Y should have numbers as values.

99 + 5 = 95 + 9  =>   ;;; is 99 + 5 equal to 95 + 9?
** <true>

X = Y  =>       ;;; does X have the same value as Y?

X =>            ;;; print out the value of X.

10 // 3 =>      ;;; prints out remainder and quotient

** 1 3
```

Complex structures, like lists of lists, can print out in a rather messy and unreadable manner using "=>"

```
[a [nested list of several words] and [another nested list
           of several words] too long to print easily] =>
** [a [nested list of several words] and [another nested list of several
     words] too long to print easily]
```

Compare how the pretty print arrow prints the same list:

```
[a [nested list of several words] and [another nested list
           of several words] too long to print easily] ==>
** [a [nested list of several words]
      and
      [another nested list of several words]
      too long to print easily]
```

When lists are deeply nested the printout from => can be very confusing, so it is then best to use ==> instead. You can in fact always use '==>'. This is especially useful for a list of several lists of similar format. E.g.: Make a list of lists of information about people

```
[
    [the mother of mary is suzy]
    [the father of mary is joe]
    [the mother of joe is miranda]
    [the father of joe is fred]
] ==>
```

Which causes the following to be printed out neatly:

```
** [[the mother of mary is suzy]
    [the father of mary is joe]
    [the mother of joe is miranda]
    [the father of joe is fred]]
```

Whereas using => would have produced:

```
** [[the mother of mary is suzy] [the father of mary is joe] [the mother
         of joe is miranda] [the father of joe is fred]]
```

Note that => or ==> can be used to terminate an imperative without a semi-colon. They act as ‘separators’ for imperatives.

2.18.2 Other printing procedures

Pop-11 provides a number of built in procedures for printing things in a more controlled format than the print arrows. In particular, the procedures pr and spr print things without starting a new line, and without printing the asterisks ‘**’. The difference is that spr prints a space after printing its argument. So:

```
pr("cat"); pr("mouse"); spr("hat"); spr("coat"); pr(99);
catmousehat coat 99
```

Similarly npr can be used to print things separated by newlines. It prints a newline after printing its argument.

```
pr("cat"); pr("mouse"); spr("hat"); npr("coat"); pr(99);
catmousehat coat
99
```

Using the built in printing procedures, it is fairly easy in Pop-11 to define your own procedures to print things out in whatever format you like. More information about printing can be found in the online file HELP PRINT. The REF PRINT file gives more complete information about printing mechanisms. There are utilities that simplify formatted printing, in particular the procedure printf, described in REF PRINT

2.19 Conditionals and conditions

One of the things which makes computers so versatile is their ability to decide for themselves what to do, instead of always doing exactly what they are told. This often depends on the use of conditionals, i.e. commands to do something if something is true. Pop-11 includes conditionals, of which a simple form would be

```
if <condition> then <imperative> endif
```

for example,

```
if x = 10 then x => endif;
```

which will print out the value of x, if it is 10, and do nothing otherwise. Conditionals must include a ‘condition’ between ”if” and ”then”, and may include any arbitrary Pop-11 imperative (which may be a sequence of imperatives) after ”then”. Conditionals are often typed over several lines when they are more complicated than the above example. For instance:

```
if x = 10 and y > 19
or member(x, list)
then
    x =>
endif
```

A condition is an expression whose value is either the object TRUE or the object FALSE. These are called ‘boolean’ objects, of which there are exactly two in Pop-11. (They are named after George Boole the logician.) There are two built in system names, ”true” and ”false” which refer to the booleans. They print out as <true> and <false>.

```
true =>
** <true>
false =>
** <false>
```

In the above conditional, the expression ‘x = 10’ could serve as a CONDITION, because its value will be either true or false. E.g.

```
vars x;
99 -> x;
x = 10 =>
** <false>
```

This in turn is because the infix operation symbol ‘=’ denotes a procedure which always returns a BOOLEAN. It compares its two arguments, and if they are the same the result is true, otherwise false. (Strictly speaking ”=” is more complex than this, since the type of comparison it performs can be made to depend on the types of objects compared, as explained in HELP CLASSES and REF KEYS. A more detailed explanation of how ”=” compares complex structures is provided in HELP EQUAL.)

2.20 Predicates and recognizer procedures

Any procedure whose result is always a boolean is called a PREDICATE. Besides ”=” there are several other predicates which compare two numbers and produce a boolean result, e.g. ”>”, ”<”.

```
vars x = 22;
x > 10  =>
** <true>
```

Some predicates take a single argument, for instance RECOGNIZER predicates, like isinteger, isword, isprocedure, islist, and several more.

```
isinteger(99) =>
** <true>
isword(99) =>
** <false>
isword("isinteger") =>
** <true>
isprocedure(isinteger) =>
** <true>
isprocedure("isinteger") =>
** <false>
```

In Pop-11 there is nothing to stop a procedure being applied to itself:

```
isprocedure(isprocedure) =>
** <true>
```

We have so far said that a condition is an expression whose result is a boolean. To be more precise, Pop-11 is defined to treat anything other than false as if it were true, between ”if” and ”then” (and in similar contexts explained later), so a condition may produce a result other than true. This is sometimes convenient, when a procedure is given the task of finding or creating something, in which it may not always succeed. Then the result false can be used to indicate failure and any other result will be the object found, which may need to be used by other procedures. Thus a program could include code like the following:

```
try_create_solution(....) -> found_thing;   ;;; may be false
if found_thing then
    do_something_with(found_thing)
else
    try_alternative(....)
endif
```

In Pop-11 this can be abbreviated, using the duplicating assignment operator ”->>”, as follows:

```
if try_create_solution(....) ->> found_thing then
    do_something_with(found_thing)
else
    try_alternative(....)
endif
```

Similarly the system predicate lmember returns either false, or the tail of a list containing the item found:

```
vars colours = [red orange yellow green blue indigo violet];
lmember("pink", colours) =>
** <false>
lmember("blue", colours) =>
** [blue indigo violet]
```

The fact that some predicates return a non-boolean result instead of true is used by the Pop-11 pattern matcher, described later.

2.21 Summary of syntactic roles in Pop-11 programs

Portions of Pop-11 programs described so far may be classified as follows – the diagram is incomplete, and may be expanded by looking back over this chapter:

```
                            imperatives
                                |
                ______________________________________________
                |                    |             |         etc...
            expressions           assignments   declarations
                |                               |          |
    __________________________          variable declarations  |
    |       |      |  |    |                               |
numerals    |      |  |  conditions             procedure definitions
       quoted words|  |
                   |  |
              strings |
                    list expressions
```

2.22 Lexical analysis and the Pop-11 itemiser

A program file read in by Pop-11 or a command typed in is basically a stream of characters. Somehow the Pop-11 system has to break this input stream into meaningful components which it can recognize and then translate into machine code instructions of various sorts. This happens in two stages.

The first stage is called 'lexical analysis' or 'tokenising' or 'itemising'. This breaks the stream of characters into separate 'text' items that form the basic building blocks of programs. These building blocks are

o words, including syntax words, procedure names, user variables, etc.

o strings, which are delimted by the string quote symbol """

o numbers, including integers, decimals, ratios and complex numbers.

The rules for breaking text up into these text items, are explained fully in REF ITEMISE. Only a subset will be explained here.

The second stage of analysis is even more complicated and involves grouping these text items into recognizable syntactic forms, such as procedure calls, assignments, loops, conditionals, declarations, definitions of procedures, and so on. This is usually called 'parsing', though Pop-11 does not create a parse tree.

There is a third stage which involves translating these recognizable forms into machine code instructions. This is sometimes called code-planting.

All the above occur at compile time. Later on, at run time, the compiled instructions can be executed.

The following sections will describe the first stage of compilation, i.e. the processes involving the itemiser.

2.23 Itemisation rules in Pop-11

We have so far assumed that we can treat Pop-11 programs as made of numbers and words which can be combined to form expressions, or imperatives, or sequences thereof. But what is actually typed in, or read in from a file is a sequence of characters. For instance the following is a sequence of five characters, which has to be broken up into four items, the number 3, the word "+" the number 55 and the word "=>" :

```
3+55=>
```

The Pop-11 'itemiser' applies quite complex rules to decide how to divide up the stream of characters into meaningful chunks. For instance, if you type:

```
[a little list,and,6*5]
```

this is read as 11 items:

```
[  a  little  list  ,  and  ,  6  *  5  ]
```

and in fact they will be interpreted as an instruction to build a list containing nine items: seven words and two numbers. (Non alphabetic characters can also be used to form words, e.g. "+++", "##@##".)

To do this Pop-11 needs 'lexical' rules saying which sorts of characters can be joined up with which, since you do not have to use spaces to separate things. Besides things like spaces, tabs and newlines, which are normally ignored by Pop-11, there are the following types of characters:

```
Numeric:         0 1 2 3 4 5 6 7 8 9

Alphabetic:      a b c d e f g  ...  z
                 A B C D E F G  ...  Z

Signs:           ! # $ & + - : < = > ? @ \ ^ | ~ / *

Underscore:      _

Separators:      ; " % ( ) , .  [  ]  {  }

String quote:    '

Character quote: `
```

2.23.1 Word formation in Pop-11

Unfortunately, Pop-11 has fairly complex rules for grouping characters in the text input stream into words, although words created by programs can contain arbitrary characters.

During program compilation, a letter followed by a series of letters and numbers will be formed into a single word, e.g. list1, list2. But if a text item starts with a number, then as soon as a non-number is reached (e.g. the "l" in "1list") Pop-11 assumes that it should insert a break. I.e. the text is separated into a number followed by a word. This can be shown by typing in the following instructions to create and print out lists:

```
[list3] =>
** [list3]

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

The second list is taken to have two elements, a number and a word. The first has a single word "list3".

A numeric character may be buried in the middle of a word which starts with letters, e.g. "list3a". Thus a word that starts with a letter can be followed by any combination of numbers and letters.

The word quote symbol """ can be used to tell Pop-11 that you wish to refer to a word, instead of using it as the name of something else (i.e. as variable):

```
"list3" =>
** list3
```

But if you give it an illegal combination of characters you will get an error:

```
"3list" =>
;;; MISHAP: IQW INCORRECT QUOTED WORD
;;; INVOLVING: 3
```

You can also make a word out of certain non-alpha-numeric characters, i.e. sign characters:

```
"*+*+*::\/^" =>
** *+*+*::\/^
```

But you cannot mix letters and sign characters:

```
"+x" =>
;;; MISHAP ...
```

and in a list they will be separated into two:

```
[+x] =>
** [+ x]
```

However, the underscore character can be used to join alphanumeric type characters to sign characters, e.g. here are two lists each containing only one word:

```
[+_x] =>
** [+_x]

[apple_#@$=>] =>
** [apple_#@$=>]
```

The underscore can also be used as a convenient way of producing long names which are readable. E.g. the following is the name of a system variable:

```
pop_readline_prompt
```

Using the underscore to join letters and sign characters

In general a sequence of characters made of "sign" characters and letters will be broken at the point where the two sorts of characters meet, unless they are joined by an underscore symbol "_", e.g.

```
fast_++          ++_lists_++
```

Two of the sign characters '/' and '*' play a special role in that they can be combined to form the 'comment brackets', explained aboved. So "/*" and "*/" cannot be used as ordinary Pop-11 words.

The separator characters cannot be used to join up with anything else, except for the use of '.' in decimal numbers (e.g. 66.35). This is because separators play a special role in the syntax of Pop-11. E.g. the following is a list of seven items

```
[(.,)a"!] =>
** [( . , ) a " !]
```

The semicolon, though normally a separator which marks the end of an imperative has a special role if repeated three times without anything between: it marks an 'end of line' comment as explained previously. E.g.

```
6 * 6 =>          ;;; this bit on the right is ignored!
** 36
```

Some of the characters have special roles which will not be explained fully till later. In particular '%' can be used both in creating procedure closures by 'partial application' and in 'unquoting' part of a list expression. (The file TEACH PERCENT gives a tutorial introduction to both.)

Strings can contain arbitrary characters

Strings, created using the string quote character can contain arbitrary characters:

```
'this is a *+*+*+* string %&$%$ of rubbish!!!'
```

except that if you wish to include the string quote itself in the string it must be preceded by the backslash character \ to indicate that it does not mark the end of the string. Here is a string containing the string quote:

```
'isn\'t it' =>
** isn't it
```

Note that strings are normally printed without the outer quotes. To make the quotes appear, do

```
true -> pop_pr_quotes;
```

2.23.2 Character quotes and string quotes

Characters themselves are represented by positive integers less than 256. Since it is difficult to remember which number represents which character (the so called 'ASCII code'), the character quote can be used to tell Pop-11 to read a character as representing the number. The character quote, sometimes referred to as the "backquote" is the backward sloping single quote character. It should not not be confused with the forward sloping (sometimes displayed as vertical) single quote character used to begin and end string expressions. Depending on the printer used the string and character quotes in this document may have different appearances.

```
Here is the character quote symbol:      `
Here is the string quite symbol:         '
```

Unfortunately neither symbol has a predictable location on keyboards: they appear in different places on different keyboards.

Here are some examples using the character quote to represent characters (as integers) without remembering their integer values. The letter 'A' has the code 65 and the numerals start from 48:

```
`A` =>
** 65
`B` =>
** 66
`a` =>          ;;; lower case codes are different
** 97
`0` =>
** 48
`5` =>
** 53
```

If you wish to include non-printing characters in a string, see the details in HELP ASCII. In particular you can use the following

```
\s  = a space
\t  = a tab
\n  = a newline
\r  = the return character (ascii 13)

'\nA string\n\twith text\n\t\s\son three lines' =>
**
A string
    with text
      on three lines
```

2.23.3 Double quotes with single quotes can form arbitrary words

If you really need to have a word containing arbitrary characters you can create it by putting word quotes around the corresponding string.

For example

```
vars funny_word = "'A word with spaces and junk:*&*=%][()))'";

isword(funny_word) =>
** <true>

funny_word =>
** A word with spaces and junk:*&*=%][()))
```

However, you would not be able to use such a word as the name of a variable, since typing something like

```
vars 'A word with spaces and junk:*&*=%][()))' = 999;
```

will produce an error.

```
;;; MISHAP - vars STATEMENT: IDENTIFIER NAME EXPECTED
;;; INVOLVING:  'A word with spaces and junk:*&*=%][()))'
```

2.23.4 Changing Pop-11's "itemiser" rules

To complicate matters further, it is possible to tell Pop-11 that you wish to alter its rules, by using the procedure item_chartype. This is especially useful when defining a new language in terms of Pop-11. Details will not be given in this introduction. The on-line documentation file REF ITEMISE gives more information, as does HELP ITEM_CHARTYPE

2.24 Revision questions

Here are some revision questions, answered earlier in this chapter.

1. What is a programming language?

2. What is declarative programming?

3. What are expressions?

4. What are imperatives?

5. What are declarations? What are initalised declarations?

6. Give examples of some of the sorts of objects Pop-11 expressions can refer to.

7. Give examples of some of the sorts of imperatives one can construct in Pop-11 and explain what they do.

8. Give an example of a Pop-11 procedure, and explain how it is used and what it does.

9. Define a procedure which takes in three numbers, adds up the first two, and then multiplies the result by the third. Your definition could start:

```
define addmult(x, y, z) -> result;
```

10. What is the difference between defining a procedure and calling, running, or invoking it?

11. Explain with examples what it means to say that a procedure takes a certain number of arguments, and does or does not produce a result.

12. What is the difference between printing something out and producing a result?

13. What is the difference between '=>' and '==>' ?

14. What is a comment, and how are comments expressed in Pop-11?

15. What will the following print:

```
pr(99);spr(100);pr(101);spr(102);spr(103);npr(66);pr(77);
```

16. What are predicates and how are they used in conditions?

17. Which of the following words are legal in Pop-11?

```
"cat5"  "5cat" "**+**"  "*cat*"  "_5cat"  "*_cat"
```

You can test each case by typing it in followed by "=>" to see if you get an error.

18. What are compile time processes and run time processes?

2.25 Built in Pop-11 data types

We have seen that a programming language allows you to construct expressions which denote objects, and we have seen some examples of Pop-11 expressions. Programming languages differ in the range of types of objects they can refer to. Often there is a fixed set of "data types" built in to the language, e.g. numbers, words and lists.

Some languages, including Pop-11, allow you to extend the set of data types indefinitely. In Pop-11 this requires using the procedure CONSKEY, or the DEFCLASS declaration (which is the basis for the RECORDCLASS and VECTORCLASS declarations), which will be explained in a later section. The OBJECTCLASS facility, based on these, allows still further extension by permitting the user to define overlapping hierarchies of types of objects with associated families of "methods" for operating on them. (This is probably the most important feature of what is known as Object Oriented Programming.)

For now we shall concentrate only on the built-in data-types, which provide the basis of all other data-types. Most types of objects are associated with special data-structures called "keys". For instance, the key known as integer_key is associated with integers, and the key known as string_key is associated with strings. The key associated with a particular type of data is another data-structure of type "key" which contains information about the type, including how to print it, which procedures are provided for manipulating its contents (if it has accessible contents), and so on.

Some data-types do not have keys because they are built-up from more primitive data-types. For example there is not a list_key because lists are built up from chains of pairs, so that the pair_key suffices.

2.26 Procedures associated with data-types

Besides keys, each data type in Pop-11 has a collection of associated procedures, some for constructing new instances of that type (e.g. consstring, consvector, consword, conspair) some for accessing or updating components of structures (e.g. subscrs, subscrv, subscrw, front, back), and some for recognizing instances (e.g. isstring, isvector, isword, ispair).

2.26.1 Generic procedures

There are also some "generic" procedures that are applicable to a wide range of data-structures, including the following:

```
o the equality and identity test procedures "=" and "==",
o pr, the maihn printing procedure
o explode which puts all components of a structure onto the Pop-11
  stack,
o datalist, which makes a list of components,
o length and datalength which return the number of fields in
  a structure.
o appdata which applies a procedure to all the components,
o mapdata which takes a structure and a mapping procedure and
```

```
  produces a copy of the structure with the components mapped
o copy, which copies a structure at top level, occasionally with
  results that surprise users, e.g. when applied to pairs.
o copydata, which copies recursively
o fill, which transfers items from the stack into the fields of
  a datastructure.
o datakey, which returns the key associated with an object's type
o dataword, which returns the word that names the object's type
o allbutfirst, a procedure for creating a copy of a list or other
  structure minus the first N elements.
o allbutlast, a procedure for creating a copy of a list or other
  structure minus the last N elements.
o <> the concatenator procedure <>, which can join or "compose"
  lists, strings, words, vectors or even two procedures, to form
  a new more complex object of the same type as its inputs.
o >< the "string concatenator" which takes any two printable objects
  and produces a string which combines their printing forms.
```

A generic feature of records, vector type objects and lists in Pop-11 is that if an object O of one of these types is applied to a positive integer N that is smaller than the length of O, then O is treated as if it were a function and the result of O(N) is the Nth component of O. Similarly the form can be used to update the N'th component, as in:

```
-> O(N);
```

The behaviour of a structure applied to an object can be re-defined for user data types using the class_apply facility, to specify how an object should behave if treated as a procedure. (See HELP CLASSES, REF KEYS)

Other generic facilities are defined in the following online Poplog documentation files:

REF DATA, REF DEFSTRUCT, REF RECORDS, REF VECTORS, REF KEYS

Yet more generic procedures can be defined by the user via the Objectclass library described briefly in Chapter 8, below.

The file REF NUMBERS describes generic mathematical operations that can be applied to all types of numbers represented in Poplog, integers, bigintegers, rationals, decimals, ddecimals and various kinds of complex numbers. E.g. "+", "-", "*", "/", and "sqrt" are generic in this sense.

2.27 Data creation, memory management and the heap

Every time a program creates a new object some space must be allocated for it in working memory. Thereafter it is referred to by a pointer. The exceptions are small integers and single precision decimals which need no more space than a pointer, so they are simply copied to wherever they are needed. They are called 'simple' objects, whereas the objects that are referred to via pointers are referred to as 'compound' objects.

The Pop-11 memory manager (in Poplog it's the Poplog memory manager, shared by all the Poplog languages) maintains a region of storage in virtual memory space, in which compound objects are created, including large numbers, strings, words, lists, vectors, and even procedures. This area is known as the 'heap'.

Every now and again the attempt to create a new object is hindered by lack of space in the heap. At that point an automatic garbage collection program is invoked which works out which objects are no longer accessible by any portion of the current program. The garbage collector then rearranges the heap so that all the accessible objects are compacted together (and all pointers to them are correspondingly changed). This leaves additional space in the heap free for new objects to be created. If there is no free space even after a garbage collection, the storage manager will try to get more space from the operating system in order to enlarge the heap (the limit to such enlargement is set by the user-assignable variable popmemlim).

It is possible that eventually there is no more space in the machine, and attempts to create new structures are foiled and the process has to be abandoned. This can depend on what else is happening on the machine at the time. The Poplog garbage collector is unusually fast for an AI development environment, so most of the time users will not notice when garbage collection happens.

Note for experts

The above is an over simplification in that the Poplog system has to be able to cope not only with ordinary Poplog structures, but also data structures linked in via external programs, which might have been written in C or Fortran. Moreover some of the Poplog structures may be made accessible to external procedures. These complications require the heap to have a mixture of sub-regions with different characteristics. For example some regions are used by Poplog structures that cannot be relocated by the garbage collector because external procedures will then be confused. Also the heap may include 'holes' corresponding to regions of memory that are used by external procedures to create structures that Poplog (and Pop-11) cannot access. Readers who wish to know more should consult the online files REF EXTERNAL and REF EXTERNAL_DATA

2.28 List of Pop-11 Data-types

For beginners most of the types listed below are irrelevant. It is possible to achieve a great deal with only the following:

```
words, strings, integers, decimals booleans, lists, vectors,
procedures, and arrays
```

Beginners can therefore safely skim most of the rest of this chapter up to the section on Objectclass.

The procedure datakey can be applied to any object to return its key. The procedure dataword can be applied to any object to return its dataword. E.g.

```
dataword(999) =>
** integer
dataword(2**40) =>
** biginteger
datakey(66.66) =>
** <key ddecimal>
dataword("word") =>
** word
dataword(dataword) =>
** procedure
dataword(word_key) =>
** key
datakey(datakey) =>
** <key procedure>
datakey({1 2 3}) =>
** <key vector>
```

What follows is a list of the data-types available in Pop-11 in Poplog Version 15.01. For each type we give the name of the key (a special record that provides information about the data-type), the dataword, and the main REF file giving information about that type. (A list similar to this can be found in REF DATA, and may be more up to date, as sometimes new types are added to the system.)

Poplog built in data types

```
 KEY NAME                DATAWORD             MAIN REF FILE
XptDescriptor_key     "XptDescriptor"      REF XptDescriptor
biginteger_key        "biginteger"         REF NUMBERS
boolean_key           "boolean"            REF RECORDS/Booleans
complex_key           "complex"            REF NUMBERS
ddecimal_key          "ddecimal"           REF NUMBERS
decimal_key           "decimal"            REF NUMBERS
device_key            "device"             REF SYSIO
dstring_key           "dstring"            REF STRINGS
exptrvec_key          "exptrvec"           REF EXTERNAL_DATA
external_ptr_key      "external_ptr"       REF EXTERNAL_DATA
ident_key             "ident"              REF IDENT
integer_key           "integer"            REF NUMBERS
intvec_key            "intvec"             REF INTVEC
key_key               "key"                REF KEYS
nil_key               "nil"                REF LISTS
pair_key              "pair"               REF LISTS
procedure_key         "procedure"
        (procedures and closures)          REF PROCEDURE
            (properties)                   REF PROPS
            (arrays)                       REF ARRAYS
process_key           "process"            REF PROCESS
prologterm_key        "prologterm"         REF PROLOG
prologvar_key         "prologvar"          REF PROLOG
ratio_key             "ratio"              REF NUMBERS
ref_key               "ref"                REF RECORDS
section_key           "section"            REF SECTIONS
shortvec_key          "shortvec"           REF INTVEC
stackmark_key         "stackmark"          REF STACK
string_key            "string"             REF STRINGS
termin_key            "termin"             REF CHARIO
undef_key             "undef"              REF IDENT
vector_key            "vector"             REF VECTORS
word_key              "word"               REF WORDS
```

In several cases there are also HELP files giving more information, e.g. HELP UNDEF, HELP SECTIONS, HELP PROCESS

Future versions may include additional standard data-types, e.g. a matchvar data-type for pattern matchers to use. Also users can extend the list of types by defining some of their own.

2.29 Further information on built in data types

The following sections give more information on the data types listed above. Here they are grouped according to their purpose, rather than being presented alphabetically. Where there is special syntax for creating instances, this is illustrated. Some of the main procedures associated with the data-type are also listed. Most of the generic procedures described previously are not mentioned again. Online HELP or REF files giving further information are listed. In many cases further information is given in other parts of this primer.

2.29.1 Words

Words are structures corresponding to a sequence of characters. They can be used in one of three main sorts of roles:

o data, i.e. objects operated on by programs, e.g. in lists

o syntax words like "if", "define", "endwhile" which form portions of programs and define the program structure

o names of objects, i.e. variable or constant identifiers

They can occur implicitly in programs as part of the code, or they can be explicitly denoted by quoted word expressions like these:

```
"cat",  "dog",  "***",  "a",  "xxx_yyy",  "(",  "!+*+*+!",  """
"'a long mixed character word including spaces %$%$%3333!)(.;;'"
```

Expressions denoting words have strict formation rules sketched previously. Normally quoted words and program text words such as variable names cannot contain spaces, or mixtures of characters of different types. However there are procedures which can construct words with arbitrary combinations of characters, and arbitrary sequences of characters can be made into a quoted word by enclosing them in string quotes with surrounding word quotes (as in the last example above). This mechanism is not available for using variable names with spaces or illegal mixtures of characters.

Words can be thought of as structures that include a string and other information. In addition they are "standardised" in a dictionary, as described below, unlike strings.

The internal representation of words

A word is represented internally by a special record which includes information about the characters making up the word (which can be accessed by applying the procedure word_string to the word).

If the word is being used as a program identifier (as opposed to merely being a data item in a list, for example) then the word record includes a pointer to an ident record (described below) giving information about the syntactic properties and associated value. This pointer can be changed depending on what the current section is. If the word has a syntactic role then the following procedures can be applied to it to discover what the properties are:

```
identprops(word)
    Returns "undef" for word with no syntactic role. Otherwise
    returns the numeric precedence, or one of "macro", "syntax",
    "syntax N" as described in REF IDENT

full_identprops(word)
```

```
    Returns "undef" for an undeclared word, or a list of all the
    keywords used in the declaration of word, e.g. "global",
    "constant", "protected" etc.  See REF IDENT

identof(word)
    Returns or updates the global (permanent) identifier currently
    associated with the word (see below). This association will
    vary according to which section is 'current'.
```

The Pop-11 dictionary

Words play a crucial role in Pop-11. Most of the items in a program text stream are words, including syntax words like "if", "define", "lvars", and also user-defined variable or procedure names, like "list", "rooms", "x_axis", etc. In order to be able to tell quickly whether a word is one that is already known, Pop-11 keeps all words, including both system words and words introduced by the user, in a single global dictionary, which makes use of hash-coding on the characters of the word for rapid access. This makes it very easy to check whether a new sequence of characters corresponds to a currently known word. If so, the existing word record is used. If not, a new word record is created and entered in the dictionary.

That explains why two occurrences of a word expression for the same word will return the very same (i.e. identical word), unlike two occurrences of a string expression, as shown when the strict equality predicate "==" is used for comparison:

```
"cat" == "cat" =>
** <true>
'cat' == 'cat' =>
** <false>
```

The dictionary in Pop-11 corresponds roughly to the symbol table of a conventional programming language, like Pascal or C, except that in those languages the symbol table is used at compilation and link time but is not normally required during program execution, whereas the incremental compiler in Pop-11 requires the dictionary to be available at all times, not merely during a compilation phase prior to execution of programs. It is also required because running programs can create words, e.g. using consword or <>.

The dictionary is not a structure that is accessible to users. However the procedure appdic can be given a procedure which it will apply to every word in the dictionary. For example the following expression will create an alphabetically sorted list of all the words in the dictionary, typically a list of several thousand items:

```
sort( [% appdic(identfn) %] )
```

Additional procedures that operate on words include consword, destword, isword, subscrw, subword, word_string, sys_current_ident, and the concatenator <>

See REF WORDS and REF IDENT for further information.

2.29.2 Strings

Examples of string expressions:

```
'a',  '21385d73::;+*)(&%',  'string with spaces'
```

A string is a vector of characters (8-bit integers). Relevant procedures include inits, consstring, deststring, isstring, subscrs, ><, explode.

Non-printing characters can be represented in strings using special conventions analogous to those used in C string expressions, as explained previously e.g.

```
\s  = a space
\t  = a tab
\n  = a newline
\r  = the return character (ascii 13)
```

A string expression like 'cat' causes a corresponding new string to be created by the lexical analyser each time such an expression is read in while programs are being compiled. Thus a list like the following will contain two strings with similar contents:

```
['cat' 'cat']
```

whereas the following list will contain contain two pointers to the very same word record, namely "cat":

```
[cat cat]
```

Moreover, as shown above, the strict equality test on two strings created at different times will return false, because the strings will be different items in the machine's memory.

In Pop-11 there is no table containing all strings, like the dictionary containing all the words.

If a string expression occurs inside a procedure, e.g. in an assignment like this

```
'The string' -> string;
```

then the string will be created only once, at compile time, rather than a new string being created each time the procedure is run. This is unlike list expressions and vector expressions which actually plant instructions to create a new instance.

Strings are useful for storing sets of small positive integers (i.e. requiring no more than 8 bits per integer). They are also often used to create text for printing. The string concatenator operator "><" is often handy for creating a string that includes the characters that would normally be used for printing some other object that is not a string. For example if you concatenate the empty string with a number the result is a string that looks like the number. However, if pop_pr_quotes is true the concatenated string will include spurious string quote characters, which can be suppressed by using sys_>< instead of ><, thus:

```
false -> pop_pr_quotes;
vars numstring = 12345 >< '';
numstring =>
** 12345
true -> pop_pr_quotes;
numstring =>
vars newstring = 12345 >< '';
newstring =>
```

```
** '12345"'
;;; Use sys_>< to suppress the effects of pop_pr_quotes true
vars laststring = 12345 sys_>< ";
laststring =>
** '12345'
false -> pop_pr_quotes;
newstring =>
** 12345"
```

The empty string is often very useful, so the built in identifier nullstring is provided with an empty string as its value.

The Poplog editor VED represents each file as a vector of strings and dstrings (described below). Empty lines are represented by nullstring.

For more information see:

```
HELP STRINGS, HELP ASCII
```

2.29.3 Dstrings

Since version 14.2, the Poplog editor VED can use a more complex representation for characters in strings. These use a special datatype known as "dstrings" (or Display Strings) which support characters with different attributes, such as bold, italic, or underlining. Ultimately dstrings will be able to support multiple fonts.

These are described in REF STRINGS/Dstrings

The VED procedure ved_chat, for modifying CHaracter ATtributes in a VED file is described in REF VEDCOMMS.

2.29.4 Idents (identifier records)

When a word is used as a syntax word or a variable or constant identifier, a new structure is associated with it which defines its role in Pop-11. This structure is a special record, known as an ident (or identifier). The record contains the following fields:

o An idval field for holding the value of the identifier if there is one. The procedure valof, applied to a word, accesses this field of the corresponding ident. So does the any program code that accesses or updates the value of a variable.

o Type information: the identtype. E.g. an ident may be restricted to take only procedure values.

o A flag indicating whether the variable is "active" and if so what its multiplicity is.

o The identprops, which determines the syntactic properties used by the Pop-11 compiler when program text is being compiled.

o A flag indicating whether the identifier is lexical or permanent

At present the ident does not specify the word that identifies it. So more than one word can share the same identifier, i.e. they can function as synonyms in programs.

The syntax word "ident" is available for accessing the ident currently associated with a word, thus:

```
vars list1 = [a b c];
ident list1 =>
** <ident [a b c]>

ident define =>
** <ident <procedure define>>
```

Note that the standard printing routine merely shows the idval field of the ident.

A word may be associated with different idents in different sections. This is why not all the information relevant to the role of a word is held in the word record itself.

Procedures concerned with idents include: consident, isident, idval, identprops, nonactive_idval, sys_current_ident, word_identifier.

The last procedure is used to create words that bypass the section mechanism so that they are guaranteed always to have the same identifier associated with them.

2.29.5 Types of numbers in Pop-11

There are several sorts of numbers in Pop-11: integers, big integers, decimals (called floats in some languages), ddecimals (sometimes called long floats), ratios, and complex numbers. The formats for typing in numbers are fully specified in REF IDENT, and the internal representation of numbers of different sorts and procedures available for operating on them are described in REF NUMBERS. What follows provides a summary of this information.

2.29.6 Integers and Bigintegers

Integers are 'simple' Pop-11 objects not represented by pointers. They occupy one machine word, typically a 32 bit word, though 2 bits are normally used for type identification in Pop-11 and therefore only 30 bits are left for the integer value. Examples are:

```
66,  -33,  99999,    -12348888, 0
```

Bigintegers are structures represented in the heap and can be arbitrarily large. Some examples are:

```
12345678900980980911,   2**40,   -9999999999999999999999
```

The only limit to the size of a biginteger is the (virtual) memory available in the machine.

2.29.7 Floating point numbers: decimals and ddecimals

In Pop-11 there are two kinds of floating point numbers, decimals and ddecimals, which differ only in their degree of precision. Normally the results of floating point calculations in Pop-11 are decimals (single precision floats). However, by making the value of the system variable popdprecision true, instead of false, which is its default, the relevant procedures are changed to return ddecimals. This increases the precision of floating point computations, but can use a lot of temporary storage space in the heap, causing garbage collections to occur (or extra paging on a machine with a small amount of physical memory.)

Decimals, like integers, are 'simple' Pop-11 objects not represented by pointers. Each occupies a single machine word, apart from the two bits needed for type information. Thus decimals in Pop-11 are typically restricted to 30 bit precision. Examples are the following.

```
66.0,  -33.0,  77.35,  9999.532,  -6666.0,  0.0s0,   5.5s-5
```

The last example represents

```
5.5 * (10 ** -5), i.e.  0.000055
```

Strictly only the last two examples will create a single precision decimal if typed in to Pop-11, since normally floating point constants are read in as ddecimals irrespective of the value of popdprecision. The last two examples use ‘s’ to force creation of a single precision float.

Examples of expressions denoting ddecimals are

```
0.0,  -9999.5, 12345.678,  0.00000001,  1.5d5,   1.5d-5,   1.5e5
```

In the last three examples the letters ”d” and ”e” are used interchangeably to indicate the exponent, in contrast with the ”s” of single precision floats. For example

```
123.45d5   is the double precision number  12345000.0
123.45d-5  is the double precision number  0.0012345
```

The last example may print as ‘0.001234’ because the global variable pop_pr_places, which controls the number of decimal places printed defaults to 6, as shown here:

```
0.123456789 =>
** 0.123457
```

As explained above, decimals and integers use single precision arithmetic, and are represented entirely within a single word of memory, usually using 30 bits, as the remaining two bits are required to distinguish pointers, integers and decimals. By contrast ddecimals use double precision arithmetic, and will be created when the value of the global Pop-11 variable popdprecision is non false.

All ddecimals take the same amount of memory space (which depends on the current implementation, but is typically three 32 bit words, one word being used to point to the ddecimal key, and the other two to hold the number.) Thus ddecimals have limited precision, though it is greater than the precision of decimals. Bigintegers are more complex structures that are unlimited in size and therefore unlimited in precision. The same applies to ratios.

2.29.8 Ratios use indefinite precision

Ratios represent the ratio of two integers or two bigintegers, and can be used for very high precision arithmetic. Examples of ways of representing ratios are

```
3_/4,  12345_/54321,  -33_/44
```

In this form a ratio expression will be read as one item, as can be shown by enclosing them in list brackets and printing out the list:

```
[3_/4  12345_/54321  -33_/44] =>
** [3_/4 4115_/18107 -3_/4]
```

Unlike this

```
[3/4  12345/54321  -33/44] =>
** [3 / 4 12345 / 54321 -33 / 44]
```

The division of two integers will normally produce either an integer, or a biginteger, or a ratio, in the case where the division is not exact. E.g.

```
10/3 =>
** 10_/3
```

Programs that involve inexact division of integers will produce ratios, and the computations will be exact, without the loss of precision involved in the use of decimals and ddecimals. However, high precision ratios, like bigintegers, can take up a lot of space, and if many of them are created, they will require temporary storage space, and the garbage collector may be invoked more often than expected. Fortunately the Poplog garbage collector is very fast. Also the frequency of garbage collections can be reduced by using the variables popmemlim and popminmemlim to expand the heap space so that space runs out less often.

The format shown above can make it difficult to take in differences between different ratios. It is possible to get Pop-11 to print out ratios as if they were floating point numbers, which is sometimes more convenient, though potentially misleading. This is done by making the global variable pop_pr_ratios false.

```
false -> pop_pr_ratios;
10/3 =>
** 3.333333
true -> pop_pr_ratios;
10/3 =>
** 10_/3
```

2.29.9 Complex numbers

Complex numbers are represented in Pop-11 by records that hold the real part and the imaginary part. The real and imaginary parts can be integers, ratios, decimals or ddecimals, though both must be of the same type. These records can be created using the two infix operators +: and -:, where users are invited to think of the colon as an approximate depiction of "i", the square root of -1. Thus, for example:

```
0 +: 1 =>
** 0_+:1

3 -: 5 =>
** 3_-:5
```

Complexes are printed out in a form in which they can be typed in as single items, using expressions that start with the real part, followed by an underscore "_", followed by "+:" or "-:" followed by the imaginary part. For example the following is a list of two complex numbers (notice how the integer values are coerced to floats where necessary)

```
[ 3_+:2.0  -3_-:4 ] =>
** [3.0_+:2.0 -3_-:4]
```

There are many built in mathematical functions that are capable of taking complex numbers as arguments and/or returning them as results. E.g. attempting to compute the square root of -1, or the logarithm of a negative number produces a complex result:

```
sqrt(-1) =>
** 0.0_+:1.0

log(-22.5) =>
** 3.113515_+:3.141593
```

Procedures that are specifically concerned with complex numbers, include the two operators mentioned above and, conjugate, destcomplex, realpart, imagpart, iscomplex.

For more information see REF NUMBERS

2.29.10 Recognizers for number types: integral, rational, decimal, complex

A number is described as "integral" if it is an integer or a biginteger. It is described as "rational" if it is integral or a ratio. It is described as decimal if it is a decimal or a ddecimal. There are various recognizer procedures for detecting the different number classes:

```
isinteger, isbiginteger, isintegral, isratio, isrational,
isdecimal, issdecimal, isddecimal, isreal, iscomplex, isnumber
```

2.29.11 Reading in numbers relative to a base

Numbers may be represented externally relative to a base, though the internal representation is not changed thereby. The base is indicated by an integer followed by a colon, preceding the number itself. Thus, binary numbers are represented with the prefix '2:'. So:

```
2:100 is the same as 4
2:1011 is the same as 11
8:101 is the same as 65
2:1.1 is the same as 1.5
8:1.1 is the same as 1.125
16:1FFA represents 8186 as a hexadecimal number.
```

Note that the prefix '10:' is redundant. 10:999 = 999

For more on notations for numbers see REF ITEMISE, or Chapter 5, below.

Additional information about the representation of numbers inside the machine is also given in chapter 5, below.

2.29.12 Characters (8 bit integers)

A character in Pop-11 is represented as a positive 8 bit integer (i.e. an integer between 0 and 255) according to the standard ascii conventions, (except in dstrings where characters have more information corresponding to font characteristics.) The printing characters correspond to the integers between 33 (the exclamation mark character) and 126 (the tilde character). Users of Pop-11 do not need to remember the mapping from integers to characters, since character quote symbols can be used to represent the integers corresponding to a character. Examples are the following, where the last two examples correspond to the backslash character and the character quote character. (The spacing in the printed result has been stretched to show the correspondence)

```
   `!`, `a`, `B`, `0`, `9`, `(`, `\s`, `\t`, `\n`, `\\`, `\" =>
** 33   97   66   48    57  40   32    9      10    92   96
```

The file HELP ASCII gives full details on character codes, including how to represent non-printing characters, such as control characters.

NOTE: ‘a‘ is a character, whereas ‘a’ is a string.

Characters are not really a distinct Pop-11 datatype, as they are (at present) simply 8 bit integers. So they have no key or dataword of their own.

2.29.13 Booleans (true and false)

In Lisp the empty list is treated as false and everything else as true. In C and many other languages the number 0 is treated as false and everything else as true. The original version of Pop2 followed the latter convention, but when Pop-11 was designed it was decided that the introduction of a boolean data type was desirable as too many obscure bugs could follow from treating the empty list or 0 as false.

Two built in identifiers are provided to refer to the two boolean values:

```
true, false =>
** <true> <false>
```

Many other expressions are capable of denoting boolean values, e.g. the result of applying a recognizer procedure to an arbitrary object, or the result of applying an equality test to two objects or an arithmetical comparison to two numbers. Here are several examples of boolean-valued expressions:

```
true,  false,   66 == 66,  66 == 99,  77 < 33, "cat" = "dog",
isinteger(true), isboolean(99), isword("cat"), isstring('cat'),
member(3, [a b c d])
```

In addition to recognizers and comparison predicates there are a few operators designed specifically to operate on boolean values, namely:

```
not, and, or,
```

These are used for forming complex conditions for conditional and looping instructions. All of these treat any non-false object as if it were true, and ”and” and ”or” return false or their last non-false argument. E.g.

```
not(99) =>
** <false>
```

```
not(false) =>
** <true>
true and false =>
** <false>
true and "cat" =>
** cat
false or 99 =>
** 99
99 or false =>
** 99
false or not(true) =>
** <false>
```

strictly ”and” and ”or” are not infix procedure names, but syntax words as they prevent their second argument being evaluated if the first argument suffices to determine a value. This can mean that the first argument can be used as a ”guard” against an error in the second, e.g.

```
false and ("cat" + "dog") =>
** <false>
```

Very many procedures return booleans as results, for use in conditional expressions and loop test expressions.

2.29.14 Pairs and lists

Pairs are records containing two fields, which can contain any type of Pop-11 item. They can be created using the procedure conspair, and have associated procedures destpair, front, back, ispair. In Pop-11 as in several other languages, pairs are used as the basis for a ‘derived’ data-type namely lists. Lists are defined recursively as follows.

o The empty list [] (defined below) is a list.

o A pair is a list if its back is a list.

