12

POP-1: AN ON-LINE LANGUAGE

R. J. POPPLESTONE
EXPERIMENTAL PROGRAMMING UNIT
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

POP-1 is an on-line computer language, whereas most of the well-known languages (FORTRAN, MAD, ALGOL, LISP, etc.) are designed for off-line use. POP-1 is for use by a person communicating directly with a computer *via* a typewriter. It differs from JOSS (Shaw 1964) in that it is primarily intended for the sophisticated computer user.* With this in mind I have aimed at a tolerable efficiency of execution, and an ability to define and name new operations, with comprehensive monitoring facilities. On the other hand actual error messages are rather simple.

At the very lowest level, the computer can be used just as a desk calculator. For example

```
23+54+72+98⇒
** 247
```

Lines preceded by double asterisks represent program output.

If the user wants to perform a number of operations using the same number, he can declare a variable to hold it.

```
VARS A;
24·56→A;
97 * A⇒
** 2382·32
(27 * A+3)/(17−5 * A)⇒
** −6·2677
```

* See also MAP (Kaplaw, Strong & Brackett 1966).

All operations are organised by means of a stack, and this basic feature shows up explicitly in the case of two types of instruction:

(i) 'argumentless' output commands (for example ⇒ by itself will cause the number at the top of the stack to be printed);
(ii) 'assignmentless expressions' which are interpreted as implicit assignments to the top of the stack.

With the above examples in mind, certain principles seem obvious.

(i) An on-line language must provide a way of returning the computer to a standard state no matter what state it is in at any time (analogous to the operation of clearing a desk calculator).

(ii) If an error is detected, a message as to the nature of the error must be displayed, and the machine must then enter a clearly defined state, with as little damage as possible to the results already built up.

The on-line user can make best use of such a system by building up complex entities in small units. For example, when calculating a large expression, it is better to work out parts of it and store these parts in variables, rather than try to do the whole thing at once.

In the above examples POP-1 has appeared as a language with a fixed vocabulary. In fact both the vocabulary, and to a limited extent the grammar, are extensible. In the terms of the earlier analogy it is as though we had a calculating machine equipped with an indefinite number of spare keys, on which we can stick new labels. For example, we can define an operation to find the HCF of two numbers by

```
FUNCTION HCF X Y
VARS QUOTIENT REMAINDER
DEFINE X//Y→REMAINDER→QUOTIENT;
IF REMAINDER=0 THEN Y EXIT
HCF(Y, REMAINDER) END
```

Where // produces two values, remainder and quotient. The instinctive reaction of the on-line user, having defined a function, is to test it:

```
HCF(12, 16)⇒
** 4
```

One may decide at this stage that it would be preferable to use an in-fixed operator, say ++ instead of HCF. To do this we need to re-assign the 'value' of the operator HCF, thus:

```
VALUE HCF→VALUE ++;
SETPRECEDENCE("++", 12, "SYMMETRIC");
```

The symbol 'SYMMETRIC' gives the signal that the operator '++' is to be used as an infix.

```
25 ++ 35⇒
** 5
```

The purpose of the SETPRECEDENCE operation is to declare ++ as being an

infixed operator with precedence 12. Thus addition (precedence 6) and multiplication (precedence 5) will be done before ++, i.e.,

```
4+6++7 * 5⇒
**5
```

The line VALUE HCF→VALUE ++ simply assigns to ++ the definition of HCF already established.

POP-1 grammar is a simple precedence grammar. Functions can be written either before their arguments as HCF(X, Y) or after as in X Y HCF.

Functions can be redefined at any time. This is obviously important for an on-line language, since the first definition of a function may be incorrect. In fact, a function definition is no more than an assignment of a constant (the definition) to the name. Thus one could in fact redefine +, *, −, / to have meanings over some field other than the reals (e.g., the complex numbers or a finite field).

