Introduction to Systematic Programming
Unit 3 - Expressing Algorithms as Ada Programs

3.1 The nature of values - types

The general category 'number' may be subdivided into two important classes:

> whole numbers which might be either positive or negative, such as –27, 0, 10037, –45, etc. and which are known as **integers**
>
> 'decimal' numbers which may have a fractional part such as 11.25, 3765.0, 1.414, –1.3333, etc. and which are known as **real** numbers.

Many real numbers such as π = 3.1415926..., √2 = 1.4142..., 4/3 = 1.3333..., etc. have non-terminating decimal expansions. Clearly it is not possible to store these values exactly in a computer and so in computer science, real values should normally be regarded as approximations only, rather than exact values. In Sun Ada real numbers are usually stored to about 7 significant decimal digits accuracy. As the total number of digits is fixed, the number of decimal places (ie. digits after the decimal point) will vary depending on the size of the number, for example 1234.567, 1.234567, 0.01234567, 123456.7, etc, and it is for this reason that real values are often referred to as **floating point** values.

Although most of the values involved in computing are numbers, other types of value are possible as well, eg. **truth** values - we can speak of a condition such as `Size>0` yielding the **value** `True` when the value stored in the variable `Size` is positive or the **value** `False` otherwise. Equally it is possible to have **textual** values, often referred to as **strings**, such as names: `"Richard"`, and addresses: `"Aston University, Aston Triangle, Birmingham"`, etc.

In Ada terminology, what kind of value we are talking about - integer, real, etc. - is known as the **type** of the value. We shall see that this concept is very important in writing Ada programs, since variables can only hold values of a particular type. Thus, for example, a variable which can only hold integer values is said to have type `Integer`. Similarly a variable which can only hold real floating point values is said to have type `Float`.

3.2 Libraries for Input and Output

The steps necessary for performing input and output in Ada are not automatically available to a program, but have to be made available from elsewhere. These **input/output (I/O) procedures** (`Get`, `Put`, etc.) are contained within library packages which are stored in the computer system; in Ada a **package** is a program unit whose purpose is simply to provide program elements (such as I/O procedures) to be used in other programs.

Just as it is essential to have variables of different types to hold real and integer values so it is necessary to have different procedures to input and output values of different types. In reality there are several versions of the output procedure `Put` each imported from a different library package:

for outputting integer values `Put` from the library package	`CS_Int_IO`	is used,
for outputting floating point values `Put` from the library package	`CS_Flt_IO`	is used,
for outputting textual values `Put` from the library package	`Ada.Text_IO`	is used.

Similarly for input:

for inputting integer values `Get` from the library package	`CS_Int_IO`	is used,
for inputting floating point values `Get` from the library package	`CS_Flt_IO`	is used.

When it is necessary to distinguish explicitly between the different versions of an I/O procedure from the various library packages, the procedure name can be **fully qualified** by preceding it with the package name to which it belongs and a full-stop, for example `CS_Int_IO.Put`, `CS_Flt_IO.Put` and `Ada.Text_IO.Put`. However in most cases the short form of the procedure name will suffice.

3.3 Converting an algorithm into an Ada program

In principle, the Ada-style notation for algorithm steps which was introduced in Unit 2 can be used directly in writing Ada programs; the forms available being `IF...` to express selection, `WHILE...` to express repetition, `...:=...` to express assignment, `Get(...)` to express input, and `Put(...)` to express output, etc. However, programming languages always impose strict rules about exactly how programs are presented. This is for a good reason - to avoid any risk of ambiguity and to simplify and hence speed up the work of computer systems - but it does mean that to convert our algorithms into correct Ada programs executable by a computer, we shall have to learn and rigorously apply a number of rules.

To see what's involved let's look at a very simple problem - given two positive whole numbers, determine the remainder when the first is divided by the second. We shall compare the algorithm to solve this with its expression as an Ada program:

Algorithm

One of the ways to solve this problem is to use repeated subtraction to find the remainder, ie. keep subtracting the second number from the first until no more subtractions can take place (note that this algorithm isn't the 'best' approach, but it will suffice for the present), thus:

```
Get(Item => Number)
Get(Item => Divisor)

WHILE Number ≥ Divisor LOOP
    Number := Number - Divisor
END LOOP
{ When the loop terminates }
{ Number holds required remainder}


Put(Item => "Remainder is ")
Put(Item => Number)
```

