Introduction to Systematic Programming
Unit 9 - More on Procedures.   Functions

9.1 Default parameters

It sometimes happens that one particular actual parameter value is used far more frequently than any other value when calls are made to a procedure. In such cases it is possible in Ada to give a **default value** to the corresponding formal parameter in the parameter specification part of the procedure heading. In Ada it is only possible to define such default values for parameters of mode `IN`. Defining a default value for a formal parameter has the following effect: if an AP is supplied in a procedure call, then (as usual) this value is copied to the FP; however if the AP is omitted from the procedure call step then the default value is used instead. Therefore, in these circumstances it is not necessary to supply all the parameters in a procedure call step; parameters with default values *may* be omitted. In order to define a default value a slightly modified parameter specification of the form:

```
Formal_parameter : IN Type_name := Default_value
```

is used. Note that this form of parameter specification is similar in many respects to an initialised variable declaration. The notation used is reasonable since we may think of a default value as initialising the FP when no AP is supplied in the procedure call step.

Example

In [5.7] a procedure `DrawLine` with two parameters `Margin` and `Length` was defined. The most frequently used value of the parameter `Margin` is likely to be zero (so that the line of stars is not preceded by any blanks). Therefore it is sensible to specify a default value of zero for this parameter:

```
PROCEDURE DrawLine (Margin : IN Integer := 0;
                    Length : IN Integer)   IS
   -- To output Margin blanks, followed by a line of Length stars
   Count : Integer;
BEGIN
   New_Line;
   -- First output Margin blanks
   Count := 1;
   WHILE Count <= Margin LOOP
      Put(Item => " ");
      Count := Count + 1;
   END LOOP;
   -- Next output Length stars
   Count := 1;
   WHILE Count <= Length LOOP
      Put(Item => "*");
      Count := Count + 1;
   END LOOP;
   New_Line;
END DrawLine;
```

Now if an AP value corresponding to `Margin` is supplied in a procedure call that value is used and the default value is ignored. Thus to produce a 10 character margin followed by 40 stars we would use the call step:

```
DrawLine(Margin => 10, Length => 40);
```

However, if an AP corresponding to the FP `Margin` is not supplied, for example in the call:

```
DrawLine(Length => 40);
```

then `Margin` is set to its default value: namely zero. Thus this procedure call is equivalent to:

```
DrawLine(Margin => 0, Length => 40);
```

Note If a FP has no default value specified for it, then an AP *must* be supplied in any procedure call step, otherwise a compilation error will occur.

9.2 Default parameters in I/O procedures

A number of the procedures in the standard I/O libraries have default values defined for some of their parameters. For example, the FP `Width` of the procedure `Put` from `CS_Int_IO` is given a default value of 1 which means, in effect, 'output the integer in a field of minimum width' (as was described in [4.5.3]). In fact the procedure `Put` has the heading:

```
PROCEDURE Put (Item  : IN Integer;
               Width : IN Integer := 1;
               Base  : IN Integer := 10);
```

and so takes a third parameter `Base`. So far this extra parameter has not been discussed; it specifies the number base to be used when outputting the number and has the default value 10. Thus to output a number in standard base 10 notation (ie. standard decimal form) in a field of width 6 (say) the call step:

```
Put(Item => IntVar, Width => 6);
```

(where `IntVar` is an integer variable) could be used. However, if output in base 2 (ie. binary) notation is required the call step:

```
Put(Item => IntVar, Width => 6, Base => 2);
```

would be used.

The procedure `Put` from `CS_Flt_IO` has four FP's in total, three of which have default values. In fact this procedure has the heading:

```
PROCEDURE Put (Item : IN Float;
               Fore : IN Integer := 1;
               Aft  : IN Integer := 2;
               Exp  : IN Integer := 0);
```

The parameters `Fore` and `Aft` are given default values of `1` and `2` which means, in effect, 'output the real number to `2` decimal places of accuracy in a field of minimum width' (see [4.5.3] for more details of this). So far the fourth parameter `Exp` has not been discussed; if a non-zero value is specified for `Exp` the real value is printed in **scientific notation** whereas if `Exp` has its default value of zero, the number is output in the usual floating point format. Here are a few examples of scientific notation:

