ISSN 0084-6198

Algol Bulletin no. 51

DECEMBER 1984

CONTENTS

The ALGOL BULLETIN is produced under the auspices of the Working Group on ALGOL of the International Federation for Information Processing (IFIP WG2.1, Chairman Peter R. King, University of Manitoba).

The following statement appears here at the request of the Council of IFIP:

> "The opinions and statements expressed by the contributors to this Bulletin do not necessarily reflect those of IFIP and IFIP undertakes no responsibility for any action that might arise from such statements. Except in the case of IFIP documents, which are clearly so designated, IFIP does not retain copyright authority on material published here. Permission to reproduce any contribution should be sought directly from the authors concerned. No reproduction may be made in part or in full of documents or working papers of the Working Group itself without permission in writing from IFIP."

Facilities for the reproduction of the Bulletin have been provided by courtesy of the John Rylands Library, University of Manchester. Word-processing facilities have been provided by the Barclay's Microprocessor Unit, University of Manchester, using their Vuwriter system.

The ALGOL BULLETIN is published at irregular intervals, at a subscription of $11 (or £6) per three issues, payable in advance. Orders and remittances (made payable to IFIP) should be sent to the Editor. Payment may be made in any currency (a list of acceptable approximations in the major currencies will be sent on request), but it is the responsibility of each sender to ensure that his payment is made in accordance with the currency requirements of his own country. Subscribers in countries from which the export of currency is absolutely forbidden are asked to contact the Editor, since it is not the policy of IFIP that anyone should be debarred from receiving the ALGOL BULLETIN for such a reason.

The Editor of the ALGOL BULLETIN is:
Dr. C. H. Lindsey,
Department of Computer Science,
University of Manchester,
Manchester, M13 9PL,
United Kingdom.

Back numbers, when available, will be sent at $4.50 (or £2.40) each. However, it is regretted that only AB32, AB34, AB35, AB36, AB38-43 and AB45 onwards are currently available. The Editor would be willing to arrange for a Xerox copy of any individual paper to be made for anyone who undertook to pay for the cost of Xeroxing.

AB51.0 EDITOR'S NOTES.

ALGOL 68 Standardization

I mentioned in the last issue that the proposals to produce an International Standard for ALGOL 68 had been to a letter ballot within ISO. This had produced an overall majority in favour, and even five coutries willing to participate in preparing the Standard (W. Germany, Belgium, Netherlands, U.S.S.R and Czechoslovakia), but unfortunately two of the would-be participators (U.S.S.R. and Czechoslovakia) were not the right kinds of member of the right ISO committees, and so the project did not get through. There now seems a possibility that a further two countries with the right status could be persuaded to participate, and it is therefore likely that we shall shortly be pressing for a second ballot to be held within ISO.

ALGOL 60 Standardization

ISO Standard 1538, "Programming languages - ALGOL 60", was published on October 9th 1984. Copies can be obtained through your own national standards organisation. The new International Standard essentially consists of the Modified ALGOL 60 Report (as previously printed in the Computer Journal), preceded by an introductory section setting out standards for conformance, etc.

Survey of viable implementations

This issue contains an updated version of the survey of viable ALGOL 68 implementations, originally published in AB47. It contains information about 5 implementations not included last time, including the first one available on a micro. I would be pleased to hear of any other implementations that may appear, with a view to publishing a further version sometime.

ALGOL Bulletin Policy

As you will see, this is again a thin issue, and it has been an awful long time since the last one. I am quite willing to publish thicker issues more often, but for this I first need the material. I welcome, indeed I solicit (and even beg) contributions connected not only with ALGOL 60 and ALGOL 68, but indeed with any topic concerned with the principles of programming languages. This is supposed to be an informal, unreferreed journal where developing ideas can be aired, but if you, the readers, have no ideas to air, then there will be little point in continuing publication.

In the meantime, because issues have been so thin and therefore have cost little to print, we have been accumulating quite significant sums of money. I have therefore decided to declare a free issue. I.e. everyone whose subscription would have expired with this issue, or later, will have his subscription extended by one issue (so, if you do get a subscription reminder this time, it will be because your subscription actually expired at AB50, and it will be one of those red "final" ones).

AB51.1 Announcements.

AB51.1.1 Book Review : Programming Languages and their Definition

By H. Bekic.
Lecture Notes in Computer Science 177, Springer Verlag 1984
ISBN 3-540-13378-X or 0-387-13378-X

Hans Bekic, a leading member of the IBM Laboratory Vienna, and for many years a member of IFIP Working Groups 2.1 and 2.2, died in 1982 (see AB49.1.1). This posthumous collection of papers has been put together by his former colleague C.B. Jones. Although he published little in the recognised computing journals, he was a recognised authority on denotational semantics and many of his technical reports written for IBM Laboratory Vienna were extensively referred to by other workers in the field. Many of these reports are here published in accessible form for the first time.

