2

The Basic SPARK Language

SPARK is a programming language based on Ada. The syntax and semantics of the Ada language are defined in the *Ada Reference Manual* (ARM, 2012). The SPARK *Reference Manual* (SPARK Team, 2014a) contains the specification of the subset of Ada used in SPARK and the aspects that are SPARK specific. As stated in Chapter 1, a major goal of SPARK 2014 was to embody the largest possible subset of Ada 2012 amenable to formal analysis. The following Ada 2012 features are not currently supported by SPARK:

- Aliasing of names; no object may be referenced by multiple names
- Pointers (access types) and dynamic memory allocation
- Goto statements
- Expressions or functions with side effects
- Exception handlers
- Controlled types; types that provide fine control of object creation, assignment, and destruction
- Tasking/multithreading (will be included in future releases)

This chapter and Chapter 3 cover many, but not all, of the features of Ada 2012 available in SPARK. We discuss those features that are most relevant to SPARK and the examples used in this book. We assume that the reader has little, if any, knowledge of Ada. Barnes (2014) presents a comprehensive description of the Ada programming language. Ben-Ari (2009) does an excellent job describing the aspects of Ada relevant to software engineering. Dale, Weems, and McCormick (2000) provide an introduction to Ada for novice programmers. Ada implementations of the common data structures can be found in Dale and McCormick (2007). There are also many Ada language resources available online that you may find useful while reading this chapter, including material by English (2001), Riehle (2003), and Wikibooks (2014).

Let us start with a simple example that illustrates the basic structure of an Ada program. The following program prompts the user to enter two integers and displays their average.

```
1  with Ada.Text_IO;
2  with Ada.Integer_Text_IO;
3  with Ada.Float_Text_IO;
4  procedure Average is
5     -- Display the average of two integers entered by the user
6     A : Integer;    -- The first integer
7     B : Integer;    -- The second integer
8     M : Float;      -- The average of the two integers
9  begin
10    Ada.Text_IO.Put_Line (Item => "Enter two integers.");
11    Ada.Integer_Text_IO.Get (Item => A);
12    Ada.Integer_Text_IO.Get (Item => B);
13    Ada.Text_IO.New_Line;
14
15    M := Float (A + B) / 2.0;
16
17    Ada.Text_IO.Put (Item => "The Average of your two numbers is ");
18    Ada.Float_Text_IO.Put (Item => M,
19                           Fore => 1,
20                           Aft  => 2,
21                           Exp  => 0);
22    Ada.Text_IO.New_Line;
23 end Average;
```

The first three lines of the program are *context items*. Together, these three context items make up the *context clause* of the program. The three *with clauses* specify the library units our program requires. In this example, we use input and output operations from three different library units: one for the input and output of strings and characters (Ada.Text_IO), one for the input and output of integers (Ada.Integer_Text_IO), and one for the input and output of floating point real numbers (Ada.Float_Text_IO). The bold words in all our examples are *reserved words*. You can find a list of all seventy-three reserved words in section 2.9 of the ARM (2012).

Following the context clause is the specification of our program on line 4. In this example, the specification consists of the name Average and no parameters. The name is repeated in the last line that marks the end of this program unit. Comments start with two adjacent hyphens and extend to the end of the line. In our listings, comments are formatted in italics.

Following the program unit's specification is the *declarative part* (lines 5–8). In our example, we declared three variables. Variables A and B are declared to be of type Integer, a language-defined whole number type with an implementation-defined range. Variable M is declared to be of type Float, a language-defined floating point number type with an implementation-defined precision and range. The initial value of all three variables is not defined.

The executable statements of the program follow the reserved word **begin**. All but one of the executable statements in our example are calls to procedures (subprograms) defined in various library packages. In the first statement, we call the procedure Put_Line in the library package Ada.Text_IO.

Except for procedure New_Line, all of the procedures called in the example require parameters. Ada provides both *named* and *positional* parameter association. You are probably very familiar with positional parameter association in which the formal and actual parameters are associated by their position in the parameter list. With named parameter association, the order of parameters in our call is irrelevant. Our example uses named association to match up the formal and actual parameters. To use named association, we give the name of the formal parameter followed by the arrow symbol, =>, followed by the actual parameter. In our call to procedure Put_Line, the formal parameter is Item and the actual parameter is the string literal of our prompt. When there is only a single parameter, named parameter association provides little useful information. When there are multiple parameters, however, as in the call to Ada.Float_Text_IO.Put, named parameter association provides information that makes both reading and writing the call easier. The formal parameters Fore, Aft, and Exp in this call supply information on how to format the real number. Details on the formatting of real numbers are given in section A.10.9 of the ARM (2012).

The only statement in our example that is not a procedure call is the assignment statement on line 15 that calculates the average of the two integers entered by the user. The arithmetic expression in this assignment statement includes three operations. First, the two integers are added. Then the integer sum is explicitly converted to a floating point number. Finally, the floating point sum is divided by 2. The explicit conversion (casting) to type Float is necessary because Ada makes no implicit type conversions. The syntax of an explicit type conversion is similar to that of a function call using the type as the name of the function.

You may feel that program Average required more typing than you would like to do. Prefixing the name of each library procedure with its package name (e.g., Ada.Integer_Text_IO) may seem daunting. Of course, most integrated development environments such as the GNAT Programming Studio (GPS) provide smart code completion, making the chore easier. Ada provides another

alternative, the *use clause*, which allows us to make use of resources from program units without having to prefix them with the unit name. It is possible that different program units define resources with the same name. For example, both Ada.Text_IO and Ada.Integer_Text_IO include a subprogram called Put. The compiler uses the signature of the subprogram call to determine which Put to call. You will see an error message if the compiler is not able to resolve a name. A shorter version of our Average program that illustrates the use clause follows. We have also used positional parameter association to shorten the parameter lists.

```
with Ada.Text_IO;              use Ada.Text_IO;
with Ada.Integer_Text_IO;      use Ada.Integer_Text_IO;
with Ada.Float_Text_IO;        use Ada.Float_Text_IO;
procedure Average is   -- Shorter version with "use clauses"
   -- Display the average of two integers entered by the user
   A : Integer;     -- The first integer
   B : Integer;     -- The second integer
   M : Float;       -- The average of the two integers
begin
   Put_Line ("Enter two integers.");
   Get (A);
   Get (B);
   New_Line;

   M := Float (A + B) / 2.0;

   Put ("The Average of your two numbers is ");
   Put (M, 1, 2, 0);
   New_Line;
end Average;
```

Throughout the remainder of this book we will make use of prefixing and named parameter association when it makes the code clearer for the reader.

One final note before leaving this simple program. Ada is case insensitive. Thus, the three identifiers – Total, total, and tOtAl – are equivalent. It is considered bad practice to use different casings for the same identifier. A commonly used switch for the GNAT compiler requests that warnings be given for uses of different casings.

2.1 Control Structures

Ada provides two statements for making decisions on the basis of some condition: the if statement and the case statement. Section 5.3 of the ARM (2012)

provides the details of the if statement and section 5.4 provides the details of the case statement. Ada provides a loop statement with several different iteration schemes. These schemes are described in detail in section 5.5 of the ARM. In this section we will provide examples of each control structure.

2.1.1 *If Statements*

Following are some examples of various forms of the if statement.

```
if A < 0 then
   A := -A;
   D := 1;
end if;
```

```
if A in 1 .. 12 then
   B := 17;
end if;
```

```
if A > B then
   E := 1;
   F := A;
else
   E := 2;
   F := B;
end if;
```

```
if A = B then
   F := 3;
elsif A > B then
   F := 4;
else
   F := 5;
end if;
```

```
if A >= B and A >= C then
   G := 6;
elsif B > A and B > C then
   G := 7;
elsif C > A and C > B then
   G := 8;
end if;
```

Ada provides the following equality, relational, logical, and membership operators commonly used in the Boolean expressions of if statements.

Equality Operators	
=	equal
/=	not equal

Relational Operators	
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Logical Operators	
not	logical negation
and	logical conjunction
or	logical disjunction
xor	exclusive or
and then	short circuit and
or else	short circuit or

Membership Operators	
in	not in

Boolean expressions that include both **and** and **or** operators must include parentheses to indicate the desired order of evaluation. Section 4.5 of the ARM (2012) gives a complete listing and description of all of Ada's operators and the six precedence levels.

2.1.2 *Case Statement*

The case statement selects one of many alternatives on the basis of the value of an expression with a discrete result. An example of a case statement follows.

```
Success := True;
case Ch is
   when 'a' .. 'z' =>
      H := 1;
   when 'A' .. 'Z' =>
      H := 2;
   when '0' .. '9' =>
      H := 3;
   when '.' | '!' | '?' =>
      H := 4;
   when others =>
      H := 5;
      Success := False;
end case;
```

The case selector may be any expression that has a discrete result. In our example, the expression is the character variable `Ch`. Variable `Ch` is of the language-defined type `Character`, a type whose 256 values correspond to the 8-bit Latin-1 values. Ada also provides 16-bit and 32-bit character types, which are described in section 3.5.2 of the ARM (2012).

Our example contains five case alternatives. The determination of which alternative is executed is based on the value of the case selector. Each alternative is associated with a set of discrete choices. In our example, these choices are given by ranges (indicated by starting and ending values separated by two dots), specific choices (separated by vertical bars), and **others**, which handles any selector values not given in previous choice sets. The **others** alternative must be given last. Ada requires that there be an alternative for every value in the domain of the discrete case selector. The **others** alternative is frequently used to meet this requirement.

2.1.3 *Conditional Expressions*

Conditional expressions are not control structures in the classical sense. They are expressions that yield a value from the evaluation of one or a number of dependent expressions defined within the conditional expression. Conditional expressions can be used in places such as declarations and subprogram parameter lists where conditional statements are not allowed. Conditional expressions

can also reduce duplication of code snippets. Ada has two conditional expressions: the *if expression* and the *case expression*.

If Expressions

If expressions are syntactically similar to if statements but yield a value rather than alter the flow of control in the program. Because they yield a single value, if expressions are more limited than if statements. Following are three examples of if statements and, to their right, equivalent assignment statements using if expressions.

```
if A > B then                C := (if A > B then D + 5 else F / 2);
   C := D + 5;
else
   C := F / 2;
end if;

if A > B then                C := (if A > B then D + 5 elsif A = B then 2 * A else F / 2);
   C := D + 5;
elsif A = B then
   C := 2 * A;
else
   C := F / 2;
end if;

if X >= 0.0 then             Y := Sqrt (if X >= 0.0 then X else -2.0 * X);
   Y := Sqrt (X);
else
   Y := Sqrt (-2.0 * X);
end if;
```

Notice that the if expressions are enclosed in parentheses and do not include **end if**. You may nest if expressions within if expressions. The expressions such as D + 5, F / 2, and 2 * A within the if expression are called *dependent expressions*. In the last example, the if expression determines whether X or −2X is passed to the square root function.

