17. Blocks and scopes

We have just seen how to initialise variables in a data brick. Within a procedure, one of the first things we often do after our declarations is to ensure that some of our variables contain particular values. Such declaration followed by assignment can be combined into an initialisation – a convenient shorthand and a means (which can be checked at compile-time) of ensuring that ref-variables contain sensible objects. It is important to remember that since local variables only exist when the procedure is being executed, a dynamic assignment will occur; because of this, the value of our initialisation is now allowed to be a general expression – that is any expression that is a valid right hand side of an assignment to the variable. Dereferencing, type-changing and operators are therefore allowed. No default values are defined, and on entry to a procedure a non-initialised variable will contain whatever value happens to be in that particular location in the run-time stack.

The syntax is precisely as in a data brick (though, of course the question of initialising arrays cannot arise since they cannot be declared locally):

```
PROC LOCAL (INT I, REAL R, REF REAL P);
    INT J:=I*20, K:=J-2, L:=I*I+INT R;
    REAL A:=3.1, B:=C:=R/I, D:=SQRT(R);
                    % PROCEDURE SQRT DEFINED ELSEWHERE %
    REF REAL Q1:=Q2:=A, Q3:=P;  % NOTE DEREFERENCING OF P %
    .
    .
ENDPROC;
```

Note that we can use variables initialised in the heading (in particular any parameters which will be initialised by a call), but that the following declaration is ridiculous:
INT T := S + 7, S := + 1;

We are trying to use S before it has been declared (and an initial value put in it) and the compiler will generate an error message to this effect.

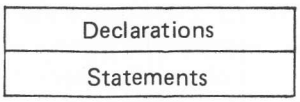
As in data bricks, it is obligatory to initialise reference variables; the reasons are those given in the last section, though perhaps it is even more important here, since we have no idea what the original contents might be when a random cell on the stack is associated with such a variable. The value that we really wish to have in a reference variable may not be known at the point of declaration; for instance it may be an array element whose subscript value has not yet been calculated. It is irritating and inefficient to have to store dummy names (and perhaps have to create them!) just to satisfy this rule. (It's not so bad in a data brick, since it consumes no effort at run-time).

To overcome this, we introduce a further kind of statement which is allowed to contain declarations (and hence initialisations) which we can then place at the desired point in our program. This also has other useful properties. This statement is the *block-statement.* It is introduced by the keyword BLOCK and terminated by the matching keyword ENDBLOCK (recall the comments about matching keywords and indentation in Section 11). Between these delimiters, the rest of the statement consists of a number of declarations (including initialisations as above) followed by a sequence of statements. These embedded statements, as usual, can be as complicated as we like: assignments, procedure-calls, goto statements, conditional statements and further block-statements.

Example:

```
BLOCK
    INT K:=2;
    REF INT Q:=K;
    IF K=Q THEN GOTO L;  END;
ENDBLOCK;
```

The internal structure of the block-statement can be pictured as:



Declarations and statements may not be mixed.

This configuration is called a *blockbody.* A moment's thought (plus a turn back to section 8 perhaps) will show that this is precisely the form of the body of a procedure and indeed the body of a procedure definition is a blockbody. Variables declared at the head of a procedure definition are local to that procedure or blockbody; similarly variables declared at the head of a blockbody are local to that blockbody; we shall return to this point below.

We now construct a more useful illustration of the use of the block-statement by rewriting our earlier 'order' example using blocks and initialisations.

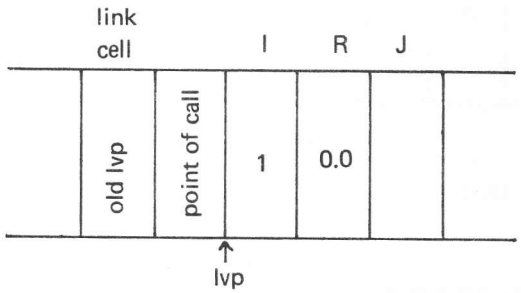
```
PROC ORDER (REF ARRAY REAL A);
INT I:=1,          % COUNTER %
    LEN:=LENGTH A;% COMPUTE LENGTH ONCE %
NEXTEL:
   BLOCK
   REF REAL MIN:=A(I);  % ASSUMED MINIMUM %
   INT J:=I+1;          % COUNTER %
CHECK:
      IF A(J)<MIN THEN MIN:=A(J);  END;
      J:=J+1;
      IF J<=LEN THEN GOTO CHECK;  END;
      BLOCK
      REAL TEMP:=A(I);  % TEMPORARY LOCATION %
         A(I):=MIN;
         VAL MIN:=TEMP;
      ENDBLOCK;
   ENDBLOCK;
   I:=I+1;
   IF I<LEN THEN GOTO NEXTEL;  END;
ENDPROC;
```

This illustrates nested block-statements; all variables are initialised on declaration, and the use of block-statements allows us to declare our local variables at just the points where we want to start using them.

Variables declared in a block-statement are local to the block; as locals, they exist as cells in the run-time stack. Similarly, a name set as a label within the block-statement is local to that block. If we have two (or more) non-overlapping block-statements, the local storage for each will be shared. This is best illustrated by an example ignoring the actual statements of the procedure.

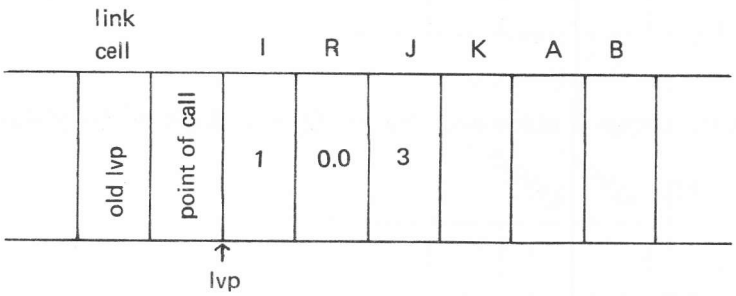
```
PROC P (INT I, REAL R);
    INT J:=3;
        .
        .
        .
L:      BLOCK
        INT K:=4;
        REAL A:=B:=2.0;
            .
            .
            BLOCK
            REF INT RI:=K;
            REF REAL RR:=R;
                .
                .
            ENDBLOCK;
        ENDBLOCK;
        BLOCK
        INT Q:=1;
        REF INT RQ:=Q;
        REAL ALPHA:=21.6;
        REF REAL GAMMA:=ALPHA;
            .
            .
        ENDBLOCK;
        .
        .
        GOTO L;
ENDPROC;
```

On entry to P (by a call of that procedure, say P(1, 0.0)) the stack will appear as:



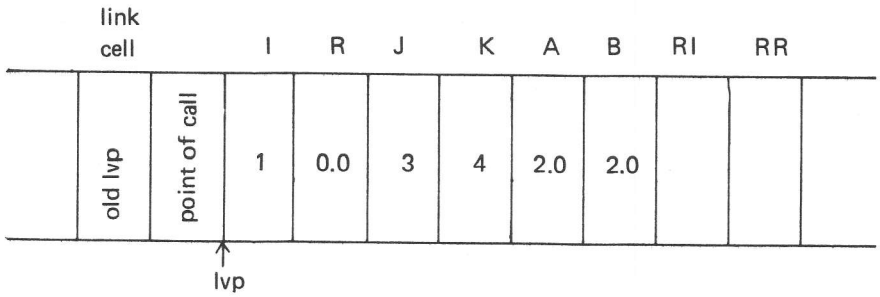
The first statement to be obeyed will initialise J.

On arrival at L, we enter a new block, and on the stack we now gain access to the variables K, A, B:

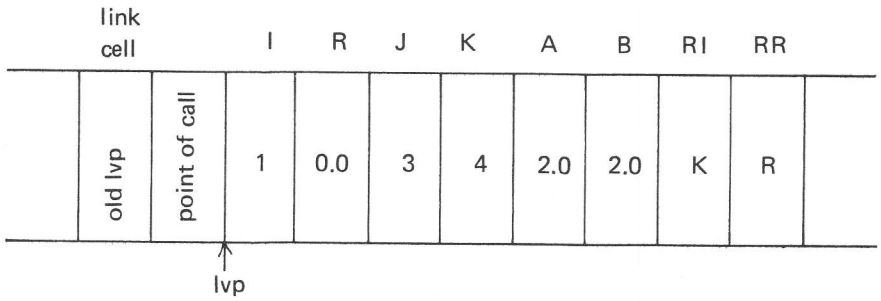


Notice that there is no change to the local variable pointer (lvp) and no creation of a new link cell; the local variables already in existence on entry to the block appear as globals within the block.

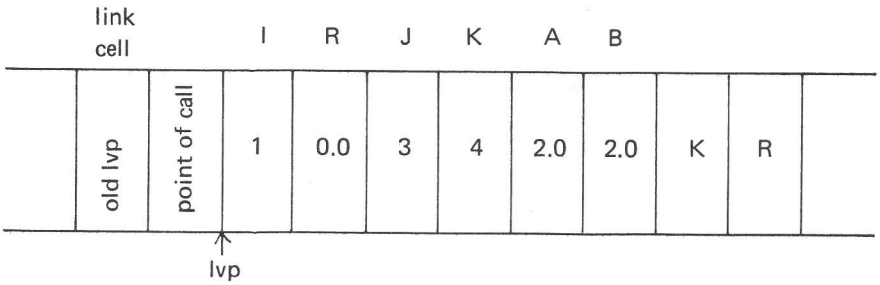
The initialisations to K, A, B will now be performed. On encountering the next block statement storage is made available for RI, RR:



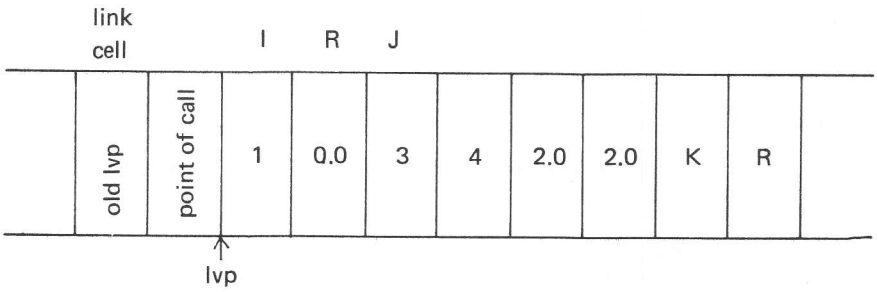
RI and RR are then initialised:



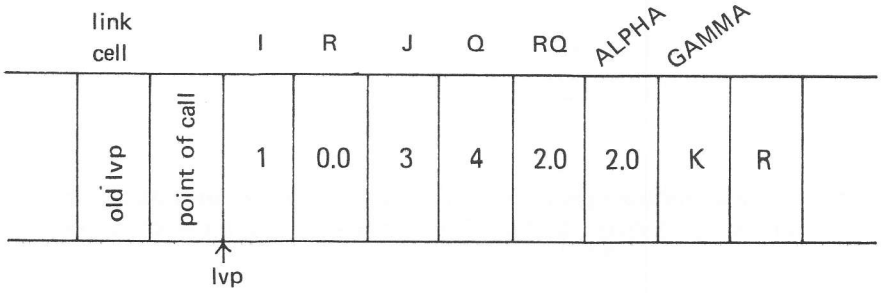
As soon as we reach ENDBLOCK, the block-statement containing RI and RR is terminated, and these variables are lost; the actual cells and their contents are still in the stack of course, but no names are associated with them, and they are inaccessible.



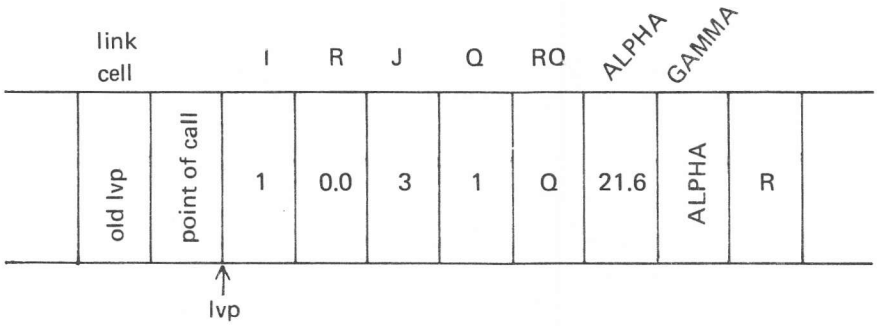
Similarly on reaching the next ENDBLOCK, K, A, B are lost:



We next encounter a new block-statement; storage is made available for Q, RQ, ALPHA, GAMMA:



The initial values are those that happen to be in the stack from the last uses of those locations; in some cases the value is incompatible with the mode of the variable – hence the need for initialisation:



On leaving this block, Q, RQ, ALPHA, GAMMA will be lost, and we will transfer control to L, where K, A, B will again be made accessible. They will use the same cells as before, but in the meantime these have been used by Q, RQ, ALPHA; hence, again the importance of initialisation. To ensure that this initialisation is performed, entry to a block-statement is only allowed through the keyword BLOCK: this will be elaborated below.

We have seen the sharing of storage in the stack. The total amount of storage is allocated on entry to the procedure, the association between a variable and a particular cell having been organised by the compiler. On entry or exit to a block-statement (or *inner block* to distinguish it from the block represented by the whole procedure definition) there is no overhead as there is on procedure entry/exit when link cell and certain other housekeeping manipulations must be performed.

We use the concept of blocks to define completely where a variable may be used and hence to enable us to formulate rules for the declaration of a variable using a name used elsewhere in the program for a different purpose – we have already seen an example of this in section 9 where we used DIAM as the name of a parameter for both CIRCLE and P2.

The *scope* of an identifier name (that is "where you can use it") is a lexicographic and dynamic constraint on its legal use : the scope of an identifier is the block in which it is declared or set as a label plus any inner blocks, unless the identifier is re-declared or reset as a label within such an inner block. This explains why we were able to continue using I, LEN in our inner blocks in ORDER : they were still "in scope" – in terms of the stack they were still accessible. An attempt to use a variable outside its valid scope is termed "taking out of scope" and is illegal. A simple example is:

```
BLOCK
    INT I;
    .
    .
ENDBLOCK;
    .
    .
I:=3;
```

The assignment to I is not in the scope of I – this is a simple lexicographic error.

A more difficult case can arise:

```
BLOCK
    INT J:=3;
    REF INT RI:=J;
    .
    .
    BLOCK
        INT K;
        .
        .
        RI:=K;
        .
        .
    ENDBLOCK;
    VAL RI:=7;
ENDBLOCK;
```

Here, there is no lexicographical illegality – all the variables are used within their scopes. Dynamically, however, when we come to perform the final assignment, RI will be dereferenced to yield as destination K, a variable no longer in existence. The fault occurs in placing the name of K in a reference variable which is declared in an embracing block, so that the name may be taken out of scope dynamically. It is not possible for the compiler to decide absolutely in this case, but it can recognise the potential danger inherent in the assignment RI:=K and issues a warning; such warning messages will be mentioned later, but the advice not to ignore them is given now!

We can deduce from our definition the scopes of variables declared in a block-statement and in a procedure heading (including the parameters). What about the other names in our program? The scope of the names of bricks (i.e. used to name a data brick or a procedure definition) and the scope of variables declared in a data brick is the whole program; that is we regard the beginning and end of our program as defining the outermost block of our nested structure. This scope of data brick variables and the names of procedures fits in with our earlier global usage of them. In this way, a block structure is imposed upon our program.

We can now see why we can only jump to a label in our current procedure:

```
PROC F1 ();
        .
        .
L:
        .
        .
ENDPROC;

PROC F2 ();
        .
        .
    GOTO L;
        .
ENDPROC;
```

The statement GOTO L is illegal since L is not in scope at this point. Similarly

```
      .
   BLOCK
      .
      .
L;
      .
   ENDBLOCK;
      .
      .
   GOTO L;
      .
```

is illegal; L is again out of scope in the goto-statement. This means that we cannot jump into a block; we can only enter it through BLOCK or through the normal procedure entry. We have already seen that this is a desirable restriction, since we wish to ensure that all initialisations (particularly of reference variables) are performed.

Of course, jumping within a block or out of it is permissible; in the case of a block-statement no housekeeping on the stack is required; in the case of a procedure we can only jump out via RETURN anyway. Thus the following are all legal:

```
   BLOCK
      .
      .
L;
      .
      GOTO L;          % L IS IN SCOPE %
   ENDBLOCK;
   BLOCK
      .
      .
      GOTO M;          % M IS IN SCOPE %
   ENDBLOCK
M;

   BLOCK
      .
      .
      RETURN;          % EXITS FROM ALL INNER BLOCKS AND THE PROCEDURE
   ENDBLOCK;
```

In the case of multiple declaration or definition of use of names we have a problem; when we encounter such a name in a statement of our program, we have to decide which variable it identifies. The rule "work outwards through the block structure until a declaration of a variable or a setting of a label with that name is found" solves it. If no such declaration or setting is found, either the variable is out of scope, or a declaration/setting has been omitted. Naturally we wish to avoid ambiguity in this process (if we end up with an integer variable *and* a label we will not know what to do) and this together with our scope rule leads to the following constraints (embodying our earlier rules), each of which is illustrated by incorrect and correct programs (the outermost block in each could be PROC. . .ENDPROC):

1. No identifier name may be declared as a variable or set as a label more than once in a block.

ILLEGAL

```
BLOCK
   REAL A;
   REF INT A;


ENDBLOCK;
```

```
BLOCK
   REAL L;

L:
   GOTO L;

ENDBLOCK;
```

```
BLOCK
L:

L:
   GOTO L;

ENDBLOCK;
```

VALID

```
BLOCK
   REAL A;

   BLOCK
      REF INT A;
   ENDBLOCK;
ENDBLOCK;
```

```
BLOCK
   REAL L;
   BLOCK
L:
   GOTO L;
   ENDBLOCK;
ENDBLOCK;
```

```
BLOCK
L:
   BLOCK
L:    % * %
   GOTO L;    % GOES TO * %
   ENDBLOCK;
ENDBLOCK;
```

2. A variable declared, or label set, in a block cannot be referenced outside that block; i.e. cannot be taken out of scope (lexicographically or dynamically).

ILLEGAL

```
BLOCK
   BLOCK
   INT I;

   ENDBLOCK;
   I:=7;
ENDBLOCK;
```

```
BLOCK
   BLOCK
   L:

   ENDBLOCK;
   GOTO L;
ENDBLOCK;
```

```
BLOCK
REF INT RI:=I;
   BLOCK
   INT J;
      RI:=J;

   ENDBLOCK;
   VAL RI:=7;
ENDBLOCK;
```

```
PROC P () REF INT;
INT I;
   RETURN (I);
ENDPROC;
```

VALID

```
BLOCK
   BLOCK
   INT I;
      I:=7;
   ENDBLOCK;

ENDBLOCK;
```

```
BLOCK
   BLOCK
   L:
      GOTO L;
   ENDBLOCK;

ENDBLOCK;
```

```
BLOCK
REF INT RI:=I;
   BLOCK
   INT J;
      RI:=J;  % COMPILER WARNING %
      VAL RI:=7;
   ENDBLOCK;

ENDBLOCK;
```

3. A variable declared, or label set, in a block is not accessible in an inner block if the name has been re-declared or re-set therein. The entity represented by the name is still technically in scope, and continues to exist, but becomes temporarily (in the inner block) inaccessible.

ILLEGAL

```
BLOCK
REAL L;
   BLOCK
   L:
      L:=0.1;
      GOTO L;
   ENDBLOCK;
ENDBLOCK;
```

VALID

```
BLOCK
REAL A;
   BLOCK
   L:
      A:=0.1;
      GOTO L;
   ENDBLOCK;
ENDBLOCK;
BLOCK
REAL L,M;
   BLOCK
   L:
      M:=0.1;
      GOTO L;
   ENDBLOCK;
   L:=M;
ENDBLOCK;
```

In general, the multiple use of names should be restricted as far as possible; it should certainly be avoided whenever it leads to loss of clarity in the program text.

