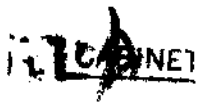
THE POPLOG PROGRAMMING SYSTEM

Steven Hardy

November 1982



Cognitive Studies Research Paper

Serial no: CSRP 003

Th University of Sussex
Cognitive Studies Programme
School of Social Sciences
Falmer
Brighton BN1 9QN

ABSTRACT

This chapter describes a typical Artificial Intelligence (AI) programming system and shows how it differs from conventional programming systems. The particular system described is POPLOG. It incorporates a powerful screen editor, a PROLOG compiler and a POP-11 compiler. POP-11 is a dialect of POP-2 which has been extensively developed at Sussex University. Other dialects of POP-2 exist, notably GLUE and WonderPOP, but all share the important features of the original which is described in [BURSTALL 71].

It is assumed that the reader is familiar with a range of conventional programming languages, such as PASCAL and BASIC. It is shown that AI programming systems, and in particular POPLOG, are used because of advantages independent of AI itself. In fact, POPLOG could usefully be employed for any application where program development costs are significant.

1) INTRODUCTION

Artificial Intelligence (AI) research involves, among other things, making computers do tasks that are easy for people but hard for computers - such as understanding English or interpreting pictures. Since this is difficult, AI researchers use programming systems which facilitate the development of programs and are prepared to sacrifice some run-time efficiency to pay for this. Programs written using AI programming systems usually need more memory and run more slowly than equivalent programs written with conventional programmins systems. However, with the decreasing cost of computer hardware, AI programming systems are now within the economic reach of many computer users and there is now no justification for non-AI programmers to use poor program development tools simply because of their cost.

This chapter describes the POPLOG programming system developed at Sussex University. It is currently implemented on the VAX range of computers manufactured by Digital Equipment Corporation (DEC). These computers are classified as large mini-computers. They have a 32-bit word size which means that programs can become *very* large - many hundreds of times larger than is possible with a 16-bit machine such as the IBM Personal Computer. A medium sized VAX computer at Sussex easily supports around twenty simultaneous users of the POPLOG programming system. To ensure portability, POPLOG is based on VMS, the standard operating system provided by DEC. (A UNIX version is in preparation and a PERQ implementation is planned for late 1983.)

software is a little old fashioned. The current vogue in AI programming system design is to use specialized hardware such as the LISP machine (see [WEINREB 79]). LISP is the AI programming system most used in the United States. The LISP machine is a special type of computer designed to run LISP with the same efficiency as a conventional system runs on a conventional computer. The LISP machine is a single user system - each programmer has his or her own computer, each broadly comparable in power to a VAX. We simply could not afford to buy twenty LISP machines and neither, we think, will many potential users of the POPLOG system.

The phrase 'programming system' is used when referring to POPLOG in place of the more usual 'programming language' because POPLOG, like BASIC, provides a complete 'environment' for the programmer to work within. Moreover, there are two programming languages available in POPLOG, POP-11 and PROLOG. Typically, a POPLOG user will invoke the POPLOG system immediately after logging in and remain there until returning to the monitor to log out. Within the POPLOG system, the programmer can perform all the activities associated with programming. Programs can be edited, compiled, debugged and documented all without once leaving POPLOG. Full access to system utilities, such as MAIL, is available. The POPLOG programming system includes all the following:

* VED, an extendible screen editor, which can display up to two 'windows' simultaneously. Among the comments made on this are 'the best available editor for the VAX' and 'worth using POPLOG just for the editor'. VED can be used to prepare both programs and documentation.

* A POP-11 compiler, which translates POP-11 programs into VAX machine code. The compiler can be invoked as a subroutine by user programs.

* Debugging tools, which enable the programmer to locate program faults. These work at the source code level; the programmer need not be aware that POP-11 is compiled into machine code. For example, during breakpoints the user can give arbitrary POP-11 commands.

* An on-line documentation system allows the user to access HELP and TEACH files which give aid and instruction respectively on how to use the POPLOG system.

* A PROLOG compiler (see chapter by Clocksin in this volume), which translates PROLOG programs into VAX machine code. PROLOG provides a relational database with limited theorem proving capability.

There are numerous advantages in combining these component into a single system. For example, while the editor is being used the state of the user's computation (such as constructed data structures) is retained. The editor and compilers share memory and so can easily communicate. When, for example, a POP-11 procedure is edited only that one procedure need be recompiled and not the entire program. Programs can be written partly in POP-11, a sequential procedural language, and partly in PROLOG, a backtracking declarative language. This provides a significant advantage over stand-alone PROLOG systems where everything must be written in 'logic' even those parts more naturally or efficiently expressed procedurally.

Procedures written in other programming languages (such as FORTRAN or C) may be incorporated into programs written using POPLOG. Although these 'external procedures' may not be altered during a single session with POPLOG this nevertheless provides a very powerful facility. A vision program, for example, may do low level array manipulation in FORTRAN, intermediate level processing in POP-11 and high level scene analysis in PROLOG.

Since PROLOG is described elsewhere in this volume, in this chapter we concentrate upon the POP-11 programming language and how it interacts with the VED screen editor (similar interactions are possible between PROLOG and VED).

2) THE POP-11 PROGRAMMING LANGUAGE

In this section a number of examples of programs written in POP-11 are presented. They get increasingly complicated. Although POP-11 has many characteristics in common with languages like PASCAL it also has a number of advanced features that permit programs to be written to solve complex problems. The majority of POP-11 users have no conception of its full power; many are content to treat it as little more than an interactive PASCAL. Indeed, POP-11 has been used to teach AI and programming to hundreds of psychology and philosophy undergraduates at Sussex University and these students are encouraged to treat POP-11 as being of the complexity of LOGO.

The final part of this section is a discussion of the way POP-11 is implemented. POP-11 is written almost entirely in POP-11 and so can easily be extended.