Lists are described in far more detail in Chapter 6, below. Apart from the syntax for creating lists, there is no special syntax for creating pairs, though they can be created using the procedure conspair:

```
conspair(3, 4),      conspair([], "cat"),   conspair("cat", [])
```

Only the last of these is a list, since its back is a list, namely the empty list. There is special syntax for creating lists, introduced in Chapter 1. Since these lists are made out of pairs, the syntax for creating lists also creates pairs. For example the list

```
[cat dog 99]
```

could be created using the expression

```
conspair("cat", conspair("dog", conspair(99, [])))
```

Many examples of lists are given in this introduction. Lists are strictly speaking a ‘derived’ data-type in Pop-11, in that they are constructed out of pairs, and therefore do not have their own key or dataword.

Most of the time users do not need to think about pairs. The facilities for building and manipulating lists, described later, are designed to hide such irrelevant details! See REF LISTS and Chapter 6. An overview of Poplog documentation relating to lists is given in HELP LISTS.

Later sections of this primer give a lot more information about lists, including the use of the pattern matcher and the Pop-11 database facility.

Procedures for operating on pairs include conspair, destpair, ispair, front, back. There is a much wider variety of procedures for operating on lists built out of pairs. See Chapter 6, below.

2.29.15 References (single component records, consref, cont).

A reference is a record with a single field that can contain an arbitrary Pop-11 object. There is no special syntax for creating references. They can be created using consref. E.g.

```
consref(0), consref("cat") =>
** <ref 0> <ref cat>
```

The contents of a reference created by consref can be accessed or updated using the field_accessor procedure cont. E.g.

```
;;; create a reference record rec, containing the number 10
vars rec = consref(10);
rec =>
** <ref 10>
;;; increment the number by 1
cont(rec) + 1 -> cont(rec);
rec =>
** <ref 11>
```

The recognizer is isref. References are often used to share variable information between different processes. For example, a procedure P can create a reference R which it gives to procedure Q. Q may eventually cause the contents of R to change. Then when control returns to P it can examine the contents of R to find out what has happened. This is sometimes more convenient than passing values via the stack (e.g. in programs using parallel co-routines) and safer than using global variables.

The procedure datalength applied to any pair will always return the same result:

```
datalength( conspair(3, 4) ) =>
** 2
datalength( [a b c d e] ) =>
** 2
```

In the second case it does not chain down the elements of the list. The procedures length and listlength do that.

```
datalength( conspair(3, 4) ) =>
length( [a b c d e] ) =>
** 5
```

However, length applied to a pair attempts to treat it as list, and this will cause an error:

```
length( conspair(3, 4) ) =>
;;; MISHAP - LIST NEEDED
;;; INVOLVING:  [3|4]
;;; DOING    :  null listlength length ...
```

2.29.16 Types of vectors: strings, full vectors, intvecs, shortvecs

There are several types of vectors in Pop-11. Each vector class has a family of associated procedures, including the following:

o initiator procedure, e.g. initv, inits, which takes an integer N and creates a vector with N fields, containing a default value, usually either undef for full vectors or 0, or 0.0 for others.

o constructor procedure, e.g. consvector, consstring, which takes N items and an integer N and creates a vector containing the N items

o subscriptor procedure, e.g. subscrv, subscrs, which takes an integer N and a vector, with at least N fields, and returns or updates the contents of the N'th field

o recognizer, e.g. isvector, isstring

o destructor procedure, e.g. destvector, deststring, which takes a vector and returns all its items on the stack, plus an integer N specifying the number of items. (Thus the results of destvector can be given to consvector to create a copy of the original.)

In addition various generic procedures mentioned previously can be used on all classes of vectors including datalength, appdata, mapdata, datalist, copy, copydata, and the concatenator <>.

The equality tester "=" is defined for all vector classes as follows: V1 = V2 is true if and only if V1 are of the same vector class (have the same datakey) and have the same length, and if corresponding components or V1 and V2 are themselves = to one another.

Users can define their own vector classes, though several standard vector classes are built in to Pop-11, including strings described above. Strings are byte vectors containing packed 8 bit integers, and have associated procedures inits, consstring, subscrs, isstring, etc.

Standard full vectors

One of the standard data types is the class of standard full vectors. Instances of this class can contain any number of items, including no items. The contents of the fields in a vector can be arbitrary Pop-11 items: i.e. they are 'full' fields.

There is special syntax for creating vectors. Examples are:

```
{},  {a b c},    {cat mouse 3 4},    {'a string' in a vector}
{[a vector] {containing some} [lists {and} vectors]}
```

Vectors are very like lists, but stored more compactly in the computer. There are several different types of vectors, besides standard full vectors. E.g. strings are "byte" vectors.

Procedures for operating on standard vectors include initv, consvector, destvector, subscrv, isvector, and the concatenator, <>, and others described in REF VECTORS.

The procedure datalength can be used to check the length of a vector.

```
datalength({}) =>
** 0
datalength({{a b} {c d} {e f}}) =>
** 3
```

Packed integer vectors: intvecs and shortvecs

An intvec is a signed packed-integer vectors (usually with 32 bit fields). So the fields of these vectors have two more bits than standard Pop-11 integers which can use only 30 bits. Intvecs have no special syntax, but can be constructed using consintvec, or initintvec. Examples are:

```
initintvec(5) =>
** <intvec 0 0 0 0 0>
consintvec(1, 2, 3, 4, 5, 6, 6) =>
** <intvec 1 2 3 4 5 6>
```

The associated procedures are: consintvec, destintvec, initintvec, isintvec, subscrintvec

A shortvec is a signed short packed-integer vectors (usually with 16 bit fields). They have no special syntax. Associated procedures are consshortvec, destshortvec, initshortvec, isshortvec, subscrshortvec

For more details see the online files: REF INTVEC, REF SHORTVEC

2.29.17 Procedures, closures, arrays, properties

A great deal has already been said about procedures, and more information will be given below. In particular, we have previously illustrated the special syntax for creating named and anonymous procedures using the formats:

```
define       ....    enddefine;
procedure    ....    endprocedure
```

Procedures are sets of instructions, which tell the computer to do something. Some are built in to the Pop-11 system. Others are added by the user. Unlike some languages, Pop-11 treats procedures as objects, just like numbers or words. E.g. procedures can be created by running procedures, can be stored in lists, or assigned to variables. There are several procedures associated with procedures

o isprocedure: recognizes procedures

o pdprops: accesses or updates the pdprops field of a procedure, which normally contains the name and possibly other information.

o updater: accesses or updates the updater of a procedure.

For example the following two expressions for updating the hd of a list are equivalent: the first is 'syntactic sugar' for the second.

```
"cat" -> hd(list);
updater(hd)("cat", list);
```

The second applies the updater of hd to "cat" and list. The first does the same, though it is easier to read.

There are three more special kinds of procedures. They are all functionally equivalent to procedures in that they all have procedure_key as their datakey, they all use procedure-calling syntax for accessing or updating their contents they can all be partially applied to form closures (defined below), and various other procedure specific procedures can be applied to them, including pdprops and updater.

closures

A closure is a combination of a procedure and some data for it to operate on. A closure may either be created using "partial application" (see HELP PARTAPPLY, HELP CLOSURES), or may be a "lexical closure" created by a procedure containing local lvars variables. See Chapter 4 below, and HELP LVARS, REF VMCODE

There is special syntax for creating closures using partial application. E.g. gensym is a procedure that can be applied to a word, to produce new words with appended numerals, e.g.

```
gensym("cat") =>
** cat1
gensym("cat") =>
** cat2
```

If we wished to partially apply the procedure gensym to the word "cat" to create a new procedure of no arguments that could be run to get the effect of applying gensym to "cat" we could do so as follows:

```
vars cat_gen = gensym(%"cat"%);
cat_gen() =>
** cat3
cat_gen() =>
** cat4
```

The datakey of a closure is the same as the datakey of any other procedure, i.e. procedure_key:

```
datakey(cat_gen) =>
** <key procedure>
```

arrays

Arrays are N dimensional structures whose components are accessed by N integers, where N can be 0 or more. E.g. a two dimensional array might represent a picture, and its components could be accessed by giving two numbers representing distance along and distance up, e.g. picture(3,5). For every class of vectors there are corresponding types of arrays. Note that Pop-11 unlike many other languages, treats arrays as procedures, for maximum flexibility. For details see REF ARRAYS.

The main procedures for creating arrays are newarray and newanyarray, and additional procedures are, as follows:

array_subscrp, arrayvector, arrayvector_bounds, boundslist, isarray, isarray_by_row.

The global variable poparray_by_row has a boolean value which determines the order in which an array stores its values in the underlying vector.

Arrays are treated like procedures in that they are applied to the numbers representing their subscripts. Every array has an associated updater procedure for updating the contents of the array.

An example of the creation of an array from a vector class follows.

Using newanyarray to create an array from an intvec

Vectors of various types can be used to create arrays of various types using newanyarray. For example to create a 2-D array of intvecs with subscript values going from -5 to 5 and -10 to 10 do

```
vars
    intarray = newanyarray([-5 5 -10 10], initintvec, subscrintvec);
```

This array has 0 as default value at each location. The value at any location can be accessed by applying the array to two integers that are within the range of the bounds given. The array can also be updated, as follows, using two integers as indexes into the array.

```
intarray(1, 1) =>
** 0
99 -> intarray(1, 1);
intarray(1, 1) =>
** 99
```

Attempting to update the array with the wrong type of item will produce an error"

```
[99] -> intarray(1, 1);
;;; MISHAP - INTEGER -2147483648 TO 2147483647 NEEDED
;;; INVOLVING:  [99]
;;; DOING    :  subscrintvec ....
```

Arrays also have the procedure_key as their datakey.

```
datakey(intarray) == procedure_key =>
** <true>
```

The procedure newarray is used to create arrays from ordinary full vectors. Both newarray and newanyarray allow several optional forms in which they can be used. See HELP ARRAYS for an introduction.

properties

The last kind of procedure is a property. A property is a table of associations between objects, based on 'hash coding' to speed up access. A property can function as a sort of memory of what is associated with what. There are several different kinds of properties in Pop-11, and different procedures for creating them, described in REF PROPS

Properties, like closures, arrays, and ordinary procedures are applied to their arguments in order to access or update the associated value. Every property has an associated updater for changing the contents of the property.

The simplest kind of property is created using newproperty, and examples are given in the next chapter, and in HELP NEWPROPERTY.

More sophisticated types of properties can be created using the rather complex procedure newanyproperty, described in HELP NEWANYPROPERTY. A special subset of its functionality is provided by the procedure newmapping, whose use is illustrated in the next chapter. See also HELP NEWMAPPING

Procedures associated with properties include appproperty, clearproperty, clearproperty, property_default, property_size, datalength, appdata, copy and explode, as described in REF PROPS

'Destroy properties'

A special kind of property which (as far as I know) is unique to Pop-11, is called a 'destroy property'. A destroy property (described in REF PROPS) allows us to associate with an object a procedure to be run when the object becomes garbage. This is particularly useful when the object has associated with it some object outside the current Poplog process that needs to be removed if the object is garbage. A typical example might be a window on the screen corresponding to the object: the destroy action might be used to ensure that such windows are removed when they are no longer needed.

For more details on properties see REF PROPS

2.29.18 Declaring a variable to be of type procedure

When you know that the value of a variable is going to be a procedure and will not ever be anything else you can declare it as being of type procedure. Thus the declaration of cat_gen above could be replaced by this form:

```
vars procedure cat_gen = gensym(%"cat"%);
```

Declaring a variable as of type procedure can cause more efficient code containing it to be compiled, and will also cause extra error checking when anything is assigned to the variable. Several identifiers can be declared to be of type procedure if they are enclosed in parentheses following the word "procedure", e.g.

```
vars procedure (p1, p2, p3);
```

2.29.19 Lightweight processes

A process is a structure containing a combination of procedure and data and a record of how far the procedure has got in its execution. The original procedure may have invoked another procedure, which in turn may have invoked other procedures, and so on. Thus a process needs to include a partial procedure calling stack. It also needs to record the values of any local variables of those procedures.

Each process can be suspended and resumed as required. These are sometimes referred to as "lightweight" processes, because switching between these processes is far less time consuming than switching between operating system processes on a time-shared computer. Processes can be used for running simulations of various kinds, e.g. simulating an operating system.

Procedures for operating on processes include consproc, consproc_to, runproc, resume, suspend, kresume, ksuspend, saveproc, isprocess, isliveprocess,

These are described in HELP PROCESS, and in more detail in REF PROCESS

The timing facilities in Pop-11, such as sys_timer, make it possible for processes to be suspended and resumed at regular intervals. Thus a time-shared multiprocessing system can be implemented using Poplog Pop-11.

2.29.20 undefs

Undefs are a special type of record used to initialise newly declared global or dynamic local variables that have not been given an initial value.

Normally when a new variable is created, it is given a default value, which is a special object which may print out something like:

```
<undef xx>
```

Meeting one of these in an error message is usually an indication that you have forgotten to "initialise" a variable with an appropriate value. Note that there is a standard Pop-11 variable called "undef" whose value is the word "undef", and which is NOT an example of an undef data-type, but is a word. (For historical reasons, the word "undef" itself is still used in some contexts).

An example follows:

```
vars xxxx;  ;;; declare a new variable.
xxxx =>
** <undef xxxx>
hd(xxxx) =>
;;; MISHAP - LIST NEEDED
;;; INVOLVING:  <undef xxxx>
;;; DOING    :  hd compile .....
```

For more information see REF IDENT and HELP UNDEF

2.29.21 Keys

A key is a record containing information about a class of objects in Poplog. Each data type has associated with it a key which is a structure containing information about all objects of that type, such as their dataword, how they are to be printed, how many elements they are made of, how they are recognized, whether it is a vector class, a record class or some other kind, and if it is a record or vector class, what is accessing procedures are, etc.

Procedures associated with keys, and described in REF KEYS, include conskey, datakey, isvectorclass, isrecordclass, class_=, class_access, class_apply, class_attribute, class_cons, class_datasize, class_dataword, class_dest, class_fast_subscr, class_field_spec, class_hash, class_init, class_print, class_recognise, class_subscr

2.29.22 Unique objects: nil, termin, stackmark

The empty list []

This unique object is used to indicate the end of a chain of pairs making up a list. In Pop-11 it is NOT used to denote FALSE as it does in many lisp systems that lack a proper boolean data type. The special identifier [] can be used to represent the empty list. For compatibility with other languages the word "nil" is also defined as a Pop-11 identifier that represents the same object.

```
nil =>
** []
nil == [] =>
** <true>
```

The stream terminator, termin

Termin is a unique object used to indicate the end of a sequence of items, e.g. the end of a sequence of characters read from a file, or the end of a sequence of items making a stream, or a dynamic list. There is no special syntax for it, though termin is often produced by typing the end-of-file character to a program reading from the terminal.

It is also the last item produced by a character repeater obtained from a file. There is no special syntax to represent termin, though a constant identifier termin is provided, which has termin as its value.

```
termin =>
** <termin>
datakey(termin) =>
** <key termin>
```

When a dynamic list is produced by a generator procedure, the end of the list is indicated by the procedure producing termin as a result. Dynamic lists are discussed in Chapter 6.

See REF CHARIO

The stack mark, popstackmark

The unique item, which prints as <popstackmark> is used in connection with Pop-11's "open" stack when building lists or vectors. Users will normally only come across it when they make errors involving attempting to take too many items off the stack (stack underflow errors)!

Roughly, whenever an list building expression [...] or a vector building expression of the form { ... } the object popstackmark is placed on the stack when the construction starts, and the final object is created by removing all items on the stack down to the last occurrence of popstackmark and putting them in a list, or a vector.

See REF STACK

2.29.23 Devices

These are records associated with files, terminals and other means of communication between the Pop-11 system and the rest of the world. There are also pseudo-devices created using consdevice.

See REF SYSIO

2.29.24 External pointers

These are pointers to external data or external procedures (functions), created using another language, e.g. Fortran, C or Pascal and linked into the Pop-11 system. See REF * EXTERNAL_DATA

2.29.25 Sections

Sections are structures that hold information mapping words in the dictionary to the idents that provide information about how the words are currently being used. Because the mapping from word to ident can be different in different sections, sections enable different programmers to use the same words for different purposes without risk of clashing even when their programs are later combined.

Identifiers in one section can access those in another by using full section "pathnames", which are similar to Unix file path names except that "$-" is used instead if "/". Thus the identifier "bite" in subsection "dog" in in sub-section "mammal" in sub-section "alive" of the top level section could be referred to as

```
$-alive$-mammal$-dog$-bite
```

Code written in the section $-alive$-mammal$-dog would merely need to use ”bite”.

As implied by this example, sections can contain subsections. For full details see REF SECTIONS. An introduction can be found in HELP SECTIONS.

The active variable current_section holds as its value the current section.

Identifiers associated with sections include:

pop_default_section, pop_section, section_cancel section_export, section_import, section_key section_name, section_pathname, section_subsect section_supersect

2.29.26 Prolog variables and terms: prologvars, prologterms

A prologvar is a one-element record used as a variable for the Prolog subsystem of Poplog. Various utility procedures are available for manipulating them.

A prologterm is an instance of a special class of vectors used to implement terms in Prolog.

Procedures relevant to the prolog sub-mechanisms in Poplog include:

consprologterm, destprologterm, initprologterm, isprologterm, isprologvar, prolog_arg, prolog_args,

See REF PROLOG for more details.

2.29.27 Data types required for the Poplog X window interface

XptDescriptors are used for managing some of the rather complex interactions between Poplog and the X window system, which is mostly written in C. Additional Pop-11 datatypes are created in the X libraries in $usepop/pop/x/*

See especially $usepop/pop/x/pop/ref/*types*

Further details are provided in: REF XptDescriptor

The REF files mentioned above give far more information than beginners can possibly cope with, but experts designing sophisticated software will probably find them indispensable.

Additional built-in data types may be provided in later versions of Poplog, and will be listed in REF DATA

Most beginners will not need to know about most of the data-types mentioned above. For very many programs it suffices to know only about procedures, words, numbers and lists. Booleans are used in conditional instructions or in tests for termination of loops. Strings are useful for formatted printing, and they are also used as names of files. Arrays are useful for representing two dimensional image data. This Primer will be mainly concerned only with the most commonly used data types.

2.30 Objectclass - An object oriented extension to Pop-11

There is a system called Objectclass, which extends the Pop-11 data-type system with object oriented facilities similar to but more general than those in languages like C++. Objectclass has more in common with the Common Lisp Object System, CLOS.

See HELP OBJECTCLASS if you are interested. It is available as part of Poplog Pop-11 from Version 14.5, as is the GO (Graphical Object) libary, which is based on objectclass.

The objectclass library creates some additional Pop-11 record and vector classes (using the Pop-11 defclass construct, defined in REF DEFSTRUCT) and then defines objectclass classes and instances and other kinds of entities in terms of those new classes. It also extends the syntax of Pop-11 to make the new facilities convenient to use.

Another, older, object-oriented extension to Pop-11 is provided in the FLAVOURS library, described in TEACH FLAVOURS and REF FLAVOURS

More information about Objectclass is provided in a later chapter.

2.30.1 Further online information

As mentioned previously, each datatype has an associated family of procedures for creating and manipulating instances of the datatype. These are described in the REF files mentioned above. However there are also more "introductory" files in the Poplog HELP directory, and these can be found by browsing HELP INDEX, and HELP HELPFILES.

In particular the following may be useful

```
HELP EQUAL, HELP MATH, HELP STRINGS,
HELP LISTS, HELP ARRAYS,
HELP RECORDCLASS, HELP VECTORCLASS,
HELP NEWPROPERTY, HELP NEWMAPPING,
HELP CLASSES,
HELP PRINT, REF PRINT
```

2.31 Additional information needed to define Pop-11

Besides the information about data-types in the Pop-11 virtual machine, and procedures provided in the Pop-11 virtual machine for operating on those data-types, a full description of Pop-11 would specify how users can define their own data-types, and define their own procedures for operating on them. A brief introduction to some aspects of this can be found in REF DEFSTRUCT. This is probably unsuitable for novices and the files HELP RECORDCLASS, HELP VECTORCLASS are easier to understand. Examples are given later, in Chapter 8.

Further information required to complete the specification of the language is given in later chapters (and in Poplog's REF files).

In particular, it is necessary to specify how the system is started up, how your program's state can be saved so that you can continue on another occasion without having to repeat what you have already done, how Pop-11 can be made to interact

- with the terminal (including how to print out various kinds of datastructures),

- with disc files,

- with the screen and with other devices,

how programs can be timed or traced or delayed, how interaction with the operating system is provided, how Pop-11 can discover what sort of machine or what operating system the program is running on, and how it can interact with programs written in other languages.

Further information would specify how "error" states can occur, at compile time or run time, how they are detected, how they are reported, how user programs can vary the standard behaviour.

In addition to all this detailed information about the specifics of Pop-11 it is possible to spend several years learning how to make use of these tools in order to design a wide range of programs, achieving such software engineering goals as maintainability, extensibility, efficiency, reliability.

Fortunately, although many of the details are specific to Pop-11, most of the concepts and techniques are not, and will be applicable when using other programming languages. Indeed having seen how to do something in a very sophisticated and flexible language like Pop-11 a programmer may have a better idea how to implement a similar technique in another language that does not provide it directly.

The next chapter returns to some details of the Pop-11 virtual machine. Besides the datastructures that are constructed in the computer's memory when Pop-11 user programs run, there are several datastructures created and used by Pop-11 itself, and areas of the memory that are used to provide "workspace" for Pop-11. An example of a datastructure used by Pop-11 is a dynamic list known as "proglist", which is used by the Pop-11 compiler to represent the stream of characters in the program or command sequence currently being read in and compiled.

Anyone wishing to extend the syntax of Pop-11 needs to know how to manipulate the symbols on proglist. Some aspects of how proglist works are described later. Full details are given in REF PROGLIST and REF POPCOMPILE.

When the Pop-11 compiler, or any other language compiler in Poplog reads in programs it has to tell the Poplog virtual machine how to create new procedures and structures corresponding to the programs read in. Specifying how to do this includes specifying the set of basic Poplog virtual machine instructions available, and the procedures for compiling instructions to machine code. This is a very complicated topic, introduced (with some simplifications) in TEACH VM and described fully in REF VMCODE.

A particularly important workspace in the Poplog virtual machine is a portion of the computer memory which is referred to as 'the stack'. This is used as a general 'communication area' by Pop-11. We shall often refer to it as if it were an object, though strictly speaking it is not. It cannot be manipulated liked an object. For example, objects can be put on the stack, but the stack cannot be put on the stack or stored in other objects. The next chapter elaborates on this.

Other workspaces, which will not be described in this primer are the procedure calling stack, and the workspace used for implementing Prolog facilities, described in REF PROLOG.

2.32 The garbage collector

One feature that Pop-11 shares with many AI language systems, though it is lacking in Fortran, Pascal, C, Cobol and other widely used languages is the "garbage collector", or store manager. This is a very important feature of Pop-11 which enormously simplifies the use of programs that create large numbers of temporary structures then discard them. It is also very complicated because of the need to cope with externally loaded programs using languages like C that do not have garbage collections.

Some aspects of the store management are described in REF SYSTEM. Understanding how to avoid unnecessary garbage collection is one aspect of designing efficient programs. There are many hints on this and similar topics in the file HELP EFFICIENCY.

2.33 Exercises

1. Explain the difference between syntax and semantics of a programming language. How do internal semantics and external semantics differ?

2. What is the difference between a compile time error and a run time error?

3. What is a virtual machine?

4. What is the connection between a programming language and a virtual machine?

5. What needs to be described in order to specify a programming language fully?

6. What are the datatypes built in to Pop-11?

7. What is the difference between a word and a string, in Pop-11? (You will not be able to answer this fully at this stage.)

8. What is the difference between a word and an ident in Pop-11?

9. There are certain conventions restricting the combinations of characters that normally can be used to create words using the word quotes ”...” in Pop-11. How can those conventions be overcome to produce words containing arbitrary characters?

10. What is the difference (at first sight) between an expression and an imperative? Why does the distinction break down in Pop-11?

11. Pop-11 contains a garbage collector as described above. Can you think of reasons why it is sometimes necessary to write programs that create large numbers of temporary structures which are then discarded? Think of reasons why some intelligent processes require searching for solutions to problems or searching for coherent interpretations of information.

12. List some of the generic procedures, which are applicable to several data-types?

13. Besides ordinary procedures Pop-11 contains three other types of structure that are treated as procedures.

14. There is one type of object that has only two instances. What is it?

15. List the Pop-11 classes that have only one instance each.

3 CHAPTER.3: PROCEDURES AND THE STACK

A very important aspect of the Pop-11 ”virtual machine” is the stack. This is a part of the computer’s memory used as a communication area in connection with procedures which take ”arguments” (inputs) and return results, and also in connection with assignment.

(Note for experts: this is not the same as the procedure calling stack, which is a different area reserved for keeping track of which procedures are currently active.)

Consider a statement in which the value of an arithmetic expression is printed, for example:

```
2 + 2 =>
** 4
```

This statement is processed in two stages; firstly the arithmetic expression is evaluated and then secondly the result is printed. The result of evaluating the arithmetic expression is left on the stack, from where the print routine removes it.

As with a stack of plates, the last thing put on the stack is the first one you get off. Because the stack can accommodate more than one result, it is possible to write several expressions separated by commas, or semi-colons, each of which causes something to be put on the stack. For example:

```
2 + 3,  10 * 2,  5 - 3 =>
```

If this instructions is obeyed then three numbers are put on the stack with 5 at the bottom, then 20 and finally 2 on the top. The print arrow prints the ENTIRE stack, starting from the bottom, and empties it, thus:

```
** 5 20 2
```

Note that the top element of the stack is printed last.

The stack is in some ways like a datastructure, e.g. a variable-sized list or a vector, to which other Pop-11 objects can be added or removed at one end only, like a stack of dishes on to which new dishes can be added, one at a time, and from which the top item can be removed.

The stack is implemented differently from a Pop-11 datastructure, for efficiency. And it is not itself an object that can be put in a list, assigned to a variable, or given as input to a procedure. In fact there is no explicit way of referring to the stack. Rather it is implicitly referred to in very many Pop-11 instructions. It is not, what is sometimes referred to as a ”first-class object”.

A language that uses a stack in a similar way to Pop-11 is Forth. Some pocket calculators also use a stack in the same sort of way.

3.1 Procedures communicate via the stack

Pop-11 procedures take their parameters, if any from the top of the stack, and leave their results, if any, on the stack. Thus the procedure SIN removes the top item from the stack, computes its sine, and place this number on the stack. When we write:

```
sin(45) =>
** 0.707107
```

Then three things happen:

```
(1) 45 is put on the stack
(2) SIN is called (invoked, applied, run...). SIN removes the
    top item (in this case, 45), computes its sine (in this case
    0.7071), and puts this on the stack
(3) The print arrow, '=>' prints and empties the stack.
```

Had we wished to be obscure, we could have written, with exactly the same effect:

```
45;
sin();
=>
```

If there is nothing on the stack for a procedure like SIN then it complains and we get a MISHAP message.

Whatever is currently on the top of the stack is used by the procedure SIN. If this expression were executed with the stack empty, a MISHAP message would be printed indicating that the stack had underflowed, thus:

```
sin() =>
;;; MISHAP - STE: STACK EMPTY (missing argument? missing result?)
;;; DOING    :  sin compile compile
```

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

Note the difference between writing

```
sin;
```

which loads the sin procedure itself onto the stack and

```
sin();
```

which actually causes the sin procedure to be executed. It may be useful to think of ”()” as the ”doit” brackets. If there’s anything between the brackets, that says ”doit to...”, e.g. sin(45).

3.2 The DOT-notation for procedure calls

Readers familiar with the language FORTH, you may be interested that Pop-11 admits an alternative notation for procedure calls, sometimes described as ”postfix” or ”reverse polish” notation. This involves placing arguments, separated by commas, before procedure names and preceeding procedure names with a dot ”.” to indicate that the procedure is to be applied.

E.g. instead of:

```
perim(3,5) =>
```

you can use

```
3, 5 .perim =>
```

I.e. the "." can be used to invoke a procedure. Thus Pop-11 allows, instead of ‘sin(45)’, ‘45.sin’, which can be interpreted as put 45 on the stack and then run SIN. Similarly

```
sin(sqrt(100))
```

can be expressed as

```
100.sqrt.sin

sin(sqrt(100)) =>
** 0.173648

100.sqrt.sin =>
** 0.173648
```

The dot notation corresponds more closely to the actual order of processing, though the former is the more conventional way of representing procedure calls. We shall not use this POSTFIX notation in this introduction, though it is popular with some Pop-11 users.

Note that if infix operators are used in postfix notation, dots are not required, e.g.

```
2 * 3 + 4 * 5 =>
```

is equivalent to

```
2, 3 *, 4, 5 * + =>
```

The two can be mixed:

```
sqrt(3) + 5
```

is the same as

```
3.sqrt, 5 +
```

Although some Pop-11 users like the postfix notation, it can make programs obscure if used excessively. A common convention is to use it only for accessing components of datastructures, as in the following expression denoting the second element of list:

```
list.tl.hd
```

3.3 Assignment as a two stage operation

The statement:

```
2 + 2 -> x;
```

takes place in two main stages, viz:

```
'2 + 2;'
```

then

```
'-> x;'
```

First the arithmetical expression is evaluated and the result, 4, is left on the stack. Then the top element of the stack is removed and put into the variable x.

We can write a series of statements which put things on the stack followed by a series which remove than and put them in variables, for example:

```
2; 3; -> x; -> y;
```

These four statements do the following:

```
(1) Put 2 on the stack
(2) Put 3 on the stack
(3) Remove the top element (ie 3) and put it in the variable x
(4) Remove the top element (ie 2) and put it in the variable y
```

So, the following will swap the values of x and y.

```
x; y; -> x; -> y;
```

Expressions may be separated by commas, and assignments need no separators, so this can be abbreviated to:

```
x, y -> x -> y;
```

Notice that '-> x' removes the top item from the stack, and assigns it to x, whereas the reverse operation:

```
x;
```

Just COPIES the value of x onto the stack, leaving x unchanged. We can demonstrate this difference as follows.

```
vars x;
77, 88, 99;      ;;; put three things on the stack
-> x;            ;;; take one off
=>
** 77 88         ;;; two things left

x; x; x; =>      ;;; copy value of x three times onto stack
** 99 99 99
```

3.4 The stack and arithmetic expressions

The stack is used while evaluating arithmetic expressions so that a statement such as:

```
2 + 3 -> x;
```

actually takes place in four steps, thus:

```
(1) Put 2 on the stack
(2) Put 3 on the stack
(3) Do the addition, that is remove the top two items on the stack
    (ie 2 and 3) and replace them by their sum (ie 5)
(4) Remove the top item from the stack (ie 5) and put it in
    the variable X.
```

A more complicated example, such as:

```
2 + 3 * 4 - 5 =>
```

takes place in EIGHT steps, which are left as an exercise for the reader to describe, some of which are:

```
(4) Do a multiplication
(5) Do an addition
(7) Do a subtraction
(8) Print the contents of the stack
```

3.5 Implicit uses of the stack

The Pop-11 stack is implicitly referred to in the following contexts:

```
1. Any Pop-11 expression which denotes some object, is implicitly
   an instruction to create or find the object and put it on the
   stack. (For most types of object this really amounts to putting a
   "pointer" to the object on the stack. The object itself remains
   somewhere in memory. For integers and single precision decimals,
   a copy of the internal representation is put on the stack.)

   E.g.
        "3 + 5"                     puts  8 on the stack
        "hd(tl([the black cat]))"   puts "black" on the stack
         x,y,z;                     puts the values of x, y and z
                                    on the stack, in that order.

2. When a procedure is invoked with some arguments, the arguments
   are put on the stack and then the procedure is run. E.g.
        3 + 4
          Puts 3 on the stack, then 4, then runs +

        hd(list)
          Puts the value of list on the stack then runs hd

3. If a procedure invoked with arguments has some input variables,
   then when the procedure is run, the appropriate number of items
```

is taken off the stack and assigned to the variables within the procedure. For example, consider the procedure perim defined in Chapter 1, with the heading

```
define perim(len, breadth) -> total;
```

The command to run perim, will cause the item on top of the stack to be removed and assigned to to the input local variable breadth, then the next item will be removed from the stack and assigned to len, and then the instructions in the procedure definition will be obeyed, using these variables.

4. If a procedure is defined to produce any results, then invoking the procedure will implicitly cause the results to be put on the stack when the procedure is finished. For example, if the numbers 3 and 4 are on the stack, and the addition procedure + runs, then the procedure, when it has finished, will replace the top two items on the stack with the single result, the number 7. So the instruction

```
3 + 4 =>
```

is equivalent to these four instructions

```
3;
4;
+;
=>
```

Another example is what happens when the perim procedure finishes. Because it has an "output" local variable in the header line, i.e. "total", then Pop-11 will ensure that just before the procedure finishes the value of the variable will be put on the stack.

Exercise: Explain why case 4 is just a special example of case 1.

5. When an assignment instruction is run, e.g.

```
x -> y
```

the left hand side, in accordance with point 1 above, causes the value of the expression to be put on the top of the stack (i.e. the value of the variable x is put on the stack), and then the rest of the instruction ("-> y") causes the item on the top of the stack to be removed and stored as the value of the variable y. (I.e. the value is stored in the area of memory associated with the variable.)

NOTE: there is a potential point of confusion. When the left hand side of such an expression puts the value of x on the stack it does not remove the value from x. Thus x still has the same value

```
as before and it can be re-used. For example the instructions:

     x; x; x;

cause the value of x to be put on the stack three times.

By contrast, when the top of the stack is assigned to a variable
it is removed from the top of the stack. Thus if there is only
one thing on the stack you can obey the instruction to move it
to y, thus
     99;          ;;; Put 99 on the stack
     -> y;        ;;; move top of stack to y

But attempting to do the same again

     -> y;        ;;; move top of stack to y

will cause an error message to be printed out if there was
originally only one thing on the stack.
;;; MISHAP - ste: STACK EMPTY (missing argument? missing result?)
```

Note: although you can assign the top of the stack to a variable, you cannot assign it to a constant object, such as the number 37, or a string, or a word. If you try any of these:

```
3 + 4 -> 7;
"cat" -> 'dog';
33 -> [a b c];
```

then you will get the following error message:

```
;;; MISHAP - iue: IMPERMISSIBLE UPDATE EXPRESSION (e.g. after -> or ->>)
        33 -> 44;
```

3.6 Example: implicit uses of the stack when running "perim"

The previous section listed five ways in which Pop-11 programs implicitly use the stack. We can now combine points 1, 2, 3, 4, and 5, and look closely at the implicit stack instructions when Pop-11 runs the procedure perim, defined in Chapter 1.

First a reminder of its definition in chapter 1:

```
define perim(len, breadth) -> result;
    (len + breadth) * 2 -> result
enddefine;
```

Assuming the definition has already been compiled, we can step through what happens when it is obeyed in the following instruction (declare the variable num, compute the perimeter of a 3 by 4 room and assign the result to num):

```
vars num;
perim(3, 4) -> num;
```

This implicitly "expands" into the following Pop-11 virtual machine instructions:

```
STEP    What happens                      VM instruction

(a)    put 3 on the stack                (i.e. PUSH 3),
(b)    put 4 on the stack                (i.e. PUSH 4),
(c)    run perim                         (i.e. CALL perim),
        this will do various things and leave a result on the stack
(d)    move the top of stack to num      (i.e. POP num)
```

We can look more closely at step (c), i.e. what happens when the procedure runs. Look at the definition, from chapter 1:

```
define perim(len, breadth) -> result;
    (len + breadth) * 2 -> result
enddefine;
```

When this runs, with 3 and 4 on the stack, as in step (c) above it does the following:

```
STEP     What happens                             VM instruction

(ca)     move top of stack (i.e. 4) to breadth    (POP breadth)
(cb)     move top of stack (i.e. 3) to len        (POP len)
(cc)     copy value of len (i.e. 3) to stack      (PUSH len)
(cd)     copy value of breadth (i.e. 4) to stack  (PUSH breadth)
(ce)     run +                                    (CALL +)
            (which takes 3 and 4 off the stack and puts back 7)
(cf)     put 2 on the stack                       (PUSH 2)
(cg)     Run the multiplication procedure         (CALL *)
            (this takes 7 and 2 off and puts back 14)
(ch)     Move top of stack (14) to result         (POP result)
(ci)     Finally, because result is an output
        local, its value is put on the stack     (PUSH result)
```

It is not absolutely essential to understand such details of implicit stack manipulation on order to develop Pop-11 programs, but it will make many things clearer and will help with more advanced programming. For practice try the following exercises:

3.7 Exercises on the stack

1. Relate each of the above steps (a) to (d) and (ca) to (ci) above to the five types of implicit uses of the stack.

2. Step through the examples with different input numbers, e.g. 4 and 5 instead of 3 and 5, keeping track of which intermediate numbers are created.

3. Do the same with the procedure volume, defined in Chapter 1. (It is easier than perim.)

4. If you have found the explanations so far confusing, you may find it helpful to work through the online Poplog file TEACH STACK, which explains these points in more detail, with more examples. TEACH DEFINE may also help.

3.8 More examples of uses of the stack

Put four things on the stack, two integers and two words.

```
3, "cat", 99, "blue";
```

The assignment arrow can be used to remove the top of the stack and store it as the value of x.

```
-> x;
```

Now print out what is left on the stack.

```
=>
** 3 cat 99
```

I.e. only three things left. What was the fourth is now the value of x:

```
x =>
** blue
```

(The non-destructive assignment arrow "->>" described later, assigns the top of the stack without removing it.)

3.9 "->" does not always represent an assignment

We have seen that "->" can also be used in the heading of a procedure definition, to define "output locals". In this context "->" does not cause anything to be assigned. This is explained more fully in the section on defining procedures.

3.10 The print arrow "=>"

The print arrow is very commonly used to show what is on the stack. It has two different uses, one inside procedures and one at "top level" i.e. when you are giving individual commands to Pop-11.

```
1. In a procedure "=>" prints only one item, the one on top of the
   stack, and removes it from the stack.

2. At "top level" "=>" prints out everything on the stack, bottom
   item first, top item last, and clears the stack.
```

The reason for the restriction in (1) is that when procedures are running there may be several items on the stack waiting to be dealt with by other procedures than the one that calls =>. So those items should not be removed. (This restriction can be overcome by using the popval procedure to simulate top level instructions, in the format: popval([=>]);

But that's recommended only for experienced programmers.)

3.11 The pretty print arrow ==>

The ”pretty print arrow” is similar to ”=>” but will print complex lists and vectors out in a more readable form. E.g. using the variable rooms defined in Chapter 1, compare:

```
rooms =>
** [[room1 10 12 8] [room2 6 11 8] [room3 15 11 8] [room4 10 12 9]
 [room5 21 11 9]]

rooms ==>
** [[room1 10 12 8]
    [room2 6 11 8]
    [room3 15 11 8]
    [room4 10 12 9]
    [room5 21 11 9]]
```

Another difference is that ==> never prints out more than ONE item from the stack.

3.12 Using assignment to store something in a structure

So far the assignment arrow has been used only with a variable on the right hand side.

Besides storing things in variables, assignment can be used to store something in a structure. E.g. we may create a list, then use assignment to alter its third element:

```
vars list = [a b c d e];
list =>
** [a b c d e]

"D" -> list(4);         ;;; alter the 4th element
list =>
** [a b c D e]
```

Here the assignment has ”updated” the previously existing list, unlike the expression

```
[a b c D e] =>
```

which creates a new list. This is an example of what is sometimes called programming with ”side effects”. I.e. when an instruction is obeyed, it not only produces results which can be assigned to variables, or used to construct new datastructures, but it also alters a pre-existing structure. Side effects are often a source of obscure bugs in programs, and there are some computer scientists who think they should not be allowed, so they prefer to use ”pure functional” languages that permit no side effects. (An example of such a language is Miranda.) However, there are many ways of designing useful software that make use of side effects, so Pop-11 allows them. If you use these facilities you will need to be very careful.

3.13 Invoking the updater of a procedure

A common way to change an existing structure is to use a procedure that has an updater. An updater is a procedure that is invoked on the right side of an assignment arrow. A procedure and its updater together are sometimes referred to as a ”doublet”. For example, the Pop-11 procedure for accessing the first element of a list, called ”hd” is really a doublet, consisting of two procedures: an accessor and an updater. To illustrate this, first create a list held in the variable person, and create three spare variables for use later.

```
vars
    person = [[name john][sex male][age 27]],
    p, x, y;
```

We’ll assign person to p, in order to illustrate some points later.

```
person -> p;
```

Notice that p is not a COPY of person. It is the very same thing:

```
person == p =>
** <true>
```

The ”accessor” of ”hd” can be used in many different contexts, e.g.

```
hd(person) =>
** [name john]

hd(hd(person)) =>     ;;; get the first element of the first element
** name

hd(person) -> y;      ;;; assign the first element to y
y =>
** [name john]

hd(hd(person)) -> x;
x =>
** name
```

Note that person is unchanged by this: the original first element is still there:

```
person =>
** [[name john] [sex male] [age 27]]
```

p is also unchanged:

```
p =>
** [[name john] [sex male] [age 27]]
```

As the previous examples show, the accessor of hd is invoked on the left hand side of an assignment arrow. We can invoke its updater on the right hand side. This will cause the list to be changed. Thus:

```
[name fred] -> hd(person);      ;;; invoke updater of hd

person =>
** [[name fred] [sex male] [age 27]]
```

The value of p is still the same as the value of person, so p is also changed:

```
p =>
** [[name fred] [sex male] [age 27]]
```

But y has been left unchanged:

```
y =>
** [name john]
```

We can use y to restore the original hd(person) thus:

```
y -> hd(person);
person =>
** [[name john] [sex male] [age 27]]
p =>
** [[name john] [sex male] [age 27]]
```

The next example is more subtle. We are going to change the first element of the first element of person. I.e. hd(hd(person)). This means that there will be two occurrences of ”hd” on the right of the assignment arrow. But one of them will not be invoked as an updater, i.e. the one inside the brackets: that is used access the item that will be updated by the call of hd outside all the brackets thus:

```
"forename" -> hd(hd(person));
```

Check the result:

```
person =>
** [[forename john] [sex male] [age 27]]
```

As before this affects p also, because it is still ”pointing to” the very same thing as person. What about y?

```
y =>
** [forename john]
```

Because y was the very same thing as the first item of person, changing part of the first item of person also changed part of y, unlike the case where replacing the whole of the first item of person left y unchanged.

One way to think of this is to think of a variable as a label attached to its value by a piece of string. When you assign something to a variable, you keep the same label, but you attach the string to something else. So

```
hd(person) -> y;
```

Takes the object that was the first element of person, and connects it to the end of y's string. Then an instruction like:

```
"forename" -> hd(hd(person));
```

changes the contents of the first item of the list person, and so that changes the contents of the item to which y's string is attached.

By contrast the instruction

```
[name fred] -> hd(person);
```

replaces the thing that was the first element of person, but y's string is still connected to what it was previously connected to, which is no longer part of the list person. So this does not change y at all. And if we THEN do

```
"forename" -> hd(hd(person));
```

it will not affect y.

