PART 4. POP-2 PROGRAM LIBRARY

INTRODUCTION

This part of the book contains a selection of programs from the POP-2 library of the Edinburgh 4100 installation. The programs have mostly been run successfully on other POP-2 implementations. They were written in POP-2 as defined in the *POP-2 Papers* and have been altered to conform to the minor changes introduced in this revised manual. Since the POP-2 compiler had not been amended to conform to the revisions the alterations could not be checked on the machine, but any oversights will be easy to correct. The changes that had to be made were

(1) *recordfns* and *stripfns* have only 2 arguments not 3
(2) *next* was previously allowed as a synonym for *dest.*
(3) The syntax word **function** is no longer allowed in a declaration after **vars**.

The *section* facility available in revised POP-2 has not been used but it would be advantageous to enclose these programs each in its own section.

For a complete list of the changes see Appendix to the Reference Manual in Part 3.

The ICL 4100 POP-2 timesharing system outputs a colon when it is waiting for a message from the console user. Thus lines beginning with a colon in the examples of program use are lines typed by the user rather than by the machine.

Diamond brackets have sometimes been used in the program descriptions to denote a syntax class, as in the Reference Manual. Thus when instructed to type '**vars** *identifier*;' you must type '**vars** *identifier*;' but for '**vars** ⟨identifier⟩;' you may type '**vars** *x*;' or '**vars** *y*;' etc.

The amount of store used is given in 'blocks'. One block is 512 words of 24 bits.

It is hoped that the library programs will be useful and also give some insight into programming techniques in POP-2.

Program Name. LIB ALLSORT
Source. D. J. S. Pullin, D. B. Anderson, DMIP; *Date of issue.* December 1968.

¶*Description.* This program will sort the elements of a list into the order specified by the user. When supplied with a suitable predicate, a list containing items of any type can be sorted. There is a function provided which can be used to sort a list of words into dictionary order.

¶*How to use program.* The program should be compiled by typing:
COMPILE(LIBRARY([LIB ALLSORT]));

The function ALLSORT takes two arguments, a list, and a function which determines the order into which the list is to be sorted, and produces as its result, the ordered list.

i.e. ALLSORT ∊ LIST, FUNCTION => LIST.
where FUNCTION ∊ ITEM1, ITEM2 => TRUTHVALUE.

The function argument, FUNCTION, must return the result TRUE if ITEM1 is to be before ITEM2, otherwise it must return the result FALSE.

For example, the function:

```
FUNCTION BEFORE X Y;
   IF X< Y THEN TRUE ELSE FALSE CLOSE
END;
```

will cause a list of numbers to be sorted into ascending order:

```
ALLSORT([9 0 3 7 6 9 8], BEFORE) =>
**[0 3 6 7 8 9 9],
```

It should be apparent that the function BEFORE is, in actual fact, just the function NONOP<, so, similarly, we could order the list into descending order by:

```
ALLSORT([90 3 7 6 9 8], NONOP>) =>
**[9 9 8 7 6 3 0],
```

A function ALFER is provided which can be used to sort lists of words into dictionary order:

```
     ALFER∊ WORD, WORD => TRUTHVALUE.
e.g. ALLSORT([AN ABLAUT IS NOT ABLE TO ACCOMPANY A],
     ALFER) =>
  **[A ABLAUT ABLE ACCOMPANY AN IS NOT TO],
```

It may be convenient to the user to use partial application with ALLSORT, in the following way, for example,

```
ALLSORT(% NONOP<%) -> NUMSORT;
NUMSORT([7 2 9 15 1]) =>
**[1 2 7 9 15],
```

¶*Method used.* An algorithm due to C. A. Hoare is used. A random element of the list is chosen, and the list is partitioned into three parts—those elements before, equal to, and after the chosen element. This procedure is now applied to these three lists, and so on, until the list is sorted.

The expected number of comparisons is 1.4 $\log_2 n$ for a list of n elements.

¶*Global variables*. HIGSEED HIGPIG QQHIGPIG ALLSORT LISTIFY PRIOR ALFER.

¶*Performance*. The average time taken to sort a list of 1000 random numbers is 25 seconds.

REFERENCES

Hoare, C. A. R. (1961) Algorithms 63 and 64. *Comm. Ass. comput. Mach.*, *4*, 321.

Hibbard, T. N. (1963) An empirical study of minimal storage sorting. *Comm. Ass. comput. Mach.*, *6*, 207.

Scowen, R. S. (1965) Algorithm 271 (quickersort). *Comm. Ass. comput. Mach.*, *8*, 669.

Blair, C. R. (1965) Certification of algorithm 271, *Comm. Ass. comput. Mach.*, *9*, 354.

Boothroyd, J. (1967) Algorithms 25, 26, 27. *Computer Journal*, *10*, 308.

```
[ALLSORT]

VARS HIGSEED; 10->HIGSEED;

FUNCTION QQHIGPIG L;
  L;INTOF(LAMBDA;
  (125*HIGSEED+1)//16384;.ERASE->HIGSEED;HIGSEED/16384;
  END.APPLY*LENGTH(L))->L;

LOOP:IF L THEN L-1->L;.TL;GOTO LOOP CLOSE;
  .HD;
END

FUNCTION NCJOIN X Y;VARS Z;
  IF X.NULL THEN Y EXIT;

  X;

LOOP:TL(X)->Z;IF Z.NULL THEN Y->TL(X) EXIT;
  Z->X;GOTO LOOP;
END

FUNCTION ALLSORT LIST ALFER;
  VARS Y Z A QQV QQW QQS;  0;
  LOOP:IF LIST.NULL OR LIST.TL.NULL THEN GOTO SPLIT CLOSE;
  NIL->QQS;NIL->Y;NIL->Z;
  LIST.QQHIGPIG->QQW;
L1: DEST(LIST)->LIST->QQV;
    IF ALFER(QQW,QQV) THEN QQV::Z->Z;
      ELSEIF ALFER(QQV,QQW) THEN QQV::QQS->QQS;

      ELSE QQV::Y->Y;
    CLOSE;
  IF LIST.NULL THEN Z; Y; 1; QQS->LIST; GOTO LOOP CLOSE; GOTO L1;
SPLIT: ->A; IF A=0 THEN LIST EXIT;  ->Y;
  IF A=1 THEN ->Z; LIST,Y.NCJOIN; 2; Z->LIST; GOTO LOOP CLOSE;
  Y,LIST.NCJOIN->LIST;
  GOTO SPLIT;
END

FUNCTION ALFER A B; VARS C D;
  [%A.DESTWORD-10%]->A;[%B.DESTWORD-10%]->B;

LOOP:IF B.NULL OR (A.DEST->A->C;B.DEST->B->D;C>D) THEN FALSE EXIT;
  IF C=D THEN GOTO LOOP CLOSE;
  TRUE;
END

2.NL;'ALLSORT NOW READY TO USE'.PR;2.NL;
```

Program name. LIB ASSOC
Source. A. P. Ambler, R. M. Burstall, DMIP; *Date of issue.* February 1969.

¶*Introduction.* It is sometimes useful to associate pairs of items together so that, given the first item of any pair, one can get the second. For example, a dictionary can be regarded as a set of pairs {(cat, chat), (dog, chenet)...}. A function is needed which, given the dictionary and "cat", will produce "chat". It is also useful to have functions to add new pairs to the dictionary and to alter the items in the pairs. One might want to add (duck, canard) to the dictionary, or alter (dog, chenet) to (dog, chien). The system of association sets described here provide such a facility. In some respects they can also be thought of as 'non-numerical strips'.

The idea of associated pairs is taken further in the accompanying library program LIB NEW STRUCTURES, which provides non-numerical arrays and pseudo-records.

¶*Association sets.* An association set is a set of pairs, a pair being two items which are 'associated' with each other (not necessarily a POP-2 pair). For the purposes of this description, one item of a pair is called a *subscript* and the other a *component.* (These names have been used because of the similarity between association sets and POP-2 strips. *Argument* and *value* could have been used equally well.) Several functions for processing association sets are desirable. These include the constructive functions which add pairs to the set, and the functions for selecting, or updating, a component of a pair in the set, given the subscript with which it is associated.

¶*Component-selecting functions.* There is a family of functions (component-selecting functions) which, given an association set and a subscript, search through the pairs for one with that subscript. If such a pair is found, then the result is the component of that pair. The action taken when such a pair is not found depends on the particular component-selecting function being used.

Example. Consider a dictionary which is an association set {(cat, chat), (dog, chien), (duck, canard)}. Then any component-selecting function, applied to it and to "dog" should produce "chien". A particular component-selecting function applied to this dictionary and "horse" would produce "undef", whereas another one might cause 'please tell me the French for horse' to be output.

¶*Component-updating functions.* There is a family of functions (component-updating functions) which, given an association set, a subscript, and a component, search through the set for a pair whose subscript is that component. If such a pair is found, then the existing component in it is replaced by the supplied component. The action taken when one is not found depends on the particular component-updating function being used.

Example. Consider a dictionary which is an association set {(cat, chat), (dog, chenet), (duck, canard)}. Then any component-updating function given "chien", "dog", and the dictionary should alter the pair (dog, chenet) to (dog, chien). A particular function, given "cheval", "horse", and the dictionary, might add the pair (horse, cheval) to the dictionary.

¶*Constructive functions for adding pairs to an association set.* A function for adding items to a set is ideally one which searches through the set before adding the item, to check that it does not already occur in the set. If the pairs of an association set are considered to be characterized by their subscripts, then two such searching functions have been described above—the component-selecting and component-updating functions. In this program the same function is used to add a new pair to the set as is used to update a pair—it is a function which searches through the set for a pair whose subscript is the same as the supplied subscript, and, if it fails to find one, constructs a new pair from the supplied subscript and component and adds it to the set. (There is also a component-selecting function which can add new pairs to a set. If this function fails to find a suitable pair it constructs a new one, the component of which is obtained by applying a suitable function to the supplied subscript.)

¶*Functions operating on association sets.* The 'empty' association set is a data structure and is not represented by a variable, but by the result of the operation ASSNIL (precedence 1).
assnull ϵ association set => truth value

The function ASSNULL tests whether an association set is empty or not.
assnull ϵ association set => truth value

The three characteristic facilities (namely component-selection, component-updating and pair-addition) are combined in one function—here called an *assoc function.* An assoc function is a doublet. It is a component-selecting function which has a component-updating (and consequently pair-addition) function in its update part. The function ASSOC is provided, together with two functions ASSOCF and ASSOCM for defining other assoc functions. These differ in the result obtained when attempting to select from an association pair which does not occur in the association set (that is, they differ in their 'fail' result). They all have the same update function ASSUF.

ASSOC is the simplest of the assoc functions. It maps a subscript and an association set onto a component. The fail result is always UNDEF.

assoc ϵ subscript, assoc set => component or UNDEF

ASSOCF and ASSOCM are used to define other assoc functions. ASSOCF applied to an item (the desired fail result) produces a function whose fail result is that item. It does not depend on the subscript.

assocf ϵ fail result => (subscript, assoc set => component or undef)

ASSOCM is used to define a constructive assoc function. Applied to a 'compute' function it produces an assoc function whose fail result is obtained by applying the compute function to the subscript. Furthermore, in the fail case, a new subscript-component pair is added to the association set.

```
assocm ϵ (subscript => component) => (subscript, assoc set
                                        => component)
```

The function which forms the update part of all these assoc functions is ASSUF. It either replaces the component of an existing pair by the supplied component, or creates a new subscript-component pair and adds it to the association set.

assuf ϵ component, subscript, assoc set => ()

The function ASSLENGTH maps an association set onto the number of pairs in it.

asslength ϵ association set => integer

¶*Association lists.* An association set is actually represented by a list. An association list is *not* a POP-2 list and POP-2 list processing functions cannot be used on it. Some association list processing functions are available. These include ASSNULL and ASSLENGTH which have been described above. Other functions on association lists are:

ASSPR	—	which prints an association list asspr ϵ association list => ()
ASSSUB	—	which maps an association list onto the subscript of the first pair in it. asssub ϵ association list => subscript
ASSCPT	—	which maps an association list onto the component of the first pair in it. asscpt ϵ association list => component
ASSTL	—	which maps an association list onto an association list by removing the first pair. asstl ϵ association list => association list
ASSNEXT	—	which maps an association list onto the first component, the first subscript and the 'tail' of the list. assnext ϵ association list => component, subscript, association list.

It should be noted that, in this representation of association sets as lists ASSUF (the update function of ASSOC, ASSOCF, ASSOCM) adds new pairs to the *end* of the association list, rather than to the beginning.

¶*Errors.* An attempt to apply an association list processing function to an item which is not an association list produces the POP-2 error 45 and SETPOP, the culprit being the item which was not an association list.

¶*Implementation of association lists and their efficiency.* Association lists are implemented in this program using records of three components—dataword ASS. The components are ASSSUB, ASSCPT and ASSTL. Such a record is called an ASSOC PAIR. An empty association list is an assoc pair whose ASSTL is FALSE. An association list is either an empty association list or it is an assoc pair whose asstl is another association list.

The empty association list requires four words of store, and each pair of items in the list requires four words.

¶*Global variables.* All global variables in the program are prefixed by the letter ASS.

¶*Store used.* The program requires approximately 3 blocks of store on the 4100.

¶*How to use the program.* The program should be compiled by typing

```
COMPILE(LIBRARY([LIB ASSOC]));
```

When the compilation has been completed, LIB ASSOC READY is output on the console and the program is ready. Use of the program is best demonstrated by the examples which follow.

