Program name. LIB RANDOM
Source. J. E. Doran, DMIP; *Date of issue.* December 1968.

¶*Description.* The function RANDOM is a simple but reliable pseudo-random number generator. It produces a real number in the range 0 to 1.

¶*How to use program.* The program should be compiled by typing:
COMPILE(LIBRARY([LIB RANDOM]));
The global variable RANSEED must now be initialized by the user to an integer value in the range 0 to 16383. The user can change the stream of random numbers from run to run by giving this variable different initial values, or can reproduce the same stream by re-initializing it to its starting value.

The function RANDOM takes no arguments and produces as its result the random number.

If numbers in a different range are required, a simple function can be written to do this, for example, to produce random integers in the range 5 to 20, we could write the function:

```
FUNCTION RANDINT;
   INTOF(RANDOM( )*15)+5
END;
```

¶*Method used.* Generation is by a congruential method with a full period of 16384, and approximate correlation between successive numbers of 0.008 ± 0.007.
The function is:

```
FUNCTION RANDOM;
   (125*RANSEED+1)//16384; . ERASE; —> RANSEED;
   RANSEED/16384
END;
```

¶*Global variables.* RANSEED RANDOM.

```
[RANDOM]

VARS RANSEED;FUNCTION RANDOM;
(125*RANSEED+1)//16384; .ERASE; ->RANSEED;
RANSEED/16384;
END;

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

Program name. LIB RANDPACK
Source. R. H. Owen (based on program obtained from Oxford Computing Laboratory program library); *Date of issue.* June 1969

¶*Description.* Procedure to generate random numbers in several standard distributions.

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

COMPILE(LIBRARY([LIB RANDPACK]));
The global variable RANSEED must now be initialized by the user to an integer value in the range 0 to 16383.

The following functions are available to the user.

RANDREAL(w)	gives a random real number in the range (0, w);
RANDINTEGER(n)	gives a random integer in the range (0, n-1);
RANDBOOLEAN(p)	gives a truthvalue which is true with probability p.
RANDNORMAL(m, d)	gives a random deviate with mean m and standard deviation d;
RANDPOISSON(w)	gives a random poisson integer variable with mean w, which should not be large.

¶*Method.* The program uses the same random number generator as LIB RANDOM.
RANDNORMAL uses the fact that
$D\sqrt{(-2.R)}.COS(\pi.R) + M$
is normally distributed with mean m and standard deviation d, provided R is uniformly distributed.
RANDPOISSON uses the fact that the n for which

$$\sum_{i=1}^{i=n} \frac{1}{i} = \frac{R\text{-}\exp(-w)}{w\exp(-w)}$$

is a Poisson Variable with mean w, if R is uniformly distributed.

¶*Globals.* RANSEED. RANDOM, RANDREAL, RANDINTEGER, RANDBOOLEAN, RANDNORMAL, RANDPOISSON

¶*Store used.* The program occupies approximately 0.5 blocks of store.

¶*Example*

```
: COMPILE(LIBRARY([LIB RANDPACK]));
: 'LIB RANDPACK READY FOR USE'
: 123 -> RANSEED:,
: RANDREAL(25) =>
**19.42,
: RANDINTEGER(100) =>
**42,
: RANDBOOLEAN(0.2) =>
**1
: RANDNORMAL(0, 1) =>
**1.03,
: RANDPOISSON(3) =>
**3
```

```
[RANDPACK]

VARS RANDREAL RANDINTEGER RANDBOOLEAN RANDNORMAL
     RANDPOISSON ;


[LIB RANDOM].LIBRARY.COMPILE;

FUNCTION RANDREAL W;RANDOM()*W;END;

FUNCTION RANDINTEGER M;INTOF(RANDOM()*M);END;

FUNCTION RANDBOOLEAN P;RANDOM()<P;END;

FUNCTION RANDNORMAL M D;VARS QA QB;
  SQRT(-2*LOG(RANDOM()))->QA;
  3.14159*RANDOM()->QB;
  QA*COS(QB)*D+M;
END;

FUNCTION RANDPOISSON W;VARS J Z D;
  EXP(-W)->D;RANDOM()-D->Z;0->J;
 L:IF Z<0 THEN J EXIT;
  J+1->J;D*W/J->D;Z-D->Z;
  GOTO L;
END;

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

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

¶*Description.* This package contains a number of useful list processing functions which are given below.