If the type of the expression is Boolean, you may leave off the final else part of the if expression. The omitted else part is taken to be True by default. Thus, the following two if expressions are equivalent:

```
(if C - D = 0 then E > 2 else True)        (if C - D = 0 then E > 2).
```

Boolean if expressions are commonly used in preconditions, postconditions, and loop invariants. Here, for example, is a precondition that states that if A is less than zero then B must also be less than zero. However, if A is not less than zero, the precondition is satisfied (True).

```
Pre => (if A < 0 then B < 0);
```

This if expression implements the implication $A < 0 \rightarrow B < 0$. We look more at implications in Chapter 5 in our discussion of the basics of mathematical logic.

Case Expressions

Case expressions return a value from a number of possible dependent expressions. As you might expect, case expressions are syntactically similar to case statements. The following case expression assigns the Scrabble letter value for the uppercase letter in the variable `Letter`[1].

```
Value := (case Letter is
            when 'A' | 'E' | 'I' | 'L' | 'U' |
                 'N' | 'O' | 'R' | 'S' | 'T'     => 1,
            when 'D' | 'G'                       => 2,
            when 'B' | 'C' | 'M' | 'P'           => 3,
            when 'F' | 'H' | 'V' | 'W' | 'Y'     => 4,
            when 'K'                             => 5,
            when 'J' | 'X'                       => 8,
            when 'Q' | 'Z'                       => 10);
```

In this example the dependent expressions are all simple integer literals. Case expressions with Boolean dependent expressions are commonly used in preconditions, postconditions, and loop invariants. While we may leave off the final else part of an if expression, a case expression must have a dependent expression for every possible value the case selector (`Letter` in our example) may take. The compiler would complain if we had omitted any of the twenty-six uppercase characters.

2.1.4 *Loop Statements*

Ada's loop statement executes a sequence of statements repeatedly, zero or more times. The simplest form of the loop statement is the infinite loop. Although it may at first seem odd to have a loop syntax for an infinite loop, such loops are common in embedded software where the system runs from the time the device is powered up to when it is switched off. Following is an example of such a

loop in an embedded temperature controller. It obtains the current temperature from an analog-to-digital converter (ADC) and adjusts the output of a gas valve via a digital-to-analog converter (DAC).

```
loop
   ADC.Read (Temperature);          -- Read the temperature from the ADC
   Calculate_Valve  (Current_Temp => Temperature,  -- Calculate the new
                     New_Setting  => Valve_Setting); -- gas valve setting
   DAC.Write (Valve_Setting);       -- Change the gas valve setting
end loop;
```

We use an exit statement within a loop to terminate the execution of that loop when some condition is met. The exit statement may go anywhere in the sequence of statements making up the loop body. A loop that reads and sums integer values until it encounters a negative sentinel value follows. The negative value is not added to the sum.

```
Sum := 0;
loop
   Ada.Integer_Text_IO.Get (Value);
   exit when Value < 0;
   Sum := Sum + Value;
end loop;
```

There are two iteration schemes that may be used with the loop statement. The *while* iteration scheme is used to create a pretest loop. The loop body is executed while the condition is true. The loop terminates when the condition is false. The following loop uses a while iteration scheme to calculate an approximation of the square root of X using Newton's method.

```
Approx := X / 2.0;
while abs (X - Approx ** 2) > Tolerance loop
   Approx := 0.5 * (Approx + X / Approx);
end loop;
```

This program fragment uses two operators not found in some programming languages. The operator **abs** returns the absolute value of its operand, and ** is used to raise a number to an integer power.

The *for* iteration scheme is used to create deterministic counting loops. A simple example of this scheme is as follows.

```
for Count in Integer range 5 .. 8 loop
   Ada.Integer_Text_IO.Put (Count);
end loop;
```

As you can probably guess, this loop displays the four integers 5, 6, 7, and 8. Let us look at the details underlying the for iteration scheme. The variable Count in this example is called the *loop parameter*. The loop parameter is not defined in a declarative part like normal variables. Count is defined only for the body of this loop. The range 5 .. 8 defines a discrete subtype with four values. The body of the loop is executed once for each value in this discrete subtype. The values are assigned to the loop parameter in increasing order. Within the body of the loop, the loop parameter is treated as a constant; we cannot modify it. To make our loops more general, we can replace the literals 5 or 8 in our example with any expression that evaluates to an integer type. We will revisit this topic when we discuss types and subtypes later in this chapter.

If we add the reserved word **reverse** to the for loop, the values are assigned to the loop parameter in decreasing order. The following for loop displays the four numbers in reverse order.

```
for Count in reverse Integer range 5 .. 8 loop
   Ada.Integer_Text_IO.Put (Count);
end loop;
```

Reversing the order of the values in our example range creates a subtype with a *null range* – a subtype with no values. A for loop with a null range iterates zero times. Such a situation often arises when the range is defined by variables. Each of the following for loops displays nothing:

```
A := 9;
B := 2;

for Count in Integer range A .. B loop  -- With a null range, this
   Ada.Integer_Text_IO.Put (Count);     -- loop iterates zero times
end loop;

for Count in reverse Integer range A .. B loop  -- With a null range, this
   Ada.Integer_Text_IO.Put (Count);             -- loop iterates zero times
end loop;
```

2.2 Subprograms

A *subprogram* is a program unit whose execution is invoked by a subprogram call. Ada provides two forms of subprograms: the *procedure* and the *function*. We use a procedure call statement to invoke a procedure. You saw examples of procedure call statements in the program Average at the beginning of this chapter. We invoke a function by using its name in an expression. A function returns a value that is used in the expression that invoked it.

The definition of a subprogram can be given in two parts: a declaration defining its signature and a body containing its executable statements. The specification for package Sorters on page 13 includes the declaration of procedure Selection_Sort. Alternatively, we can skip the subprogram declaration and use the specification at the beginning of the body to define the signature. We will take this second approach in this section and use separate declarations when we discuss packages. Section 6 of the ARM (2012) provides the details on subprograms.

2.2.1 *Procedures*

Let us start with a complete program called Example that illustrates the major features of a procedure.

```
1  with Ada.Text_IO;
2  with Ada.Integer_Text_IO;
3  procedure Example is
4
5     Limit : constant Integer := 1_000;
6
7     procedure Bounded_Increment
8       (Value   : in out Integer;  -- A value to increment
9        Bound   : in     Integer;  -- The maximum that Value may take
10       Changed :    out Boolean)  -- Did Value change?
11    is
12    begin
13       if Value < Bound then
14          Value   := Value + 1;
15          Changed := True;
16       else
17          Changed := False;
18       end if;
19    end Bounded_Increment;
20
21    Value    : Integer;
22    Modified : Boolean;
23
24 begin -- procedure Example
25    Ada.Text_IO.Put_Line ("Enter a number.");
26    Ada.Integer_Text_IO.Get (Value);
27    Bounded_Increment (Bound   => Limit / 2,
28                       Value   => Value,
29                       Changed => Modified);
```

```
30   if Modified then
31         Ada.Text_IO.Put_Line ("Your number was changed to ");
32         Ada.Integer_Text_IO.Put (Item  => Value,
33                                  Width => 1);
34      end if;
35   end Example;
```

The first thing you might notice is that our program is itself a procedure. It is called the *main procedure*. Each procedure consists of a declarative part where all of its local resources are defined and a sequence of statements that are executed when the procedure is called. The declarative part of procedure Example contains four declarations: the named constant Limit, the procedure Bounded_Increment, and the two variables Value and Modified. Named constants are assigned values that we may not change. This program also introduces the language-defined type Boolean with possible values True and False.

Execution of our program begins with the executable statements of the main procedure (line 25 of procedure Example). The first statement executed is the call to procedure Put_Line that displays the prompt "Enter a number." The program then obtains a value from the user, calls procedure Bounded_Increment, and finally, based on the actions of the procedure just called, it may display a message.

Parameter Modes

Many programming languages require that programmers assign parameter passing mechanisms such as pass-by-value and pass-by-reference to their parameters. Ada uses a higher level means based on the direction of data flow of the parameter rather than the passing mechanism. Procedure Bounded_Increment illustrates all of the three different modes we can assign to a parameter.

in Used to pass data from the caller into the procedure. Within the procedure, an in mode parameter is treated as a constant. The actual parameter may be any expression whose result matches the type of the formal parameter. In our example, parameter `Bound` has mode in.

out Used to pass results out of the procedure back to its caller. You should treat the formal parameter as an uninitialized variable. The actual parameter must be a variable whose type matches that of the formal parameter. In our example, parameter `Changed` has mode out.

in out Used to modify an actual parameter. A value is passed in, used by the procedure, possibly modified by the procedure, and returned to the caller. It is like an out mode parameter that is initialized to the value of the actual parameter. Because a value is returned, the actual parameter must be a variable. In our example, parameter `Value` has mode in out.

As you might imagine, the SPARK analysis tools make use of these parameter modes to locate errors such as passing an uninitialized variable as an in parameter.

Scope

The scope of an identifier determines where in the program that identifier may be used. We have already seen one example of scope in our discussion of the for loop. The scope of a loop parameter is the body of the loop. You may not reference the loop parameter outside the body of the loop. The scope of every other identifier in an Ada program is based on the notion of *declarative regions*. Each subprogram defines a declarative region. This region is the combination of the subprogram declaration and body. A declarative region is more than the declarative part we defined earlier.

Let us look at the declarative regions defined in program Example on page 28. The declarative region of procedure Example begins after its name (on line 3) and ends with its **end** keyword on line 35. Similarly, the declarative region for procedure Bounded_Increment begins just after its name (on line 7) and ends with its **end** keyword on line 19. Note that Bounded_Increment's declarative region is nested within the declarative region of Example. Also note that Bounded_Increment's declarative region contains the definition of its three parameters.

Where a particular identifier may be used is determined from two rules:

- The scope of an identifier includes all the statements following its definition, within the declarative region containing the definition. This includes all nested declarative regions, except as noted in the next rule.
- The scope of an identifier does not extend to any nested declarative region that contains a locally defined *homograph*.[2] This rule is sometimes called *name precedence*. When homographs exist, the local identifier takes precedence within the procedure.

