Information Technology: Research and Development (1983), *2* (109–122)

POPLOG: A MULTILANGUAGE PROGRAM DEVELOPMENT ENVIRONMENT

A. SLOMAN, S. HARDY AND J. GIBSON

Cognitive Studies Programme, University of Sussex, Brighton BN1 9QN, UK

(*Received 5 April 1983*)

ABSTRACT

POPLOG, an integrated combination of the programming languages POP–11, PROLOG and LISP is described and its applications in artificial intelligence research are discussed. A system overview describes how PROLOG and LISP are built on top of the core language POP–11, and how the system can be used interactively. The facilities provided for program development are then described, followed by a breakdown of the subsystem. Some examples of the system in operation are also provided.

1. INTRODUCTION

POPLOG is an integrated combination of the programming languages POP–11, PROLOG and LISP. The core language, POP–11, is very similar in power to LISP, though more conventional in appearance. Both provide most features found in conventional programming languages, but are also interactive and provide unusual facilities to reduce program development time.

POP–11 is a dialect of POP–2, originally designed at Edinburgh University for research in Artificial Intelligence, and extensively developed at Sussex University. Unlike some LISP systems, POP–11 uses an incremental compiler for efficiency. Unlike most compiled languages, it is interactive, and individual procedures can be recompiled without relinking. Other dialects of POP–2 exist, notably WonderPOP for the DEC–10, but all share the important features of the original, described in (Burstall 1971). An extended version of POP–11, which is extensively optimized, is used for building the system.

POPLOG also includes PROLOG, the 'logic programming' language (Kowalski, 1979). The syntax is compatible with that of the DEC–10 PROLOG as described in (Clocksin, 1981). PROLOG and POP–11 are integrated in that POP–11 procedures can call PROLOG procedures and *vice versa*. Moreover, the same screen editor VED can simultaneously be used to manipulate both sorts of files, taking appropriate default action. Thus, in a complex design it is possible to implement modules in whichever language is more suitable. Combining a relatively conventional AI language with a 'logic programming' language permits programs to have the best of both worlds. For those who prefer a more uniform syntax there is a PROLOG-like extension to POP–11, using POP–11 syntax.

An incremental LISP compiler is also included, so far used mainly for teaching and demonstration purposes. It too is integrated into the system, including the editor. LISP is at present the most widely used language for artificial intelligence research, although there are many different dialects, not all compatible! The POPLOG dialect is designed to support the examples given in Winston's 1977 textbook on AI. Like MACLISP it uses an incremental compiler rather than an interpreter, although an interpreter could easily be added. LISP is an older language than POP-11, and in most implementations has less clean semantics. The syntax of LISP is very elegant and economical, and has many addicts. However the richer syntax of POP-11 makes it more readable and allows more extensive compile time checking, thus reducing the time spent investigating run-time errors.

All three languages compile into a common virtual machine language, POP-ASSEMBLER. This in turn is compiled to the machine language for the host machine, for efficiency. The code generator is one of the few parts of the system which have to be changed for different machines. In system-building mode, instead of compiling POP-ASSEMBLER to machine code, there is a special program, POPAS, which generates ASSEMBLER for the host machine. This design makes it possible to extend the system to provide additional compilers, e.g. it should be possible to add incremental interactive compilers for PASCAL and FORTRAN.

2. WHY USE SEVERAL LANGUAGES?

There are several different reasons. The most important is that different tasks can be best served by different languages. Moreover, there may be library programs written in different languages which it would be wasteful to convert. POPLOG not only combines interactive AI languages, but also (in the VAX VMS version) allows library programs written in other languages to be linked in.

AI languages and environments have been developed to serve needs which are not restricted to AI research. In fact, POPLOG could be employed for any application where program development costs are significant or where the restrictions built in to more conventional languages introduce design difficulties. Its integral 'help' and teaching facilities can help an experienced programmer to learn AI techniques, and can also be used to teach novices programming. At Sussex University the system is in regular use both for teaching absolute beginners (including Arts students) and for advanced research, including speech processing and image interpretation. In addition, the text-processing facilities provided by the editor are used by many non-programmers for document preparation.

