RTL/2
Training manual

RTL/2 Training manual

RTL/2 Reference : 3
Version : 1

Contents

1. Introduction

The history of RTL/2, the criteria employed during its design and the benefits which can be expected from its use are outlined in the manual "Introduction to RTL/2".

Programming a computer to perform any task is basically a process of translating from the language of human thought and expression to one which is convenient to the machine. The 9th century Arabian mathematician al-Kuwarizmi is credited with originating the method of symbol manipulation which lies at the root of abstract mathematics and logic. His name is remembered in 'algorithm', something which symbolically represents a sequence of decisions and calculations to perform some well-defined process. RTL/2 is basically an algorithmic language in which we can define symbolic procedures in a style similar to mathematics and English, without worrying about the basic instructions or organisation of a particular computer. The translation of this into a form suitable for execution by a machine is performed by a program - a *compiler.* For this to be possible, the language must be defined in a precise, unambiguous manner. There are two aspects to this. Firstly, there are the rules of grammatical structure (*syntax*) that must be obeyed before a construction can be regarded as admissible; secondly, there are other (*semantic*) rules that govern its use and interpretation in practice and attach a meaning to a construction in its particular context. A formal presentation of these rules is given in the RTL/2 Specification Manual, which is intended as a reference manual and not a source for learning RTL/2. This Training Manual, however, is intended to provide a sequential text to teach RTL/2, not only its rules, but also its philosophy, underlying structures and some of the ways in which it can be used.

The stages involved in writing and running programs from the formulation of the problem to the existence of a dependable tool are well known. In this Manual we are concerned with the process of formulating and writing the logical sequence which will solve the problem. This process leads to a program which is presented to an RTL/2 compiler through some suitable input medium. This Manual is about the whole RTL/2 language and not just the bits which are simple to learn and use; an overall understanding is essential if the maximum benefit of RTL/2 is to be gained in the design and writing of programs. There are many things which this Manual is not about: it does not concern itself with actual machines or operating systems; it does not describe what compilers may do; above all, it is not a handbook to enable existing programs to be transliterated into RTL/2.

Text books usually have prefaces and introductions which explain for whom the work is intended, the level of knowledge assumed and how the book is organised; this Manual is no exception and the following remarks will probably be ignored completely or dismissed as padding. Knowledge of any other language is not assumed, and the average intelligent person who has brushed with computers and has a reasonable level of mathematics should find little difficulty. The style is designed to be readable without complex technical definitions and a mass of numbered paragraphs bristling with decimal points. The automatonous reader is asked to excuse the occasional glimpses of humanity which have crept into an otherwise turgid subject. The experienced programmer will proceed rapidly. No sections are starred as being "of less importance" or "can be omitted at a first reading"; the aim is to present the whole language in a fairly logical order; if you find difficulty, follow the French mathematician's advice to the conscientious not to linger too long over the hard parts but 'go on, and faith will come to you' — it will all fall into place. Finally, the usual exhortation to try all the examples; they (and the worked examples) are essential — the test of understanding is usage. Answers are provided in Appendix 3 but these do not purport to be 'best' solutions; they merely solve the problems with the constructions available at that point.

One aim of any high-level language is to bring the potential of computers within reach of a wider group of users, including those who regard such machines as mere tools and who are not interested particularly in the hardware structure involved. It is intended that RTL/2 can be used to program processes independent of the actual computer used. This makes precise definition impossible in some areas; other vague areas may arise where the interface between RTL/2 programs and the particular operating system is involved. We recognise these limitations and indicate them by the phrases "machine dependent", "implementation dependent" and "system dependent"; whenever possible a minimum interpretation expected or the spirit intended is given, but the reader is advised to consult the documents relevant to the particular computer or system when he finds himself in such areas: it is clearly good programming practice, when designing transportable software, to collect any 'fuzzy' areas into one place and pay particular attention to documenting the assumptions made.

Regardless of high-level languages, computers will only do what they have been told to do. Whatever the claims for computers, it is true that they will faithfully reproduce in their output all the errors of the input, and this applies equally to the logical structure of the program as well as to any numerical data. RTL/2 and its compilers cannot check your logic though they may help you – you have been warned.

2. Places and Objects

Executing a program in a computer consists of the manipulation of numerically coded instructions and items of numerical data, held in the store of the machine. We can picture this store as a collection of cells. Since there is a large number of different cells where information and instructions can be located, some means must be provided to identify each one uniquely, so that a particular instruction can be specified or a particular item of data accessed. This is solved by giving each cell an identification number called its *address*; thus each cell can be thought of as having a nameplate bearing a unique reference number. It is important to note that this nameplate contains no information about the nature of what is in the cell (i.e. instruction or datum) nor about the contents itself.

In a high-level language we are not interested in which particular cells of the machine we use. We do wish to manipulate information in the cells and to be able to refer uniquely to a cell (or a group of cells) wherever it is physically situated in the store.

We shall see how we do this in RTL/2 and distinguish between the nameplate and the contents of the cell in a simple case.

The most primitive objects that we wish to manipulate in RTL/2 are numbers; some of the numbers involved are *real numbers.* Inside a machine they are represented in some floating-point form. As far as the program writer is concerned, he can think of them in a form similar to the common scientific notation, in which a real number is regarded as a fraction between 0.1 and 1.0 times a power of ten. For example 276.23 can be written 0.27623 x 10^3. In any given machine numbers can only be held with a certain finite number of significant figures. The majority of numbers will therefore be approximated to some degree of precision. Since the number of significant figures held varies from machine to machine, the language does not define the *accuracy* with which real numbers will be held: that is the accuracy is implementation dependent. Similarly, because machines are finite, there will be a limit to the magnitude of real numbers which can be held within the machine. It will not be possible to hold very large nor very small numbers. The *range* of numbers which can be held is again implementation dependent. A typical implementation on a small computer might allow a range of 10^{-20} to 10^{+20} with 6 figure accuracy.

How do we write a real number in RTL/2? We use a combination of the decimal digits 0 to 9, the letter E and the decimal point (.). RTL/2 allows three forms for a real number and leaves the compiler to construct the form required internally:

i) A simple decimal number which contains a decimal point and at least one digit both before and after the point;

ii) A simple decimal number as in (i) with a power of ten (*exponent*) appended in the form E followed by a signed or unsigned number (without any point);

iii) A decimal number without a point with an exponent as described in (ii).

You may wonder how negative numbers can be dealt with if we do not prefix a sign to the number. Any leading sign is not regarded as a part of the number, but as an operator acting on the number; this will be dealt with in Section 5. Note too that such a number is an RTL/2 item (see Section 3) which is terminated as soon as a character is found which cannot be interpreted as part of it; thus we cannot have any spaces in the number particularly between the decimal and exponent parts.

The following are examples of legal real numbers written in the different permissible forms:

i)	263.27	0.5	13.0
ii)	2.6327E+2	5.0E−1	1.3E1
iii)	26327E−2	5E−1	13E0

The following examples show illegal attempts to write real numbers:

16.	no digit following decimal point
.02	no digit preceding decimal point
E6	no number for the exponent to apply to
1.2E1.0	decimal point in exponent
1632.7 E−1	space precedes exponent

The compiler may also fail a syntactically valid real number if its size exceeds the range for the machine. This is an example of a semantic error — the machine will not be able to attach a sensible meaning to the RTL/2 item.

A number appearing in a program in one of the above forms is called a *real constant.*

We will wish to manipulate such real numbers in our program, and will thus require to store them in the cells of our machine. Just as in the machine we identified each cell uniquely by an address, we wish to identify uniquely, within the RTL/2 program, the quantities we are going to manipulate. During the execution of a program (i.e. at run time) a quantity will take on many different numerical values, but, as in algebra, such a *variable* quantity has only one value at a particular instant. In our machine this is reflected by one cell which contains different real numbers as the program progresses. We wish to identify this particular cell. This is done using an *identifier.* Identifiers play a vital role in RTL/2; we use them to name various things, including variables. To cope with identifiers we introduce the RTL/2 item *name*. A name is a sequence of the letters A to Z and the digits 0–9 (these characters are known as alphanumerics) with the proviso that the first character must be a letter. Thus the following are valid RTL/2 names:

```
A
XY27
FRED
INCOME
WHATALONGNAMEIAMWRITING
```

Whilst the following sequences are illegal as names:

A*B	contains a non-alphanumeric
3PQ	does not start with a letter
FRED BLOGGS	contains a non-alphanumeric (space)

The language RTL/2 does not place any limit on the number of characters that may appear in a name. However, for practical reasons, there will be a finite limit. There is in any case a limit imposed by the number of characters you can get on a line! It is unwise to use names which are too long since writing them becomes tedious, the machine independence is possibly reduced (see Section 30), and the chance of making spelling mistakes increases!

So to identify our variables in a program we have a free choice of names; well, almost a free choice! The character set of RTL/2 does not provide enough distinct symbols for the use of the language and so some names are reserved and have a pre-defined meaning. There are 57 such *keywords* which are listed in Appendix 2; we shall explain their use as we introduce them individually through this manual.

We can now name the variable quantities we wish to use in our program. The selection of a particular name for a variable does not in any way affect the meaning of our program, in the sense that the meaning will not be changed if throughout our program we consistently replace it by another name (different from all the other names used in the program). In short programs, brief, uninspired names, like A, X27, FRED are all right, but in longer, more complicated cases, involving several hundred variables perhaps, it becomes important to use mnemonic names so that a variable's name suggests what it stands for. The use of such names (e.g. TEMP, PRESS, ERROR) simplifies the writing of a program, reduces the number of errors made, and makes it far more comprehensible to another reader (and to yourself some months after you first wrote it!)

We can now link together in RTL/2 the ideas of the contents of a machine cell and the nameplate attached to it. The name of a real variable is simply the nameplate of some cell which contains real numbers during the course of a program. We shall see that real numbers are not the only quantities we wish to manipulate. During compilation of an RTL/2 program a suitable cell (or set of cells) has to be reserved for every variable appearing in the program. In order that the compiler can allocate the appropriate storage the programmer must convey some information about the objects which will be contained in the cells named by the identifiers used in the program. This is achieved by a *declaration.* If we wish to use TEMP, PRESS, ERROR to name cells containing real quantities we write:

```
REAL TEMP,PRESS,ERROR
```

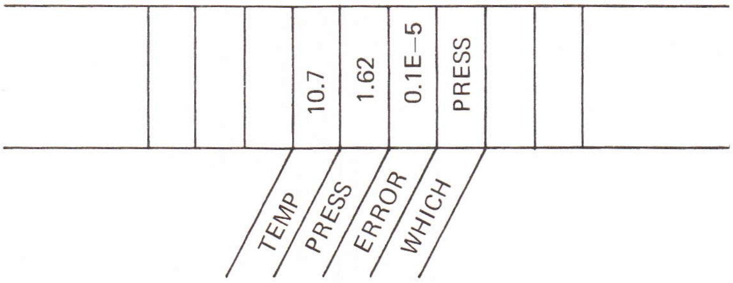
The exact position in the program where we put this will be discussed later. The form is the same for all declarations. We have a description of the *type* (i.e. the nature) of the contents (in this case the reserved word REAL) followed by a list of the names of the variables required, separated by commas. So we have our first keyword, REAL, and our first piece of RTL/2; the function of this declaration is simply to announce that the identifiers TEMP, PRESS, ERROR will be used in the subsequent text of the program to identify places which can contain real numbers. The importance of this seemingly pedantic insistence on the name "identifying a place which can contain" something rather than saying "the variable *is* a real number" will become apparent.

One of the objects we shall wish to manipulate is the name of a variable, or in general, an identifier. That is we wish the contents of some cell to be the "name" of some other place. We shall wish to name this cell too in our program, and, like all variable names in RTL/2, we must inform the compiler as to how it will be used by means of a declaration. The type this time is a reference to a real cell which is reflected in the declaration:

```
REF REAL WHICH
```

REF is another keyword and in this case we are only declaring one name. What have we done? We have announced that the name WHICH will be used to identify a place which can contain the name of a real variable, that is which can contain a name which in its turn identifies a place which can contain a real number. Thus the contents of a ref-real variable is a *pointer* to another cell. In practice the contents will probably be the address of the cell of that name; reference variables can thus be used for indirect addressing.

If at some stage of our program the values in cells TEMP, PRESS, ERROR are 10.7, 1.62, 0.1E–5 and WHICH happens to be pointing to PRESS, we can picture the machine cells in the following way:



Note that a declaration, as well as informing the compiler of the usage intended for a name also reminds us of the range of values which a variable can contain. Real variables, once declared can contain real numbers, the form of which we have already examined; we do not need to declare real constants since no cells are involved and both we and the compiler can recognise them as such by the presence of a decimal point or the exponent symbol E (or of course both). In other words the constant is unambiguously real. For variables this is not true: the language supplies the structure, but the programmer provides the vocabulary, and the compiler must be informed how names are to be used.

The concentration on real numbers and floating point arithmetic in the early part of this Manual does not mean that they are regarded as most important; they are merely the simplest to deal with and enable the basic concepts of RTL/2 to be introduced without involving other unnecessary considerations at this stage.

Section 2 examples

1 Which of the following real constants are illegal and why?

a) 72.6
b) 0.3
c) 673.0E+1
d) 0001.34
e) 3B.7
f) 27.
g) 32,767
h) 0.00015
i) 100.001
j) .3
k) 27E010
l) 2.3E−2
m) 16E−9
n) 16 E9
o) 1.2*E−3
p) 0.9E+1.0
q) 0.06E10
r) 11.74E0
s) .125 E0
t) 27E2

2 Rewrite the legal real constants in example 1 in the form of a fraction in the range [0.1, 1.0) (i.e. a fraction f $0.1 \leqslant f < 1.0$) and an exponent

3 Which of the following identifier names are illegal and why?

a) FREQUENCY
b) A
c) RATE-OF-FLOW
d) FRED'S
e) 6H20
f) A7E2
g) #27
h) MESSAGE
i) REF
j) FEET/SEC
k) MUCHTOOLONG
l) NAME
m) 77E2
n) FISH&CHIPS
o) N204
p) REALDATA
q) A29461
r) TAX
s) ACCURACY
t) VALVE SETTING

4 Write suitable declarations for variables to be used in the evaluation of tax liability

3. Presentation

Ultimately a piece of program text must move from the planning and manuscript stage to a form which is suitable for input to a machine, and in particular for presentation to an RTL/2 compiler. This process can be performed on any data preparation equipment which produces a suitable input medium (e.g. punched cards, punched tape) for the particular machine on which the compiler is run. This input will be a sequence of RTL/2 characters, each one drawn from the "language subset" of ISO 7 – a character code proposed by the International Standards Organisation. On some machines, text may be presented in some other code (e.g. EBCDIC) and there will be a translation into ISO 7 before entering the compiler; at this stage only the characters concern us; their internal representation will become critical in later sections. We have already encountered some of the valid characters: the letters A–Z, the digits 0–9, the comma (,), and the signs +, –. Others will be seen in later sections and full details of the set are given in Appendix 1. It is perhaps worth pointing out that care is needed in distinguishing between the letters I, O and the digits 1, 0.

We mentioned in Section 2 that characters are grouped together to form *items*. So far we have seen two RTL/2 items, the real constant and the name. The former was characterised by the fact that it contained a decimal point or an exponent symbol E (or both), the latter by the fact that it started with a letter. Within the character sequence of the text, an item is terminated as soon as a character is reached which cannot be interpreted as part of the item; for example if a name item is started by the letter W, say, this item will be terminated as soon as a non-alphanumeric character is found. Often this will be a space character, an ISO 7 character not mentioned specifically before. The space in RTL/2 is a *layout character.* There are two other layout characters, the tab character (for horizontal tabulation: no settings are defined in the language) and the newline character (corresponding to an explicit line-feed, possibly with a carriage return, on paper tape and implicitly present as the end of a card). In general, any item will be terminated by a layout character. Apart from that, layout characters may be inserted freely into the program text. RTL/2 is a free format language within this item structure, and the judicious use of layout characters aids considerably the clarity and legibility of a piece of program. In particular the indentation of lines is a simple but effective way of indicating the structure of a program.

RTL/2 programs are meant to be to a large extent self documenting. A program may be annotated by the insertion of a *comment.* A comment is an item and can appear wherever an item can appear – this is virtually everywhere but note that you cannot have a comment in the middle of a name nor in the middle of a real constant! It consists of a per-cent symbol (%) followed by almost any sequence of characters forming the explanatory material and terminated by a further per-cent symbol. Since the per-cent symbol terminates the comment, it is obvious that the sequence cannot contain a per-cent symbol! The only other restriction is that the sequence may not contain a newline character. The reason for this is that if inadvertently the closing per-cent sign were omitted, the compiler would otherwise treat all the program text as a comment until it reached another per-cent sign, and this might lead to the (possibly disastrous) creation of a program with a large chunk missing. An example of a comment is:

```
% EXAMPLE OF COMMENT %
```

Such comments are completely ignored by the compiler, do not affect the meaning of the program, but are intended to help the reader of the program. As with mnemonic names, free use of the facility (with no penalty) is encouraged to make programs more comprehensible to others and to yourself some months after originally writing them. Much (possibly excessive!) use will be made of them in this Manual.

As an example, we will see how we might annotate a solution to examples 2 number 4:

```
REAL  INCOME,      % TOTAL EARNED INCOME IN POUNDS %
      NETTAXPAY,   % TAXABLE PAY AFTER PENSION DEDUCTIONS %
      ALLOWANCES,  % ALL PERSONAL ALLOWANCES BUT NOT EARNED %
                   % INCOME RELIEF %
      TAXCODE,     % CODE NUMBER IN TAX TABLES %
      TAX          % TAX PAYABLE %
% NOTE THE FREE USE OF LAYOUT CHARACTERS, AND THE COMMENTS %
% (LIKE THIS ONE) SPREAD ACROSS TWO LINES %
```

4. Assignment

We have seen how to declare the name of a place to contain a real number, and how to write real constants in RTL/2. During our program we will obviously wish to change the contents of various locations (cells) or, at least, to put some numbers in them! How do we do this? At execution time, a program tells the machine what operations to perform, that is, it specifies a series of actions to be taken. In an RTL/2 program, the actions to be performed are defined by *statements*. Note that the program which we write is not composed solely of actions; we have already seen the declaration, which supplies information to the computer and the human reader, and the comment, providing annotation.

Various declarations and statements are usually separated from one another by semi-colons. We regard the examples as being parts of some larger program and so append ';' to the last statement. This point will be discussed fully in a later section.

The statement in RTL/2 which changes the contents of locations is called the *assignment statement.* Suppose we have a variable named RATIO (declared as a real) and we wish to place the real number 0.7 in the place named RATIO. Then our declaration and statement appear as:

```
    REAL RATIO;
    RATIO := 0.7;
```

The assignment statement consists of three parts, the left hand side (the name RATIO), the symbols :=, and the right hand side (the real constant 0.7). Let us examine these constituents in turn.

The left hand side specifies the *destination;* that is it states the name of the place whose contents are to be changed. Clearly, the only sensible thing that can be allowed here *is* the name of a place. This change of contents is a replacement of the old contents by the new, and in the process, the old object is destroyed. The destination supplies us with another piece of information. As the name of a place, it must have been declared, and that declaration tells us the nature of the contents of that place: such an object must be delivered by the right hand side.

The sequence := ('colon equals' or 'becomes') is an item formed by the concatenation of the two characters ':' and '='; as an item, of course, it cannot have any layout characters in it. It behaves as a single entity, separates the left and right hand sides of the statement, and specifies the action of assignment. As such, it may be thought of as a replacement operator. Note that '=' alone is not used, since equality is not involved; the combination ':=' is used to emphasize the asymmetry of the action and to stress that an *operation* and not a relation is involved.

The right hand side supplies the object which is to be the new contents of the destination. In our example, the destination requires a real number, and the real constant 0.7 is clearly a valid object to be placed in RATIO. As we shall see later, the right hand side will often be a much more complex expression, but the principle will remain: the right hand side must deliver an object which can be stored in the destination specified by the left hand side.

So much for getting numbers into locations. At some stage we shall wish to use the contents of our various named locations (i.e. use our variables). Suppose we have another real variable in which we wish to remember the contents of RATIO prior to assigning a new value to the placė RATIO (which action would of course destoy the old value). What do we mean by:

```
    REAL RATIO, OLDRATIO;
    RATIO:=0.7;
    OLDRATIO:=RATIO;
```

The first two lines are identical to the last example except that we have now declared a second real variable OLDRATIO. What does the second assignment mean? The left hand side is the name of a place which is therefore a valid destination. Further it is the name of a place which contains a real number, so the right hand side must deliver a real number. On the right hand side, however, we have the name of a place again, namely RATIO. The only sensible interpretation which can result in a real object is that the contents of RATIO is required, and this is indeed the meaning. The act of extracting the contents of a named location is called *dereferencing*. Dereferencing on the right hand side of an assignment statement is automatic, and we do not need to insert any explicit notation in our program to perform this. Note that there is no change to the contents of the place named RATIO; we are merely 'reading' its contents. To be meaningful some sensible number must previously have been assigned to the place, as in our example.

It is very tedious to talk about "the contents of the place named" but this distinction between the name of a place and the object in that cell is vital. We shall talk about "the value of a variable" meaning "the object currently in the place named" and this naturally refers to the object most recently assigned to that location.

In Section 2 we introduced the idea of a ref-real variable, the contents of which is the name of a real variable, providing at any time, a pointer to a real variable. How will such variables fit into assignment statements?

```
REAL RATIO, OLDRATIO;
REF REAL WHICH;
    RATIO :=0.7;
OLDRATIO := RATIO;
   WHICH := OLDRATIO;
```

The left hand side of the third assignment statement is the name of a place and is therefore a valid destination. It is a place that can contain the name of a real variable, and so the right hand side must deliver such an object. The right hand side, OLDRATIO, is the name of a real variable. The action then is to place the name OLDRATIO in the cell named WHICH. What happens if we now add the statement:

```
RATIO := WHICH;          ?
```

We go through a similar process of reasoning: the left hand side is the name of a place which can contain a real number. The right hand side is the name of a place which contains the name of a real variable. This is not a suitable object, so we invoke dereferencing which yields the contents of WHICH which is the name of a real variable (in this case OLDRATIO). This is still not a suitable object, so we invoke dereferencing again, that is we take the contents of the contents of WHICH: the contents of WHICH is a real variable and the contents of that must be a real number (in this case the contents of OLDRATIO which happens to be 0.7). This real number is a suitable object and is assigned to the place RATIO. The dereferencing is performed automatically, and again there is no effect on the contents of WHICH nor on the contents of its contents (!) (that is the value of OLDRATIO in this case is not affected, it is merely "read"). This provides a facility of *indirect access* to the values of variables.

The reverse process must also be considered. Can we (and, if so, how) assign indirectly? Consider the statement

```
WHICH := 0.4;
```

The left hand side indicates that the name of a real variable is required; the right hand side is a real number which can in no way deliver the name of a real variable. The right hand side is incompatible and the statement is therefore illegal. If our intention was to place the number 0.4 in the location whose name was contained in WHICH (i.e. to assign indirectly, in this case to OLDRATIO), then we must arrange for the name of a real to be the left hand side. To do this, we use the keyword VAL which forces dereferencing on the left hand side.

```
VAL WHICH := 0.4;
```

This says that the destination is the contents of WHICH. Since WHICH is a ref-real variable its contents is the name of a real variable which is the destination; being a real variable, a real number is required, and the right hand side delivers such an object. VAL can only be used on the left hand side of an assignment statement, and this is the only situation in which dereferencing needs to be forced and hence mentioned explicitly in our program.