This example should help to show why programming with side-effects can be confusing, and mistakes can be made. However, it is also often very convenient, as long as you thing VERY clearly about what you are doing!

3.14 Other updaters: subscrv, subscrs

The procedures hd and tl are used for accessing and updating the head of a list or its tail (the sublist containing everything except the first element). Each is a doublet, i.e. each has an updater.

The procedure subscrv is available for accessing and updating a particular element of a vector, as illustrated here:

```
vars vec = {the cat on the mat};

subscrv(3, vec) =>       ;;; access the third element of vec
** on

"under" -> subscrv(3, vec);     ;;; update the third element
vec =>
** {the cat under the mat}
```

For accessing individual components of a string, use subscrs. The components of a string are characters, which, in Pop-11 are the 8-bit ascii character codes, as described in HELP ASCII. E.g. the code for A is 65, for B is 66, etc., and these can also be represented as ‘A‘, ‘B‘, etc. Thus we can access or modify characters in a string:

```
vars string = 'ABCDE';

string =>
** ABCDE

subscrs(3, string) =>           ;;; get the third character
** 67

`Z` -> subscrs(3, string);      ;;; update the third character
string =>
** ABZDE
```

3.15 Using a numerical subscript to access or update a structure

In Pop-11, it is possible to access or update the N'th element of a list or vector type datastructures simply by applying the structure to the number N.

```
vars
    list1   = [a list of words],
    vec1    = {contents of vector},
    string1 = 'ABCDE';
```

In each case we can access the second element simply by applying the structure to the number 2:

```
list1(2) =>
** list

vec1(2) =>
** of

string1(2) =>
** 66
```

Similarly a numerical subscript can be used for updating:

```
"set" -> list1(2);
list1 =>
** [a set of words]

"without" -> vec1(2);
vec1 =>
** {contents without vector}

`?` -> string1(2);
string1 =>
** A?CDE
```

This facility makes it easy to write a program that will operate on objects of different types, e.g.:

```
define iselement(item, structure) -> answer;
    lvars num;
    for num from 1 to length(structure) do
        if item == structure(num) then
            true -> answer;
            return();
        endif;
    endfor;

    false -> answer;
enddefine;

iselement("set", list1) =>
** <true>

iselement("set", vec1) =>
** <false>

iselement(`?`, string1) =>
** <true>
```

3.16 updaters, the stack and "explode"

Typically an updater procedure, like the updater of hd, removes one item from the top of the stack, and stores it somewhere. Pop-11 updaters are not required to remove exactly one item. For example, if you have a three element list or vector it can be used to store three items from the top of the stack using the updater of the procedure explode, as explained below.

Explode itself puts all the elements of a structure on the stack:

```
vars vec = {a b c};
explode(vec) =>     ;;; put all the elements of vec on the stack
** a b c
```

This leaves vec unchanged.

```
vec =>
** {a b c}
```

We can use the updater of explode to take three things off the top of the stack and store them in vec:

```
[list 1] , [list 2], [list 3] -> explode(vec);
vec =>
** {[list 1] [list 2] [list 3]}
```

So vec is now a vector containing three lists instead of three words.

The procedure "fill" is similar to the updater of explode, but also returns the "filled" datastructure as its result. See HELP FILL)

3.17 Defining updaters

Any Pop-11 procedure is allowed to have an updater associated with it, whether it is concerned with accessing datastructures or not. Equally, it is not essential for an updater procedure to take something from the stack. For sophisticated programmers this is often a powerful tool, which allows related pairs of procedures to be closely associated, and used, for example, in connection with Pop-11's "dynamic local expression" mechanism, introduced in HELP DLOCAL.

The syntax for defining the updater of a procedure is of the form:

```
define updaterof ....
```

which is illustrated in HELP DEFINE, and discussed in a later chapter.

By default a newly created procedure is given an updater that does nothing except produce an error message, like this:

```
66 -> sqrt(4) =>
;;; MISHAP - COMPILING CALL TO NON-EXISTENT UPDATER
;;; INVOLVING:  <procedure sqrt>
```

HELP UPDATER gives more information. Experts can look at REF PROCEDURE.

3.18 Non−destructive assignment ”->>”

The normal assignment arrow REMOVES the top item of the stack. It is sometimes convenient to assign the top element of the stack yet leave it there. This is done with the arrow ”->>”. For instance this can be used to assign the same thing to several different variables:

```
0 ->> x ->> y -> z;
```

This puts 0 on the stack, then COPIES the top of the stack into x, then COPIES the top into y the MOVES the top of the stack into z.

3.19 Exercise

Work out and write down what happens when the following imperative is obeyed:

```
(2 + 3) * 4 + 5 - 2 -> subscrv(4 - 2, vec1);
```

Try it out and compare the result with what you expect.

3.20 Multiple assignments

All the following imperatives use the stack:

```
(a) expression =>
(b) expression;
(c) expression -> variable;
(d) -> variable;
```

we can replace ‘expression’ by a sequence of expressions separated by commas and we can replace ‘-> variable’ by a sequence of such things, for example:

```
1, 2, 3 -> z -> y -> x;
```

After this imperative has been executed z will have the value 3 and x will have the value 1. Because the assignments remove the items from the stack in the reverse order from their placement on the stack, this notation can be confusion. So Pop-11 allows a special syntax for assigning to several variables in the natural order. The above instruction is equivalent to:

```
1, 2, 3 -> (x, y, z);
x =>
** 1
```

More usefully, after executing:

```
x, y -> x -> y;
```

or, expressing the same thing more clearly:

```
x, y -> (y, x);
```

the values of x and y will have been swapped.

Beware of writing programs which leave things lying around on the stack for later use in a wanton fashion. Some people do this for the sake of efficiency, since stack operations are very fast. However it can make programs hard to understand, debug, or modify.

3.21 Some procedures which work on the stack

STACKLENGTH can be used to find out how many things are on the stack. When invoked it counts the number of items on the stack and then leaves that number on the stack, for example:

```
2 + 3, 9 - 5, stacklength() =>
** 5 4 2
```

Notice that in Pop-11 a procedure which takes no arguments is invoked by writing its name followed by two parentheses.

DUP

This procedure merely duplicates the top element of the stack.

```
1,2,3,4, dup() =>
** 1 2 3 4 4
```

SETSTACKLENGTH

setstacklength(N)

Increases or decreases the stack length to N, if necessary by removing items or adding occurrences of [], the empty list (because the procedure is used by the Poplog Lisp compiler). Examples:

```
1,2,3,4, setstacklength(2), setstacklength(6) =>
** 1 2 [] [] [] []
```

3.22 Clearing items from the stack (erase)

ERASE removes one item from the stack, for example:

```
2 + 3, 9 - 5, erase() =>
** 5
```

or

```
5 + 5, 66, erase(), 8 * 8, 10 - 5, erase() =>
** 10 64
```

The assignment arrow followed by ";" or "," (or any other expression terminator) has the same effect as "erase()". Thus:

```
5 + 5, 66, -> , 8 * 8, 10 - 5, -> =>
** 10 64
```

Whether you use this or "erase" is a matter of taste. (However, "erase" has to be used when a stack-clearing procedure is referred to explicitly, e.g. assigned to another procedure as input.)

ERASENUM takes an integer and removes that number of items from the stack. E.g.

```
1,2,3,4,5,6; erasenum(2) =>
** 1 2 3 4
```

Which is equivalent to: erasenum(1,2,3,4,5,6,2) =>

SETPOP clears the stack. For example;

```
2 + 3;
setpop();
stacklength() =>
** 0
```

Setpop does many others things besides clear the stack; it also aborts any running program. It also prints a message whenever it is called. An immediate call of setpop occurs whenever you type CTRL-C (unless you have redefined the procedure INTERRUPT, described later).

For more on procedures that manipulate the stack, see REF STACK

3.23 Removing unwanted items from the stack

The procedure erase simply removes one item from the stack, so in:

```
erase(2 + 2) =>
```

the expression in the parentheses will be evaluated but no result will be printed. Erase is sometimes useful if you want only one of the results of a procedure which produces two results. E.g. the operation '//' performs integer division putting the remainder and the quotient on the stack:

```
23 // 6 =>
** 5 3
```

If you wanted to assign only the remainder to X, you could do:

```
erase(23 // 6) -> x;
x =>
** 5
```

or, using an extra "assignment to nothing" instead of erase:

```
(23 // 6) -> -> x;
x =>
** 5
```

(Actually the infix operation REM does the same:

```
23 rem 6 -> x;
x =>
** 5
```

)

To assign the divisor first assign the top of the stack, then erase the second result:

```
erase(23 // 6 -> x);
x =>
** 3
```

or, using assignment to nothing

```
23 // 6 -> x ->;
x =>
** 3
```

(This can be done instead using the infix operator ‘div’:

```
23 div 6 =>
** 3
```

)

3.24 Swapping items using the stack

As described earlier, the following swaps the values of X and Y:

```
x, y -> (y, x);
```

And this is equivalent to

```
x, y -> x -> y;
```

The statement:

```
-> (y, x); x, y;
```

has a rather different meaning. Effectively, it swaps the top two elements of the stack and as a side-effect stores the top two items in x and y. E.g.

```
1, 2, 3, 4;
-> (y, x); x, y;

=>
** 1 2 4 3

x, y =>
** 4 3
```

If we wanted to swap the first (ie top) and third elements of the stack we could do:

```
vars x, y, z;
-> x -> y -> z; x, y, z;
```

or, more clearly,

```
-> (z, y, x); x, y, z;
```

3.25 Conditionals and the stack

Previously we noticed that the use of a conditional like

```
if x = 10 then x=> endif
```

depended on the fact that "=" is a predicate, i.e. a procedure producing a boolean result. This result is left on the stack. You can think of a conditional imperative of the form

```
if <condition> then <action> endif
```

as being roughly equivalent to something like the following:

```
1. Put value of <condition> on stack

2. If the top element of the stack is true perform the
    <action>, otherwise jump to after 'endif'.
```

This means that if the expression between 'if' and 'then' produces no result, then an error will occur. For example, "3 = x" produces a boolean result, whereas the assignment "3 -> x" is simply an imperative, and does not, so:

```
if 3 -> x then x => endif;

;;; MISHAP - ste: STACK EMPTY (missing argument? missing result?)
```

The portion of Pop-11 between 'if' and 'then' produced no result.

Strictly speaking, Pop-11 is defined so that anything other than false can be used as equivalent to true in a condition:

```
if 99 then 66 => endif;
** 66
```

For this reason it is sometimes convenient to define a procedure so that it returns either false or some object it is discovered. For example, a procedure which searches down a list looking for a word, and if it finds it returns that word as its result, otherwise returns false:

```
define findword(list) -> result;
    lvars item;
    for item in list do
        if isword(item) then
            item -> result;
            return();
        endif;
    endfor;
    false -> result;
enddefine;

findword([1 2 3 4 5 6]) =>
** <false>

findword([1 2 3 4 five 6]) =>
** five
```

This might be used with the non-destructive assignment arrow thus:

```
vars w;
if findword(list) ->> w then
        .... do something with w ....
else
        report_failure();
endif;
```

4 CHAPTER.4: PROCEDURES IN POP-11

This chapter reviews some of the features of procedures in Pop-11, including several that are not found in other languages, and also provides more information on how to define procedures.

4.1 Procedures as ”first class objects”

In Pop-11 procedures (which are referred to by Lisp programmers as functions) are ”first class objects”. This is a phrase used by some computer scientists to characterise programming languages with rich facilities for manipulating procedures, in contrast with programming languages in which you can merely define procedures and then run (invoke, execute) them, but cannot do anything else with them: in the latter case procedures are not ”first class” objects.

In Pop-11 you can do several other things with procedures besides defining them and running them. If P is a procedure, then:

o You can give P as input to another procedure (which is also possible in several other common languages e.g. Pascal). E.g. you can give the procedure sqrt as input to the procedure maplist, to get the square roots of a list of numbers thus:

```
maplist([4 16 25 36], sqrt) =>
** [2.0 4.0 5.0 6.0]
```

o You can assign P to one or more variables, and use those variables to invoke it:

```
vars my_sqrt;
sqrt -> my_sqrt;
my_sqrt(16) =>
** 4.0
```

o you can return P as the result of running another procedure

o you can store P in a datastructure, like a list or a vector

```
[^sqrt ^hd ^maplist] =>
** [<procedure sqrt> <procedure hd> <procedure maplist>]
```

o you can combine (or”compose” P with other procedures to form more complex procedures using the Pop-11 concatenation operator <>, as shown in a later section of this chapter. (See REF * PROCEDURE/’Generic Datastructure Procedures’)

o you can combine P with some data to form a new procedure using ”partial application”. (See HELP * CLOSURES). For example, the built in Pop-11 procedure member, referred to in Chapter 1, takes some item and a list and decides whether the item is in the list. So if you have a list of colour words, you can define a procedure iscolourword by combining member with the list, thus.

```
vars colours = [red orange yellow green blue indigo violet];

vars iscolourword = member(% colours %);
```

After that, iscolourword is a ”Pop-11 closure”, a combination of a procedure with some data, which can be treated as a new procedure, thus:

```
iscolourword("red") =>
** <true>

iscolourword("square")=>
** <false>
```

o If P is defined inside another procedure Q, and accesses some lexical variables of Q, then you can (in some cases) form a "lexical closure" of P which is another procedure with associated data involving the values of the variables in Q at the time the closure was created. (See HELP * CLOSURES, HELP * LVARS/'lexical closures')

o You can give P a new updater, e.g. using the form:

```
<new_updater> -> updater(P);
```

o You can use P as the updater of another procedure, e.g. using:

```
P -> updater(Q);
```

Most of the points mentioned above have either been illustrated in previous chapters or will be explained below. See also REF PROCEDURE

These features make Pop-11 a very powerful programming language. In particular, the ability to re-use one procedure in a variety of ways to create new procedures, helps with the design of modular programs and also facilitates the creation of relatively compact, easily maintained, but very general programs.

This is closely related to, though distinct from the re-use of "methods" in object oriented programming (also available in Pop-11 in the FLAVOURS and OBJECTCLASS packages).

In addition to Pop-11 programs being able to use existing procedures and combine them to form new procedures according to the needs of the occasion, Pop-11 also allows a running program to create new procedures from procedure definitions. One way of doing this for a running program to compile a file of procedure definitions which will then extend the capabilities of the program. Another way is for a program to create a list of text items then apply the procedure popval to it. Popval is able to invoke the compiler to build a new procedure, or obey instructions in the list. For example this will create a new procedure called "greet":

```
popval( [ define greet(x); [hello ] => x => enddefine; ]);
```

Popval can be given lists that have been created by programs rather than by the programmer. In this way a program can extend itself in a way that depends on the environment when it runs, rather than on the programmer having decided in advance what the extension should be. This can be a powerful learning mechanism.

4.2 Other languages able to manipulate procedures

There are not many other languages with similar generality: certainly none of the most commonly used languages allow these uses of procedures, e.g. Pascal, C, Fortran, Cobol.

Some of the "functional" languages, e.g. ML, Miranda and Scheme, come close to Pop-11, though they do not have updaters or partial application. Some of them have other powerful features. (E.g. Miranda has "lazy evaluation", which is only partially mirrored in Pop-11 by "dynamic lists", described in a later chapter.)

4.3 Example: creating new procedures from old

This section illustrates some of the ways in which procedures can be manipulated to create new procedures. We'll create a procedure called double, then show how to combine it with itself to get a new procedure that quadruples. We'll also show how to combine it with a general list-manipulating procedure to produce a new specialised procedure for operating on lists of numbers.

The example will use procedure composition and partial application. Suppose you have a procedure that takes a number and doubles it:

```
define double(num) -> num;
    ;;; Double num then return it as a result
    num + num -> num;
enddefine;

double(53) =>
** 106
```

We can combine this procedure with itself to produce a procedure that calculates quadruples:

```
(double <> double)(4) =>
** 16
```

We can also illustrate how the procedure (or function) double can be given as input to a procedure that takes a list and a procedure, applies the procedure to every element of the list, and returns a list of the results. We can define such a "list mapping" procedure thus:

```
define list_results(list, proc) -> newlist;

    lvars item;
    [% ;;; start making a list
        for item in list do

            proc(item)          ;;; apply proc to the item
                                ;;; leaving the result on the stack
        endfor
        ;;; now finish making the list of results
    %] -> newlist
enddefine;
```

(There is actually a built in procedure in Pop-11 called maplist, that does something very similar, except that it includes instructions to produce an error message if the second argument is not a procedure.)

We can give double as second argument to list_results, thus:

```
list_results( [1 2 3 4 5], double ) =>
** [2 4 6 8 10]
```

4.4 Using "partial application" to create a closure

Previously we saw how to define the procedure iscolourword by using partial application to combine member with a list. We can produce another closure, as follows:

If we often wanted to double all the numbers in a list, we could create a new procedure by partially applying list_results to double, and then re-using the new procedure as needed.

To do this we use the partial application brackets "(% ... %)" which have the effect NOT of running a procedure but of creating a NEW procedure that can be run later on with the data between the brackets.

If P is a procedure that requires three numbers to run, then P(% 3, 4 %) is a combination of P and two numbers, which can be run later on if the third number is given.

Similarly the procedure list_results can be partially applied to the procedure double, to create a new procedure that can be run later on if a list is given to be provided as the first argument of list_results. E.g.

```
vars double_elements = list_results(% double %);
```

or, using a better, but equivalent syntax, which shows more clearly that we are defining a new procedure:

```
define double_elements = list_results(% double %)
enddefine;
```

This defines double_elements to be a procedure made from the procedure list_results by combining it with the procedure double as its "frozen argument".

We now have a new procedure:

```
double_elements =>
** <procedure double_elements>
```

We can apply double_elements to lots of different lists:

```
double_elements([11 22 33]) =>
** [22 44 66]

double_elements([66.5 103.45 89  77.002 1000000]) =>
** [133.0 206.9 178 154.004 2000000]
```

4.5 Using both partial application and procedure composition

Now suppose we wanted to do the same using not double, but a procedure that produces the double of the double, i.e. one that quadruples its input. We can create such a procedure by composing double with itself, thus:

```
(double <> double)(2) =>
** 8
```

and we can give that procedure as the second argument input to list_results:

```
list_results([1 2 3 4 5], double <> double) =>
** [4 8 12 16 20]
```

Now if we want to make a procedure that takes a list of numbers, and creates a new list containing the quadruples of all those numbers we can partially apply list_results to a procedure made by composing double with itself, e.g.

```
vars quadruple_elements = list_results(% double <> double %);
```

or, equivalently

```
define quadruple_elements = list_results(% double <> double %)
enddefine;
```

so

```
quadruple_elements([1 2 3 4 5]) =>
** [4 8 12 16 20]
```

Incidentally, this is equivalent to

```
list_results( list_results( [1 2 3 4 5], double ), double ) =>
```

or

```
[1 2 3 4 5], double, list_results(), double, list_results() =>
```

(as explained in Chapter 3.)

Here we have invoked list_results twice. This creates a list of doubles, then a new list of doubles of the doubles. This is wasteful because creating the intermediate temporary list uses up space that is not needed again and will have to be reclaimed by the Pop-11 garbage collector. This is one of many examples of ways in which two mathematically equivalent methods of doing something may be very different in terms of efficiency. (See also HELP EFFICIENCY)

4.6 Exercises on procedure creation and manipulation

1. Using the notation defined above, use procedure composition and partial application to create a procedure called octuple_elements which, when applied to a list of numbers, creates a new list containing the results of multiplying those numbers by 8, i.e. the double of the double of the double of each number.

2. Note that the built-in Pop-11 procedure maplist can be used instead of list_results, in most of the examples. Check this by changing all the examples to use maplist instead, and see how they work. If you give a non-procedure as the second argument of maplist an error will result.

3. Use procedure composition and partial application to combine the procedures maplist double and sqrt to produce a new procedure called list_sqrt_doubles that when applied to a list of numbers produces the square roots of the doubles of those numbers, and test it thus:

```
list_sqrt_doubles([2 8 12.5]) =>
```

It should produce

```
** [2.0 4.0 5.0]
```

4. Just to test your understanding, in preparation for the next section, try to work out what the following do, remembering that a vector, or list can be used as a procedure by applying it to a number. So when a list or vector of objects is given as second argument to maplist, an error results, whereas list_results is more generous! For example:

```
list_results([3 4], {cat dog mouse bird rabbit}) =>
** [mouse bird]

list_results([1 3], [a b c d e]) =>
** [a c]
```

What will this one produce?

```
list_results([2 4], [[list 1] [list 2] [list 3] [list 4]]) =>
```

5. What will be printed out as a result of these instructions:

```
vars colourwords = [red orange yellow green blue indigo violet];

define select_colours = list_results(% colourwords %) enddefine;

select_colours([2 4]) =>

select_colours([1 3 5]) =>
```

6. What does it mean to say that a programming language treats procedures as first class objects? Why is this important?

7. What sorts of things can Pop-11 programs do with procedures besides calling them?

8. In what ways can Pop-11 programs create new procedures while they are running? How does the useof popval differ from the others.

4.7 Using non-procedures (e.g. lists, vectors) as procedures

So far we have seen how procedures can be used as data, e.g. giving double as argument to the composition operator, and saving the created procedure in a variable, etc.

Not only can many different things be done with procedures in Pop-11: it is also possible to treat certain objects as if they were procedures in order to obtain general and modular programs. For example, consider a list of three words:

```
vars numlist = [cat dog mouse];
```

Although this is not actually a procedure, it has, from the mathematical point of view, some of the properties of a procedure (or a function) in that you can think of it as mapping the first three integers onto three words. We can define a procedure that does this:

```
define int_to_word(int) -> result;
    if int == 1 then "cat" -> result;
```

```
        elseif int == 2 then "dog" -> result
        elseif int == 3 then "mouse" -> result
        else mishap('NUMBER TOO BIG', [^int])
        endif
    enddefine;
```

then we can use the procedure thus:

```
    int_to_word(1) =>
    ** cat
    int_to_word(2) =>
    ** dog
    int_to_word(3) =>
    ** mouse
    int_to_word(4) =>
    ;;; MISHAP - NUMBER TOO BIG
    ;;; INVOLVING:  4
    ;;; DOING    :  int_to_word compile ...
```

However, in Pop-11 you do not need to define such a procedure. You can use the list itself as a procedure by applying it to integers as if it were a procedure, or mathematical function, thus:

```
    numlist(1) =>
    ** cat
    numlist(3) =>
    ** mouse
    numlist(4) =>
    ;;; MISHAP - BAD ARGUMENTS FOR INDEXED LIST ACCESS
    ;;; INVOLVING:  4 [cat dog mouse]
    ;;; DOING    :  ...
```

Moreover, because numlist is a list, its contents can be changed, altering the behaviour of the procedure or function that it simulates:

```
    "snake" -> numlist(2);
```

then

```
    numlist(2) =>
    ** snake
```

Similarly vectors and strings can be treated as procedures, in Pop-11.

```
    vars
        vec = {966 77 88 99},
        string = 'the cat';

    vec(3) =>
    ** 88
```

```
string(1) =>
** 116

string(2) == `h` =>
** <true>
```

(Remember that the strings are actually vectors of 8 bit integers, representing character codes, as described in HELP ASCII, HELP STRINGS)

(NOTE: This use of integers for indexed access does not work with records in Pop-11: they are assumed to have components that are accessible only via particular named procedures associated with the record class.)

4.8 Arrays are procedures in Pop-11

An array in Pop-11 is something like a vector of vectors, or a list of lists. For example, suppose you wished to create a 3 by 4 array for later use to store information.

You could do this:

```
vars vecarray =
    { {undef undef undef}
      {undef undef undef}
      {undef undef undef}
      {undef undef undef}};

vecarray ==>
** {{undef undef undef}
    {undef undef undef}
    {undef undef undef}
    {undef undef undef}}
```

(Note that "undef" is a word that is provided in Pop-11 for use when a structure has to contain elements that are not yet defined.) We can store the word "cat" in the third field of the second vector thus:

```
"cat" -> vecarray(2)(3);

vecarray ==>
** {{undef undef undef}
    {undef undef cat}
    {undef undef undef}
    {undef undef undef}}
```

or access it thus:

```
vecarray(2)(3) =>
** cat
```

Then we can access or update any element of this structure by giving it two numbers, the first between 1 and 4, and the second between 1 and 3. This is in effect a two dimensional array with bounds 1 and 4 in the first dimension and bounds 1 and 3 in the second dimension.

However, Pop-11 provides a mechanism for creating entities called arrays which can have any number of dimensions and whose bounds can be any integers, positive or negative. So we can create a 3D array with bounds 2 to 5, 3 to 10, and -5 to +5 thus:

```
vars array3d = newarray([2 5 3 10 -5 5]);
```

To access its elements we can give it three numbers, one for each dimension, e.g.

```
array3d(3, 4, -4) =>
** undef
```

Initially the word "undef" is the value of each field. We can update it thus:

```
[a list] -> array3d(3, 4, -4);
```

then

```
array3d(3, 4, -4) =>
** [a list]
```

This shows that just as a vector or list can be treated as a procedure applicable to one number at a time, to get at a particular location, so an N-dimensional array is an object that can be treated as a procedure that is applicable to N integers to access or update a field in the array.

Because arrays in Pop-11 are treated as procedures, all the operations available on procedures can be used with them, including partial application. Thus if you can partially apply array3d to the number -4, to produce what is effectively a 2d array, which corresponds to one 2-d "plane" in array3d, thus:

```
vars array2d = array3d(% -4 %);

array2d(3, 4) =>
** [a list]
```

Again this ability in Pop-11 to abstract from the differences between arrays and real procedures and treat them all as procedures provides powerful facilities for generalisation and re-use of the same mechanisms in different contexts.

More will be said on arrays in a later chapter.

4.9 Properties as procedures

An N dimensional array provides a mapping or association from sets of N integers to values stored in the array.

Pop-11 also allows the creation of entities called "properties" that will map or associate any objects with any others. There are various ways of creating properties in Pop-11, depending on the particular sort of mapping that is required.

4.9.1 newmapping associates things compared using "="

A major decision in setting up an association or property is to decide what to do about two objects that are structurally similar with identical components, but are not the very same object. Should they be mapped onto the same thing or not? If so, use newmapping, otherwise use newproperty. (Actually newmapping is based on a more general procedure, newanyproperty, which will not be described here.)

Newmapping takes four arguments

```
a. A list giving an initial mapping. (This list will be empty
   in our examples here.)

b. An integer indicating the size the mapping is likely to grow to.
   This is actually used to decide the amount of space initially
   allocated to store the associations.

c. A default item, with which everything is assumed to be
   associated. (Often the Pop-11 object false, or undef is used.)

d. A boolean, i.e. the value true or false, which controls
   whether the space allocated for the mapping should be increased
   when appropriate, in order to increase the speed with which
   values are found.
```

(The second and fourth items merely affect efficiency, in most cases, and make no difference to the logic of the program.)

For example, suppose you were storing information about individuals, where an individual was always represented by a list like [john smith], [marylou walker], etc.

Then if you store information about each individuals by associating it with the lists, then if you re-created a list [john smith] later on, you might want the new list to be associated with the same information as the old list. In that case you should use newmapping, not newproperty.

For example, here are several mappings created using newmapping, with table sizes 10, default values 0 or false, and expandability set to true:

```
vars
    age = newmapping([], 10, 0, true),
    spouse = newmapping([], 10, false, true),
    occupant = newmapping([], 10, false, true);
```

We can then store information about different people, and rooms:

```
33 -> age([john smith]);
27 -> age([marylou walker]);

[john smith] -> spouse([judy smith]);
[mary low walker] -> spouse([bob walker]);

[john smith] -> occupant([room 3]);
[bob walker] -> occupant([room 5]);
[ted bloggs] -> occupant([room 6]);
```

We can then ask for information:

```
age([marylou walker]) =>
** 27

age(spouse([judy smith])) =>
** 33

occupant([room 4]) =>
** <false>

occupant([room 3]) =>
** [john smith]
```

Properties created using newmapping behave like procedures

Notice that the syntax for finding out what each property associates with an object is the same as the syntax for applying a procedure to an object. The properties, age, occupant and spouse are all both properties and procedures. We can test for this using the built in predicates isproperty and isprocedure:

```
isproperty(age) =>
** <true>

isprocedure(age) =>
** <true>
```

Compare:

```
isproperty(hd) =>
** <false>

isprocedure(hd) =>
** <true>
```

Using newmapping we created properties that would give the same result for lists created at different times, as long as they had the same elements. This is not so for properties created using newproperty: it requires exactly the same object as before in order to remember the mapping. A merely similar object will not do. (See HELP * EQUAL)

4.9.2 newproperty associates things compared using ”==”

Newproperty requires a list, an integer representing the space allocated for the association table, the default value with which objects are associated, and a fourth argument that specifies how space will be reclaimed by the garbage collector, which for now we'll assume must be ”perm”, meaning that space will not be reclaimed, unless the whole property is. For further details (See HELP * NEWPROPERTY and REF * newproperty)

For example, here are several mappings created using newproperty, with table sizes 10, default values 0 or false.

```
vars
    salary = newproperty([], 10, undef, "perm"),
    address = newproperty([], 10, undef, "perm");
```

We shall now again use lists to represent individuals, but this time we associate each list with a variable, so that later we can get at the very same list.

```
vars
    john = [john smith],
    mary = [marylou walker];
```

We can then store salary and address information;

```
15000 -> salary(john);
25000 -> salary(mary);

[room 3] -> address(john);
[room 5] -> address(mary);
```

Now having set up the mappings we can ask for the information, using the value of the variable as input to the properties, e.g.:

```
salary(john) =>
** 15000
address(mary) =>
** [room 5]
```

But if we try to use new copies of the items with which the information is associated, we will not be able to drive the associations, because properties created with newproperty need the input to be "==" to the original, not merely "=" to it. Thus

```
address([john smith]) =>
** undef
salary([marylou walker]) =>
** undef
```

So it does not recognize the new inputs that are merely similar to the old. But if it is given exactly the same, identical input, it will be recognized:

```
salary(mary) =>
** 25000
```

The difference occurs because, although the following is true:

```
mary = [marylou walker] =>
** <true>
```

This one is not:

```
mary == [marylou walker] =>
** <false>
```

The difference between properties created by newmapping and properties created by newproperty is that

```
(a) the former are more general and flexible in their use
(b) the latter are much more efficient in terms of time
```

The reason for (b) is that the use of "==" as a test is generally much faster than the use of "=".

Properties can be composed using <>, given to maplist, etc.

For example, using the mapping occupant, defined above, and maplist, we can find the occupants of rooms:

```
maplist([[room 3][room 4][room 5][room 6]], occupant) =>
** [[john smith] <false> [bob walker] [ted bloggs]]
```

We can compose properties to find spouses of occupants:

```
maplist([[room 3][room 4][room 5][room 6]], occupant <> spouse) =>
** [<false> <false> [mary low walker] <false>]
```

So only room 5 has an occupant whose spouse has been identified.

4.10 Exercise using properties for the rooms database

In chapter 1, an example was developed at length involving the dimensions of rooms, using a list of lists to represent all the information, in this format:

```
vars rooms=
    [[room1 10 12 8]
     [room2 6 11 8]
     [room3 15 11 8]
     [room4 10 12 9]
     [room5 21 11 9]];
```

A disadvantage of this kind of representation is that if the number of items in the list is very large, then discovering the information about particular rooms may require lengthy searches down the list. Properties provide a far more efficient mechanism in such cases.

Try representing all this information using properties. Create a list of names of all the rooms:

```
vars rooms = [ room1 room2 ...];
```

Then for each aspect of a room define a property or mapping from the room to its value for that aspect. E.g. room_length could be a property associating room names (e.g. "room1") with numbers (e.g. 10). Similarly room_breadth, room_height. Define these properties and set up their initial values. Then rewrite all the procedures from chapter 1 that previously searched through the list rooms so that they instead make use of these properties.

Another approach would be to have a single property called room_features, and associate each room with a list of three numbers, as in

```
[6 11 8] -> room_features("room2");
```

Which method is best for which purposes, and why?

4.11 DEFINING PROCEDURES IN POP-11

We have already seen some examples of procedure definitions. Here is a partial specification of the most common format for a procedure definition:

```
define
    then <name of procedure>
    then <formal parameters in parentheses, separated by commas>
    then -> (<output locals>) (if required, separated by commas).
    then semi-colon ;
    then local variable declarations, dlocal expressions,
    then actions to be performed
        (this may include conditionals, loops, additional
        local variable declarations, lexical blocks,
        nested "local" procedure definitions, etc.)
    then
enddefine;
```

Example - a procedure, named DOUBLESUM which takes two numbers, and produces a new number got by doubling the sum of the two.

```
define doublesum (num1, num2);
    2 * (num1 + num2)
enddefine;
```

This definition produces a result, even though it does not use an output local. This is because the procedure calls first '+' (add) then '*' (multiply), and the latter leaves its result on the stack. So since DOUBLESUM does not remove the result of '*', it is left as the result of DOUBLESUM also. (See Chapter 3, and TEACH STACK).

We can make it clearer to ourselves, and readers of our programs if we indicate that a procedure is to produce a result, by using an 'output local' variable in the procedure heading, as we did previously, thus:

```
define doublesum (num1, num2) -> result;
    2 * (num1 + num2)    -> result
enddefine;
```

(NOTE for experts: though clearer this is marginally less efficient, since it has an extra local variable, and does an extra assignment and stack operation. usually clarity is worth more than efficiency!)

Another example - if you already have the POP-11 rc_graphic package loaded, (See TEACH RC_GRAPHIC), which includes a procedure called rc_draw, which takes a number and draws a line, and a procedure called rc_turn, which takes a number in degrees representing the angle to turn, then you can define a procedure named SQUARE which will take a number, and will draw a square of the appropriate size:

```
define square (side);
    repeat 4 times
        rc_draw(side); rc_turn(90);
    endrepeat;
enddefine;
```

This procedure produces no result. So there is no need for an output local.

There are additional forms of procedure definitions, some of which have already been illustrated in previous chapters. E.g. procedures may produce multiple results, like the system procedure explode and the procedure join_two defined in Chapter 2.

It is also possible to define procedures whose names are used as infix operators (like rem, mod, +, * and <>). More on this below.

It is also possible to use the ”define ... enddefine” syntax to specify other things than procedure definitions. This because the form can be extended by using the define_form mechanism, described in the only file HELP * DEFINE_FORM

4.12 Using ”define <name> = ... enddefine”

There is an alternative syntax that is often useful when existing procedures are being used to create new procedures. We have already used it in defining closures, above:

```
define
    then <name of procedure>
    then =
    then an expression evaluating to a procedure
    then
enddefine;
```

Examples: A procedure formed by concatenating two other procedures

```
define pr_sqrt = sqrt <> pr enddefine;

pr_sqrt(16);
4.0
```

A procedure formed by partial application

```
define list_of_doubles =
    maplist(%procedure (x); x + x endprocedure%)
enddefine;

list_of_doubles([1 2 3 4]) =>
** [2 4 6 8]
```

A procedure that's actually a property

```
define father_of =
    newproperty([[tom joe][mary joe][harry dick]],100,false,true)
enddefine;

father_of("mary") =>
** joe
father_of("joe") =>
** <false>
```

4.13 Specifying the syntactic type of a procedure identifier

It is possible to specify that a procedure name is global, constant, a procedure identifier, a lexical constant a lexical variable, or temporarily redefined local to a procedure, using forms like

```
define global <name> ...

define global constant procedure <name> ...

define global vars procedure <name> ...

define lconstant procedure <name> ...

define dlocal procedure <name> ...
```

etc.

In short, the kinds of specifiers that can occur in a variable declaration can also occur in a procedure header.

See HELP VARS, HELP LEXICAL, REF IDENT, REF SYNTAX

4.14 Executing (calling, running, applying) a procedure

An imperative telling Pop-11 to run, or invoke, a procedure, is formed by writing the procedure name, followed by parentheses, with any arguments (if any are needed) between the parentheses. E.g.

```
square(5);
```

I.e.: execute the procedure square with 5 as argument. i.e. the input variable, or formal parameter, used in the procedure definition, namely side, gets the value 5. Similarly, we can invoke doublesum with two arguments, both numbers:

```
doublesum(3,99) =>
** 204
```

I.e.: execute the procedure doublesum, with 3 and 99 as arguments. I.e num1 gets the value 3 and num2 the value 99. The result (204) is printed out by =>

Instead of printing out the result, we can assign it to a variable, e.g. x, thus:

```
doublesum(4, 60) -> x;
```

Alternatively the result of doublesum can be used as input to another procedure, such as square, as in

```
square(doublesum(4, 60));
```

Note that this is equivalent to:

```
4, 60, doublesum(), square();
```

4.15 Procedures with more than one output local

If a procedure is to return one or more results, it can be given OUTPUT LOCALS. We have already seen examples of procedures with one output local variable, e.g. doublesum, above, and several others before that. Sometimes it is convenient to have a procedure produce more than one result.

For example, here is a procedure which takes a list of numbers and produces the sum of all the numbers, and their average, i.e. two results. The two results are left on the stack when the procedure exits.

```
define stats(numlist) ->(average, sum);
    0 -> sum;

    lvars item;
    for item in numlist do
        item + sum -> sum;
    endfor;

    sum/length(numlist) -> average
enddefine;

vars s a;
stats([ 1.0 3.5 5.3 9.62 ]) -> (a, s);
s =>
** 19.42
a =>
** 4.855
```

NB output locals are like ordinary local variables in that the values assigned to them during execution of the procedure are not available after the procedure has finished. E.g. if you try to access the values:

```
sum =>
** <undef sum>
average =>
** <undef average>
```

4.16 Output locals and the stack

Output locals can be assigned to anywhere in the procedure. When the procedure finishes, the values of the output locals are left on the stack. In the example above the values are then taken off the stack after the procedure STATS has finished, and assigned to A and to S.

It may be useful to remember the following "symmetry" between the "input variable" numlist, and the "output" variables, average and sum, in the definition of stats:

```
define stats(numlist) ->(average, sum);
    etc...
```

(a) The input variables are used without any assignment arrow, yet values are ASSIGNED TO THEM from the stack when the procedure starts up.

(b) The output variables appear to be preceded by an assignment arrow, yet their values are copied to the stack when the procedure finishes: the reverse of an assignment.

This is the reverse of what happens when a procedure is invoked with actual arguments:

```
stats(list) -> (a, s);
```

Here, before the procedure runs: (c) a value is taken FROM the variable "list" and copied to the stack,

(d) and when the procedure has finished two items at the top of the stack are moved into the variables a and s.

So (a) is the reverse of (c) and (b) is the reverse of (d). Nevertheless the same syntactic form is used in both the header of a procedure definition and in a later invocation of the procedure. The procedure header gives a sort of "template" for using the procedure.

As a reminder of points made in the previous chapter, the procedure form:

```
define foo(i1, i2) -> (r1, r2);
    <instructions>
enddefine;
```

can be thought of as roughly meaning

```
To do sumsq;
    make i1, i2, r1, r2 lexical local variables for the procedure
            sumsq.
    -> i2;
    -> i1;
    <do the procedure instructions>
    put value of r1 on stack.
    put value of r2 on stack
```

Note that the order in which values for i1 and i2 are taken off the stack is the reverse of the order in which they are put on the stack when the procedure is invoked. This is because it is a stack!

4.17 Precedence and parentheses

There is a syntactic complication in Pop-11, like most programming languages, which we have so far illustrated without commenting on. Some procedures are run by writing their names before the parentheses, with inputs between parentheses as in the imperative 'member(x, list)'.

By contrast some procedures, such as the arithmetic procedures '+', '*', '-', are invoked by writing the name between the inputs, as in

```
3 * 5, 77 + 7
```

etc. These are known as 'infix' operators. They are names of ordinary procedures, but have special syntactic properties to simplify their use. If the same expression contains two such infix operators there may be an ambiguity. For instance: how is 3 + 4 * 5 to be understood? One way is to use parentheses to indicate whether the addition or the multiplication is to be done first:

```
(3 + 4) * 5 =>
** 35

3 + (4 * 5) =>
** 23
```

Parentheses are also the principal way that we get round Pop-11's requirement that everything be written 'linearly'. So the 'ordinary' way of writing a division

```
7.50 + 19.43
────────────
     27
```

becomes

```
(7.50 + 19.43)/27 =>
**  0.99740741
```

Each of the infix operations of Pop-11 has a 'precedence' associated with it. A precedence is a number which determines the order in which the operations are applied. Both '+' and '-' have a precedence of 5. '*', '/', and '//' however have a precedence of 4, indicating that these operations are applied before addition and subtraction where brackets are not used to remove any ambiguity. The concatenator operator '<>' has a precedence of 5.

Because of the precedence rules, in this example:

```
3 + 4 * 5 =>
** 23
```

the multiplication is done first because its precedence is lower than that of '+'.

Generally, infix procedures with the LOWEST numerical precedence are evaluated first. The precedence of '*' is 4 and that of '+' is 5 so the above expression is equivalent to '3 + (4 * 5)'.

Note, unlike many other programming languages Pop-11 uses larger precedence to represent larger scope. I.e operators with smallest precedence bind their arguments "most tightly" to themselves. In many other languages the reverse convention is used.

In the case of operators of equal precedence they are applied from left to right, thus, since "*" and "/" have equal precedence:

```
6 / 2 * 5 =>
** 15
```

not:

```
6 / (2 * 5) =>
** 0.6
```

4.17.1 Precedences of arithmetical operations

The precedence table for arithmetic operations is as shown below.

```
Operation       Precedence
div                2
rem                2
mod                2
**                 3
* / //             4
+ -                5
> < >= <=          6
=                  7
```

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

```
3 - 2.5  ** 2 * 1.5 / 3 =>
```

will be

```
** -0.125
```

The order of evaluation is as follows:

```
Original expression                    3 - 2.5 ** 2 * 1.5 / 3
Apply operations of precedence  3      3 - 6.25 * 1.5  /3
Apply operations of precedence  4      3 - 3.125
Apply operations of precedence  5      -0.125
```

Another way of looking at this is to see that the original expression is transformed into a collection of instructions to Pop-11 that use the stack. I.e.

```
3 - 2.5 ** 2 * 1.5 / 3 =>
```

translates to:

```
3, 2.5, 2, **, 1.5, *, 3, /, - =>
```

Notice how the left right order of operators here corresponds to their numerical precedence.

Pop-11 can be thought of as obeying the above instruction through the following sequence of transformations:

```
3, 2.5, 2, **, 1.5, *, 3, /, - =>
Procedure ** runs, taking two inputs and producing one result
3, 6.25, 1.5, *, 3, /, - =>
Procedure * runs, taking two inputs and producing one result
3, 9.375, 3, /, - =>
Procedure / runs, taking two inputs and producing one result
3, 3.125 - =>
Procedure - runs, taking two inputs and producing one result
** -0.125
```

Parentheses can override operator precedence

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

```
(3 - 2.5) ** 2 * 1.5 / 3 =>
** 0.125
```

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

Parentheses may be nested to any depth, the expressions within inner parentheses being evaluated first.