AI languages are often thought to be wasteful because they produce inefficient programs. However, as computing power becomes cheaper, and software development costs rise, run-time efficiency becomes just one factor in a complex equation. Some current attitudes are a relic of the enormous expense of early computers.

When problems are very complex and programming is difficult, it is often worth sacrificing run-time efficiency for greater ease of program development and maintenance. This is especially true when tasks are initially ill defined, and explorations using the computer are required for clarifying the requirements for the final program. For example, the task of defining a system which understands a natural language is ill-defined because we cannot initially specify exactly what understanding a language such as English requires. Accordingly, AI researchers have devised programming languages and environments which emphasize ease of program development and interactive exploration of ideas in order to clarify and refine them.

3. EXTENDABLE LANGUAGES

Since different problems may require different formalisms for naturally representing information and processes, no single programming language can be expected to meet all requirements. Thus, programmers need to be able to tailor the language for the work in hand. For this reason, AI languages are usually extendable, in that the programmer can modify the syntax to simplify the current task. POP-11 provides a powerful 'macro' facility which allows the user to add new syntactic constructions. If POP-11 did not already include the construction

UNTIL condition DO actions ENDUNTIL

then it would be easy to define in terms of IF and GOTO, using the built-in macro and syntax facilities. Some of the constructs used with 'pattern' matching were added to the language by library programs using this extension mechanism, as was a very flexible 'CASES' construct. Even PROLOG, with its totally different syntax, was implemented in the core language, POP-11. Since compiler subroutines are available to the user, this sort of extension can be done very efficiently. Because the full power of the POP-11 system is available at compile time for user-defined macros, the facility has far greater scope than most languages providing macros.

POP-11 and LISP provide the programmer with facilities for numerical computations, and in addition a set of building blocks for nonnumerical computations, including lists, arrays, strings, words, records, properties and procedures. For example, POP-11 procedures can build and manipulate POP-11 procedures. This is used also in the PROLOG system which creates POP-11 procedures to correspond to PROLOG rules. Users can define new data-types.

Unlike more conventional languages, POPLOG data-structures can be type free, i.e. lists can contain arbitrary objects. This makes it possible to write programs with greater generality than in more conventional languages. For instance, there is a *set* manipulation package in the POP-11 library which can be used for sets of arbitrary objects. There is a sorting program which can reorder lists of arbitrary objects provided that the user supplies an ordering predicate. In PASCAL, for example, it would not be possible to define a procedure which takes an argument that is a predicate which could be applied to arbitrary objects: the compile-time type-restrictions provide a gain in run-time efficiency and consistency checking at the cost of intolerable limitations (and increased compilation time). Type restrictions do enable some useful checking to be done by a compiler, but the additional compile-time checking does not compensate for the greatly reduced generality and the restricted run-time debugging aids normally associated with conventional languages.

A possible development of POPLOG would be to provide *optional* compile-time facilities for more elaborate consistency checks and greater efficiency.

4. AI LANGUAGES AND EFFICIENCY

Despite the comparative run-time inefficiency, POPLOG is being used for research on speech and image processing involving 'number crunching'. Moreover, linking in subroutines written in other languages enables code which is critical to be written in the most efficient language. Thus, an image understanding system can use FORTRAN or C for the lowest level algorithms, POP-11 for intermediate processing and PROLOG for the high-level, rule-based processing.

The garbage collector also saves programmer effort, concerned with reclaiming memory space no longer in use. It facilitates frequent recompilation of individual procedures during testing and development, since memory occupied by old definitions is reclaimed. Procedures and structures required merely for the initialization phase of a program can be discarded after the initial structures have been set up. The garbage collector will enable their space to be reused. POPLOG uses a fast 'stop and copy' garbage collector which takes a time proportional only to the amount of memory still in use. A novel mechanism minimizes time taken by garbage collections when a large portion of the workspace is known to be permanently in use. When combined with the very large address space of 33-bit machines, these techniques considerably reduce the proportion of time required for garbage collection.