¶*How to use program.* The program should be compiled by typing:
COMPILE(LIBRARY([LIB SETS]));
All the following functions will now be available for use.

(1) MEMBER(X, XLIST, XEQ);
If X is equal to any member of the list XLIST, under the equivalence relation XEQ, then the function produces the result TRUE, otherwise FALSE. For example,
MEMBER(1, [1 2 3], NONOP =) =>
** 1,

(2) UNION(ALIST, BLIST, ABEQ);
This produces a list whose elements are also elements of either the lists ALIST or BLIST, using the function ABEQ to test for equality of elements. For example,
UNION([1 2 3], [2 7 9], NONOP=) =>
** [1 2 3 7 9],

(3) INTERSECTION(ALIST, BLIST, ABEQ);
Produces a list whose elements are members of both the list ALIST and BLIST, again using the function ABEQ to test for equality. For example,
INTERSECTION([1 2 3 4], [2 2 7 1], NONOP=) =>
** [1 2],

(4) SUBTRSET(ALIST, BLIST, ABEQ);
Produces a list of all those elements of ALIST which are not equal, according to the function ABEQ, to any of those in BLIST. For example,
SUBTRSET([A B C], [C D E], NONOP =) =>
** [A B],

(5) SUBSET(SETLL, PLL);
Produces a list of all those elements of the list SETLL which satisfy the predicate PLL. For example,
SUBSET([1 2 9 3 5], LAMBDA X; X<4 END) =>
**[1 2 3],

(6) EXISTS(SETLL, PLL);
This function gives the value TRUE if at least one element of the list SETLL satisfies the predicate PLL, otherwise the result is FALSE. For example,
EXISTS([2 9 3 14], LAMBDA X; BOOLOR(X=2, X=1) END) =>
** 1,

(7) LIT(XS, Y, F);
If XS is a list $[l_1\ l_2\ l_3 \ldots\ l_n]$, Y is an item l_0, and F is a function of two arguments and one result, then in the above, the result is:
$F(\ldots F(F(l_0, l_1), l_2)\ldots, l_n)$
It should be noted that LIT associates to the right. There could be a corresponding LIT function to associate to the left, but this is not implemented in this package. If F is commutative, the two LIT functions produce identical results when F is their common last argument. For example,
LIT([1 2 3 4 5], 0, NONOP+) =>
** 15,

that is, 0+1+2+3+4+5 = 15.

```
LIT(%NIL, UNION(% NONOP= %) %) -> SUMSET;
SUMSET([%[1 2], [4 7], [9 3]%]) =>
** [1 2 4 7 9 3],
LIT([9 3 21 14 7 1 11 2],-100, LAMBDA X Y;
                                  IF X>Y THEN X ELSE Y CLOSE
                               END) =>
** 21,
```

that is, the max of the given list.

(8) SUMSET(LISTLIST) (produced by 'partial application' of LIT as shown above);

This is the function given in the above example. It takes as argument a list of lists, and unions them. For example,

```
SUMSET([[A B] [C D] [E F] [G]]) =>
**[A B C D E F G],
```

¶*Global variables.* SETFN SUMSET LIT MEMBER UNION INTERSECTION SUBTRSET SUBSET EXISTS.