The paper "Formalization of Storage Properties" arose from a dissatisfaction with the underlying model of the original ALGOL 68 Report. He proposed that locations to hold stowed values should be regarded as having sub-locations for the components, that *flex* should be a property of names rather than of values, that the concept "instance of a value" was unnecessary, that the linearization of multi-dimensional arrays was superfluous, and much else besides. The amazing thing is that this paper was quite unknown to the editors of the Revised ALGOL 68 Report (certainly to this editor, at least), and yet all of these changes were incorporated in the Revision in almost the same form as he had proposed them. (It should be noted in general that, where this book refers to ALGOL 68, it is always the original ALGOL 68 Report that is implied.)

Several papers refer to shortcomings in PL/1, and give a good description of the techniques used in the original "VDL" formal definition of that language. This definition contained an operational semantics based on an extremely baroque model of the machine state (only to be expected, I suppose, for such a baroque language). This gave him a healthy distaste for operational semantics, and most of the rest of the book is therefore concerned with denotational semantics. This includes a definition of a substantial subset of PL/1 in the denotational style (Cliff Jones has edited out the bulk of the details, but enough is left to show how the method worked). However, his main interest was in finding a sound denotational model for nondeterminacy and parallelism.

This proceeded in two phases. The 1971 paper "Towards a Mathematical Theory of Processes" tried to regard the parallel composition of the denotations of two actions as a set of functions. This paper has been very widely referenced but, unfortunately, it contains a serious bug, and several subsequent items in the book discuss its possible fixes. A later group of papers, dating from 1981, examines an alternative formulation of parallelism in which the composition results in a function returning a set of possible outcomes, each carefully indexed by a value from an indexing set recording a possible merging of the component primitive actions. The treatment here is very sketchy, since Hans was still developing these ideas at the time of his death.

C.H. Lindsey

AB51.1.2 Books Received : An Analysis of Sparse Matrix Storage Schemes

by M. Veldhorst
Mathematical Centre Tracts 150, Mathematisch Centrum, Amsterdam 1982.
ISBN 90 6196 242 0

This is a reprint of a thesis from the University of Utrecht. It deals with an extension, known as TORRIX-SPARSE, of the TORRIX matrix handling package, as developed by the author and S.G. van der Meulen - see Mathematical Centre Tracts 86). Two storage methods for sparse matrices are considered, one in which rows are stored with leading and trailing zeroes omitted, and one in which the matrix is partitioned into blocks and subblocks, with the possibility that subblocks filled entirely with zeroes need not be stored. Methods of organising matrices to fit into one or other of these strategies are investigated, and examples of the use of the techniques are given.

AB51.1.3 Books Received : Abstraction, Specification and Implementation Techniques

by H.B.M. Jonkers
Mathematical Centre Tracts 166, Mathematisch Centrum, Amsterdam 1983.
ISBN 90 6196 263 3

This is a reprint of a thesis based on research conducted at the Mathematical Centre. It describes, by way of a case study, the design and implementation of a garbage collector for a machine-independent ALGOL 68 implementation. However, the prime purpose of the work is to explore the systematic way in which the garbage collector was arrived at, by way of a specification language for algorithms and data structures, and correctness-preserving transformations to an efficient implementation.

AB51.1.4 New Journal : Parallel Computing

North-Holland has recently published volume 1, issue 1 of PARALLEL COMPUTING, a new international journal presenting the theory and use of parallel computer systems, including vector, pipeline, array and fifth generation computers. Within this context, the journal covers all aspects of high-speed computing. Its publication style is intended to be generally oriented to the international and interdisciplinary character of the parallel computing community.

PARALLEL COMPUTING features original research work, tutorial and review articles, as well as accounts on practical experience with, and techniques for, the use of parallel computers.

A free copy of PARALLEL COMPUTING can be obtained by writing to:
North-Holland, a Division of Elsevier Science Publishers, attn:
Karin van Schouten, P.O. Box 1991, 1000 BZ Amsterdam, The Netherlands.

AB 51.3.1 Survey of Viable ALGOL 68 Implementations

This survey, which we hope to republish from time to time as further information becomes available, has been restricted to implementations which you can actually obtain and use. Each of them has an identifiable person or organisation responsible for its maintenance, and has been used on at least one site other than that where it was developed.