Further, during program development POPLOG can be more machine efficient than most compiled languages, because the compiler is very fast and need only recompile altered portions of a program. With editor and compiler integrated into the run-time system, recompiling and relinking a modified procedure definition can take only a fraction of a second. For this reason, a machine can support more POP-11 programmers than say C or PASCAL programmers for a given rate of program development per user.

Since much of the system is written in itself, and can therefore be tested interactively during development, the authors were able to develop it very quickly. The initial version of the compiler for the VAX took about three man-months, and the first version of the screen editor about four weeks. The initial PROLOG implementation took a few weeks, and the LISP implementation even less. Because most of the system is written in itself, the process of development and rebuilding makes heavy use of the system, thoroughly testing the compiler, editor etc. This means that most bugs are detected and ironed out before users can suffer from them. New system facilities can first be tested interactively, as if they were ordinary user programs before being optimized and linked into the main system.

Typically, a user will work entirely within POPLOG after logging in. A command to the operating system can be given by typing a line beginning with a dollar symbol, which will cause POPLOG to suspend itself until after the command is obeyed. Many such commands are replaced by POP-11 procedures. For example, the editor can be used to send mail, interrogate directories or purge files. The advantage of remaining within one system is that user programs do not have to be restarted if the programmer finds a need to examine a directory, send some mail or use an operating system utility in the middle of a session. This can save both programmer time and computer time, which would otherwise be required for recreating the context after the interruption.

5. MACHINE COMPATIBILITY

POPLOG was not designed to be squeezed on to small machines. On the assumption that hardware costs, especially memory costs, would continue to fall, programmer convenience and ease of maintenance of the system were given the highest priority. So it will not run on the microcomputers which have been available for the last few years. However, it should be transferrable to the new generation of microcomputers with 32-bit address spaces.

The current preference in AI programming system design is to use specialized hardware providing a personal computer, such as the LISP machine (Weinreb, 1979) or PERQ. These provide a very large address space, excellent graphical display facilities and rapid response from a powerful dedicated processor. However, at present most people cannot afford such specialized hardware. Providing this sort of environment requires a machine with a very large address space for user processes. Given a limited budget with which to provide an interactive service for a large number of users, time-sharing a single virtual-memory machine will be the most economical solution for many groups of users who cannot afford to buy a number of LISP machines or PERQs. The requirement for a large address space rules out the cheaper personal computers. However, as hardware costs fall, it will eventually become more sensible to use a network of personal computers.

6. SYSTEM OVERVIEW

6.1. Languages

The core language is POP-11, with PROLOG and LISP built on top of this. Unlike PROLOG, POP-11 is relatively conventional; it is based on familiar concepts, such as sequential execution, variables, data-structures and procedures, and the syntax has much in common with the ALGOL family. A programmer familiar with, say, PASCAL would not (at first) experience any culture shock. Several students familiar with PASCAL have been able to teach themselves to use POP-11 with relatively little help.

The PROLOG subsystem is very different. It is based on the use of *predicate logic*, which is given both the usual declarative interpretation and also a procedural interpretation. It is especially suitable for the design of programs which are given factual information from which inferences are to be made, for instance the design of databases where much of the information is implicit in general principles. PROLOG can also be used for problem-solving programs where the strategy can be defined in terms of collections of condition–action rules. Fault tracing would be an example. Alternative actions may be associated with the same sorts of conditions, and the system will automatically try out alternatives, using back-tracking where a line of exploration is unsuccessful. See (Clocksin, 1981).

Facilities are provided to enable a PROLOG program to hand control temporarily over to a POP-11 procedure, or *vice versa.*

POP-11 extends POP-2 in a variety of ways. The syntax has been enriched, making it more redundant, so that it is easier to read, and the compiler can give more helpful error messages. In addition, there is a wider variety of looping constructs, a built-in pattern matcher, additional procedures for manipulating data structures (including bypassing run-time type checks where efficiency is crucial), enhanced control facilities, 'autoloading' of library files, more convenient 'section facilities', a mechanism for precompiling frequently used programs, a timed interrupt mechanism, the ability to link in external routines (though not at run time) and procedures providing system calls, including UNIX-like stream I/O.

