ALGOL 68 with fewer tears

C. H. Lindsey

Department of Computer Science, University of Manchester, Manchester M13 9PL

ALGOL 68 is a new programming language, designed by Working Group 2.1 of IFIP—the body that produced the 1962 revision of ALGOL 60. This paper, which is intended for the reader with some familiarity with ALGOL 60, is intended as a guide to the new language, emphasising particularly the new features which give it its great power. Many implementations of ALGOL 68 are under way, and the one implementation actually running has successfully compiled this paper.

(Received October 1971)

begin

comment 1. INTRODUCTION.

This is a program written in ALGOL 68. It is a valid program, syntactically, although it does not purport to do anything sensible. Rather, its purpose is to introduce the language ALGOL 68, which is defined in van Wijngaarden and described in more detail in Lindsey and van der Meulen (1971).

There are many similarities between ALGOL 68 and ALGOL 60, and I have rather taken these for granted, concentrating my description on the facilities which are new, or changed.

ALGOL 68 brings with it some new terminology, with which you ought to become familiar. For example, the familiar block structure is still there, but now we must talk of "ranges" rather than blocks when speaking of the region of a program within which a variable has its "scope".

A program in ALGOL 68, then, consists of a range enclosed between begin and end. This is itself presumed to be enclosed within an outer range within which a variety of constants, built-in procedures, etc. are declared.

Comments may be inserted anywhere within a program (i.e. not just between statements), except in the middle of compound characters such as begin and end. I am in the middle of a comment at the moment, and I shall terminate it by writing

comment is also used at the beginning of a comment (I am in another one now). As an alternative to comment # may be used, at either end. Other alternatives are eo etc. These four symbols, of course, cannot be used within comments. As in ALGOL 60, spaces and newlines are everywhere ignored within the program.

The remainder of this paper consists of a series of inner ranges, each describing some section of the language, and each containing all the declarations that are needed for the understanding of that section. It should be noted that, as in ALGOL 60, identifiers consist of a letter, followed possibly by further letters and/or digits. Only one alphabet (either upper or lower case) is required, but implementations are at liberty to add a second, or even more.

# 2. BASIC DECLARATIONS AND STATEMENTS.

begin

comment There are 5 basic types of variable which can be declared:

real a, b;
int i, j, k;
bool p, q;
char c, d;  # character#
compl w, z;  # complex#

# There are also multi-precision versions of real, int and compl: #
long real xxx;
long long # and as many more longs as you like # int yyy;
comment A given implementation is only required to pay attention to a certain number of longs. The number actually effective in the particular implementation is given by the constants "int lengths" and "real lengths" (compl being the same as real). These two built-in constants are termed "environment enquiries" and may be used wherever an int may be used. Here are some more: #

i := max int   # The largest value representable by an int#;
yyy := long long max int # The largest value representable by a long long int#;
#and as many more longs as you like#
a := max real;
xxx := long real; # and so on#
a := small real # The smallest real value such that the result of 1 ± small real # is 1#;
xxx := long small real; # and so on#

comment It is up to your implementation how the values of these various "modes" are packed into machine words. If you want to get at machine words yourself, you should declare: #

bits i, t/;
bytes r;
# bits regards the machine word as a sequence of bools and bytes as a sequence of chars. If you want to know how many, there are some more environment enquiries: #
i := bits width;
i := bytes width;
# and if you like to take your machine words two or more at a time, you can try: #
long bits tt;
long long bytes rrr;

comment There is another mode for taking chars several at a time and this is called string. #

string s;
# The number of chars in a string is flexible, which is very pleasant for the user, even though it makes his implementor work a little harder. #

comment A variable may be initialised when it is first declared: #

real x := 25.427, y := 3.412;
int n := entier x;
# Such initialisations may refer to earlier initialised declarations, even within the same range (but the initialisation of y could not have referred to x because both are part of the same

*The implementation referred to in the summary is that made by the Royal Radar Establishment at Malvern.
declaration). The value of a variable thus initialised may subsequently be changed by assignment.

An identifier may alternatively be made to "possess" the value of some expression by means of an "identity-declaration":#

\[ \text{real } e = 2.718281828, xy = x \times y; \]
\[ \text{int } m = 47; \]

#The values of e, xy and m are fixed and may not be reassigned. In the case of e and m the compiler can treat them as literals and take advantage of any hardware facilities for putting literals in the operand field of an instruction.

The declarations real pi = 3.141592654 etc., long real long pi = the same thing, and so on, are built into the outer range.

The value of a variable thus initialised may be ass\[ a \]

Comment You can now assign values to variables in all the usual ways.#
\[ a := a \times b \times k + x \times y / e; \] #real#
\[ p := q \text{ and } i < j ; q := \text{ true } \] #bool#+
\[ i := f \div 10; \] #int. \div\ implies an integral result and truncation towards zero#\n\[ i := 123; \] #a literal RHS#
\[ c := i ; c := "A" \] #char#
\[ s := "ABCD" \] #string#
\[ w := z \times (i 10) \] #compl. Note the operator i (or \L) for constructing a complex value out of two reals.#+
\[ xxx := \text{long 3.141592654} \]

Comment You can now assign longs to reals;#
\[ a := f \div k; \]
#but if you want to turn reals into ints, you will have to use enter or round, e.g.#
\[ i := f \times \text{ enter } x; \]

Likewise if you want to mix your precisions;#
\[ xxx := \text{which was declared long} := \text{ leng x x leng y}; \]
\[ \# or \# xxx := \text{ leng } (x \times y) \] #which might not be quite the same thing#\n\[ x := \text{short xxx} \] #which may lose you some information#;
#You can also use reals in complex expressions without formality and bytes can be turned into strings;#
\[ s := r; \]
#but a special operator is needed to go the other way;#
\[ r := \text{ctb } s; \]

The number of chars in s had better be \leq bytes length. r is filled out at the right hand end, if necessary, with some null character.#

Comment There is a wide selection of operators built in, both monadic and dyadic.

Operating on reals, ints and sensible mixtures thereof are:
\[ +, -, \times, / \] (between compl and any of compl, real or int),
\[ \Uparrow, \Downarrow \]
\[ \text{re, im, abs} \] (these 3 yield real),
\[ \text{conj} \] (conj \( a = (\text{real } i - \text{ ima}) \)),
\[ i \] (or \L, which we have already met above),
\[ =, \neq \]

On bool and bits there are:
\[ \text{or, and, not, } =, \neq, \]
\[ \text{abs} \] (yields 1 for true, 0 for false and a positive int for any bits).