The use of inner blocks is recommended. It leads to efficient use of storage on the stack and increases legibility since declarations occur at appropriate points rather than in a messy muddle at the head of the procedure. It also removes the need for dummy initialisations in procedures and, in fact, leads to less errors caused by using a variable (*thought* not to contain anything significant) for temporary calculation purposes. We can also now see that a procedure call is *in effect* an inner block:

```
PROC SWAP (REF REAL A,B);
REAL TEMP:=A;
   VAL A:=B;
   VAL B:=TEMP;
ENDPROC;

PROC MAIN ();
REAL P,Q;
      .
      .
      SWAP(P,Q);
      .
      .
ENDPROC;
```

is equivalent to:

```
PROC MAIN ();
REAL P,Q;
      .
      .
      BLOCK
      REF REAL A:=P,B:=Q;
      REAL TEMP:=A;
         VAL A:=B;
         VAL B:=TEMP;
      ENDBLOCK;
      .
      .
ENDPROC;
```

Finally, in this section, we introduce two concepts connected with the implementation of RTL/2 through the use of a run-time stack. We are interested here in establishing the meanings of two words, rather than a detailed investigation.

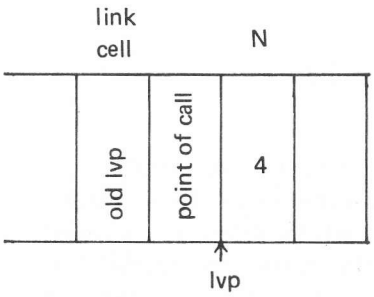
If, in the body of a procedure, a call is made of that same procedure, it is said to be a *recursive procedure.* The simplest example is:

```
PROC P ();
   P();
ENDPROC;
```

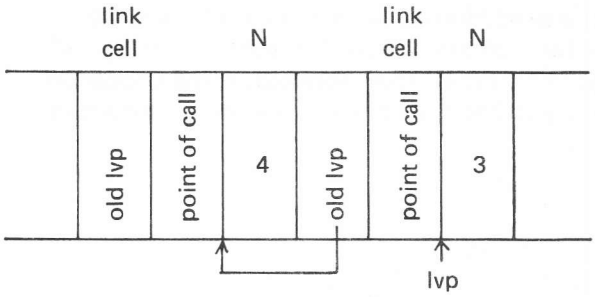
This example is futile in that when P is called it will simply call itself ad infinitum. Consider the following:

```
PROC FACTORIAL (INT N) INT;
   RETURN(IF N=1 THEN 1 ELSE FACTORIAL(N-1)*N END);
ENDPROC;
```

This procedure calls itself (provided N#1) and is therefore recursive; it computes factorial N, i.e. the product N(N−1)(N−2)...3.2.1. This is possible since successive calls generate new incarnations of the local variables in the run-time stack. Let us follow the stack through a call FACTORIAL (4):

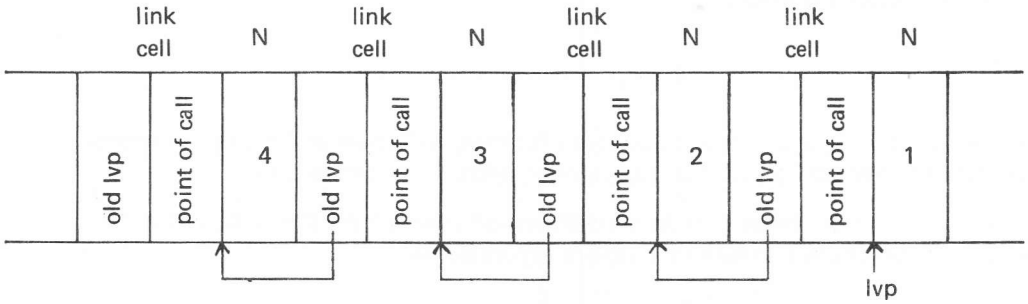


4#1 hence FACTORIAL (3) will be called:



This generates a new link cell and a new N.

Similarly FACTORIAL (2) and FACTORIAL (1) will be called:



On this call, N=1 and so a result of 1 is returned; we unwind and now form the product 1*2 in the incarnation FACTORIAL (2); this is returned and we unwind again to form (1*2)*3 in the incarnation FACTORIAL (3); one further procedure exit gives us 1*2*3*4 which is then returned as the result of FACTORIAL (4).

Recursion is made possible by the stack mechanism. It can be very powerful and useful; it can also be very wasteful of time and stack as the following example shows. You are invited to sketch the stack manipulations for a call FIB(99).

```
PROC FIB (INT N) INT;
   % GENERATES THE NTH FIBONACCI NUMBER %
   % U(N) = U(N-1) + U(N-2) %
   RETURN(IF N=1 THEN 1 ELSEIF N=2 THEN 1
                ELSE FIB(N-1) + FIB(N-2) END);
ENDPROC;
```

This concept should not be confused with a call of a procedure one of whose parameters involves a call of the procedure; SQRT(SQRT(81.0)) is not a recursive situation. The root of 81.0 is evaluated and then becomes the parameter for a second call of SQRT; SQRT does not call itself.

Recursion can be hidden in the sense that P1 calls P2 which calls P3 which calls P1 which calls ... This situation is termed *mutual recursion.*

As a final example of recursion, we re-present our earlier example of extracting the highest common factor written in a recursive manner:

```
PROC HCF (INT A,B) INT;
   RETURN(IF B=0 THEN A ELSE HCF(B,A MOD B) END);
ENDPROC;
```

If we consider two users running two programs in two independent stacks (or if we simply consider our case of recursion above) who use a common procedure (we investigate how this is made possible in a later section) we can see that a situation can arise in which a procedure which is already in use can be called and entered. We have seen that the stack(s) make this possible for the local variables. We must also ensure that the actual instructions in the procedure are the same whenever we call it. This is achieved by making the code *read-only;* i.e. no procedure (including itself) affects the instruction sequence. These two properties (read-only and separate incarnation of locals) make RTL/2 procedures *re-entrant.* Thus if a procedure is being executed it can be interrupted, another program can call the same procedure, and on return (re-entry) nothing will have been affected and processing can safely continue. This is not true, however, if the procedure manipulates data brick variables, since new incarnations of these are not created on a procedure call.

Section 17 examples

1. Rewrite the procedure solving a quadratic equation (coping with real and complex roots) using blocks to code the evaluation of the square root within the procedure.
2. Smoothing of data points in an array F is to be performed (except on the first and last points in the array) to reduce the effect of random errors using

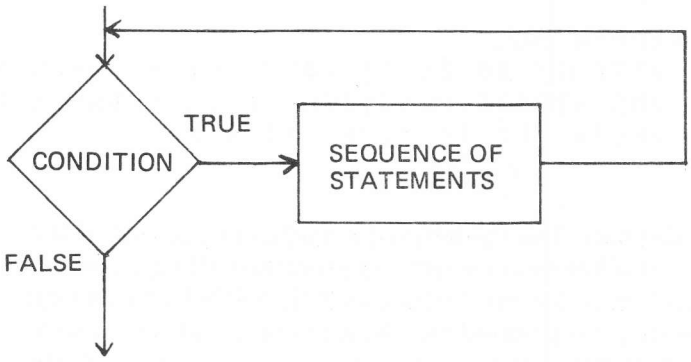
$$F_i := \frac{F_{i-1} + F_i + F_{i+1}}{3}$$

   Write a procedure to do this.
3. Write a procedure to evaluate the mean and standard deviation of a set of values supplied in an array parameter.
4. Rewrite the earlier matrix product example using inner blocks.
5. Write a procedure to return the value of a real number raised to an integer power. Attempt to write a recursive procedure to do this. (Hint: to reduce multiplication, decompose the exponent into a suitable form).
6. Recursive procedures provide a neat way of coding integers in punched card code prior to physical output to a card punch in card column order. Given that the card codes for the digits 0, 1, ..., 9 are 512, 256, 128, ..., 2, 1, write a simple procedure to output the appropriate codes to a card punch (via a second procedure PROC CARDCOLUMN (INT CD);) for a positive integer, X. No formatting is required. Digits are determined by successive division of X by 10 and use of the remainders to index an array of codes. It is worthwhile to attempt this with and without the use of recursion.

18. Loops I: While statement

From section eleven onwards, we have seen a number of examples where a problem has been solved by an iterative approach (e.g. extraction of square root, ordering an array). In general, an iterative loop consists of a series of actions to be obeyed a specified number of times (as in the case of ordering elements) or until a given condition is satisfied (as in reaching an acceptable degree of accuracy.). The former case can always be cast into the latter form. The conditional situation is treated in this section.

Up to now we have programmed loops by a combination of conditional statements and goto statements. We now develop a shorthand form called a *while-statement* or *while-loop.* The while-statement programs the situation represented by the simple flowchart:



In the syntax for this, we require three delimiters; one to mark the beginning of the statement, one to mark the end, and one to terminate the condition and thereby indicate the beginning of the statement sequence. These delimiters are the keywords WHILE, REP (for repeat), DO respectively. We can represent the syntax in the symbolic form:

```
WHILE     condition is true     DO
              sequence of statements
REP;
```

WHILE and REP are matching keywords. As we have now encountered a number of such pairs, it is appropriate to point out that such compound statements formed by the use of matching keywords must be strictly nested within one another; interleaving is not allowed.

Thus
```
IF A<B THEN
    BLOCK
        .
        .
        .
END;
    ENDBLOCK;
```

is illegal, since the inner block statement has not been terminated when we attempt to terminate the outer conditional statement by END. Legal versions are

```
IF A<B THEN                    BLOCK
  BLOCK                          IF A<B THEN
        .                              .
        .                              .
        .                              .
  ENDBLOCK;                      END;
END;                           ENDBLOCK;
```

%NOTE THE INDENTATIONS%

The condition in the while-statement is of precisely the same form as that used within a conditional statement.

Example:

```
LET OPEN=1;
LET SHUT=0;

DATA PLANTDATA;
   ARRAY (NOOFVALVES) INT POSITION;
ENDDATA;

      .
      .
      WHILE POSITION(VALVENO)=OPEN DO
         CLOSE(VALVENO);     % ATTEMPT TO CLOSE VALVE WITH GIVEN NUMBER %
                             % AND UPDATE POSITION IF SUCCESSFUL %
         DELAY(10);          % DELAY FOR 10 TIME UNITS %
      REP;
```

This behaves precisely as indicated in the flowchart. The contents of a particular element of the array POSITION is inspected; if it is the value OPEN then we call the procedure CLOSE which initiates an attempt to close the valve in question and if successful sets POSITION (VALVENO) to the value SHUT. We then delay for a specified time period (to allow the physical action to be performed). Encountering REP sends us back to evaluate the condition; if it is still true, a further attempt is made to close the valve; otherwise, we continue our processing with the statement immediately following REP. Hence we have a loop terminated as soon as a condition fails to be satisfied (or as soon as the opposite, the valve being shut, is satisfied).

We can re-write our square-root extraction procedure using a while-statement:

```
LET EPS=0.001;

PROC SQRT (REAL X) REAL;
REAL OLDGUESS:=1.0, NEWGUESS:=(X+1.0)*0.5;
                  % FIRST ITERATION PERFORMED IN INITIALISATION %
   WHILE ABS(NEWGUESS-OLDGUESS)>EPS*NEWGUESS DO
      OLDGUESS:=NEWGUESS;
      NEWGUESS:=(NEWGUESS + X/NEWGUESS)*0.5;
   REP;
   RETURN(NEWGUESS);
ENDPROC;
```

Clearly the while-statement

```
WHILE condition DO
        sequence of statements
REP;
```

is equivalent to the following labelled if-statement:

```
L:  IF condition THEN
        sequence of statements
        GOTO L;
    END;
```

All we have done in introducing the while statement is to leave the burden of organising the label control to the compiler (it may be able to do this more efficiently). We have also added to the clarity, and explicitly indicated the iterative nature of the situation.

Section 18 examples

1. Rewrite the program to order the elements of an array using while-statements.
2. It is known that the equation f(x)=0 has just one root in the range x=a to x=b. This is characterised by the fact that f(a) and f(b) have different signs. One method of finding the root is by "repeated bisection"; at each iteration the range is reduced to a half of its previous value but such that the function values at the end points have different signs. As soon as the range is less than the desired accuracy, either endpoint (or the midpoint of the range) is an approximation to the root required.

   Write a program to do this for some arbitrary function (which you need not define) using while-statements.
3. A procedure PROC SCAN () REAL reads a value from an instrument. PROC DELAY (INT N) delays for a time interval of N centi-seconds. Write a loop which reads the instrument at suitable intervals until switch-on transients have decayed to some small percentage level.

19. Loops II: For and to statements

A more complex structure in RTL/2 provides the form of iterative scheme in which the number of times the loop is performed is important. A *for-statement* signifies that a series of values is given in succession to a variable (of mode integer) and that for each value a sequence of statements is to be performed. As before this is a shorthand, and in many situations where up to now we have used if-statements and goto-statements, we would normally write a for-statement. The following example recodes the initialisation of the tables in the iterpolation example (section 14, number 4) using a for-statement, and will be used to indicate its structure and rules.

```
FOR I:=1 BY 1 TO 100 DO
REAL XVAL:=I*0.1;
   X(I):=XVAL;
   Y(I):=F(XVAL);
REP;
```

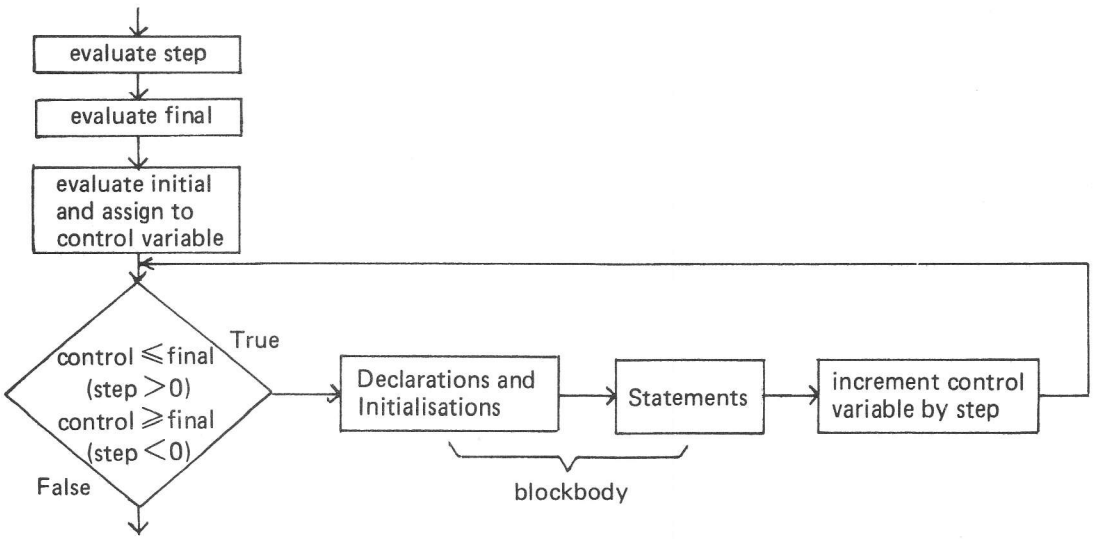
In a similar way to the while-statement, three keywords delimit the statement; it is introduced by FOR and terminated by REP (FOR, REP form another matching pair); the keyword DO separates the heading of the statement from its body. This body starts with the initialised declaration of a real-variable XVAL; the main body of a for-statement is a blockbody and thus consists of declarations followed by a sequence of statements; of course, a blockbody doesn't *have* to contain declarations – we may have a null set of declarations. The heading specifies the sequence of values to be given to the *control variable*; in this case the variable is named I and will take the values 1, 2, 3, ..., 99, 100. The syntax of the heading is:

1. The keyword FOR introducing the statement.
2. An assignment to an integer control variable; the right hand side is any integer expression and is the *initial value* for the loop. The nature of the control variable will be discussed in more detail below.
3. The keyword BY introducing a *step value*; this value is specified by an arbitrary integer expression, which may be positive or negative.
4. The keyword TO introducing the *final value*; this again is specified by an integer expression.
5. The keyword DO terminates the heading.

If the initial value is i, the step s, and the final value f, then the sequence of values taken by the control variable is:

i, i+s, i+2s, i+3s,... while $i+ns \leqslant f$ $(s>0)$
$i+ns \geqslant f$ $(s<0)$

Clearly the terminating condition depends on whether the step value is positive or negative. The three integer expressions in the heading are evaluated once upon entry to the loop, and we can now present a flowchart of the action of the for-statement.



1. Note the order of evaluation: step, final, initial. This is important if any of the integer expressions contain procedure calls which have side effects or manipulate data brick variables affecting the remaining expressions. It needs to be emphasised that, although the values are dynamic (i.e. expressions calculated at run-time) they are only evaluated once on entry to the loop.
2. The sign of the step will affect the nature of the termination test to be performed; if it is positive we check that the current value of the control variable does not exceed the final value; if it is negative we check that the control variable is not less than the final value. The action if the step is zero is not defined — the program may well loop infinitely.
3. Note the inclusive nature of the loop: we do not actually have to hit the final value; as soon as the control variable passes it, the loop is terminated. On termination, of course, the next statement is the one immediately following REP. Since the control variable is integral, there are no cumulative error problems should we wish the final value to be taken.
4. The control variable is declared by its presence after FOR and, of course is always initialised. It is deemed to be declared of mode integer in the blockbody of the loop. This means that on exit from the loop, (either by normal completion or by explicitly jumping out), the control variable is inaccessible since it is out of scope.

```
        FOR I:=1 BY 2 TO 20 DO
            .
            .
            IF A=B THEN GOTO L;  END;
            .
            .
        REP;
L;      EXITVALUE:=I;       % ILLEGAL; I OUT OF SCOPE %
```

We must write:

```
        FOR I;=1 BY 2 TO 20 DO
            .
            .
            IF A=B THEN EXITVALUE:=I;  GOTO L;  END;
            .
            .
        REP;
        EXITVALUE;=21;      % NOTE TERMINATION VALUE OF I %
L;
```

The control variable has one other property; it is read-only. This means that an assignment cannot be made to it, neither directly nor by assigning its name to a reference variable. The programmer cannot therefore corrupt the loop control nor play funny games with it! (Naturally, there is assignment to the control variable in the 'increment-by-step' stage, but this is out of the control of the programmer).

5. As the body of a for-statement is a blockbody, we cannot jump into it; entry must be through its heading. Thus the following is illegal.

```
        FOR I:=A BY B TO C DO
            .
M;
            .
        REP;
        GOTO M;
        % ILLEGAL; M OUT OF SCOPE %
```

However, as usual, we can jump out of the block.

6. Note that, as with the while-statement, the loop may be performed zero times. The loop test is at the head of the loop, and if the condition is false on first entry, then the loop sequence will not be obeyed; the next instruction will be the one following REP.

   We now present some examples (some of them familiar) illustrating the use of the for-statement.

```
%* 1 *%
PROC TRACE (REF ARRAY (,) INT Q) INT;
% RETURNS SUM OF THE DIAGONAL ELEMENTS OF Q ASSUMED NON-NULL AND %
% SQUARE MATRIX %
INT SUM:=0;
   FOR I:=1 BY 1 TO LENGTH Q DO
      SUM:=SUM + Q(I,I);
   REP;
   RETURN(SUM);
ENDPROC;
```

```
%* 2 *%
PROC ORDER (REF ARRAY REAL A);
% PUTS ARRAY INTO ASCENDING NUMERICAL ORDER %
% NOTE THAT FOR A NULL ARRAY THE ACTION IS NULL %
   FOR I:=1 BY 1 TO LENGTH A - 1 DO
   REF REAL MIN:=A(I);
      FOR J:=I+1 BY 1 TO LENGTH A DO
      % NOTE THAT THE INNER LOOP'S INITIAL VALUE IS A FUNCTION OF %
      % THE OUTER LOOP VARIABLE AS IN OTHER CONSTRUCTIONS %
         IF A(J)<MIN THEN MIN:=A(J);  END;
      REP;
      BLOCK
      REAL TEMP:=A(I);
         A(I):=MIN;
         VAL MIN:=TEMP;
      ENDBLOCK;
   REP;
ENDPROC;
```

```
%* 3 *%
LET PRIME=1;
LET NONPRIME=0;
LET NOOFELS=100;

DATA NUMBERS;
   ARRAY (NOOFELS) INT P:=(PRIME(NOOFELS));
ENDDATA;

PROC SIEVE ();
% FINDS PRIME NUMBERS UP TO NOOFELS; P(I)=PRIME IF I IS A PRIME %
% NUMBER. METHOD IS THE SIEVE OF ERATOSTHENES %
   % 1 IS ASSUMED PRIME; SO IS 2 %
   FOR I:=2 BY 1 TO NOOFELS DO
      IF P(I)=PRIME THEN
         % MARK ALL MULTIPLES OF I AS NON-PRIME %
         FOR J:=I BY I TO NOOFELS DO
            % NOTE DYNAMIC STEP %
            P(J):=NONPRIME;
         REP;
      END;
   REP;
ENDPROC;
% THIS IS A VERY CRUDE METHOD AND IS SLOW FOR LARGE VALUES OF NOOFELS %
```

```
%* 4 *%
% ILLUSTRATES NEGATIVE STEP %

   FOR COUNT:=10 BY -1 TO 0 DO
      WRITEINTEGER(COUNT);      % OUTPUT NUMBER %
      IF COUNT#0 THEN
         DELAY(100);      % WAIT FOR 1 SECOND %
      END;
   REP;
   BLASTOFF();
```

In many examples the step value will be 1; as an additional shorthand, the 'BY step' part of the heading may optionally be omitted; if it is omitted, a step value of +1 will be used by default. Note that if the initial value is greater than the final value and the BY part is omitted, a default value of −1 will *not* be assumed; +1 is assumed and the loop will be obeyed zero times.

