Introduction to Systematic Programming
Unit 8 - The Type `Boolean`. Initialised Variables and Constants

8.1 The type `Boolean`

So far, we have used integer and real values and have declared variables (of types `Integer` and `Float`) in which to store such values. We have also met the idea of truth values, and in just the same way as before we may declare variables which can store truth values. The name used for the 'truth value' type in Ada is `Boolean`. Hence we can declare variables able to store truth values by writing (in the declaration section of the main program or of a procedure), for example:

```
            OK, MoreDataLeft : Boolean;
```

Such variables are known as **boolean variables** (after the 19th century mathematician and logician George Boole).

There are only two different boolean values, namely `True` and `False` that can be stored in such variables. These values can be assigned directly to boolean variables, for example:

```
            OK := True;
            MoreDataLeft := False;
```

However, this is not the only way to give a value to a boolean variable. Conditions yield boolean values, and hence the value given by a condition can be assigned to a boolean variable; eg. assuming the declarations:

```
            TransactionOK : Boolean;
            Debit, Limit  : Integer;
```

we could write:

```
            TransactionOK := Debit <= Limit;
```

When this step is executed, the current value of the variable `Debit` is tested to see whether it is less than or equal to the current value of `Limit`; if this is so, then the value `True` is stored in `TransactionOK`, otherwise `False` is stored there. Just as with any other construction which uses values of variables, the execution of this step is sensible only if previously executed steps (input or assignment steps) have caused values to be stored in `Debit` and `Limit`.

We have seen that an `IF` step uses the value `True` or `False`, ie. a boolean value, to select between alternatives. One way to get a boolean value is the result of a condition; another way is the value of a boolean variable. Hence as well as (say):

```
            IF Debit <= Limit THEN ... ELSE ... END IF
```

we can also write:

```
            IF TransactionOK THEN ... ELSE ... END IF
```

When this step is executed, the `THEN`-part or the `ELSE`-part is obeyed according to whether the value stored in `TransactionOK` is `True` or `False`, respectively.

Boolean variables are useful in a number of circumstances, which include:

a) where several separate `IF`'s depend on one condition which may be worked out in advance and assigned to a `Boolean` variable (thus it is not necessary to re-write and hence re-evaluate the condition for each `IF`);

b) where it is convenient to work out a condition at a different place in a program from the point where the corresponding `True` or `False` value is needed;

c) where the conditions are complicated.

You will meet examples of all of these cases later on.

8.2 Compound conditions
Using AND and OR

A decision often depends on a combination of a number of conditions, rather than on one single condition. The logical operators AND and OR allow such cases to be expressed in Ada. For example:

a)
```
IF WaterTemp>0 AND WaterTemp<100 THEN[1]
     Put(Item => "Liquid");
END IF;
```

b)
```
IF WaterTemp<=0 OR WaterTemp>=100 THEN
     Put(Item => "Not Liquid");
END IF;
```

Let C_1 and C_2 stand for any two conditions (of arbitrary complexity) then:

C_1 AND C_2	yields True if C_1 *and* C_2 *both* yield True, otherwise it yields False
C_1 OR C_2	yields True if *either* C_1 *or* C_2 (or both) yield True, otherwise it yields False

More precisely the values yielded are given by the following **truth tables**:

C_1	C_2	C_1 AND C_2	C_1 OR C_2
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

More than two conditions may be combined by using further AND and OR operators. For example:

```
X=Z AND X<Y AND X>0
```

would be True when the three conditions: X = Z, 0<X and X<Y were all True.

Notice that the AND (and similarly the OR) operator has a lower priority of evaluation than the relational operators, =, >, <, etc, so that it is not necessary to bracket relational expressions (X=Z, X<Y and X>0 in the above). However, you should note that it is an error to have conditions involving both AND and OR, unless brackets are included to make clear the order of evaluation. For example:

```
X=Z OR X<Y AND X>0        -- !? illegal in Ada
```

In such cases brackets must be used to indicate the order in which the conditions are to be combined together with the operators AND and OR. For example:

```
X=Z OR (X<Y AND X>0)
```

would be True when X and Z were equal or when both X>0 and X<Y, whereas:

```
(X=Z OR X<Y) AND X>0
```

would be True when X>0 and when either X was equal to Z or when X<Y.