Ada Program

```
WITH Ada.Text_IO; USE Ada.Text_IO;
WITH CS_Int_IO;   USE CS_Int_IO;

-- Program to find a remainder
-- by repeated subtraction.

PROCEDURE Remainder IS

   Number   : Integer;
   Divisor  : Integer;

BEGIN
   Put(Item => "Type 2 positive ");
   Put(Item => "whole numbers: ");
   Get(Item => Number);
   Get(Item => Divisor);

   WHILE Number >= Divisor LOOP
      Number := Number - Divisor;
   END LOOP;
   -- When the loop terminates
   -- Number holds the remainder.

   New_Line;
   Put(Item =>"Remainder is ");
   Put(Item => Number);
   New_Line;
END Remainder;
```

To get from the algorithm to a complete Ada program, we had to follow these rules:

i) Make available all the procedures from the library `Ada.Text_IO` which are used for pe rforming output of text:

```
WITH Ada.Text_IO;  USE Ada.Text_IO;
```

ii) Similarly, make available the procedures from the library `CS_Int_IO` which are used f or performing input and output of integer (whole number) values.

iii) Next insert a program heading:

```
PROCEDURE Remainder IS
```

iv) **Declare** the names and types of all the variables used in the program. In this case th ere are just two integer variables:

```
            Number  : Integer;
            Divisor : Integer;
```

v) Form the algorithm steps into a **program body** by :

   preceding them with the keyword `BEGIN` to indicate the start of the program text proper;

   marking the conclusion of the program text by adding the keyword `END` followed by the na me of the program:

```
END Remainder;
```

   Note that this name must be exactly the same as that which appears in the program heading.

vi) Terminate each complete program step by a semi-colon (`;`). This also applies to the ad ditional **specification steps** introduced for (i), (ii) and (iv).

vii) Rewrite mathematical symbols in a form which can be entered from a normal keyboard (ie. >= for ≥). Note that these symbols should not include any spaces, ie. >= *not* > =.

viii) Rewrite the comments which have been included in the algorithm to aid its understan ding by preceding them with two hyphens `--` rather than enclosing them in braces `{....}`.

ix) Add various output commands to improve the appearance of the output and, when th e program used interactively, to provide instructions to the user of the program.

Some of these are very simple, but others are more involved and will be considered in greater detail in the following sections.

3.4 Using library packages for input and output

The I/O procedures used in the program `Remainder` above (ie. `Put`, `Get` and `New_Line`) are contained in two separate packages whose names are `Ada.Text_IO` and `CS_Int_IO`. The **context clause**:

```
WITH Ada.Text_IO;
```

informs the Ada compiler (and any human reader) that the program will be using procedures from the package `Ada.Text_IO` to output textual values to the VDU screen. Similarly the context clause:

```
WITH CS_Int_IO;
```

enables the program `Remainder` to use procedures from the package `CS_Int_IO` which perform input/output of integer values. Thus whenever the contents of a package are to be used, the package must be made available by including a clause:

```
WITH Package_Name;
```

As we have seen in [3.2] there are different versions of the I/O procedures each belonging to its own library package for use with different data types. In order to be able to refer to the procedures in the package `Ada.Text_IO` simply by using their name rather than the fully-qualified form `Ada.Text_IO.`*`Name`* it is necessary to insert another form of context clause:

```
USE Ada.Text_IO;
```

after the corresponding `WITH` clause and similarly for `CS_Int_IO`. This is convenient as it cuts down on typing and also allows the Ada compiler to automatically use the correct I/O procedure for each type of data value. In one of the recommended text books, Feldman and Koffman have preferred to omit `USE` *`Package_Name`* clauses from their programs and instead to use the fully-qualified names of I/O procedures, while in the other recommended text book, Rudd includes the `USE` *`Package_Name`* clauses. Both methods have advantages and disadvantages: Feldman and Koffman's approach makes it clear exactly which I/O procedures are being used at every step but programmers must be careful always to specify the appropriate I/O procedure for each type of data value whereas the approach adopted in Rudd's text book and these units has the advantages of brevity and simplicity but at the cost of some slight loss of clarity.

Observe that the above gives some indication of the composition of large-scale Ada software, which is not usually composed of a single large program, but rather from a number of smaller units whose components are made available to each other (by suitable `WITH` and `USE` clauses). Thus in the above we would refer to `Ada.Text_IO` and `CS_Int_IO` as being packages, and `Remainder` as being a (main) program. We also see the importance of a sensible and systematic approach to the naming of programs and packages.