Also on bits there are:
\[ \geq, \leq \] (\( i \geq t \) is true if each bit of \( t \) implies the corresponding bit of \( i \)),
\[ \Uparrow, \Downarrow \]

\[ \text{elem} \] (\( i \text{ elem } t \) yields its ith bool—also \( i \text{ elem } r \) yields the ith character of a bytes).

Two operators are associated with char:
\[ \text{abs} \] (yields a unique positive integer for each character),
\[ \text{repr} \] (yields the character corresponding to an integer),
and for char, bytes and string there are the relations
\[ <, \leq, =, >, \geq, >. \]

For people who dislike writing#
\[ a := a + b; \]
#(and if \( a \) is subscripted this may indeed compile badly) there is#
\[ a \text{ plus } b; \]
#which has exactly the same effect. Thus there are:#
\[ \text{plus, minus, times, overb, div} \]
or, if you prefer,
\[ +: =, -: =, \times: =, \div: = \]
which work between all pairs of real, int and compl for which the implied assignment would be valid (i.e. \( \times \) plus \( x \) will not do).

\[ \text{overb implies }\div\text{ and div implies }/. \] There is also:
\[ \text{modb} \] (or \( \div: =, /: = \) for ints only; \( a \text{ modb } b \) means \( a := a \mod b \)).

plus works quite well for strings;#
\[ s \text{ plus } "EFGH"; \]
#and there is also a back to front animal called prus;#
\[ "WXYZ" \text{ prus } s; \]
\[ "WXYZABCDEFGH" \]
which work between all pairs of real, int and compl for which the implied assignation would be valid (i.e. \( i \text{ plus } x \) will not do).

\[ \text{overb implies }\div\text{ and div implies }/. \] There is also:
\[ \text{modb} \] (or \( \div: =, /: = \) for ints only; \( a \text{ modb } b \) means \( a := a \mod b \)).

Each operator has a priority, which determines which is considered first in complicated expressions. Low priority means "do this one last". The priorities are:
\[ \text{minus, plus, times, overb, div, modb, prus} \]
or
\[ 2, 3 \]
and
\[ 4, 5 \]
\[ 6, 7 \]
\[ 8, 9 \]
followed by all the monadic operators.

These rules imply that#
\[ i := j \times \text{ enter } x; \] #means the same as#
\[ i := j \times -(\text{ enter } x); \]
#also that#
\[ x := -y \div j; \] #means the same as#
In the case of operators of the same priority, there is implied bracketing as follows:

\[ x := (-y)/j; \]  
# which is not what you might expect.

Thus:

\[ x := \sin (y); \]
\[ y := \text{random}; \]  
# uniformly distributed such that
\[ 0 \leq y < 1 \#

There is also a real variable last random

\[ \text{last random} := y; \]  
# The next call of random will yield the successor of y.

There are also corresponding long real, long long real, etc. procedures called:

long sqrt, long long sqrt, etc. etc.

These take long(s) operands and yield correspondingly long(s) results.

**Comment** In ALGOL 68, an assignation may have a value, and hence it may be used in an expression. Consider the following:

\[ a := b := x + y; \]

# which is the same as

\[ a := (b := x + y); \]

# Of more interest is

\[ a := x + (b := y \times e); \]

# Here, the intermediate result \( x \times e \) is stored in b, which is then added to x. The brackets must be included in order to avoid ambiguity.

**Comment** We will now consider conditional-clauses. In order to permit a further conditional-clause within either half, the delimiter fi. The syntax is

\[ \text{if } \langle \text{serial boolean expression} \rangle \text{ then } \langle \text{serial clause} \rangle \text{ end}; \]

**Comment** A "serial-clause" is the sequence of declarations and "unitary-clauses" separated by semicolons which, for scope purposes, constitutes a range. It can be either a "serial-statement" or a "serial-expression" and is much the same as an ALGOL 60 block or compound statement, except that the delimiters begin and end are not always needed (if they are actually present, it is called a "closed-clause"). Moreover, begin and end may be replaced by "("and")".

This is a closed-statement:

\[ \text{begin} \]

**Comment** I shall first make some declarations.

\[ \text{real } u, v, w; \]

# this real w supersedes the compl w declared earlier and will remain in force until we get to the end#

\[ u := 0; \]  
# unlabelled statements may occur in the midst of the declarations#

\[ \text{int } l; \]  
# note declarations (and unlabelled statements) separated by ;

**Comment** The whole of this declaration part is, of course, optional, but if present it must come at the head of the clause.

\[ w := l := 0; \]  
# but not l := w := 0#

**Loop**:  
# This is a label, and is valid for this range only.##

\[ \text{if } l = 99 \text{ then goto last fi}; \]  
# We shall say more about conditionals later.#
if $i = 1$
then if $j = 1$
then if $k = 1$
then $x := y$
else $x := a$
else $x := a$
fi
fi;

# or "else if";

# Particularly in the 2nd case, it is easy very to lose count of the number of fis. The first case may be rewritten using thef, thus:
if $i = 1$ thenf $j = 1$ thenf $k = 1$ thenf $x := y$ elsef $x := a$ fi;

# The second case may be rewritten using elf, thus:
if $i = 1$ thenf $j := y$
elsef $i = 2$
thenf $x := a$
elsef $i = 3$
thenf $a := x$
elsef $i = 4$
thenf $y := a$;

#\begin{align*}
# & b := x \\
# & \text{else} \ y := x
# \end{align*}#$

fi;

comment The symbols if, then, else, fi, thef, elf may be replaced by respectively, (, |, |, ), |, : |, : |

This looks nearer in the case of conditional-expressions (which are, incidentally, undefined if the else part is omitted).#

$x := y \times (i < j \land a | - a)$;

comment Note that one of the alternatives in a conditional-expression may not yield a value (i.e. it may be a jump).#

$x := y \times (i < j \land a | \text{goto help})$ # The implementor will have fun disentangling his stack here #

help; $b := 0$;

# Now try this one #
$i := (\text{int} \ f := 0; (j \leq a \land e: \ f := j + 1; (j \leq a \land e \ f j - l))$;

# In fact, this is the way in which the meaning of $i := \text{enter a}$ could be defined.#

comment cases in ALGOL 68 do rather more than switches did in ALGOL 60. The example with the elfs above could be replaced by:

case $i$
in $x := y$;

$\begin{align*}
& x := a \\
& a := x \\
& (y := a; b := x) \\
& \text{out} \ v := x
# \end{align*}$#

# Each case is matched by an esc. The out $y := x$ could be omitted, in which case what would happen if $i$ were not in the range 1 to 4 is not defined.

