MPREGNABLE.

Program name. LIB FULL MEMOFNS
Source. David Marsh, DMIP; *Date of issue.* September 1970

¶*Description.* This is a very much extended and revised version of the memo-function facilities proposed by Michie (1967, 1968) and initially implemented by Popplestone (1967). Marsh (1970a) describes the present implementation.

(a) *Purpose.* To improve function evaluation time by the automatic storage of previously computed results.

(b) *Operation of a memo-function.* Basically, arguments and results of a function are stored in a special rote of a specified size. A linear search of this rote is carried out each time the function is evaluated and, if the arguments are on the rote, the result is taken directly, otherwise the function is evaluated, and the rote updated. Full memo-functions differ from the other library versions in the following ways:

(1) They handle functions of more than one argument and result.
(2) The structure of the rote entries is under the control of the user. For example, arguments which map onto the same results may be grouped together in any desired fashion as in, perhaps, a set description.
(3) The function results may be standardized before being inserted on the rote or being left as the function result.
(4) In LIB TREE MEMOFNS (Marsh 1970b) a tree-structured rote is used. In this implementation this facility is not available.

The following features are, however, still available in this full version:
(1) Completely new entries are inserted at the head of the rote.
(2) Entries which are used in lookup are promoted one place.
(3) When the rote exceeds a specified size, entries are deleted from the bottom of the rote.
(4) With recursive functions only top level arguments and results are inserted on the rote, though the rote is searched on every call.

(c) *Economies.* Extensive timing tests have not been carried out. For the log gamma function we found that the overheads of the current implementation precluded any pay-off in terms of increased speed (Marsh 1970a). However, along with Michie (Michie 1971), we have shown them to be a valuable tool in automatic problem solving.

(d) *Terminology and data standards.* (1) The function being memoized will be called the main function.
(2) We shall use X to indicate an argument or arguments and Y to indicate a result, and XY to indicate an argument-result pair.
(3) A rote entry is always a POP-2 pair, whose front contains the arguments and whose back contains the result. Initially these are set to UNDEF so that an empty rote entry is [UNDEF . UNDEF].
(4) An argument X may be presented to the component functions of the memo apparatus in various ways: (a) when the main function takes one argument, X will be either an atom, or a list within a list, if the single argument is itself a list. (b) when the main function takes more than one argument, X will be a list of those arguments, whose head is the first argument.
(5) The structure of the result is governed by the standardization function—YSTAND (q.v.).
(6) If the result of the main function is UNDEF no entry will be made in the rote under any circumstances.

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

COMPILE(LIBRARY([LIB FULL MEMOFNS]));

(a) *Creation of a memo-function.* The function NEWMEMO is provided for this:

NEWMEMO ∊ F, N, NARG, SEREQUIV, UPEQUIV, ROTEUPDATE,
YSTAND=>memo-function

(1) Arguments

F — The main function, to be memoized.

N — An integer, specifying the maximum size of the rote.

NARG — An integer, specifying the number of arguments that the main function takes.

SEREQUIV — SEREQUIV ∊ X, ROTE-ENTRY => TRUTHVALUE

This is the *equiv*alence function used in the initial *sear*ch of the rote.

The result of this function will be TRUE if the main function result for this argument X is to be taken from the rote. The function could be of several forms:

e.g. (a) if the arguments are numerical,
LAMBDA X ROTE; X=FRONT(ROTE) END

(b) if the rote contains an argument set,
LAMBDA X ROTE; MEMBER(X, FRONT(ROTE)) END

(c) LAMBDA X ROTE;
BOOLOR(INRANGE(X, RANGEPAIR(FRONT(ROTE))),
MEMBER(X, EXCEPTIONS(FRONT(ROTE))))
END

where the front of each rote-entry has two components, RANGEPAIR and EXCEPTIONS. RANGEPAIR is used by a function INRANGE, perhaps LAMBDA X PAIR;
BOOLAND(X>FRONT(PAIR), X<BACK(PAIR))
END

and EXCEPTIONS gives a list of exception conditions.

UPEQUIV — either (a) FALSE or (b) UPEQUIV ∊ XY, ROTE-ENTRY => TRUTHVALUE;

(a) If UPEQUIV is FALSE there will be no search of the rote when it is updated and a new entry will be made at the top of the rote by ROTEUPDATE (q. v.).

(b) This function looks for a match between any features of a new entry after the main function has been evaluated, and previous entries in the rote. If it returns TRUE then ROTEUPDATE will extend the rote-entry, for example, at its simplest it will test for equality between results:

LAMBDA XY ROTE; BACK(XY) = BACK(ROTE) END

ROTEUPDATE — ROTEUPDATE ∊ XY, ROTE-ENTRY => NEW-ROTE-ENTRY; This will be entered under two circumstances:

(a) There has been no search of the rote (UPEQUIV=FALSE) or the search (using UPEQUIV) has failed and a new rote entry is to be made. In these cases the ROTE-ENTRY parameter will be UNDEF, and most typically the function will be:

LAMBDA XY ROTE; XY END

(b) There has been a search of the rote and UPEQUIV has given the result TRUE. The arguments will be the same as those given to UPEQUIV and the result must be a new entry for the rote. For example,

```
LAMBDA XY ROTE;
FRONT(XY)::FRONT(ROTE) -> FRONT(ROTE);
ROTE
END
```

YSTAND This function converts the main function results into a canonical element. It takes as many arguments as the main function leaves results. It must leave only one result. This result is the Y component of the XY pair found as parameters of the UPEQUIV and ROTEUPDATE functions. It must be structured in the same way as the arguments of the main function are structured for presentation to the component functions of the memo apparatus. That is, if there is a single result it will be an atom, unless that single result is a list, in which case it will be a list whose head is that result; or if there is more than one result it will be a list whose elements are those results.

For example, if the main function leaves two results, both real numbers, and the canonical representation of these results is two integers, the function YSTAND may be:

```
LAMBDA R1 R2;
[% INTOF(R1), INTOF(R2) %]
END
```

Note that the results from the main function will always be in the standardized form whether or not the result has been found on the rote. It is left to the memo apparatus to ensure that the results on exit are presented as separate items, in this example two numbers.

(2) Result
The result of NEWMEMO is a memo-function with a rote of size N. This result must be assigned back to the variable which contained the original function.

(b) *Subsidiary functions*

ISMEMO(F); Produces the result TRUE if the function F is a memo-function, otherwise FALSE.

UNMEMO(F); Produces the original function from a memo-function F. n.b. UNMEMO is destructive and once applied to a memo-function, that memo-function should not be used again.

DICTOF(F); Produces the rote of the memo-function F. The rote is implemented as a strip. This may be printed using the standard function DATALIST, or with PRDICTOF.

PRDICTOF(F); Prints the rote of the memo-function F. Each XY pair is output on a new line.

ROTELENGTH(F); Produces an integer corresponding to the number of entries in the rote of the memo-function F.

NEWINITMEMO(F, N); Memoizes function F with a rote of size N. F must be a unary function taking an integer argument.

(c) *Functions used in the implementation.* The rote is implemented as a POP-2 strip with the argument-result pairs as POP-2 pairs. When a memo-function is created, all the storage required by the rote is claimed. Care must therefore be taken not to specify a rote size greater than that actually required.

ASSOCVAL(X, ROTE, SEREQUIV, N); Finds the item associated with the item X under the equivalence SEREQUIV in the rote ROTE of size N.

ASSOCUD(Y, X, ROTE, N, UPEQUIV, ROTEUPDATE); Uses ROTEUPDATE to associate the items X and Y (arguments and results) on the rote ROTE of size N, the nature of the association being determined by UPEQUIV.

NEWASSOCFN(N, SEREQUIV, UPEQUIV, ROTEUPDATE); Produces a rote of size N whose selector function is ASSOCVAL and whose updater function is ASSOCUD.
CONSARG(NARG); Forms a list of the arguments of the memo-function taking each argument off the stack. If the function is unary, taking a simple item as argument, then this item is the result of CONSARG.
DESTARG(ARG); Places the arguments of the memo-function contained in ARG on the stack.
DOMEMO(F, A, YSTAND, NARG); The general purpose memo-function which NEWMEMO converts to a particular memo-function.

(d) *Storage requirements.* The memo apparatus occupies approximately 2. 8 blocks of core store. Once the program is compiled the only overheads for each function memoized will be the rote.

¶*Errors*
A memo-function should be redefined if a run-time error occurs.

¶*Example of the use of the program.* Consider the function
LOGGAMMA ∈ REAL => REAL

In POP-2 we can define this, approximately, as

```
FUNCTION LOGGAMMA X;
IF X < 2 THEN
   LOG(1. 5749—0. 5749*X)
ELSE
   LOG(X—1) + LOGGAMMA(X—1)
CLOSE
END;
```

Now suppose that the result of LOGGAMMA is required to only 2 significant figures. Then YSTAND is SIG2 where SIG2, which we shall assume has already been defined, performs the desired truncation. Many arguments will now map onto the same result. These arguments could be grouped together explicitly on the rote. Since LOGGAMMA is monotonic we can represent them more economically by a pair of numbers corresponding to the least and greatest values which have been found to give the same standardized result. Initially, however, only one argument will be associated with a given result and that argument will be stored as a single number.

SEREQUIV tests whether a particular argument is already represented on the rote:

```
FUNCTION SEREQUIV X ROTEXY;
VARS ROTEX;
FRONT(ROTEXY) —> ROTEX;
IF ISNUMBER(ROTEX) THEN
      X = ROTEX
ELSE
      BOOLAND(X>= FRONT(ROTEX), X=< BACK(ROTEX))
CLOSE
END;
```

UPEQUIV is defined as:

```
FUNCTION UPEQUIV XY ROTEXY;
BACK(XY)=BACK(ROTEXY)
END;
```

If UPEQUIV produces TRUE as result then, when ROTEUPDATE is

called, it will extend the appropriate lower or upper bound to accommodate the new argument, otherwise it will add the new argument and result to the rote. It is defined as:

```
FUNCTION ROTEUPDATE XY ROTEXY;
VARS X ROTEX;
IF ROTEXY=UNDEF THEN XY EXIT;
FRONT(XY)->X; FRONT(ROTEXY)->ROTEX;
IF ISNUMBER(ROTEX) THEN
    IF X>ROTEX THEN
       ROTEX::X
    ELSE
       X::ROTEX
    CLOSE->FRONT(ROTEXY)
ELSEIF X<FRONT(ROTEX) THEN
    X->FRONT(ROTEX)
ELSE
    X->BACK(ROTEX)
CLOSE;
ROTEXY
END;
```

We can now memoize LOGGAMMA, with a rote size of, say, 4:

```
NEWMEMO(LOGGAMMA, 4, 1, SEREQUIV, UPEQUIV, ROTEUPDATE,
SIG2) -> LOGGAMMA;
```

Memoization does not affect the way LOGGAMMA is called. Thus,

```
LOGGAMMA(10.1) =>
** 13.0,
```

The call will, however, have resulted in an entry in the rote:

```
PRDICTOF(LOGGAMMA);
[10.1 . 13.0]
```

If we now call

```
LOGGAMMA (10.5) =>
** 13.0,
```

an argument range will be constructed:

```
PRDICTOF(LOGGAMMA);
[[10.1 . 10.5] . 13.0]
```

Further calls will augment the rote:

```
LOGGAMMA(8.1), LOGGAMMA(11.6), LOGGAMMA(11.7),
LOGGAMMA(16) =>
** 8.7, 16.0, 16.0, 28.0,
PRDICTOF(LOGGAMMA);
[16 . 28.0]
[[11.6 . 11.7] . 16.0]
[8.1 . 8.7]
[[10.1 . 10.5] . 13.0]
```

If we now call

```
LOGGAMMA(10.3) =>
** 13.0,
```

the result will be obtained by look-up and the appropriate rote-entry will be promoted:

```
PRDICTOF(LOGGAMMA);
[16 . 28.0]
[[11.6 . 11.7] . 16.0]
[[10.1 . 10.5] . 13.0]
[8.1 . 8.7]
```

If we now evaluate
LOGGAMMA(20) =>
** 40.0,

the bottom entry of the rote will be lost.
PRDICTOF(LOGGAMMA);
[20 . 40.0]
[16 . 28.0]
[[11.6 . 11.7] . 16.0]
[[10.1 . 10.5] . 13.0]

¶*Global variables used.* ASSOCVAL ASSOCUD NEWASSOCFN CONSARG DESTARG DOMEMO ISMEMO UNMEMO DICTOF PRDICT PRDICTOF ROTELENGTH NEWINTMEMO

REFERENCES

Marsh, D. (1968) LIB MEMOFNS. *Multi-POP Program Library Documentation.* Edinburgh: Department of Machine Intelligence and Perception (reproduced in this volume).