For instance to do this for a finite field of characteristic P one writes:

```
VARS P
FUNCTION ADD X Y
DEFINE ADD(X, Y) // P→X→Y; X END
VALUE ADD VALUE+ →VALUE ADD →VALUE+ ;
```

Because of the stack basis of this sequence, the result is to interchange the values of 'ADD' and '+' and similarly for SUB and MULT. Note that the last line has the desired effect of exchanging the values of 'ADD' and '+' because it consists of a sequence of assignments first to, and then from, the stack. Further, since it is executed before the function definition itself is entered (which occurs on the first occasion that the function is called) the operator use of ADD in 'DEFINE ADD (X Y)' already has the meaning previously possessed by +. It is necessary to introduce the names ADD SUB MULT DIV because the old definitions of these operations are needed by the new ones.

Now we get, for example,

```
7→P;
4+5⇒
** 2
```

DIV can be defined using the euclidean algorithm.

Algorithm:

```
FUNCTION DIV X Y DEFINE X * RECIPROCAL (Y) END
FUNCTION RECIPROCAL X
VARS U V H
DEFINE X P EUCLID →U →V →H;

IF NOT (H=1) THEN NL TEXT P NOT PRIME; EXIT
U END
FUNCTION EUCLID X Y
VARS QUOTIENT REMAINDER U V H
DEFINE X//Y→REMAINDER →QUOTIENT',
```

```
IF REMAINDER=0 THEN Y, 1, 0 EXIT
Y REMAINDER EUCLID →U →V →H;
H, U SUB(U, MULT (V, QUOTIENT)), V END
```

The symbol NL means 'new line', i.e., it is a formal function.

The effect of VALUE is to make the next word be treated as a variable. Its opposite is OBEY which makes the next word be treated as a function.

ARRAYS

These are treated as functions with a difference—the difference being that they have a meaning when preceded by an → (apart from the above meaning). Declaring that a variable is to hold an array is distinct from assigning the space to hold that array. The word ARRAY in any list of variables indicates that all following variables are to be treated as arrays (until NORMAL or FUNCTION or the end of the list is reached). Rectangular arrays are created by the function NEWARRAY. Thus, to handle permutations, one might write the following group of functions:

```
VARS LENGTH;
FUNCTION READPERM
VARS ARRAY X NORMAL I
DEFINE 1→I; ⟨⟨1 LENGTH⟩⟩NEWARRAY→ VALUE X;
L 1: NEXTATOM → X(I);
IF I=LENGTH THEN VALUE X EXIT
I+1→I; GOTO NEXT END
```

The brackets ⟨⟨ ⟩⟩ enclose items which are to be evaluated and then made into a list

```
FUNCTION MULTPERM ARRAY X Y
VARS I ARRAY Z
DEFINE 1→I; 1 LENGTH NEWARRAY→VALUE Z;
NEXT: I X Y→ Z(I);
IF I=LENGTH THEN VALUE Z EXIT
I+1→I; GOTO NEXT END
3→LENGTH;
VARS A B C D
READPERM 1 2 3→A; READPERM 3 2 1→B; READPERM 1 3 2→C;
VALUE MULTPERM → VALUE *;
A * B⇒
** 3 2 1
B * C⇒
** 2 3 1
```

Since arrays are created by function, one can write functions to create special arrays, e.g., in a chess program

```
FUNCTION NEWBOARD
VARS I J K ARRAY A
```

```
DEFINE <<1 8 1>> NEWARRAY→VALUE A;
1→I;
L1: IF I>8 THEN VALUE A EXIT
1→J;
L2: IF J>8 THEN I+1→I GOTO L1
ALSO (I MASK 1)+(J MASK 1) MASK 1 THEN "BLACK" ELSE "WHITE"
ALSO →A(I, J), J+1→J GOTO L2 END.
```

Quotation marks enclose atoms. MASK forms the logical AND of its arguments considered as bit patterns. When the conditional THEN ··· ELSE ··· ALSO is obeyed, the commands between THEN and ELSE are obeyed if the top of the stack is 'true' (i.e., non zero) before THEN. Otherwise the commands between ELSE and ALSO are obeyed. ALSO is thus used as a terminator of conditionals rather similar in action to the semi-colon which is used to terminate assignments.