For instance, a very crude method of calculating the nth power of a real would be:

```
PROC POWER (REAL A, INT N) REAL;
REAL POW:=1.0;
   FOR I:=1 TO ABS N DO      % NOTE LACK OF BY ELEMENT %
      POW:=POW*A;
   REP;
   RETURN(IF N<0 THEN 1.0/POW ELSE POW END);
ENDPROC;
```

Another situation which occurs is that in which a loop is to be performed a fixed number of times and in which no control variable is required.

A further shorthand form of the loop is provided for this case; it has the syntactic form

```
TO integer expression DO
    blockbody
REP;
```

It is equivalent to a standard for-statement with initial and step values of 1, and final value the given expression. Note that it still defines a new block.

Hence we can write

```
PROC POWER (REAL A, INT N) REAL;
REAL POW:=1.0;
   TO ABS N DO POW:=POW*A;  REP;
   RETURN(IF N<0 THEN 1.0/POW ELSE POW END);
ENDPROC;
```

This form can often be compiled more efficiently; in particular the compiler can take advantage of any loop instructions in the computer's instruction set. Whenever the control variable is not required explicitly in a loop, a simple TO construction can always be used, the required number of iterations being given by

(FINAL–INITIAL):/STEP + 1

Section 19 examples

1. Rewrite the matrix product example using for-statements.
2. Rewrite the example calculating the mean and standard deviation of a set of values using for-statements.
3. Write a procedure to return the mean of a set of values and replace each element of the set by its deviation from the mean.
4. Rewrite the smoothing example (section 17 number 2) using for-statements.
5. A look-up table (similar to the function tables for Y in section 14 number 4) is provided by means of two arrays, one containing integer index values, the second the corresponding real value of some parameter. (Thus given an index q, if integerarray (i)=q then realarray(i) is the required value). Write a procedure to derive the value of the real parameter from a given integer index.
6. Two arrays INPUT and OUTPUT are in a data brick. A tape containing N + 1 integers where N is the value of the first integer on the tape can be read sequentially by calls of the procedure PROC IREAD( ) INT.

   Each integer, in the range 1–5 specifies one of the following actions:

   1: Read in a new array

   2: Print out the array OUTPUT

   3: Replace the values in OUTPUT by their deviations from the mean of their values (use example 3).

   4: Scale the input array into the output array, the conversion being from Fahrenheit to Centigrade.

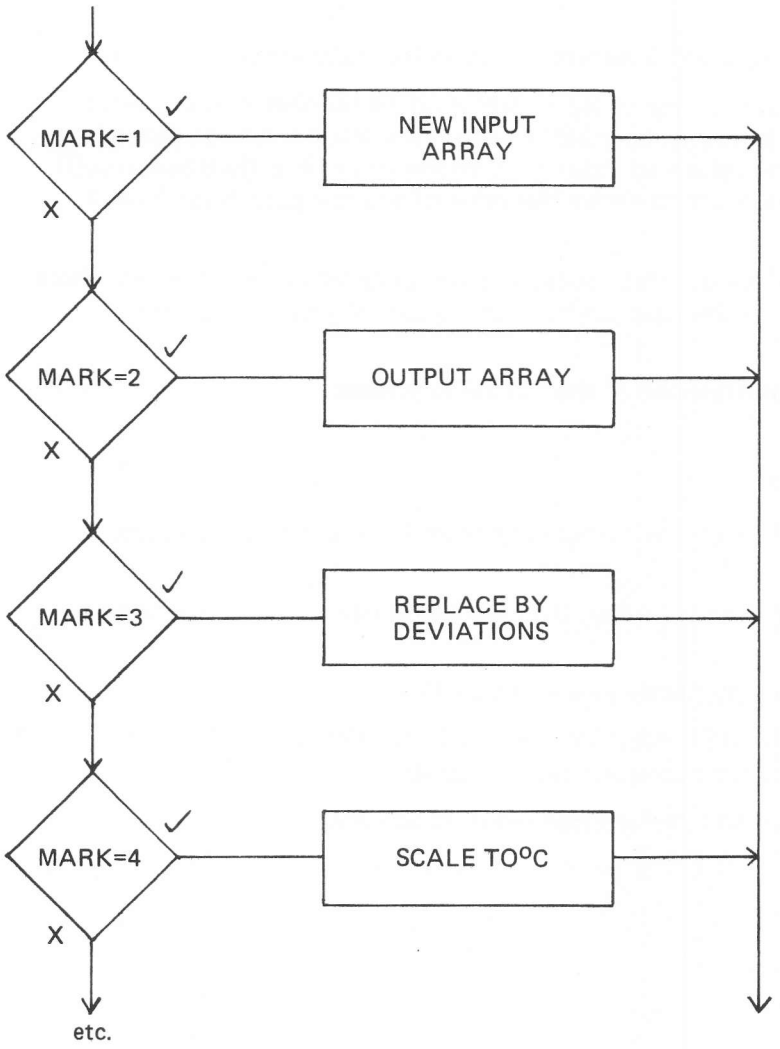
   5: Smooth the contents of the input array (use example 4).

   (The procedures PROC ARRAYIN (REF ARRAY REAL Q) and PROC ARRAYOUT (REF ARRAY REAL Q) perform the necessary input/output).

   Write a procedure to read the tape and perform the required actions.

20. Transfer of control: switch statement

The last example of section 19 used an integer read from a tape to select a particular processing strategy. This is another fairly common requirement in programming, to deal with situations where there are several different paths or strategies at a given point, one of which must be chosen on the basis of some parameter. One way of doing this was illustrated in the solution: a complex if-statement simply churned through all the possible values until it found a match:



If the number of possibilities becomes large, the number of tests involved is also large, and this method becomes inefficient both in terms of the space occupied by the instructions, and the time taken to execute them. The fundamental property of such a situation is the need to transfer control to the correct set of instructions given a simple index.

RTL/2 provides such a construction in the form of a *switch-statement*. It is a transfer-of-control statement which causes the flow of control to pass to one of a set of labels, the decision being based on an integer index value. The syntax consists of the keyword SWITCH followed by an integer expression (as usual as complex as you like provided it delivers an integer value) which is used as the index; the keyword OF separates this from a list of labels (separated by commas) which are the "destinations" and may be regarded as being numbered from 1 to n where n is the total number of labels in the list.

Examples:

```
SWITCH I OF L,M,L,M,EXIT,M,M,L;

SWITCH MARK OF INAR,OUTAR,DEV,SCALE,SM;
```

There is no reason why a label should not appear more than once in the list; the restriction on the labels in the list is the usual one for all names – each label must be in scope at the point of the statement.

The required action of the statement is obviously to branch to the i th label of the list where i is the value of the integer expression. This is all very well provided i belongs to the set 1, 2, 3, ..., n-1, n. We must define what will happen when the integer expression yields a value out of this range. In such a case ($i \leq 0$ or $i > n$) there is no action, in the sense that the next statement to be performed is the one following the switch-statement; that is control 'drops through' the switch and there is no transfer. Thus, unlike the goto-statement, we can arrive at an unlabelled statement immediately following the switch, and unlike arrays an out-of-range index is valid. In our example, when we had exhausted the tests MARK=1, MARK=2, we signified an error condition; this condition will now follow the switch. Our example now becomes:

```
LET LA=100;    % LENGTH OF MESSAGE ARRAYS %

DATA MESSAGES;
   ARRAY (LA) REAL INPUT,OUTPUT;
ENDDATA;

PROC PROCESSTAPE ();
   % READ NUMBER OF INTEGERS ON TAPE FOR LOOP CONTROL %
   TO IREAD() DO
   INT MARK:=IREAD();
      SWITCH MARK OF INAR,OUTAR,DEV,SCALE,SM;
      % SUITABLE ACTION FOR UNDEFINED INTEGER VALUE %

INAR:     ARRAYIN(INPUT);    GOTO NEXT;

OUTAR:    ARRAYOUT(OUTPUT);    GOTO NEXT;

DEV:      MEAN(OUTPUT);          GOTO NEXT;

SCALE:    FOR I:=1 TO LA DO
             OUTPUT(I):=( INPUT(I) - 32.0 ) / 1.8;
          REP;
          GOTO NEXT;

SM:       SMOOTH(INPUT);

NEXT:
   REP;
ENDPROC;
```

To prevent the various actions from running into one another we have to program the continuation at a common point explicitly. This is done by appending a goto-statement at the end of each sequence and labelling a common point; in this case the common point is the REP causing the next iteration of the loop to be performed. Note that the last sequence, at SM, does not need a GOTO NEXT since control will proceed to that point automatically.

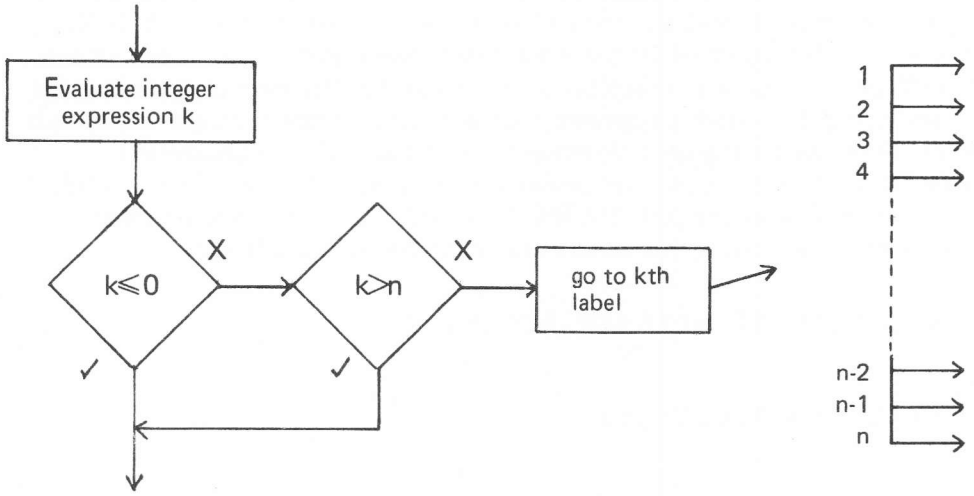
If we have no need of the value of MARK in the various sequences, we could simply write

```
TO IREAD() DO
   SWITCH IREAD() OF INAR,OUTAR,...ETC
```

IREAD delivers an integer which will be used as index. In the case when this value is out of range, we will drop into the error sequence. Note, however, that we cannot regard the action of

the switch-statement as null; the call of IREAD has read the next integer; this is a typical example of the side effect of a procedure call.

Diagrammatically we can present the statement as:



The advantage of the switch-statement clearly lies in the reduction of the number of tests required and, perhaps, a slight increase in clarity for the reader. The disadvantages are the need to program explicitly continuation to a common point from the various possibilities (sometimes, of course, this is not required anyway) and the fact that valid indices must form a consecutive range of values. This latter problem can be solved by using a suitable expression to return a series of consecutive integers or by a look-up table; any 'holes' in a sequence can be filled by labelling the default path and using that label.

```
    SWITCH I OF L1,L2,FAIL,L4,L5,FAIL,L7,L8;
       % I=3,6 SHOULD NOT ARISE %

FAIL:     ERRORACTION();
          % FAIL IS REACHED FOR I<=0,I>8,I=3,6 %


    SWITCH N+3 OF P1,P2,P3,P4,P5;
       % VALID FOR N IN THE RANGE -2 TO +2 %
```

This is the tenth kind of RTL/2 statement (including the dummy one) that we have described, and in fact we have now covered all but one of the RTL/2 statement types (the last one occurs in section 31). It seems appropriate to point out that the range of statements can give considerable flexibility in the method of coding a particular program, some ways being more efficient than others; which is more efficient in a given case will depend on the situation and the criteria for "efficient". We re-iterate the comment made in the introduction that our examples (and answers) usually show only one of the possible methods that could be used for a particular solution.

Section 20 examples

1. Write a procedure which, given two real parameters will return the sum, product, difference, quotient, average or the greater or lesser of them depending on the value of a third (integer) parameter.

21. Worked example

PROBLEM: A man borrows £200 to be paid back over 2 years by monthly instalments. Interest is charged on the remaining debt at R% per annum, compounded monthly. Calculate R for the following monthly instalments:

£8.50, £8.70, £8.90, £9.10, £9.30, £9.50.

Write the program in such a way that the following problems can also be solved:

i) Calculate the value after 5 years of an initial investment of £100 with monthly deposits of £10 at an interest rate of 5% per annum compounded monthly.

ii) Calculate the monthly repayments required to repay a mortgage of £4000 over 25 years, the rate of interest being 8½% per annum charged once a year.

iii) Calculate the initial investment required in a deposit account if £10 is to be drawn monthly for 20 years (when the account is exhausted) interest being at 6% per annum compounded monthly.

DISCUSSION: This problem is basically concerned with compound interest. If a principal value (P) is invested at time zero for a number of years (N) with additional deposits (Y) made at the same rate (T times per year) as interest is compounded (the interest rate being R% per annum) then the five quantities P, N, Y, T, R and the final value F are related by the formula:

$$F = (P + \frac{Y.100.T}{R})(1 + \frac{R}{100T})^{NT} - \frac{Y.100.T.}{R}$$

The case of repayment of a loan is covered by the same formula; P in this case is the amount borrowed, Y the periodic repayment (negative to indicate this) and F will be zero.

All the parts of the problem require us to find one of the quantities given the other five. A cursory glance shows that the formula will look much simpler if we put a new variable r equal to R/100T):

$$F = (P + Y/r)(1 + r)^{NT} - Y/r$$

This formula gives us F in terms of the other five; rearranging, we can obtain expressions for P, Y in terms of the remaining five variables.

$$P = (F + Y/r)(1 + r)^{-NT} - Y/r$$

$$Y = r\{F - P(1 + r)^{NT}\} / \{(1 + r)^{NT} - 1\}$$

When it comes to calculating the value of r (and hence R), we cannot make r the subject of the formula, nor is there a simple algebraic expression for a solution. However, we know that a practical solution (if one exists) will satisfy $0 \leqslant R \leqslant 100$ and we can approximate a solution by using the method of repeated bisection of [0, 100] to yield a range (as small as we please) that contains the required solution. Any range containing a solution will have the property that the signs of the expression

$$(P + Y/r)(1 + r)^{NT} - Y/r - F$$

evaluated at the end points of the range will be different. If the signs are not different for the initial range, we know that our problem has no solution.

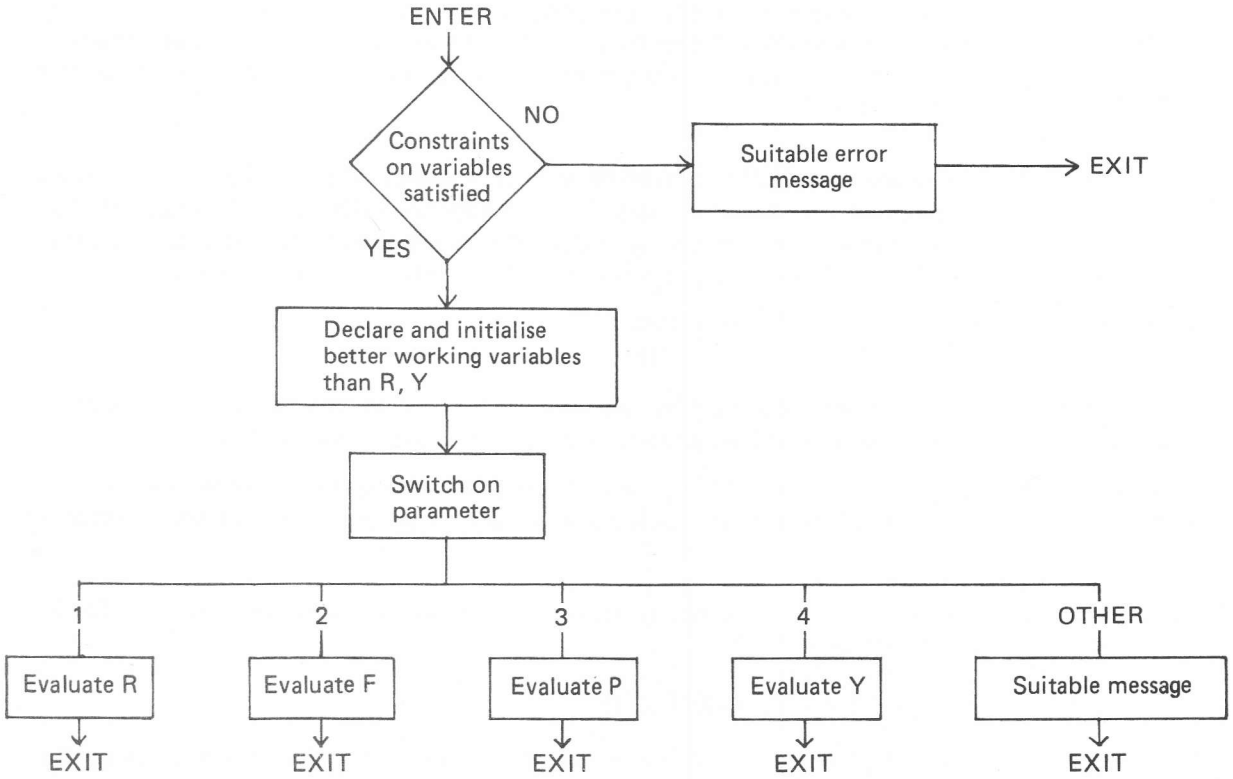
Mathematically, we now know how to solve our problem. Next we must plan our program. To make our solution flexible and cope with all the problems presented, we choose to have a procedure to calculate the missing piece of information, signalling which item is required by an integer parameter. We could pass the other variables as parameters and return the unknown quantity as a result; instead we choose to have F, P, Y, R, N, and T as global variables which we initialise before a call of our procedure, and use our procedure to "fill in" the missing value. Using this approach makes it simpler to communicate values to other procedures.

Our procedure COMPOUND will also test that the various quantities supplied are sensible; the unknown quantity is given a default value of 1.0 so that it will pass these tests – this means that we can make all the checks in one place before we decide (in COMPOUND) which is the unknown. From practical considerations, the sensible ranges of the variables are as follows; we can also decide on their mode at this stage:

F	Can be positive or negative – no constraints	REAL
P	$P \geqslant 0$	REAL
Y	Again positive or negative – no constraints	REAL
R	$0 < R < 100$ (not allowing end points is arbitrary)	REAL

N	N>0	We choose a whole number of years	INTEGER
T	1≤T≤365		INTEGER

We can now draw a flowchart for COMPOUND; in doing so we see that we will need procedures to calculate a power of a real number, and that it will be efficient (in the bisection calculation) to provide a procedure to return the sign of our expression — this is where the communication of the problem's parameters through data variables is valuable — the values are available without passing them all as parameters.