[1] Note that in Ada it is illegal to use abbreviated mathematical notation such as 0<WaterTemp<100:

```
IF 0<WaterTemp<100 THEN ...     -- !? illegal in Ada
```

instead one must write:

```
IF 0<WaterTemp AND WaterTemp<100 THEN ...
```

The conditions $\mathtt{C}_1$ and $\mathtt{C}_2$ above do not have to involve relational operators; anything producing a boolean value will do. One common example is the use of a boolean variable, for example:

```
IF TransactionOK AND Debit<100 THEN
    Accept it
ELSE
    Request identification document, etc, ...
END IF;
```

The operator `NOT`

The operator `NOT` can be used to **negate** (or 'reverse') a boolean value: applied to `True` it yields `False`, and applied to `False` it yields `True`. Thus for example:

```
    IF NOT TransactionOK THEN Put on reject list END IF;
```

will result in execution of the steps in the `THEN`-part only when `TransactionOK` is `False`, and not when `TransactionOK` is `True`. Like `AND` and `OR`, `NOT` is an Ada keyword. The operator `NOT` has the highest possible priority. Thus if it is to be applied to any expression involving other operators, then that expression must be bracketed, for example to negate the truth value of a logical expression such as `TransactionOK AND Debit<100` write `NOT(TransactionOK AND Debit<100);` without the brackets the meaning would be `(NOT TransactionOK) AND Debit<100.`

Note Inexperienced programmers often misuse boolean variables in selections as follows:

```
      IF BoolVar = True THEN ...     -- ?? poor style in Ada
```
or
```
      IF BoolVar = False THEN ...    -- ?? poor style in Ada
```

where (of course) `BoolVar` is assumed to be a variable of type `Boolean`. Although these conditions are syntactically correct they show very poor programming style (due, perhaps, to a lack of understanding of boolean values); it is far neater to express these conditions as follows:

```
      IF BoolVar THEN ...
```
and
```
      IF NOT BoolVar THEN ...
```

respectively.

8.3 Multiple selections

Using the selection:

```
    IF Condition THEN Steps_1 ELSE Steps_2 END IF;
```

we have seen how to choose one of two alternative courses of action depending on whether a condition yields `True` or `False`. However in many problems we may need to select between three, or more, courses of action. We can always achieve this by using nested selection statements, for example:

```
IF WaterTemp<0 THEN
    Put(Item => "Ice");
ELSE
    IF WaterTemp<100 THEN
        Put(Item => "Liquid");
    ELSE
        Put(Item => "Steam");
    END IF;
END IF;
```

However, this construction is rather clumsy, particularly if there are more than three alternatives, since one `END IF` is needed for *each* `IF` and the amount of indentation required can often become excessive. Secondly, in the above example, the three alternatives are really at the same 'logical level' and yet in the second selection appears at a lower logical level (ie. as a nested `IF`). To overcome both of these problems, Ada provides a more general form of the `IF` statement which allows multiple selections to be combined into a single statement. The keywords `ELSE` and `IF` in the nested selection(s) are combined to form the single keyword `ELSIF`. The end of this generalised `IF` statement is signalled by a single `END IF`.

Using this construction we could rewrite the program fragment above more compactly and symmetrically as follows:

```
IF WaterTemp<0 THEN
   Put(Item => "Ice");
ELSIF WaterTemp<100 THEN
   Put(Item => "Liquid");
ELSE
   Put(Item => "Steam");
END IF;
```

Note that it is customary to use the same amount of indentation for the keywords `IF`, `ELSIF`, `ELSE` and `END IF` in this construction and (of course) to indent the steps governed by the selection by a further three or four spaces relative to these keywords.