2.1) AN OLD FAVOURITE: FACTORIAL

Bowing to an old tradition, we present as our first example of a POP-11 procedure the definition of FACTORIAL. The simplest way of writing this procedure is recursively, thus:

```
define factorial(number) -> result;
    if number = 1 then
        1 -> result
    else
        number * factorial(number - 1) -> result
    endif
enddefine;
```

We can invoke this procedure, thus:

```
factorial(5) =>
** 120
```

The symbol '=>' is called the print arrow and instructs POP-11 to print out the value of the expression to its left. The output is preceded by two asterisks, for example:

```
factorial(5) + factorial(5) =>
** 240
```

The fact that the formal parameter to FACTORIAL is called NUMBER has no significance to the POP-11 compiler. It might just as well have been called, say, X. Any type of object (eg number or string) can be assigned to any variable. This is not to say that POP-11 is a typeless language. Every object has a type and it is not possible, for example, to subtract a number from a string. The subtraction procedure checks at run time that the arguments it has been given are numbers and if not it reports an error to the user. This would happen if we applied FACTORIAL to, say, a string:

```
factorial('seven') =>
MISHAP:    NON NUMBERS FOR ARITHMETIC OPERATION
INVOLVING: 'seven' - 1
DOING:     factorial
```

The definition of FACTORIAL uses what is termed an 'output variable', in this case called RESULT. Whatever is 'assigned' to this variable is the result of the procedure call. Assignment statements in POP-11 go from right to left, thus:

```
expression -> variable
```

For example:

```
1 -> result
```

This assigns the number 1 to the variable RESULT. The final value of this variable is the result of the procedure call. As with the formal parameter NUMBER, there is no significance to the word RESULT. The procedure could have been equivalently written as:

```
    if x « 1 then 1 -> y else x * factorials - 1) -> y endif
enddefine;
```

Notice, too, that newlines have no significance in POP-11.

FACTORIAL can also be defined iteratively, thus:

```
define factorial(number) -> result;
    vars index;
    1 -> result;
    for index from 1 to number do
        index * result -> result
    endfor
enddefine;
```

The second line of this definition uses the keyword VARS to tell the POP-11 compiler that the variable INDEX is 'local[1] to this procedure. The semicolon after the declaration is a 'separator[1]. If a procedure definition has several steps then they must be separated by semicolons. This procedure has two main steps - assigning 1 to RESULT and a 'for loop[1]. Notice that the value of the variable RESULT is changed several times when this procedure is applied; it is the final value of RESULT that is the result of the procedure.

The meaning of 'for loops[1] in POP-11 is very similar to other programming languages. In this case the controlled statement:

```
index * result -> result
```

is executed with INDEX being first 1, then 2, then 3 and so up to the value of NUMBER.

Notice that in POP-11 most syntactic constructions have an opening keyword (eg FOR) and a closing keyword formed by prefixing END onto the opening keyword (eg ENDFOR). Although this is verbose it is easy to read and remember and means that when it detects syntax errors the POP-11 compiler can give more useful information than would be possible if the same closing keyword were used for many constructions (as in LISP).

example, can occur anywhere in the body of a loop and immediately stops the loop. QUITLOOP is syntactic sugar for a simple GOTO to an implicit label placed after the end of the loop. Other 'jumpout' constructions in POP-11 are more powerful. CATCH and THROW, for example, provide a structured form of non-local GOTOs.

2.2) VARIABLES

POP-11 is a 'dynamically scoped' language. This refers to the way 'free variables' are treated. Had the variable INDEX not been declared as local to FACTORIAL then the POP-11 compiler would have looked in the environment of the procedure that invoked FACTORIAL to see if INDEX were a local variable of that. If not, the POP-11 would look in the environment of that procedure's invoker and so on until it either found an active procedure with INDEX as a local variable or else reached the end of the procedure calling chain and found the global variable INDEX. This is to be contrasted with the mechanism used in 'lexically scoped' languages such as ALGOL where free variables are looked for in lexically enclosing procedures. A technique called 'shallow binding' is used to ensure that POP-11 variables can be accessed extremely rapidly (with only one memory reference). See section 2.8 on how POPLOG is implemented for more details.

Surprisingly, procedure definitions in POP-11 are simply syntactic sugar for assignment statements. When a definition of, say, FACTORIAL is encountered the POP-11 compiler simply assigns a 'procedure record' to the variable FACTORIAL. Thus the following is permissible:

```
factorial -> temp;
temp(5) =>
** 120
```

```
5 -> x;
factoriaUx) =>
** 120
factorial -> temp; x -> factorial; temp -> x;
x(factorial) =>

** 120
```

Treating procedures as objects like any other, eg numbers, which can be moved around from variable to variable has many advantages. For example, it is easy to redefine procedures in POP-11 - one simply assigns a new procedure to an existing variable. Also, procedures may be passed as arguments, returned as results, stored in data structures and, as we see later, dynamically constructed like any other data structure! (See section 2.6 on procedure closures for an example of this.)

2.3) DATA STRUCTURES

2.3.1) SYSTEM DATA TYPES

P0PL06 includes a rich set of ~~predefined~~ types of data structure including:

integers	in the range -536870911 to 536870911
decimals	'real' numbers (both single and double precision)
strings	one dimensional arrays of 'bytes'
vectors	one dimensional arrays of any object
arrays	any number of dimensions
properties	hash coded 'arrays', any object can be a subscript
pairs	as in LISP
procedures	structures containing machine code
externals	procedures written in some language other that POP-11 such as FORTRAN or assembly language
closures	a combination of a procedure and an environment
processes	see Section 2.7
devices	descriptors of external objects like disc files

One additional vitally important data structure is the 'word' which is modelled on the LISP atom. In essense, a word is a string with additional properties that has been entered in the P0PL06 dictionary. There is only copy of any word (unlike strings where there may be several strings containing the same letters). Words in POP-11 are also

variables; amongst the properties that words have are its value and its syntactic type.

2.3.2) USER DEFINED STRUCTURES

The user may define new types of data structure if the predefined types of data structure are inappropriate. One way of declaring a new type of data structure is to use a •recordclass[1] statement, for example:

```
recordclass person name age sex;
```

This tells POP-11 that PERSON is a new type of data structure which has three components a NAME, an AGE and a SEX. Instances of data structures are not declared in POP-11 but created using a •constructor[1] procedure, for example:

```
conspersonC'steve", 33, "male") -> p;
```

This creates a new PERSON structure whose NAME is STEVE, whose AGE is 33 and whose SEX is MALE. The new structure is stored in an area of memory called the "heap[1] and a pointer to (ie the address of) the new structure is assigned to the variable P. We can use the new structure thus:

```
age(p) =>
** 33
age(p) + 1 -> age(p);
age(p) =>

** 34
```

Notice that structures are accessed using procedures; the procedure assigned to the variable AGE accesses the second component of a PERSON structure. This convention makes it particularly easy to make a program independent of the form in which data is stored since there is no syntactic distinction between accessing a field of a structure and applying any other procedure to the structure. We might, for example, decide to alter our representation of a PERSON thus:

```
recordclass person name yearofbirth sex;
```

If we have a global variable containing the current year we can now write an ordinary procedure called AGE, thus:

```
define age(x) -> result;
    currentyear - yearofbirth(x) -> result
enddefine;
```

If the value of CURRENTYEAR changes then this will be reflected in the result of future calls of AGE. For example:

```
consperson("steve", 1949, "male") -> p;
1982 -> currentyear;
age(p) =>
** 33
currentyear + 1 -> currentyear;
age(p) =>
** 34
```