¶*Examples*

Example 1. Use of ASSOC to create an English-French dictionary.

```
: VARS DICT;  ASSNIL -> DICT;         (creates 'empty' dictionary)
: "CHAX" -> ASSOC("CAT",DICT);        (adds the pair (cat, chax))
: ASSOC("CAT", DICT) =>               (selects the component
** CHAX,                               associated with "cat")
```

```
: "CHAT" —> ASSOC("CAT",DICT);            (replaces "CHAX" by
: ASSOC("CAT",DICT) =>                     "CHAT")
** CHAT,
: ASSOC("DOG",DICT) =>                    (attempt to select from a
** UNDEF,                                  pair which does not exist)
: "CHIEN" —> ASSOC("DOG",DICT);           (adds the pair (dog, chien))
: ASSPR(DICT);                            (prints the association list)
    CAT  CHAT
    DOG  CHIEN
```

Example 2. Use of ASSOCF in using stored data.

```
: VARS DATA;  ASSNIL —> DATA;             (creates 'empty' structure)
: 10 —> ASSOC("WEIGHT",DATA);             (adds a pair to the structure)
: ASSOC("HEIGHT",DATA)*ASSOC              (attempt to select component
("WEIGHT",DATA) =>                        which does not exist.
ERROR 47                                  The result is UNDEF, which
CULPRIT UNDEF                             gives an arithmetic error)
SETPOP:
: VARS NUMASS;  ASSOCF(0) —> NUMASS;      (creates a new assoc func-
                                          tion whose fail result is 0.)
: NUMASS("HEIGHT",DATA)*NUMASS            (again there is an attempt
("WEIGHT",DATA) =>                        to select from a pair which
** .                                      does not exist. The result
                                          is now 0—a suitable argu-
                                          ment for arithmetic
                                          operations)
```

Example 3. Use of ASSOCM in constructing and using a dictionary.

```
: VARS FRENCH;  ASSNIL —> FRENCH;         (creates an 'empty'
: VARS FIND;                              dictionary)
: ASSOCM(LAMBDA X;  NL(1);
:           PR([WHAT IS THE FRENCH
            FOR]<>[%X%]);
:           ITEMREAD();                   (creates an assoc function
:           END;) —> FIND;                whose fail action is to ask
                                          for words it cannot find)
: FIND("CAT",FRENCH) —> X;                (attempt to select from pair
                                          which does not exist)
[WHAT IS THE FRENCH FOR CAT]              (the system is waiting for
                                          a reply)
: CHAT                                    (reply typed by user. The
                                          pair (cat, chat) is now
:                                         added to the dictionary)
: FUNCTION TRANSLATE SENTENCE             (definition of a function to
            DICT;                         convert words from one
:  MAPLIST(SENTENCE, FIND(%DICT%));       language into another)
: END;
: "LE" —> FIND("THE",FRENCH);             (adding new pairs to the
: "POISSON" —> FIND("FISH",FRENCH);       French dictionary)
: TRANSLATE([THE CAT EATS THE             (the program cannot trans-
  FISH],FRENCH) =>                        late the sentence until it
  [WHAT IS THE FRENCH FOR EATS]           knows the French for EATS)
: MANGE                                   (reply typed by user)
** [LE CHAT MANGE LE POISSON]             (output by system as soon
                                          as all the needed words are
                                          in the dictionary).
```

```
[ASSOC]

VARS ASSNILCONS  ASSNULL  ASSNEXT ASSLENGTH ASSPR ASSCONS
  ASSTL ASSSUB ASSCPT  ASSF ASSUF ASSCF ASSUCF;

RECORDFNS("ASS",[0 0 0])->ASSTL ->ASSSUB ->ASSCPT
       ->ASSNEXT  ->ASSCONS;

VARS OPERATION  7  ASSNIL;

ASSCONS(%UNDEF,UNDEF,FALSE%)-> ASSNILCONS;
ASSNILCONS-> NONOP ASSNIL;


FUNCTION ASSNULL ASS; NOT(ASSTL(ASS)); END;

FUNCTION ASSLENGTH  ASS;  VARS ASSN; 0->ASSN;
LOOP:
  IF ASSNULL(ASS) THEN ASSN
    ELSE ASSN+1->ASSN; ASSTL(ASS)->ASS; GOTO LOOP;
  CLOSE;
END;

FUNCTION ASSPR ASS; VARS ASSCPTT ASSSUBT; NL(1);
LOOP:
  IF ASSNULL(ASS) THEN RETURN
    ELSE ASSNEXT(ASS)-> ASS -> ASSSUBT ->ASSCPTT;  SP(4);
      PR(ASSSUBT); SP(4); PR(ASSCPTT); NL(1);
  GOTO LOOP;
  CLOSE;
END;

FUNCTION ASSF X ASS  U;
LOOP:
  IF ASSNULL(ASS) THEN U
    ELSEIF X=ASSSUB(ASS) THEN ASSCPT(ASS)
    ELSE ASSTL(ASS)-> ASS; GOTO LOOP;
  CLOSE;
END;

FUNCTION ASSUF Y X ASS;
LOOP:
  IF ASSNULL(ASS) THEN
    X->ASSSUB(ASS); Y->ASSCPT(ASS); ASSNIL->ASSTL(ASS);
    ELSEIF X=ASSSUB(ASS) THEN Y->ASSCPT(ASS);
    ELSE ASSTL(ASS)-> ASS; GOTO LOOP;
  CLOSE;
END;

FUNCTION ASSCF ASS X U;
  ASSF(X,ASS,U);
END;

FUNCTION ASSUCF Y ASS X;
 ASSUF(Y.X,ASS);
END;

FUNCTION ASSMEMO X ASS ASSMEMOF;
LOOP:
  IF ASSNULL(ASS) THEN
    X.ASSMEMOF->Y; X->ASSSUB(ASS); Y->ASSCPT(ASS);
    ASSNIL->ASSTL(ASS); Y   ;
    ELSEIF X=ASSSUB(ASS) THEN ASSCPT(ASS)
    ELSE ASSTL(ASS)-> ASS; GOTO LOOP;
  CLOSE;
END;
```

```
FUNCTION ASSMEMOC ASS ASSMEMOF X;
  ASSMEMO(X,ASS,ASSMEMOF);
END;


FUNCTION NEWCPTMEMO ASSKEY ASSMEMOF; VARS ASSFF;
  ASSMEMOC(%ASSMEMOF,ASSKEY%)-> ASSFF;
  ASSUCF(%ASSKEY%) ->UPDATER(ASSFF); ASSFF;
END;
VARS ASSOC;
ASSF(%UNDEF%)->ASSOC;
ASSUF->UPDATER(ASSOC);

FUNCTION ASSOCF FAIL; VARS ASSOF;
  ASSF(%FAIL%)->ASSOF;
  ASSUF->UPDATER(ASSOF); ASSOF;
END;

FUNCTION ASSOCM ASSFN; VARS ASSOM;
  ASSMEMO(%ASSFN%)->ASSOM;
  ASSUF->UPDATER(ASSOM); ASSOM;
END;

'LIB ASSOC READY'.PR;
```

Program name. LIB CALL AND EXPLAIN
Source. R. M. Burstall, DMIP; *Date of issue.* February 1969.

¶*Description.* This program enables one to make functions self-explanatory so that they can be used more easily by someone unfamiliar with them, and to use functions so prepared.

It is often difficult to use a function provided by someone else, for example, a library function, because one has forgotten: (a) what it does, (b) what parameters it needs, and (c) what results it produces.

Similarly, if a package of functions is available, it is often difficult to remember exactly what it is for, and what functions it contains.

The EXPLAIN facility allows a person ('the documenter') to create explanatory material in a file about a package of functions so that any other person ('the user') can use them more easily. This is intended to give an aide-memoire rather than a substitute for written documentation, and is thus of the same ilk as POPCHAT in the POPSTATS system, but less comprehensive, shorter, and simpler.

Two functions are provided for the user. The EXPLAIN function enables him to get an explanatory message about a package or any function in it. The CALL function enables him to call any function in the package and then be asked for its parameters by their names (he must respond by typing a suitable POP-2 expression terminated by ** for each parameter requested), and finally to have the results of the function printed with their names.

¶*How to use the program.* Assume in the following that ⟨package⟩ is the name of a package of functions, for example, LIB SETS.

A. *How to use a self-explanatory package.* Compile the explanatory package by typing:

COMPILE(LIBRARY([EXPLAIN⟨package⟩])); (Assuming an explanation file is available)

This may be repeated for any other packages you wish to use.

To get an explanation of the package, type:

EXPLAIN ⟨package⟩;

To get an explanation of a function ⟨f⟩, type:
EXPLAIN ⟨f⟩; (a reply of UNDEF means that no explanation is available).

To use the CALL facility with a function ⟨f⟩, type:

```
CALL ⟨f⟩;        The machine will now print:
ENTER ⟨f⟩
⟨p⟩=:            where ⟨p⟩ is the name of the first parameter.
```

The user must now type in the required value for this parameter (any POP-2 expression) and terminate it with **, the machine will now output the name of the next parameter, and so on for all parameters of the function.

The machine now computes the values of ⟨f⟩ and types:

```
EXIT ⟨f⟩
 ⟨r1⟩ = ⟨v1⟩     where ⟨r⟩ is the name of the result,
 ⟨r2⟩ = ⟨v2⟩        and ⟨v⟩ its value, for the ⟨n⟩ results of the function.
   .
   .
 ⟨rm⟩ = ⟨vm⟩
```

If the function ⟨f⟩ had not previously been made self-explanatory, the use of CALL results in the message NOSPEC (no specification) being typed.

Notes

(1) Making a function self-explanatory affects its FNPROPS, but does not otherwise alter it. This means that it is still possible to use ⟨f⟩ in the normal way, for example, in some other function by f(x, y).
(2) If the DEBUG package is in use, do not CALL a function which has been BUGged. A self-explanatory function has already been SPECced, since SPECIFY, a function similar to SPEC, is used to make it self-explanatory.

(3) To remove the current explanations and free some store, the user should type: . ASSNIL —> EXPLS;

Example of the use of a self-explanatory package. In the following all user's input is underlined.

```
COMPILE(LIBRARY([LIB CALL AND EXPLAIN]));
COMPILE(LIBRARY([RANDOM PROG]));
COMPILE(LIBRARY([EXPLAIN RANDOM PROG]));

EXPLAIN [RANDOM PROG];

'RANDOM PROG CONTAINS FUNCTIONS TO GENERATE
RANDOM NUMBERS:
RAND01 TO GET A RANDOM NUMBER X, 0 =< X< 1.
RANDINT TO GET A RANDOM INTEGER I,  L =< I =<U.
RANDNORM TO GET A NORMALLY DISTRIBUTED RANDOM NUMBER.
TO RESET RANDOM SEQUENCE, ASSIGN A NUMBER TO RANSEED'

CALL RANDNORM;

ENTER RANDNORM
  MEAN =: 30**
  STANDEV =: 6. 5**
EXIT RANDNORM
  RANDOM = 30. 38

EXPLAIN RANDINT;

'RANDINT NEEDS A LOWER BOUND AND AN UPPER BOUND. IT
PRODUCES A RANDOM INTEGER EVENLY DISTRIBUTED BETWEEN
THEM'

CALL RANDINT;

ENTER RANDINT;
  LOWBOUND =: 3**
  UPBOUND =: 10**
EXIT RANDINT;
  RANDOM = 8
```

Note. The above is an example using a hypothetical program, there being no library file called RANDOM PROG.

B. *How to make a package self-explanatory.* The documenter must create a file EXPLAIN ⟨package⟩ where ⟨package⟩ is the name of the package. This file is then compiled by the user when he wants to use the package in self-explanatory mode.

'. . . .' means an explanatory message, that is, a string or list of strings, ⟨f⟩ means the name of a function, ⟨p⟩ the name of a parameter and ⟨r⟩ of a result. Then the file EXPLAIN package is:

```
'...' -> EXPLAIN(⟨package⟩)
'...' -> EXPLAIN(⟨f⟩);                       }  repeated
SPECIFY(⟨f⟩, [⟨p⟩..⟨p⟩], [⟨r⟩..⟨r⟩]);         }  for each function ⟨f⟩
```

¶*Example.* The file [EXPLAIN RANDOM PROG] might be:

```
'RANDOM CONTAINS etc.' -> EXPLAIN([LIB RANDOM]);
'RAND01 MAKES A RANDOM etc.'-> EXPLAIN(RAND01);
SPECIFY (RAND01, [], [RANDOM]);
'RANDINT NEEDS etc.' -> EXPLAIN(RANDINT);
SPECIFY(RANDINT, [LOWBOUND UPBOUND], [RANDOM]);
'RANDNORM PRODUCES etc.' -> EXPLAIN (RANDNORM);
SPECIFY(RANDNORM, [MEAN STANDEV], [RANDOM]);
```

¶*Method used.* Explanations are kept in a simple association list against names of packages or functions.

Parameter names and result names are hung on the fnprops list as in LIB DEBUG, in fact SPECIFY is the same function as SPEC. CALL just uses this information to request the parameters and print the results.

¶*Names of global variables and functions.* Global variables: EXPLS EXPLAIN(operation 2), CALL(operation 2) ITSREAD, ASSNIL, ASS, ASSU.

R E F E R E N C E N C E S

Burstall, R. M. (1968) The helpful civil servant, a conversational control routine. *Research Memorandum MIP-R-38.* Edinburgh: Department of Machine Intelligence and Perception.
Michie, D. & Weir, S. (1968) Application of Burstall's control routine to conversational statistics. *Research Memorandum MIP-R-39.* Edinburgh: Department of Machine Intelligence and Perception.
Library file LIB POPSTATS.
Library file LIB DEBUG.

A D D E N D U M T O L I B C A L L A N D E X P L A I N

Source. D. L. Marsh, DMIP; *Date of issue.* June 1969

A further command function has been added called TEACH.

This operates in the same way as the CALL function except that on exit from the function the machine does not type the function results but waits for the user to input them. These must by typed in the same format as the arguments, that is, a POP-2 expression terminated by **. The machine will check the user's result with the actual result and will reply 'CORRECT' or with the message 'NO — THE CORRECT ANSWER IS' followed by the actual result.