LIST PROCESSING

POP-1 was originally a pure list-processing language, and list-processing facilities are well developed. Let us consider the function REV to reverse a list.

```
FUNCTION REV X
DEFINE IF X ATOM THEN X EXIT
REV(X TL) <> X HD :: 1 END
```

The operator <> means join, i.e., concatenate the lists which are its operands, and :: is CONS in LISP.

Let us translate some dog-English into dog-French. Suppose we have read an English sentence as a list, and suppose we have a dictionary DIC giving what part of speech each English word is, and its French equivalent, together with information as to whether a word (if it is an adjective) comes before or after the word it qualifies. If a sentence is defined as a nounphrase followed by a transitive verb followed by a nounphrase then:

```
FUNCTION SENTENCE X D
VARS U V T
DEFINE NOUNPHRASE (X, D)→T→V;
  "SENTENCE" :: V <> LOOKUP (T HD, D) HD <> (T TL NOUNPHRASE→U)::1;
  U END
FUNCTION NOUNPHRASE X D
  VARS U V T
  DEFINE LOOKUP(X HD, D)→U;
  IF U HD="NOUN" THEN "NOUNPHRASE" : U :: 1; X TL EXIT
  NOUNPHRASE(X TL, D)→T→V;
  V <> U :: 1; T END
```

These functions work by finding the first syntactic unit of the type that bears their name in the list X, giving as a result an analysis of that unit, and whatever remains in the list X. Having produced a parsed version of the sentence, we translate it with the following function:

```
FUNCTION TRANSLATE X G
DEFINE
  IF X HD="SENTENCE" THEN TRANSLATE (X TL HD, G); FRENCH (X TL TL
    HD) SPR;
  TRANSLATE (X TL TL TL HD, G) EXIT
  IF X HD="NOUN" THEN FRENCH (X) SPR EXIT
  IF X HD="ADJECTIVE" THEN IF G="MASCULINE" THEN FRENCH (X) HD SPR
    EXIT
  FRENCH (X) TL HD SPR EXIT

  IF X HD="NOUNPHRASE" THEN GENDER (X TL HD)→G;
  TRANSSELECT (X TL YL, G, "ARTICLE");
  TRANSSELECT (X TL, TL, G, "BEFORE");
  TRANSLATE (X TL HD, G);
  TRANSSELECT (X TL TL, G, "AFTER")
  EXIT
  NL "FAIL" PR  END
```

This works as follows. X is the tree representing the sentence, or a part of it, in the parsed form, containing all the information necessary for a translation. G is the gender of the part of the sentence being translated, if this is relevant. Thus the sentence 'The cat eats fish' is parsed as

```
SENTENCE

  NOUNPHRASE                VERB      NOUNPHRASE

    NOUN      ADJECTIVE     MANGE      NOUN      ADJECTIVE

      CHAT MASC [LE LA LES LES] ARTICLE  POISSON MASC

                                        [LE LA LES LES] ART
```

Note that lists are enclosed in square brackets.

When applying TRANSLATE to this structure, first we apply TRANSLATE to the nounphrase (X TL HD), then we find the French equivalent of the verb, and print it preceded by a space. To translate the nounphrase, we first extract the gender of the noun (GENDER (X TL HD)). We then apply the function TRANSSELECT to the list of adjectives following the noun.

TRANSSELECT is defined by:

```
FUNCTION TRANSSELECT X G S
DEFINE
NEXT: IF X ATOM THEN EXIT
IF POSSELECT (X HD)=S THEN TRANSLATE (X HD, G) ALSO
X TL → X; GOTO NEXT END
```

Thus transselect translates only those items in the list X which have the atom S in the position POSSELECT. Hence the first call translates the articles 'the' and 'a', and the second those adjectives that come before the noun in French. Next, we translate the noun itself, and then those adjectives which come after the noun.