Floating Point Form	Normal Scientific Form	Computer Scientific Form
`256.1`	2.561×10^{2}	`2.561E2`
`-1.98`	-1.98×10^{0}	`-1.98E0`
`0.0023`	2.3×10^{-3}	`2.3E-3`

The values of `Fore` and `Aft` control the number of print positions used before and after the decimal point and the value of `Exp` (if greater than zero) specifies the number of print positions used to output the **exponent** (ie. the part after the `E`). If the value specified for `Exp` is positive but smaller than the number of print positions required to output the exponent, the exponent is output in full using the minimum number of print positions (ie. the exponent is never truncated in any way).

The procedure `New_Line` from `Ada.Text_IO` also has a formal parameter `Spacing` with a default value of 1. The parameter `Spacing` specifies the number of new lines which are to be output. So far we have only used `New_Line` as if it were a procedure with no parameters, but now we can see that the procedure call:

`New_Line;`   is equivalent to the call: `New_Line(Spacing => 1);`

Furthermore a procedure call step such as:

```
New_Line(Spacing => 3);
```

is equivalent in its effect to the three separate calls:

```
New_Line;
New_Line;
New_Line;
```

9.3 Parameter association by position

So far in procedure call steps we have associated formal and actual parameters in procedure calls using expressions of the form:

`Formal_parameter => Actual_parameter` or *`FP => AP`*

This method is known as **parameter association by name**. In Ada a second method is also allowed: namely **parameter association by position**[1]. Using this method only the actual parameters appear in the procedure call step and the association with formal parameters is by position: the first AP is associated with the first FP in the parameter specification in the procedure heading, the second AP with the second FP and so on. A few examples using the procedure `Put` from `CS_Int_IO` should clarify this:

a)
```
Put(Number, 8, 10);
```
is equivalent to:
```
Put(Item => Number, Width => 8, Base => 10);
```

b)
```
Put(Number, 7);
```
is equivalent to:
```
Put(Item => Number, Width => 7);
```
Here `Number` is associated with the first FP (namely `Item`) and `7` with the second FP (namely `Width`) and, since the third AP is missing, the default value `10` of the third FP `Base` is used.

c)
```
Put(Number);
```
is equivalent to:
```
Put(Item => Number);
```
Here `Number` is associated with the first FP (namely `Item`) and, since the second and third AP's are missing, their default values (namely `1` and `10`) are used.

d) Note that to express the equivalent of the procedure call:
```
Put(Item => Number, Base => 2);
```
when using parameter association by position, requires that the default value 1 for the second FP `Width` be supplied explicitly:
```
Put(Number, 1, 2);
```
If we were to write:
```
Put(Number, 2);
```
the AP value `2` would be associated with `Width` and the default value `10` of `Base` would be used.

The major advantage of using parameter association by position is obviously brevity. Secondly one need not know the precise FP names to use association by position. However there are some serious disadvantages:

i) Particularly when there are three or more parameters there is a risk of putting the AP's in the wrong order and so producing an erroneous program. The chances of making a mistake when using parameter association by name is much reduced as the order is not critical.

ii) Reduced program clarity with positional parameters as the association of AP's and FP's is not explicitly stated in the procedure call.

iii) Parameter association by position when some parameters are omitted is less flexible than named association and is more error prone (see example (d) above).

[1] In fact, many programming languages (eg. C, Basic, Pascal, Lisp) only allow parameter association by position.

As a general rule of thumb parameter association by name is the preferred method particularly if a procedure has several parameters. However association by position is acceptable in cases where a procedure has only one parameter (where there is no problem with order) or if a procedure has two (or more) parameters whose order is unimportant (as, for example, with the procedure `PrintMax` considered in [6.1]), and we will adopt this practice in the course units.