Operators associate to left, unless precedence is negative

Where parentheses are not present, and the precedences do not uniquely define an order of evaluation, then evaluation proceeds from left to right. I.e. operations associate to the left. So

```
3 / 4 / 5
```

is equivalent to

```
(3 / 4) / 5
```

which denotes 0.15, unlike

```
3 / (4 / 5)
```

which denotes 3.75.

Otherwise infix procedures with a positive precedence are evaluated left to right.

Thus if +++ were defined as an infix procedure of precedence 3, then

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

would be equivalent to

```
(((a +++ b) +++ c) +++ d)
```

However, if the precedence is negative, the operation will associate to the right. So if +++ had precedence -3, then

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

would be equivalent to:

```
(a +++ (b +++ (c +++ d)))
```

For more on precedence see HELP DEFINE

The HELP PRECEDENCE file gives more information on precedences in Pop-11. Precedences in Pop-11 may be positive or negative numbers in the range -12.7 to +12.7.

4.17.2 Using identprops to discover precedence

The procedure called 'identprops' can be used to find out the precedence of an operator. For an ordinary variable the precedence is 0. For example:

```
vars xx, list3;
identprops("xx") =>
** 0
identprops("list3") =>
** 0
```

For an infix operator it will be some other number. For a syntax word or macro (see later) identprops produces the word "syntax" or "macro". If applied to something which has not been declared by the user or the system, it produces the result undef. Thus:

```
identprops("+") =>
** 5

identprops(">") =>
** 6

identprops("if") =>
** syntax

identprops("help") =>
** macro

vars x;
identprops("x") =>
** 0

identprops("xxxxx") =>
** undef
```

4.18 Defining new infix procedures

(Beginners should omit this section.)

Users may define their own new infix operators by specifying the precedence after the word 'define'. E.g. to define an operation of precedence -3.5 /// which divides its first argument by the square of its second, and associates to the right:

```
define -3.5 x /// y;
    x / (y * y)
enddefine;

3 /// 4 =>
** 0.1875
4 /// 5 =>
```

```
** 0.16
3 /// 4 /// 5 =>
** 117.187
(3 /// 4) /// 5 =>
** 0.0075
```

Infix procedures are available for other purposes besides numerical operations. We have already seen that ”<>” is an infix operation which can join two lists or two procedures together, for instance, and ”matches” is an infix operation used for comparing two lists.

```
identprops("matches") =>
** 8
```

The equality symbols ”=” and ”==” are also used as infix operators applicable to arbitrary objects.

4.19 Different types of procedures: normal, infix, macros, syntax

(This section may be omitted by beginners)

Procedures come in four types: normal, infix, macros and syntax (including syntactic operators, described below.).

Strictly speaking, it is the identifiers that come in four types - their values are in all cases simple procedure records.

A normal procedure, with a normal identifier, is one which is invoked by writing its name, followed by an opening parenthesis, its arguments (if any) separated by commas and finally a closing parenthesis, for example:

```
foo(x + 1, y)
```

This applies the procedure which is the value of the variable FOO to the results of evaluating the expressions ‘X + 1’ and ‘Y’.

4.19.1 Infix operators have a ” precedence”

Infix operators were defined and discussed above.

An infix procedure is invoked by writing its name between its arguments. (Typically, infix procedures have two arguments). For example ‘+’ is an infix procedure, thus:

```
3 + 4
```

Some ”infix” operators take a single argument. E.g. we could define ”root” as an infix operator of precedence 2, taking one argument.

```
define 2 root x -> result;
    sqrt(x) -> result;
enddefine;

root 16 =>
** 4.0

root 16 + root 25 =>
** 9.0
```

4.19.2 Macros and syntax words

Macro procedures and syntax procedures can be used to extend the syntax of Pop-11. They are evaluated at compile time (ie when read in by the compiler). They are explained in more detail after further discussion of forms of expression that can be used in defining ordinary procedures.

A syntax word can be identified by applying the procedure identprops to it. If the argument is a syntax word, identprops returns either the word "syntax" or a special kind of word like "'syntax 2'", "'syntax 5'" which includes a number. These are syntactic operators. E.g.

Ordinary syntax words:

```
identprops("if") =>
** syntax

identprops("elseif") =>
** syntax

identprops("endif") =>
** syntax
```

Syntactic operators:

```
identprops("(") =>
** syntax -1

identprops("and") =>
** syntax 9

identprops("or") =>
** syntax 10
```

. Syntactic operators

As illustrated above, there are some syntax words, like "and" and "or" and "(" which are not merely syntax words, but also have an associated precedence. This helps the Pop-11 compiler sort out the complex rules for grouping expressions in a text stream.

4.20 Some important constructs used in defining procedures

4.21 Loops: instructions to do something repeatedly

Several looping constructs are provided.

For compatibility with older systems, wherever the word DO is used in the examples below, the word THEN is also accepted. Similarly most looping constructs will accept CLOSE or ENDDO as the closing bracket, for consistency with POP-2 and earlier versions of Pop-11. It is probably best to ignore these variant forms, as they may be withdrawn later.

The examples of looping constructs below are among the most widely used ones. There are others described in HELP LOOPS and HELP FOR. Moreover, because Pop-11 is an extendable language, users can define new syntax words and macros defining new constructs, as explained below. Examples are the library definitions of "foreach", "forevery", and "switchon", each of which has its own help file. In each case the library definitions can be inspected using the "showlib" command, e.g.

```
ENTER showlib foreach
```

4.21.1 UNTIL <condition> DO <action> ENDUNTIL

This means, check if the <condition> is true, and if not then do the action. Then test again to see if the condition is true. And so on. The condition is tested again each time AFTER <action> is done. For example, to print out all the numbers from 3 to 99 do:

```
vars num;
3 -> num;
until num > 99
do    num =>
  num + 1 -> num;
enduntil;
```

The <action> element of a loop (like a conditional) may be an arbitrarily complex pop-11 imperative, or sequence of imperatives. So to print out the words "THE" "CAT" "SAT" "ON" "THE" "MAT", you could make a list of the words, then keep printing out elements of the list. We use the system procedure TL which, when given a list, returns a new list containing all except the first element of the original. chop one off. until the list is [], thus:

```
vars list;
[the cat sat on the mat ]  -> list;
until list == []
do
    list(1) =>                   ;;; print first element
    tl(list) -> list;            ;;; prepare for next
enduntil;
```

NB the loop will not be terminated immediately if the condition becomes TRUE in the middle of executing the action. Note also that the condition is always tested at least once BEFORE anything else is done.

4.21.2 REPEAT <number> TIMES <action> ENDREPEAT

After the word REPEAT you can have a number, e.g. 4, or a variable whose value is a number, e.g. REPEAT N TIMES... or a more complex expression which calculates a number,

```
  e.g. repeat 66 + 53 times....   endrepeat;
```

The <action> will be done the specified number of times.

Example: to print out 10 blank lines, do

```
repeat 10 times pr(newline) endrepeat;
```

Make a list of 10 randomly generate numbers between 1 and 20

```
[% repeat 10 times random(20) endrepeat %] =>
** [12 15 19 18 9 5 4 11 17 19]
```

See also the definition of procedure SQUARE, above.

For indefinite iteration do

```
REPEAT <action> ENDREPEAT;
```

e.g.

```
repeat
    [you are wonderful] =>
endrepeat;
```

You will need to interrupt (e.g. with CTRL-C).

(See HELP LOOPS for more on interrupting loops)

4.21.3 WHILE <condition> DO <action> ENDWHILE

This means, test the condition. If it is true, then do the action. Then test the condition again. And so on. This is similar to UNTIL, except that WHILE does the action each time the condition is found to be TRUE whereas UNTIL does the action each time the condition is found to be FALSE. In both cases the condition is tested first.

E.g. to find the first integer whose square is greater than 1000, you could do:

```
vars num;
1 -> num;
while num * num < 1000 do
    num + 1 -> num
endwhile;
```

The value of num and its square can now be printed out:

```
num =>
** 32
num * num =>
** 1024
```

4.21.4 'FOR ... ENDFOR' loops

We have seen examples of the use of the 'for ... endfor' construction, to iterate over lists. In fact there are several different formats.

The following is a very general for loop, which can be used in many contexts, although a more specific version will often be preferred.

```
FOR <initiate> STEP <step> TILL <condition> DO
    <action>
 ENDFOR;
```

This is equivalent to:

```
<initiate>;
UNTIL <condition> DO <action>; <step> ENDUNTIL;
```

In other words, do the initialisation, then, until the condition evaluates to TRUE, repeatedly do the action followed by the step.

For example, to print out all the numbers from LO to HI, separated by spaces:

```
for   lo-> x
step  x+1 -> x
till  x > hi
do    spr(x);
endfor;
```

N.B. The ‘step’ is not done until after the ‘action’.

Suppose FOO is a procedure of two arguments, a word and a number. If you are given a list of words and a list of numbers and wish to apply FOO to the first element of each, then the second element of each, etc., until either there are no more words, or no more numbers, you could do something like:

```
vars w, n;
for   words -> w; numbers -> n;
step  tl(w) -> w; tl(n) -> n;
till  w = [] or n = []
do    foo(hd(w), hd(n));
endfor;
```

. FOR X IN LIST DO <action> ENDFOR;

X will take on as its value the first element of LIST, then the second element, then the third, etc. Each time the <action> is performed the latest value of X will be used. E.g. printing out the squares of a list of numbers

```
for num in numbers do
    pr('\nThe square of: '); pr(num); pr(' is: '); pr(num*num);
endfor;
```

(Note: this uses strings. ‘\n’ in a string causes a new line to be printed.)

It is also possible to iterate over several lists at the same time, e.g.

```
for x1, x2, x3, in list1, list2, list3 do
    <something involving x1, x2, x3>
endfor;
```

. FOR L ON LIST DO <action> ENDFOR;

Here, the first time the <action> is done L will refer to the whole LIST. The second time it will refer to the TAIL of the list (i.e. a list of all but the first element). The next time a still shorter list, and so on. E.g.

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

Again this can be done with several lists at once.

. FOR X FROM <number> BY <number> TO <number> DO

FOR X FROM <number> BY <number> TO <number> DO <action> ENDFOR;

e.g. to print out numbers from 2 to 20 going up in steps of 7

```
for x from 2 by 7 to 20 do  x => endfor;
** 2
** 9
** 16
```

or going down:

```
for x from 50 by -12.5 to -20 do x => endfor;
** 50
** 37.5
** 25.0
** 12.5
** 0.0
** -12.5
```

the 'FROM <number>' and 'BY <number>' portions can be omitted. The starting value defaults to 1, as does the increment.

```
vars x;
for x to 6 do pr(x); pr(space) endfor;
1 2 3 4 5 6
```

4.21.5 There are several forms of conditionals.

All the <conditions> in what follows should be expressions which evaluate to TRUE or FALSE.

```
(1) IF <condition> THEN <action> ENDIF;
```

```
(2) IF <condition> THEN <action1> ELSE <action2> ENDIF;
```

```
     if the <condition> is true, then <action1> will be executed,
     otherwise <action2> will be executed.

(3) IF <condition1> THEN <action1>
    ELSEIF <condition2> THEN <action2>
    ELSEIF <condition3> THEN <action3>
    ELSEUNLESS <condition4> THEN <action4>
    .............
    .............
    ELSE <default action>
    ENDIF;
```

This ‘multi-branch’ conditional says:

```
    try <condition1> then <condition2> etc in turn until an ELSEIF
    condition is found which is TRUE, or an ELSEUNLESS condition is
    found which is FALSE. If either is found, execute the
    corresponding <action>.  If none of the conditions comes out
    TRUE then do the thing following ELSE, i.e. <default action>.
```

Note that in an imperative you do not have to have the ELSE <default action> bit. You must, however, have it in an expression intended to denote, something, for then it should denote something under all conditions. You can include as many ELSEIF clauses as you like.

```
  (4)   UNLESS <condition> THEN  <action> ENDUNLESS;
       this is equivalent to:
       IF NOT(<condition>) THEN <action> ENDIF;
```

UNLESS can also have ELSEUNLESS and ELSEIF and ELSE clauses.

Note: the words NOT, AND and OR are available for use in formulating complex conditions. E.g.

```
    IF <condition1> OR (<condition2> AND NOT(<condition3>))
```

. Examples of conditionals

To test whether the value of X is bigger than the value of Y, and print out the bigger value do:

```
    if    x > y
    then  x =>
    else  y =>
    endif;
```

Compare:

```
    if       x > y
    then     x =>
    elseif   y > x
    then     y =>
```

```
else     "same" =>
endif;

if    2 < x and x < 6
then  true
else  false
endif =>
```

Note that the last example is exactly equivalent to:

```
2 < x and x < 6  =>
```

If LIST1 and LIST2 are two lists and you want to print out the one which is shorter you could do:

```
if    length(list1) < length(list2)
then  list1
else  list2
endif =>
```

If N and M are two numbers, and you wish to assign the bigger one to the variable MAX then do:

```
if    m > n
then  m
else  n
endif -> max;
```

4.22 Using conditionals to jump out of, or re-start a loop.

Several of the looping constructs have their own natural terminating conditions. E.g. the ”until” loop form stops as soon as its condition evaluates to TRUE.

However, it is sometimes useful to have additional tests for ”unusual” exits within the ”body” of a loop. The syntax words ”quitloop”, ”quitif”, and ”quitunless” provide this. They can also be used with an integer specifying which loop to jump out of. The default is 1, meaning jump out of the smallest enclosing loop, whereas quitloop(3) means jump out of the third enclosing loop. These expressions are normally used with conditional tests, e.g.

```
if list == [] then quitloop(2) endif;
```

Such a test can be abbreviated thus:

```
quitif(list == [])(2);
```

Similarly there is an analogue of ”unless”, i.e.

```
quitunless(x < y);
```

is equivalent to

```
unless x < y then quitloop() endunless;
```

4.22.1 Re-starting a loop using ”nextloop”

Just as it is sometimes useful prematurely to terminate action so is it sometimes useful also to restart a loop prematurely. For example if a loop's body includes a sequence of instructions, then it may be possible to use a time-saving instruction of this form:

```
if x > y then nextloop(3) endif;
```

This means that if x > y ever evaluates to true, then the third enclosing loop should be re-started. This does not mean that if it is a ”for” loop the loop variable should be reset to its initial value. It is equivalent to jumping to just before the end of the loop, e.g. just before ”endrepeat”, ”endwhile”, etc., unlike ”quitif” and its friends, which jump to just after the end.

4.22.2 Other abnormal exit commands

In addition to the above commands for quiting or re-starting a loop there are several procedures that can be used to exit from the current procedure, or some other procedure that is 'higher up' the control chain. In particular

”return” can be used to terminate execution of a procedure.

Other abnormal exits.

The following syntax words can be used to deal with more complex cases:

chain, chainto, breakto, catch, jumpout

See HELP CONTROL for more information.

4.22.3 Switch statements

Pop-11 provides a command GO_ON which can be used thus

```
GO_ON <numerical expression> TO <label1> <label2> <label3> ...<labeln>;
```

The expression must evaluate to a number in the range 1 to N, if N is the number of labels. The labels must be repeated elsewhere in the procedure, followed by colons. For instance, the following will translate numerals:

```
define trans(num);
  go_on num to la lb lc ld;
  la: "one" ; return;
  lb: "two" ; return;
  lc: "three" ; return;
  ld: "toobig";
enddefine;
trans(2) =>
** two
```

A GO_ON instruction may end with ELSE <default label>; See HELP GO_ON.

4.23 Other syntactic constructs

The preceding sections merely introduce a subset of the constructs available in constructing procedure definitions. Additional facilities are provided for different kinds of loops, and for transferring control between procedure activations (e.g. using exitto). Moreover the Pop-11 ”process” mechanism adds additional facilities not found in many languages.

Further reading can be found in later chapters, and in the following online help files

```
HELP CONTROL
HELP LOOPS
HELP PROCESS

REF SYNTAX
REF POPSYNTAX
```

4.24 Tracing procedures

To trace some procedures, type TRACE, followed by the names of the procedures. example:

```
trace square doublesum;
```

will cause the two procedures to be traced whenever they are executed. UNTRACE is used to undo tracing, e.g.

```
untrace square doublesum;
```

For more on this see TEACH TRACE. HELP TRACE gives a summary and explains the tracing mechanisms in more detail.

4.25 Defining macros and syntax words to extend the language.

4.25.1 Macros

Macro procedures can be used to extend the syntax of Pop-11. They are evaluated at compile time (ie when read in by the compiler).

A macro typically reads in items (e.g. words, strings, numbers) from the current program text input stream and then creates new text items to replace those read in. The original items need not have been legal Pop-11. The replaced items will be legal Pop-11, unless they include macros, which can cause further changes.

An example would be a macro procedure to read in two variable names and produce instructions to swap their values. It transforms illegal Pop-11 expressions like,

```
swap x y;
swap list1 list2;
```

into legal Pop-11 like

```
x, y -> x -> y;
list1, list2 -> list1 -> list2;
```

which are equivalent to

```
x, y -> (y, x);
list1, list2 -> (list2, list1);
```

as explained in Chapter 2.

This is how the macro ”swap” could be defined.

```
define macro swap var1 var2;
    ;;; Transform "swap x y" into "x, y -> x, -> y"

    var1, ",", var2, "->", var1, "->", var2

enddefine;
```

This could be used as follows:

```
vars x = 99, y = 66;
x, y =>
** 99 66

swap x y;
x, y =>
** 66 99
```

The word ”nonmac” can precede a macro name to prevent its evaluation, as in this example

```
nonmac swap =>
** <procedure swap>
```

For more details and further examples see HELP MACRO.

Macros can be recursive

Macros can invoke other macros (or themselves) recursively. If the text put on the input stream by one macro includes another macro, the second macro's procedure will be invoked when it is read in. This can happen several times, causing the program text to be rearranged several times, until all that is left is legal Pop-11 syntax words and other text items, whereupon the program will simply be translated into machine code, i.e. compiled.

Unlike some languages with macros, Pop-11 macros can operate after some user programs have been compiled, and may therefore invoke those programs. (In fact a macro is just a user program.) This means that users can write macros that do very complex manipulations of the text stream, as in Lisp. An example of a fairly complex macro is ”switchon” defined in a Pop-11 library program LIB SWITCHON, and described in HELP SWITCHON.

Macro arguments are text items, not expressions

NOTE: The arguments of a macro are bound to the individual following TEXT ITEMS, not the two following EXPRESSIONS: so if the macro swap has two arguments then the expression

```
'swap x y' is correct
```

but

```
'swap hd(x) hd(y)' is wrong
```

because the latter would give the next two text items "hd" and "(" as arguments to swap. So it would be transformed into:

```
hd, (, -> hd, -> (, x) hd(y)
```

which will cause a mishap to occur, in this case:

```
;;; MISHAP - COMPILING ASSIGNMENT TO PROTECTED IDENTIFIER
;;; INVOLVING:  hd
```

It would be possible to write a macro that coped with expressions like

```
swap hd(x) hd(y)
```

by translating them into

```
hd(x), hd(y) -> hd(x) -> hd(y)
```

This would require making use of the procedure pop11_comp_expr and the variable pop_syntax_only. However this will not be explained here, as there is a more powerful facility already available, explained in the next section.

See REF PROGLIST, REF POPCOMPILE, REF VMCODE for full details.

4.25.2 Using "define :inline" to define the SWAP macro

It is possible to use a facility described in HELP INLINE to produce the required version of swap, which accepts arbitrary expressions. The following defines a new macro SWAP

```
define :inline SWAP(expr1, expr2);

    expr1, expr2 -> (expr2, expr1)

enddefine;

;;; check that it is a macro

identprops("SWAP") =>
** macro
```

It can be invoked in the format

```
SWAP(<expresion1>, <expression2>)
```

and will cause the values of the two expressions to be exchanged.

We can test this as follows. First the easy case, where the expressions are individual identifiers.

```
vars x = 66, y = 88;

x,y =>
** 66 88

SWAP(x, y);
x,y =>
** 88 66
```

The format can also be used when the expressions are complex, e.g.

```
vars list = [cat dog mouse];

SWAP(list(1), list(3));

list =>
** [mouse dog cat]
```

Note that SWAP thus defined is a macro, not a procedure. It is run at compile time, i.e. while procedures are being compiled, not at run time. Unlike ordinary procedures, it does not evaluate its two arguments and then do something with the values. Rather, during the compilation process it creates new code using the expressions to which it has been applied, not the values. This can be seen by tracing the macro

```
trace SWAP;
SWAP(list(1), list(3));
> SWAP
< SWAP list ( 1 ) , list ( 3 ) -> ( list ( 3 ) , list ( 1 ) )
```

Because the "define :inline" form produces macros, not ordinary procedures, users are STRONGLY advised to use names that give a reminder that something unusal is involved, e.g. by choosing upper case names, as in the example above.

For more information see HELP INLINE.

4.25.3 Note on efficiency of macros

Often the use of macros produces "inline" code instead of procedure calls and can therefore be very efficient. However, in some cases the use of macros defined as above can be very wasteful as the repetition of complex expressions may cause the same thing to be recomputed unnecessarily.

For example

```
vars list1 = [cat mouse dog flea], list2 = [red orange yellow];

SWAP( hd(tl(tl(list1))) , hd(tl(tl(list2))) );

list1, list2 =>
** [cat mouse yellow flea] [red orange dog]
```

To achieve that exchange, the use of SWAP produces the equivalent of the following Pop-11:

```
hd(tl(tl(list1))), hd(tl(tl(list2)))
    -> ( hd(tl(tl(list2))), hd(tl(tl(list1))) );
```

I.e. tl(tl(list1)) is computed twice, and tl(tl(list2)) is computed twice, once before and once after the assignment arrow. It would have been more efficient to store their values in two variables, temp1, and temp2, and then produced code to do:

```
hd(temp1), hd(temp2) -> ( hd(temp2), hd(temp1) );
```

Note, however that it would not do to replace the above with these two expressions:

```
hd(tl(tl(list1))) -> temp1; hd(tl(tl(list2))) -> temp2;

temp1, temp2 -> (temp2, temp1);
```

Why not? In order to answer that question, the reader needs a deep understanding of how expressions are evaluated, how complex structures are referred to, and what the relationship is between a variable and its value.

If you try all the above you will see that all it does is assign two values to temp1 and temp2 and then swap their values, without altering the contents the lists, list1 and list2. For that it is essential to run the updater of the procedure hd, which is done by the expression after the assignment arrow here:

```
hd(temp1), hd(temp2) -> ( hd(temp2), hd(temp1) );
```

4.25.4 Syntax procedures

Syntax procedures provide even more powerful facilities for extending the syntax of Pop-11.

Unlike macros they do not take arguments. Like macros, they too are executed at compile time. System syntax words, such as ”if”, ”define”, ”until”, ”for”, etc. correspond to syntax procedures: they typically read in some or all of a Pop-11 expression and tell the compiler how to understand it.

There are other syntax words that do nothing except flag the end of a complex syntactic construction. Examples are ”)”, ”endif”, ”enddefine” and other closing brackets. Some syntax words, like ”then”, ”do”, ”elseif” are used to indicate intermediate expression boundaries in complex expressions, as in

```
if <condition> then <action> elseif <condition> then <action> endif
```

The difference between syntax words and macros is quite subtle. Both macros and syntax words read items in from the program input text stream called ”proglist”, a dynamic list, defined in REF PROGLIST. A macro will rearrange text items, put them back on the front of proglist, then let the compiler continue.

A syntax word will typically invoke a procedure that reads in items from proglist, and then gives instructions to the compiler to create machine code corresponding to the text read in. Strictly speaking, the syntax procedure ”plants code” for the Poplog virtual machine PVM (described briefly in Chapter 2) rather than the physical machine on which Pop-11 is running. The virtual machine instructions are then translated to real machine code by a VM compiler.

This means that a MACRO definition can extend Pop-11 syntax only by introducing new constructs that translate into existing legal Pop-11, whereas a SYNTAX word can introduce any syntactic extension that is capable of being translated into PVM instructions. Since the Poplog virtual machine is extremely general, syntax words can extend Pop-11 in enormously varied ways. (For example, this is how languages like ML and Prolog are defined in Pop-11, languages which could not be translated into Pop-11).

This is both a strength of the language, and a problem: it means on the one hand that extensions relevant to particular applications are relatively easily produced, in order to make programs easier to develop and maintain. But it also means that the ways in which Pop-11 programs can vary syntactically are so great that it is difficult to produce tools for manipulating them automatically, e.g. cross reference tools, program validation tools etc. However, to some extent the power of the language is such that users can provide their own tools to suit their own extensions.

4.25.5 Example: defining a syntax word: loop

If we did not already have the construct ”repeat <actions> endrepeat” we could define a new syntax format ”loop <actions> endloop”, as follows.

```
;;; first define the closing bracket to be a syntax word
global constant syntax endloop;

;;; Now define the main syntax word

define syntax loop;
    ;;; First define labels for the beginnings and ends of the loop.
    ;;; We need the end label for use with "quitloop"

    lvars Lab, Endlab;
    sysNEW_LABEL() -> Lab;      pop11_loop_start(Lab);
    sysNEW_LABEL() -> Endlab;   pop11_loop_end(Endlab);

    ;;; Plant the first label
    sysLABEL(Lab);
    ;;; read and compile up to "endloop"
    pop11_comp_stmnt_seq_to([endloop]) ->;
    ;;; Now code to jump back to the beginning of the loop
    sysGOTO(Lab);
    ;;; Now plant the send label, for clean exits from the loop
    sysLABEL(Endlab);
enddefine;

;;; test it.

vars x = 3;
loop
    x =>
    x - 1 -> x;
    if x < 0 then quitloop() endif;
endloop;
** 3
** 2
** 1
```

```
** 0
```

More detailed information is provided in the online files: HELP SYNTAX and REF SYNTAX.

We now give examples of some of the more commonly used forms of syntax.

5 CHAPTER.5: NUMERICAL AND LOGICAL FACILITIES IN POP-11

This chapter outlines the facilities for manipulating numbers of various sorts in Pop-11. More details can be found in HELP MATH, REF NUMBERS REF ITEMISE details the conventions for typing in representations of numbers in Pop-11 programs, which are also summarised below.

Pop-11 provides a standard collection of mathematical facilities, including powerful tools for doing very high precision calculations using ”bigintegers” or ”ratios”. These are not provided in many programming languages, partly because they lead to complex store management problems. These problems are solved by the use of an automatic garbage collector in Pop-11. The same is true of modern lisp systems and some other languages, e.g. ML and some implementations of Prolog. (All the Poplog languages share these facilities.)

5.1 Mathematical knowledge presupposed

It is assumed in what follows that the reader knows enough mathematics to understand the distinction between

1. The integers:

```
        0, 1, -1, 2, -2, 3, -3, etc.
```

2. The ratios (with numerators and denominators):

```
        1/2, -1/2, 1/3, -1/3, 2/3, -2/3, 3/4 etc.
```

3. Floating point numbers:

```
        0.135, 1.35, 13.5 -0.135, -1.35, etc.
```

4. Complex numbers with real and imaginary parts, such as the square root of -1.

It is also assumed that readers know that floating point numbers can be represented as manitissa and exponent, and will therefore understand what is printed out by the following, where each number in the first row is represented as <mantissa>e<exponent>:

```
   1.234e4, 1.234e3, 1.234e2, 1.234e1, 1.234e0, 1.234e-1, 1.234e-2 =>
** 12340.0  1234.0   123.4    12.34    1.234    0.1234    0.01234
```

(Extra spaces have been inserted in the printout, to simplify comparisons.)

Readers who do not know all about these topics may nevertheless be able to pick up the information from the explanations given below. Alternatively they can omit sections on unfamiliar kinds of numbers, e.g. complex numbers.

Note: in Pop-11 complex numbers have real and imaginary parts that themselves can be integral (i.e. positive or negative integers) or ratios, or floating point numbers, or any combination of these.

5.2 The machine representation of numbers in Poplog

In Poplog there is a major division between two kinds of data items: simple items, and compound items. Simple items are directly represented in the machine as bit patterns. Compound items are represented indirectly as bit patterns that are pointers, or addresses, of data structures that contain the actual information.

The simple items can all be completely represented in a single "machine word", which is normally 32 bits, but may be larger on some machine architectures. Compound items are larger structures, and each includes information about what TYPE of item it is, so that run time checks can be made. (See also Chapter 2 and REF DATA). This type information is represented by another pointer, to a KEY for the type of data in question.

5.2.1 Simple items in Pop-11: integers and decimals

There are only two sorts of simple items in Poplog, integers and decimals. Both are represented within a single machine word. However not all the bits of the word can be used for this purpose, as two bits in every word are reserved for specifying whether the item is a compound item, an integer, or a decimal. In a 32 bit machine that leaves 30 bits for the integer or decimal, of which 1 will be for the sign.

This "direct" representation by bit patterns means that operations on integers or decimals do not create new Poplog datastructures, since they merely involve operations on bit patterns in machine registers.

5.2.2 Compound items in Pop-11

By contrast, operations that create compound items, such as bigintegers, ddecimals, ratios and complexes will require additional records to be allocated in the storage "heap", and their use can therefore trigger garbage collections every now and again. (For more information see HELP EFFICIENCY.)

Besides the numerical compound items just mentioned, all other Pop-11 data types besides integers and decimals are compound items, including lists, words, strings, procedures, properties, arrays, etc.

5.2.3 Fixed precision and indefinite precision arithmetic

Another distinction that is important is between those items that have fixed size in memory and those that can be as big as needed. Integers, decimals, ddecimals, and complexes composed only of integers, decimals and ddecimals all have fixed sizes. So typically a ddecimal will occupy three words, one for the key which identifies the object as a ddecimal, and two for the actual number.

By contrast bigintegers and ratios, and complexes composed of bigintegers and ratios provide "infinite" precision arithmetic and can take up as much store as is available. So the only limit to the size of a biginteger is the (virtual) memory available in the machine. To illustrate this try the following:

```
define factorial(n);
    if n == 0 then 1
    else n * factorial(n - 1)
    endif
enddefine;

factorial(3) =>
** 6
factorial(30) =>
** 265252859812191058636308480000000

factorial(100) =>
** 9332621544394415268169923885626670049071596826438162146859296389
    95217599993229915608941463976156518286253697920827223758251185
```

```
    2109168640000000000000000000000000

factorial(1000) =>
```

The number printed out by the last example will take about 37 lines (depending on your screendwidth).

Similarly ratios can represent small numbers very accurately, e.g.

```
237/factorial(37) =>
** 79_/458791769707544834877199319386030080000000

1/(2**150 - 1) =>
** 1_/1427247692705959881058285969449495136382746623
```

5.2.4 Packed integer or decimal numbers

Pop-11 provides facilities for creating records and vectors that include numbers represented compactly as bit patterns. Examples are

bitvectors, (described in HELP BITVECTORS,)

strings (which contain packed 8 bit integers, described in REF STRINGS)

shortvecs (which contain packed 16 bit integers, described in REF INTVEC)

intvecs (which contain packed 32 bit integers, described REF INTVEC)

NOTE: The numbers of bits per field in shortvecs and intvecs is implementation dependent.

Using the "defclass" syntax construct users can define new record types and vector types containing packed integers of various field sizes.

Because Pop-11 can tell the types of all these integers from their occupancy of fields of particular types, it does not need to use 2 bits per integer to recognize them. However, extraction or insertion of these field values, e.g. assigning to or from a variable, necessitates conversion to and from the standard Poplog representation. This is performed automatically by the corresponding record or vector access and update procedures.

It is also possible to have vectors or records with packed floating point numbers, and similar remarks apply to them.

5.3 Types of numbers in Pop-11

We can now list the kinds of numbers that can be created and manipulated in Pop-11, namely integers, bigintegers, decimals, ddecimals, ratios and complexes. Examples of each follow, preceded by the corresponding "dataword", and an indication of whether they are simple or compound.

5.3.1 decimals and ddecimals

Decimals are single-precision floating point numbers, whereas ddecimals are double-precision floating point numbers. They can be typed in directly or produced as the result of applying procedures.

```
DATAWORD              EXAMPLES
```

```
decimal (simple):
        Input:  66.0s0  2.54s3  2.54s-3  sqrt(4)
        Output: 66.0    2540.0  0.00254  2.0

ddecimal (compound):
        66.0  -33.0  77.35  9999.532  -6666.0
        66e0  2.54e5  2.54d3 2.54d-3
```

The symbols ”e”, ”s” and ”d” are used in a representation of floating point numbers consisting of mantissa and exponent. The symbol is preceded by the mantissa and followed by the exponent. So 2.54e5 is the same as 2.54*(10 ** 5), where ** is the ”raise to the power” operator.

The notation using ”s” followed by an exponent is required to force a number typed in to be a single precision decimal. This is because the ”constant” decimal numbers typed in are interpreted by the Pop-11 itemiser as double precision decimals, i.e. ddecimals. So, 66.0 will be read in as a ddecimal, as will 66.0e0, 66.0d0, whereas 66.0s0 will be a ddecimal, with the same numerical value.

```
dataword(3.5s0) =>
** decimal

dataword(3.5d0) =>
** ddecimal

dataword(3.5e0) =>
** ddecimal
```

Whether floating point results of arithmetical computations are single or double precision is controlled by the global variable popdprecision. If it is false (which is the default value) then the results are decimals. If it is true, then as long as at least one ddecimal or non decimal number is involved initially, floating point computations will produce double-precision ddecimal results. This is described more fully below.

5.3.2 Integers, bigintegers, ratios and complex numbers

```
DATAWORD            EXAMPLES
integer (simple):
        66  -33  99999 -12348888

biginteger (compound):
        12345678900980980911   2**40   -99999999999999999999999

ratio (compound):
        3_/4  12345_/54321 -33_/44

complex (compound):
        sqrt(-1)  33_+:44   55.3_-:22.5   3_/4_+:5_/8

The third example is a complex number with floating point (ddecimal)
real and imaginary parts (55.3 and -22.5, respectively). The last
example is a complex number with ratios as its real and imaginary
parts (3/4 and 5/8 respectively).
```

5.4 Forming numerical expressions

Any of the above forms of numerical expression can be used where Pop-11 requires an expression, e.g. to assign values to variables, or to give arguments to procedures requiring numerical arguments. Mathematical procedures provided by Pop-11 are listed below, including the standard arithmetical operators (+, -, *, /), trigonometrical functions (sin, cos, tan) and others.

Starting from expressions denoting numbers we can create more complicated expressions denoting numbers. E.g.

```
33 + 44       denotes the number 7

sqrt(9)       denotes the number 3.0

max(4, 77)    denotes the number 77.
```

Arithmetic expressions can be embedded in others. For example the expression

```
    tan(23 + 22)
```

contains sub-expressions 23, 22, and 23 + 22.

Finally

```
max(min(66, 33), min(999, 9))
```

denotes the number 33, as does this:

```
max( min(66-5, 33), min(999, 9+9) ) =>
** 33
```

Exercise: How many different expressions does the last example contain?

5.4.1 Using a radix other than 10 to represent numbers on input

Pop-11 will read in binary numbers if they are preceded by ‘2:’

```
2:111 =>
** 7
```

Similarly octal numbers may be preceded by ‘8:’

```
8:111 =>
** 73
```

This sort of prefix can be used for non-integer numbers too:

```
8:11.11 =>
** 9.140625

10:55.55 =>
** 55.55
```

The integer base used for reading in non-decimal numbers must be in the range 2-36. If the base is 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.

Note: the same internal representation is used, no matter how the number is read in. So how the number prints out is not affected. E.g.

```
2:111, 7 =>
** 7   7

2:111 == 7 =>
** <true>
```

5.4.2 Using pop_pr_radix to control radix used in printing

How numbers are printed is controlled using pop_pr_radix, which defaults to 10. By making it 16 numbers can be printed in hexadecimal, for instance:

```
16 -> pop_pr_radix;
16 =>
** 10
15 =>
** F
```

You can define a procedure to print numbers in binary form thus:

```
define pr_binary(num);
    dlocal pop_pr_radix = 2;
    pr(num)
enddefine;

vars x;
for x from 1 to 15 do pr_binary(x); pr(space) endfor;
1 10 11 100 101 110 111 1000 1001 1010 1011 1100 1101 1110 1111
```

pop_pr_radix, pop_pr_places and pop_pr_exponent

The printing of decimal (floating point) numbers is also controlled by pop_pr_radix.

```
16 -> pop_pr_radix;
15.55 =>
** F.8CCCCD
```

Compare

```
pr_binary(15.55);
1111.100011
```

The number of decimal places shown is controlled by pop_pr_places, which defaults to 6.

```
10 -> pop_pr_radix;
sqrt(2) =>
** 1.41421

2 -> pop_pr_places;
sqrt(2) =>
** 1.41

6 -> pop_pr_places;
sqrt(2) =>
** 1.41421
```

A value of 0 causes decimal numbers to be printed as integers.

If pop_pr_exponent is made true (it defaults to false) then exponent form is used for printing decimals and ddecimals:

```
true -> pop_pr_exponent;
sqrt(20000) =>
** 1.41421e+2
false -> pop_pr_exponent;
sqrt(20000) =>
** 141.421
```

As explained above, the exponent notation can also be used for reading in decimals:

```
1.41421e+2 =>
** 141.421
```

5.5 Additional facilities for printing numbers

```
prnum(NUM, INT_PLACES, FRAC_PLACES)
    This takes any non-complex number NUM and prints it in the form
        <pre>.<post>
    INT_PLACES is an integer specifying the number of character
    positions that <pre> should occupy, including leading spaces, and
    minus sign, if NUM is negative. FRAC_PLACES specifies the number of
    positions the <post> part should occupy, excluding the ".", and
    including trailing zeros if necessary. E.g.

    pr("|"); prnum(-3.52, 4, 5);pr("|");
    |  -3.5200|

radix_apply(ITEM_1, ITEM_2..., ITEM_N, P, RADIX)
    Runs the procedure P, which should take N arguments, ITEM_1 to
    ITEM_N, in an environment in which pop_pr_radix has the integer
```

```
value RADIX. So any printing done by P uses that value of
pop_pr_radix. E.g.

radix_apply(45,5,5,prnum,16);
   2D.0000
```

More sophisticated general purpose printing procedures are defined in REF PRINT, including:

```
printf, nprintf, pr_field, format_print
```

5.6 Maximum and minimum integer sizes: pop_max_int, pop_min_int

(Almost) all the arithmetic operators in Poplog can be applied to all the different kinds of numbers and will produce appropriate kinds of results. So for example if you add or multiply integers together you'll get integers until the result is too large to fit into the maximum space allowed for integers, and in that case the result will be a biginteger. Where the transition occurs will depend on the word size of your machine. The relevant information can be obtained by loading the INT_PARAMETERS library:

```
lib int_parameters
```

This makes available two variables: pop_max_int, which is the largest (i.e. most positive) integer value and pop_min_int, which is the smallest (i.e. most negative) integer value. On a typical 32 bit machine these give:

```
pop_max_int, pop_min_int =>
** 536870911 -536870912
```

So any integer operation producing a number bigger than the first or smaller (more negative) than the second will produce a biginteger, not an integer.

```
isinteger(pop_min_int) =>
** <true>
isinteger(pop_min_int - 1) =>
** <false>
isbiginteger(pop_min_int - 1) =>
** <true>
```

5.7 The representation of floating point numbers: lib float_parameters

In order to obtain information about decimals and ddecimals in your version of Pop-11 use the following library

```
lib float_parameters
```

This provides the following implementation dependent constants, all of which are explained fully in REF NUMBERS:

```
pop_most_positive_decimal
pop_least_positive_decimal
pop_least_negative_decimal
```

```
pop_most_negative_decimal
pop_most_positive_ddecimal
pop_least_positive_ddecimal
pop_least_negative_ddecimal
pop_most_negative_ddecimal

pop_plus_epsilon_decimal
pop_plus_epsilon_ddecimal
pop_minus_epsilon_decimal
pop_minus_epsilon_ddecimal
```

Additional information about the representation of individual floating point numbers can be obtained using facilities defined below.

5.8 Basic arithmetical facilities

There are several procedures for manipulating numbers of all types. The most commonly used ones are represented by 'infix' identifiers. That is they can be used to create expressions without using parentheses, e.g.

```
3 + 5,  77 * 9,
```

although it is legal to write +(3, 5) or *(77, 9) instead if you prefer.

The operations listed below are available for constructing arithmetic expressions. They all take numbers of every variety in all combinations. I.e. they are "overloaded" or "polymorphic" operators. (The exception is "mod" which cannot be applied to complex numbers.)

When the result is a floating point number, whether it is represented as a decimal (single precision) or ddecimal (double precision) will depend on whether the global variable popdprecision is false or not, as explained below.

In general the rules in Pop-11 for deciding on the type of result to be produced by a mathematical procedure conform to standard mathematical expectations. For example, a decimal divided by an integer will give a decimal, unless popdprecision is true, in which case it will give a ddecimal. Division of one integer by another may produce a surprise in the form of a ratio, as described below.

5.9 Arithmetical operators

```
OPERATOR PRECEDENCE DESCRIPTION
    +       5       add two numbers
    -       5       subtract two numbers, or negate one number.
        (The syntactic context is used to determine whether this is
         a binary subtraction operator or a unary negation operator)
    *       4       multiply two numbers
    **      3       exponentiation:
                        e.g. A ** 3 means A*A*A i.e. A cubed.
    /       4       divide first number by second (See note below)
    //      4       divide first number by second, produce
                        remainder and quotient.

 X div Y    2       returns the number of times Y goes into X
```

```
 X rem Y     2        returns the remainder on dividing X by Y.
 X mod Y     2        returns the remainder on dividing X by Y.
        mod, unlike rem, requires both numbers to be real, i.e.
        non-complex. It always returns a result with the same sign as Y
```

Examples of arithmetical expressions

Here are some expressions formed using arithmetic operators:

```
3 + 5 * 2 =>
** 13

(3 + 5) * 2 =>
** 16
```

The expression:

```
(X + Y) * (99 - Z/3)
```

has many sub-expressions, e.g. X, Y, 99, Z, 3, X + Y, Z/3, etc.

Notice how parentheses can affect the order in which operations are applied.

Illustrating the use of // (which produces two results)

The operator // is unusual in that it produces two results. It takes two integers, and produces two integers, a remainder and a divisor, as in:

```
10 // 3 =>
** 1 3

23 // 10 =>
** 3 2
```

Thus one can do things like

```
vars remainder, quotient;  223 // 10 -> (remainder, quotient);
remainder, quotient =>
** 3 22
```

or, combinding the declaration with a multiple initialisation:

```
vars (remainder, quotient) = 223 // 10;
```

To illustrate the use of // on different kinds of numbers:

```
Dividing integers

    10 // 3 =>
```

```
    ** 1 3
    10 // -3 =>
    ** 1 -3
    -10 // 3 =>
    ** -1 -3
    -10 // -3 =>
    ** -1 3

Dividing decimals (or ddecimals)
    10.5 // -3.2 =>
    ** 0.9 -3

Dividing ratios
    (7/3) // (11/23) =>
    ** 29_/69 4

Dividing complexes
    (33 + sqrt(-57)) // sqrt(-5) =>
    ** 1.69505_+:0.841633 3_-:14
```

5.9.1 Binary and unary negation

Most of the above are 'binary' operations. That is, they require TWO arguments. For example the expression

```
3 + 4
```

applies the operation + to the two arguments 3 and 4. It denotes the number 7. The arguments may themselves be complex expressions, e.g.

```
(3 + 4) + (5 + 6)
```

The operation - can be binary (two arguments) or unary (one argument), and Pop-11 works out from the context which it is. When it is unary, as in

```
- 4
```

or

```
- (3 + 5)
```

it produces a number by NEGATING its argument.