¶*Store used.* The program occupies approximately 1.5 blocks of store.

```
[SETS]


FUNCTION DELETE ITEMLL LISTLL EQUIVLL;
L0: IF LISTLL.NULL THEN NIL EXIT
    IF .EQUIVLL(LISTLL.HD, ITEMLL) THEN LISTLL.TL-> LISTLL  GOTO L0 CLOSE;
    LISTLL.HD::DELETE(ITEMLL,LISTLL.TL, EQUIVLL)
END

FUNCTION SUBSET SETLL PLL;
L0: IF SETLL.NULL THEN NIL EXIT
    IF SETLL.HD.PLL THEN SETLL.HD::SUBSET(SETLL.TL,PLL) EXIT
    SETLL.TL -> SETLL; GOTO L0
END

FUNCTION MEMBER ITEMLL LISTLL EQUIVLL;
L0: IF LISTLL.NULL THEN FALSE EXIT
    IF EQUIVLL(ITEMLL, LISTLL.HD) THEN TRUE EXIT
    LISTLL.TL -> LISTLL; GOTO L0
END

FUNCTION CONSSET ITEMLL SETLL EQUIVLL;
    IF MEMBER(ITEMLL,SETLL,EQUIVLL) THEN SETLL EXIT;
    ITEMLL::SETLL
END

FUNCTION EXISTS SETLL PLL;
L0: IF SETLL.NULL THEN FALSE EXIT;
    IF SETLL.HD.PLL THEN TRUE EXIT;
    SETLL.TL -> SETLL;  GOTO L0
END


FUNCTION LIT XS Y YFXY; VARS X;
LOOP: IF XS.NULL THEN Y
      ELSE XS.DEST->XS->X ;
      YFXY(Y,X)->Y; GOTO LOOP;
   CLOSE;
END;

FUNCTION SETFN XS1 XS2 EQUIVLL T1 T2; REV(XS1)->XS1; VARS X XS3 ;
      IF T1 THEN XS2->XS3; ELSE NIL->XS3;CLOSE;
LOOP: IF XS1.NULL THEN XS3
       ELSE XS1.DEST->XS1->X;
         IF MEMBER(X,XS2,EQUIVLL)=T2 THEN X::XS3->XS3 CLOSE;
            GOTO LOOP
      CLOSE;
END;

VARS SUMSET,UNION,INTERSECTION,SUBTRSET;
 SETFN(% TRUE,FALSE %)->UNION;
 SETFN(% FALSE,TRUE %)->INTERSECTION;
 SETFN(% FALSE,FALSE %)->SUBTRSET;
LIT(% NIL,UNION(%NONOP=%) %)->SUMSET;

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

NOTE ADDED IN PROOF

A LANGUAGE EXTENSION: GENERALIZED JUMPS AND BACKTRACKING

After the main text of this book was completed it became apparent to us, largely through work at Aberdeen (Elcock *et al.* 1971) and at MIT (Hewitt 1969), that a facility for 'backtrack' programming would be very useful. Using some ideas of Leavenworth (1970) we have added to the POP-2 language primitives for state saving to provide backtracking and swapping between processes. These are related to the generalized jumps of ISWIM (Landin 1966), CPL (Strachey 1967), PAL (Evans 1968) and GEDANKEN (Reynolds 1970). We define this extension by adding a section to the Primer and one to the Reference Manual. We wish to thank Bruce Anderson, who used macros to implement and experiment with backtracking in POP-2, and Ray Dunn who implemented the new primitives on the 4130 POP-2 system over a weekend.

A PRIMER OF POP-2 PROGRAMMING, SECTION 25. STATE SAVING AND BACKSCATTERING

There are occasions when one wishes to jump to another point in the program in a different function application. It is possible using *jump-out* to jump to the exit point of a function which has already been entered. But one may wish to jump back into a function application from which one has already exited. On jumping back we may wish to reset some but not all the variables to their previous values. Applications include search algorithms which involve exploring a tree of decision points by 'backtracking', for example, syntax analyzers and a number of artificial intelligence problems, also 'swapping' between apparently simultaneous processes, as in simulation programs, co-routines, and programs servicing more than one user.

Standard functions *appstate* and *reinstate* enable part of the current state of the POP-2 system to be saved as a data structure and later reinstated, that is, the computation returns to the point where the state was saved, resetting some or all of the local variables to their previous values. Global variables and components of data structures are not reset. A function *barrierapply* is used to indicate which local variables are to be saved and reset. It sets up a 'barrier' separating the computation between its entry and exit from that outside. The variables saved are the locals, formals and output locals of those functions which have been entered inside this barrier. *appstate* must be given as argument a function *g*. It constructs a state data structure and applies *g* to it; normally *g* will be used to store the state away for future reinstatement. *appstate* exits normally but on reinstatement control returns to the exit point of *appstate*. States cannot be reinstated outside the barrier in which they were constructed.

Thus we may compare

```
appstate(lambda s; s-->s0 end);
x+1-->x;
...
reinstate(s0);
```

|

```
l: x+1-->x;
   ...
   goto l;
```

The differences are that a state, unlike a label, may be made the value of a variable, and that when *s0* is reinstated the variable *x* will (if suitably declared) be reset to its previous value instead of being incremented.

Here is an example, a program which given y tries to find z such that $z^2=y^3$ by setting z successively to the numbers greater than y.