POP-11 also includes hash-coded association tables, and a process package (e.g. for coroutines, back-track searching etc.). The compiler routines which are available to the user at run time make it possible to modify the compiler so that instead of merely compiling one language it will accept others. All are compiled into a common intermediate language, which is then compiled to machine code for efficiency.

Many (not all) POP-2 programs will run with no alteration in POP-11, as the syntax is handled by autoloadable library 'macros', i.e. procedures which run at compile time and rearrange text before it gets to the compiler. Syntax extensions written in POP-11 can be relatively efficient because POP-11 compiler subroutines can be called directly by user procedures, unlike POP–2 which only allowed editing of the input stream by macros.

6.2. Text editor

POP–11 incorporates a screen editor, called VED, in the run-time system, allowing the user to edit and recompile portions of a program as it is being tested. VED can also be used for interrogating help and reference files, and for producing documentation.

The editor can include several files simultaneously, with switching between them, and more than one can be displayed on the screen at the same time. Although it can be used in batch mode, POPLOG is primarily intended for interactive use. The screen editor is interposed between the user and other subsystems as shown in Figure 1.

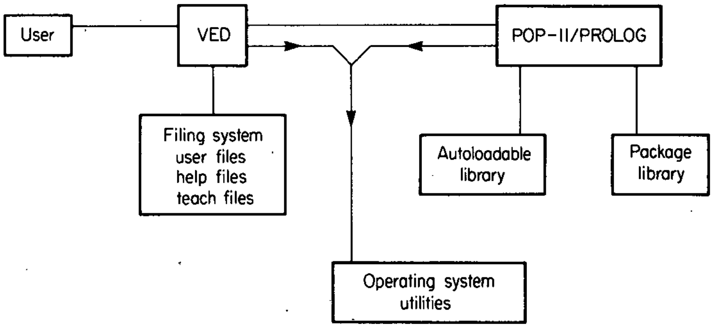

FIG. 2. How the user views the POPLOG system

The user's screen displays a portion of some selected file or files: user files, help files, teach files etc. The editor is used to read teach files which explain how to use the editor! Files can be displayed simultaneously in different 'windows'. Edit windows can scroll left or right as well as up or down, so that long lines can be typed in or read.

Simple editor commands cause part of the current file to be sent to the compiler for POP–11, LISP or PROLOG. The editor decides which on the basis of the file-name suffix '.P', '.LSP' or '.PL'. The system compiles the fragment of text sent to it and sends back any output to VED which splices the output into a file (this may be the current file if desired) and then displays the output on the user's screen. Output stored in an edit file can easily be reviewed after scrolling off the top of the VDU screen, making memory in the VDU unnecessary.

A typical interaction will consist of the user typing in procedure definitions (with documentation) and test commands, pressing the DOIT key and observing the output. If a command or procedure definition needs to be modified, a few keystrokes after editing suffice to have the text recompiled and reexecuted. VED also includes some simple text formatting facilities for nonprogram text, e.g. documentation.

7. THE PROGRAM DEVELOPMENT ENVIRONMENT

Explicit debugging tools are less necessary than with conventional programming languages, mainly because the compiler is part of the run-time system, so that, for example, it can be invoked at break points. This allows the user to give any POPLOG command, not only to examine or change variables, interrogate files etc., but also to edit and recompile procedures. An entirely new procedure, e.g. to print parts of some data structure, can be defined and run during a break point. Break points occur whenever there is an error, whenever code execution reaches a declared break point (set, perhaps, by editing a procedure definition to include a call of the compiler) or when the user interrupts a running program, by typing an interrupt character at the terminal.

A second reason why no separate debugging tools are needed is that POPLOG procedures can be manipulated by POPLOG programs. This means that such debugging tools as are required can be written in POPLOG itself and tailored to the programmer's own needs. For example, a simple built-in program makes it easy to add (or remove) trace printing instructions to specified procedures, although users can easily produce their own variants.