When it is a binary operator, as in

```
3 - 4
```

or

```
(2 + 1) - (2 + 2)
```

it subtracts the second argument from the first. So the latter denotes the number -1.

Roughly, the rule is that if "-" occurs as part of an expression in which it is immediately preceded by an expression, and not a comma, semicolon, or opening bracket of any kind, then it is treated as binary and represents a subtraction. Otherwise it is unary and represents a negation (change of sign).

The following are all binary (i.e. subtractions)

```
x - y,    (33 * x) - sqrt(y),   x + y - z,  x * y - z - 1
```

The following are all unary (i.e. negation)

```
-3 + 5,   sqrt(- 22),  if x > 0 then -x else x endif
```

One consequence of this is that although the other arithmetical operators can be used as normal functions, e.g.

```
+(3, 5),  *(9, 10) =>
** 8 90
```

The minus symbol will normally be interpreted as a unary negation symbol in this sort of context, producing an unexpected result, e.g.

```
-(6, 3) =>
** 6 -3
```

which is equivalent to

```
6, -3 =>
```

or

```
6, 3, - =>
** 6 -3
```

However, according to the rule given, the following will work as a binary negation, which may also be surprising:

```
6, 3 - =>
** 3
```

When unary negate is required, this can be specified unambiguously by invoking the procedure negate, e.g.

```
3 + negate(4) * 5 =>
```

Compared with

```
3 + -4 * 5 =>
```

Similarly, because "nonop -" is always taken to refer to the subtraction procedure, the name "negate" must be used when unary minus is to be given as argument to a procedure, e.g.

```
maplist([ 1 2 3 4 5 6], negate) =>
** [-1 -2 -3 -4 -5 -6]
```

Compare using the subtraction operator concatenated with the sqrt procedure:

```
define sqrt_sub = nonop - <> sqrt enddefine;

sqrt_sub(8, 4) =>
** 2.0
```

Or the subtraction operator partially applied to 3:

```
define sub_3 = nonop - (% 3 %) enddefine;

sub_3(66) =>
** 63

sub_3(5.4) =>
** 2.4
```

5.9.2 WARNING: division of integers using "/" can produce ratios

The division operator "/" when given two integers (or bigintegers) will will not produce a decimal or ddecimal as result, but an integer (or biginteger) or ratio. Thus

```
10/5 =>
** 2

10/6 =>
** 5_/3
```

The reason for printing out ratios in this way, i.e. with the numerator and denominator joined by "_/" is that this will allow them to be read in again as ratios without violating Pop-11's syntactic rules for lexical items, mentioned briefly in chapter 1, and described fully in REF ITEMISE

This sort of result of integer division can cause confusion if you are used to a language which in which non-integer results of division are automatically converted (i.e. coerced) to decimals (reals, floats). This means that if you require such conversion you should ensure that one of the arguments to "/" is a decimal. e.g.

```
10/6.0 =>
** 1.66667
```

or

```
10.0/6 =>
** 1.66667
```

5.10 Infix predicates on numbers

Besides arithmetical operations that take numbers as arguments and produce numbers as results, Pop-11 includes various predicates for testing properties of numbers, or relations between numbers. These predicates all have boolean (i.e. true or false) results. The most widely used infix predicates are:

```
OPERATOR PRECEDENCE DESCRIPTION
    >       6       test two numbers: TRUE if first greater than second.
    <       6       test two numbers: TRUE if first less than second.
    >=      6       test two numbers: TRUE if first greater than or
                    equal to second.
    <=      6       test two numbers: TRUE if first less than or equal
                    to second.

    ==      7       Exact identity (i.e. the very same object in the
                    machine.)
    =       7       Equality: i.e. the same type of object with
                    equivalent contents.
    /==     7       Not identical
    /=      7       Not equal
    ==#     7       Same type of number and same numeric value.
                    (E.g. 8 = 8.0 is TRUE 8 ==# 8.0 false)
```

The rules for ”=” and ”/=” when the two arguments are of different types are quite complicated. If X and Y are of different types and are integers, bigintegers or ratios then they cannot be either ”=” or ”==”. If floating point numbers (decimals and ddecimals) are compared with integers, bigintegers or ratios then they are first converted to ddecimals and then compared. Further details are given in HELP EQUAL and REF NUMBERS

What sort of thing is denoted by an arithmetical expression depends on the ”top level” operator. E.g. consider:

```
    99 > (X + Y)
```

This takes the expression ‘99’ and the expression ‘(X + Y)’ each of which may denote a number, and creates a new expression which denotes true or false, depending on whether the first number is greater than or less than the other. I.e. > is a binary operator, taking two numbers and producing a TRUTH-VALUE, i.e. a BOOLEAN as a result. So

```
    99 > 66         denotes TRUE
    66 > 99         denotes FALSE
```

5.11 Recognizer predicates for number types

There are recognizer procedures which can be applied to any object and will always return a boolean (true or false) result.

```
isinteger(ITEM)
        Returns <true> if ITEM is a simple integer, <false> otherwise.

isbiginteger(ITEM)
```

Returns <true> if ITEM is a biginteger, <false> otherwise.

isintegral(ITEM)
Returns <true> if ITEM is a simple integer or a biginteger, <false> otherwise.

isratio(ITEM)
Returns <true> if ITEM is a ratio, <false> otherwise.

isrational(ITEM)
Returns <true> if ITEM is a simple integer, a biginteger or a ratio, and <false> otherwise.

isdecimal(ITEM)
Returns <true> if ITEM is a decimal or a ddecimal, <false> otherwise.

issdecimal(ITEM)
Returns <true> if ITEM is a single length decimal, <false> otherwise.

isddecimal(ITEM)
Returns <true> if ITEM is a ddecimal, <false> otherwise.

isreal(ITEM)
Returns <true> if ITEM is any number except a complex, <false> otherwise.

iscomplex(ITEM)
Returns <true> if ITEM is a complex number, <false> otherwise.

isnumber(ITEM)
Returns <true> if ITEM is any kind of number, <false> otherwise.

5.12 Coercing numbers from one type to another

number_coerce(NUM1, TO_NUM) -> NUM2
Produces a number NUM2 which is the number NUM1 converted to the representation class (i.e. rational, single-float decimal or double-float ddecimal) of the number TO_NUM.

```
number_coerce(3.5, 1) =>
** 7_/2

number_coerce(3.5, 1_/2) =>
** 7_/2

number_coerce(1_/2, 3.5) =>
** 0.5
```

5.13 Other arithmetic procedures

Other arithmetic procedures available in Pop-11 include the following, most of which can be applied to numbers of all types.

```
abs(x)       absolute value (modulus) of x
exp(x)       exponential of x (e to the power x)
max(x,y)     the bigger of two numbers
min(x,y)     the smaller of two numbers
sign(x)      -1 if x is negative, + 1 if positive, 0 if x = 0, or 0.0
random(x)    a random integer in range 1 to x
round(x)     the integer nearest to x, unless x is complex in which
             case the result is a complex number with real part and
             imaginary parts both rounded.
arccos(x)    trigonometric arc cosine of angle
arcsin(x)    trigonometric arc sine of angle
arctan(x)    trigonometric arc tangent of angle
cos(x)       trigonometric cosine of angle
log(x)       natural logarithm of a number - inverse of exp
log10(num)   returns the base 10 logarithm of num
negate(x)    negation of the number (i.e. -x)
sin(x)       trigonometric sine of angle
sqrt(x)      square root of number
tan(x)       trigonometric tangent of angle
```

NOTE: The trigonometric procedures use degrees, unless the variable popradians is set TRUE. The default is FALSE (i.e. use degrees.) (This is one of a large collection of user definable global variables controlling the behaviour of Pop-11. See HELP POPVARS).

5.14 Illustrating popradians

The variable popradians controls whether the trigonometric procedures take arguments, or produce results in degrees or radians. We can use the fact that "pi" is a built in constant thus:

```
true -> popradians;
sin(90) =>
** 0.893997

sin(pi/2) =>
** 1.0

arcsin(1) =>
** 1.5708

false -> popradians;
sin(90) =>
** 1.0

sin(pi/2) =>
```

```
** 0.027412

arcsin(1) =>
** 90.0
```

5.15 Other global variables controlling arithmetical computations

5.15.1 popdprecision

The 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. If it is false then only decimals (single precision floats) are produced. If it is the word "ddecimal" then ddecimal results are produced by arithmetic operators only if at least one of the arguments is ddecimal. If the value is anything else (e.g. true), then a ddecimal result can be forced if at least one of the arguments is integral or rational, even if the others are all decimals.

In NO case is there an increase in precision of floating point computations if all arguments are single-float decimal to start with. The default value of popdprecision is false.

Examples:

```
Case 1: Only single precision results

    false -> popdprecision;

    dataword(sqrt(3.5d0)) =>
    ** decimal

    dataword(sqrt(3.5s0)) =>
    ** decimal

    dataword(3.5d0 + 3.5s0) =>
    ** decimal

    dataword(sqrt(2)) =>
    ** decimal

Case 2: Double precision whenever a ddecimal is involved initially

    "ddecimal"-> popdprecision;

    dataword(sqrt(3.5d0)) =>
    ** ddecimal

    dataword(sqrt(3.5s0)) =>
    ** decimal

    dataword(3.5d0 + 3.5s0) =>
    ** ddecimal
```

```
    dataword(sqrt(2)) =>
    ** decimal

Case 3: Double precision produced if either ddecimal or non-decimal
    number is involved initially.

    true -> popdprecision

    dataword(sqrt(3.5d0)) =>
    ** ddecimal

    dataword(sqrt(3.5s0)) =>
    ** decimal

    dataword(3.5d0 + 3.5s0) =>
    ** ddecimal

    dataword(sqrt(2)) =>
    ** ddecimal
```

This means that if popdprecision is non false, then the evaluation of arithmetical expressions in which intermediate floating point numbers are produced will sometimes create temporary ddecimal numbers that are then discarded. As these are compound items (explained above) this can cause garbage collections, leading to reduced speed, the price of greater accuracy.

5.15.2 pop_reduce_ratios

This is normally true. Making it false prevents Pop-11's normal behaviour in which a ratio result is always reduced to its lowest common terms (and therefore to an integral result if the denominator becomes 1).

5.16 Miscellaneous operations on integers, ratios, floats

5.16.1 checkinteger, gcd_n, lcm_n

checkinteger(ITEM, LOW_INT, HI_INT)

Checks ITEM to be an integer within the range specified by lower bound LOW_INT and upper bound HI_INT (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(INT1, INT2, ..., INT_N, N) -> GCD

Computes the greatest common divisor of the all the N integers INT1, INT2, ..., INT_N, where the number N itself (a simple integer >= 0) appears as the rightmost argument. If N = 0, then GCD = 0; if N = 1, then GCD = INT1.

lcm_n(INT1, INT2, ..., INT_N, N) -> LCM

Computes the least common multiple of the all the N integers INT1, INT2, ..., INT_N, where the number N itself (a simple integer >= 0) appears as the rightmost argument. If N = 0, then LCM = 1; if N = 1, then LCM = INT1.

5.16.2 destratio, numerator, denominator

```
destratio(RAT)   -> (NUMERATOR, DENOMINATOR)
numerator(RAT)   -> NUMERATOR
denominator(RAT) -> DENOMINATOR
    These procedures return (on the stack) the numerator and denominator
    parts of a rational number, either together (-destratio-), or
    separately (-numerator- and -denominator-). When RAT is an integer
    or biginteger, then NUMERATOR = RAT, and DENOMINATOR = 1.
```

5.16.3 Operations on floats (decimals and ddecimals)

5.16.4 intof, fracof, float_digits, float_precision

```
fracof(x)    fractional part of a decimal number
intof(x)     integer part of a decimal number, positive or negative

    intof(-123.456), fracof(-123.456) =>
    ** -123 -0.456

float_digits(FLOAT) -> DIGITS
    Returns an integer, the number of digits represented in the internal
    (usually binary) floating-point format of the argument. (I.e. DIGITS
    has only two possible values, one for decimals and one for
    ddecimals. In all current Poplog implementations, b = 2 and DIGITS
    is around 22 for decimals, 53-56 for ddecimals.) On a sparcstation:

    float_digits(1.0e0), float_digits(1.0s0) =>
    ** 53 22

float_precision(FLOAT) -> SIGDIGITS
    Same as -float_digits-, except that the number of significant bits
    in the argument is returned. This will in fact be identical to
    float_digits(FLOAT), except that float_precision(0.0) = 0
```

5.16.5 float_decode, float_scale, float_sign

```
float_decode(FLOAT, INT_MANTISSA) -> (MANTISSA, INT_EXPO, FLOAT_SIGN)
    This procedure takes a floating-point number and splits it into its
    component parts, i.e. mantissa, exponent and sign. For full (gory)
    details see REF NUMBERS

float_scale(FLOAT1, INT_EXPO) -> FLOAT2
    This is equivalent to
        FLOAT1 * 2**INT_EXPO
    but is more efficient and avoids any intermediate overflow or
    underflow. 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_sign(FLOAT_SIGN, FLOAT1) -> FLOAT2
    Returns a floating-point number FLOAT2 of the same type and absolute
    value as FLOAT1, but which has the sign of the float FLOAT_SIGN. If
    FLOAT1 is false> then FLOAT2 is returned as a 1.0 or -1.0 of the
    same type and sign as FLOAT_SIGN.
```

For more details see REF NUMBERS

5.17 Complex Specific Operations

```
NUM1  +:  NUM2  ->  NUM3
NUM1  -:  NUM2  ->  NUM3
    These two operators are the basic way of creating complex numbers.
    Effectively, they both multiply their second argument by "i" (the
    positive square root of -1), and then either add the result to (+:)
    or subtract the result from (-:) the first argument.

+: NUM1  ->  NUM2
-: NUM1  ->  NUM2
    As prefix operators, +: and -: are equivalent to unary_+:(NUM1) and
    unary_-:(NUM1) respectively.

unary_+:(NUM1)  ->  NUM2
unary_-:(NUM1)  ->  NUM2
    Single-argument versions of +: and -:, which multiply their argument
    by i and -i respectively.

conjugate(NUM1) -> NUM2
    Returns the complex conjugate of its argument. The conjugate of a
    real number is itself, while for a complex number it is

        realpart(NUM1) -: imagpart(NUM1)

destcomplex(NUM) -> (REALPART, IMAGPART)
realpart(NUM)    -> REALPART
imagpart(NUM)    -> IMAGPART
    These procedures return the real and imaginary parts of a complex
    number, either together (-destcomplex-), or separately (-realpart-
    and -imagpart-). When NUM is real, then REALPART = NUM, and a zero
    of the same type as NUM is returned for IMAGPART.
```

5.18 random and oneof

Pop-11 provides a useful (but relatively sophisticated) pseudo-random number generator, controlled by an integer variable ranseed whose value is changed whenever the generator is called. By re-setting ranseed to the same initial value (e.g. 0) one can sure that the same sequence of ”random” numbers will be generated every time. If false is assigned to ranseed then an unpredictable integer value depending on the exact time of day will be assigned to it when the next random number is generated.

```
random0(NUM) -> RANDOM
```

Given a non-zero positive integer or floating-point number, this procedure generates a random number of the same type, in the range:

```
        0 <= RANDOM < NUM
```

where the distribution of RANDOM will be approximately uniform.

```
random(NUM) -> RANDOM
```

Same as -random0-, except that whenever the latter would return 0 or 0.0, the original argument NUM is returned instead. It can thus be defined as

```
    random0(NUM) -> RANDOM;
    if RANDOM = 0 then NUM else RANDOM endif;
```

Hence the range of the result is

```
    0 < RANDOM <= NUM
```

for a float, or

```
    1 <= RANDOM <= NUM
```

for an integer.

Examples

```
    repeat 10 times random(5) endrepeat =>
    ** 3 5 2 1 2 1 2 4 3 3

    2 -> pop_pr_places;
    repeat 10 times random(5.0) endrepeat =>
    ** 0.49 4.3 1.48 3.65 0.52 4.25 2.49 0.14 2.37 3.47
```

A closely related procedure is oneof, which takes a list and returns a randomly chosen element.

```
    repeat 10 times oneof([1 2 3 4 5]) endrepeat =>
    ** 5 4 5 5 4 2 4 5 3 3
```

5.19 Additional mathematical functions

Less common mathematical functions defined in REF NUMBERS include:

```
phase(NUM)
    Returns the complex phase angle of NUM as a floating-point
    quantity.

cis(REALANGLE)
    Returns the float-complex number cos(REALANGLE) +: sin(REALANGLE)

arctan2(REAL_X, REAL_Y) -> REALANGLE
    Computes the arctangent of REAL_Y / REAL_X, but using the signs of
    the two numbers to derive quadrant information.


sinh(ANGLE)
cosh(ANGLE)
tanh(ANGLE)
    These procedures compute the hyperbolic sine, hyperbolic cosine and
    hyperbolic tangent of ANGLE. The result is a floating-point, or a
    float-complex if ANGLE is complex.

arcsinh(NUM)
arccosh(NUM)
arctanh(NUM)
    These procedures compute the hyperbolic arcsine, hyperbolic
    arccosine and hyperbolic arctangent of NUM. For NUM complex, the
    result is a float-complex. For NUM real, the result will be a real
    float, except in the following cases:

        arccosh:    NUM < 1
        arctanh:    abs(NUM) > 1

    For -arctanh-, it is an error if NUM = 1 or -1.


linearfit(LIST) -> (M,C)
    The library LIB LINEARFIT makes available the procedure linearfit,
    which takes a list of pairs of numbers representing co-ordinates of
    points, works out the best straight line through the points, and
    returns its slope M, and its Y-intercept C.

        linearfit([% conspair(0,0), conspair(1.01, 0.98),
            conspair(1.85, 2.005), conspair(3.0, 3.0) %]) =>
        ** 1.015095 0.009136

    For vertical or nearly vertical lines it will produce an error.
```

5.20 Exercises

1. What are the data types of each of the following?

```
33
33.0
8:777
"cat"
'asdf;lkj876 *+*++ '
[1 2 3 4]
33 + sqrt(-5)
123.45e3
123.45s-3
```

2. How would the data types of the preceeding example change if popdprecision had the value false, "ddecimal" or true ?

3. What do the following expresions denote?

```
33 + 3
3 + 4 * 5
(3 + 4) * 5
6 - 3.0
sign(random(20))
1.5e5
1.5e-5
2:11111
```

If you have access to a computer running Pop-11 you can test your answers using '=>'. E.g.

```
6 - 3.0 =>
```

4. What is the effect of the variable POPRADIANS?

5. What variable can be given the value 2 to make pop print numbers in binary notation?

6. What happens if that variable is given the value 1?

7. How can you make pop-11 print in hexadecimal form?

8. How would you represent the numbers 16, 21, 30, 35, 40

```
(a) in binary form?
(b) in hexadecimal form (i.e. 16:???????)?
```

9. Define a procedure which produces a random decimal between -1 and 1

10. Define a procedure which produces a random ratio between -10 and 10

5.21 Testing for equality and inequality

It is often necessary to use conditional imperatives, in order to write flexible programs which do not always do the same thing. This requires the ability to test certain conditions, to see if they are true or false. An important class of such tests is testing for equality or similarity.

```
==  test any two objects. TRUE if they are identical:
    i.e. not really two objects but one and the same.
    (Strict equality). In Lisp this is referred to as EQ.

=   test any two objects. TRUE if they are identical,
    OR if they are of the same type with the same elements,
    i.e. if they are 'similar'. In Lisp this is referred to
    as EQUAL
```

E.g.

```
3 == 3      is TRUE because it's the same thing, the number 3
            that's referred to on both sides.
3 == 5 - 2  is also TRUE for the same reason.

[A B C] == [A B C]  is FALSE, since they are two lists
            Each time Pop-11 reads in '[ .... ]' it creates a
            new list, even if there was a similar one earlier.

[A B C] = [A B C]  is TRUE, since they are two SIMILAR lists.

'A STRING' == 'A STRING'    is FALSE, because there are two strings
but
'A STRING' = 'A STRING'     is TRUE, because the two strings are
                            similar
```

Thus it is possible to have two lists or strings with the same components, but which are not the very same object, e.g. if you type in the string 'silly' twice.

```
'silly' == 'silly' =>
** <false>
```

If you type in the same number twice Pop-11 will not treat the two expressions as denoting two different objects. So 999 will always refer to the same number. Strings and lists are different.

Words have to be treated like numbers. Pop-11 has to know about certain words, e.g. "define", "if", "(" and to be able to recognize new occurrences of the very same word. So words are entered in a dictionary when they are first read in, and if an expression denoting a word with the same characters is read in later, then instead of creating a new object Pop-11 finds the same word in its dictionary and re-uses that.

Thus, words, unlike strings, are 'standardised' in a dictionary, so that you cannot have two different words with the same characters. If you attempt to type in a second one, Pop-11 will find the original in its dictionary, and assume you wanted to refer to it. So both the following are TRUE:

```
"CAT" == "CAT" and  "CAT" = "CAT"
```

Writing two expressions on either side of an equality operation produces a new expression, which denotes a truth-value, TRUE or FALSE. In other words = and == each take two arguments and produce one BOOLEAN result.

5.22 Bitwise (Logical) integer operators

Pop-11 provides a collection of operations for manipulating integers considered as bit patterns. In order to see which bits are involved in a positive integer, use the pr_binary procedure defined above, e.g.

```
pr_binary(-100);
1100100
```

The rightmost bit corresponds to bit 0, and if there are N+1 bits the Nth bit is the leftmost one thus printed, except that negative numbers conceptually have 1 bits extending indefinitely to the left of the sign bit representation. In all the following definitions remember that the lowest value for N is 0, not 1.

5.22.1 Bit accessing procedures for integers

```
testbit(INT, N) -> BOOL
    This procedure tests the bit at position N in the integer INT
    returning true for 1 and false for 0. It has an unusual updater,
    which also returns a result:

BOOL -> testbit(INT, N) -> NEWINT
    This clears or sets the N'th bit of INT to 1 if BOOL is true or 0 if
    false, and the returns NEWINT as the resulting integer.

integer_leastbit(INT) -> N
    Returns the bit position N of the least-significant bit set in the
    integer INT. E.g.
        integer_leastbit(4), integer_leastbit(5) =>
        ** 2 0

integer_length(INT) -> N
    Returns the length in bits of INT as a two's-complement integer.
    That is, N is the smallest integer such that

        INT  <   ( 1 << N),   if INT >= 0
        INT  >=  (-1 << N),   if INT <  0

    Put another way: if INT is non-negative then the representation of
    INT as an unsigned integer requires a field of at least N bits;
    alternatively, a minimum of N+1 bits are required to represent INT
    as a signed integer, regardless of its sign.

integer_bitcount(INT)
    Counts the number of 1 or 0 bits in the two's-complement
    representation of INT. If INT is non-negative, N is the number of 1
    bits; if INT is negative, it is the number of 0 bits.
```

For more information on any of theses see REF NUMBERS

5.22.2 Infix and prefix bitwise (logical) operators

```
OPERATOR  PRECEDENCE  DESCRIPTION
    &&       4       Logical and of bits in two integers

    &&~~     4       Logical and of first argument and negation of
                     second. (Useful for clearing bits.)

    ||       4       Logical "inclusive or" bits in two integers.

    ||/&     4       Logical "exclusive or" of bits in two integers.

    &&/=_0   6
    &&=_0    6
                 These two operators provide the same results as the
                 boolean expressions

                     INT1 && INT2  /==  0
                     INT1 && INT2  ==   0

                 but are more efficient and avoid producing intermediate
                 results.
```

5.22.3 Unary bitwise negation ~~

```
The unary operator ~~ has precedence 4
    It produces the logical complement of its argument, i.e. there is a
    1 in the result for each bit position for which the argument has 0.
    It is always true that ~~ INT = -(INT + 1)
```

5.22.4 Bitwise (logical) shift operators

```
OPERATOR  PRECEDENCE  DESCRIPTION
 I << N      4       Shifts the bits in I N places left (or -N places
                     right if N is negative).

 I >> N      4       Shifts the bits in I N places right (or -N places
                     left if N is negative).


integer_field(SIZE, POSITION) -> ACCESS_P
    This procedure, described fully in REF NUMBERS can be used to create
    very efficient procedures for accessing and updating 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.
```

5.23 Iteration over numbers

5.23.1 for num from ... by ... to ... do ... endfor

There are many ways of expressing iteration over sets of numbers in Pop-11, including using repeat ... endrepeat, while ... endwhile, and until ... enduntil. However, it is usually "for" loops that are most convenient. These allow a variable to take successive values in a series of numbers defined by repeatedly adding or subtracting a fixed amount.

The basic syntactic form provided to support this kind of iteration is the following (where line-breaks are equivalent to spaces):

```
for <variable> from <number> by <number> to <number> do
    <actions>
endfor;
```

However the "from" and "by" components may be omitted where the number in question is 1. So the following forms are also permitted:

```
for <variable> from <number> to <number> do <actions> endfor
        (Default: "by 1")

for <variable> by <number> to <number> do <actions> endfor
        (Default: "from 1")

for <variable> to <number> do <actions> endfor
        (Defaults: "from 1", "by 1")
```

Examples of arithmetical for loops are:

```
vars x;
for x to 10 do spr(x) endfor;
1 2 3 4 5 6 7 8 9 10

for x by 0.5 to 5 do spr(x) endfor;
1 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

for x by 1/3 to 5 do spr(x) endfor;
1 4_/3 5_/3 2 7_/3 8_/3 3 10_/3 11_/3 4 13_/3 14_/3 5

for x from 1 to 6 do spr(x) endfor;
1 2 3 4 5 6

for x from 19 by -1 to 8 do spr(x) endfor;
19 18 17 16 15 14 13 12 11 10 9 8

for x from -5 by -0.6 to -9 do spr(x) endfor;
-5 -5.6 -6.2 -6.8 -7.4 -8.0 -8.6
```

For loops may be nested. For example, to make a list of all possible pairs of numbers between 3 and 5 do

```
vars x, y;
[% for x from 3 to 5 do
    for y from 3 to 5 do
        [^x ^y]
    endfor
endfor %] =>
** [[3 3] [3 4] [3 5] [4 3] [4 4] [4 5] [5 3] [5 4] [5 5]]
```

5.23.2 Using fast_for

When it is known that the values of the loop variable and the increment or decrement will all be integers, then it is permissible to use the fast_for instead of for, e.g.

```
vars x;
fast_for x from 1 by 300 to 10000 do x endfor =>
** 1 301 601 901 1201 1501 1801 2101 2401 2701 3001 3301 3601 3901
    4201 4501 4801 5101 5401 5701 6001 6301 6601 6901 7201 7501 7801
    8101 8401 8701 9001 9301 9601 9901
```

Using this form will eliminate some of the run-time checks that are otherwise required. However, it is risky in that errors may not be detected. (See HELP EFFICIENCY).

5.23.3 Iterating over non-arithmetical progressions

For sequences of numbers not defined by constant intervals it may be convenient to use the following form:

```
for <action> step <action> till <condition> do <actions> endfor
```

For example to produce a list of the powers of 2 less than 10000 do

```
vars x;
[% for 2 -> x step x * 2 -> x; till x >= 10000 do x endfor%] =>
** [2 4 8 16 32 64 128 256 512 1024 2048 4096 8192]
```

5.24 Using external mathematical libraries

For some programs it is desirable to use existing mathematical libraries written in C or Fortran or other languages, e.g. the NAG libraries.

This is possible by externally loading the libraries into Poplog. This can save work duplicating code, and also produce faster performance than rewriting the programs in Pop-11.

The mechanisms for linking in external programs are described in

```
REF EXTERNAL
REF EXTERNAL_DATA
REF DEFSTRUCT
    These files describe the basic mechanisms for linking in and
```

```
    communicating with "external" programs.

HELP EXTERNAL
HELP NEWEXTERNAL
HELP NEWC_DEC
HELP FORTRAN_DEC
    These files describe higher level language-specific tools
    for managing external programs.
```

6 CHAPTER.6: LIST PROCESSING IN POP-11

In Chapter 1 an example of the use of lists to represent a collection of items of information about rooms was used to introduce a number of features of the language Pop-11. We have occasionally used lists to illustrate other points in preceding chapters. This chapter explains in more detail why lists are important, how they are implemented and how they can be used.

The chapter will provide more general information on syntax for constructing lists, together with an overview of the basic procedures available for operating on lists including the two list-pattern matchers provided in Pop-11.

6.1 On knowing about lists

There are several things a Pop-11 user needs to know about lists:

```
What they are used for
How to construct them
How to extract information from them
How they are represented in the machine
How dynamic lists and static lists differ
How to compare them or test them to see whether they satisfy
    certain conditions relevant to deciding what to do.
Which facilities are available for manipulating lists
How to design your own facilities for manipulating lists
How to optimise your list-processing programs
```

A comprehensive overview of all these topics would require a whole book. This chapter, introduces the basic ideas and gives some examples of how to use them.

6.2 Why use lists?

For work in Artificial Intelligence, and many other applications where we need to represent data in a variety of forms, lists are a very useful data type, partly because it is possible for two lists to share common sublists, partly because it is possible for lists to change their size by being extended in the middle or at either end, and partly because there is a way of representing them that is simple and flexible.

In Lisp and closely related languages, such as Scheme and T, it is also the case the lists can represent interpreted procedures: this is sometimes useful, though not as useful as the early inventors of Lisp supposed it would be. Pop-11 has a different philosophy regarding procedures and some of the things that a Lisp programmer would do by building a list and interpreting it, a Pop-11 programmer would do by using partial application to build closures that can be run (as described in chapter 4).

Also, the incremental compiler provided in Pop-11 allows lists of text items representing procedure definitions to be compiled at run time. For example

```
vars name = "silly", func = "last", list = [a b c];
popval([define ^name; ^func(^list); enddefine;]);
```

creates a new COMPILED procedure called silly, which can be run like any other procedure and then applies last to [a b c]

```
silly() =>
** c
```

This is sometimes useful, in writing programs that create new programs.

However, Pop-11 does not provide the equivalent of Lisp's EVAL, which interprets Lisp procedure definitions.

Nevertheless lists play a very important role in representing information in many Pop-11 programs.

6.2.1 Lists can contain a mixture of elements of any type in Pop-11

Lists can be used to represent or store many kinds of information. In Pop-11, like most AI languages, lists can contain any type of object, including, for instance, numbers, words, procedures, and other lists. Moreover, the same list can contain a mixture of items of different types. This is impossible to achieve in strongly typed languages, including even some languages with a polymorphic type system (e.g. ML).

This generality has a number of implications. First of all it is possible conveniently to use lists to represent all kinds of information in a common format, especially information that changes its structure while a program is running. Moreover, because one does not need to use different datastructures for different kinds of information it is possible to produce a collection of very general re-useable procedures that work on lists containing different kinds of data. The Pop-11 "pattern matcher", described below, is an example of a very powerful general purpose utility procedure for operating on many types of lists. Another example is the Pop-11 database package, based on the matcher.

Being able to use such "generic" procedures with lists of different kinds can simplify program development and lead to more compact and robust programs.

Illustrating generality: isinlist

To illustrate here is a simple example. In Pop-11 it is possible to provide a single "isinlist" procedure, which takes an item and a list and returns a boolean result, which is true if the item is in the list otherwise false, as follows:

```
define isinlist(target, list) -> found;

    lvars item;
    for item in list do
        if item = target then
            true -> found;
            return();
        endif
    endfor;
    false -> found
enddefine;
```

This procedure is very general, insofar as it can then be used on lists of numbers, words, strings, lists, etc. in any combination, as it makes no assumptions about the types of the values of the variables "target" and "list", except that the value of "list" should be a list: if not a run time error will occur, because it uses the list iteration construct. Although it assumes that the second argument will always be a list, it makes no assumptions about the contents of the list. In some languages, such as Pascal it would be necessary to produce a different version of the procedure for each type of list, and it would not be possible to mix items of different types in one list.

The versatile behaviour of isinlist can be illustrated thus:

```
vars person_data =
    [name joe age 23 wife mary salary 20000 children [sue fred]];

isinlist("mary", person_data) =>
** <true>

isinlist(20000, person_data) =>
** <true>

isinlist("fred", person_data) =>
** <false>

isinlist([sue fred], person_data) =>
** <true>
```

(In fact the built in Pop-11 procedure "member" behaves like isinlist, so it is not necessary for users to define isinlist. Try replacing "member" in all the above examples.)

There are many other examples of re-usable general procedures that operate on lists no matter what their contents. For example, applist, maplist, syssort, the concatenator <>, and other procedures described below.

The library procedure assoc can create an associative memory made of two-element lists, no matter what the contents of the lists are.

```
vars person =
    assoc([[name fred]
           [sex male]
           [age 30]
           [kids [sue tom dick]]]);

;;; person is now an association mechanism.
person("age") =>
** 30
person("age") + 1 -> person("age");
person("age") =>
** 31
person("kids") =>
** [sue tom dick]
```

6.3 Lists vs other representations

How to choose between the use of lists or other means of representation is not always obvious: it can take many years of experience to make good decisions, weighing up such criteria as ease of program design, ease of testing, ease of long term maintenance, compactness, speed, generality.

Sometimes the reason for using a particular representation is simply that there already exist utilities that do the job one needs and one does not wish to have to rewrite them.

Although in principle lists can be used for every type of data, it is sometimes useful on grounds of compactness of data, or speed of access, or more useful run-time checking to use a more specific data-type for a particular problem.

When more specialised representations are required, Pop-11 provides records, vectors, arrays, strings, properties and user defined record classes and vector classes. (See the list of data types in Chapter 2.)

Moreover, the object-oriented extensions to Pop-11 (Objectclass and Flavours) provide additional means of structuring large programs. See Chapter 8 below for a brief introduction to Objectclass.

6.4 Lists in AI

A central thesis of much work in Artificial Intelligence is that certain sorts of computational processes provide a good way to represent many of the processes we call thinking, seeing, reasoning, speaking, learning. But what sorts of computations? Not only the manipulation of numbers found in much scientific and engineering computation, but also a wide variety of non-numerical symbol manipulations. Intelligence seems to involve the manipulation of many kinds of symbols that can be used to store information about many kinds of things and their properties and relationships.

The role of symbolic and non-symbolic processing is the subject of considerable debate among those interested in how intelligent systems work. For now, we shall avoid getting embroiled in such debates and merely point out that for many purposes where symbolic manipulation is useful, lists can provide a powerful and general representation. They can also be used for programs operating on numerical data, though in that case some other representation will often be more efficient and more natural, e.g. arrays or vectors.

6.5 Constructing lists in Pop-11

The simplest way to construct a list is to use square brackets. But there are several others:

6.5.1 Lists are constructed using [ ]

These "list constant brackets" may contain text items, i.e words, numbers strings, lists, or anything else. E.g.:

```
[ A B C D]            is a list of four words
[1 cat 2 dog 3 pig ]  is a list of six items.
[string 'a short string' 66]
   is a list with a word a string and a number.
```

[] is the empty list. There's more on this below.

List brackets may also be used to construct lists which contain lists, E.g.:

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

is a list with two lists of numbers and one number. It contains exactly three elements, of which the first contains two elements.

A list can also contain vectors, signified with the vector brackets { } which themselves can enclose further lists or vectors to any depth: e.g. (using more spaces than necessary, for clarity):

```
[ a b { c d [ e { f } ] } [ g h ] ]
```

This is a list containing four items: two (quoted) words "a" and "b", a vector { c d [e { f }] } and a two element list [g h].

6.5.2 List brackets quote their contents

The words in a list are by default quoted, apart from the list and vector brackets, which signify embedded lists and vectors.

So even if the words are Pop-11 identifiers, their values are not inserted in the list, just the words themselves. (This is unlike the convention in Lisp, where, by default, the values are inserted.)

Example:

```
[true false if + * then sqrt ] =>
** [true false if + * then sqrt]
```

6.5.3 Unquoting using ˆ and %

In order to get the contents of a list (or vector) expression evaluated, or unquoted, it is possible to use either ˆ or matching pairs of % ... % as in this deliberately confusing example

```
vars x = "cat", cat = "x";

[x cat ^x ^cat] =>
** [x cat cat x]
```

which is equivalent to each of

```
[x cat %x% %cat%] =>
[x cat %x, cat%] =>
[% "x", "y", x, cat %] =>
```

Another example would be

```
vars n1 = 5, n2 = 7;
[the sum of n1 and n2 is %n1 + n2%] =>
** [the sum of n1 and n2 is 12]
```

So, in order to insert the value of a single variable, it is simplest to use ˆvariable, whereas pairs of percents % ... % may be more appropriate for longer expressions to be evaluated.

NOTE: ˆ cannot cannot be used outside a list or vector expression. However, % has an additional use in forming closures, using partial application, as described in Chapter 4, and HELP PERCENT

Between the percent signs in a list or vector expression the normal syntactic rules for Pop-11 apply, so that, for example, expressions must be separated by commas, and in order to quote a word the word quote symbol "" must be used.

6.5.4 ^(...) is equivalent to % .... %

For historical reasons, the use of the percent symbols is equivalent to the use of ^ followed by a parenthesized expression. Thus, the last two examples could be written

```
[ ^( "x", "y", x, cat )] =>
[the sum of n1 and n2 is ^(n1 + n2)] =>
```

There is no difference in meaning or efficiency between ^(...) and % ... %.

If embedded list or vector expressions are used inside these "unquoted" portions of a list, then by default they too quote their contents, unless % or ^ is used to unquote, e.g.

```
[% [two words], [% n1, "a"%], [% n2 % b c] %] =>
** [[two words] [5 a] [7 b c]]
```

is a list of three lists, whose contents depend on the values of the identifiers n1 and n2, but treats all other words as quoted.

The symbol "%" can occur anywhere in a list expression (or vector expression). But it must occur in pairs. Roughly speaking, in a list the first occurrence of each pair means: "switch from quoting to non-quoting mode", and the second occurrence means "switch from non-quoting to quoting mode". In non-quoting mode variables are replaced by their values and any procedures are run and their values are inserted in the list instead of the names of the procedures being put in the list.

6.5.5 Loops can occur in unquoted portions of a list

There is no restriction at all on the sequence of expressions that can occur in the unquoted portion of a list. The Pop-11 instructions are run, and any results produced that are left on the stack will be incorporated in the final list.

So for example, it is possible to use a loop between % .. % to create a list of numbers.

```
vars n;
[% for n from 1 to 20 do n endfor % ] =>
** [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]
```

Similarly the contents of the list, may be conditional on something else, as in

```
[The list was % if n > 15 then "big" else "small" endif % ] =>
** [The list was big]
```

6.6 WARNING lists in procedures are not "constants"

An expression using the list (and vector) brackets, and containing no occurrences of the un-quoters "^", "^^" or "%" may LOOK as if it represents a constant object, in the way that a word expression, e.g. "cat" or a string expression does e.g. 'the cat'. However, in Pop-11 an expression like

```
[ a b { c d [ e { f } ] } [ g h ] ]
```

compiles into a collection of instructions to CREATE a list of words, vectors and lists. Although the very same words are used each time (they are are stored in the Pop-11 dictionary for that purpose), the other structures are recreated each time the procedure is run. So for example if you define a procedure to return what looks like a constant list, it will return a new copy each time, thus:

```
define a_list();
    [a list]    ;;; create a list and leave on stack
enddefine;

;;; Use it to create a list
vars x1 = a_list();
x1 =>
** [a list]

;;; create another list using the same procedure
vars x2 = a_list();
x2 =>
** [a list]
;;; test for equality
x1 = x2 =>
** <true>
;;; test for identity
x1 == x2 =>
** <false>
```

So although the different lists returned by the procedure are = to each other (i.e. they are of the same type and they contain objects that are = to each other) nevertheless they are not == to each other. They are not the same object, even though they have the same structure and contain objects that are = to each other.

Similarly two apparently identical list expressions will create copies of each other but not identically the same object:

```
[a list] = [a list] =>
** <true>

[a list] == [a list] =>
** <false>
```

Contrast the case of words, which are identical if they look the same:

```
"list" == "list" =>
** <true>
```

Similarly, if a list is assigned to a variable outside a procedure then as long as nothing new is assigned to the variable, it will always point to the same list.

```
vars x2 = [another list];

x2 == x2 =>
** <true>
```

For more on the difference between "=" and "==" see HELP EQUAL

6.6.1 Creating a truly constant list expression

A consequence of all this is that a procedure that uses a list expression will create a new copy each time it runs, even though this may not be strictly necessary, and this can cause more garbage collections to occur than the program really needs. (The garbage collector is described briefly in Chapter 2). This will happen if you define a procedure that uses the pattern matcher and includes something like

```
if list matches [room ?name ?len ?width ?height] then
```

In such cases the pattern to the right of ”matches” will be recreated on every execution of the procedure.

For many programs where speed is not crucially important this will not matter as the Pop-11 garbage collector is very fast, and the creation of unnecessary temporary structures will not make very much difference. Where it does matter, there are two things you can do to ensure that exactly one list is created and then re-used.

Using lconstant

If inside a list you declare a variable using lconstant, then you MUST initialise it, and the expression to initialise it will be evaluated only once, at compile time, and thereafter the same result will be used on every call of the procedure. E.g.

```
define the_list();
    lconstant list = [the list];
    list
enddefine;

the_list() =>
** [the list]

the_list() == the_list() =>
** <true>
```

If ”lvars” or ”vars” had been used instead of ”lconstant” that last identity comparison would have been false.

6.6.2 using #_< ... >_# to evaluate an expression at compile time

Another way to get a truly constant constant list is to use these brackets which cause their contents to be evaluated once, at compile time and then the result is re-used each time the procedure is run.

```
define another_list();
    #_< [another list] >_#
enddefine;

another_list() =>
** [another list]
another_list() == another_list() =>
** <true>
```

So you can use these brackets in expressions like

```
if list matches #_< .... >_# then
```

However this will NOT work properly if the list expression includes items whose values are not known until the procedure is actually running, ie.g. list or vector expressions containing any of

```
^  ^^  % ... %
```

as described above.

6.6.3 WARNING constant lists can cause strange behaviour

If you use a truly constant list, and your program changes that list then the change will be remembered between invocations of the procedure, and the list will then not be a constant.