¶*Example*

```
: COMPILE(LIBRARY([EXPLAIN LIB SETS]));
: TEACH UNION;

ENTER UNION
SET1 = : [1 2 3] **
SET2 = : [2 3 4] **
EQUIV = : NONOP = **

EXIT UNION
SET = : [1 2 3 2 3 4 ] **
'NO — THE CORRECT ANSWER IS' [1 2 3 4]
:
: TEACH UNION;

ENTER UNION
SET1 = : [ 5 6 7 9] **
SET2 = : [A 7 10] **
EQUIV = : NONOP = **

EXIT UNION
SET = : [5 6 9 A 10] **
'NO — THE CORRECT ANSWER IS' [5 6 7 9 A 10]
:
: TEACH UNION;

ENTER UNION
SET1 = : [1 2 3] **
SET2 = : [2 3 4] **
EQUIV = : NONOP = **

EXIT UNION
SET = : [1 2 3 4] **
'CORRECT'
:
: TEACH MEMBER;

ENTER MEMBER
ELEMENT = : "FUMIGATE" **
SET = : [FICKLE FANDANGO FISSIPAROUS PUSTULE
PERSPICACIOUS]**
EQUIV =
: NONOP = **

EXIT MEMBER
TRUTHVAL = : TRUE **
'NO — THE ANSWER IS' 0
```

```
[CALL AND EXPLAIN]

VARS TEACHEQUIV;

FUNCTION ASSNIL; NIL::NIL END;

FUNCTION ASS X A; VARS L; A.TL->L;
 L0:
  IF L.NULL THEN UNDEF
  ELSEIF EQUAL(X,L.HD.FRONT) THEN L.HD.BACK
  ELSE L.TL->L; GOTO L0 CLOSE
END;


FUNCTION ASSU Y X A; VARS L;   A.TL->L;
 L0:
  IF L.NULL THEN X::Y::A.TL->A.TL
  ELSEIF EQUAL(X,L.HD.FRONT) THEN Y->L.HD.BACK
  ELSE L.TL->L; GOTO L0 CLOSE
END;

ASSU->UPDATER(ASS);

VARS EXPLS OPERATION 2 EXPLAIN; .ASSNIL->EXPLS;

FUNCTION EXPLAIN F;
  NL(1); PR(ASS(F,EXPLS));   NL(1);
END;

LAMBDA E F; E->ASS(F,EXPLS) END-> UPDATER(NONOP EXPLAIN);

FUNCTION SPECIFY F SPEIN SPEOUT;
  IF F.FNPROPS.ATOM THEN F.FNPROPS::NIL->F.FNPROPS CLOSE;
  [%F.FNPROPS.HD,SPEIN,SPEOUT%]
    ->F.FNPROPS.HD
END;

FUNCTION ITSRD; VARS I IL; NIL->IL;
L: .ITEMREAD->I; IF I="**" THEN POPVAL(REV(IL)<>[;GOON]) EXIT;
  I::IL->IL; GOTO L
END;

VARS CALLPRINT OPERATION 2 CALL; PR->CALLPRINT;

FUNCTION CALL F; VARS SPEIN SPEOUT PARS;
  IF F.FNPROPS.HD.ATOM THEN "NOSPEC".PR EXIT;
  NL(1); "ENTER".PR;SP(1); F.FNPROPS.HD.HD.PR; NL(1);
  F.FNPROPS.HD.TL.HD->SPEIN;
  APPLIST(SPEIN,LAMBDA NAME;SP(1); NAME.PR; 1.SP; "=".PR; 2.SP; .ITSRD END);
  .F; NL(1); "EXIT".PR; SP(1); F.FNPROPS.HD.HD.PR; NL(1);
  F.FNPROPS.HD.TL.TL.HD->SPEOUT;
  MAPLIST(SPEOUT,ERASE); .REV;
  APPLIST(SPEOUT,LAMBDA NAME; .DEST->PARS; SP(1); NAME.PR; 1.SP; "=".PR;
                   .CALLPRINT; PARS
                  END);.ERASE ;NL(1)
END
FUNCTION TEACHPRINT ANS;
  VARS GUESS;.ITSRD->GUESS;
  IF  TEACHEQUIV(GUESS,ANS) THEN PR("CORRECT")
                    ELSE PR('NO - THE ANSWER IS');2.SP;
                           PR(ANS);
  CLOSE;
  1.NL;
END;

[TEACHPRINT]->FNPROPS(TEACHPRINT);

VARS OPERATION 2 TEACH;

FUNCTION NT12A F; CALLPRINT ; TEACHPRINT->CALLPRINT;
  CALL F;->CALLPRINT;
END;

FUNCTION TEACH;
  POPMESS([OPERATOR 'USING TEACH. CHECK AUTHORISATION']);
  NT12A->NONOP TEACH; .NT12A;
END;

2.NL; 'CALL AND EXPLAIN READY FOR USE'.PR; 2.NL;
```

Program name. LIB DCOMPILE
Source. D. J. S. Pullin, DMIP. *Date of issue.* December 1968.

¶*Description.* This program provides a simple line editing facility for use with Multi-POP and the disc.

When DCOMPILE is entered, all input from the teletype, as well as being compiled in the normal way, is copied to the user's disc track. Line numbers are output at the beginning of each line.

It should be noted that DCOMPILE does not allow insertion of lines in the middle of a file; it allows the user to compile a selected area up to a specified line, when the user can continue typing from that point (see examples).

¶*How to use program.* The program should be compiled by typing:
COMPILE(LIBRARY([LIB DCOMPILE]));

(1) To initially set up DCOMPILE so that the teletype input is being copied to the disc, as well as being compiled, type:
DCOMPILE(TR, N, 0, 0);   where TR is your disc track, and N is a suitable sector.

A line number will be output at the beginning of each line. If a compile error occurs, the usual error messages are output, the disc file is closed, and an exit is made from DCOMPILE.

(2) To restart after an error in line M+1 inside a function which starts at line L, type:
DCOMPILE(TR, N, L, M);

The lines L to M will be output on the teletype, the next line number is output, and the user may now continue typing.

(3) To restart after an error in the first line of a function starting at line L, type:
DCOMPILE(TR, N, L, 0);

In all cases when the user wishes to close the disc file the HALT character (CTRL and T), should be typed, the message:
'DISC FILE CLOSED. NEXT FREE SECTOR IS' S

where S is an integer, will then be output, and DCOMPILE will exit.

The complete file can be used as a normal file, using DISCIN, and it should be noted that the line numbers do not appear on the disc.

¶*Global variables.*
DCOMPILE, DCOMPFUN.

¶*Store Used*
The program requires approximately 1. 5 blocks of store.

¶*Example of the use of dcompile*

```
COMPILE(LIBRARY([LIB DCOMPILE]));

: DCOMPILE(68, 50, 0, 0);
 1:
 2:  FUNCTION RECFACT N;
 3:      IF N=0 THEN 1 ELSE N*RECFACT(N-1) CLOSE
 4:  END;
 5:  RECFACT (4)=>

**   24,
 6: FUNCTION INFACT N;
 7: VARS TOT;   1 -> TOT;
 8: LOOP:
 9:   IF N=0 THEN TOT ELSE
10:   N*TOT -> TOT; N-1 -> N;
11:   GOTO LOOP;   END;                              (mistake made)

ERROR 22
IN FUNCTION INFACT
CULPRIT END
SETPOP:
:                                                    (restart at
: DCOMPILE(68, 50, 5, 10);                           beginning of
 6: FUNCTION INFACT N;                               function INFACT)
 7: VARS TOT;   1 -> TOT;
 8:   LOOP:                                          (lines 6 to 10 output)
 9:     IF N=0 THEN TOT ELSE
10:     N*TOT -> TOT; N-1 -> N;
11: GOTO LOOP;   CLOSE;                              (type in the
12: END;                                             correct lines)
13:                                                  (note-HALT typed)
'DISC FILE CLOSED.  NEXT FREE SECTOR IS' 51
:
:  INFACT(4)=>

**24,
[DCOMPILE]


VARS DCOMPFUN;
FUNCTION DCOMPILE A B C D;VARS V W X Y Z SS;
  FUNCTION ERRF Z SS;TERMIN.Z;SS->ERRFUN;.ERRFUN END;

  FUNCTION FRED1;VARS U;
    .A->U;U.B;U.Z;U
  END;

  FUNCTION FRED A B C D Z ZZ SS;VARS P;
    .ZZ->P;
    IF P=TERMIN THEN SS->ERRFUN; [DISC FILE CLOSED . NEXT SECTOR IS] .PR;
      Z.DSECTOR.PR;GOTO L1 CLOSE;
    IF P=17 THEN  (C+1).PR;C+1->FROZVAL(3,DCOMPFUN);
    IF NOT(A=CHARIN) THEN
      D-1->FROZVAL(4,DCOMPFUN);
      IF D=0 THEN CHARIN->FROZVAL(1,DCOMPFUN);ERASE->FROZVAL(2,DCOMPFUN)
             ELSE PR(":") CLOSE
      CLOSE
    CLOSE;
  L1: P
  END;

  ERRFUN->SS;
  A,B.DISCOUT->Z;ERRF(%Z,SS%)->ERRFUN;
  IF C=0 THEN 1->C;C.PR;L5: CHARIN->A;ERASE->B;GOTO L1 CLOSE;
  A,B.DISCIN->A;ERASE->B;
  D-C->D;C->V;
L3:V-1->V;IF V=0 THEN GOTO L2 CLOSE;
L4:IF .FRED1=17 THEN GOTO L3 CLOSE;
  GOTO L4;
L2: C.PR;IF D<0 THEN GOTO L5 CLOSE;
   CHAROUT->B;":" .PR;
L1:FRED(%A,B,C,D,Z,FRED1,SS%)->DCOMPFUN;DCOMPFUN.COMPILE;
END;

2.NL; 'DCOMPILE NOW READY FOR USE'.PR; 2.NL;
```

Program name. LIB DEBUG
Source. R. M. Burstall, R. D. Dunn, DMIP; *Date of issue.* December 1968.

¶*Description.* This program provides a powerful debugging aid for POP-2 programs by allowing the user to output the values of selected arguments and results of functions, when they are obeyed.

¶*How to use the debugging aids.* In the 4100 implementations of POP-2 these routines are implemented as standard functions. Please refer to the Functional Specification of the system you are using, if the debugging aids are not available as standard, then type:

```
COMPILE(LIBRARY([LIB DEBUG]));
```

The user first specifies any functions he may wish to have traced, by using the macro SPEC.
⟨specification⟩ ::=⟨formal parameter list⟩ ⟨output local list⟩;

Please refer to section 4.1 of the reference manual. Thus the specification of a function looks like the first line of a function definition, with the word FUNCTION replaced by the word SPEC.

For example, consider the function:

```
FUNCTION ADD X Y;
  X+Y
END;
```

Using the output local list facility in the language, this can be written as:

```
FUNCTION ADD X Y => RESULT;
  X+Y -> RESULT;
END;
```

To specify ADD one would do:

```
SPEC ADD X Y => RESULT;
```

Note. The function does *not* need to be written in the second form (using the output local list facility) in order that it may be specified.

Similarly, if the function F has four arguments, say A, B, C, and D, and produces three results, say X, Y, and Z, then F may be specified by doing:

```
SPEC F A B C D => X Y Z;
```

It is not necessary however to specify all parameters and results, just the last *m* parameters, and *n* results may be specified if desired ($m \geqslant 0, n \geqslant 0$). Also if, in the specification, a parameter or result is given as the item *, then that parameter or result will not be printed when tracing.

For example, if with function F above, it is required to trace the values of C, X, and Z only, then the specification would appear as:

```
SPEC F C * => X * Z;
```

SPEC may be applied any number of times, if it is required to change the specification.

Specifying a function has no effect on its running speed, and only uses a few words of store for each function, so that it can well be done after each function, or group of functions, when writing the program in the first place.

To cause a particular function, or functions, to be traced, the macro BUG should be used in the following way:

BUG ADD;	will cause the function ADD to be traced,
BUG ADD F1 F;	will cause the functions ADD, F1, and F, to be traced.
etc.	

From then on, tracing will occur on entry to, and on exit from, the bugged functions *provided that the variable DEBUG is set to TRUE.*

Thus the tracing can be controlled over all functions bugged, by setting the variable DEBUG to FALSE, for no output, or TRUE, for output.

The printout given while tracing is, on entry to the function, the name of the function on a new line, preceded by the symbol '>', and followed by the names of the selected arguments and their values. On exit from the function the name of the function is printed on a new line preceded by the symbol '<', and is followed by the names of the selected results and their values.

If a function which has not been specified with SPEC, is bugged, then the printout consists of the name of the function only.

If a doublet is BUGged then a doublet will be produced, however the update part of the doublet must be SPECced separately.

On each function entry the printout is indented by one space, and on exit the indentation is reduced by one space, the amount of indentation being held in the variable DEBSP. The user may change the value of this variable during a trace if a special lay-out is required. DEBSP is automatically re-set to zero if an error occurs, or if the user interrupts the process by pressing the bell character on the console, on all implementations which have these facilities as standard, otherwise DEBSP must be reset by the user.

The values of parameters and results are printed during tracing by the function DEBPR, which initially has the value PR. This function may be redefined by the user, if it is required to print special values, arrays, or records, for example. The definition given to DEBPR must be a function which takes one argument and leaves no results. If this is not done the result of the debugging will be undefined.

To stop tracing a particular function, or functions, the macro UNBUG should be used in a similar way to BUG, for example,

UNBUG ADD;	will cause the debugging mechanism to be removed from ADD,
UNBUG FF GETY;	removes the debugging mechanism from the functions FF and GETY.
etc.	

It should be noted that BUG and UNBUG affect the function stored in the function variable, and if the value of the function has to be taken out and used elsewhere, for example, by partially applying it, or tying it up in a data structure, they will not affect the incorporated function.

¶ *Global variables used.* DEBUG DEBSP SPEC BUGIO BUGIN BUGOUT BUG UNBUG.

¶ *Examples of the use of the debugging aids*
```
: FUNCTION ADD X Y;
:     X+Y
: END;
```

```
:
: SPEC ADD X Y => SUM;
:
: FUNCTION ADD3 U V W;
:    ADD(ADD(U, V), W)
: END;
:
: SPEC ADD3 V * => SUM3;
: BUG ADD ADD3;
:
: TRUE -> DEBUG;
:
: ADD3(1, 2, 3) =>
   >ADD3;  V= 2, *,
     >ADD;     X= 3, Y= 2,
     <ADD;     SUM= 5,
     >ADD;     X= 5, Y= 1,
     <ADD;     SUM= 6,
   <ADD3;  SUM3= 6,

   **  6,

: UNBUG ADD;
: ADD3(2, 3, 4) =>
   >ADD3;  V= 3, *,
   < ADD3; SUM3= 9,

   ** 9,

:
```

```
[DEBUG]


VARS DEBPR DEBSP DEBUG;

MACRO SPEC;
VARS FN X;
  ITEMREAD()->FN;
  MACRESULTS([% [%
    [% FN,
       [% (L0: ITEMREAD()->X;
            IF NOT(X=";") AND NOT(X="=>") THEN X GOTO L0 CLOSE) %],
       IF X=";" THEN NIL ELSE
       [% (L1: ITEMREAD()->X; IF NOT(X=";") THEN X GOTO L1 CLOSE) %]
       CLOSE %] %],
    "->",FN,".","FNPROPS",";" %])
END;

FUNCTION BUGIO F;
VARS X;
  NL(1); SP(DEBSP); PR(); FNPROPS()->X;
  IF NULL(X) THEN L0: PR(X); PR(";") EXIT;
  HD(X)->X;
  IF ATOM(X) THEN GOTO L0 CLOSE;
  PR(HD(X)); PR(";"); F(TL(X))->X; REV(MAPLIST(X,ERASE));
  APPLIST(X,
    LAMBDA U;
      NEXT()->X; SP(1); PR(U);
      IF NOT(U="*") THEN PR("="); ->U; DEBPR(U); U CLOSE;
      PR(","); X
    END); ->X
END;

FUNCTION DEBFN X;
  IF DEBUG THEN BUGIO(X,">",HD); DEBSP+1->DEBSP; X();
    DEBSP-1->DEBSP; BUGIO(X,"<",LAMBDA; HD(TL()) END)
  EXIT;
  X()
END;

MACRO BUG;
VARS X;
  MACRESULTS([%
(L0:
  ITEMREAD()->X;
  IF NOT(X=";") THEN "DEBFN","(%",X,"%)","->",X,";" GOTO L0 CLOSE) %])
END;

MACRO UNBUG;
VARS X;
  MACRESULTS([%
(L0:
  ITEMREAD()->X;
  IF NOT(X=";") THEN 1,",",X,".","FROZVAL","->",X,";" GOTO L0 CLOSE) %])
END;

PR->DEBPR; 0->DEBSP; FALSE->DEBUG;
```

Program name. LIB EASYFILE
Source. R. M. Burstall, R. H. Rae, A. P. Ambler, DMIP; *Date of issue.* February 1969.