We have now seen real numbers, real variablés, ref-real variables and (on the left hand side only) ref-real variables preceded by VAL, in assignment statements. Let us enumerate the 12 possibilities, indicate which are legal, and the interpretations to be put upon them.

```
REAL X,Y;
REF REAL P,Q;
   X:=Y:=0.0;  P:=Q:=X;   % ENSURE SENSIBLE CONTENTS %

% IN THESE STATEMENTS THE LEFT HAND SIDES ARE NOT NAMES OF PLACES %
   1.6:=2.4;   % ILLEGAL %
   1.6:=X;     % ILLEGAL %
   1.6:=P;     % ILLEGAL %

% REAL OBJECTS ARE SUPPLIED TO A REAL VARIABLE IN : %
    Y :=2.4;
    Y :=X;     % X DEREFERENCED %
    Y :=P;     % P DEREFERENCED TWICE %

% Q IS A REF-REAL VARIABLE AND REQUIRES THE NAME OF A REAL VARIABLE %
    Q :=2.4;   % ILLEGAL %
    Q :=X;
    Q :=P;     % P DEREFERENCED %

% THE FOLLOWING ARE SIMILAR TO THE ASSIGNMENTS TO Y EXCEPT THAT %
% THE DESTINATION IS OBTAINED BY DEREFERENCING Q %
 VAL Q:=2.4;
 VAL Q:=X;     % X DEREFERENCED %
 VAL Q:=P;     % P DEREFERENCED TWICE %
```

It is extremely important to distinguish

```
        Q:=X;
    VAL Q:=X;
```

which specify quite different actions.

In the above, we have placed each statement on a separate line. This is not essential bearing in mind the free-format nature of RTL/2. As noted in Section 3 the presentation should be designed to be as clear as possible for the reader.

If we wish to assign the same object to more than one location, we can do this in one statement, a *multiple assignment.* Since we are assigning one object, all the destinations must be places which require that kind of object. We merely add the name and a ":=" item for each variable to the left hand side thus:

```
    X := Y := Z := 1.3;        % X,Y,Z REAL VARIABLES %
```

Note that RATIO := WHICH := 0.1; is illegal since RATIO demands a real whilst WHICH demands the name of a real variable; dereferencing of WHICH is not automatic, since it is here part of a left hand side. Indirect assignment would require:

```
    RATIO := VAL WHICH := 0.1;
```

The action of this statement is to set up the real object 0.1, find the contents of WHICH and overwrite the contents of that real variable with the object 0.1, and replace the value of RATIO with 0.1. Note the order of performing the actions from right to left; this will be mentioned again later.

Let us trace the actions of a set of assignment statements:

```
REAL A,B,C;
REF REAL P,Q,R;
    A:=1.0E-6;   B:=62.8;   C:=3E7;
    P:=B;   Q:=C;   R:=A;
    % ALL VARIABLES NOW CONTAIN SENSIBLE VALUES %
    VAL Q:=A:=C;        % 1 %
    VAL P:=1.6;         % 2 %
    B:=R;               % 3 %
    C:=0.7;             % 4 %
    R:=B;               % 5 %
    VAL R:=C;           % 6 %
    A:=B;               % 7 %
```

If we number the statements as shown we can draw up a table showing the values of the variables after each assignment:

VARIABLE STAT	A	B	C	P	Q	R
initial	1.0E–6	62.8	3E7	B	C	A
1	3E7	62.8	3E7	B	C	A
2	3E7	1.6	3E7	B	C	A
3	3E7	3E7	3E7	B	C	A
4	3E7	3E7	0.7	B	C	A
5	3E7	3E7	0.7	B	C	B
6	3E7	0.7	0.7	B	C	B
7	0.7	0.7	0.7	B	C	B

You are invited to check this: any step can be verified by considering the nature of the destination.

Section 4 examples

1 Draw up a table to show the result of performing the following assignment statements:

```
REAL W,X,Y,Z;
REF REAL A,B,C,D;
    W:=X:=0.0;
    Y:=Z:=1.0;
    A:=B:=Y;
         D:=Z;
    VAL B:=X;
    Z:=C;
    VAL D:=3.2;
    C:=W;
    B:=D;
    Y:=C;
    X:=Z;
    W:=0.3;
    VAL B:=C;
```

One assignment statement is meaningless; which one and why?

5. Monadic operators

The assignment statement enables us to move values around the machine, but hardly gives us any scope to manipulate those values. As mentioned in the last section, the right hand side will normally consist of a more complex *expression.* An expression is simply a rule or formula for computing an object; as before, the destination determines the nature of this object; and hence dereferencing may be needed during the evaluation of an expression.

We introduced the statement to express a required action to be carried out in our RTL/2 program. The actions required in the evaluation of an expression are specified by *operators.* In this section we are concerned with *monadic operators*; such operators act on one object to produce another object. In the RTL/2 text, a monadic operator precedes the object it is acting on. We need to know four pieces of information about such operators:

i) the symbol used for the operator
ii) the nature of the object on which it is acting – the *type* or *mode of the operand*
iii) the nature of the object produced – the *mode of the result*
iv) what action will actually be performed.

Let us consider three simple monadic operators in RTL/2:

OPERATOR	OPERAND	RESULT	INTERPRETATION
+	Real	Real	No change : identity
–	Real	Real	Negate the operand
ABS	Real	Real	Negate the operand if it is negative, otherwise no change

ABS is a reserved word, and must be terminated by a non-alphanumeric (see Section3); thus we write ABS X and not ABSX which would be interpreted as a name

Thus these three operators all act on real objects to produce new real objects, and perform the simple algebraic functions of identity, negation and absolute value.

Example:

```
REAL X,Y;
REF REAL WHICH;
      WHICH := Y;                     % 1 %
            Y := +0.3;                % 2 %
            X := –26.2E2;             % 3 %
VAL WHICH := ABS X;                   % 4 %
```

The four assignment statements will be interpreted as follows:

1 the familiar assignment of the name Y to the location WHICH
2 the value 0.3 is assigned to the location Y, the plus sign merely emphasizes the positivity.
3 the negative value –26.2E2 is assigned to the location X.
4 the left hand side is the name of a real, so a real object is required; ABS acts on a real, so the contents of X are accessed (i.e. X is dereferenced), the absolute value taken, and this real value stored. With the current values, this will result in 26.2E2 being assigned to Y. There is no effect on the contents of X.

Note that the signs preceding the constants are regarded as operators and not part of the constant (mentioned in Section 2).

There is no reason why the left hand and right hand sides of an assignment statement should not contain the same name. Thus:

```
REAL X;
X := 3.2
X := –X;
```

has the effect of storing 3.2 in X and then negating the contents of X. Note carefully the distinction between the left hand side where X is the name of a destination and the right hand side where the contents of X are accessed since a real object is required. The reason for the use of ':=' is now seen more strongly; X=−X would be algebraically misleading! We can prefix more than one operator if we wish, but we must then be sure in what order the actions will be performed. What do we mean if we write

```
REAL X, Y;
Y := 1.65;
X := −ABS Y;          ?
```

The first assignment statement should be clear by now. The second has X as destination which therefore demands a real object. The minus sign also requires a real object; ABS cannot be interpreted as a real quantity but ABS Y can. So we take the name Y, dereference it to yield a real value, perform the operation ABS which yields a new real value, which becomes the operand for −. Thus −1.65 will be stored in X. Another way of expressing this rule is to say that monadic operators are applied from right to left, i.e. the "innermost" operations are performed first.

An expression consisting of some monadic operators (or none!) and an object is called a *term* in RTL/2. We shall see how to combine terms together in later sections, and also meet new monadic operators as we progress through this Manual.

Section 5 examples

1 Write down the values of the variables X and Y after each of the following assignment statements.

```
REAL X,Y;
    X:=26.3;
    Y:=-0.27;
    X:=ABS-X;
    Y:=+ABS-Y;
    X:=+-Y;
    Y:=-+X;
    Y:=-ABS-2.3;
    X:=+0.01;
    X:=-ABS+Y;
```

2 Rewrite Example 1 with new declarations and first assignment

```
REAL X,P;
REF REAL Y;
    Y:=P;
```

Simplify the monadic operators to remove redundant operations, and remove redundant assignment statements, but do not use the name P nor replace names by current constant values!

6. Dyadic operators

In order to construct more complex expressions, we need to combine terms together. Operators will again be required to define precisely what action is to be performed. Such operators defining how two terms are to be combined are called *dyadic operators.* As with monadic operators, there are items of information we need to know, namely the symbol used for the operator, the modes of the two operands, the mode of the result, the interpretation, and a further item which will be explained below.

In this section we introduce four dyadic operators which provide simple addition, subtraction, multiplication and division of real numbers. We will present these and subsequent operators in tabular form:

OPERATOR	FIRST OPERAND	SECOND OPERAND	RESULT	INTERPRETATION
+	Real	Real	Real	Form the sum of the operands
–	Real	Real	Real	Subtract the second operand from the first
*	Real	Real	Real	Form the product of the operands
/	Real	Real	Real	Divide the first operand by the second

The operators are written in the RTL/2 text between the operands just as in mathematics, with the left operand naturally being regarded as the first. Thus we write

```
REAL A;
A := A*2.54;
```

to scale the contents of A from inches into centimetres. There are a number of important observations to be made.

1 The result of any of the above operations may lie outside the range of real numbers which can be held on the particular machine on which the program is running. This condition is termed *overflow*. For example, if the range of real numbers is 10^{-20} to 10^{+20} and we attempt any of the following operations (between valid real numbers), overflow will occur:

```
9.9E19 + 9.7E19
1.2E9 * 1.3E11
1.001E17 / 0.001
```

The action taken on encountering overflow is machine dependent; strictly speaking the behaviour of the program will be undefined when such a condition arises, but the intention is that any implementation will supply some means (not necessarily automatic) of detecting overflow.

2 The multiplication sign must always be supplied explicitly; the mere juxtaposition of two variables as in mathematical notation does not imply multiplication. Nor may the point '.' be used as a 'times' operator.

3 Division is an operation performed as accurately as possible (i.e. the same degree of precision as that mentioned in Section 2) and there is no question of a remainder.

The result of applying one of the above dyadic operators is a further real object which may itself be used as an operand. A new problem arises when we have more than one dyadic operator in an expression. Let A, B, C be declared as real variables. Then if we write A+B+C we clearly mean to add together the three real numbers in the locations A, B, C. However, if we write A+B*C what do we mean? We could mean add together the contents of A and B and then multiply by the contents of C or, alternatively, we might require the contents of A to be added to the product of the contents of B and C. In algebra, this problem is quite familiar and is resolved by requiring multiplications to be performed first; if a product is to be distributed over a sum, brackets are inserted to indicate this. Thus a+bc is our latter interpretation; whilst for the former we would write (a+b)c. The same approach is followed in RTL/2. If we write

```
REAL A,B,C,D;
A := 2.3
B := 1.6;
C := 0.1;
D := A+B*C;
D := (A+B)*C;
```

the result of the fourth assignment is to place 2.46 into the location D whilst the fifth results in 0.39 becoming the value of D. A number of simple rules govern the way in which an expression will be evaluated in RTL/2.

Firstly, any expressions in round brackets are evaluated first. It is permissible to have bracketed expressions within brackets thus (A+(B+C)+D)*C (known as *nested* brackets) but note that there is only one sort of bracket. In such cases the rule is to evaluate innermost brackets first.

Secondly, every dyadic operator has a number associated with it (this is the extra piece of information referred to earlier) known as its *precedence*. This is a measure of its priority of performance in an expression and reduces the number of brackets needed to ensure that an expression is evaluated in the way intended. The rule we shall give is applicable to all dyadic operators in RTL/2 and not just to the four introduced between real quantities in this section. Suppose we have three terms ∝, β, γ, two operators □ and ○ and we write the expression ∝□β○γ. If the precedence of □ is greater or equal to the precedence of ○ then the expression will be evaluated as (∝□β)○γ (i.e. will proceed simply from left to right); otherwise it will be ∝□(β○γ). Note that in particular, when □ and ○ are the same, they will have the same precedence and so the expression 0.6—0.1—0.5 will result in 0.0.

These rules do not state that ∝□β will be evaluated from left to right; this point will be discussed in Sections 8 and 14.

The fact that bracketed expressions are evaluated first means that their presence overrides the precedence considerations which would otherwise be invoked. Brackets can be safely used, when the precedence of two operators has been forgotten, to ensure the correct interpretation, and it is sometimes useful to insert redundant brackets to aid legibility and to reinforce the precedence in the reader's mind. The general moral is not to hesitate to insert brackets at all points where one is unsure of the priorities or where they increase legibility. Any pair of redundant brackets will not affect the meaning of the expression. The function of brackets (no matter how complex the enclosed expression may be) is to shield the contents from the rest of the expression; from outside, the bracketed expression can be regarded as a simple term.

We now re-present our four dyadic operators, showing their precedences:

OPERATOR	PRECEDENCE	FIRST OPERAND	SECOND OPERAND	RESULT	INTERPRETATION
+	1	Real	Real	Real	Form the sum of the operands
—	1	Real	Real	Real	Subtract the second operand from the first
*	5	Real	Real	Real	Form the product of the operands
/	5	Real	Real	Real	Divide the first operand by the second

From this we can see that addition and subtraction have equal precedence as have multiplication and division and that multiplication and division have the higher precedence as in algebra. This last fact is sometimes expressed in the form that multiplication and division are "more tightly binding".

Example:

```
% VALUE OF EXPRESSION SHOWN IN COMMENT %
REAL A,B,C;
A := 27.0/9.0;                  %3.0%
B := 13.2*1.1+6.7;              %21.22%
C := 1.0–1.0/2.5;               %0.6%
B := B*B–A*C;                   %448.4884%
C := B/(2.0*A);                 %74.748067%
```

Of course, in the last two assignments, implicit dereferencing has occurred. Note that there is no 'power' operator so B^2 must be written out as a multiplication; precedence demands that the products B*B and A*C be performed before the subtraction. In the last assignment, the brackets are essential for the evaluation of B/2A; B/2*A would mean (B/2)*A since / and * have equal precedence.

Dyadic operators may also be combined with monadic operators in expressions. The rule for precedence here is embodied in the fact that the operands of a dyadic operator are terms and any monadic operators are part of a term. Thus all monadic operators have a higher precedence than the dyadic operators. This precedence again can be overridden by the use of brackets.

Thus –(A–B) is equivalent to –A+B
whereas –A–B is equivalent to –A+–B

Note that A–B is the same as A+–B; it is an algebraic fact that +,– can be used either as monadic or dyadic operators (they are the only such ones).

It is unnecessary to labour further the algebraic interpretation of such operators; the important lessons from this section are the concepts of bracketing and precedence which will apply to less familiar operators which we will encounter.

Example:

Write a sequence of assignment statements to store in A,B,C the values of

$8.3+1.7y-0.9y^2+3.0y^3+1.0/x$

for the values of x stored in P,Q,R at y = 12.4

```
REAL A,B,C,P,Q,R,Y,AUX;
    Y:=12.4;
    AUX:=8.3 + Y*(1.7 + Y*(-0.9 + 3.0*Y));
% NOTE THE USE OF AN AUXILIARY VARIABLE TO CALCULATE PART OF    %
% THE EXPRESSION ONLY ONCE, AND THE USE OF FACTORISATION AND    %
% BRACKETS TO REDUCE THE NUMBER OF MULTIPLICATIONS              %
    A:=AUX + 1.0/P;
    B:=AUX + 1.0/Q;
    C:=AUX + 1.0/R;
```

Section 6 examples

1 Declaring suitably chosen identifier names, write simple assignment statements to:

i) calculate the percentage error between two readings of an instrument
ii) convert temperatures expressed in the Fahrenheit scale to centigrade
iii) calculate simple interest
iv) evaluate $(1+x)^5$

2 Construct a table showing the values of the variables between each of the following assignment statements:

```
REAL X,Y,X1,X2,Y1,Y2,YMEAN,SQUARES;
    X1:=1.0;  X2:=2.0;
    Y1:=X1*3.0 + 4.0;
    Y2:=X2*3.0 + 4.0;
    X:=1.6;
    Y:=Y1 + (Y2-Y1) / (X2-X1) * (X-X1);
    YMEAN:=(Y + Y1 + Y2) / 3.0;
    Y :=ABS(YMEAN - Y );
    Y1:=ABS(YMEAN - Y1);
    Y2:=ABS(YMEAN - Y2);
    SQUARES:=Y*Y + Y1*Y1 + Y2*Y2;
```

7. Function calls

In Section 6 we saw the same calculation repeated for a number of different variables, and in example 2 we had

```
    ∝ := ABS(YMEAN−∝)
```

for ∝ = Y,Y1 and Y2. It often happens in a program that a relatively simple (or complex!) piece of calculation recurs. In elementary mathematics, when we wish to use a trigonometrical ratio, we do not have to draw a triangle each time and measure this ratio, we use a set of tables where this information has been recorded. For similar values in our program, to store sets of tables in our computer would take up valuable space. There are ways of calculating such values, but we do not wish to have to write out some complex algorithm every time we want to use it. In RTL/2 we define a function name to represent the sequence of operations which must be performed to give us the desired value. At each point at which we require to perform the calculation, we simply write this function name as a shorthand. This use of functions not only saves programming time, but also conserves computer storage space and results in considerable advantages in the organisation of a program, at a slight cost in execution time.

Some languages provide certain functions (such as trigonometric functions, square root) as part of the definition of the language. RTL/2 has no such built-in functions, but a particular RTL/2 system may offer standard facilities, or a more general mathematical package. We shall restrict our attention for the moment to functions which, given a real number, supply a real result – the trigonometric functions, square root, logarithm all fall into this class. We are considering the *call* of the function here, that is, its use and not its definition – we are concerned with putting money into a slot machine and getting some goods in return, and not with the mechanical workings of the machine.

In giving us one real value from another, such a function behaves exactly as a monadic operator; a function call is just a complicated primary constituent of an expression which, with any monadic operators which may be applied to it, behaves as a term (and can thus be the operand of a dyadic operator). The real object we give to the function is a *parameter* or *argument*; the real object used in our expression is the *result*. To insert a function call at the appropriate point we write the function name and the real object which we wish to be the parameter within brackets. Thus if a function SQRT is defined which finds the positive square root of the given parameter, then

```
    X := SQRT(81.0);
```

will result in 9.0 being stored in the location X. The brackets here enclose the parameter and are obligatory; they have no connection with the grouping of terms or precedence.

We have said that the parameter should be a real object; this does not mean that it has to be a real constant. Any expression can be written as the parameter as long as it delivers a real object; the parameter behaves as if it were the right-hand side of an assignment statement whose destination is a real variable. Thus:

```
    REAL X,Y;
    REF REAL REFX;
    Y := SQRT(X);
    Y := SQRT(REFX);
```

are possible; in the first assignment, X will be dereferenced once to yield a real, in the second, REFX will be dereferenced twice. In particular the expression may itself contain a function call:

```
    X := SQRT(SQRT(81.0) );
```

The parameter of SQRT is a real namely SQRT(81.0); i.e. the result (9.0) returned from the inner SQRT is used as the parameter and hence 3.0 will be stored in location X.

Assuming that the trigonometric functions SIN, COS for sine and cosine of a parameter expressed in radians, and LOG for the natural logarithm of a positive real have been defined in addition to SQRT, we now show some examples of function calls used in general expressions:

```
         SIN2X := 2.0*SIN(X)*COS(X);
    QUADROOT := (−B+SQRT (B*B−4.0*A*C) )/(2.0*A);
             D := LOG(ABS( (1+SIN(X) )/COS(X) ) );
```

We have been discussing the call of a function; clearly a definition must exist somewhere, and, for each call, the computer must obey a set of instructions to find the result. It would defeat the object of saving space if the code for this were inserted at each calling point! Hence, the coding for the evaluation of the function is elsewhere and there is a change of sequence at the point of call. This is our first example of the execution of a program not being strictly sequential as defined by the text. A function call is quite complex; at the point of call, there is a (temporary) change of sequence (albeit an implicit one beyond the control of the writer). The point of call must be remembered, the function evaluated elsewhere, and the result returned to the calling point so that the complete expression can be calculated. The form of the definition will be dealt with in the next section and the question 'where is the code?' answered at a later stage.

Up to now we have restricted our thoughts to functions which have a real parameter and a real result and our examples have all been taken from elementary mathematics. These will normally be supplied by someone else and the functions we will write will be specifically for the particular problem we are trying to solve. Sometimes we will wish to work with more than one parameter. For example, a program may wish to calculate compound interest at various points; we could define a function with name COMPOUND which would require three parameters, namely the principal, the rate of interest and the time and a typical call would appear as:

```
    INTEREST := COMPOUND(11250.0,RATE,25.0);
```

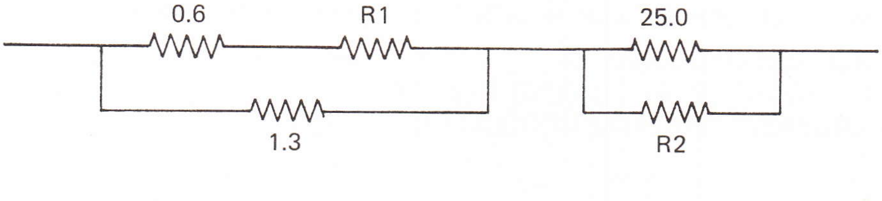
The three parameters are supplied as a list between the brackets, separated by commas; the order of presentation of the parameters must correspond to that of the definition, otherwise there will be some unexpected results!

Methods of obtaining more than one result, and the use of parameters requiring other than real objects will be discussed later.

Section 7 examples

1 Devise some useful functions, and illustrate their use in assignment statements.

2 The functions SERIES and PARA each have two real parameters and return as a real result the effective resistance of two resistances joined respectively in series and in parallel, all the quantities being expressed in ohms. Write assignment statements to evaluate the effective resistance of the following network:



8. Procedure bricks and data bricks

We saw in the last section how to use a function call in an expression. At the time of writing a system or in our individual programs we wish to write, in RTL/2, the coding that produces the result from the parameters supplied in our function calls; that is, we wish to be able to define a function and specify exactly how it works. Such a definition in RTL/2 is known as a *procedure brick.* A procedure is an object (like a real!) which happens to be a *process* rather than a numerical constant. Moreover, it is an executable process, that is, it is a piece of program, a set of instructions associated with a name which will be obeyed every time that name is used as a function call.

Let us write a function to evaluate y = $3x^2-7$ for a given value of x, and then explain the various parts of the definition.

```
PROC Y(REAL X) REAL;
RETURN(X*X*3.0–7.0);
ENDPROC
```

i) PROC introduces the definition and is a natural abbreviation for procedure; it is a keyword and the abbreviation is compulsory – PROCEDURE will be taken as an ordinary name!

ii) This is followed (after suitable spacing to terminate the keyword PROC) by the name of the function; this associates that name with the following coding. The name is formed in the usual way.

iii) An obligatory left bracket opens a *parameter specification list.*

iv) In this case we have one parameter. The 'REAL X' looks very much like the form we have used previously in declarations; this is indeed a declaration. This is our first example of where a declaration can actually occur. It says that X is to be a location which can contain a real object. It has two other properties: it is *local* to this procedure brick, that is it only exists when this procedure's coding is being obeyed (this will be dealt with in Section 9) and it is a parameter which will contain a definite value specified in each call of the function. Declaring parameters enables us to manipulate the particular arguments supplied at the function call; a parameter can be thought of as an open space in the procedure which is filled in by the procedure call. This will be investigated more fully later.

v) The right bracket terminates the parameter list.

vi) REAL then specifies that this function will deliver a real object as its result; this tells us that whenever the function is called, it behaves similarly to a real variable being dereferenced to yield a real value.

vii) The semi-colon is obligatory, and terminates the *procedure heading* which defines fully the nature and specification of the procedure brick.

viii) There follows the *body* of the procedure; in this case it is a simple statement (which will be analysed below), but in general it will be a sequence of statements which process the parameters to evaluate the required result.