Form example, on each call of this procedure, it will return the value of item, which is got from the head of the stored list.

```
define counter() -> item;
    lconstant list = [0];

    list(1) -> item;        ;;; get the first item.
    item + 1 -> list(1);    ;;; increment the stored value
enddefine;

counter() =>
** 0
counter(), counter(), counter() =>
** 1 2 3
```

If ”lconstant” were replaced by ”lvars” then on every execution the procedure would construct a new list containing [0], and would therefore always print out the same result.

If a constant list is returned as a result, and then altered OUTSIDE the procedure that produced it, this will change the future behaviour of the procedure.

```
define announce() -> list;
    lconstant list = [0 is a number];
enddefine;

announce() =>
** [0 is a number]
```

But if the result is tampered with....

```
vars x = announce();
999 -> x(1);
x =>
** [999 is a number]
```

then that ”corrupts” the procedure:

```
announce() =>
** [999 is a number]
```

Some of the issues relating to constant lists are described in the online file HELP EFFICIENCY. The are two dangerous procedures described in REF sys_grbg_list and sys_grbg_destpair which explains how a list once created can be returned to the "free" store so that it can be re-used without creating unnecessary garbage collections. These procedures are dangerous and should only be used by very experienced programmers who know how to make sure when it a list really cannot be used again, and therefore can safely be returned to the free store.

It is precisely because that sort of thing can be very difficult to determine that Pop-11 uses an AUTOMATIC garbage collector, which is very much safer.

6.7 Concatenating lists using <>

If two lists are to be joined together, the general-purpose Pop-11 concatenator can be used, as in

```
vars list1 = [a b c], list2 = [d e f];
list1 <> list2 =>
** [a b c d e f]

list1 <> list2 <> list1 <> list2 =>
** [a b c d e f a b c d e f]
```

6.8 Merging lists using the double up-arrow

A more flexible facility than <> is the use of ^^ which allows the contents of one list to be spliced into another at any location, whereas <> allows only joining lists "end on". For example:

```
vars list1 = [a b c], list2 = [d e f];

[1 2 ^^list1 3 4 ^^list2 ] =>
** [1 2 a b c 3 4 d e f]
```

The same list can be spliced in at several different points:

```
[ ^^list1 x y z ^^list1] =>
** [a b c x y z a b c]
```

It is possible to use an arbitrary Pop-11 expression that evaluates to a list after ^^ provided that it is enclosed in parentheses.

```
[a set of numbers ^^( [% 1 + 2, 3 + 4, 5 + 6%])] =>
** [a set of numbers 3 7 11]
```

Compare the use of ^ without the embedded list brackets and % symbols:

```
[a set of numbers ^( 1 + 2, 3 + 4, 5 + 6)] =>
** [a set of numbers 3 7 11]
```

So inside a list, the form

```
^^( [% <Pop-11 instructions> %] )
```

is exactly equivalent to

```
^( <Pop-11 instructions> )
```

except that the former wastefully creates a temporary list and then discards it after its contents have been spliced into the final list. so it is worth using ^^ only when a list already exists, although it may be necessary to use a complex expression to access it, e.g.

```
vars list = [a [b c d] e f];
[another list with : ^^(list(2)) ] =>
** [another list with : b c d]
```

Here list(2) was used to get at the second element of list.

If the above had used ^ instead of ^^, the result would have been:

```
[another list with: ^(list(2)) ] =>
** [another list with : [b c d]]
```

In other words ^^ removes a layer of list brackets that ^ will leave. The prefix "^^" can be thought of as meaning "remove the list brackets".

If the value is not a list, an error will result.

```
vars x = 99;
[a b c ^^x] =>

;;; MISHAP - LIST NEEDED
;;; INVOLVING:  99
```

6.9 Lists are a derived data-type

Pairs are the primitive datatype used: conspair, front, back

LISTS in Pop-11 are built out of a data type called PAIRS, which are two-element records. Pairs can be created using the procedure conspair.

```
vars pair1 = conspair(3, "cat");

pair1 =>
** [3|cat]
```

Note how pairs are printed almost like lists, except for the vertical bar to indicate that they are not two element lists.

The built in system procedures front and back can be used to access or update their elements. E.g.

```
front(pair1) =>
** 3

"dog" -> back(pair1);
pair1 =>
** [3|dog]
```

Pairs can be chained together, e.g. by creating a new pair whose back is the old one.

```
vars pair2 = conspair(2, pair1);
pair2 =>
** [2 3|dog]
```

Notice how the printing procedure starts printing pair2 as if it were a list, that is a chain of pairs, then suddenly it finds that the list ends in a pair that has a word as its back, rather than another pair or the special end of list object [], so it indicates that it's not a proper list, by using the vertical bar again.

6.9.1 destpair(pair) -> (pair_front, pair_back)

The procedure destpair, when given a pair, puts both the front and the back on the stack. So using destpair can sometimes be more efficient than first calling front then back.

```
destpair(pair1) =>
** 3 dog
```

The front and the back of the second pair created above, pair2, can also be accessed in one Pop-11 instruction, using destpair:

```
destpair(pair2) =>
** 2 [3|dog]
```

For example, a loop doing something to every item in a chain of pairs might have:

```
while ispair(chain) do
    destpair(chain) -> (item, chain);
    .... instructions involving item .....
endwhile;
```

instead of

```
while ispair(chain) do
    front(chain) -> item;
    .... instructions involving item .....
    back(chain) -> chain;
endwhile;
```

A chain of pairs ending in [] is a list

The chain of two pairs created above involving pair1 and pair2 is not yet a proper list, since every list must end with the empty list [] as the back of the final pair (except for dynamic lists, described below.)

```
vars
    pair1 = conspair("b", "c"),
    pair2 = conspair("a", pair1);
```

So neither pair1, nor pair2 is a proper list, as is indicated by the way they are printed, e.g.:

```
pair2 =>
** [a b|c]
```

If we now change the back of pair1, by assigning [] to it, we'll get a proper list:

```
[] -> back(pair1);
pair1 =>
** [b]
```

and this has also changed pair2 into a properly terminated list.

```
pair2 =>
** [a b]
```

NOTE: The empty list [] is a special, unique object, with its own dataword (like termin and popstackmark).

```
dataword([]) =>
** nil
```

The system identifier "nil" can also be used to refer to the empty list:

```
nil =>
** []

nil == [] =>
** <true>

conspair(3, conspair(4, nil)) =>
** [3 4]
```

6.9.2 List expressions are "syntactic sugar"

From this, it should be clear that the list expression [a b] is actually just "syntactic sugar" for the expression:

```
conspair("a", conspair("b", []))
```

and [a ^x b] is syntactic sugar for

```
conspair("a", conspair(x, conspair("b", [])))
```

and [a %x + y% b] is syntactic sugar for

```
conspair("a", conspair(x + y, conspair("b", [])))
```

So, inside a list expression, words are quoted by default and "^" or "%" is used to unquote them, whereas outside a list words are by default not quoted and """ is used to quote them. The same applies to vector expressions using { ... }

```
Exercise: What is [the cow is brown] syntactic sugar for?
```

6.9.3 Recursively chaining down list links

Because all lists end in the unique object [], there are many programs that chain down a list made of pairs by assigning the first pair to a variable, then the second pair, then the third, and so on, stopping only when it is found that the variable points to [].

For example here is a recursive procedure to copy a list (which could be written more compactly but would be less clear):

```
define copy_list(oldlist) -> newlist;
    lvars newtail ;

    if oldlist == [] then
        [] -> newlist
    elseif ispair(oldlist) then
        ;;; recursively copy the back of the list
        copy_list(back(oldlist)) -> newtail;
        ;;; and create a new list using the old front
        ;;; and the new tail
        conspair(front(oldlist), newtail) -> newlist
    else
        mishap('LIST NEEDED', [^oldlist])
    endif
enddefine;

copy_list([a b c d]) =>
** [a b c d]
```

Recursive procedures can often be understood more easily if traced, using the trace mechanism described in Chapter 4 (or HELP TRACE). So try running the previous command after first doing:

```
trace copy_list
```

Note that the Pop-11 system procedure copylist does exactly what copy_list does, so there's no need for the user to define copy_list.

6.9.4 Recursing down the front and the back of a ”tree”

Note also that if there were an embedded list, it would not be copied: it would reappear in the new list, as in [a b [c d] e]. Lists like this have a ”tree” structure, as shown by the boxes diagrams below.

Complete copying of a tree can be achieved by a ”doubly recursive” procedure, which copies list elements that are lists as well as copying the ”top level” list.

For example, here’s a doubly recursive version, which simply leaves its results on the stack instead of using an output local variable:

```
define copy_tree(tree);
    if ispair(tree) then
        ;;; copy the front if its a pair, otherwise use it, and
        ;;; then put the result in a new pair, with a copy of the
        ;;; old back
        conspair(
            if ispair(tree) then copy_tree(front(tree))
                else tree endif,
            copy_tree(back(tree)) )
    else
        ;;; just return the item that is not a pair
        tree
    endif
enddefine;

vars tree = [[a 1 2] [b 3 4] [c [x y] 5 6] d];
copy_tree(tree) =>
```

Running that example with copy_tree traced, will show all the recursive calls.

We turn now to showing in more detail how the static lists are actually represented.

6.9.5 Numeric subscripts and lists

We have already seen on numerous occasions that lists can be treated as if they were one dimensional arrays, whose elements can be accessed via numerical subscripts, starting from 1.

```
vars list = [a bird in the hand];

list(2) =>
** bird

"fish" -> list(2);
list(2) =>
** fish

list =>
** [a fish in the hand]
```

If the number is too small or too large an error will result.

```
list(8) =>
;;; MISHAP - BAD ARGUMENTS FOR INDEXED LIST ACCESS
;;; INVOLVING:  8 [a fish in the hand]
;;; DOING    :  compile ....
```

6.9.6 Iterating down list links

Here is a rather different program that uses iteration rather than recursion. It searches down the list links until if finds a target element and then returns a copy of the rest of the list, starting from the target.

```
define tail_list(target, oldlist) -> newlist;

    oldlist -> newlist;

    repeat
        if newlist == [] then
            mishap('TARGET NOT IN LIST', [^target ^list])
        else
            if front(newlist) = target then
                return();   ;;; result is newlist
            else
                ;;; get next link in the list
                back(newlist) -> newlist
            endif
        endif
    endrepeat
enddefine;

tail_list("cat", [The black cat sat on the mat]) =>
** [cat sat on the mat]
```

Exercise: re-write that procedure using the for loop format:

```
for <var> on list do .... endfor
```

6.10 Why use "hd" and "tl" instead of "front" and "back" ?

The need to hide implementation details

We have seen that a list is actually a chained collection of pairs. But most of the time the user does not need to know about this. High level facilities for manipulating lists are provided, which conceal from the user the details of how they are represented in the machine. For example Pop-11 provides special syntax using the list brackets [...] together with "^", "^^" and "%" for creating lists by specifying what they should "look like", without worrying about their implementation at a lower level.

Additional facilities provided that "hide the implementation details" are the following procedures:

```
matches, copylist, applist, maplist, and the list concatenator <>.
```

In addition there are looping constructs, e.g. using

```
for <var> in <list expression> do ... endfor
foreach <pattern> in <list> do ... endforeach
```

which also make it convenient to manipulate lists without knowing how they are implemented. (The faster version: "fast_for ... endfast_for" described in REF FASTPROCS) assumes that there are no dynamic lists (described below), and therefore does not do as much checking.)

For the sake of efficiency it is sometimes useful to know how lists are implemented and to use the procedures conspair, destpair, front and back as in previous examples. This would have the disadvantage that if the implementation were ever to change and something other than simple chained pairs were used for lists, then programs that used these procedures would stop working.

In fact, in some versions of the Pop language that is already the case, because Pop2, Pop10 and Poplog Pop-11 support what are called "dynamic" lists, which have a slightly different implementation from the lists described so far, which are all "static" lists. These will be described below.

Meanwhile, the main justification for using hd, tl, and dest can be thought of as being to "hide the implementation details", which allows the implementation to be changed.

Note: "hd" and "tl" in Pop-11 correspond to "CAR" and "CDR" in LISP.

6.10.1 using :: instead of conspair

For this reason it is desirable to use something other than conspair, which is guaranteed always to give a list. An infix list constructor operator is provided, namely "::". It's infix precedence is 4. It is similar to conspair, except that it will complain if its second argument is clearly not a list.

```
"a" :: [b c d] =>
** [a b c d]

conspair("a", "b") =>
** [a|b]

"a" :: "b" =>
;;; MISHAP - LIST NEEDED
;;; INVOLVING:  b
;;; DOING    :  :: (etc.)
```

Warning: Unfortunately, for historical reasons, "::" associates to the left rather than to the right, so parentheses are needed to produce a flat list, as with conspair:

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

Without the parentheses, it is equivalent to

```
((1 :: 2) :: 3) :: [] =>
```

And the embedded invocation of ”1 :: 2” will produce a LIST NEEDED error.

For this reason, when constructing a list from a number of known items it is usually simpler and clearer to use the ”syntactic sugar” of list expressions than to use multiple occurrences of ”::”, e.g.

```
[1 2 a ^x ^y]
```

is used instead of

```
1 :: (2 :: ("a" :: (x :: (y :: [])))))
```

6.10.2 The difference between :: and <>

It is important to be aware of the difference between the list constructor :: and the list concatenator <>.

The procedure :: takes a potential list-head and a potential list-tail (which must already be a (possibly empty) list and makes a new list with that head and tail (and without copying the tail). Thus the first argument of :: is the first element of the new list.

By contrast the concatenator <> takes a potential initial segment of a list and a potential final segment, and makes a new list containing those segments. The first argument of <> is not the first ELEMENT of the new list: though its elements are the initial elements of the new list.

The difference can be interested with the following two examples, where both operators are applied to two lists:

```
[a b c] :: [d e f] =>
** [[a b c] d e f]

[a b c] <> [d e f] =>
** [a b c d e f]
```

The first example produces a list of four items, the second a list of six items.

Each could be defined in terms of the other. To illustrate we can use <> to define an infix operator like :: called :+:, and we can use :: to define an infix operator like <> called <+>. In each case we give the infix precedence as an integer following ”define”.

```
define 4 :+: (item, list);
    [^item] <> list
enddefine;

;;; Test it:

"a" :+: ("b" :+: []) =>
** [a b]
```

This is wasteful because each invocation of :+: creates a temporary list to give to <>, which is then copied and discarded.

We can also define a concatenator in terms of ::, though this time we use recursion. The idea is that in order to create list1 <> list2 we first create tl(list1) <> list2, then use hd(list1) and :: to finish the job. The recursive call on the tail of list1 will, of course, use :: repeatedly. When it gets down to list1 == [], the concatenator simply needs to return list2, since [] <> list2 = list2.

```
define 5 <+> (list1, list2);
    if list1 == [] then list2
    else
        hd(list1) :: (tl(list1) <+> list2)
    endif
enddefine;

;;; Test it:

[a b c] <+> [d e f] =>
** [a b c d e f]
```

Notice that list2 is used exactly as it is: it is not copied. However, :: creates new list links for the elements of list1.

6.11 Diagrams showing static lists represented as pairs

This section shows how (non-dynamic) lists are represented as chained collections of pairs linked together. It is based on the Poplog file TEACH BOXES, originally written by Steve Hardy.

Consider the following set of instructions in Pop-11. We first explain what each one means to a user, and then present a graphical representation of the effects on the computers memory.

```
vars x;
```

This puts an entry for the word "x" in the Pop-11 dictionary (unless it was already there), and, in the current section, associates it with an ordinary identifier record (identprops = 0). The "valof" cell in the record originally has an <undef x> record as its contents.

```
     .---.
    x! *-+----> <undef x>
     .---.
```

```
"a" -> x;
```

This creates an entry for the word "a" in the Pop-11 dictionary (unless it was already there) and stores a pointer to the word in the valof cell of the identifier record associated with "x"

```
Dictionary      Identifiers in
                Current section

 <word v>
                .========.
 <word x>----> |ident * |
                .======+=.
                       |
 <word a> <-----------+

 <word if>------>....
```

Here the notation *—-> is used to represent a ”pointer” where the ”*” indicates a bit pattern that gives the address in (virtual) memory of the item at the other end of the arrow. When a memory cell contains a pointer to the word ”a” we shall sometimes abbreviate this by displaying the word directly in the cell. Also instead of separately representing a word in the dictionary and its identifier record, we can write the word immediately to the left of a box representing its ”valof” cell. Thus the above representation of the effect of ”a” -> x can be abbreviated as follows:

```
  .===.
 x| a |
  .===.
```

We’ll use this sort of abbreviation in subsequent examples.

```
[a] -> x;
```

This is equivalent to

```
conspair("a", []) -> x;
```

It creates a new list, consisting of a single pair record, with a pointer to the word ”a” in its front and a pointer to the empty list [] in its back.

```
  .===.    .---.---.
 x| *-+-->| a | *-+----->[]
  .===.    .---.---.
```

To simplify such examples, we’ll display ”[]” in the box, instead of showing a pointer to it. So the above example becomes

```
  .===.    .---.--.
 x| *-+-->| a |[]|
  .===.    .---.--.
```

```
[how now brown cow] -> x;
```

This creates records for the four words ”how”, ”now”, ”brown” and ”cow” and puts them in the dictionary (so that they’ll be found and re-used if the words occur somewhere else). It does not create identifier records for value cells for the words, because they are merely quoted in the list, not used as variable names. It then creates a four element list, made of four list links, the first containing a pointer to ”how” in its front and a pointer to the second in its back, the second containing a pointer to ”now” in its front, and so on. The fourth link contains a pointer to [] in its back. When the list has been constructed a pointer to the first pair is put on the stack. Then the assignment removes it from the stack and puts it in the valof cell for ”x”, with the following result:

```
  .===.    .---.---.    .---.---.    .-----.---.    .---.--.
 x| *-+-->|how| *-+-->|now| *-+-->|brown| *-+-->|cow|[]|
  .===.    .---.---.    .---.---.    .-----.---.    .---.--.
```

```
[[a]] -> x;
```

This creates a list consisting of a single pair, whose front contains ”a” and whose back contains [], then creates a new list whose front points to the original list and whose back also points to [], thus:

```
  .===.    .---.--.
x|  *-+-->| *  |[]|
  .===.    .-+-.--.
             |          (Note: the backs of both pairs point to the
             v           very same item, not two copies of [].)
           .---.--.
           | a  |[]|
           .---.--.
```

```
    [[the man] kicked [a dog]] -> x;
```

This one is easier to show than to describe. It might be a good idea for the reader to draw the diagram corresponding to this before looking at the one given here.

```
  .===.    .---.---.   .------.---.    .---.--.
x|  *-+-->| *  | *-+-->|kicked| *-+-->| *  |[]|
  .===.    .-+-.---.   .------.---.    .-+-.--.
             |                            |
             v                            v
           .---.---.   .---.--.        .---.---.   .---.--.
           |the| *-+-->|man|[]|        | a  | *-+-->|dog|[]|
           .---.---.   .---.--.        .---.---.   .---.--.
```

```
    [a b c] -> x; tl(x) -> y;
```

This creates the expected two-element list containing pointers to the quoted words ”a”, ”b” and ”c” (added to the dictionary if necessary), and inserts a pointer to the first link into the valof cell for ”x”.

The procedure ”tl” then takes a copy of the pointer in the back of the first link pointed to by ”x” (i.e. its tail, or ”tl”) and the assignment puts it in the valof cell for the word ”y”, so that ”x” and ”y” now both provides route to the second pair in the list, though it can be reached more directly from ”y”.

```
  .===.    .---.---.   .---.--.      .---.--.
x|  *-+-->| a  | *-+-->| b  | * +--->| c  |[]|
  .===.    .---.---.   .---.--.      .---.--.
                          ^
  .===.                   |
y|  *-+-------------------*
  .===.
```

After the above has been constructed, the expression ”y == tl(x)” evaluates to TRUE. I.e. it is only the pointer that has been copied across: the list cell pointed to has not been copied. So if we were to assign something to hd(y) then that would also change hd(tl(x)) and vice versa.

The next example illustrates the use of the structure concatenator ”<>” when used with lists. An expression of the form list1 <> list2 creates a new list, which starts with the elements of list1 and continues with the elements of list2. However although it makes a complete copy of list1, it re-uses the list2, so that we end up with two lists sharing a common ”tail”, as above.

```
[a b] -> x; [c d] -> y; x <> y -> z;
```

This creates two two-element lists, putting pointers to the first in the valof cell for ”x”, the second in the valof cell for ”y”. It then creates a third list which starts with a *complete* copy of the list links in the first list (i.e. x) , and then instead of ending the copy with [], puts a pointer to the first link of the second list, with the following overall result:

```
 .===.    .---.---.    .---.--.
x| *-+-->| a | *-+-->| b |[]|        (Copied to start off z)
 .===.    .---.---.    .---.--.

 .===.    .---.---.    .---.--.
y| *-+-->| c | *-+-->| d |[]|
 .===.    .---.---.    .---.--.
            ^
            |
            *---------------*
                            |
 .===.    .---.---.    .---.-+-.
z| *-+-->| a | *-+-->| b | * |
 .===.    .---.---.    .---.--.
```

Note that the two occurrences of ”a” are actually two copies of a pointer to the very same word ”a” in the dictionary. Similarly the two occurrences of ”b”. So both of the following are true:

```
hd(x) == hd(z),         hd(tl(x)) == hd(tl(y))
```

Also y == tl(tl(z)) is true.

```
[a b] -> x; [c d] -> y; y -> tl(tl(x));
```

This is similar except that instead of making a copy of the two pairs in x and making the final link of the COPY point to the list in y, it puts a pointer to the second list in the back of the last link of x, producing this:

```
 .===.    .---.---.    .---.--.
x| *-+-->| a | *-+-->| b | * |
 .===.    .---.---.    .---.-+-.
                             |
            *---------------*
            |
            v
 .===.    .---.---.    .---.--.
y| *-+-->| c | *-+-->| d |[]|
 .===.    .---.---.    .---.--.
```

6.11.1 The lack of symmetry between hd and tl

Notice the lack of symmetry between hd and tl. hd(list) is the same as list(1). But tl(list) is not the same as list(2): it is a list of ALL remaining elements, or [] if there are not any more.

Simlar comments may be made about front and back when applied to lists.

6.12 Dynamic lists: generators and pdtolist

Pop-11 includes mechanisms for creating and using ‘dynamic lists’, whose elements are created as needed by a generator procedure. This is an example of what is sometimes referred to as ”lazy evaluation”.

Dynamic lists are created using the (horribly named) pdtolist procedure (derived from ”ProceDure TO LIST”). This procedure takes a ”generator procedure” as argument and produces a dynamic list as result.

6.12.1 Generator procedures

A generator procedure is a procedure that can be called repeatedly without any arguments and will produce a new result each time; and if it ever produces as its result the unique Pop-11 object termin, that signifies that there are no more items to be generated.

Here is a generator procedure that uses a private list and goes on forever generating even numbers, i.e. 0, 2, 4, 6, 8, 10, etc.

```
define gen_evens() -> next;
    ;;; create a local, constant, private list containing 0
    ;;; the number will be changed each time the procedure is run.
    lconstant store = [0];

    store(1) -> next;
    next + 2 -> store(1);
enddefine;

gen_evens() =>
** 0
gen_evens() =>
** 2
gen_evens(), gen_evens(), gen_evens() =>
** 4 6 8
```

We can use a procedure like gen_evens to create a conceptually ”infinite” list, thus

```
vars gen_list = pdtolist(gen_evens);
```

6.12.2 Printing dynamic lists

If you try to print out gen_list, Pop-11 will only show the part of the ”infinite” list that corresponds to how far the generator procedure has actually got. We can force it to generate the first N additional items by accessing the first N items of the list:

```
gen_list =>
** [...]
```

The dots indicate an unexpanded dynamic list.

```
gen_list(1) =>
** 10
gen_list(2) =>
** 12
```

Now look at the list

```
gen_list =>
** [10 12 ...]
```

Part of it is ”static”, or has been ”solidified”. But part is still dynamic, waiting to be expanded, as shown by the three dots. We can move things on:

```
gen_list(15) =>
** 38

gen_list =>
** [10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 ...]
```

Using gensym to make a dynamic list

The potential of dynamic lists can be illustrated using the procedure gensym. This takes a word as input and each time it is invoked it produces a new word got by appending a number to the word, e.g.

```
;;; Make the gensym library procedure accessible
uses gensym;
gensym("cat"), gensym("dog"), gensym("cat"), gensym("dog") =>
** cat1 dog1 cat2 dog2
```

By partially applying gensym to a word we can create a generator. By applying pdtolist to the generate we make a dynamic list.

```
1 -> gensym("pig");     ;;; make sure it starts with pig1, pig2,...

vars pig_gen = pdtolist(gensym(%"pig"%));
pig_gen =>
** [...]
```

The three dots show that the dynamic list is still unexpanded. We can force pig_gen to run the generator procedure three times, by asking for the third element of the list, after which it will print out as partially expanded:

```
pig_gen(3) =>
** pig3

pig_gen =>
** [pig1 pig2 pig3 ...]
```

To restart gensym, use cleargensymproperty().

```
cleargensymproperty();

;;; This will affect the contents of the dynamic list
pig_gen(4) =>
** pig1
pig_gen(10) =>
** pig7
pig_gen =>
** [pig1 pig2 pig3 pig1 pig2 pig3 pig4 pig5 pig6 pig7 ...]
```

Note that gensym will be replaced after V14.5

From Poplog V14.5, the procedure gensym is superseded by the procedure gen_suffixed_word, with the same functionality, and the action

```
cleargensymproperty()
```

is replaced by by

```
clearproperty(gen_suffixed_word_prop)
```

See REF *gen_suffixed_word, after V14.5

6.12.3 Accessing components of dynamic lists: hd, tl, dest

The procedures front, back and destpair would not work on on a dynamic list as expected. Instead we need procedures that can check whether a list is dynamic or not, and if it is, then run the generator procedure if necessary to get another element of the list, then create a static initial portion of that list, leaving the generator on the end.

The procedures hd, tl and dest are designed to do exactly that. Otherwise they work like front, back, and destpair respectively.

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

So nearly all the general purpose procedures for operating on lists are implemented using hd and tl and dest rather than front and back and destpair, so that they can also work on dynamic lists.

An example of a procedure that will NOT work on dynamic lists is matches. There are also certain "fast" procedures, described in the files HELP EFFICIENCY and REF FASTPROCS, that operate on static lists but not on dynamic lists. Their use can lead to obscure errors if you suddenly start using dynamic lists!

null(list) vs list == []

Another case that has to be handled is the case of a dynamic list that is empty.

Let's define a procedure that produces nothing but termin.

```
define gen_nothing() -> result;
    termin -> result;
enddefine;

gen_nothing(), gen_nothing(), gen_nothing() =>
** <termin> <termin> <termin>
```

If used to create a dynamic list this will effectively create an empty list:

```
vars nothing_list = pdtolist(gen_nothing);

nothing_list =>
** [...]
```

Now if you try to tell whether this is an empty list by comparing it with [], you'll get the wrong result:

```
nothing_list == [] =>
** <false>
```

The procedure null, however, recognizes both [], and this dynamic list as empty lists:

```
null([a b]) =>
** <false>
null([]) =>
** <true>
null(nothing_list) =>
** <true>
```

And it changes the internal representation of the dynamic list to tell the Pop-11 print routines that the generator is finished, so that now it prints AS IF it were the same as []

```
nothing_list =>
** []
```

But it is not

```
nothing_list == [] =>
** <false>
```

It's a null dynamic list.

So if you are going to sometimes use dynamic lists then define all your procedures to use hd, tl, and null rather than front, back and == [].

6.12.4 The representation of dynamic lists

Readers interested in knowing how dynamic lists are actually represented should read the Poplog file REF LISTS. Essentially they depend on the fact that if the final pair in the chain of lists contains as its back a procedure rather than [], and a non-false front, then the list is taken to be dynamic. However if the generator procedure has already produced termin, then the representation is changed by adding a pair whose front is false and whose back is the generator procedure.

We can look at the contents of nothing_list by using the procedure destpair, which, when applied to a pair, produces its front and its back:

```
destpair(nothing_list) =>
** <false> <procedure gen_nothing>
```

By contrast, the procedure dest, which works on lists which may be

dynamic, and returns the hd and the tl of the list, will complain about an empty dynamic list:

```
dest(nothing_list) =>
;;; MISHAP - NON-EMPTY LIST NEEDED
;;; INVOLVING:  []
;;; DOING    :  dest ...
```

6.12.5 The uses of dynamic lists

Dynamic lists are often useful for mathematical purposes using number generators of various kinds. Another use is to represent a stream of input to the computer. In Pop-11 input devices are often ”converted” to generator procedures (sometimes called ”producers”) which, each time they are called produce the next item of input (if it is ready, otherwise they usually wait).

The Pop-11 compiler, called ”compile” (named ”popval” in earlier versions of Pop-11 and Pop-2), is able to take a list of text items (consisting of words, strings, numbers, and possibly other objects) and cause them to be compiled into instructions for the computer.

This list is called ”proglist” and is described in detail in the files REF PROGLIST and REF POPCOMPILE. It is possible for the list to be a simple static list, as in

```
compile([3 + 3 => ]);
** 6
```

More usefully, when a file is being compiled the text items in the file are transformed into a dynamic list, which is then compiled. It would be possible in principle to read in the whole file and make a huge static list, then compile that, but usually that’s not sensible as a lot of space will be wasted while the early parts of the file are being compiled.

It is also useful when Pop-11 is being used interactively to treat the terminal as if it were an infinite file of text items. This is done by creating a dynamic list containing the items that are being typed in. This is based on a generator procedure that returns an item at a time, which it obtains from a generator procedure (charin) that returns a character at a time from the terminal, each time it is applied. You can test it interactively by giving this command:

```
charin(),charin() =>
```

It will prompt you for input. You can type a letter or number followed by the RETURN key. It will then print out two numbers, one being the ascii code for the letter or number, and the second being the ascii code for a newline, i.e. 10. (There is a separate character repeater for 'raw' mode interaction used by the editor, which does not wait for you to type RETURN after typing another character.).

The item repeater for interactive Pop-11 is created by applying the system procedure incharitem to charin, thus:

```
incharitem(charin)
```

This creates a new procedure, which is an item repeater. Each time the item repeater is run it consumes enough characters to make a complete text item, and it returns that item, and then the next time it is run it returns the next item, and so on.

The dynamic list constructor pdtolist, can be applied to this item repeater, and that will create a dynamic list of program text items. That, in fact, is what proglist is when programs are being compiled either from a file or from the terminal. I.e.

```
pdtolist(incharitem(charin)) -> proglist;
```

Because proglist is a dynamic list, if the compiler is reading in an expression of some kind and the user has not finished typing it, the compiler simply waits till more is typed (every time the user presses the RETURN key, the items typed so far are made available as an extension to proglist).

Note that the above is highly modular: instead of compiling from the current input stream Pop-11 can compile from any sequence of characters, including the characters stored in a string, as illustrated by the following command, which uses stringin to turn a string into a character repeater:

```
compile(pdtolist(incharitem(stringin('66*33 =>'))));
** 2178
```

You can experiment with an infinite list connected to the current input stream as follows:

```
vars inputlist = pdtolist(incharitem(charin));

inputlist =>
** [...]
```

An example of an infinite list of input

This example will work well only in the editor.

The following will prompt for input until you have typed enough for three full text items. Suppose you respond by typing "the cat" on one line, followed by RETURN, followed by "on" on the next line:

```
inputlist(3) =>
** on
```

You can now print out the current state of inputlist:

```
inputlist =>
** [the cat on ...]
```

This will now prompt for two further input items:

```
    inputlist(5) =>
    inputlist =>
```

For more information on dynamic lists and the compiler input stream see REF LISTS, and REF PROGLIST

For writing interactive programs it is not usually convenient to use charin and proglist. Generally it is safer for beginners to use the built in procedure readline, illustrated in TEACH RESPOND, which might be defined thus:

```
global vars pop_readline_prompt = '? ';
define readline() -> list;
    lvars item, list,
        procedure rep = incharitem(charin);     ;;; item repeater

    ;;; temporarily change two global variables.
    dlocal
        popnewline = true,  ;;; make newlines recognizable
        popprompt = pop_readline_prompt;

    ;;; Make a list items to next newline
    [% until (rep() ->> item) == newline do item enduntil %] -> list;

enddefine;

;;; Test it
readline() =>
? pretty polly pretty polly
** [pretty polly pretty polly]


'Please say something: ' -> pop_readline_prompt;

readline() =>
Please say something: how are you today?
** [how are you today ?]
```

In the last example the words after ":" were typed in, and made into a list.

6.13 Some procedures for manipulating lists

We have already met some expressions denoting lists. This section will introduce some of the procedures used for manipulating lists, including new procedures for creating new lists, or modified versions of old ones.

6.13.1 cons, conslist, initl, sysconslist

```
cons(ITEM, LIST) -> LIST
```

This is strictly equivalent to ITEM :: LIST. It is included in Pop-11 merely as an analogue for the function CONS in Lisp.

```
conslist(ITEM1, ITEM2, ..., ITEMN, N) -> LIST
```

Returns a list constructed from the top N items on the stack. E.g.

```
conslist("a", "b", [c d], 3) =>
** [a b [c d]]

conslist(#| vars i; for i from 2 to 13 do i*i endfor|#) =>
** [4 9 16 25 36 49 64 81 100 121 144 169]
```

```
initl(N) -> LIST
```

This constructs a list of empty lists, of length N. E.g.

```
initl(6) =>
** [[] [] [] [] [] []]
```

```
sysconslist() -> LIST
```

Collects all items placed on the stack since the last stackmark, into a list. This is used by Pop-11 list constructor syntax. E.g.

```
[% 1, 2, 3, 4 %]

is equivalent to

popstackmark, 1, 2, 3, 4; sysconslist();
```

See REF LISTS, for a description of sysconslist_onto(LIST_1), which is used when ^^ precedes the last item in a list expression.

6.13.2 allbutfirst, allbutlast

```
allbutfirst(N, LIST) -> SUB_LIST
allbutlast(N, LIST) -> SUB_LIST
```

These two can be used to "chop off" an initial or final segment of a list, of length N.

```
allbutfirst(2, [a b c d e]) =>
** [c d e]
allbutlast(2, [a b c d e]) =>
** [a b c]
```

In the first case, the result will share list links with the input list.

6.13.3 dl or explode, destlist

The procedure dest (or destpair) when given a list puts its and its tail on the stack. Sometimes it is necessary to put all the elements of a list on the stack, e.g. in order to use them to make a copy of the original list. The procedure explode will do this.

```
explode([a b [c d] e f]) =>
** a b [c d] e f
```

However, explode works on a variety of different structures, including words, strings, vectors and lists. A version specific to lists is also available, known as ”dl”.

```
dl([a b [c d] e f]) =>
** a b [c d] e f
```

destlist is similar, except that it also returns the number of items in the list. It's the reverse of conslist.

```
destlist([a b [c d] e f]) =>
** a b [c d] e f 5
```

6.13.4 applist, maplist, ncmaplist

Both of these take a list and a procedure and apply the procedure to every element of the list. The only difference is that maplist makes a list of everything put on the stack.

```
applist([1 2 3 4 5], nonop +(% 10 %)) =>
** 11 12 13 14 15

maplist([1 2 3 4 5], nonop *(% 10 %)) =>
** [10 20 30 40 50]
```

ncmaplist, non-constructive maplist, is like maplist, except that it re-uses the links of its first argument.

6.13.5 recursive_front

This procedure can be applied to any object. If it's a list or pair, it repeatedly applies the procedure front, until a non-pair is found, and returns that as its result. This is useful for digging out items deeply embedded in lists.

```
recursive_front([[[[[a]  b ] c]] d]) =>
** a
```

6.13.6 expandlist

```
expandlist(LIST) -> LIST
```

Returns LIST unchanged, unless the list is dynamic, in which case it runs the generator procedure to completion and makes the list static. It will loop forever if the list is of infinite length.

6.13.7 rev and ncrev

```
rev(LIST) -> LIST
```

rev, when given a list, produces a new version which has the same elements in reverse order:

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

The procedure does not alter the order of elements in the original list: that is left unchanged.

```
ncrev(LIST) -> LIST
```

This is like rev, except that it re-uses the list links of the original list, so that it does not use any new store. This can reduce garbage collections but is dangerous if there is any risk that something else was dependent on the original list surviving unchanged.

```
vars list1 = [a b c], list2 = ncrev(list1);
list2 =>
** [c b a]
list1 =>
** [a]
```

6.13.8 setfrontlist

```
setfrontlist(ITEM, LIST_1) -> LIST_2
```

Returns LIST_2 formed by moving the ITEM to the front of LIST_1, or adding the ITEM if not already present. (This is used by the editor VED whenever a file in vedbufferlist becomes ”current”.)

6.13.9 sort and syssort

```
sort(LIST_1) -> LIST_2
```

The list should contain either numbers only or words only or strings only or a mixture of words and strings. Any other items will produce an error. It returns a list of sorted items. If it contains numbers only, then the result is equivalent to

```
syssort(LIST_1, nonop <)
```

otherwise

```
syssort(LIST_1, alphabefore)

sort([the cat sat on the mat]) =>
** [cat mat on sat the the]

sort([ 111 222 33 ]) =>
** [33 111 222]
```

```
syssort(LIST, P) -> LIST
syssort(LIST, BOOL, P) -> LIST
```

The first argument is a list, the last argument is a procedure which takes two items and returns a boolean result (e.g., nonop < for numbers or -alphabefore- for string and words, etc). The items 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 non-copying, and merely re-arranges the elements of the argument list, like ncrev. A merge sort algorithm is used. Example

```
syssort([[the] [cat] [sat] [on] [the] [mat]],
            procedure(l1, l2);
                alphabefore(hd(l2), hd(l1))
            endprocedure) =>
** [[the] [the] [sat] [on] [mat] [cat]]
```

6.13.10 last, lastpair

last finds or updates the final element of a list:

```
last([cat dog mouse]) =>
** mouse

last([[tom brown] [mary green] [suzy white]]) =>
** [suzy white]
```

Note that in the latter example, the procedure LAST was applied to a list of lists, and produced as its result a list, i.e. the last list.

```
vars list = [a b c d];
list =>
** [a b c d]

999 -> last(list);
list =>
** [a b c 999]
```

NOTE: last can also work on words, vectors and strings, though the updater cannot be used with words.

```
lastpair(LIST) -> PAIR
PAIR -> lastpair(LIST)
```

Returns or updates the last PAIR of the list LIST, i.e. the last link in the chain. LIST cannot be null. Having a pointer to the last pair of a list, instead of the last item in the list makes various things much simpler. For example, here is one way to define a queue:

```
vars queue = [a b c d], queue_start = lastpair(queue);

queue_start =>
** [d]
```

To add something to the queue at the far end, do this:

```
conspair("e", []) -> back(queue_start);
back(queue_start) -> queue_start;
queue =>
** [a b c d e]
queue_start =>
** [e]
```

Remove something at the left end:

```
tl(queue) -> queue;
queue =>
** [b c d e]
```

6.13.11 oneof, shuffle

oneof applied to a list chooses an element at random. Thus using it repeatedly on the same list may or may not produce different results:

```
repeat 4 times
    oneof([ [you are gorgeous]
            [everyone loves you]
            [how masterful you are]
            [you are quite stunning]
        ]) =>
endrepeat;
** [you are quite stunning]
** [everyone loves you]
** [you are quite stunning]
** [you are quite stunning]
```

```
shuffle(LIST_1) -> LIST_2
```

Returns a copy of its argument with the elements randomly re-ordered. It uses -oneof-.

```
repeat 4 times shuffle([a b c d e]) => endrepeat;
** [b d c a e]
** [b d a c e]
** [b e c d a]
** [d c e b a]
```

Since oneof and shuffle are defined in terms of random, illustrated previously, (in Chapter 5) their behaviour can be controlled by assigning to the variable ranseed, as in the case of random.

6.13.12 delete, ncdelete

delete produces copies of a list which do not contain a certain element. It has several different formats

```
delete(ITEM, LIST_1)          -> LIST_2
delete(ITEM, LIST_1, EQ_P)    -> LIST_2
delete(ITEM, LIST_1, N)       -> LIST_2
delete(ITEM, LIST_1, EQ_P, N) -> LIST_2
```

These all delete occurrences of ITEM from LIST_1, producing a new list LIST_2 (which shares the largest possible trailing sublist of the original).

The parameter EQ_P is an optional argument; if supplied, it must be a procedure of the form

```
EQ_P(ITEM, LIST_ELEMENT) -> BOOL
```

EQ_P is then used to compare ITEM against each list element, and those for which it returns true are deleted. If not supplied EQ_P defaults to nonop = (i.e. structure equality). (See HELP EQUAL)

N is a second optional argument: if supplied, it is an integer >= 0 which specifies how many matching elements should be deleted (e.g, if 1 then only the first occurrence will be removed). If not supplied, all occurrences are deleted.

For example,

```
delete(1, [1 2 3 4 5 6 1 9 8]) =>
** [2 3 4 5 6 9 8]
delete(1, [1 2 3 4 5 6 1 9 8], 1) =>
** [2 3 4 5 6 1 9 8]

delete("cat", [mouse cat dog flea], nonop == ) =>
** [mouse dog flea]

delete('cat', ['mouse' 'cat' 'dog' 'flea'], nonop == ) =>
** [mouse cat dog flea]

delete('cat', ['mouse' 'cat' 'dog' 'flea'], nonop = ) =>
** [mouse dog flea]
```

(The difference between the last two is due to the fact that two strings can be "=" but never "==").

```
ncdelete(ITEM, LIST_1)          -> LIST_2
ncdelete(ITEM, LIST_1, EQ_P)    -> LIST_2
ncdelete(ITEM, LIST_1, N)       -> LIST_2
ncdelete(ITEM, LIST_1, EQ_P, N) -> LIST_2
```

Non-constructive delete. Same as delete, but does not copy list pairs that need to be changed, and thus (may) destructively change the original list. The result LIST_2 will be == to LIST_1 unless there are one or more leading matching occurrences of ITEM that are deleted.

6.13.13 flatten and flatlistify

```
flatten(LIST_1) -> LIST_2
```

Explodes LIST_1 and all sub-lists in LIST_1. I.e. it converts a tree into "flat" list of elements at the "fringe" of the tree.

```
flatten([ a [ b c [d e] [f]] g [ h [i] ] j]) =>
** [a b c d e f g h i j]
```

```
flatlistify(STRUCT) -> LIST
```

Given a structure, STRUCT, made of lists and/or vectors embedded arbitrarily, -flatlistify- will return a list, LIST. The result contains all the words needed to create a list isomorphic with the original one, if given to compile.

```
flatlistify([ a [ b c [d e] [f]] g [ h [i] ] j]) =>
** [a [ b c [ d e ] [ f ] ] g [ h [ i ] ] j]
```

NB. That is a 20 item list containing "a", "[", "b", etc.