Based on these rules, the variables Value and Modified may be used by the main procedure Example but not by procedure Bounded_Increment. The constant Limit could be used in both procedure Example and procedure Bounded_Increment. Because Limit is declared within procedure Example's declarative region, Limit is said to be *local* to Example. As Limit is declared in procedure Bounded_Increment's enclosing declarative region, Limit is said to be *global* to Bounded_Increment. The three parameters, Value, Bound, and Changed, are local to procedure Bounded_Increment.

Although global constants are useful, the use of global variables is usually considered a bad practice as they can potentially be modified from anywhere. We use the style of always declaring variables after procedures so that the

variables may not be accessed by those procedures. In Chapter 3, we use global variables whose scope is restricted to implement variable packages. We may include a global aspect to indicate that a procedure accesses or does not access global variables. Should a procedure violate its stated global access, the SPARK tools will give an error message.

2.2.2 *Functions*

Functions return a value that is used in the expression that invoked the function. While some programming languages restrict return values to scalars, an Ada function may return a composite value such as an array or record. SPARK restricts all function parameters to mode *in*. This mode restriction encourages programmers to create functions that have no side effects. For the same reason, SPARK functions may use but not modify global variables. Here is an example of a function that is given a real value and acceptable error tolerance. It returns an approximation of the square root of the value.

```
function Sqrt (X : in Float; Tolerance : in Float) return Float is
   Approx : Float;   -- An approximation of the square root of X
begin
   Approx := X / 2.0;
   while abs (X - Approx ** 2) > Tolerance loop
      Approx := 0.5 * (Approx + X / Approx);
   end loop;
   return Approx;
end Sqrt;
```

The signature of this function includes its parameters and the type of the value that it returns. Approx is a local variable that holds our approximation of the square root of X. Execution of the **return** statement completes the execution of the function and returns the result to the caller. You may have multiple return statements in a function. Should you need to calculate a square root in your programs, it would be better to use a function from a library rather than our example code.

Expression Functions

Expression functions allow us to write functions consisting of a single expression without the need to write a function body. The executable code of an expression function is an expression written directly within the specification of the function. Here, for example, is a Boolean expression function that returns

True if the second parameter is twice the first parameter regardless of the sign of either number:

```
function Double ( First  : in Integer ;
                  Second : in Integer ) return Boolean is
   (abs Second = 2 * abs First );
```

Note that the expression implementing function Double is enclosed in parentheses. A common use of expression functions is within assertions such as preconditions, postconditions, and loop invariants. Replacing complex expressions with well-named function calls can make assertions easier to read. Of course, we can also easily reuse the expression in other assertions.

2.3 Data Types

A computer program operates on data stored in memory. Our programs reference this data through symbolic names known as *objects*. An object is declared as either variable or constant. In Ada, every object must be of a specific type. The *data type* establishes the set of possible values that an object of that type may take (its domain) and the set of operations that can be performed with those values. For example, an integer type might have a domain consisting of all of the whole numbers between $-2,147,483,648$ and $+2,147,483,647$. Operations on these numbers typically include the arithmetic operators ($+$, $-$, $\times$, $\div$), equality operators ($=$, $\neq$), and relational operators ($<$, $>$, $\leq$, $\geq$).

The primary predefined types in Ada are Boolean, Character, Integer, Float, Duration, and String. You are familiar with most of these types from your prior programming experiences. Duration is a real number type used to keep track of time in seconds. As you saw in the examples earlier in this chapter, an Ada variable is declared by writing its name followed by the name of its type. Here are some more examples:

```
Found    : Boolean;
Letter   : Character;
Count    : Integer ;
Distance : Float;
```

Constant declarations are similar but require the addition of the word **constant** and a value:

```
Pay_Taxes : constant Boolean   := True;
Negative  : constant Character := 'N';
Maximum   : constant Integer   := 1_247_962;
-- Note the use of underscores to
```

```
Origin     : constant Float     := 0.0;
-- improve readability of the literal
File_Name : constant String     := "data.txt";
```

Ada allows us to define our own simple and complex types. Using these types, we can create accurate models of the real world and provide valuable information to the SPARK tools so we can identify errors before the program is executed. Let us look at an example program with an obvious error:

```
with Ada.Float_Text_IO;
procedure Bad_Types is
   Room_Length    : Float;    -- length of room in feet
   Wall_Thickness : Float;    -- thickness of wall in inches
   Total          : Float;    -- in feet
begin
   Ada.Float_Text_IO.Get (Room_Length);
   Ada.Float_Text_IO.Get (Wall_Thickness);
   Total := Room_Length + 2.0 * Wall_Thickness;
   Ada.Float_Text_IO.Put (Item => Total, Fore => 1, Aft => 2, Exp => 0);
end Bad_Types;
```

In this example we have defined three variables, each of which holds a real number. The programmer ignored the comments given with each of these variable declarations and neglected to convert the wall thickness measurement from inches to feet before adding it to the room length measurement. Although the error in this short program is obvious, finding similar errors in large programs requires a great deal of effort in testing and debugging. Ada's type model helps eliminate a wide class of errors from our programs. However, as the example illustrates, we can still have such errors in our Ada programs when we do not take full advantage of the type system to model our values.

SPARK's types are a subset of Ada's types. Ada's access types (pointers) and controlled types are not amenable to formal analysis and therefore not part of SPARK. At the time of this writing, SPARK does not support task types. We expect that to change in the near future. Figure 2.1 shows the relationships among SPARK's various types. The hierarchy in this figure is similar to a class inheritance hierarchy. Type Boolean *is an* enumeration type. An enumeration type *is a* discrete type. The types whose names are in italics in Figure 2.1, such as *Scalar*, are abstract entities used to organize the classification of types. The set of operations available for all Ada types[3] include assignment (:=) and equality testing (= and /=).

Figure 2.1 shows that types are divided into two groups: atomic and composite. A composite type is one whose values may be decomposed into smaller

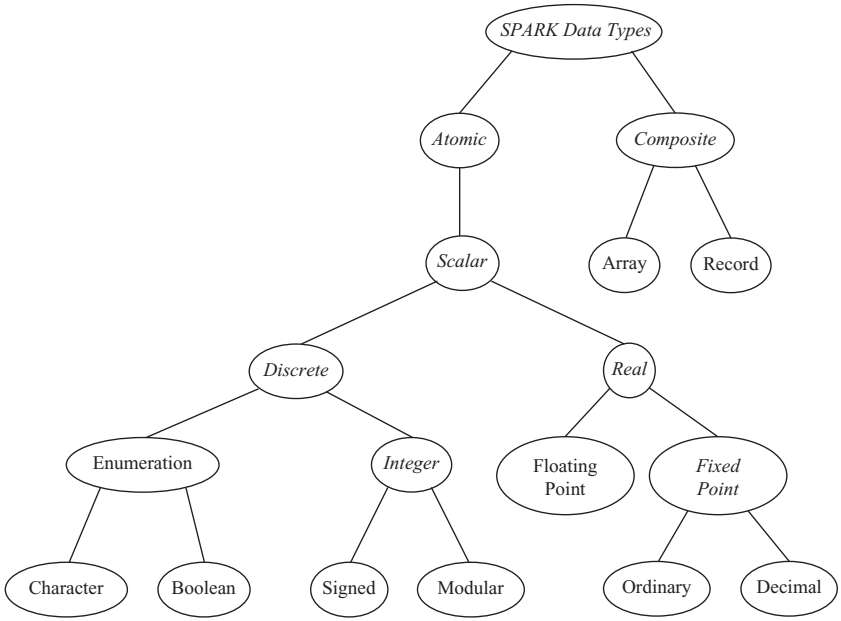


Figure 2.1. The SPARK type hierarchy.

values. A string type is a composite type. A string value is composed of characters. We can access and use the individual characters making up a string. An atomic type is one whose values cannot be decomposed into smaller values. A character type is an atomic type. A character value cannot be decomposed into smaller values. Integers and real numbers are also atomic types.

2.3.1 *Scalar Types*

A scalar type is an atomic type with the additional property of ordering. We can compare scalar values with the relational operators (<, <=, >, and >=). Characters, integers, and real numbers are all scalar types.

One of the principles of object-oriented programming is the development of classes that accurately model the objects in the problem. We can apply this same approach to the design of our scalar types. By using scalar types that more accurately reflect the nature of the data in a problem we are solving, we can write better programs. One research study on the nature of costly software faults indicates that poor models of scalar quantities were responsible for nearly 90 percent of the errors in the cases studied (Eisenstadt, 1997; McCormick, 1997). Ada allows programmers to define their own scalar data types that accurately model the scalar values in the problem domain.

Figure 2.1 shows that there are two kinds of scalar types. Real types provide the mechanisms for working with real numbers. A discrete type is a scalar type with the additional property of unique successors and predecessors. We will look at specific real and discrete types in the next sections.

Real Types

The storage and manipulation of real numbers is the substance of the discipline of numerical analysis. The underlying problem with computations involving real numbers is that very few real numbers can be represented exactly in a computer's memory. For example, of the infinite number of real numbers in the interval between 10,000.0 and 10,001.0, only 1,024 are represented exactly in the IEEE 754 single precision representation. The numbers with exact representations are called *model numbers*. The remaining numbers are approximated and represented by the closest model number. The representational error for a particular real number is equal to the difference between it and the model number used to represent it.

Floating Point Types

Here is a revised version of our simple program for adding room dimensions. In place of the language-defined type Float, we have defined two new floating point types: Feet and Inches. Because the variables Room_Length and Wall_Thickness are now different types, the SPARK tools will catch the inappropriate addition of feet and inches we had in our earlier erroneous program.

```
with Ada.Text_IO;
procedure Good_Types is
   -- Declarations of two floating point types
   type Feet    is digits 4 range 0.0 .. 100.0;
   type Inches  is digits 3 range 0.0 .. 12.0;

   -- Instantiation of input/output packages for feet and inches
   package Feet_IO is new Ada.Text_IO.Float_IO (Feet);
   package Inch_IO is new Ada.Text_IO.Float_IO (Inches);

   function To_Feet (Item : in Inches) return Feet is
   begin
      return Feet (Item) / 12.0;
   end To_Feet;

   Room_Length    : Feet;
   Wall_Thickness : Inches;
   Total          : Feet;
```

```
begin
   Feet_IO.Get (Room_Length);
   Inch_IO.Get (Wall_Thickness);
   Total := Room_Length + 2.0 * To_Feet (Wall_Thickness);
   Feet_IO.Put (Item => Total, Fore => 1, Aft => 1, Exp => 0);
end Good_Types;
```

The addition of feet and inches requires that we convert a value from one unit to another. We have included a function that makes this conversion. The function To_Feet first does an explicit type conversion (3 inches is converted to 3 feet), which is then divided by 12 to complete the unit conversion.