The most general form of an `IF` statement is as follows:

```
IF  Condition_1 THEN
      Program_steps_1
ELSIF Condition_2 THEN
      Program_steps_2
ELSIF Condition_3 THEN
      Program_steps_3
........
ELSIF Condition_N THEN
      Program_steps_N
ELSE
      Program_steps_N+1
END IF;
```

where the `ELSIF`-parts and the `ELSE`-part are all optional.

8.4 Declaring initialised variables

As we saw in Unit 3 it is necessary to instruct the Ada compiler to allocate suitable storage for program variables by means of variable declarations of the form:

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

A variable declaration of this form causes the compiler to allocate the appropriate amount of storage for each variable. However, it does not store any values whatsoever in these storage locations and thus they will contain random values ('junk' values perhaps left over from a previous program). Very often in programming it is necessary to initialise such variables (ie. give them suitable starting values) before proceeding with the main part of an algorithm.

For example, suppose it is required to calculate the average of a number of non-negative real data values (terminated by a negative value); then, assuming the above declarations, it is necessary to initialise `Count` and `RunningTotal` to zero at the start of the algorithm. Thus we might write the following Ada fragment:

```
Count := 0;
RunningTotal := 0.0;
Get(Item => Number);
WHILE Number >= 0.0 LOOP
     Count := Count + 1;
     RunningTotal := RunningTotal + Number;
     Get(Item => Number);
END LOOP;
Average := RunningTotal/Float(Count);
```

Unfortunately it is all too easy to forget to initialise all the necessary variables (particularly in a large program with many variables) and so end up with a program which produces erroneous results. To help overcome this problem Ada allows an initial value of a variable to be specified as part of the variable declaration. A suitable initialised variable declaration for the above problem would have the form:

```
Count : Integer := 0;
RunningTotal : Float := 0.0;
Number, Average : Float;
```

which allocates appropriate storage for the variables in the normal manner and then initialises `Count` to `0` and `RunningTotal` to `0.0`. We may then omit the first two steps of the Ada program fragment above to produce the following:

```
Get(Item => Number);
WHILE Number >= 0.0 LOOP
    Count := Count + 1;
    RunningTotal := RunningTotal + Number;
    Get(Item => Number);
END LOOP;
Average := RunningTotal/Float(Count);
```

Note that it is *not* necessary to initialise the variables `Number` and `Average` as any 'junk' values that they contain are overwritten by the `Get` step in line 1 and the assignment step in line 7 of the above fragment respectively. It is considered poor programming style to initialise variables (such as `Number` and `Average`) when it is not necessary. Such initialisation steps are totally redundant; they simply increase program size and slow down program execution. More importantly they are misleading and so reduce the clarity of a program; if a variable is initialised with a certain value, a reader might reasonably expect that the value would be used by the program in some way rather than simply being overwritten.

The general form of a variable declaration consists of one or more lines of the form:

```
List_of_variables_separated_by_commas : Type_name := Initial_value;
```

This causes appropriate storage to be allocated for each of the variables and then each of the variables is assigned the initial value specified. The '`:= Initial_value`' part is optional and, if it is omitted, the variables are established with 'junk' values.

Whenever it is consistent with the logic of a problem, variables which will need to be given initial values *should* be initialised as they are declared. Clearly if declaration and initialisation occur together rather than at two (perhaps widely separated) points the risk of error is considerably reduced.