9.4 Functions

In Ada **functions** are similar to procedures. A procedure is designed to perform some particular computation, and (by means of a procedure declaration) is given a name. Then, by quoting the name of the procedure, and supplying whatever parameter information is expected, the defined computation can be executed whenever required. A **function** is also called by quoting its name and specifying appropriate parameters, but in addition to performing some defined computation, a function delivers (or **returns**) a value which is *immediately available for use*, eg. as an operand in an arithmetic expression. In fact a function may *only* be called in a place in a program where its returned value can used. In Ada functions and procedures together are often referred to as **subprograms**.

9.4.1 Defining a function

Functions may have most of the features of procedures, such as a heading, parameters, local variables and a body, but in addition the type of the value returned must also be specified in the heading. In fact the form for function definition is:

```
FUNCTION Identifier(Parameter_specification) RETURN Result_type IS
   Local_variable_declarations
BEGIN
   Step(s)
END Identifier;
```

The differences from a procedure definition are:

a) The keyword `FUNCTION` is used instead of `PROCEDURE`.

b) The result type must be specified after the parameter specification part by writing the keyword `RETURN` followed by the type of the value to be delivered.

c) The parameter specification is more restricted than for procedures: all the parameters of a function must be of mode `IN` (`OUT` or `IN OUT` modes are not permitted and any attempt to use them will result in a compilation error). We see therefore that a function can only deliver a single result for use in the calling program. As all the parameters must be of mode `IN`, this keyword may be (and usually is) omitted from the parameter specification which thus consists of one or more parts of the form:

   ```
   List_of_parameters_separated_by_commas : Type_name := Default_value
   ```

   These parts are separated by semi-colons (just as in procedure parameter specifications). The '*`:= Default-value`* ' part is optional.

d) The function body *must* include one or more steps which specify the value to be returned. This is done by writing the keyword `RETURN` followed an expression which, when evaluated, causes the function to complete its execution and produces the required value to be returned, ie:

   ```
   RETURN Expression;
   ```

   The type of the value returned should be the same as the type specified in the function heading.

Notes i) Although there is no theoretical limit on the number of `RETURN` steps that can appear in a function body, it is generally thought to be bad programming style to write functions containing many `RETURN` steps as this makes the program difficult to follow.

ii) It is also possible to use the program step:

   ```
   RETURN;
   ```

   in the steps of a procedure. When this step is executed, the procedure 'returns' immediately to the calling program, but, of course, since a procedure doesn't deliver a value no expression is required after the keyword `RETURN`.

9.4.2 Calling a function

A function call resembles a procedure call in several respects: a function is called by quoting its name and supplying the required parameters in brackets. Each actual parameter is associated with the corresponding formal parameter in the normal manner (ie. using association by name of the form *FP* => *AP* or using association by position). However function and procedure calls differ in one important respect: a procedure call forms a complete program step whereas a function call does not. A function call directly returns a value which the calling program *must* use in some manner. Thus a function call may be used anywhere in an expression where a variable (of the same type) would be valid. The following examples illustrate these points.

9.4.3 Examples

A function to calculate the larger of two real values could be defined as:

```
FUNCTION Max (First, Second : Float) RETURN Float IS
BEGIN
   IF First > Second THEN
      RETURN First;
   ELSE
      RETURN Second;
   END IF;
END Max;
```

Once it has been defined (and assuming the declaration of appropriate `Float` variables) the function `Max` might be called in any of the following ways:

```
Larger := Max(First => A, Second => B);

Put(Max(First => MyBalance, Second => YourBalance));

HalfMax := Max(First => A, Second => B) / 2.0;

IF Max(First => A, Second => B) < 0.0 THEN Do_something END IF;
```

In Unit 6 a procedure `FindMax` was defined which also computed the larger of its two `IN` mode parameters. However `FindMax` passed this value back to the calling program by means of an `OUT` mode parameter. The question naturally arises: which approach is better? To answer this question, compare the use of the procedure `FindMax` and the function `Max`. Firstly suppose we wish to assign the greater of two `Float` values `A` and `B` to the variable `Larger`. Using the procedure `FindMax` we would write:

```
FindMax(First => A, Second => B, Max =>Larger);
```

whereas using the function `Max` we would write:

```
Larger := Max(First => A, Second => B);
```