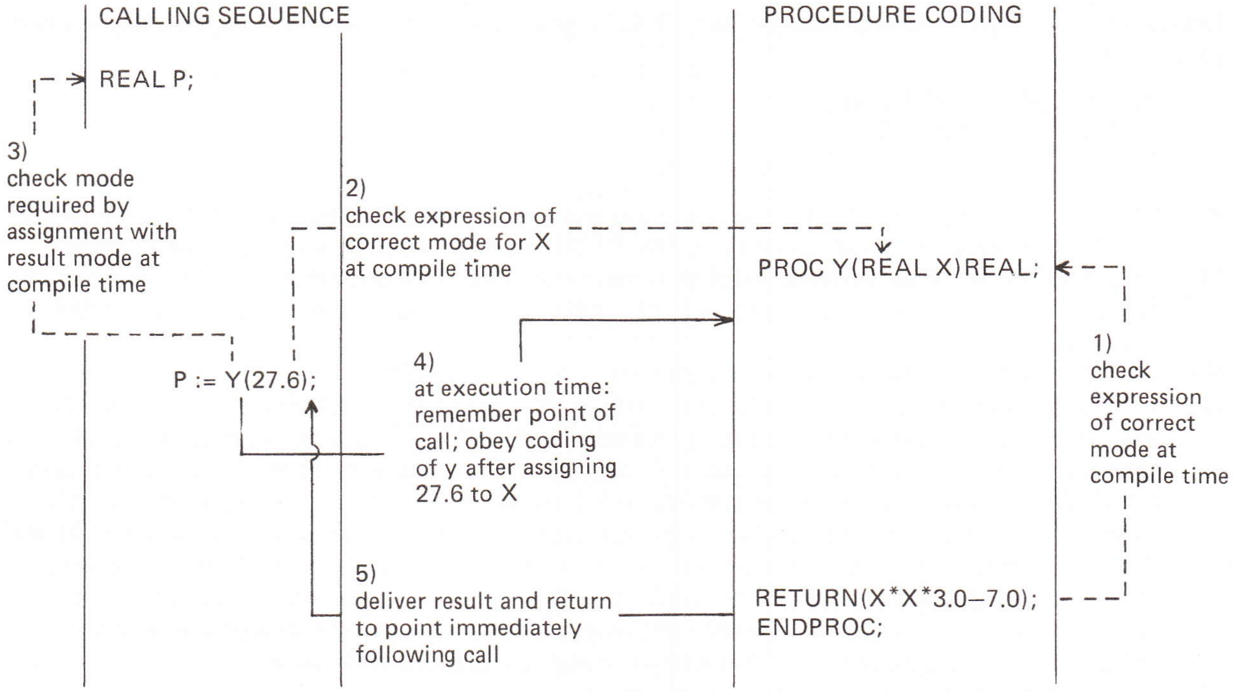
ix) Finally, the brick is terminated by ENDPROC, another keyword which matches PROC and informs us that the definition is complete. If definitions of further procedures follow, then a semi-colon is used as a separator between these bricks.

The body of the procedure is the guts of the structure, the actual code which does the work. In this case it consists of a single statement. This introduces a new statement type, the *return statement.* It performs two distinct actions. Firstly it sets up the result which is to be delivered by the function call and secondly it causes an explicit change of sequence by returning to the point of the function call (strictly, the point immediately following the function call.) The syntax is straightforward; a return statement consists of the keyword RETURN followed by an expression enclosed in brackets. Note, once again, that these brackets are required by the syntax and have no connection whatsoever with any brackets that may occur in the expression. Naturally, if the specification says that a real object will be returned as the result, the expression must yield such a result, and dereferencing will be applied as if the expression were to be assigned to a real variable.'In our example, the statement merely calculates the expression $3x^2-7$ from the value supplied in the parameter X. As a further point, the specification of a result implies the rule that there must be a RETURN somewhere in the body of the procedure to yield a result.

The concept of matching the object with the specification also applies to the parameters. The parameter mechanism is that of assignment and each call must present as parameters a correctly ordered list of expressions which deliver objects suitable for assignment to the parameter locations specified in the procedure heading.

Finally, not only must the expression in the return statement be of the correct type to match the specification, this result must also be valid in the context of the function call.

We can picture the coding of a procedure brick and a function call, the checks made at compile time and the actions taken at run time for our procedure Y as follows:



Consider the following definition of a procedure:

```
PROC FOURTHPOWER (REAL X) REAL;
   X:=X*X;          % REDUCE MULTIPLICATIONS %
   RETURN(X*X);
ENDPROC;
```

This first squares the parameter and then squares the result to deliver a fourth power. Suppose we have the calling sequence:

```
REAL P, P4;
.
.        % DOTS INDICATE FURTHER STATEMENTS %
.
P:=0.3;
.
.
P4:=FOURTHPOWER(P);
.
```

At the call of the function, a real object is required for X and so P is dereferenced and 0.3 assigned to X; the body of the procedure is then obeyed which immediately means that the value of X is replaced by its square 0.09. *There is no effect on P.* The assignment to X only involves a dereferencing of X, a multiplication and the actual assignment; where the original 0.3 came from is irrelevant and unknown to FOURTHPOWER.

Our next example shows how to write a function definition having more than one parameter, and also introduces a non-real parameter.

```
PROC CIRCLE (REAL DIAM, REF REAL WHERE) REAL;
   VAL WHERE:=3.14159 * DIAM;
               % SETS UP  CIRCUMFERENCE %
   RETURN(0.25 * 3.14159 * DIAM * DIAM);
               % AND RETURNS THE AREA %
ENDPROC;
```

Our parameter list consists of two declarations; note that they are separated in the parameter list by a comma and not by a semi-colon as in our earlier examples. A call of CIRCLE will require two expressions as actual parameters, the first of which must deliver a real object suitable for assignment to DIAM, the second the name of a real variable since this is the only valid object to assign to the ref-real variable WHERE. What happens on our call?

```
REAL AREA,CIRCUM;
   AREA:=CIRCLE(3.2,CIRCUM);
```

The parameter assignments and execution of the body will proceed as if:

```
DIAM:=3.2;      % PARAMETER ASSIGNMENT %
WHERE:=CIRCUM; %  DITTO %
CIRCUM:=3.14159 * 3.2;
   % THIS IS THE ACTION OF THE STATEMENT VAL WHERE:= .. %
   % WHERE IS DEREFERENCED TO GIVE THE LOCATION CIRCUM %
   % FOLLOWED BY AN ASSIGNMENT OF A REAL OBJECT : %
   % THUS AN EFFECT IS MADE OUTSIDE THE BODY %
AREA:=0.25 * 3.14159 * 3.2 * 3.2;
   % THIS IS THE EFFECT OF DEREFERENCING DIAM AND %
   % RETURNING THE RESULT OF THE FUNCTION CIRCLE %
```

The use of ref-real parameters thus enables us to manipulate the contents of variables outside the procedure, and secondly gives us a method of returning more than one result – we have here set up both the area and circumference by one function call.

Note that we could make the body of the procedure more efficient by writing RETURN(0.25*WHERE*DIAM) to reduce the number of multiplications: the use of WHERE in a real expression will cause it to be dereferenced twice, yielding firstly the name of a real location (CIRCUM) and then the real value in that location (in this case the value of 3.14159*DIAM which therefore gives us the correct formula).

To labour the point, a call such as CIRCLE(3.2,1.6) would clearly be illegal; we are attempting to assign a real object (1.6) to a ref-real variable.

Care must be taken over the use of ref-real variables to return multiple results; suppose we wish to use CIRCLE to evaluate the surface area of a cylinder whose diameter and height are known, and we write:

```
REAL SURFACE,CIRCUM,HEIGHT;
   HEIGHT:=2.7;
   .
   .
   SURFACE:=2.0*CIRCLE(3.2,CIRCUM)+CIRCUM*HEIGHT;
```

We mentioned in Section 6 that the order of evaluation of the operands of a dyadic operator had not been defined; no order is defined by the language; in this case, if the term CIRCUM*HEIGHT were evaluated first, we would calculate the wrong answer, since CIRCUM would not yet have been set by the call of CIRCLE and its contents would be indeterminate. The distinction between the lexicographic order of presentation (i.e. the straightforward sequential order of the RTL/2 text) and the dynamic order used at execution time should not be forgotten, and the use of *side-effects* (such as returning results through ref-real parameters) of function calls should be used in an unambiguous manner. We can ensure the correct answer in the above by splitting the calculation of SURFACE into two separate calculations, using SURFACE itself to hold the intermediate term:

```
SURFACE:=2.0*CIRCLE(3.2,CIRCUM);

SURFACE:=SURFACE + CIRCUM*HEIGHT;
```

Since statements *are* performed sequentially, CIRCUM is guaranteed to have been set up in the second assignment statement by the function call in the first.

How do we write the parameter list in the procedure heading when we have more than one parameter of the same type? We can write a separate declaration for each, but normally we will simply put a list of names (separated as usual by commas) after the type description. As an example, here is the definition of a procedure which returns the average of three real numbers supplied as parameters:

```
PROC AVERAGE (REAL A,B,C) REAL;
   RETURN( (A+B+C)/3.0 );
ENDPROC;
```

Note that we may also have ref-real results. The heading in such a case will be

PROC SOMETHING (REAL A) REF REAL;

The result will be the name of a real (which can be dereferenced in an expression or assigned to a ref-real variable) and hence the expression in the return-statement must deliver such an object. It is not possible to give a sensible example at this stage.

We mentioned in Section 2 that all variable names in RTL/2 have to be declared, and we have just seen where the declaration information is written in the case of parameters. The names of parameters were described as being local to that particular procedure brick.

If local parameters only exist whilst the coding of their procedure is being obeyed, we must clearly have some other kind (or kinds) of variable that can exist throughout the life of a complete program and which contain the items of information in which we are interested. In a more complex situation, such variables may need to exist for use by a number of "programs". Such variablcs are *global* in nature, and the structure which contains their declaration is known as a *data brick*; it consists merely of a named set of declarations and eventually corresponds with a named area of the machine's store. This area exists throughout the life of the program, and the variables therein are *static* — that is, they are always available to the program and for each variable there is a unique location in store.

The syntax of a data brick is the keyword DATA, followed by the name-of the data brick and a semi-colon which is obligatory. This name identifies the whole structure and may be chosen freely in the usual way. The 'body' of the brick consists solely of declarations as described in Section 2, the names of a list being separated by commas and declarations by semi-colons. The brick is terminated by the matching keyword ENDDATA and, as usual, is separated from any following bricks by a semi-colon.

Example:

```
DATA ATTRIBUTES;
   REAL     TEMP,PRESSURE,DENSITY;
   REF REAL WHERE,WHICHREAL;
ENDDATA;
```

Global variables declared in a data brick are available throughout the life of a program; this means that they can occur in statements. We have seen that we can write statements in procedure bricks. Where in general do our statements go?

We extend the concept of a procedure brick being the definition of a function to the more general idea of a procedure brick being a process, that is, a series of statements which define a sequence of actions to be performed, and require that all statements belong to some procedure brick. In particular, procedure bricks will be used to manipulate data brick variables. Calling a procedure brick is a way of temporarily breaking the sequential flow of control through the program to execute some pre-determined actions and then resume. These actions may require information from the main stream (i.e. parameters) and/or return information to it (i.e. results).

Calls of such procedure bricks are themselves statements and therefore belong to some procedure brick; our "program" then will consist of one basic procedure which has a sequence of operations, including calls of subservient procedures. A moment's thought will result in the problem that this basic procedure must itself be called if its instructions are to be obeyed which implies a statement which implies . . . This difficulty of an "uncaused first cause" is in the area of the interface between a program and the system.

The definition of a procedure brick is identical to that already described, except that now the specification of parameters and the specification of the result may not be present. Thus the following are all valid procedure bricks (the bodies have not been written out explicitly but implied by <body>):

```
PROC ACTION ();
   % NO PARAMETERS BUT BRACKETS STILL REQUIRED %
   % NO RESULT %

   < BODY >

ENDPROC;


PROC ROOT2 () REAL;
   % NO PARAMETERS BUT A RESULT %
   % ALWAYS RETURNS ROOT OF CONTENTS OF GLOBAL REAL, SAY %

   < BODY >

ENDPROC;


PROC SETPOINTS (REAL X);
   % NO RESULT BUT A PARAMETER %
   % SETS UP CERTAIN GLOBAL DATA PERHAPS %

   < BODY >

ENDPROC;


PROC CIRCLE (REAL DIAM, REF REAL WHERE) REAL;
   % PARAMETERS AND RESULT AS BEFORE %

   < BODY >

ENDPROC;
```

How do we call these and what is the mechanism? Our earlier function calls and definitions were merely special cases of the more general procedures we have just described. The parameter mechanism is exactly the same, and the syntax of the call is identical, the name of the procedure followed by a parameter list enclosed in brackets. In the case of there being no parameters, the brackets are still required. There are some differences of course; a procedure without any result cannot be used in an expression. A call of a procedure which merely performs a sequence of actions is a statement on its own called (of course!) a *procedure statement.*

Hence, to call the procedures ACTION and SETPOINTS above we would write (in some suitable procedure) the statements:

```
SETPOINTS(0.0);
ACTION();
```

Note that our procedure ROOT2 has no parameters but does return a real result so that it can be used in an expression:

```
AREA:=CIRCLE(ROOT2(), CIRCUM);
    % ROOT2 DELIVERS A REAL - A VALID PARAMETER FOR CIRCLE %
    % NOTE THE NULL PARAMETER LIST FOR ROOT2 %
```

Finally, since functions *are* procedures, we can invoke them by a procedure statement; this will result in the action: defined by the procedure being performed, but in the absence of any expression, the result is simply lost. This is not necessarily ludicrous; we may wish to call the procedure because of its side-effects — in particular it may amend global data variables or return other results through parameters.

In the above, if we merely wished to set up the circumference of our circle we would write the statement:

```
    CIRCLE(26.0,CIRCUM);
```

the area would be lost, but CIRCUM would be set up correctly.

Within the body of a procedure, if it has no result, there is clearly no obligation to provide a return-statement; indeed we cannot provide one in the form described so far, since there is no valid expression to insert within the brackets. How then, during the execution of the coding of the procedure, do we know when to return to the calling sequence? There are two ways:

i) The sequence is completed by encountering the terminating ENDPROC
ii) The use of a return-statement which does not return any result; the syntax of this statement is simply the keyword RETURN. No brackets are required, but as usual it must be separated from any following statements by a semi-colon. Since at present we have no way of changing the sequence of statements to be obeyed, this construction is unnecessary at this point; it is included now for completeness.

Local variables in the form of parameters have been declared in procedure headings. We may also wish to have other local variables, that is variables which are only used in a particular procedure brick for instance to hold temporary or intermediate information. We can declare such variables (using the normal syntax rules) between the procedure heading and the first statement of the body. To illustrate this we show a procedure which swaps the values in two given real locations; note the way ref-real variables are used to indicate which global variables are to be operated on by the procedure.

```
DATA GLOBALREALS;
    REAL A,B,C,D;
ENDDATA;

PROC SWAP (REF REAL X,Y);
    REAL TEMPORARY;          % LOCAL DECLARATION %
        TEMPORARY:=X;
        VAL X:=Y;
        VAL Y:=TEMPORARY;
ENDPROC;

PROC MAIN ();
    % CONTAINS MAIN SEQUENCE OF ACTIONS %
    .
    .
    SWAP(A,B);
    SWAP(C,D);       % ILLUSTRATES CALLS OF SWAP %
    .
    .
ENDPROC;
```

The reader is encouraged at this point to work carefully through the actions performed at the calls and to test his understanding of the dereferencing involved.

RTL/2 is a procedure-orientated language and the concept is fundamental to its structure. It not only provides a convenient and efficient method of programming (for instance, changes in logic are concentrated in one definition and not repeated at each point of use) but also a mechanism for effectively extending the language:

i) As we have already seen, standard mathematical functions do not need to be defined as part of the language but can be implemented simply as procedures.

ii) Operations which are similar but whose details necessarily vary from machine to machine can be implemented as procedures: the specifications can be standard (i.e. same number of parameters, same result type and same name) with bodies differing from implementation to implementation. Programs containing calls are independent of the machine on which we wish to run them: this is one way of defining input/output and real-time operations.

iii) The body of a procedure may be written in a different language, particularly machine code, because RTL/2 does not provide the required facilities. However, calls can still be made in the normal way and the procedure can be documented in RTL/2 style, thus preserving legibility.

Section 8 examples

All variables used should be declared in suitable bricks.

1 Write the procedures SERIES and PARA used in examples 7 no. 2. (The effective resistance, r, of two resistances r_1, r_2 is given by: $r = r_1 + r_2$ when in series

$\frac{1}{r} = \frac{1}{r_1} + \frac{1}{r_2}$ when in parallel).

2 Write a procedure to return the value of (1 + X) to the fourth power for a given value of X.

3 Write a procedure to return the maximum of two given real numbers.

4 Write a procedure to permute the values of the global real variables p, q, r to the order q, r, p.

5 Repeat No. 4 for three global variables supplied as parameters.

6 Write suitable data and procedure bricks to evaluate:

$y = px^4 + qx^2 + r$ for
p, q, r = 1.0, 2.0, 3.0 at x = −0.5, −10.2, 673.7
and p, q, r = 2.5, −3.5, 1.0 at x = 0.1, 7.2, 12.6

7 Write procedures to set up the roots of a quadratic equation

i) with the coefficients passed as parameters;
ii) with the coefficients in a global data brick.

(Assume that a PROC SQRT(REAL X)REAL to return the root of a positive real has been defined).

9. The run-time stack

The 'local' aspect of local variables is not simply a lexicographical one. Local variables only exist during the execution of the body of the procedure as the result of a procedure call; thus they are *dynamic* in nature – compare this with the static properties of data brick variables which are always available. We now consider how this is achieved, where local variables are held in the machine and how the parameter and procedure call mechanism works.

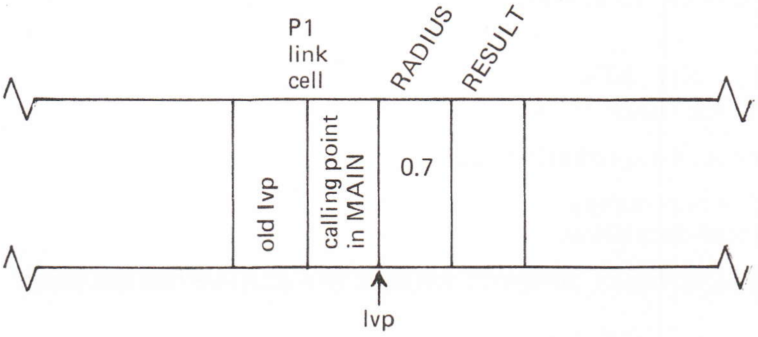
At execution time, each program has a *run-time stack*, an area of core which provides space for dynamic variables and certain other *housekeeping* items. The area of this stack actually in use expands and contracts with each procedure call and return. As each procedure is called, an incarnation of its local variables is made. At any one time there may be several 'levels of locals' due to procedure A calling procedure B which in its turn calls procedure C and so on. However we are only interested in the locals of the procedure which is currently being obeyed. For this we have the concept of a *local variable pointer* which effectively tells us where the current locals are in the stack. On a procedure call, we must remember the current value of the local variable pointer and also the point of the call in the current procedure (the *return link*); this information is held in the stack in a *link cell.* The local variable pointer is then set to point at the new local variables and the called procedure is obeyed (i.e. the required change in sequence is made). On return, clearly we have to reset the local variable pointer, and return to the point remembered in the link cell. In this way, local variables only exist during the execution of their procedure brick and only one set of locals is accessible directly at any point in time.

In the example which follows, the layout of the stack is purely illustrative and the position of the lvp conceptual. The exact layout and use will vary between implementations, but the underlying principles will not.

Consider the following piece of program:

```
PROC CIRCLE (REAL DIAM, REF REAL WHERE) REAL;
        .
        % AS BEFORE %
        .
ENDPROC;

PROC P2 (REAL DIAM) REAL;
REAL CIRCUM,AREA;
        AREA:=CIRCLE(DIAM,CIRCUM);                  % 3 %
        RETURN(AREA*0.5);                           % 4 %
ENDPROC;

PROC P1 ();
REAL RADIUS,RESULT;
        .
        RADIUS:=0.7;                                % 1 %
        .
        RESULT:=P2(RADIUS*2.0);                     % 2 %
        .
ENDPROC;
```

At point 1 the stack layout will be of the form:

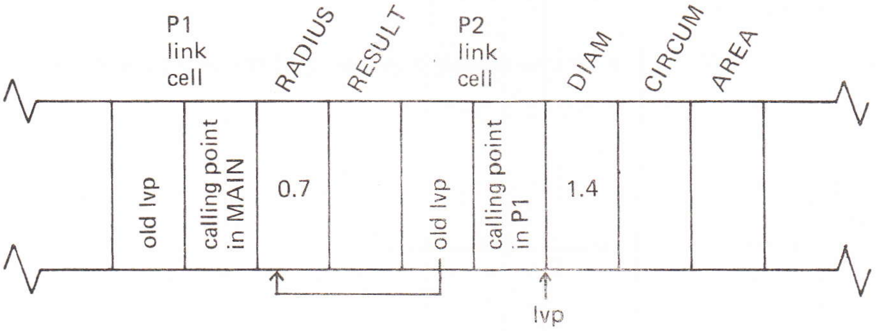


Here we have assumed that P1 was called from some procedure called MAIN.

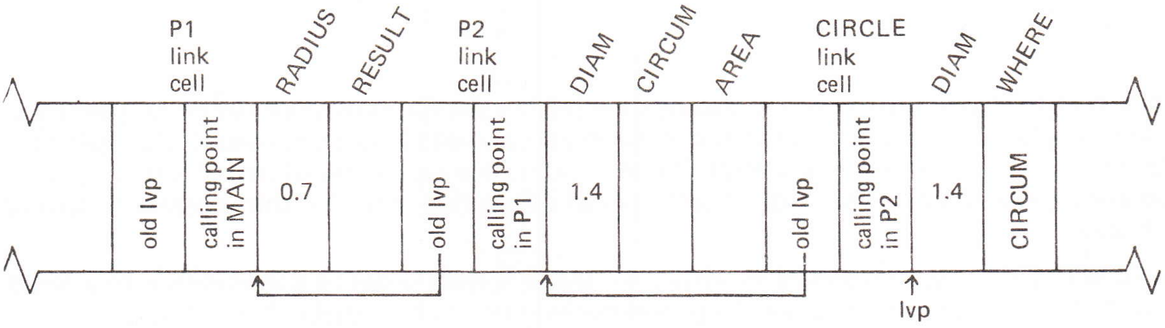
Statement 2 will now invoke procedure P2; this involves the following (any housekeeping checks/actions are excluded):

i) Creating a new link cell for P2
ii) Setting up the parameters in their correct locations
iii) Remembering the current value of the local variable pointer (lvp) in the new link cell – note that this also enables us to remember the position in the stack of the current link cell.
iv) Remember the position in P1 in the link cell.
v) Set lvp to point at the new local variables (those of P2).
vi) Execute the body of P2

As vi) is commenced the stack layout will be:



Statement 3 is now obeyed which involves a call of CIRCLE, and the same process causes the stack layout to become:

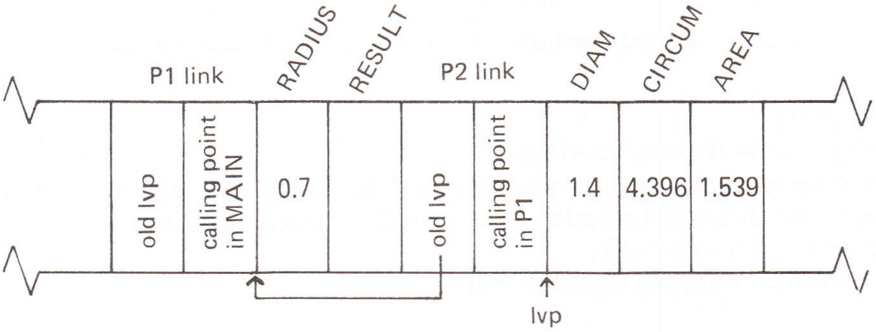


The body of CIRCLE will now be executed, during which CIRCUM in P2 will be set up – this may seem to contradict the fact that only the current locals are accessible, but this is not so; CIRCUM is set up indirectly as the result of dereferencing and the deliberate passing of its name as a parameter, but not by an explicit assignment. Note here that the existence of the name DIAM in both P2 and CIRCLE causes no ambiguity – only one of them is accessible at this point dynamically (via the lvp to CIRCLE) and lexicographically since within CIRCLE we cannot use the DIAM declared to be local to P2 (the multiple declaration of names will be discussed later). CIRCLE will then return a result; this involves the reverse process:

i) Return to the previous link cell in the stack by resetting the local variable pointer.