Marsh, D. (1970a) Memo functions, the Graph Traverser and a simple control situation. *Machine Intelligence 5*, pp. 281-300 (eds. B. Meltzer & D. Michie). Edinburgh: Edinburgh University Press.

Marsh, D. (1970b) LIB TREE MEMOFNS. *Multi-POP Program Library Documentation.* Edinburgh: Department of Machine Intelligence and Perception, University of Edinburgh.

Michie, D. (1967) Memo functions: a language facility with 'rote-learning' properties. *Research Memorandum MIP-R-29* Edinburgh: Department of Machine Intelligence and Perception, University of Edinburgh.

Michie, D. (1968) Memo functions and machine learning. *Nature, 218,* 19-22.

Michie, D. (1971) Mechanization of plan-formation. *Proceedings NATO Advanced Study Institute on Artificial Intelligence and Heuristic Programming.* Edinburgh: Edinburgh University Press (In press).

Popplestone, R. J. (1967) Memo functions and the POP-2 language. *Research Memorandum MIP-R-30* Edinburgh: Department of Machine Intelligence and Perception, University of Edinburgh.

```
[FULL MEMOFNS]

FUNCTION ASSOCVAL X ROTE SEREQUIV N;
VARS P P0 U;
SUBSCR(N+1,ROTE)->P; P->P0;
L1: SUBSCR(P,ROTE)->U;
    IF BACK(U)=UNDEF THEN UNDEF EXIT;
    IF SEREQUIV(X,U) THEN BACK(U);
        IF NOT(P=P0) THEN P-1->P0;
           IF P0=0 THEN N->P0 CLOSE;
           SUBSCR(P0,ROTE),U, ->SUBSCR(P0,ROTE); ->SUBSCR(P,ROTE)
        CLOSE;
    EXIT;
P+1->P;
IF P>N THEN 1->P CLOSE;
IF (P=P0) THEN UNDEF EXIT;
GOTO L1
END



FUNCTION ASSOCUD Y X ROTE N UPEQUIV ROTEUPDATE;
IF EQUAL(Y,UNDEF) OR EQUAL(Y,[%UNDEF%]) THEN EXIT;
VARS P U P0 XY; X::Y->XY;
SUBSCR(N+1,ROTE)->P; P->P0;
IF NOT(UPEQUIV) THEN GOTO L2 CLOSE;
L1: SUBSCR(P,ROTE)->U;
IF (BACK(U)=UNDEF) THEN GOTO L2 CLOSE;
    IF UPEQUIV(XY,U) THEN ROTEUPDATE(XY,U)->SUBSCR(P,ROTE) EXIT;
    P+1->P;
    IF P>N THEN 1->P CLOSE;
    IF NOT(P=P0) THEN GOTO L1 CLOSE;
L2: IF P0=1 THEN N->P0 ELSE P0-1->P0 CLOSE;
ROTEUPDATE(XY,UNDEF)->SUBSCR(P0,ROTE);
P0->SUBSCR(N+1,ROTE)
END



FUNCTION NEWASSOCFN N SEREQUIV UPEQUIV ROTEUPDATE;
VARS U V N1;
INIT(N+1)->U; 1->SUBSCR(N+1,U);
1->N1;
L1: IF N1=<N THEN UNDEF::UNDEF->SUBSCR(N1,U); N1+1->N1; GOTO L1 CLOSE;
ASSOCVAL(% U,SEREQUIV,N %)->V;
ASSOCUD(% U,N,UPEQUIV,ROTEUPDATE %)->UPDATER(V);
V
END



FUNCTION CONSARG NARG;
VARS L; ->L; IF L.ATOM.NOT THEN L::NIL->L; CLOSE;
L1: IF NARG=1 THEN L EXIT;
    ::L->L; NARG-1->NARG;
GOTO L1
END



FUNCTION DESTARG ARG;
IF ARG.ATOM THEN ARG EXIT;
L1: IF ARG.NULL THEN IF NOT(ARG=NIL) THEN ARG CLOSE; EXIT;
    ARG.DEST->ARG;
GOTO L1
END
```

```
FUNCTION DOMEMO F A YSTAND NARG;
VARS ARG RES; NARG.CONSARG->ARG;
A(ARG)->RES;
IF F.FNPROPS.TL.HD THEN
    IF RES=UNDEF THEN
       FALSE->F.FNPROPS.TL.HD;
       YSTAND(F(DESTARG(ARG)))->RES;
       RES->A(ARG);
       TRUE->F.FNPROPS.TL.HD
    CLOSE;
       DESTARG(RES)
    ELSE
    IF RES=UNDEF THEN
       DESTARG(YSTAND(F(DESTARG(ARG))));
       ELSE
       DESTARG(RES);
    CLOSE;
CLOSE
END


FUNCTION NEWMEMO F N NARG SEREQUIV UPEQUIV ROTEUPDATE YSTAND;
VARS U V;
IF F.FNPROPS.ATOM THEN F.FNPROPS::NIL->F.FNPROPS CLOSE;
1::TL(FNPROPS(F))->TL(FNPROPS(F));
NEWASSOCFN(N,SEREQUIV,UPEQUIV,ROTEUPDATE)->U;
DOMEMO(% F,U,YSTAND,NARG %)->V;
UPDATER(U)->UPDATER(V);
[MEMO]->FNPROPS(V);
V
END


FUNCTION ISMEMO F;
F.FNPROPS->F;
L1: IF F.ATOM
    THEN F="MEMO"
    ELSE F.HD->F; GOTO L1
    CLOSE
END


FUNCTION UNMEMO F;
VARS MF;
IF F.ISMEMO THEN FROZVAL(1,F)->MF;
IF MF.FNPROPS.TL.NULL.NOT THEN TL(TL(FNPROPS(MF)))->TL(FNPROPS(MF)) CLOSE;
MF
ELSE
UNDEF
CLOSE
END


FUNCTION DICTOF F;
IF F.ISMEMO THEN FROZVAL(1,FROZVAL(2,F)) ELSE UNDEF CLOSE;
END


FUNCTION PRDICT F PRFUN;
VARS P P1 N STRIP;
F.DICTOF->STRIP;
1.NL; IF STRIP=UNDEF THEN UNDEF.PR EXIT;
FROZVAL(3,FROZVAL(2,F))->N;
SUBSCR(N+1,STRIP)->P; 1->P1;


L1: IF BACK(SUBSCR(P,STRIP))=UNDEF THEN EXIT;
PRFUN(SUBSCR(P,STRIP)); 2.NL; P+1->P; P1+1->P1;
IF P1>N THEN EXIT;
IF P>N THEN 1->P CLOSE;
GOTO L1
END
```

```
VARS PRDICTOF;

PRDICT(%PR%)->PRDICTOF;

FUNCTION ROTELENGTH F;
IF F.ISMEMO.NOT THEN 0 EXIT;
VARS N P P1 STRIP;
F.DICTOF->STRIP;
IF STRIP=UNDEF THEN 0 EXIT;
FROZVAL(3,FROZVAL(2,F))->N;
SUBSCR(N+1,STRIP)->P; 0->P1;
L1: IF BACK(SUBSCR(P,STRIP))=UNDEF THEN P1 EXIT;
P+1->P; P1+1->P1;
IF P1>N THEN N EXIT;
IF P>N THEN 1->P CLOSE;
GOTO L1
END

VARS NEWINTMEMO;
NEWMEMO(% 1,LAMBDA X ROTE; X=FRONT(ROTE); END,0,
              LAMBDA XY ROTE; XY END,LAMBDA Y; Y END%)
->NEWINTMEMO;

COMMENT
NEWMEMO(F,N,NARG,SEREQUIV,UPEQUIV,ROTEUPDATE,YSTAND)->F
NEWINTMEMO(F,N)->F
SEREQUIV(X,ROTE)=>0/1
UPEQUIV(X::Y,ROTE)=> 0/1
ROTEUPDATE(X::Y,ROTE) =>ROTE
;
```

Program name: LIB GRAPH TRAVERSER

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

¶*Description.* The Graph Traverser is a heuristic problem solving algorithm developed by Doran and Michie (1966). This package provides the basic facilities required to apply the Graph Traverser to a problem. Familiarity with the general ethos of the Graph Traverser is assumed.

The main function provided (GROWTREE) grows a search tree. Nodes on this tree are POP-2 records, called JOBS. These jobs are held on a list, JOBLIST, which is ordered, the 'better' jobs being at the head.

Because the Graph Traverser can be used in a great variety of situations it is not possible to specify in advance how a search tree might be handled. On the one hand, one might be solving a problem such as the Eight-puzzle where the achievement of a goal state and the printing out of a path leading to this state are the important factors. On the other hand, one might be using the Graph Traverser as a planning or control routine where there is no specific goal, and the importance lies in the pruning of the partial search tree and the selection and application of a specific operator. So, while facilities, such as finding a path or pruning a search tree, are provided, it is left to the user to adopt them in a manner suited to his objectives.

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

A. *Data structures*
(1) A node on the Graph is termed a JOB and is held as a POP-2 record of 5 components.
(a) STATE The representation of the problem state.
(b) USAGE The number (an integer > =1) of the operator next to be applied to this state.
(c) PARENT The parent node, either another JOB, or UNDEF if the JOB is at the root of the tree.
(d) VALUE The value of a state, as worked out by the evaluation function.
(e) OPUSED The number of the operator which was applied to the parent to reach this state.
(2) The tree is an ordered collection of jobs held on a list, the current best job is at the head of this list.

B. *The growtree function and its parameters*
GROWTREE ϵ JOBLIST, PREDICT, EVAL, ISLIMIT, ISDEVELOPED, BETTERTHAN
=> JOBLIST;
This function grows a search tree to the specification of the parameters described. *These parameters must be defined by the user.* They are as follows:

(1) JOBLIST This is a list of jobs. It may be:
(a) A new tree with just one job, the root of the tree. A function NEWTREE is provided to construct this.
(b) A tree left over from a previous search.

(2) PREDICT
PREDICT ϵ POSITION, OPNO => POSITION;
(n.b. POSITION and OPNO are synonyms for STATE and USAGE.

These latter names are not used here because they are already defined as selector functions on the job records.)

PREDICT is the function for applying an operator to a state. The representation of this state is determined by the user. The operator number is an integer in the range from 1 to a limit appropriate to ISDEVELOPED (q.v.). This limit may be variable to allow for a varying number of operators. The function could be of the form:

```
(a)  FUNCTION PREDICT POSITION OPNO;
        IF OPNO = 1 THEN
           APPLY (POSITION, FIRSTOP)
        ELSEIF OPNO = 2 THEN
           APPLY (POSITION, SECONDOP)
        ELSE
           APPLY (POSITION, THIRDOP)
        CLOSE
     END;
```

Where FIRSTOP, SECONDOP and THIRDOP are functions.

The operators could be held in a list so:

```
(b)  FUNCTION PREDICT POSITION OPNO;
        APPLY (POSITION, GET (OPNO, OPERATORLIST))
     END;
```

where GET(N, L) finds the Nth item of a list L.

(3) EVAL

EVAL ϵ JOB => NUMBER;

This is the desirability function, which controls the direction of growth of the Graph Traverser. Remember that a JOB has a component STATE. The result of this function is assigned to the VALUE component of a JOB.

(4) ISLIMIT

ISLIMIT ϵ JOBLIST => TRUE or FALSE;

This function decides when a search shall terminate. This may be:
(a) because the goal state has been found, that is, (ISGOAL(EH (JOBLIST)) = TRUE) or
(b) because the tree has reached a maximum size, that is, (LENGTH(JOBLIST) >= TREEMAX)
where ISGOAL and TREEMAX must be specified by the user.
This function is entered as soon as the search function is entered and then every time a new job is added to the tree.

(5) ISDEVELOPED

ISDEVELOPED ϵ JOB => TRUE or FALSE;

This function returns TRUE if it is *not* applicable to apply any operators before the limit of the maximum possible has been reached.

```
e.g.   IF STATE(JOB) = UNDEF OR STATE(JOB) = "FAIL"
       OR JOB.ISREPEAT THEN TRUE ELSE FALSE CLOSE;
```

(6) BETTERTHAN

BETTERTHAN ϵ JOB1, JOB2 => TRUE or FALSE;

This function specifies the ordering of the JOBLIST, or, more precisely, which job should be developed first. It produces TRUE if the first JOB is "better" than the second.
e.g. VALUE(JOB1) < VALUE(JOB2) if the goal state has small values.