Name of System	Hardware	Operating System	Principal Sublanguage features	Principal Superlanguage features
FLACC	IBM 370 Amdahl Siemens	OS/VS/MVS /MFT/MVT CP/CMS MTS		exception handling Fortran interface
ALGOL 68C Release 1	IBM 360 IBM 370	OS/MVT OS/VS2 OS/MVS OS/MFT OS/VS1	no *sema* no *flex* no *format* restricted transput	automatic *op:=* for any *op* *upto*, *downto* and *until* in loop-clauses displacement operator (*:=:=*) *andf*, *orf* and *thef* separate compilation scopes not checked
		VM/CMS		
	DEC-10 DEC-20	TOPS-10 TOPS-20		
	DEC VAX	Berkeley UNIX 4.2		
	Prime			
	Tele- funken TR440 TR445	BS3	improved transput available	
	CYBER 205	VSOS 2.1		

Most of the column headings are self-explanatory. "Deviation" means that it is possible to write some program, valid and with defined meaning both in the given implementation and according to the Revised Report, which will provide results different from those defined by the Revised Report. Under "Money", "nominal" usually means under $200, "yes" means a realistic commercial rate. "MC Test" means that it has been tested using the MC Test Set (see AB 44.1.2) and that the implementor claims it ran correctly. In all cases, the people listed in the last column should be able to provide further information.

Deviations?	Money?	MC Test?	Other features	Where to obtain it
No	Yes	Yes	load and go version available very complete checking (hence not-so-fast running)	Chion Corporation Box 4942, EDMONTON Alberta Canada T6E 5G8
Yes	Nominal to Univer- sities	No	fast running no garbage collector	ALGOL 68C Distribution Service Computer Laboratory Corn Exchange Street CAMBRIDGE CB2 3QC United Kingdom
				Robert Hill Computing Service University of Leeds LEEDS LS2 9JT United Kingdom
				Dr R. G. Blake Computing Service University of Essex Wivenhoe Park COLCHESTER CO4 3SQ United Kingdom
				ALGOL 68C Distribution Service (see above)
				Dr E. F. Elsworth Dept. of Computer Science University of Aston in Birmingham BIRMINGHAM United Kingdom
				Hanno Wupper Rechenzentrum der Ruhr-Universitaet Postfach 102148 D-4630 BOCHUM Federal German Republic
				Hanno Wupper (see above)

Name of System	Hardware	Operating System	Principal Sublanguage features	Principal Superlanguage features	Deviations?	Money?	MC Test?	Other features	Where to obtain it
	TESLA 200 (similar to IBM 360)		no *flex* (except *string*) no *union* no *sema* no *heap*	bounds in formal-declarers	No	No	No	TRACE facility independent compilation of routines fast running	J. Nadrchal Institute of Physics Czechoslovak Academy of Sciences 180 40 PRAHA 8 Na Slovance 2 Czechoslovakia
CONTROL DATA ALGOL 68	CDC 6000 -7000 170 series	NOS 2 NOS/BE SCOPE 2	one *long* flexibility is an attribute of a multiple value	no transient name restriction ICF macros allow definition of operators in machine instructions	No	Yes	Yes	separate compilation	Control Data Services P.B. 111 RIJSWIJK (24) The Netherlands
A68S	CDC Cyber PERQ	NOS 2 NOS/BE SCOPE 2.1 PNX 2	official sublanguage (SIGPLAN Notices 12 5 May 1977 or Informal Introduction Appendix 4) but *heap* is allowed		No	Nominal	No	very complete checking fast compilation slow running	Dr C. H. Lindsey Dept. of Computer Science University of Manchester MANCHESTER M13 9PL United Kingdom
A68RS	ICL 2900 Honeywell Series 60 Level 68 DEC VAX	VME/B MULTICS VMS	indicators to be declared before use no *sema* scopes not checked	*mode vector* indexable structures *forall* elements of array no transient name restriction	Yes Yes Yes	Yes Nominal to Universities Yes	Yes Yes Yes	modular compilation re-entrant compiler and object code source-level symbolic debugger	ICL local sales office Richard Wendland Praxis Systems Limited 6/7 Trim Street BATH BA1 1HB United Kingdom Products Group SPL International Research Centre The Charter ABINGDON OX14 3LZ United Kingdom
	UNIVAC 1100 series	EXEC VIII	no garbage collector scopes not checked	*bin* of any primitive mode complex mathematical functions *min* and *max* matrix and vector operators exception handling	Yes	No	Yes	French representations (inhibitable by pragmat) independent compilation of routines	Daniel Taupin Laboratoire de Physique des Solides Universite de Paris XI 91405 ORSAY France
	Victor Sirius Apricot	CPM MSDOS	no *par*, *format*, *op*, *format*, *union*, *goto*, *bytes*, *long short*, *heap*, *flex* no anonymous routine texts temporary transput restricted scope of arrays, restricted balancing, restricted *[][]* modes	*mode address* for access to memory-mapped addresses access to CPM primitives, machine coded subroutines and 8087 chip features uses interpreted intermediate language	No	Yes	No	incremental compilation with immediate execution many of the missing features will appear in later releases	Algol Applications Ltd 369 Ipswich Road COLCHESTER CO4 4HL United Kingdom

AB51.4.1 Implementing *random* in ALGOL 68

by C. H. Lindsey and I. Domville
(University of Manchester)