6.13.14 length and listlength

These can both be applied to a list and will return the number of elements. The difference is that length can be applied to many other types of objects besides lists, e.g. strings, vectors.

```
listlength([a [ b c d] e ]) =>
** 3
listlength({ a b c }) =>
;;; MISHAP - LIST NEEDED
;;; INVOLVING:  {a b c}
```

6.13.15 copy, copylist, copydata, copytree

The procedure copy when given a Pop-11 data-structure returns a copy that is = to the original. In the case of lists, it merely copies the first link, returning a new pair with the same front as the oldone and the same back as the old one. This means that the two lists share tails.

```
vars list1 = [a b c d], list2 = copy(list1);
list2 =>
** [a b c d]

33 -> list1(3);
list1 =>
** [a b 33 d]
list2 =>
** [a b 33 d]
```

An assignment to the first element of a will not affect list2, and vice-versa, but replacing any other element of either will affect the other.

In order to avoid this what is needed is a complete copy of the original list, made entirely of new list links. The procedure copylist is provided for that purpose. The above example may be tried with copylist instead of copy.

Even if copylist did not exist, there are several ways it could be defined by users, including the following:

```
define copylist1(list);
    maplist(list, identfn)
enddefine;

define copylist2(list);
    [% appplist(list, identfn) %]
enddefine;

define copylist3(list);
    [% explode(list) %]
enddefine;

define copylist4(list);
    conslist(#| explode(list) |#)
enddefine;

define copylist5(list);
    conslist(destlist(list))
enddefine;
```

```
copytree(LIST_1) -> LIST_2
```

This makes a list, LIST_2, which is a copy of LIST_1. Any elements of LIST_1 which are themselves lists are recursively copied.

copydata is a generalisation of copytree that works on arbitrary datastructures. Thus if a list contains vectors containing lists, etc. then, in order to obtain a completely new copy use copydata, not copylist.

6.13.16 subscrl, fast_subscrl

```
subscrl(N, LIST) -> ITEM
ITEM -> subscrl(N, LIST)
```

Returns or updates the N-th element of the list LIST (where the first element is has subscript 1). Because this procedure is the -class_apply- procedure of pairs, this can also be used in the form

```
LIST(N) -> ITEM
ITEM -> LIST(N)
```

```
fast_subscrl(N, LIST) -> ITEM
ITEM -> fast_subscrl(N, LIST)
```

This is like subscrl, but does not check argument data types, and does not expand dynamic lists.

6.14 Predicates on lists

6.14.1 atom, islist, ispair, islink, null

```
atom(ITEM)-> BOOL
```

This will be true for all items except pairs, and false for pairs. This is therefore equivalent to not(ispair(ITEM)).

```
islist(ITEM) -> BOOL
```

This is intended to recognize the empty list [], dynamic lists, or ordinary lists. However, for efficiency it does not check that all the links in the chain have lists in their tail, and so it can be fooled. e.g.

```
islist(conspair(3,4)) =>
** <false>
islist(conspair(2,conspair(3,4))) =>
** <true>
```

The latter is not really a list because it does not end in []. A "proper" list recognizer would be something like this

```
define is_really_list(item);
    if islist(item) then
        if null(item) then true
        elseif atom(item) then false
        else is_really_list(tl(item))
        endif
    else
        false
    endif
enddefine;

is_really_list(conspair(2,conspair(3,4))) =>
** <false>
is_really_list(conspair(1, conspair(2,conspair(3,4)))) =>
** <false>
is_really_list(conspair(1, conspair(2,conspair(3,[])))) =>
** <true>
```

The above definition could be optimised slightly by checking for dynamic lists directly on the basis of the definition of dynamic lists instead of using islist to do that.

```
ispair(ITEM) -> BOOL
```

A recognizer for pair datastructures.

```
islink(ITEM) -> BOOL
This is equivalent to
        ispair(TIEM) and not(null(item))
```

```
null(ITEM) -> BOOL
```

This recognizes empty lists, including empty (exhausted) dynamic lists. If ITEM is a dynamic list that has not been expanded it will expand it by one step, so that front and back can be used on that link instead of hd and tl, in order to avoid repeating the test for a dynamic list.

6.14.2 isdynamic

```
isdynamic(ITEM) -> P
```

Recognizes dynamic lists. If ITEM is a dynamic list it returns the generator procedure, otherwise it returns false.

6.14.3 member, lmember

```
member(ITEM, LIST) -> BOOL
```

This procedure returns true if ITEM is an element of the list LIST, otherwise false, equality being determined with the operator "=".

```
lmember(ITEM, LIST) -> SUB_LIST
  This is like member except that
    1. The test for equality is "==" not "="
    2. If ITEM is found to be == to an element of the list, then the
       procedure returns the trailing portion of LIST starting  with
       that element

    lmember("on", [the cat sat on the black mat]) =>
    ** [on the black mat]
```

6.14.4 user defined predicates

In a language like Pop-11 it is very easy to define additional list processing procedures. For example, a slight variant of maplist would take a list and two procedures and return a list of two element lists:

```
    define map2list(list, proc1, proc2) -> list;
        [%
            lvars item;
            for item in list do [%proc1(item), proc2(item)%] endfor
        %] -> list
    enddefine;

    map2list([1 -2 3 -4 5 -6], identfn, negate) =>
    ** [[1 -1] [-2 2] [3 -3] [-4 4] [5 -5] [-6 6]]
```

A procedure to merge two lists by interleaving elements:

```
    define merge(list1, list2) -> list;
        lvars item1, item2;
        [%
            for item1, item2, in list1, list2 do item1, item2, endfor
        %] -> list
    enddefine;

    merge([1 2 3 4], [a b c d]) =>
    ** [1 a 2 b 3 c 4 d]
```

Note if either list has more elements than the other this procedure will ignore trailing items in the longer list.

The variety of possible user defined list manipulating procedures is endless. In Poplog Pop-11 several examples can be found in the library directories, i.e. $usepop/pop/lib/auto and $usepop/pop/lib/lib

6.15 Iterating on lists

The main Pop-11 iteration constructs were given in an earlier chapter. Here we give a number of examples of iteration involving lists. Using iteration we can easily perform a subset of the operations previously illustrated using recursion. Iteration is sometimes easier to understand, and can be more efficient (i.e. easier for the computer). It has the disadvantage that you cannot use TRACE to make explicit what is happening when the program runs.

There are many different forms of iteration over lists. The simplest is a serial scan over the elements of the list. Here is a new definition of TRAVERSE, using an UNTIL loop.

```
define traverse(list);
   until list = [] do
     hd(list) =>
     tl(list) -> list
   enduntil;
 enddefine;
 traverse([a b c d]);
 ** a
 ** b
 ** c
 ** d
```

6.16 Using for ... in ... do .... with lists

E.g., to add all the numbers in a list

```
define addall(list) -> total;
    lvars x, total = 0;
    for x in list do
        x + total -> total
    endfor
enddefine;

addall([3 5 7 9]) =>
** 24
```

6.16.1 Using for inside [% ..... %]

We could use the FOR construct to define a procedure like delete. We use a local variable, say ‘x’, to denote successive elements in a list. If the element is not the same as the item to be deleted, then leave the element on the stack. If all this is done inside the list brackets, using ‘%’, then the items left on the stack will be made into a list.

```
define new_delete(item, list) -> result;
    lvars x;
    [% for x in list do
         unless x = item then x /* left on stack */  endunless;
       endfor
    %] -> result
enddefine;

new_delete([a b], [[1 2] [3 4] [a b] [c d]]) =>
** [[1 2] [3 4] [c d]]
```

6.17 Iterating over two or more lists

if list1, list2, list3, ... listN, are lists, then it is possible to use N variables and iterate over all the N lists at once, thus:

```
for item1, item2, ... itemN in list1, list2, ... listN do
    <instructions using item1, item2, ...itemN >
endfor
```

The instructions will be obeyed N times, each time taking one item from each list. If any lists are shorter than the shortest list, then their trailing items will be ignored.

E.g.

```
vars item1, item2, item3;
for item1, item2, item3 in
    [the every each all some],
    [men pig cow cars],
    [stank slept fell ate]
do
    [^item1 ^item2 ^item3] =>
endfor;
** [the men stank]
** [every pig slept]
** [each cow fell]
** [all cars ate]
```

6.18 Iteration vs Recursion

Instead of using FOR, it is always possible to recurse on the TL of the list. The recursive technique has the great advantage that successive calls of the procedure can be TRACEd. Constructions utilising loops (REPEAT, WHILE, UNTIL, FOR) do not admit of easy monitoring through TRACE.

Often, however, it is very quick and easy, using VED, to temporarily alter the definition so that each time round the loop the procedure prints something (e.g. the value of ‘x’ in the last example). In a system such as Poplog altering and recompiling a procedure is so fast that temporary changes for debugging purposes can be very much more useful than in a conventional language.

Moreover, if a looping construct is used, then the list brackets [...] (or the vector brackets { ..... }) can be used to collect together all the items left on the stack into a list, if that is desired. (This is one of the features of Pop-11 not available in LISP, which uses alternative mechanisms.) Compare the following two different definitions of procedures equivalent to maplist.

```
define maplist1(list, proc);
    if null(list) then []
    else
        proc(hd(list)) :: maplist1(tl(list), proc)
    endif
enddefine;

define maplist2(list, proc);
    lvars item;
    [%for item in list do proc(item) endfor%]
enddefine;
```

The former uses the so-called "functional" style of programming. However, I have no doubt that the second is easier to understand and that programmers using the second style will make fewer mistakes.

Another example: here are two ways make a list of all the non-empty tails of a list:

```
define all_tails1(list);
    if null(list) then []
    else
        list :: all_tails1(tl(list))
    endif
enddefine;

define all_tails2(list);
    lvars tail;
    [%for tail on list do tail endfor%]
enddefine;

all_tails1([ a b c d ]) =>
** [[a b c d] [b c d] [c d] [d]]
```

An advantage of the recursive version is that when it is traced it shows clearly what is going on, whereas extra printing instructions have to be put into programs using for ... endfor to show what is happening on each iteration.

Try

```
trace all_tails1;
```

then redo the above command. (See TEACH TRACE)

Using the integrated editor VED makes it very quick and easy to add or remove printing instructions and recompile, so the difference in traceability is not very great.

6.19 Exercises

What should the following print out? Which of the expressions produce words, and which produce lists? (Work out your won answers and write them down before running the commands in Pop-11).

```
last(hd([[tom brown] [mary green] [suzy white]])) =>

last(last([[tom brown] [mary green] [suzy white]])) =>

hd(hd([[tom brown] [mary green] [suzy white]]))=>

hd(tl([[tom brown] [mary green] [suzy white]]))=>

tl(hd([[tom brown] [mary green] [suzy white]]))=>

tl(tl([[tom brown] [mary green] [suzy white]]))=>
```

6.20 More exercises on lists

```
1. Write down some examples of expressions denoting:
    a list of numbers
    a list of words
    a list of numbers and words
    a list of lists of words
    a list of lists of numbers

2. What is denoted by each of the following:

    hd( [ once upon a time ] )
    last( [ mary had a little lamb ] )
    hd( [ [mary had] [a little lamb] ] )
    rev( [ mary had a little lamb ] )
    last( hd( [ [ mary had ] [ a little lamb ] ] ) )
    delete( "cat", [mouse pig cat dog cat] )
    delete( "cat", [mouse pig cat dog cat], 1 )


3. What does oneof do?
    NB. Your answer should take the form:
    ONEOF is a procedure which takes one argument (or one input
    item) and produces one result. The argument must be a ....
    The result will be .....

4. What does delete do?
    (I.e. how many arguments does it take? What sorts of things
    can they be? How many results does it produce? How is the
    result related to the arguments?)

5. What do the following denote:

    delete(3, rev([ 1 2 3 4 5]) )

    rev( delete( "little", [mary had a little lamb] ) )
```

6.21 Exercises on the meaning of the single and double up-arrows

Here are some puzzles to test your understanding. If we assign [a b] to x, thus:

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

then the list [^^x c] is [a b c] and the list [^x c] is [[a b] c].

What values would have to be assigned to x, y etc so that the following were true (some are impossible - which ones?):

```
[^^x ^^x] = [a b a b]          ;;; answer is [a b] -> x;
[^x ^^y] = [[a b] b c]         ;;; answer is [a b] -> x; [b c] -> y;
[^^x mother ^^y] = [i love my mother]
[the height of steve is ^^x] = [the height of steve is 70 inches]
[every ^^x is a ^^y] = [every fire man is a civil servant]
[every ^x is a ^y] = [every fire man is a civil servant]
[^^x i ^^y you] = [sometimes i hate people like you]
[[^x ^^y] ^^z] = [[a b c d]]
[^x [^^y] ^z] = [[a b] [c d] [e f]]
[i saw ^^n ships] = [i saw 3 ships]
[i saw ^n ships] = [i saw 3 ships]
[i ^x you] = [i hate computers]
[^x ^y ^z] = [i hate computers]
```

Use the computer to check your answers. For example, if you think the answer to the fourth one is:

```
[6 feet] -> x;
```

then try printing the list:

```
[the height of steve is ^^x] =>
```

Later you will see that the procedure MATCHES could be used to find the answers.

7 CHAPTER.7: THE POP-11 PATTERN MATCHER AND DATABASE

We have already seen how the equality symbol "=" can be used to compare two lists. The operation MATCHES provides more sophisticated facilities for matching lists against a partially specified pattern.

(In later versions of Pop-11 the equality operator "=" will be generalised to include this matching capability, based on a new kind of matchvar data-type).

The pattern matcher makes it possible to express some complex ideas in a way that is both much clearer and easier to read than normal procedural formulations, and also easier to get right first time when programs are being developed. So for many kinds of programs it can substantially speed up the development time. There are additional facilities based on the pattern matcher in the Pop-11 database, and these add to the power of the language for solving quite complex problems, including developing expert systems.

The matcher can be used in several different formats. The most common is an expression of this form, which always evaluates to TRUE or FALSE, depending whether the list does or does not match the pattern.

```
    <list> matches <pattern>

;;; Simple examples, using the matcher like an equality tester:

    [a b c d] matches [a b c d] =>
    ** <true>

    [a b c d] matches [d b c a] =>
    ** <false>
```

The use of pattern elements allows the matcher to perform more complex comparisons. The pattern matcher assumes that information is stored in lists whose contents can be searched by specifying a pattern (a sort of template list) containing special pattern elements described below. Some of these pattern elements are used to match individual items in a list. Others can match arbitrarily long subsets of lists and are called "segment" pattern elements. We illustrate the power of segment pattern elements first.

7.0.1 The anonymous segment pattern element: "=="

We start by illustrating the use of the segment pattern element "==" to create templates that can be used to match lists.

The symbol "==", which we have previously met as an equality symbol can be used in patterns with an entirely different function, to represent an unspecified number of unknown elements. For example both the following complex expressions denote TRUE

```
    [who is the father of joe] matches [who is == ]

    [where is the father of joe] matches [where is == ]
```

The following also denote TRUE:

```
    [you are my favourite programming pupil] matches [== pupil]

    [the little dog laughed to see such fun] matches [== dog ==]
```

7.1 Using matches to define a procedure

MATCHES is a Pop-11 procedure which takes two arguments, a list and a pattern (which is also a list), and produces one result, a boolean, i.e. either TRUE or FALSE. The name of the procedure "matches" is defined to be an infix operator (like "=", "+", "<>" and "::") so that it can be used in the form

```
list matches pattern
```

It takes two arguments, both lists, and returns one result, a boolean. The first argument is called the DATUM (what is given) and the second the PATTERN, against which the datum is to be compared.

The pattern may contain special pattern elements, of which "==" is a simple example. Other types of pattern elements are explained below. Notice the asymmetry: the pattern must be the second argument, never the first.

To illustrate, here is a procedure which uses MATCHES to decide whether a question requires a person or a place as its answer:

```
define type_of_answer(list) -> type;
    if     list matches [who is ==] then
        "person" -> type
    elseif list matches [where is == ] then
        "place" -> type
    else
        "undef" -> type
    endif
enddefine;
```

The procedure type_of_answer takes as its input a list, and produces a word as its result. The word is one of "person" "place" "undef". Notice the use of the multi-branch conditional form:

```
if condition1 then
    action1
elseif condition2 then
    action2
else
    default action
endif
```

An expression using "matches" can be used as a condition because it always produces a boolean result. We can test the above definition thus:

```
type_of_answer([who is your father]) =>
** person

type_of_answer([where is your father]) =>
** place

type_of_answer([is Joe your father]) =>
** undef
```

”undef” is a word often used in in Pop-11 to indicate something unknown.

Of course, this definition is not at all adequate to recognizing requests for information about persons or places, e.g.

```
type_of_answer([where was joe born]) =>
** undef
```

The matching procedure MATCHES is used by the database procedures ADD, REMOVE, PRESENT, LOOKUP, and FLUSH and the database looping constructs FOREACH and FOREVERY, about which more is said below. These provide very useful problem-solving tools for both AI programming and other kinds of programming requiring manipulation of complex, changing structures.

7.2 Exercise

Using the previous definition as a model, define a procedure called FRIENDLY, which takes a list of words as input, and produces a result which is TRUE, or FALSE or UNDEF. The result should be TRUE if the list contains the words ‘LIKES YOU’ and FALSE if the list contains the words ‘HATES YOU’, and UNDEF if it contains neither. The definition can start:

```
define friendly(list) -> result;
```

Hint: you’ll need to use ”==” as a pattern element more than once in the same pattern.

You may test your definition with commands like:

```
friendly([everybody hates you today]) =>
friendly([father christmas likes you ]) =>
friendly([I can not bring myself to like you]) =>
```

What results should these print out?

7.3 Use of the matcher to extract the contents of a list

In the previous uses of the matcher with ”==” the portion of the list that corresponded to ”==” was ignored. We might want to revise the above procedure so that it extracted that portion and returned it as a result. For that we use a ”segment variable” pattern element, in which a variable name is preceded by ”??”. The use of ”??” indicates that the variable should match an arbitrary number of items in the datum, and if the whole match is successful a list containing those items should be assigned to the variable. Here is an example, in which we change the above procedure to return two results, the type of question, and the contents of the question. Note that we have to use ”vars” to declare a variable as a pattern variable:

```
define type_of_answer(list) -> (type, contents);

    vars remainder;

    if     list matches [who is ??remainder] then
        "person" -> type;
```

```
        remainder -> contents;
    elseif list matches [where is ??remainder ] then
        "place" -> type;
        remainder -> contents;
    else
        "undef" -> type;
        [] -> contents;
    endif
enddefine;
```

This can now be tested:

```
type_of_answer([why can pigs fly]) =>
** undef []

type_of_answer([who is the murderer]) =>
** person [the murderer]

type_of_answer([where is the murderer]) =>
** place [the murderer]
```

7.4 The use of ”?” variables

The variable preceded by ”??” in the above example matched a segment of the list. We can also use a variable to match a single element of a list, and have that element assigned as its value. For example, here is a way of getting the third element of a list assigned to the variable ”item”. We use ”=” as an anonymous pattern element to match exactly one element of the datum without saving the value and ”?item” to match one element and save the value in the variable item.

```
vars item;
[ the cat sat on the mat] matches [ = = ?item ==] =>
** <true>

item =>
** sat
```

Here we use the matcher to extract the third and fourth elements of a list:

```
vars third, fourth;
[the cat sat on the black mat] matches [= = ?third ?fourth ==] =>
** <true>

third =>
** sat
fourth =>
** on
```

7.4.1 Example: Using the matcher to define next_item

Consider how to define a procedure that searches down a list looking for a given item, and if it finds it returns the next item. You could write that using normal list processing procedures as follows:

```
define next_item(item, list) -> next;
    repeat
        if list = [] then
            false -> next;
            return();
        elseif hd(list) = item then
            ;;; found the target, get the next item and stop
            hd(tl(list)) -> next;
            return()
        else
            ;;; move one step down the list.
            tl(list) -> list;
        endif
    endrepeat
enddefine;
```

Here is a version using the pattern matcher.

```
define next_item(item, list) -> next;
    vars found;  ;;; this is to be used as a pattern element

    if list matches [ == ^item ?found ==] then
        found -> next
    else
        false -> next
    endif
enddefine;
```

Note that we use ”ˆitem” as explained previously to insert the value of the variable item in the pattern. This means that instead of our programs always having fixed patterns they can have dynamically constructed patterns.

Now test the procedure

```
vars person = [name sue age 30 job teacher home york];
next_item("age", person) =>
** 30
next_item("home", person) =>
** york
next_item("height", person) =>
** <false>
```

That example should illustrate how much easier it is to use the matcher than ordinary list processing in such cases. Programs using the standard list processing method will run faster, however. If that is important, program using the matcher can be ”translated” after they have been developed.

7.4.2 Exercise: define previous_item

Using the above example as a template produce a definition of a procedure called previous_item that behaves thus:

```
vars person = [name sue age 30 job teacher home york];
previous_item(30, person) =>
** age
previous_item("york", person) =>
** home
previous_item("height", person) =>
** <false>
```

7.5 The matcher arrow "-->"

The matcher arrow is often very useful when you are sure two things will match, but you want to use the matcher to examine the contents of one of them. That way you don't use a conditional expression of the form:

```
if ... matches ... then ....
```

Instead we use the matcher arrow "–>". "–>" should not be confused with "->". The latter is the assignment arrow.

The infix operation "–>" could have been defined thus:

```
define 8 datum --> pattern;
    unless datum matches pattern then
        mishap('NON MATCHING ARGUMENTS FOR -->',[%datum,pattern%])
    endunless
enddefine;
```

The '8' in the procedure heading indicates that an infix operation of precedence 8 is being defined.

This definition looks as if it doesn't do anything when the datum matches the pattern. We shall see that, on the contrary, it can be used to 'decompose' a list.

7.5.1 Examples of the use of "-->"

The simplest use of '–>' is to check that a list has a certain format:

```
list --> [junction == ];
```

checks that LIST starts with "JUNCTION". If not, an error occurs.

```
    [a b c d] --> [junction ==];

    ;;; MISHAP - NON MATCHING ARGUMENTS FOR -->
    ;;; INVOLVING:  [a b c d] [junction ==]
    ;;; DOING    :  --> compile
```

The operation –> can also be used to decompose a list, i.e. to assign some of its components to variables.

```
vars first, second, rest;
vars list = [dogs like wagging their tails];

list --> [?first ?second ??rest];
first =>
** dogs
second =>
** like
rest =>
** [wagging their tails]
```

That would have caused an error if LIST had had fewer than two elements. Since there are at least two elements, it gives FIRST the first element, as its value, SECOND the second element, and REST a list containing all the remainder.

Here's another example.

```
vars first, second, rest;

[mary had a little lamb] --> [?first ?second ??rest];

first =>
** mary

second =>
** had

rest =>
** [a little lamb]
```

Notice the different effects of "?" and "??". The former means, roughly "match ONE element", the latter means, roughly "match any number of elements". The use of "??" and "==" can mean that there are alternative ways of matching a list againt a pattern. In that case the matcher will find only one of them. Which one it finds is not defined. The Birmingham ftp directory has a library called "doesmatch" that finds all the possible matches.

7.6 Findroom revisited

We can illustrate the use of the matcher to simplify the definition of a searching procedure, by redefining the procedure 'findroom' introduced in chapter 1, and defined thus:

```
define findroom(name, list_of_lists) -> data;
    ;;; search list_of_lists for one starting with name

    for data in list_of_lists do
        if data(1) = name then
            return();       ;;; i.e. stop the procedure
        endif;
```

```
        endfor;

        ;;; produce a mishap message
        mishap('DATA NOT FOUND', [^name ^list_of_lists])
    enddefine;
```

Compare this with the following:

```
    define findroom(name, list) -> data;
        vars len, breadth, height;  ;;; pattern variables

        if list matches [ ==  [^name ?len ?breadth ?height] == ]
        then
            [^name ^len ^breadth ^height] -> data;
        else
            mishap('DATA NOT FOUND', [^name ^list])
        endif
    enddefine;
```

The line

```
        if list matches [ == [^name ?len ?breadth ?height] == ]
```

runs the operation called 'matches' with two inputs, the value of the variable 'list', and the pattern on the right, which instructs the matcher what to look for in the list. It says, look for any number of elements (matched against "=="), then a list starting with the given name and containing three other things, followed by any number of elements (matched against "==" again).

7.7 Setting a value ("?") vs Using a value ("^")

Notice the difference between "?" and "^" here. The searched-for information is represented by the list

```
    [^name ?len ?breadth ?height]
```

where the symbol "^" says that the given name must be found. "^" can be read as 'use the value of', whereas the occurrences of "?" can be read as 'set the value of'. I.e. the three variables will be given values depending on what numbers are found after the name, once the list with the required name is found. If we had used "?name" instead of "^name", then ANY name would have been accepted.

If the MATCHES operation produces the result TRUE, having found what is required then the instruction after 'then' is obeyed, which ensures that the output local 'data' has an appropriate value, to be returned as the result of the procedure.

If MATCHES cannot find what is required in list, then it produces the result FALSE, and the instruction after 'else' is obeyed, causing an error message to be printed out.

Using the fact that "–>" will cause an error when a match fails, we could adopt the even shorter definition:

```
    define findroom(name, list) -> data;

        ;;; Use vars for pattern variables (with "?")
```

```
    vars len, breadth, height;

    list --> [ ==  [^name ?len ?breadth ?height] == ];

    [^name ^len ^breadth ^height] -> data;
enddefine;
```

7.8 List pattern matching

We now provide a more general motivation for the pattern matcher, and give a complete account of its operation.

If we wish to search down a list for an item we can use a test something like

```
item = hd(list)
```

to accomplish this. But list representations of more complex situations do not exhibit their significant features in terms of single items considered in isolation. Rather the norm is one of a context or co-occurrence of elements. Thus we are much more likely to want to know whether a list contains two designated elements occurring in a particular order, or even whether it satisfies some rather more complex condition. E.g. does the sentence start with a noun phrase? For now we consider only the relatively simple cases.

Supposing we have this list

```
[a b c d e] -> x;
```

In some problem we might wish to test might be to establish whether it contains ”b” and ”d” occurring in that order. Now

```
member("b",x) =>
** <true>
member("d", x) =>
**<true>
```

tell us that both are present i.e.

```
member("b", x) and member("d",x) =>
** <true>
```

but we know nothing of their relative positions in x nor indeed of their intervening or surrounding context.

We can capture this ordering of ”b” and ”d” for example:

```
x(1) == "b" and x(2) == "d"
```

which specifies that ”d” should immediately follow ”b: in the list. In fact the relationship that does obtain in X has the form

```
x(2) == "b" and x(4) == "d"
```

In this way we can specify any arbitrary patterning of elements in a list structure. It will however be very tedious to try all possible combination of pairs of successive numbers. The situation gets even worse if you merely want to test whether ”d” occurs SOMEWHERE to the right of ”b”, although you do not mind where exactly.

This need to be able to give a more complex specification of the structure of a list is met by the procedure MATCHES that tests a given list for the presence of some specified pattern.

7.9 Describing the shape of a list pattern

The B, D example is of course but one of an infinite variety of patternings that we might want to look for. So our specification of a pattern has to be couched in a suitably descriptive language. The pattern for 'B immediately followed by D' would be:

```
[== b d ==]
```

But the 'B followed somewhere by D' pattern would be specified like this:

```
[== b == d ==]
```

Where '==' denotes any number of list elements (including no elements). Thus all of these lists should meet that specification:

```
[a b c d e]
[b d]
[b a a c d f]
```

To test a given list, say X, for presence of this specified pattern we'd call MATCHES like this

```
x matches [== b == d ==] =>
** <true>
```

What result would you expect from the following:

```
x matches [a == e] =>
x matches [a == d == e] =>
```

The pattern specification we've adopted clearly admits of a lot of variation in x - it's a rather sloppy fit: and often that is a useful way of representing generality.

There are a number of ways in which we can tighten it up, when required. We might for example want just one intervening element between B and D:

```
x matches [== b = d ==] =>
** <true>
```

These two symbols == and = are basic descriptors of pattern shape and we may think of them as 'Gobbling up' intervening list items. We can call '=' Gobble-one and '==' Gobble-any. (These names were suggested by the late Max Clowes.)

Gobble-one and Gobble-any help us characterise the linear shape of a pattern. We may also want to characterise its structural organisation. For example that the first element in the target list must itself be a list - as it is in CUPBOARD for example.

```
cupboard matches [[==] blanket ==] =>
** true
```

And this device may of course be used to dig arbitrarily 'deep' into a list structure.

The pattern is then a sort of picture with lots of missing details, of the kind of list we are looking for.

7.10 Using variables in a pattern specification

So far we've described our patterns in very literal terms. i.e. that there needs to be a "B" and a "D", or a list etc. In practice the items we want to include in our specification may have been constructed by other procedures (and as we shall see by procedures that use MATCHES). Typically these items will be the value of variables.

Suppose we have a variable "box" whose value is a list representing the contents of a box.

```
vars box = [shoes tins brushes] ;
```

And a variable cupboard corresponding to a cupboard that contains the box and other things:

```
vars cupboard = [[shoes tins brushes] blanket hats coats umbrellas];
```

If we write

```
cupboard matches [box blanket ==] =>
** <false>
```

The result is false because the first element of cupboard is not the word "box" but a list which is the same as the value of the variable box. We need to enrich the pattern specification language so as to distinguish between words used literally and words used as variable names. We do this with the up-arrow ˆ prefix, as explained in the chapter on lists. Thus

```
cupboard matches [^box blanket ==] =>
** <true>
```

The use of ˆ (up-arrow) in the pattern specification here is the same as that introduced earlier. Thus

```
[box blanket ==] =>
** [box blanket ==]
```

But

```
[^box blanket ==] =>
** [[shoes tins brushes] blanket ==]
```

And it is of course only this latter version that matches CUPBOARD.

7.11 Matching a "segment" of a list

Had the value of cupboard been

```
** [shoes tins brushes blanket pillow]
```

then that MATCH would have failed.

We need to 'strip off' the list brackets of the value of BOX if we are to match this new kind of CUPBOARD, and we can do this by using a double up-arrow:

```
[^^box blanket ==] =>
** [shoes tins brushes blanket ==]
```

This uses the value of the variable box to insert a collection of items to form a SEGMENT of the list, rather than inserting a single list as an ELEMENT of the enclosing list.

To illustrate the power of matches and the pattern language, consider the following definition of the procedure ismember which could be compared with the definition of iselement in Chapter 3:

```
define ismember(item, list) -> trueorfalse;
    list matches [== ^item ==] -> trueorfalse
enddefine;

ismember(3, [ 1 2 4 ] ) =>
** <false>
ismember(3, [ 1 2 3 4 ] ) =>
** <true>
ismember("c", [ a b c d e]) =>
** <true>
```

This uses the fact that matches returns a ‘boolean’ result, i.e. either TRUE or FALSE.

7.12 Retrieving details of the target list

In some of our examples we have merely been concerned to establish whether a particular target list meets some pattern specification, where that specification only cited various list fragments embedded within a context whose precise composition was not germane to the recognition of this key configuration.

We may however want to know what that context is, when a match is obtained. For instance, a sentence analysing program which finds a verb in a sentence may want to know what came before and after the verb. It is also used in programs which interrogate the Pop-11 database not merely to see if a particular piece of information is there, but also to find out exactly what it is. E.g. you may not merely wish to find out if the database has an item of the form

```
[age fred ....]
```

you may also want to know what the age is.

Thus in recognizing the ordered co-occurrence of ”B” and ”D” in the list

```
vars x = [a b c d e];
```

we might want to be ‘told’ what the value of the intervening list element(s) is (are).

To set a variable say P to take on the value of single list element prefix the variable name with ”?” Thus

```
vars p;
x matches [== b ?p d ==] =>
** <true>
p =>
** c
```

Similarly to set the value of the variable to be some SEQUENCE of list elements use ”??” Thus:

```
vars q;
x matches [a ??q d ==] =>
** <true>
q =>
** [b c]
```

Think of ”?” as ”get ONE element” and ”??” as get ANY elements”, by analogy with ”=” and ”==”.

Try the following, which gives LIST a new value, then uses it:

```
[i like talking to you] matches [i ??list you] =>
** <true>
[you ^^list me?] =>
** [you like talking to me?]
```

This is typical of the sort of trick used by Eliza.

Whenever the MATCH fails because the list does not match the pattern the queried variables may have their values altered in an unpredictable way as part of the process of determining that the match fails.

```
[i like talking to you] matches [i ??list you alone] =>
** <false>
list =>
** [like talking to you]
```

7.13 Using a ”restriction” to control or check the match

The basic concept of matching as illustrated so far uses identity between elements of the target list and corresponding elements of the pattern specification. We can generalise this by requiring that a target element have some specified PROPERTY.

For example in the ELIZA world, the occurrence of a word indicating reference to the family e.g. ”son”, ”sister”, ”father”, ”mother”, etc. in an input sentence is an important response-determining cue. To detect that a list element belongs to such a set, rather than being identical with a given pattern element, we can use a ”restriction procedure” as an affix to a queried variable in the pattern specification. Thus if we have a procedure that produces TRUE when applied to family word and false otherwise, then we can use it to restrict the value of ”x” in the following match:

```
vars x;
[my father loved me] matches [== ?x:family ==] =>
** <true>
x =>
** father

[you remind me of my brother] matches [== ?x:family ==] =>
** <true>
x =>
```

```
** brother


[you remind me of my car] matches [== ?x:family ==] =>
** <false>
```

where FAMILY is a (previously-defined) procedure that might be something like this:

```
define family(word) -> result;
  member(word, [son sister father mother brother]) -> result
enddefine;
```

You can gain experience with MATCHES ?, ??, ^, ^^, and variables by working through TEACH RESPOND.

See also TEACH MATCHES, TEACH ARROW

7.14 Summary of match notations

Basic format:

```
list MATCHES pattern
```

Pattern specification can contain the following:

```
1) [cat mouse x 99 'string']
    literal words numbers, strings, and other items to be
    checked in the target list

2) =
    The 'Gobble-one' spacer

3) ==
    The 'Gobble-any' spacer

4) ^A
    Put into the pattern an object which is the value of the
    variable A. That object will then be compared with the
    corresponding object in the target list.
   ^(<expression>)
    Put into the pattern whatever results from evaluation of
    the expressin.

5) ^^A
    The variable A MUST have a list as value. Put into the pattern
    all the elements of the list, for comparison with elements of
    the target list.

6) ?A
    Set the value of the variable A to be a single element in the
    matching target list.
```

```
7) ??A
    Set the value of the variable A to be a list containing a
    sequence of elements in the target list.

8)  ?A:TEST
    Only allow the variable A to match an element such that TEST(A)
    is not FALSE.

9) ??A:TEST
    Like (8), but the TEST is applied to a LIST of successive elements
    from the target list to be matched against the variable A.
```

(Future versions of Pop-11 with a more sophisticated matcher will use additional pattern element formats.)

NOTE: If the predicate TEST in the last two cases returns not TRUE but some other non-FALSE result, then the result will be assigned to the variable A instead of the item or list of items from the target list. For example, define a procedure to recognize a list of colours, and return a list of their initials:

First a procedure to produce a one character word from a word:

```
define initial(word);
    subword(1,1,word)
enddefine;
```

Use that to define a procedure that returns a list of initials if given a list of colour words:

```
define all_colours(list) -> result;
    ;;; given a list of colour words return a list of single
    ;;;  character words, using the first character of each word.
    lvars item;
    for item in list do
        unless member(item, [red green blue indigo violet orange])
        then
            false -> result;
            return();
        endunless;
    endfor;
    ;;; create list of initials
    maplist(list, initial) -> result;
enddefine;

all_colours([red square]) =>
** <false>
all_colours([green red orange]) =>
** [g r o]
```

This can be used to transform the output of the matcher, because it returns the transformed list rather than simply true. E.g.

```
vars colours;
```

```
[the big green blue red orange thing on the wall]
    matches [== ??colours:all_colours thing ==] =>
** <true>

colours =>
** [g b r o]
```

See HELP MATCHES/RESTRICTIONS

7.15 MATCHing on a corpus of lists - the DATABASE concept

The description of some task situation may take the form of a large corpus of lists, representing propositions. For instance we might know that B1, B2, ... etc are blocks, that each has a colour and a size, and that some are on others. These facts could be represented as a database consisting of a list of lists:

```
[[b1 isa block]
 [size b1 big]
 [colour b1 green]
 [on b1 b2]
 .........
 ]
```

We could also describe the contents of an image in terms of which objects are in it what their properties are, and how they are related. The library program SEEPICTURE builds up just such a list representation of a PICTURE pattern to provide a basis for recognition.

7.16 Adding and removing database items

Since this facility is often required, Pop-11 provides a collection of automatically loaded library procedures for manipulating lists of lists. They all make use of a global variable 'database' which may contain arbitrary information.

We can build up a DATABASE using ADD

```
vars database;
[] -> database;
add([a]);
add([b]);

database =>
** [[b] [a]]
```

Notice that items appear in reverse order to the order of ADDing.

More concisely we can use ALLADD

```
alladd([[c] [d]]);
database =>
** [[d] [c] [b] [a]]
```

Complementing ADD and ALLADD we have REMOVE and ALLREMOVE.

```
remove([c]);
database =>
**[[d] [b] [a]]
```

The order of items in a call of ALLREMOVE is not important i.e. it does not need to reflect the ordering of the DATABASE

```
allremove([[a] [b] [d]]);
database =>
** []
```

The procedure REMOVE will remove at most one item from the database, even if it is given a pattern which matches several. Thus REMOVE([==]); instead of removing everything, removes just one item. Moreover, REMOVE will generate a MISHAP if it does not find one item to remove. Similarly, ALLREMOVE will generate a mishap if it can't remove something for every element of the list of patterns given to it as argument.

The procedure FLUSH is provided without these restrictions. FLUSH deletes everything in the database that matches its argument, but if there is nothing that matches, then FLUSH does nothing!

The argument given to FLUSH is a pattern specification, that is FLUSH uses MATCHES to determine the list items that it will DELETE. It is however very powerful in its action... it removes ALL the matching DATABASE entries. Thus with the DATABASE [[D] [C] [B] [A]] the action

```
flush([=]);
```

clears the DATABASE, thus:

```
database =>
** []
```

Thus in this situation FLUSH([=]) is equivalent to:

```
allremove([[a] [b] [c] [d]]);
```

Whenever the database contains only one-element lists, the command

```
flush([=]);
```

will remove them all.

Similarly

```
flush([==]);
```

will remove all lists from the database no matter what is in them. It is therefore equivalent to

```
[] -> database;
```

7.17 Using ”it” to record what was removed

After using FLUSH or REMOVE the variable IT will hold the item last removed. E.g.

```
add([dogs like meat]);
remove([dogs like == ]);
it =>
** [dogs like meat]
```

The procedure ALLREMOVE, uses the variable THEM instead. This will be a list of all the items removed. Similarly, ADD updates IT, and ALLADD records things in THEM.

7.18 Finding items in the database

One of the most commonly used procedures is PRESENT, which takes a pattern and returns true or false depending on whether there is something in the database that matches the pattern. If the pattern contains variables preceded by ”?” or ”??”, and the match is true, then the values of those variables will give information about the list that matched the pattern. For example:

alladd([[the big red box] [the long square pole][the tiny blue flea]]) vars x y;

present([the ?x ?y pole]) => ** <true> x, y => ** long square

Exercise: what will the values of x and y be after

```
present([the ?x ?y flea]) =>
```

7.18.1 How PRESENT works.

It does this by trying to MATCH the pattern against every item in the database. If the match is ever successful then true is returned as the result of PRESENT, otherwise the result is false.

```
alladd([[a b c d] [d c b a] [a b d c]]);

present([== b ==]) =>
** <true>

present ([== b c]) =>
** <false>
```

Notice that PRESENT finds the ‘first’ matching item.

PRESENT is frequently employed in a conditional e.g.

```
if present(pattern) then action
```

The ‘IF... THEN’ construction ‘uses up’ the boolean value returned before THEN ie. try:

```
if present([== b ==]) then => endif;
```

The ”STACK EMPTY” error message arises because Pop-11 treats the value returned by PRESENT between ”IF” and ”THEN” as <TRUE> or <FALSE>. So the value is no longer there for ”=>” to print out, and that produces the error message.

7.19 Using "it" to record what was matched

The value returned by present when successful is TRUE. But that does not determine what exactly matched the pattern in the database. For many purposes however we will need to know what the matched item is for example to use it in the THEN branch of the conditional. For this purpose the DATABASE variable IT is set to have the value of the matching item, when PRESENT finds something.

```
alladd([[a b c d] [d c b a] [a b d c]]);

if present([== b ==]) then
  it =>
  remove(it);
endif;
** [a b c d]
```

Thus we see that the item matched, and subsequently removed was the first item in the database.

Note that it found only ONE item, matching the pattern, and removed it. FOREACH, explained below, shows how you can search for ALL items matching some pattern.

7.20 Retrieving values from within a matching ITEM

Since all of the apparatus of MATCHES is utilised in those DATABASE procedures, PRESENT can be used to set values of appropriately queried variables in the pattern specification.

```
vars x;
present([?x b ==]) =>
** <true>
x =>
** a
it =>
```

Notice that if "?" or "??" is used before a word in a pattern, then that word should be declared as a variable name. Hence the "vars x;" above.

If the variables in patterns are not declared to be local, to the procedures which use them, then different procedures can mess each other up. This applies to procedures which use MATCHES or any of the database operations PRESENT, FLUSH, LOOKUP, REMOVE, etc.

7.21 Using LOOKUP to extract information from the database

We do not always want to know what the matching item was. Sometimes we will know in advance that an item matching the pattern is present in the DATABASE and want only the value of some fragment of the item.

For this purpose the procedure LOOKUP is provided. It does not return <TRUE> or <FALSE> but merely sets the value of queried pattern variables when a match is found, and causes an error if no match is found.

```
alladd([[a b c d] [d c b a] [a b d c]]);

lookup([= ?x b ==]);
```

```
x =>
** c
```

In the event of no match being found an error will result

```
lookup([?x == e]);
;;; MISHAP - LOOKUP FAILURE
;;; INVOLVING:  [? x == e]
;;; DOING    :  sysprmishap mishap lookup ......
```

Note that PRESENT is to MATCHES as LOOKUP is to –>. PRESENT and MATCHES check a condition and return a true/false result, and may simultaneously bind pattern variables. But only PRESENT searches over a list. LOOKUP and –> do not return any result, and only LOOKUP searches.

```
procedure   returns          iterates over      mishap on
            a boolean        the database       failure
-----------------------------------------------------------
matches      yes                  no                no
-->          no                   no                yes
present      yes                  yes               no
lookup       no                   yes               yes
```

7.22 FOREACH: iterates over all items PRESENT matching a pattern

There will often be a need to find not just one matching item in the DATABASE, (or to set the value of queried pattern variable for just one matching item) but to find all of them. For this purpose the looping construct FOREACH is provided, We can use it, for example to find all the items in the database in which "b" precedes "a":

```
[[a b c d] [d c b a] [a b d c] [b d c a]] -> database;

foreach [== b == a ==] do
  it =>
endforeach;
```

** [d c b a] ** [b d c a]

Note that FOREACH is a "syntax" word (like IF and DEFINE), not a procedure name, and hence the pattern specification that follows need not be enclosed in round brackets '(' ')'.