Our two new floating point types are defined by the type definitions at the beginning of the program. To define a new floating point type, we must specify the minimum number of decimal digits we require in the mantissa in the floating point numbers. This number follows the word **digits** in the type definition. The specification of a range for a floating point type is optional. If the range is omitted, the compiler will create a floating point type with the widest range possible.

We select the minimum number of digits in the mantissa based on the expected precision of our largest value. For our room length, we selected 100 feet as the upper bound of our domain. We estimated that the precision of a measurement of a 100-foot-long room is one-tenth of a foot. Therefore, we need four digits of precision to represent 100.0 – three account for the digits to the left of the decimal point and one for the digit to the right of the decimal point. Should we use a laser range finder with a precision of a thousandth of a foot in place of a tape measure, we would increase the number of digits of precision to six, three on each side of the decimal point. Similarly, we estimated that the precision of a measurement of a 12-inch-thick wall is one-tenth of an inch. So we need a total of three digits of precision for our wall thickness type. The precisions we select are minimums we will accept. The compiler will select the most efficient floating point representation available on the hardware with at least the precision we specify. The most common representations used are those specified by the IEEE 754 standard for floating point representation. We usually consider the precisions specified in our floating point type definitions as documentation on the precision of our actual data.

We cannot use the procedures in the library package Ada.Float_Text_IO to do input and output with values of type Feet and Inches. Ada provides a generic library package that may be instantiated to obtain packages for doing input and output with our own floating point types. You can see the two instantiations for packages Feet_IO and Inch_IO immediately following the definitions of our

two floating point types. We talk more about Ada's generic facilities in Sections 2.4.2 and 3.3.3.

Fixed Point Types

As illustrated in Figure 2.1, Ada provides support for two representations of real numbers: fixed point and floating point. Fixed point numbers provide a fixed number of digits before and after the radix point. When we write a real number on paper, we usually use a fixed point format such as

12.75 0.00433 1258.1

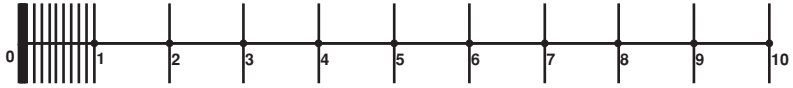
In a floating point number, the radix point may "float" to any location. Floating point is the computer realization of scientific notation. A floating point value is implemented as two separate numbers: a mantissa and an exponent. The following are all valid representations of 1,285.1:

$.12581 \times 10^4$ 1.2581×10^3 12.581×10^2
125.81×10^1 1258.1×10^0 $12581. \times 10^{-1}$

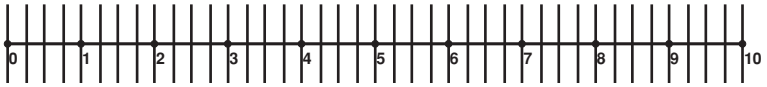
Floating point is by far the more commonly used representation for real numbers. In most programming languages, floating point is the only type available for representing real numbers. Floating point types support a much wider range of values than fixed point types. However, fixed point types have two properties that favor their use in certain situations. First, fixed point arithmetic is performed with standard integer machine instructions. Integer instructions are typically faster than floating point instructions. Some inexpensive embedded microprocessors, microcontrollers, and digital signal processors do not support floating point arithmetic. In such cases, fixed point is the only efficient representation available for real numbers.

The second advantage of fixed point is that the maximum representational error is constant throughout the range of the type. The maximum representational error for a floating point type depends on the magnitude of the number. This difference is a result of the distribution of model numbers in each of the representations. The distance between model floating point numbers varies through the range; it depends on the value of the exponent. The distance between model fixed point numbers is constant throughout the range.

Figure 2.2 illustrates the difference in model number distributions. Figure 2.2a shows the model numbers for a very simple floating point representation. There are ten model numbers between 1.0 and 10.0, ten model numbers between 0.1 and 1.0, and ten model numbers between 0.01 and 0.1. Figure 2.2b shows the model numbers for a simple fixed point representation. The distance between model numbers is constant throughout the range. Figure 2.2 shows



a. A simple floating point type (digits = 1)



b. A simple fixed point type (delta = 0.25)

Figure 2.2. Distribution of model numbers.

that the representational error for a floating point number gets larger as the number gets larger, whereas the representational error for a fixed point number is constant throughout its range.

Does the choice of real number representation really make a difference in our applications? "On February 25, 1991, a Patriot missile defense system operating at Dhahran, Saudi Arabia, during Operation Desert Storm failed to track and intercept an incoming Scud [missile]. This Scud subsequently hit an Army barracks, killing 28 Americans" Blair, Obenski, and Bridickas (1992). The Patriot battery failed because of a software problem related to the storage and use of floating point numbers. The system stored time, in tenths of a second, in a floating point variable. Table 2.1, taken from the Government Accounting Office report, shows the magnitude of the error in representing this time as a floating point value. As with all floating point representations, the magnitude of the error increases with the magnitude of the value.

Table 2.1. Magnitude of range gate error when modeling time as a floating point real number

Time		Absolute inaccuracy (Seconds)	Approximate shift in missile range gate (Meters)
Hours	Seconds		
0	0.0	0.0	0
1	3600.0	0.0034	7
8	28800.0	0.0275	55
20	72000.0	0.0687	137
48	172800.0	0.1648	330
72	259200.0	0.2472	494
100	360000.0	0.3433	687

Table 2.1 shows that the floating point representation error grows as the number grows. After twenty hours, the time is off enough that the target is outside the range gate and the Patriot missile fails to launch against a threat. After the tragedy, the software was corrected by replacing the floating point time variables with fixed point variables. Let us note that Ada's predefined type Duration is a fixed point type for seconds.

To declare a fixed point type, we specify the maximum distance between model numbers that we are willing to accept. The maximum representational error is half of this distance. We may also specify an optional range for the type. Here are two examples:

```
type Thirds is delta 1.0 / 3.0 range 0.0 .. 100_000.0;
type Volts  is delta 2.0**(-12) range 0.0 .. 5.0;
```

Thirds is a fixed point type with a specified distance of $\frac{1}{3}$ (0.33333...) between model numbers and Volts is a fixed point type with a specified distance of $\frac{1}{4096}$ (0.000244140625) between model numbers. Both of these types are called *ordinary fixed point types*. The actual distance between model numbers in our fixed point type may be smaller than our request. The actual distance between model numbers is the largest power of two that is less than or equal to the value given for delta. So although we specified a delta value of $\frac{1}{3}$ for Thirds, the actual delta used is the power of two, $\frac{1}{4}$ (2^{-2}). The delta we specified for Volts is a power of two so it is used directly. Because the distance between model numbers is some power of two, ordinary fixed point types are sometimes called *binary fixed point types*.

Neither floating point nor ordinary fixed point types are appropriate for currency calculations. Neither is capable of accurate storage of decimal fractions that are so important in commercial applications. Ada's *decimal fixed point types* are the more appropriate choice for such values. Here is an example:

```
type Dollars is delta 0.01 digits 12;
```

For decimal fixed point types, we must specify both a delta that is a power of ten and the number of decimal digits. A range is optional. A value of type Dollars contains twelve decimal digits. Because the distance between model numbers is 0.01, two of these digits are to the right of the decimal point, leaving ten digits for the left side of the decimal point.

We use the generic packages Ada.Text_IO.Fixed_IO and Ada.Text_IO.Decimal_IO to instantiate packages for the input and output of ordinary and decimal fixed point types. You may find the details for the

available I/O operations in section A.10.9 of the ARM (2012). Here are the instantiations for our example types:

```
package Thirds_IO is new Ada.Text_IO.Fixed_IO (Thirds);
package Volts_IO  is new Ada.Text_IO.Fixed_IO (Volts);
package Dollar_IO is new Ada.Text_IO.Decimal_IO (Dollars);
```

Ada's rules that prevent the mixing of different types are more relaxed for fixed point type multiplication and division. Multiplication and division are allowed between any two fixed point types. The type of the result is determined by the context. So, for example, if we assign the result of multiplying a Volts value and a Thirds value to a Volts variable, the result type of the multiplication would be Volts. Similarly, if we assign the same product to a Thirds variable, the result type of the multiplication would be Thirds. Additionally, a fixed point value may be multiplied or divided by an integer yielding the same fixed point type.

2.3.2 *Discrete Types*

Recall that a scalar type is an atomic type with the additional property of ordering. A discrete type is a scalar type with the additional property of unique successors and predecessors. The language-defined types Boolean, Character, and Integer are all discrete types. In the next sections we will look at defining our own discrete types.

Enumeration Types

An enumeration type provides a means for defining a type by enumerating (listing) all the values in the domain. The following program illustrates the definition and use of three enumeration types:

```
with Ada.Text_IO;                use Ada.Text_IO;
with Ada.Integer_Text_IO;        use Ada.Integer_Text_IO;
procedure Enum_Example is
   --Declaration of three enumeration types
   type Day_Type is (Monday, Tuesday, Wednesday, Thursday,
                     Friday, Saturday, Sunday);
   type  Traffic_Light_Color  is (Red, Green, Yellow);
   type  Pixel_Color          is (Red, Green, Blue, Cyan,
                                  Magenta, Yellow, Black, White);

   package Day_IO is new Ada.Text_IO.Enumeration_IO (Day_Type);

   function Next_Day (Day : in Day_Type) return Day_Type is
   begin
```

```
        if Day = Day_Type'Last then
           return Day_Type'First;
        else
           return Day_Type'Succ (Day);
        end if;
     end Next_Day;

     Today    : Day_Type;
     Tomorrow : Day_Type;
     Count    : Integer;
  begin
     Put_Line ("What day is today?");
     Day_IO.Get (Today);
     Tomorrow := Next_Day (Today);
     Put ("Tomorrow is ");
     Day_IO.Put (Item  => Tomorrow,
                 Width => 1,
                 Set   => Ada.Text_IO.Lower_Case);
     New_Line;

     if Today > Tomorrow then
        Put_Line ("Today must be Sunday");
     end if;
     New_Line;

     Put_Line ("The week days are ");
     for Day in Day_Type range Monday .. Friday loop
        Day_IO.Put (Day);
        New_Line;
     end loop;
     New_Line (2);

     for Color in Traffic_Light_Color loop
        Put_Line ( Traffic_Light_Color 'Image (Color));
     end loop;
     New_Line (2);

     Count := 0;
     for Color in Pixel_Color range Red .. Yellow loop
        Count := Count + 1;
     end loop;
     Put (Item => Count, Width => 1);
  end Enum_Example;
```

Each of our three enumeration types is defined by listing literals for all of the values in the domain. These literals are case insensitive. We could also have typed MONDAY or monday for the first value of Day_Type. Notice that Red is both a Pixel_Color literal and a Traffic_Light_Color literal.