¶*Description.* This program provides several basic commands to handle disc files by name (the name being any list) and enables one to:

1. Read a new file, typed on the console, to disc; (DREAD)
2. Read in a file to disc from paper tape; (DPTIN)
3. Read in a file to disc from a character repeater specified by the user; (DREPIN)
4. Have a disc file typed out at the console; (DTYPE)
5. Have a disc file listed on the line printer; (DLP80)
6. Have a disc file copied onto paper tape; (DPTOUT)
7. Obtain the character repeater of a disc file; (DIN)
8. Compile a disc file; (DCOMP)
9. Edit a disc file, using LIB POPEDIT and typing edit commands on the console; (DEDIT)
10. Recover the original version of a disc file which has just been edited; (DRECOVER)
11. Copy a disc file, giving it a new name; (DCOPY)
12. Discard a disc file no longer required; (DKILL).

The system currently uses a simple disc filing system and LIB POPEDIT. It is anticipated that other editing or filing systems may be substituted for these without changing the effect of the basic commands listed above.

¶*How to use the system.* Compile the program by typing:
COMPILE (LIBRARY ([LIB EASYFILE]));

When the compilation is complete 'TYPE TRACK NUMBER' is output on the console, and if this is *not* the first time LIB EASYFILE has been used with this track you should type the number of the disc track you wish to use. The facilities described below may then be used immediately (example 2). If this is the *first* time LIB EASYFILE has ever been used with this track you should type 0 instead as you still have to initialize it (see below and example 1).

¶*Initializing disc tracks.* The first time LIB EASYFILE is used with a disc track it must be initialized to the format required by the filing system. To do this type

DISCINIT ([⟨n1⟩ ⟨n2⟩. . .]);
where ⟨n1⟩, ⟨n2⟩, . . . are the numbers of the tracks to be initialized.

The system is now ready to accept and handle files in this and subsequent POP-2 sessions. The first 10 sectors of each track so initialized are used by the system for storing the track's file directory. The rest of the track will be used for your files. You may at any time add further tracks to your filing system by calling DISCINIT again.

¶*Changing tracks.* There is a separate directory for each track in use. You may change from one track to another by typing

DTRACK(⟨n⟩);
where ⟨n⟩ is the number of the track you wish to use (see example 3).

¶*Commands available.* In this description ⟨filename⟩ denotes a file name (for example, RMB THEOREMS or APA SYS 4).

A. *Commands which create new disc files.* DREAD([⟨filename⟩]); reads characters from the console until a halt-code is depressed (CTRL and T). The characters are put into a file (on disc) called ⟨filename⟩. If a file with this name already exists, it is lost when the halt-code is read from the console.

DPTIN([⟨filename⟩]); inputs the file ⟨filename⟩ from paper tape and creates a disc file with the same name.

DREPIN([⟨filename⟩],⟨charrep⟩); creates a disc file with the name ⟨filename⟩ from the character repeater ⟨charrep⟩. This command can be used to put files already on disc into the filing system. (For example, when you first decide to use LIB EASYFILE, and already have some files on disc which are indexed in some other way).

DCOPY([⟨filename1⟩], [⟨filename2⟩]); copies the file ⟨filename1⟩ to the file ⟨filename2⟩.

DEDIT([⟨filename⟩]); is a command which creates a new disc file. Its use is described below under D.

When the transfers involved in the above commands have been completed (that is, when a halt-code or TERMIN has been read) the message

[⟨filename⟩ COMPLETE DISCEND IS NOW⟨n⟩]

is output on the console. For the significance of DISCEND and ⟨n⟩ see below under HOUSEKEEPING. If a transfer to the disc is interrupted before it is terminated (for example, by pressing CTRL and G) then the new file is not entered in the directory.

B. *Commands to copy a disc file onto another device.* DTYPE ([⟨filename⟩]); outputs to the console the file ⟨filename⟩. To interrupt just depress CTRL and G.

DLP80([⟨filename⟩]); lists ⟨filename⟩ on the lineprinter.

DPTOUT([⟨filename⟩]); outputs ⟨filename⟩ onto paper tape.

C. *Commands for compiling disc files.* DCOMP([⟨filename⟩]); compiles the file⟨filename⟩. When the compilation has been completed the message
[⟨filename⟩ COMPILED]
is output on the console.

D. *Commands for editing disc files.* DEDIT([⟨filename⟩]); edits a file discarding the previous version. It expects edit commands from the keyboard in the format of POPEDIT (see library documentation of LIB POPEDIT). When the editing has been completed the message

[⟨filename⟩ COMPLETE DISCEND IS NOW ⟨n⟩]

is output on the console. For the significance of DISCEND see below under housekeeping. The original file is not lost until this message has been output and it is therefore possible to recover from an obviously wrong edit merely by interrupting it—depressing CTRL and G. (To recover *after* the message has been output DRECOVER will have to be used—see below.) In LIB POPEDIT when an edit ends not under the control of SH the comment 'END OF INPUT FILE. OUTPUT FILES NOT CLOSED' is output. This comment is not true when DEDIT is used—the output file *is* closed.

DRECOVER([⟨filename⟩]); puts back into the directory the version of filename which has just been displaced by editing. The edited version is discarded. DRECOVER can only be used successfully if there has been no intervening call of DEDIT, DTIDY (see below) or DTRACK. If there has been such a call, or if the file is wrongly named, then the message

[SORRY NOT POSSIBLE TO RECOVER ⟨filename⟩]
is output.

E. *Other commands to handle files.* DIN([⟨filename⟩]); has as result the input character repeater of the file ⟨filename⟩. DIN is analogous to DISCIN.

DKILL([⟨filename⟩]); discards the file ⟨filename⟩. The area of disc on which it was stored will become available for other files.

¶*Housekeeping: Commands peculiar to the present filing system.*
In the system used at present files are added to the disc track consecutively, and the sectors freed by editing or killing files are not automatically made available for re-use. To do this YOU must give the command:

.DTIDY;
which shuffles the files and makes the freed space available. (See example 6.) It is desirable to use this command whenever your track is in danger of becoming full. The variable DISCEND points to the next free sector on the track being used. Whenever a new file is added to the track the new value of DISCEND is printed so that you may know how full your track is. DTIDY also prints the new value of DISCEND. (There are 160 sectors on a disc track.)

The system uses a directory which stores the actual position and size (in sectors) of each file and free area.

DISCDIR =>
prints out this file directory. (See example 6.)

¶*Errors.* An attempt to call a file by name which is not a list causes ERROR 54. An attempt to access a file which is not in the directory produces ERROR 57. Overfilling the track causes ERROR 79.

¶*Suggestion for efficient use of the system.* It should be more convenient to keep a large program as a lot of short files with a master file to compile them (or list them). (See example 5.) Not only is editing much easier, but also the progress of compilation of a large program is indicated by the messages output as each subsidiary file is compiled. Also, if a large file is more than 80 sectors long it will not be possible to edit it, as two copies of the file cannot exist on the 160 sectors of the track.

¶*Global variables.* These are:
DISC DISCTIDY DISCOFF DISCSINK DISCTRAC
DISCINIT
DOFIND DOTO DDIO
DUSER DREAD DPTIN DREPIN DCOPY DOUT DIN
DTYPE DLP80 DPTOUT DKILL DCOMP DEDIT DFREE
DRECOVER DTIDY DTRACK

DISCDIR DISCEND DISCUSER
DREC1 DREC2

¶*Store required.* This program requires approximately $4\frac{1}{2}$ blocks of core.

¶*Examples of use.* In the examples which follow, a halt-code (depressing CTRL and T) is indicated by Ⓗ. Characters typed in by the user are underlined, to distinguish them from output from the machine.

Example 1. Initializing a track, reading a file in from the console, editing and compiling it.

```
: COMPILE(LIBRARY([LIB EASYFILE]));
'TYPE TRACK NUMBER': 0
: DISCINIT([94]);
: DREAD([APA 1]);
: THE CAT SAT ON THE MAT]. HD. PR;
: Ⓗ
[[APA 1] COMPLETE DISCEND IS NOW 11]
: DTYPE([APA 1]);
THE CAT SAT ON THE MAT]. HD. PR;
:  DEDIT([APA 1]);
'POPEDIT READY FOR USE'
'READY: TYPE EDIT COMMANDS'
: IS  [
: SH
'EDIT FINISHED. OUTPUT FILES CLOSED'
[[APA 1] COMPLETE DISCEND IS NOW 12]
: DCOMP([APA 1]);
THE
[[APA 1] COMPILED]:
```

Example 2. A subsequent POP-2 session. Reading in a file from disc. Use of DRECOVER.

```
:  COMPILE(LIBRARY([LIB EASYFILE]));
'TYPE TRACK NUMBER': 94
:  DREPIN([SYSTEM], DISCIN(94, 100));          (Bring file SYSTEM
[[SYSTEM] COMPLETE DISCEND IS NOW 38]          under the control of
:DEDIT([SYSTEM]);                              the filing system from
'POPEDIT READY FOR USE'                        track 94 sector 100 on
'READY: TYPE EDIT COMMANDS'                    the disc.)
                                               (Edit this file.)
:  FL THEN
:  DC X
:  SH
'EDIT FINISHED. OUTPUT FILES CLOSED'
[[SYSTEM] COMPLETE DISCEND IS NOW 45]
:  DCOMP([SYSTEM]);.                           (Attempt to compile
ERROR 22 IN FUNCTION SYS                        the new file SYSTEM
CULPRIT CLOSE                                   uncovers a syntax
SETPOP                                          error in it.)
:  DRECOVER([SYSTEM]);                         (Recover the old file
:  DCOMP ([SYSTEM]);                           SYSTEM.)
[ [SYSTEM] COMPILED]:                          (This file compiles
                                                correctly.)
```

Example 3. Changing tracks (supposing that 93 and 94 have been initialized at some earlier POP-2 session).

```
: COMPILE(LIBRARY([LIB EASYFILE]));
'TYPE TRACK NUMBER': 94
: DPTIN([APA 2]);
[[APA 2] COMPLETE DISCEND IS NOW 141]
: DTRACK (93);                               (Change the track on
: DPTIN([THEOREMS]);                          which the system is
[[THEOREMS] COMPLETE DISCEND IS NOW           currently operating
33]                                           from 94 to 93, and
:                                             read in a paper-tape
                                              file to that track).
```

Example 4 Use of DTIDY

```
: DPTIN([SYSTEM 2]);
[[SYSTEM 2] COMPLETE DISCEND IS NOW 152]
: .DTIDY;
DISCEND NOW 127
:
```

Example 5 Storing a large program as several small files. Assume that there are on the disc the files [APA SYS1], ... , [APA SYS6] and they together comprise a program [APA SYSTEM]. Make up a master file [APA SYSTEM] as follows:

```
: DREAD([APA SYSTEM]);                            (GENFUN is an arbi-
: GENFUN([APA SYS1]);   GENFUN([APA SYS2]);       trary name for a
: GENFUN([APA SYS3]);   GENFUN([APA SYS4]);       function which can be
: GENFUN([APA SYS5]);   GENFUN([APA SYS6]);       redefined so that the
Ⓗ                                                 file can perform
[[APA SYSTEM] COMPLETE DISCEND IS NOW             various actions.)
97]
:
```

if you wish to compile [APA SYSTEM] do:

```
: DCOMP —> GENFUN;                               (Define GENFUN to be
: DCOMP ([APA SYSTEM]);                           DCOMP. This will
[[APA SYS1] COMPILED]                             cause the various sub
[[APA SYS2] COMPILED]                             files to be compiled
                                                  when SYSTEM is com-
                                                  piled.)

     .
     .
     .
     .
[[APA SYS6] COMPILED]
[[APA SYSTEM] COMPILED]
:
```

if you wish to list [APA SYSTEM] on the line printer do:

```
: DLP80—>GENFUN;  DCOMP([APA SYSTEM]);  (GENFUN is defined
[[APA SYSTEM] COMPILED]                  to be DLP80, this
:                                        will cause a listing
                                         of the sub files.)
```

if you wish to copy [APA SYSTEM] to paper tape do:

```
: DPTOUT —> GENFUN;  DCOMP([APA SYSTEM]);
```
etc.

if you wish to edit [APA SYSTEM] you need only edit the relevant subsidiary file—say [APASYS4]. Since the new version of [APA SYS4] has the same name you do not need to alter your master file.

Example 6. The disc file directory and DTIDY. This example shows how the directory develops, starting from a newly-initialized track.

```
COMPILE(LIBRARY([LIB EASYFILE]));
'TYPE TRACK NUMBER' : 0
: DISCINIT([94]);
: DISCDIR =>

** NIL,
DPTIN([APA 1]);
[[APA 1] COMPLETE DISCEND IS NOW 15]
: DPTIN([APA 2]);
[[APA 2] COMPLETE DISCEND IS NOW 27]
: DISCDIR =>
** [[[APA 2] 15 12] [[APA 1] 10 5]],
: DPTIN([APA 1]);
[[APA 1] COMPLETE DISCEND IS NOW 32]
: DISCDIR =>
** [[[APA 1] 27 5] [[APA 2] 15 12] [FREE 10 5]],
: .DTIDY;
[DISCEND IS NOW 27]
: DISCDIR =>
** [[[APA 1] 21 5] [[APA 2] 10 12]],
:
```

ADDENDUM TO EASYFILE: THE DISC-FILING SYSTEM ON WHICH THE CURRENT LIB EASYFILE IS BASED

Source: R.H.Rae DMIP.

¶*Description.* The present MULTIPOP facilities for using the disc as backing store require the user to know the number of the sector at which each of his files begins. If many files are being held on disc this rapidly becomes cumbersome, and some sort of filing scheme becomes desirable. With [DISCFILE] the user associates a symbolic title with each of his files and all files are read to disc and accessed from disc by these symbolic titles. He need not know the file's actual position on the track.

Facilities are provided for reading a character repeater onto disc and giving the resultant file any desired title, for accessing any existing disc file by its title, for deleting any unwanted file, and for compacting all existing files as far down a track as possible, hence freeing as large a single block of disc track as possible for any future large file.

If the user has more than one track or has permission to read another user's files, there is a simple switch that enables him to use the same system for other tracks. However, each track must individually be under the control of this filing system.

Each track must have a certain area set aside for holding a directory and other executive information. This area is taken as the first ten sectors of the track, usually, leaving the full 150 others for holding files.

No check can be made by program to estimate whether there is enough room on the track to accommodate a new file before the file is read onto disc. The user must estimate this himself by examining the global variable DISCEND. Subtracting the value of DISCEND from 160 indicates the numbers of sectors remaining for the next file. If the user accidentally writes off the end of his track, error 79 will be called as normal and the offending file will not be added to the user's directory.