The 'macro' facility makes it particularly easy for users to define commands which can be put into procedures to conditionally interrupt processing, to allow communication with the user who may dynamically alter the conditions. Without changing the source code, the macros can be switched off so that fully tested procedures are compiled as if the debug commands were not included.

During break points, produced by errors, planned pauses or the typing of an interrupt character, the procedure POPREADY enables the user to communicate with the system in the same language as is used for writing programs. In some cases, POP enables execution to be restarted at an earlier point, after one or more procedures have been redefined, or the environment has been altered. This uses the CHAINTO mechanism.

The POP-11 error handler takes default action (including printing an error message) which can be altered by globally or locally redefining the relevant procedure. Temporary redefinitions are often very useful during debugging.

7.1. Library

An essential component of POPLOG is the program library. It includes an 'auto-loading' mechanism that causes library files to be automatically compiled and included into any user program that references them. The user can alter the list of 'autoload' directories to be searched. In addition, there is a library of programs which are loaded by an explicit 'LIB' command. A team of programmers can share an autoload library. The editor can be instructed to alter the autoloadable library depending on what sort of file is being used, since the editor runs a user-definable procedure every time a file is set on the screen.

7.2. Documentation system

There are several levels of documentation: 'help' files, 'teach' files, 'reference' files and 'manuals'. Having a screen editor built into the system simplifies access to documentation for the online user.

There are over 600 help files, and aids to access are planned. Some simple aids already exist. A user who doesn't know the precise name of a help file, but thinks it may include, for example, 'word', can type, to the editor

```
H WORD
```

and will be given an ordered menu of possibly relevant help files, e.g.

```
CONSWORD ISWORD WORDS DATAWORD WORDSWITH
```

Because of liberal cross referencing, finding a partly relevant file will often lead the user to the precise information required.

HELP can be used as a tree-structured menu system, starting with

```
HELP HELPFILES
```

and working down to more and more specific files. However, since the editor allows arbitrary switching between files there is no need to be restricted to a particular tree structure. While looking at file A, you can go up or down within it, or switch to a new file B, or return to where you were in a previous file C, saving your location in A. This overcomes the problem that a tree-structured menu can be too restrictive. To simplify this use of the editor a simple key sequence is provided to invoke HELP on a word indicated by the VDU cursor.

The files intended as off-line documentation can be accessed via the DOC editor command. The off-line documentation is growing more slowly than online documentation, owing to shortage of staff. Off-line manuals and primers will be available later.

7.3. Teaching aids

POPLOG was developed for teaching as well as research. A collection of 'teach files' explains not only aspects of the programming language but also aspects of AI in general.

The approach to using the computer as a tutor is somewhat unconventional. One of the problems of teaching programming is the enormous range of aptitudes revealed by students. This requires a very flexible approach, allowing students to progress at their own speeds. Instead of providing a program which rigidly controls the learning process, monitoring all the students' activities and passing judgement either explicitly or implicitly, the authors favour leaving the student fully in control, with the computer available as a friendly but not very intelligent 'adviser'. Thus, the student working through a teach file using the editor, is free to jump forwards, or backwards, or to switch temporarily or permanently to another teach file. Most importantly, he can switch freely and rapidly between using the compiler to try things out and reading more of the teach file for advice.

Apart from providing information, teach files make suggestions for programming exercises. The student may be asked simply to copy certain commands, in order to see what happens. After that, variants can be explored freely before moving on. The student is in complete control; he can try as many or as few variants as he wishes. He may be invited to write a program to perform some task, or may be given a program and asked to modify it or complete it. The computer does not assess the student's performance. Instead, the student is left to test his own program to see if it performs as required. If not, it is up to him to try to find out what went wrong. As students become more advanced, they find increasing cross references to other files.

8. THE POP-11 SUBSYSTEM

POP-11 is a mixture of relatively conventional and some less conventional features. Although POP-11 has many characteristics in common with languages like PASCAL, it also has a number of features not found in conventional languages. Many POP-11 users have no conception of its full power, and are content to treat it as little more than an interactive PASCAL. Indeed, POP-11 has been used to teach AI and programming to linguistics, psychology and philosophy undergraduates at Sussex University, and these students are encouraged to treat POP-11 as being hardly more complex than LOGO.