1. The Problem.

The ALGOL 68 Revised Report [1] specifies a pseudo-random number generator that is to be provided in the standard-prelude. The precise form of this poses some problems not usually encountered by implementors of such generators. Here, to refresh your memory, is the specification taken from R10.2.3.12.k and R10.5.1.b.

```
proc L next random = (ref L int a) L real:
     (a := c the next pseudo-random L integral value after 'a' from a
           uniformly distributed sequence on the interval [L 0, L max int] c
      ;
      c the real value corresponding to 'a' according to some mapping of
           integral values [L 0, L max int] into real values [L 0, L 1) (i.e.,
           such that 0 ⩽ x < 1} such that the sequence of real values so
           produced preserves the properties of pseudo-randomness and uniform
           distribution of the sequence of integral values c) ;

L int L last random := round (L max int / L 2) ;

proc L random = L real: L next random (L last random) ;
```

Remember that *L* (L) is to be replaced, consistently, by a suitable number of *longs* (**longs**) or *shorts* (**shorts**).

Now for the problems:

a) Unlike most random number generators, we are asked to produce a sequence of ints ($0 \leqslant r \leqslant$ *maxint*) rather than of reals. Assuming *maxint* is of the form 2^n-*1*, this means we are in the business of producing *n*-bit words composed of random bits.

b) A random number generator depends on a "seed" which is initialized to some value (such as round(*maxint*/*2*)) and updated after each call of the generator. The random number produced is some function of the last seed. In the case of the generator *next random* specified by the RR, the random int produced <u>is</u> the new seed, and therefore the seed itself is of mode int. This rules out generators of the type proposed by Wichmann and Hill [2,3] in which the seed consisted of three separate numbers, each producing its own pseudo-random sequence, these being combined to produce a final sequence with much better randomness and with a much longer cycle (although not the product of the three individual cycle lengths as Wichmann and Hill had originally supposed [3]).

c) Now that 16-bit micros are in common use, one would expect to implement ALGOL 68 on them. Since the seed is required to be a positive int, this leaves us just 15 usable bits and a maximum cycle length of 32768, which is distinctly on the short side. One cannot afford an actual cycle length significantly shorter than this. Moreover, it is unlikely that a generator with such a short cycle will give as good randomness as longer ones.

d) We would like to have an algorithm that is portable to all implementations. Therefore, it must be written in a high-level language and cannot take advantage of any special facilities (such as multi-length arithmetic) provided in the order code of the particular hardware. On the other hand, all such implementations (with a given word length) will then give the same sequence of pseudo-random integers.

e) The sequence of real numbers returned by *next random* (and hence by *random*) is to be derived from the sequence of ints. It is not allowed to use a long int seed to produce the real sequence (a long int seed can only be used to produce a long real sequence).

2. Implementation.

2.1. Types of generator.

The simplest random number generator is the "multiplicative congruential generator". This takes the following form:

$$X_{n+1} = (X_n \times P) \bmod Q$$

where *X* and *P* are positive integers and Q is (usually) a prime number. Any value of *P* which gives the maximal cycle length Q-*1* is known as a "primitive root" of Q. To decide if a number is a primitive root, the following theorem is used.

<u>Theorem.</u> The number A is a primitive root of a prime number *P* iff
$A \not\equiv 0$ *(modulo P)* and $A^{(P-1)/Q} \not\equiv 1$ *(modulo P)* for any prime divisor Q of *P-1*.
where $X \not\equiv Y$ *(modulo Z)* means $X \bmod Z \neq Y \bmod Z$.

This is taken from a more general case given by Knuth [4], to which the interested reader should refer for a more detailed discussion.

Two types of generator were considered:

<u>SMC(*n*/*P*)</u> Q is a prime number as close as possible to 2^n. *P* is a primitive root of Q, chosen to be close to $\sqrt{Q}$. This gives a cycle of length Q-*1*.

<u>OP5(*n*/*P*)</u> Q is 2^n and *P* is an odd power of *5*. Moreover, $X_n = 4 \times seed+1$, and the next seed is $(X_{n+1}-1)+4$ (the bottom two bits of X_n abd X_{n+1} are always *01*). The advantage of this method is that the cycle length is the full 2^n, and that the *mod* operation is trivial to implement. We needed to implement generators with *n* equal to *15 and 31* (for 16- and 32-bit machines) and *48* (for CDC Cyber machines).

The following table shows the values of Q and *P* that were tried.