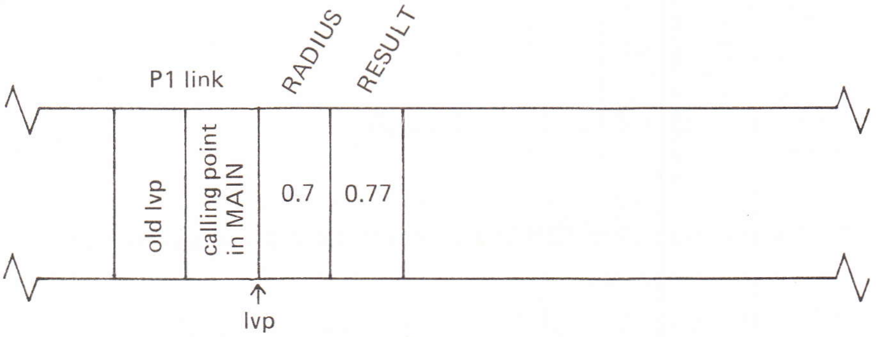
ii) Return to the correct point in the calling procedure; in this case it is the assignment of the result to AREA.

Thus statement 3 is completed and the stack appears as:



Note that all the locals of CIRCLE have disappeared — this is the reason why they cannot be accessed elsewhere — they do not exist!

P2 now returns its result, the stack is unwound to the previous link cell and the assignment to result is made:



The link cells form one type of housekeeping item held in the run-time stack; other locations are used to keep global information about the program and unnamed locations used to store partial results necessary in a complex calculation (for instance in the evaluation of (a+b)*(c+d), a temporary location may be required for (a+b) whilst calculating (c+d)) — this is generally termed a *work area.*

You will notice that the parameters are always arranged in their order of declaration with a fixed relation to their link cell. This makes the parameter mechanism extremely efficient: on a procedure call there is no overhead in specifying where the parameters are to be found, they can automatically be assigned to the correct locations in the stack.

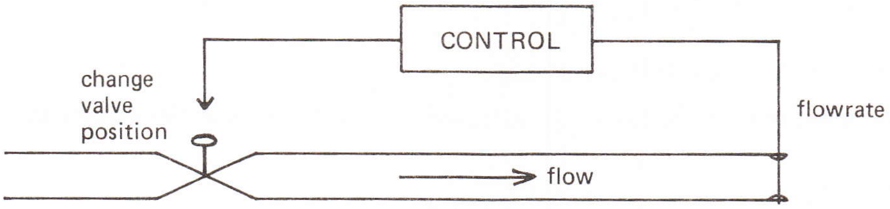
We are now familiar with static variables declared in data bricks which name fixed locations in core, and dynamic variables which have a transient life span; successive calls of a procedure lead to new independent incarnations of the local variables which may not name the same location in the stack on each call, because of differing orders of calls. Global variables can be thought of as ones used to communicate information between procedures. The organisation of variables (involving the decisions on which variables to make global and which local) is an important aspect of program design and management.

No mention has been made of where the procedure body itself is; within the machine, all it consists of is a sequence of machine instructions. These are kept independently of the data bricks and the run-time stack (the linkage mechanism providing the necessary control information); the bodies contain no slots for variables and the code is designed to be *read-only*: that is there can be no modification of it at execution time. We shall come back to this point later. The bodies of data bricks, on the other hand are named locations which can be manipulated as desired at execution time, and can be used to communicate information between procedure bricks.

10. Application

Although we have not learned many constructions in RTL/2, we have, nevertheless, seen some of the structure behind a program and are in a position to investigate in this section an application, albeit a vastly simplified one; note that our method is just one of many possible approaches.

A typical process control situation is the use of a DDC (direct digital control) algorithm to control some closed loop of action without human intervention. The aim is basically to maintain a desired value for a quantity by measuring its current value and empioying the difference between this and the desired value to initiate action to reduce the difference.



Consider a simple system consisting of a valve and some means of measuring the flow-rate through the system. The required rate is a known quantity which may vary with time, (for example it may be set by the required conditions downstream), and the flow rate will vary according to the valve position and conditions downstream. The box marked 'control' in the diagram must use the measured flow-rate and the required rate (*set-point*) to calculate the necessary change in valve position.

Three different approximation terms are commonly used in making error corrections, and for those unfamiliar with this subject, a brief summary of them will be given.

a) Proportional Control
The correction applied is a simple constant times the observed error. Thus

$P_{new} = P_{old} + KE(x_n)$

where P is the controlled quantity
x_n is the current flow-rate
$\overline{x}$ is the set point value
$E(x_n) = \overline{x} - x_n$ is the current error
and K is the proportional constant, chosen to make an optimum correction (no consideration will be given here to the theoretical or empirical means by which K might be chosen); clearly too small a value for K results in an excessive continuous error whilst too large a value gives instability or repeated over-correction.

b) Integral Control
The correction applied is a constant times an integral over time of the error. Thus

$P_{new} = P_{old} + L \int_o^t E(x_n)dt$

Integral control is able to eliminate errors completely since corrections are added whilst any error remains — there is no steady-state error. However, if the set point changes, the accumulated correction must itself be integrated out before the desired correction is complete. This type of control is sluggish in response and can easily lead to overcorrections and oscillatory responses.

c) Derivative Control
In this case the correction is a constant times the derivative of the error with respect to time. Thus

$P_{new} = P_{old} + M \frac{d}{dt} E(x_n)$

This method is often very useful since it can detect a change and apply a large correction while the error is forming rather than waiting until the error itself has increased to a large value. However, derivative control can never be used alone. By its mathematical structure it can only detect a changing error and hence a large steady error would generate no correction.

Also, it is difficult to evaluate accurately such a derivative for a physical process and such a term is particularly susceptible to noise (instrument variations are often of short duration and hence have large time derivatives).

In practice, some combination of these terms is used in the controller – proportional, proportional plus integral or proportional plus integral plus derivative. The third of these is the most general, the other two being obtained by putting L=M=O and M=O respectively.

Hence our valve position can be expressed as

$$P = P_o + KE(x_n) + L \int_o^t E(x_n)dt + M\frac{d}{dt} E(x_n)$$

(We assume that all the quantities have appropriate units.)

Since our measurements are taken at discrete time intervals, we can rewrite this in the difference form

$$\Delta P = K\Delta E + LE\Delta t + \frac{M}{\Delta t}\Delta(\Delta E)$$

where Δt is the sampling interval and ΔE is the difference between the errors of the current measurement and the previous one, and, similarly, $\Delta(\Delta E)$ is the second order change in error (i.e. $\Delta(\Delta E) = \Delta E(x_n) - \Delta E(x_{n-1})$). ΔP is the necessary change in valve position.

How are we going to program this in RTL/2?

We shall assume at this stage that the measurement is somehow read into a global variable and that a suitable output command can be generated from ΔP. (Although input and output are fundamental to any program, since one presumably wishes to have some result, it is not convenient to discuss them at this point). We can put all our variables in a data brick, set up constants and measurements and simply grind out the formula. Investigation of the formula shows that the three latest measurements and set-points are required. Concentrating then on the relevant declarations and formula evaluation, and ignoring the problems of timing and contact with the physical process, our program appears as:

```
% CONTROLLER FOR VALVE ADJUSTMENT %

DATA SYSTEM;
   REAL MEASURED,SETPOINT,             % CURRENT VALUES %
      LASTMEASURE,OLDPOINT,            % PREVIOUS VALUES %
      VERYOLDMEASURE,VERYOLDPOINT,  % ONE BEFORE LAST %
      INTERVAL,                        % TIME INTERVAL BETWEEN READINGS %
      CORRECTION,                      % DELTA P - VALVE CHANGE %
      K,L,M;                           % CONSTANTS %
ENDDATA;

PROC DDC ();

% SET UP CONSTANTS AND INPUT MEASUREMENT %

   CORRECTION:=K*((SETPOINT-MEASURED) - (OLDPOINT-LASTMEASURE))
         + L*(SETPOINT-MEASURED)*INTERVAL
         + M*(((SETPOINT-MEASURED) - (OLDPOINT-LASTMEASURE))
             -((OLDPOINT-LASTMEASURE) - (VERYOLDPOINT-VERYOLDMEASURE))
             ) / INTERVAL;
   % RESET FOR NEXT SAMPLING POINT %
   VERYOLDMEASURE:=LASTMEASURE;  VERYOLDPOINT:=OLDPOINT;
   LASTMEASURE:=MEASURED;  OLDPOINT:=SETPOINT;

% NOW OUTPUT CONTROL SIGNAL %

ENDPROC;
```

This is not very efficient! The formula could be simplified algebraically and, in fact, only the errors need to be remembered for the subsequent steps. So we will rewrite the sequence with these improvements. To indicate a different structure, we will place the calculation of each combination of correction terms in a separate procedure brick, and thus provide a choice of control algorithms, to be selected within the body of DDC.

```
DATA SYSTEM;
   REAL MEASURED,SETPOINT,           % CURRENT VALUES %
      NEWERR,OLDERR,VERYOLDERR,      % LAST THREE ERROR TERMS %
      INTERVAL,                      % TIME INTERVAL %
      CORRECTION,                    % DELTA P - VALVE CHANGE %
      K,L,M;                         % CONSTANTS %
ENDDATA;

PROC PROP () REAL;
   RETURN(K*(NEWERR-OLDERR));
ENDPROC;

PROC PROPINT () REAL;
   RETURN( PROP() + L*NEWERR*INTERVAL);
ENDPROC;

PROC PROPINTDERIV () REAL;
   RETURN( PROP() + PROPINT ()
      + M*(NEWERR-2.0*OLDERR+VERYOLDERR) / INTERVAL);
ENDPROC;

PROC DDC ();
% SET CONSTANTS AND INPUT MEASUTEMENT %
   NEWERR:=SETPOINT-MEASURED;
   % NEXT STATEMENT DEPENDS ON CONTROL SELECTED %
   CORRECTION:=PROP();
   % OR %
   CORRECTION:=PROPINT();
   % OR %
   CORRECTION:=PROPINTDERIV();
   % RESET FOR NEXT CALL %
   OLDERR:=NEWERR;
   VERYOLDERR:=OLDERR;
   % ETC %
ENDPROC;
```

```
% IN THIS METHOD WE COULD CALCULATE THE COMPONENT ADJUSTMENTS IN THE %
% INDIVIDUAL PROCEDURES AND SELECT THE REQUIRED FORM OF CONTROL BY A %
% STATEMENT CORRECTION:=PROP()+PROPINT()+PROPINTDERIV() ; HAVING SET %
% APPROPRIATE NUMBERS FOR THE CONSTANTS. OR WE COULD OMIT K,L,M FROM %
% THE PROCEDURES AND WRITE K*PROP()+L*PROPINT()+M*PROPINTDERIV() WITH%
% K,L,M SET IN LINE BEFORE THE STATEMENT %
```

Notice that we cannot reset the remembered errors within the procedures PROP etc. since we do not know which one may be called, and in any case they call each other. Note too the logical error in the order of the last two statements of DDC; by re-setting OLDERR first we lose its current value and hence reset VERYOLDERR to NEWERR as well! Such an error might not show up in the actual execution since we are only dealing in approximations anyway, but might lead to inefficient operation; it is an example of the sort of error that cannot be found for you by a compiler!

To show yet another approach, we code up the same formula in a procedure which returns the correction as result; all the required process information is passed through parameters. Clearly this could be used for a number of different valves, and the required algorithm selected by the choice of suitable values for K, L, M on the various calls. Note the way the errors required for future calls for a particular valve are handled by ref-real parameters.

```
% CONTROLLER FOR VALVE ADJUSTMENT %
PROC DDC (REAL K,L,M,MEASURED,SETPOINT,INTERVAL,
          REF REAL OLDERR,VERYOLDERR) REAL;
REAL NEWERR,CORRECTION;
   NEWERR:=SETPOINT-MEASURED;
   CORRECTION:=K*(NEWERR-OLDERR)
              +  L*NEWERR*INTERVAL
              +  M*(NEWERR-2.0*OLDERR+VERYOLDERR) / INTERVAL;
% RESET ERROR TERMS %
   VAL VERYOLDERR:=OLDERR;
   VAL OLDERR:=NEWERR;
   RETURN(CORRECTION);
ENDPROC;
```

These various approaches are intended to illustrate the flexibility of RTL/2 and to emphasize the need to decide (for the particular problem and its constraints) which program structure is appropriate.

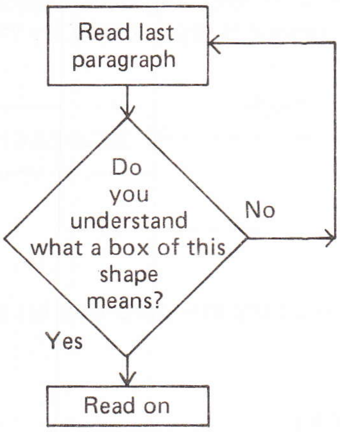
In our examples, we have mentioned selecting a particular algorithm. We need further statement types to make a choice at a particular point in a program. This forms the subject matter of the next section.

11. Conditional statement

In the DDC example of the last section we suggested that the particular algorithm required could be selected. It is essential that certain parts of any computing process be executed if and only if specified conditions are satisfied. For example, in a banking program, we will wish to take special action for an overdrawn account; in a process control situation error conditons will need to be detected; in a numerical analysis calculation, we will need to test that some desired accuracy has been achieved.

It has already been pointed out that the execution of a procedure by means of a call involves an implicit transfer of control to a different chunk of program text; however, this transfer does not involve any change in the logical flow during the execution, the procedure mechanism is effectively an efficient shorthand for writing out all the instructions at every point where those actions are required. The decision taken to execute part of a program based on some specified condition implies that at run-time different logical paths will be followed depending on the various conditions involved. In planning the sequence of operations required in a problem, a *flowchart* provides a useful visual aid. A flowchart consists of a number of boxes, the shapes of which indicate the nature of the operations described within the boxes, together with connecting lines and arrows which show the "flow of control" between the boxes. In this Manual we use a rectangle (▭) to indicate any general processing operation and a rhombus (◇) to indicate a decision; the lines leaving the latter box are labelled with the outcome of the decision that causes that particular path to be followed.

Example:



Within the decision box there is a condition to be tested. A condition is simply a proposition that may be true or false (i.e. either it is satisfied or it is not). The usual condition imposed on a child before it is allowed some treat is "if you are good . . ."; the proposition "you are good" may be true or false (though how this is decided may be somewhat difficult and arbitrary!). In RTL/2 a condition consists of a proposition about a simple relationship between two expressions. Syntactically, it consists of two expressions (such as we have already seen on the right hand side of an assignment statement) separated by a *relational operator* or *comparator*; these are just fancy names for any of the six mathematical symbols of equality and/or inequality:

= 'equal to'
# 'not equal to'
< 'less than'
> 'greater than'
<= 'less than or equal to'
>= 'greater than or equal to'

Before we investigate how to use them, a few comments are required:

a) The ISO7 character set is not as standard as its name implies and even less so after it has been interpreted by a manufacturer! In particular there is a lack of uniformity in the treatment of the number sign (hash #) and the currency symbols £ and $. Because of this, the three symbols are considered to be interchangeable within RTL/2, the intention being that whatever the data preparation equipment, the key marked # may be used.

b) The combinations <= and >= are items (like :=) regarded as indivisible and hence must not contain any spaces or other layout characters.

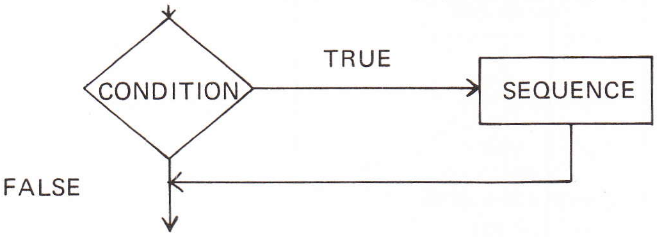
Examples of RTL/2 conditions are:

```
    B*B  < 4.0*A*C
SIN(PHI) >= 0.6
    TIME = 0.0
ABS(X-Y) <= EPSILON
    7.9  # 10.8        % ALWAYS TRUE %
    3.1  > 4.6         % ALWAYS FALSE %
```

The comparators =, # should not in general be used to compare two real quantities since they do not take into consideration the accuracy to which numbers are held within the machine (e.g. A–B=0 may be satisfied whilst A=B is not). Instead, a comparison of the form ABS(A–B)<TOLERANCE should be employed.

The comparisons are performed ultimately between two real numbers; since each of the two expressions must deliver a real object to enable this to be done, variable names are automatically dereferenced (once or twice) just as on the right hand side of an assignment statement with a destination demanding a real.

We use conditions in *conditional statements* to decide whether or not to execute a particular piece of program (this is equivalent to selecting which piece from a number of possibilities). The simplest form of conditional statement is represented by the flowchart:

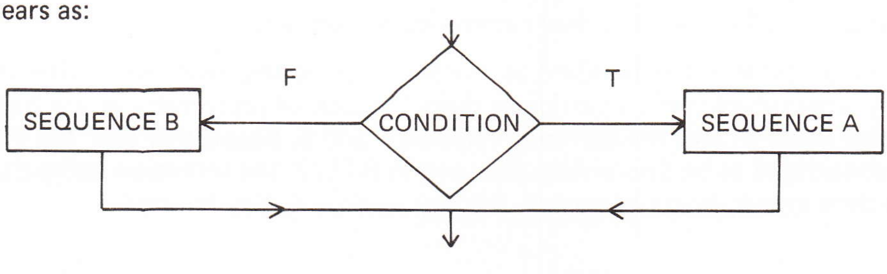


The RTL/2 form of this is similar to the standard English construction "if it rains then we will stay indoors". We write:

```
IF Q < 0.0 THEN
    NEGATIVERESPONSE ();
    Q:=-Q;
END;
```

IF and END are keywords which delimit the statement; every IF must have a matching END. The final semi-colon (as usual) separates the statement from the next one. THEN is also a keyword; it marks the end of the condition and the beginning of the sequence of code which is to be obeyed if the condition is true. The example shown behaves just as the flowchart requires. The condition Q<0.0 is tested; if this is true then (notice that the English hardly adds much to the comprehension!) the procedure NEGATIVERESPONSE is called, Q is negated and the next statement to be obeyed is the one following the END; if the condition is false then the statement following END is obeyed immediately. Note the way we have lined up the IF and END and indented the intermediate statements; this is not essential, but aids legibility. A good rule with matching keywords like IF, END (and there will be others) is to put both of them on the same line if the statement can be contained on one line, otherwise to indent them to the same level.

Our next sort of conditional statement deals with the case of alternative actions; in English we say "if it's fine then we will go to the seaside otherwise we'll go to the cinema". Our flowchart now appears as:



In RTL/2 we use the keyword ELSE to introduce the alternative (rather than 'otherwise') and our statement will take the form:

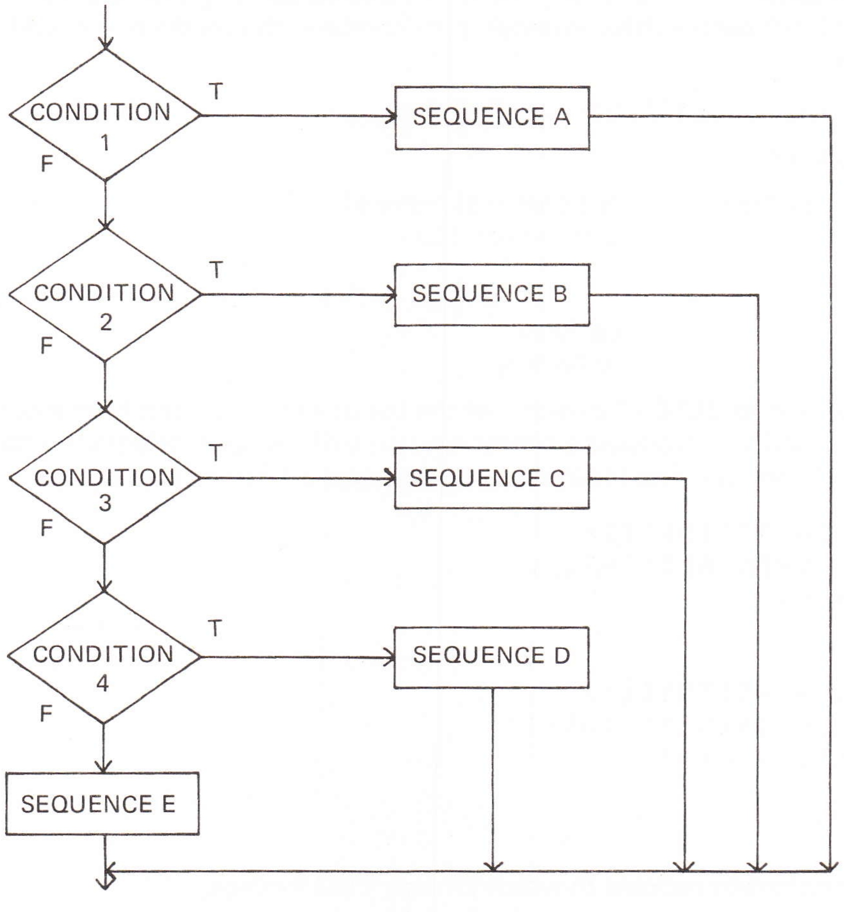
```
IF Q < 0.0 THEN
   NEGATIVERESPONSE ();
   Q:=-Q;
ELSE
   POSITIVERESPONSE ();
   Q:=Q+INTERVAL;
END;
```

Strictly speaking the semi-colons preceding the ELSE and END are redundant, but they do no harm and in the event of adding further statements in these positions avoid the mistake of omitting the separating semi-colons. The concept of having statements within a conditional statement may be confusing: the construction as a whole (from IF to END) behaves as a single statement and this is how it appears from outside; however, within, it can contain as many complex statements as you like; it is similar to a bracketed expression which behaves as a single term but may contain many complex terms and operators.

Again, our example behaves as the flowchart; the condition Q<0.0 is tested; if it is true then the procedure NEGATIVERESPONSE is called, Q is negated and the next statement is the one following END; if it is false then the procedure POSITIVERESPONSE is called and Q is increased by INTERVAL, the next statement being the one following END.

Clearly the two alternatives are exclusive, and to avoid the execution of both there must be some change in sequence at run-time; the changes are explicitly programmed by the use of IF,ELSE, END, the actual changes being organised by the compiler.

A more complex situation arises if we have a number of possible actions at run-time depending on a number of conditions for example the particular range in which a variable lies. This is illustrated by the following flowchart:



To cope with a succession of conditions, we introduce the further keyword ELSEIF. As this is an item, there must be no spaces — ELSE IF will be treated as two keywords. The RTL/2 text corresponding to the flowchart then appears as:

```
IF Q < 0.0 THEN
   NEGATIVERESPONSE ();
   Q:=-Q;
ELSEIF Q < 1.0 THEN
   FRACTIONACTION ();
ELSEIF Q < 10.0 THEN
   POSITIVERESPONSE ();
   Q:=Q+INTERVAL;
ELSEIF Q < 100.0 THEN
   Q:=Q/10.0;
   RECYCLE ();
ELSE ERRORACTION ();
END;
```

This is the most general form of the conditional statement. The behaviour of this and the earlier forms may be described as follows: the condition following IF is tested; if this is true then the sequence following THEN is executed and this completes the statement (i.e. the sequence delimited by ELSEIF, ELSE or END; by completion we mean that the next statement is the one following END — this is where the idea of the conditional statement as an entity is important); if the condition is false then the conditions following the optional keywords ELSEIF are tested until one is found to be true; the sequence following its THEN is executed and this completes the statement; in the absence of ELSEIF conditions of if they are all false then the sequence following the optional ELSE is executed and this completes the statement; in the absence of an ELSE part, the statement is completed and the overall effect is nothing. That's not quite true — here is a warning! Within the various conditions, we may be calling functions which, as side effects, alter the values of global variables for instance; only the first condition is guaranteed to be tested (since if it is true any ELSEIF parts will be ignored) so be careful with functions in conditions which produce side-effects!