8.1. The POP virtual machine

An understanding of the machine underlying POP-11 (and POP-2) is of great help in understanding POP-11 itself. This machine is conceptually very simple, and the mapping between POP-11 instructions and virtual machine instructions is also simple. Expressions in POP-11 are translated into instructions for a 'stack-oriented' machine. For example, the imperative

```
X + Y —> Z;
```

translates into the virtual machine instructions

```
PUSH   X   Put the value of variable X on the stack
PUSH   Y   Put the value of variable Y on the stack
CALL   +   Call the addition procedure, which removes two elements from
           the stack and replaces them by their sum
POP    Z   Remove one element from the stack and store in the variable Z
```

A second 'system' stack is used to save the address of procedures and the values of local variables during procedure calls. For example, the procedure

```
DEFINE TWICE(X); X * 2 ENDDEFINE;
```

translates to:

```
SAVE      X   Save the value of variable X on the system stack
POP       X   Set variable X from the user stack
PUSH      X   Put the value of X onto the user stack
PUSHQ     2   Put the integer 2 onto the user stack
CALL      *   Call the multiplication procedure, which takes two items off
              the stack and puts its result on the stack
RESTORE   X   Restore the value of X from the 'system' stack
```

These instructions are packaged into a procedure record, which is then assigned to the variable TWICE. In the VAX implementation for POP, virtual machine instructions are further translated into VAX machine code; a cleaner solution would be to augment the VAX microcode to recognize POP virtual machine instructions, although on the VAX most POP virtual instructions produce only one or two machine code instructions.

Understanding this two stack mechanism makes it easy to understand many features of POP-11. For example, it is clear that procedures can have more than one result (that is, procedures can leave more than one thing on the stack); it is even possible for procedures to have a variable number of results (though this can be abused in obscure programs). Further, a loop instruction can leave items on the stack to be collected into a list, for example

```
[% FOR X FROM 1 TO 20 DO X ENDFOR %]
```

makes a list of numbers from 1–20. The items left on the stack are collected into a list by '[% . . . . %]'.

This virtual machine is enriched by the provision of a collection of system procedures which may be given as argument to CALL, for allocating memory, manipulating data structures, reading or writing data to files or the terminal etc. Many of these system procedures are themselves defined in terms of the same POP virtual machine, although a few are not and have to be reimplemented for each new computer.

8.2. Variables

POP-11 is a 'dynamically scoped' language, using 'shallow binding', like some LISP implementations. All occurrences of the same variable name (say X) refer to the same location; on entry to a procedure the current value of its local variables are saved and then restored on exit. This contrasts with the technique used in 'lexically scoped' languages, such as ALGOL and PASCAL. Both have advantages and disadvantages. Ideally a language should provide both options.

A feature of POP-11 not usually found in LISP is that procedures are themselves just values of the variables used as their names. This means that redefining a procedure simply requires the system to build and assign a new procedure record to be the value of the variable. If the old procedure is no longer accessible, its place will be reclaimed by the garbage collector.

Moreover, the local variable mechanism can be used to alter, temporarily, the procedure associated with a name, thus changing the behaviour of procedures which use the name. This means that a procedure F which calls the database procedure ADD can be made to behave differently in different contexts by giving the variable 'ADD' different values. Powerful use of this is made by the POP-11 error handler, which, for example, calls a procedure for printing error messages, which can be temporarily redefined by a user procedure to alter the format or even suppress the message. Similarly, the procedure INTERRUPT can be temporarily redefined in certain critical contexts, so that normal interrupts are disabled. This is done simply by making INTERRUPT a local variable of the procedure which needs to give it a new value. On exit from that procedure the old value is automatically restored.

