Answers to Exercises

Specimen answers are given to all the exercises. In some cases they do not necessarily represent the best technique for solving a problem but merely one which uses the material introduced at that point in the discussion.

Answers 2

Exercise 2.2

1
```
package Simple_Maths is
   function Sqrt(F: Float) return Float;
   function Log(F: Float) return Float;
   function Ln(F: Float) return Float;
   function Exp(F: Float) return Float;
   function Sin(F: Float) return Float;
   function Cos(F: Float) return Float;
end Simple_Maths;
```

The first few lines of the program Print_Roots could now become

```
with Simple_Maths, Simple_IO;
procedure Print_Roots is
   use Simple_Maths, Simple_IO;
```

Exercise 2.4

1
```
for I in 0 .. N loop
   Pascal(I) := Next(I);
end loop;
```

2
```
for N in 0 .. 10 loop
   Pascal2(N, 0) := 1;
   for I in 1 .. N–1 loop
      Pascal2(N, I) := Pascal2(N–1, I–1) +
                                    Pascal2(N–1, I);
   end loop;
   Pascal2(N, N) := 1;
end loop;
```

3
```
type Month_Name is (Jan, Feb, Mar, Apr, May,
                    Jun, Jul, Aug, Sep, Oct, Nov, Dec);

type Date is
   record
      Day: Integer;
      Month: Month_Name;
      Year: Integer;
   end record;

Today: Date;
...
Today := (24, May, 1819);
```

Answers 3

Exercise 3.1

1
```
package Buffer_System is              –– visible part
   type Buffer is private;
   Buffer_Error: exception;
   procedure Load(B: in out Buffer; S: in String);
   procedure Get(B: in out Buffer;
                 C: out Character);
   function Is_Empty(B: Buffer) return Boolean;
private                               –– private part
   Max: constant Integer := 80;
   type Buffer is
      record
         Data: String(1 .. Max);
         Start: Integer := 1;
         Finish: Integer := 0;
      end record;
end Buffer_System;

package body Buffer_System is
   procedure Load(B: in out Buffer; S: in String) is
   begin
      if S'Length > Max or B.Start <= B.Finish then
```

```
      raise Buffer_Error;
    end if;
    B.Start := 1;
    B.Finish := S'Length;
    B.Data(B.Start .. B.Finish) := S;
  end Load;

  procedure Get(B: in out Buffer;
                C: out Character) is
  begin
    if B.Start > B.Finish then
      raise Buffer_Error;
    end if;
    C := B.Data(B.Start);
    B.Start := B.Start + 1;
  end Get;

  function Is_Empty(B: Buffer) return Boolean is
  begin
    return B.Start > B.Finish;
  end Is_Empty;
end Buffer_System;
```

The parameter Buffer of Load now has to be **in out** because the original value is read. Also, we could replace the test in Get by

```
    if Is_Empty(B) then
```

Exercise 3.2

1

```
package Objects is
  type Object is tagged
    record
      X_Coord: Float;
      Y_Coord: Float;
    end record;

  function Distance(O: Object) return Float;
  function Area(O: Object) return Float;
end Objects;

package body Objects is
  function Distance(O: Object) return Float is
  begin
    return Sqrt(O.X_Coord**2 + O.Y_Coord**2);
  end Distance;

  function Area(O: Object) return Float is
  begin
    return 0.0;
  end Area;
end Objects;

with Objects;  use Objects;
package Shapes is
  type Circle is new Object with
    record
      Radius: Float;
    end record;

  function Area(C: Circle) return Float;

  type Point is new Object with null record;

  type Triangle is new Object with
    record
      A, B, C: Float;
    end record;

  function Area(T: Triangle) return Float;
end Shapes;

package body Shapes is
  function Area(C: Circle) return Float is
  begin
    return π * C.Radius**2;
  end Area;

  function Area(T: Triangle) return Float is
    S: constant Float := 0.5 * (T.A + T.B + T.C);
  begin
    return Sqrt(S * (S – T.A) * (S – T.B) * (S – T.C));
  end Area;
end Shapes;
```

Note that we can put the use clause for Objects immediately after the with clause.

Exercise 3.3

1

```
procedure Add_To_List(The_List: in out List;
                      Obj_Ptr: in Pointer) is
  Local: List := new Cell;
begin
  Local.Next := The_List;
  Local.Element := Obj_Ptr;
  The_List := Local;
end Add_To_List;
```

or more briefly using a form of allocation with initial values

```
procedure Add_To_List(The_List: in out List;
                      Obj_Ptr: in Pointer) is
begin
  The_List := new Cell'(The_List, Obj_Ptr);
end Add_To_List;
```

2

```
package body Objects is
  function Distance(O: Object) return Float is
  begin
    return Sqrt(O.X_Coord**2 + O.Y_Coord**2);
  end Distance;
end Objects;
```

3 We have to add the function Area for the type Point.

4 We cannot declare the function Moment for the abstract type Object because it contains a call of the abstract function Area.

5
```
function MO(OC: Object'Class) return Float is
begin
   return MI(OC) + Area(OC) * Distance(OC)**2;
end MO;
```

Answers 4

Exercise 4.2

1 The default field is 6 for a 16-bit type Integer and 11 for a 32-bit type Integer so

```
Put(123);       -- "sss123"  and  "ssssssss123"
Put(-123);      -- "ss-123"  and  "sssssss-123"
```

Exercise 4.4

1
```
with Ada.Text_IO, Etc;
use Ada.Text_IO, Etc;
procedure Table_Of_Square_Roots is
   use My_Float_IO, My_Elementary_Functions;
   Last_N: Integer;
   Tab: Count;
begin
   Tab := 10;
   Put("What is the largest value please? ");
   Get(Last_N);
   New_Line(2);
   Put("Number");  Set_Col(Tab);
   Put("Square root");
   New_Line(2);
   for N in 1 .. Last_N loop
      Put(N, 4);  Set_Col(Tab);
      Put(Sqrt(My_Float(N)), 3, 6, 0);
      New_Line;
   end loop;
end Table_Of_Square_Roots;
```

2
```
with Ada.Text_IO;
package My_Numerics.My_Float_IO is
      new Ada.Text_IO.Float_IO(My_Float);

with Ada.Text_IO;
package My_Numerics.My_Integer_IO is
      new Ada.Text_IO.Integer_IO(My_Integer);

with Ada.Numerics.Generic_Elementary_Functions;
package My_Numerics.My_Elementary_Functions is
      new Ada.Numerics.
           Generic_Elementary_Functions(My_Float);
```

3
```
package Objects is ...

with Ada.Numerics.Elementary_Functions;
use Ada.Numerics.Elementary_Functions;
package body Objects is ...

with Objects;  use Objects;
package Shapes is ...

with Ada.Numerics.Elementary_Functions;
use Ada.Numerics.Elementary_Functions;
package body Shapes is ...

with Shapes;  use Shapes;
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;
procedure Area_Of_Triangle is
   T: Triangle;
begin
   Get(T.A);  Get(T.B);  Get(T.C);
   Put(Area(T));
end Area_Of_Triangle;
```

We should really check that the sides do form a triangle, if they do not then the call of Sqrt in Area will have a negative parameter and so raise Ada.Numerics.Argument_Error. See Program 1.

4
```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
with Ada.Numerics.Discrete_Random;
procedure Sundays is
   type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   package Random_Day is
         new Ada.Numerics.Discrete_Random(Day);
   use Random_Day;
   G: Generator;
   D: Day;
   Number_Of_Sundays: Integer;
begin
   Number_Of_Sundays := 0;
   for I in 1 .. 100 loop
      D := Random(G);
      if D = Sun then
         Number_Of_Sundays :=
                              Number_Of_Sundays + 1;
      end if;
   end loop;
   Put("Percentage of Sundays in selection was ");
   Put(Number_Of_Sundays);
   New_Line;
end Sundays;
```

5
```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Triangle is
   Size: Integer;
begin
   Put("Size of triangle please: ");  Get(Size);
   declare
      Pascal: array (0 .. Size) of Integer;
      Tab: Count;       -- indentation at start of row
   begin
      Tab := Count(2*Size + 1);
      Pascal(0) := 1:
      for N in 1 .. Size loop
```

```
         Pascal(N) := 1;
         for I in reverse 1 .. N–1 loop
            Pascal(I) := Pascal(I–1) + Pascal(I);
         end loop;
         Tab := Tab – 2;
         New_Line(2);  Set_Col(Tab);
         for I in 0 .. N loop
            Put(Pascal(I), 4);
         end loop;
      end loop;
      New_Line(2);
      if 2*Size > 8 then
         Set_Col(Count(2*Size – 8));
      end if;
      Put("The Triangle of Pascal");
      New_Line(2);
   end;
end Triangle;
```

It is instructive to consider how this should be written to accommodate larger values of Size in a flexible manner and so avoid the confusing repetition of the literal 2. A variable Half_Field might be declared with the value 2 in the above but would need to be 3 for values of Size up to 19 which will go off the screen anyway. Care is needed with variables of type Count which are not allowed to take negative values.

Answers 5

Exercise 5.3

1 The following are not legal identifiers

(b) contains &
(c) contains hyphens not underlines
(e) adjacent underlines
(f) does not start with a letter
(g) trailing underline
(h) this is two legal identifiers
(i) this is legal – but it is a reserved word and not an identifier

Note that (a) is of course a legal identifier but it would be unwise to declare our own variable called Ada because it would conflict with the predefined package of that name.

Exercise 5.4

1 (a) legal – real
(b) illegal – no digit before point
(c) legal – integer
(d) illegal – integer with negative exponent
(e) illegal – closing # missing
(f) legal – real
(g) illegal – C not a digit of base 12
(h) illegal – no number before exponent
(i) legal – integer – case of letter immaterial
(j) legal – integer
(k) illegal – underline at start of exponent
(l) illegal – integer with negative exponent

2 (a) $224 = 14 \times 16$
(b) $6144 = 3 \times 2^{11}$
(c) 4095.0
(d) 4095.0

3 (a) 32 ways

41, 2#101001#, 3#1112#, ... 10#41#, ... 16#29#, 41E0, 2#101001#E0, ... 16#29#E0

(b) 40 ways. As for example (a) plus, since 150 is not prime but $2 \times 3 \times 5^2 = 150$, also

2#1001011#E1, 3#1212#E1, 5#110#E1, 5#11#E2, 6#41#E1, 10#15#E1, 15#A#E1, 15E1

Answers 6

Exercise 6.1

1 F: Float := 1.0;

2 Zero: **constant** Float := 0.0;
One: **constant** Float := 1.0;

but it might be better to write real number declarations

Zero: **constant** := 0.0;
One: **constant** := 1.0;

3 (a) **var** is illegal – this is Ada not Pascal
(b) terminating semicolon is missing
(c) a constant declaration must have an initial value
(d) no multiple assignment in Ada
(e) nothing – assuming M and N are of integer type
(f) 2Pi is not a legal identifier

Exercise 6.2

1 There are four errors

(1) no semicolon after declaration of J, K
(2) K used before a value is assigned to it
(3) = instead of := in declaration of P
(4) Q not declared and initialized

Exercise 6.4

1 It is assumed that the values of all variables originally satisfy their constraints.

(a) the ranges of I and J are identical so no checks are required and consequently Constraint_Error cannot be raised,

(b) the range of J is a subset of that of K and again Constraint_Error cannot be raised,

(c) in this case a check is required since if K > 10 it cannot be assigned to J in which case Constraint_Error will be raised.

Exercise 6.5

1 (a) –105 (b) –3 (c) 0 (d) –3 (e) –3 (f) illegal (g) –1 (h) 2

2 All variables are of type Float

```
(a) M*R**2
(b) B**2 – 4.0*A*C
(c) (4.0/3.0)*π*R**3
                      -- parentheses not necessary
(d) (P*π*A**4) / (8.0*L*η)
                      -- parentheses are necessary
```

Exercise 6.6

1 (a) Sat
(b) Sat   note that Succ applies to base type
(c) 2

2

```
(a) type Rainbow is (Red, Orange, Yellow, Green,
                     Blue, Indigo, Violet);
(b) type Fruit is (Apple, Banana, Orange, Pear);
```

3 Groom'Val((N–1) **mod** 8)

or perhaps better

Groom'Val((N–1) **mod** (Groom'Pos(Groom'Last) +1))

4 D := Day'Val((Day'Pos(D) + N – 1) **mod** 7);

5 If X and Y are both overloaded literals then X < Y will be ambiguous. We would have to use qualification such as T'(X) < T'(Y).

Exercise 6.7

1

```
T: constant Boolean := True;
F: constant Boolean := False;
```

2 The values are True and False, not T or F which are the names of constants.

(a) False (b) True (c) True (d) True (e) False

3 The expression is always True. The predefined operators **xor** and **/=** operating on Boolean values are the same. But see the note at the end of Section 11.3.

Exercise 6.8

1 (a) False (b) Sat

Exercise 6.9

1 All variables are of type Float except for N in example (c) which is Integer.

```
(a) 2.0*π*Sqrt(L/G)
(b) M_0/Sqrt(1.0–(V/C)**2)
(c) Sqrt(2.0*π*Float(N)) * (Float(N)/E)**N
```

2 Sqrt(2.0*π*X) * Exp(X*Ln(X)–X)

Answers 7

Exercise 7.1

1

```
declare
   End_Of_Month: Integer;
begin
   if Month = Sep or Month = Apr
             or Month = Jun or Month = Nov then
      End_Of_Month := 30;
   elsif Month = Feb then
      if (Year mod 4 = 0 and Year mod 100 /= 0)
                      or Year mod 400 = 0 then
         End_Of_Month := 29;
      else
         End_Of_Month := 28;
      end if;
   else
      End_Of_Month := 31;
   end if;
   if Day /= End_Of_Month then
      Day := Day + 1;
   else
      Day := 1;
      if Month /= Dec then
         Month := Month_Name'Succ(Month);
      else
         Month := Jan;
         Year := Year + 1;
      end if;
```

```
   end if;
end;
```

If today is 31 Dec 2399 then Constraint_Error will be raised on attempting to assign 2400 to Year.

2
```
if X < Y then
   declare
      T: Float := X;
   begin
      X := Y;  Y := T;
   end;
end if;
```

Exercise 7.2

1
```
declare
   End_Of_Month: Integer;
begin
   case Month is
      when Sep | Apr | Jun | Nov =>
         End_Of_Month := 30;
      when Feb =>
         if (Year mod 4 = 0 and Year mod 100 /= 0)
                              or Year mod 400 = 0 then
            End_Of_Month := 29;
         else
            End_Of_Month := 28;
         end if;
      when others =>
         End_Of_Month := 31;
   end case;
   -- then as before
   ...
end;
```

2
```
subtype Winter is Month_Name range Jan .. Mar;
subtype Spring is Month_Name range Apr .. Jun;
subtype Summer is Month_Name range Jul .. Sep;
subtype Autumn is Month_Name range Oct .. Dec;
...
case M is
   when Winter => Dig;
   when Spring => Sow;
   when Summer => Tend;
   when Autumn => Harvest;
end case;
```

Note that if we wished to consider winter as December to February then we could not declare a suitable subtype.

3
```
case D is
   when 1 .. 10 => Gorge;
   when 11 .. 20 => Subsist;
   when others => Starve;
end case;
```

We cannot write 21 .. End_Of_Month because it is not a static range. In fact **others** covers all values of type Integer because although D is constrained, nevertheless the constraints are not static.

Exercise 7.3

1
```
declare
   Sum: Integer := 0;
   I: Integer;
begin
   loop
      Get(I);
      exit when I < 0;
      Sum := Sum + I;
   end loop;
end;
```

2
```
declare
   Copy: Integer := N;
   Count: Integer := 0;
begin
   while Copy mod 2 = 0 loop
      Copy := Copy / 2;
      Count := Count + 1;
   end loop;
   ...
end;
```

3
```
declare
   G: Float := -Ln(Float(N));
begin
   for P in 1 .. N loop
      G := G + 1.0/Float(P);
   end loop;
   ...
end;
```

We assume that Ln is the function for natural logarithm.