	OP5 generators		SMC generators	
n	Q	*P*	Q	*P*
15	2^{15} =*32768*	*125, 3125*	2^{15}-*19* =*32749*	*171, 172, 175, 176, 182, 189*
31	2^{31}	*125, 3125*	2^{31}-*1*	*46339, 46340, 46341, 46342*
48	2^{48}	*125, 3125*	2^{48}-*59*	*16777213, 16777214, 16777215*

2.2. Stretching.

Whereas the Revised Report requires the generator to produce ints uniformly in the range *0..maxint*, the SMC generators produce them in the range *1..Q-1* so that, for example, in the SMC(15) generators the numbers *32749..32767* are never

generated at all. To ensure that the mean value of the numbers generated is *maxint/2*, we need to "stretch" the output of the generator so that the missing numbers are distributed uniformly throughout the range, and to "shrink" it again before the generator is called the next time. The shrinking formula is

$$X_n := seed + (1 - entier(seed/shrinker))$$

where $shrinker = (maxint+1)/((maxint+1) - (Q-1) + 1)$.

The stretching formula is

$$newseed := X_{n+1} + entier((X_{n+1} - 1)/stretcher) - 1$$

where $stretcher = (Q-2)/((maxint+1) - (Q-1) + 1)$.

2.3. Multi-length Arithmetic.

Obviously, if we multiply an n-bit seed by P, we shall cause an overflow. We must therefore resort to multi-length arithmetic which, for portability, must be coded entirely using int arithmetic.

We split the n-bit seed into *MIDBITS* (the top $m = (n+1)\div 2$ bits) and *LOBITS* (the bottom $l = n\div 2$ bits). We also provide *HIBITS* to hold the overflowed part of the product. This gives, for the 16-bit machine,

$$\underbrace{\lceil HIBITS \rceil}_{-} \quad \underbrace{\lceil MIDBITS \rceil}_{m=8} \quad \underbrace{\lceil LOBITS \rceil}_{l=7}$$

Each part is multiplied by P, and any overflows from *LOBITS* and *MIDBITS* are carried forward to *MIDBITS* and *HIBITS* respectively. In all the generators to be considered, $2^l < P < 2^m$. Thus $LOBITS*P < 2^n$, but $MIDBITS*P$ might exceed *maxint*. *MIDBITS* is therefore decreased by 2^{m-1} beforehand, thus effectively using the sign bit of an int to allow products up to $2*maxint$. Finally, we recombine *MIDBITS* and *LOBITS* into one n-bit value in *LOBITS*.

The next calculation, to take the result S (now residing in *HIBITS* and *LOBITS*) modulo Q, is not quite so simple. We make use of the fact that Q is only slightly less than 2^n, and thus $S \bmod 2^n$ (which is already held in *LOBITS*) is a good approximation to $S \bmod Q$.

Let $q = 2^n - Q$
Observe that $S < 2^n * P$ and hence $S/2^n < P$

$$\begin{aligned}
\text{Then } S+Q &= S+(2^n-q) \\
&\leqslant S/(2^n-q) \\
&< (S/2^n)*(1+q/2^n) \\
&< S/2^n + P*q/2^n \qquad (\text{since } S/2^n < P) \\
&< S+2^n+1 + P*q/2^n \\
&< S+2^n+2 \quad \text{provided } P*q < 2^n \\
&\qquad \text{which is true for all the } qs \text{ considered.}
\end{aligned}$$

Thus, either $S+Q = S+2^n$ or $S+Q = S+2^n+1$.

$$\begin{aligned}
&\text{If } S+Q = S+2^n \\
&\text{Then } S \bmod Q = S - S+Q*Q \\
&\qquad = S - S+2^n*(2^n-q) \\
&\qquad = S \bmod 2^n + S+2^n*q \\
&\qquad = LOBITS + HIBITS*q \\
&\text{Else } S+Q = S+2^n+1 \\
&\text{And } \ S \bmod Q = S - S+Q*Q
\end{aligned}$$

$$\begin{aligned}
&= S - (S+2^n+1)*(2^n-q) \\
&= S \bmod 2^n + S+2^n*q - (2^n-q) \\
&= LOBITS + HIBITS*q - Q
\end{aligned}$$

The full algoritm is given is section 5 below.

3. Testing.

Since *maxint* is of the form 2^n-1 on all the machines considered, we are essentially trying to generate random sequences of bits. It was therefore decided to test the generators by regarding the ints produced as being made up of octal digits, and seeing whether these occurred randomly. In most of the tests, we considered the 1st, 2nd, 3rd, etc. digits of each number, and compared them with the corresponding digits of the succeeding numbers. The following tests, based on those described by Wichmann and Hill [2], were tried.

a) Poker test. This test begins by taking octal digits from five consecutive numbers to form poker hands of five "cards". The type of "hand" (all different, 1 pair, 2 pairs, 3 of a kind, full house, 4 of a kind and 5 of a kind) is recorded. After 4800 hands, the number of occurrences of each type is compared with its theoretical expectation, and a χ^2 value is computed.

b) Coupon collector's test. This test gets its name because we have a "coupon" with all the digits 0 to 9 on it. For each number r, $entier(r/2^n*10)$ (effectively, the first decimal digit of r) is computed and the corresponding digit on the coupon is ticked. When the coupon is completed, the length of the sequence of digits is noted. After 10000 numbers, the distribution of sequence lengths is compared with its expectation (we considered all lengths from 11 to 75, plus a class for sequences outside this range). A χ^2 value is computed.