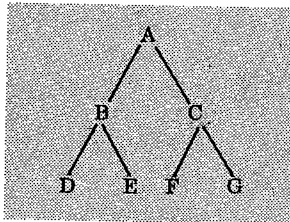
C. Other functions available to the user

(1) PRUNE

PRUNE ϵ JOBLIST => JOB, JOBLIST;

This function retraces from the current best job (HD(JOBLIST)) to the root of the tree; leaves the root JOB as a result, marks the descendent

of the root (on the retraced path) as a new root and then removes from the JOBLIST any JOB which does not descend from this new root.
for example, take a tree



If we show just the state component of a JOB, the JOBLIST corresponding to this tree might be [D A B E F C G], so:
PRUNE([D A B E F C G]) =>
** A, [D B E],

(2) FINDPATH
FINDPATH ϵ JOB => PATHLIST;
This will take any JOB and produce a list of jobs on the path from the root of the tree to that JOB. So, from the previous example,
FINDPATH(D) =>
** [A B D]
To print out the states from the root to the current best, one would write:
APPLIST(FINDPATH(HD(JOBLIST)), PRJOBSTATE);
where PRJOBSTATE is declared by the user.

(3) NEWTREE
NEWTREE ϵ POSITION => JOBLIST;
This takes a position (state representation) and produces a joblist, containing one JOB, the root of a tree.

(4) RETRACE
RETRACE ϵ JOB, PRFUN => (); where PRFUN ϵ JOB => ();
This is similar to FINDPATH except that a pathlist is not created. The jobs on the path from the root of the tree to a specified JOB are supplied in turn to the function PRFUN; it could be used for just counting the number of jobs in a path.

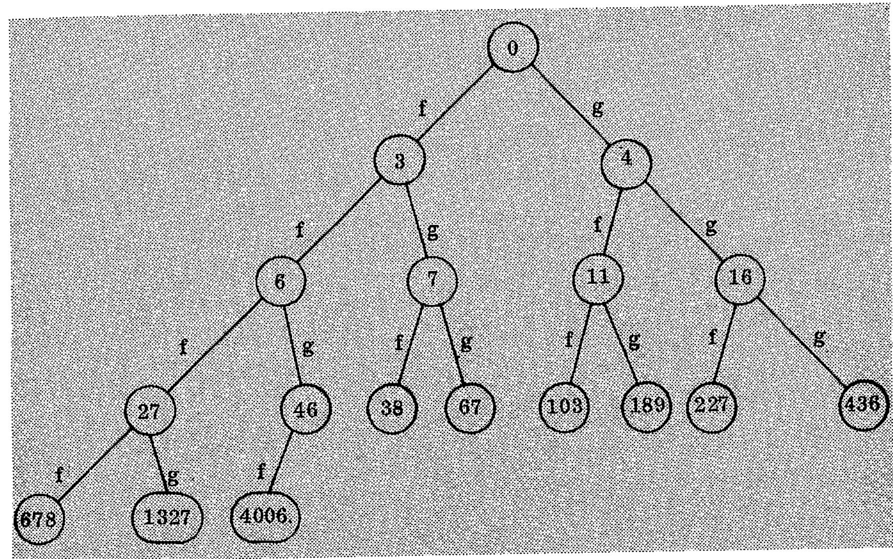
(5) ISREPEAT
ISREPEAT ϵ JOB, EQUIVSTATE => TRUE or FALSE, where
EQUIVSTATE ϵ POSITION, POSITION => TRUE or FALSE.
This has the result TRUE if there are any states equivalent to the state of the given JOB on the path between that JOB and the root of the tree.

(6) ISROOT
ISROOT ϵ JOB => TRUE or FALSE;
Result is TRUE if the job is a root (that is, PARENT(JOB) = UNDEF)

(7) CURTAIL
CURTAIL ϵ JOBLIST, SIZE => JOBLIST;
This limits the length of the list to a specified size by removing items from the tail end of a list. It is useful in some circumstances when it may not be appropriate to prune the joblist, for example, where the current best is still the root of the tree, and has not been fully developed. In such circumstances we can prune by removing the least promising (worst) nodes from the tree.

(8) INSERT
INSERT ϵ ITEM, LIST => LIST;
This inserts an item into an ordered list, ordered under the global (to INSERT) function BETTERTHAN.

¶*Example of use of program.* Let us take the example from Burstall (1968). The tree is defined in terms of a function of two integers
$f(n) = n^2 - 2n + 3$
$g(n) = 2n^2 - 5n + 4$
The problem is to find some value of n with some goal property p by repeatedly applying either f or g (operators) to a starting value n_0.
For example,
$n_0 = 0$
$p(n) = 30 < n \leq 40$
Part of the corresponding search tree is shown in the tree below.



The evaluation function $r(n) = |n - 35|$

To search this by partial development (that is, applying one operator at a time) we would do:

```
: COMPILE(LIBRARY([LIB GRAPH TRAVERSER]));

: FUNCTION PREDICT N OPNO;
:     1.NL; PR(N); 5.SP;                              (Monitoring
:     IF OPNO = 1                                      progress)
:     THEN PR("F");  (N*N—2*N+3)
:     ELSE PR("G");  (2*N*N—5*N+4)
:     CLOSE. PRINT;
: END;

: FUNCTION EVAL JOB;
: VARS N;
:     STATE(JOB)—35—>N;                               (| n—35 |)
:     IF N< 0 THEN —N ELSE N CLOSE
: END;

: FUNCTION ISLIMIT JOBLIST;
: VARS N;
:     STATE(HD(JOBLIST)) —> N;                        (30< (best job) =<
:     BOOLAND(N>30, N=<40);                           40)
: END;
```

```
: FUNCTION ISDEVELOPED JOB;
:     USAGE (JOB)>2
: END;

: FUNCTION BETTERTHAN JOBA JOBB;
:     VALUE(JOBA)<VALUE(JOBB)
: END;

:  VARS TREE PATH;
:  GROWTREE(NEWTREE(0), PREDICT, EVAL, ISLIMIT,
:  ISDEVELOPED, BETTERTHAN) -> TREE;

0    F   3
3    F   6
6    F   27
27   F   678
27   G   1327
6    G   46
46   G   4006
3    G   7
7    F   38 :
:  FINDPATH(HD(TREE)) -> PATH;
:  MAPLIST(PATH, STATE) =>
** [0 3 7 38]
```

However, we may want to proceed (as in Burstall's example) by applying all the operators at once to a particular state, that is, complete development. This is done by not applying any operators to a state until its parent is fully developed. We redefine ISDEVELOPED:

```
: FUNCTION ISDEVELOPED JOB;
: BOOLOR(USAGE(JOB)>2,
           IF JOB. ISROOT. NOT
              THEN USAGE(PARENT(JOB)) =< 2
              ELSE FALSE
           CLOSE);
: END;
```

So now:

```
: GROWTREE(NEWTREE(0), PREDICT, EVAL, ISLIMIT, ISDEVELOPED,
: BETTERTHAN) -> TREE;

0    F   3
0    G   4
4    F   11
4    G   16
16   F   227
16   G   436
11   F   102
11   G   191
3    F   6
3    G   7
7    F   38:

:  FINDPATH(HD(TREE)) -> PATH;
:  MAPLIST(PATH, STATE) =>
** [0 3 7 38],
:  OPUSED(HD(TL(PATH))) =>
**   1
:  MAPLIST(TREE, STATE) =>
**   [38 16 11 7 6 4 3 0 102 191 227 436],
```

Find operator used in going from root to first job in this path.

```
: PRUNE(TREE) —> TREE; .STATE =>
** 0                                   The state of the 'old' root
: MAPLIST(TREE, STATE) =>              of the tree.
** [38 7 6 3],
: MAPLIST(FINDPATH(HD(TREE)),          The tree is now pruned,
: STATE) =>                            and the path correspond-
** [3 7 38]                            ingly shorter.
```

¶*Global variables used.* PARENT USAGE STATE VALUE OPUSED DESTJOB CONSJOB GROWTREE INSERT ISROOT DEVELOP PRUNE FINDPATH NEWTREE ISREPEAT RETRACE CURTAIL

¶*Stored use.* The basic program uses approximately 2 blocks. Any more will depend on the state representation and the size of the partial search tree. A rough guide is to add 6 words for each node on the tree.

REFERENCES

Burstall, R. M. (1968) Writing search algorithms in functional form. *Machine Intelligence 3,* pp. 373-85 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Doran, J. E. & Michie D. (1966) Experiments with the Graph Traverser program. *Proc. R. Soc. A, 294,* 235-59.

```
[GRAPH TRAVERSE]

VARS INSERT ISROOT DEVELOP;
VARS PARENT USAGE STATE VALUE OPUSED DESTJOB CONSJOB;
RECORDFNS("JOB",[0 0 0 0 0])->OPUSED->VALUE->PARENT->USAGE->STATE
          ->DESTJOB->CONSJOB;




FUNCTION GROWTREE JOBLIST PREDICT EVAL ISLIMIT ISDEVELOPED BETTERTHAN;
  VARS JOB JOBPOINT;
  L0: IF JOBLIST.ISLIMIT THEN JOBLIST EXIT
      JOBLIST->JOBPOINT;
      L1: IF JOBPOINT.NULL THEN JOBLIST EXIT
          IF HD(JOBPOINT).ISDEVELOPED
             THEN JOBPOINT.TL->JOBPOINT;  GOTO L1
          CLOSE;
  DEVELOP(HD(JOBPOINT))->JOB;
  INSERT(JOB,JOBLIST)->JOBLIST;
  GOTO L0
END




FUNCTION INSERT X L;
  VARS HL; L->HL;
  IF L.NULL THEN X::NIL EXIT
  L1: IF L.TL.NULL
      THEN IF BETTERTHAN(X,L.HD)
           THEN HD(L), X  -> HD(L) -> X; CLOSE;
           X::TL(L)->TL(L); HL  EXIT
      IF BETTERTHAN(X,L.HD)
      THEN HD(L)::TL(L) -> TL(L);
           X-> HD(L); HL  EXIT
  TL(L)->L;
  GOTO L1
END




FUNCTION ISROOT JOB;
  PARENT(JOB)=UNDEF;
END




FUNCTION DEVELOP JOB;
  VARS NEW;  PREDICT(STATE(JOB),USAGE(JOB))->NEW;
  CONSJOB(NEW,1,JOB,0,USAGE(JOB))->NEW;
  EVAL(NEW)->VALUE(NEW);
  USAGE(JOB)+1->USAGE(JOB);
  NEW;
END




FUNCTION PRUNE JLIST;
  FUNCTION DELETE L PRED;
    VARS HL; L->HL;
    L1: IF L.NULL THEN HL EXIT
        IF L.TL.NULL THEN HL EXIT
        IF PRED(L.TL.HD) THEN L.TL.TL->L.TL;
           ELSE L.TL->L; CLOSE;
    GOTO L1
  END;
```

```
  VARS JOB; JLIST.HD->JOB;
  L1: IF ISROOT(PARENT(JOB))
      THEN PARENT(JOB);
           "OLD"-> PARENT(PARENT(JOB));
           UNDEF->PARENT(JOB);
      ELSE PARENT(JOB)->JOB;
           GOTO L1
      CLOSE;
  DELETE(JLIST,
    LAMBDA JOB;
      VARS UP DOWN OLD;
      "BASE"->DOWN;
      L1: PARENT(JOB)->UP;
          DOWN->PARENT(JOB);
          IF UP = "UNDEF" THEN FALSE->OLD; GOTO L2 CLOSE;
          IF UP = "OLD"   THEN TRUE ->OLD; GOTO L2 CLOSE;
          JOB->DOWN;
          UP->JOB;
        GOTO L1
      L2: PARENT(JOB)->DOWN;
          IF OLD THEN "OLD" ELSE UP CLOSE -> PARENT(JOB);
          IF DOWN = "BASE" THEN OLD EXIT;
          JOB->UP;
          DOWN->JOB;
        GOTO L2
    END);
END


FUNCTION FINDPATH JOB;
  VARS PATH; NIL->PATH;
  L1: JOB::PATH->PATH;
      IF JOB.ISROOT THEN PATH EXIT
      PARENT(JOB)->JOB;
  GOTO L1
END


FUNCTION NEWTREE S;
  VARS JOB; CONSJOB(S,1,UNDEF,0,UNDEF)->JOB;
  EVAL(JOB)->VALUE(JOB);
  JOB::NIL;
END;


FUNCTION ISREPEAT JOB EQUIV;
  VARS JOBSTATE;  STATE(JOB)->JOBSTATE;
  L1: IF JOB.ISROOT THEN FALSE EXIT
      PARENT(JOB)->JOB;
      IF EQUIV(JOBSTATE,STATE(JOB)) THEN TRUE EXIT
  GOTO L1
END



FUNCTION RETRACE JOB PRFUN;
  IF JOB.ISROOT THEN PRFUN(JOB);
  ELSE RETRACE(PARENT(JOB),PRFUN); PRFUN(JOB); CLOSE;

END


FUNCTION CURTAIL L MAX;
  VARS HL; L->HL;
  L1: IF L.NULL THEN HL EXIT
      IF MAX=0 THEN NIL->L.TL; HL EXIT
      MAX-1->MAX; L.TL->L; GOTO L1
END

2.NL;PR('LIB GRAPH TRAVERSER READY');3.NL;
```

Program name. LIB INDEX
Source. R. H. Owen, DMIP; *Date of issue.* February 1969

¶*Description.* The program inputs a sequence of words or phrases with associated page or section numbers and orders them alphabetically, collecting together the associated numbers of phrases which occur more than once. The resulting index may be output to any device in a form either suitable for re-input, or for direct use.