```
vars a s;0—>a;
function greaterthan x;
    appstate(lambda state;state—>s end);
    comment reinstate(s) jumps back in to here;
    a+1—>a;1.nl;'a =`.pr;a.pr;
    x+a
end;
function g y=>z;
    greaterthan(y)—>z;
    'y =`.pr;y.pr;'z =`.pr;z.pr;
    y*y*y—>y;z*z—>z;
    if z=y then 'solved`.pr else reinstate(s) close
end;
barrierapply(4,g,1);
```

```
a = 1 y = 4 z = 5
a = 2 y = 4 z = 6
a = 3 y = 4 z = 7
a = 4 y = 4 z = 8 solved
```

BACKTRACK PROGRAMMING

To illustrate the use of the state saving mechanism we give function and macro definitions to provide 'backtrack' or 'non-deterministic' programming (Floyd 1967), usable without knowledge of the *appstate* and *reinstate* functions. We define an operation *fail* and macros **either, orelse, orlast,** ndbegin and **ndend** to provide two new statements:

⟨*non-deterministic statement*⟩::= **ndbegin** ⟨*function body*⟩ **ndend**
⟨*either statement*⟩::= **either** ⟨*imperative sequence*⟩ ⟨*orelse clause*?*⟩ ⟨*orlast clause*⟩ **close**
⟨*orelse clause*⟩::= **orelse** ⟨*imperative sequence*⟩
⟨*orlast clause*⟩::= **orlast** ⟨*imperative sequence*⟩

After entering a non-deterministic statement one can evaluate 'either statements' which cause the computation to branch, notionally into parallel branches although in fact the branches are executed in succession from left to right. Each branch consists of executing one of the imperative sequences and then passing on to the next statement. When *fail* is applied the branch terminates. When all branches of an either statement have terminated it acts as a *fail* and control goes back to the previous either statement, giving an error if there is none. After **ndend** control cannot pass back to an either statement evaluated during the evaluation of the non-deterministic statement. A macro **crossvar** declares a 'cross-talk' operation variable which is not reset on failing. The mechanism is that **either** constructs a state, adding it to a 'trail' and *fail* reinstates the last state of the *trail*. More sophisticated search strategies such as 'graph traversing' can be added using extra functions.

```
macro 'crossvar;vars x;.itemread—> x;
macresults([vars operation 1]<>(x::[;
                cont(%consref("])<>(x::["::undef)%)—>nonop])<>(x::nil))
end;
crossvar btrail;vars swbak eitherf operation 2 (dectestsw fail);
function eitherf1 appstat;crossvar esw;0—>esw;
        appstat(lambda s;s::btrail—>btrail end);1+esw—>esw;esw
end;
```

```
function dectestsw; swbak—1—>swbak; swbak= 0 end;
macro either; macresults([. eitherf—>swbak; if dectestsw then])
end;
macro orelse; macresults([elseif dectestsw then])
end;
macro orlast;
    macresults([elseif dectestsw then btrail. tl—>btrail; ])
end;
function fail; btrail. hd. reinstate end;
macro ndbegin;
    macresults([barrierapply(lambda; vars eitherf;
    crossvar btrail; eitherf1 (%.appstate%.)—> eitherf; nil—>btrail; ])
end;
macro ndend; macresults([end. 0); ]) end;
```

For example the following program builds two towers from a set of bricks. The function *twotowers* takes the heights of the towers and a list of the heights of the bricks. Thus *twotowers (3, 5, [1 1 1 2 2 4])* prints *[2 1]* and *[4 1]*, having tried to build the first tower as *[1 1 1]* and backtracked on finding that it then fails in all attempts to build the second tower.