We may instantiate packages for the input and output of enumeration values. In our example program, we instantiated the Ada library generic package Enumeration_IO to create the package Day_IO that allows us to get and put day values. You may find the details of the input and output operations available for enumeration values in section A.10.10 of the ARM (2012). Like the defining literals, the input values are not case sensitive. For output, we may select between all uppercase or all lowercase values. The first portion of the main procedure in our example calls procedures get and put in package Day_IO to get a day and display the next day.

Our main subprogram calls the function Next_Day to determine the day that follows the day entered by the user. This function has our first use of attributes. An *attribute* is an operator that yields a characteristic of a type or object. Some attributes require parameters. Here are the most common attributes for scalar types:

'First	Returns the lower bound of the type
'Last	Returns the upper bound of the type
'Image	Returns a string equivalent to the given value
'Value	Returns a value equivalent to the given string

And here are two additional attributes available for discrete types:

'Succ	Returns the successor of the given value
'Pred	Returns the predecessor of the given value

Let us look at the attributes used in program Enum_Example. The expression Day_Type'Last (read "day type *tick* last") in the if statement of our function uses the attribute 'Last to determine the last (largest) value in the domain of the type. As you might expect, the attribute 'First returns the first (smallest) value in the domain of the type. The attribute 'Succ requires a parameter. It returns the successor of the value passed to it. Because there is no successor for the value Sunday in type Day_Type, passing Sunday to the successor attribute function is an error. The purpose of the if statement in function Next_Day is to avoid this error by returning the first day of our type (Monday) for the one case in which the successor function fails. The attribute 'Pred returns the predecessor of a value passed to it. It is an error to use this function to determine the predecessor of the smallest value in the type's domain. You may find descriptions of the attributes available for all scalar types in section 3.5 of the ARM (2012). Additional

attributes for all discrete types are described in section 3.5.5. Sections 3.5.8 and 3.5.10 describe attributes available for all floating point and fixed point types.

The next portion of the main subprogram of our example illustrates the use of relational operators with enumeration values. These operators use the order of the literals in each enumeration type definition. Day_Type defines an order of days in which Monday is the smallest day and Sunday is the largest day. The if statement that asks whether Today is greater than Tomorrow is true only when Today is Sunday.

The remainder of our example program illustrates additional variations of the for loop. Our previous for loop examples used only integer loop parameters. A loop parameter may be of any discrete type. Recall that a loop parameter is not declared before the loop. Its type and range are defined by the discrete subtype definition following the reserved word **in**.

The second loop in our example program uses a type name without a range. This loop iterates through all three values of type Traffic_Light_Color displaying each value. We used another approach for displaying the traffic light colors in this loop. The 'Image attribute function returns an uppercase string equivalent to the enumeration parameter. We then used Ada.Text_IO.Put_Line to display this string. The advantage of the 'Image attribute is its simplicity. However, it does not provide the control available for formatting enumeration values available in the put procedures created in instantiations of enumeration I/O packages.

Integer Types

Most programmers use the integer type defined in their language for all variables that hold whole numbers. As we saw earlier in our room length example, using the same type for different quantities may result in logic errors requiring debugging effort. By again using different and appropriate types for our integers, we can have more confidence in our software. Ada provides both signed and unsigned integer types.

Signed Integers. To define a new signed integer type, we need only specify the range. Here are the definitions of three signed integer types and the declaration of a variable of each of those types:

```
type Pome      is range   0 .. 120;
type Citrus    is range −17 .. 30;
type Big_Range is range −20 .. 1_000_000_000_000_000_000;

Apples  : Pome;
Oranges : Citrus;
Fruit   : Big_Rang;
```

The range of each type is specified by a smallest and largest value, not by some storage unit size such as byte or word. Because they are different types, a comparison of Apples and Oranges is illegal. Of course, should you really want to combine apples and oranges in an expression, you can use explicit type conversions as in

```
Fruit := Big_Range (Apples) + Big_Range (Oranges);
```

Operations available for signed integers include +, −, *, /, **, **abs**, **rem**, and **mod**. The **rem** (remainder on division) and **mod** (mathematical modulo) operators return the same result when both of their operands are positive. Should one of your operands be negative, you should consult the formal definitions of these two similar operators given in section 4.5.5 of the ARM (2012). The attributes 'First, 'Last, 'Succ, 'Pred, and 'Image discussed for enumeration values are also available for signed integers.

To do input and output of our own integer types, we need to instantiate a package from the generic integer I/O package available in the Ada library. Here are the instantiations for the three integer types we defined:

```
package Pome_IO  is new Ada.Text_IO.Integer_IO (Pome);
package Citrus_IO is new Ada.Text_IO.Integer_IO (Citrus);
package Big_IO   is new Ada.Text_IO.Integer_IO (Big_Range);
```

Modular Integers. Modular integer types are unsigned integer types that use modular arithmetic. The value of a modular integer variable wraps around after reaching an upper limit. To define a modular integer type, we need only specify a modulus. Here are some modular integer type definitions and variable declaration:

```
type Digit   is mod 10;     -- range is from 0 to 9
type Byte    is mod 256;    -- range is from 0 to 255
type Nybble  is mod 16;     -- range is from 0 to 15
type Word    is mod 2**32;  -- range is from 0 to 4,294,967,295

Value : Nybble;
```

The following assignment statement illustrates the modular nature of this type:

```
Value := 12 + 8;    -- Value is assigned 4
```

In addition to the usual arithmetic operators, the logical operators **and**, **or**, **xor**, and **not** are available for modular integer types. These operators treat the values as bit patterns. The result of the **not** operator for a modular type is defined as the difference between the high bound of the type and the value of

the operand. For a modulus that is a power of two, this corresponds to a bit-wise complement of the binary representation of the value of the operand.

You may recall using the logical operators and bit masks in your assembly language, C, C++, or Java programs to clear, set, and toggle individual bits in a computer's memory. Thus, you might think that Ada's modular types provide the mechanism for Ada programs to manipulate individual bits. Ada provides a much higher level approach to bit manipulation. This topic is beyond the scope of this work. See McCormick, Singhoff, and Hugues (2011) for a full discussion of low level programming with Ada.

Again, we must instantiate packages to do input and output with the types we define. You can find the details on these packages and the get and put procedures in section A.10.8 of the ARM (2012). Here are the instantiations for our four modular types:

```
package Digit_IO  is new Ada.Text_IO.Modular_IO (Digit);
package Byte_IO   is new Ada.Text_IO.Modular_IO (Byte);
package Nybble_IO is new Ada.Text_IO.Modular_IO (Nybble);
package Word_IO   is new Ada.Text_IO.Modular_IO (Word);
```

2.3.3 *Subtypes*

By defining our own types, we make our programs easier to read and safer from type errors and allow range checking by the SPARK tools. In some cases, values with different constraints are related so closely that using them together in expressions is common and desired. Although explicit type conversion allows us to write such expressions, Ada provides a better solution – the *subtype*. Subtypes allow us to create a set of values that is a subset of the domain of some existing type. Subtypes inherit the operations from their base type. Subtypes are compatible with the type from which they were derived and all other subtypes derived from that type. A subset is defined by specifying an existing type and an optional constraint. Let us look at some examples:

```
subtype Uppercase is Character range 'A' .. 'Z';
subtype Negative  is Integer   range Integer'First .. -1;

type    Day_Type is (Monday, Tuesday, Wednesday, Thursday,
                     Friday, Saturday, Sunday);
subtype Weekday is Day_Type range Monday .. Friday;
subtype Weekend is Day_Type range Saturday .. Sunday;

type    Pounds      is digits 6 range 0.0 .. 1.0E+06;
subtype UPS_Weight  is Pounds   range 1.0 .. 100.0;
```

```
subtype FedEx_Weight is Pounds   range 0.1  ..  1.0;

subtype Column_Number is Ada.Text_IO.Count; -- A synonym

Total    : Pounds;
Box      : UPS_Weight;
Envelope : FedEx_Weight;
```

The domain of the subtype Uppercase is a subset of the domain of the language-defined type Character. Objects of subtype Uppercase may be combined with or used in place of objects of type Character. Similarly, the domain of the subtype Negative is a subset of the domain of the language-defined type Integer. The domain of subtypes Weekday and Weekend are both subsets of the programmer-defined type Day_Type. The following assignment statement illustrates the combining of subtypes with the same base type:

```
-- Adding two different subtypes with same base type
Total := Box + Envelope;
```

Subtype definitions may also be used to create synonyms – subtypes with the same domain as their base type. Synonyms are often used to provide a more problem specific name for a type whose name is more general or to eliminate the need to prefix a type name defined in a package. The subtype Column_Number is an example of such a synonym.

There are two commonly used language-defined subtypes defined in Ada. Positive is a subtype of Integer with a range that starts at one. Natural is a subtype of Integer with a range that starts at zero.

2.3.4 *Scalar Types and Proof*

Defining and using our own types and subtypes to accurately model real-world values helps prevent errors in our programs. Selecting appropriate ranges for variables provides an additional benefit when using the SPARK proof tools. Take, for example, the following assignment statement:

```
A := B / C;
```

To prove that the program containing this statement is correct requires the tool to prove that C can never be zero. This proof is much simpler if C is declared to be some type or subtype that does not include zero in its range (for example, subtype Positive). Here is another example whose proof can be simplified by using appropriate ranges:

```
A := (B + C) / 2;
```

Can you see the error lurking in this simple statement? If B and C are both very large numbers, their sum may exceed that of the processor's accumulator. To ensure that this statement is correct, the tool must prove that the sum of the two numbers cannot exceed the maximum accumulator value. By declaring types or subtypes with limited ranges for B and C, the proof tool's job is much easier. Using types and subtypes with appropriate ranges rather than the predefined types Integer and Float will reduce the effort both you and the proof tools expend to verify your program.

2.3.5 *Array Types*

Arrays are composite types whose components have the same type. We access a specific component by giving its location via an index. Defining an array type requires two types: one type or subtype for the component and one type or subtype for the index. The component may be any type. The index may be any discrete type. Here are some examples of array definitions:

```
type Index_Type       is range 1..1000;
type Inventory_Array  is array (Index_Type) of Natural;

subtype Lowercase        is Character range 'a' .. 'z';
type    Percent          is range 0..100;
type    Frequency_Array  is array (Lowercase) of Percent;

type Day_Type is (Monday, Tuesday, Wednesday, Thursday,
                  Friday, Saturday, Sunday);
type On_Call_Array is array (Day_Type) of Boolean;

Inventory : Inventory_Array;
Control   : Frequency_Array;
Unknown   : Frequency_Array;
On_Call   : On_Call_Array;
```

Variable Inventory is an array of 1,000 natural numbers indexed from 1 to 1,000. Control and Unknown are arrays of 26 percentages indexed from a to z. On_Call is an array of seven Boolean values indexed from Monday to Sunday.