c) Runs up and down. This test studies the length of monotonic runs up and down, using Grafton's algorithm [5], and was thought to be one of the most powerful tests. The first 10000 numbers were tested, producing separate χ^2 values for runs up and for runs down. The test was repeated using, in place of each r, $r - entier(r/2^n*10)*2^n+10$ (effectively, after removing successive leading decimal digits).

d) Serial test. From a sequence of octal digits, the triplet commencing at each position was considered, and the dependency of the third digit of each triplet on the 64 combinations of the other two was examined. We obtained 64 distributions of the third digit, and 64 χ^2 values showing how they deviated from the uniform distribution expected. These χ^2 values were divided into 14 groups, each containing all the χ^2 values in a given range, and an overall χ^2 value was computed.

3.1. Results of tests.

a) Poker test.

Values of χ^2 with 5 degrees of freedom
(5% upper limit is 11.070)

	SMC(15/182)	OP5(15/125)
1st digit	6.136	6.720
2nd digit	0.754	6.932
3rd digit	3.937	107.564
4th digit	6.996	1849.905
5th digit	6.858	18605.714

The figures for SMC are entirely reasonable, as are the first 2 digits of OP5. The disaster in the later octal digits of OP5 reflects the fact that, on careful examination of the algorithm, one sees that the bottom few bits are bound to follow a rather simple cycle. Thus OP5 is not an acceptable generator, although it was not until we began to examine octal digits as opposed to decimal ones that this became apparent.

b) Coupon collector's test.

Values of χ^2 with 64 degrees of freedom
(5% upper limit is 83.672)

SMC(15/182)	SMC(15/176)	OP5(15/125)	SMC(31/46340)	OP5(31/125)
958.8	956.8	1369	72.64	59.54

The high figures for the 15-bit generators reflect the difficulty of making a satisfactory generator for a 16-bit machine. The 31-bit generator is entirely reasonable.

c) Runs up and down.

Values of χ^2 with 6 degrees of freedom
(5% upper limit is 12.590)

digits	SMC(15/182)		SMC(15/176)		OP5(15/125)		SMC(31/46340)		OP5(31/125)	
	up	down	up	down	up	down	up	down	up	down
1	7.89	1.94	1.54	1.17	3.07	2.30	3.21	2.89	19.22	10.48
2	4.18	1.93	1.92	5.44	0.80	4.77	1.22	4.17	18.55	2.63
3	4.52	6.41	1.51	1.31	2.23	1.80	1.65	5.26	12.50	8.19
4	3.46	12.59	3.54	1.82	9.12	4.84	6.33	6.49	4.18	5.05
5	2.69	8.40	1.83	1.59	2.74	4.98	3.62	6.36	6.25	10.55

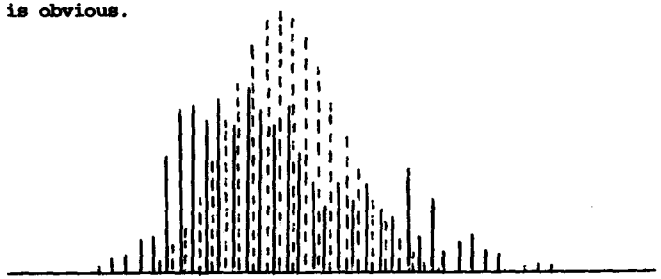
These figures are all reasonable which only goes to show, in view of the results of the next test, that runs up and down is not a very good test.

d) Serial test.

Up to this point, SMC(15/182) had seemed the best candidate, but on this test it proved to be a disaster. Consider triplets starting with a particular pair of digits, e.g. of the form *(3,7,-)*. Suppose the observed frequency with which the third digit takes the values *0,1,2,..,7* in 80 trials is

6 17 10 9 11 8 9 10

Does the digit *1* occur so many more times than expected because of a reasonable sampling error, or does the pair *3,7* really prefer to be followed by a *1*? If, after examining triplets starting with all 64 combinations of two digits, we find that unusually high frequencies like 17 turn up unexpectedly often, then something is clearly wrong with the generator. The following histogram shows how often the various frequencies turned up in the SMC(15/182) generator. It also shows the binomial distribution which these frequencies should have followed. The discrepancy is obvious.



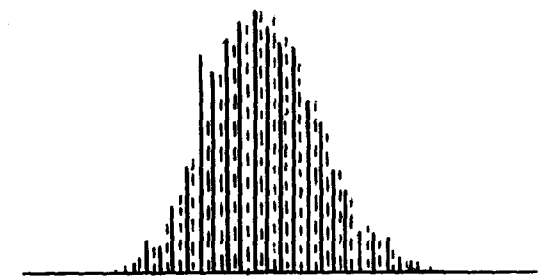
The χ^2 values yielded by this test, and given in the following table, show the extent to which the observed distribution of the frequencies differred from the binomial one. After the failure of SMC(15/182), various other primitive roots of *32749* were tried, as the table shows.