Answers 8

Exercise 8.1

1
```
declare
   F: array (0 .. N) of Integer;
begin
   F(0) := 0;  F(1) := 1;
   for I in 2 .. F'Last loop
      F(I) := F(I-1) + F(I-2);
   end loop;
   ...
end;
```

2
```
declare
   Max_I: Integer := A'First(1);
   Max_J: Integer := A'First(2);
   Max: Float := A(Max_I, Max_J);
begin
   for I in A'Range(1) loop
      for J in A'Range(2) loop
         if A(I, J) > Max then
            Max := A(I, J);
            Max_I := I;  Max_J := J;
         end if;
      end loop;
   end loop;
               -- Max_I, Max_J now contain the result
end;
```

3
```
declare
   Days_In_Month: array (Month_Name) of Integer
     := (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
   End_Of_Month: Integer;
begin
   if (Year mod 4 = 0 and Year mod 100 /= 0)
                          or Year mod 400 = 0 then
      Days_In_Month(Feb) := 29;
   end if;
   End_Of_Month := Days_In_Month(Month);
   ...      -- then as Exercise 7.1(1)
end;
```

4
```
Yesterday: constant array (Day) of Day :=
               (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

5
```
Bor: constant array (Boolean, Boolean) of Boolean
               := ((False, True), (True, True));
```

6
```
Unit: constant array (1 .. 3, 1 .. 3) of Float :=
              ((1.0, 0.0, 0.0),
               (0.0, 1.0, 0.0),
               (0.0, 0.0, 1.0));
```

Exercise 8.2

1
```
type Bbb is array (Boolean, Boolean) of Boolean;
```

2
```
type Ring5_Table is array (Ring5, Ring5) of Ring5;

Add: constant Ring5_Table :=
              ((0, 1, 2, 3, 4),
               (1, 2, 3, 4, 0),
               (2, 3, 4, 0, 1),
               (3, 4, 0, 1, 2),
               (4, 0, 1, 2, 3));

Mult: constant Ring5_Table :=
              ((0, 0, 0, 0, 0),
               (0, 1, 2, 3, 4),
               (0, 2, 4, 1, 3),
               (0, 3, 1, 4, 2),
               (0, 4, 3, 2, 1));

A, B, C, D: Ring5;
...
D := Mult(Add(A, B), C));
```

Exercise 8.3

1
```
Days_In_Month: array (Month_Name) of Integer :=
              (Sep | Apr | Jun | Nov => 30,
               Feb => 28,
               others => 31);
```

2
```
Zero: constant Matrix := (1 .. N => (1 .. N => 0.0));
```

3 This cannot be done with the material at our disposal at the moment. See Exercise 9.1(**6**).

4
```
type Molecule is (Methanol, Ethanol, Propanol,
                                     Butanol);
type Atom is (H, C, O);

Alcohol:
      constant array (Molecule, Atom) of Integer :=
         (Methanol => (H =>  4, C => 1, O => 1),
           Ethanol => (      6,      2,      1),
          Propanol => (      8,      3,      1),
           Butanol => (     10,      4,      1));
```

Note the danger in the above. We have used named notation in the first inner aggregate to act as a sort of heading but omitted it in the others to avoid clutter. However, if we had written H, C and O in other than positional order then it would have been very confusing because the positional aggregates would not have had the meaning suggested by the heading.

Exercise 8.4

1
```
Roman_To_Integer:
         constant array (Roman_Digit) of Integer :=
            (1, 5, 10, 50, 100, 500, 1000);
```

2
```
declare
   V: Integer := 0;
begin
   for I in R'Range loop
      if I /= R'Last and then
         Roman_To_Integer(R(I)) <
                     Roman_To_Integer(R(I+1)) then
         V := V – Roman_To_Integer(R(I));
      else
```

```
      V := V + Roman_To_Integer(R(I));
    end if;
  end loop;
  ...
end;
```

Note the use of **and then** to avoid attempting to access R(I+1) when I = R'Last.

Exercise 8.5

1 AOA(1 .. 2) := (AOA(2), AOA(1));

2
```
Farmyard: String_3_Array := ("pig", "cat", "dog",
                             "cow", "rat", "hen");
...
Farmyard(4)(1) := 's';
```

3
```
if R'Last >= 2 and then
              R(R'Last–1 .. R'Last) = "IV" then
  R(R'Last–1 .. R'Last) := "VI";
end if;
```

Exercise 8.6

1 White, Blue, Yellow, Green, Red, Purple, Orange, Black

2 (a) Black (c) Red
(b) Green

3 **not** (True **xor** True) = True
not (True **xor** False) = False

The result follows.

4 An aggregate of length one must be named.

5 "123", "ABC", "Abc", "aBc", "abC", "abc"

6 (a) 1 (c) 5
(b) 5

Note that the lower bound of the result of & may depend upon the order of the operands; the same applies to **and**, **or** and **xor**.

7 (a) 1 .. 10 (c) 6 .. 15
(b) 1 .. 10 (d) 0 .. 9

Exercise 8.7

1 C1, C2, C3: Complex;

(a) C3 := (C1.Rl+C2.Rl, C1.Im+C2.Im);
(b) C3 := (C1.Rl*C2.Rl – C1.Im*C2.Im,
C1.Rl*C2.Im + C1.Im*C2.Rl);

2
```
declare
  Index: Integer;
begin
  for S in People'Range loop
    if People(S).Birth.Year >= 1980 then
      Index := S;
      exit;
    end if;
  end loop;
  -- we assume that there is such a student
end;
```

Exercise 8.8

1 In a straightforward manner 8 as follows

((1, 0), (0, 1)), ((1, 0), [0, 1]), ([1, 0], (0, 1)),
[(1, 0), (0, 1)], [[1, 0], (0, 1)], [(1, 0), [0, 1]],
([1, 0], [0, 1]), [[1, 0], [0, 1]]

2 (**for** K **in** 1 .. 10 => K*(K+1)/2)

Answers 9

Exercise 9.1

1 B := N **in** 3 | 5 | 7 | 11 | 13 | 17 | 19;

2 Letter **in** 'a' .. 'e' | 'A' .. 'E' | 'v' .. 'z' | 'V' .. 'Z'

Exercise 9.2

1
```
Days_In_Month := (if M in Sep | Apr | Jun | Nov
then 30 elsif M = Feb then (if Year mod 4 = 0
then 29 else 28) else 31);
```

Exercise 9.3

1
```
L := (case Today is
        when Monday | Friday | Sunday => 6
        when Tuesday => 7
        when Thursday | Saturday => 8
        when Wednesday => 9);
```

2
```
Pension := Integer(
             (if Age in 50 .. 69 then 50.0
              elsif Age in 70 .. 79 then 60.0
              elsif Age in 80.. 100 then 70.0
              else 0.0)
               *
              (if Gender = Female then 0.9 else 1.0)
              *
              (if Disabled then 1.05 else 1.0)
              +
               (if Age = 100 then 100.0 else 0.0));
```

It is probably better to use a case expression for the first part thus

```
(case Age is
   when 50 .. 69 => 50.0
   when 70 .. 79 => 60.0
   when 80 .. 100 => 70.0
   when others => 0.0)
```

assuming that Age is a static subtype of Integer.

Exercise 9.4

1

```
(for all K in A'First .. A'Last – 1 =>
                                    A(K) <= A(K+1))
```

This assumes that the index type of the array is an integer type. In the general case we have to use T'Pred and T'Succ where T is the type of the index thus

```
(for all K in A'First .. T'Pred(A'Last) =>
                              A(K) <= A(T'Succ(K)))
```

Exercise 9.6

1

```
Tn := ([for J in 1..N => J*(J+1)/2]'Reduce("+", 0));
```

Answers 10

Exercise 10.1

1

```
function Even(X: Integer) return Boolean is
begin
   return X mod 2 = 0;
end Even;
```

2

```
function Factorial(N: Natural) return Positive is
begin
   if N = 0 then
      return 1;
   else
      return N * Factorial(N–1);
   end if;
end Factorial;
```

3

```
function Outer(A, B: Vector) return Matrix is
   C: Matrix(A'Range, B'Range);
begin
   for I in A'Range loop
      for J in B'Range loop
         C(I, J) := A(I) * B(J);
      end loop;
   end loop;
   return C;
end Outer;
```

4

```
type Primary_Array is
      array (Integer range <>) of Primary;

function Make_Colour(P: Primary_Array)
                                    return Colour is
   C: Colour := (F, F, F);
begin
   for I in P'Range loop
      C(P(I)) := T;
   end loop;
   return C;
end Make_Colour;
```

Note that multiple values are allowed so that Make_Colour((R, R, R)) = Red.

5

```
function Inner(A, B: Vector) return Float is
begin
   if A'Length /= B'Length then
      raise Constraint_Error;
   end if;
   return Result: Float := 0.0 do
      for I in A'Range loop
         Result := Result + A(I) * B(I+B'First–A'First);
      end loop;
   end return;
end Inner;
```

6

```
function Make_Unit(N: Natural) return Matrix is
begin
   return M: Matrix(1 .. N, 1 .. N) do
      for I in 1 .. N loop
         for J in 1 .. N loop
            if I = J then
               M(I, J) := 1.0;
            else
               M(I, J) := 0.0;
            end if;
         end loop;
      end loop;
   end return;
end Make_Unit;
```

We can then declare

```
Unit: constant Matrix := Make_Unit(N);
```

7

```
function GCD(X, Y: Natural) return Natural is
begin
   if Y = 0 then
      return X;
   else
      return GCD(Y, X mod Y);
   end if;
end GCD;
```

or

```
function GCD(X, Y: Natural) return Natural is
   XX: Integer := X;
   YY: Integer := Y;
   ZZ: Integer;
begin
   while YY /= 0 loop
      ZZ := XX mod YY;  XX := YY;  YY := ZZ;
   end loop;
   return XX;
end GCD;
```

Note that X and Y have to be copied because the formal parameters behave as constants.

Exercise 10.2

1
```
function "<" (X, Y: Roman_Number) return Boolean is

   function Value(R: Roman_Number)
                                          return Integer is
      V: Integer := 0;
   begin
      ...   -- then loop as in Exercise 8.4(2)
      return V;
   end Value;

begin
   return Value(X) < Value(Y);
end "<";
```

2
```
function "+" (X, Y: Complex) return Complex is
begin
   return (X.Rl + Y.Rl, X.Im + Y.Im);
end "+";

function "*" (X, Y: Complex) return Complex is
begin
   return (X.Rl*Y.Rl – X.Im*Y.Im,
           X.Rl*Y.Im + X.Im*Y.Rl);
end "*";
```

3
```
function "<" (P: Primary; C: Colour) return Boolean is
begin
   return C(P);
end "<";
```

4
```
function "<=" (X, Y: Colour) return Boolean is
begin
   return (X and Y) = X;
end "<=";
```

5
```
function "<" (X, Y: Date) return Boolean is
begin
   if X.Year /= Y.Year then
      return X.Year < Y.Year;
   elsif X.Month /= Y.Month then
      return X.Month < Y.Month;
   else
      return X.Day < Y.Day;
   end if;
end "<";
```

Exercise 10.3

1
```
procedure Swap(X, Y: in out Float) is
   T: Float;
begin
   T := X;  X := Y;  Y := T;
end Swap;
```

2
```
procedure Rev(A: in out Vector) is
   R: Vector(A'Range);
begin
   for I in A'Range loop
      R(I) := A(A'First + A'Last – I);
   end loop;
   A := R;
end Rev;
```

We might then write

```
Rev(Vector(R));
```

If we had two parameters and built the result directly in the out parameter thus

```
procedure Rev(A: in Vector; R: out Vector) is
begin
   for I in A'Range loop
      R(I) := A(A'First + A'Last – I);
   end loop;
end Rev;
```

then a call with both parameters denoting the same array would result in a mess if passed by reference because the result would overwrite the data. Both parameters denote the same object and are said to be aliased. This is a bounded error.

3 The fragment has a bounded error because the outcome depends upon whether the parameter is passed by copy or by reference. If by copy then A(1) ends up as 2.0; if by reference then A(1) ends up as 4.0. There is aliasing because A and V both refer to the same object.

Exercise 10.4

1
```
function Add(X: Integer; Y: Integer := 1)
                                    return Integer is
begin
   return X + Y;
end Add;
```

The following 6 calls are equivalent

```
Add(N)                     Add(N, 1)
Add(X => N, Y => 1)        Add(X => N)
Add(N, Y => 1)             Add(Y => 1, X => N)
```

2
```
function Favourite_Spirit return Spirit is
begin
   case Today is
      when Mon .. Fri => return Gin;
      when Sat | Sun => return Vodka;
   end case;
end Favourite_Spirit;

...

procedure Dry_Martini(Base: Spirit := Favourite_Spirit;
                      How: Style := On_The_Rocks;
                      Plus: Trimming := Olive);
```

This example illustrates that defaults are evaluated each time they are required and can therefore be changed from time to time. Incidentally, we could just declare the specification of Favourite_Spirit first and then declare the bodies of both subprograms.

Answers 11

Exercise 11.2

1
```
procedure Append(First: in out Cell_Ptr;
                 Second: in Cell_Ptr) is
   L: Cell_Ptr := First;
begin
   if First = null then
      First := Second;
   else
      while L.Next /= null loop
         L := L.Next;
      end loop;
      L.Next := Second;
   end if;
end Append;
```

2
```
function Size(T: Node_Ptr) return Integer is
begin
   if T = null then
      return 0;
   else
      return Size(T.Left) + Size(T.Right) + 1;
   end if;
end Size;
```

3
```
function Copy(T: Node_Ptr) return Node_Ptr is
begin
   if T = null then
      return null;
   else
      return new Node'(Copy(T.Left),
                       Copy(T.Right), T.Value);
   end if;
end Copy;
```

4
```
function "+" (A: A_String) return String is
begin
   return A.all;
end "+";
```

and then Put(+Zoo(3)); will output the string "camel".

5
```
function "&" (X, Y: A_String) return A_String is
begin
   return new String'(X.all & Y.all);
end "&";
```

Exercise 11.4

1
```
type G_String is access constant String;
type G_String_Array is
         array (Positive range <>) of G_String;

Aardvark: aliased constant String := "aardvark";
Baboon: aliased constant String := "baboon";
...
Zebra: aliased constant String := "zebra";

Zoo: constant G_String_Array :=
      (Aardvark'Access, Baboon'Access, ...,
                                   Zebra'Access);
```

2
```
N: Integer := ... ;
M: Integer := ... ;
World: array (1 .. N, 1 .. M) of Cell;
Abyss: constant Cell := (0, 0, (1 .. 8 => null));
...
-- offsets of 8 neighbours starting at North
type Offset is array (1 .. 8) of Integer;
H_Off: Offset := (+0, +1, +1, +1, +0, –1, –1, –1);
V_Off: Offset := (+1, +1, +0, –1, –1, –1, +0, +1);

-- now link up the cells
for I in 1 .. N loop
   for J in 1 .. M loop
      -- link to eight neighbours except on boundary
      declare
         H_Index, V_Index: Integer;
      begin
         for N_Index in 1 .. 8 loop
            H_Index := I + H_Off(N_Index);
            V_Index := J + V_Off(N_Index);
            if H_Index in 1 .. N and
                     V_Index in 1 .. M then
               World(I, J).Neighbour_Count(N_Index) :=
                     World(H_Index, V_Index).
                              Life_Count'Access;
            else
               -- edge of world, link to abyss
               World(I, J).Neighbour_Count(N_Index) :=
                     Abyss.Life_Count'Access;
            end if;
         end loop;
```

```
    end;
  end loop;
end loop;
```

Clearly the repetition of World(I, J) could be eliminated by introducing an access type to the cell itself. Or we could use renaming as described in Section 13.7. It would be better if we did not have so many occurrences of the literal 8. Indeed, the enthusiastic reader might like to consider how this example might be extended to three or more dimensions. In three dimensions of course the number of neighbours is $3^3 - 1 = 26$.

3
```
type Cell;
type Ref_Cell is access constant Cell;
type Ref_Cell_Array is
      array (Integer range <>) of Ref_Cell;
type Cell is
  record
    Life_Count: Integer range 0 .. 1;
    Total_Neighbour_Count: Integer range 0 .. 8;
    Neighbour: Ref_Cell_Array(1 .. 8);
  end record;
...
C.Total_Neighbour_Count := 0;
for I in C.Neighbour'Range loop
  C.Total_Neighbour_Count :=
            C.Total_Neighbour_Count +
                C.Neighbour(I).Life_Count;
end loop;
```

The other changes are that World and Abyss have to be declared as aliased

```
World: array (1 .. N, 1 .. M) of aliased Cell;
Abyss: aliased constant Cell := (0, 0, (1 .. 8 => null));
```

and the expressions assigned to the neighbours omit Life_Count as in

```
World(I, J).Neighbour(N_Index) := Abyss'Access;
```

Exercise 11.6

1 The conversion to Ref1 is checked dynamically; it passes for the call of P with X1'Access and fails with X2'Access. The conversion to Ref2 is checked statically and passes.

2 The conversion to Ref1 is checked dynamically; it passes for X1 and fails for X2 and X3. The conversion to Ref2 is checked dynamically and passes in all cases. The conversion to Ref3 is checked statically and passes.

Note that the case of X3 and Ref2 is where the accessibility is adjusted on the chained call; without this adjustment it would unnecessarily fail. The point is that considering P1 as a whole, since the type A2 is inside, the conversion is always safe. But since the type A2 is outside P2 which actually does the conversion, it has to be checked dynamically; the adjustment ensures that it always passes.

Exercise 11.7

1 The first and last assignments are legal.

Exercise 11.8

1
```
function G(T: Float) return Float is
begin
  return Exp(T) * Sin(T);
end G;
...
Answer: Float := Integrate(G'Access, 0.0, P);
```

2
```
function Solve(F: access function (X: Float)
                         return Float) return Float;
...
function G(X: Float) return Float is
begin
  return Exp(X) + X - 7.0;
end G;
...
Answer := Solve(G'Access);
```

3
```
function Integrate(F: access function (X, Y: Float)
                                    return Float);
             LX, HX, LY, HY: Float return Float is
  function Outer(X: Float) return Float is
    function Inner(Y: Float) return Float is
    begin
      return F(X, Y);
    end Inner;
  begin
    return Integrate(Inner'Access, LY, HY);
  end Outer;
begin
  return Integrate(Outer'Access, LX, HX);
end Integrate;
```

The functions have to be nested so that the inner one can access the parameter X of the outer one.

Answers 12

Exercise 12.1

1
```
package Random is
   Modulus: constant := 2**13;
   subtype Small is Integer range 0 .. Modulus;
   procedure Init(Seed: in Small);
   function Next return Small;
end;

package body Random is
   Multiplier: constant := 5**5;
   X: Small;

   procedure Init(Seed: in Small) is
   begin
      X := Seed;
   end Init;

   function Next return Small is
   begin
      X := X * Multiplier mod Modulus;
      return X;
   end Next;

end Random;
```

2
```
package Complex_Numbers is
   type Complex is
      record
         Re, Im: Float := 0.0;
      end record;