Hence the general form may be symbolically represented as:

IF condition THEN sequence

ELSEIF condition THEN sequence — optional — as many of these as you like
.
.
.
ELSE sequence — optional
END — obligatory

ELSEIF is, in fact, equivalent to ELSE IF except that the use of the compound form saves the need for a matching END and also produces a clearer rendering of the appropriate flowchart (and may be compiled more efficiently). The following two sequences are identical in action:

```
IF ALPHA>BETA THEN ACTION1();
ELSEIF ALPHA<0.0 THEN ACTION2();
ELSE ERRORACTION ();
END;

IF ALPHA>BETA THEN ACTION1();
ELSE   IF ALPHA<0.0 THEN ACTION2();
       ELSE ERRORACTION();
       END;
END;
```

Note the different indentations to indicate to which IF each END belongs.

We can now write our earlier procedure to return the maximum of two real numbers using a test rather than an arithmetic formula:

```
PROC MAX (REAL A,B) REAL;
% RETURNS THE MAXIMUM OF A AND B %
   IF A>B THEN RETURN(A);   END;
   RETURN(B);
ENDPROC;
```

Notice that we have not used an ELSE part; you may complain that when A is greater than B, the sequence following THEN will be obeyed and then the statement following the END will be obeyed so that B will always be returned. This is not so. The sequence following THEN contains a return-statement which sets up a result and exits from the function to the point of call so that the rest of the body is not executed. It would be perfectly legitimate to write:

IF A>B THEN RETURN(A) ELSE RETURN(B) END;

However, the compiler may not be very clever in spotting that each logical path contains a RETURN statement and may think an exit from the function is possible without setting up a result!

Whenever we wish simply to select a value according to certain conditions, the conditional statement becomes tedious and messy to use, since we may need to introduce auxiliary variables merely to remember temporarily the value selected. To avoid this, RTL/2 allows the use of a *conditional expression* which behaves like a bracketed expression. In form, it is similar to a conditional statement with the following important differences:

i) Instead of sequences, each possibility consists of an expression.
ii) The ELSE part must be present: this is obviously sensible, since an expression must deliver some object, regardless of the conditions.
iii) Since we are in an expression, there must be no semi-colons.

Our procedure becomes:

```
PROC MAX (REAL A,B) REAL;
   RETURN(IF A>B THEN A ELSE B END);
ENDPROC;
```

As usual, the expressions for each possibility can be as complex as we like (and may therefore contain conditional expressions!) but each path must deliver an object of the required sort. Thus dereferencing will be applied where necessary. Consider the following:

```
REAL A,B,MAX;
REF REAL BIGGER;

   BIGGER:=IF A>B THEN A ELSE B END;
    MAX   :=IF A>B THEN A ELSE B END;
```

The right hand sides look identical, but their actions will be different. In the first assignment, the destination is a ref-real which requires the name of a real identifier; the conditional expression will therefore yield the name A or the name B according as the value of A is greater than B or not. In the second case, we require a real object to be delivered, so the contents of A or the contents of B is the object delivered by the expression.

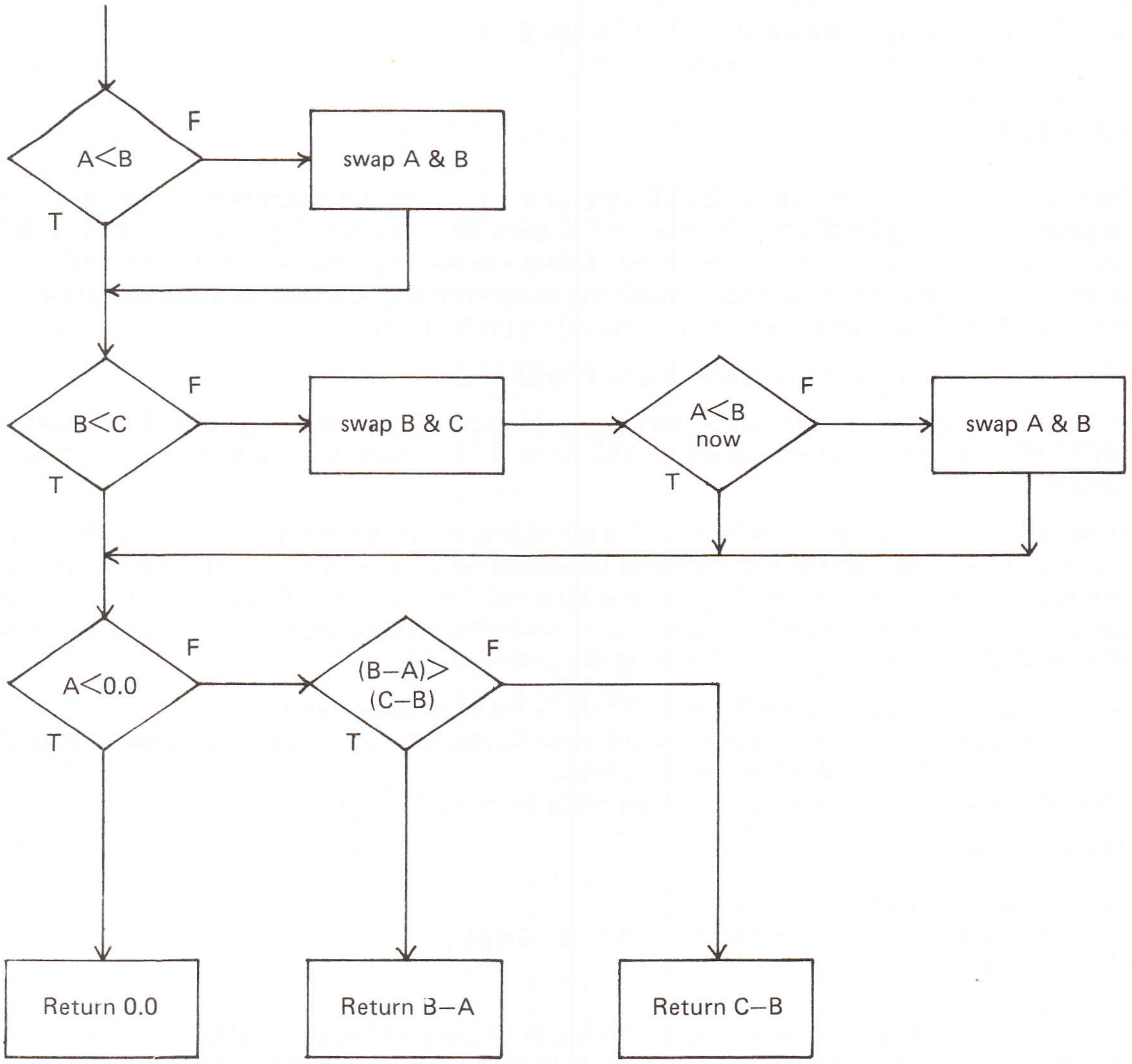
By now it should go without saying that conditional expressions may occur wherever terms may occur so that we may use them in parameters and write expressions such as:

A := C+IF A>=B THEN B*B ELSE 697.2–(A+C)END*3.0;

We conclude this section with a worked example:

Write a procedure to rearrange three given variables in ascending order and return the maximum difference between them, unless the minimum value is negative, in which case zero is the result.

Since we wish to rearrange values, the procedure will have ref-real parameters; call these A,B,C. We can draw a flowchart:



We will use our earlier procedure SWAP to perform the changes.

```
PROC SWAP (REF REAL X,Y);
REAL TEMP;
   TEMP:=X;
   VAL X:=Y;
   VAL Y:=TEMP;
ENDPROC;

PROC MINASC (REF REAL A,B,C) REAL;
% REARRANGES VALUES OF A,B,C INTO ASCENDING NUMERICAL ORDER IN A,B,C %
% AND RETURNS MAXIMUM DIFFERENCE OR ZERO IF MINIMUM IS NEGATIVE %
   IF A>B THEN SWAP(A,B);  END;
   IF B>C THEN
      SWAP(B,C);
      IF A>B THEN SWAP(A,B);  END;
   END;
   RETURN(IF A<0.0 THEN 0.0
          ELSEIF B-A > C-B THEN B-A ELSE C-B  END);
ENDPROC;
```

Notes:

i) Some of the inequalities are the reverse of the ones shown in the decision boxes, to make the action part follow the THEN and give a null ELSE part which can be omitted – we could have written for instance:

```
IF A<B THEN          % NULL ACTION %
ELSE SWAP(A,B);
END;
```

when documenting such cases, it is probably best to reverse the inequality (and the true and false labels!) of the decision box.

ii) Note the use of a conditional statement within a conditional statement.

iii) Note that VAL is not required within MINASC – all dereferencing (once or twice) is automatic – you are recommended to work through the procedure and indicate where dereferencing occurs.

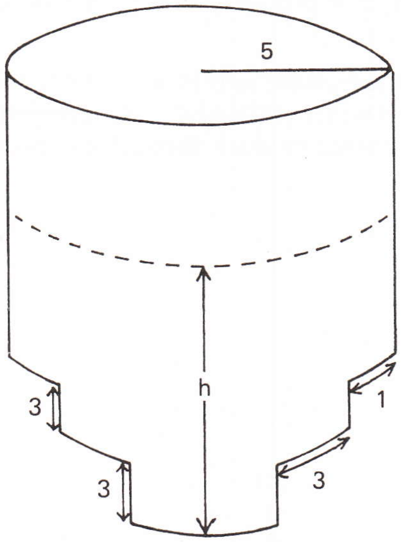
Section 11 examples

1 Rewrite the DDC example of section 10 using conditional statements to select the control algorithm required. Illustrate the use of a conditional expression used as a parameter by writing a call of the final version of DDC.

2 Write a procedure to return the step function:

$$f(x) = \begin{cases} 0.0 \text{ for } x<0.0 \\ 0.5 \text{ for } x = 0.0 \\ 1.0 \text{ for } x>0.0 \end{cases}$$

3 A cylindrical tank has the following shape:



Write a function to compute the volume of fluid contained for a given depth h; you may assume that the tank does not overflow.

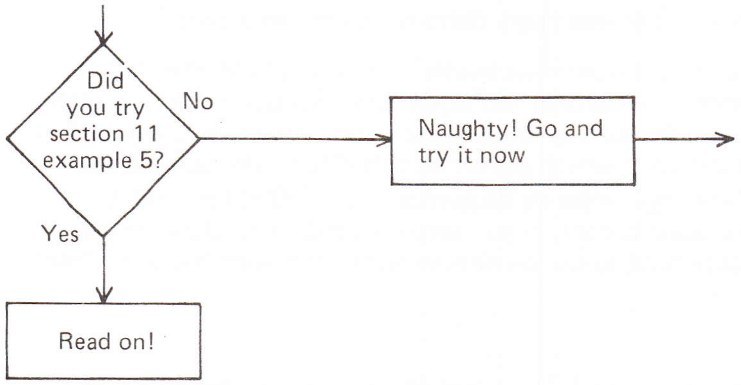
4 Rewrite the ABS operator as a function.

5 The Newton-Raphson method for the extraction of a square root computes successive approximations to $\sqrt{a}$ from the formula

$x_{n+1} = \frac{1}{2}(x_n + a/x_n)$ (where x_1 is a first guess)

terminating when two successive approximations differ by less than some desired accuracy. Draw a flowchart to represent this process, and write a procedure in RTL/2 which will return the square root of a positive number.

12. Labels; transferring control



If you still haven't attempted the examples look at the answers, where you will see that the solution consists of an indefinite number of repetitions of three statements. Even with a large number, n, of these repetitions we cannot guarantee (for a particular value of EPS) that a value will be found of the correct accuracy for all possible values of the parameter A. In any case, it would be very tedious and error prone if, in iterative situations, we had to write out n repetitions of the same sequence (Dijkstra compares it to writing out punishment lines at school!); we introduced procedures partly to save drudgery, so we want to make this situation simpler too.

The basic problem is that we want to say "as before" or "go back and do it again". We need to make an explicit transfer of control; the point is illustrated in the flowchart at the head of this section. On the 'NO' route we wish to return to the previous section; we have left an arrow dangling. You know where to go though, because the example is numbered.

We naturally use the same principle in RTL/2. Firstly we wish to identify a particular point in the program text, that is, to *label* it. What can we label? Statements. Any statement can be labelled, even if it is a statement which is a constituent of a more complex statement – for example an assignment embedded in a conditional statement; only statements can be labelled. Syntactically, all we do is to precede the statement we wish to label by a name followed by a colon (:). Thus,

```
   NEXTX:=(CURX + A/CURX) * 0.5;
L: IF ABS(NEXTX-CURX) < EPS THEN RETURN(NEXTX);  END;
   CURX:=NEXTX;
```

The conditional statement has been labelled, and we have chosen to call that point L. We have in no way affected the logic of the program. After the assignment to NEXTX, the conditional statement is obeyed; that is, passing through a label in the normal lexicographical flow causes no action to be taken.

What is a label? A label is an object (similar to a real number) which happens to be a particular (fixed) point in the program text. We will learn how to manipulate such objects in later sections.

There is nothing to stop us labelling a statement that is already labelled! We merely precede it with another name and colon:

M: L: IF ABS(NEXTX–CURX)<EPS THEN . . .

This may seem absurd (or an academic nicety) but in fact can have practical advantages, for example:

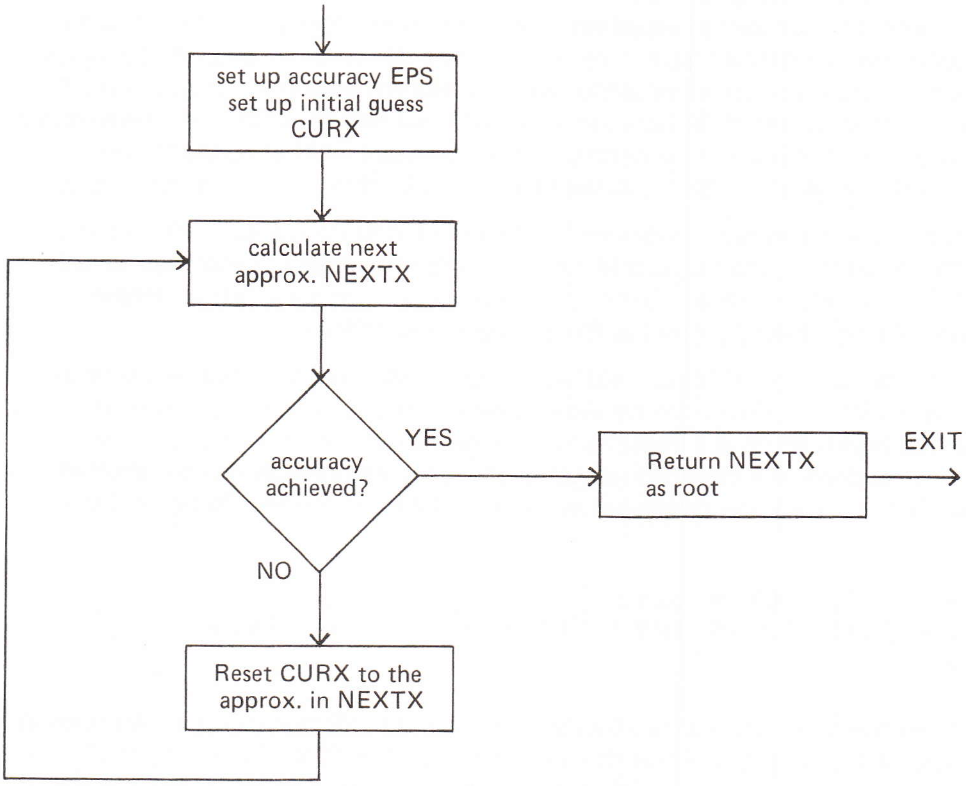
i) Where uses of the point are made in widely differing parts of the program two mnemonically significant names may increase the clarity of the program.
ii) Two labels may emphasize similarities with other sections of program.
iii) Two labels may be useful if, at some later date, the actions performed there are likely to become distinct.

Since a label is affixed to some statement, it must occur in a procedure brick. Although a label is in some sense a 'constant' (it is a fixed point and is not a place that can be assigned to) its name is regarded as local to the procedure in which it is *set* (i.e. occurs) and it can only be used in that procedure (the reasons and rules for this will become more definite in later sections).

The sort of use we want to make, of course, is to cause an explicit transfer of control to a labelled point — we cannot make a transfer to an unlabelled statement! Normally the successor statement to a statement is the one which follows it in the program text. An explicit transfer of control must break this sequence. The statement which performs this effectively defines the successor statement. The keyword GOTO (single item so no spaces — GO TO is two names!) followed by the name of a label (as mentioned before, in the same procedure brick) forms a *goto-statement* and says that the next statement to be performed is the one with the given label name. So we simply write

```
GOTO L;
```

We can now construct a flowchart and produce the RTL/2 text for our square root example:



The picture of continually repeating the last three steps until the accuracy is reached (and the RETURN statement finishes the procedure) is precisely what we wrote in our earlier solution; however we now only need to write it once and include a label and a goto-statement:

```
PROC SQRT (REAL A) REAL;
REAL  EPS,       % ACCURACY %
      CURX,      % CURRENT APPROXIMATION %
      NEXTX;     % NEXT APPROXIMATION %
   EPS:=0.001;
   CURX:=1.0;
TRYAGAIN:    NEXTX:=(CURX + A/CURX) * 0.5;
             % RELATIVE ERROR TEST PREFERABLE TO ABSOLUTE ONE %
             IF ABS(NEXTX-CURX) < EPS*NEXTX THEN
                RETURN(NEXTX);
             END;
             CURX:=NEXTX;
             GOTO TRYAGAIN;
ENDPROC;
```

Any such *iterative loop* returning a result must contain a test which will eventually lead to the termination of the loop – or else the machine will obey the same group of instructions infinitely! The classic infinite loop would be:

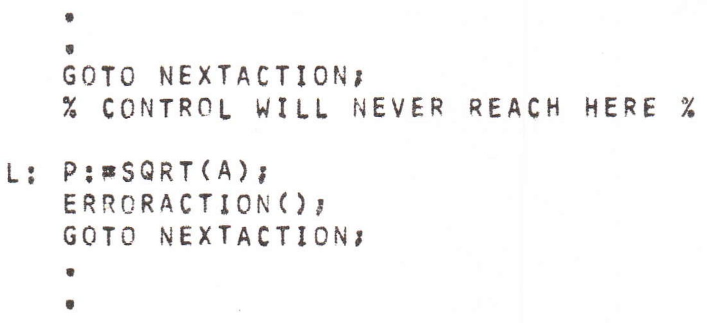
```
L : GOTO L;
```

Endless loops may be used sensibly in real-time applications, but these will normally be interrupted frequently, and/or contain instructions to delay their activities. The goto-statement is not the first transfer statement that we have encountered; you will remember that the return-statement performs the action of exiting from a procedure body and returning to the point of call; we also said that encountering ENDPROC caused a return to the point of call. The following two examples are therefore equivalent:

```
PROC FRED();
.
.
.
RETURN;
.
.
.
.
ENDPROC;

PROC FRED();
.
.
.
GOTO L;
.
.
.
L:
ENDPROC;
```

You may argue that L is not labelling a statement, since we have not introduced any 'endproc-statement'. In order to allow the placing of such labels we define a *dummy-statement.* This is simply an empty statement which does nothing and can now be forgotten! Just remember it if you have a conscience about putting a label before ENDPROC and similar keywords!

Above, we said that falling through a label had no effect; it does have the effect of obeying the statement at that point of course. Sometimes we will wish to write pieces of program which can only be reached by an explicit transfer of control. Where can we put them? The obvious place is *after* a transfer of control statement; any statement following, say, a goto-statement cannot be reached unless it is labelled and jumped to from somewhere else:



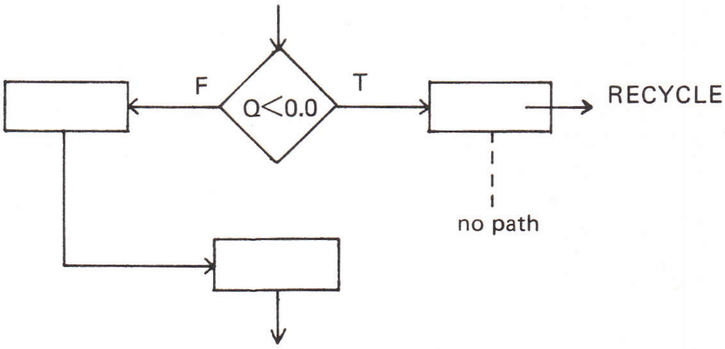
```
    .
    .
    GOTO NEXTACTION;
    % CONTROL WILL NEVER REACH HERE %

L:  P:=SQRT(A);
    ERRORACTION();
    GOTO NEXTACTION;
    .
    .
```

The coding between L: and the next goto-statement is completely self-contained and will only be executed if an explicit *branch* (jump) is made to L.

So far, all our jumps have been backward ones, that is, to points earlier in the text. There is no restriction on forward jumps to points later in the program; the only rule that has to be obeyed is that the label in a goto-statement has to be set in the same procedure body.