¶*Facilities offered.* (1) The facilities are available when LIB EASY-FILE has been compiled.

(2) To change the track being accessed
DISCTRACK(⟨tracknumber⟩);
where ⟨tracknumber⟩ is the number of the track the user wants to change to.

(3) To read a file onto the disc:
⟨repeater⟩ –> DISC(⟨title⟩);

where ⟨repeater⟩ is the character repeater function defining the new file. The file is closed as soon as ⟨repeater⟩ produces TERMIN.

⟨title⟩ is the name to be associated with this file. It must be a list. Any existing file already with that title will be deleted.

(4) To access a disc file:
DISC(⟨title⟩);
where ⟨title⟩ is the name, defined when the file was read onto the disc, of the file required. It must be a list.
The result of a call of DISC will be a character repeater.
Often it is more convenient to create a file character by character rather than all at once. This can be done by using:

DISCSINK(⟨title⟩);
where ⟨title⟩ is the name the completed file will have.
The result is a character sink.
The file is only created when the final TERMIN is output.

Example. Consider the user wanting to create a file, [FRED], from the teletype. Then he could use:

```
: VARS DOUT;
: DISCSINK([FRED]) –> DOUT;
:
: FUNCTION GOBLUP; VARS CHAR;
: LQ: .CHARIN –> CHAR;   CHAR . DOUT;
: IF NOT (CHAR=TERMIN) THEN GOTO LQ
: CLOSE
: END
: GOBLUP();
:
```

This will read characters from the teletype up to, and including a TERMIN.
The result would be identical to calling:

```
: CHARIN –> DISC([FRED]);
:
```

(5) To delete an existing file:
DISCOFF(⟨title⟩);
where ⟨title⟩ is the name of the file to be deleted. This function produces no result.

(6) If files have been deleted, by calling DISCOFF or by using a name a second time, there will be unused sectors between the files on the track in question. We want as large a contiguous block of the track as possible for reading future files onto, so we want to compact the existing files as far down the track as possible, filling up the gaps left by deletion.

This is done by calling:
DISCTIDY();
This function takes no arguments and produces no results.
It may take an appreciable length of time to be completed.
A system failure, logging off, or exiting from DISCTIDY in any other way before it has been completed, may prove fatal.
DISCTIDY should be called when, but only when, it is needed.

(7) Before using the system for the first time, each of the user's tracks must be initialized. This is done by:

DISCINIT(⟨user track list⟩);
where ⟨user track list⟩ is a list of the track numbers of each of the user's tracks. The ordering is irrelevant.

¶*Errors.* If a title that is not a list is ever the argument of DISC, error 54 will be called.
If the user attempts to access, via DISC, a file that is not on his current track, error 57 will be called.

If the user writes off the end of his track, or tries to write a file onto someone else's track, the usual disc error messages will appear.

¶*Identifiers and variables used.* All identifiers and variables used begin either DISC... or DD... All identifiers and variables used have four or more characters in their name.

(a) Global identifiers all begin DISC... They are:

DISC
    The function used to read files to and from disc.
DISCSINK
    The function used to create a character sink.
DISCOFF
    The function used to delete a file from the user's directory.
DISCTIDY
    The function used to compact the user's files.
DISCTRACK
    The function used when a change of track is wanted.
DISCINIT
    The function used to initialize the system for a user.
DISCDIR
    This variable holds the user's current directory.
DISCEND
    This variable holds the number of the sector which will be the start of the next file added to the directory.
DISCUSER
    This variable holds the number of the track currently in use.

None of these identifiers should be the destination of any assignment.

(b) Local variables all begin DD... :
DDEND, DDFUN, DDMP, DDG1 — DDG6 are all of type general.
DDFIND, DDTO, DDCOMP, DDF1, DDF2 are all of type function.
These identifiers can be used as the user wishes.

¶ *The directory.* Files are accessed via a directory. This directory is a list that is held both in store and on disc. The copy in store is the current version of the user's directory and the disc version is just a copy of this. The store version is dumped to disc whenever a new file is added or whenever DISCTIDY has been called. Except during a call of DISCTIDY, the version in store is always a correct picture of the positioning of the user's files. This store version is held in the global variable DISCDIR. If, however, a user attempts to output a file to a track which is not his own, his store version of DISCDIR for that track will get corrupted. Recovery is possible by doing a DTRACK for that track. The directory is NIL or a list of file entries.

A file entry is a list of three items held as:
[⟨filetitle⟩ ⟨filestart⟩ ⟨filelength⟩]
A ⟨filetitle⟩ is "FREE" or a list.
A ⟨filestart⟩ is UNDEF or an integer not less than 10.
A ⟨filelength⟩ is a non-negative integer.
If ⟨filestart⟩ of a file entry is UNDEF, the ⟨filelength⟩ must be 0.
The directory must be ordered properly, and must contain no gaps.
Roughly, the conditions are:
If FE(N) is the nth file entry, and the directory contains D entries, D>0,

```
        filestart(FE(1))+filelength(FE(1))=DISCEND,
        filestart(FE(D))=10,
        filestart(FE(N+1))+filelength(FE(N+1))=filestart(FE(N))
unless  filestart(FE(N+1))=UNDEF
when    filelength(FE(N+1))=0.
```

A typical directory could look like:

```
[[[FRED] 42 29][FREE 17 25][[BILL] 15 2][[CHARLIE] UNDEF 0]
[[EDITOR] 10 5]]
```