Values of χ^2 with 13 degrees of freedom
(5% upper limit is 22.362)

	SMC(15/171)	/172	/175	/176	/182	/189	OP5(15/125)
1st digit	55.44	27.78	34.42	33.87	959.42	50.52	48.56
2nd digit	79.03	43.56	55.28	40.59	781.06	64.89	482.47
3rd digit	41.14	49.89	31.84	14.27	136.53	43.02	2432.00
4th digit	31.92	16.06	14.58	11.45	27.62	36.77	388.31

	SMC(31/46339)	/46340	/46341	/46342	SMC(48/16777213)	/16777214	/16777215
1st digit	10.44	9.66	995.20	10.52	12.31	22.00	12.23
2nd digit	12.23	8.25	1086.84	13.17	5.52	7.86	2.94
3rd digit	13.41	8.80	971.53	8.17	7.00	12.16	10.12
4th digit	11.30	21.06	1011.84	7.86			
5th digit	8.33	8.87	1058.87	10.67			

The table shows how every prime number seems to have its maverick primitive root (and also the fact that the OP5 generator degenerates into a cycle in its later digits shows up clearly). The generators finally chosen were SMC(15/176), SMC(31/46340) and SMC(48/16777215). The histogram below is for SMC(15/176) and may be compared to that shown for SMC(15/182) above.



4. real random numbers.

next random is required to return a real number *r*, $0.0 \leq r < 1.0$. We could use

r = *newseed*/Q

but the number of significant bits in a real is likely to be more than the number of bits in an int, and so the less significant bits of *r* will not appear very random. We therefore used

r = *newseed*/Q + *oldseed*/Q^2

thus constructing *r* out of two random integers.

5. The algorithm expressed in ALGOL 68

```
proc nextrandom = (ref int seed)real:
     co version for 16-bit machine.
        differences for 32- and 48-bit machines given between #...#.
     co
     begin
     co n=15, l=7, m=8 co # n=31, l=15, m=16; n=48, l=24, m=24 #
     int  twol = 128, # 32768; 16777216 #
          twom = 256, # 65536; 16777216 #
          P    = 176, # 46340; 16777215 #
          Q    = 32749, # 2147483647; 281474976710597 #
          q    =  19, # 1; 59 #
     real shrinker  = 1560.381, # 715827882.334; 4614343880501.61 #
          stretcher = 1559.381, # 715827881.667; 4614343880502.55 #
     loc int s := seed;
     s +:= (1-entier(s/shrinker));
     real lshalfofrand = s/Q/Q;
     loc int lobits := s +* twol * P;
     loc int midbits := (s+twol - twol)*P + lobits+twol;
     loc int hibits := ( midbits>=0 | midbits+twom | (midbits+1)+twom-1 );
     midbits +*:= twom +:= P*twol;
     hibits +:= midbits+twom;
     midbits +*:= twom;
     lobits +*:= twol +:= midbits*twol;
     s := lobits - Q + hibits*q;
     if s<0 then s +:= Q fi;
     real rand = seed/Q + lshalfofrand;
     seed := s + entier((s-1)/stretcher) - 1;
     rand
     end
```

6. Conclusions.

A pseudo-random number generator for 16- 32- and 48-bit machines, in strict conformance with the requirements of the Revised Report, has been presented. The versions for 32- and 48-bit machines are entirely satisfactory; that for 16-bits is somewhat less so (suggesting that a fully satisfactory 16-bit generator strictly within the ALGOL 68 specification may be unachievable). Further details and discussion may be found in reference [6].

7. References.

[1] A. van Wijngaarden et al, Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica 5 1,2,3 Dec 1975, also SIGPLAN Notices 12 5 May 1977.

[2] Wichmann, B. A. and Hill, I. D., A Pseudo-Random Number Generator, NPL Report DITC 6/82, 1982.

[3] Wichmann, B. A. and Hill, I. D., An Efficient and Portable Pseudo-random Number Generator, Algorithm AS 183, Applied Statistics, 31 pp188-190 (see also Correction in Applied Statistics 33 p123).

[4] Knuth, D. E., Seminumerical Algorithms, Vol 2 of The Art of Computer Programming, Addison-Wesley 1969.

[5] GRAFTON, R. G. T., Algorithm AS 157: The runs-up and runs-down tests, Applied Statistics 30 pp81-85, 1981.

[6] Domville, I., A Runtime System for ALGOL 68S, 3rd year Project Report, University of Manchester Department of Computer Science, 1984.

AB51.4.2

Interactive Algol68

Peter G. Craven
(Algol Applications Ltd.)