Ada provides a rich set of array operations. Let us start with a selection operation. We use *indexing* to select a particular component in an array. We can use indexing to obtain a value from an array or change a value in an array. Here are some examples using the variables we just defined:

```
Inventory (5)    := 1_234;
Control ('a')    := 2 * Control ('a');
```

```
On_Call (Sunday) := False;

Total_Days := 0;
for Day in Day_Type loop
   if On_Call (Day) then
      Total_Days := Total_Days + 1;
  end if;
end loop;
```

Assignment is another operation available with arrays. As usual, strong typing requires that the source and target of an assignment statement be the same type. The following assignment statement makes the array Unknown a copy of the array Control.

```
Unknown := Control;
```

We can use the equality operators to compare two arrays of the same type. If the array components are discrete values, we can also compare two arrays with any of the relational operators. The relational operators are based on lexicographical order (sometimes called dictionary order) using the order relation of the discrete component type. We frequently use relational operators with arrays of characters (strings). Here is an example that uses two of the array variables we defined earlier:

```
if Control = Unknown then
   Put ("The values in the two arrays are identical");
elsif Control < Unknown then
   Put ("The values in Control come lexicographically before those in Unknown");
end if;
```

Slicing is another selection operation. It allows us to work with sub-arrays. We use slicing to read a portion of an array or write a portion of an array. A range is used to specify the portion of interest. Here are two examples of slicing:

```
-- Copy elements 11-20 into locations 1-10
Inventory (1 .. 10) := Inventory (11 .. 20);

-- Copy elements 2-11 into locations 1-10
Inventory (1 .. 10) := Inventory (2 .. 11);
```

Our second example illustrates slice assignment with overlapping ranges. Such assignments are useful in shuffling the components in an array when inserting or deleting components in an array-based list.

Although indexing and slicing both access components, their results are quite different. The indexing operation accesses a single component. The

slicing operation accesses an array. So while the expressions Inventory (5) and Inventory (5..5) are very similar, their result types are very different. Inventory (5) is a natural number, whereas Inventory (5..5) is an array consisting of one natural number.

Our slicing examples also illustrate the sliding feature of array assignment. The index ranges of the source and target of these assignments are different. The ten values in the source array slide into the target array. The range of the indices of the source and target may be different. The two restrictions for array assignment are that the target and source be the same type and that the number of components is the same.

We define multidimensional arrays by defining multiple indices. The following examples illustrate two ways to define an array in which we use two indices to locate a component:

```
type Row_Index is range  1 .. 1000;
type Col_Index is range −5 .. +5;

type Two_D_Array is array (Row_Index, Col_Index) of Float;

type One_Row           is array (Col_Index) of Float;
type Another_2D_Array is array (Row_Index) of One_Row;

Canary : Two_D_Array;       −− A 2−dimensional array variable
Finch  : Another_2D_Array; −− An array of arrays  variable
```

The syntax for indexing a two-dimensional array is different from that for indexing an array of arrays. Here are examples of each kind:

```
−− Assign zero to row 12, column 2 of the 2−D array
Canary (12, 2) := 0.0;

−− Assign zero to the second component of the 12th array
Finch (12)(2)  := 0.0;
```

Slicing is limited to one-dimensional arrays. We cannot slice the array variable Canary. However, we can slice an array in our array of arrays variable Finch.

Constrained and Unconstrained Array Types

All the previous array examples were of constrained arrays. A constrained array type is an array type for which there is an index range constraint. An

unconstrained array type definition provides only the type of the index, it does not specify the range of that type. Here is an example:

```
type Float_Array is array ( Positive range <>) of Float;
```

This statement defines the unconstrained array type Float_Array. The components of this array type are type Float and the index of this array is subtype Positive. We did not specify the range of the positive index. The box symbol, <>, indicates that this is the definition of an unconstrained array type. Because there is no range constraint for this array type, we cannot use an unconstrained array type to declare an array variable because the compiler cannot determine how much memory to allocate to such an array variable.

```
Illegal : Float_Array; -- This declaration will not compile
```

The two important uses of unconstrained array types are (a) as a base for a constrained array subtype and (b) for the type of a formal parameter. Here are examples of constrained array subtypes:

```
subtype Small_Array is Float_Array (1 .. 10);
subtype Large_Array is Float_Array (1000 .. 9999);

Small : Small_Array; -- An array of 10 Float values
Large : Large_Array; -- An array of 9,000 Float values
```

Because the arrays Small and Large have the same base type, we can combine them in expressions like these:

```
Large (1001 .. 1010) := Small;              -- Copy 10 values
if Small /= Large (2001 .. 2010) then       -- Compare 10 values
   -- Copy 21 values
   Large (2001 .. 2021) := Small & 14.2 & Small;
end if;
```

The second assignment statement in the preceding example illustrates another array operation, *concatenation*. The & operator may be used to concatenate two arrays, an array and a component, or two components. The result in all cases is an array. In our example, we created a 21 component array by concatenating a copy of array Small with the value 14.2 and a second copy of Small.

Here is an example of a subprogram specification with an unconstrained array parameter:

```
function Average (Values : in Float_Array) return Float;
```

The formal parameter Values will match any actual parameter that is a constrained array subtype of Float_Array. The formal parameter will take on the

index constraint of the actual parameter. Here are some calls to function Average using our two previously defined array variables:

```
Avg := Average (Small);                -- Average of 10 values
Avg := Average (Large);                -- Average of 9,000 values
Avg := Average (Large (2001..2010));   -- Average of 10 values
Avg := Average (Large & Small);        -- Average of 9,010 values
```

Array Attributes

As you just saw, we can pass different size arrays to function Average. We do not need to pass the size of the array or its starting or ending indices. The function makes use of attributes to obtain the properties of the actual array parameter. Earlier we introduced some of the most commonly used attributes for discrete types. There are attributes for array types as well. The attributes most commonly used with arrays are as follows:

'First	Returns the lower bound of the index range
'Last	Returns the upper bound of the index range
'Length	Returns the number of components in the array
'Range	Returns the index range of the array ('First .. 'Last)

Here is the body of function Average, which uses two of these attributes:

```
function Average (Values : in Float_Array) return Float is
   Sum : Float;
begin
   Sum := 0.0;
   for Index in Values'Range loop
      Sum := Sum + Values (Index);
   end loop;
   return Sum / Float (Values'Length);
end Average;
```

When working with unconstrained array parameters, you should not make any assumptions about the first or last index values. Although many arrays use 1 as a starting index, you should use the attributes rather than make such an assumption.

Section 3.6 of the ARM (2012) provides the details on array types and attributes. There you may also find how to use attributes with multidimensional arrays.

Array Aggregates

An array aggregate is a collection of components enclosed in parentheses. One might think of an array aggregate as an array literal. Aggregates are

commonly used to initialize array objects. Here are some examples of using array aggregates to initialize the array variable Small defined earlier:

```
Small := (0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 4.0, 3.0, 2.0, 1.0); -- by position
Small := (1 .. 10 => 0.0);                                    -- by name
Small := (1 .. 5 => 0.0, 6 .. 10 => 1.0);
Small := (1 | 3 | 5 | 7 | 9 => 0.0, others => 1.0);
Small := (others => A + Sqrt (B));
```

In the first example, the ten components assigned to the array Small are given by position: 0.0 is assigned to Small(1), 1.0 is assigned to Small(2), and so on. In the other examples, we use array indices as names to assign values to specific locations within the array. Similar to the case selector, we can use ranges and individual index values separated by vertical bars to associate locations with values. The **others** keyword may be used last to associate a value with locations not previously specified. The last example, which calculates a value from the variables A and B, demonstrates that aggregates are not true literals, but simply collections of components.

Array aggregates are available for multidimensional arrays. Let us look at an example. Here is a two-dimensional array of integers with three rows indexed from -1 to 1 and four columns indexed from 1 to 4:

```
type Row_Range  is range -1 .. 1;
type Col_Range  is range  1 .. 4;
type Table_Array is array (Row_Range, Col_Range) of Integer;

Table : Table_Array;  -- 2-dimensional array with 12 elements
```

And here are examples of using aggregates to assign values to Table:

```
-- Assign all elements by position
Table := ((1, 2, 3, 4),
          (5, 6, 7, 8),
          (9, 8, 7, 6));
-- Assign rows by name, columns by position
Table := (-1 => (1, 2, 3, 4),
           0 => (5, 6, 7, 8),
           1 => (9, 8, 7, 6));
-- Assign all elements by name
Table := (-1 => (1 .. 4 => 2),
           0 => (1 .. 3 => 3, 4 => 6),
           1 => (2 => 5, others => 7));
Table := (-1 .. 1 => (1 .. 4 => 0));
Table := (others => (others => 0));
```

Strings

We conclude our discussion of unconstrained arrays with a very brief discussion of Ada's predefined fixed-length string type. Type String is predefined as an unconstrained array of characters with a positive index.

```
-- Type String is defined in Ada.Standard as
type String is array ( Positive range <>) of Character;
```

Here are examples of declarations of a string subtype, a string constant, and several string variables:

```
subtype Name_String is String (1 .. 20);  -- A constrained array type

Currency : constant String := "Dollars";

Name    : Name_String;          -- A string containing 20 characters
Address : String (1 .. 40);     -- A string containing 40 characters
City    : String (3 .. 22);     -- A string containing 20 characters
```

Fixed-length strings are efficient but require more thought in their use than varying-length strings. For example, you may not assign a string of one length to a string of another length. We can make use of array slicing to make assignments to different length strings.

```
Name := Address; -- Illegal Assigning 40 characters to a 20 character string
Name := "Peter"; -- Illegal Assigning  5 characters to a 20 character string
Address := Name; -- Illegal Assigning 20 characters to a 40 character string

City := Name;                 -- Legal Both source and target contain 20 characters
Name := Address (1 .. 20);    -- Legal Both source and target contain 20 characters
Address := Name & Name;       -- Legal Both source and target contain 40 characters
Address (9 .. 28) := Name;    -- Legal Both source and target contain 20 characters
```

In the last statement, only twenty of the forty characters in the variable Address are changed by the assignment.