¶*How to use the program.* The program should be compiled by typing:
COMPILE(LIBRARY([LIB INDEX]));
This will also cause the library file LIB ALLSORT, which is used by the program, to be input.
The following functions will now be available

(1) NEWINDEX();
This function must be called to initialize the program's workspace before a new index is input.

(2) READINDEX(⟨charrep⟩);
This function is used to read in the index from an input device whose character repeater is ⟨charrep⟩. The input must have the following format:
⟨word or phrase⟩;⟨non-empty sequence of numbers⟩;
and must be terminated by an asterisk, for example,

```
ALEXANDER THE GREAT; 41 42;
BUCEPHALUS; 42;
AGAMEMNON; 43;
BUCEPHALUS: 43;
MENELAUS; 43 44;
*
```

As above, more than one number may be associated with a phrase if desired, and a phrase can occur more than once. When the index is ordered, the associated numbers of similar phrases will be grouped together.

(3) MAKEINDEX (⟨charrep⟩);
This outputs the index in alphabetical form to the output device whose character repeater is ⟨charrep⟩, in the form:

```
A
AGEMEMNON                 43
ALEXANDER THE GREAT       41 42
B
BUCEPHALUS                42 43
M
MENELAUS                  43 44
```

(4) STOREINDEX(⟨charrep⟩);
This function outputs the index to a device, whose character repeater is ⟨charrep⟩, in a form suitable for reinput directly by READINDEX.

¶*Method used.* The index is stored as a list of records in an array. The phrase is represented as a character strip in one component of the record, and the sequence of numbers as a list in the other component of the record.
LIB ALLSORT is used for the ordering process.

¶*Global variables used.* READSTRING LISTOSTR READNOS ORDERSTRING EQSTRING NAMEINDEX NOINDEX DESTINDEX LETTER CONSINDEX AORDER ABORDER COMPRESS NORDER SORTLETTER IOUTPUT STOUTPUT FORMAT and the globals in LIB ALLSORT.

¶*Store used.* The program occupies approximately 4 blocks of store.

¶*Examples.* (a) to read in an index.

```
:NEWINDEX();                              (Initialize workspace)
:READINDEX(CHARIN);                       (Read index from console)

:HILBERT, DAVID; 1 2 3;                   (Type in index)
:KLEENE, STEPHEN; 2;
:LESNIEWSKI; 2;
:LORENZEN, PAUL; 3;
:FRIEDBERG; 3;
:KLEENE, STEPHEN; 3;
:MUCNIK; 4;
:HILBERT, DAVID; 4 5;
:*                                        (Terminate with asterisk)
:MAKEINDEX(CHAROUT);                      (Output index to console)

F                                         (The ordered index)
FRIEDBERG            3

H
HILBERT, DAVID       1 2 3 4 5

K
KLEENE, STEPHEN      2 3

L
LESNIEWSKI           2
LORENZEN, PAUL       3

M
MUCNIK               4

:
```

(b) to store an index on, for example, paper tape.

```
:STOREINDEX( POPMESS([PTOUT INDEX]));
```

to read the index from paper tape

```
:NEWINDEX();
:READINDEX(POPMESS([PTIN INDEX]));
:
```

```
[INDEX]

VARS GOBBLE READSTRING LISTOSTR READNOS NOINDEX NAMEINDEX
     DESTINDEX CONSINDEX AORDER READINDEX ORDERSTRING ABORDER
     NORDER COMPRESS SORTLETTER EQSTRING STOREINDEX STOUTPUT
     NEWINDEX MAKEINDEX IOUTPUT FORMAT  CUOUT2  CUOUT CUCHARIN  ;

VARS CHARCOUNT MARGIN; 0->CHARCOUNT;30->MARGIN;

FUNCTION READSTRING ; VARS A B C ; NIL->A;1->C;
 L: CUCHARIN()->B;
  IF B=26 THEN  IF NOT(CUCHARIN=CHARIN) THEN GOBBLE(CUCHARIN) CLOSE;
    SETPOP() EXIT;
  IF B=11 THEN IF C=0 THEN 16::A->A CLOSE; LISTOSTR(A) EXIT;
  IF NOT(B=16) THEN B::A->A;0->C;
    ELSEIF C=0 THEN B::A->A;C+1->C;
    ELSE C+1->C; CLOSE;
  GOTO L;
END;

FUNCTION LISTOSTR A; VARS B C ;
  LENGTH(A)->B;INITC(B)->C;
  APPLIST(A,LAMBDA X;X->SUBSCRC(B,C);B-1->B;END);
  C;
END;

FUNCTION READNOS;VARS A B READITEM;
  NIL->B;CUCHARIN.INCHARITEM->READITEM;
  L: .READITEM->A;
  IF A=";" THEN B EXIT;
  IF A.ISNUMBER THEN A::B->B; GOTO L CLOSE;
  'PLEASE TYPE PAGE NUMBERS'=>;.READNOS;
END;

RECORDFNS("INDEX",[0 0])->NOINDEX->NAMEINDEX->DESTINDEX->CONSINDEX;

FUNCTION AORDER LETTER ;VARS V W;
  L:
    READSTRING()->V;SUBSCRC(1,V)->W;
    IF W<33 OR W>58 THEN
    'PLEASE TYPE A SEQUENCE OF CHARACTERS BEGINNING WITH A LETTER'=>;
      1.NL; GOTO L; CLOSE;
  CONSINDEX(V,READNOS())::LETTER(W)->LETTER(W);
END;

FUNCTION READINDEX INCR;INCR->CUCHARIN;
  LL: AORDER(LETTER); GOTO LL;
END;

FUNCTION ORDERSTRING  A B; VARS AL BL I C IA IB ;
  DATALENGTH(A)->AL; DATALENGTH(B)->BL;1->I;
  IF AL<BL THEN AL ELSE BL CLOSE->C;
 L: IF I>C THEN AL<BL   EXIT;
  SUBSCRC(I,A)->IA; SUBSCRC(I,B)->IB;
  IF IA>IB THEN TRUE
    ELSEIF IA<IB THEN FALSE
    ELSE I+1->I; GOTO L; CLOSE;
END;

COMPILE([LIB ALLSORT].LIBRARY);

FUNCTION ABORDER LETTER; VARS I;33->I;
  L: IF I>58 THEN EXIT;
  ALLSORT(LETTER(I),LAMBDA X Y;ORDERSTRING(X.NAMEINDEX,Y.NAMEINDEX);
  END)->LETTER(I);1+I->I; GOTO L;
END;

FUNCTION NORDER NLIST;
  ALLSORT(NLIST,NONOP <  );
END;
```

```
FUNCTION FORMAT X;
  APPLIST(X.DATALIST,CUCHAROUT);
END;

FUNCTION COMPRESS LETTER I => NEWLETI;
  VARS  B C ;NIL->B;NIL->C;
  APPLIST(LETTER(I),LAMBDA X;IF B.NULL THEN [% X %]->B;
                    ELSEIF EQSTRING(B.HD.NAMEINDEX,X.NAMEINDEX)
                    THEN CONSINDEX(X.NAMEINDEX,
                    B.HD.NOINDEX<>X.NOINDEX)::NIL->B;
                    ELSE B<> C->C;[%  X%] ->B; CLOSE;
                  END);
  B<>C->NEWLETI;
END;

FUNCTION SORTLETTER LETTER ; VARS I;33->I;
 L: IF I>58 THEN EXIT;
  COMPRESS(LETTER,I)->LETTER(I);
  I+1->I;
  GOTO L;
END;

FUNCTION EQSTRING A B;EQUAL(DATALIST(A),DATALIST(B));
END;


FUNCTION STOREINDEX OUTCR ;VARS I ;
  OUTCR->CUCHAROUT;33->I;
 LL:IF I>58 THEN PR("*");1.NL;OUTCR(TERMIN); EXIT;
  APPLIST(LETTER(I),STOUTPUT);1+I->I;
  GOTO LL;
END;

FUNCTION STOUTPUT X ;
  X.NAMEINDEX.FORMAT;PR(";");
  APPLIST(X.NOINDEX,PR);PR(";");1.NL;
END;

FUNCTION CUOUT2 C;
  IF C=17 THEN 0->CHARCOUNT;CUOUT(C);
    ELSEIF C=63 THEN
      IF CHARCOUNT> MARGIN THEN 2.SP
        ELSE SP(MARGIN-CHARCOUNT) CLOSE;
    ELSE CUOUT(C);CHARCOUNT+1->CHARCOUNT; CLOSE;
END;


FUNCTION NEWINDEX ;
 NEWARRAY([33 58],LAMBDA X; NIL;END)->LETTER;
END;

FUNCTION MAKEINDEX OUTCR; VARS I LET;
  CUOUT2->CUCHAROUT;OUTCR->CUOUT;
  ABORDER(LETTER);SORTLETTER(LETTER);33->I;
 LL:IF I>58 THEN 3.NL;OUTCR(TERMIN); EXIT;
  LETTER(I)->LET;
  IF LET.NULL.NOT THEN 3.NL;CUCHAROUT(I);1.NL;CLOSE;
  APPLIST(LET,IOUTPUT);I+1->I; GOTO LL;
END;

FUNCTION IOUTPUT X;
  1.NL;X.NAMEINDEX.FORMAT;CUCHAROUT(63);
  APPLIST(X.NOINDEX.NORDER,LAMBDA Y;Y.PR;1.SP;END);
END;

FUNCTION GOBBLE XT ;
  LL:XT()->B;IF B=TERMIN THEN EXIT;GOTO LL;
END;


2.NL;
'LIB INDEX IS READY FOR USE'.PR;
2.NL;
```

Program name. LIB INVTRIG
Source. S. Arrell, Esk Valley College; *Date of issue.* February 1969.

¶*Description.* This package contains three inverse trigonometric functions ARCSIN, ARCTAN, and ARCCOS.

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

¶*Method used.* Evaluation is by a truncated Chebyshev polynomial expansion, and is accurate to within 4 significant figures.

¶*Global variables.* The program uses 2 functions:
ABS(X); the modulus of X
REALSIGN(X); the standard function SIGN

¶*Store used.* Approximately a $\frac{1}{2}$ block of store is used on the 4100.

```
[INVTRIG]

VARS ABS ARCTAN ARCSIN ARCCOS REALSIGN ;

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

FUNCTION REALSIGN X;
    ( X > 0 ) - ( X < 0 ) ;
END;

FUNCTION ARCTAN X;
    VARS A;X*X->A;
    IF ABS(X)>1.0 THEN (1.570793-ARCTAN(1/ABS(X)))*REALSIGN(X)
    ELSE X*(((((-0.0134222*A+0.0573305)*A-0.12111)*A+0.195589)*A
    -0.332989)*A+0.9999955); CLOSE;
END;

FUNCTION ARCSIN X;
    VARS A;;X*X->A;
    IF ABS(X)>0.7071 THEN (1.5707963-ARCSIN(SQRT(1-A)*REALSIGN(X)))
    ELSE X*1.4142136*(((((0.0681325*A-0.0171236)*A+0.0430539)*A
    +0.0515667)*A+0.117918)*A+0.7071063);CLOSE;
END;

FUNCTION ARCCOS X;
    IF ABS(X)>0.7071 THEN ARCSIN(SQRT(1-X*X)*REALSIGN(X)) ELSE
    1.571-ARCSIN(X); CLOSE;
END;
```

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

¶*Description.* KALAH is an old Arabic game which is normally played using holes dug out of the sand, and pebbles. This program plays a game with the user, requesting moves, automatically replying, and displaying the board state between each move.