For curiosity's sake, Fig. 1 shows some worked examples done on the console typewriter as a demonstration.

MACROS

Since POP-1 is a list-processing language, one might expect that it would be easy to provide facilities for writing macros. This is indeed the case. First we define the function MACRO by

```
FUNCTION MACRO          VARS X
DEFINE READ→X;
  SETTYPE(X,5,3,15,0);
  <<"FUNCTION" X>> <> INSTREAM→INSTREAM
END
SETTYPE(MACRO,5,3,15,0);
```

This is in fact itself a macro definition, and works as follows. The final SETTYPE command has the effect of making the function MACRO be executed as soon as it is read. Thus if

```
MACRO THIRD . . .
```

occurs in the input instream, then when MACRO is read, it is immediately obeyed. Thus the first action of MACRO is to read the next word, i.e., THIRD off the input stream and store it in X. INSTREAM is a list which, if non-empty, contains the atoms which are next to be read off the input stream. Thus after the evaluation of MACRO the next items to be read off the input stream will be

```
FUNCTION THIRD . . .
```

Thus the effect of MACRO . . . is to define THIRD as a function which is itself evaluated as soon as it is read. Now let us consider how to define THIRD as a macro which will take the third member of a list.

```
  MACRO THIRD
VARS X
  DEFINE MACARGS→X;
    <<"(""("X>> <>[]TL TL HD)] <> INSTREAM→ INSTREAM END
```

Let us see that THIRD defined above will produce the correct reverse Polish input for the assembler.

Suppose THIRD is called by

```
THIRD (A <> THIRD(B))
```

as soon as THIRD is read (and its precedence has been set by settype so that it is not transferred behind the list of arguments, which would be its fate if it were an ordinary function) it is evaluated. The function MACARGS, which is

```
23 + 54 + 72*96 - 298*34 =>

** -3143


VARS A B C%

[Semicolon  Does not occur on the typewriter so use % instead]

19 -> A%  25 -> B%
A*B + 98*(75 + B) + 4*A =>

** 10351

FUNCTION REV X
DEFINE
    IF X ATOM THEN NIL EXIT
    REV(X TL) <> X HD END

REV( [1 2 3 4 5 6]) =>

** 1
[No good so redefine REV]

FUNCTION REV X
DEFINE
    IF X ATOM THEN NIL EXIT
    REV(X TL)  <>  <<X HD >> END


REV( [1 2 3 4 5 6] ) =>

** [6 5 4 3 2 1 ]

[OK. Now load translation program from paper tape]


[THE BIG BLACK CAT EATS THE RED MEAT] SENTENCE

**PLEASE TELL ME ABOUT "BIG"
**PART OF SPEECH                                ADJECTIVE
**FRENCH
**SINGULAR                                      GRAND
**FEMININE                                      GRANDE
**PLURAL                                        GRANDS
**FEMININE                                      GRANDES
**
**IS IT BEFORE OR AFTER OR ARTICLE              BEFORE
**PLEASE TELL ME ABOUT "CAT"
**PART OF SPEECH                                NOUN
**FRENCH                                        CHAT
**IS IT MASCULINE OR FEMININE                   MASCLINE
**YOU MUST REPLY ONE OF MASCULINE OR FEMININE
                                                MASCULINE
```