Should we want to store fewer characters in a fixed-length string than its length, we can make use of additional variables to keep track of the string's "real" length and use that variable to slice the string. Here is an example that demonstrates how we might store five characters in a fixed-length string with a length of twenty.

```
Count := 5;
Name (1 .. Count) := "Peter";
Put (Name (1 .. Count));          -- Display Peter
```

Here is a complete program that uses type String to illustrate constrained array subtypes, slicing, unconstrained array parameters, and array attributes:

```
with Ada.Text_IO;
procedure Palindrome is

   function Is_Palindrome (Item : in String) return Boolean is
      Left_Index  : Natural;  -- Two indices mark the beginning and
      Right_Index : Natural;  -- end of the unchecked portion of Item
   begin
      Left_Index  := Item'First;
      Right_Index := Item'Last;
      loop
         exit when Left_Index >= Right_Index or else
                   Item (Right_Index) /= Item (Left_Index);
         Left_Index  := Left_Index + 1;
         Right_Index := Right_Index - 1;
      end loop;
      return Left_Index >= Right_Index;
   end Is_Palindrome;

   Max_Length : constant Positive := 100;
   subtype Line_Type is String (1 .. Max_Length);

   Line  : Line_Type;  -- Characters entered by user
   Count : Natural;    -- Number of characters entered

begin
   Ada.Text_IO.Put_Line ("Enter a line.");
   -- Get_Line reads characters to end of line
   -- Last is the index of the last character read
   Ada.Text_IO.Get_Line (Item => Line,
                         Last => Count);
   -- Slice off garbage before calling Is_Palindrome
   if Is_Palindrome (Line (1 .. Count)) then
      Ada.Text_IO.Put_Line ("is a palindrome");
   else
      Ada.Text_IO.Put_Line ("is not a palindrome");
   end if;
end Palindrome;
```

The fixed-length string variable Line contains 100 characters. The variable Count keeps track of the index of the last "good" character in Line. The call to procedure Get_Line fills in Line with the characters typed at the keyboard. The

Get_Line procedure reads characters until either the string is filled (100 characters for Line) or it encounters a line terminator. If our user types the word `toot` and then presses the enter key, the first four characters of Line will be `t o o t` and the remaining ninety-six characters will be undefined. When we call the function Is_Palindrome, we slice off the garbage passing only the characters entered by the user.

2.3.6 *Record Types*

Arrays are homogeneous composite types – the components are all of the same type. Records are heterogeneous composite types – the components may be different types. In an array we access a specific component by giving its position in the collection. We access a specific component in a record by giving its name. The following declarations define a simple record type for an inventory system.

```
subtype Part_ID is Integer range 1000 .. 9999;
type    Dollars is delta 0.01 digits 7 range 0.0 .. 10_000.0;

type Part_Rec is
   record
      ID       : Part_ID;
      Price    : Dollars;
      Quantity : Natural;
   end record;

Part     : Part_Rec;
Discount : Dollars;
```

There are three components (fields) defined in type Part_Rec. Each component is identified by a name. The name is used with the variable name to select a particular component in the record. Here are some examples that illustrate component selection:

```
Part.ID       := 1234;
Part.Price    := 1_856.25;
Part.Quantity := 597;
Discount      := 0.15 * Part.Price;
```

Record Aggregates

In the previous example, we used three assignment statements to give the record variable Part a value. We can use a record aggregate to assign a value to a record variable with a single assignment statement. A record aggregate

is a record value written as a collection of component values enclosed with parentheses. The association of values in the aggregate and the record field may be given by position or by name. Here are examples of each:

```
Part := (1234, 1_856.25, 597); -- Assign values by position

Part := (ID       => 1234,    -- Assign values by name
         Quantity => 597,
         Price    => 1_856.25);
```

When using named association in a record aggregate, we can order the fields as we like.

Discriminants

We often parameterize record types with one or more discriminants. A *discriminant* is a record component on which other components may depend. Whereas ordinary record components can be any constrained type, discriminants are limited to discrete types. The following declarations use a discriminated record to define an array-based list of inventory records. The discriminant, Max_Size, is used to define the index constraint of the array component.

```
type Part_Array is array (Positive range <>) of Part_Rec;

type Inventory_List (Max_Size : Positive) is
   record
      Size  : Natural;
      Items : Part_Array (1..Max_Size);
   end record;
```

When we use a discriminated record in the declaration of an object, we supply an actual value for the discriminant. The following declaration defines an inventory list that holds a maximum of 1,000 part records:

```
Inventory : Inventory_List (Max_Size => 1000);
```

Inventory is a record with three components: Max_Size, Size, and Items. The last component is an array of part records. Here is some code that accesses the information in this data structure:

```
-- Append a new part to the inventory list
if Inventory.Size = Inventory.Max_Size then
   Put_Line ("The inventory list is full");
else
   Inventory.Size := Inventory.Size + 1;
   Inventory.Items (Inventory.Size) := New_Part;
end if;
```

2.3.7 *Derived Types*

Subtypes allow us to define subsets of existing types whose values may be combined with any other subtype that shares the same base type. Sometimes we would like to create a new type that is similar to an existing type yet is a distinct type. Here is an example of a definition of a derived type:

```
-- Define a floating point type
type Gallons is digits 6 range 0.0 .. 100.0;
-- Define a new type derived from Gallons
type Imperial_Gallons is new Gallons;
```

Type Imperial_Gallons is a derived type. Gallons is the *parent* of Imperial_Gallons. Derived types are not limited to scalar types. We can derive types from array and record types as well. The domain of the derived type is a copy of the domain of the parent type. Because they are different types, however, values of one type may not be assigned to objects of the other type. We may use explicit type conversions to convert a value of one type to a value of the other type.

The most common use of derived types is in the creation of class hierarchies associated with object-oriented programming. A detailed discussion of classes and inheritance is not in the scope of this book.

2.4 Subprograms, More Options

2.4.1 *Overloading*

We may have two subprograms with the same name as long as their signatures differ. The signature of a subprogram consists of the number, the types, and the order of the parameters. For functions, the type of the returned value is also considered. The formal parameter names and modes are not part of the signature. Here are examples of specifications of three different procedures with the same name:

```
procedure Calc (A : in  Integer;
                C : out Integer);

procedure Calc (A : in  Integer;
                C : out Float);

procedure Calc (A : in  Integer;
                B : in  Integer;
                C : out Integer);
```

We may also overload operators. Let us look at an example. Here is a record type for the coordinates of a point on a plane and two point variables:

```
type Point is      -- Cartesian coordinates of a point
   record
      X_Coord : Float;
      Y_Coord : Float;
   end record;

P1 : Point;
P2 : Point;
```

The following function, that overloads the <= operator, may be used to determine whether the Left point is the same distance from or closer to the origin than the Right point. Notice that we must enclose the operator symbols in double quotes.

```
function "<=" (Left : in Point; Right : in Point) return Boolean is
-- Returns True when Left is the same distance from
-- or closer to the origin than Right
   Left_Squared  : Float;
   Right_Squared : Float;
begin
   Left_Squared  := Left.X_Coord ** 2 + Left.Y_Coord ** 2;
   Right_Squared := Right.X_Coord ** 2 + Right.Y_Coord ** 2;
   return Left_Squared <= Right_Squared; -- Calls <= for Float numbers
end "<=";
```

Finally, some code that calls function "<=". The call may be made as either an infix operator or a normal function call (prefix operator). When the function call syntax is used, the operator must be enclosed in double quotes.

```
-- Function called as an infix operator
if P1 <= P2 then
   Put_Line ("P1 is not further from the origin than P2");
else
   Put_Line ("P1 is further from the origin than P2");
end if;

-- "Normal" function call
if "<=" (P1, P2) then ...
```

2.4.2 *Generic Subprograms*

Ada provides parameterized *generic units* for writing reusable software components. Generic units are templates that can be instantiated to create a unit for a specific application. While the initial effort to create a generic unit may be higher, the long-term savings can be substantial. Not only can we reuse the code in the template, but also we need only test one instance to verify all future instantiations.[4] We have already seen how to use generic packages to create packages for doing input and output with the types we define. Now let us look at writing our own generic units.

Suppose we have written the following function to count the number of times a particular character occurs in a string of characters:

```
function Count (Source  : in String;
                Pattern : in Character) return Natural is
-- Returns the number of times Pattern occurs in Source
   Result : Natural := 0;
begin
   for Index in Source'Range loop
      if Source (Index) = Pattern then
         Result := Result + 1;
      end if;
   end loop;
   return Result;
end Count;
```

While this function is specific for counting characters in a string, the same logic could be used for counting occurrences of objects in any array. For each different application we need different types for the parameters Source and Pattern. Rather than write a new counting function for each application, we can write a single generic function and instantiate it as appropriate. We supply the different types for each application through *generic parameters*. We define *generic formal parameters* in the specification of the generic unit and supply *generic actual parameters* when we instantiate the generic unit for a particular application. Ada provides many different kinds of generic formal parameters with specific rules for what actual parameters may be supplied. Table 2.2 shows the commonly used generic formal types with descriptions of what generic actual types are acceptable. See the ARM (2012), Barnes (2014), or Wikibooks (2014) for a complete list.

Let us start with our type for Pattern. This type is also the type of the components in the array we wish to process. There are no restrictions on what

Table 2.2. Some of Ada's generic formal types

Generic formal type	Acceptable generic actual types
type T **is range** <>;	Any signed integer type.
type T **is mod** <>;	Any unsigned integer (modular) type.
type T **is** (<>);	Any discrete type.
type T **is digits** <>;	Any floating point type.
type T **is delta** <>;	Any ordinary fixed point type.
type T **is delta** <> **digits** <>;	Any decimal fixed point type.
type T **is array** (Indx) **of** Cmp;	Any array type with index of type `Indx` and components of type `Cmp`. The formal and actual array parameters must both be constrained or both be unconstrained.
type T **is private**;	Any type for which assignment and equality testing are available (nonlimited).
type T **is limited private**;	Any type at all.

type the component of an array can be. We do, however, need to compare a component in our array to Pattern so this type must be one for which the equality operator is defined. The generic formal parameter type **private** meets this requirement. Any type that has assignment and equality testing may be supplied as an actual parameter for a private generic formal type – that is every type we have seen at this point.

The generic formal parameter type for the array is a little more complicated. We have three decisions to make:

1. What types should we allow for the index of the array?
2. Should the array type be constrained or unconstrained?
3. What types should we allow for the component of the array?