¶*How to use program.* Program should be compiled by typing:
COMPILE(LIBRARY([LIB KALAH])):
The program will ask several questions of the user, and will, on request, explain the game fully. Please terminate all replies with carriage-return/line-feed.

A variable, DEPTH, may be used to control the degree of difficulty of the game played by the computer. DEPTH has the initial value 2, and this produces a reasonable game for the beginner, the machine taking less than 30 seconds to make a move. If the value of DEPTH (which must be integral) is increased, the move-time increases by a large factor.

¶*Method used.* The program makes its moves by a mini-maxing look-ahead procedure to a depth of DEPTH half-moves ahead, applying the alpha-beta heuristic to reduce the size of the search-tree. No storage of board positions is made. The evaluation function, in effect, does a half move as well, and the value of the position is calculated from a multiple of the difference between the number of stones in the two Kalahs, together with a multiple of the expected differences after all possible moves have been made.

¶*Store required.* The program occupies approximately 5 blocks of store.

REFERENCES

Samuel, A. L. (1959) Some studies in machine learning using the game of checkers. *IBM. J. Res. Dev., 3,* 211-29.

Samuel, A. L. (1960) Programming computers to play games. *Advances in Computers, Vol. 1,* pp. 165-92. (ed. Alt, F. L.) New York and London Academic Press.

Russell, R. (1964) Kalah—the game and the program. *Stanford Artificial Intelligence Project Memo No, 22.* Stanford University.

```
[KALAH]

VARS RESIGN WIN RF MESS ENDGAME HALF P1 P2 VAL XXX N Q S T DEPTH ALPHA BETA
     MYSCORE YOURSCORE STONES DRAWS IWINS YWINS FREESTORE FREECHAIN
     GAMEOVER BOARD CHARS DUMP TOSUBSCR;


3 -> DEPTH;  -100000 -> ALPHA;  100000 -> BETA;  UPDATER(SUBSCR) -> TOSUBSCR;
NIL -> FREESTORE;  NIL -> FREECHAIN;



FUNCTION INPR X;
  IF X < 10 THEN SP(1); CLOSE;
  PR(X);
END;



FUNCTION NICEPR X;
  IF X=32 OR X=23 THEN EXIT
  CHARS(X);
END;



FUNCTION BPR;
VARS K;
  NL(1); SP(4); 13->K;
 ZB:
  INPR(BOARD(K)); K-1->K;
  IF K > 7 THEN GOTO ZB CLOSE;
  NL(1); INPR(BOARD(14)); SP(21);  PR(BOARD(7)); NL(1);
  1->K;  SP(4);
 ZC:
  INPR(BOARD(K)); K+1->K;
  IF K < 7 THEN GOTO ZC CLOSE; NL(1);
END;



FUNCTION MOVE; -> N;
  14, 7, IF N>7 THEN ->S ->T ELSE ->T ->S CLOSE;
  BOARD(N)-> Q; 0->BOARD(N); N+1->N;
 LOOP:
  IF N = S THEN GOTO SNOOP CLOSE;
 LOOP1:
  BOARD(N)+1 -> BOARD(N); Q-1->Q;
  IF Q THEN
   SNOOP:
    N+1-> N; IF N = 15 THEN 1 -> N; GOTO LOOP1 CLOSE;
    GOTO LOOP;
  CLOSE;
  IF N=T THEN TRUE EXIT;
  IF BOARD(N) = 1 AND N<T AND N>=T-6 AND BOARD(14-N) THEN
    BOARD(T)+BOARD(14-N)+1 -> BOARD (T); 0 -> BOARD(14-N);
    0 -> BOARD(N);
  CLOSE;
  FALSE
END;



FUNCTION NEWBOARD; VARS X;
  INIT(14)->X;
  TOSUBSCR(%X%);
  SUBSCR(%X%)->X;  ->UPDATER(X);
  X
END;
```

```
FUNCTION EXPLORE SIDE ALPHA BETA;
VARS BMOVE P DUMP;
  IF ATOM(FREESTORE) THEN
    NIL::FREECHAIN->FREECHAIN;
    .NEWBOARD->DUMP;
  ELSE
    FREESTORE.FRONT->DUMP;  FREESTORE.BACK;
    FREECHAIN->FREESTORE.BACK;  FREESTORE -> FREECHAIN;  ->FREESTORE;
  CLOSE;
  SIDE+6->P;  1->ENDGAME;
LOOP:
  IF BOARD(P) THEN
    14->N;
   SET:
    BOARD(N)->DUMP(N); N-1->N; IF N THEN GOTO SET CLOSE;
    0 -> ENDGAME;
    IF MOVE(P) THEN EXPLORE(SIDE,ALPHA,BETA)->XXX;
    ELSEIF DEPTH THEN
      DEPTH-1->DEPTH;
      EXPLORE(IF SIDE THEN 0 ELSE 7 CLOSE,ALPHA,BETA)->XXX;
      DEPTH+1->DEPTH;
    ELSE
      BOARD(14)->P1; BOARD(7)->P2;
      IF P1>=HALF OR P2>=HALF THEN .GAMEOVER->XXX; 2000*(MYSCORE-YOURSCORE);
        ELSE P1-P2 CLOSE
    CLOSE -> VAL;
    BOARD, DUMP->BOARD ->DUMP;
 TEST:
    IF SIDE THEN
      IF VAL<BETA THEN
       IF VAL>ALPHA THEN VAL->ALPHA; P->BMOVE CLOSE
      ELSE BETA, BMOVE; GOTO SAVESPACE CLOSE
    ELSE
      IF VAL>ALPHA THEN
        IF VAL<BETA THEN VAL->BETA; CLOSE
      ELSE ALPHA, BMOVE; GOTO SAVESPACE CLOSE
    CLOSE
  CLOSE;
  P-1 -> P;  IF P>SIDE THEN GOTO LOOP CLOSE;
  IF ENDGAME THEN
    0->ENDGAME;
    .GAMEOVER->XXX; 2000*(MYSCORE-YOURSCORE)->VAL;  GOTO TEST
  CLOSE;
  IF SIDE THEN ALPHA ELSE BETA CLOSE, BMOVE;
 SAVESPACE:
  DUMP->FREECHAIN.FRONT;  FREECHAIN.BACK;  FREESTORE->FREECHAIN.BACK;
  FREECHAIN->FREESTORE;  ->FREECHAIN;
END;



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



FUNCTION MOVIN X;
VARS DUMP;
  NL(2);  PR(X);
 IN:
  READNEXT()->X;
  IF X = "HELP" THEN
    .NEWBOARD->DUMP;
    14->N;
   SET:
    IF N<8 THEN BOARD(N+7) ELSE BOARD(N-7) CLOSE -> DUMP(N);
    N-1->N;  IF N THEN GOTO SET CLOSE;
    DUMP,BOARD->DUMP->BOARD;
    EXPLORE(7,ALPHA,BETA) -> X;  .ERASE;  NL(1);
    PR('TRY MOVE'); PR(X-7);
    DUMP -> BOARD;  GOTO IN
  ELSEIF X="RESIGN" THEN 1->RESIGN; FALSE EXIT;
  IF NOT(ISINTEGER(X)) OR X<1 OR X>6 OR BOARD(X)=0 THEN
    PR('ILLEGAL MOVE - TRY AGAIN');  GOTO IN;
  CLOSE;
  MOVE(X);  BPR();
END;
```

```
FUNCTION GAMEOVER;
  6->N;  0->MYSCORE;  0->YOURSCORE;
 L0:
  MYSCORE + BOARD(N+7) -> MYSCORE;
  YOURSCORE + BOARD(N) -> YOURSCORE;
  N-1->N;  IF N THEN GOTO L0 CLOSE;

  IF MYSCORE=0 OR YOURSCORE=0 OR BOARD(14)>=HALF
  OR BOARD(7)>=HALF OR RESIGN THEN
    TRUE;
  ELSE
    FALSE
  CLOSE;
  MYSCORE+BOARD(14)->MYSCORE;  YOURSCORE+BOARD(7)->YOURSCORE;
END;



FUNCTION PLAY;
  0 -> WIN;  0 -> RESIGN;  .BPR;
  PR('
DO YOU WANT TO START');
  IF .READNEXT="NO" THEN 'MY FIRST MOVE:' -> MESS;  GOTO MYMOVE CLOSE;
 LOOP:
  'YOUR MOVE' -> MESS;
 LOOP1:
  IF GAMEOVER() THEN GOTO STOPPE CLOSE;
  IF MOVIN(MESS) THEN 'YOU MOVE AGAIN' -> MESS;  GOTO LOOP1 CLOSE;
  'MY MOVE:' -> MESS;
 MYMOVE:
  IF GAMEOVER() THEN GOTO STOPPE CLOSE;
  NL(2); PR(MESS); EXPLORE(7,ALPHA,BETA) ->RF; ->XXX;
  IF XXX>999 AND NOT(WIN) THEN
    PR(' I AM GOING TO WIN, MOVE:'); 1->WIN;
  ELSEIF XXX< -999 THEN
    PR(' I RESIGN.


*YOU WIN*'); YWINS+1->YWINS;
    GOTO GAMESTOT;
  CLOSE;
  PR(RF-7); NL(2);
  IF BPR(MOVE(RF)) THEN
    'ME AGAIN:' -> MESS;  GOTO MYMOVE
  ELSE
    GOTO LOOP
  CLOSE;

 STOPPE:
  IF RESIGN THEN
    IF ERASE(EXPLORE(7,ALPHA,BETA))<999 THEN
      PR('

I THOUGHT I COULD STILL LOOSE, NEVERTHELESS -');
    ELSE
      PR('

VERY WISE');
    CLOSE;
    PR('

*I WIN*');IWINS+1->IWINS;
    GOTO GAMESTOT;
  CLOSE;
PR('


GAME FINISHED.  MY SCORE IS'); PR(MYSCORE);
  PR('.  YOUR SCORE IS'); PR(YOURSCORE);  PR('.

');
  NL(2);
  PR(IF MYSCORE=YOURSCORE THEN
      '*WE DRAW*';  DRAWS+1->DRAWS;
     ELSEIF MYSCORE>YOURSCORE THEN
      '*I WIN*';  IWINS+1->IWINS;
     ELSE '*YOU WIN*'; YWINS+1->YWINS;
     CLOSE);
 GAMESTOT:
  PR('
```

```
GAMES TO ME -'); PR(IWINS); PR('.  GAMES TO YOU -');
  PR(YWINS);  PR('.  DRAWS -'); PR(DRAWS); PR('.


');
END;




FUNCTION CONTINUE B;
VARS N P T CUCHAROUT;;
  CUCHAROUT->CHARS;  NICEPR->CUCHAROUT;
  0->DRAWS;  0->IWINS;  0->YWINS;  0->T;
  .NEWBOARD->BOARD;  1->N;
 SET:
  NEXT(B)->B->P;  P+T->T;  P->BOARD(N);
  N+1->N;  IF N<15 THEN GOTO SET CLOSE;
  INTOF(T/2)->HALF;  PLAY();
END;




VARS OPERATION 1 KALAH;




FUNCTION KALAH;
VARS START X CUCHAROUT;
  0->DRAWS; 0->IWINS; 0->YWINS;
  CUCHAROUT->CHARS; NICEPR->CUCHAROUT;
  PR('

PLEASE TERMINATE ALL REPLIES WITH RETURN/LINE-FEED.

DO YOU KNOW HOW TO PLAY THE GAME');
  IF READNEXT()="NO" THEN
    PR('


                   ME
      6    5    4    3    2    1
  K                                  K
      1    2    3    4    5    6
                  YOU
');  PR('

    THE ABOVE IS A DRAWING OF THE KALAH BOARD.  EACH PLAYER HAS
A ROW OF SIX "PITS" IN FRONT OF HIM, NUMBERED AS SHOWN, AND A
KALAH PIT TO HIS RIGHT.  INITIALLY ALL THE PITS HAVE AN EQUAL'); PR('
NUMBER OF STONES IN THEM EXCEPT THE KALAHS, WHICH ARE EMPTY.
    A MOVE CONSISTS SIMPLY OF TAKING STONES FROM ONE OF YOUR OWN
PITS, AND DISTRIBUTING THEM ANTI-CLOCKWISE ONE BY ONE INTO THE
OTHER PITS, INCLUDING YOUR OWN KALAH, BUT NOT YOUR OPPONENTS.'); PR('
    THE TWO RULES ARE:-
       1)  IF YOUR LAST STONE LANDS IN YOUR KALAH, YOU TAKE
           ANOTHER MOVE,'); PR('
       2)  IF YOUR LAST STONE LANDS IN AN EMPTY PIT ON YOUR OWN
           SIDE, THAT STONE, TOGETHER WITH ALL THE STONES IN THE
           PIT OPPOSITE, ARE PUT IN YOUR KALAH.');  PR('

    THE OBJECT OF THE GAME IS TO COLLECT AS MANY STONES IN YOUR
KALAH AS POSSIBLE,  THE GAME BEING OVER WHEN EITHER PLAYER HAS
NO STONES LEFT TO MOVE, OR WHEN ONE PLAYER HAS MORE THAN HALF'); PR('
THE TOTAL NUMBER OF STONES IN HIS KALAH.  THE TOTAL SCORE IS
THE NUMBER OF STONES IN YOUR KALAH PLUS THE SUM OF THE STONES LEFT
IN THE PITS ON YOUR SIDE.'); PR('

    A MOVE IS MADE BY TYPING THE NUMBER OF THE PIT FROM WHICH YOU
WISH TO PLAY (SEE BOARD ABOVE).  YOU MAY ASK THE COMPUTER TO
SUGGEST YOUR BEST MOVE BY TYPING HELP.
    IF YOU WISH TO RESIGN TYPE RESIGN.
```

```
   SIX STONES PER PIT IS CONSIDERED THE BEST GAME, THREE PER PIT
GIVING A GOOD BEGINNERS GAME.
');
  CLOSE;
 L0:
  PR('
HOW MANY STONES PER PIT WOULD YOU LIKE');
  .READNEXT->STONES;  STONES*6->HALF;
  IF NOT(ISINTEGER(STONES)) THEN PR('COME NOW'); GOTO L0 CLOSE;
  .NEWBOARD->BOARD;
  14->N;
 SET:
  STONES->BOARD(N); N-1->N; IF N THEN GOTO SET CLOSE;
  0->BOARD(7);  0->BOARD(14);
  PLAY();
  PR('WOULD YOU LIKE ANOTHER GAME');
  IF .READNEXT="YES" THEN GOTO L0 ELSE PR('
BACK TO POP-2 THEN.

'); CLOSE
END;

PR('


TO ENTER PROGRAM TYPE  KALAH;


');
```