We may even define an 'updater' for AGE that will allow us to 'assign' to the AGE of a PERSON even though there is no AGE field as such. With this new representation, an assignment to AGE should be translated to an assignment to the YEAROFBIRTH. The appropriate definition is:

```
define updaterof age(newage, x);
    currentyear - newage -> yearofbirth(x);
enddefine;
```

We can use this new procedure thus:

```
age(p) =>
** 34
37 -> age(p);
yearofbirth(p) =>
** 1946
```

2.3.3) MEMORY MANAGEMENT

POPLOG includes a 'garbage collector'. This manages the heap where all structures created by the user are stored. When a temporary data structure is no longer accessible by the user (for example, no user variables 'points' to it) then the garbage collector reclaims the space occupied by the inaccessible structure. It is not possible for the user to find the actual address of a structure and, in fact, the garbage collector will rearrange the heap periodically to coalesce reclaimed

2.4.1) STRUCTURE EXPRESSIONS

The most commonly used data structures in POP-11 are lists and words. These are modelled on the data structures provided in LISP. A list is a linked collection of primitive data structures called PAIRs. PAIRs have two components, a HD and a TL. A three element list of the words STEVE/ IS and HAPPY could be created by the following command:

```
conspair("steve"[11], conspairC'is", conspair("happy", nil))) -> x;
```

The HD of the first PAIR is a pointer to the first element of the list; the TL of the first pair is a pointer to a second pair whose HD is a pointer to the second element and whose TL is a pointer to a third pair whose HD is a pointer to the third element of the list and whose TL is a distinguished object, thus:

```
hd(x) =>
** steve
hd(tl(x)) =>
** is
hd(tl(tl(x))) =>
** happy
```

Although there is a default way of printing data structures, the user is allowed to override this default and specify precisely how structures of any given type are to be printed. POP-11 already has a special way of printing PAIRs, thus:

```
x =>
** Csteve is happyD
```

Additionally, square brackets "C" and "3" have been defined as syntax words in POP-11 so that users may create lists without explicit use of CONSPAIR. (see section 2.8 for a discussion of how new syntax words are created.) The above list could have been created more simply with the statement:

```
Csteve is happyD -> x;
```

Expressions such as these are termed 'structure expressions'. They evaluate to data structures. In the example above, the structure could be created at 'compile time' since all its constituents are known then. This is not usually the case; usually data structures are created out of components that are being held in variables and are not known at compile time. For example, if the variable ADJ contains some word then the user can create a three element list of the words STEVE, IS and the value of ADJ thus:

```
[steve is ^adj]
```

```
**************************************************
*                NOTE TO PUBLISHER               *
* The symbol ^ does not come out very well on our *
* printer. It should be an upwards pointing arrow *
* and not a circumflex                            *
**************************************************
```

The up-arrow, "^", tells POP-11 that it is the value of ADJ that is wanted and not the word ADJ itself. If the value of a variable is a list, the double up-arrow, "^^" tells POP-11 to insert <u>all</u> the elements of that list as elements of the new list, for example:

```
[red orange yellow] -> x;
[blue ^^x] =>
** [blue red orange yellow]
[^^x blue] =>
** [red orange yellow blue]
[^^x are all colours] =>
** [red orange yellow are all colours]
```

Since most POP-11 users use structure expressions, there is no need for them to know about CONSPAIR. This eliminates many minor programming errors. Structure expressions can be used to create all types of datastructure.

To complement structure expressions, POP-11 provides a 'pattern assignment' statement for decomposing lists without the explicit use of HD and TL. A typical pattern assignment is:

```
x --> [?a ?b ?c];
```

This statement takes the list which is the value of the variable X and assigns the first element of the list to A, the second to B and the third to C. Unless the list has exactly three elements, an error message is generated. A more complicated example of a pattern assignment is:

```
x --> [?a ??b ?c];
```

The double question mark indicates that B is to be given a list as value. The first element of X is assigned to A, the last to C and a list of the intervening elements to B. If X was initially [STEVE IS VERY HAPPY], then A will be set to STEVE, B to [IS VERY] and C to HAPPY. Using structure expressions and pattern assignments, it is easy to write a procedure to produce a copy of a list with some given item deleted, thus:

```
define delete(item, list) -> result;
    vars x, y;
    list --> [??x ^item ??y];
    [^^x ^^y] -> result
enddefine;
delete("very", [steve is very happy]) =>
** [steve is happy]
```

2.4.3) A SIMPLE DATABASE PACKAGE

Structure expressions and pattern assignments are used to provide a simple 'database' facility in POP-11. The variable DATABASE is a list of structures. The procedure ADD inserts a structure in this list, REMOVE removes a structure and LOOKUP locates a structure, thus:

```
define add(item);
    [^item ^^database] -> database
enddefine;
```

```
define remove(item);
    vars x, y;
    database --> [??x ^item ??y];
    [^^x ^^y] -> database
enddefine;

define lookup(item);
    vars x, y;
    database --> [??x ^item ??y]
enddefine;
```

LOOKUP will usually be given a pattern as argument; the effect of using LOOKUP is to assign to the variable in that pattern, thus:

```
[] -> database;
add([steve is happy]);
add([aaron is busy]);
database =>
** [[aaron is busy] [steve is happy]]
lookup([?p is happy]);
p =>
** steve
```

Finally, FOREVERY constructions allow programs to iterate over all instances of a set of patterns in a given list, for example:

```
forevery [[?x is a bachelor] [?x is rich]] in database do
    add([mary wants [^x loves mary]])
endforevery;
```

The definition of FOREVERY is stored in the auto-loadable library (see section 3.4). The mechanism for defining new syntax words is discussed in section 2.8.3; the definition uses co-routining (see section 2.7).

2.5) A SIMPLE PARSER

Pattern assignments may contain restrictions; these are elements in the pattern which restrict the values which can be assigned to a variable. For example, the statement:

```
[i saw 3 ships] --> [??x ?y:isinteger ??z]
```

is an example of a restricted pattern assignment. The variable Y is constrained to accept as value only an integer. Thus the result of the above assignment would be to give X the value CI SAWD, Y the value 3 and Z the value [SHIPS].

If we have written procedures called NOUNPHRASE and VERBPHRASE which, respectively, return TRUE if given a noun phrase and a verb phrase then the following statement will succeed (ie not cause an error) only if the value of LIST is a 'sentence[1]:

```
list -> C??x:nounphrase ??y:verbphraseD
```

The variables X and Y will be set, respectively, to the portion of LIST which is the noun phrase and the portion which is the verb phrase. Should the sub-procedures NOUNPHRASE and VERBPHRASE return anything other than TRUE or FALSE (say parse trees) then X and Y will be set to the results of those procedures.