3.5 Comments

In Ada as in all programming languages it is good practice to include **comments**, that is explanatory text which is not part of the program proper but which is included to improve program clarity. In Ada comments begin with two hyphens `--` and continue until the end of the line is reached. The comment text is ignored by the Ada compiler and is there solely for the benefit of (human) readers. The example program in [3.3] contains four comment lines: two to describe briefly the purpose of the program and two to explain a particular point of the algorithm. Comments may also be used (for example) to give instructions on how the program is to be compiled and run, on the form of the input data required, to give information about the origin of the program: its author, date, etc, and to indicate when and what part of a program was modified at a later date.

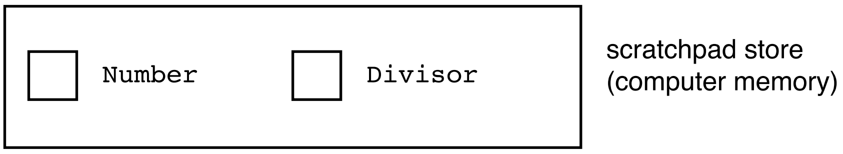
3.6 The variable declaration section

When an algorithm is obeyed, particular values have to be assigned to the variables it uses. If the algorithm is realised as a program executed by a computer, then the computer must physically place these values somewhere. A computer uses locations ("pigeon holes") in its "scratch-pad" store (or memory) for this purpose. So that the computer system can appropriately associate the available locations with the variables to be used, a program must explicitly state (or **declare**) its variable requirements. This is done in a **variable declaration section** which lies between the keyword `IS` in the program heading and the keyword `BEGIN` which marks the start of the executable steps of the program. In this section we must name each variable used by the program and also state the type of value each variable is to store. This is necessary since different types of values, eg. integers and reals, are stored differently inside computers. This also allows the Ada system to do some checking of our program and hence prevent us from making certain types of error.

The full form (or syntax) of variable declarations is a little complicated, so let us begin by looking at the declaration used in the example above [3.3]:

```
Number  : Integer;
Divisor : Integer;
```

Its effect is to reserve two locations suitable for storing integers and to associate the specified names with those locations.



As we have seen before, `Number` and `Divisor` are then said to be variables capable of storing integer values, or more simply said to be **integer variables**. As an alternative we could have achieved the same effect using the slightly more concise declaration:

```
Number, Divisor  : Integer;
```

The general form of the declaration used here is:

`List_of_variables_separated_by_commas` `:` *`Type_for_those_variables`*`;`

There can be any number of these parts; for example suppose that we have a modified version of the algorithm in [2.2] which reads in a number of real floating point data values until a terminating zero value is encountered. It is required to count the number of data values read and also to compute their total. Two variables of type `Float` will be needed `Number` (to hold the current data value) and `RunningTotal` as well as a variable `Count` (say) of type `Integer`, thus we might see:

```
Number, RunningTotal : Float;
Count : Integer;
```

These parts can be in any order, and must be terminated by semi-colons.

3.7 Restrictions on the use of variables

When any variable is declared, a type is associated with it. *Only values of the specified type can then be stored in (ie. assigned to or read into) that particular variable*. For example, assuming the declarations:

```
a : Integer;
x : Float;
```

then consider :

```
a := x;     --  ?! illegal in Ada
```

and:

```
x := 3;     --  ?! illegal in Ada
```

Both are invalid (attempting to store a real floating point value in a integer variable and vice-versa). Similarly, if `Get(Item => a)` were obeyed and the next data available was the text value `THREE`, a run-time error would occur. Since the variable `a` is of type `Integer`, the `Get` procedure expects to only encounter a data value of type `Integer`. Such a data value can only be expressed using digits and thus `THREE` is a text value, not an integer value. A similar run-time error would occur if the data value `3.0` were encountered as this is a real value.

Note To avoid confusion, the symbol `?!` will be used to point out example Ada constructions which are in any way incorrect.