Finally we write the RTL/2, expanding the flowchart and discussion above into a program, annotated with comments to enable the reader to follow it. Incidentally, as it stands, the program contains examples of every RTL/2 statement that we have discussed.

```
% COMPOUND INTEREST PROBLEM %

DATA GLOBAL;
   REAL  P,     % PRINCIPAL %
         F,     % FINAL VALUE %
         Y,     % INSTALMENT : NEGATIVE IF REPAYMENT %
         R;     % RATE (PER CENT) PER ANNUM %
   INT   N,     % NUMBER OF YEARS %
         T;     % NUMBER OF TIMES PAID AND COMPOUNDED PER ANNUM %
ENDDATA;

PROC MAIN ();
   % RATES CALCULATED %
   P:=200.0;  F:=0.0;  N:=2;  T:=12;
   R:=1.0;  % DEFAULT %
   FOR I:=850 BY 20 TO 950 DO
      Y:=-I/100.0;
      COMPOUND(1);
      PRINT(R);
      % WE ASSUME SOME OUTPUT FUNCTION PRINT TO SUPPLY THE RESULTS %
   REP;
```

```
    % PROBLEM 1 TO FIND THE FINAL VALUE OF INVESTMENT %
    P:=100.0;  Y:=10.0;  R:=5.0;  N:=5;  T:=12;
          % NOT STRICTLY NECESSARY TO RESET T %
    F:=1.0;  % DEFAULT %
    COMPOUND(2);
    PRINT(F);

    % PROBLEM 2 TO FIND MONTHLY REPAYMENT %
    P:=4000.0;  F:=0.0;  R:=8.50;  N:=25;  T:=1;
    Y:=1.0;  % DEFAULT %
    COMPOUND(4);
    % SETS ANNUAL REPAYMENT AS A NEGATIVE QUANTITY %
    PRINT(-Y/12.0);

    % PROBLEM 3 TO FIND INITIAL DEPOSIT %
    F:=0.0;  R:=6.0;  Y:=-10.0;  N:=20;  T:=12;
    P:=1.0;  % DEFAULT %
    COMPOUND(3);
    PRINT(P);
ENDPROC;

PROC POWER (REAL X, INT M) REAL;
    % POSITIVE POWER ONLY : RETURNS X TO THE MTH %
    RETURN (IF M=0 THEN 1.0 ELSEIF M=1 THEN X
              ELSE POWER(X*X,M:/2)*POWER(X,M MOD 2) END);
ENDPROC;

PROC SIGN (REAL R) INT;
    % RETURNS -1,0,+1 ACCORDING TO SIGN OF THE EXPRESSION %
REAL  NEWR:=R/T*0.01,
      NEWY:=Y/NEWR,
      Q:=POWER(NEWR+1.0,N*T)*(P+NEWY) - F - NEWY;
    RETURN(IF Q<0.0 THEN -1 ELSEIF Q>0.0 THEN +1 ELSE 0 END);
ENDPROC;

PROC COMPOUND (INT MARK);
    % MARK=1 FINDS RATE %
    % MARK=2 FINDS FINAL VALUE %
    % MARK=3 FINDS PRINCIPAL %
    % MARK=4 FINDS MONTHLY INSTALMENT %

% CHECK RANGES OF PARAMETERS; DEFAULT OF 1.0 ALWAYS WORKS %
% SEE SECTION 26 FOR A NEATER METHOD %
    IF T<1 THEN GOTO FAIL;  END;
    IF T>365 THEN GOTO FAIL;  END;
    IF P<0.0 THEN GOTO FAIL;  END;
    IF R<=0.0 THEN GOTO FAIL;  END;
    IF R>=100.0 THEN
FAIL:    % SUITABLE ERROR MESSAGE %
         % UNKNOWN STILL CONTAINS DEFAULT VALUE %
         RETURN;
    END;
```

```
BLOCK
   % NOW SAFELY CALCULATE BETTER VARIABLES %
   REAL  NEWR:=R/T*0.01,      % PERIODIC FRACTIONAL RATE %
         NEWY:=Y/NEWR,
         FACT:=POWER(NEWR+1.0,N*T);     % RATE FACTOR %

   SWITCH MARK OF RATE,FINAL,PRIN,MONTH;
      % OUTPUT SUITABLE MESSAGE FOR ILLEGAL PARAMETER IN CALL %
      RETURN;

RATE:
   BLOCK
   REAL BOT:=0.1,TOP:=99.9;  % INITIAL SEARCH INTERVAL %
   INT  ST:=SIGN(99.9);      % INITIAL SIGN FOR TOP OF RANGE %
      IF ST=SIGN(0.1) THEN
         % NO SOLUTION IN RANGE 0.1 TO 99.9 : SUITABLE MESSAGE %
         RETURN;
      END;
      WHILE TOP-BOT>0.01 DO
         BLOCK
         REAL MID:=(TOP+BOT)*0.5;       % BISECT RANGE %
         INT  SM:=SIGN(MID);            % SIGN OF MID-POINT %
            IF SM=0 THEN
               % EXACT SOLUTION %
               R:=MID;
               RETURN;
            END;
            % DECIDE NEW RANGE %
            IF SM#ST THEN
               % SOLUTION IN TOP HALF OF RANGE %
               BOT:=MID;
            ELSE
               % SOLUTION IN BOTTOM HALF OF RANGE %
               TOP:=MID;
            END;
         ENDBLOCK;
      REP;
      R:=(TOP+BOT)*0.5; % TOP AND BOT DIFFER BY LESS THAN 0.01 %
                        % CHOOSE MID-POINT AS SOLUTION %
      RETURN;
   ENDBLOCK;

FINAL:
   F:=(P+NEWY)*FACT - NEWY;
   RETURN;

PRIN:
   P:=(F+NEWY)/FACT - NEWY;
   RETURN;

MONTH:
   Y:=(F-P*FACT)/(FACT-1.0)*NEWR;

ENDBLOCK;
ENDPROC;
```

22. Binary

Up to this point, all numbers in this manual have been written in decimal notation, and in describing the action of operators we have been working implicitly in decimal. Computers store numbers, not in decimal notation, but as *binary patterns*. By means of operators to be described later, such binary representations can be used for the compact storage and manipulation of data.

Binary numbers use just two digits, 0 and 1; the use of binary digits is more natural within a computer since the digits 0, 1 can be used to represent a two state electronic component: i.e. either 'off' or 'on'. The binary pattern representing a number is simply a shorthand for a decomposition of the number into a sum of powers of 2; thus

$$1100 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$
(= 12 in decimal)

$$10.11 = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$
(= 2.75 in decimal)

The binary point fixes the position of the digit representing the multiple of 2^0 (= 1).

We can devise rules for converting integers and fractions from one number system to the other:

1. To convert a decimal integer to binary we divide repeatedly by 2; at each division, we write down the remainder, starting from the right; the sequence generated is the binary number.
   e.g. 279

Divide by 2	gives	139	remainder		1
"	"	69	"		1
"	"	34	"		0
"	"	17	"		1
"	"	8	"		0
"	"	4	"		0
"	"	2	"		0
"	"	1	"		1
"	"	0	"		

   And 100010111 is the required binary pattern.

2. To convert a decimal fraction to binary we use an iterative technique:
   a) multiply the fraction by 2.
   b) the digit to the left of the decimal point is the next binary digit in the binary fraction.
   c) the fraction part of the product is the new fraction.
   d) return to a).

   The process is terminated as soon as sufficient digits have been generated; in general, the fraction will not be exact in binary.
   e.g. 0.72

Multiply by 2	1.44	giving digit	1	
Multiply fraction by 2	0.88	"	"	0
"	1.76	"	"	1
"	1.52	"	"	1
"	1.04	"	"	1
"	0.08	"	"	0
"	0.16	"	"	0
"	0.32	"	"	0
"	0.64	"	"	0
"	1.28	"	"	1

   To ten binary places the fraction is .1011100001.

3. To convert a binary integer to decimal start from the left-most digit; multiply it by two and add in the next digit; continue multiplying by two and adding in the next digit until the

digits are exhausted.

e.g. .11010101

1	multiply by 2 gives		2 + next digit 1 gives			3
3	"	"	6	"	0 "	6
6	"	"	12	"	1 "	13
13	"	"	26	"	0 "	26
26	"	"	52	"	1 "	53
53	"	"	106	"	0 "	106
106	"	"	212	"	1 "	213

And 213 is the required decimal number.

4. To convert a binary fraction to decimal, we start from the right, divide by 2 and add in the next digit in a similar way; the last digit added in (the left most one and hence the leading digit of the fraction) must also be divided.
   e.g. .010111011

digit 1	divide by 2 gives	0.5	add digit	1 gives	1.5
1.5	"	" 0.75	"	0 "	0.75
0.75	"	" 0.375	"	1 "	1.375
1.375	"	" 0.6875	"	1 "	1.6875
1.6875	"	" 0.84375	"	1 "	1.84375
1.84375	"	" 0.921875	"	0 "	0.921875
0.921875	"	" 0.4609375	"	1 "	1.4609375
1.4609375	"	" 0.7304875	"	1 "	1.73046875
0.73046875	"	" 0.365234375			

And 0.365234375 is the required fraction.

In the computer, integer binary patterns may be thought of as being held in precisely the form used above. Each binary digit is termed a *bit* which may thus be 0 or 1. Starting from the right the bits represent the multiples of 2^0, 2^1, 2^2, etc. and the binary point is conceptually at the right hand end of the *word* (the collection of bits).

...	2^4	2^3	2^2	2^1	2^0	
	0	0	1	0	1	. (representing 5)

The top bit (or left-most bit) is special; it behaves as a multiple of -2^m (where m + 1 is the number of bits in the word). Hence if our word has 6 bits:

-2^5	2^4	2^3	2^2	2^1	2^0	
1	0	0	0	0	0	(representing −32)
1	0	1	0	1	1	(representing −32 + 8 + 2 + 1 = −21)
0	1	0	1	0	1	(representing +16 + 4 + 1 = +21)

This method of holding negative and positive numbers is called *2's complement* form. RTL/2 states explicitly that integers will be held in this form.

For our 6 bit word, the smallest integer is clearly

1	0	0	0	0	0	(representing −32+0=−32)

and the largest integer is:

0	1	1	1	1	1	(representing −32x0+16+8+4+2+1=31)

Thus the *range* of integers is −32 to +31.

In general, if our computer has (m + 1) bits in its word, the integer range is -2^m to 2^m-1. Now convince yourself that whatever the value of m, the integer −1 will always be held as a pattern consisting of ones in every bit.

We can now understand the choice of our ranges $[-\alpha, \alpha)$, $[-\alpha^2, \alpha^2)$ for the discussion of integers and big integers. The big integer effectively occupies two words and our ranges are $[-2^m, 2^m)$, $[-2^{2m}, 2^{2m})$; it might be argued that we now have 2m+2 bits and hence our range should be $[-2^{2m+1}, 2^{2m+1})$: however because of the common machine treatment of double length quantities and their sign bits, we have the more restricted range.

If, in our 6-bit example "machine" we perform

```
INT I:= 17*5;
```

we will generate two integer patterns

0	1	0	0	0	1

0	0	0	1	0	1

multiply them together to give a big integer occupying an 11 bit word

0	0	0	0	1	0	1	0	1	0	1

We must then narrow our big integer to an integer for assignment to I. Clearly this is impossible, as 85 does not belong to the range [−32, 31]. Overflow has occurred: our binary pattern will not fit into the available space. On addition, similarly we can generate (by performing the addition theoretically) patterns that will not fit in and the overflow condition arises.

The number of bits in a word (the *wordlength*) varies from computer to computer: hence the integer range $[-2^m, 2^m-1]$ also varies and this explains why the range of integer values is said to be machine dependent. RTL/2 specifies a minimum of a 16-bit word, and hence a minimum range of integer values of $[-2^{15}, 2^{15}-1]$ or [−32768, 32767]. When designing machine-independent (transportable) programs, and particularly when using logical representations (see section 27) the wordlength must be borne in mind, in particular (at this stage) the consideration of overflow.

RTL/2 allows integer constants to be written in binary form – this being of particular use when binary patterns are required for logical manipulation. This new form of the integer constant consists of the keyword BIN (for binary) and a 'number' formed of binary digits 0, 1. A problem arises here; BIN is a keyword and must be terminated, neither 0 nor 1 would terminate it, they would create names BIN0... or BIN1... . Our integer constant is an item (refresh your memory with section 3 if necessary!) but now we allow layout characters to separate the keyword BIN from its digit sequence; indeed not only allow but demand that at least one layout character (space, tab or newline) occurs. After any layout characters the item is terminated as soon as a non-binary digit is encountered.

```
BIN 1011
BIN     010
BIN  11011010
```

We can think of such integer constants either as numbers represented in their binary form, or as simple patterns.

A list of binary digits can become quite long, and is difficult to read and somewhat error prone.

If we take a binary integer, and, starting from the right, partition the bits into groups of three and replace each group by its decimal equivalent we obtain an *octal* representation.

Thus 1011101101011 partitioned is 1/011/101/101/011 is octal 13553
11011110 partitioned is 11/011/110 is octal 336

The maximum value of a group is 111 in binary, i.e.7.

Thus the octal representation of a number is simply its representation using a base of 8: it bears a direct relationship to the binary form since 8 is an integral power of 2.

Similarly we can partition the bits into groups of four. In this case the maximum value of the groups is 1111 in binary which is 15 in decimal; we now represent the decimal equivalents 10, 11, 12, 13, 14, 15 by the letters A, B, C, D, E, F and the resulting representation (expressing the number to a base of 16) is called a *hexadecimal* representation.

Thus 1011101101011 partitioned is 1/0111/0110/1011 is hexadecimal 176B
11011110 partitioned is 1101/1110 is hexadecimal DE.

The conversions from octal or hexadecimal to decimal follow from our earlier methods by replacing 2 by 8 or 16, and conversion to binary is achieved simply by rewriting each group in binary.

hexadecimal 71B is binary 0111/0001/1101
is decimal $7 \times 16^2 + 1 \times 16^1 + 11 \times 16^0 = 1819$
octal 5034 is binary 101/000/011/100
is decimal $5 \times 8^3 + 0 \times 8^2 + 3 \times 8^1 + 4 \times 8^0 = 2588$

Octal and hexadecimal forms of integer constants are allowed in RTL/2; their syntax is similar to the binary form with the keywords OCT and HEX being used. The allowed 'digits' in the number are 0, 1, ...,7 and 0, 1, ..., 9, A, B, ..., F respectively.

Examples:

```
HEX 71B
OCT 5034
HEX       176B
OCT             13553
```

We cannot say how real numbers will be held in a particular implementation, except that some binary form will probably be used. We do not stipulate a particular form or minimum range in RTL/2 and hence no other forms for the real constant are available. It is worth pointing out at this stage that a consideration of the number of binary digits used to store information about a real is needed to decide the degree of accuracy to which a real can be held. The reader is referred to the documents on individual implementations for this information. In general, a real number will occupy more space than an integer.

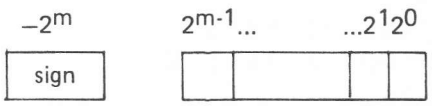
Fractions and conversions between decimal and binary have been mentioned in this section. The manipulation of fractions in RTL/2 is covered in section 23.

Section 22 examples

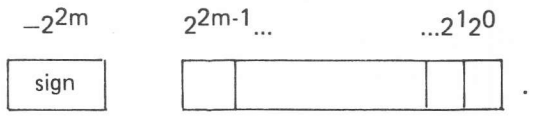
1. Rewrite the following decimal integer constants in binary, octal and hexadecimal forms.
   6
   27
   84
   317
   2120
   32677

23. Fractions

In section 22 we discussed how an integer is held within the computer as a binary pattern with a conceptual binary point at the right hand end of the word; we can picture this as a box:



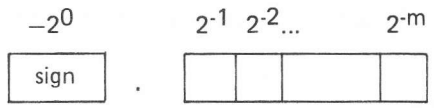
The range is a function of the word-length of the machine; if the word has m+1 bits then the integer range is $-2^m \leqslant \text{integer} < 2^m$ and the accuracy that can be obtained is clearly integral, that is any integer is a multiple of 1. A big integer is a similar form, having a larger range:



range: $-2^{2m} \leqslant$ big integer $< 2^{2m}$
accuracy: 1

We also saw in the last section how conversions between decimal and binary fractions can be achieved. RTL/2 possesses an arithmetic mode to cope with fractions in a fixed point way. This mode is the subject of this section.

In elementary terms, a fraction value is something less than one in magnitude. In a computer a fraction consists of a binary pattern with a conceptual binary point at the left hand end of the word; in this case the sign bit now represents multiples of -2^0 that is of -1.



If we have m + 1 bits in our word again, the range that can be held is clearly $-1 \leqslant$ fraction < 1 and the accuracy is now in terms of multiples of 2^{-m} (that is $1/2^m$). Any implementation of RTL/2 will use the same word-length for fractions as for integers. The range is independent of m though, and this is one of the advantages of fractions; if we change machines, there is no change of range (as there may be for integers and reals); the penalty may be a change of accuracy. Many machines do not possess hardware for performing floating-point arithmetic and such operations can be lengthy (in space and time) and inefficient. Fractions provide a means of writing fixed point arithmetic in a machine independent way.

Fractions are normally fractions of something. For instance we may have an instrument which reads a voltage between 0 and 24 volts. For a reading of 18 volts, we can either regard this in absolute terms or as a reading 3/4 of full-scale. With the second viewpoint, we can record readings as fractions, and keep the full-scale value elsewhere as a *scaling factor*. In working with fractions the aim will be to maintain scale factors at appropriate points in order that the fraction range -1 to 1 is maintained and to ensure no loss of accuracy – this will entail keeping as much information as possible in the most significant bits of the word. Note that any tuning performed on a program to ensure that scaling achieves these objectives is stable in the sense that the same tuning is applicable to all machines because of the independence of the range. Before exploring this further, we will investigate the syntax of fractions in RTL/2.

Fraction is a mode (like integer or real) in which the objects manipulated are fraction values in the range $[-1, 1)$:

1. The form of a fraction constant is a real constant together with a *scaling factor.* This factor consists of the letter B followed by a (possibly signed) integer and specifies a binary scale factor by which the value is to be multiplied (compare the decimal exponent in the real constant). Thus the number 10.2B–4 will be held as the fraction 10.2×2^{-4} or 10.2/16. Note that as usual no layout characters may occur in the sequence (these would terminate the item). For a fraction constant to be valid, the value obtained from the combination of the real constant and binary scale must lie in the range $[-1, 1)$.

```
% EXAMPLES OF VALID FRACTION CONSTANTS : %
   0.2B2      0.7E0B+0      13E2B-11
% THE FOLLOWING EXAMPLES ARE INVALID : %
   0.4  B-2          % CONTAINS LAYOUT CHARACTER %
   7B-4              % INITIAL CONSTANT NOT A REAL %
   4.93E-1B2         % VALUE OUT OF RANGE %
```

2. Variables, reference variables and arrays of variables to contain fraction values or the names of fraction variables may be declared as for the integer and real modes. The mode fraction is specified by the keyword FRAC.

   Example:

```
DATA GLOBAL;
   FRAC F:=F1:=0.2B0, G, H;
   REF FRAC RF:=F1;
   ARRAY (7) FRAC AF:=(0.1B0,0.2B0,0.0B0(5));
   REF ARRAY FRAC RAF:=AF;
ENDDATA;

PROC ACTION (REF FRAC X) FRAC;
FRAC LF:=0.0B0;
      .
      .
```

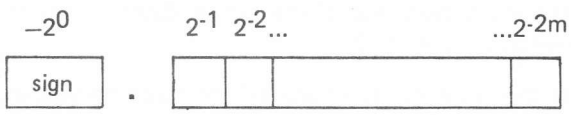
   The rules for declarations, initialisations, use as parameters and as result mode, use in conditional expressions, dereferencing and the use of VAL all follow in a similar way to those for integer and real variables.

3. The usual monadic operators are available for fractions:

OPERATOR	OPERAND	RESULT	INTERPRETATION
+	Fraction	Fraction	Identity: no action
–	Fraction	Fraction	Negate the operand
ABS	Fraction	Fraction	Negate the operand if it is negative, otherwise no change.

4. When we come to consider dyadic operators between fractions, multiplication and division again present problems. Two numbers in the range [–1, 1) when multiplied together will give another fraction in the range (except for the case of –1 times –1); however if we multiply two values of order $1/2^m$ together, the result will be of order $1/2^{2m}$, and we need twice as many bits to retain accuracy in the calculation.

   We again have an intermediate mode to cater for this situation and to mirror the double-length operations available on most machines. In this case our double length quantity will again have 2m + 2 bits of which 2m will be available to hold the value, but now the binary point is at the left hand end (note that the unused bit may occur at either end of the word or indeed in the middle, but this will not concern the user):



   Range: [–1, 1)
   Accuracy: multiples of $1/2^{2m}$
   Because the accuracy is greater this mode is termed a *fine fraction.*

   Division, again regarded as the reverse of multiplication, has a fine fraction as its first operand, a normal fraction as its second and, naturally, produces a result of mode fraction. The compound separator / / is used as the symbol for fraction division.