   I: constant Complex := (0.0, 1.0);

   function "+" (X: Complex) return Complex;
   function "–" (X: Complex) return Complex;

   function "+" (X, Y: Complex) return Complex;
   function "–" (X, Y: Complex) return Complex;
   function "*" (X, Y: Complex) return Complex;
   function "/" (X, Y: Complex) return Complex;
end;

package body Complex_Numbers is

   function "+" (X: Complex) return Complex is
   begin
      return X;
   end "+";

   function "–" (X: Complex) return Complex is
   begin
      return (–X.Re, –X.Im);
   end "–";

   function "+" (X, Y: Complex) return Complex is
   begin
      return (X.Re + Y.Re, X.Im + Y.Im);
   end "+";

   function "–" (X, Y: Complex) return Complex is
   begin
      return (X.Re – Y.Re, X.Im – Y.Im);
   end "–";

   function "*" (X, Y: Complex) return Complex is
   begin
      return (X.Re*Y.Re – X.Im*Y.Im,
                 X.Re*Y.Im + X.Im*Y.Re);
   end "*";

   function "/" (X, Y: Complex) return Complex is
      D: Float := Y.Re**2 + Y.Im**2;
   begin
      return ((X.Re*Y.Re + X.Im*Y.Im)/D,
                 (X.Im*Y.Re – X.Re*Y.Im)/D);
   end "/";

end Complex_Numbers;
```

Exercise 12.2

1 Inside the package body (or that of a child package, see Section 13.3) we could write

```
function "*" (X: Float; Y: Complex) return Complex is
begin
   return (X*Y.Re, X*Y.Im);
end "*";
```

but outside we could only write

```
use Complex_Numbers;

function "*" (X: Float; Y: Complex) return Complex is
begin
   return Cons(X, 0.0) * Y;
end "*";
```

and similarly with the operands interchanged.

2
```
declare
   C, D: Complex_Numbers.Complex;
   F: Float;
begin
   C := Complex_Numbers.Cons(1.5, –6.0);
   D := Complex_Numbers."+" (C, Complex_Numbers.I);
   F := Complex_Numbers.Re_Part(D) + 6.0;
   ...
end;
```

3
```
package Rational_Numbers is
   type Rational is private;

   function "+" (X: Rational) return Rational;
   function "–" (X: Rational) return Rational;

   function "+" (X, Y: Rational) return Rational;
   function "–" (X, Y: Rational) return Rational;
   function "*" (X, Y: Rational) return Rational;
   function "/" (X, Y: Rational) return Rational;

   function "/" (X: Integer; Y: Positive) return Rational;
```

```
   function Numerator(R: Rational) return Integer;
   function Denominator(R: Rational) return Positive;

private
   type Rational is
      record
         Num: Integer := 0;     -- numerator
         Den: Positive := 1;    -- denominator
      end record;
end;

package body Rational_Numbers is

   function Normal(R: Rational) return Rational is
      -- cancel common factors
      G: Positive := GCD(abs R.Num, R.Den);
   begin
      return (R.Num/G, R.Den/G);
   end Normal;

   function "+" (X: Rational) return Rational is
   begin
      return X;
   end "+";

   function "-" (X: Rational) return Rational is
   begin
      return (-X.Num, X.Den);
   end "-";

   function "+" (X, Y: Rational) return Rational is
   begin
      return Normal((X.Num*Y.Den + Y.Num*X.Den,
                                        X.Den*Y.Den));
   end "+";

   function "-" (X, Y: Rational) return Rational is
   begin
      return Normal((X.Num*Y.Den - Y.Num*X.Den,
                                        X.Den*Y.Den));
   end "-";

   function "*" (X, Y: Rational) return Rational is
   begin
      return Normal((X.Num*Y.Num, X.Den*Y.Den));
   end "*";

   function "/" (X, Y: Rational) return Rational is
      N: Integer := X.Num*Y.Den;
      D: Integer := X.Den*Y.Num;
   begin
      if D < 0 then D := -D;  N := -N; end if;
      return Normal((Num => N, Den => D));
   end "/";

   function "/" (X: Integer; Y: Positive)
                                  return Rational is
   begin
      return Normal((Num => X, Den =>Y));
   end "/";

   function Numerator(R: Rational) return Integer is
   begin
      return R.Num;
   end Numerator;

   function Denominator(R: Rational)
                                  return Positive is
   begin
      return R.Den;
   end Denominator;

end Rational_Numbers;
```

4 Although the parameter types are both Integer and therefore the same as for predefined integer division, nevertheless the result types are different. The result types are considered in the hiding rules for functions. See Section 10.5.

Exercise 12.3

1
```
package Metrics is
   type Length is new Float;
   type Area is new Float;
   function "*" (X, Y: Length) return Length
                                  is abstract;
   function "*" (X, Y: Length) return Area;
   function "*" (X, Y: Area) return Area is abstract;
   function "/" (X, Y: Length) return Length
                                  is abstract;
   function "/" (X: Area; Y: Length) return Length;
   function "/" (X, Y: Area) return Area is abstract;
   function "**" (X: Length; Y: Integer) return Length
                                  is abstract;
   function "**" (X: Length; Y: Integer) return Area;
   function "**" (X: Area; Y: Integer) return Area
                                  is abstract;
end;

package body Metrics is

   function "*" (X, Y: Length) return Area is
   begin
      return Area(Float(X) * Float(Y));
   end "*";

   function "/" (X: Area; Y: Length) return Length is
   begin
      return Length(Float(X) / Float(Y));
   end "/";

   function "**" (X: Length; Y: Integer) return Area is
   begin
      if Y = 2 then
         return X * X;
      else
         raise Constraint_Error;
      end if;
   end "**";

end Metrics;
```

Exercise 12.4

1

```
package Stacks is
   type Stack is private;
   Empty: constant Stack;
   ...
private
   ...
   Empty: constant Stack := ((1 .. Max => 0), 0);
end;
```

Note that Empty has to be initialized because it is a **constant** despite the fact that Top which is the only component whose value is of interest is default initialized anyway. Another approach is to declare a function Empty. This has the advantage of being a primitive operation of Stack and so inherited if we derived from Stack.

2

```
function Is_Empty(S: Stack) return Boolean is
begin
   return S.Top = 0;
end Is_Empty;

function Is_Full(S: Stack) return Boolean is
begin
   return S.Top = Max;
end Is_Full;
```

Whereas Is_Empty can test for an empty stack it cannot be used to set a stack empty. A constant or function Empty plus equality can do both.

3

```
function "=" (S, T: Stack) return Boolean is
begin
   return S.S(1 .. S.Top) = T.S(1 .. T.Top);
end "=";
```

4

```
function "=" (A, B: Stack_Array) return Boolean is
begin
   if A'Length /= B'Length then
      return False;
   end if;
   for I in A'Range loop
      if A(I) /= B(I+B'First–A'First) then
         return False;
      end if;
   end loop;
   return True;
end "=";
```

This uses the redefined = (via /=) applying to the type Stack. This pattern for array equality clearly applies to any type. Beware that we cannot use slice comparison (as in the previous answer) because that would call the function being declared and so recurse infinitely.

Equality and inequality returning Boolean arrays might be

```
type Boolean_Array is
      array (Integer range <>) of Boolean;

function "=" (A, B: Stack_Array)
                              return Boolean_Array is
   Result: Boolean_Array(A'Range);
begin
   if A'Length /= B'Length then
      return (A'Range => False);
   end if;
   for I in A'Range loop
      Result(I) := A(I) /= B(I+B'First–A'First);
   end loop;
   return Result;
end "=";
```

It might be better to raise Constraint_Error in the case of arrays of unequal lengths.

```
function "/=" (A, B: Stack_Array)
                              return Boolean_Array is
begin
   return not (A = B);
end "/=";
```

Recall from Section 8.6 that **not** can be applied to all one-dimensional Boolean arrays.

5

```
package Stacks is
   type Stack is private;
   procedure Push(S: in out Stack; X: in Integer);
   procedure Pop(S: in out Stack; X: out Integer);
private
   Max: constant := 100;
   Dummy: constant Integer := 0;
   type Integer_Vector is
         array (Integer range <>) of Integer;
   type Stack is
      record
         S: Integer_Vector(1 .. Max) :=
                                   (1 .. Max => Dummy);
         Top: Integer range 0 .. Max := 0;
      end record;
end;

package body Stacks is
   procedure Push(S: in out Stack; X: in Integer) is
   begin
      S.Top := S.Top + 1;
      S.S(S.Top) := X;
   end;

   procedure Pop(S: in out Stack; X: out Integer) is
   begin
      X := S.S(S.Top);
      S.S(S.Top) := Dummy;
      S.Top := S.Top – 1;
   end;
end Stacks;
```

Note the use of Dummy as a default Integer value for unused components of the stack.

6
```
function "=" (X, Y: Rational) return Boolean is
begin
   return X.Num * Y.Den = X.Den * Y.Num;
end "=";
```

Overflow would soon occur without reduction after each operation; this would not be sensible.

Exercise 12.5

1 In the case of the access formulation, although Is_Empty is straightforward, it is difficult to write an appropriate function Is_Full; we will return to this when exceptions are discussed in detail in Chapter 14.

```
function Is_Empty(S: Stack) return Boolean is
begin
   return S = null;
end Is_Empty;
```

2
```
function "=" (S, T: Stack) return Boolean is
   SL: Cell_Ptr := Cell_Ptr(S);
   TL: Cell_Ptr := Cell_Ptr(T);
begin
   while SL /= null and TL /= null loop
      if SL.Value /= TL.Value then
         return False;
      end if;
      SL := SL.Next;
      TL := TL.Next;
   end loop;
   return SL = TL;
end "=";
```

3
```
package Queues is
   Empty: exception;
   type Queue is limited private;
   procedure Join(Q: in out Queue; X: in Item);
   procedure Remove(Q: in out Queue; X: out Item);
   function Length(Q: Queue) return Integer;
private
   type Cell;
   type Cell_Ptr is access Cell;
   type Cell is
      record
         Next: Cell_Ptr;
         Data: Item;
      end record;
   type Queue is
      record
         First, Last: Cell_Ptr;
         Count: Integer := 0;
      end record;
end;

package body Queues is

   procedure Join(Q: in out Queue; X: in Item) is
      L: Cell_Ptr;
   begin
      L := new Cell'(Next => null, Data => X);
      if Q.Count = 0 then          -- queue was empty
         Q.First := L;
         Q.Last := L;
      else
         Q.Last.Next := L;
         Q.Last := L;
      end if;
      Q.Count := Q.Count + 1;
   end Join;

   procedure Remove(Q: in out Queue; X: out Item) is
   begin
      if Q.Count = 0 then
         raise Empty;
      end if;
      X := Q.First.Data;
      Q.First := Q.First.Next;
      Q.Count := Q.Count – 1;
   end Remove;

   function Length(Q: Queue) return Integer is
   begin
      return Q.Count;
   end Length;

end Queues;
```

It would be tidy to assign **null** to Q.Last in the case when the last item is removed but it is not strictly necessary. See also Section 25.4 and in particular Exercise 25.4(**1**).

Exercise 12.6

1
```
private
   Max: constant := 1000;        -- no of accounts
   type Key_Code is new Integer range 0 .. Max;
   subtype Key_Range is
               Key_Code range 1 .. Key_Code'Last;
   type Key is
      record
         Code: Key_Code := 0;
      end record;
end;

package body Bank is
   Balance: array (Key_Range) of Money :=
                                       (others => 0);
   Free: array (Key_Range) of Boolean :=
                                       (others => True);

   function Valid(K: Key) return Boolean is
   begin
      return K.Code /= 0;
   end Valid;
```

```
   procedure Open_Account(K: in out Key;
                          M: in Money) is
   begin
      if K.Code = 0 then
         for I in Free'Range loop
            if Free(I) then
               Free(I) := False;
               Balance(I) := M;
               K.Code := I;
               return;
            end if;
         end loop;
      end if;
   end Open_Account;

   procedure Close_Account(K: in out Key;
                           M: out Money) is
   begin
      if Valid(K) then
         M := Balance(K.Code);
         Free(K.Code) := True;
         K.Code := 0;
      end if;
   end Close_Account;

   procedure Deposit(K: in Key;
                     M: in Money) is
   begin
      if Valid(K) then
         Balance(K.Code) := Balance(K.Code) + M;
      end if;
   end Deposit;

   procedure Withdraw(K: in out Key;
                      M: in out Money) is
   begin
      if Valid(K) then
         if M > Balance(K.Code) then
            Close_Account(K, M);
         else
            Balance(K.Code) := Balance(K.Code) – M;
         end if;
      end if;
   end Withdraw;

   function Statement(K: Key) return Money is
   begin
      if Valid(K) then
         return Balance(K.Code);
      end if;
   end Statement;

end Bank;
```

Various alternative formulations are possible. It might be neater to declare a record type representing an account containing the two components Free and Balance.

Note that the function Statement will raise Program_Error if the key is not valid. Alternatively we could return a dummy value of zero but it might be better to raise our own exception as described in Chapter 15.

2 An alternative formulation which represents the home savings box could be that where the limited private type is given by

```
type Box is
   record
      Code: Box_Code := 0;
      Balance: Money;
   end record;
```

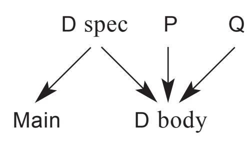
In this case the money is kept in the variable declared by the user. The bank only knows which boxes have been issued but does not know how much is in a particular box. The details are left to the reader.

3 Since the parameter is of an explicitly limited type, it is always passed by reference and nothing can go wrong. However, if the record type had not been explicitly limited then it might be passed by copy or by reference. If it is passed by copy then the call of Action will succeed whereas if it is passed by reference it will not.

4 Again, since the parameter is of an explicitly limited type it will be passed by reference and so cannot be changed. However, even if it had not been explicitly limited and passed by copy then he would still have been thwarted because of the rule mentioned in Section 10.3 that a record type with any default initialized components is always copied in precisely so that junk values cannot be created.

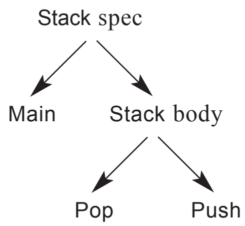
Answers 13

Exercise 13.1



1 The number of possible orders of compilation is (a) 120, (b) 18. The source model thus gives the programmer much more flexibility.

Exercise 13.2



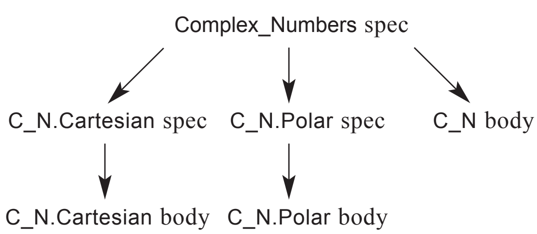
1 The number of possible orders of compilation is (a) 120, (b) 8. The source model again is much more flexible. Indeed the number of combinations for the source model is always $n!$ (where n is the number of units) irrespective of the dependency structure.

Exercise 13.3

1
```
package Complex_Numbers is
   type Complex is private;
   ...
   function "+" (X, Y: Complex) return Complex;
   ...
private
   ...
end Complex_Numbers;

package Complex_Numbers.Cartesian is
   function Cons(R, I: Float) return Complex;
   function Re_Part(X: Complex) return Float;
   function Im_Part(X: Complex) return Float;
end Complex_Numbers.Cartesian;

package Complex_Numbers.Polar is
   ...   -- as before
end Complex_Numbers.Polar;
```

The bodies are as expected.

2 We have used the abbreviation C_N for Complex_Numbers.



3 The only subprograms inherited are the arithmetic operations in the package Complex_Numbers. Those in the child packages are not primitive operations of the type Complex and so are not inherited.

Exercise 13.4

1
```
private package Rational_Numbers.Slave is
   function Normal(R: Rational) return Rational;
end;

package body Rational_Numbers.Slave is

   function GCD(X, Y: Natural) return Natural is
      ...
   end GCD;

   function Normal(R: Rational) return Rational is
      G: Positive := GCD(abs R.Num, R.Den);
   begin
      return (R.Num/G, R.Den/G);
   end Normal;

end Rational_Numbers.Slave;

with Rational_Numbers.Slave;
package body Rational_Numbers is
   use Slave;
   -- as before without the function Normal
   ...
end Rational_Numbers;
```

An alternative is to make Normal a private child function.

2
```
package Complex_Numbers.Trig is
   function Sin(X: Complex) return Complex;
   function Cos(X: Complex) return Complex;
   ...
end;

private function Complex_Numbers.Trig.
            Sin_Cos(X: Complex) return Complex is
begin
   ...
end;

with Complex_Numbers.Trig.Sin_Cos;
package body Complex_Numbers.Trig is
   ...
end Complex_Numbers.Trig;
```

The function Sin_Cos can be called directly as such within the body of Trig without a use clause.

Exercise 13.7

1
```
function Monday return Diurnal.Day
      renames Diurnal.Mon;
```

2 This cannot be done because Next_Work_Day is of an anonymous type.

3
```
Pets: String_3_Array renames Farmyard(2 .. 3);
```
Note that the bounds of Pets are 2 and 3.

4
```
function "+" (X, Y: Complex_Numbers.Complex)
                     return Complex_Numbers.Complex
      renames Complex_Numbers."+";
```

5
```
This_Cell: Cell renames World(I, J);
```

Exercise 13.8

1
```
package P is
   B: Boolean;
end;

with P;
package Q is
   ...   -- spec of Q
end;

package body Q is
   ...
begin
   P.B := True;
end Q;

with P;
package R is
   ...   -- spec of R
end;

with Q;
pragma Elaborate(Q);
package body R is
   ...
begin
   P.B := False;
end R;
```

We need to ensure that the bodies are elaborated in a specific order. Placing the pragma Elaborate for Q on the body of R ensures that the body of Q is elaborated before that of R. Thus P.B will be finally set False. Note also that R has to have a with clause for Q. The pragma could alternatively be placed on the specification of R since a body is always elaborated after its specification.

Answers 14

Exercise 14.1

1
```
P: Point := (Object(C) with null record);
```

2
```
R: Reservation:
...
R.Flight_Number := 77;
R.Date_Of_Travel := (5, Nov, 2007);
...
NR: Nice_Reservation := (R with Window, Green);
```

3
```
package Reservation_System.Supersonic is
   type Supersonic_Reservation is
                                 new Reservation with
      record
         ...
      end record;

   procedure Make(SR: in out Supersonic_Reservation);
   ...
end Reservation_System.Supersonic;
```

Exercise 14.2

1
```
procedure Print_Area(OC: Object'Class) is
begin
   Put(Area(OC));   -- dispatch to appropriate Area
end;
```

2
```
type Person is tagged
   record
      Birth: Date;
   end record;

type Man is new Person with
   record
      Bearded: Boolean;
   end record;

type Woman is new Person with
   record
      Children: Integer;
   end record;
```

3
```
procedure Print_Details(P: in Person) is
begin
   Print_Date(P.Birth);
end;

procedure Print_Details(M: in Man) is
begin
   Print_Details(Person(M));
   Print_Boolean(M.Beard);
end;
```