Program name. LIB MATRIX
Source. R. J. Popplestone, DMIP; *Date of issue.* December 1968.

¶*Description.* This program provides several operations on functions treated as matrices. These are matrix addition, subtraction, multiplication, and scalar multiplication, a function which calculates the determinant of a square matrix, and functions for creating, inputting, and outputting matrices.

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

The following functions will then be available:

NEWMATRIX(X, Y, F);	Creates a matrix with dimensions 1-X and 1-Y from the function F.
MATPR(A);	Prints the matrix A on the current output device.
READMAT( );	Forms a matrix from information typed in. This should be in the form: number of rows, number of columns, semi-colon, the values of the elements of the matrix typed in row by row, and terminated by a semi-colon.
A ++B;	Adds the matrices A and B.
A —— B;	Subtracts the matrix B from the matrix A.
A ** B;	Multiply the matrix A by the matrix B. <u>Note</u>. ++ and —— are operations of precedence 5, and ** is an operation of precedence 4.
SUBMAT(ILO, IHI, JLO, JHI, A);	Produces the submatrix of A which consists of the partition between rows ILO and IHI, and the columns JLO and JHI.
HMULT(A, B);	Horizontal concatenation of A and B.
VMULT(A, B);	Vertical concatenation of A and B.
SMULT(X, A);	Scalar multiplication of A by the number X.
DET(A);	Produces the determinant of the square matrix A.

¶*Method used.* NEWMATRIX creates a new function from the one supplied, setting its FNPROPS to contain the dimensions of the matrix. This is in the format, [X ROWS COLS], where X is the name of the function (which can be overwritten, or used by SPEC), ROWS is the number of rows, and COLS the number of columns in the matrix.

¶*Errors.* If the arguments given to any of the functions are of the wrong type, an error message is printed, and SETPOP is entered.

For example, if A is a non-square matrix:
DET(A);
** 'DET ERROR. NON SQUARE MATRIX',
SETPOP:

¶*Global variables used.* FUNTOMAT MATPR ROWNO COLNO DLISTOF ++—— ** SIGMA SUBMAT HMULT VMULT QUAD DELROWCOL DET READMAT COPYMAT SMULT NEWMATRIX.

¶*Store used.* The program occupies some 3.5 blocks of store.

¶*Examples of use*

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

READMAT();

4 4;
2 4 5 3
1 1 9 4
3 4 1 7
3 0 8 0;  -> A;

MATPR(A);

2 4 5 3
1 1 9 4
3 4 1 7
3 0 8 0

SMULT(0.49, A) -> C;
A ++C -> B;
MATPR(B);

2.98 5.96   7.45   4.47
1.49 1.49  13.41   5.96
4.47 5.96   1.49  10.43
4.47 0     11.92   0

SMULT(0.04, B) -> D;
D ** A -> E;
MATPR(E);

1.907  1.907  4.47   3.397
2.503  2.444  3.278  4.172
2.026  1.192  6.437  1.907
1.788  2.622  1.371  3.874

SUBMAT(1, 4, 3, 4, E) -> F;
MATPR(F);

4.47   3.397
3.278  4.172
6.437  1.907
1.371  3.874

MATPR(HMULT(A, F));

2 4 5 3 4.47   3.397
1 1 9 4 3.278  4.172
3 4 1 7 6.437  1.907
3 0 8 0 1.371  3.874

DET(A) =>

** 'DET ERROR.  NON SQUARE MATRIX'
SETPOP:

DET(E) =>

** 4.073
```

```
[MATRIX]


VARS COLNO ROWNO DLISTOF SMULT COPYMAT;

FUNCTION FUNTOMAT DLIST F;
  F(% %) -> F;
  IF F.FNPROPS.ATOM THEN F.FNPROPS::NIL->F.FNPROPS CLOSE;
  DLIST -> F.FNPROPS.TL;  F
END

FUNCTION NEWMATRIX X Y F;
  FUNTOMAT([%X,Y%],F);
END;


FUNCTION MATPR F;
VARS MAXI MAXJ;  F.ROWNO -> MAXI;  F.COLNO -> MAXJ;
  2.NL;  VARS I J;
  1 -> I;
L0: IF I > MAXI THEN EXIT
  2.NL;  1 -> J;
L1: IF J > MAXJ THEN I + 1 -> I; GOTO L0 CLOSE;
  PR(F(I,J));  J + 1 -> J;  GOTO L1
END


FUNCTION ROWNO A;
  A.FNPROPS.TL.HD
END


FUNCTION COLNO A;
  A.FNPROPS.TL.TL.HD
END

LAMBDA; .FNPROPS.TL END ->DLISTOF;


VARS OPERATION 5 (++ --) OPERATION 4 **;


FUNCTION ++ A B;
     IF EQUAL(DLISTOF(A) , DLISTOF(B)) THEN
       FUNTOMAT(DLISTOF(A),
         LAMBDA I J F G;  F(I,J) + G(I,J) END (%A, B%))
     EXIT

     [++ ERROR] => ;  .SETPOP;
END


FUNCTION -- A B;
     IF EQUAL(DLISTOF(A), DLISTOF(B)) THEN
       FUNTOMAT(DLISTOF(A),
         LAMBDA I J F G;  F(I,J) - G(I,J) END (%A, B%))
     EXIT

     [-- ERROR] => .SETPOP
END


FUNCTION SIGMA ISIG FSIG; VARS SSIG;  0 -> SSIG;
L0: IF ISIG = 0 THEN SSIG EXIT
     FSIG(ISIG) + SSIG -> SSIG;  ISIG - 1 -> ISIG;  GOTO L0
END


FUNCTION ** A B;
     IF A.ISNUMBER THEN SMULT(A,B) EXIT;
     IF B.ISNUMBER THEN SMULT(B,A) EXIT;
     IF A.COLNO = B.ROWNO THEN
       FUNTOMAT([%A.ROWNO, B.COLNO%],
         LAMBDA I J F G;
           SIGMA(F.COLNO,  LAMBDA K; F(I,K) * G(K,J) END)
         END(% A, B%)    )
     EXIT

     [** ERROR] =>  .SETPOP
END
```

```
FUNCTION SUBMAT ILO IUP JLO JUP A;
    ILO - 1 -> ILO;  JLO - 1 -> JLO;
    FUNTOMAT([% IUP-ILO, JUP-JLO%],
     LAMBDA I J ILO JLO A;  A(I+ILO, J+JLO) END (%ILO, JLO, A%)  )
END



FUNCTION  HMULT A B;
    IF A.ROWNO = B.ROWNO THEN
      FUNTOMAT([%A.ROWNO, A.COLNO + B.COLNO %],
        LAMBDA I J OFFSET A B;
          IF J =< OFFSET THEN A(I,J) ELSE B(I,J-OFFSET) CLOSE
        END (%A.COLNO, A, B%)  )
    EXIT
    [HMULT ERROR] => .SETPOP
END



FUNCTION VMULT A B;
    IF A.COLNO = B.COLNO THEN
      FUNTOMAT([%A.ROWNO + B.ROWNO,  A.COLNO%],
        LAMBDA I J OFFSET A B;
          IF I =< OFFSET THEN A(I,J) ELSE B(I-OFFSET,J) CLOSE
        END (% A.ROWNO, A, B %) )
    EXIT
    [VMULT ERROR] => .SETPOP
END



FUNCTION QUAD A B C D;
    HMULT(VMULT(A,C), VMULT(B,D))
END



FUNCTION DELROWCOL I J A;
    QUAD(SUBMAT(1,I-1, 1,J-1, A),  SUBMAT(1,I-1,  J+1, A.COLNO, A),
      SUBMAT(I+1, A.ROWNO,  1, J-1, A),  SUBMAT(I+1, A.ROWNO,  J+1, A.COLNO, A))

END



FUNCTION DET A;
    IF A.ROWNO = A.COLNO THEN
      IF A.ROWNO = 1 THEN A(1,1) ELSE
      SIGMA(A.COLNO,  LAMBDA I; (-1)↑(I+1) * DET(DELROWCOL(1,I,A)) * A(1,I) END)

      CLOSE
    EXIT
    [DET ERROR] => .SETPOP;
END



FUNCTION READMAT;
VARS A IMAX JMAX I J;
  1 -> I;  ITEMREAD() -> IMAX;  ITEMREAD() -> JMAX;

  IF NOT (ITEMREAD() = ";") THEN [ITEMREAD ERROR] => .SETPOP CLOSE;
  COPYMAT(FUNTOMAT([%IMAX, JMAX%],  LAMBDA I J; 0 END)) -> A;
 L0:
  IF I > IMAX THEN IF ITEMREAD() = ";" THEN A EXIT [READMAT ERROR] => A EXIT;

  1 -> J;
 L1:
  IF J > JMAX THEN I + 1 -> I;  GOTO L0 CLOSE;
  ITEMREAD() -> A(I,J);  J+1 -> J;  GOTO L1
END
```

```
VARS OPERATION 7 ==> ;  MATPR -> NONOP ==> ;



FUNCTION COPYMAT A;
  FUNTOMAT(A.DLISTOF, NEWARRAY([%1, A.ROWNO, 1, A.COLNO %], A)) -> A;
  A.FNPROPS -> A.FNPART.FNPROPS;  A.FNPART
END



FUNCTION SMULT X A;
  FUNTOMAT(A.DLISTOF,  LAMBDA I J X A;  X*A(I,J) END (%X, A%) )
END

PR('