Examples

i) In a banking program we might have a declaration of the form:

```
AccountNew      : Boolean := True;
Overdrawn       : Boolean := False;
Balance         : Float := 10.00;    -- £10 bonus for new a/c
OpeningDeposit  : Float;
NumStandingOrders, NumDirectDebits : Integer := 0;
```

ii) The local variables of procedure `ProduceFactors` at the end of Units 5 and 6 could be declared and assigned initial values as follows:

```
PROCEDURE ProduceFactors (Number : IN Integer) IS
   NumOfDivisors : Integer := 0;
   TrialDivisor  : Integer := 2;
BEGIN
   .......
```

Notes

a) The type of the initial value specified must be the same as the type specified in the variable declaration.

b) It is possible to use the initialised form of declaration for both global and local variables. For global variables the initialisation is performed once and once only - just before the steps of the main program are executed. For local variables the initialisation takes place every time the procedure is called – just before the steps of the procedure body are executed.

c) Usually the initial value specified is a literal constant (of the appropriate type), but in general it can be any expression yielding a value of the appropriate type *provided that* the expression can be fully evaluated at the time the initialisation takes place. In particular, this means that all the objects (variables, formal parameters, etc.) comprising the expression must exist and have values at the time the initialisation occurs. A few examples should make this clear:

i)

```
Radius   : Float := 39.3;
Diameter : Float := 2.0 * Radius;
```

The second line is valid as `Radius` has been declared and initialised previously and so its value may be used when initialising `Diameter`.

ii) Suppose a procedure is defined as follows:

```
PROCEDURE Example (X, Y : IN Integer) IS
    Quotient  : Integer := X/Y;
    Remainder : Integer := X REM Y;
BEGIN
    ..........
```

and, for example, suppose that the procedure is called as follows:

```
Example(X => 45, Y => 12);
```

As the procedure call takes place the values `45` and `12` are copied into the formal parameters `X` and `Y` respectively. Next a local `Integer` variable `Quotient` is created and initialised with the value of `X/Y`, that is to `45/12 = 3` and a local `Integer` variable `Remainder` is created and initialised with the value of `X REM Y`, that is `45 REM 12 = 9`.

8.5 Constant Declarations

Realistic programs often involve mathematical or scientific constants, such as the value of π (≈3.1415926), the acceleration of a falling body due to gravity (≈9.807 ms^{-2}), conversion factors for different units (1 mile = 1.609344 km.), etc. Other programs involve values which remain constant during the execution of a program and are not going to be given as data (eg. the current rate of income tax, the number of lines of output which will fit onto one page, etc.). It is of course possible to use the appropriate literal values directly at the relevant points in a program, but it is much clearer to give each such constant a suitable name at the head of a program and then, within the remainder of the program, use those names to refer to the constants. This is done by including a constant declaration in a program, for example:

```
Pi : CONSTANT Float := 3.1415926;
g  : CONSTANT Float := 9.807;
TaxRate : CONSTANT Float := 0.25;
LinesPerPage : CONSTANT Integer := 60;
```

Thus a constant declaration is similar to an initialised variable declaration except that the keyword `CONSTANT` is inserted into the declaration just after the colon and before the type name.

Once a constant has been defined, then its name can be used instead of the corresponding value at any point in the program, for example:

```
AreaOfCircle := Pi * Radius**2;

Speed := g * Time;   -- Speed of a falling body

Pay := Pay * (1.0 - TaxRate);

IF LineNum = LinesPerPage THEN Start a new page END IF;
```

It is also possible to use expressions on the right-hand side of constant declarations *provided that* all the values are known (or can be calculated) at the time the constant declaration occurs. For example:

```
Pi : CONSTANT Float := 3.1415926;
PiBy2 : CONSTANT Float := Pi / 2.0;
LitresPerGall : CONSTANT Float := 4.546;
GallsPerLitre : CONSTANT Float := 1.0 / LitresPerGall;
```

Note that it is illegal in Ada to attempt to change the value of a constant during execution of a program. For example, given the constant declaration on the previous page, it is illegal to attempt to give a new value to `TaxRate` by an assignment, for example:

```
TaxRate := 0.40;      -- !? illegal in Ada
```

Similarly it is not permitted to use a constant as an `OUT` mode or `IN OUT` mode parameter in a procedure call statement (since if this were allowed the procedure could change the constant's value). Once a constant has been set up by a constant declaration, its value is 'retrieve access only' for the rest of its existence. This is similar to the behaviour of formal `IN` parameters of a procedure. Recall that the value of a formal `IN` parameter may not be changed by steps in the procedure (see Unit 5). This is also consistent with our use in Unit 5 of the term 'local constants' when referring to `IN` parameters.