```
procedure Print_Details(W: in Woman) is
begin
   Print_Details(Person(W));
   Print_Integer(W.Children);
end;

procedure Analyse_Person(PC: Person'Class) is
begin
   Print_Details(PC);                 -- dispatch
end;
```

4
```
package body Queues is

   procedure Join(Q: access Queue;
                  E: in Element_Ptr) is
   begin
      if E.Next /= null then    -- already on a queue
         raise Queue_Error;
      end if;
      if Q.Count = 0 then       -- queue was empty
         Q.First := E;
         Q.Last := E;
      else
         Q.Last.Next := E;
         Q.Last := E;
      end if;
      Q.Count := Q.Count + 1;
   end Join;

   function Remove(Q: access Queue)
                        return Element_Ptr is
      Result: Element_Ptr;
   begin
      if Q.Count = 0 then
         raise Queue_Error;
      end if;
      Result := Q.First;
      Q.First := Result.Next;
      Result.Next := null;
      Q.Count := Q.Count - 1;
      return Result;
   end Remove;

   function Length(Q: Queue) return Integer is
   begin
      return Q.Count;
   end Length;

end Queues;
```

Exercise 14.3

1
```
function Further(X, Y: Object) return Object is
begin
   if Distance(X) > Distance(Y) then
      return X;
   else
      return Y;
   end if;
end Further;
```

Since it returns the type Object it becomes abstract when inherited by Circle and so has to be overridden. This is very frustrating because the text is essentially unchanged. But Point is OK because the extension is null.

2
```
package Objects is
   type Object is abstract tagged
      record
         X_Coord: Float;
         Y_Coord: Float;
      end record;

   function Distance(O: Object'Class) return Float;
   function Area(O: Object) return Float is abstract;
end Objects;

with Objects; use Objects;
package Shapes is
   type Point is new Object with null record;

   function Area(P: Point) return Float;

   type Circle is new Object with
      record
         Radius: Float;
      end record;

   function Area(C: Circle) return Float;
   ...      -- etc.
end Shapes;
```

3
```
function Further(X, Y: Object'Class)
                        return Object'Class is
begin
   if Distance(X) > Distance(Y) then
      return X;
   else
      return Y;
   end if;
end Further;
```

This calls the class wide function Distance and can be applied to all objects without change. Moreover, it can be used to compare the distances of two different types of object such as a triangle and a circle.

4 The function Bigger cannot be written for the type Object because it would contain a call of the abstract function Area. Functions could of course be written for Circle and Point but would need to be written out for each. A better solution is again to use class wide parameters as in

```
function Bigger(X, Y: Object'Class)
                        return Object'Class is
begin
   if Area(X) > Area(Y) then
      return X;
   else
      return Y;
   end if;
end Bigger;
```

This dispatches to the appropriate functions Area and can be used to compare the areas of any two objects in the class. However, it would be better to manipulate references to the objects rather than the objects themselves. Indeed if we wish to know which is bigger we don't really want to be given a copy of the bigger one, we want an access value referring to it. The extension problems with the function results would then all go away.

5

```
package body Reservation_System.Subsonic is

   procedure Make(BR: in out Basic_Reservation) is
      Select_Seat(BR);
   end Make;

   procedure Make(NR: in out Nice_Reservation) is
      Make(Basic_Reservation(NR));
      Order_Meal(NR);
   end Make;

   procedure Make(PR: in out Posh_Reservation) is
      Make(Nice_Reservation(PR));
      Arrange_Limo(PR);
   end Make;
   ...   -- Select_Seat etc as before
end Reservation_System.Subsonic;
```

Exercise 14.4

1

```
function "=" (C, D: Circle) return Boolean is
begin
   return Object(C) = Object(D) and
               abs(C.Radius – D.Radius) < Epsilon;
end "=";
```

2

```
function "=" (C, D: Circle) return Boolean is
begin
   return Old_Equality(Object(C), Object(D)) and
               abs(C.Radius – D.Radius) < Epsilon;
end "=";
```

Exercise 14.5

1 It would fail to compile because, as mentioned in Section 14.2, Order_Meal is not a primitive operation of all types in the class rooted at Reservation.

2

```
procedure Select_Seat(NR: in out Nice_Reservation) is
begin
   if NR.Seat_Sort = Aisle then
      -- choose aisle seat
   else
      -- choose window seat
   end if;
end Select_Seat;
...
procedure Make(R: in out Reservation) is
begin
   Select_Seat(Reservation'Class(R));  -- redispatch
end Make;
```

Observe that we know that the component NR.Seat_Sort must exist because this is a nice reservation or derived from it. Naturally enough all seats are window or aisle seats in nice and posh categories.

Exercise 14.6

1 We could not declare the function Further with Shape as parameter and result because it has a controlling result and would prevent further extension from the partial view. But the class wide version taking Shape'Class is acceptable.

Exercise 14.7

1

```
procedure Adjust(Object: in out Thing) is
begin
   The_Count := The_Count + 1;
end Adjust;
```

The procedures Initialize and Finalize are as before.

2 The type Key and the control procedures could be

```
type Key is new Limited_Controlled with
   record
      Code: Key_Code;
   end record;

procedure Initialize(K: in out Key) is
begin
   K.Code := 0;
end Initialize;

procedure Finalize(K: in out Key) is
begin
   Return_Key(K);
end Finalize;
```

We have chosen to use Initialize to set the initial value but we could have left this to be done by the default mechanism as before.

Exercise 14.9

1 (a) legal – both tags statically the same
(b) illegal – tags statically different
(c) illegal – cannot mix static and dynamic cases
(d) legal – but tags checked at run time

2
```
procedure Convert(From: in Set'Class;
                  To: out Set'Class) is
begin
   if From'Tag = To'Tag then
      To := From;
   else
      declare
         Temp: Set'Class := From;
         -- and so on
      end;
   end if;
end Convert;
```

3
```
procedure Convert(From: in Stack'Class;
                  To: out Stack'Class) is
begin
   To := Empty;
   declare
      Temp1: Stack'Class := From;   -- copy original
      Temp2: Stack'Class := To;     -- the siding
      E: Element;
   begin
      while Temp1 /= Empty loop
         Pop(Temp1, E);
         Push(Temp2, E);
      end loop;
      while Temp2 /= Empty loop
         Pop(Temp2, E);
         Push(To, E);
      end loop;
   end;
end Convert;
```

A double loop is required otherwise the elements end up in reverse order. So having copied the whole stack into Temp1 to avoid destroying the original, we then move the individual elements into a second temporary which we can think of as a siding using a railroad analogy and finally reverse them out into the final destination.

Other approaches are possible such as introducing a procedure Rev but these result in lots of copying. Note the use of the inner block so that Temp2 can be initialized with an empty stack of the same type as To.

The linked stack might be

```
type Linked_Stack is new Stack with
   record
      Component: Inner;
   end record;
```

where Inner is as for the Linked_Set. The deep copy mechanism is identical. As in the answer to Exercise 12.5(**2**), equality can be defined as

```
function "=" (S, T: Linked_Stack) return Boolean is
   SL: Cell_Ptr := S.Component.The_Set;
   TL: Cell_Ptr := T.Component.The_Set;
begin
   ... -- as Exercise 12.5(2)
end "=";
```

The array stack might be

```
type Array_Stack is new Stack with
   record
      S: Element_Vector(1 .. Max);
      Top: Integer range 0 .. Max := 0;
   end record;
```

with equality as in Section 12.4.

Answers 15

Exercise 15.1

1
```
procedure Quadratic(A, B, C: in Float;
                    Root_1, Root_2: out Float;
                    OK: out Boolean) is
   D: constant Float := B**2 – 4.0*A*C;
begin
   Root_1 := (–B+Sqrt(D)) / (2.0*A);
   Root_2 := (–B–Sqrt(D)) / (2.0*A);
   OK := True;
exception
   when Constraint_Error =>
      OK := False;
end Quadratic;
```

2
```
function Factorial(N: Integer) return Integer is

   function Slave(N: Natural) return Positive is
   begin
      if N = 0 then
         return 1;
      else
         return N * Slave(N–1);
      end if;
   end Slave;

begin
   return Slave(N);
exception
   when Constraint_Error | Storage_Error =>
      return –1;
end Factorial;
```

Exercise 15.2

1
```
package Random is
   Bad: exception;
   Modulus: constant := 2**13;
   subtype Small is Integer range 0 .. Modulus;
   procedure Init(Seed: in Small);
   function Next return Small;
end;

package body Random is
   Multiplier: constant := 5**5;
   X: Small;

   procedure Init(Seed: in Small) is
   begin
      if Seed mod 2 = 0 then
         raise Bad;
      end if;
      X := Seed;
   end Init;

   function Next return Small is
   begin
      X := X * Multiplier mod Modulus;
      return X;
   end Next;

end Random;
```

2
```
function Factorial(N: Integer) return Integer is

   function Slave(N: Natural) return Positive is
   begin
      if N = 0 then
         return 1;
      else
         return N * Slave(N-1);
      end if;
   end Slave;

begin
   return Slave(N);
exception
   when Storage_Error =>
      raise Constraint_Error;
end Factorial;
```

3
```
function "+" (X, Y: Vector) return Vector is
   R: Vector(X'Range);
begin
   if X'Length /= Y'Length then
      raise Constraint_Error;
   end if;
   for I in X'Range loop
      R(I) := X(I) + Y(I+Y'First-X'First);
   end loop;
   return R;
end "+";
```

4 No. A malevolent user could write

```
raise Stack.Error;
```

outside the package. It would be nice if the language provided some sort of 'private' exception that could be handled but not raised explicitly outside its defining package.

5
```
procedure Push(S: in out Stack; X: in Integer) is
begin
   S := new Cell'(S, X);
exception
   when Storage_Error =>
      raise Error;
end;

procedure Pop(S: in out Stack; X: out Integer) is
begin
   if S = null then
      raise Error;
   else
      X := S.Value;
      S := S.Next;
   end if;
end;
```

Exercise 15.3

1 Four checks are required. The one inserted by the user, the overflow check on the assignment to Top plus the two for the assignment to S(Top) which cannot be avoided since we can say little about the value of Top (except that it is not equal to Max). So this is the worst of all worlds thus emphasizing the need to give appropriate constraints.

Exercise 15.4

1 The subprograms become

```
procedure Push(X: Integer) is
begin
   if Top = Max then
      raise Error with "stack overflow";
   end if;
   Top := Top + 1;
   S(Top) := X;
end Push;

function Pop return Integer is
begin
   if Top = 0 then
      raise Error with "stack underflow";
   end if;
   Top := Top - 1;
   return S(Top + 1);
end Pop;
```

and the handler might become

```
when Event: Error =>
   Put("Stack used incorrectly because of ");
   Put(Exception_Message(Event));
   Clean_Up;
```

Exercise 15.5

1
```
pragma Assert(for all K in A'First .. A'Last – 1 =>
                                        A(K) <= A(K+1));
```

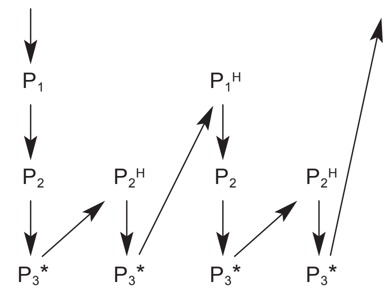
See also the answer to Exercise 9.4(**1**).

Exercise 15.6

1
```
package Bank is
   Alarm: exception;
   type Money is new Natural;
   type Key is limited private;
   ...      -- as before
end;

package body Bank is
   Balance: array (Key_Range) of Money :=
                                       (others => 0);
   Free: array (Key_Range) of Boolean :=
                                       (others => True);

   function Valid(K: Key) return Boolean is
   begin
      return K.Code /= 0;
   end Valid;

   procedure Validate(K: Key) is
   begin
      if not Valid(K) then
         raise Alarm;
      end if;
   end Validate;

   procedure Open_Account(K: in out Key;
                          M: in Money) is
   begin
      if K.Code = 0 then
         for I in Free'Range loop
            if Free(I) then
               Free(I) := False;
               Balance(I) := M;
               K.Code := I;
               return;
            end if;
         end loop;
      else
         raise Alarm;
      end if;
   end Open_Account;

   procedure Close_Account(K: in out Key;
                           M: out Money) is
   begin
      Validate(K);
      M := Balance(K.Code);
      Free(K.Code) := True;
      K.Code := 0;
   end Close_Account;

   procedure Deposit(K: in Key;
                     M: in Money) is
   begin
      Validate(K);
      Balance(K.Code) := Balance(K.Code) + M;
   end Deposit;

   procedure Withdraw(K: in out Key;
                      M: in out Money) is
   begin
      Validate(K);
      if M > Balance(K.Code) then
         raise Alarm;
      else
         Balance(K.Code) := Balance(K.Code) – M;
      end if;
   end Withdraw;

   function Statement(K: Key) return Money is
   begin
      Validate(K);
      return Balance(K.Code);
   end Statement;
end Bank;
```

For convenience we have declared a procedure Validate which raises the alarm in most cases. The Alarm is also explicitly raised if we attempt to overdraw but as remarked in the text we cannot also close the account (unless we are assured that the key is either explicitly limited or tagged). An attempt to open an account with a key which is in use also causes Alarm to be raised. We do not however raise the Alarm if the bank runs out of accounts but have left it to the user to check with a call of Valid that he was issued a genuine key; the rationale is that it is not the user's fault if the bank runs out of keys.

2 Suppose *N* is 2. Then on the third call, P is not entered but the exception is raised and handled at the second level. The handler again calls P without success but this time, since an exception raised in a handler is not handled there but propagated up a level, the exception is handled at the first level. The pattern then repeats but the exception is finally propagated out of the first level to the originating call. In all there are three successful calls and four unsuccessful ones. The diagram opposite might help.



An * indicates an unsuccessful call, H indicates a call from a handler.

More generally suppose C_n is the total number of calls for the case $N = n$. Then by induction

$$C_{n+1} = 2C_n + 1$$

with the initial condition $C_0 = 1$ since in the case $N = 0$ it is obvious that there is only one call which fails. It follows that the total number of calls C_N is $2^{N+1} - 1$. Of these 2^N are unsuccessful and $2^N - 1$ are successful.

I am grateful to Bob Bishop for this amusing example.

Answers 16

Exercise 16.1

1
```
type My_Boolean is new  Boolean
   wiith Default_Value => True;
```

In the case of most Boolean aspects such as Inline we can omit True which is then taken by default. But this does not apply to Default_Value since it can apply to any type and not just Boolean.

Exercise 16.2

1
```
overriding
function Area(S: Square) return Float
   with Pre => S.Side > 0.0 and
                                    S.X_Coord > S.Side,
        Post => abs (Area'Result – S.Side**2) >
                                                  Eps;
```

The precondition ensures that all of the square is in the positive half-plane for some reason. The postcondition requires the answer to be correct within some small value Eps.

Exercise 16.3

1
```
function Is_Unduplicated(S: Stack)
                                     return Boolean is
begin
   for I in 1 .. S.Top–1 loop
      for J in I+1 .. S.Top loop
         if S.S(I) = S.S(J) then
            return False;
         end if;
      end loop;
   end loop;
   return True;
end Is_Unduplicated;
```

2
```
function Is_Unduplicated(S: Stack)
                                     return Boolean is
(not(for some I in 1 .. S.Top–1 =>
   (for some J in I+1 .. S.Top => S.S(I) = S.S(J))));
```

Note how this closely mimics the previous answer. Of course this expression function cannot be used as the precondition directly because it needs access to the implementation details. So it is just used as a completion. However, remember that it can be given in the private part as a completion and so is visible to the human reader of the package specification.

Exercise 16.4

1
```
subtype Primary is Rainbow
   with Static_Predicate =>
                         Primary in Red | Yellow | Blue;
```

2
```
subtype Curious is Integer
   with Dynamic_Predicate => Curious in 1 .. 999
                              and Curious mod 37 = 1;
```

If we wanted to use a static predicate then we would have to write the possible values out thus