We can now draw up our table showing the dyadic operators for fractions:

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
+	1	Fraction	Fraction	Fraction	Form the sum of the operands
–	1	Fraction	Fraction	Fraction	Subtract the second operand from the first
*	5	Fraction	Fraction	Fine Fraction	Form the product of the operands
/ /	5	Fine Fraction	Fraction	Fraction	Quotient on dividing first operand by second

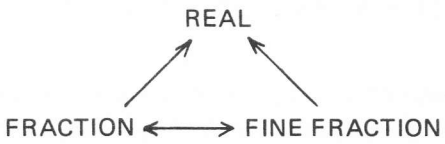
Notes: i) As usual, overflow is possible in all four cases.
For example:
0.5B0 + 0.5B0
–0.6B0 – 0.6B0
–1.0B0 *– 1.0B0
0.1B0* 0.1B0 / / 0.001B0

ii) Division is like integer division in terms of its accuracy and the possible existence of a remainder.

5. The comparators = #, <, >, <=, >= may also be used to form conditions involving fractions in the normal way.
6. The rules governing mode transfers are the same as those we encountered when considering the combination of integers and reals:

i) Mode transfers between normal and intermediate modes are automatic.

ii) Mode transfers in which there is no loss of information are performed automatically.
Thus the diagram for automatic transfers is:



iii) Mode transfers in which information is lost must be programmed explicitly.

In the case of a transfer from real to fraction, the keyword FRAC is used as a monadic operator. As in the case of INT, this is also defined for completeness for a fraction operand.

OPERATOR	OPERAND	RESULT	INTERPRETATION
FRAC	Fraction	Fraction	Identity: no action
	Real	Fraction	Rounds real to fraction

This operation can clearly give rise to an overflow condition. Rounding is to the nearest fraction value; if the original real value lies midway between two fraction values, then the algebraically greater fraction is the result.

Hence:

```
FRAC F;
REAL R;
R:=0.7;              % FAMILIAR %
R:=0.1B0;            % RIGHT HAND SIDE DELIVERS A FRACTION %
                     % AUTOMATICALLY WIDENED TO 0.1 %
R:=0.4B0*0.1B0;      % RIGHT HAND SIDE DELIVERS A FINE FRACTION %
                     % AUTOMATICALLY WIDENED TO 0.04 %
                     % NOTE DIRECT; NO INTERMEDIATE NARROWING %
                     % TO FRACTION OCCURS %

F:=0.1B0;            % FAMILIAR %
F:=0.4B0*0.1B0;      % RIGHT HAND SIDE DELIVERS FINE FRACTION %
                     % NARROWED AUTOMATICALLY TO NORMAL FORM %
F:=FRAC 0.7;         % RIGHT HAND SIDE INITIALLY DELIVERS A REAL %
                     % THIS IS EXPLICITLY NARROWED %
```

The combination of integer and fraction values will be discussed in a later section.

As in the integer/real case, the same rules apply when considering the evaluation of an expression involving both reals and fractions. The presentation of ordered operator tables is postponed until section 25.

Examples:

```
0.6B0*0.1B0//0.2B0          % PERFORMED DIRECTLY %
0.06B0//0.3B0               % WIDEN TO FINE FRACTION %
0.6B0/0.3B0                 % WIDEN TO REALS %
0.6/(0.3B0*0.2B0)           % WIDEN FINE FRACTION TO REAL %
0.6B0*0.1B0//0.2B0*0.4B0    % FINE FRACTION NARROWED TO FRACTION %
```

The problem of using an appropriate binary scale, and the use of scaling to retain accuracy will be discussed in subsequent sections. The following example uses fractions in a simple way, where the scaling involved is of a simple 0 to full-scale reading form:

In an exothermic process, temperatures are read as positive fractions representing actual values between 0 and 80°C (full-scale reading). The temperature (EXITTEMP) of the effluent fluid is kept near a desired value (TARGET) by setting the temperature of the incoming fluid according to a feed-forward control equation of the form:

$$\text{SETPOINT} = A/L^2 + B/L + C$$

Where A, B, C are real constants and L is the plant load. In practice the setpoint is also required as a fraction. To this setpoint is added a trimming term given by a feed-back proportional control equation:

$$\text{NEWTRIM} = \text{OLDTRIM} + K\,(\text{TARGET} - \text{EXITTEMP})$$

where K is a fraction derived from some expression which caters for the change in residence time of the fluid with plant load, plant characteristics, frequency of updating etc. The absolute values of the trim term and the setpoint are to be restricted as follows:

$$-5^{\circ}C < \text{TRIM} < +5^{\circ}C$$
$$10^{\circ}C < \text{SETPOINT} < 70^{\circ}C$$

Our procedure calculates the new values of the trim term and setpoint and imposes these constraints; variables are assumed to be global. The full scale value is presented in a LET statement, this value being used to scale the setpoint (without trim term) to a fraction.

```
LET TEMPSCALE=80.0;

DATA CONTROL;
   FRAC  TARGET,
         EXITTEMP,    % BOTH AS FRACTIONS; SCALE IS 0 TO TEMPSCALE %
         TRIM,
         K,           % PROPORTIONAL CONSTANT %
         SETPOINT;
   REAL  A,B,C,       % FEED FORWARD CONSTANTS %
         LOAD;
ENDDATA;

PROC TEMPCONTROL ();
FRAC SP;
   TRIM:=TRIM + K*(TARGET-EXITTEMP);
   IF TRIM<-0.0625B0 THEN TRIM:=-0.0625B0;  END;
   IF TRIM>+0.0625B0 THEN TRIM:=+0.0625B0;  END;
      % APPLY ABSOLUTE LIMITS OF -5 TO +5 DEG.C %
      % IN TERMS OF TRIM THIS IS -1/16 TO +1/16 OF FULL-SCALE %
   SP:=FRAC( ( (A/LOAD+B)/LOAD + C) / TEMPSCALE)  +  TRIM;
   IF SP<0.125B0 THEN SP:=0.125B0;  END;
   IF SP>0.875B0 THEN SP:=0.0875B0;  END;
      % APPLY ABSOLUTE LIMITS OF 10.0 TO 70.0 ON INPUT SETPOINT %
      % IN TERMS OF SP THIS IS 1/8 TO 7/8 OF FULL-SCALE %
   SETPOINT:=SP;    % UPDATE SETPOINT %
ENDPROC;
```

Section 23 examples

1. Which of the following fraction constants are illegal, and why?
   a) 0.2B0
   b) 2B–3
   c) 0.7
   d) 75E–2B0
   e) 27.B–7
   f) 0.01E + 1B–0
   g) 38.7B–5
   h) 0.0625B3
   i) 2.2B + –2
   j) 38.1B–7E–1
2. Write procedures to perform the conversion between absolute values and fraction values for an instrument reading in terms of the full-scale reading.

24. Arithmetic shifts

We are all familiar in elementary arithmetic with moving the decimal point to the right or left to effect multiplication or division by a power of ten:

```
34.56
3.456
3456.
```

We could regard this operation as one in which we keep the position of the point fixed and move the digits:

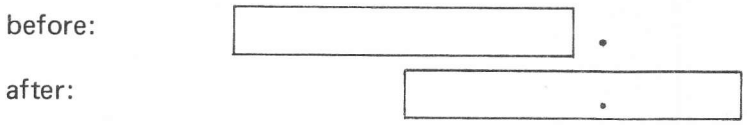
```
  34.56
   3.456
3456.
```

Naturally we can perform the same operations with binary numbers, though in this case, the effect is multiplication or division by a power of two:

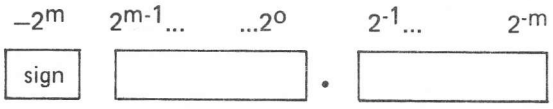
```
  110.101
    .110101
110101.
```

This operation is known as *shifting* and can be performed in RTL/2 by the use of dyadic operators, the second operand specifying the number of places the first operand is to be shifted. In a computer, however, there is the problem that the word-length is finite, and therefore we must define very carefully the actions to be taken. This will involve extensive considerations of intermediate modes.

A right shift is specified by the keyword SRA (Shift Right Arithmetic); this has the effect of a division by a power of two (supplied in the second operand). If we shift an integer to the right, we obtain significant digits (binary digits hence bits) to the right of the point



The part to the right is a fraction, whilst the part to the left of the point is an integer; we thus have, as the result of shifting, a *mixed number*. We introduce a new intermediate mode to cope with this; it will have the same range as an integer, but its accuracy is that of a fraction; because of this increased accuracy it is termed a *fine integer.*



range: $-2^m \leqslant$ fine integer $< 2^m$
accuracy: multiples of $1/2^m$

If we write

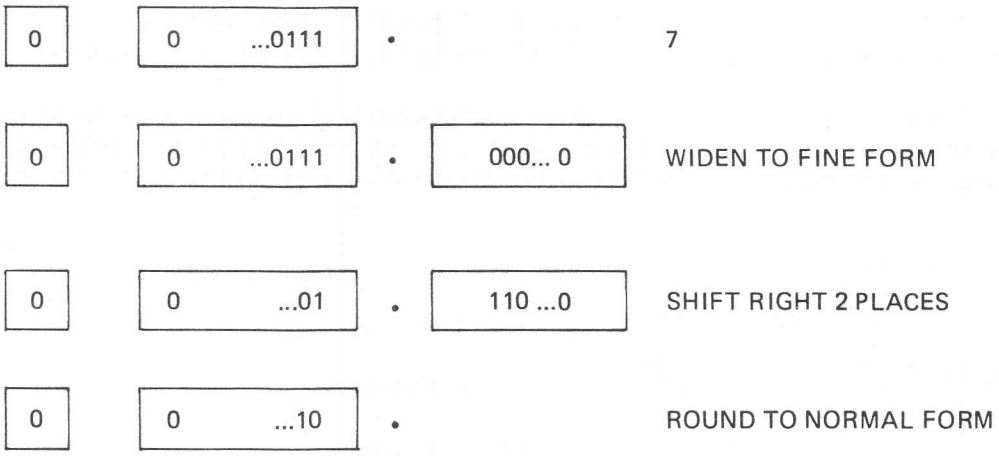
```
INT I := 7;
  I := I SRA 2;
```

what is stored in I? Initially I will contain 7; a shift right of 2 places is equivalent to a division by 2^2(=4) yielding the fine integer 1¾. This double length quantity cannot be stored in an integer so we must define the nature of the narrowing operation (performed automatically since we are dealing with an intermediate mode) that is required. The mode transfer from fine integer to normal integer is one of rounding, that is to the nearest normal value, the algebraically greater being taken in the case when the fine value lies midway between two normal values. Thus 2 will be stored in I.

This shift operation is not defined when the second operand is negative or greater than the number of bits in the word; what happens in practice will vary from machine to machine and may be unpredictable.

One of the uses of shifts is to accomplish scaling and the manipulation of double length quantities. For this reason, we define the action of shifts on intermediate modes also. There is

no extension to triple or greater length quantities; the resemblance to actual machines is retained, and only double-length intermediate modes are involved. If we shift right a big integer (formed by a multiplication) we obtain a big integer as a result, any bits going beyond the binary point being irretrievably lost and hence no question of rounding arises; any mode transfer from big integer to normal integer is the familiar contraction of range (with the possibility of overflow). Thus (3* 3)SRA 2 gives the result 2. If we shift right a fine integer (formed perhaps by a shift), we obtain another fine integer as result. The right shift on the normal integer is in fact performed as a shift on a fine integer, the widening to the fine form being carried out first by the addition of a zero fractional part. I:=7 SRA 2 may be pictured then as:



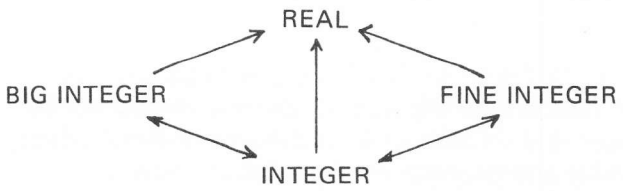
Note that zeroes appear in the word at the left; if the number had been negative, and hence the sign bit a one, ones would have appeared; this ensures that the sign is maintained and the division nature retained and is embodied in the fact that this is an *arithmetic* shift (see also section 26).

With the additional information that the precedence of all shifts is 6 – the highest precedence of all and hence the most tightly binding of dyadic operators – we can draw up a table for the SRA operator:

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
SRA	6	Fine Integer	Integer	Fine Integer	Shift operand 1 right arithmetically
		Big Integer	Integer	Big Integer	by operand 2 binary places

Note the way that the normal integer case (for the first operand) with its immediate widening to the fine form is catered for by making the fine integer case the first entry and using the typechecking rules already discussed in section 13. No overflow condition can arise from the right shift operation itself, but may occur on storing the result in an integer variable.

A mode transfer from a fine integer to a real will proceed directly (and not via an integer) as in the big case; in this way maximum accuracy is retained. In the cases of automatic transfers between the big and fine forms, these will occur via the normal form (and of course overflow is possible when a big integer is converted to fine and rounding occurs when a fine integer is converted to big). Our diagram depicting automatic mode transfers between integer and reals becomes



Examples:

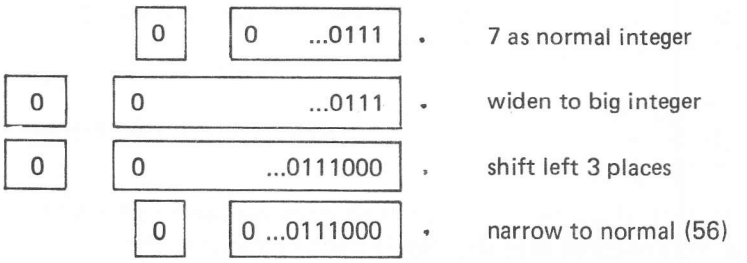
```
INT I;
REAL R;

I:=(3*3) SRA 2;          % BRACKETS ESSENTIAL FOR PRECEDENCE %
                         % BIG INTEGER RESULT NARROWED TO 2 %
I:=7 SRA 2;              % WIDEN TO FINE SHIFT AND ROUND TO 2 %
I:=17 SRA 2 SRA 2;       % FIRST SHIFT PRODUCES FINE 4 1/4 %
                         % SECOND SHIFT GIVES 1 1/16 ROUNDS TO 1 %

R:=(3*3) SRA 2;          % BIG INTEGER WIDENED TO 2.0 %
R:=7 SRA 2;              % FINE RESULT 1 3/4 WIDENED TO 1.75 %
R:=17 SRA 2 SRA 2;       % FINE RESULT 1 1/16 WIDENED TO 1.0625 %
```

The left shift follows a similar pattern. The keyword SLA specifies an arithmetic shift left; operating on a normal integer is equivalent to multiplication and hence will produce a big integer result; this is performed by first widening the integer to the big form and then shifting it.

I : = 7 SLA 3



Zeroes will be introduced in the right-hand end of the word. With a left shift (as a multiplication) there is always the possibility of overflow. The rule that "if the second operand is negative or greater than the word length of the machine then the action is undefined" again holds. Shifting a fine integer left will give another fine integer, and so our table appears as

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
SLA	6	Big Integer	Integer	Big Integer	Shift operand 1 left arithmetically by
		Fine Integer	Integer	Fine Integer	operand 2 binary places

Examples:

```
INT I;
REAL R;

I:=(3*3) SLA 3;          % BIG INTEGER NARROWED TO 72 %
I:=7 SLA 3;              % WIDEN TO BIG AND SHIFT THEN NARROW TO 56 %
I:=17 SRA 3 SLA 2;       % FINE INTEGER 2 1/8 SHIFTED LEFT IS 8 1/2 %
                         % ROUND TO NORMAL INTEGER 9 %

R:=(3*3) SLA 3;          % BIG INTEGER WIDENED TO 72.0 %
R:=7 SLA 3;              % BIG INTEGER WIDENED TO 56.0 %
R:=17 SRA 3 SLA 2;       % FINE INTEGER WIDENED TO 8.5 %
```

So far all our examples have involved explicit shifts, in the sense that the second operand has been an integer constant in each case. There is no reason why the second operand should not be a general expression; it must simply yield an integer value. Cases arise (in dynamic scaling) where the direction of the shift is not known until run-time and depends on the value of some

expression. For this situation, a general arithmetic shift operator is provided; in this case (since we do not know the direction of the shift) the normal integer is not widened before the shift; the big and fine forms result in big and fine forms, whilst the normal integer on shifting gives a normal integer. Naturally, overflow or loss of accuracy can occur. The direction of the shift is defined by the sign of the second operand – left if it is positive, right if it is negative. The number of binary places shifted is the magnitude of the second operand; if this magnitude exceeds the word-length of the machine, the action is once again undefined. The keyword SHA (Shift Arithmetic) is used for this operation:

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
SHA	6	Integer	Integer	Integer	Shift operand 1 by ABS (operand 2) binary places, left if operand 2 positive, otherwise right
		Big Integer	Integer	Big Integer	
		Fine Integer	Integer	Fine Integer	

Examples:

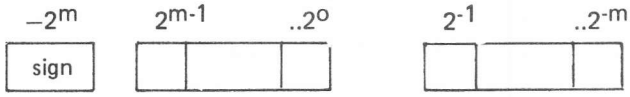
```
% IN PRACTICE THESE SHIFTS WOULD BE WRITTEN USING SRA OR SLA . %
% WE ASSUME HERE THAT THE VALUES USED AS SECOND OPERANDS ARE %
% DERIVED AT RUN TIME FROM SOME EXPRESSION %
INT I;
REAL R;

I:=(3*3) SHA -1;        % BIG INTEGER NARROWED TO 4 %
I:=7 SHA -2;            % 1; NOTE NO FINE INTEGER IS FORMED AND NO %
                        % ROUNDING. COMPARE WITH 7 SRA 2 %
I:=(17 SRA 4) SHA 2;    % FINE INTEGER 8 1/2 ROUNDED TO 9 %

R:=(3*3) SHA 3;         % BIG INTEGER WIDENED TO 72.0 %
R:=7 SHA -2;            % NO ROUNDING; WIDENED TO 1.0 %
R:=(17 SRA 4) SHA 2;    % WIDEN FINE INTEGER TO 8.5 %
```

Scaling and hence shifting are required for fractions. The three operators SRA, SLA, SHA are also defined on fractions; the second operand, specifying the number of binary places will still be an integer. The rules are identical, the only difference in these operations being the position of the binary point.

Shifting a fraction right gives a fine fraction, an intermediate mode already encountered. Shifting left will generate digits to the left of the point and gives rise to the fourth (and last) intermediate mode, the *big fraction.*



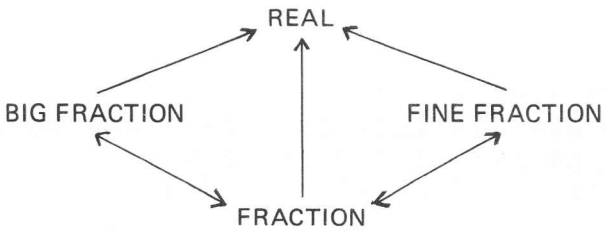
range: $-2^m \leqslant$ big fraction $< 2^m$
accuracy: multiples of $1/2^m$

A big fraction is thus a mixed number.
The tables for the operators now follow immediately:

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
SRA	6	Fine Fraction	Integer	Fine Fraction	Shift operand 1 right by operand 2 binary places
		Big Fraction	Integer	Big Fraction	
SLA	6	Big Fraction	Integer	Big Fraction	Shift operand 1 left by operand 2 places
		Fine Fraction	Integer	Fine Fraction	
SHA	6	Fraction	Integer	Fraction	Shift operand 1 by ABS (operand 2) places, left if operand 2 is positive, otherwise right
		Big Fraction	Integer	Big Fraction	
		Fine Fraction	Integer	Fine Fraction	

As in the integer case, for SRA and SLA a normal fraction will be widened (by the addition of suitable zeroes) to the fine and big form respectively, before performing the shift.

The automatic conversion from big fraction to fraction is similar to the integer case; the integer part is simply ignored, though an overflow condition may arise. Big and fine fractions are widened directly to reals if necessary, and so the automatic transfer diagram appears as



25. Mixed mode arithmetic: combining integers and fractions

If you look at the two intermediate modes representing mixed numbers (i.e. big fraction and fine integer) you will see that their internal representations, ranges and accuracies are identical; the distinction is made to indicate which "half" we are primarily interested in, and to define explicitly their behaviour under mode transfers. We introduce further uses of the monadic operators INT and FRAC to perform the necessary transfers between big fractions and fine integers:

OPERATOR	OPERAND	RESULT	INTERPRETATION
INT	Big Fraction	Fine Integer	Change mode of operand
FRAC	Fine Integer	Big Fraction	to that specified in result

There is no change of value nor possibility of overflow or loss of accuracy with these operations.