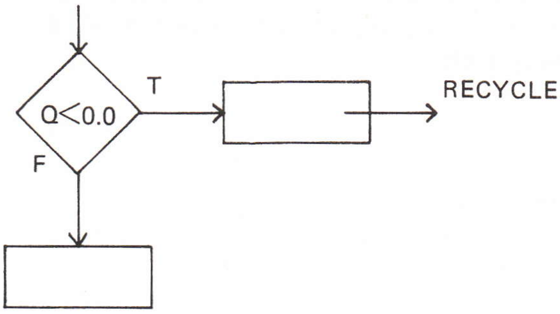
Finally, in this section, a short note on efficiency of coding. Consider the following conditional statement:

```
IF Q<0.0 THEN
    NEGATIVERESPONSE();
    Q:=-Q;
    GOTO RECYCLE;
ELSE
    POSITIVERESPONSE();
    Q:=Q+INTERVAL;
END;
NEXTACTION();
```



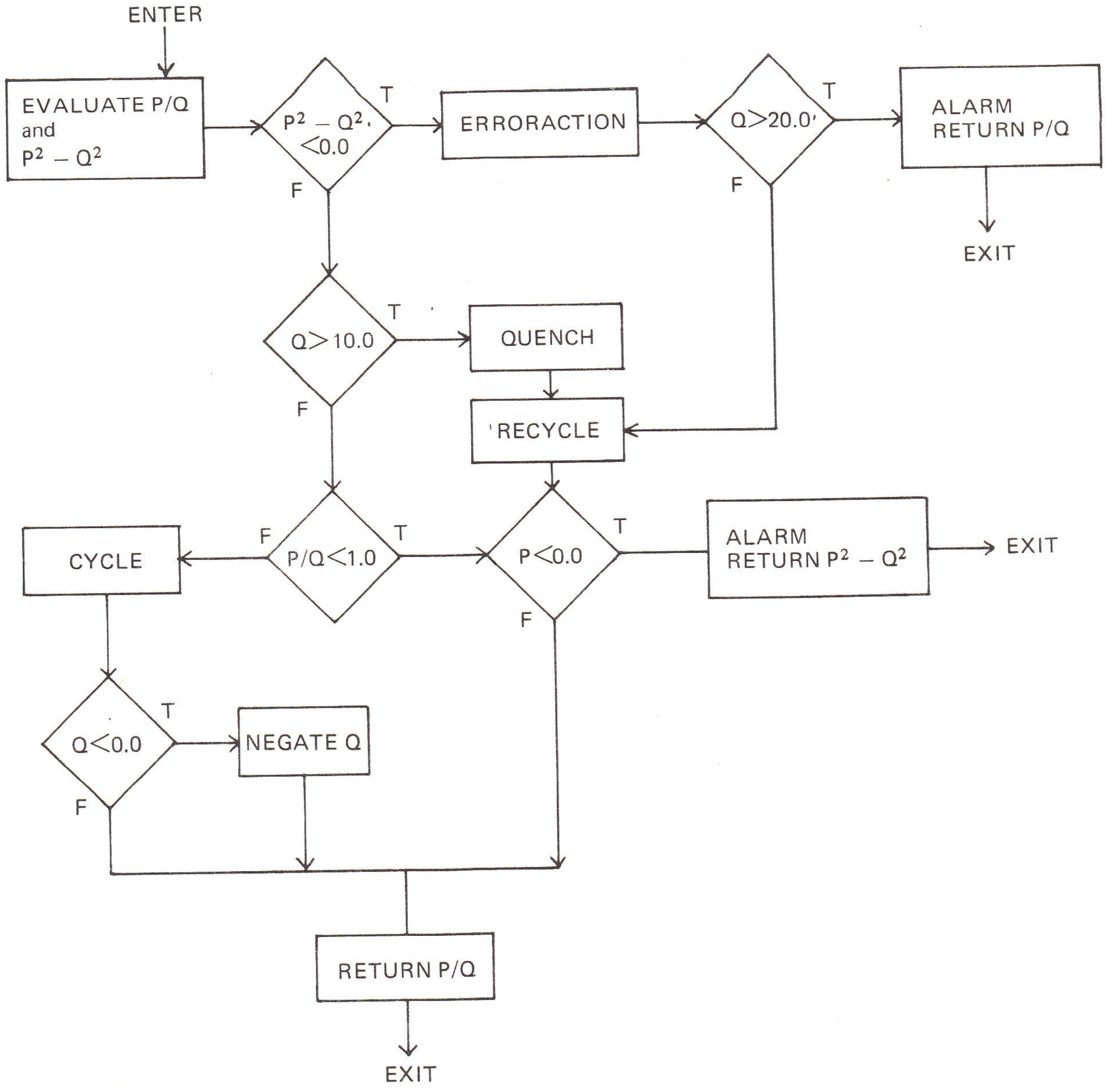
Since the actions when the condition is true contain a jump to RECYCLE, the only way the statement following END can be reached is if the condition is false; therefore we might as well put *all* the actions to be taken for a false condition after the END. This removes the ELSE part entirely, and might be compiled more efficiently since some of the inherent jumps in a conditional statement will not be required. The RTL/2 is also slightly clearer:

```
IF Q<0.0 THEN
    NEGATIVERESPONSE();
    Q:=-Q;
    GOTO RECYCLE;
END;
POSITIVERESPONSE();
Q:=Q+INTERVAL;
NEXTACTION();
```



Section 12 examples

1 Assuming that ERRORACTION, CYCLE, RECYCLE, QUENCH, ALARM are parameter-less, result-less procedures and that P, Q are real parameters, write a procedure brick to perform the following:



2 One way of checking the accuracy of real arithmetic is to multiply together two complex numbers whose modulus is unity, and then to repeatedly multiply the product by one of them. Write a procedure to do this; devise a counting mechanism for the number of multiplications, and return this count as result as soon as the modulus of the product differs from unity by more than 0.0001.

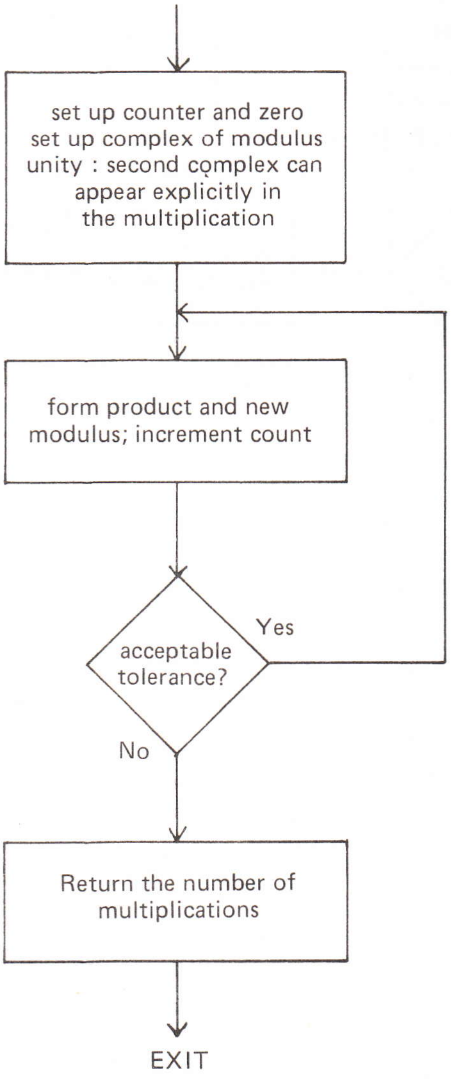
Hints:

i) Each complex number x + iy will have to be represented by two reals X,Y; its modulus is $\sqrt{(x^2+y^2)}$ and the product of two complex numbers is given by $(x_1 + iy_1)(x_2 + iy_2) = x_1x_2 - y_1y_2 + i(x_1y_2 + x_2y_1)$

ii) Pythagoras' Theorem provides good complex numbers of modulus unity:

$$\left(\frac{3}{5}\right)^2 + \left(\frac{4}{5}\right)^2 = 1, \quad \left(\frac{5}{13}\right)^2 + \left(\frac{12}{13}\right)^2 = 1$$

iii) Skeleton flowchart:


set up counter and zero
set up complex of modulus unity : second complex can appear explicitly in the multiplication
form product and new modulus; increment count
acceptable tolerance?
Yes
No
Return the number of multiplications
EXIT

13. Integers

So far we have seen four kinds of object (a real, a name of a real variable, a procedure and a label – note that a data brick is not an object in this sense) but we have only been able to manipulate real values and names. Declarations must have seemed somewhat redundant, since (apart from ref-reals) everything could be interpreted correctly purely from its context. The last example showed the deficiencies of real arithmetic when used for a simple counting task. We now introduce another *plain* (i.e. arithmetic) mode, with which we can perform such operations accurately. This is the mode *integer*, the objects being whole numbers.

We can retrace our steps through sections 1 to 12 seeing how integers fit in to the pattern, and noting how, in general, they behave as the mode real does. The form of the *integer constant* is simply a sequence of decimal digits; when representing whole numbers we can dispense with decimal points and exponents, and give the number explicitly in a fixed point representation. Thus:

```
    3            267              1065487932
```

are valid integer constants. Note that we cannot follow the written English practice of using commas to group the digits. Integers are held exactly, so there is no question of accuracy as in the case of reals. The finiteness of machines means that there is a restricted range of values (machine dependent) which can be held and this is usually much smaller than the real number range. Effectively, with integers, one purchases exactitude at the expense of a limited range of values, though for the majority of uses this will not matter.

As an item, an integer constant is terminated by any non-digit character, and is characterised by its beginning with a digit and containing no decimal point, and, like the real constant, does not need to be declared.

We can declare integer variables to contain such numbers, and ref-integer variables; we use the keyword INT for this purpose. The declarations

```
INT COUNT, INDEX;
REF INT WHICHONE;
```

inform us that the locations named COUNT and INDEX will contain integers, and the location WHICHONE the name of an integer variable. The usual rules for forming an identifier name apply.

The assignment statement follows exactly as in the real case, with the left hand side defining a destination, and the nature of the object which must be delivered by the right hand side. The rules for dereferencing and the use of VAL are identical:

```
INT COUNT, INDEX;
REF INT WHICHONE;
      WHICHONE := INDEX;
           COUNT := 21;
VAL WHICHONE := COUNT;
```

You are left to work out the effect of these statements.

The same monadic operators which we met for reals may be used with integers, and, as before, negative numbers are regarded as the result of applying the monadic operator '–' to a (positive) integer constant:

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

Rules for combining monadics are the same as before.

When we come to consider dyadic operators, the situation becomes a little more involved. Addition and subtraction pose no problems. When we multiply together two numbers each in the range $[-\propto,\propto)$ (for those unfamiliar with this notation, this means a number x satisfying $-\propto\leqslant x<\propto$), the product will be in the range $(-\propto^2,\propto^2]$. This is true of real and integer multiplication. In the real case the range is large and the situation is completely dealt with by overflow of the range and any associated actions. In the integer case the situation is more likely to be encountered; also, most machines have some multiple length instructions for coping with such situations, of which we can take advantage. We therefore define an intermediate mode known as a *big integer* which has a range $[-\propto^2,\propto^2)$ when integers have a range $[-\propto,\propto)$. Note that this range may not be sufficient for all multiplications and that overflow may still occur; the reason for its choice is discussed later. Intermediate modes can only arise during the evaluation of an expression and cannot be stored or manipulated in the same way as reals and integers. Their purpose is to explain precisely what is happening. The result of a multiplication between two integers is therefore a big integer. What happens if we want to store the product in another integer location? A mode conversion must occur from big integer to integer (also called normal integer). This is performed automatically – but since there is a contraction of the range *(narrowing)* there is the possibility of arithmetic overflow at this point also.

Division between integers is defined in an elementary way, in which we have a quotient (exact and integral) and a 'remainder' (also exact and integral); two operators supply the relevant results. Regarding division as an inverse to multiplication, the dividend expected for such an operation is naturally a big integer. What happens if we want to divide a normal integer? Again a mode conversion is required and performed automatically; in this case, since there is an expansion of the range (*widening*) there is no possibility of overflow. We now define these dyadic operators for integers:

OPERATOR	PRECEDENCE	FIRST OPERAND	SECOND OPERAND	RESULT	INTERPRETATION
+	1	Integer	Integer	Integer	Form the sum of the operands
–	1	Integer	Integer	Integer	Subtract the second operand from the first
*	5	Integer	Integer	Big Integer	Form the product of the operands
:/	5	Big Integer	Integer	Integer	Quotient on dividing first operand by second
MOD	5	Big Integer	Integer	Integer	Remainder on dividing first operand by second

Notes:

i) Arithmetic overflow can occur in all cases; this may seem odd in the case of division, but think of β:/1 where $\beta=2\propto$ and $\propto$ is the range for integers; similarly the division implicit in the MOD operation can cause overflow.

ii) MOD is a reserved word (short for modulo). If negative quantities are involved, there is clearly a need to define explicitly how a sign is to be attached to any remainder. The rule is that the remainder has the same sign as the dividend; thus:

```
 22 MOD   6   is   4
 22 MOD  –6   is   4
–22 MOD   6   is  –4
–22 MOD  –6   is  –4
```

iii) Having fixed the sign for any remainder, we can easily decide how to define the quotient when negative quantities are involved. The rule here states that the result of ordinary division is truncated towards zero (i.e. any fraction part is forgotten!) Hence:

```
 22 :/  6  is  3
 22 :/ -6  is -3
-22 :/  6  is -3
-22 :/ -6  is  3
```

Like :=, :/ behaves as a single item – "integer divide".

The rules for forming expressions, precedence and bracketed expressions then follow just as for reals.

Example:

```
INT COST, CHANGE, NUMBER, UNITCOST, CAPITAL;
    COST   := UNITCOST*NUMBER;
  CHANGE   := CAPITAL–COST;
  NUMBER   := CAPITAL :/ UNITCOST;
  CHANGE   := CAPITAL MOD UNITCOST;
  CHANGE   := CAPITAL – CAPITAL :/ UNITCOST*UNITCOST;

%LAST TWO ASSIGNMENTS EQUIVALENT – NOTE PRECEDENCE%
```

When it comes to functions, procedures and data bricks, integers behave in exactly the same way as reals; locals and globals can be declared similarly, we can have parameters and results that are integer. However since the parameter mechanism is that of assignment, if we define a procedure brick having an integer parameter, then we must provide an integer expression in the call.

The six comparators =, #, <, >, <=, >= may all be used between integer expressions in conditions. Since we are now dealing with exact quantities, = and # can be used confidently. We can write conditional expressions for integers; each possible path through the expression must deliver an integer:

```
COUNT := IF INDEX #0 THEN COUNT +1 ELSE COUNT –1 END;
```

We now present some bricks using integers; each procedure calculates the highest common factor of two numbers (with varying degrees of efficiency!) The reader is left to see how they work and to ensure that he understands the RTL/2 involved.

```
PROC HCF1 (INT P,Q) INT;
INT MAX,TRY;
   IF Q>P THEN
      % SWAP USING MAX AS A TEMPORARY VARIABLE %
      MAX:=Q;
      Q:=P;
      P:=MAX;
   END;
   MAX:=TRY:=1;
A: IF TRY<=Q THEN
      IF P MOD TRY = 0 THEN
         IF Q MOD TRY = 0 THEN MAX:=TRY;  END;
      END;
      TRY:=TRY+1;
      GOTO A;
   END;
   RETURN(MAX);
ENDPROC;

PROC HCF2 (INT P,Q) INT;
INT TEMP;
L: IF Q=0 THEN RETURN(P);  END;
   TEMP:=Q;
   Q:=P MOD Q;
   P:=TEMP;
   GOTO L;
ENDPROC;
```

```
PROC HCF3 (INT P,Q) INT;
M: IF P>Q THEN
      IF Q=0 THEN RETURN(P);  END;
      P:=P-Q;
   ELSE
      IF P=0 THEN RETURN(Q);  END;
      Q:=Q-P;
   END;
   GOTO M;
ENDPROC;

PROC HCF4 (INT P,Q) INT;
REF INT BIG,SMALL;
N: IF P>Q THEN
         BIG:=P;  SMALL:=Q;
   ELSE  BIG:=Q;  SMALL:=P;
   END;
   IF SMALL=0 THEN RETURN(BIG);  END;
   VAL BIG:=BIG-SMALL;
   GOTO N;
ENDPROC;
```

We have discussed the way in which dyadic operators are combined, the precedence rules that govern the combination of monadic and dyadic operators, and in this section we have mentioned the automatic *mode transfers* between the normal and big integers. There remains the problem of combining together, if possible, integers and reals.

The rules governing mode transfers are very simple; we have already met the first one:

1 All mode transfers between normal and intermediate modes are automatic.
2 All mode transfers in which no information is lost (i.e. the case of widening) are performed automatically.
3 Any mode transfer (not involving intermediate modes) in which information is lost (i.e. the case of narrowing) must be programmed explicitly.

The only possible explicit narrowing which arises at present is that of transferring from real to integer. To indicate this, we introduce a further monadic operator and a second usage of the keyword INT:

OPERATOR	OPERAND	RESULT	INTERPRETATION
INT	Real	Integer	Rounds operand to nearest integer value

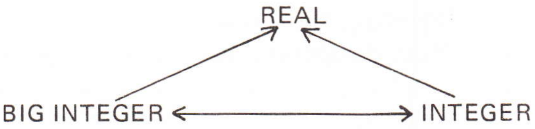
The interpretation of conversion to the nearest integer poses a problem when the real quantity lies midway between two integer values; in this case the algebraically greater of the two possible integers is taken. Hence:

INT 6.32 is 6
INT –4.95 is –5
INT 7.50 is 8
INT –7.50 is –7

We have seen three arithmetic modes so far, namely real, integer and big integer; we can investigate the six possible assignment statement types (only six, since a big integer cannot appear as a destination).

```
INT I;
REAL R;

    R:=0.7;       % FAMILIAR %
    R:=4;         % RIGHT HAND SIDE DELIVERS AN INTEGER %
                  % AUTOMATICALLY WIDENED TO 4.0 %
    R:=62*71;     % RIGHT HAND SIDE DELIVERS A BIG INTEGER %
                  % AUTOMATICALLY WIDENED TO 4402.0 %

    I:=4;         % FAMILIAR %
    I:=62*71;     % RIGHT HAND SIDE DELIVERS A BIG INTEGER %
                  % AUTOMATICALLY NARROWED TO NORMAL FORM %
    I:=INT 0.7;   % RIGHT HAND SIDE INITIALLY DELIVERS %
                  % A REAL WHICH MUST BE NARROWED EXPLICITLY %
                  % INFORMATION IS LOST AND 1 STORED IN I %
```

Note that in the case of assigning a big integer to a real, the conversion is performed directly without first narrowing to an integer; this is important since a value outside the normal integer range can be handled without overflow developing. We can express the automatic mode transfers diagrammatically:



The same rules apply when considering the evaluation of an expression containing a number of modes. How are they applied? Consider an operator □ and the expression ∝□β; there are a number of possibilities which will be treated as follows:

1 The operator □ is only defined between one pair of modes (for instance / between two reals). Then there are three cases

a) ∝, β are of the correct mode and the expression can be evaluated immediately.
e.g. 0.6/1.3
27 :/ INT 3.2

b) Either or both of ∝, β can be converted to the correct mode by a widening operation: the necessary widening is performed automatically and the expression evaluated.
e.g. 3/4 %WIDENED TO 3.0/4.0%
This situation applies also to any necessary narrowing or widening of intermediate modes:
e.g. 7*4 :/ 3*5 %BIG INTEGER NARROWED%
3*4 / 7 %BIG INTEGER WIDENED%

c) Either or both of ∝, β cannot be coverted to the correct mode (any monadic mode transfers specified explicitly are, of course, taken into account for the terms ∝, β); the expression is illegal and a compilation failure will occur:
e.g. 3.2 :/ 7

2 The operator □ is defined for a number of pairs of modes (for instance + is defined between two integers and two reals). Exactly the same rules are applied as in 1 except that we repeat them in a specific order for the various pairs of possible modes, and a failure under 1 c) is only recorded when all the possibilities have been tried. This information is shown by arranging our entries in the tabular presentation of operators in the order in which the mode matching will be attempted.

```
e.g.    1 + 4.7    % CANNOT BE PERFORMED AS AN ADDITION %
                   % BETWEEN TWO INTEGERS BUT CAN BE AN %
                   % ADDITION BETWEEN TWO REALS IF WE   %
                   % WIDEN TO 1.0 + 4.7                 %
```

Similar considerations are made in the case of monadic operators. Note that for completeness INT is also defined for an integer operand.

We now gather together all the operators met so far to indicate the ordering; remember this is only significant for a particular operator; the ordering of the operators themselves is arbitrary.

Monadics:

OPERATOR	OPERAND	RESULT	INTERPRETATION
+	Integer Real	Integer Real	Identity: no action
−	Integer Real	Integer Real	Negate the operand
ABS	Integer Real	Integer Real	Negate the operand if it is negative: otherwise no change
INT	Integer Real	Integer Integer	Identity: no action. Rounds operand to nearest integer value

Dyadics:

OPERATOR	PRECEDENCE	FIRST OPERAND	SECOND OPERAND	RESULT	INTERPRETATION
+	1	Integer Real	Integer Real	Integer Real	Form the sum of the operands
−	1	Integer Real	Integer Real	Integer Real	Subtract the second operand from the first
*	5	Integer Real	Integer Real	Big Integer Real	From the product of the operands
/	5	Real	Real	Real	Divide the first operand by the second
:/	5	Big Integer	Integer	Integer	Quotient on dividing first operand by second
MOD	5	Big Integer	Integer	Integer	Remainder on dividing first operand by second

Although the automatic widening operations may appear to lift some of the onus of mode conversion from the programmer, there are pitfalls to be avoided, and care should be taken in the choice of modes for variables and constants. For instance, a large exact constant, say a million, may be required in real arithmetic and the temptation is to write it in the exact form 1000000; however, on a small machine of small integer range, such a constant may fail at compile time, because the compiler will treat it as an integer and find it to be out of range, even though its usage is correct. This is easily avoided by writing it in the form 1000000.0 or 1E6; this is more efficient, anyway, since the constant is used as a real; the program becomes independent of the machine's integer range, and the formation of the real can be made at compile-time rather than converting the integer value at run-time.

Section 13 examples

1 Write suitable bricks to evaluate the roots of a quadratic equation with integer coefficients.

2 The Fibonacci numbers are defined by the relations

$U_n = U_{n-1} + U_{n-2}$ (n>2) $U_1 = U_2 = 1$

The ratio $\frac{U_{n-1}}{U_n}$ converges to a limit as n increases.

Write suitable bricks to calculate this limit.

3 Assuming that the operator :/ and MOD do not exist, write procedures that will perform their functions (i.e. you may use / etc.)

4 Assuming that the monadic operator INT does not exist, write a procedure that will perform its function.

14. Arrays

When names were first introduced in section 2 we darkly hinted that a name might be used for groups of locations as well as for a single cell. This forbode the idea of a named structure; in fact we have already had examples in the name of a data brick (naming a set of declarations and a static area at run-time) and of a procedure brick (naming a collection of instructions). Consider the following examples:

Example 1:
  Arrange three given numbers in ascending order of magnitude.

```
DATA NUMBERS;
    REAL A,B,C;     % THREE GIVEN NUMBERS %
ENDDATA;

PROC ARRANGE ();
REAL SWAP;
    IF B<A THEN
        SWAP:=B;
        B:=A;
        A:=SWAP;
    END;
    IF C<B THEN
        SWAP:=C;
        C:=B;
        B:=SWAP;
        IF B<A THEN
            SWAP:=B;
            B:=A;
            A:=SWAP;
        END;
    END;
ENDPROC;
```

Example 2:
  Arrange 3000 given numbers in ascending order of magnitude.

We do not attempt to solve this yet! Clearly the choice of 3,000 names and the writing of a large number of conditional statements and swapping sequences is not a very efficient method, and extremely tedious to write out. This situation of wishing to operate on sets of data treating each individual datum in the same way is common in programming. The facility we desire is to be able to name a set collectively but work with the individual members of the set.

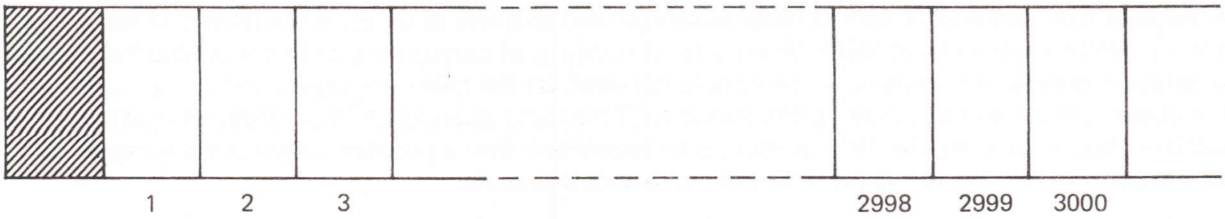
In RTL/2 this is achieved through the use of arrays. An *array* is an indexed set of variables identified by a single name. It is comparable to a vector (having a certain number of components) or to a set formed by attaching subscripts to a name. In such a structure, all the *elements* of the set must be of the same mode, and this mode is an attribute of the array. We must also know precisely how many elements belong to it. As usual, such information is given to the reader and the compiler in the declaration of the name. Thus, in our example, to declare an array for the given 3000 numbers we would write:

  ARRAY (3000) REAL A;

The keyword ARRAY says that we are declaring a structure; an integer constant (necessarily positive) in brackets specifies the number of elements (the *length* of the array) and a mode description (REAL) informs us of the nature of each element; there follows the usual list of names. Note that in this list, each array declared will have space for 3000 real variables. Similarly we can declare arrays of integers:

  ARRAY (317) INT FUNNY, ANOTHER;

Arrays are static structures, and hence such declarations are only permitted in data bricks. At execution time we will have a named area of core each element of which can be named individually by indexing the name of the structure, from 1 to the number (*bound*) specified in the declaration.



The name A belongs to the total structure and we have not shown this on the diagram which is purely illustrative – arrays may not be laid out in this way; the hatched location will contain information about the size of the structure – the use of this will be examined later. Note that it is legal (and sometimes useful) to declare an array of zero length; this structure will have a name but no elements.