```
function rmember xl; vars x; comment gives some member of xl;
loop: if xl. null then fail close;
    xl. dest—>xl—x;
    either x orlast goto loop close
end;
function delete y xl; vars x; comment copies xl missing out y;
    xl. dest—>xl—>x;
    if x=y then xl else x::delete(y, xl) close
end;
function tower h =>t; vars b; nil—>t;
loop: if h=0 then exit;
    rmember(free)—>b; if b>h then fail close;
    h-b—>h; b::t—>t; delete(b, free)—>free; goto loop
end;
function twotowers h1 h2 bricks;
    ndbegin vars t1 t2 free; bricks—>free;
    either tower(h1)—>t1; tower(h2)—>t2; t1=> t2=>
    orlast "noluck"=> close
    ndend
end;
```

POP-2 REFERENCE MANUAL. SECTION 5.7. SAVED STATES AND REINSTATEMENT

There is a standard function *barrierapply*
barrierapply ∈ *item, . . . , item, function, integer =>item, . . . , item*

Suppose that f is a function of n arguments and m results such that $f(x_1, \ldots, x_n) = y_1, \ldots, y_m$. Then *barrierapply* applies f to its arguments, taking the number of arguments as an additional parameter, thus *barrierapply* $(x_1, \ldots, x_n, f, n) = y_1, \ldots, y_m$

We call an application of *barrierapply* a *Barrier* and other functions, including the parameter function f, which are applied between this entry to *barrierapply* and the corresponding exit are said to be applied *inside* this barrier. Any attempt to take items off the stack in excess of $x_1, \ldots, x_n$ during the application of *barrierapply* causes an error.

There is a standard data structure called a *saved state*. Its dataword is *"savedstate"* and it has no constructor, destructor or select/update functions.

The function *barrierapply* has a standard local variable *appstate* whose value is a function, a different function for each barrier. This function constructs a saved state whenever it is applied and applies a given argument function to it.

appstate ∈ *(saved state=>())=>()*

A saved state constructed by the *appstate* function of a barrier is said to *belong* to that barrier. The application of an *appstate* function is only allowed inside its barrier.

A variable is said to *belong* to a saved state if it is a local, formal or output local of the application of a function, which has been entered but not exited when the state is constructed, and this application is inside the barrier to which the state belongs.

There is a standard function *reinstate*
reinstate ∈ *saved state =>()*

When *reinstate* is applied to a saved state the computation proceeds again from immediately after the exit of the application of the *appstate* function which constructed the state. The value of each variable belonging to the state is reset to the value which it had when the state was constructed, in spite of any assignments which may have intervened. The values of other variables and the components of data structures are not reset. Any functions which were active, that is entered but not exited from, when the state was constructed become active once again. *reinstate* may only be applied to a state inside the barrier to which the state belongs.

BIBLIOGRAPHY

DOCUMENTATION

the language: POP-2 Papers, R. M. Burstall, J. S. Collins & R. J. Popplestone. Edinburgh: Edinburgh University Press 1968 (made obsolete by this book).

the compilers: 'POP-2 in POP-2', J. G. P. Barnes and R. J. Popplestone. Property of the University of Edinburgh Round Table, Department of Machine Intelligence and Perception, 1968.

the POP-2/4100 implementation: 'POP-2/4100 Users' Manual', R. D. Dunn. Edinburgh: Department of Machine Intelligence and Perception, 1970.

'A plain man's guide to Multi-POP implementation', D. J. S. Pullin. *Mini-MAC Report No. 2.* Edinburgh: Department of Machine Intelligence and Perception, 1967.