How many non-computing departments have fallen for the bait of an "easy" start and are now teaching their students BASIC as a first programming language ? And how do we answer colleagues even in computing who scorn the "unnnecessary complexity" of Algol68 ?

AB readers can now reply that there is an available Algol68 (subset) implementation in which all one has to type is

```
     1+1;
```

and the answer

```
     2
```

will be displayed immediately. Where one goes from there depends perhaps on the audience, but hardened programmers seem to be impressed by the fact that after:

```
     PROC ferment = (STRING fruit)STRING:

          IF   fruit = "hops"     THEN  "Beer"

          ELIF fruit = "grapes"   THEN  "Wine"

          ELIF fruit = "apples"   THEN  "Cider"

                                  ELSE  "Try it and see !"

          FI;
```

one has only to type:

```
     ferment("apples");
```

to get:

```
     "Cider"
```

.

So what is really happening ? The Algol Applications compiler is a one-pass affair in which forward references are disallowed. In the interactive version it is expecting a series, and after each complete phrase at the global level the newly compiled code is executed immediately. If the phrase was a non-VOID unit, the resulting value is then displayed as a side-effect of the voiding demanded by the language.

Exactly how the values should be displayed is open to discussion. We have taken the view that "straightening" is not appropriate and that beginners would probably prefer to see the value in more or less the same form as they would have to type it in (a denotation or display). Thus

```
[]INT(1, 2+3, ABS "A");
```

will be returned as

```
(1, 5, 33)
```

and for structures we also display the field names, so

```
2 I 3 ^ 2;
```

gives

```
({re} -5.0000000000, {im}  12.000000000)
```

.

The author believes strongly that REFs should be identified as such, so that the vital distinction between

```
REAL x = 7;  x;
```

giving

```
7.000000000
```

and

```
REAL y := 7;  y;
```

giving

```
[6] refers to 7.0000000000
```

is there for all to see. More esoterically, after

```
PROC p = INT: 3;

REF PROC INT r := LOC PROC INT := p;
```

typing

```
r;
```

gives

```
[14] refers to [16] refers to PROC delivers 3
```

whereas

```
r+1;
```

gives

```
4
```

.

The ability of most constructs to deliver values is something most languages regard as an unnecessary luxury, but in an interactive context it gives exactly the right "feel". Indeed, beginners are most disappointed to find that

```
FOR i TO 5 DO i+1 OD;
```

does NOT return

```
(2, 3, 4, 5, 6)
```

WG 2.1 please note !

The compiler is self-compiling and uses recursive descent parsing. The execution model is stack based, using separate static and dynamic stacks as in Algol68C, and all expressions are evaluated on the stack. At present, compilation is to an interpreted intermediate code (bearing somewhat more resemblance to BCPL's O-code than Pascal's P-code) and all operations conceptually demanded by the language are slavishly performed. If one uses a statement-oriented programming style this leads to some inefficiency, since for example after an assignment a separate code is issued to void the result. However by making good use of the expression capablilities of the language, algorithms can be expressed quite compactly, and the code for the current batch compiler occupies only 30K bytes. A nice feature of stack evaluation is that identity declarations generate no code other than that for the right hand side - the compiler simply remembers where the result is and carries on.

The extension to provide incremental compilation was made with remarkably little effort. Effectively the outermost block is parsed specially, and the normal voiding at a semicolon is replaced by a call to execute the newly generated code. At this point the resulting value is on the execution stacks, and the mode information is on the compiler's semantic stack. A recursive procedure now generates and executes the code to strip off REFs and PROCs one by one, displaying to the screen as it goes (and taking care not to dereference NIL nor to deprocedure a comorph!). Finally a plain or stowed value is reached which can be displayed and voided.

Another aspect which gave less trouble than expected is error recovery. At the beginning and again after each successful execution, a snapshot is taken of the compiler and execution states, and a backtrack is performed to this point in the event of a compilation or execution error in the next phrase. In practice errors causing general corruption are rare, and it suffices to snapshot the compiler and execution stack pointers plus a few key variables. Altogether, backtracking is probably easier than continuing after a syntax error in a batch compiler.

More tedious was arranging that the display to the user is consistent with the compiler's view of the world. In the current system, if a compilation error is detected on a line part way through a procedure, a message is inserted above the line and the cursor moves up to the top of the procedure (the backtrack point). One can then use the "cursor down" key to re-submit as much as was good without re-typing. Alternatively extra text can be inserted - maybe to fix up an undeclared identifier - and in this case one has the satisfaction of seeing the error message politely disappear when the offending line is re-submitted in the correct context .

The use of an intermediate code naturally results in execution slower than the raw speed of the processor (Intel 8086) by at least a factor 10. However for the interactive user, the more relevant comparison is with interpreted BASIC, and here Algol68 is seen to be faster by a factor 10. It will be interesting to see whether this fact can be used to woo the many who value speed more than structure !