LIB MATRIX NOW READY FOR USE

');
```

Program name. LIB MEMOFNS
Source. D. Michie, R. J. Popplestone, D. L. Marsh, DMIP; *Date of issue.* December 1968.

¶*Description.* This is a revised version of the memo-function facilities proposed by D. Michie (1967, 1968) and implemented by R. J. Popplestone (1967).

Purpose. To improve function evaluation time by the automatic storage of previously computed results. For a more general discussion see the above references.

Operation of a memo-function. Each memo-function has a rote attached to it. This rote is composed of argument-result pairs which have been derived from previous evaluations of the function. Before each new argument is evaluated, the rote is searched to see whether that argument has occurred in some previous evaluation. If it has, the function result is then taken from the rote, otherwise the function is evaluated and the new argument-result pair is inserted on the rote.

New pairs are inserted at the top of the rote, which is limited to a certain size. When it exceeds that size, pairs are removed from the bottom of the rote.

Whenever an argument-result pair is used, that pair is promoted one place on the rote. This means that the more frequently used arguments are nearer the top of the rote.

Recursive functions. A memoized recursive function distinguishes between two types of function calls. Top-level calls are those whose arguments are provided from outside the function body, and other calls are those from within the body of the recursive function itself.

The rote is searched at every call of the function, but only the argument and result from the top-level call of the function will be inserted in the rote. This ensures that the distribution of argument values in the rote will reflect the distribution of the main function arguments, and will not be skewed by the sub-arguments resulting from recursion.

Economies. The memo-apparatus may be effectively applied to functions under the following conditions:

(a) The average function evaluation time is greater than 100msecs for recursive functions, or greater than 200msecs for non-recursive functions.

(b) The function will be applied to not more than about 100 different arguments.

(c) The function is used frequently enough to allow the rote to grow to the maximum specified size.

With rotes containing about 80 per cent of the possible number of argument values, 10-fold speed-ups in evaluation time can result.

Very generally, it was found that, with a numerical function (log factorial) where the arguments were distributed both uniformly and normally, a rote size equal to the standard deviation of the argument population produced a 3-fold speed-up with a recursive definition, and a 1.2-fold speed-up with an iterative definition. Larger speed-ups were associated with larger rotes.

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

Creation of a memo-function. The function NEWMEMO is provided for this:

NEWMEMO ∈ F, N, EQUIV => memo-function.

1. Arguments

F — The function to be memoized. This must be a function which takes one argument and produces one result.

N — The maximum size to which the rote is allowed to grow.

EQUIV — The equality function to decide whether two arguments are identical, for example, NONOP= for numbers, EQUAL for lists.

2. Results

The result of the function NEWMEMO is a memo-function with a rote of a specified size. This result must be assigned back to the variable which contained the original function.

3. Example of the use of NEWMEMO
NEWMEMO(PRIME, 20, NONOP =) —> PRIME;
NEWMEMO(NUMSORT, 30, EQUAL) —> NUMSORT;
where NUMSORT is a function to sort a list of numbers into ascending order.

Subsidiary functions

ISMEMO(F); Produces the result TRUE if the function F is a memo-function, otherwise FALSE.

UNMEMO(F); Produces the original function from a memo-function F. n.b. UNMEMO is destructive and once applied to a memo-function, that memo-function should not be used again.

DICTOF(F); Produces the rote of the memo-function F. The rote is implemented as a strip. This may be printed using the standard function DATALIST, or with PRDICTOF.

PRDICTOF(F); Prints the rote of the memo-function F. Each argument-result pair is output on a new line.

Functions used in the implementation. The rote is implemented as a POP-2 strip with the argument-result pairs as POP-2 Pairs.

When a memo-function is created, all the storage required by the rote is claimed. Care must therefore be taken not to specify a rote size greater than that actually required.

ASSOCVAL(X, ROTE, EQUIV, N); Finds the item associated with the item "X" under the equivalence "EQUIV" in the rote "ROTE" of size "N".
ASSOCUD(Y, X, ROTE, N); Associates the items "X" and "Y" (argument and result) on the rote "ROTE" of size "N".
NEWASSOCFN(N, EQUIV); Produces a rote of size "N" whose selector function is ASSOCVAL and whose updater function is ASSOCUD.

DOMEMO(X, F, A); The general purpose memo-function which NEWMEMO converts to a particular memo-function by partially applying the function "F" and a rote "A".

Storage requirements. The memo apparatus occupies approximately 9 sectors on the disc and 2 blocks of core store. Once the program is in store the only overheads for each function memoized will be the rote.

¶*Errors.* A memo-function should be redefined if a run time error occurs. This is because the FNPROPS of the functions are used to indicate whether a particular function call is at top level or not. Consequently, when an error occurs during the evaluation of a recursive function which has been memoized, this indicator will be left in the wrong state, and subsequent results may not be left on the rote.

¶*Global variables used.* NEWMEMO, UNMEMO, ISMEMO, DICTOF, PRDICTOF.

¶*Examples of the use of the program.* Consider the function log-factorial:

```
FUNCTION LOGFACT N;
IF N=1 THEN 0 ELSE LOG(N)+LOGFACT(N-1) CLOSE
END;
```

This may be memoized with a rote size of, say, 4:

```
NEWMEMO(LOGFACT, 4, NONOP=) -> LOGFACT;
```

There is no change in the way it is called:

```
LOGFACT(5) =>
** 4.787,
```

This call will, however, have resulted in an entry in the rote:

```
PRDICTOF(LOGFACT);
[5 . 4.787]
```

Further calls will extend the rote:

```
LOGFACT(3);  LOGFACT(6) =>
** 1.792, 6.579,
PRDICTOF(LOGFACT);
[6 . 6.579]
[5 . 4.787]
[3 . 1.792]
```

Evaluation of LOGFACT(4); will terminate when the argument 3 is found on the rote, thus saving three function calls. The rote entry for argument 3 will be promoted by one place:

```
PRDICTOF(LOGFACT);
[4 . 3.178]
[6 . 6.579]
[3 . 1.792]
[5 . 4.787]
```

The rote is now full size. Further entries will cause the bottom entry to be lost:

```
LOGFACT(7) =>
**  8.525,
PRDICTOF(LOGFACT);
[7 . 8.525]
[6 . 6.579]
[4 . 3.178]
[3 . 1.792]
```

Obviously:

```
ISMEMO(LOGFACT) =>
** 1,
```

Finally, to obtain the original log-factorial function:

```
UNMEMO(LOGFACT) -> LOGFACT;
```

R E F E R E N C E S

Michie, D. (1967) Memo functions: a language facility with 'rote-learning' properties. *Research Memorandum MIP-R29* Edinburgh: Department of Machine Intelligence and Perception.

Michie, D. (1968) Memo functions and Machine Learning. *Nature, 218,* 19-22.

Popplestone, R. J. (1967) Memo functions and the POP-2 language. *Research Memorandum MIP-R-30* Edinburgh: Department of Machine Intelligence and Perception.

```
[MEMOFNS]

FUNCTION ASSOCVAL X ROTE EQUIV N;
VARS P P0 U;
SUBSCR(N+1,ROTE)->P; P->P0;
L1: SUBSCR(P,ROTE)->U;
    IF EQUIV(X,FRONT(U)) THEN BACK(U);
        IF NOT(P=P0) THEN P-1->P0;
           IF P0=0 THEN N->P0; CLOSE;
           SUBSCR(P0,ROTE),U, ->SUBSCR(P0,ROTE); ->SUBSCR(P,ROTE);
        CLOSE;
    EXIT
P+1->P;
IF P>N THEN 1->P;CLOSE;
IF (P=P0) OR (FRONT(U)=UNDEF)  THEN UNDEF EXIT
GOTO L1
END


FUNCTION ASSOCUD Y X ROTE N;
VARS P U;
SUBSCR(N+1,ROTE)->P;
IF P=1 THEN N->P; ELSE P-1->P; CLOSE;
X->FRONT(SUBSCR(P,ROTE));
Y-> BACK(SUBSCR(P,ROTE));
P-> SUBSCR(N+1,ROTE);
END

FUNCTION NEWASSOCFN N EQUIV;
VARS U V N1;
INIT(N+1)->U; 1->SUBSCR(N+1,U);
1->N1;
L1: IF N1=<N THEN UNDEF::UNDEF->SUBSCR(N1,U); N1+1->N1; GOTO L1 CLOSE;
ASSOCVAL(% U,EQUIV,N %)->V;
ASSOCUD(%  U,N %)->UPDATER(V);
V
END


FUNCTION DOMEMO X F A;
VARS Y;
A(X)->Y;
IF F.FNPROPS.TL.HD THEN IF  Y = UNDEF THEN FALSE -> F.FNPROPS.TL.HD;
  X.F->Y; Y->A(X);
  TRUE->F.FNPROPS.TL.HD;
  CLOSE;;
  Y;
  ELSE IF Y = UNDEF THEN F(X); ELSE Y
  CLOSE;
CLOSE;
END


FUNCTION NEWMEMO F N EQUIV;
VARS U V;
   IF F.FNPROPS.ATOM THEN F.FNPROPS::NIL->F.FNPROPS CLOSE;
    [% 1, TL(FNPROPS(F)) %] ->TL(FNPROPS(F));
    NEWASSOCFN(N,EQUIV,) -> U;
    DOMEMO(% F, U, %) -> V;
    UPDATER(U) -> UPDATER(V);
IF V.FNPROPS.ATOM THEN V.FNPROPS::NIL ->V.FNPROPS CLOSE;
    "MEMO"::TL(FNPROPS(V)) ->TL(FNPROPS(V));
    V
END

FUNCTION ISMEMO F;
IF F.FNPROPS.ATOM THEN FALSE EXIT
IF F.FNPROPS.TL.NULL THEN FALSE EXIT
 F.FNPROPS.TL.HD = "MEMO"
END


FUNCTION UNMEMO F; VARS MF;
IF F.ISMEMO THEN
 FROZVAL(1,F)->MF;
    IF MF.FNPROPS.TL.NULL.NOT THEN TL(TL(FNPROPS(MF))) -> TL(FNPROPS(MF));
    CLOSE; MF
  ELSE UNDEF
CLOSE;
END
```

```
FUNCTION DICTOF F;
IF F.ISMEMO THEN FROZVAL(1,FROZVAL(2,F)) ELSE UNDEF CLOSE;
END


FUNCTION PRDICTOF F;
VARS P P1 N STRIP;
F.DICTOF->STRIP;
1.NL; IF STRIP=UNDEF THEN UNDEF.PR; EXIT
FROZVAL(3,FROZVAL(2,F))->N;
SUBSCR(N+1,STRIP)->P; 1->P1;
L1: IF HD(SUBSCR(P,STRIP))=UNDEF THEN EXIT
PR(SUBSCR(P,STRIP)); 1.NL; P+1->P; P1+1->P1;
IF P1 > N THEN EXIT
IF P > N THEN 1->P; CLOSE;
GOTO L1
END


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

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

¶*Description.* This program provides a new kind of data structure system more versatile than POP-2 arrays or records, although less efficient. This system can also be used to provide simple 'memo-functions' (see library program LIB MEMOFNS).

The facilities provided, are *non-numerical arrays, function arrays, pseudo-records*, and *function-components*. They are each described separately, although their implementations are very similar and are trivial applications of *association sets,* defined by LIB ASSOC which is used by this package.

¶*Non-numerical arrays.* A POP-2 array provides a means of storing a fixed amount of information indexed by integers. It is sometimes convenient to index by other items (for example, by names when referring to people) and to have no restrictions on the ultimate size of the array (so that, for example, more people can be added to the array). The *non-numerical arrays* described here provide these facilities.