¶ *Example of the use of the basic facilities.* The user, with track 89, has a paper tape file, [EXAMPLE], that he wants to read onto disc. Having done this he compiles the disc file, and finds he has made some mistakes. He compiles POPEDIT, examines the faulty disc file, edits it, checks the edited file, and then compiles it. As the compilation is OK, he changes the name of the file, deletes the redundant file, calls DISCTIDY to clear up his track, and logs off.

```
: COMPILE(LIBRARY([LIB EASYFILE]));
: 'TYPE TRACK NUMBER': 89
:
: COMMENT READ THE PAPER TAPE FILE TO DISC;
: POPMESS([PTIN EXAMPLE])->DISC([EXAMPLE]);
:
: COMMENT COMPILE THE RESULTING DISC FILE;
: COMPILE(DISC([EXAMPLE]));

COMMENTS [A]

ERROR 47
IN FUNCTION FACT
CULPRIT A
SETPOP:
:
: COMMENT COMPILE POPEDIT41;
: COMPILE(LIBRARY([LIB POPEDIT]));
:
: COMMENT PRINT OUT THE OFFENDING DISC FILE;
: DTYPE([EXAMPLE]);
```

```
FUNCTION SIGMA L;
   IF L.NULL THEN 0 ELSE L.HD+SIGMA(TL(L))
   CLOSE
END;

FUNCTION FACT N; N*FACT(N—1);
END;

"A"—>A;
1,2,3—>L;
FACT(A)=>
SIGMA(LIST)=>

: COMMENT EDIT THE DISC FILE;
: POPEDIT(DISC([EXAMPLE], CHARIN, NIL)—>DISC([EDITED
  EXAMPLE]);

[READY : TYPE EDIT COMMANDS]
:
:   FL END
:   FE
:   FC ;
:   IL  IF N=0 THEN 1 ELSE
:   FC)
:   IS  CLOSE
:   FL  END
:   FE
:   IB
:
:   VARS A LIST;
:   2—>A; [%1, 2, 3%]—>LIST;
:
: ↑
: DL 1
: DE
: SH
:
: =>

**
: COMMENT HAVING DONE THE EDIT, PRINT OUT THE EDITED
  FILE;
: DTYPE([EDITED FILE]);
ERROR  57
CULPRIT [EDITED FILE]
SETPOP:
:
: COMMENT MADE A MISTAKE. REALLY WANTED [EDITED
  EXAMPLE]. TRY AGAIN;
: DTYPE([EDITED EXAMPLE]);

FUNCTION SIGMA L;
   IF L.NULL THEN 0 ELSE L.HD+SIGMA(TL(L))
   CLOSE
END;

FUNCTION FACT N;
   IF N=0 THEN 1 ELSE N*FACT(N—1) CLOSE;
END;
```

```
VARS A LIST;
 2->A; [%1, 2, 3%]->LIST;

FACT(A)=>
SIGMA(LIST)=>

: COMMENT AS IT LOOKS O.K. TRY COMPILING IT;
: COMPILE(DISC([EDITED EXAMPLE]));

** 2,

** 6,
:
: COMMENT THIS IS ALRIGHT, BUT THE FILE IS TO BE CALLED
  [EXAMPLE];
: DISC([EDITED EXAMPLE])->DISC([EXAMPLE]);
:
: COMMENT AS FILES HAVE BEEN DELETED, CALL DISCTIDY TO
  CLEAN UP THE TRACK;
: DISCTIDY();
:
: LOGOFF();
```

```
[EASYFILE]
VARS DISCDIR DISCEND DISCUSER DDEND DDMP
DISC DISCTIDY DISCOFF DISCSINK;
FUNCTION DDFIND DDG1;VARS DDG2;
 IF DDG1.ATOM THEN DDG1,54.ERRFUN EXIT;DISCDIR->DDG2;
L:IF DDG2.NULL THEN[%DDG1,UNDEF,0%]::DISCDIR->DISCDIR;DISCDIR.HD EXIT;
 IF DDG2.HD.HD.ATOM.NOT THEN IF EQUAL(DDG1,DDG2.HD.HD) THEN DDG2.HD EXIT
  ELSEIF NOT(DDG2.HD.HD="FREE")THEN DDG2.HD.HD;54.ERRFUN
 EXIT;DDG2.TL->DDG2;GOTO L
END
FUNCTION DDTO DDF1 DDG1;
VARS DDG2 DDF2;DISCUSER,DDG1.DISCOUT->DDF2;
L:.DDF1->DDG2;DDG2.DDF2;IF DDG2=TERMIN THEN DDF2.DSECTOR ELSE GOTO L CLOSE
END
LAMBDA DDG1 DDG2;DDG1.DDG2->DDG1;
 IF NOT(DDG1.TL.HD=UNDEF)THEN DISCUSER,DDG1.TL.HD.DISCIN
  ELSE DDG1.HD;57.ERRFUN
 EXIT
END(%DDFIND%)->DISC;
LAMBDA DDF1;VARS DDG1;CUCHAROUT->DDG1;DISCUSER,0.DISCOUT->CUCHAROUT;
"DISCDIR".DDF1;"DISCEND".DDF1;"DISCUSER".DDF1;TERMIN.CUCHAROUT;DDG1->CUCHAROUT
END(%LAMBDA DDG1;DDG1.VALOF.PR;"->".PR;DDG1.PR;";".PR;1.NL END%)->DDMP;
LAMBDA CUSTART DISCDUMP DISCWRITE;
VARS NEWSTART CUDIR SAVEDIR FILE1 FILE2 NAME START;
 REV(DISCDIR)->CUDIR; CUDIR->SAVEDIR;
L0:
 IF NULL(CUDIR) THEN
 ELSE HD(CUDIR)->FILE1;
IF HD(FILE1)="FREE" OR HD(TL(FILE1))=UNDEF THEN "FREE"->FILE1.HD; CUSTART->FILE1
.TL.HD;
L1: IF NULL(TL(CUDIR)) THEN TL(REV(SAVEDIR))->DISCDIR;DISCEND-HD(TL(TL(FILE1)))-
>DISCEND
ELSE HD(TL(CUDIR))->FILE2; HD(FILE2)->NAME; HD(TL(FILE2))->START;
IF NAME="FREE" OR START=UNDEF THEN
FILE1.TL.TL.HD+FILE2.TL.TL.HD->FILE1.TL.TL.HD; TL(TL(CUDIR))->TL(CUDIR); GOTO L1

ELSE "TIDY1"->HD(FILE1);
DISCWRITE(DISCIN(DISCUSER,START),CUSTART)->NEWSTART;
"TIDY2"->HD(FILE2); CUSTART->HD(TL(FILE1));
NEWSTART-CUSTART->HD(TL(TL(FILE1))); NAME->HD(FILE1);
START+HD(TL(TL(FILE2)))-NEWSTART->HD(TL(TL(FILE2)));
"FREE"->HD(FILE2); REV(SAVEDIR)->DISCDIR; DISCDUMP();
NEWSTART->CUSTART; FILE2->FILE1; TL(CUDIR)->CUDIR; GOTO L1
CLOSE CLOSE
    ELSE TL(CUDIR)->CUDIR; CUSTART+HD(TL(TL(FILE1)))->CUSTART; GOTO L0
    CLOSE
 CLOSE; DISCDUMP()
END(% 10,DDMP,DDTO %)->DISCTIDY;
LAMBDA DDG1 DDG2 DDG3;"FREE"->DDG1.DDG2.HD;.DDG3 END(%DDFIND,DDMP%)->DISCOFF;
LAMBDA DDG1 DDG2 DDG3 DDG4 DDG5;VARS DDG6;
L:DDG2.DDG3->DDG6;IF NOT(DDG6.TL.HD=UNDEF)THEN DDG2.DDG4;GOTO L CLOSE;
DISCEND->DDG6.TL.HD;DDG1-DISCEND->DDG6.TL.TL.HD;DDG1->DISCEND;.DDG5
END(%DDFIND,DISCOFF,DDMP%)->DDEND;
LAMBDA DDG1 DDG2=>DDG3;IF DDG1.ATOM THEN DDG1,54.ERRFUN EXIT;
 LAMBDA DDG1 DDG2 DDG3 DDG4;
  IF DDG2.NOT THEN DISCUSER,DISCEND.DISCOUT->DDG2;DDG2->FROZVAL(1,DDG3)
  CLOSE;DDG1.DDG2;
  IF DDG1=TERMIN THEN DDG2.DSECTOR.DDG4
  CLOSE
 END(%0,0,DDG2(%DDG1%)%)->DDG3;DDG3->FROZVAL(2,DDG3)
END(%DDEND%)->DISCSINK;
LAMBDA DDG1 DDG2 DDG3 DDG4;
IF DDG2.ATOM THEN DDG2,54.ERRFUN EXIT;DDG1,DISCEND.DDG3;DDG2.DDG4;
  NL(1);PR([%DDG2,"COMPLETE","DISCEND","IS","NOW",DISCEND%]);NL(1);
END(%DDTO,DDEND%)->DISC.UPDATER;
FUNCTION DISCTRAC;0.DISCIN.COMPILE END
FUNCTION DISCINIT DDG1;NIL->DISCDIR;10->DISCEND;L:IF DDG1.NULL THEN EXIT;
DDG1.HD->DISCUSER;DISCUSER,TERMIN,DDG1.NEXT->DDG1;0.DISCOUT.APPLY.DISCTRAC
.DISCTIDY;GOTO L
END;
VARS DREC1 DREC2 POPEDIT;

FUNCTION DREAD FILE;CHARIN->DISC(FILE);END;
FUNCTION DPTIN FILE;POPMESS([PTIN 20]::FILE)->DISC(FILE);END;
FUNCTION DREPIN FILE INF; INF->DISC(FILE);END;
FUNCTION DCOPY FILE1 FILE2; DISC(FILE1)->DISC(FILE2);END;
FUNCTION DOUT FILE;VARS OUTF;DISCSINK(FILE)->OUTF;
  LAMBDA U OUTF; OUTF(U); IF U=TERMIN THEN NL(1);
      PR([%HD(FILE),"COMPLETE","DISCEND","IS","NOW",DISCEND%]);NL(1);
  CLOSE END(%OUTF%);
END;
```

```
FUNCTION DDIO INF OUTF;VARS U;
  L:INF()->U;OUTF(U);IF U=TERMIN THEN EXIT;GOTO L; END;
FUNCTION DTYPE FILE; DDIO(DISC(FILE),CHAROUT);END;
 FUNCTION DLP80 FILE; DDIO(DISC(FILE),POPMESS([LP80 20]::FILE));END;
FUNCTION DPTOUT FILE;DDIO(DISC(FILE),POPMESS([PTOUT 20]::FILE));END;
VARS FUNCTION (DIN DKILL); DISC->DIN; DISCOFF->DKILL;
FUNCTION DCOMP FILE; FILE.DISC.COMPILE;NL(1);PR([%FILE,"COMPILED"%]);END;
FUNCTION DEDIT FILE;HD(DDFIND(FILE))->DREC1;HD(TL(DDFIND(FILE)))->DREC2;
  IF NOT(EQUAL(FNPROPS(POPEDIT),[POPEDIT])) THEN COMPILE(LIBRARY([LIB POPEDIT]))
; CLOSE;
  POPEDIT(DISC(FILE),CHARIN,NIL)->DISC(FILE);NL(1);
END;
FUNCTION DFREE N; VARS DDIR DDG;DISCDIR->DDIR;
L: IF DDIR.NULL THEN 'FILES IN A MESS'.PR;.SETPOP;EXIT;
NEXT(DDIR)->DDIR->DDG;
    IF HD(DDG)="FREE" AND HD(TL(DDG))=N THEN DDG ELSE GOTO L;
  CLOSE
END;
FUNCTION DRECOVER FILE; VARS CFILE;
  DDFIND(FILE)->CFILE; IF HD(TL(CFILE))=UNDEF THEN  TL(DISCDIR)->DISCDIR; FALSE-
>CFILE; CLOSE;
  IF NOT(ATOM(DREC1))THEN IF EQUAL(DREC1,FILE) THEN FILE->HD(DFREE(DREC2));
IF CFILE THEN "FREE"->HD(CFILE); CLOSE;
  EXIT; CLOSE;
  NL(1); PR([SORRY NOT  POSSIBLE TO RECOVER]<>[%FILE%]);NL(1);
END;
FUNCTION DTIDY; .DISCTIDY;UNDEF->DREC1; UNDEF->DREC2;
  PR([%"DISCEND","NOW",DISCEND%]);NL(1);
END;
FUNCTION DTRACK N; N->DISCUSER; DISCUSER.DISCTRAC;UNDEF->DREC1; UNDEF->DREC2; EN
D;
LAMBDA;VARS U; 1.NL;'TYPE TRACK NUMBER'.PR; CHARIN.INCHARITEM.APPLY->U;
IF U THEN U->DISCUSER; DISCUSER.DISCTRACK CLOSE END.APPLY;
```

```
[READABLE DISCFILE]
```
(A more readable version of the Discfile part of Easyfile)

```
VARS DISCDIR DISCEND DISCUSER MAKEFILE UPDEXEC
  DISC DISCTIDY DISCOFF DISCSINK;

FUNCTION FINDFILE FILENAME;
VARS SAVEDIR CUFILE;
 IF ATOM(FILENAME) THEN ERRFUN(FILENAME,54)
 ELSE DISCDIR->SAVEDIR;
LOOP:
    IF NULL(SAVEDIR) THEN [% FILENAME,UNDEF,0 %]::DISCDIR->DISCDIR; HD(DISCDIR)
    ELSE NEXT(SAVEDIR)->SAVEDIR->CUFILE;
       IF NOT(ATOM(HD(CUFILE))) THEN
          IF EQUAL(FILENAME,HD(CUFILE)) THEN CUFILE
          EXIT
       ELSEIF NOT(HD(CUFILE)="FREE") THEN ERRFUN(HD(CUFILE),54)
       EXIT;
       GOTO LOOP
    CLOSE
 CLOSE
END;


FUNCTION TODISC REPEATER STARTSECTOR;
VARS CUCHAR DREPEATER;
 DISCOUT(DISCUSER,STARTSECTOR)->DREPEATER;
LOOP:
 REPEATER()->CUCHAR; DREPEATER(CUCHAR);
 IF CUCHAR=TERMIN THEN DSECTOR(DREPEATER)
 ELSE GOTO LOOP
 CLOSE
END;

LAMBDA FILENAME FILEFIND;
 FILEFIND(FILENAME)->FILENAME;
 IF NOT(HD(TL(FILENAME))=UNDEF) THEN DISCIN(DISCUSER,HD(TL(FILENAME)))
 ELSE ERRFUN(HD(FILENAME),57)
 CLOSE
END(% FINDFILE %)->DISC;
```

```
 LAMBDA DWRITER;
 VARS SAVECUCHAROUT;
  CUCHAROUT->SAVECUCHAROUT;
  DISCOUT(DISCUSER,0)->CUCHAROUT;
  DWRITER("DISCDIR"); DWRITER("DISCEND"); DWRITER("DISCUSER");
  CUCHAROUT(TERMIN); SAVECUCHAROUT->CUCHAROUT
 END(% LAMBDA WORD;
        PR(VALOF(WORD)); PR("->"); PR(WORD); PR(";"); NL(1)
       END %)->UPDEXEC;

 LAMBDA FILENAME FILEFIND UPDEXEC;
  "FREE"->HD(FILEFIND(FILENAME)); UPDEXEC()
 END(% FINDFILE,UPDEXEC %)->DISCOFF;

 LAMBDA ENDSECTOR FILENAME FILEFIND UPDEXEC;
 VARS CUFILE;
 LOOP:
  FILEFIND(FILENAME)->CUFILE;


 IF HD(TL(CUFILE))=UNDEF THEN DISCEND->HD(TL(CUFILE));
    ENDSECTOR-DISCEND->HD(TL(TL(CUFILE))); ENDSECTOR->DISCEND; UPDEXEC()
 ELSE DISCOFF(FILENAME); GOTO LOOP
 CLOSE
END(% FINDFILE,UPDEXEC %)->MAKEFILE;

 FUNCTION DISCTRACK;
  COMPILE(DISCIN(0))
 END;

 LAMBDA FILENAME MAKEFILE;
  IF ATOM(FILENAME) THEN ERRFUN(FILENAME,54)
  ELSE
     LAMBDA CHAR DREPEAT TRACKNO FILEMAKER;
      DREPEAT(CHAR);
      IF CHAR=TERMIN THEN
         IF NOT(TRACKNO=DISCUSER) THEN DISCUSER; DISCTRACK(TRACKNO); ->TRACKNO
         CLOSE;
         FILEMAKER(DSECTOR(DREPEAT));
         IF NOT(TRACKNO=DISCUSER) THEN DISCTRACK(TRACKNO)
         CLOSE
      CLOSE
     END(% DISCOUT(DISCUSER,DISCEND),DISCUSER,MAKEFILE(% FILENAME %) %)
  CLOSE
 END(% MAKEFILE %)->DISCSINK;

 LAMBDA REPEAT FILENAME ONTODISC MAKEFILE;
  IF ATOM(FILENAME) THEN ERRFUN(FILENAME,54)
  ELSE MAKEFILE(ONTODISC(REPEAT,DISCEND),FILENAME)
  CLOSE
 END(% TODISC,MAKEFILE %)->UPDATER(DISC);

LAMBDA CUSTART DISCDUMP DISCWRITE;
VARS NEWSTART CUDIR SAVEDIR FILE1 FILE2 NAME START;
 REV(DISCDIR)->CUDIR; CUDIR->SAVEDIR;
L0:
 IF NULL(CUDIR) THEN
 ELSE HD(CUDIR)->FILE1;
    IF HD(FILE1)="FREE" OR HD(TL(FILE1))=UNDEF THEN "FREE"->HD(FILE1); CUSTART->
HD(TL(FILE1));
L1:     IF NULL(TL(CUDIR)) THEN TL(REV(SAVEDIR))->DISCDIR;
           DISCEND-HD(TL(TL(FILE1)))->DISCEND
        ELSE HD(TL(CUDIR))->FILE2; HD(FILE2)->NAME; HD(TL(FILE2))->START;
           IF NAME="FREE" OR START=UNDEF THEN HD(TL(TL(FILE1)))+HD(TL(TL(FILE2)))
->HD(TL(TL(FILE1)));
              TL(TL(CUDIR))->TL(CUDIR); GOTO L1
           ELSE "TIDY1"->HD(FILE1);
              DISCWRITE(DISCIN(DISCUSER,START),CUSTART)->NEWSTART;
              "TIDY2"->HD(FILE2); CUSTART->HD(TL(FILE1));
              NEWSTART-CUSTART->HD(TL(TL(FILE1))); NAME->HD(FILE1);
              START+HD(TL(TL(FILE2)))-NEWSTART->HD(TL(TL(FILE2)));
              "FREE"->HD(FILE2); REV(SAVEDIR)->DISCDIR; DISCDUMP();
              NEWSTART->CUSTART; FILE2->FILE1; TL(CUDIR)->CUDIR; GOTO L1
           CLOSE
        CLOSE
    ELSE TL(CUDIR)->CUDIR; CUSTART+HD(TL(TL(FILE1)))->CUSTART; GOTO L0
    CLOSE
 CLOSE; DISCDUMP()
END(% 10,UPDEXEC,TODISC %)->DISCTIDY;
```

```
;

; UPDEXEC()

ACKNO); ->TRACKNO

NAME %) %)

D(FILE1); CUSTART->

2))->START;
)+HD(TL(TL(FILE2)))

ART;

LE1);
)));
MP();
R; GOTO L1

; GOTO L0
```

```
FUNCTION DISCINIT TRACKLIST;
 NIL->DISCDIR; 10->DISCEND;
LOOP:
 IF NULL(DISCDIR) THEN RETURN
 ELSE HD(TRACKLIST)->DISCUSER;
    DISCUSER,TERMIN,TRACKLIST.NEXT->TRACKLIST;
    0.DISCOUT.APPLY.DISCTRACK.DISCTIDY;
    GOTO LOOP
 CLOSE
END;

COMMENT
'WHEN [DISCFILE] IS COMPILED, THE NUMBER OF THE FIRST TRACK THE USER
 WANTS TO ACCESS SHOULD BE ON THE TOP OF THE STACK. IF THIS IS NOT SO,
 AN ERROR EXIT WILL OCCUR AT THIS POINT IN THE PROGRAM. THE PROGRAM WILL
 BE FULLY USABLE, BUT "DISCUSER" WILL BE UNDEFINED';

->DISCUSER; DISCTRACK(DISCUSER);
```

Program name. LIB EQUATIONS
Source. R. J. Popplestone, DMIP; *Date of issue.* June 1969.

¶*Description.* This program provides a simple facility for checking the manipulation of algebraic expressions. In particular it will check the solution of a polynomial of one variable.

The user types in an algebraic expression which the program will store. This may be any POP-2 arithmetic expression containing X as its free variable, A, B, C, and D as its constants, and any arithmetic functions or operations. The expression must be terminated by a semi-colon.

```
e.g.    (X+A)*B = D/3 ;
        4+D+C = X*X–3*X ;
        X–A*B+5 ;
```

(If any two signs, e.g., +, *, etc., including =, are adjacent, they must be separated by spaces.)

The user can then manipulate this expression and the program will state whether he has done so correctly or not.

¶*How to use the program.* The program is compiled by typing:
COMPILE(LIBRARY([LIB EQUATIONS]));
and is entered by the function EQUATION.

On entry the message 'TYPE EQUATION' is output, and the user must type SOLVE followed by an algebraic expression of the type given above. The program now outputs 'TYPE NEXT LINE', and any reduction or expansion of the initial expression may be input. If it is consistent with the previous one, the message 'OK. TYPE NEXT LINE' is output and the user may continue. If however no relationship exists between them, the message 'MISTAKE. TRY AGAIN' is output and the correct version should be input.

The manipulation of any expression may be abandoned at any time either by typing FINISH, which causes an exit from the program, or by typing SOLVE followed by a new expression.

All legal POP-2 arithmetic operations may be used, and two expressions separated by OR may be typed if more than one solution exists. If a non-legal POP-2 expression is given, the message 'I DO NOT UNDERSTAND' is output, and the user should re-input the expression correctly.

¶*Method used.* The program redefines = and OR to be subtraction and multiplication respectively. Any constants are given an arbitrary value and the resulting expression is treated as a lambda expression of one argument X. The function ZERO finds a solution to the equation, and the original expression is applied to this solution to give a result which should not be appreciably different from Ø. If it is, then the expression is incorrect. If any expression is meaningless, ERRFUN is called, and due to its redefinition by the program, it causes the message 'I DO NOT UNDERSTAND' to be output and returns the user to the point at which he may type in a new expression.

¶*Notes*
1. OR and = are redefined in the program and on premature exit from the function EQUATION need to be redefined (for example, after control and G) by typing:

```
EQS–>NONOP = ;
ORR–>NONOP OR ;
```

Subsequently = and OR are no longer syntax words and can be redefined at will.

2. Certain classes of legal expressions will cause errors in the program with 'I DO NOT UNDERSTAND' being output. This is usually due to arithmetic overflow in function ZERO, caused by an attempt to deal with an equation with no real roots.

¶*Global variables.*
ZERON EQS ORR A B C D IDZERO ERF ABS EQUATION
ZERO STDR STATREAD EQCHECK JUMPEND.

¶*Store used.* The program uses one block of store.