Armed with these operators and the concepts of the four intermediate modes, we can now investigate further the combination of integers and fractions in expressions.

If we multiply two integers together we obtain a big integer, if we multiply two fractions together we obtain a fine fraction. What happens if we multiply an integer by a fraction or a fraction by an integer? We are multiplying a value in the range $[-2^m, 2^m)$ by a value in the range $[-1, 1)$ and hence the answer lies in the range $[-2^m, 2^m]$; in terms of accuracy we have a multiple of 1 and a multiple of $1/2^m$ and hence the result is a multiple of $1/2^m$. Thus the answer is a big fraction apart from the case of multiplying together the most negative values (-2^m and -1) which will cause overflow. The result could have been chosen to be a fine integer, but we must choose either big fraction or fine integer so that there is no ambiguity; it is felt that such a multiplication will usually occur when we are primarily concerned with fraction arithmetic, and so the big fraction result is the appropriate one.

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
*	5	Integer	Fraction	Big Fraction	Form product of
		Fraction	Integer	Big Fraction	operands

Let J, K be declared as integers, and F as a fraction; note carefully the difference between the two expressions

```
J * (K * F)
J * INT (K * F)
```

In the first, K * F yields a big fraction which is narrowed to a fraction and a big fraction is the final result; in the second the INT operator yields a fine integer which is narrowed to an integer and a big integer is the final result.

In the reverse process, division, there is no ambiguity in allowing both fine integer and big fraction as the dividend; this is basically because division is not a reflexive operation and hence the two cases can be distinguished. The division operators (:/,// and MOD) are extended to cover the reverse cases of the multiplication just discussed; which operator to use for division depends on the nature of the result required:
thus :/ always yields an integer, // always yields a fraction; the result of MOD then follows by considering the nature of the remainder in each integer division case:

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
:/	5	Big Fraction	Fraction	Integer	Quotient on dividing
		Fine Integer	Fraction	Integer	Operand 1 by operand 2
MOD	5	Big Fraction	Fraction	Fraction	Remainder on dividing
		Fine Integer	Fraction	Fraction	Operand 1 by operand 2
//	5	Big Fraction	Integer	Fraction	Quotient on dividing
		Fine Integer	Integer	Fraction	Operand 1 by operand 2

The rules governing the divisions are as before, with truncation towards zero and the sign of the remainder being the same as the sign of the dividend; note that overflow can occur in all cases.

We can use these operations to extract (in a machine independent manner) the integer and fraction parts of a mixed number without the possibility of overflow, but we must form the mixed number twice:

```
INT I,J;
FRAC F,G;

    J:= -I*F :/ -1.0B0;
    G:= I*F MOD -1.0B0;
% NOTE THAT J:=INT(I*F) IS NOT SATISFACTORY SINCE ROUNDING OCCURS %
% ON ASSIGNMENT %
```

In section eleven when discussing conditional expressions we showed that dereferencing would be applied where necessary to ensure that the expression delivered the same kind of object whichever route is taken at run-time.

```
REAL R;
REF REAL RR:=R;
    .
    .
    R:=IF R=0.0 THEN RR ELSE R*0.1 END;
    % RR DEREFERENCED TWICE; EXPRESSION YIELDS A REAL %
```

Similarly automatic typechanging is applied to ensure that an expression yields a definite unique mode regardless of the run-time path; further, the typechanging is such that an intermediate mode will not be yielded.

```
INT I;
FRAC F;
REAL R;

IF I=0 THEN I ELSE R END          % YIELDS REAL %

IF R=0.0 THEN I*I ELSE I SLA 1 END
    % YIELDS INTEGER EVEN THOUGH EACH ROUTE GIVES A BIG INTEGER %

IF R=0.0 THEN F*F ELSE I*F END % YIELDS FRACTION %

IF R=0.0 THEN I ELSE F END
    % YIELDS REAL - ONLY MODE TO WHICH BOTH INTEGER AND FRACTION %
    % CAN BE WIDENED %
```

This can cause some difficulties, since there may be undesirable loss of accuracy by the narrowing to a normal form. To allow widening to be forced at the appropriate moment, we introduce the monadic operator REAL which is the only explicit widening operator; this gives the ability to float a fixed point number at the appropriate point in a program; it is also defined for a real operand for completeness.

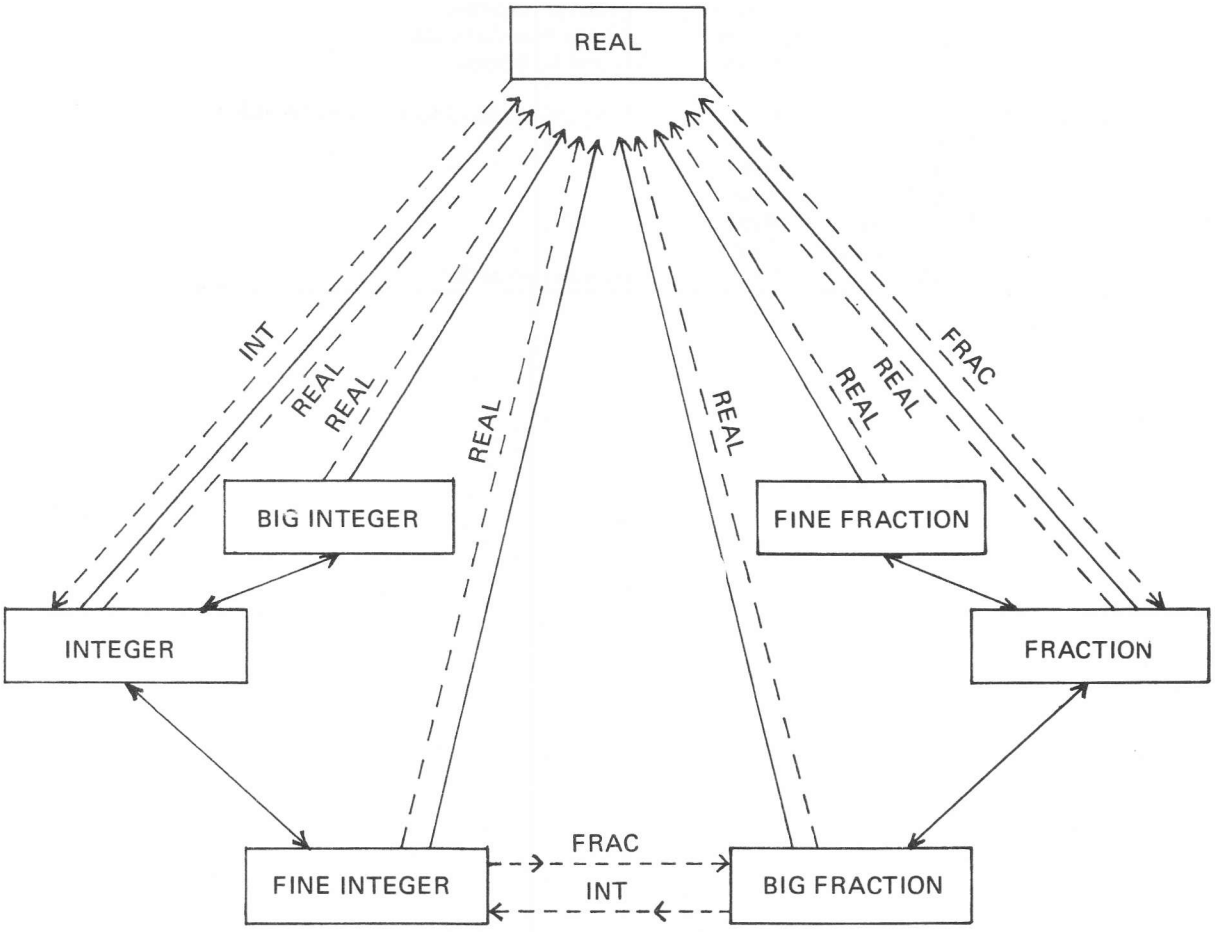
OPERATOR	OPERAND	RESULT	INTERPRETATION
REAL	Integer	Real	Floats the operand to
	Big Integer	Real	give a real number
	Fine Integer	Real	
	Fraction	Real	
	Big Fraction	Real	
	Fine Fraction	Real	
	Real	Real	Identity: no action

```
R:=IF R=0.0 THEN I*I ELSE I END;
R:=IF R=0.0 THEN REAL(I*I) ELSE I END;
   % SECOND CASE AVOIDS POSSIBILITY OF OVERFLOW BY FORCING %
   % EXPRESSION TO YIELD A REAL. IN THE FIRST CASE IT YIELDS %
   % AN INTEGER WHICH IS THEN WIDENED %

R:=IF R=0.0 THEN 3*0.5B0 ELSE 0.06B0 // 0.15B0 END;
   % EXPRESSION YIELDS FRACTION; FIRST ALTERNATIVE CAUSES OVERFLOW %
R:=IF R=0.0 THEN 3*0.5B0 ELSE REAL(0.06B0 //0.15B0) END;
   % ONE EXPLICIT WIDENING TO REAL CAUSES EXPRESSION TO BE %
   % EVALUATED AS A REAL %
```

Having seen so many typechange situations and operators in this section, we now present the typechange diagram and operator tables as completed so far, to show the order in which operations will be attempted, as described in section 13.

Mode Transfer Diagram



Automatic transfer ——————>

Explicit transfer (with operator) ——INT——>

Monadic Operators

OPERATOR	OPERAND	RESULT	INTERPRETATION
+	Integer	Integer	Identity: no action
	Fraction	Fraction	
	Real	Real	
–	Integer	Integer	Negate the operand
	Fraction	Fraction	
	Real	Real	
ABS	Integer	Integer	Negate if the operand is negative,
	Fraction	Fraction	otherwise no action
	Real	Real	
LENGTH	Array	Integer	Return length of array
INT	Integer	Integer	Identity: no action
	Big Fraction	Fine Integer	Change mode of value
	Real	Integer	Round to integer
FRAC	Fraction	Fraction	Identity: no action
	Fine Integer	Big Fraction	Change mode of value
	Real	Fraction	Round to fraction
REAL	Integer	Real	Float the operand to give a real number
	Big Integer	Real	
	Fine Integer	Real	
	Fraction	Real	
	Big Fraction	Real	
	Fine Fraction	Real	
	Real	Real	Identity: no action

Dyadic Operators

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
+	1	Integer Fraction Real	Integer Fraction Real	Integer Fraction Real	Form sum of operand
–	1	Integer Fraction Real	Integer Fraction Real	Integer Fraction Real	Subtract operand 2 from operand 1
*	5	Integer Integer Fraction Fraction Real	Integer Fraction Integer Fraction Real	Big Integer Big Fraction Big Fraction Fine Fraction Real	Form product of operands
/	5	Real	Real	Real	Divide operand 1 by operand 2; no remainder
:/	5	Big Integer Fine Integer Big Fraction	Integer Fraction Fraction	Integer Integer Integer	Integer quotient when operand 1 divided by operand 2; truncation to zero
MOD	5	Big Integer Fine Integer Big Fraction	Integer Fraction Fraction	Integer Fraction Fraction	Remainder when operand 1 divided by operand 2; has sign of operand 1.
//	5	Fine Integer Big Fraction Fine Fraction	Integer Integer Fraction	Fraction Fraction Fraction	Fraction quotient when operand 1 divided by operand 2; truncation to zero
SRA	6	Fine Integer Big Integer Fine Fraction Big Fraction	Integer Integer Integer Integer	Fine Integer Big Integer Fine Fraction Big Fraction	Arithmetically shift operand 1 right by operand 2 binary places
SLA	6	Big Integer Fine Integer Big Fraction Fine Fraction	Integer Integer Integer Integer	Big Integer Fine Integer Big Fraction Fine Fraction	Arithmetically shift operand 1 left by operand 2 binary places
SHA	6	Integer Big Integer Fine Integer Fraction Big Fraction Fine Fraction	Integer Integer Integer Integer Integer Integer	Integer Big Integer Fine Integer Fraction Big Fraction Fine Fraction	Arithmetically shift operand 1 by ABS (operand 2) binary places; to the left if operand 2 is positive, otherwise right.

Notes

1. The form of the first operand and result in the arithmetic shifts can be condensed into the following table, entries showing the result:

OPERAND 1 \ OPERATOR	SLA	SHA	SRA
BIG	BIG	BIG	BIG
NORMAL	BIG	NORMAL	FINE
FINE	FINE	FINE	FINE

2. Fractions and integer values are only combined (the second operand in the shifts is somewhat different) in multiplication and divisions. Thus for an integer I, and a fraction F, the expression I+F will be evaluated as a real with appropriate widening performed.

The following example illustrates the use of fractions. In it, instead of choosing to scale readings (in this case integer readings) to a fraction of a full-scale value, we choose the first power of two greater than the full-scale value; reduction to fraction form can then be achieved by a simple shift rather than a division. The problem dealt with here is an electrical network with a number of nodes and lines joining them. Sample procedures show how readings might be treated and how the electrical properties of the network might be recalculated on switching into use an additional line. No details of the physics of the situation are given; it is hoped that the RTL/2 (with its comments of course) will indicate the use of fractions and scaling and the mixing of these variables with integers and reals.

```
LET ZERO=0.0B0;          % FRACTION ZERO %
LET NODES=70;            % NUMBER OF NODES %
LET LINES=100;           % NUMBER OF LINES %

LET IMPPLACES=6;         % FULL-SCALE IMPEDANCE IS 50 OHMS %
                         % BINARY SCALE IS THEREFORE 64 OR 6 PLACES %
LET CURRENTPLACES=14;% FULL SCALE CURRENT IS 10000 AMPS; BINARY 163
LET POTPLACES=19;        % FULL-SCALE POTENTIAL IS 500000 VOLTS-19 PLAC

LET DIFFPLACE=1;         % THIS IS THE SCALING DIFFERENCE BETWEEN %
                         % POTENTIAL AND CURRENT*IMPEDANCE %

DATA NETWORK;
   ARRAY (NODES,NODES) FRAC IMPED;
      % EFFECTIVE IMPEDANCE BETWEEN TWO NODES OF NETWORK %
      % IMPED(I,J) IS THE IMPEDANCE BETWEEN NODES I AND J %
   ARRAY (LINES) FRAC LINEIMP;
      % IMPEDANCES OF UNLOADED LINES %
   ARRAY (NODES) FRAC
      CURRENT,     % NET CURRENT ENTERING NODE %
      POTENTIAL,   % POTENTIAL AT NODE %
      TEMP;        % WORK ARRAY %
   ARRAY (LINES) INT LONODE:=HINODE:=(-1(LINES));
      % LINE NUMBER L JOINS LONODE(L) TO HINODE(L) WITH LONODE(L) %
      % NUMERICALLY SMALLER. -1 INDICATES LINE NOT IN USE %
   ARRAY (NODES) INT LINENDS;
      % NUMBER OF LINES IN USE AT NODE %
ENDDATA;
```

```
PROC READPOT (INT POT) FRAC;
% SCALES ABSOLUTE INTEGER READING INTO FRACTION OF CORRECT SCALE %
   RETURN( FRAC(POT SRA POTPLACES) );
ENDPROC;

PROC OUTPOT (FRAC POTVAL) INT;
% RETURNS ACTUAL VALUE FROM FRACTION %
   RETURN( INT(POTVAL SLA POTPLACES) );
ENDPROC;

% SIMILAR PROCEDURES FOR CURRENT AND IMPEDANCE %

PROC ADDLINE ( INT LINE,      % NEW LINE NUMBER %
                   FROM,      % LOWER NODE NUMBER %
                   TOP        % UPPER NODE NUMBER %      );
% THIS SWITCHES A NEW LINE INTO USE %
% ASSUME A RETURN EXIT FOR ERROR ACTION %
   IF LINE>LINES THEN
      % ERROR : ILLEGAL LINE %
   END;
   IF FROM>NODES THEN
NOTNODE; % ERROR : ILLEGAL NODE OR ORDER %
   END;
   IF TOP>NODES THEN GOTO NOTNODE;  END;
   IF FROM>=TOP THEN GOTO NOTNODE;  END;
   IF LONODE(LINE)#-1 THEN
      % ERROR : LINE ALREADY IN USE %
   END;

% LINE INFORMATION NOW ACCEPTED %

   LONODE(LINE):=FROM;        % ENTER LINE INFORMATION %
   HINODE(LINE):=TOP;
   LINENDS(FROM):=LINENDS(FROM)+1;      % UPDATE NODE INFORMATION %
   LINENDS(TOP):=LINENDS(TOP)+1;

   FOR I:=1 TO NODES DO
      % CALCULATE INTERMEDIATE IMPEDANCE VARIABLE %
      TEMP(I):=IMPED(I,FROM) - IMPED(I,TOP);
   REP;

   BLOCK
   REAL MUTUAL:=LINEIMP(LINE) + TEMP(TOP);
   % MUTUAL IMPEDANCE; REAL USED BECAUSE OF RANGE AND ARITHMETIC BELOW
      % NOW UPDATE IMPEDANCE MATRIX %
      % THIS IS STRICTLY A TRIANGULAR MATRIX BUT WE REPEAT ENTRIES %
      FOR I:=1 TO NODES DO
      FRAC P:=TEMP(I);
         FOR J:=1 TO NODES DO
         REAL PJ:=P-TEMP(J);
            IMPED(I,J):=IMPED(I,J) - FRAC(PJ*PJ/(MUTUAL*4.0));
         REP;
      REP;
   ENDBLOCK;
ENDPROC;
```

```
PROC SETPOTENTIAL ();
% SETS UP THE POTENTIAL OF EACH NODE %
   FOR I;=1 TO NODES DO
   FRAC POTI;=ZERO;
      FOR J;=1 TO NODES DO
         % ADD CONTRIBUTION FROM EACH POSSIBLE LINE %
         % IN PRACTICE FURTHER SCALING MIGHT BE NECESSARY HERE %
         % IN THE ACCUMULATION - DEPENDS ON PHYSICAL CONSTRAINTS %
         POTI;=POTI + (IMPED(I,J)*CURRENT(J))SLA DIFFPLACE;
      REP;
      POTENTIAL(I);=POTI;
   REP;
ENDPROC;

PROC LINEFLOW (INT LINE) FRAC;
% RETURNS CURRENT FLOWING IN THE LINE %
% RETURNS ZERO IF THE LINE IS NOT IN USE %
   IF LONODE(LINE)=-1 THEN RETURN(ZERO);  END;
   RETURN( (POTENTIAL(HINODE(LINE)) - POTENTIAL(LONODE(LINE)))
                     SRA DIFFPLACE
                % SCALE NOW AS FOR IMPEDANCE*CURRENT %
                                          // LINEIMP(LINE)    );
ENDPROC;
```

26. Extension of conditions

In the compound interest example, when we wished to check that a parameter was in a desired range, we had to employ a number of conditional statements:

```
IF T < 1 THEN GOTO FAIL; END;
IF T > 365 THEN GOTO FAIL; END;
```

The need to test whether a value lies within a range is common; it is also common to perform a set of actions when some combination of conditions is satisfied. To combine conditions together, RTL/2 has two keywords AND and OR which are formal versions of the natural English words. Since any condition is an assertion which is true or false, we can describe the action of AND as if it were an operation between the values true and false; that is we can construct a *truth table:*

AND	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

Thus the combined condition formed by connecting two conditions with the keyword AND is only true if both of the 'subconditions' are true.