A non-numerical-array (n-n-array) is a function which is similar to a POP-2 array in that it is a doublet which maps a subscript onto a component, but dissimilar in that subscripts are not restricted to integers, but can be items of any type. The array is of unlimited length, but only one dimension. The function NEWNNARRAY produces a new n-n-array with, as yet, no subscript-component 'pairs'. These 'pairs' are added by using the n-n-array in the update sense. An attempt to select from a subscript-component pair which does not yet exist produces the failure result UNDEF.
NEWNNARRAY ∊ () => (SUBSCRIPT ==> COMPONENT or UNDEF)
The contents of an n-n-array may be printed using the function PRNNARRAY
PRNNARRAY ∊ N-N-ARRAY => ()

Example 1

```
: COMPILE(LIBRARY([LIB NEW STRUCTURES]));
: VARS DMIP; NEWNNARRAY() -> DMIP;      (creates a new, empty,
                                         n-n-array)
: "MICHIE" -> DMIP("CHAIRMAN");         (adds items to the array.
                                         MICHIE is indexed by
                                         "CHAIRMAN").
: 'FORREST HILL' -> DMIP("ADDRESS");
: DMIP("CHAIRMAN") =>                   (selects item indexed by
                                         "CHAIRMAN")
**MICHIE,
: "GREGORY" -> DMIP("CHAIRMAN");        (updates the item indexed
                                         by "CHAIRMAN")
: DMIP("ROBOT") =>                      (attempt to select an item
                                         not yet in the array)
**UNDEF,
: "FREDDY" -> DMIP("ROBOT");            (adds a new item to the
                                         array)
: PRNNARRAY(DMIP);                      (prints the contents of the
                                         array)

    CHAIRMAN    GREGORY
    ADDRESS     'FORREST HILL'
    ROBOT       FREDDY
```

It is not always convenient to have the failure result UNDEF. For example, an n-n-array might be used to store functions, and a function might therefore be a more appropriate failure result. If the n-n-array is defined using NEWNNFARRAY then a different result for the failure case can be specified by the user. (NNF stands for non-numerical with specified failure result.)

```
NEWNNFARRAY ϵ FAIL RESULT => (SUBSCRIPT ==> COMPONENT)
                                        (OR FAILRESULT)
```

Example 2

```
: VARS POPLIB;
: NEWNNFARRAY(LAMBDA; 'NO SUCH PROGRAM'. PR; TERMIN;
  END; ) —> POPLIB;                        (creates a
                                            n-n-array with
                                            function as the
                                            failure result)
: LIBRARY([LIB POPEDIT]) —> POPLIB ("EDIT"); (the LIB POPEDIT
                                            file, indexed by
                                            "EDIT" is added)
: COMPILE(POPLIB("EDIT"));                  (The library file
  (followed by the usual output);           LIB POPEDIT is
                                            compiled)
: COMPILE(POPLIB("ABC"));                   (attempt to select
  'NO SUCH PROGRAM'                          and compile an
                                             item which is not
                                             in the array. The
                                             fail result is a
                                             function and so
                                             Error 37 is
                                             avoided.)
```

¶*Function-arrays.* Michie, Popplestone, and March describe memo-functions elsewhere. The data structures described now are a kind of simplified memo-function, omitting the rote optimization and forgetting facilities. They consist of a non-numerical array, together with a function which computes the component value for any subscript not occurring in the array. The newly-computed value is stored in the array. Thus they generalize the previous concept of a non-numerical array by producing a result which, in the failure case, depends on the subscript, rather than being a fixed result. They are produced by the function:

```
NEWFNARRAY ϵ (SUBSCRIPT => COMPONENT) => (SUBSCRIPT ==>
                                          COMPONENT)
              the function to be                  function-
              incorporated in the function part   array
```

Example 3

The use of function arrays as simple memo-functions.

```
: FUNCTION LOGFACT N;
: IF N=1 THEN 0 ELSE LOG(N)+LOGFACT(N—1); CLOSE;
: END;
: NEWFNARRAY(LOGFACT) —> LOGFACT;        (converts LOGFACT
                                          to a function array)
: LOGFACT(5) =>                           (computes the result
                                          and adds it to the
** 4.787,                                 array)
: PRNNARRAY(LOGFACT);                     (prints the contents
                                          of the array)
  5     4.787
```

Example 4

The use of function arrays to construct data structures. (In this example we have a class of students. A student is a data structure, and we want to be able to update it without having to worry about whether we have heard of the student before.)

```
: VARS CLASS;
: NEWFNARRAY(LAMBDA X; X::UNDEF; END;) -> CLASS;
: "GERMAN" -> BACK(CLASS("JOHN"));      (CLASS is a function
: "FRENCH" -> BACK(CLASS("BILL"));       array which constructs
: PRNNARRAY(CLASS);                      a new "student" when it
     JOHN   [JOHN . GERMAN]              cannot find one of the
     BILL   [BILL . FRENCH]              right name)
```

¶*Pseudo-records.* The standard POP-2 function RECORDFNS, when applied to suitable arguments, produces, for a particular record class, a constructor function, a destructor function, and a fixed number of component-accessing doublets. Suitable records, which have a fixed size, have to be created using the constructor function, and are accessed using the doublets. There are situations, however, in which it is not known in advance what, or even how many, components the records in a class will have. The records may not all have the same components, for example, all men do not have a wife. In this case the *pseudo-records* described here may be used. There is a single doublet-generating function (NEWCPT) and a single constructor function (NILPSREC). The doublet-generating function, given the name of a component (that is, the subscript), produces a doublet for accessing components of that name from *any* pseudo-record. The constructor function creates a pseudo-record which has, as yet, no components. Components are added by using the doublets described above in the update sense. An attempt to select a component which does not exist produces the failure result UNDEF.

NEWCPT ∊ SUBSCRIPT => (PSEUDO—RECORD => COMPONENT or UNDEF)

NILPSREC ∊ () => PSEUDO-RECORD, with, as yet, no components.

A pseudo-record may be printed using PRPSREC

PRPSREC ∊ PSEUDO-RECORD => ()

Example 5

The use of pseudo-records.

```
: VARS JIM JOHN NAME;
: NILPSREC() -> JIM; NILSPREC() -> JOHN;   (creates two pseudo-
                                             records)
: NEWCPT("NAME") -> NAME;                  (creates a component-
                                             accessing function)
: "JIM" -> NAME(JIM); "JOHN" -> NAME(JOHN);  (adds name
                                             components to JIM
                                             and JOHN)
: JIM -> FATHER(JOHN);
ERROR 38                                   (because the function
CULPRIT [FATHER. UNDEF]                    FATHER has not been
SETPOP:                                    defined)
: NEWCPT("FATHER") -> FATHER;              (definition of FATHER.
: JIM -> FATHER(JOHN);                      Contrast with a
: PRPSREC(JOHN);                            POP-2 record class
     NAME JOHN                              where it would be
                                            impossible to define
                                            a new component.)
     FATHER ASS....                        (ASS... is the record
                                            held in JIM)
```

Pseudo-records can be used to store a variety of component types. It is not always convenient to have UNDEF as the fail result. If the component-accessing doublet is defined using the function NEWCPTF, then a different result for the failure case may be specified by the user.
NEWCPTF ∊ SUBSCRIPT, FAIL RESULT => (PSEUDO-RECORD => COMPONENT or FAIL RESULT)

Example 6
(continuing example 5)

```
: VARS WEIGHT;
: NEWCPTF("WEIGHT", 130)-> WEIGHT;       (creates a component-
: 30 --> WEIGHT(JOHN);                    accessing function
                                          whose fail result is
                                          130)
: WEIGHT(JOHN)+WEIGHT(JIM) =>            (WEIGHT fails to find a
** 160,                                   component in JIM and
: PRPSREC(JOHN);                          so assumes the result
     NAME JOHN                            is 130. If the result
     FATHER ASS...                        had been UNDEF then
     WEIGHT 30                            there would have been
                                          an arithmetic error).
```

¶*Function-components.* Function-components provide for pseudo-records a facility similar to that of function-arrays. They have two parts—a 'look up' doublet and a function which computes the component value for any subscript not occurring in the pseudo-record. The new subscript-component pair so formed is added to the pseudo-record.
NEWCPTFN ∊ SUBSCRIPT, (SUBSCRIPT => COMPONENT)
=> (PSEUDO-RECORD ==> COMPONENT)

Example 7
(continuing examples 5 and 6)

```
: VARS FOOD;
: NEWCPTFN("FOOD", LAMBDA X; NL(1);
:           PR([WHAT IS THE FOOD OF] <> [%NAME(X)%]);
:           ITEMREAD();                    (creates a function
:         END) --> FOOD;                    which asks for
: "PORRIDGE" --> FOOD(JOHN);                information when it
                                            cannot find a suitable
                                            component)
: MAPLIST([%JOHN, JIM%], FOOD) --> BREAKFAST; BREAKFAST =>
[WHAT IS THE FOOD OF JIM]                  (question asked by
                                            system, which waits
                                            for a reply)
: KIPPERS                                  (reply typed by user)
**[PORRIDGE KIPPERS],                      (breakfast)
: PRPSREC(JIM);                            (prints out the pseudo-
    NAME   JIM                              record JIM. Note
    FOOD   KIPPERS                          that the pair (food,
                                            kippers) has been
                                            added).
```

¶*Method used.* The data structures and functions described here are implemented using ASSOCIATION SETS. (See accompanying description of the library program LIB ASSOC.) When LIB NEW STRUCTURES is compiled it automatically brings LIB ASSOC into store. A n-n-array is a closure of the function ASSF and an association list. The function NEWNNARRAY creates a n-n-array which has a 'NULL' association list. Updating of such an array causes subscript-component 'pairs' to be

added to the *end* of the association list. The association list of an n-n-array may be obtained using CONTNNARRAY.

A pseudo-record is an association list. The function NILPSREC is the same function as the operation ASSNIL. A component-accessing doublet is a closure of a function with a subscript. Used in the update sense it adds subscript-component 'pairs' to the end of a pseudo-record.

N-n-arrays and the component-accessing functions of pseudo records are closely related. They are obtained from the same function, ASSF, but in the first case it is partially applied to an association list, and in the second case the arguments are rearranged, and it is partially applied to a subscript.

In this program, association lists are implemented using POP-2 records of three components, data word ASS. An empty association list requires four words of store, and each subscript-component pair requires an additional four words.

¶*Errors.* An attempt to apply a function created by NEWCPT, etc, to an item which is not a pseudo-record, gives ERROR 45, the culprit being the item.

¶*Global variables used.* In general all global variables used in LIB DATA STRUCTURES are prefixed by the letters ASS. The exceptions are: NEWNNARRAY, NEWNNFARRAY, NEWFNARRAY, NEWCPT, NEWCPTF, NEWCPTFN, NILPSREC, CONTNNARRAY, PRNNARRAY, PRPSREC.

¶*Store used.* The program occupies approximately 1 block of store.

R E F E R E N C E S

Library program LIB ASSOC

Library program LIB MEMOFNS

Ambler, A. P. & Burstall, R. M. (1969) Question-answering and syntax analysis. Experimental Programming Reports: No 18, Edinburgh: Department of Machine Intelligence and Perception, University of Edinburgh.

Michie, D. (1967) Memo functions: a language feature with 'rote-learning' properties. *Research Memorandum MIP-R-29.* Edinburgh: Department of Machine Intelligence and Perception.

Popplestone, R. J. (1967) Memo functions and the POP-2 language. *Research Memorandum MIP-R-30.* Edinburgh: Department of Machine Intelligence and Perception.

```
[NEW STRUCTUR]

VARS NEWNNARRAY NEWNNFARRAY NEWFNARRAY NEWCPT NEWCPTF
  NEWCPTFN NILPSREC PRNNARRAY PRPSREC CONTNNARRAY POPLIBCUC;

CUCHAROUT->POPLIBCUC;  ERASE->CUCHAROUT;

COMPILE(LIBRARY([LIB ASSOC]));

POPLIBCUC->CUCHAROUT;


FUNCTION NEWNNFARRAY U; VARS ASS ASSFF;
  ASSNIL->ASS; ASSF(%ASS,U%)->ASSFF; ASSUF(%ASS%)->UPDATER(ASSFF);
  ASSFF;
END;

NEWNNFARRAY(%UNDEF%)->NEWNNARRAY;

FUNCTION NEWCPTF ASSKEY U; VARS ASSFF;
  ASSCF(%ASSKEY, U %)-> ASSFF;
  ASSUCF(%ASSKEY%)->UPDATER(ASSFF); ASSFF;
END;

NEWCPTF(%UNDEF%) -> NEWCPT;

FUNCTION NEWFNARRAY ASSMF;
  VARS ASS; ASSNIL->ASS;
  ASSMEMO(%ASS,ASSMF%)->ASSFF; ASSUF(%ASS%)->UPDATER(ASSFF);
  ASSFF;
END;

FUNCTION NEWCPTFN ASSKEY ASSMF; VARS ASSFF;
  ASSMEMOC(%ASSMF,ASSKEY%)->ASSFF;
  ASSUCF(%ASSKEY%) -> UPDATER(ASSFF); ASSFF;
END;

ASSNILCONS->NILPSREC;

FUNCTION PRNNARRAY NNARRAY; VARS ASS;
  FROZVAL(1,NNARRAY)->ASS;
  ASSPR(ASS);
END;

ASSPR->PRPSREC;

FUNCTION CONTNNARRAY NNARRAY;
  FROZVAL(1,NNARRAY);
END;


'LIB NEW STRUCTURES READY FOR USE'.PR; 2.NL;
```