¶*Example of use*

```
: COMPILE(LIBRARY([LIB EQUATIONS]));
'LIB EQUATION IS READY FOR USE'
: .EQUATION;
'TYPE EQUATION': SOLVE A*(B+(1+X)↑(—1)) = D;         To manipulate
'TYPE NEXT LINE': (1+X)*A*B+A=D*(1+X) ;                    a simple
'OK TYPE NEXT LINE': (1+X)(A*B—D)=(—A) ;                  algebraic
'I DO NOT UNDERSTAND': (1+X)*(A*B—D)=(—A);              expression.
'OK TYPE NEXT LINE': 1+X = A/(D—A*B);
'OK TYPE NEXT LINE': X=A/(D—A*B) —1 ;
'OK TYPE NEXT LINE': SOLVE X↑2+3*X+2=Ø ;                    To solve
'TYPE NEXT LINE': X=((—3)+SQRT(9—4*2))/2;                          a
'OK TYPE NEXT LINE':X=(—1.5) +SQRT(1)/2;                   quadratic
'OK TYPE NEXT LINE': X=1—1.5;                               equation.
'MISTAKE. TRY AGAIN': X=Ø.5—1.5;
'OK TYPE NEXT LINE': X=1 ;
'MISTAKE. TRY AGAIN': X+1=Ø ;
'OK TYPE NEXT LINE': X= —2;
'OK TYPE NEXT LINE': X= —2 OR X+1=Ø ;
'OK TYPE NEXT LINE': FINISH ;


'EXIT EQUATIONS'
:
```

```
[EQUATION]


VARS  ZERON EQS EPS  ORR;
  0.01 -> EPS; NONOP OR -> ORR ;NONOP = -> EQS;

CANCEL = OR ;
VARS OPERATION 6 =  OPERATION 7 OR  ;
  ORR->NONOP OR; EQS -> NONOP = ;

VARS IDZERO  ERF  ABS EQUATION ZERO STRD STATREAD
  JUMPEND ORIGINAL ;

ERRFUN->ERF;

FUNCTION EQCHECK X; VARS NEWEQ ;
  JUMPOUT(LAMBDA F N Z ;1.NL;PR('I DO NOT UNDERSTAND');
  Z->ERRFUN;END(% ERF %),0)->ERRFUN;
  IF EQ(X.HD,"FINISH") THEN EQS->NONOP = ;ERF->ERRFUN;
    ORR-> NONOP OR ;2.NL;PR('EXIT EQUATIONS');1.NL;JUMPEND();
  CLOSE;
  IF EQ(X.HD,"SOLVE") THEN X.TL->X;1.NL;PR(' TYPE NEXT LINE'); TRUE
    ELSE FALSE
  CLOSE->NEWEQ ;
  VARS FUN;POPVAL([LAMBDA X;]<>X<>[END ; GOON])->FUN;
  IF NEWEQ THEN FUN->ORIGINAL;EXIT;1.NL;
  IF NOT(IDZERO(FUN)) THEN
    IF ABS(ORIGINAL(ZERO(FUN)))>=2*EPS
    THEN ' MISTAKE.TRY AGAIN'.PR;EXIT;
  CLOSE;
  'OK TYPE NEXT LINE'.PR;
END;


FUNCTION EQUATION; VARS X ORIGINAL ;
  JUMPOUT(LAMBDA ; END,0)->JUMPEND;
  NONOP - -> NONOP = ; NONOP * -> NONOP OR ;
  1.NL;' TYPE EQUATION'.PR;
 L:.STRD->X;EQCHECK(X);GOTO L;
END;

FUNCTION IDZERO F;
  IF  EQ( F(0.2),0) AND  EQ(0, F(0.4)) AND
    EQ(0, F(0.6)) AND  EQ(0, F(0.8)) THEN TRUE ELSE FALSE;
  CLOSE;
END;



VARS A B C D;
    1.5 -> A;  2.5 -> B;  3.5 -> C;  4.5 -> D;


FUNCTION ABS X;
    IF X < 0 THEN -X ELSE X CLOSE
END

FUNCTION STATREAD;  .STRD END


FUNCTION STRD;
    VARS V;  .ITEMREAD -> V;  IF  EQS( V, ";") THEN NIL EXIT
    V :: STRD()
END
```

```
VARS ZERON;  10 -> ZERON;



FUNCTION ZERO F;
  VARS U1 U2 N;  ZERON -> N;  -1 -> U1;  +1 -> U2;

L0: IF  EQS( N, 0) THEN UNDEF EXIT;
    IF ABS(U1-U2) < EPS THEN U1 EXIT
    IF EQ( F(U2),F(U1)) THEN  (U1+U2)/2
      ELSE (U1 * F(U2) - U2 * F(U1) )/ (F(U2) - F(U1))
    CLOSE;,  U1 -> U2; -> U1;
    N - 1 -> N;  GOTO L0
END;

2.NL; 'EQUATIONS READY FOR USE'.PR; 2.NL;
```

Program name. LIB FOR *Source.* R. J. Popplestone, DMIP.; *Date of issue.* June 1969

¶Description
This package provides FOR statements in POP-2, their syntax being :
⟨for spec⟩ ::= ⟨variable⟩, ⟨expression⟩ |
                    ⟨variable⟩, ⟨expression⟩ WITH ⟨for spec⟩
⟨for statement⟩ ::= FOR ⟨for spec⟩ DO ⟨statement *⟩ REPEAT
e.g. FOR X, [1 2 3] WITH Y, [1 2 3] DO
    SUM(X, Y) REPEAT;
The ⟨expression⟩ must evaluate to a sequence (see below). The WITH statement has the same effect as nested FOR loops.

¶How to use the program. The program should be compiled by typing:
COMPILE(LIBRARY([LIB FOR]));
All the FOR statement facilities may then be used in the user's program.

Note: When using a program containing FOR statements, LIB FOR must be input BEFORE any such statements are mentioned.

The ⟨expression⟩ of the FOR statement must be one of three forms:

(a) a list, whose elements will be the successive values of the ⟨variable⟩
(b) a repeater function, which will produce successively each value of the ⟨variable⟩, and TERMIN as the terminator.
(c) a record whose data-word is SEQ and whose components are SEQFUNOF and SEQPROPSOF. This is produced by the function STEP.

STEP (X, Y, Z) is used to construct a SEQ record of a sequence starting at X in steps (positive or negative) of Y, until Z;
e.g.   FOR X, STEP(1, 1, 5) DO PRINT(X ↑ 2) REPEAT;
will cause the squares of 1 through 5 to be output. FOR X, [1 2 3 4 5] DO PRINT (X↑2) REPEAT; has the same effect, as does
FOR X, F DO PRINT(X↑2) REPEAT; where F is defined as

```
FUNCTION F;
IF N<6 THEN N; N + 1—>N EXIT;
TERMIN
END;
1—>N;
```

Sequences in the form of SEQ records or lists can be used as many times as required, in different places, and simultaneously; they can be handed on as parameters of functions and given as results. For example, if all indexing in a program was from one to eight, then the sequence STEP (1, 1, 8)—>ONEEIGHT; or [1 2 3 4 5 6 7 8]—>ONEEIGHT; could be set up globally, and ONEEIGHT used as the ⟨expression⟩ in all FOR statements in the program.

Note. FOR statements may not appear at execute level, as they use labels and GOTO statements.

¶Method used. The basic words of the package, FOR, WITH, DO, REPEAT, are all defined as macros.

The function REPEATEROF is applied to the ⟨expression⟩ and a repeater function is created (if necessary) and assigned to a variable whose name is invented by the function NEWNAME, and declared by the FOR statement apparatus.

A loop is created in which this repeater function is applied to no arguments, the result assigned to the ⟨variable⟩, and a test planted to see whether it is equal to TERMIN. At the REPEAT a GOTO statement to close the loop, and a label as the destination of the GOTO statement in the test for TERMIN, are planted.

¶Global variables
SEQFUNOF STEPFUN CONSSEQ SEQPROPS NEWNAME FORFUN REPEATEROF STEP SEQPROPSOF FOR DO WITH REPEAT FORLIST NAMCOUNT.

¶Store used
Approximately 2 blocks of store are used by the package.

¶Example of use
```
: COMPILE(LIBRARY([LIB FOR]));
: FUNCTION TEST;
: VARS X Y;
:    FOR X, STEP(1, 1, 3) WITH Y, [5 6 7]DO
:       PR(X);  PR(Y);  NL(1);
:    REPEAT;
: END;
:
: TEST();
 1 5
 1 6
 1 7
 2 5
 2 6
 2 7
 3 5
 3 6
 3 7
:
: FUNCTION SUM SEQ;
: VARS TOT U;
:    Ø-> TOT;
:    FOR U, SEQ DO TOT+U-> TOT REPEAT;
:    TOT
: END;
:
: SUM([23 24 45])=>

**     92,
:
: SUM(STEP(17,-1, 1))=>

**     153,
:
: STEP(1, 1, 1Ø)->X;
: SUM(X)=>

**     55,
:
: SUM(X)+SUM([1 2 3])=>

**     61,
:
```

```
[FOR]

VARS SEQFUNOF STEPFUN CONSSEQ SEQPROPS NEWNAME
FORFUN REPEATEROF STEP SEQPROPSOF ;

VARS FORLIST;  NIL -> FORLIST;

VARS NAMCOUNT;  0 -> NAMCOUNT;


FUNCTION NEWNAME;
    NAMCOUNT//10//10//10//10//10//10, 33, 8.CONSWORD;
    NAMCOUNT + 1 -> NAMCOUNT
END


MACRO FOR;
    VARS L;  .NEWNAME -> L;
    [%.ITEMREAD, L, L%] :: FORLIST -> FORLIST;
END


FUNCTION FORFUN W;
    VARS NLAB NVAR;  .NEWNAME -> NLAB;  .NEWNAME -> NVAR;
    MACRESULTS([% ";", "VARS", NVAR, ";",
    ".",  "REPEATEROF",  "->",  NVAR,

      ";",  NLAB,  ":",  ".",  NVAR,  "->",  FORLIST.HD.HD, ";",

    "IF",  FORLIST.HD.HD,  "=",  "TERMIN",  "THEN",

    "GOTO",  FORLIST.HD.TL.HD,   "CLOSE",  ";" %]);

    IF NOT(W = "DO") THEN .ITEMREAD -> FORLIST.HD.HD CLOSE;
    NLAB -> FORLIST.HD.TL.HD
END


MACRO DO;
    FORFUN("DO")
END


MACRO WITH; FORFUN("WITH") END


MACRO REPEAT;
    MACRESULTS([%
      ";", "GOTO", FORLIST.HD.TL.HD,  FORLIST.HD.TL.TL.HD,  ":" %]);

    FORLIST.TL -> FORLIST
END


FUNCTION REPEATEROF X;
    IF X.ISFUNC THEN X EXIT
    IF X.ATOM THEN
      IF X = NIL THEN LAMBDA ; TERMIN END RETURN
        ELSEIF X.ISCOMPND AND X.DATAWORD = "SEQ"
          THEN X.SEQFUNOF.APPLY RETURN
        ELSE 'WRONG FOR SEQ' => X.PR;.SETPOP
      CLOSE
      ELSE LAMBDA XREF;
              VARS U;  XREF.CONT -> U;
              IF U.NULL THEN TERMIN EXIT;
              U.DEST -> XREF.CONT
            END (% CONSREF(X) %)
    CLOSE
END
```

```
FUNCTION STEP X Y Z;
    CONSSEQ(STEPFUN(%X,Y,Z, IF Y > 0 THEN NONOP >  ELSE NONOP < CLOSE%),
    [%"STEP", X, Y, Z%])
END


FUNCTION STEPFUN X Y Z COMP;
    LAMBDA XREF Y Z COMP;
      IF COMP(XREF.CONT, Z) THEN TERMIN EXIT
      XREF.CONT;  XREF.CONT+Y -> XREF.CONT
    END(%CONSREF(X),Y,Z,COMP%)
END


RECORDFNS("SEQ", [0 0]) -> SEQPROPSOF -> SEQFUNOF;.ERASE; -> CONSSEQ;
```

Program name. LIB FOURS
Source. D. J. S. Pullin, DMIP; *Date of issue.* June 1969

¶*Description.* FOURS is a game of three-dimensional noughts and crosses, played on a 4 × 4 × 4 board. The program plays a game against the user, requesting moves, automatically replying, and displaying the board whenever it is the user's turn to play.

¶*How to use the program.* The program should be compiled by typing:
COMPILE(LIBRARY([LIB FOURS]));
The program asks a couple of questions of the user. These should be answered by either YES or NO and terminated with carriage-return/ line-feed. A full explanation of the game, and how to play against the program, is output if required.

¶*Method used.* The program evaluates all sequences of forcing moves, and if any leads to a win, either plays it or blocks it. If there is no winning sequence it selects the position which minimizes the opponent's free lines and maximizes its own, ignoring the forcing positions which have already been rejected. More weight is given to opponent's lines than its own, and to long lines than short ones. No lookahead is used in the non-forcing case.