FIG. 1. An example of a POP-1 "session".

```
**PLEASE TELL ME ABOUT "RED"
**PART OF SPEECH                          AJECTIVE
**I DO NOT KNOW THIS PART OF SPEECH
**PART OF SPEECH                          ADJECTIVE
**FRENCH
**SINGULAR                                ROUGE
**FEMININE                                ROUGE
**PLURAL                                  ROUE##ROUGES
**FEMININE                                ROUGES
**
**IS IT BEFORE OR AFTER OR ARTICLE        AFTER

-> A -> B%

A =>
** 1   [This is the null list, and is the unparsed part of
the input]

B =>
** [SENTENCE [NOUNPHRASE [NOUN CHAT MASCULINE ]
**[ADJECTIVE [NOIR NOIRE NOIRS NOIRES ]AFTER ]
**[ADJECTIVE [GRAND GRANDE GRANDS GRANDES ]BEFORE ]
**[ADJECTIVE [LE LA LES LES ]ARTICLE ]][VERB MANGE
**][NOUNPHRASE [NOUN VIANDE FEMININE ][ADJECTIVE
**[ROUGE ROUGE ROUGES ROUGES ]AFTER ][ADJECTIVE
**[LE LA LES LES ]ARTICLE ]]]

[This is the tree structure of the sentence represented in
bracketed form - Let us explore this tree]

3 HD =>
** SENTENCE

B TL HD =>
** [NOUNPHRASE [NOUN CHAT MASCULINE ][ADJECTIVE
**[NOIR NOIRE NOIRS NOIRES ]AFTER ][ADJECTIVE
**[GRAND GRANDE GRANDS GRANDES ]BEFORE ][ADJECTIVE
**[LE LA LES LES ]ARTICLE ]]

B TL TL HD =>
** [VERB MANGE ]

B TL TL TL HD =>
** [NOUNPHRASE [NOUN VIANDE FEMININE ][ADJECTIVE
**[ROUGE ROUGE ROUGES ROUGES ]AFTER ][ADJECTIVE
**[LE LA LES LES ]ARTICLE ]]

B TL TL TL TL HD
ERROR 30 COMPILER   HD
ERROR 30
[An attempt has been made to evaluate HD with an atomic argument]

B 1 TRANSLATE
 LE GRAND CHAT NOIR MANGE LA VIANDE RO
UGE

FUNCTION SEN##TRANS X
DEFINE X SENTENCE TRANSLATE END

[THE LITTLE B##BLACK BOY EATS THE MEAT] TRANS
 LE PETIT GARCON NOIR MANGE LA VIANDE

[ The character  "##"  causes the atom just typed to be ignored]
```

a library function defined in POP-1, reads the string of atoms (A <> THIRD (B)) and converts it into the list [A <> THIRD(B)] (square brackets denote list structures, round brackets are just symbols). Assuming INSTREAM to be empty (i.e., we are not in the middle of a macro call), this first call of THIRD will leave

((A <> THIRD(B)) TL TL HD)

in INSTREAM. Processing this will result in A being compiled and <> being set aside before THIRD is called again. Thus at the second call of THIRD INSTREAM will contain (B)) TL TL TL HD), and the second call will turn this into ((B) TL TL HD)) TL TL HD). Hence B TL TL HD will now be compiled, and then the <> operator which was set aside, and then the TL TL HD, so the total effect of the macro calls is

A B TL TL HD <> TL TL HD

which is the correct reverse Polish form of the input stream.

IMPLEMENTATION

The present implementation (on an Elliott 4100) is by means of an interpreter. A program being input is first processed to set variables and functions into the correct reverse Polish order. The resulting stream is then translated into pseudo-machine code which is obeyed by the interpretive routine. Certain words such as FUNCTION, THEN, EXIT have the effect of activating special routines as soon as they are input. The user himself can define such words, e.g., in making macro definitions. Thus the whole system is easily extensible.

AIMS

POP-1 was originally conceived as a list-processing language which has acquired an increasingly on-line emphasis. It is now adopted as a language for exploratory programming projects at the Experimental Programming Unit of the University of Edinburgh, and as such will be incorporated in the on-line time-sharing system being developed there.

EDITOR'S NOTE

POP-1 has now been superseded by a greatly extended version, POP-2, which is the subject of Chapter 14.

REFERENCES

Kaplaw, R., Strong, S., & Brackett, J. (1966). MAP: a system of on-line mathematical analysis. *Technical report MAC-TR-24*, Massachusetts Institute of Technology.

Shaw, J. C. (1964). JOSS: A designer's view of an experimental on-line computing system. In *AFIPS Conference Proceedings*, Vol. 26. Baltimore and London: Spartan Books.