Finally, all the standard printing procedures assume that the characters to be output will go to a certain procedure CUCHAROUT which consumes characters. Normally, this procedure sends the characters to the terminal, or prints them into a VED file, but any procedure which calls the print routines can temporarily redefine CUCHAROUT to do something different, such as printing the characters into a disk file, storing them in an array etc. Using the local variable mechanism, rather than altering global variables on entering and leaving a procedure, ensures that the environment is reset automatically, even if the procedure exits abnormally. This dynamic alterability of procedures is one of the features of POP which is not available with the same generality in most LISP systems.

The simple representation for procedures, combined with the ability to call the compiler recursively, gives the user great flexibility. The user can also write programs which CONSTRUCT procedures by assembling the text for the procedure and then compiling that text. This can be used in writing compilers (like the LISP compiler) and programs which modify themselves.

The intermediate POP–ASSEMBLER language allows system building to be done by altering the final stage of the compiler so that instead of generating machine code, it produces a file of ASSEMBLER for the desired machine. These files can then be compiled using the host assembler, and linked together with a small number of assembler routines written by hand for the host machine. Thus, a version of POP–11 running on one machine can be modified to generate a system to run on another machine, without requiring the other machine to provide a good high-level systems programming language.

The time required for entering and leaving a user procedure (excluding the time required for intervening procedure calls), is either 12 or 17 μs plus four μs for each local variable. The faster time is the result of declaring the procedure name to be of type 'PROCEDURE' so that run-time checks are reduced. Thus POP–11 can achieve up to 80 000 procedure calls per second on a VAX–11/780. Simple system procedures are even faster.

8.2.1. Typed variables

It is not generally recognized that typed variables, as in ALGOL68 or PASCAL, greatly restrict the type of procedure that can be written. Consider, for example, the following procedure to find an element of a given list satisfying a given predicate

```
define find(xs, p);
    vars x;
    for x in xs do
        if p(x) then return(x) endif
    endfor;
    return(false);
enddefine;
```

FIND takes a list (named XS) and a predicate (named P). It applies P to each element X of XS, returning the first X for which P(X) returns a nonfalse result. If no such element exists, the procedure FIND returns FALSE. For example

```
find([I saw 3 ships], isinteger) =>
** 3
find([I saw 3 ships], isprocedure) =>
** <false>
```

(The square brackets are POP–11 syntax for a list, " =>" is the print arrow, causing printout preceded by '**'.)

Such procedures could not be written in a strongly typed language, because the precise type of the list XS and the procedure P, and therefore of FIND, are not known at compile time (it is known that P must return a truth value, but it is not known what type of argument it requires; in fact it might be given *any* type of argument).

The run-time type-checks can, of course, slow processes down in comparison with conventional languages. Some limited facilities are provided for bypassing these checks in robust programs.

8.3. Structure expressions

It is often useful when designing intelligent programs to be able to represent data structures within a program as well as procedure definitions. Convenient syntax for doing this is provided, and can easily be extended using the macro mechanism. In particular, the brackets '[', ']' are used for building lists whose structure is known at compile time, and '{', '}' for building vectors. These structure expressions can be nested within each other to arbitrary depths. Moreover, constants and evaluable subexpressions are readily mixed, since the symbol '%' can be used to switch from quoted to evaluated mode.

The special syntax word 'CONS WITH' can be used to alter the behaviour of the curly braces. For example, if CONSFAMILY is a procedure which takes a collection of names and a number saying how many names, and creates a data structure using those names, then

```
            CONS WITH CONSFAMILY
{TOM MARY %CHILDREN("TOM", "MARY")%}
```

will create a structure with all the names, including those produced by the procedure CHILDREN. The prefix '^' can be used for inserting the value of a variable, and the prefix '^^' is provided for splicing the contents of one list into another.

The authors found the number of trivial programmer errors dropped when this feature was introduced. Also useful is the notion of 'structure matching'. POP-11 incorporates a simple structure matcher which can compare a given data structure to a pattern and 'select' components of the structure as a result of the match.

The macro facility referred to above makes it possible to define new sorts of structure expressions for particular applications. For example, instructions to build a 2D array of characters could be represented by a 'picture' of its contents.