So that we can tell whether LIST is a sentence (it may not be) we use the procedure MATCHES This performs a pattern assignment and returns TRUE unless the assignment would cause an error in which case it returns FALSE. The following is a procedure which recognizes a 'sentence[1] and returns a parse tree for it (or else returns FALSE if its argument is not a 'sentence[1]):

```
define sentence(list) -> result;
    vars X, Y;
    if list matches C??X:nounphrase ??Y:verbphraseD then
        [sentence "X "YD -> result
    else
        false -> result
    endif
enddefine;
```

The definition of NOUNPHRASE is equally straightforward:

```
define nounphrase(list) -> result;
    vars X, Y;
    if list matches [?X:determiner ??Y:adjnoun] then
        [nounphrase ^X ^Y] -> result
    else
        false -> result
    endif
enddefine;
```

This says that a noun phrase is a DETERMINER followed by an ADJNOUN. An ADJNOUN is either simply a NOUN or else an ADJECTIVE followed by an ADJNOUN:

```
define adjnoun(list) -> result;
    vars X, Y;
    if list matches [?X:noun] then
        X -> result
    elseif list matches [?X:adjective ??Y:adjnoun] then
        [adjnoun ^X ^Y] -> result
    else
        false -> result
    endif
enddefine;
```

A verb phrase is defined as a verb followed by a nounphrase, thus:

```
define verbphrase(list) -> result;
    vars X, Y;
    if list matches [?X:verb ??Y:nounphrase] then
        [verbphrase ^X ^Y] -> result
    else
        false -> result
    endif
enddefine;
```

To complete our parser we need procedures to recognize the lexical categories, thus:

```
define determiner(word) -> result;
    if member(word, [the each]) then
        [determiner ^word] -> result
    else
        false -> result
    endif
enddefine;
```

```
define noun(word) -> result;
    if member(word, [cardinal shoe]) then
        [noun ^word] -> result
    else
        false -> result
    endif
enddefine;

define adjective(word) -> result;
    if member(word, [purple]) then
        [adjective ^word] -> result
    else
        false -> result
    endif
enddefine;

define verb(word) -> result;
    if member(word, [discarded]) then
        [verb ^word] -> result
    else
        false -> result
    endif
enddefine;
```

We can now test our parser, thus:

```
sentence([the cardinal discarded each purple shoe]) =>
** [sentence
       [nounphrase [determiner the] [noun cardinal]]
       [verbphrase [verb discarded]
                   [nounphrase [determiner each]
                               [adjnoun [adjective purple]
                                        [noun shoe]]]]
```

The form of this parser is so straightforward that is easy to write a program that will generate the above program given a description of the grammar. POP-11 contains such a program in the library (see section 3.4). The programs generated by the library package are less simplistic than that shown above which is not recommended as a way of writing parsers; the example is shown only to illustrate structure expressions and pattern assignments.

The grammar and example shown here are similar to those in the chapter on PROLOG in this book. This will enable the reader to compare a POP-11

2.6) PROCEDURE CLOSURES

POP-11 provides a number of mechanisms for dynamically creating new procedures as a program is running. In this section, which can be skipped on first reading, we illustrate the way one such technique, called partial application, is used for input and output in POP-11 and also how it could be used to implement PROLOG.

2,6.1) CLOSURES IN INPUT AND OUTPUT

Partial application is a technique whereby a procedure and some arguments for that procedure can be 'frozen' together to create a new procedure (or 'closure' as it sometimes called). A closure needs fewer arguments than the original procedure; if all the arguments were frozen in at the time the closure was created then it will need need no additional arguments at all. Partial application is used to provide elegant input/output mechanisms in POP-11 arid this application is a good introduction to closures. POP-11 provides a number of primitive procedures for accessing disc files. With some simplifications, two of these are:

* SYSOPEN which takes as argument the name of a disc file and returns a 'device descriptor' for reading from that file.
* SYSREAD which takes as argument a device descriptor, created by SYSOPEN, and returns the 'next' character from the disc file.

Thus:

```
sysopen('foobaz') -> d
```

The variable D will now hold a device descriptor for the file called FOOBAZ. To read the first character from this file we would do:

```
sysread(d) =>
```

second character and so on. If we 'partiaUy apply[1] SYSREAD to the device descriptor D, thus:

```
sysread(Xd%) -> p;
```

then the variable P will hold a closure. Notice that partial application is denoted by •decorated parentheses[1], •(%• and • %) ' . We can now simply apply P to read succesive characters from the file, thus:

```
p<> =>
```

SYSREAD and SYSOPEN are packaged up into a procedure called DISCIN which takes as argument a file name and returns a procedure which reads from that file, thus:

```
define discinCfilename) -> result;
    sysread(%sysopen(filename>%) -> result
enddefine;
discinC'foobaz[1]) -> p;
p0 =>
```

2.6.2) CLOSURES AND PROLOG

In this section a highly simplified account of the way PROLOG is implemented in POPLOG is given; CMELLISH 83] is a fuller account.

2.6.2.1) CONTINUATION PROGRAMMING

PROLOG is implemented using a technique called Continuation passing[1]. In this technique, procedures are given an additional argument, called a continuation. This continuation (which is a procedure closure) describes whatever computation remains to be performed once the called procedure has finished its computation. In conventional programming, the continuation is represented implicitly by the 'return address[1] and code in the calling procedure. Suppose, for example that we have a procedure, called PROG, that has just two steps: calling the subprocedure F00 and then when that has finished execution calling the subprocedure BAZ,

```
define prog();
    foo();
    baz();
enddefine;
```

Were this procedure to be re-written using explicit continuations, then BAZ would be passed as an extra argument to FOO since BAZ is the continuation for FOO. Actually, it is not quite that simple since PROG itself would also have a continuation and this must be passed to BAZ as its continuation, thus:

```
define prog(continuation);
    foo(baz(%continuation%))
enddefine;
```

Thus, if we invoke PROG we must give it explicit instructions, CONTINUATION, as to what is to be done when it has finished. PROG invoked FOO, giving FOO as its continuation the procedure BAZ which has been partially applied to the original continuation since that is what is to be done when BAZ (now invoked by FOO as its continuation) has finished its task.

This apparently round about way of programming has an enormous advantage - since procedures have explicit continuations there is no need for them to 'return' to their invoker. Conventionally, sub-procedures returning to their invokers means:

I have finished - continue with the computation

With explicit continuations we can assign a different meaning to a sub-procedure returning to its invoker, say:

Sorry - I wasn't able to do what you wanted me to do

PROG accomplishs its task by first doing FOO and then doing BAZ. The power of continuation programming is made clear if we define a new procedure NEWPROG, thus:

Try doing FOO but if that doesn't work then try doing BAZ

This is represented thus:

```
define newprog(continuation);
    foo(continuation);
    baz(continuation);
enddefine;
```

If we now invoke NEWPROG (with a continuation) then it first calls FOO (giving it the same continuation as itself). If FOO is succesful then it will invoke the continuation. If not then it will return to NEWPROG which then tries BAZ. If BAZ too fails (by returning) then NEWPROG itself fails by returning to its invoker.