The symbols case, in, out and esc may be replaced by respectively, (, |, |, ).

Here is a case used in an expression.#

days := (m o n t h | 31, (y e a r \ mod 4 = 0 \ and \ y e a r \ mod 100 \ &\ 0) \ or \ y e a r \ mod 400 = 0 | 29 | 28),

31, 30, 31, 30, 31, 31, 30, 30, 31, 30, 31, 30, 31);
to have an expression on the LHS also, provided only that the
type it yields is a name. Thus:

\[(i < 0 \mid x \mid y) = a;\]

Such expressions will not in general be as complex as those
on the right, since the language has lots of built-in operators
for arithmetic quantities, but very few for names.

\[xx := x; x := 2.0;\]
\[yy := y; y := 2.0;\]

comment On occasions, it is necessary to ask whether two
names are equal (i.e. are the same name). Great care is needed
here.#

\[p := xx := yy;\]

This will not do, because the operator "=" is defined for pairs
of reals, but not for pairs of names. This example will give true
because, after double dereferencing, the value of both sides is
2.0. There is therefore a special construction, known as an
"identity-relation":

\[p := (\text{ref real: xx}) := (\text{ref real: yy});\]

On account of the assignments above, this is comparing the
name \(x\) with the name \(y\), and therefore gives false. However,
after:

\[yy := x;\]

it would have given true#

\[p := yy := y;\]

To make the modes the same, coercions may be applied to one side only, but in this case no
coercion is needed because both are already of mode ref ref real.
So this compares the name of the ref real variable \(xx\) with the name of the ref real variable \(yy\), and of course these
are never the same. The variables themselves might be, but to
test this a dereferencing must be forced:

\[p := (\text{ref real: xx}) := (\text{ref real: yy});\]

Begin heap real \(w\) local to
it. Naturally, \(w\) will be stored on a conventional
stack.#

\[w := 10.5;\]
\[xx := w \# xx \text{ refers to the name } w \text{ which refers to the value } 10.5\#
end;

Now we are outside the range. The "Top of stack" has been
moved back; \(w\) has gone and so has its value 10.5. \(xx\) is still
with us, however, but to whom does it refer? The answer is
undefined. The assignment \(xx := w\) ought never to have been
made, because the "scope" of the name \(w\) is less than the scope
of the name \(xx\). The compiler could have caught this case, but
other cases can be constructed which could only be detected
by a run-time check. If we wish \(xx\) to continue to refer, in-
directly, to 10.5; then the 10.5 must be stored somewhere else,
in a region which is termed the "heap".

Begin heap real \(w := 10.5;\)
comment This declares \(w\) to be the name of a place, and
moreover, this name is to refer to a certain fixed
place (not local to this range) where a real can be
put, and which has been initialised to 10.5#
\[xx := w\]
end;
# xx now holds the name of the place which still contains the value 10.5, and which used to be named w.#

comment Now, perhaps, you are becoming confused because the declaration real x creates an object of mode ref real, so let me summarise the situation.#

begin

# You can declare constants#
real e = 2.718281828;
# an identity-declaration always has an “=” in it. e is a real constant. Its mode is real and its value cannot be changed.#
# You can declare variables#
real y; real x := 3.142;
# a variable-declaration never has an “=” in it. x and y are real variables. Their mode is ref real and the values they refer to can be changed. Note that a something variable always has a ref something mode.#
# Of course, you can always declare a constant name#
ref real anotherx = x;
# anotherx is a ref real constant of mode ref real. Alternatively, since it refers to a real value, it could be regarded as a real variable, but this would not stop its mode from being ref real.#
skip end
end;

# 5. PROCEDURES.#
begin

comment There are essentially 4 types of procedure. Viz. a procedure may or may not be accompanied by parameters, and it may or may not deliver a value. When it is declared, the mode(s) of all the parameter(s) (if any) and the mode of the delivered value (if any) must be specified.

It must be emphasised that a proc variable is just like any other variable. It has a “name” which refers to the “place” where its “value” is kept, and its “value” is the “routine” which constitutes its body. A routine is a perfectly respectable value which may be assigned to a proc variable just as a real value can be assigned to a real variable.

Let us, therefore, first consider procedure-variable-declarations.#

proc #void# f1;  
# f1 is the name of a place where a routine will be put. The routine must have no parameters and will deliver no value.#

proc (real, int) #void# f2;  
# f2 likewise refers to a routine which must have a real parameter and an int parameter.#

proc real f3;  
# f3 likewise refers to a routine with no parameters but delivering a real result.#

proc (real, int) real f4;  
# f4 likewise refers to a routine with a real parameter and an int parameter delivering a real result.

When a routine delivers no value, we say it delivers "void", as indicated by the comments in the first two declarations. Most implementations in fact permit, or even require, the symbol void without the comment-symbols around it, and the official specification of the language will probably be brought into line with this in due course.

None of these procs has a body yet. It will have to be provided by assignment later on. However, we can provide a body by initialisation.#

proc g1 := (#void# (l goto l)) ; # i.e. a loop stop;#
proc g2 := (real x, int j) #void# ((j = enter x | l l));
proc g3 := real : p/4;
proc g4 := (real x, int j) real: x x j;
# The right hand side of the initialisation is called a routine-denotation, and consists of a declaration of the formal-parameters, if any, and of the mode of the value delivered (or void), followed by a: "=" followed by a suitable expression or statement, as the case may be. When the right hand side is such a routine-denotation, the left hand side can be simplified (as compared with the cases f1 to f4 above). Now, for example, g1 is the name of a place which contains a routine (which is actually a loop stop—although it may be changed later on).

More usually, however, a proc constant will be declared to possess a specific routine which is not to be changed later.#

proc h1 := # void #: (l goto l);
proc h2 := (real x, int j) #void#: (j = enter x | l l);
proc h3 := real : p/4;
proc h4 := (real x, int j) real: x x j;

# All of these are identity-declarations, in which the routine-denotations follow an equals-symbol rather than a "=".

We can, of course, declare:#

ref proc (real, int) real g4 := g4;

#g4 is now the "name" of a "place" where I may keep the "name" of some other "place" in which I may put a routine with a real parameter and an int parameter that delivers a real value. Initially, the first "place" is to contain the name g4, which has already been declared to be the name of a suitable "other place" in which a routine has already been initially put.