8.4. Looping with the matcher

Suppose the value of the variable DATA is a list of lists of words, and the user wishes to search for a list which includes the words 'I' and 'you'. He also wants to make a list of all the intervening words and assign it to the variable FOUND, which is then to be given to the procedure called PROCESS. He would need to write a set of nested loops in a conventional language. The POP-11 version involves

```
IF      DATA MATCHES [ = = [ = = I ??FOUND you = = ] = = ]
THEN
        PROCESS(FOUND)
ENDIF
```

Here '= =' stands for any collection of elements of the matching list. By using patterns which show the structure as thought of by the programmer, programming errors are considerably reduced.

Patterns may have embedded patterns and may also contain 'restriction specifiers' limiting the possible matches. PROLOG also uses pattern matching. It is more powerful than POP–11, in that patterns can be matched against patterns, and matches can be 'undone' if an alternative has to be tried. Moreover, the PROLOG matcher allows variables to pick up values in a wider variety of ways. However, in some contexts the POP matcher leads to more readable programs.

8.5. Control facilities

POP–11 allows procedures to be recursive or mutually recursive, with local variables whose values are saved on procedure entry and restored on exit, as explained above.

It also provides the usual collection of looping constructs, and a few which are less common.

```
UNTIL condition DO action ENDUNTIL
WHILE condition DO action ENDWHILE
REPEAT number TIMES action ENDREPEAT
REPEAT FOREVER action ENDREPEAT
FOR var FROM number BY number TO number DO action ENDFOR
FOR var IN list DO action ENDFOR
FOR var ON list DO action ENDFOR
```

And two forms which make use of the pattern matcher

FOREACH pattern IN list-of-lists DO action ENDFOREACH, i.e. for every match between the pattern and a list do the action, with pattern variables appropriately bound

FOREVERY list-of-patterns IN list-of-lists
DO action ENDFOREVERY
i.e. do the action for all consistent combinations of matches

QUITIF and QUITUNLESS are available for abnormal exits from loops. For example

QUITIF(condition)(3)

causes control to leave the third enclosing loop if 'condition' does not evaluate to FALSE.

NEXTLOOP(2)

causes the second enclosing loop to be restarted.

IF, UNLESS, ELSEIF, ELSEUNLESS and ELSE are available for multibranch conditional expressions. There is a 'switch' statement which allows evaluation of a numerical expression to control a jump to a label, with a default option if the number is out of range. A very general CASES facility is provided in the form of a library macro.

CHAIN is a procedure which exits from its caller and then runs its argument, allowing one active procedure to be replaced by another. CHAINTO and CHAIN-FROM generalize this by unwinding the control stack to a specified active procedure (or just beyond it) and then running the new procedure. EXITTO and EXITFROM, CATCH and THROW, and JUMPOUT are additional facilities for 'abnormal' procedure exits.

SYSSETTIMER takes a procedure and an integer specifying a time interval, and causes that procedure to interrupt processing after the time interval. This can, for instance, be used to write a scheduler for subprocesses.

The 'process' mechanism allows the state of a computation to be saved and then resumed later on. This facilitates the design of a program as a collection of co-operating processes communicating by message sending. It also permits 'back-tracking' or 'nondeterministic' programs.

The logic programming subsystem also allows programs to save their state for the purpose of backtracking if continued execution reaches a failure point. POP-11 also provides GOTO and labels, although it is rarely necessary to use them, given the other more structured facilities for transferring control.

ACKNOWLEDGEMENTS

The authors would like to thank C. Mellish, J. Cunningham, R. Evans, S. Isard, D. Hogg, T. Khabaza and D. Roberts.

REFERENCES

Burstall, Collins and Popplestone (1971) *Programming in POP-2.* Edinburgh, UK: Edinburgh University Press.
Kowalski, R. (1979) *Logic for problem solving.* Berlin: Springer-Verlag.
Clocksin, W. and Mellish, C. (1981) *Programming in PROLOG.* Berlin: Springer-Verlag.
Winston, P. H. (1977) *Artificial Intelligence.* London: Addsion Wesley.