Each element of the array is to behave as a single variable; what is the name of such a variable? We have said that an element is named by indexing the name of the total structure. In writing RTL/2, we do this by following the array name with an integer expression (*subscript*) in round brackets, the value of the expression indicating the element we wish to identify. Thus A(3) is the third variable element of the structure; as usual the expression can be as simple or as complex as we like, but must deliver an integer value for use as an index. Having formed a name in this way, the element behaves just as a variable of the mode associated with the array, and may occur in expressions, as a destination in an assignment, in parameters and conditions. The following example illustrates how this can happen – it is not meant to be a sensible or useful program.

```
DATA STRUCTURES;
   ARRAY (10) REAL AR;
   ARRAY (20) INT AI;
ENDDATA;

PROC ACTION ();
INT  J,K;
REF INT WHERE;
REAL P,Q;
REF REAL WHICH;
   WHERE:=AI(3);        % PUTS NAME OF ELEMENT IN WHERE %
   WHICH:=AR(7);        % SIMILAR %
   J:=K:=6;
   P:=AR(J);            % DEREFERENCE AR(J) TO YIELD REAL %
   AI(1):=7;            % AI(1) IS DESTINATION %
   AR(K-2):=P*AR(J+AI(3));   % MORE COMPLEX SUBSCRIPTS %
   IF AI(17)<AR(2) THEN      % AI(17) WIDENED TO REAL %
      AI(20):=INT AR(K);     % EXPLICIT NARROWING %
   END;
   PERFORM( AR(K+J-2) , AI( INT(AR(3)+AR(K)) ) );
      % WHERE PERFORM IS A PROCEDURE OF THE FORM %
      % PROC PERFORM (REAL A, INT B); %
      % NOTE THE USE OF INT TO FORCE AN INTEGER SUBSCRIPT %
ENDPROC;
```

We can see that the subscript expression can itself contain the names of array elements and this nesting can occur indefinitely. When the subscript contains variable names, its value can only be determined at run-time, when the contents of those variables are known; what happens in the last statement above if the integer expression INT (AR(3) + AR(K)) yields a value of 30 or a value of –73? Only the elements AI(1), AI(2), ..., AI(20) are defined by our declaration; a variable specified with a subscript which is out of the range of the declaration is not defined. The possible effect can vary; if the name is to be assigned to a ref-variable or if it is the destination in an assignment statement there is a danger that we will overwrite some location which may not belong to our program if we try to interpret the variable; to prevent this, a check (*bound check*) is made at run-time and an out of range subscript will give rise to an error condition. If we are simply dereferencing the variable, there is no possibility of corruption, only the probability of accessing meaningless values; a check here is optional. In the case of integer constant subscripts, the check can be, and is, made by the compiler. This topic is discussed more fully in a later section; the important point at this stage is to appreciate that a problem exists if the integer expression lies outside the range 1 to the number of elements.

Example 3:

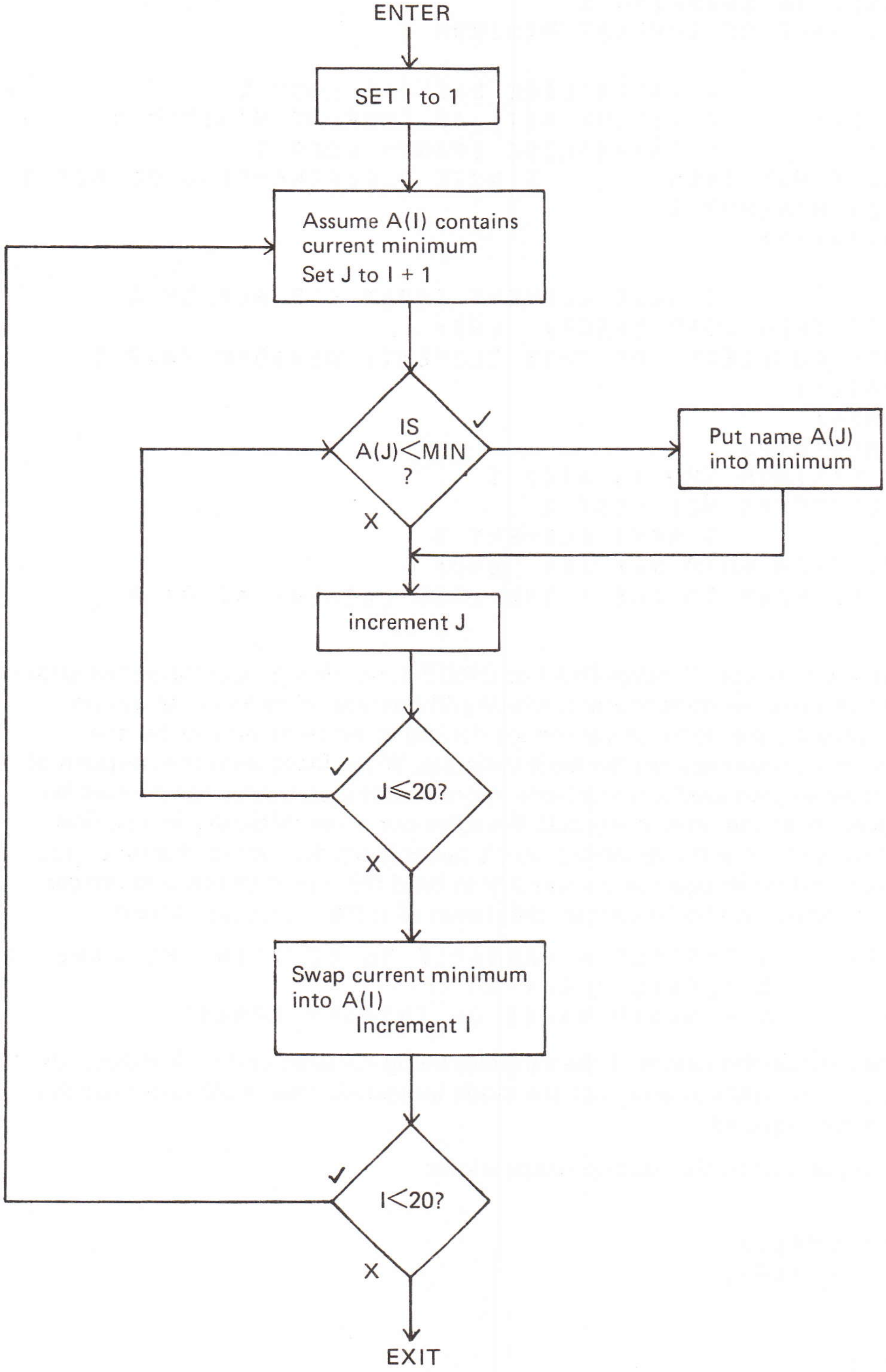
Write a procedure to form the sum of the elements of an integer array containing 100 elements.

```
DATA SUMARRAY;
   ARRAY (100) INT Q;
ENDDATA;

PROC SUMMATION () INT;
   % FORMS AND RETURNS SUM OF ELEMENTS OF Q IN A SIMPLE LOOP %
INT SUM,    % FORM SUM OF ELEMENTS %
    I;      % ELEMENT COUNTER %
   SUM:=0;
   I:=1;
NEXTEL:
   SUM:=SUM + Q(I);
   IF I>100 THEN
      % FINISHED %
      RETURN(SUM);
   END;
   I:=I+1;
   GOTO NEXTEL;
ENDPROC;
```

Example 4:

Write a procedure to rearrange the elements of a real array into ascending numerical order.

Let us take an array named A of 20 elements; the logic of the solution is to seek the name of the minimum element and swap the contents of this location with A(1) [This may of course be a swap with itself]. We repeat this process with the remaining 19 elements using A(2) and so on until the array is correctly ordered. Two loops are required [counted by I and J] as shown in the flowchart.


ENTER
SET I to 1
Assume A(I) contains current minimum
Set J to I + 1
IS A(J)<MIN ?
✓
X
Put name A(J) into minimum
increment J
J⩽20?
✓
X
Swap current minimum into A(I)
Increment I
I<20?
✓
X
EXIT

Noting that the minimum is a name and must therefore be contained in a ref-real variable we can write the corresponding RTL/2. (This is not the only method; we could for instance maintain an integer containing the subscript for the current minimum element).

```
DATA GLOBAL;
   ARRAY (20) REAL A;
ENDDATA;

PROC ORDER ();
% PUTS ARRAY A INTO ASCENDING NUMERICAL ORDER %
INT I,J;    % LOOP COUNTERS %
REAL TEMP;  % USED IN SWAPPING %
REF REAL MIN;  % NAME OF CURRENT MINIMUM %

         I:=1;          % INITIALISE ELEMENT LOOP %
NEXTEL:  MIN:=A(I);     % ASSUME A(I) IS CURRENT MINIMUM %
         J:=I+1;        % INITIALISE SEARCH LOOP %
CHECK:   IF A(J) < MIN THEN       % NOTE DEREFERENCING OF MIN %
            % NEW MINIMUM %
            MIN:=A(J);
         END;
         J:=J+1;        % NEXT ELEMENT INDEX FOR SEARCH %
         IF J<=20 THEN GOTO CHECK;  END;
         % SEARCH COMPLETE FOR THIS ELEMENT; PERFORM SWAP %
         TEMP:=A(I);
         A(I):=MIN;
         VAL MIN:=TEMP;
         % NEXT MINIMUM NOW IN A(I) %
         % OLD CONTENTS NOT LOST %
         I:=I+1;        % NEXT ELEMENT %
         IF I<20 THEN GOTO NEXTEL;  END;
         % NOTE NO NEED TO CHECK THE LAST ELEMENT A(20) %
ENDPROC;
```

This is all very well, but when we call SUMMATION or ORDER, we always operate on the arrays Q and A respectively. In practice we want routines which will operate on different arrays on different calls. Arrays, however, are static, so we cannot declare an array parameter for the procedures – remember that parameters are dynamic variables. When faced with the problem of affecting our parameters earlier, we used a ref-variable in order that a particular name could be made available to a procedure at the time of the call. We solve our array difficulty in a similar way, by defining new modes (further ref-variables) which can contain the names of arrays. Just as different modes ref-real and ref-integer were necessary to hold the names of real and integer variables, we must have different modes to contain the names of different sorts of arrays:

```
REF ARRAY REAL A;    % DEFINES A VARIABLE TO CONTAIN THE NAME OF  %
                     % A REAL ARRAY %
REF ARRAY INT N,P;   % CONTAIN NAMES OF INTEGER ARRAYS %
```

The declarations specify as usual the names of the variables being declared and their modes: in this case their contents will be names of arrays of the mode indicated. That is all; note that the bounds of the arrays are not required.

We can assign to such variables using the usual considerations:

```
DATA GLOBAL;
   ARRAY (3) INT SMALL;
   ARRAY (300) INT BIG;
ENDDATA;

PROC SOMENAME ();
REF ARRAY INT P;
   P:=SMALL;
   P:=BIG;
ENDPROC;
```

The left-hand sides of the assignment statements are the names of variables and therefore valid destinations; they require the names of arrays of integers; the right-hand sides *are* the names of integer arrays and therefore deliver acceptable objects. The bounds of the arrays are not involved. Note, however, that the analogy with other ref-variables is not complete. We cannot write

```
    VAL P := SMALL;
```

and expect all the contents of the elements of the array SMALL to be copied into the elements of the array whose name is currently in P (there would be a bound problem here!). Such "whole array" operations are not allowed; the name of an array is not a valid destination since it does not identify a unique place, so statements of the form BIG := SMALL are also illegal.

What we can manipulate via ref-array variables are the elements of the arrays whose names are held in such variables. To do this we simply append a subscript to the name of the ref-array variable:

```
    P(I + J)
```

P is a ref-array variable; a subscript applied to it is meaningless, just as + applied to a ref-real is meaningless. We interpret this as meaning apply the index given by the subscript to the *contents* of P; that is, we dereference P and use the name of the array contained in it to yield, with the index, the name of a variable which then behaves in the normal way . That's all! The dereferencing is always automatic, since no other sensible interpretation can be put on the construction.

In both examples 3 and 4 we used the number of elements in the arrays explicitly. Clearly, if no information about the number of elements involved is contained in the declaration of a ref-array variable, we can no longer do this when we write general procedures to operate on arrays – the number of elements involved will differ from call to call.

To cope with this we introduce a new monadic operator:

OPERATOR	OPERAND	RESULT	INTERPRETATION
LENGTH	Array name	Integer	No. of elements in the array

LENGTH is a keyword, and the only operator amongst the monadics and dyadics which operates on a name rather than a value. Note that the mode of the elements of the array operand is immaterial.

For our example above, LENGTH BIG will yield the integer value 300; the use of LENGTH with an actual array name as in this case is somewhat redundant, since the answer can always be obtained by consulting the declaration. LENGTH will generally be used to ascertain the length of an array whose name is contained in a ref-array variable; naturally the ref-array variable will be dereferenced to yield an array name as a suitable operand for LENGTH. As the result is an integer, the operator may occur quite generally in an expression, and all the usual rules governing monadic operators and widening and narrowing apply:

```
    REF ARRAY REAL A;
    INT I;
    I := LENGTH A + 3;
```

We can now rewrite our two examples with ref-array parameters:

```
PROC SUMMATION (REF ARRAY INT Q) INT;
INT SUM,I;
   SUM:=0;
   I:=1;
NEXTEL:  SUM:=SUM+Q(I);
            % Q DEREFERENCED THEN THE NAME OBTAINED BY INDEXING %
            % THE CONTENTS OF Q ALSO DEREFERENCED %
         I:=I+1;
         IF I>LENGTH Q THEN RETURN(SUM);  END;
         GOTO NEXTEL;
ENDPROC;


PROC ORDER (REF ARRAY REAL A);
INT I,J;
REAL TEMP;
REF REAL MIN;
         I:=1;
NEXTEL:  MIN:=A(I);
         J:=I+1;
CHECK:   IF A(J)<MIN THEN MIN:=A(J);  END;
         J:=J+1;
         IF J<=LENGTH A THEN GOTO CHECK;  END;
         TEMP:=A(I);
         A(I):=MIN;
         VAL MIN:=TEMP;
         I:=I+1;
         IF I<LENGTH A THEN GOTO NEXTEL;  END;
ENDPROC;
```

Not much change! Except that, each time Q or A occurs, it will now be dereferenced to yield an array name. Make sure you understand all the dereferencing in these examples; for instance the statement A(I) := MIN involves four dereferences: A is dereferenced to give an array name since a subscript follows; I is dereferenced to give an integer which is used to index the array name thus giving a real destination; MIN must then be dereferenced twice to deliver a real number.

What can we have arrays of? So far we have seen real arrays and integer arrays. We can also declare arrays of ref-reals and ref-ints; each element is now a ref-variable which can contain the name of a variable of the appropriate mode. The name of an element of such an array is formed from the array name and a subscript and this element behaves just as a ref-variable:

```
DATA S;
   ARRAY (10) REAL AR;
   ARRAY (20) INT AI;
   ARRAY (3) REF REAL T;     % NOTE THAT THE MODE OF EACH ELEMENT %
                             % FOLLOWS THE LENGTH INFORMATION %
   ARRAY (7) REF INT U;
ENDDATA;

PROC MAIN ();
      .
      .
   T(2):=AR(7);          % NAME OF ELEMENT AR(7) %
   AI(17):=U(6);         % DEREFERENCE U(6) TWICE TO YIELD INTEGER %
   VAL T(1):=0.3;        % FORCE DEREFERENCING OF T(1) TO YIELD %
                         % A REAL VARIABLE DESTINATION %
ENDPROC;
```

Of course we can also declare references to arrays of ref-variables. Note here carefully though, that VAL is only needed to force dereferencing of the individual element; dereferencing of the ref-array variable is implicit in the use of a subscript.

```
DATA S;
   ARRAY (10) REAL AR;
   ARRAY (20) INT AI;
   ARRAY (3) REF REAL T;
   ARRAY (7) REF INT U;
ENDDATA;

PROC MAIN ();
REF ARRAY REF REAL A;
REF ARRAY REF INT B;
A:=T;             % NAME OF ARRAY OF CORRECT MODE %
                  % NOTE A:=AR ILLEGAL SINCE A DEMANDS NAME OF ARRAY %
                  % OF REF-REALS %
   B:=U;
   A(1):=AR(6);   % A DEREFERENCED AUTOMATICALLY; NAME AR(6) STORED %
   AI(2):=B(3);
      % AI(2) DEMANDS AN INTEGER; B IS DEREFERENCED TO YIELD AN ARRAY %
      % OF REF-INTS WHICH IS INDEXED BY 3 TO GIVE A REF-INT. THIS IN %
      % TURN IS DEREFERENCED TWICE TO YIELD THE REQUIRED INTEGER %
   VAL B(1):=7;
      % DEREFERENCING OF B IS AUTOMATIC; VAL APPLIES TO THE REF-INT %
  % VARIABLE OBTAINED BY INDEXING THE CONTENTS OF B BY 1 %
ENDPROC;
```

What other modes have we had? Arrays.

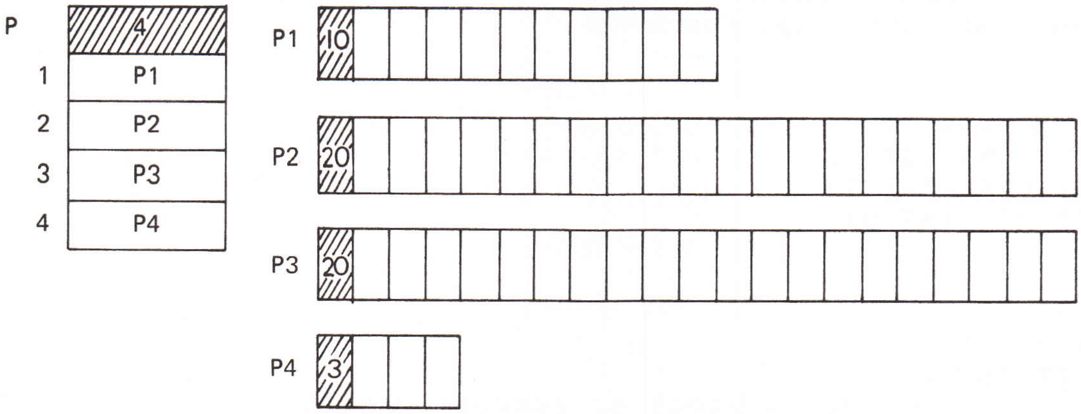
Can we have arrays of arrays then? No; if we could, each element would not be a variable, but a structure consisting of many variables. However, we can declare an array whose elements are ref-array variables; that is each element can contain the name of an array. As usual the mode of the elements of such arrays must all be the same.

Consider the following:

```
DATA S;
 ARRAY (10) INT P1;
 ARRAY (20) INT P2,P3;
 ARRAY (3) INT P4;
 ARRAY (4) REF ARRAY INT P;
ENDDATA;

PROC SETP ();
 P(1):=P1;
 P(2):=P2;
 P(3):=P3;
 P(4):=P4;
ENDPROC;
```

An element, such as P(1), is a ref-array-int variable and as the destination of an assignment demands the name of an integer array; P1 is such an acceptable name.

What sort of structure have we created with P?



It is basically a two level structue; the actual integer variables require two levels of indexing from the name P. To name the 17th element of P3 using P we need to write:

```
P(3)(17)
```

The index 3 on P gives a ref-array-int variable; the following subscript forces dereferencing to give the name of an array (in this case P3) which is indexed by 17 to give the name of an integer variable. This then behaves just as an ordinary integer variable.

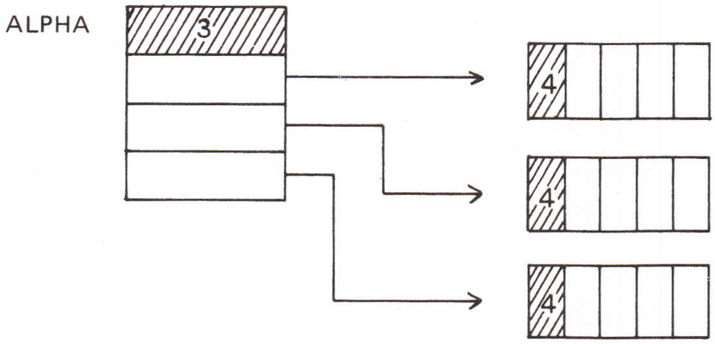
Such structures can be used to represent any organised data with a two-dimensional structure, e.g. matrices. In a matrix, we normally have the same number of elements in each row vector, unlike the differing lengths in our structure P. In RTL/2 we allow direct declaration of such two-dimensional structures:

```
ARRAY (3,4) INT ALPHA;
```

This declaration is equivalent to declaring

```
ARRAY(3)REF ARRAY INT ALPHA;
```

and assigning to each element the names of integer arrays each of length 4. Such a structure will be created, but no intermediate names (in the RTL/2 sense) will exist.



[The arrows indicate
some internal
naming mechanism]

If we make any assignments to the elements of the first level of ALPHA we destroy the structure generated by the compiler. For example if we write ALPHA(1) := P2; we can only regain the original rectangular two-dimensional structure if we have previously remembered the old contents of ALPHA(1) in a suitable ref-array-int variable.

The word dimension refers to the number of bounds possessed by the array and not to the extents of the vectors which compose it. Thus ALPHA has dimension 2, and:

LENGTH ALPHA is 3

whilst LENGTH ALPHA(1) is 4 (since ALPHA(1) contains the internally generated name of an array of four elements). We may name the individual elements in precisely the same way, ALPHA(2)(3) is the name of an integer variable found by indexing the contents of ALPHA(2).

Following the normal algebraic practice we also allow the forms ALPHA (2,3) and P(3,17).

As usual, the subscripts can be any expressions provided they deliver integer values within the bounds of the array.

Naturally, if we wish to pass such a two-dimensional array as a parameter to a procedure, we need to be able to declare a reference to such a structure. A variable suitable to contain the name of a structure such as ALPHA must have the mode REF ARRAY REF ARRAY INT; we allow the shortened form REF ARRAY(,)INT, the comma in the empty bound list indicating that it is a reference to a 2 dimensional structure. Our earlier declarations could also have shown the dimension involved explicitly:

```
REF ARRAY() REF REAL A;
REF ARRAY() REF INT B;
```

We can extend this to construct arrays of as many dimensions as we desire (though in practice a compiler may impose an upper limit) and suitable ref-variables to contain their names. In the ref-variable case, the number of commas required in the empty bound list is equal to one less than the dimension of the array contents; in the simple one-dimensional case, the "list" is optional. Provided you think carefully of the sort of variable involved at each stage of the subscripting and the dereferencing involved, you will have no difficulty – there are no new concepts involved. For instance, we could declare (in a data brick) a three dimensional array and wish to order one of its constituent vectors using our procedure ORDER:

```
DATA SS;
 ARRAY (5,6,7) REAL DIM3;
ENDDATA;

PROC MAIN ();

 ORDER ( DIM3(3,4) );
      % CONVINCE YOURSELF THAT THE PARAMETER DOES INDEED YIELD THE
      % NAME OF A REAL ARRAY %
ENDPROC;
```

You will observe that because of the intermediate levels required, it is more efficient to arrange (if possible) for the bounds to occur in ascending numerical order.