the Uni-POP/1900 implementation: 'A Supplementary Manual for the Lancaster POP-2 System', John Scott. University of Lancaster, 1968.

the Uni-POP/System 4 implementation: 'System 4 POP-2 compiler-internal conventions', J. G. P. Barnes. *Research memorandum MIP-R-41.* Edinburgh: Department of Machine Intelligence and Perception, 1968.

the Basic POP/903 implementation: 'The Basic POP/903', Horace Townsend. Regional Clinical Neurophysiology Service, Western General Hospital, Edinburgh 1968.

the Uni-POP/System 4 implementation: 'System 4 POP-2 Users Guide' and 'System 4 Regime J Batch Environment', J. G. P. Barnes. Edinburgh: Department of Machine Intelligence & Perception 1968.

REFERENCES

Barron, D. W., Buxton, J. N., Hartley, D. F., Nixon, F., & Strachey, C. S., (1963) The main features of CPL. *Comput. J.*, **6**, 134-43.

Burstall, R. M. & Popplestone, R. J. (1968) POP-2 reference manual. *Machine Intelligence 2*, pp. 207-46 (eds. E. Dale & D. Michie). Edinburgh: Edinburgh University Press.

Elcock, E. W., Foster, J. M., Gray, P. M. D., McGregor, J. J., & Murray, A. M. (1971) ABSET. A programming language based on sets; motivation and examples. *Machine Intelligence 6*, pp. 467-90 (eds. B. Meltzer & D. Michie). Edinburgh: Edinburgh University Press.

Evans, A. (1968) PAL—A language designed for teaching programming linguistics. *Proc. Ass. comput. Mach. 23rd Nat. Conf. 1968*, pp. 395-403. Brandon Systems Press, Princeton, N.J.

Floyd, R. W. (1967) Non-deterministic algorithms. *J. Ass. comput. Mach.*, **14**, 636-44.

Hewitt, C. (1969) PLANNER: a language for proving theorems in robots. *Proc. Joint Int. Conf. on Artificial Intelligence*, Washington, D.C. pp. 295-301.

Landin, P. J. (1964) The mechanical evaluation of expressions. *Comput. J.*, **6**, 308-20.

Landin, P. J. (1965) The correspondence between ALGOL 60 and Church's lambda notation. *Comm. Ass. comput. Mach.*, **8**, 89-101, 158-165.

Landin, P. J. (1966) The next 700 programming languages. *Comm. Ass. comput. Mach.*, **9**, 157-66.

Leavenworth, B. M. (1970) Definition of quasi-parallel control processes in a high level language. *Proc. Int. Computing Symposium, Bonn, Assoc. Comput. Mach.*, pp. 442-71.

McCarthy, J. (1960) Recursive functions of symbolic expressions. *Comm. Ass. comput. Mach.*, **3**, 184-95.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., & Levin, M. I. (1962) *LISP 1.5 Programming Manual*. Cambridge, Mass.: MIT Press.

Marks, S. L. (1967) The JOSS User's Reference Manual. *Memorandum RM-5219-PR*. Rand Corporation, Santa Monica.

Mooers, C. M. (1966) TRAC, a procedure describing language for the reactive typewriter. *Comm. Ass. comput. Mach.*, **9**, 215-24.

Popplestone, R. J. (1968) POP-1: an on-line language. *Machine Intelligence 2*, pp. 185-94 (eds. E. Dale & D. Michie). Edinburgh: Edinburgh University Press.

Popplestone, R. J. (1968a) The design philosophy of POP-2. *Machine Intelligence 3*, pp. 393-402 (ed. D. Michie). Edinburgh: Edinburgh University Press.

Reynolds, J. C. (1970) GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept. *Comm. Ass. comput. Mach.*, **13**, 308-19.

Shaw, J. C. (1964) Joss: a designer's view of an experimental on-line computing system. *AFIPS Conference Proceedings* (FJCC 1964), **26**, 455-64.

Strachey, C. (1967) Ed. CPL reference manual. Privately circulated. Programming Research Unit, Oxford University.

Sussman, G. J. & Winograd, T. (1970) Micro-planner reference manual. *A. I. Memo, No. 203, Project MAC*, MIT, Cambridge, Mass.

Wirth, N. & Hoare, C.A.R. (1966) A contribution to the development of Algol. *Comm. Ass. comput. Mach.*, **9**, 413-32.

INDEX TO THE PRIMER

INDEXES to the reference manual

TECHNICAL TERMS

INDEXES to the reference manual

TECHNICAL TERMS

SYNTAX DEFINITIONS

STANDARD FUNCTIONS AND VARIABLES

OPTIONAL FUNCTIONS

These are defined in Appendix 2.

SYNTAX WORDS