```
subtype Curious is Integer
   with Static_Predicate =>
      Curious in 38 | 75 | 112 | 149 | 186 | 223 ... ;
```

Exercise 16.6

1
```
subtype Even is Integer
with Dynamic_Predicate => Even mod 2 = 0,
     Predicate_Failure => raise Constraint_Error
                    with "something odd about even!";
```

Answers 17

Exercise 17.1

1 P: on A: like Integer, on B: like Short_Integer
Q: on A: like Long_Integer, on B: like Integer
R: on A: not possible, on B: like Long_Integer

2 No, the only critical case is type Q on machine A. Changing to one's complement changes the range of Integer to

–32767 .. +32767

and so only Integer'First is altered.

3 (a) Integer'Base
(b) illegal – need explicit conversion
(c) My_Integer'Base
(d) Integer'Base
(e) *root_integer*
(f) My_Integer'Base

4
```
type Longest_Integer is
        range System.Min_Int .. System.Max_Int;
```

Exercise 17.2

1 (a) 16#EF#
(b) 120
(c) 222
(d) 239

2
```
type Ring5 is mod 5;
A, B, C, D: Ring5;
...
D := (A + B) * C;
```

3 (a) 4
(b) 1

DeMorgan's theorem that

not (A **and** B) = **not** A **or not** B

does not hold if the modulus is not a power of two.

4
```
type Bearing is mod 360 * 60;
Degree: constant Bearing := 60;
SE: constant Bearing := 135*Degree;
NNW: constant Bearing := 337*Degree+30;
NE_by_E: constant Bearing := 56*Degree+15;
```

Exercise 17.3

1 (a) illegal
(b) Integer'Base
(c) *root_real*
(d) Integer'Base
(e) *root_real*
(f) *root_integer*

2
```
R: constant := N * 1.0;
```

Exercise 17.4

1
```
type Real is digits 7;
type Real_Vector is
      array (Integer range <>) of Real;

function Inner(A, B: Real_Vector) return Real is
   type Long_Real is digits 14;
   Result: Long_Real := 0.0;
begin
   for I in A'Range loop
      Result := Result +
                    Long_Real(A(I)) * Long_Real(B(I));
   end loop;
   return Real(Result);
end Inner;
```

Exercise 17.5

1 The literal 0.5 is *universal_real* and this matches *universal_fixed*.

2
```
function "**" (X: Complex; N: Integer)
                                  return Complex is
   Result_θ: Angle := 0.0;
begin
   for I in 1 .. abs N loop
      Result_θ := Normal(Result_θ + X.θ);
   end loop;
   if N < 0 then
      Result_θ := –Result_θ;
   end if;
   return (X.R**N, Result_θ);
end "**";
```

We cannot simply write

```
   return (X.R**N, Normal(X.θ * N));
```

because if **abs** N is larger than 3 the multiplication is likely to overflow; so we have to repeatedly normalize. A clever solution which is faster for all but the smallest values of **abs** N is

```
function "**" (X: Complex; N: Integer)
                                  return Complex is
   Result_θ: Angle := 0.0;
   Term: Angle := X.Theta;
   M: Integer := abs N;
begin
   while M > 0 loop
      if M rem 2 /= 0 then
         Result_θ := Normal(Result_θ + Term);
      end if;
      M := M / 2;
      Term := Normal(Term * 2);
   end loop;
   if N < 0 then
```

```
    Result_θ := –Result_θ;
  end if;
  return (X.R**N, Result_θ);
end "**";
```

This is a variation of the standard algorithm for computing exponentials by decomposing the exponent into its binary form and doing a minimal number of multiplications. In our case it is the multiplier N which we decompose and then do a minimal number of additions.

Recognizing that our repeated addition algorithm is essentially the same as for exponentiation, we can in parallel compute X.R**N by the same method. A little manipulation soon makes us realize that we might as well write

```
function "**" (X: Complex; N: Integer)
                                    return Complex is
  One: constant Complex := Cons(1.0, 0.0);
  Result: Complex := One;
  Term: Complex := X;
  M: Integer := abs N;
begin
  while M > 0 loop
    if M rem 2 /= 0 then
      Result := Result * Term;       -- Complex *
    end if;
    M := M / 2;
    Term := Term * Term;             -- Complex *
  end loop;
  if N < 0 then Result := One / Result; end if;
  return Result;
end "**";
```

This brings us back full circle. This is the standard algorithm for computing exponentials applied in the abstract to our type Complex. Note the calls of "*" and "/" applying to the type Complex. This version of "**" can be declared outside the package Complex_Numbers and is quite independent of the internal representation (but it will be very inefficient unless it is polar).

3

```
private
  type Angle is delta 0.05 range –720.0 .. 720.0;
  for Angle'Small use 2.0**(–5);
  type Complex is
    record
      R: Float;
      θ: Angle range 0.0 .. 360.0;
    end record;
  I: constant Complex := (1.0, 90.0);
end;
...
function Normal(φ: Angle) return Angle is
begin
  if φ >= 360.0 then
    return φ – 360.0;
  elsif φ < 0.0 then
    return φ + 360.0;
  else
    return φ;
  end if;
end Normal;
```

The choice of delta and small is deduced as follows. We need 10 bits to cover the range 0 .. 720 plus one bit for the sign thus leaving 5 bits after the binary point. So *small* will be 2^{-5} and thus any value of delta greater than that will do. We have chosen 0.05.

Answers 18

Exercise 18.1

1

```
Trace((M'Length, M))
```

If the two dimensions of M were not equal then Constraint_Error would be raised. Note that the lower bounds of M do not have to be 1; all that matters is that the number of components in each dimension is the same since sliding is permitted when building the aggregate.

2

```
package Stacks is
  type Stack(Max: Natural) is private;
  Empty: constant Stack;
  ...
private
  type Integer_Vector is
      array (Integer range <>) of Integer;
  type Stack(Max: Natural) is
    record
      S: Integer_Vector(1 .. Max);
      Top: Integer := 0;
    end record;
  Empty: constant Stack(0) := (0, (others => 0), 0);
end;
```

We have naturally chosen to make Empty a stack whose value of Max is zero. Note that the function "=" only compares the parts of the stacks which are in use. Thus we can write S = Empty to test whether a stack S is empty irrespective of its value of Max.

3

```
function Is_Full(S: Stack) return Boolean is
begin
  return S.Top = S.Max;
end Is_Full;
```

4

```
S: constant Square := (N, Make_Unit(N));
```

Exercise 18.2

1 Z: Polynomial := (0, (0 => 0));

The named notation has to be used because the array has only one component.

2
```
function "*" (P, Q: Polynomial) return Polynomial is
   R: Polynomial(P.N+Q.N) := (P.N+Q.N, (others => 0));
begin
   for I in P.A'Range loop
      for J in Q.A'Range loop
         R.A(I+J) := R.A(I+J) + P.A(I)*Q.A(J);
      end loop;
   end loop;
   return R;
end "*";
```

It is largely a matter of taste whether we write P.A'Range rather than 0 .. P.N.

3
```
function "–" (P, Q: Polynomial) return Polynomial is
   Size: Integer;
begin
   if P.N > Q.N then
      Size := P.N;
   else
      Size := Q.N;
   end if;
   declare
      R: Polynomial(Size);
   begin
      for I in 0 .. P.N loop
         R.A(I) := P.A(I);
      end loop;
      for I in P.N+1 .. R.N loop
         R.A(I) := 0;
      end loop;
      for I in 0 .. Q.N loop
         R.A(I) := R.A(I) – Q.A(I);
      end loop;
      return Normal(R);
   end;
end "–";
```

There are various other ways of writing this function. We could initialize R.A by using slice assignments

```
R.A(0 .. P.N) := P.A;
R.A(P.N+1 .. R.N) := (P.N+1 .. R.N => 0);
```

or even more succinctly by

```
R.A := P.A & (P.N+1 .. R.N => 0);
```

4
```
procedure Truncate(P: in out Polynomial) is
begin
   if P'Constrained then
      raise Truncate_Error;
   else
      P := (P.N–1, P.A(0 .. P.N–1));
   end if;
end Truncate;
```

5 Any unconstrained Polynomial could then include an array whose range is 0 .. Integer'Last. This will take a lot of space. Since most implementations are likely to adopt the strategy of setting aside the maximum possible space for an unconstrained record it is thus wise to keep the maximum to a practical limit by the use of a suitable subtype such as Index.

6
```
type Polynomial(N: Index := 0) is
   record
      A: Integer_Vector(0 .. N) :=
                        (0 .. N–1 => 0) & (N => 1);
   end record;
```

We cannot write a single aggregate because an aggregate can only have one dynamic choice which must be the only choice. An alternative approach is to declare a function thus

```
function F(N: Integer) return Integer_Vector;

type Polynomial(N: Index := 0) is
   record
      A: Integer_Vector(0 .. N) := F(N);
   end record;

...

function F(N: Integer) return Integer_Vector is
   R: Integer_Vector(0 .. N);
begin
   for I in 0 .. N–1 loop
      R(I) := 0;
   end loop;
   R(N) := 1;
   return R;
end F;
```

If the full type Polynomial is declared in the private part of a package then the function specification can also be in the private part with the function body in the package body. It does not matter that F is referred to before its body is elaborated provided that it is not actually called. If we declared a polynomial (without an initial value) before the body of F then Program_Error would be raised.

Clearly any initial value can be computed this way even if it cannot be written as aggregates.

7
```
package Rational_Polynomials is
   Max: constant := 10;
   subtype Index is Integer range 0 .. Max;
   type Rational_Polynomial(N, D: Index := 0) is
                                                private;

   function "+" (X: Rational_Polynomial)
                              return Rational_Polynomial;
   function "–" (X: Rational_Polynomial)
                              return Rational_Polynomial;

   function "+" (X, Y: Rational_Polynomial)
                              return Rational_Polynomial;
   function "–" (X, Y: Rational_Polynomial)
                              return Rational_Polynomial;
   function "*" (X, Y: Rational_Polynomial)
                              return Rational_Polynomial;
   function "/" (X, Y: Rational_Polynomial)
                              return Rational_Polynomial;

   function "/" (X, Y: Polynomial)
                              return Rational_Polynomial;
   function Numerator(R: Rational_Polynomial)
                                      return Polynomial;
   function Denominator(R: Rational_Polynomial)
                                      return Polynomial;
private
   type Rational_Polynomial(N, D: Index := 0) is
      record
         Num: Polynomial(N) := (N, (0 .. N => 0));
         Den: Polynomial(D) :=
                              (N, (0 => 1) & (1 .. N => 0));
      end record;
end;
```

It might be more elegant to write functions Zero and One taking a parameter giving the value of N like the function F of the previous exercise.

We make no attempt to impose any special language constraint on the denominator as we did in the type Rational where the denominator has subtype Positive.

8
```
function "&" (X, Y: V_String) return V_String is
begin
   return (X.N + Y.N, X.S & Y.S);
end "&";
```

Exercise 18.3

1
```
procedure Shave(P: in out Person) is
begin
   if P.Sex = Female then
      raise Shaving_Error;
   else
      P.Bearded := False;
   end if;
end Shave;
```

2
```
procedure Sterilize(M: in out Mutant) is
begin
   if M'Constrained and M.Sex /= Neuter then
      raise Sterilize_Error;
   else
      M := (Neuter, M.Birth);
   end if;
end Sterilize;
```

3
```
type Figure is (Circle, Point, Triangle);

type Object(Shape: Figure) is
   record
      X_Coord: Float;
      Y_Coord: Float;
      case Shape is
         when Circle =>
            Radius: Float;
         when Point =>
            null;
         when Triangle =>
            A, B, C: Float;
      end case;
   end record;

function Area(X: Object) return Float is
begin
   case X.Shape is
      when Circle =>
         return Pi * X.Radius**2;
      when Point =>
         return 0.0;
      when Triangle =>
         return ... ;
   end case;
end Area;
```

Note the similarity between the case statement in the function Area and the variant part of the type Object.

4
```
type Category is (Basic, Nice, Posh);
type Position is (Aisle, Window);
type Meal_Type is (Green, White, Red);

type Reservation(C: Category) is
   record
      Flight_Number: Integer;
      Date_Of_Travel: Date;
      Seat_Number: String(1 .. 3) := "   ";
      case C is
         when Basic => null;
         when Nice | Posh =>
            Seat_Sort: Position;
            Food: Meal_Type;
            case C is
               when Basic | Nice => null;
               when Posh =>
                  Destination: Address;
            end case;
      end case;
   end record;
```

Note the curiously nested structure which is forced upon us by the rule that the individual components must have distinct names.

Exercise 18.4

1
```
type Boxer(W: Weight; Sex: Gender) is
      new Person(Sex => Sex) with
   record
      ...
   end record;
```

We have chosen to give the new discriminant Sex the same name as the old one.

2
```
function Geometry.Polygons.Four_Sided.
    Make_Quadrilateral(Sides, Angles: Float_Array)
                             return Quadrilateral is
   P: Polygon := Make_Polygon(Sides, Angles);
begin
   if P.No_Of_Sides /= 4 then
      raise Queer_Quadrilateral;
   end if;
   return (P with null record);
end Geometry.Polygons.Four_Sided.
                             Make_Quadrilateral;
```

3
```
package Geometry.Polygons.Four_Sided.
                                Conversions is
   function To_Parallelogram(Q: Quadrilateral'Class)
                             return Parallelogram;
   function To_Square(Q: Quadrilateral'Class)
                                return Square;
end;

package body Geometry.Polygons.Four_Sided.
                                Conversions is
   function To_Parallelogram(Q: Quadrilateral'Class)
                             return Parallelogram is
   begin
      if Q.Sides(1) /= Q.Sides(3) or
                    Q.Sides(2) /= Q.Sides(4) then
         raise Poor_Parallelogram;
      end if;
      return (Quadrilateral(Q) with null record);
   end To_Parallelogram;

   function To_Square(Q: Quadrilateral'Class)
                                   return Square is
      P: Parallelogram := To_Parallelogram(Q);
   begin
      if P.Sides(1) /= P.Sides(2) or
                 P.Angles(1) /= P.Angles(2) then
         raise Silly_Square;
      end if;
      return (P with null record);
   end To_Square;

end Geometry.Polygons.Four_Sided.Conversions;
```

These two functions suffice. Putting them in a child package ensures that there are no problems of going abstract on extension. Using a class wide parameter gives greater flexibility. We can convert towards the type Quadrilateral by normal type conversion. Other intermediate types such as Rectangle or Rhombus could be dealt with in a similar manner.

Note also the use of the extension aggregates in the return statements – and especially that the ancestor type must be specific and so a type conversion is required in To_Parallelogram in order to convert the class wide formal parameter Q to the specific type Quadrilateral.

Exercise 18.5

1
```
function Heir(P: Person_Name) return Person_Name is
   Mother: Womans_Name;
begin
   if P.Sex = Male then
      Mother := P.Wife;
   else
      Mother := P;
   end if;
   if Mother = null
              or else Mother.First_Child = null then
      return null;
   end if;
   declare
      Child: Person_Name := Mother.First_Child;
   begin
      while Child.Sex = Female loop
         if Child.Next_Sibling = null then
            return Mother.First_Child;
```

```
      end if;
      Child := Child.Next_Sibling;
    end loop;
    return Child;
  end;
end Heir;
```

2
```
procedure Divorce(W: Womans_Name) is
begin
  if W.Husband = null or W.First_Child /= null then
    return;           -- divorce not possible
  end if;
  W.Husband.Wife := null;
  W.Husband := null;
end Divorce;
```

3
```
procedure Marry(Bride: Womans_Name;
                Groom: Mans_Name) is
begin
  if Bride.Father = Groom.Father then
    raise Incest;
  end if;
  ...  -- then as before
end Marry;
```

Note that there is no need to check for marriage to a parent because the check for bigamy will detect this anyway. Our model does not allow remarriage if there are children.

4 Marry is unchanged. Note however that calls of Marry with parameters of the wrong sex will be detected at compile time whereas with variants this is detected at run time.

5
```
function Spouse(P: Person_Name)
                          return Person_Name is
begin
  if P in Man then
    return Mans_Name(P).Wife;
  else
    return Womans_Name(P).Husband;
  end if;
end Spouse;
```

This is not good because not only is there a check required in doing the membership test but also a conversion to the appropriate specific type so that the component can be selected; this conversion requires yet another check (which always passes and might be optimized away).

A better solution is to make Spouse a primitive abstract operation of Person with an access parameter

```
function Spouse(P: access Person)
               return Person_Name is abstract;
```

and then provide specific functions for each sex

```
function Spouse(P: access Man)
                      return Person_Name is
begin
  return P.Wife;
end Spouse;

function Spouse(P: access Woman)
                      return Person_Name is
begin
  return P.Husband;
end Spouse;
```

A call of Spouse will then resolve at compile time to the correct function if the parameter is of a specific type (such as Womans_Name) or will dispatch if it is class wide (Person_Name); this gives the best of all worlds.

6
```
function New_Child(Mother: Womans_Name;
                   Boy_Or_Girl: Gender;
                   Birthday: Date)
                      return Person_Name is
  Child: Person_Name;
begin
  if Mother.Husband = null then
    raise Out_Of_Wedlock;
  end if;
  case Boy_Or_Girl is
    when Male =>  Child := new Man;
    when Female =>  Child := new Woman;
  end case;
  Child.Birth := Birthday;
  ...     -- and so on as before
end New_Child;
```

This feels most uncomfortable. It seems strange to have a parameter giving the sex because we have not otherwise had to introduce the type Gender. We could pass the tag (such as Man'Tag) as a parameter T but that seems really dirty and anyway we would still have to write a conditional statement

```
  if T = Man'Tag then
    Child := new Man;
  else
    Child := new Woman;
  end if;
```

This example illustrates a common problem with polymorphism; it all works fine for output when we know what we have but it is difficult for input when we do not.

Exercise 18.6

1 If the user declared a constrained key with a nonzero discriminant thus