OR	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

The condition formed with OR is only false if both the subconditions are false. Our example then becomes

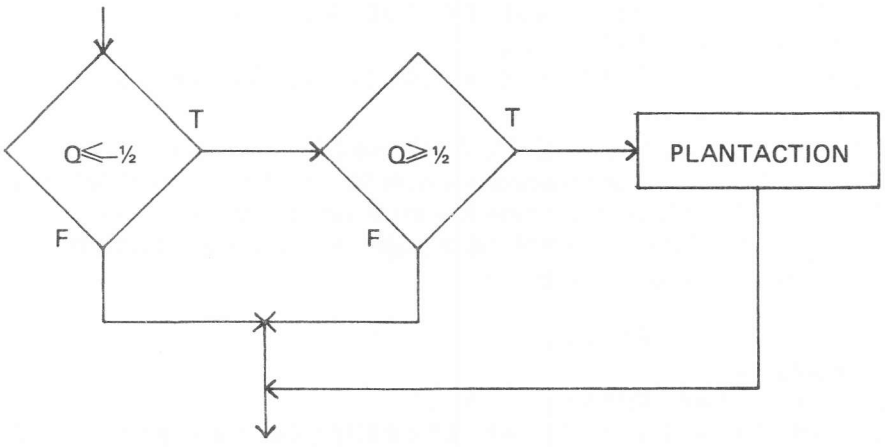
```
IF T < 1 OR T > 365 THEN GOTO FAIL; END;
```

If we want to perform some plant action if and only if some fraction variable Q lies between –½ and +½ then we would write

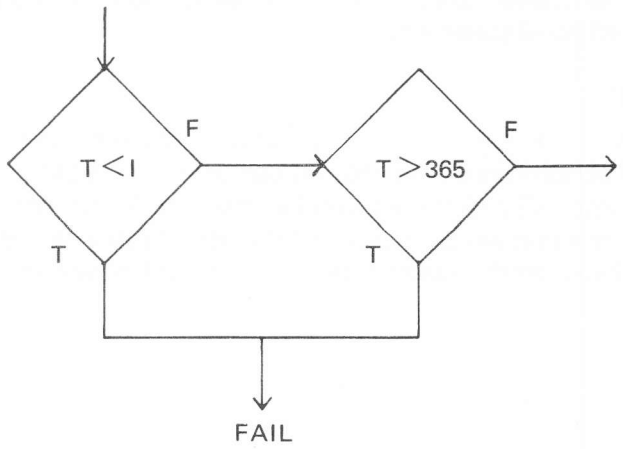
```
IF Q>=-0.5B0 AND Q<=+0.5B0 THEN
   PLANTACTION ();
END;
```

In our earlier discussion on conditional statements, we pointed out that when we have a whole list of conditions embedded within IF, ELSEIF etc, we would only evaluate conditions until we found the correct path; this could cause trouble (or be used to advantage) when procedures with side effects were involved in these conditions. Similarly, when AND and OR are used, subconditions are only evaluated until we can decide the truth value of the whole condition; thus if for example T is zero the first condition in our OR example (T<1) is true; we know T is outside the range and we can immediately transfer to FAIL; similarly if the first condition in our AND example (Q≤–0.5B0) is false we know the value of Q is out of range, the whole condition is false and we continue with the statement following END. The actions can be represented by the following flowcharts:

AND:



OR:



The similarity between the flow diagrams shows how by reversing the subconditions we could rewrite the statements:

```
IF T>=1 AND T<=365 THEN
ELSE GOTO FAIL;
END;

IF Q<=0.5B0 OR Q>+0.5B0 THEN
ELSE PLANTACTION ();
END;
```

However we have lost some clarity.

More complex conditions can be written by connecting a number of subconditions with AND and OR; the fundamental rule that subconditions will be evaluated from left to right only until the truth value of the whole condition is determined still holds.

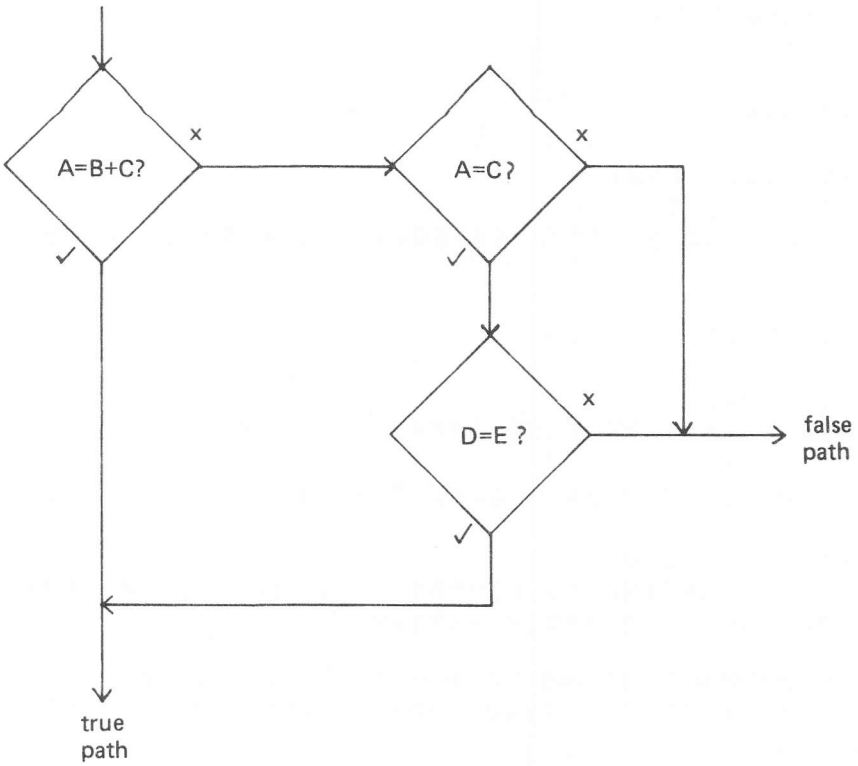
```
IF A=1 AND B=2 AND C=3 THEN ...
% TRUE WHEN ALL THREE CONDITIONS ARE SATISFIED %
IF READ()=1 OR READ()=2 OR READ()=3 THEN ...
% IF READ READS THE NEXT VALUE ON A TAPE, NOTE THAT THE NUMBER %
% OF VALUES READ DEPENDS ON THOSE VALUES. THUS ONE VALUE IS READ %
% IF THE FIRST VALUE IS 1; IF IT IS NOT ANOTHER NUMBER IS READ %
% IF THIS IS NOT 2 THEN A THIRD NUMBER IS READ. THIS MAY OR MAY %
% NOT BE WHAT YOU INTEND %

% THE NEXT EXAMPLE TAKES ADVANTAGE OF THE RULE %
IF J<=LENGTH P AND P(J)#0 THEN ...
% P(J) ONLY ACCESSED IF J CONTAINS A VALID SUBSCRIPT %
```

As soon as we try to construct a condition containing AND and OR we encounter the precedence problem, first met with + and *. In this case however we wish to preserve the strict left to right evaluation of subconditions and be able to determine the truth value as soon as possible. Bracketing of subconditions is therefore not permitted(though naturally a subcondition may contain a normal bracketed arithmetic expression).

```
IF (A=B OR A=C) AND D=E THEN ...
   % THIS IS ILLEGAL %
IF A=(B+C) OR A=C AND D=E THEN ...
   % THESE BRACKETS FORM PART OF AN ARITHMETIC EXPRESSION %
```

What is the meaning of the second condition? AND is regarded as having the higher precedence (or to be the more tightly binding); thus AND, OR behave similarly to *,+. The condition is true then if either A=B+C is true or A=C AND D=E is true (or if both are true).




```
IF A>=1 AND A<=26 OR B>=0 AND B<=9 THEN ...
% TRUE IF A IS IN RANGE 1 TO 26 OR B IS IN 0 TO 9.%
```

How would we write the example in which we attempted to use brackets? We could write it as a compound condition

```
    IF A=B AND D=E OR A=C AND D=E THEN...
```

in which we repeat one of the tests, or we could split it into two nested conditional statements:

```
    IF A=B OR A=C THEN
      IF D=E THEN...
```

In this section our examples so far have been conditional statements; complex conditions involving AND and OR can be used whenever a condition is valid and hence we may use them in conditional expressions and in while-statements:

```
P:=IF A=B AND C=D OR Q=R THEN 0.0 ELSE 3.1 END;
IF A=B THEN ...
ELSEIF C#D AND A+B<2 THEN ...
END;
WHILE X>100 OR Y<50 DO ...
REP;
```

Simple conditions operate at the numerical level; that is, we are comparing actual real, integer or fraction values. If a reference variable occurs in an expression on one side of a comparator it is dereferenced twice to yield an appropriate value. Sometimes, however, we wish to compare the contents of two reference variables, and make an assertion concerning the names contained in them; only one level of dereferencing is then required; similarly we may wish to compare the contents of a reference variable with a given name. No type changing is involved here, and names and reference variables must be of the appropriate modes. The only comparisons that are meaningful are those of equality and inequality; to represent the comparison at the name level,

we use the compound comparators :=: and :#:. These separators are formed by the concatenation of three symbols and must not contain any spaces; as usual, # may also be represented by the symbol £ or $.

```
INT I,J;    FRAC F;    REAL R,S;
REF INT WHICH:=I,QI:=J;
REF FRAC RF:=F;
REF REAL WHERE:=R,WHO:=S;

    IF WHICH:=:QI THEN  ...  END;
    % BOTH DEREFERENCED ONCE %
    % TESTS WHETHER WHICH AND QI BOTH REFERENCE THE SAME INTEGER %
    % VARIABLE %

    IF RF:#:F THEN  ..  END;

    WHILE R:#:S DO  ..  REP;
    % NO DEREFERENCING; ALWAYS TRUE SO INFINITE LOOP %

    J:=IF WHICH:=:I OR WHO:#:R THEN 2 ELSE 3 END;

    IF WHERE:=:QI THEN  ...  END;
    % ILLEGAL : WE ARE ATTEMPTING TO COMPARE THE NAME OF A REAL %
    % VARIABLE WITH THAT OF AN INTEGER VARIABLE %
```

In a condition, therefore, the amount of dereferencing is governed by the nature of the comparator used. Naturally there is no restriction on mixing various sorts of conditions within a complex condition using AND or OR.

```
IF A<6+B*B OR WHICH :#:R AND WHO #S THEN...
```

Make sure you can indicate where dereferencing occurs.

27. Logical operations

Suppose that we wish to construct a very simple personnel record system, holding for each person his (or her) age, sex, location and salary scale. The age information will be a number in the range 15-70, sex is a simple binary choice, the location might be a code number in the range 0-6 and the salary scale some number in the range 0-10. We can access this information by assigning to each person an index number and holding the various pieces of information in arrays. For a large number of people, the space taken by four integer arrays (one for each item of information) may be excessive; in any case it is wasteful to use an integer simply to record 0 or 1 for male or female. We can take advantage of our knowledge of the binary representation of numbers in the computer to *pack* this information more efficiently. The integers will then be not so much numerical values as logical patterns containing various pieces of information.

If we consider a machine having a word-length of 16 bits (the minimum for RTL/2), we can investigate the items in our example and see how they could be packed into a single integer word.

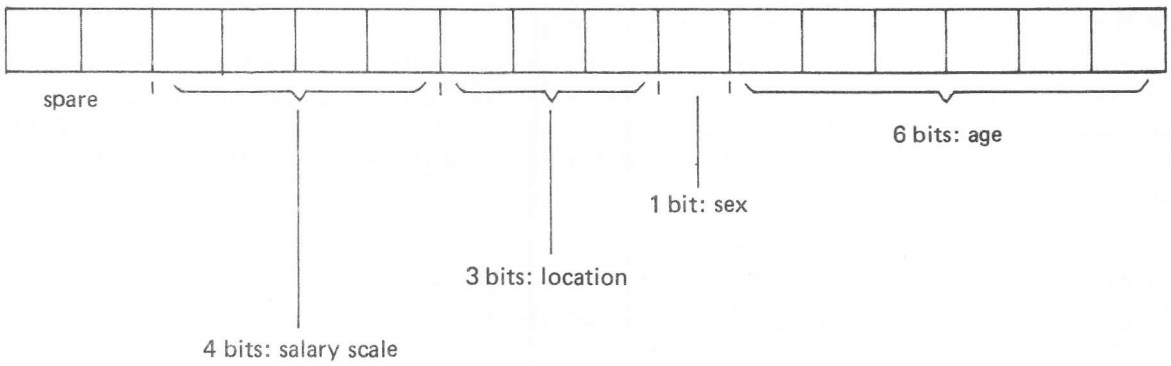
Age: range is 15-70 which we could store as (age – 15) i.e. in the range 0-55; such a range can be held in 6 bits (giving numbers 0 to 2^6-1; that is, 0-63).

Sex: clearly 1 bit is sufficient : 0 for male
: 1 for female

Location: range is 0-6 which can be held in 3 bits (allows numbers 0-7).

Salary scale: range is 0-10 which can be held in 4 bits (allows numbers 0-15).

Hence the bits in our integer word might be utilised as follows:



Now one array will suffice to store the personnel details

```
LET EMP=1000;    % NUMBER OF EMPLOYEES %

DATA PERSONNEL;
   ARRAY (EMP) INT PEOPLE;
ENDDATA;
```

To store the details of a man aged 32 on scale 3 in location 4, whose index in the scheme is 21 we would then write:

```
PEOPLE(21):=BIN 0000111000010001;
% VALUES 3,4,0,17 HELD IN PACKED FORM %
% NOTE THE USE OF BINARY CONSTANT FORM %
```

This is simple enough; how, though, are we going to unpack this information (dynamically) for use in our program? For such purposes and the manipulation of logical quantities we define in RTL/2 a number of *logical operators.* These are operators that act on (or between) binary patterns, and are defined in terms of these; our tables will still show them acting on integers, but we are regarding them as binary patterns.

The first two operations are bitwise operations between two integer operands and are logically similar to the AND/OR operations between conditions; the keywords LAND and LOR (for logical and/or) are used, and the result of the operations between bits can be summarised in a way similar to truth tables (they are in fact identical – we are simply using a bit of 0,1 to represent false, true).

LAND	0	1
0	0	0
1	0	1

LOR	0	1
0	0	1
1	1	1

Thus the result of a LAND operation (a further integer pattern) has a one bit where both operands had ones, the result of a LOR has a one bit in every position where at least one of the operands had a one.

As dyadic operators they have the following properties:

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
LAND	4	Integer	Integer	Integer	Bit-wise logical and.
LOR	3	Integer	Integer	Integer	Bit-wise logical or.

Examples:

BIN 101101 LAND BIN 110011 is BIN 100001
BIN 1101011 LOR BIN 10100 is BIN 1111111

The LAND operator is particularly useful for extracting a particular *"field"* from a packed pattern. In our example, the four pieces of information could be isolated (still in the appropriate bits of integers and not as direct numerical values) by the use of the correct binary patterns; these *masks* will have ones just in the field positions so that any ones there in the pattern of interest will be extracted.

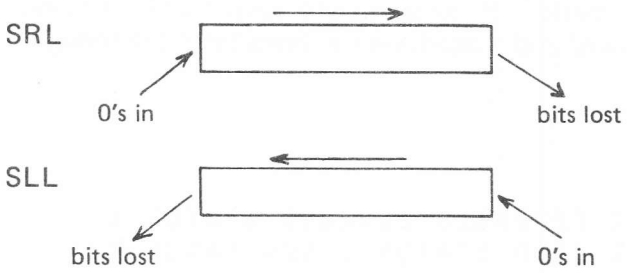
```
AGE:=PEOPLE(I) LAND OCT 77 + 15;
% PICKS UP BOTTOM 6 BITS AND ADDS 15 FOR RANGE 15 TO 70 %
% NOTE USE OF PRECEDENCE OF LAND OVER + %
SEX:=PEOPLE(I) LAND OCT 100;
LOC:=PEOPLE(I) LAND OCT 1600;
SCALE:=PEOPLE(I) LAND OCT 36000;
```

The LOR operation can be used to set specific bits in a word; thus I LOR OCT 77 will ensure that the bottom six bits of the result all contain ones. Utilising the precedences of LAND and LOR (note that these follow the same binding rule as AND/OR) we can fill in new values for our field by masking out the old value and LOR-ing in the new one:

```
PEOPLE(I):=PEOPLE(I) LAND OCT 1777 LOR NEWSCALE;
% FIRST LAND OPERATION RETAINS OTHER THREE FIELDS %
% THE FORM OLD LAND MASK LOR NEW IS VERY USEFUL %
```

Working arithmetically with values in the 'wrong' part of an integer is tedious and error prone (particularly with respect to overflow); we have the bit pattern of a value, but in the wrong bits of the word. We therefore define a number of shift operations which will enable us to align values correctly and pack values efficiently. These *logical shifts* are quite different from the arithmetic shifts defined earlier. They are only defined on integers, and no intermediate double-length quantities are formed; there is no question of overflow and any bits going beyond the end of the word are irretrievably lost. In logical shifts the sign bit behaves in precisely the same way as any other bit; zero bits are introduced at the opposite end of the word. However, the rule that the shift is undefined if the second operand is negative or greater than the word-length of the machine still holds; again, a general shift whose direction is determined by the sign of the second operand is also available. The keywords used are SRL, SLL, SHL (Shift Right Logical, Shift Left Logical, Shift Logical) and they have the same precedence as the earlier arithmetic shifts.

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
SRL	6	Integer	Integer	Integer	Shift pattern in operand 1 right by operand 2 places
SLL	6	Integer	Integer	Integer	Shift pattern in operand 1 left by operand 2 places
SHL	6	Integer	Integer	Integer	Shift pattern in operand 1 by ABS (operand 2) places, left if operand 2 positive, otherwise right



SHL acts as SLL if the second operand is positive, otherwise it behaves as SRL.

Thus to obtain the numerical value of the salary scale we write:

```
SALSCALE:=(PEOPLE(I)LAND OCT 36000) SRL 10;
```

It is probably clearer to perform the shift first, and use a simpler mask to isolate the four bits; no brackets are then required for the precedence.

```
SALSCALE:=PEOPLE(I)SRL 10 LAND OCT 17;
```

If we can be sure that the top two 'spare' bits contain zeroes always, the LAND operation is unnecessary, since any other one bits will have been lost:

```
SALSCALE:=PEOPLE(I) SRL 10;
```

But the mask is essential for the location value:

```
LOCAT:=PEOPLE(I) SRL 7 LAND 7;
% NOTE THAT DECIMAL NUMBERS CAN BE USED - THEY ARE INTEGERS %
% BINARY PATTERNS ARE CLEARER FOR LARGE PATTERNS %
```

Note that we can also accommodate "short" signed integers in a field; suppose that 5 bits are used to hold numbers in the range −15 to +15 with the top bit of the field behaving as a sign bit for this "short word" (i.e. as a multiple of $-2^4 = -16$). Then we can unpack this field as follows, but note that the number of places of shift is machine dependent.

```
LET   WORDBITS4 = 12;
LET   WORDBITS9 =  7;          % FOR 16 BIT MACHINE %

VALUE:= (PACKEDPATTERN LAND HEX 1FO)    % PICKS OUT FIELD %
        SLL   WORDBITS9        % SHIFTS SHORT SIGN BIT INTO PROPER SIGN%
        SRA   WORDBITS12;      % SHIFT DOWN TO NORMAL INTEGER%
                               % SIGN BIT IS PROPAGATED%
```

We can pack up information by shifting up the appropriate field and LOR-ing the fields together: this is better than addition (which will fail in the case when the sign bit is being used logically).

```
PEOPLE(I):=SALSCALE SLL 10 LOR LOCAT SLL 7 LOR SEX SLL 6 LOR(AGE-15);
% ALTERNATIVELY %
PEOPLE(I):=((SALSCALE SLL 3 LOR LOCAT)SLL 1 LOR SEX)SLL 6 LOR (AGE-15
```

One other logical dyadic operator (and incidentally this is the last dyadic operator to be introduced) is available. This enables us to isolate the bits that differ between two patterns; this is especially useful in comparing two consecutive values of a pattern containing status information

to ascertain what has changed. It is again a bit-wise operation and uses the keyword NEV (not equivalent, exclusive or).

NEV	0	1
0	0	1
1	1	0

Its precedence completes the scale 1 to 6.

OPERATOR	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
NEV	2	Integer	Integer	Integer	Bit-wise exclusive or.

Note that if A NEV B = 0 the patterns A, B are identical. If the status of 6 valves are held as the six bottom bits of a word with 0 representing closed and 1 open, we can investigate the changes as follows:

```
INT STATUS,OLDSTATUS;
    .
    .
    OLDSTATUS:=STATUS;              % REMEMBER CURRENT STATUS %
    READSTATUS(STATUS);             % READ STATUS : NEW VALUE %
    IF OLDSTATUS NEV STATUS # 0 THEN
       % SOME CHANGES IN POSITION OF VALVES %
    END;
    .
    .
```

We should of course be worried in the following case!

```
IF (OLDPERSON(I) NEV PERSON(I)) LAND OCT 100 # 0 THEN ...
% OLDPERSON CONTAINS PREVIOUS INFORMATION %
```

One further logical operator is provided in RTL/2; this enables a complementary pattern to be generated; application of the keyword NOT to a binary pattern changes the value of each bit.

OPERATOR	OPERAND	RESULT	INTERPRETATION
NOT	Integer	Integer	Reverse each bit in pattern.

We must realise here that the definition of the logical operators and the fact that their use may be combined with arithmetic operations makes it imperative to know the representation of integers within the machine (and in some cases to know the word-length of the machine); for this reason, RTL/2 explicitly specifies that integers will be held in two's complement form.