2.6.2.2) PROLOG PREDICATES AND CONTINUATIONS

Consider the following PROLOG procedure:

```
happy(X) :- healthy(X), wise(X).
```

This says that X is HAPPY if X is HEALTHY and WISE. If this is the only definition of HAPPY then we may translate this to the following POP-11 procedure:

```
define happy(x, continuation);
    healthy(x, wise(%x, continuation%))
enddefine;
```

A call of this procedure can be interpreted as meaning:

```
Check that X is happy and if so do the CONTINUATION
```

This is accomplished by passing X to HEALTHY but giving HEALTHY a continuation which then passed X across to WISE. Let us suppose that someone is HEALTHY if they either JOG or else EAT CABBAGE, ie:

```
healthy(X) :- jogs(X).
healthy(X) :- eats(X, cabbage).
```

This can be translated as:

```
define healthy(x, continuation);
    jogs(x, continuation);
    eats(x, "cabbage", continuation);
enddefine;
```

Finally, let us assume that we know that CHRIS and JON both JOG, thus:

```
jogs(chris).
jogs(jon).
```

We can represent this as a POP-11 procedure thus:

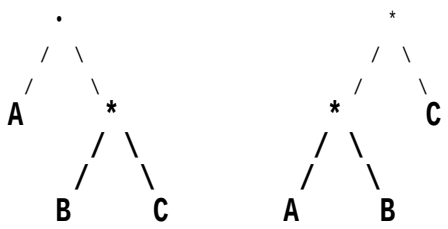
```
define jogs(x, continuation);
    if x = "chris" then continuation() endif;
    if x = "jon" then continuation() endif;
enddefine;
```

The translation of JOGS is too simplistic. It does not cater for the case where X is unknown and we wish to find someone who JOGS. This is dealt with in the actual PROLOG sub-system of POPLOG by representing unknowns by data structures whose contents are initially UNDEF, a unique word. Instead of simply comparing X with the word CHRIS, JOGS instead trys to 'unify' the data structure with the word CHRIS. (Also, PROLOG procedures are translated directly into virtual machine code and not into POP-11 procedures.) A full account of this process is inappropriate in a paper of this length.

2.7) CO-ROUTINES AND GENERATORS

POPLOG allows programs to be be written as a number of cooperating processes. Only one process may be active at any given moment and it must explicitly relinquish control of the processor before any other process may run. This restricted multi-processing facility is called co-routining. To illustrate the use of co-routines, a program to solve the 'same fringe' problem is presented.

The same fringe problem consists of writing some program to take two arbitrarily shaped trees and to determine if the 'fringe' of the two trees is identical. This is true for the following two trees:



What makes this problem difficult is that while one can write a recursive program to traverse one tree it is very hard to write a program to traverse two differently shaped trees. The solution is to have one process per tree. These processes would perform a simple recursive traversal of their single tree pausing after finding each element of the fringe.

We can easily represent trees like those above with lists, thus:

CA CB *Cll* CCA BD CD

We represent a node of the tree as either a word (meaning we are at a tip of the tree and the word is on the fringe) or else as a list of subtrees. For simplicity, we say that nodes are either tips or else have exactly two sub-trees. A procedure to traverse such a tree is:

```
define traverse(tree);
    if islist(tree) then
        traverse(treed));
        traverse(tree(2>);
    else
        suspend(tree, 1)
    endif
enddefine;
```

The notation 'TREEd)[1] means the first subtree. POP-11 allows lists (and many other structures) to be accessed as if they were one dimensional arrays. Thus, if the TREE given to TRAVERSE is a list, then TRAVERSE calls itself recursively - first on the first subtree and then on the second subtree. If the TREE given to TRAVERSE is not a list then it must be a tip, ie a word on the fringe. In this case TRAVERSE calls the

invoked it. To create a process, we use the procedure CONSPROC. Basically, CONSPROC takes as argument a procedure and returns a process which when invoked with RUNPROC will call the given procedure. CONSPROC must also be given the arguments that will be needed by the procedure and a count of the number of arguments. The following procedure, FRINGEPROCESS, takes as argument a tree and returns as result a process which will traverse the given tree:

```
define fringeprocess(tree);
    consproc(tree, 1, procedure x; traverse(x); termin endprocedure)
enddefine;
```

(The notation PROCEDURE ... ENDPROCEDURE specifies an 'anonymous' procedure; these are called 'lambda expressions' in LISP). We can apply the above procedure to some particular tree, thus:

```
fringeprocess([[a b] [c [d e]]]) -> x;
```

The value of X is now a process which can be run with RUNPROC. RUNPROC also needs to know whether any data items are to be passed to the process; in this case there are none. RUNPROC will start the process which will enter TRAVERSE and continue execution until it reaches a call of SUSPEND. The parameter to SUSPEND will then be passed back to the calling process as the result of RUNPROC, thus:

```
runproc(0, x) =>
** a
```

We have now got the first element of the fringe of the tree given to FRINGEPROCESS. The process held in the variable X is now suspended half way through the execution of TRAVERSE. If we invoke RUNPROC repeatedly we will get the subsequent elements of the fringe, thus:

```
runproc(0, x) =>
** b
runproc(0, x) =>
** c
```

Eventually, the process in X will terminate and produce TERMIN as its result so that we know we have reached the end of the fringe. (TERMIN is a unique object used by POP-11 programs to signify that a •stream[1] has terminated.)

It is a simple matter to write a procedure which takes two trees, create processes for each of them and then compare the 'results[1] given by the two processes, thus:

```
define samefringe(t1, t2);
    vars x1, x2, p1, p2;
    fringeprocess(ti) -> p1;
    fringeprocess(t2) -> p2;
    repeat forever
        runproc(0, p1) -> x1;
        runproc(0, p2) -> x2;
        if x1 » termin and x2 = termin then return(true) endif;
        unless x1 = x2 then return(false) endunless;
    endrepeat;
enddefine;
```

RETURN is a key word to force immediate termination of a procedure call. This is used as soon as two fringe elements are different. If the trees have identical fringes then the two sub-processes will eventually return TERMIN simultaneously and the REPEAT loop will be broken. If the fringes are different then the POPLOG garbage collector (see section 2.3) will recover the space occupied by the now unwanted processes.