Here there is not much to choose between the two methods in terms of convenience. However suppose we wished to output a suitable message depending on the value of the larger of the two values `A` and `B`. Using procedure `FindMax` we would write:

```
FindMax(First => A, Second => B, Max => Temp);
IF Temp < Limit THEN
   Put(Item => "Limit not exceeded");
END IF;
```

whereas with function `Max` we have:

```
IF Max(First => A, Second => B) < Limit THEN
   Put(Item => "Limit not exceeded");
END IF;
```

Here the program fragment using the function `Max` is clearly the neater of the two; everything can be coded in a single selection and the extra variable `Temp` is not needed. Furthermore, since the value returned by function `Max` does not depend on whether the AP `A` is associated with the FP `First` and

B with Second or vice-versa, we may use, with equal clarity, parameter association by position to obtain:

```
IF Max(A, B) < Limit THEN
   Put("Limit not exceeded");
END IF;
```

As a general rule if a program fragment is to calculate a single value then it should be written in Ada as a function rather than a procedure with an OUT parameter. As a function's value is immediately available for use in expressions, a program using a function call is usually more compact and clearer than a comparable program using a procedure call. However if two or more values are to be computed and passed back to the calling program, a procedure with the appropriate number of OUT parameters *must* be used since a function can only return a single value.

As a second example consider the factorial function. The factorial of a positive integer N (written N!) is equal to N*(N-1)*(N-2)*...3*2*1, ie. N! is the product of all the positive integers which are less than or equal to N.

```
FUNCTION Factorial (N : Integer) RETURN Integer IS
   ProductSoFar : Integer := 1;
   Count : Integer := 1;
BEGIN
   WHILE Count <= N LOOP
      ProductSoFar := ProductSoFar * Count;
      Count := Count + 1;
   END LOOP;
   RETURN ProductSoFar;
END Factorial;
```

which might be called using:

```
Put("Factorial of 8 is ");
Put(Factorial(8));
```

9.4.4 Predicate functions

Functions returning a boolean result are often useful; they are referred to as **predicates**. For example, suppose we want to be able to check whether a particular year is a leap year or not. We could write:

```
FUNCTION IsLeap(Year : Integer) RETURN Boolean IS
BEGIN
   RETURN (Year REM 4 = 0 AND Year REM 100 /= 0) OR
          Year REM 400 = 0;
END IsLeap;
```

Note that on the Gregorian Calendar (which is in almost universal use today) a year is a leap year if the year number is exactly divisible by 4 but not by 100, or if the year number is exactly divisible by 400.[2] Thus the year 1900 was not a leap year but the year 2000 will be. A typical use of this function might be:

```
IF IsLeap(ThisYear) THEN
   DaysInFeb := 29;
ELSE
   DaysInFeb := 28;
END IF;
```

[2] Historical note: The Gregorian Calendar is named after Pope Gregory. In Britain it replaced the older Julian Calendar (named after Julius Caesar) in the middle of the 18th Century. The Julian Calendar used a simpler leap year algorithm: a year is a leap year if the year number is exactly divisible by 4. The average length of year on the Gregorian Calendar is 365.2425 days which coincides almost exactly with the time it takes Earth to complete one orbit of Sun. By contrast the Julian leap year

9.5 A library package of mathematical functions

Functions are applicable to all types of programming, but are very commonly used in mathematical work. In Ada 95 the standard mathematical functions are defined in a library package called `Ada.Numerics.Elementary_Functions`. If we wish to use these mathematical functions we may import them from this package into our programs in the standard way:

```
WITH Ada.Numerics.Elementary_Functions;
USE Ada.Numerics.Elementary_Functions;
```

Here is a list of some of the most commonly used functions provided in the package `Ada.Numerics.Elementary_Functions`:

`Exp`	exponential function	e^x
`Log`	natural logarithm	$\log_e x$
`Sqrt`	square root function	x
`Sin`	sine function	$\sin x$
`Cos`	cosine function	$\cos x$
`Tan`	tangent function	$\tan x$
`Arcsin`	arc-sine (inverse sine) function	$\sin^{-1} x$
`Arctan`	arc-tangent (inverse tangent) function	$\tan^{-1} x$

All the above functions have a single formal parameter `X` of type `Float` and return a `Float` result. When calling these functions it is customary to use parameter association by position. This usage corresponds more closely with standard mathematical notation.