There is also nothing to stop a routine giving, as its value, another routine#

proc generate := (int n) proc real:
begin
  case n in h3, g3, f3, real: p/4 esac
end;
# Finally, let me declare a few variables that we shall need.#
real a, b, x, int i, j;

comment So far, I have declared a bewildering set of procedures, but only 8 of them actually do anything, and of these 4 are at liberty to change their minds. The following proc assignations show how this may be remedied. As usual, the RHS must be (or be coercable to) an expression yielding a value and the LHS must yield the name of a place able to hold that value. The examples below may be compared with some of those in the section on names above.#

f4 := h4;  # (but not h4 := f4)#
# RHS expressions tend to be simple, there being no built-in operators for procs.#
f4 := g4;  # Here, g4 must be dereferenced to yield the routine to which it refers.#
gg4 := f4;  
gg4 := if i < 0 then f4 else gg4 fi;  
f4 := gg4;  #gg4 must be doubly dereferenced#

(ref proc (real, int) real: gg4 := h4;  
(i < 0 | f4 | gg4) := h4;  
gg4 := nil;

# Comment Of more interest are assignations in which the RHS is the routine-denotation itself.#
f4 := (real x, int j) real: x x j;  
f3 := real: p/4;
# In the case of procedures without parameters, such as the last one, the clause which is to become the routine body is a sufficient RHS on its own. Thus:#

f3 := p/4;  # This is a coercion known as "proceduring". However,#

f3 := real: (i < 0 | a | -a);  # means one thing, whereas#
f3 := (i < 0 | a | -a);  
# is quite different, meaning in fact:#

f3 := if i < 0 then real: a else real: -a fi;

comment Having now determined what a given procedure does, let us consider how it may be called:#
h1;  # A loop stop#
g1;  # This one is a loop stop too, at the moment#
The effect of all these calls is as if the body of the routine-denotation had replaced the call.

Note the difference in meaning between $x := h3$ and $g3 := h3$.

The former is a call on $h3$, and actually involves a coercion hi.

If the routine-denotation contained formal-parameters, then these must be replaced by the actual-parameters. If I call $h2$ thus:

```plaintext
h2(a, i);  
```

# then I must consider the routine possessed by $h2$, which is

```plaintext
(real x, int j) # void#: (j = enter x | l: l)  
```

and amend it thus:

```plaintext
(real x = a, int j = i; (j = enter x | l: l))  
```

It will be seen that the routine has now become a respectable closed-clause in which the formal-parameters have become declarations for the constants $x$ and $j$, which now possess the values of the actual-parameters $a$ and $i$. This closed-clause may now be put in place of the call.

It will now be noticed that $x$ and $j$ have been declared as constants so that they cannot be altered by the routine. We have therefore (almost) achieved the effect of the ALGOL 60 "call by value". If we do wish to alter the value of a formal-parameter, then it must be declared ref:

```plaintext
proc enter(i, int a);  
```

# which produces the closed clause

```plaintext
(ref int i) = a; (j = enter x | l: l)  
```

in which it is declared that $j$ is the name of a place holding an integer, and that this name is to be the name $i$ (which is already declared to name a place holding an integer). Thus, when we assign the value enter $x$ to the place named $j$, it is to the place named $i$ that it goes.

This "call by reference" is a little more like the ALGOL 60 "call by name" and will suffice for most of the cases where that is required.

A bound, instead of being "fixed", may be "flexible":

```plaintext
proc series = (int k, ref int i, proc (int) real term) real:  
```

```plaintext
begin  
real sum := 0;  
for j to k do  
begin  
int i := j;  
sum plus term  
end;  
end;  
end;  
x := series (100, i, real: 1/i);  
```

```
# i.e. I may substitute a routine-denotation as the actual-parameter. Since term has no parameter, I could substitute an expression for proceeding instead:  
x := series (100, i, 1/i);  
# which is, of course, Jensen’s device.

This example used a proc without parameters. I could, alternatively, have made the ref int $i$ a parameter of term, as in the following:

```plaintext
begin  
proc series = (int k, proc (int) real term) real:  
```

```plaintext
begin  
real sum := 0;  
for j to k do sum plus term (j);  
end;  
x := series (100, (int i) real: 1/i);  
```

Note the difference in meaning between $x := h3$ and $g3 := h3$.

A call upon a ref proc is a call upon the proc to which it currently ref:

```plaintext
x := gg4 (a, i) # calls f4 or g4 as the case may be;  
```

Similarly for a call upon a procedure delivering a procedure:

```plaintext
x := generate (i)  
```

# the effect of which is to call $h3$, $g3$, $f3$ or real: $pi/4$ according to the value of $i$.#

```plaintext
end;  
```

6. Multiple values.

```plaintext
begin  
int i, j, k, l, m, n;  
comment “Multiple value” is now the in word for “array”. A “multiple”, which is a convenient abbreviation for it, can be declared thus:  
[1 : n] real x1, y1 # one dimensional#;  
[2 : n + 1] real z1;  
[1, m : n] real x2 # two dimensional#;  
[1 : n, 1 : n] real z2;  
```

The bounds of these multiples are "fixed" by the values of $m$ and $n$ at the time of declaration.

```plaintext
int:  
```

```plaintext
end;  
```

These are multiples in which both bounds are flexible and are initially [1 : 0]. They are therefore empty but may be filled later on.

One can, of course, have names of multiples,

```plaintext
ref [] real xx1;  
```

(because the name of the multiple value referred to cannot be specified in the declaration),

```plaintext
and multiples of names,  
[1 : n] ref real xx2;  
```

and names of multiples of names of multiples,

```plaintext
ref [] ref [] real xx3;  
```

and multiples of procedures.

```plaintext
[1 : 4] proc # void#: switch = (goto e1, goto e2, goto e3, goto e4);  
```

```plaintext
```

Here, I have in fact declared 4 separate procedures whose bodies are given by the 4 elements of the row-display. Each element is in fact a procedureable clause—I could have used routine-denotations but I would have taken more space. I can call one of these procedures by:

```plaintext
switch [i];  
```

```plaintext
comment  
```

Assignments of multiple values can assign the whole multiple at once:

```plaintext
The Computer Journal
```

182 The Computer Journal
If one or both of the bounds on the LHS is fixed, the corresponding bound on the RHS must match it exactly (in the case above, both sides were \([1 : n]\)). If one or both of the LHS bounds is flexible, then it is reset from the corresponding bound(s) of the RHS:

\[ \text{y2} := \text{y1}; \]

\[ \text{y2} \text{ now has bounds \([1 : n]\), but the \text{n} is still flexible, and may be changed again later.} \]

The value of a row-display can be assigned provided the lower bound of the LHS is 1 (or is flexible, in which case it becomes 1). #

\[ \text{i} := (2, 3, 4, 5, 6); \]

\[ \text{y4} := (1.2, 2.3, 3.4); \]

# A true row-display has at least two elements. For multiple values with only one element (by virtue of a coercion known as “rowing”) a simple arithmetic value (in general, any multiple with one dimension less) does instead: #

\[ \text{y4} := 1; \] # The bounds of \text{y4} are now \([1 : 1]\) again.

For multiples with no elements, \(<\text{empty}>\) may be used, but by convention it is always made into a closed-clause so as to be more conspicuous:

\[ \text{y4} := (); \] # The bounds of \text{y4} are now \([1 : 0]\).

# comment One may take a part of a multiple value, by using a “trimmer”, and assign to or from it. A trimmer specifies two bounds, which select a part of a row of the multiple. After this selection, the bounds are “slid down” so that the lower bound becomes 1 (or as otherwise specified by an \text{at}). #

\[ \text{y3} := \text{xl}[2 : n - 1]; \] #\text{y3} which has flexible bounds

\[ \text{y3} := \text{xl}[2 : n - 1 \ at \ 2]; \] #\text{y3} is now \([2 : n - 1]\) on account of the \text{at} 2.

If a lower- (upper-) bound in a trimmer is omitted, the existing lower- (upper-) bound of the multiple is implied:

\[ \text{y4} := \text{y3}[; , n - 2]; \] #\text{y3}[n - 2] is implied, since \text{y3} is currently \([2 : n - 1]\). \text{y4} becomes \([1 : n - 3]\).

\[ \text{y4} := \text{y3}[; ]; \] # where both bounds are omitted, the \text{a} is omitted also, and there is no “sliding down”.

\[ \text{y4} \text{ is therefore now } [2 : n - 1]. \]

\[ \text{xl}[2 : n - 1] := \text{y3}[\text{at} 1]; \] # both sides, after sliding, are \([1 : n - 2]\), but it is \([2 : n - 1]\) of \text{xl} that gets altered.

Note that \text{xl}[2 : n - 1] := \text{y3} would not have been accepted, but:

\[ \text{xl}[2 : n - 1 \ at \ 2] := \text{y3}; \] # would.

Of course, \text{xl} := \text{z1} is not allowed (the bounds do not match even though both have \text{n} elements), but:

\[ \text{xl}[\text{at} 1] := \text{z1}[\text{at} 1]; \] # will achieve its presumably intended effect. Note that the effect of:

\[ \text{xl}[2 : n] := \text{xl}[1 : n - 1]; \] # is well behaved, and the implementor had better ensure that he starts copying from the top end. #

# comment One may take an individual element out of a multiple by means of a “subscript”:

\[ i := i[\text{xl}]; \]

\[ \text{i}[3] := \text{z2}[4]; \]

# If the multiple has two or more dimensions, then both trimmers and subscripts may appear:

\[ \text{x1} := \text{z2}[4]; \]

\[ \text{x1} := \text{z2}[4, ]; \]

\[ \text{x1}[\text{: } m] := \text{z2}[4, : m]; \]

\[ \text{x1} := \text{z2}[1, 4]; \] # assigns the 4th column of \text{z2}.

\[ \text{x1}[2 : n - 1] := \text{z2}[2 : n - 1, 4]; \] # assigns part of the 4th column of \text{z2}.

Do not be afraid of creating names which refer to parts of multiples:

\[ \text{xxl} := \text{x1}; \] # The value of \text{xxl} is the name of \text{x1}.

\[ \text{xxl} := \text{x1}[2 : n - 1]; \] # The value of \text{xxl} is the name of part of that multiple, the whole of which is named \text{x1}.

However, it is not permitted to assign the name of a part of a flexible multiple, for this would become meaningless if ever the original multiple has its bounds altered. Thus \text{xxl} := \text{y3}[3 : n - 2] is not allowed.

Operators \text{lwb} and \text{upb} are provided to discover the actual bounds of any multiple value. Thus:

\[ i := 2 \text{upb} \text{x2}; \] # sets \text{i} to the 2nd upper bound of \text{x2} (i.e. \text{n}) and #

\[ i := \text{lwb} \text{z1}; \] # sets \text{t} to the 1st (and only) lower bound of \text{z1} (i.e. \text{2}). I.e. \text{lwb} and \text{upb} exist in both dyadic and monadic forms.

When multiple values are used as formal-parameters in procedures, it must be specified whether the bounds of the actual multiples that are to be substituted are expected to be fixed (which is the default state) or flexible (indicated by \text{flex} or either (either)). In the following example, the lower bound of \text{z2} is to be fixed, and its upper bound is to be either. Thus \text{x1} and \text{y2} could be substituted as actual parameters, but \text{y3} could not.

begin
real \text{x};
proc \text{sum} = ([: \text{either}] \text{real} \text{z1}) \text{real}:
begin \text{real} \text{sum} := 0;
for \text{i} from \text{lwb} \text{z1} to \text{upb} \text{z1} do \text{sum plus} \text{z1}[/];
\text{sum end};
\text{x} := \text{sum(x1)};
end;

# comment Note that \text{x1} := \text{x1} + \text{x1} is not a valid statement of the language, because the meaning of the operator “+” is not defined for multiples. However, it will be shown in Section 10 how such a meaning for “+” could be defined by the user.

# comment Now, at last, you can be told the truth about string.

begin
# In Section 1 there was declared
string \text{s};
but it would have meant exactly the same thing had I written \([1 : \text{oflex}]\) char \text{s};

# So! string is just an abbreviation. But look what can now be done with it.

\[ \text{s} := "ABCD"; \] # clearly, the string-denotation

"ABCD" implies the bounds \([1 : 4]\).

\[ \text{string} \text{t} := \text{s}[2 : 3]; \] # it can be trimmed.

\[ \text{char} \text{c} := \text{s}[4]; \] # or subscripted.

\[ \text{s} := \text{s}[1]; \] # or shortened.

\[ \text{i} := \text{upb} \text{s}; \] # or measured.

\[ \text{s} := ( ); \] # or emptied.

\[ \text{s} := "ABCD"[\text{i}] + \text{t} + \text{c}; \] # or restored to its former glory.

Now I shall invert the middle of it:

\[ \text{s}[2 : 3] := \text{s}[3, \text{s}[2]]; \] # using a row-display;

and finally it shall be enclosed in a quotation:

\[ "* * * * * * * * *" \]

# Note how a pair of quotes-symbols (" *) is used as a denotation for the quotes-symbol itself—confusing, possibly, but
can you think of a better way that does not sacrifice one of the other characters on your teleprinter, which you might have liked to use as a character on its own account?

Now, in case you are lost, let us print it out and see what it looks like:

```plaintext
print(5)
whose gives, on some external medium, the six characters
"ACBD"
end;
```

# 7. STRUCTURES.*

```plaintext
begin
int i, compl, w, z; real x;
comment A "structure" is a set of values of various types that are associated together as a unit that may be handled like any other value (e.g. it may be assigned to other similar structures).