The reader will note the similarity between testing the •outputs[1] of two processes and testing two lists for equality. POP-11 provides a mechanism by which a process can be made to appear to be a list of its outputs. The following expression:

```
pdtolist(runproc(%0, fringeprocess(tree)%))
```

evaluates to a 'dynamic list'. Dynamic lists are data structures that appear to be simple lists. The normal list processing procedures like HD and TL will work on them. Actually, the elements of a dynamic list are evaluated only when they are required. If we were to access the third element of the dynamic list created by the above expression then POP-11 would arrange for the process to be run three times without our being aware of it.

2.8) HOW POPLOG IS IMPLEMENTED

The POPLOG system is implemented almost entirely in POPLOG. There are three reasons for this:

* It gives the user almost unlimited power to tailor the system to his own requirements.

* It is the easiest way of doing it - POPLOG is a powerful programming tool and it is very convenient to be able to make use of it in its own construction.

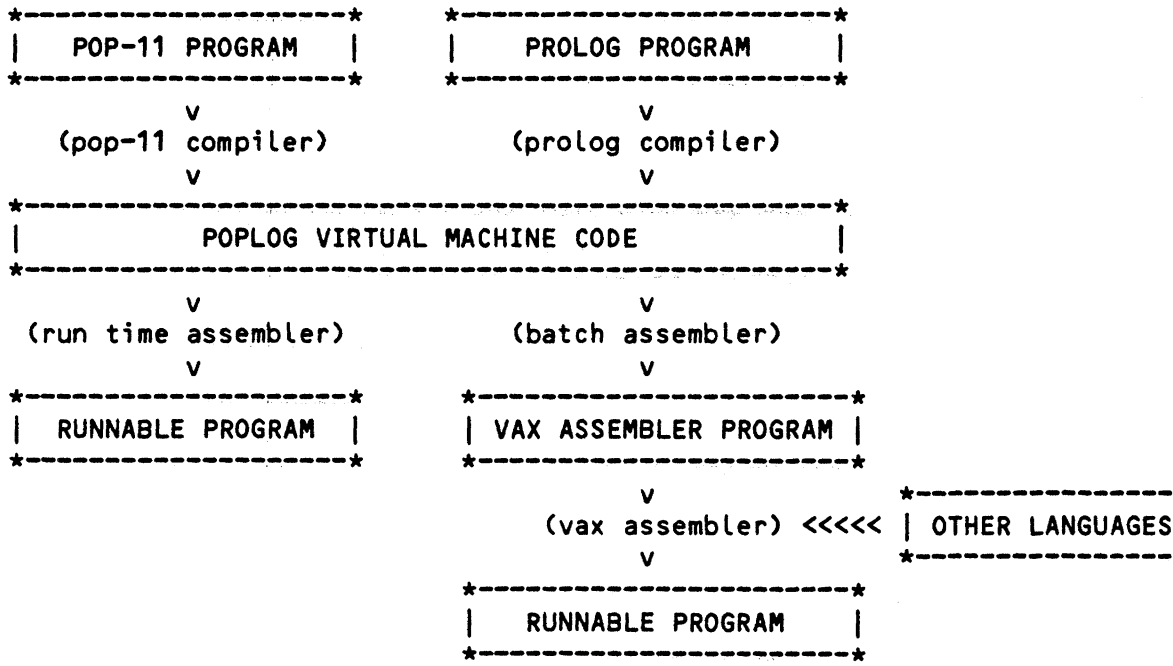
* It makes the POPLOG system inherently portable. At a time when there are so many exciting developments in hardware design we wanted the POPLOG system to be relatively independent of any actual computer. About three man months work should be sufficient to move the system to any 32-bit computer. Since the essential programming tools, such as a screen editor, are already in POPLOG the user of POPLOG is also relatively independent of the operating system.

2.8.1) THE OVERALL STRUCTURE

The POPLOG system can be visualized as an inverted pryamid resting upon the POPLOG virtual machine. The POP-11 and PROLOG compilers translate programs into instructions for this machine. Virtually all of the POPLOG system, including the screen editor and the compilers themselves, is written in POP-11 and so can be translated into POPLOG virtual machine code.

POPLOG virtual machine code is itself the apex of a pyramid and can be translated into the machine code of the actual machine on which POPLOG is running. For user programs, POPLOG contains an in-core assembler which translates POPLOG virtual machine code directly into running programs in memory.

To produce the running code for the POPLOG system itself, a different mechanism must, of course, be used. This is accomplished by having a 'boot strap' version of the POPLOG system which simply makes a list of POPLOG virtual machine code instead of translating it into running programs in memory. A stand-alone program (written in POP-11, naturally) translates this list into the assembly language provided by the host operating system. Once in this form, it is easy to 'link in' programs written in languages other than POP and then use the standard assembler to produce a running program. This arrangement can be illustrated in diagrammatic form, thus:



To transport the POPLOG system to a new computer only the two assemblers need be re-written for the new computer. The batch assembler (used for compiling the POPLOG system itself) makes many machine dependent 'optimizations'. Most importantly, it replaces code calls of certain key procedures (such as addition) by suitable inline code. As the batch assembler is table driven it is easy to modify it for a new type of computer.

2.8.2) THE VIRTUAL MACHINE

An understanding of the POPLOG virtual machine is of great help in understanding POP-11. This machine is conceptually very simple and the mapping between POP-11 instructions and virtual machine instructions is also simple. POPLOG is based on a stack oriented machine. Expressions in POP-11 are translated into instructions for this machine. For example, the assignment statement:

```
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 stack is used to save the values of local variables during procedure calls. For example, the procedure:

```
define double(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
restore x     - Restore the value of X from the system stack
```

up into a 'procedure record' which is then assigned to the variable DOUBLE.

Understanding this simple 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 lead to obscure programs).

Notice, too, that POP-11 has a simple way of assigning scope to variables. 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 on procedure exit they are restored. That is, POP-11 is 'dynamically bound' with 'shallow binding'. This has the advantage that procedures are not associated with any particular environment; a procedure is simply a collection of instructions that can be executed any time and in any environment.

Also, it can be seen that it is necessary for the addition procedure, say, to check that it has been given two numbers. (Since the addition procedure can be invoked at any time whatever the state of the stack).

In POPLOG it is not variables that are typed but objects. A POP-11 procedure cannot INSIST that it be given only, say, integers; if this is crucial to the procedure's operation then it must check for itself. It is not generally recognized that typed variables greatly restrict the type of procedure that can be written. For example, the SORT procedure in the POP-11 library takes two arguments: a list of any type of object and a boolean procedure embodying the ordering criteria. A procedure

such as this could not be written in a strongly typed language, such as PASCAL, because the precise type of the procedure is not known at compile time (it is known that it 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 simple representation for procedures, combined with the ability to call the compiler recursively, gives the user great flexibility. Crucially, the user can write programs which CONSTRUCT procedures by assembling the text for the procedure and then compiling that text. The ability to do this is often of great importance in AI programs where a program may wish to extend itself on the basis of the data it is given.

2.8.3) DEFINING SYNTAX WORDS

The P0PL06 virtual machine is available to the user as a set of procedures for 'planting' virtual machine code instructions. This means that the POP-11 compiler itself can be written as a set of POP-11 procedures. Indeed, some infrequently used syntactic constructions are defined by quite ordinary POP-11 procedures stored in the 'auto loadable[9] library (see section 3.4).

Suppose, for example, that POP-11 did not already have an UNTIL loop. The form of such a loop is:

```
UNTIL condition DO actions ENDUNTIL
```

The POPLOG virtual machine instructions for such a loop are:

```
label    L1
<code for the condition>
ifso     L2
<code for the actions>
goto     L1
label    L2
```

element from the top of the user stack and if it is not [illegible] control to the given label. .

The user could add UNTIL loops to POP-11 by defining the following procedure:

```
define syntax until;
    vars L1, L2;
    gensym() -> L1;
    gensym() -> L2;
    plantlabel(L1);
    compileto("do");
    plantifso(L2);
    compileto("enduntil");
    plantgoto(L1);
    plantlabel(L2);
enddefine;
```

(This definition has been simplified for clarity.) The procedure COMPILETO reads in and compiles text up to the given closing keyword, in this case ENDUNTIL. The various PLANT procedures create POPLOG virtual machine code. The procedure GENSYM creates unique words for use as labels.

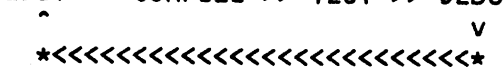
Should one UNTIL loop be nested inside another then the above definition of UNTIL will be invoked recursively by COMPILETO.

---

3) THE ENVIRONMENT

As we have shown, POP-11 is a powerful and flexible programming language. Programming languages, however, are only one of the tools the programmer needs. Programmers typically spend much of their time interacting with ancillary programs like editors and debugging packages. Perhaps the most crucial advantage POPLOG offers the programmer is that all the necessary tools are integrated into a single 'programming environment'. Most significantly, a powerful screen editor, called VED,

is intimately linked to the compilers and the run time system. This means that the familiar cycle in which so much programmer time is spent:

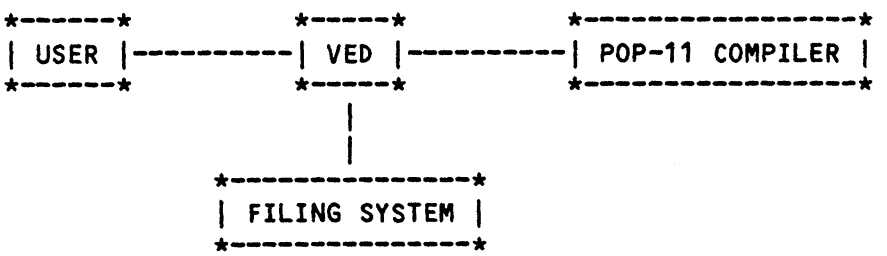


can be appreciably speeded up. It is the BASIC programming environment that has made that language so useful, not BASIC itself. In POPLOG we have married powerful programming languages to a superb programming environment

3.1) THE TEXT EDITOR

Since programmers spend more time modifying programs than running them, it is important they have good editing facilities. POPLOG accomplishes this by building the editor into the run time system so allowing the user to edit and re-compile portions of his program as it is being developed and tested.

POP-11 is primarily intended for interactive use from a terminal. In earlier implementations of POP-11, the compiler read a stream of definitions and imperatives from the user's terminal and wrote back any output. In this new implementation for the VAX a text editor, called VED, is interposed between the user and the POP-11 compiler, thus:



Notice that the user is always communicating with VED, the text editor. (This section is confined to how POP-11 fits into the POPLOG environment. The PROLOG compiler occupies a similar place to the POP-11 compiler in this scheme.)

The user's VDU screen continuously displays a portion of some selected file or files. These files may belong either to the user or be files •belonging' to POPLOG (such as documentation or tutorial files). When user presses the •DOIT' button part of the •current' file is sent to the POP-11 compiler. The POP-11 compiles the fragment of text sent to it and sends back any output to VED which splices the output into the current file and hence displays the output on the user's VDU screen. Since the output is stored in an edit file it is easy to review any output that has scrolled off the top of the VDU screen.

This may sound *jery* complicated, but in practice it is very simple since the scope of the DOIT button will default to any fresh text typed in since the DOIT button was last pressed. A simple interaction with POPLOG will consist of the user typing in a command, pressing the DOIT button and observing the output; this cycle is then repeated. If a definition needs to be modified two or three keystrokes after editing suffice to have the procedure re-compiled and incorporated into the existing compiled program.

The editing procedures can be invoked directly by POP-11 (and PROLOG) programs. A program wanting to create a disc file can therefore use the full power of the editor to aid it in this task. Moreover, any program wanting simple graphical output can accomplish this using the editor.

disc file; the procedure DRAW invokes editing procedures to alter the disc file and VED ensures that the user's terminal is updated as the file is updated.

* A demonstration problem solving package, called SOLVER, demonstrates means-end analysis and forward chaining heuristic search. The search space is shown as a dynamically changing tree. It is intended that a similar facility will be incorporated into the PROLOG system to provide a uniquely powerful debugging tool.

POPLOG makes full use of the VAX virtual memory system to minimize actual disc IO.

Without a demonstration it is difficult to know how an editor 'feels'. In a paper one is reduced to giving a written description which is an inadequate substitute for practical experience. However, VED is sufficiently convenient to have attracted a number of users who have no interest in programming and simply want to use it as a word-processing system.

The task of describing VED is complicated by the fact that it can be customized by the individual user. In essence, VED provides an extensible set of 'editing procedures'. Any particular procedure can be 'attached' to one of the keys on the user's terminal. Normally, for example, the key marked 'A' will, when pressed, invoke a procedure to insert a letter A into the current file. The terminals in use at Sussex University have over thirty 'function' keys in addition to the normal QWERTY keyboard; these additional keys are allocated to commonly used procedures such as:

MOVE-CURSOR-TO-RIGHT-OF-NEXT-WORD

Escape sequences are used to get access to procedures not allocated to a key (such as the procedure to select some new file for editing).

Since the editing procedures are written in POP-11, the user can write new editing procedures and, if desired, attach them to keys or keystroke sequences.

In summary, POPLOG incorporates a powerful text editing facility that can be used both by the programmer to modify his program and also by the program itself to simplify output. The editor is written entirely in POP-11 and can be extended or otherwise modified by the programmer.

3.2) DEBUGGING TOOLS

The design of POP-11 makes the provision of explicit debugging tools less necessary than with other programming languages. The principal reason for this is that the compiler can be invoked during breakpoints. This allows the user to give any POP-11 command - for example to examine or change variables, or even to edit and recompile procedures. Breakpoints occur whenever there is an error, whenever code execution reaches a declared breakpoint (set, perhaps, by editing a procedure definition to include a call of the compiler) or when the user interrupts a running program.

Since POP-11 procedures can be manipulated by POP-11 programs, debugging tools can be written in POP-11 itself. (This feature of POP-11 is shared by LISP and other AI languages). For example, in the POPLOG library, there is a short program (about 50 lines) that will add 'trace' printing instructions to specified procedures.