The general form of FOREACH is

```
FOREACH <pattern> DO <action> ENDFOREACH;
```

For example, to print out every item in the database representing something blue:

```
vars x;
foreach [??x is blue] do
    x ==>
endforeach;
```

Inside the <action> the variable IT is available to represent the database item which has matched the pattern. E.g. suppose you have various database entries of forms:

```
[p1 is a person]
[b3 is a block]
[c5 is a cat]
```

etc. Then, to print out all the blocks do:

```
foreach [??x isa block] do
    x =>
endforeach;
```

you can use "[%" and "%]" to make a list of all the blocks:

```
[% foreach [??x isa block] do
    x
   endforeach
%] -> blocks;
```

Each time round the value of X is left on the stack and the [%...%] brackets make a list of all of them. The variable IT can be used each time round the loop to refer to the database entry which was found to match the pattern given after FOREACH.

FOREACH can be followed by IN to specify a list other than DATABASE to search in, i.e.

```
FOREACH <pattern> IN <list> DO <action> ENDFOREACH;
```

If you have access to a working Poplog system you can get more information and examples in:

```
TEACH MATCHES; TEACH DATABASE; TEACH FOREACH; TEACH RIVER2;
```

7.23 Checking a set of patterns against the database

Just as ALLADD and ALLREMOVE can be given a list of patterns to add or remove, similarly, ALLPRESENT can be given a list of patterns to look for in the database. If it finds items for all of them it returns a list of the found items (and also assigns it to the variable THEM). Otherwise its result is false. E.g. to find a grandson of TOM:

```
alladd([
      [dick father harry]
      [tom father jack]
      [bill father tom]
      [jack father dick]]);

vars x, y;
if allpresent([[tom father ?x] [?x father ?y]]) then
  y =>
```

```
endif;
** dick
them =>
** [[tom father jack] [jack father dick]]
```

For more summary information on the database procedures see:

```
HELP DATABASE,  HELP ADD,   HELP PRESENT,
HELP REMOVE,    HELP FLUSH, HELP LOOKUP.
HELP FOREACH,   HELP FOREVERY
```

7.24 Forevery: simultaneously satisfying a collection of patterns

Just as ALLPRESENT is a generalisation of PRESENT, so Pop-11 includes FOREVERY which is a generalisation of FOREACH, and allows some fairly powerful manipulations of the database. In particular, we can find all possible combinations of a SET of patterns in the database. E.g. to find all paternal grandfather relations, i.e. all combinations of the form

```
[?x father ?y][?y father ?z]
```

Here's an example

```
[
    [dick father harry]
    [tom father jack]
    [bill father tom]
    [dick father mary]
    [jack father dick]] -> database;

vars x, y, z;
forevery [[?x father ?y] [?y father ?z]] do
    [^x is the paternal grandfather of ^z] =>
endforevery;
** [tom is the paternal grandfather of dick]
** [bill is the paternal grandfather of jack]
** [jack is the paternal grandfather of harry]
** [jack is the paternal grandfather of mary]
```

Note that FOREVERY made sure that whatever matched "y" in in the two patterns was the same.

The permitted formats of FOREVERY are:

```
FOREVERY <list of patterns> DO <actions> ENDFOREVERY;

FOREVERY <list of patterns> IN <database> DO <actions> ENDFOREVERY;
```

When 'IN <database>' is omitted, then the value of the variable DATABASE is used as the database.

Another example: find all the blocks and print out their colours:

```
forevery [[?x isa block] [colour ?x ?col]] do
    [^x is a ^col block] =>
endforevery;
```

Or to find all maternal grandfather relationships:

```
forevery [[?x father ?y] [?y mother ?z]] do
    [^x is grandfather of ^y] =>
endforevery;
```

In addition to these procedures the Pop-11 library contains some more powerful procedures SCHECK and SCHOOSE, for matching a whole database pattern against a database, and indicating whether the match was partially successful, what was missing, what was surplus, etc. For details see TEACH SCHEMATA

A powerful expert system shell based on the matcher is described in HELP NEWPSYS, and TEACH PSYSRIVER

Although the database procedures described here are very powerful compared with what most programming languages provide. They are still not as powerful as the facilities built into the language Prolog. See, for example:

```
W. Clocksin and C.S. Mellish
PROGRAMMING IN PROLOG, Springer Verlag.
```

7.25 LIB FMATCHES overcomes some limitations of the matcher

The pattern matching facilities have serious restrictions in that they will not work properly with sections, and they cannot be used with lexically scoped variables, which is why all pattern variables have to be declared with ”vars”, not ”lvars”. (See HELP LVARS).

There is a modified version of the matcher described in HELP FMATCHES which overcomes these restrictions, though it is no longer an infix operator, but a syntax word.

FMATCHES is superseded by the DOESMATCH library and the READPATTERN library, both available from the Birmingham FTP directory.

In a later version of Pop-11 both may be superseded.

8 CHAPTER.8 AN AI APPLICATION: A GENERAL PROBLEM SOLVER

As explained in many AI text books, it is often necessary to search an abstract space in order to find a solution to a problem.

This chapter, which will be of use only to fairly experienced programmers, uses a subset of some of the contents of a Pop-11 "teach" file developed by the author, to define an illustrative general purpose problem solver which searches for a solution to a variety of problems.

In order to use the procedure solve_problem you have to choose a representation for your problem and the search space it defines. That requires choosing a representation for each state, including the initial state, the goal state or goal states if there are several alternative ways of solving the problem, and other intermediate states.

8.1 Procedures to be supplied by users

You also need to define four procedures, as follows:

is_goal_state

```
is_goal_state(state, goal) -> boolean
```

This is given a state, and a specification of the current goal and returns true if the state is acceptable as a goal state, otherwise false. This procedure will be different for different sorts of problems.

next_states

```
next_states(state) -> newstates;
```

The procedure next_states is given a state and returns a list of states that can be reached in one step from that state. The procedure will be have to be defined differently for different problems.

same_state

```
same_state(state1, state2) -> boolean
```

This procedure is required so that the problem solver can tell whether a particular new state is equivalent to one that has previously been examined, so that time is not wasted examining it again and trying to explore its successors. Since the notion of equivalence will depend on the kind of problem and how states are represented, this has to be defined by the user.

If this procedure is provided, then the general problem solver can keep a "history" list of the states that have already been examined, and whenever new states are generated it can discard any of them that are "the same" as a state already on the history list.

insert_state

```
insert_state(state, oldstates) -> newstates
```

This procedure can embody some "heuristic" information. It is given a new state (e.g. something produced by the procedure next_states) and list of states waiting to be explored, and it returns a new list of states, with an order which represents the best current guess as to which one should be examined first. For example it is often useful to prune a search space by working first on items that are going to be most difficult to fit into a complete solution, so that those that are unusable are quickly eliminated from further combinations of items.

is_in_list

The problem solver described below needs this utility procedure which can be applied to a state, a list of states, and the user's procedure same_state, to check whether the first state is equivalent to one of those in the list. It returns a true or false result.

```
define is_in_list(newitem, list, sameitem) -> boole;
    ;;; Return true if and only if the newitem is the same as something in
    ;;; in the list, as compared using the procedure sameitem

    lvars procedure sameitem;   ;;; declare it as a procedure, for speed

    lvars olditem;

    for olditem in list do
        if sameitem(newitem, olditem) then
            ;;; found an olditem which is the same, so return with true
            true -> boole;
            return();
        endif
    endfor;
    ;;; did not find anything that was the same
    false -> boole
enddefine;
```

8.2 The definition of solve_problem

If the user has defined the above procedures they can be given to the following general problem solver. It takes six arguments and produces a result which is the goal state if one is found and otherwise the value FALSE.

The first argument is a representation of the initial state. The second is a representation of the goal to be achieved. The next four arguments are procedures, which are the users versions of the four procedures listed above: the goal recogniser, the procedure for generating new states from old ones, the state equivalence recogniser, and the procedure to insert a new state into a list of states.

Here is one of many ways to define such a problem solver:

```
define solve_problem
    (initial, current_goal, isgoal, nextstates, samestate, insert)
        -> result;

    lvars
```

```
        initial,        ;;; the initial state
        current_goal,   ;;; the second argument for isgoal
        ;;; four procedures defining the problem and a strategy
        procedure (isgoal, nextstates, samestate, insert),
        result;

    lvars
        alternatives = [^initial],  ;;; the list of alternative states
        history = [] ;              ;;; the list of previous states

    vars current_state, rest;       ;;; use "vars" for pattern variables

    repeat
        if null(alternatives) then
            ;;; failed
            false -> result;
            return();
        else
            alternatives --> [?current_state ??rest];
            rest -> alternatives;

            ;;; Check if current_state is a goal state
            if isgoal(current_state, current_goal) then
                ;;; problem solved
                current_state -> result;
                return();
            else
                ;;; Keep a history list, avoid circles
                [ ^current_state ^^history ] -> history;

                ;;; generate successor states to current_state
                lvars states;
                nextstates(current_state) -> states;

                ;;; put all the "new" states onto the alternatives list
                lvars state;
                for state in states do
                    unless is_in_list(state, history, samestate) then
                        insert(state, alternatives) -> alternatives
                    endunless;
                endfor;
                ;;; now go back to the beginning of the repeat loop
            endif
        endif
    endrepeat
enddefine;
```

8.3 Using solve_problem to solve a simple problem

Suppose you have a set of parent child relationships stored in a database (see TEACH * DATABASE).

For example you could set up a database like this:

```
define start_family_tree();
    [
        [father [tom jones] [dick jones]]
        [father [tom jones] [sue smith]]
        [mother [ginny jones] [dick jones]]
        [mother [ginny jones] [sue smith]]
        [father [dick jones] [mary jones]]
        [mother [sue smith] [joe smith]]
        [mother [sue smith] [fred smith]]
        [father [jack smith] [joe smith]]
        [father [jack smith] [fred smith]]
        [father [fred smith] [angela green]]
        [mother [angela green] [willy green]]
    ] -> database;

enddefine;
```

Then the following command will set up the initial database:

```
start_family_tree();
;;; print out database to check
database ==>
```

Now suppose you had to find out whether someone A is an ancestor of someone else B.

One way you could do that is set up a search space consisting of a start node A, and all paths from A to descendents of A. Then the solve_problem procedure could be given the task of searching for a path through the family tree starting from A and ending with B. If such a path is found, it would answer the question positively: A is an ancestor of B.

You could represent a state in the search space as consisting of a list of names representing a path found so far through the family tree. For example if we wanted to find out whether [ginny jones] is an ancestor of [angela green] we would have a succession of states showing routes up the tree to ginny jones thus

```
[[ginny jones]]                 ;;; initial state
[[dick jones] [ginny jones]]    ;;; a successor state
[[sue smith] [ginny jones]]     ;;; another successor state
```

If we use this representation, then it is easy to tell whether the problem has been solved: check whether the target person B is the first item of the list.

Thus we can define the following procedures to use with solve_problem

```
define is_ancestor_goal(state, target) -> boole;
    ;;; The state is a list of names defining a route up the family tree
    ;;; Does it start with target?
```

```
    lvars state, target, boole;
    state matches [^target == ]  -> boole
enddefine;
```

However, we also have to create a procedure for generating new states from a given state. Remember that a state is a list of names of people

```
[personN ... person3 person2 person1]
```

where person1 represents a parent of person2, person2 a parent of person3, and so on. Person1 is the one given as the original alleged ancestor, and personN is the latest person found in searching down the tree from person1.

Then in order to find successors to this state we need to find individuals who are immediate descendants of personN, and form a set of new routes up the tree from each of them. First we need a procedure to find the immediate descendants. We can use the database searching procedure foreach (described in TEACH FOREACH)

```
define immediate_descendants(person) -> list;
    lvars person, list;
    vars next;  ;;; a pattern variable used with foreach below

    ;;; start making a list
    [%
        ;;; first collect all those to whom person is father
        foreach [father ^person ?next] do
            next;   ;;; just leave the next person on the stack
        endforeach;

        ;;; now collect all those to whom person is mother
        foreach [mother ^person ?next] do
            next;   ;;; just leave the next person on the stack
        endforeach;

    %] -> list
enddefine;
```

You can test this as follows

```
immediate_descendants([ginny jones]) =>
** [[dick jones] [sue smith]]
immediate_descendants([sue_smith]) =>
** []
```

Using the procedure immediate_descendants we can define a procedure to create extensions to a state represented as a route up the tree

```
define family_next_states(state) -> newstates;
    lvars state, states;
    ;;; make a list of new states that start with one descendant
    ;;; and the rest of state
```

```
    lvars nextpeople, person;
    immediate_descendants(hd(state)) -> nextpeople;

    ;;; Now make a list of the new states
    [%
        for person in nextpeople do
            ;;; make a new state and leave it on the stack, to go into
            ;;; the list being created by [% ... %]
            [^person ^^state]
        endfor
    %] -> newstates

enddefine;
```

Now test it

```
family_next_states([[ginny jones]]) ==>
** [[[dick jones] [ginny jones]] [[sue smith] [ginny jones]]]
```

Note that the output was a list of TWO lists.

```
[[dick jones] [ginny jones]]
```

and

```
[[sue smith] [ginny jones]]
```

Try extending the first list:

```
family_next_states([[dick jones] [ginny jones]] ) ==>
** [[[mary jones] [dick jones] [ginny jones]]]
```

This time there is only one extension, containing three names. You may of course get a different answer with your database of names.

We now need a test for whether a state is the same as one we have tried before and for this problem it is easy - just see if the two states are the same. (It is not always that easy.)

```
define family_same_state(state1, state2) -> boole;
    state1 = state2 -> boole
enddefine;
```

And finally we define the simplest possible state insertion procedure for combing a new state with a list of states: just put it at the front, so that it will be examined next. (This defines a depth first search strategy).

```
define family_insert(state, states) -> newstates;
    [^state ^^states] -> newstates
enddefine;
```

You should now be able to use the solve_problem procedure to see whether one person is a descendant of another. We can define a new procedure called check_ancestor which does the work, as follows:

```
define check_ancestor(person1, person2) -> result;
    ;;; use solve_problem to do the work
    solve_problem
    ([^person1],        ;;; start state (must be a list with a name)
        person2,            ;;; goal state information
        is_ancestor_goal,   ;;; goal state recogniser
        family_next_states, ;;; generator for next states
        family_same_state,  ;;; circle detector procedure
        family_insert       ;;; insertion procedure
    ) -> result
enddefine;

;;; Now test it
check_ancestor([ginny jones], [mary jones]) ==>
** [[mary jones] [dick jones] [ginny jones]]

;;; Compare this case, with a different target descendant
check_ancestor([ginny jones], [tom jones]) ==>
** <false>
```

Try some other tests using an extended version of the database.

8.4 Exercises

1. The above strategy is not always the most efficient. See if there are ways of improving it e.g. by re-defining the next states generator or the insertion procedure.

2. Try using the above problem solver to set up a route finding program, using a database of information about road or rail links between towns, and then trying to find a route between any two towns. If you want the shortest route to be found you will have to have distance information and the insert procedure or the next states procedure, or both could use the distance measure to order the states, so that shorter routes are tried first. Text books on AI define alternative algorithms for doing this sort of thing.

3. Try using the problem solver to solve the problem of selecting some blocks from a pile of blocks to build a tower of exactly a given height. E.g. you may be given a list of numbers representing the heights of the available blocks:

```
[3 16 12 22 5 5 24 14 8 7 22 11]
```

and the task of finding blocks to make a tower exactly 30 units high.

4. The above problem solver stops as soon as it finds a solution. If for some reason you wish to look for another solution to the same problem that could be very wasteful. Try extending the procedure so that it not only returns the solution, but enough information about what it has done so far to enable it to be re-started with that information. This will require a change to the procedure to allow it to be run with information saved from a previous run, which might default to [].

9 CHAPTER.9 RECORDS, VECTORS AND OBJECTCLASS

This chapter is about different ways in which users can extend the variety of types of data-structures their programs use, and also about the associated procedures that are created, either automatically or explicitly.

Records and vectors have been part of the language for many years. Objectclass is a recent extension providing ”Object oriented” programming facilities.

9.1 Records

A recordclass is a class of objects all of the same general structure, where each object contains zero or more fields that can contain data. A record is an instance of a recordclass. All instances of the same class have the same number of data fields.

Examples of recordclasses that are predefined in Pop-11 are pairs, created by conspair, which are used for lists, and references, created by consref. E.g. all pairs have two fields, all refs have one field.

A field may be ”full”, i.e. unrestricted in type, and therefore able to contain any legal Pop-11 data-type, or restricted, e.g. to 7 bit integers, or 16 bit decimals, etc.

Each record class has an associated ”key” and a family of procedures, as follows:

```
o A constructor procedure, for creating new instances (e.g. conspair)
o An exploder (or destructor) which can be applied to an instance
  and puts all of its contents on the stack (e.g. destpair)
o A recognizer
o A collection of accessor/updater procedures, one for each field
  in the recordclass. (For instance pairs have front and back, and
  their updaters.)
o A class_print procedure for printing instances of the class.
  (The class_print procedure for pairs, knows about printing
  list structures using "[" and "]")
o A class_apply procedure for deciding what to do if an instance
  is applied to some other object, as if it were a procedure.
  (The class_apply procedure for pairs is invoked if you apply
  a list to an integer N. It returns the N't item of the list.)
```

Each key is itself an instance of a special recordclass data-type called keys, and its contents provide a lot of information about the class and its instances.

In addition to the class-specific procedures there are generic procedures that can be applied to instances of many different data-types.

9.2 Defining new record types in Pop-11

The procedure conskey provides the most general mechanism for creating new record classes (and new vector classes). However there are two syntax words that are generally easier to use.

9.2.1 DEFCLASS and RECORDCLASS

The defclass construct defined in REF DEFSTRUCT can be used to create other new record classes or new vector classes (see below for vector classes). The syntax word ”defclass” is defined in the autoloadable library, on the basis of conskey.

However, for many purposes the older recordclass construct can be used as a simpler mechanism for defining new record classes. (”recordclass” is also a library syntax word defined in terms of ”conskey”).

9.2.2 An example: recordclass point3D

Suppose it were necessary to represent points in 3-D by means of four element triples, one for each coordinate of the point and one for the colour. The corresponding recordclass could be defined thus:

```
recordclass point3D point_x point_y point_z point_colour;
```

The newer syntax for this, using defclass would appear thus:

```
defclass point3D {point_x, point_y, point_z, point_colour};
```

As mentioned above, this automatically creates a family of new procedures associated with the new recordclass, including conspoint3D, destpoint3D, ispoint3D, point_x, point_y, etc.

An instance of the class can then be created using the procedure conspoint3D and its components can be accessed or changed using the field selector/updater procedures whose names correspond to the field names above. For example, create two points, one at the origin:

```
vars
    point1 = conspoint3D(0, 0, 0, "green"),
    point2 = conspoint3D(10, 10, 10, "red");

;;; print them out
point1 =>
** <point3D 0 0 0 green>

point2 =>
** <point3D 10 10 10 red>

point_colour(point2) =>
** red

"yellow" -> point_colour(point2);

point2 =>
** <point3D 10 10 10 yellow>
```

9.2.3 When should records be used?

Lists (and vectors, explained below) can be used whenever records are used. However, records have certain advantages in some contexts. Records are are used in preference to lists when:

o Very large numbers of instances are to be created: records normally take less space in memory than the corresponding lists

o Elements of the records need to be accessed and updated rapidly. Accessing or changing the fourth element of a list normally takes longer than accessing or updating the fourth element of a record. That is because with lists it is necessary to ”chain” down the list links, whereas the Nth component of a record can be accessed in one go. (Using offsets from the beginning of the record.)

o Run time type-checking is desirable for robustness or debugging. If you attempt to access the fourth element of a list it will work as long as the list has four or more elements, even if something has gone wrong and the wrong sort of list has been found. If you apply a recordclass field accessor to the wrong sort of structure, even if it has the right number of fields, this will produce a run time error message. For example:

```
front(point1) =>
;;; MISHAP - PAIR NEEDED
;;; INVOLVING:  <point3D 0 0 0 green>
;;; DOING    :  sysprmishap mishap front .....
```

Lists are particularly useful when you wish to extend a structure by inserting new items in the middle, or when you wish to have a number of structures with shared ”tails”, as in these examples:

```
vars
    list1 = [a b c],
    list2 = [d e f],
    list3, list4;
list1 =>
** [a b c]
;;; Extend list1 by adding a new item after "a"
conspair("item", tl(list1)) -> tl(list1);
list1 =>
** [a item b c]

;;; Make two new lists that both share list2 as their tail

[apple ^^list2] -> list3;
[orange ^^list2] -> list4;
list3 =>
** [apple d e f]
list4 =>
** [orange d e f]
;;; they have identical tails, sharing memory
tl(list3) == tl(list4) =>
** <true>
```

That cannot be done with vectors or records.

9.3 Defining new vector types in Pop-11

A vector class is a class of data whose instances may be of different lengths, though once an instance is created, its length cannot be changed.

Pop-11 includes two main built in vector classes, namely strings and vectors. A string is an instance of a vector class that is constrained to contain an ordered set of characters, i.e. 8-bit integers. Strings may be of different lengths, but each component of a string must be an 8-bit character. E.g.

```
'a tiny string'         'a much longer string of characters'
```

Ordinary Pop-11 vectors can also be of different lengths, but their contents are unconstrained.

```
{1 2 3}     {1 2 3 a vector with numbers [words and] [lists]}
```

Vectors play a major role in the construction of arrays. See REF ARRAYS Roughly speaking a multidimensional array in Pop-11 is a combination of a one dimensional vector and a procedure for mapping between the high dimensional array and the low dimensional vector.

9.3.1 Using DEFCLASS and VECTORCLASS

The defclass construct described in REF DEFSTRUCT can be used to create new vector classes as well as new record classes. However it is often simpler to use the vectorclass construct.

9.3.2 Example of creation of a new vector class.

This example is based on HELP VECTORCLASS. The imperative

```
vectorclass short 16;
```

declares a new vector class called "short" which is restricted to containing 16 bit integers in its fields. This creates procedures

```
initshort, consshort, destshort, isshort
subscrshort, fast_subscrshort,
```

and declares the variable

```
short_key
```

with the new key as its value.

The newer syntax for doing this uses defclass, as follows:

```
defclass short :16;
```

To create an instance of short, with 32 fields do this:

```
vars shortie = initshort(32);
shortie =>
** <short 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0>
```

By default the number 0 is put into each of the fields of a newly created short vector. But this can be changed, e.g. by using the updater of the subscriptor function subscrshort:

```
99 -> subscrshort(3, shortie);
shortie =>
** <short 0 0 99 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0>
```

Because of the use of the number "16" in its declaration, vectors of the class "short" can only contain integers in the range 0 to (2 ** 16) - 1. Attempting to assign something else will cause an error. E.g.

```
"cat" -> subscrshort(5, shortie);

;;; MISHAP - INTEGER 0 TO 65535 NEEDED
;;; INVOLVING:  cat
;;; DOING    :  subscrshort ....
```

By contrast, the word "undef" is used as the default value for fields of unconstrained vectors. For example, create a five element ordinary vector:

```
vars vec5 = initv(5);

vec5 =>
** {undef undef undef undef undef}
```

Ordinary vectors have associated procedures initv, consvector, isvector, subscrv, destvector. See REF VECTORS

The anomaly that some of the names end in "v" and some in "vector" is a historical accident. Similarly for strings the corresponding procedures are inits, consstring, isstring, subscrs, deststring. (See REF STRINGS).

For creating ordinary vectors the 'curly braces' can be used to simplify the syntax.

There are several procedures that apply equally to different sorts of vectors, notably <> which can be used to join two vectors of the same type, e.g. two strings, or two full vectors:

```
'one string ' <> 'and another' =>
** one string and another

{one vector} <> {and another} =>
** {one vector and another}
```

As explained above, there are other generic procedures that apply equally to vectors and records, e.g. appdata, fill, explode, datalist, copydata.

9.4 OBJECTCLASS: an Object-Oriented extension to Pop-11

Note: in some versions of Poplog it will not be possible to compile Objectclass directly, with the "lib" or "uses" command. In that case try

compile('$usepop/pop/lib/proto/objectclass.p');

LIB OBJECTCLASS makes available an object-oriented programming (OOP) extension to Pop-11. This provides a natural extension to the notion of a recordclass with the added features:

o Different classes can share fields, and associated procedures.

o Information can be inherited from one class to other classes (its sub-classes). In Objectclass "multiple inheritance" is supported, so that classes can inherit from more than one superclass.

o Procedures (known as methods) can be defined which behave differently depending on the classes of their arguments. This is already true of some built in procedures: e.g. the arithmetic operators such as + behave differently depending on whether their arguments are integers, bigintegers, decimals, ratios, etc. Because their behaviour depends on the classes (or types) of more than one argument they are sometimes known as "multi-methods". Objectclass includes multi-methods.

o Object Oriented programming also adds an element of "data encapsulation" or "data hiding" in that details of an implementation of a particular type may be hidden from the users of that type. (This is not unique to OOP: it is an aspect of all good programming, and was already an aspect of vectors and records in Pop-11.)

Some OOP systems, e.g. Smalltalk, are based on the notion of "message passing". This is not used by Objectclass: it follows the Common Lisp Object System (CLOS) in using multi-methods instead. All the behaviour of a message passing system can be obtained as a special case.

The Objectclass library is remarkable in the way that it neatly maps OOP ideas onto existing Pop-11 ideas. The correspondence of ideas is as follows :-

```
methods:
    procedures, which act according to the type of their
    arguments.

instances:
    records, the instance variables of which are simply the
    fields of the record and are accessed by appropriate methods.

classes:
    keys, although the keys that  play the role of classes are
    given some extra fields (via hidden properties) so that they can
    provide inheritance of behaviour.
```

Because this is a natural mapping, the overhead of a method call is relatively low. This means that the objectclass package provides an efficient means of doing object-oriented programming.

If your Poplog system includes Objectclass, you can get an introduction by looking at TEACH OBJECTCLASS_EXAMPLE. More detailed technical information is provided in REF OBJECTCLASS.

9.4.1 Some examples of the use of Objectclass

Here is a simple definition for an objectclass to represent people. We assume each person has a name, an age and a sex. The class definition has "slots" for each of these. We can specify default values for some of these attributes, as in the following example:

```
define :class person;
    slot person_name = "no_name";
    slot person_age  = 0;
    slot person_sex;
enddefine;
```

Slots for which no default value is provided will have the word "undef" as value.

You can then use this to create an instance of the class. Various syntactic forms are available for this. Here is one of them, to create an instance of class person, where the variable xxx will have the new instance as its value:

```
define :instance xxx:person;
    person_age = 27;
    person_sex = "female"
enddefine;

;;; look at the value of xxx
xxx =>
** <person person_name:no_name person_age:27 person_sex:female>

;;; xxx can be given a name, as if it were a record with a field
;;; accessor/updater person_name

"mary" -> person_name(xxx);
xxx =>
** <person person_name:mary person_age:27 person_sex:female>
```

9.4.2 Defining an objectclass method

Objectclass methods are procedures that act differently depending on the types (keys) of their arguments. Compare the way the Pop-11 operator ”<>” can be applied to words, lists, strings, procedures, etc.

Different definitions of a method are needed for each combination of objectclasses. For example, you could write a method for giving a person a birthday.

```
define :method birthday( p:person );
    lvars age;
    person_age(p) + 1 -> age;
    age -> person_age(p);
    [Happy birthday ^(person_name(p)) - now aged ^age] =>
enddefine;
```

This method can be applied to xxx

```
person_age(xxx) =>
** 27

birthday(xxx);
** [Happy birthday mary - now aged 28]
person_age(xxx) =>
** 28
```

It would be possible to define a different version of the ”birthday” method for cars. For example, the car version might print out a reminder to get the the car’s licence renewed and its insurance revalued. Similarly if ”child” were a subclass of ”person” then a special birthday method could be defined for instances of child and it would take precedence over the default method for person. E.g. it might light some candles and sing a song.

9.4.3 Defining a subclass of a class

We can define a new subclass of persons, called adults, with default age 18, who are able to have a spouse, as in the next example. The spouse may be nonexistent, which we'll make the default, represented by the boolean false. Note that the second line of this definition specifies that the adult class is a subclass of class "person". That means that all the slots and methods previously defined for persons will automatically be applicable to adults.

```
define :class adult;
    is person;                          ;;; specify superclass
    slot person_spouse = false;         ;;; and additional slot
    slot person_age = 18;               ;;; A new default age
enddefine;

define :instance yyy:adult;
    person_sex = "male";
enddefine;
```

Let's give yyy the name "fred".

```
"fred" -> person_name(yyy);
yyy =>
** <adult person_name:fred person_age:18 person_sex:male
        person_spouse:<false>>
```

The TEACH files referred to give more information including how to define a method called marry, that is applicable to two adults and makes each of them the spouse of the other.

The Graphical Object library gives many more examples, used for defining objects that can be useful components of graphical interfaces, some of whose methods are invoked by actions performed by the user with a mouse.

[To be continued]

APPENDIX

9.4.4 Additional topics and relevant documentation files

The previous chapters merely provide an introduction to the Pop-11 language. A more complete definition can be found in the online Poplog REF files.

The rest of this section provides some pointers to further information that might be useful.

Words and identifiers

```
words, the dictionary, identifiers, cancel, synonyms,
valof, idval, identprops, identtype, sections
(REF * WORDS, REF * IDENT, REF * SECTIONS))
(HELP * WORDS)
```

Ascii character codes, special graphic codes in VED

```
HELP * ASCII, REF * ITEMISE/Graphics
```

Data structures, classes,

```
defclass, recordclass, vectorclass keys, conskey, class_print,
class_apply, etc.
(REF * DEFSTRUCT, HELP * RECORDCLASS, REF * DATA, REF * KEYS)
```

Control facilities for use in Pop-11 procedures

```
chain, chainto, chainfrom, breakto, breakfrom, catch, jumpout, etc.
 (REF * PROCEDURE, HELP * CONTROL)
```

Character and item streams

```
discin, discout, charin, charout,
itemiser, incharitem, proglist
macros, syntax words, and code planting procedures
(REF * CHARIO,  REF * ITEMISE)
```

File handling

```
syscreate, sysopen, sysread, syswrite, sysdelete, pop_file_versions
discin, discout, discappend
(REF * SYSIO)
```

Pipes and mail boxes

```
syspipe, pipein, pipeout
(REF * SYSIO)
```

Arrays

```
newarray, newanyarray, arrayvector, boundslist
 (REF * ARRAYS)
```

Interrupt handling

```
interrupt, vedinterrupt, popready,
signal handling (sys_send_signal, sys_signal_handler)
(See REF * SYSTEM, REF * SIGNALS)
```

Active variables (with associated procedures that run when the variable is accessed or updated).

```
HELP ACTIVE, HELP DLOCAL, REF IDENT
REF * active, * nonactive
```

Dynamic local expressions (expressions whose values are automatically accessed and saved on procedure entry and restored via updater procedures on procedure exit):

```
HELP DLOCAL
REF VMCODE has two relevant sections, entitled
    --  Dynamic Local Expressions
    -- More On Dynamic Local Expressions
```

Processes ("lightweight processes")

```
consproc, consprocto, runproc, suspend, resume
(REF * PROCESS, HELP * PROCESS,
HELP * ACTOR (may be called NEWACTOR))
```

Library mechanisms,

```
autoloading, search lists, popautolist, popuseslist,
document browsing facilities, vedgetsysfile
Different categories of documentation and libraries
private help (etc) libraries
(REF * LIBRARY)
```

'fast_' procedures. Efficiency "tips"

```
See HELP * EFFICIENCY, REF * FASTPROCS
```

The garbage collector and memory management procedures.

```
sysgarbage, popgcratio, popmemlim, pop_callstack_lim
(REF *SYSTEM)
```

Error handling and warnings.

```
prmishap, mishap, prwarning, prautoloadwarn, popwarnings
(REF * MISHAPS)
```

Timing facilities

```
time, profile, timediff, sys_timer,
(See REF * TIMES)
```

Debugging, tracing and profiling aids

```
(REF * TRACE, HELP * DEBUGGER, HELP * PROFILE)
```

Printing

```
pr, syspr, sys_syspr, =>, ==>, pop_pr_quotes, pop_pr_radix
(REF * PRINT)
```

VED as a general purpose front end

```
(REF * VEDPROCS, REF * VEDCOMMS, REF * REGEXP from Version 14.5)
```

Calling external procedures

```
(HELP * EXTERNAL
REF * EXTERNAL, REF * EXTERNAL_DATA, REF * DEFSTRUCT)
```

Saved images

```
syssave, sysrestore, sys_lock_system
(REF * SYSTEM, HELP * MKIMAGE, HELP * SYSSAVE)
```

Initialisation, and startup files.

```
init.p, vedinit.p vedinitfile, vedfiletypes etc.
(REF * SYSTEM, REF * VEDTERMINALS
HELP * INITIAL, HELP * TERMINAL, HELP * VEDFILETYPES)
```

Exiting Poplog

```
sysexit and related procedures, popexit, vedpopexit
ved_xx, ved_qq
(REF * SYSTEM)
```

Syntactic sugar available in Pop-11

```
    switchon
    form
    prefix
```

Defining new define_forms

```
HELP * DEFINE_FORM
```

Defining new looping constructs

```
HELP * FOR_FORM
```

A selection of useful or interesting library packages

```
database, add, remove, foreach, forevery,
sets
profile
showtree
vturtle/turtle
grammar/tparse/facets
format
msblocks
prefix
```

Programming style in Pop-11

```
(TEACH * PROGSTYLE, HELP * EFFICIENCY)
```

The X window facilities in Poplog Pop-11 are very rich and complex. You might start with

```
HELP * X
REF * X
TEACH * RC_GRAPHIC
TEACH * PROPSHEET
    (Warning: up to version 14.2 the latter is incomplete!)
```

For a more complex list of detailed REF files expanding on the definition of Pop-11 see REF * INDEX

For a list of Pop-11 ref files concerned with the X interface see

```
REF * XPOPINDEX
```

The file HELP NEWS always gives information on recent changes to Poplog. It will usually refer you to older versions of the file giving a reverse chronological history of the development of Pop-11 and Poplog. There are other news files for Prolog, Lisp, ML, and X.

9.4.5 Overview of REF files

The following is an extract from REF * REFFILES

```
REF *INDEX
    --- List of REF files

REF *ARRAYS
    --- Arrays and array procedures (see also HELP *ARRAYS)

REF *ASYNC
    Describes asynchronous traps and signals in Poplog.

REF *CHARIO
    --- Character stream input and output (See also HELP *IO)
```

REF *DATA
    --- General data procedures and datatypes (See also
    HELP *PROGRAMMING)

REF *DEFSTRUCT
    --- POP11 syntax for defining and accessing structures

$usepop/pop/ref/doc_index
    --- This is not a ref file, as explained above.

REF *DOCUMENTATION
    --- The Poplog online documentation system

REF *ENVIRONMENT_VARIABLES (Unix only)
    --- Environment variables defined by the Poplog system.

REF *EXTERNAL
    --- Loading & calling external procedures from Poplog (See also
    HELP *EXTERNAL, *PROGRAMMING)

REF *EXTERNAL_DATA
    --- Using external data structures in Poplog

REF *FASTPROCS
    --- Fast procedures (See also HELP *FASTPROCS, LIB *SLOWPROCS)

REF *FLAVOURS
    --- The flavours object oriented language (See also
    HELP *FLAVOURS). Made obsolete by Objectclass.

REF *IDENT
    --- Identifiers (constants and variables)

REF *INTVEC
    --- Signed integer vectors (See also REF *DATA, *VECTORS,
    *STRINGS, HELP *PROGRAMMING)

REF *ITEMISE
    --- Itemisation and lexical syntax. See also REF *POPCOMPILE

REF *KEYS
    --- Information on classes and keys (see also HELP *PROGRAMMING,
    *CLASSES, *DATASTRUCTURES)

REF *LIBRARY
    --- Poplog library mechanisms, including autoloading

REF *LISTS

--- Lists and pairs (see also HELP *LISTS)

REF *LOGICAL_NAMES (VMS only)
--- Logical names defined by the Poplog system.

REF *MISHAP_CODES
--- Some error messages include a "code" abbreviating the message. This file includes full explanations and cross-references to relevant information about the error. (See also HELP *MISHAP)

REF *MISHAPS
--- Poplog mishap (error) handling (See also HELP *MISHAP, *PROGRAMMING)

REF *NEWC_DEC
--- Procedures concerned with the "new" interface to externally loaded C programs. See HELP * NEWC_DEC for more details.

REF *NUMBERS
--- Number data types and numerical procedures (See also HELP *NUMBERS)

REF *OBSOLETE
--- Obsolete features still supported for backwards compatibility

REF *POPCOMPILE
--- POP11 compiler procedures

$usepop/pop/ref/popindex.*
--- These files (which really should be in another place) are used by the *POPINDEX and *VED_SOURCEFILE mechanisms.

REF *POPSYNTAX
--- POP11 syntax in diagram form

REF *PRINT
--- Printing procedures, etc (See also HELP *IO, HELP *PRINT)

REF *PROCEDURE
--- The nature of procedures and closures, list of predicates operating on procedures (See also HELP *PROGRAMMING, *DEFINE)

REF *PROCESS
--- The Poplog "process" mechanism. (See also HELP *PROCESS)

REF *PROGLIST
--- The input stream used in Poplog by many system modules,

including the POP11 compiler. (See also REF *ITEMISE, REF *CHARIO, HELP *PROGRAMMING)

REF *PROLOG
--- Procedures that support the Prolog system in Poplog

REF *PROPS
--- Properties (association tables) (See also HELP *PROPERTIES)

REF *PWM
--- The original Poplog Window Manager (now defunct, as it requires Sunview).

REF *RECORDS
--- The reference and boolean data types (See also HELP *RECORDS)

REF *REFFILES
--- Overview of REF files, with the information presented here.

REF *REFFORM
--- Describes the format of REF files, explains the conventions used to indicate the types of arguments and results of procedures, and provides some VED utilities to help creation of REF files in the proper format.

REF * REGEXP
--- Describes the Pop-11 regular expression matcher, used in VED and other Poplog facilities, for searching in strings.

REF *SECTIONS
--- Sections (hierarchies of permanent identifiers) (See also HELP *SECTIONS)

REF *SHADOWCLASS
--- Mechanisms for bridging the gap between Pop-11 data structures and external structures, e.g. for C programs.

REF *SIGNALS
--- Contents now transferred to REF *ASYNC.

REF *SOCKETS
--- A set of procedures for creating and operating on Unix sockets.

REF *STACK
--- The POP11 stack and procedures for operating on it (See also HELP *STACK, TEACH *STACK)

REF *STRINGS
--- Strings (See also HELP *STRINGS)

REF *SUBSYSTEM
--- Describes mechanisms for handling sub-systems like Prolog, Pop-11, Lisp, ML and switching conveniently between them.

REF *SYNTAX
--- Information on POP11 syntax words (See also *PROGRAMMING)

REF *SYSIO
--- Device Input & Output procedures, including pipes and mailboxes

REF *SYSTEM
--- Various Poplog system control procedures, including starting up, interrupts, saving and restoring saved images, etc.

REF *SYSUTIL
--- Operating System utility procedures

REF *TIMES
--- Date and Time and Timer procedures

REF *TRACE
--- Information about tracing Pop-11 procedures.

REF *VECTORS
--- Standard full vectors (see also HELP *VECTORS)

REF *VED
--- Overview of REF files about the Poplog editor, VED

REF *VEDCOMMS
--- Summary of VED <ENTER> commands

REF *VEDPROCS
--- Summary of VED system procedures

REF *VEDTERMINALS
--- Summary of VED terminal types and terminal initialisation procedures

REF *VEDVARS
--- Summary of VED's global control variables

REF *VMCODE
--- The Poplog Virtual Machine (See also TEACH *VM)

```
REF *WORDS
    --- Words and the Poplog dictionary mechanism.

REF *WVED
    --- Variables and procedures associated with Windowed VED,
    i.e. XVed and PWM.
```

There are additional REF files to be found in

```
$usepop/pop/x/ved/ref
$usepop/pop/x/pop/ref
```

and, at least in Poplog versions 14.5 and 15.0

```
$usepop/pop/lib/proto/go/ref
    --- Information about the graphical objects system

$usepop/pop/lib/proto/objectclass/ref
    --- Information about the objectclass system
```

In later versions of Poplog those two sets of files may be moved somewhere else.

9.5 VMS DCL or UNIX shell commands in Pop-11

(A) For UNIX users:

You can give a UNIX Shell command by typing a dollar as the first symbol on the line, followed by the command, e.g.

```
$ ls -l *.p
```

On UNIX if you use a percent symbol '%' instead of the dollar, you'll get the CSHELL rather than the SHELL. In that case you can use the character '~' in file names to identify a user's login directory. E.g.

```
% ls ~fred
```

will get a listing of fred's directory (if it is readable by you) whereas

```
$ ls ~fred
```

will produce a message about ~fred being "not found".

If you use '!' as the shell "escape" character, then you will get whichever shell is the currently value of the Unix environment variable $SHELL

To switch temporarily to a sub-SHELL, so that you can give a series of commands, type one of '%', '$' or '!' on its own, at the beginning of a line. Pop-11 will be temporarily suspended, and you can give SHELL commands. Later, you can leave the shell by typing the end of file character, or typing 'exit', after which you will return to where you were in POP, or VED.

(B) For VAX VMS users:

DCL commands can be given by typing the dollar as the first symbol on the line, e.g.

```
$ show time
```

or

```
$ dir *.p
```

If you type the dollar on its own at the beginning of a line it will spawn a new sub-process running DCL. You can type several DCL commands, and then type "q" to terminate the DCL process and return to Pop-11.

See also the following facilities for invoking a shell or DCL from a VED window:

```
ved_imcsh ved_imsh ved_imdcl
```

9.6 ADDITIONAL READING

```
See
TEACH * TEACHFILES

HELP * POP
    Summary of online documentation about Pop-11.

REF * REFFILES
    Overview of the online REF files which give the "definitive"
    definition of the language.

HELP * OBJECTCLASS
    This gives an overview of the main object oriented extension to
    Pop-11, still under development.
    TEACH * OBJECTCLASS_EXAMPLE
        This can be read after doing:
                lib objectclass

TEACH * FLAVOURS
    An older object oriented extension to Pop-11, based on message
    passing and much influenced by Smalltalk.

HELP * DOCUMENTATION
    Gives an overview of online documentation available in Poplog.

See also the books and articles in
HELP * POPREFS

James Anderson, editor, Pop-11 Comes of Age
Ellis Horwood, 1989

This is a collection of papers on the history of dialects of Pop, the
features and benefits of the language, and some applications using
Pop-11.
```

--- $poplocal/local/teach/primer