3.8 Identifiers

Variable names and the program name are examples of **identifiers**. There are a number of simple rules governing the form and use of identifiers:

a) An identifier consists of a sequence of letters, digits and the underscore character beginning with a letter. No other characters are permitted. In particular, spaces are not allowed in identifiers. Thus we cannot use `Running Total` as an identifier but must instead use `RunningTotal` (or possibly `Running_Total`).

b) Keywords (such as `BEGIN`, `END`, `WITH` etc.) cannot be used as identifiers.

c) The case of letters is not significant in Ada (except within strings). Thus `TOTAL`, `total` and `Total` all denote the same identifier and will be regarded as identical by the Ada compiler. Although case is not significant a sensible use of capital letters can greatly improve the readability of programs. In these units the conventions adopted are that keywords are written completely in capitals and when an identifier consists of two or more English words the first letter of each component word is capitalised. The identifier `TotalSoFar` is somewhat easier to read than `totalsofar`.

d) For clarity, and to avoid confusion, *meaningful* identifiers should always be chosen. Thus identifiers such as `TimeOfDay` and `RunningTotal` are preferable to identifiers such as `T` or `TOD` and `R` or `RT`.

e) Generally, identifiers may be of any length, and all characters are significant, just as in English words. Thus `AnExtremelyLongIdentifier` is a valid identifier in Ada. However one should avoid using identifiers which are overlong as these make programs difficult to read (and type!). A maximum of 16 characters is a useful rule of thumb.

f) All of the identifiers used in the variable declaration section of a program must be different. These identifiers should also be different from the names of any input/output procedures which are imported into a program.

3.9 Adding explanatory output and improving output layout

When the example program in [3.3] is executed not only does it send output to the user's VDU screen but it also 'expects' input from the keyboard. Thus the program interacts directly with the user and is said to be an **interactive** program. When such a program needs input (ie. when a `Get` step is obeyed) the program will wait until the user types the required data value (and presses the return key). In order that a user will know what is required (and not sit there indefinitely waiting for the program to do something!) the program should output (using `Put` steps) a suitable message prompting the user to type in the required data before each input step.

Similarly as we have seen in Unit 2 a program should output (using `Put` steps) suitable explanatory text to make clear the meaning of the numerical results produced. Often it will also improve the appearance of program output if blank lines are used in order (for example) to clearly separate input and output on the VDU screen. This is achieved by including calls to the output procedure `New_Line` which causes the next output step to begin its output at the start of a new line. The procedure `New_Line` is part of the library package `Ada.Text_IO`. Liberal use of `New_Line`, for example to produce double-line-spacing, can often vastly improve the look of program output. These points are illustrated in the example program in [3.3] which will produce the following output on the VDU (assuming the user types in the values `19` and `4`):

```
Type 2 positive whole numbers: 19 4

Remainder is 3
```

Note that as the user types in data at the keyboard it is 'echoed' on the VDU screen. For clarity user input is indicated by underlining in the diagram above (of course this underlining does not appear in the actual VDU output). In the above diagram it has been assumed that the user pressed the return key after entering the number `4`. Alternatively the user could press the return key twice, once after typing in the value `19` (instead of typing a blank) and again after entering the value `4`. In this case the following VDU output would be produced:

```
Type 2 positive whole numbers: 19
4

Remainder is 3
```

3.10 Use of semi-colons

Program steps (including specification steps such as variable declarations) must be terminated by semi-colons, but it is only *complete* steps that must be terminated, not parts of steps. A complete `IF...` or `WHILE...` construction constitutes a program step, but some of their individual components do not. Thus a semi-colon is required on the last line of the example below to terminate the whole (5 line) `IF...` step and also on lines (2) and (4) to terminate the steps making up the `THEN` and `ELSE` parts. However no semi-colons should be used on lines (1) and (3) as the `IF...` step is not yet complete.

```
1)   IF x > y THEN
2)        Max := x;
3)   ELSE
4)        Max := y;
5)   END IF;
```

Similarly a semi-colon is required on line (4) of the example below to terminate the `WHILE...` step and also on lines (2) and (3) to terminate each step making up the body of the loop, but not on line (1).

```
1)   WHILE Number >= Divisor LOOP
2)        Quotient := Quotient + 1;
3)        Number := Number - Divisor;
4)   END LOOP;
```

Note that comments do not form programs steps (they are ignored by the computer - being there only for the benefit of the human reader of a program) and so need not be terminated by semi-colons.