Some advantages of using constants

a) The use of constant declarations (with meaningful identifiers) makes programs easier to understand. For example it is easier to understand the intention behind a program step such as:

```
Pay := Pay * (1.0 - TaxRate);
```

rather than the more cryptic form:

```
Pay := Pay * 0.75;
```

b) They reduce the risk of typing errors since it is only necessary to type multi-digit numbers (such as 3.1415926 for the value of π) once instead of possibly many times in a program.

c) They make programs easier to modify. We can change a constant throughout a long program simply by editing the constant declaration at the head of the program instead of having to search through the program for all occurrences of that literal constant. For example after Budget Day it might be necessary to edit a pay-roll program to change the declaration of the constant `TaxRate`. Of course it would then be necessary to recompile the modified program.

The position of constant declarations

In Ada 95 variable and constant declarations and procedure definitions may appear in any order in the declaration part of a program. However it is recommended that constant declarations appear before any procedure definitions so that

a) the constant is available for use if necessary in procedures (In Ada the declaration of a constant must precede any use of that constant).

b) it makes for easier modification of the program as discussed in (c) above.

It is also possible to place constant declarations in the declaration section of a procedure definition. This causes the constants to be local to that procedure. It is (of course) sensible always to use meaningful identifiers in constant declarations.

8.6 Example

A simple number guessing program is to be written to do the following:

i) A secret number (a positive integer lying in a the range 0 to 1000) is to be generated using the procedure `GetNextRandom` from the library package `CS_Random`.

ii) The user is then asked to guess this secret number, but is only allowed 15 guesses.

iii) If the user's guess is not correct, the program should output a message indicating whether the guess was too high or too low and then ask the user to try again.

iv) Depending on whether the user guesses the correct number within 15 goes or not, suitable messages giving the number of guesses taken etc. are to be output.

v) The whole number guessing process is then repeated from step (i) with a new secret number. If at any stage the user types a negative number (thus indicating that they wish to 'quit') the program should terminate.

First thoughts

Generate the secret number and ask the user to make a guess at its value. Need to test the guess (an integer value) to see if it is negative and against the secret number (four possible outcomes so will need to use `IF` with `ELSIF`) to output an appropriate message. Unless the guess is correct or all guesses have been used up or the user wishes to quit, we need to read in next guess and repeat. (Repetition terminated by correct guess or by the number of guesses reaching 15 or by the user typing a negative number => compound condition controlling a `WHILE` loop). During this, need to count number of guesses (an `Integer` value). After the loop is finished need to know which conditions caused termination of the loop in order to output appropriate message – thus we should use two boolean variables: one to record whether the user has guessed the number and the other to signal that he/she wants to quit. All the above must be repeated unless the user has typed a negative number to quit -- thus we need to nest all the above in another `WHILE` loop controlled by the second boolean. Constants should be declared to hold the maximum number of guesses allowed (15) and the largest possible value of the secret number (1000).

Is it feasible to guess the secret number in 15 or fewer guesses? Can you think of a suitable strategy for the user?

Full program

This is presented here without explanation of how to get from 'first thoughts' to the program. Nevertheless, you should be able to read through and understand this program.

Note that the variables `NumGuesses` and `Guessed` are only needed within the procedure `TryToGuess`, so they are declared as local variables of the procedure. Also, they need to be 're-initialised' for each new secret number; this will happen since they are initialised each time they are created when the procedure is called. The constant `MaxGuesses` is also only used in this procedure, so it could be declared as a local constant, however, it is easier to find and alter (if necessary) if it is declared globally. The constant `MaxSecret`, however, *must* be declared as a global constant as it is used by the main program and by the procedure `TryToGuess`.