As a simple example of the use of two-dimensional arrays, we will write a procedure to return the trace (the sum of the diagonal elements) of a square matrix presented as a parameter.

```
PROC TRACE (REF ARRAY (,) REAL A) REAL;
   % WE ASSUME THAT THE PARAMETER IS A SQUARE ARRAY, THAT IS %
   % LENGTH A = LENGTH A(1) %
REAL SUM;   % FORM TRACE %
INT I;      % COUNTER %
   I:=1;
   SUM:=0.0;
NEXTEL:  SUM:=SUM + A(I,I);       % ADD IN NEXT DIAGONAL ELEMENT %
         I:=I+1;
         IF I <= LENGTH A THEN GOTO NEXTEL;  END;
         RETURN(SUM);
ENDPROC;
```

Because the left-hand side of an assignment statement can now contain array element names which may be derived from complex subscript expressions (involving procedure calls which may affect data variables or other variables via the parameter mechanism) we must now define explicitly the order of actions (mentioned in section 4) in assignment, and, indeed, in the order of evaluation of the subscript expressions themselves. We would need to be sure of this, if (for some reason) we wished to write something of the form:

```
DATA ANYDATA;
    ARRAY (8,10,20) INT BIG;
    INT K;
ENDDATA;

PROC SET (REF INT X) INT;
    VAL X:=X+1;
    RETURN(7);
ENDPROC;

PROC MAIN ();
INT J;
    K:=3;   J:=1;
    BIG(J,K,SET(K)):=BIG(SET(J),K,J):=SET(J);
% A COMMENT WOULD BE ADVISABLE HERE - IN GENERAL SUCH OPAQUE %
% PROGRAMMING IS NOT RECOMMENDED %
ENDPROC;
```

At first glance you might expect this statement to be equivalent to BIG(1,3,7) :=BIG(7,4,2):=7; with J,K containing the values 3,4 respectively.

This is not the case; the rules are as follows:

i) An array and its subscripts are evaluated from left to right. [Note that the array must be included in this, since the name may be obtained from a ref-variable whose contents could be changed by a function call in one of the subscript expressions: this rule says that the array name will be evaluated before its subscripts].

ii) In an assignment, the right-hand side is evaluated first; the destinations are evaluated and the assignments made from right to left.

Thus in our example the following sequence will be obeyed:

evaluate SET(J): this yields 7 and resets J to 2
evaluate BIG(SET(J),K,J): this indexing process from left to right will yield the name BIG(7,3,3) and reset J to 3
evaluate BIG(J,K,SET(K)): this yields the name BIG(3,3,7) and resets K to 4

The overall effect therefore is to perform

BIG(3,3,7) := BIG(7,3,3) :=7

with J, K containing 3, 4 respectively.

The rules describe the action of any assignment statement unambiguously provided we add the further information about any procedures involved:

iii) The parameters are evaluated from left to right before examining and calling the procedure itself [The reason for including the procedure itself will become apparent later].

It is important to realise that we only need worry about these considerations if we are using procedures which have side effects on data variables or through the parameter mechanism, and then only if we are using those variables affected elsewhere in our statement.

Note that the order of evaluation of the terms forming the operands of a dyadic operator was not defined; thus $\propto\Box\beta$ may result in β being evaluated before $\propto$. Again this will only lead to ambiguity if the evaluation of one term affects the other via side-effects. Since the order is not defined such constructions should be avoided (see also page 29).

Section 14 examples

1 The co-ordinates of a point in space are held in an array of length 3. Write a procedure to find its distance from the origin.

$(\sqrt{(x^2 + y^2 + z^2)})$

2 Write a procedure to calculate the mean and standard deviation of a set of numbers, supplied via an array parameter.

3 Write a procedure to form in a third array the matrix product of two given arrays. (An element of the third array is given by

$C_{ij} = \sum_k A_{ik} B_{kj}$)

4 In two arrays X, Y set up the values of x and some function of x for a given range of values. Use this table of values to calculate, by linear interpolation, the value of the function for a given x in the range.

15. Let

In a large program there may be many arrays all having the same length, and a need to use this length explicitly at various points in the program. Should the specification of the program change, or the length change for some other reason, there will be many places where the integer value will need to be amended: it is almost certain that some occurrences will be missed in this amendment process, and this error may not be detected for some time, since the program will still be syntactically correct. One solution would be to put this value in a variable (kept unchanged throughout the program) but all the arrays would still need the explicit integer in their declarations.

RTL/2 has a facility which is of particular use in such a situation. We define a sequence of text which is to be associated with a name (formed by the usual rules). This definition occurs outside our bricks and takes the form of the keyword LET followed by the name, an equals sign, the sequence we are defining, and a semi-colon to terminate it. For example:
LET Z = 16;

On every subsequent occurrence of the name (in this case Z) in the program, the sequence (16) will be inserted in its place. This is, therefore, a mechanism of textual replacement which is performed by the compiler whilst reading a program, Thus:

```
LET NO=75;        % NUMBER OF ELEMENTS %

DATA S;
   ARRAY (NO) INT A;
   ARRAY (NO) REAL B;
ENDDATA;

PROC MAIN ();
INT I;
      .
      .
      IF I<=NO THEN
         A(I):=7 + A(NO-I);
         B(NO-I):=0.01*B(I);
      END;
      .
      .
ENDPROC;
```

is equivalent to the program:

```
DATA S;
   ARRAY (75) INT A;
   ARRAY (75) REAL B;
ENDDATA;

PROC MAIN ();
INT I;
      .
      .
      IF I<=75 THEN
         A(I):=7 + A(75-I);
         B(75-I):=0.01*B(I);
      END;
      .
      .
ENDPROC;
```

However, if the number of elements changes at some later date, we only need to amend the first version in one place — the definition of NO — rather than in (at least) five places.

The textual replacement specifies exactly what happens; syntactically, the definition sequence must be valid wherever we use the name. If we write:
LET P = 6 + 7;
then 6 + 7 (and not 13) is what we get for each occurrence of P. If we attempt to put
ARRAY(P)INT. . .
somewhere in our program, it will fail to compile since we have not supplied a simple integer constant as the array length.

It is vital to realise that we have not declared a variable P; no cell named P exists at run time. This explains why = is used and not the assignment symbol. P is a name used as a shorthand, which usefully concentrates changes of information in one spot. As we have seen this can simplify re-definition of lengths of data structures. Similarly it can be used to represent other numerical constants:

```
LET PI= 3.14159;
LET INSTOCM=2.54;     % CONVERSION FACTOR INCHES TO CENTIMETRES %
```

This use is strongly recommended as it ensures that the same value is used throughout and minimises the chances of typographical errors. It also means that changes of accuracy can be made at one point, particularly when transferring to another machine.

Another use is to give a name to a fixed element of an array. An array of six elements might be used to store the current values of the co-ordinates of a point in a fluid, its density, temperature and pressure, the array being used so that all the information could be passed to various procedures by a single parameter. For clarity and documentation we could give names to the various elements; these names would be replaced by the appropriate array element on reading the program:

```
LET XCOORD  = PLACE(1);
LET YCOORD  = PLACE(2);
LET ZCOORD  = PLACE(3);
LET DENSITY = PLACE(4);
LET TEMP    = PLACE(5);
LET PRESS   = PLACE(6);

DATA FLUIDINFORMATION;
   ARRAY (6) REAL PLACE;
      .
      .
ENDDATA;

PROC SET ();
      .
      TEMP:=10.6;     % SETS PLACE(5); %
      .
ENDPROC;
```

As the facility is one of replacement on reading a program, such replacement of the name will only occur in that part of the program following the definition. We can, if we wish, re-define the name to be some other sequence and this definition will be used to replace the name in subsequent parts of the program. This can be useful for documentation purposes and for preserving similarity between various sections of program.

```
LET ERRORACTION=FAIL1;  % FAIL1 IS PROC GIVING MESSAGE AND RESTART %

PROC P1 ();
      .
      .
      IF ....... THEN ERRORACTION ();  END;
      .
      .
ENDPROC;

LET ERRORACTION=FAIL2;  % FAIL2 IS PROC GIVING MESSAGE ONLY %

PROC P2 ();
      .
      .
      IF .... THEN ERRORACTION ();  END;
      .
      .
ENDPROC;
```

The exact requirements for ERRORACTION in the various cases may have been decided after the initial program was written. The use of the LET facility ensures that all occurrences of ERRORACTION are replaced correctly; again if the specification changes, the textual changes required are localised.

What sort of things can we include in such definitions? Between the equals sign and the semi-colon we can write any sequence of the items we have encountered so far, that is names, real and integer constants, comments (though this is unlikely to be of much use since they are removed anyway!) and separators (but not the semi-colon of course!). However, we cannot make further definitions within a LET definition – hence the keyword LET is illegal in the sequence.

Since one can put almost any sequence of items in the definition, some very devious definitions could be made:

```
LET GUESS= IF A=B THEN ;

DATA S;
   INT A,B;
ENDDATA;

PROC MAIN ();
INT WHAT;
      .
      .
      GUESS WHAT := 6;  END;
      .
ENDPROC;
```

This is not transparent programming, not well documented and extremely error prone! The LET facility is not intended for this sort of use.

We can use, in a LET-definition, a name that has itself been defined by a LET-definition. This name has been defined and so, on reading it, the compiler will replace it with the appropriate sequence.

```
LET EL=5;
      .
      .
LET TEMP=PLACE(EL);  % EQUIVALENT TO PLACE(5) %
```

Note that this replacement is made once in the definition and not on each occurrence of TEMP;

```
LET EL=5;
       .
       .
LET TEMP=PLACE(EL);
       .
       .
LET EL=7;
       .
       .
       TEMP:=10.7;
       .
```

will result in the statement PLACE(5) :=10.7; since PLACE(5) is the current definition of TEMP at this point, even though EL is now defined to be 7.

16. Data Initialisation

It would be extremely tedious, if we wished to ensure that a large array contained certain values at the beginning of our program, to have to write out a vast number of assignment statements:

```
    A(1) := 7;
    A(2) := 16;
        .
        .
        .
    A(300) := 0;
```

The instructions would also occupy a large amount of store.

We introduce at this point the concept of *initialising* variables in a data brick. This is a facility which allows the programmer to specify the contents of his data brick (static) variables on entry to the program.

To do this, clearly we must be able to calculate the required contents of these variables at compile time; hence expressions and any dereferencing or type-changing is forbidden – we are restricted to using "constant" information of the correct mode.

For a simple variable we have to supply one piece of information, namely the required contents. This is specified by an assignment-like sequence within the declaration containing the definition of the variable.

```
DATA INITIALISED;
   REAL A:=1.0,B:=-3.2,C:=D:=+0.7E-1,E,F,G:=-6E7;
   INT I:=1,J:=K:=L:=-30,M:=N:=+3,P,Q:=0;
ENDDATA;
```

From this example we can deduce the general rules:

i) Variables to be initialised are followed by := and the required initial value; "multiple initialisation" is allowed in a similar way to multiple assignment.

ii) The initial value may be a signed constant; it must be of the correct mode.

iii) Groups of names and initialisations are separated by commas ; note that variables without initial values may be included, in the normal way, in the list of names.

Static arithmetic variables which are not initialised in the program text will be initialised by default to zero (a zero of the correct mode!). Thus on entry to our program, E, F above will contain 0.0 and P will contain 0. Q need not have been explicitly initialised to zero, but it is not wrong, and such an initialisation is good and useful documentation.

When it comes to initialising ref-variables, the "constants" we can have are the names of variables of the appropriate mode. We have mentioned earlier that before a ref-variable is used in an expression, we must ensure that it contains something sensible, or else on dereferencing twice we shall have no idea what location we shall be accessing. The wisdom of initialising such variables is forced onto the programmer : all ref-variables must be initialised (some earlier examples are illegal in this respect) and hence there is no question of any default value. As before, the syntax consists of assignments embedded in the declarations.

```
DATA MORE;
   REAL A,B;
   INT I:=J:=K:=3;
   ARRAY (10) INT LIST;
   REF REAL WHICHREAL:=A,YOU:=ME:=B;
   REF INT PLACE:=K,WHERE:=LIST(2);
   REF ARRAY INT POINTER:=LIST;
ENDDATA;
```

We must remember that no local variables are available at this point – they only exist whilst the call of a procedure is being executed. Variables whose names are used as initial values in data bricks must therefore belong to some data brick; that is, they must be static so that their location is known at compile-time.

An array element is such a simple variable, but only if the subscript is a constant (again so that the exact location is known at compilation). Hence initialising WHERE to LIST(2) is quite legitimate; an initial value of LIST(I) (even though I has been initialised to 3) would be illegal as dereferencing of I is required. Similarly an element of a two (or more) dimensional array A(3, 4) is illegal since an implicit dereferencing is required to find the name in A(3) before indexing it by 4.

Now we can consider our opening problem – how to initialise an array. An array consists of a named structure containing many simple variables; to initialise it we must therefore supply a set of the appropriate "constants". This is done by writing a list of values separated by commas and enclosed in brackets as the initial value for the array. The list must contain exactly the right number of constants for the number of elements.

```
DATA LISTS;
   INT I,J,K;
   ARRAY (3) REAL A1:=(1.0,2.0,3.0);
   ARRAY (10) INT I1:=I2:=(1,2,3,4,5,6,7,8,9,10),
                  I3:=(20,30,0,0,0,0,0,0,0,0),
                  I4;
   ARRAY (2) REF INT RI:=(K,J);
ENDDATA;
```

For arrays of simple variables, the same default rules apply; hence I4 above will be initialised to ten zeroes.

In the case of I3 we only wished to initialise the first two elements and leave the rest zero. It is not too bad to have to write (and count) the necessary eight zeroes and the commas. For a large array, however, much writing would be laborious and error prone; the requirement to do this would probably deter people from using the initialisation facility in such a case, relying on the default value and filling in the first few elements by explicit assignments in the program. Where a single value is required for many elements of an array a *repitition factor* may be used to specify the number of elements to be initialised to this value. A repetition factor consists of an integer constant in brackets following the initial value. Thus we would initialise I3 by writing

    ARRAY (10) INT I3: = (20, 30, 0(8));

This facility can be used for any number of elements in any position in the array and for any required value – though the rules for the mode of the value and the total number of elements must be obeyed of course.

Example:

```
ARRAY (15) REAL ALPHA:=(1.0(2),3.1(3),0.0,6.1,7.2(2),1.0E1(5),27.2);
ARRAY (7) INT BETA:=(7(2),1(3),6,5);
```

will declare arrays which on entry to the program will appear as:

ALPHA
15
1.0
1.0
3.1
3.1
3.1
0.0
6.1
7.2
7.2
1.0E1
1.0E1
1.0E1
1.0E1
1.0E1
27.2

BETA
7
7
7
1
1
1
6
5

The LET facility can be useful in such situations:

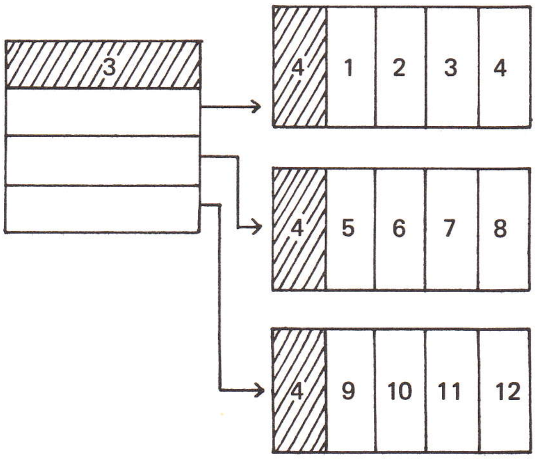
```
LET NOEL=100;            % NUMBER OF ELEMENTS %
LET NOELLESS2=98;        % SET TO NOEL-2 %
   % WHEN NOEL IS CHANGED WE MUST ALWAYS CHANGE NOELLESS2 : THIS  %
   % MAKES CHANGES EASY TO IMPLEMENT %

   DATA INF;
      ARRAY (NOEL) INT GAMMA:=(1(NOEL));
      ARRAY (NOEL) REAL DELTA:=(0.01,0.07,0.0(NOELLESS2));
   ENDDATA;
```

When we come to multi-dimensional structures, we must be a little more careful.

When we declare explicitly a two dimensional array, we are creating all the levels, and so can present the initial values as an array of array initial values and the syntax reflects this.

```
DATA NONAME;
   ARRAY (3,4) INT BIG:=
         ( (1,2,3,4), (5,6,7,8), (9,10,11,12) );
ENDDATA;
```

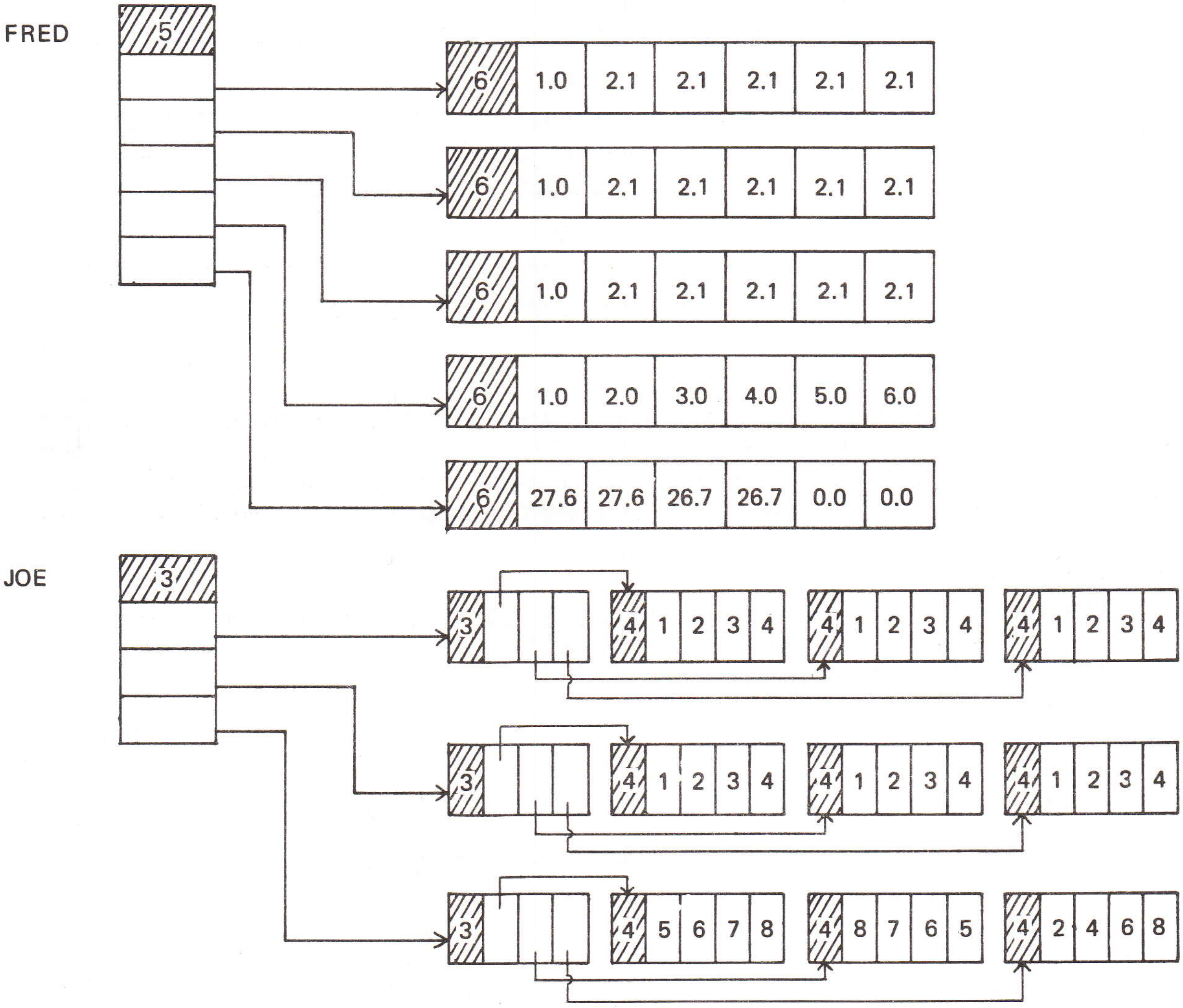
This declaration creates the structure:



We may use repetition factors to repeat a particular array of values, and the scheme may be extended to more dimensions:

```
DATA MULTIDIM;
    ARRAY (5,6) REAL FRED:=
        ( (1.0,2.1(5)) (3),
          (1.0,2.0,3.0,4.0,5.0,6.0),
          (27.6(2),26.7(2),0.0(2))    );
    ARRAY (3,3,4) INT JOE:=
        (  (   (1,2,3,4)(3)   )(2),
           (   (5,6,7,8), (8,7,6,5), (2,4,6,8)   )   );
ENDDATA;
```

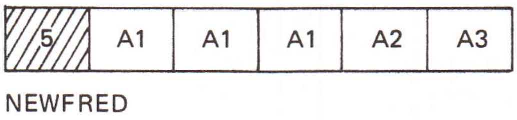
This will create the following structures:



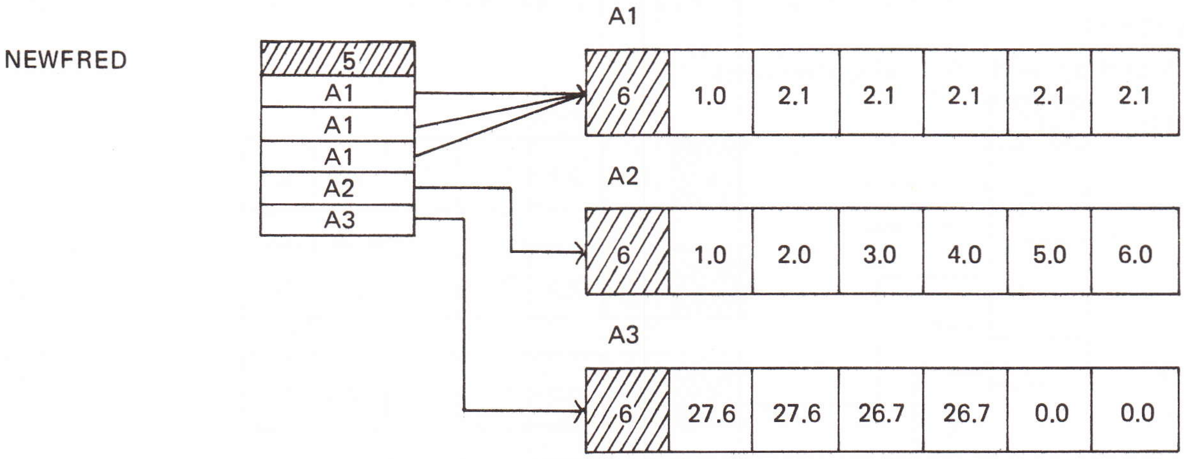
When, however, we are setting up a two-dimensional structure by the use of an array of references, we are only declaring a single level, and hence can only initialise that level to a set of names of appropriate arrays (these arrays may be initialised in their own declarations of course).

```
DATA STRUCT;
    ARRAY (5) REF ARRAY REAL NEWFRED:=(A1,A1,A1,A2,A3);
        % COULD USE REPETITION FACTOR AND WRITE (A1(3),A2,A3) %
    ARRAY (6) REAL
        A1:=(1.0,2.1(5)),
        A2:=(1.0,2.0,3.0,4.0,5.0,6.0),
        A3:=(27.6(2),26.7(2),0.0(2));
ENDDATA;
```

Note that the structure NEWFRED consists of an array of 5 names:



The total structure thus created is *not* the same as our earlier FRED. Not only are the final array levels named, there are not so many of them! In general also, they need not be of the same length.



Initialisation, apart from saving a large number of assignments to be performed initially at run time, and ensuring that ref-variables contain sensible names, is also useful for setting up tables of values, decision tables, look-up tables etc. For instance, in our DDC example, we may use different (say temperature) setpoints at various stages of the process and have a 'stage number' at any moment:

```
DATA PLANT;
   ARRAY (5) REAL TEMPS:=(150.0,180.7,230.0,195.3,100.0);
         % TEMPERATURE SET POINTS %
   INT STAGENUMBER;
ENDDATA;

PROC DDC ();
      .
      .
      SETPOINT:=TEMPS(STAGENUMBER);    % SIMPLE LOOK-UP %
      .
      .
ENDPROC;
```

Section 16 example

1. Write a procedure which will be called at 00 hours each day in a real-time system to update the values of integers DAY, MONTH and YEAR held in a suitable data brick. (Note that you must allow for the various lengths of the months but you may ignore the problem of leap years if you wish).