```
K: Key(7);
```

then he will have bypassed Get_Key and be able to call the procedure Action without authority. Note also that if he calls Return_Key then Constraint_Error will be raised on the attempt to set the code to zero because the key is constrained.

Hence forged keys can be recognized since they are constrained and so we could rewrite Valid to check for this

```
function Valid(K: Key) return Boolean is
begin
   return not K'Constrained and K.Code /= 0;
end Valid;
```

We must also insert calls of Valid into Get_Key and Return_Key.

Exercise 18.7

1 The first assignment is illegal because there is a type mismatch, the second inserts the appropriate conversion but still fails because of the dynamic accessibility check. The third is illegal because the type is limited.

Answers 19

Exercise 19.1

1
```
generic
   type Item is private;
package Stacks is
   type Stack(Max: Natural) is private;
   procedure Push(S: in out Stack; X: in Item);
   procedure Pop(S: in out Stack; X: out Item);
   function "=" (S, T: Stack) return Boolean;
private
   type Item_Array is
         array (Integer range <>) of Item;
   type Stack(Max: Natural) is
      record
         S: Item_Array(1 .. Max);
         Top: Integer := 0;
      end record;
end Stacks;
```

The body is much as before. To declare a stack we must first instantiate the package thus

```
package Boolean_Stacks is
      new Stacks(Item => Boolean);
```

and then

```
use Boolean_Stacks;
S: Stack(Max => 30);
```

2
```
generic
   type Thing is private;
package P is
   procedure Swap(A, B: in out Thing);
   procedure CAB(A, B, C: in out Thing);
end P;

package body P is
   procedure Swap(A, B: in out Thing) is
      T: Thing;
   begin
      T := A;  A := B;  B := T;
   end;

   procedure CAB(A, B, C: in out Thing) is
   begin
      Swap(A, B);  Swap(A, C);
   end;
end P;
```

Exercise 19.2

1
```
function "not" is new Next(Boolean);
```

2
```
generic
   type Number is range <>;
package Rational_Numbers is
   type Rational is private;
   function "+" (X: Rational) return Rational;
   function "–" (X: Rational) return Rational;
   function "+" (X, Y: Rational) return Rational;
   function "–" (X, Y: Rational) return Rational;
   function "*" (X, Y: Rational) return Rational;
   function "/" (X, Y: Rational) return Rational;

   subtype Positive_Number is
                     Number range 1 .. Number'Last;
   function "/" (X: Number; Y: Positive_Number)
                                    return Rational;
   function Numerator(R: Rational) return Number;
   function Denominator(R: Rational)
                             return Positive_Number;
private
   type Rational is
      record
         Num: Number := 0;
         Den: Positive_Number := 1;
      end record;
end;
```

3
```
generic
   type Index is (<>);
   type Floating is digits <>;
   type Vec is array (Index range <>) of Floating;
```

```
   type Mat is array (Index range <>,
                                 Index range <>) of Floating;
function Outer(A, B: Vec) return Mat;

function Outer(A, B: Vec) return Mat is
   C: Mat(A'Range, B'Range);
begin
   for I in A'Range loop
      for J in B'Range loop
         C(I, J) := A(I) * B(J);
      end loop;
   end loop;
   return C;
end Outer;

function Outer_Vector is
      new Outer(Integer, Float, Vector, Matrix);
```

4
```
package body Set_Of is

   function Make_Set(L: List) return Set is
      S: Set := Empty;
   begin
      for I in L'Range loop
         S(L(I)) := True;
      end loop;
      return S;
   end Make_Set;

   function Make_Set(E: Element) return Set is
      S: Set := Empty;
   begin
      S(E) := True;
      return S;
   end Make_Set;

   function Decompose(S: Set) return List is
      L: List(1 .. Size(S));
      I: Positive := 1;
   begin
      for E in Set'Range loop
         if S(E) then
            L(I) := E;  I := I + 1;
         end if;
      end loop;
      return L;
   end Decompose;

   function "+" (S, T: Set) return Set is
   begin
      return S or T;
   end "+";

   function "*" (S, T: Set) return Set is
   begin
      return S and T;
   end "*";

   function "–" (S, T: Set) return Set is
   begin
      return S xor T;
   end "–";

   function "<" (E: Element; S: Set) return Boolean is
   begin
      return S(E);
   end "<";

   function "<=" (S, T: Set) return Boolean is
   begin
      return (S and T) = S;
   end "<=";

   function Size(S: Set) return Natural is
      N: Natural := 0;
   begin
      for E in Set'Range loop
         if S(E) then
            N := N + 1;
         end if;
      end loop;
      return N;
   end Size;

end Set_Of;
```

Sadly, we cannot use renaming as bodies for "+", "*" and "–". This is because the functions become frozen at the end of the package specification (see Section 25.1) and are given convention Ada by default whereas the predefined operations "or" etc. have convention Intrinsic and so do not match. On the other hand, we could put such renamings in the private part of the package and then the new operations would take the convention of the renamed operations and so themselves be Intrinsic.

5
```
private
   type Element_Array is array (Element) of Boolean;
   type Set is
      record
         Value: Element_Array := (Element => False);
      end record;

   Empty: constant Set :=
                        (Value => (Element => False));
   Full: constant Set :=
                        (Value => (Element => True));
end;
```

We have to make the full type into a record containing the array as a single component to give it a default initial expression. Sadly, this means that the body needs rewriting and the functions become rather untidy. Also we cannot write the default expression as Empty.Value. This is because we cannot use the component name Value in its own declaration. In general, however, we can use a deferred constant as a default value before its full declaration. Note also that we have to use the named notation for the single component record aggregates.

Exercise 19.3

1 First we have to declare our function "<" which we define as follows: if the polynomials have different degrees, the one with the lower degree is smaller; if the same degree, then we compare coefficients starting at the highest power. So

```
function "<" (X, Y: Polynomial) return Boolean is
begin
   if X.N /= Y.N then
      return X.N < Y.N;
   end if;
   for I in reverse 0 .. X.N loop    -- or X.A'Range
      if X.A(I) /= Y.A(I) then
         return X.A(I) < Y.A(I);
      end if;
   end loop;
   return False;       -- they are identical
end "<";

procedure Sort_Poly is
      new Sort(Integer, Polynomial, Poly_Array);
```

2

```
type Mutant_Array is
      array (Integer range <>) of Mutant;

function "<" (X, Y: Mutant) return Boolean is
begin
   if X.Sex /= Y.Sex then
      return X.Sex > Y.Sex;
   else
      return Y.Birth < X.Birth;
   end if;
end "<";

procedure Sort_Mutant is
      new Sort(Integer, Mutant, Mutant_Array);
```

Note that the order of sexes asked for is precisely the reverse order to that in the type Gender and so we can directly use ">" applied to that type. Similarly, younger first means later birth date first and so we use the function "<" we have already defined for the type Date but with the arguments reversed.

We could not sort an array of type Person because we cannot declare such an array anyway since Person is indefinite.

3 We cannot do this because the array is of an anonymous type.

4 We might get Constraint_Error. If C'First = Index'Base'First then the attempt to evaluate Index'Pred(C'Last) will raise Constraint_Error. Considerable care can be required to make such extreme cases foolproof. The easy way out in this case is simply to insert

```
if C'Length < 2 then return; end if;
```

5 The generic body corresponds closely to the procedure Sort in Section 11.2. The type Vector is replaced by Collection. I is of type Index. The types Node and Node_Ptr are declared inside Sort because they depend on the generic type Item. The incrementing of I cannot be done with "+" since the index type may not be an integer and so we have to use Index'Succ. Care is needed not to cause Constraint_Error if the array embraces the full range of values of Index. But, the key thing is that the generic specification is completely unchanged and so we see how an alternative body can be sensibly supplied.

6

```
with Ada.Exceptions;  use Ada.Exceptions;
generic
   type Index is (<>);
   type Item is limited private;
   type Collection is array (Index range <> ) of Item;
   with function Is_It(X: Item) return Boolean;
   Ex: Exception_Id := Constraint_Error'Identity;
function Search(C: Collection) return Index;

function Search(C: Collection) return Index is
begin
   for J in C'Range loop
      if Is_It(C(J)) then return J; end if;
   end loop;
   Raise_Exception(Ex);
end Search;
```

7

```
generic
   type Item is private;
   type Vector is array (Integer range <>) of Item;
   with function "=" (X, Y: Item) return Boolean is <>;
function Equals(A, B: Vector) return Boolean;

function Equals(A, B: Vector) return Boolean is
begin
   ...   -- body exactly as for Exercise 11.4(4)
end Equals;
```

We can instantiate by

```
function "=" is new Equals(Stack, Stack_Array, "=");
```

or simply by

```
function "=" is new Equals(Stack, Stack_Array);
```

in which case the default parameter is used.

Note that it is essential to pass "=" as a parameter otherwise predefined equality would be used and the whole point is that we have redefined "=" for the type Stack.

8
```
generic
   type Floating is digits <>;
   with function F(X: Floating) return Floating;
function Solve return Floating;

function G(X: Float) return Float is
begin
   return Exp(X) + X – 7.0;
end;
...
function Solve_G is new Solve(Float, G);
...
Answer: Float := Solve_G;
```

Exercise 19.4

1
```
package body Generic_Complex_Functions is
   use Elementary_Functions;

   function Sqrt(X: Complex) return Complex is
   begin
      return Cons_Polar(Sqrt(abs X), 0.5*Arg(X)));
   end Sqrt;

   function Log(X: Complex) return Complex is
   begin
      return Cons(Log(abs X), Arg(X));
   end Log;

   function Exp(X: Complex) return Complex is
   begin
      return Cons_Polar(Exp(Rl_Part(X)), Im_Part(X));
   end Exp;

   function Sin(X: Complex) return Complex is
      Rl: Float_Type := Rl_Part(X);
      Im: Float_Type := Im_Part(X);
   begin
      return Cons(Sin(Rl)*Cosh(Im), Cos(Rl)*Sinh(Im));
   end Sin;

   function Cos(X: Complex) return Complex is
      Rl: Float_Type := Rl_Part(X);
      Im: Float_Type := Im_Part(X);
   begin
      return Cons(Cos(Rl)*Cosh(Im), –Sin(Rl)*Sinh(Im));
   end Cos;

end Generic_Complex_Functions;
```

2
```
package Poly_Vector is new General_Vector
                     (Integer, Polynomial, Poly_Array);

procedure Sort_Poly is new Sort(Poly_Vector);
```

3
```
generic
   with package Signature is new Group(<>);
   use Signature;
function Power(E: Element; N: Integer)
                                    return Element;

function Power(E: Element; N: Integer)
                                    return Element is
   Result: Element := Identity;
begin
   for I in 1 .. abs N loop
      Result := Op(Result, E);
   end loop;
   if N < 0 then Result := Inverse(Result); end if;
   return Result;
end Power;
...
package Integer_Addition_Group is
      new Group(Element => Integer, Identity => 0,
                  Op => "+", Inverse => "–");

function Multiply is
      new Power(Integer_Addition_Group);
```

4
```
generic
   type Element is (<>);
   Identity: in Element;
   with function Op(X, Y: Element) return Element;
   with function Inverse(X: Element) return Element;
package Finite_Group is end;

generic
   with package Signature is new Finite_Group(<>);
   use Signature;
function Is_Group return Boolean;

function Is_Group return Boolean is
begin
   –– check the operation is closed,
   –– the actual parameter could be constrained
   for E in Element'Range loop
      for F in Element'Range loop
         declare
            Result: Element;
         begin
            Result := Op(E, F);
         exception
            when Constraint_Error =>
               return False;
         end;
      end loop;
   end loop;

   –– check identity is OK
   for E in Element'Range loop
      if Op(E, Identity) /= E or
                           Op(Identity, E) /= E then
         return False;
      end if;
   end loop;
```

```
   -- check inverse is OK
   for E in Element'Range loop
      if Op(E, Inverse(E)) /= Identity or
                  Op(Inverse(E), E) /= Identity then
         return False;
      end if;
   end loop;

   -- check associative law OK
   for E in Element'Range loop
      for F in Element'Range loop
         for G in Element'Range loop
            if Op(E, Op(F, G)) /= Op(Op(E, F), G) then
               return False;
            end if;
         end loop;
      end loop;
   end loop;
   return True;
end Is_Group;
```

Exercise 19.5

1
```
generic
   type Floating is digits <>;
package Generic_Complex_Numbers is
   ...
end Generic_Complex_Numbers;

generic
package Generic_Complex_Numbers.Cartesian is
   ...
end Generic_Complex_Numbers.Cartesian;

generic
package Generic_Complex_Numbers.Polar is
   ...
end Generic_Complex_Numbers.Polar;

with Ada.Numerics.Generic_Elementary_Functions;
use Ada.Numerics;
with Generic_Complex_Numbers;
with Generic_Complex_Numbers.Cartesian;
with Generic_Complex_Numbers.Polar;
generic
   with package Elementary_Functions is
         new Generic_Elementary_Functions(<>);
   with package Complex_Numbers is
         new Generic_Complex_Numbers
               (Elementary_Functions.Float_Type);
   with package Cartesian is
         new Complex_Numbers.Cartesian;
   with package Polar is
         new Complex_Numbers.Polar;
package Generic_Complex_Functions is
   use Complex_Numbers, Cartesian, Polar;

   function Sqrt(X: Complex) return Complex;
   ...
end Generic_Complex_Functions;
```

Note that the generic formals ensure that the packages passed as actuals are correctly related. For example if we did two instantiations of the hierarchy (one for Float and one for Long_Float) then we need to ensure that we do not use the parent from one with a child from the other. However, there is no such guarantee if the hierarchy has a generic subprogram as a child since we can only express the requirement that the profile is correct; this will be enough in most cases but is not foolproof. Of course the program would not crash, just do something silly.

Answers 20

Exercise 20.1

1
```
procedure Shopping is
   task Get_Salad;

   task body Get_Salad is
   begin
      Buy_Salad;
   end Get_Salad;

   task Get_Wine;

   task body Get_Wine is
   begin
      Buy_Wine;
   end Get_Wine;

   task Get_Meat;

   task body Get_Meat is
   begin
      Buy_Meat;
   end Get_Meat;

begin
   null;
end Shopping;
```

Exercise 20.2

1
```
task body Char_To_Line is
   Buffer: Line;
begin
   loop
      for I in Buffer'Range loop
         accept Put(C: in Character) do
            Buffer(I) := C;
         end;
      end loop;
      accept Get(L: out Line) do
```

```
      L := Buffer;
    end;
  end loop;
end Char_To_Line;
```

Exercise 20.3

1
```
with Calendar;
generic
  First_Time: Calendar.Time;
  Interval: Duration;
  Number: Integer;
  with procedure P;
procedure Call;

procedure Call is
  use type Calendar.Time;
  Next_Time: Calendar.Time := First_Time;
  Now: Calendar.Time := Calendar.Clock;
begin
  if Next_Time < Now then
    Next_Time := Now;
  end if;
  for I in 1 .. Number loop
    delay until Next_Time;
    P;
    Next_Time := Next_Time + Interval;
  end loop;
end Call;
```

2 The trouble with writing

```
  delay Next_Time – Clock;
```

is that the task might be temporarily suspended between calling Clock and issuing the delay; the delay would then be wrong by the amount of time for which the task did not have a processor.

3
```
use Calendar;
Date: Time;
Y: Year_Number;
M: Month_Number;
D: Day_Number;
S: Day_Duration;
High_Noon: Time;
...
Split(Clock, Y, M, D, S);
High_Noon := Time_Of(Y, M, D, 43_200.0);
if S > 43_200.0 then      –– afternoon, so add a day
  High_Noon := High_Noon + 86_400.0;
end if;
```

Exercise 20.4

1
```
protected Variable is
  entry Read(Value: out Item);
  procedure Write(New_Value: in Item);
private
  Data: Item;
  Value_Set: Boolean := False;
end Variable;

protected body Variable is
  entry Read(Value: out Item) when Value_Set is
  begin
    Value := Data;
  end Read;

  procedure Write(New_Value: in Item) is
  begin
    Data := New_Value;
    Value_Set := True;          –– clear the barrier
  end Write;
end Variable;
```

The problem with this solution is that it no longer allows multiple readers because the function has been replaced by an entry.

2 This is a bit of a trick question. It cannot be done because a discriminant is not static and therefore cannot be used to declare the type Index. One possible alternative is to make the index type of the array and the type of the variables In_Ptr and Out_Ptr to be type Integer and to use the **mod** operator to do the cyclic arithmetic. The structure would then be