The use of the procedure `GetNextRandom` from the library package `CS_Random` needs a little explanation; this procedure has two formal parameters:

i) an `OUT` mode formal parameter `Number` of type `Integer` which is used to pass the random number generated by the procedure back to the calling program, and

ii) an integer `IN` mode formal parameter `MaxRand`. The random number generated lies between 0 and the value of `MaxRand` (inclusive). For technical reasons which need not concern us here, the value of `MaxRand` can be no larger than `2**24 - 1` (=`16777215`).

In order to make it clear that the procedure `GetNextRandom` is imported from the package `CS_Random` we use the fully qualified name and hence there is no need to place the context clause:

```
USE CS_Random;
```

at the head of the program. To generate a random number `Secret` in the range 0 to 1000, we will use the procedure call step:

```
CS_Random.GetNextRandom(Number => Secret, MaxRand => 1000);
```

```
WITH Ada.Text_IO; USE Ada.Text_IO; -- To import Put for text and New_Line
WITH CS_Int_IO;   USE CS_Int_IO;   -- To import Get and Put for integers
WITH CS_Random;                    -- To import procedure GetNextRandom

PROCEDURE GuessingGame IS
   --  Program for Unit 8 of ISP Ada course.
   --  Illustrates the use of Boolean variables, compound
   --  conditions, multiple selections, initialised variable
   --  and constant declarations.
   --  Written by A. Barnes, September 1993.

   -- Constant declarations
   MaxSecret  : CONSTANT Integer := 1000; -- Largest secret number
   MaxGuesses : CONSTANT Integer := 15;   -- Number of guesses allowed

   PROCEDURE TryToGuess(Number : IN Integer; Continuing : OUT Boolean) IS

      NumGuesses : Integer := 0;         -- Number of guesses made
      Guess      : Integer;              -- Current guess
      Guessed    : Boolean := False;     -- True if user guesses number

   BEGIN
      Continuing := True;      -- Initialise OUT parameter
      Put(Item => "You have "); Put(Item => MaxGuesses);
      Put(Item => " goes to guess my secret number."); New_Line;
      Put(Item => "It lies between 0 and ");
      Put(Item => MaxSecret); New_Line;
      Put(Item => "At any time type a negative integer to quit.");
      New_Line;

      WHILE Continuing AND NOT Guessed AND NumGuesses<MaxGuesses LOOP
         Put(Item => "Type in your guess please> ");
         Get(Item => Guess);
         NumGuesses := NumGuesses + 1;

         IF Guess < 0 THEN
            Continuing := False;
         ELSIF Number > Guess  THEN
            Put(Item => "Your guess is too low. Try again!");
         ELSIF Number < Guess THEN
            Put(Item => "Your guess is too high. Try again!");
         ELSE
            Put(Item => "Correct! You took "); Put(Item => NumGuesses);
            Put(Item => " goes to guess my number."); New_Line;
            Guessed := True;
         END IF;
         New_Line;
      END LOOP;

      IF Continuing AND NOT Guessed THEN
         Put(Item => "Hard luck!  All your goes are up."); New_Line;
         Put(Item => "The secret number was ");
         Put(Item => Number); New_Line; New_Line;
      END IF;

   END TryToGuess;

   -- Global variables
   Secret      : Integer;        -- The pseudo-random number to be guessed
   StillPlaying : Boolean := True;  -- False if user wants to quit

BEGIN   -- Main Program
   WHILE StillPlaying LOOP
      CS_Random.GetNextRandom(Number => Secret, MaxRand => MaxSecret);
      TryToGuess(Number => Secret, Continuing => StillPlaying);
   END LOOP;
   Put(Item => "Quitting ..."); New_Line;
END GuessingGame;
```