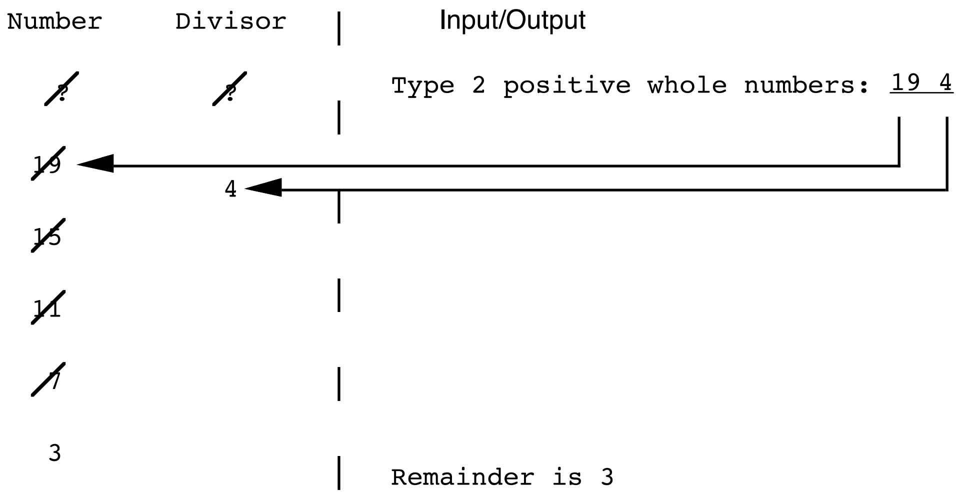
3.11 Layout of programs

Spaces and new lines are used in Ada to separate individual program items such as identifiers, keywords, arithmetic symbols and numbers. However only the first space or new line is significant and the rest are ignored by the Ada compiler. Thus programs may be laid out exactly as desired, and it is important to take advantage of this freedom to introduce blank spaces and blank lines wherever this helps to illuminate the structure of a program. Blank lines should be used to split programs into 'paragraphs' of logically related steps. Spaces should be used to indent programs to emphasize their logical structure. Thus the keywords `IF`, `THEN`, `ELSE` and `END IF` in selections and `WHILE` and `END LOOP` in repetitions should be indented by the same amount and the bodies of the selections and repetitions should be indented by a further fixed amount (say three or four spaces) relative to these keywords. Similarly, the variable declaration section and the steps in the program body should be indented relative to the keywords `PROCEDURE`, `BEGIN` and `END` *`ProgramName`* in the main program.

In Ada it is possible to type two or more program steps on the same line provided each is terminated by a semi-colon. Normally each line of the program should contain only one program step, otherwise the program may appear cramped and so be difficult for humans to read. However occasionally it is clearer to place two or more program steps on a single line when the steps are short *and* are closely related logically, as, for example, is the case with the context clauses in the example program in [3.3].

3.12 Tracing the execution of the example program

Suppose that the example program [3.3] is obeyed, and that data values of `19` and `4` are supplied as input. To help us understand the execution of the program we can **trace** the contents of the variables `Number` and `Divisor` as the execution proceeds. It is also necessary to keep track of the data input (ie. the effect of using `Get`) and of output produced (by using `Put`).



The columns of figures under the variable names show the sequence of values successively stored in those variables as the program is executed. As each value is replaced by a new value, the previous value is (of course) over-written, and this is shown by lightly crossing-out the previous value. Notice that immediately after the variables have been declared (so that locations "exist") but before the `Get` has been obeyed, the variables contain **undefined** or 'junk' values (shown as `?`). The output produced on the VDU screen is also shown. Also, as each data value is read in, it is irrevocably 'consumed'; this too is shown by appropriate crossing-out.

Hand tracing of a program (or algorithm) can often be very helpful in practice as it aids understanding of the program (or algorithm). It is also particularly useful in checking programs (or algorithms) and in finding errors. Hence you should certainly practice and master this technique. For example, it is a good exercise to trace the example program above for various input values including cases where the user types in unexpected values, eg. negative or zero values.

Later in the course and in the laboratory sessions you will learn about programming tools such as **run-time debuggers** which help in tracing and debugging programs. With the use of a debugger, a program can be executed one step at a time and the values of program variables can thus be inspected as the program runs. This, of course, provides much the same information as hand tracing. Nevertheless hand tracing is still important since it can also be used on an algorithm before it is converted into an Ada program. Such hand tracing can be used to detect logical errors in the algorithm at an early stage in the design process and these errors may therefore be corrected before program coding in Ada commences.