In order to declare a structure, one may first invent a new mode to describe it:

struct person = (string name, int age, bool male, ref person

spouse, [1 : 0flex] ref person children);

# Then one may declare various items to be of this mode.#

person mary := ("MARY", 15, false, nil, ());
# note the use of (empty) for children of mary.#

tom := ("THOMAS", 19, true, nil, ());

person albert := ("ALBERT", 42, true, skip, (tom, mary));

person edna := ("EDNA", 39, false, albert, children of

albert),

ann, giles;

# We shall now fill in albert's spouse:#

spouse of albert := edna;

# We can now write#

ann := ("ANN", 18, false, nil, ());

# Let us now marry ann and tom and produce the inevitable consequences:#

spouse of tom := ann;

spouse of ann := tom;

children of tom := children of ann := giles :=

("GILES", 0, true, nil, ());

# We shall now calculate the sum of the ages of albert's children#

i := (int sum := 0;

for i to upb children of albert do

sum plus age of (children of albert)[i];

sum);

comment Note that the mode compl is really a structure, declared in the outer range by:

struct compl = (real re, im).

Thus it is possible to refer to, e.g. re of z and im of w:#

x := re of w; # which has the same effect as#

x := re w;

# Contrariwise, re w := x is not permitted, because re yields a value, not a name. However,#

re of w := x # is all right because w is a name and so is

re of w.#

end;
```

# 8. MODES.*

```plaintext
begin
comment It is possible to invent new modes identical to other modes, or consisting of combinations of them, and the invented modes hold throughout the range in which they are declared (unless redeclared differently within an inner range).#

mode reel = real;

is not really helpful, but#

mode funnyarray = ref [ ] ref [ ] real;

might save some ink, if I had to declare a lot of them. Moreover,#

mode fixedarray = [1 : 5] real;

has built-in fixed bounds [1 : 5].

A new kind of structure can be created thus:#

mode person = struct(string name, int age, bool male, ref

person spouse, [1 : 0flex] ref person children);

# which is just a slightly different way of writing the line which began#

struct person = (string name, . . .

in the previous section.

A mode which is a combination of several others can be declared thus:#

union intreal = (int, real);

and variables of this mode thus:#

intreal xi;

or thus:#

union (int, real) yj;

The place named xi can be used to hold either an int or a real, but this does not actually save any space, since the place must also hold, at run time, an indication of which mode is currently in use.#

real x, y; int i, j; bool p;

comment The following assignations are now all legitimate:#

xi := x #xi now holds a real#;

xi := i #xi now holds an int#;

xi := yj #yi now holds whatever yj held#;

xi := 1.0 #xi now holds a real#;

xi := 1 #xi is now quite definitely int#;

# However, i := xi and x := xi are not legitimate, because there is no knowing what might happen if the current mode of xi was not the right one.

I may discover the current mode of a variable such as xi by means of a "conformity-relation":#

p := i :: xi; # makes p true if the current mode of xi is the same as that of i (i.e. int).

Note that the first operand of a conformity-relation must yield a name. The other may be either a name or a value. The value of a united operand such as xi may be assigned by means of a "conforms-to-and-becomes" operator:#

i ::= xi;

# (Remember that i ::= xi is not allowed). The assignment only takes place if i ::= xi is true, and the value of the whole thing is true or false.

As an alternative to a conformity-relation, there is the "case-conformity":#

case x, i ::= xi in print(x), print(i) esac;

# which means the same as:#

if x ::= xi then print(x)

else i ::= xi then print(i)

fi;

comment Saving space was not, however, the prime reason for having unions in the language. They are needed where one has a procedure which may be called with an actual-parameter of one of several different modes:#

proc checkdate = (int day, union(string, int) month) bool:

begin
int m, string s;
```

The Computer Journal
[1:12] struct[[1:3]] char m, int d) table =
("JAN", 31), ("FEB", 29), ("MAR", 31),
("APR", 30), ("MAY", 31), ("JUN", 30),
("JUL", 31), ("AUG", 31), ("SEP", 30),
("OCT", 31), ("NOV", 30), ("DEC", 31));

case s, m := month
in for i to 12 while m := i; m of table[i] ≠ s[1:3]
do skip
skip
esac;
day ≤ d of table[m]
end;

# after which both the following calls should give false#
p := checkdate(30, "FEBRUARY");
p := checkdate(30, 2)

comment Another important application of unions occurs when handling tree structures, wherein a node of the tree may contain either pointers to other nodes of the tree or (if the node is a tip) some useful piece of information. An example of this will appear in the next section.#

end;

# 9. GENERATORS.#

begin
comment So far, I have described how storage may be obtained in fixed chunks (on the stack) or in flexible chunks (on the heap), by means of variable-declarations. There exist applications, however, in which storage requirements are quite unpredictable in advance, and in which chunks of storage need to be grabbed and released in a higgledy-piggledy order in accordance with the data for the particular problem. Traditionally in the past, these applications have been written in some List Processing language (e.g. LISP).#

struct link = (string title, ref link next);
ref link start := nil;  # initially, the chain is empty#

proc insert = (string name) # void #: # a procedure to insert a new link at its correct alphabetical position#
begin
ref link pointer := start;
# pointer can point at any ref link variable such as start or, more importantly, the next field of a link variable (a field of a struct variable is itself a variable)#
while ((ref link: pointer) : ≠: nil | name ≥ title of pointer | false)
do pointer := next of pointer;
end;

# a generator to remove a link from the chain#

end # of insert#;
insert("AB"); insert("BC"); insert("BA");

proc remove = (string name) # void #: # a procedure to remove a link from the chain#
begin
ref link pointer := start;
while if (ref link: pointer) : ≠: nil then if name = title of pointer then (ref link: pointer) := next of pointer; # At this point, the offending link has been bypassed in the chain. There is now no ref link value in the whole system which still points to it, and so it has become quite inaccessible—# it is garbage. If ever storage space becomes tight, a LISP-like garbage collector will be called in to make its storage space available for further use.#
else name > title of pointer
fi
else false
fi
do pointer := next of pointer
end # of remove#;
remove("BA");

proc make tree = (string s) ref operand:
begin
ref operand expr, term, factor, ref formula f;
expr := term := factor := heap operand := s[1];
for i from 2 by 2 to upb s − 1 do
begin
char c = s[i];
if c = "\n" then (ref operand: factor) := heap formula :=
(factor, c, s[i + 1])
else c = "\*" or c = "^" then (ref operand: term) := f := heap formula :=
(term, c, s[i + 1]);
factor := right of f
else c = "\+" or c = "−" then (ref operand: expr) := f := heap formula :=
(expr, c, s[i + 1]);
factor := term := right of f
else # invalid string#
goto exit # which is the standard way to terminate a program#
fi
end;
expr
end # of make tree#;
ref operand tree := make tree(\texttt{'X Y Z + A B C'});
proc print polish = (operand a) \# void\#:
begin
  char ch, ref formula rf;
case ch, rf := \ldots a
in
  print(ch),
  (operator of rf); print polish(left of rf);
  print polish(right of rf))
esac
end;
print polish(tree)
\# which should print the characters:
+ $\times$ $\times$ $X$ $Y$ $Z$ $\times$ $A$ $B$ $C$ \#
end;

\# 10. OPERATORS.\#
begin
comment All the existing dyadic operators have an associated
priority, as listed earlier. One may invent new operators, or
redfine existing ones, as follows:\#
priority min = 9;
\# However, priority + = 6, although valid, would lose all the
existing definitions of +.
One may now invent meanings for new operators, or alter the
meaning of existing ones, by providing a routine which
determines what the operator is to do:\#
\#10. OPERATORS.\#
\# which should print the characters:
\# + $\times$ $\times$ $X$ $Y$ $Z$ $\times$ $A$ $B$ $C$ \#
end;

\# 11. TRANSPUT.\#
begin
comment A given implementation is aware of a number of
different types of device (tape readers, punchers, printers,
teletypes, mag tapes, discs, wind tunnels, etc.). The Report
refers to a device type as a "channel".
All transput goes to or from "files". A file may be a deck of
cards, a bundle of lineprinter output, a magnetic tape, an area
on the disc, etc. The user is initially provided with one opened
file on each of the 3 standard channels \texttt{stand in channel},
\texttt{stand out channel} and \texttt{stand back channel}. If he wishes to use
any more files on these or other media, he must first declare his
file names:\#
\# file input, my output:\#
\# These names are identifiers, by which the file is referred to
from within the program. On some devices (e.g. mag tape), a
file also possesses an external string identification. File is a
struct built-in to the outer range by
struct file = (string term, proc bool logical file end, physical
file end, format end, value error, proc(ref
char)bool char error, proc(int)bool other error)
The user may then open these files by means of procedures
provided, and likewise he may close them again later on. The
names of the 3 standard files declared and opened within the
outer range are:
\# file \texttt{stand in}, \texttt{stand out}, \texttt{stand back}.\#
The filepointer, which gives the position in a file at which the
next transput will start, consists of a page number, a line number,
and a character number. Maximum values for each of these
quantities are associated with each channel according to the
implementation. The user may always enquire what is the
current position of the file pointer, and in the case of a random
access device he may set it wherever he likes. Otherwise, it
always starts off at (1, 1, 1) when the file is opened or reset (i.e.
rewound) and automatically moves forward as transput pro-
ceds.

The following useful procedures are predefined within the
outer range (the list is by no means exhaustive):

\begin{verbatim}
proc open = (ref file file, string idf, int channel) void:
  Open a new file, with the specified external identification (where
  appropriate), on the channel specified.

proc create = (ref file file, int channel) void:
  Open an empty file on the channel specified, with the file, page
  and line lengths set to the default values for the channel.

proc establish = (ref file file, string idf, int mp, ml, mc,
  channel) void:
  Open an empty file on the channel specified, to be identified by
  idf, and with the number of pages, page size and line length
  equal to mp, ml and mc (all to be within the channel default sizes).

proc space = (file file) void:
  Advance the filepointer by 1 char (you must not overflow a line).
  I.e. give one space on output, or ignore one character on input.

proc backspace = (file file) void:
  Move the filepointer back 1 char (but do not get yourself before
  the beginning of the current line).

proc newline = (file file) void:
  Move the filepointer to the beginning of the next line (but do
  not get yourself off the end of your page).

proc newpage = (file file) void:
  Move the filepointer to the beginning of the next page.

proc close = (file file) void:
  Close the file.
\end{verbatim}

Transput is effected by means of special procedures which will
now be described. They are divided into 6 categories, according
to whether the transput is in or out, and whether it is format-
less, binary or formatted.

\begin{verbatim}
Formalless Output
There is a proc called put:
proc put = (file file, [ ] ???) void:
\end{verbatim}
which outputs the ???s to the specified file. For [ ]???, one may substitute any single item (be it int, real, bool, char, bits, bytes, string, compl or any struct, or multiples of any of these things, or a proc(file) such as space, newline or newpage) or a row-display whose elements are a mixture of such single items, e.g.:#
real x, y; int n; [1 : n] real x1; string s;
put (my output, x \times y);
put (my output, "X \times Y = \ldots", x ! y, \ldots");
x1[2 : n - 1], newline, s);
#(Note the "\ldots" denotation for the char "space") All the
items listed are then output on the named file in order:
ints as a space, followed by enough digits to accommodate
max int preceded by a sign, leading zeroes being turned into
spaces.
reals as a space, followed by a sign and enough digits of
fractional mantissa to give the accuracy implied by small
real, followed by a sign and enough exponent digits to
accommodate max real.
compls as two reals separated by "\ldots".
bools as 1 or 0.