If logarithms to other bases are required, the base must be supplied as a second parameter of type `Float`. For example (assuming a suitable declaration of `Float` variable `ComLog3`), the program step:

```
ComLog3 := Log(X => 3.0, Base => 10.0);
```

would set `ComLog3` to the common logarithm (base 10) of 3, i.e. $\log_{10} 3$.

The package `Ada.Numerics.Elementary_Functions` also provides another form of the exponentiation operator `**`. This form takes two operands of type `Float` and produces a result of type `Float`. For example the program step:

```
FourthRoot := X ** 0.25;
```

would set `FourthRoot` to the 4th root of the value of variable `X` (assuming suitable declarations of `Float` variables `FourthRoot` and `X`).

9.6 Functions with no parameters

Very occasionally we may wish to define a function with no parameters (or a **parameterless function**). In this case the parameter specification part and the pair of parentheses `()`, are omitted from the function definition. Such parameterless functions will almost always involve an input step unless the function simply returns the same value each time it is called. For example, we could define a function `NextInput` which, like the procedure `Get` in `CS_Int_IO`, 'reads' values from an input device, but which also makes this value immediately available for use in expressions:

```
FUNCTION NextInput RETURN Integer IS
   Number : Integer;
BEGIN
   Get(Number);
   RETURN Number;
END NextInput;
```

This appears to have some advantages: for example to read in a data value and add it to a variable `Total` we could simply write:

```
Total := Total + NextInput;
```

rather than: `Get(Number);`

algorithm gave rise to an average year length of 365.25 days which is too long b y about 11 minutes.

```
              Total := Total + Number;
```

However there are a number of disadvantages:

i) The function call looks like a reference to a variable and so reduces program clarity.

ii) More insidiously, because of the input step involved, undefined program behaviour may occur if two calls are made to `NextInput` in the same expression. For example, suppose the input stream contains two integers `5` and `3` (say) and that the program step:

```
        Result := NextInput * (10 - NextInput);
```

is executed. You might expect that the expression would be evaluated from left to right so that the variable `Result` would be set to 5 × (10 – 3) = 35, but can you be sure? Perhaps the computer evaluates the term in brackets first and so `Result` is set to 3 × (10 – 5) = 15. One computer might evaluate the expression in one way and another in the opposite way. Thus a program could produce different results when run on different machines. In any case the expressions:

```
        NextInput * (10 - NextInput)
```

and:

```
        (10 - NextInput) * NextInput
```

would produce different results. This sort of behaviour is very obviously highly undesirable.

9.7 Functions with side-effects

A procedure or function which modifies the 'environment' of the program by consuming input data or producing output is said to have **side-effects**. A procedure may also affect the 'environment' by altering the values of variables in the calling program since it can pass values back to the calling program via `OUT` or `IN OUT` mode parameters.

With procedures such side-effects are acceptable; how else may a program handle I/O or a procedure pass back computed values to the calling program? However, as we have seen above in [9.6(ii)], functions with side-effects can make a program difficult to understand (and debug) and, in certain circumstances, may lead to undefined program behaviour. Such undefined behaviour is not possible with procedures since a procedure call is a complete program step and so the order in which procedure calls occur is completely determined by program sequencing.

A function is normally called solely for its value and it is generally regarded as poor programming style to define functions which produce side-effects. Instead one should define a procedure and use an `OUT` (or `IN OUT`) parameter to pass back the 'return' value to the calling program. Unfortunately some library packages *do* provide parameterless functions with side-effects and so if one needs to use these packages one should be aware of the pitfalls – *in particular never use such a function more than once in any expression*.

You can now perhaps also appreciate why it is illegal in Ada to define functions with `OUT` or `IN OUT` parameters; such functions could have the side-effect of altering the values of variables in the calling program and this would increase the scope for producing undefined program behaviour.