Our selection for the type of Pattern has already given us the answer to the third question – it can be any type. As an unconstrained array type can be of any length; it is more general than a constrained array type. Indeed, our original example's array type was an unconstrained array of characters indexed by positive numbers. We could choose to use a positive index for our array. However, we can be more general. The only restriction on the index of an array is that it must be a discrete type. The formal generic parameter type (<>) will match any discrete actual type. By now your head is probably spinning with all

of this new terminology so let us go right to the code that implements all this text. Here is the specification of a general object counting function:

```
generic
   type Component_Type is private; -- Any type with assignment and equality testing
   type Index_Type     is (<>);    -- Any discrete type
   type Array_Type     is array (Index_Type range <>) of Component_Type;
function Generic_Count (Source  : in Array_Type;
                        Pattern : in Component_Type) return Natural;
-- Returns the number of times Pattern occurs in Source
```

A generic unit begins with the keyword **generic**, followed by a list of generic formal parameters, followed by the unit specification. Let us review the choice of generic formal parameters in this example. Component_Type is a formal type that will match any actual type that allows assignment and equality testing. Index_Type is a formal type that will match any discrete actual type. Finally, Array_Type is a formal type that will match any actual unconstrained array type that is indexed by the actual type given for Index_Type and has components of the actual type given for Component_Type.

Now let us create some actual counting functions from our generic function. Here are the types for an unconstrained array of percentages indexed by character:

```
type Percent is range 0 .. 100;
type Percent_Array is array (Character range <>) of Percent;
```

Here is the instantiation of a function to count how many times a particular percentage value occurs in an array:

```
function Percent_Count is new Generic_Count
                        (Component_Type => Percent,
                         Index_Type     => Character,
                         Array_Type     => Percent_Array);
```

And here is a call to our newly created function using an array aggregate with ten values for the Source and the literal 5 for the Pattern:

```
The_Count := Percent_Count (Source  => (5, 6, 7, 5, 3, 4, 19, 16, 5, 23),
                           Pattern => 5);
```

Let us look at another instantiation of the generic function. This time, we will look at one that counts the number of times a particular character occurs

in a string – an equivalent of the nongeneric function we gave at the beginning of this section – and a sample call.

```
function Char_Count is new Generic_Count
                       (Component_Type => Character,
                        Index_Type     => Positive,
                        Array_Type     => String);

The_Count := Char_Count (Source => "How now brown cow",
                         Pattern => 'w');
```

Writing a separate specification for nongeneric subprograms is optional. Generic subprograms must be written with separate specifications and bodies. The executable code of Generic_Count is identical to the original function we wrote for counting character occurrences in a string.

```
function Generic_Count (Source  : in Array_Type;
                        Pattern : in Component_Type) return Natural is
   Result : Natural := 0;
begin
   for Index in Source'Range loop
      if Source (Index) = Pattern then
         Result := Result + 1;
      end if;
   end loop;
   return Result;
end Generic_Count;
```

When writing the body, we have no idea of what actual types will be supplied for Array_Type and Component_Type. So, what operations can we apply to the parameters Source and Pattern? All we know about Source is that it is an unconstrained array indexed by some discrete type. We made use of the array attribute 'Range and array indexing in this body. We have no idea of the type of the parameter Pattern. However, we do know that it supports assignment and equality testing. It is only the latter that we needed to implement the function.

Let us make this counting logic even more general. Perhaps we would like to know how many values in an array of percentages are greater than ninety. Or, how many values in an array of Cartesian points are inside a circle of radius of 1.0 centered on the origin. To accomplish these tasks, we need to replace the equality test in the if statement with a test for some other property. To accomplish this task, we make use of generic formal subprograms. As our last

example was of a generic function, this time we will write a generic procedure. Here is its specification:

```
generic
   type Component_Type is limited private; -- Any type
   type Index_Type     is (<>);            -- Any discrete type
   type Array_Type     is array (Index_Type range <>) of Component_Type;
   with function Selected (From_Source : in Component_Type;
                           Pattern     : in Component_Type) return Boolean;
procedure Tally (Source  : in Array_Type;
                 Pattern : in Component_Type;
                 Result  : out Natural);
-- Returns the number of items in Source that are selected for a given Pattern
-- Calls function Selected to determine if an element in Source qualifies
```

This generic procedure has three generic formal type parameters and one generic formal function parameter. We have changed the type of the generic formal parameter Component_Type to **limited private**. This change allows the component to be of any type, even if it should not have assignment and equality testing operations. The new generic formal parameter Selected is a function that takes two Component_Type parameters and returns a Boolean value. In our application, we can pass any actual function that has the same signature as function Selected. Here, for example, is an instantiation of this generic procedure that tallies the number of percentage values in an array that is greater than some value. The actual function passed for the generic formal function parameter Selected is the > for percents so it returns True when a percent value in the array Source is greater than the percentage in Pattern.

```
-- Instantiate a procedure to determine how many percentages
-- in an array indexed by characters are greater than some value
procedure Tally_Percents is new Tally (Component_Type => Percent,
                                       Index_Type     => Character,
                                       Array_Type     => Percent_Array,
                                       Selected       => ">");
```

Types Percent and Percent_Array are defined on page 61. Here is a sample call of the procedure we instantiated from the generic procedure Tally to count all the values in an array greater than five.

```
-- Determine how many values in My_Percents are greater than 5
Tally_Percents (Source  => My_Percents,
                Pattern => 5,
                Result  => The_Count);
```

Summary

SPARK is a subset of the Ada language. This is a summary of the Ada features in the SPARK subset that are discussed in this chapter.

- Ada provides all of the control structures expected in a high level programming language.
- If expressions and case expressions may be used to select a value from a number of dependent expressions.
- The loop statement and exit statement may be used to implement any iteration scheme.
- The for loop option may be used to implement counting loops.
- The while loop option may be used to implement pre-test loops.
- Ada provides two kinds of subprograms: procedures and functions.
- Parameter passing modes are based on direction of data flow not on the underlying passing mechanism.
- The nested structure of an Ada program provides a powerful mechanism for controlling the scope of identifiers.
- Ada's type model is perhaps the most important feature, giving Ada a significant advantage over other programming languages.
- Programmer-defined scalar types allow us to more accurately model the problem we are solving.
- Ada provides both floating point and fixed point representations for real numbers.
- Ada provides both signed and unsigned (modular) representations for integer numbers.
- Enumeration types allow us to create types by listing all possible values in the domain.
- Attributes provide information about a type or object.
- Subtypes allow us to create a set of values that is a subset of the domain of some existing type.
- We may index our arrays with any discrete data type; we are not limited to integer indices.
- Indexing allows us to access an element of an array.
- Slicing allows us to access a portion of an array; a slice is an array.
- Unconstrained array types allow us to define formal array parameters that match any size actual array parameter.
- Records are heterogeneous composite data types.
- Derived types allow us to create a new and different type from an existing type.

- Subprogram names may be overloaded provided each has a different signature. The signature consists of the number and types of parameters and, for functions, the type returned.
- Generic units allow us to write code that may be reused in other applications.

Exercises

2.1 What is the purpose of the *with clause*?

2.2 Where is the *declarative part* of an Ada program located?

2.3 Define the following terms:

a. formal parameter
b. actual parameter
c. dependent expression
d. loop parameter
e. scope
f. local variable
g. global variable
h. data type
i. model number
j. attribute

2.4 What is meant by *parameter association*? Describe how parameters are associated with *named parameter association* and *positional parameter association*.

2.5 What is the purpose of the *use clause*?

2.6 True or False? Ada is case sensitive.

2.7 Write an if statement that checks three integer variables, A, B, and C, and displays one of the three messages: "Two of the values are the same," "All three values are the same," or "All of the values are different."

2.8 Ada has two different operators for *and* (**and** and **and then**) and two different operators for *or* (**or** and **or else**). The latter operator in each case is called the *short circuit* form. Read section 4.5.1 of the ARM and then explain the difference between the normal and short circuit forms of these logical operators.

2.9 Which of the four if statements in the examples given on page 22 could be translated into if expressions?

2.10 Why cannot the case statement example on page 23 be translated into a case expression?

2.11 Write a loop that displays all of the integers between 0 and 100 that are evenly divisible by 3.

2.12 True or False? A for loop with a null range executes exactly one iteration.

2.13 Write a procedure that swaps the contents of its two integer parameters.

2.14 Write a function that returns the larger of its two real parameters.

2.15 What two characteristics does a data type define?

2.16 Suppose we have a type for complex numbers. Why is this type not an atomic type?

2.17 Why is 5.3 not the unique successor of 5.2?

2.18 Errors in quantities can be expressed in absolute terms or relative terms. For each of the following quantities, determine which type of error (absolute or relative) makes more sense, then declare the most appropriate Ada type for that quantity:
   a. Distances in light years to various galaxies in the universe
   b. Altitude in feet of an aircraft
   c. Number of gallons of gasoline in a car's fuel tank
   d. Bank account balance

2.19 Write a Boolean function called Nearly_Equal that returns True if two float numbers are nearly equal. Two numbers are considered nearly equal if the absolute value of their difference is less than 0.001 percent of the smaller number.

2.20 Suppose we have a type for a choice in the game rock-paper-scissors with a domain consisting of the three possible player choices.
   a. Why is this type an atomic type?
   b. Is this type a scalar type? Explain your answer.

2.21 What, if any, are the restrictions on the types we may use for the *components* of an array type? What, if any, are the restrictions on the types we may use for the *index* of an array type?

2.22 Declare an array type whose components are positive whole numbers indexed by Pixel_Color (defined on page 40).

2.23 Given the following unconstrained array type

```
type Int_Array is array (Character range <>) of Integer;
```

complete the following function that returns the value of the largest value in the array:

```
function Max (Items : in Int_Array) return Integer is
```

2.24 Given the following array type that defines an unconstrained array of real numbers:

```
type Float_Array is array ( Positive range <>) of Float;
```

2.25 Why is the following variable declaration illegal? Write a declaration for a string variable suitable for holding a person's name.

```
Name : String;
```

Write an instantiation of the generic procedure Tally given on page 63 that tallies up the number of floating point numbers in an array that are nearly equal to a given number. Make use of the function Nearly_Equal you wrote for Exercise 2.19.

2.26 Given the following array types that define an unconstrained array of Cartesian points (defined on page 58) indexed by Natural integers, an array variable that holds 100 points, and a variable that holds a count of points:

```
type    Point_Array is array (Natural range <>) of Point;
subtype Point_List  is Point_Array (101 .. 200);

My_Points   : Point_List ;
Point_Tally : Natural;
```

a. Write an instantiation of the generic procedure Tally given on page 63 that tallies up the number of points in an array that are not further from the origin than a given point. You may use the function "<=" defined on page 58.

b. Write a call to the procedure you instantiated to tally the number of points in the array My_Points that are on or inside a circle of radius 1.0. Put the result into the variable Point_Tally. You may either declare a variable to hold a pattern point or use a record aggregate in your procedure call.