chars and strings just as they are.
If strings run into the end of a line, physical file end of the file
is called. This is one of the "error procedure" fields of the mode
file. If you have previously assigned a suitable procedure
to this field, it will now be called, in default of which the result
is undefined. The other error procedures associated with the
file may similarly be used to trap other exception conditions.
If an item of a mode other than string would overflow a line,
a newline (or page) is started before the item.
A multiple value (struct) is treated as the sequence of items
that constitutes its elements (fields). If a multiple is more than
one dimensional, it appears row by row, and so on. Note that
strings, chars, bits and bytes, when printed, are not separated
by spaces in any way.
There is also a proc called print:
proc print = ([ ] ???) void:
which outputs all the ???s, as above, on the standard file stand out.

Formatted Input
There is a proc called get:
proc get = (file file, [ ] ???) void:
which reads the ???s from the specified file. For [ ]???, one may substitute the name of any single item (be it int, real, bool, char, bits, bytes, string or any struct, or multiples of any of these things, or a proc(file) such as space, newline or newpage) or a row-display whose elements are a mixture of such single items, e.g.:#
get (my input, x);
get (my input, (x, y, newline, s, x[2 : n - 1]));
#The items are then sought, in turn, on the given input file.
In the case of multiple values, the elements thereof are sought in
turn. The existing number of elements in the multiple is the
number that is sought for except in the case of a multiple of
chars with at least one flex bound (e.g. a string—see below).

When seeking an int, spaces and newlines (or pages) are ignored
until a sign, or a digit is found. After + or −, further spaces
are ignored until a digit. Digits are then read until a non-
numeric character or the end of the line is found, and thus
the int is determined.
When seeking a real, it first seeks an integer, as given above. It
will then accept "point" and a fractional part immediately
following, and a "\ldots" and exponent immediately following
that. The first space, newline, or other unexpected character
encountered terminates the whole operation.
When seeking a compl, it seeks two reals separated by "\ldots".
When seeking a bool, it accepts 1 or 0, ignoring spaces, new-
lines, or new pages.

When seeking a char, it ignores newlines or new pages, but
otherwise takes the first character offered.
When seeking a string, it takes chars up to the end of the line,
or until it encounters one of the chars which the user may
have included in the term field of his file (which field is empty
by default). The terminating char itself is not yielded. When
seeking a multiple of chars with fixed bounds, it seeks the
appropriate number of chars and, if the end of the line occurs
before this, physical file end of the file is called.
There is also a proc called read:
proc read = ([ ] ???) void:
which reads all the ???s, as above, on the standard file stand in.

Binary Output
There is a proc called put bin:
proc put bin = (file file, [ ] ???) void:
where [ ]??? is as described for put above. The last operation on
file must not have been a read, unless it is a random access
device.
The specified items are written in sequence to the file, starting
at the current page/line/char and continuing over as many
chars, lines and pages as may be necessary. The form in which
it is written is such that it may be recovered by a subsequent
get bin, but is otherwise undefined.
There is also a proc called write bin:
proc write bin = ([ ] ???) void:
which writes the ???s as above on the standard file stand back.

Binary Input
There is a proc called get bin:
proc get bin = (file file, [ ] ???) void:
where [ ]??? is as described for get above. The last operation on
file must not have been a write, unless it is a random access
device.
The specified items are read in sequence from the file, starting
at the current page/line/char.
There is also a proc called read bin:
proc read bin = ([ ] ???) void:
which reads the ???s as above from the standard file stand back.

Formatted Transput
We shall now introduce a new mode called format. #
format a, b, c;
# formats are like any other variables. They can be assigned to
each other, occur in multiples or structures, have names, be
initialised or possessed at declaration, be united with other
modes, be delivered by procedures, etc. There are, however, no
facilities for operating on them, nor means of creating oper-
ators that would do so. Therefore, the value of any format must
eventually be traced back to a format-denotation such as the
following, which occurs in an assignation:#
int bin;
a := $ p "table \_\_\_ of " x 11a,
n(bin - 1) (2l2zd,
3(8k \_1.12de+2d\_+j \_ x " si +.10de+2d l) p $ ;
# comment A format-denotation is enclosed between $ and $.
Between these, it consists of a number of "pictures", separated
by commas. There are two pictures in the above example.
The first is
p "table \_\_\_ of " x 11a
and the second is
n(bin - 1) (2l2zd, 3( . . . )) p
which is to be interpreted as a list of (bin - 1) pictures each of
which holds
2l 2zd, 3( . . . )
which itself consists of two pictures, the second of which itself
consists of 3 further pictures.
Thus the format may be expanded into an arbitrarily long list of
basic pictures. Each picture now corresponds to a single
value that is to be transput.
A picture consists of a pattern, with insertions scattered before, after, and in the middle of it. Thus in:

\[
p \cdot \text{table of } x \cdot 11a
\]

there are the insertions

\[
p \cdot \text{table of } x \cdot \text{meaning move to start of next page}
\]

\[
\text{outf (file format, [ ] ???);}
\]

\[
\text{This will read in, according to the format } a. \text{ Where literal strings occur in the format (e.g. \text{table of } and \text{ +jx }), they must appear exactly in the input stream. If, however, the file pointer is moved (for example by insertions involving } p, \text{ l, x, k etc.), then any parts of the input stream skipped over may contain rubbish. When trying to match patterns, the exact number of characters implied by the pattern must be present (but there is a special pattern } i \text{ which can be used to read a string up to the end of the line, or the term of the file).}
\]

Here are some more examples:

\[
\text{outf (my output, } 3d, 3d \cdot 999999); \text{ prints 999,999}
\]

\[
\text{outf (my output, } c(\text{Sun, Mon, Tues, Wednes, Thurs, Fri, Sat}) \cdot 3d, 4); \text{ prints Wednesday.}
\]

A picture consists of a pattern, with insertions scattered and the pattern

\[
\text{end}
\]

\section*{12. COLLABORAL PHRASES.}

\begin{itemize}
  \item \textbf{comment} A collateral-statement consists of a list of unitary-statements separated by commas, the whole being enclosed in brackets. The only forms of collateral-expression are the row-display and structure-display which have already been introduced. A collateral-declaration consists of a list of unitary-declarations separated by commas (but without brackets). Note that real a; real b; really stands for real a, real b;
  \item The notable thing about such collateral-statements and declarations is that their constituents may be executed simultaneously.
\end{itemize}

\begin{itemize}
  \item Similarly, the two sides of an assignment, the actual parameters of a call, the operands of an expression—in fact almost all pairs of objects that do not actually have a semicolon between them—are \enquote{elaborated collaborally}. The only practical effect of this in most cases is that the order in which the constituents are executed becomes undefined, to the consternation of users of side effects (hooray!).
  \item If, however, a collateral-statement is preceded by \textbf{par}, this signifies that the user really intended the constituents to run in parallel because he had a compiler and some hardware that could profitably do this. In this case, he can make use of two special operators defined in the outer range:
\end{itemize}

\begin{itemize}
  \item \textbf{op down = (sema dijkstra) void:}
  
  sema (for semaphore) is a special mode used for communication between cooperating processes. If \textit{dijkstra} is \leq 0, the part of the \textbf{par} statement in which the \textit{down} occurred is halted.
  
  Otherwise \textit{dijkstra} is reduced by 1.
  \item \textbf{op up = (sema dijkstra) void:}
  
  All the statements previously halted on account of \textit{dijkstra} are resumed, and \textit{dijkstra} is increased by 1. For a full discussion of this technique, see Dijkstra (1968).
\end{itemize}

\begin{itemize}
  \item \textbf{References}
\end{itemize}

\begin{itemize}
\end{itemize}