9.8 Example

An interactive program is to be written to ask the user to input a positive integer `N` (≥`2`). The program is to output all the primes ≤`N`. The primes are to be output 6 to a line. After the table of primes a summary giving the number of primes ≤`N` is to be output followed by the estimated number of such primes according to the Prime Number Theorem: namely `N/log`$_e$`(N)` (rounded to the nearest integer).

First thoughts

We need to test all odd numbers ≤ N for primality (all even numbers are divisible by 2 and so except for 2 itself even numbers cannot be prime) and keep a count (`NumPrimes` say) of the number of primes found. Thus we will need a repetition. In order to test whether an odd number is prime or not, we need to check to see if it is exactly divisible by all odd numbers less than or equal to its square root. Thus we need a repetition which should terminate as soon as an exact divisor is found or when all trial divisors up to and including its square root have been tried.

The library package `Ada.Numerics.Elementary_Functions` will be needed for the square root and logarithm functions. We will need to be careful as these functions expect parameters of type `Float` and so we will need to use the conversion function `Float` to convert whole number values before calling these two functions.

In order to output 6 primes to a line we will need a selection step of the form:

```
    IF NumPrimes REM 6 = 0 THEN New_Line; END IF;
```

but we introduce a named constant `NumberPerLine` (`:= 6`) for clarity.

Final Program

```
WITH Ada.Text_IO;
USE Ada.Text_IO;                 -- Import Put (for strings) and New_Line
WITH CS_Int_IO; USE CS_Int_IO;   -- Import Put and Get for integers
WITH Ada.Numerics.Elementary_Functions;
USE Ada.Numerics.Elementary_Functions;     -- Import Sqrt and Log

PROCEDURE Primes IS
   -- Program for Unit 9 of the ISP Ada course.
   -- Prints all the primes less than or equal to a number N (user input)
   -- Also outputs the number of such primes and the estimate N/log(N)
   -- obtained from the Prime Number Theorem.
   -- Written by Alan Barnes, September 1993.

   NumberPerLine : CONSTANT Integer := 6; -- Number of primes output/line

   FUNCTION ISqrt (Number : Integer) RETURN Integer IS
      -- Returns square root of Number rounded to the nearest integer
   BEGIN
      RETURN Integer(Sqrt(Float(Number)));
   END ISqrt;

   FUNCTION IsPrime (Number : Integer) RETURN Boolean IS
     -- Returns True if number is prime and False if it is composite
     TrialDivisor : Integer := 3;
     MaxTrial   : Integer;
     Composite : Boolean := (Number REM 2 = 0); -- True if 2 is a divisor
   BEGIN
      MaxTrial := ISqrt(Number);
      WHILE NOT Composite AND TrialDivisor <= MaxTrial LOOP
         IF Number REM TrialDivisor = 0 THEN
            Composite := True;
         END IF;
         TrialDivisor := TrialDivisor + 2;    -- Try next odd integer
      END LOOP;
      RETURN NOT Composite;
   END IsPrime;

   N : Integer;                   -- Upper limit of search (user input)
   Candidate : Integer := 3;      -- Candidate primes (we know 2 is prime)
   NumPrimes : Integer := 1;      -- Count of primes found so far, ie
                                  -- counting the prime 2

BEGIN   -- Main Program
   Put("Please input a positive integer >= 2 ");
   Get(N);
   New_Line(2);

   Put(Item => 2, Width => 10);   -- We know 2 is prime
   WHILE Candidate <= N LOOP
      IF IsPrime(Candidate) THEN
         Put(Item => Candidate, Width => 10);
         NumPrimes := NumPrimes + 1;
         -- If the output line is full, start a new one.
         IF NumPrimes REM NumberPerLine = 0 THEN New_Line; END IF;
      END IF;
      Candidate := Candidate + 2;     -- Next odd integer
   END LOOP;

   New_Line(2);
   Put("The number of primes <= "); Put(N);
   Put(" is "); Put(NumPrimes); New_Line(2);

   Put("The number predicted by the Prime Number Theorem is ");
   Put(Integer(Float(N)/Log(Float(N)))); New_Line;
END Primes;
```