```
generic
  type Item is private;
package Buffers is
  type Item_Array is array (Integer range <>) of Item;
  protected type Buffering(N: Integer) is
    –– and so on
  end Buffering;
end Buffers;
```

3
```
protected Buffer is
  entry Put(X: in Item);
  entry Get(X: out Item);
private
  V: Item;
  Is_Set: Boolean := False;
end;

protected body Buffer is
  entry Put(X: in Item) when not Is_Set is
  begin
    V := X;
    Is_Set := True;
  end Put;

  entry Get(X: out Item) when Is_Set is
  begin
    X := V;
    Is_Set := False;
  end Get;
end Buffer;
```

4
```
protected Char_To_Line is
   entry Put(C: in Character);
   entry Get(L: out Line);
private
   Buffer: Line;
   Count: Integer := 0;  -- number of items in buffer
end;

protected body Char_To_Line is
   entry Put(C: in Character) when
                               Count < Buffer'Last is
   begin
      Count := Count + 1;
      Buffer(Count) := C;
   end Put;

   entry Get(L: out Line) when Count = Buffer'Last is
   begin
      Count := 0;
      L := Buffer;
   end Get;
end Char_To_Line;
```

Exercise 20.7

1
```
protected type Mailbox is
   entry Deposit(X: in Item);
   entry Collect(X: out Item);
private
   Full: Boolean := False;
   Local: Item;
end;

protected body Mailbox is

   entry Deposit(X: in Item) when not Full is
   begin
      Local := X;
      Full := True;
   end Deposit;

   entry Collect(X: out item) when Full is
   begin
      X := Local;
      Full := False;
   end Collect;

end Mailbox;
```

This mailbox is reusable whereas the task version was not (indeed the task just terminated after use). We could prevent the protected object from being reused by further state variables.

The advantages of the protected object are that there need be no concern with termination in the event of it not being used and it is of course much more efficient. A possible disadvantage is that the closely coupled form is not possible.

Exercise 20.8

1
```
task type Buffering is
   entry Put(X: in Item);
   entry Finish;
   entry Get(X: out Item);
end;

task body Buffering is
   N: constant := 8;
   type Index is mod N;
   A: array (Index) of Item;
   In_Ptr, Out_Ptr: Index := 0;
   Count: Integer range 0 .. N := 0;
   Finished: Boolean := False;
begin
   loop
      select
         when Count < N =>
         accept Put(X: in Item) do
            A(In_Ptr) := X;
         end;
         In_Ptr := In_Ptr + 1;  Count := Count + 1;
      or
         accept Finish;
         Finished := True;
      or
         when Count > 0 =>
         accept Get(X: out Item) do
            X := A(Out_Ptr);
         end;
         Out_Ptr := Out_Ptr + 1;  Count := Count – 1;
      or
         when Count = 0 and Finished =>
         accept Get(X: out Item) do
            raise Done;
         end;
      end select;
   end loop;
exception
   when Done =>
      null;
end Buffering;
```

This curious example illustrates that there may be several accept statements for the same entry in the one select statement. The exception Done is propagated to the caller and also terminates the loop in Buffering before being quietly handled. Of course the exception need not be handled by Buffering because exceptions propagated out of tasks are lost, but it is cleaner to do so.

2 The server aborts the caller during the rendezvous thereby placing the caller into an abnormal state. Although the caller cannot be

properly completed until after the rendezvous is finished (the server might have access to the caller's data space via a parameter), nevertheless the caller is no longer active and does not receive the exception Havoc.

3
```
select
   Trigger.Wait;
then abort
   loop
      ...   -- compute next estimate in Z
      -- then store it in the protected object
      Result.Put_Estimate(Z);
      -- loop back to improve estimate
   end loop;
end select;
```

Exercise 20.9

1 If we wrote

```
   requeue Reset with abort;
```

then there would be a risk that the task that called Signal was aborted before it could clear the occurred flag. The system would then be in a mess since subsequent tasks calling Wait could proceed without waiting for the next signal.

2
```
protected Event is
   entry Wait;
   procedure Signal;
private
   Occurred: Boolean := False;
end Event;

protected body Event is
   entry Wait when Occurred is
   begin
      if Wait'Count = 0 then
         Occurred := False;
      end if;
   end Wait;

   procedure Signal is
   begin
      if Wait'Count > 0 then
         Occurred := True;
      end if;
   end Signal;
end Event;
```

The last of the waiting tasks to be let go clears the occurred flag back to false (the last one out switches off the light). It is important that the procedure Signal does not set the occurred flag if there are no tasks waiting since in such a case there is no waiting task to clear it and the signal would persist (remember this is a model of a transient signal).

An amazing alternative solution is

```
protected Event is
   entry Wait;
   entry Signal;
end Event;

protected body Event is
   entry Wait when Signal'Count > 0 is
   begin
      null;
   end Wait;

   entry Signal when Wait'Count = 0 is
   begin
      null;
   end Signal;
end Event;
```

This works because joining an entry queue is a protected action and results in the evaluation of barriers (just as they are evaluated when a protected procedure or entry body finishes). Note that there is no protected data (and hence no private part) and that both entry bodies are null; in essence the protected data is the Count attributes and these therefore behave properly. In contrast, the Count attributes of task entries are not reliable because joining and leaving task entry queues are not protected in any way.

3
```
task Controller is
   entry Sign_In(P: Priority; D: Data);
private
   entry Request(Priority) (P: Priority; D: Data);
end;

task body Controller is
   Total: Integer := 0;
begin
   loop
      if Total = 0 then
         accept Sign_In(P: Priority; D: Data) do
            Total := 1;
            requeue Request(P);
         end;
      end if;
      loop
         select
            accept Sign_In(P: Priority; D: Data) do
               Total := Total + 1;
               requeue Request(P);
            end;
         else
            exit;
         end select;
      end loop;
```

```
      for P in Priority loop
         select
            accept Request(P) (P: Priority; D: Data) do
               Action(D);
            end;
            Total := Total – 1;
            exit;
         else
            null;
         end select;
      end loop;
   end loop;
end Controller;
```

The variable Total records the total number of requests outstanding. Each time round the outer loop, the task waits for a call of Sign_In if no requests are in the system, it then services any outstanding calls of Sign_In. The calls to Sign_In requeue onto the appropriate member of the entry family Request. The task then deals with a request of the highest priority. Observe that we had to make P a parameter of the entry family as well as the index; this is because requeue can only be to an entry with the same parameter profile (or parameterless).

Note that the solution works if a calling task is aborted; this is because the requeue does not specify **with abort** and so is considered as part of the abort deferred region of the original rendezvous.

4
```
package Monitor is
   protected Call is
      entry Job(D: Data);
   end;
private
   task The_Task is
      entry Job(D: Data);
   end;
end;

package body Monitor is
   protected body Call is
      entry Job(D: Data) when True is
      begin
         Log_The_Call(Calendar.Clock);
         requeue The_Task.Job;
      end Job;
   end Call;
   ...
end Monitor;
```

Exercise 20.10

1
```
package Cobblers is
   procedure Mend(A: Address; B: Boots);
end;

package body Cobblers is
   type Job is
      record
         Reply: Address;
         Item: Boots;
      end record;

   package P is new Buffers(Job);
   use P;
   Boot_Store: Buffer(100);

   task Server is
      entry Request(A: Address; B: Boots);
   end;

   task type Repairman;
   Tom, Dick, Harry: Repairman;

   task body Server is
      Next_Job: Job;
   begin
      loop
         accept Request(A: Address; B: Boots) do
            Next_Job := (A, B);
         end;
         Put(Boot_Store, Next_Job);
      end loop;
   end Server;

   task body Repairman is
      My_Job: Job;
   begin
      loop
         Get(Boot_Store, My_Job);
         Repair(My_Job.Item);
         My_Job.Reply.Deposit(My_Job.Item);
      end loop;
   end Repairman;

   procedure Mend(A: Address; B: Boots) is
   begin
      Server.Request(A, B);
   end;
end Cobblers;
```

We have assumed that the type Address is an access to a mailbox for handling boots. Note one anomaly; the server accepts boots from the customer before checking the store – if it turns out to be full, he is left holding them. In all, the shop can hold 104 pairs of boots – 100 in store, 1 with the server and 1 with each repairman.

Answers 21

Exercise 21.1

1
```
function Volume(C: Cylinder) return Float is
begin
   return Area(C.Base) * C.Height;
end Volume;

function Area(C: Cylinder) return Float is
begin
   return 2.0*Area(C.Base) +
                    2.0*Pi*C.Base.Radius*C.Height;
end Area;
```

We cannot apply the function Moment to a cylinder because the type Cylinder is not in Object'Class. We are thus protected from such foolishness.

Exercise 21.3

1
```
with Ada.Finalization;  use Ada;
generic
   type Raw_Type is tagged private;
package Tracking is
   type Tracked_Type is new Raw_Type with private;
   function Identity(TT: Tracked_Type) return Integer;
private
   type Control is new Finalization.Controlled with
      record
         Identity_Number: Integer;
      end record;

   procedure Initialize(C: in out Control);
   procedure Adjust(C: in out Control);
   procedure Finalize(C: in out Control);

   type Tracked_Type is new Raw_Type with
      record
         Component: Control;
      end record;
end Tracking;

package body Tracking is
   The_Count: Integer := 0;
   Next_One: Integer := 1;

   function Identity(TT: Tracked_Type)
                                    return Integer is
   begin
      return TT.Component.Identity_Number;
   end Identity;

   procedure Initialize(C: in out Control) is
   begin
      The_Count := The_Count + 1;
      C.Identity_Number := Next_One;
      Next_One := Next_One + 1;
   end Initialize;

   procedure Adjust ...
   procedure Finalize ...
end Tracking;
```

2
```
with Objects;  use Objects;
with Tracking;
package Hush_Hush is
   type Secret_Shape is new Object with private;
   function Shape_Identity(SS: Secret_Shape)
                                    return Integer;
   ...
private
   package Q is
         new Tracking(Raw_Type => Object);
   type Secret_Shape is new Q.Tracked_Type with
      record
         ...   -- other hidden components
      end record;
end Hush_Hush;

package body Hush_Hush is
   function Shape_Identity(SS: Secret_Shape)
                                    return Integer is
   begin
      return Identity(SS);
   end Shape_Identity;
   ...
end Hush_Hush;
```

Note carefully that the type Secret_Shape inherits the function Identity from Q.Tracked_Type. Of course we could have laboriously written

```
   return Q.Identity(Q.Tracked_Type(SS));
```

However, we do not actually have to write out a body for Shape_Identity but can simply use a renaming thus

```
function Shape_Identity(SS: Secret_Shape)
                  return Integer renames Identity;
```

Exercise 21.4

1
```
package body Lists is
   procedure Insert(After: Cell_Ptr; Item: Cell_Ptr) is
   begin
      if Item = null or else Item.Next /= null then
         raise List_Error;
      end if;
      if After = null then
         raise List_Error;
      end if;
      Item.Next := After.Next;
      After.Next := Item;
   end Insert;
```

```
  function Remove(After: Cell_Ptr) return Cell_Ptr is
     Result: Cell_Ptr;
  begin
     if After = null then
        raise List_Error;
     end if;
     Result := After.Next;
     if Result /= null then
        After.Next := Result.Next;
        Result.Next := null;
     end if;
     return Result;
  end Remove;

  function Next(After: Cell_Ptr) return Cell_Ptr is
  begin
     if After = null then
        raise List_Error;
     end if;
     return After.Next;
  end Next;
end Lists;
```

Note that we do not have to do anything about a dummy first element. The user has to do that by declaring a list with one cell already in place by for example

```
The_List: Cell_Ptr := new Cell;
```

2 Using null exclusions enables most of the checks to be omitted. The subprograms become

```
  procedure Insert(After: not null Cell_Ptr;
                   Item: not null Cell_Ptr) is
  begin
     if Item.Next /= null then
        raise List_Error;
     end if;
     Item.Next := After.Next;
     After.Next := Item;
  end Insert;

  function Remove(After: not null Cell_Ptr)
                                  return Cell_Ptr is
  begin
     return Result: Cell_Ptr := After.Next do
        if Result /= null then
           After.Next := Result.Next;
           Result.Next := null;
        end if;
     end return;
  end Remove;

  function Next(After: not null Cell_Ptr)
                                  return Cell_Ptr is
  begin
     return After.Next;
  end Next;
```

Exercise 21.5

1

```
package List_Iteration_Stuff is
      new Iteration_Stuff(Lists.List,
                          Lists.Iterators.Iterator);

procedure Green_To_Red is
      new Generic_Green_To_Red(I_S =>
                          List_Iteration_Stuff);
```

2

```
package Iterators is
   type Structure is interface;
   procedure Iterate(S: in Structure;
      Action: access procedure (C: in out Colour));
end;

package Trees is
   type Tree is new Structure with private;
   ...
   procedure Iterate(T: in Tree;
      Action: access procedure (C: in out Colour));
private
   type Node;
   type Node_Ptr is access Node;

   type Node is
      record
         Left, Right: Node_Ptr;
         C: Colour;
      end record;
   type Tree is new Structure with
      record
         Root: Node_Ptr;
      end record;
end;

package body Trees is
   ...
   procedure Iterate(T: in Tree;
    Action: access procedure (C: in out Colour)) is
      procedure Inner(N: in Node_Ptr) is
      begin
         if N /= null then
            Action(N.C);            -- indirect call
            Inner(N.Left);
            Inner(N.Right);
         end if;
      end Inner;
   begin
      Inner(T.Root);
   end Iterate;

end Trees;
```

Note that Action is an access to procedure parameter of the primitive procedure Iterate of the interface Structure. This is called from within the body of Iterate. We then have

```
function Count(S: Structure'Class; C: Colour)
                                    return Natural is
   Result: Natural := 0;

   procedure Action(C: in out Colour) is
   begin
      if C = Count.C then
         Result := Result + 1;
      end if;
   end Action;

begin
   Iterate(S, Action'Access);   -- dispatch on S
   return Result;
end Count;

Oak: Tree;                      -- declare some tree
...                             -- build the tree
N := Count(Oak, Green);
```

Note that this has a mixture of dispatching and access to subprogram calls.

Exercise 21.9

1 The tagged type case is easy; we simply write

```
package People is
   type Person_Name(<>) is private;
   type Mans_Name (<>) is private;
   type Womans_Name(<>) is private;

   function Man_Of(P: Person_Name)
                              return Mans_Name;

   function Woman_Of(P: Person_Name)
                              return Womans_Name;
   function Person_Of(M: Mans_Name)
                              return Person_Name;
   function Person_Of(W: Womans_Name)
                              return Person_Name;
   ...   -- other subprograms
private
   type Person;
   ...
   type Person_Name is access all Person'Class;
   type Mans_Name is access all Man;
   ...
   type Person is abstract tagged ...
   ...
end People;
```

where the private part is exactly as before. The conversion functions are simply

```
function Man_Of(P: Person_Name)
                              return Mans_Name is
begin
   Mans_Name(P);
end;
```

and so on. Remember that conversion is allowed between general access types referring to derived types in the same class. Constraint_Error is raised if we attempt to convert a person to a name of the wrong sex. Conversion in the opposite direction (towards the root) always works.

The variant formulation requires more care. We cannot write, in the private part, something like

```
type Person_Name is access Person;
type Mans_Name is Person_Name(Male);
```

because the full type always has to be a new type. But we can write

```
type Person_Name is access all Person;
type Mans_Name is access all Person(Male);
type Womans_Name is access all Person(Female);
```

where we have again used general access types so that that we can convert between them.

Answers 22

Exercxise 22.2

1

```
procedure Gauss_Seidel is
   N: constant := 5;
   subtype Full_Grid is Integer range 0 .. N;
   subtype Grid is Full_Grid range 1 .. N–1;
   type Real is digits 7;
   Tolerance: constant Real := 0.0001;
   Error_Limit: constant Real := Tolerance * (N–1)**2;
   Converged: Boolean := False;
   Error_Sum: Real;

   function F(I, J: Grid) return Real is separate;

   task type Iterator is
      entry Start(I, J: in Grid);
   end;

   protected type Point is
      procedure Set_P(X: in Real);
      function Get_P return Real;
      function Get_Delta_P return Real;
      procedure Set_Converged(B: in Boolean);
      function Get_Converged return Boolean;
   private
      Converged: Boolean := False;
      P: Real;
      Delta_P: Real;
   end;

   Process: array (Grid, Grid) of Iterator;
   Data: array (Full_Grid, Full_Grid) of Point;
```

```
  task body Iterator is
    I, J: Grid;
    P: Real;
  begin
    accept Start(I, J: in Grid) do
      Iterator.I := Start.I;
      Iterator.J := Start.J;
    end Start;

    loop
      P := 0.25 * (Data(I–1, J).Get_P +
                   Data(I+1, J).Get_P +
                   Data(I, J–1).Get_P +
                   Data(I, J+1).Get_P – F(I, J));
      Data(I, J).Set_P(P);
      exit when Data(I, J).Get_Converged;
    end loop;
  end Iterator;

  protected body Point is
    procedure Set_P(X: in Real) is
    begin
      Delta_P := X – P;
      P := X;
    end;

    function Get_P return Real is
    begin
      return P;
    end;

    function Get_Delta_P return Real is
    begin
      return Delta_P;
    end;

    procedure Set_Converged(B: in Boolean) is
    begin
      Converged := B;
    end;

    function Get_Converged return Boolean is
    begin
      return Converged;
    end;
  end Point;

begin    –– of main subprogram; tasks now active

  for I in Grid loop
    for J in Grid loop   –– tell them who they are
      Process(I, J).Start(I, J);
    end loop;
  end loop;

  loop
    Error_Sum := 0.0;
    for I in Grid loop
      for J in Grid loop
        Error_Sum := Error_Sum +
                     Data(I, J).Get_Delta_P**2;
      end loop;
    end loop;

    Converged := Error_Sum < Error_Limit;
    exit when Converged;
  end loop;

  –– tell protected objects that system has converged

  for I in Grid loop
    for J in Grid loop
      Data(I, J).Set_Converged(True);
    end loop;
  end loop;

  –– output results

end Gauss_Seidel;
```

Note that there are protected objects on the boundary points but we have not shown how to initialize them. The central computation could be made neater by using renaming in order to avoid repeated evaluation of Data(I, J) and so on; this would also speed things up. Thus we could write

```
function Get_P1 return Real
                        renames Data(I–1, J).Get_P;
function Get_P2 return Real
                        renames Data(I+1, J).Get_P;
...
procedure Set_P(X: in Real)
                          renames Data(I, J).Set_P;
function Get_Converged return Boolean
                   renames Data(I, J).Get_Converged;
```

and then

```
Set_P(0.25 *
     (Get_P1+Get_P2+Get_P3+Get_P4 – F(I, J)));
exit when Get_Converged;
```

Exercise 22.3

1

```
protected A_Map is new Map with
  procedure Insert(K: in Key; V: in Value);
  procedure Find(K: in Key; V: out Value);
private
  ...
end A_Map;
```

Exercise 22.4

1
```
procedure Handle(CD: access Cannon_Data;
                 E: in Exception_Occurrence) is
begin
   if Exception_Identity(E) = Bang'Identity then
      Put_Line("Cannon seems to have exploded.");
      Put("Perhaps ");
      Put(CD.Pounds_Of_Powder);
      Put(" pounds of powder was too much!");
   else
      Put("Some other catastrophe ...");
   end if;
end Handle;
```

2 The root package might be

```
package Root_Activity is
   type Root_Descriptor is abstract tagged private;

   function No_Of_Cycles return Integer;
   function No_Of_Cycles(D: access
            Root_Descriptor'Class) return Integer;
private
   Total_Count: Integer := 0;

   type Root_Descriptor is tagged
      record
         Instance_Count: Integer := 0;
      end record;
end;

package body Root_Activity is

   function No_Of_Cycles return Integer is
   begin
      return Total_Count;
   end;

   function No_Of_Cycles(D: access
            Root_Descriptor'Class) return Integer is
   begin
      return D.Instance_Count;
   end;

end Root_Activity;
```

The type Descriptor and associated operations are now placed in the child package together with the task type Control whose body is modified to update the counts on each cycle. Remember that the body of a child package can see the private part of its parent. So we have

```
package Root_Activity.Cyclic is
   type Descriptor is new Root_Descriptor with ...
   ...
   task type Control(Activity: access Descriptor'Class);
end;

package body Root_Activity.Cyclic is
   ...
   task body Control is
      Next_Time: Calendar.Time := Activity.Start_Time;
   begin
      loop
         Total_Count := Total_Count + 1;
         Activity.Instance_Count :=
                     Activity.Instance_Count + 1;
         delay until Next_Time;
         ...
   end Control;
end Root_Activity.Cyclic;
```

3 It could be rewritten as follows. Type extension is necessary to pass the additional data.