Considering the importance of debugging it is surprising that so few languages make proper allowance for it. It is not uncommon for language [illegible] of the available facilities. Frequently there

To summarize, the basic design of POP-11 makes the provision of explicit debugging tools less necessary than with other non-AI languages. The language was designed with debugging in mind.

3.3) THE DOCUMENTATION SYSTEM

One big advantage of having a screen editor built into the POPLOG system is that it greatly simplifies providing documentation for the on-line user. A simple editor command, such as:

HELP FOR

tells the editor that the user want to look at the 'help file' for 'for' (one of the iterative constructions in POP-11). The editor assigns a 'window' on the user's VDU screen and within that window displays the wanted documentation. The documentation is visible as the user ponders over the file he is editing which is also visible. If the information given is insufficient, the user can give the command:

TEACH FOR

This selects the 'teach file' for 'for'. Teach files are generally much longer and are tutorial introductions to the use of POPLOG.

A third level of documentation, 'reference files', are provided for those wanting more precise details of how the system works. These are primarily intended for experienced programmers.

POPLOG was developed with the needs of teaching as well as research firmly in mind. A collection of many dozens of 'teach files' exist. They explain not only aspects of POPLOG but also aspects of AI in general. Undergraduates at Sussex University can spend up two years studying AI and much of their reading will be of teach files. Typically, a course tutor can tell a student to go and read a particular teach file as a

week's work. Teach files usually include exercises and assignments for students. Some are designed for on-line study while others can be printed and read away from the terminal.

3.4) THE LIBRARY

An essential component of POPLOG is the associated program library. POPLOG has an 'auto-loading' mechanism that causes library files to be automatically compiled and included into any user program that references them. As part of the undergraduate teaching program a number of simplified AI programs have been written and these are available for incorporation into user programs. These include a suite of programs for operating on line-drawings, an ELIZA-like program, a structure database and a parser generator program that writes parsers in POP-11 given a context free grammar.

All library programs may be perused with the editor and the SHOWLIB command. In this way, many users have developed their skill and understanding.

---

4) CONCLUSIONS

The POPLOG system for the VAX computer, although originally developed for AI research, has features that make it useful in more general applications. The system provides an excellent environment for the programmer. He need not ever leave this environment since it includes needed utilities such as text editors, documentation and debugging tools. The compiler itself is not intrusive, generally doing the right

compiler, screen editor etc are all written in POP-11). The system runs on an unmodified VMS operating system and will be made available on other machines and operating systems.

BIBLIOGRAPHY

CBURSTALL 19713
Programming in POP-2
Burstall, Collins and Popplestone
Edinburgh University Press, 1971

This book describes the earliest version of POP-2. In addition to a primer it also has many demonstration programs which, unfortunately, *are* badly written with excessive use of GOTOs etc.

CCLOCKSIN 19813
Programming in PROLOG
William Clocksin and Chris Mellish
Springer-Verlag, 1981

This book a reference manual for, and introduction to, PROLOG.

CGREEN 19743
An Easily-implemented Language for Computer Control of Complex Experiments
T. Green and D Guest
International Journal of Man-Machine Studies (1974) 6

This paper describes GLUE, a dialect of POP-2, used for controlling Experimental Psychology experiments.

CHEWITT 19763
Viewing Control Structures as Patterns of Passing Messages
Carl Hewitt
Artficial Intelligence memo 410
Massachusetts Institute of Technology, 1976

Reading the paper, which introduces the ACTOR model of computation, wil help the reader understand more about the continuation passing style o' programming used in Section 2.6.2.

CHOLLOWAY 19803
The SCHEME-79 Chip
Jack Holloway, Guy Lewis, Gerald Sussman and Alan Bell
Artficial Intelligence Memo 559
Massachusetts Institute of Technology, 1980

This describes a hardware implementation of the SCHEME interpreter. Se CSUSSMAN 19753 for an account of this language.

[illegible] 1980]
A Real Time Garbage Collector
that can Recover Temporary Storage Quickly
Henry Lieberman and Carl Hewitt
Artficial Intelligence memo 569
Massachusetts Institute of Technology, April 1980

One of the serious disadvantage of the P0PL06 programming environment is that periodically it takes time out to reorganize its memory - a process called garbage collection. This paper describes one method of overcoming this problem. As P0PL06 is not used for real-time work we have not attempted to use any of the ideas in this paper. With the decreasing cost of computer memory, the problem may go away as we might expect computers to have such massive amounts of memory that either garbage collection will be unnecessary or else can be postponed till a non critical time (like overnight).

CMEAD 19793
Introduction to VLSI Systems
Carver Mead and Lynn Conway
Addison Wesley, 1979

It is becoming increasingly important for programming system designers to consider whether their design could exploit special purpose hardware. This was not an issue in the design of P0PL06 since we wished to use standard, commercially available hardware; the interested reader, however, may wish to consult this introduction to VLSI design.

CMELLISH 19833
Implementing PROLOG in P0P-11
Chris Mellish and Steven Hardy
Cognitive Studies Memo
University of Sussex

This paper (in preparation) describes the way in which PROLOG has been implemented in the POPLOG system and outlines the advantages of putting a PROLOG compiler into a good programming environment.

CPAPERT 19803
Mindstorms: ChiIdren, Computers and Powerful Ideas
Seymour Papert
Harvester Press, 1980

An entertaining and enthusiastic account of the benefits of using computers in children's education.

SUSSMAN 19753
SCHEME: An Interpreter for Extended Lambda Calculus
Gerald Jay Sussman and Guy Lewis Steele
Artficial Intelligence Memo 349
Massachusetts Institute of Technology, 1975

A *very* worthwhile paper describing a collection of issues about implementing interpreters for the Lambda Calculus. In particular, there is a useful discussion of the relative merits of 'dynamic[1] and 'static[1] binding.

[WEINREB 1979]
LISP Machine Manual
Daniel Weinreb and David Moon
Massachusetts Institute of Technology

The reference manual for the MIT LISP machine. This is only for the very dedicated as it explains very little of the philosophy behind using single user computers.