At first sight NOT BIN 1101100 is 0010011; but if we are working on a 16 bit machine, the actual operation will be NOT BIN 0000000001101100 and the result is therefore BIN 1111111110010011. We leave as an exercise in binary arithmetic the proposition that NOT I = −I−1 is true regardless of the word-length of the machine.

Note too that to obtain a pattern with a certain number of zero bits at the bottom of a word (for use as a mask say) in a machine independent manner, the use of negative integers or NOT is required:

```
    −8          gives      11.....1111000
or NOT 7        gives   NOT BIN 00...00111
                which is   11....11000
```

Whereas

```
    HEX FFF8  generates 11...11000 on a 16 bit machine
              but ...000011...11000 with any larger word-length.
```

Section 27 examples

1. Plant information is recorded as follows: an integer value in the range −100 to +100, a sequence number 0, 1, 2, or 3, a plant status value 0 to 7 and three indicator flags A, B, C (set if 1). It is required to store consecutive readings of these data in an array of integers. Devise a packing of the information and write procedures to update and read the individual items.

   Write a procedure (using those above) to scan the array and detect the following alarm conditions.

   i) sequence number =3 and the integer value within 10 of its limits.
   ii) instability: one of the following conditions has occurred.

	A set	A not set
Status value even	B set	both B and C not set
Status value old	C set	B or C set

28. Bytes

This section introduces the fourth and last of the plain (arithmetic) modes in RTL/2, the mode *byte.* The values that can be taken are a subset of the integer values; a byte value is an 8 bit binary pattern which represents integer values in the range 0 to 255. Its range is thus fixed, its accuracy is exact and we have a mode which is completely machine independent. Bytes will be used to save space when dealing with small range information (e.g. status flags), for character handling, and to achieve machine independence in certain areas.

There is no byte constant in RTL/2. Constant values for the mode byte are integer constants whose values lie in the range 0 to 255. We can think of this as a semantic restriction on the integer constant. The following are all permissible values for a byte:

```
27
BIN 1101
OCT 300
HEX FF
```

To facilitate character handling, a further form of the integer constant is available in RTL/2 – the *character constant.* Each character in the RTL/2 language subset of the ISO7 character set has a value associated with it, the value of the binary pattern (ignoring the parity bit) formed on 8-hole tape by punching the character on ISO7 preparation equipment. These values are given in Appendix 1. This numerical value can be expressed in RTL/2 by enclosing the character in single quotation marks; thus 'A' is used to represent the numerical value of the character A (which happens to be 65). One member of the character set is the space character; its value can be written in the form ' '. The character constant is an item and therefore cannot contain any non-significant layout characters; although we can write ' ' we cannot write ' B' nor ' ' nor 'X '; the syntax demands three symbols in the form 'character'. Unfortunately, not all the characters in the RTL/2 set may be used in this way; there are three groups of exceptions which are detailed below, with the reason for their explicit exclusion from this construction:

a) The layout characters tab and newline; these are excluded because of the difficulties of placing them correctly in RTL/2 text.

b) #, £ and $; manufacturers treat these characters in differing ways (we have already said that they are interchangeable in RTL/2) and so we cannot guarantee an invariant value.

c) The double quote symbol ''; the reason for not allowing '"' will be seen below.

Given these restrictions, character handling situations can now be programmed relatively efficiently and with clarity in the RTL/2 text.

Variables to hold byte values may be declared, following the familiar rules applicable to reals, integers and fractions, by the use of the keyword BYTE. Hence we can have byte variables, references to byte variables, arrays of bytes and so on, and can declare appropriate variables in data, locally in procedures and inner blocks, as parameters, and specify byte results. The mechanism of assignment is identical, an integer constant being a valid byte value if and only if it lies in [0, 255] as described above.

```
LET SP=' ';     % SPACE CHARACTER %

DATA MESSAGE;
   BYTE B1:='A',B2,B3;
   REF BYTE RB:=B2;
   ARRAY (6) BYTE FLAGS:=(0,1,2,3(3));
   ARRAY (7) BYTE ALARM:=('F','A','I','L','U','R','E');
ENDDATA;
```

```
PROC WRITE (REF ARRAY BYTE TEXT) INT;
INT L:=LENGTH TEXT;
   FOR I:=1 TO L DO
      OUTPUT(TEXT(I));
      % ASSUME THAT OUTPUT IS A PROCEDURE TO OUTPUT A SINGLE %
      %CHARACTER, DEFINED BY  PROC OUTPUT(BYTE B);  %
   REP;
   RETURN(L);    % RESULT IS NUMBER OF CHARACTERS OUTPUT %
ENDPROC;

PROC MAIN ();
BYTE X;
       .
       .
       X:=';';
       WRITE(ALARM);
       .
       .
ENDPROC;
```

The example shows the initialisation of two byte arrays, the first using familiar integer constants, the second employing character constants to build up a message. This is a tedious way to have to write it. RTL/2 provides a more compact form for the initialisation of a byte array known as a *string.* As its name implies this is a string of characters and is simply a shorthand for the bracketed list of character constants separated by commas. The characters which can occur legally between single quotes (in a character constant) are called *stringchars.* A string is simply a sequence of stringchars enclosed in double quotes; we can now see why " is not an allowed stringchar – if it were, we would not be able to distinguish between " standing as a character of a string and the " terminating the string. We can rewrite our initialisation as follows:

```
ARRAY(7) BYTE ALARM := "FAILURE";
```

Now no brackets are required; the initialisation is delimited by the *string quotes* ("). We must still ensure that our initialisation matches the length of the array – there must be precisely the correct number of stringchars.

A string is an item, but it can contain spaces; this is because the space is a valid stringchar and is standing as a significant thing and not as a layout aid in the text.

```
"THIS IS A VALID STRING"   %CONTAINS 22 CHARACTERS%
```

For a long message or piece of text, we may not be able to get our string on to one line in the program. We cannot insert a newline since it is not a valid stringchar; also we may not want a newline character as part of the string. This problem is overcome by concatenating adjacent strings; in this context, adjacent means that the only characters separating the strings are layout characters (tabs, newlines or spaces); naturally, such characters are not regarded as part of the total string.

```
"THIS IS A LONG STRING "
"SPREAD OVER TWO LINES"
```

Here only spaces and newlines separate the two strings, and so they will be concatenated and treated as the single item

```
"THIS IS A LONG STRING SPREAD OVER TWO LINES"
```

This can be extended to concatenate successive strings on many lines provided they are 'adjacent'.

In the following example, a non-layout item (in this case a comment) separates the two strings which will therefore be treated as being distinct.

```
"PART ONE"                %SECTION HEADING%
"CHAPTER ONE"
```

The exclusion of newline characters from strings has the advantage (as in the case of comments) that if the closing " is inadvertently omitted, masses of program will not be swallowed as part

of the string! A compiler error message will be generated as soon as a newline character is encountered.

In realistic text handling and other applications we do need to be able to get newline characters and other non-stringchars into the strings themselves when they are to be used as messages. This is achieved by a facility inside a string which temporarily reverts to the normal syntax of initialisation. This inner sequence is enclosed between a pair of # characters. As usual, $ or £ may also be used, but the characters must occur in matched pairs. As the # character is not a stringchar, there is no ambiguity caused by its use, and, of course, it does not contribute to the length of the string. Within the sequence, values must be in the range 0 to 255 and be separated by commas; we may use repetition factors, and names defined by LET definitions will be replaced – this facility is recommended for clarity. As we are now operating as if in an ordinary array initialisation, comments and the layout characters space and tab are allowed, but the newline is not, because we are still fundamentally within a string and we wish to minimise the dangers of swallowing program text as described above.

```
LET NL=10;          % NEWLINE CHARACTER %
LET POUND=36;       % VALUE OF £ IN THIS IMPLEMENTATION %
        .
        .
    ARRAY (14) BYTE INC:="#NL#SALARY IS : #POUND#";
```

will generate an array INC of the form

INC	14	10	'S'	'A'	'L'	'A'	'R'	'Y'	' '	'I'	'S'	' '	' : '	' '	36

and on output it will appear on a newline as

SALARY IS : £

This method can also be used to introduce control characters into a string:

```
LET ENQ=5;      % ENQUIRE %

    ARRAY (12) BYTE ASK:="WHAT NEXT ?#ENQ  % TERMINATE WITH ENQUIRE % #";
```

and our earlier examples could have been written

```
ARRAY (6) BYTE FLAGS:="#0,1,2,3(3)#";
ARRAY (7) BYTE ALARM:="#'F','A','I','L','U','R','E'#";
```

though these forms are not so clear.

Note that there is no reverse facility

ARRAY(7) BYTE ALARM:= ('F', "AILU", 'R', 'E') is illegal. When concatenation is involved we must terminate the string correctly with ", and this means terminating any inner sequence first:

```
"#NL#CLOSE#NL(2),ENQ#"
"#NL#OPEN AGAIN"
```

Finally on the general syntax of strings, note that strings must occur explicitly for concatenation to occur; that is, if one string is present implicitly via a LET definition, concatenation will not occur.

```
LET STRING="DOUBLE";
        .
        .
        STRING "STRING"
        .
        .
```

will be treated as two strings – "DOUBLE" followed by "STRING".

Strings can also be used to initialise sub-arrays in a multi-dimensional array declaration; note here that the outer brackets are required for the outer levels of the array, and that the subarrays can be presented as a mixture of strings and bracketed lists.

```
ARRAY (4,7) BYTE MIX:=
   (  "FAILURE",
      (0(3),255(4)),
      "#NL#GO ?#NL,ENQ#",
      "END    "     );
   % LAST STRING PADDED WITH 4 SPACE CHARACTERS FOR LENGTH OF 7 %
```

In most cases strings will be used for messages and the contents of an array so initialised will remain unchanged throughout the run of a program. With this consideration, they are in some sense "constants". It is also useful to be able to write a message explicitly at its point of use rather than the name of an array which is initialised elsewhere. A further use is therefore defined for a string; it may be used as a *literal* array and be assigned to a ref-array-byte variable. In this sense it behaves as the name of an array of bytes; however it is not permitted to subscript it directly; thus "STRING"(2) is illegal. Such strings are allocated storage in a *pool* and have a length and an internal (compiler generated) name. Identical strings *of this form* in a program will probably share the same storage, and hence should not be altered by program, but used in a read-only manner.

```
DATA S;
   ARRAY (6) BYTE FLAG:="ALLSET";
      % THIS STRING IS SHORTHAND FOR INITIALISATION %
   REF ARRAY BYTE RAB:="FAILURE";
      % THIS IS A LITERAL STRING; RAB IS INITIALISED TO CONTAIN %
      % A REFERENCE TO IT %
ENDDATA;

PROC ACTION ();
REF ARRAY BYTE R1:=FLAG,      % STANDARD ASSIGNMENT %
               R2:="FAILURE";
               % STRING IS A LITERAL; STORAGE SHARED WITH EARLIER %
               % OCCURRENCE %
      .
      .
   R1:="NOGO";     % NEW ASSIGNMENT USES LITERAL STRING %
   R2(3):='A';     % OVERWRITES STRING IN LITERAL POOL %
                   % NOT RECOMMENDED %
      .
      .
   WRITE("FAILURE");
   % PROC WRITE (REF ARRAY BYTE TEXT);  OUTPUTS A STRING OF CHARACTERS
   % AT COMPILE TIME THIS LITERAL STRING WILL BE SHARED WITH THE %
   %EARLIER OCCURRENCES BUT AT RUN-TIME THIS STRING HAS BEEN CORRUPTED
   % AND THE STRING FAALURE WILL BE OUTPUT : DONT OVERWRITE STRINGS %
      .
ENDPROC;
```

Assignment to a ref-array-byte parameter is likely to be the major use of literal strings. Their internal syntax is identical to that discussed above and so we can make initialisations of the form

```
REF ARRAY BYTE Q:="#0(2),NL,255(7)#";
% BUT Q:=(0(2),NL,255(7)) IS NOT LEGAL %
```

We can also declare an array of such references:

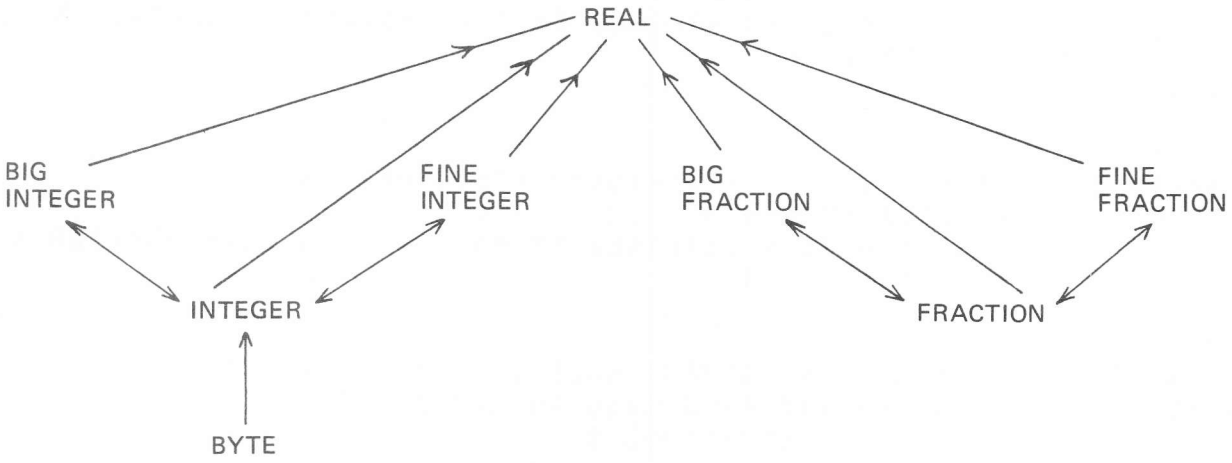
```
ARRAY (5) REF ARRAY BYTE MESSAGE:=
   (  "VALVE MALFUNCTION",
      "TEMPERATURE OVERLOAD",
      "START UP",
      "",        % NULL STRING OF ZERO LENGTH %
      "CALL OPERATOR"   );
```

Note that the strings (literal) are of different lengths, and we have declared effectively a *ragged* structure. In fact we generate an array containing five internal names of five strings entered in the literal pool. Note the difference between this one dimensional array, and the explicit two dimensional array MIX above which had to be rectangular (and hence all the strings or subarrays had to be of the same length) and in which the individual characters of the strings were filled into elements of the array.

A typical procedure to output a message would be PROC TWRT (REF ARRAY BYTE TEXT); and a call of this TWRT (MESSAGE(3)) would case START UP to be output.

We now have to consider the operations which can be performed on or between byte values, and the way in which they interact with the other modes. Bytes are very economic in terms of storage space and are mainly intended for handling characters and simple logical situations; they are not designed for arithmetic use and extensive use of bytes in arithmetic may be inefficient. Hence for "small integer" work, bytes should be used for static storage (especially when a large number of them is needed) whereas local integers are appropriate for the calculations involved.

For mode transfers, a byte value is automatically widened to integer where necessary; the full automatic mode transfer diagram is thus:



The keyword BYTE also serves as a monadic operator for narrowing a real or an integer value (a fraction value would first be widened to a real, but this case is unlikely to arise in practice); the action is to mask the integer value to obtain the bottom eight bits (and hence a value in [0,255]); any real value is first converted to integer by the usual rounding process. Overflow may occur at the stage real to integer, but no overflow will be indicated at the masking stage: this operation is equivalent to working modulo 256. As usual, an identity operation is included for completeness.

OPERATOR	OPERAND	RESULT	INTERPRETATION
BYTE	Byte	Byte	Identity: no action
	Integer	Byte	Mask to byte value
	Real	Byte	Round to integer and mask to byte

BYTE	72	is	72	BYTE	0.1	is	0
BYTE	293	is	37	BYTE	275.6	is	20
BYTE	−17	is	239	BYTE	−3.2	is	253

The operator REAL is also defined for byte values to enable type changes to be made at appropriate points.

OPERATOR	OPERAND	RESULT	INTERPRETATION
REAL	Byte	Real	Float operand to real number

The other monadic operators defined for byte operands are the usual ones +,–, and ABS. Naturally, + and ABS perform no action (since the value is guaranteed positive) and are included for completeness. The act of negating a value in the range [0,255] is bound to produce a result lying outside this range; negating a byte is therefore defined to yield an integer value, the action being the usual negation

OPERATOR	OPERAND	RESULT	INTERPRETATION
+	Byte	Byte	Identity: no action
–	Byte	Integer	Negate to give integer value
ABS	Byte	Byte	No action

Entries for byte operands occurring in the tabular description of monadic operators occur first; otherwise automatic mode transfers would be invoked. Note the absence of NOT; a byte operand will be widened to an integer. This is to ensure efficient implementation.

The only dyadic operators defined between byte operands are the logical operators LAND, LOR and NEV. They act in the same bitwise fashion as for integers on 8 bit byte patterns (naturally producing as result a further 8 bit byte pattern) and the complete tabular description for these operators can now be given:

OPERAND	PRECEDENCE	OPERAND 1	OPERAND 2	RESULT	INTERPRETATION
LAND	4	Byte Integer	Byte Integer	Byte Integer	Bitwise logical and
LOR	3	Byte Integer	Byte Integer	Byte Integer	Bitwise logical or
NEV	2	Byte Integer	Byte Integer	Byte Integer	Bitwise exclusive or

We have now completed all the cases for monadic and dyadic operators.

Note that what may be regarded as omissions are covered by the mode transfer rules detailed earlier; the sum of two byte variables, for instance, is evaluated as an integer involving widening at run time. This is why arithmetic between bytes may be inefficient and should be programmed using integers. If we want to put the sum back into a byte, we must include an explicit narrowing operation.

```
BYTE A,B,C;
INT DIGIT;
    A:=BYTE(B+C);
    % NOTE THE USE IN CHARACTER HANDLING OF SIMILAR EXPRESSIONS %
    % TO CONVERT A DIGIT INTO ITS CHARACTER VALUE - THIS RELIES %
    % ON THE SEQUENTIAL ORDERING OF ISO7 CHARACTER VALUES %
    A:=BYTE(DIGIT + '0');    % DIGIT 0 TO 9 INTO CHAR '0' TO '9' %
```

We may use byte values in comparisons. The notions of equality and inequality are well defined, and =, # (or its variants) may be used between bytes. The other comparisons are also well defined logically, but some hardware restricts their efficient implementation. For this reason, the use of <, <=, > and >= between bytes will result in an integer comparison (this is defined by saying that mode conversion occurs as for subtraction); this in no way affects the truth values of assertions involving these comparators.

EXAMPLE 1:
We could implement a multi-length arithmetic capability in a machine independent fashion by using bytes. Each number is decomposed into a number of bytes (the bytes placed together forming the total binary pattern of the number) held in an array. This is equivalent to working in a number system with base 256. A number in array A can be thought of as

$$A(1)*256^0 + A(2)*256^1 + \ldots + A(N)*256^{N-1}$$

We give a possible version of a procedure for adding together two such positive 'numbers'. The reader is left to devise a scheme for coping with signs and writing further arithmetic routines.

```
PROC ADD (REF ARRAY BYTE A,B);
   % PERFORMS A:=A+B IN MULTI-LENGTH ARITHMETIC %
   %  LENGTH A = LENGTH B ASSUMED %
INT CARRY:=0;
   FOR I:=1 TO LENGTH A DO
   INT SUM:=A(I) + B(I) + CARRY;
      A(I):= BYTE SUM;      % EQUIVALENT TO SUM MOD 256 %
      CARRY:=SUM SRL 8;     % EQUIVALENT TO SUM :/  256 %
   REP;
   IF CARRY#0 THEN WARN("OVERFLOW");
   END;
   % WARN IS A PROCEDURE TO OUTPUT A MESSAGE %
ENDPROC;
```

EXAMPLE 2:
Assuming a PROC OUT(BYTE B) which outputs a simple ISO7 character, the following procedure will print out an integer preceded by an appropriate sign. The procedure is recursive but does not allow for the possibility of overflow – it will not print out the most negative number for the machine's word length.

```
PROC INTEGERPRINT (INT N);
   IF N<0 THEN
      N:=-N;    % OVERFLOW POSSIBLE %
      OUT('-');
   ELSE
      OUT('+');
   END;
   IF N>9 THEN INTEGERPRINT(N:/10);  END;
   OUT(BYTE(N MOD 10 + '0'));
ENDPROC;
```