A HELP facility is provided, and for this the program performs the above operations for the user instead of for itself.

```
[FOURS]

FUNCTION PLAY;
VARS MYLIST FORCEMOVES LINES POINTS OWNER LENGTHS USEDPTS
     BOARD A1 LINENO THISLINE COUNT1 COUNT2 X3 X2 COUNT3 X1
     A3 A2 COUNT SCALE RANSEED
     FORMLINES GETSPACE MOVEVALUE VALUES RANDOM
     TESTFORCE FREEPT FINDL UNMOVE UNWRAP MOVEIT HISMOVE
     PRINTBOARD MYITEMREAD NICEPR OCHAROUT CUCHAROUT;


FUNCTION SETUP;VARS X;
  INIT(64)->BOARD; INIT(64)->USEDPTS;
  INIT(76)->LENGTHS; INIT(76)->OWNER;
  INIT(64)->POINTS; 76->X;
L1: 4->SUBSCR(X,LENGTHS); 0->SUBSCR(X,OWNER);
  IF X<65 THEN
    INIT(8)->SUBSCR(X,POINTS); 1->SUBSCR(X,USEDPTS)
    ; 0->SUBSCR(X,BOARD)
  CLOSE;
  X-1->X; IF X>0 THEN GOTO L1 CLOSE;
  INIT(76)->LINES; .FORMLINES;
  NIL->FORCEMOVES; 0->MYLIST;
END;


FUNCTION MOVE X A B; VARS T U V W Y Z;
  SUBSCR(X,POINTS)->Y;1->Z;FALSE->U;
  IF B THEN .GETSPACE->T;
  CLOSE;
L1: SUBSCR(Z,Y)->W;
  IF W<0 THEN
    IF B THEN T;X CLOSE;
    0->SUBSCR(X,USEDPTS);
    U
  EXIT;
  SUBSCR(W,LENGTHS)->V;
  IF B THEN V+SUBSCR(W,OWNER)->SUBSCR(Z,T) CLOSE;
  IF SUBSCR(W,OWNER)=0 THEN
    IF V=0 THEN GOTO L2 CLOSE;
    A->SUBSCR(W,OWNER); 3->SUBSCR(W,LENGTHS);
  ELSEIF SUBSCR(W,OWNER)=A THEN
    V-1->SUBSCR(W,LENGTHS);
    IF V=1 THEN TRUE->U CLOSE;
  ELSE 0->SUBSCR(W,OWNER); 0->SUBSCR(W,LENGTHS);
  CLOSE;
L2: Z+1->Z; GOTO L1;
END;


FUNCTION LINEFORM; VARS Y Z;
  4->COUNT1; INIT(4)->THISLINE;
L1: X+1->SUBSCR(COUNT1,THISLINE);
  SUBSCR(X+1,USEDPTS)->Y; SUBSCR(X+1,POINTS)->Z;
  LINENO->SUBSCR(Y,Z); -1->SUBSCR(Y+1,Z);
  Y+1->SUBSCR(X+1,USEDPTS); X+A1->X;
  COUNT1-1->COUNT1; IF COUNT1>0 THEN GOTO L1 CLOSE;
  THISLINE->SUBSCR(LINENO,LINES);
  LINENO+1->LINENO
END;


FUNCTION GETSPACE;
  IF MYLIST=0 THEN INIT(8)
  ELSE MYLIST; SUBSCR(8,MYLIST)->MYLIST
  CLOSE
END;


FUNCTION FREESPACE X;
  MYLIST->SUBSCR(8,X); X->MYLIST;
END;
```

```
FUNCTION FORMLINES;
  1->A1;4->A2; 16->A3; 1->LINENO;
L1: 0->X; 1->X1; 4->COUNT3;
L5: X->X2; X->X3;
L2: 4->COUNT2;
L3: .LINEFORM;
  X2+A2->X2;
  X2->X;COUNT2-1->COUNT2;
  IF COUNT2>0 THEN GOTO L3 CLOSE;
  X3+A3->X3;X3->X2;X3->X;COUNT3-1->COUNT3;
  IF COUNT3>0 THEN GOTO L2 CLOSE;
  IF X1>0 THEN X1-1->X1;A3;A2;A1;A2+A1;
    A3->A2->A1;0->X;GOTO L5
  CLOSE;
  IF X1=0 THEN X1-1->X1; ->X;A1-X-X->A1;X;
    IF A1>0 THEN A1->X
    ELSE -A1->A1;3->X;
    CLOSE;
    GOTO L5
  CLOSE;
  ->A3 ->A1 ->A2;
  IF A1>1 THEN GOTO L1 CLOSE;
  0->X;21->A1; .LINEFORM;
  3->X;19->A1; .LINEFORM;
  12->X; 13->A1; .LINEFORM;
  15->X;11->A1; .LINEFORM;
1->X;
L4: 1->SUBSCR(X,USEDPTS);
  X+1->X;IF X<65 THEN GOTO L4 CLOSE;
END;
```

```
FUNCTION TESTMOVES;VARS X A B C D P1 P2;
  FINDL(1,0)->X;
  IF NOT(X=0) THEN
L4: X.FREEPT->A;
    IF SUBSCR(A,OWNER)=B THEN A EXIT;
    FINDL(1,X)->B;
    IF NOT(B=0) THEN B->X;GOTO L4 CLOSE;
    IF FORCEMOVES.NULL THEN
L5:   A
    EXIT;
    IF A=HD(FORCEMOVES) THEN GOTO L0 CLOSE;
    NIL->FORCEMOVES;GOTO L5
  CLOSE;
  IF NOT(NULL(FORCEMOVES)) THEN
L0: HD(FORCEMOVES); TL(FORCEMOVES)->FORCEMOVES;
  EXIT;
  8->P1; 16->P2;
  IF .TESTFORCE THEN GOTO L0 CLOSE;
  P1;P2->P1; ->P2;
  IF .TESTFORCE THEN HD(FORCEMOVES); NIL->FORCEMOVES; EXIT;
  INTOF(SCALE* .RANDOM)->COUNT; COUNT->COUNT1;
  IF SCALE>4 THEN SCALE//2->SCALE->X;
  CLOSE;
  -1000000->X; .VALUES; 1->D;
L1: IF D=65 THEN C EXIT;
  IF SUBSCR(D,USEDPTS)=0 THEN GOTO L2 CLOSE;
  MOVEVALUE(D,Y,Z,V,W)->B;
  IF B>X THEN
    COUNT1->COUNT;
L3: B->X; D->C;
  ELSEIF B=X THEN
    IF COUNT>1 THEN COUNT-1->COUNT;GOTO L3 CLOSE;
  CLOSE;
L2: D+1->D; GOTO L1
END;
```

```
FUNCTION TESTFORCE; VARS X W FIRST;
  0->X;
L4: FINDL(2,X)->X;
  IF X=0 THEN FALSE EXIT;
  IF SUBSCR(X,OWNER)=P2 THEN GOTO L4 CLOSE;
  TRUE->FIRST;
  X.FREEPT->Y ->W;
L2: MOVE(W,P1,TRUE)->V; MOVE(Y,P2,TRUE)->V;
L3: FINDL(1,0)->Z;
  IF Z>0 THEN
    IF SUBSCR(Z,OWNER)=P1 THEN Z.FREEPT::NIL->FORCEMOVES; .UNWRAP; EXIT;
    MOVE(Z.FREEPT,P1,TRUE)->V;
    FINDL(1,0)->V;
    IF V=0 OR SUBSCR(V,OWNER)=P2 THEN
      .UNMOVE; GOTO L1;
    CLOSE;
    MOVE(V.FREEPT,P2,TRUE)->V;
    GOTO L3;
CLOSE;
  IF .TESTFORCE THEN .UNWRAP EXIT;
L1: .UNMOVE; ->Y;Y.UNMOVE; IF NOT(Y=W) THEN GOTO L1 CLOSE;
  IF FIRST THEN
    FALSE->FIRST; X.FREEPT->W->Y; GOTO L2
CLOSE;
  GOTO L4
END;


FUNCTION UNMOVE T X; VARS Z U V W;
  1->SUBSCR(X,USEDPTS);
  SUBSCR(X,POINTS)->Z; 1->U;
L1: SUBSCR(U,Z)->V;
  IF V<0 THEN FREESPACE(T); EXIT;
  SUBSCR(U,T)->W;
  LOGAND(W,7)->SUBSCR(V,LENGTHS);
  LOGAND(W,24)->SUBSCR(V,OWNER);
  U+1->U; GOTO L1
END;


FUNCTION UNWRAP;
L1: .UNMOVE; ->X; X.UNMOVE;
  X::FORCEMOVES->FORCEMOVES;




  IF NOT(X=W) THEN GOTO L1 CLOSE;
  TRUE
END;


FUNCTION FINDL Y X;
L1: X+1->X; IF X=77 THEN 0 EXIT;
  IF SUBSCR(X,LENGTHS)=Y THEN X EXIT;
  GOTO L1;
END;


FUNCTION VALUES; VARS X;
  0->Y;0->Z;0->V;0->W;1->X;
L1: IF SUBSCR(X,LENGTHS)=2 THEN
    IF SUBSCR(X,OWNER)=P2 THEN Y+1->Y
    ELSE Z+1->Z
    CLOSE
  ELSEIF SUBSCR(X,LENGTHS)=3 THEN
    IF SUBSCR(X,OWNER)=P2 THEN V+1->V
    ELSE W+1->W
    CLOSE
  CLOSE
; X+1->X; IF X<77 THEN GOTO L1 CLOSE;
END;
```

EXIT;

```
FUNCTION MOVEVALUE X A B C D; VARS W Y Z;
SUBSCR(X,POINTS)->X; 1->Z;
L1: SUBSCR(Z,X)->W;
  IF W<0 THEN C-10*D+100*A-1000*B EXIT;
  SUBSCR(W,LENGTHS)->Y;
  IF Y=4 THEN C+1->C
  ELSEIF Y=3 THEN
    IF SUBSCR(W,OWNER)=P2 THEN A+1->A ELSE D-1->D CLOSE
  ELSEIF Y=2 THEN
    IF SUBSCR(W,OWNER)=P1 THEN B-1->B CLOSE
  CLOSE;
  Z+1->Z; GOTO L1;
END;



FUNCTION FREEPT X;VARS Y Z;
1->Y; SUBSCR(X,LINES)->X;
  L1: SUBSCR(Y,X)->Z;
  IF SUBSCR(Z,USEDPTS)=1 THEN Z CLOSE;
Y+1->Y; IF Y<5 THEN GOTO L1 CLOSE;
END;



FUNCTION MOVEIT X;MOVE(X,8,FALSE);
  8->SUBSCR(X,BOARD);
  (((X-1) //4) //4) .PR; .PR; .PR; .PRINTBOARD; 10.NL;
END;



FUNCTION HISMOVE; VARS L M N;
L0: PR('YOUR MOVE');
L1: .MYITEMREAD->L; IF L= "RESIGN" THEN L EXIT;
  IF L= "HELP" THEN
    IF NOT(NULL(FORCEMOVES)) THEN
      PR('I CAN WIN NO MATTER WHAT YOU DO.');
      GOTO L0
    CLOSE;
      FINDL(1,0)->L;IF L>0 THEN L.FREEPT->C;GOTO L8 CLOSE;
    16->P1;8->P2;
    IF .TESTFORCE THEN
      LENGTH(FORCEMOVES)->M; HD(FORCEMOVES)->N;
      PR('YOU CAN FORCE A WIN IN');PR(M);
      PR(' MOVES, STARTING AT');
L5:   (((N-1) //4) //4) .PR; .PR; .PR;
      1.NL; NIL->FORCEMOVES; GOTO L0
    CLOSE;
    P1; P2->P1; ->P2;
    IF .TESTFORCE THEN
      HD(FORCEMOVES)->N;
      PR('I AM IN A STRONG POSITION. HOWEVER, TRY');
      GOTO L5
    CLOSE;
    -1000000->X; .VALUES; 1->D;
L6: IF D=65 THEN
L8:       PR('TRY');(((C-1) //4) //4) .PR; .PR; .PR;
      1.NL; GOTO L0
    CLOSE;
    IF SUBSCR(D,USEDPTS)=0 THEN GOTO L7 CLOSE;
    MOVEVALUE(D,Y,Z,V,W)->B;
    IF B>X THEN B->X;D->C CLOSE;
L7: D+1->D; GOTO L6
  CLOSE;
  IF NOT(ISNUMBER(L)) THEN GOTO L1 CLOSE;
  IF L<0 OR L>3 THEN
L2:     PR('NO SUCH POSITION. PLEASE RETYPE.');
        GOTO L0;
  CLOSE;
```

OSE

L;

CLOSE;

```
L3: .MYITEMREAD->M;
  IF NOT(ISNUMBER(M)) THEN GOTO L3 CLOSE;
  IF M<0 OR M>3 THEN GOTO L2 CLOSE;
L4: .MYITEMREAD->N;
  IF NOT(ISNUMBER(N)) THEN GOTO L4 CLOSE;
  IF N<0 OR N>3 THEN GOTO L2 CLOSE;
  16*L+4*M+N+1->L;
  IF NOT(SUBSCR(L,BOARD)=0) THEN
    PR('THAT POSITION IS ALREADY OCCUPIED, STOP
TRYING TO CHEAT.`);
    GOTO L0
CLOSE;
  MOVE(L,16,FALSE);
  16->SUBSCR(L,BOARD);
END;


FUNCTION PRINTBOARD; VARS X Y Z;
  2.NL; 5.SP;0.PR;8.SP;1.PR;8.SP;2.PR;8.SP;3.PR;
  2.NL; 4->X;
L1: 2.SP;0.PR;1.PR;2.PR;3.PR;X-1->X;IF X>0 THEN GOTO L1 CLOSE;
  0->Y;1->Z;
L2: 1.NL;Y.PR;4->X;4->W;
L3: 1.SP;
  IF SUBSCR(Z,BOARD)=0 THEN PR(".")
  ELSEIF SUBSCR(Z,BOARD)=16 THEN PR("X")
  ELSE PR("O")
  CLOSE;
  Z+1->Z; X-1->X;
  IF X>0 THEN GOTO L3  CLOSE;
  W-1->W;
IF W>0 THEN 4->X;Z+12->Z;2.SP;GOTO L3 CLOSE;
  Y+1->Y;
  IF Y<4 THEN Z-48->Z;GOTO L2 CLOSE;
  2.NL;
END;


FUNCTION MYITEMREAD;
  APPLY(INCHARITEM(CHARIN));
END;


FUNCTION NICEPR X;
  IF X=23 OR X=32 THEN EXIT;
  X.OCHAROUT
END;



 CUCHAROUT->OCHAROUT;
  NICEPR->CUCHAROUT;
  .SETUP; INTOF(100*HD(POPDATE()))->RANSEED;
L1: 1.NL;
  PR('DO YOU KNOW HOW TO PLAY AGAINST THIS PROGRAM`);
  .MYITEMREAD->X;
  IF X= "YES" THEN  GOTO L3 CLOSE;
L2: PR('THE GAME IS PLAYED ON A 4*4*4 CUBE, THE OBJECT
BEING TO PLACE 4 PIECES IN A STRAIGHT LINE. YOUR
PIECES ARE SHOWN AS X , MINE AS O . TO MAKE A
MOVE YOU HAVE TO TYPE IN 3 NUMBERS, INDICATING
PLANE, ROW AND COLUMN; EACH IN THE RANGE 0 TO 3.
THUS IN THE BOARD SHOWN BELOW, YOU HAVE A PIECE
AT 1 3 2  AND I HAVE ONE AT 2 0 3.`);
  16->SUBSCR(31,BOARD); 8->SUBSCR(36,BOARD);
  .PRINTBOARD;
  0->SUBSCR(31,BOARD);0->SUBSCR(36,BOARD);
'YOU CAN ASK THE COMPUTER TO SUGGEST A MOVE BY TYPING  HELP
AND CAN CONCEDE DEFEAT BY TYPING  RESIGN
` .PR;
L3: 'DO YOU WANT TO START
` .PR;
  .MYITEMREAD->X; .PRINTBOARD;
  IF X= "YES" THEN 16->SCALE;GOTO L4 ELSE 8->SCALE;GOTO L6 CLOSE;
```

```
L4: .HISMOVE->X;
    IF X= "RESIGN" THEN
      IF NULL(FORCEMOVES) THEN
        PR('I HAD NOT REALISED THAT MY POSITION WAS IMPREGNABLE.
')    ELSE PR('FAIR ENOUGH. ')
    CLOSE;
    GOTO L5
  CLOSE;
  IF X THEN .PRINTBOARD;PR('YOU WIN. ');GOTO L5 CLOSE;
L6: 1.NL;PR('MY MOVE'); .TESTMOVES;
  IF .MOVEIT THEN 1.NL;PR('I WIN. ');GOTO L5 CLOSE;
  GOTO L4;
L5: PR('DO YOU WANT TO PLAY AGAIN');
  .MYITEMREAD->X;
  IF X= "YES" THEN .SETUP;GOTO L3 CLOSE;
  PR('BACK TO POP2 THEN');
END;



ERASE->CUCHAROUT;
COMPILE(LIBRARY([LIB RANDOM]));
CHAROUT->CUCHAROUT;
4.NL;
PR('TO ENTER PROGRAM , TYPE FOURS;




');
VARS OPERATION 1 FOURS;
PLAY->NONOP FOURS;
```