```
task type Control(Activity: access Descriptor'Class);

task body Control is
   Next_Time: Calendar.Time := Activity.Start_Time;
begin
   loop
      delay until Next_Time;
      Activity.Action(Activity);          -- indirect call
      Next_Time := Next_Time + Activity.Interval;
      exit when Next_Time > Activity.End_Time;
   end loop;
   Activity.Last_Wishes(Activity);        -- indirect call
exception
   when Event: others =>
      Activity.Handle(Activity, Event);  -- indirect call
end Control;

package Root_Activity is
   type Descriptor is tagged;
   type Action_Type is access
        procedure (D: access Descriptor'Class);
   type Last_Wishes_Type is access
        procedure (D: access Descriptor'Class);
   type Handle_Type is access
        procedure (D: access Descriptor'Class;
                   E: in Exception_Occurrence);

   procedure Null_Last_Wishes
             (D: access Descriptor'Class) is null;
   procedure Default_Handle
             (D: access Descriptor'Class;
              E: in Exception_Occurrence);

   type Descriptor is tagged
      record
         Start_Time, End_Time: Calendar.Time;
         Interval: Duration;
         Action: Action_Type;
         Last_Wishes: Last_Wishes_Type :=
                        Null_Last_Wishes'Access;
         Handle: Handle_Type :=
                        Default_Handle'Access;
      end record;
end;
```

```
package body Root_Activity is

   procedure Default_Handle
                  (D: access Descriptor'Class;
                   E: in Exception_Occurrence) is
   begin
      Put_Line("Unhandled exception");
      Put_Line(Exception_Information(E));
   end Default_Handle;

end Root_Activity;

use Root_Activity;

type Cannon_Data is new Descriptor with
   record
      Pounds_Of_Powder: Integer;
   end record;

procedure Cannon_Action
                  (D: access Descriptor'Class) is
begin
    Load_Cannon(Cannon_Data(D.all).
                              Pounds_Of_Powder);
    Fire_Cannon;
end Cannon_Action;

The_Data: aliased Cannon_Data :=
      (Start_Time => High_Noon;
       End_Time => When_The_Stars_Fade_And_Fall;
       Interval => 24*Hours;
       Action => Cannon_Action'Access;
       ...
       Pounds_Of_Powder => 100);

Cannon_Task: Control(The_Data'Access);
```

So here is a deep point. The access problems are overcome by type extension itself and not by the dispatching. However, the dispatching approach is neater because the default subprograms are automatically inherited and do not clutter the record.

Exercise 22.5

1

```
package Start_Up
      with Elaborate_Body is
end;

with Ada.Task_Termination;
use Ada.Task_Termination;
package body Start_Up is
begin
   Set_Dependents_Fallback_Handler
                              (RIP.One'Access);
end Start_Up;

with Start_Up;  pragma Elaborate(Start_Up);
package Library_Tasks is
   ...      -- declare library tasks here
end;
```

Exercise 22.8

1 Good luck with the contemplation. The website might have a solution.

Exercise 22.9

1 Presumably the optimistic programmer writes

```
pragma Conflict_Check_Policy(No_Conflict_Checks);
```

and the pessimistic programmer writes

```
pragma Conflict_Check_Policy(All_Conflict_Checks);
```

Answers 23

Exercise 23.3

1 Index(S, Decimal_Digit_Set **or** To_Set('.'))

2 "begins" which seems to be the longest word in English with the letters in alphabetical order. Other Ada words, "abort" and "first", are good runners-up.

3

```
function Make_Map(K: String)
                        return Character_Mapping is
   Key_Set, Non_Key_Set: Character_Set;
   In_letters, Out_Letters: String(1 .. 26);
begin
   Key_Set := To_Set(To_Upper(K));
   Non_Key_Set := To_Set(('A', 'Z')) – Key_Set;
   In_Letters := To_Sequence(Key_Set) &
                        To_Sequence(Non_Key_Set);
   Out_Letters := To_Sequence(To_Set(('A', 'Z')));
   return To_Mapping
            (In_Letters & To_Lower(In_Letters),
             Out_Letters & To_Lower(Out_Letters));
end Make_Map;
...
Translate(S, Make_Map("Byron"));
```

4

```
function Decode(M: Character_Mapping)
                        return Character_Mapping is
begin
   return To_Mapping(To_Range(M), To_Domain(M));
end Decode;
...
Translate(S, Decode(Make_Map("Byron")));
```

The functions To_Domain and To_Range produce the domain and range of the original mapping and the reverse map is simply created by calling To_Mapping with them reversed. Note that To_Mapping raises Translation_Error anyway if the first argument has duplicates and so no additional check is required.

Exercise 23.4

1
```
with Ada.Numerics.Elementary_Functions;
use Ada.Numerics;
package body Simple_Maths is

   function Sqrt(F: Float) return Float is
   begin
      return Elementary_Functions.Sqrt(F);
   exception
      when Argument_Error =>
         raise Constraint_Error;
   end Sqrt;

   function Log(F: Float) return Float is
   begin
      return Elementary_Functions.Log(F, 10.0);
   exception
      when Argument_Error =>
         raise Constraint_Error;
   end Log;

   function Ln(F: Float) return Float is
   begin
      return Elementary_Functions.Log(F);
   exception
      when Argument_Error =>
         raise Constraint_Error;
   end Ln;

   function Exp(F: Float) return Float
         renames Elementary_Functions.Exp;

   function Sin(F: Float) return Float
         renames Elementary_Functions.Sin;

   function Cos(F: Float) return Float
         renames Elementary_Functions.Cos;

end Simple_Maths;
```

We did not write a use clause for Elementary_ Functions because it would not have enabled us to write for example **return** Sqrt(F); since this would have resulted in an infinite recursion.

2
```
type Hand is (Paper, Stone, Scissors);
type Jacks_Hand is new Hand;
type Jills_Hand is new Hand;
type Outcome is (Jack, Draw, Jill);
Payoff: array (Jacks_Hand, Jills_Hand) of Outcome :=
      ((Draw, Jack, Jill),
       (Jill, Draw, Jack),
       (Jack, Jill, Draw));
package Random_Hand is
      new Discrete_Random(Hand);
use Random_Hand;
Jacks_Gen: Generator;
Jills_Gen: Generator;
Result: Outcome;
...
Reset(Jacks_Gen);  Reset(Jills_Gen);
loop
   Result := Payoff(Jacks_Hand(Random(Jacks_Gen)),
                    Jills_Hand(Random(Jills_Gen)));
   case Result is
      when Jack =>
         ...
      when Draw =>
         ...
      when Jill =>
         ...
   end case;
end loop;
```

The types Jacks_Hand and Jills_Hand are introduced simply so that the array Payoff cannot be indexed incorrectly. There are clearly lots of different ways of doing this example. A more object oriented approach might be to declare a type Player containing the personal generator and perhaps the player's score.

Exercise 23.5

1
```
with Ada.Direct_IO;
generic
   type Element is private;
procedure Rev(From, To: in String);

procedure Rev(From, To: in String) is
   package IO is new Ada.Direct_IO(Element);
   use IO;
   Input: File_Type;
   Output: File_Type;
   X: Element;
begin
   Open(Input, In_File, From);
   Open(Output, Out_File, To);
   Set_Index(Output, Size(Input));
   loop
      Read(Input, X);
      Write(Output, X);
      exit when End_Of_File(Input);
      Set_Index(Output, Index(Output)–2);
   end loop;
   Close(Input);
   Close(Output);
end Rev;
```

Remember that **reverse** is a reserved word. Note also that this does not work if the file is empty (the first call of Set_Index will raise Constraint_Error).

Exercise 23.6

1 The output is shown in string quotes in order to reveal the layout. Spaces are indicated by s. In reality of course, there are no quotes and spaces are spaces.

(a)	"Fred"	(f)	"ss8#170#"
(b)	"sss120"	(g)	"–3.80000E+01"
(c)	"sssss120"	(h)	"sssss7.00E–2"
(d)	"120"	(i)	"3.1416E+01"
(e)	"–120"	(j)	"1.0E+10"

2
```
with Ada.Text_IO;
with Ada.Float_Text_IO;
use Ada;
package body Simple_IO is

   procedure Get(F: out Float) is
   begin
      Float_Text_IO.Get(F);
   end Get;

   procedure Put(F: in Float) is
   begin
      Float_Text_IO.Put(F);
   end Put;

   procedure Put(S: in String)
                              renames Text_IO.Put;

   procedure New_Line(N: in Integer := 1) is
   begin
      Text_IO.New_Line(Text_IO.Count(N));
   end New_Line;

end Simple_IO;
```

We have used the nongeneric package Ada.Float_Text_IO. We have to use the full dotted notation to avoid recursion. We cannot use renaming for Get and Put for the type Float because those in Float_Text_IO have additional parameters (which have defaults).

The other point of note is the type conversion in New_Line.

Exercise 23.7

1
```
procedure Date_Read(Stream: not null access
                            Root_Stream_Type'Class;
                            Item: out Date) is
   Month_Number: Integer range 1 .. 12;
begin
   Integer'Read(Stream, Item.Day);
   Integer'Read(Stream, Month_Number;
   Item.Month := Month_Name'Val(Month_Number – 1);
   Integer'Read(Stream, Item.Year);
end Date_Read;
...
for Date'Read use Date_Read;
```

Answers 24

Exercise 24.2

1
```
private with Ada.Containers.Doubly_Linked_Lists;
package Queues is
   Empty: exception;
   type Queue is limited private;
   procedure Join(Q: in out Queue; X: in Item);
   procedure Remove(Q: in out Queue; X: out Item);
   function Length(Q: Queue) return Integer;
private
   use Ada.Containers;
   package Q_Container is
                  new Doubly_Linked_Lists(Item);
   type Queue is
            new Q_Container.List with null record;
end;

package body Queues is

   procedure Join(Q: in out Queue; X: in Item) is
   begin
      Append(Q, Item);
   end Join;

   procedure Remove(Q: in out Queue; X: out Item) is
   begin
      if Is_Empty(Q) then
         raise Empty;
      end if;
      X := First_Element(Q);
      Delete_First(Q);
   end Remove;

   function Length(Q: Queue) return Integer is
   begin
      return Integer(Count_Type'(Q.Length));
   end Length;

end Queues;
```

We have used a private with clause since there is no need for access to the container in the visible part of the package.

The type Queue is not visibly tagged but is visibly limited. The fact that the full type is tagged but not limited does not matter.

The function Length which results from the instantiation and type derivation might appear to clash with the function Length that we have to provide. But the new one has result of type Count_Type (which is declared in Ada.Containers). Thus the call of Q.Length can be qualified to select the correct one and the result is then converted.

Exercise 24.3

1
```
private with Ada.Containers.Vectors;
package Queues is
   Empty: exception;
   type Queue is limited private;
   procedure Join(Q: in out Queue; X: in Item);
   procedure Remove(Q: in out Queue; X: out Item);
   function Length(Q: Queue) return Integer;
private
   use Ada.Containers;
   package Q_Container is new Vectors(Item);
   use Q_Container;
   type Queue is new Vector with null record;
end;
```

The body remains unchanged. But it will be terribly slow because each call of Remove will result in the vector sliding.

2
```
with Ada.Containers.Vectors;
with Ada.Containers.Doubly_Linked_Lists;
use Ada.Containers;
generic
   with package DLL is
                        new Doubly_Linked_Lists(<>);
   with package V is new Vectors(
            Element_Type => DLL.Element_Type;
            others => <>);
function Convert(The_Vector: V.Vector)
                                  return DLL.List;

function Convert(The_Vector: V.Vector)
                                  return DLL.List is
   The_List: DLL.List;
begin;
   for Ind in The_Vector.First_Index ..
                    The_Vector.Last_Index loop
      The_List.Append(The_Vector.Element(Ind));
   end loop;
   return The_List;
end Convert;
```

We have to ensure that the element types of both containers are the same. But the Index_Type for the vector does not matter nor do the equality operations have to be the same – so we have used the **others** => <> notation to cover them.

The function body could be written in many ways. We have chosen to use the index facilities of the vector container for illustration. We could equally have used a cursor thus

```
function Convert(The_Vector: V.Vector)
                                  return DLL.List is
   The_List: DLL.List;
   V_Cursor: V.Cursor;
begin;
   V_Cursor := The_Vector.First;
   loop
      exit when V_Cursor = V.No_Element;
      The_List.Append(The_Vector.Element);
      V.Next(V_Cursor);
   end loop;
   return The_List
end Convert;
```

3
```
with Ada.Containers.Vectors;
with Ada.Containers.Doubly_Linked_Lists;
use Ada.Containers;
generic
   with package DLL is
                        new Doubly_Linked_Lists(<>);

   with package V is new Vectors(
            Index_Type => <>;
            Element_Type => DLL.Element_Type;
            "=" => DLL."=");
function Equals(The_Vector: V.Vector;
             The_List: DLL.List) return Boolean;

function Equals(The_Vector: V.Vector;
             The_List: DLL.List) return Boolean is
begin;
   if The_List.Length /= The_Vector.Length then
      return False;
   end if;
   for Ind in The_Vector.First_Index ..
                    The_Vector.Last_Index loop
      if The_List.Find(The_Vector.Element(Ind)) =
                             DLL.No_Element then
         return False;
      end if;
   end loop;
   return True;
end Equals;
```

In this case it is necessary for the equality operations to be the same, but naturally the Index_Type does not matter.

This simple solution assumes that there is no duplication of elements. It checks that the two containers have the same number of elements and that for each element in the vector there is an element in the list with the same value. The reader is invited to extend the solution to avoid the assumption of no duplication.

Exercise 24.4

1
```
with Ada.Containers.Ordered_Maps;
with Ada.Tags;  use Ada.Tags;
package body Tag_Registration is
   package Map_It is
                  new Ada.Containers.Ordered_Maps(
                        Key_Type => Character,
                        Element_Type => Tag);
   use Map_It;
   The_Map: Map;

   procedure Register(The_Tag: Tag;
                                 Code: Character) is
   begin
      The_Map.Insert(Code, The_Tag);
   end Register;

   function Decode(Code: Character) return Tag is
      C: Cursor := The_Map.Find(Code);
   begin
      If C = No_Element then
         return No_Tag;
      else
         return Element(C);
      end if;
   end Decode;

end Tag_Registration;
```

The test for No_Element in Decode could be done in several ways. We could use Has_Element or we could even crudely call the function Element that directly takes a Code and this would raise Constraint_Error which could then be handled to return No_Tag. Thus

```
   function Decode(Code: Character) return Tag is
   begin
      return The_Map.Element(Code);
   exception
      when Constraint_Error =>
         return No_Tag;
   end Decode;
```

Shorter but not sweeter.

Exercise 24.5

1
```
with Abstract_Sets;
private with Ada.Containers.Ordered_Sets;
package Container_Sets is
   type C_Set is
                  new Abstract_Sets.Set with private;
   function Empty return C_Set;
   function Unit(E: Element) return C_Set;
   function Union(S, T: C_Set) return C_Set;
   function Intersection(S, T: C_Set) return C_Set;
   procedure Take(From: in out C_Set;
                                       E: out Element);
private
   use Ada.Containers;
   package S_Container is
            new Ordered_Sets(Element);
   use S_Container;
   type C_Set is new Abstract_Sets.Set with
      record
         The_Set: Set;      -- that is S_Container.Set
      end record;
end;

package body Container_Sets is
   function Empty return C_Set is
   begin
      return (The_Set => Empty_Set);
   end Empty;

   function Unit(E: Element) return C_Set is
   begin
      return R: C_Set := (The_Set => <>) do
         R.The_Set.Insert(E);
      end return;
   end;

   function Union(S, T: C_Set) return C_Set is
   begin
      return (The_Set => S.The_Set or T.The_Set);
   end Union;

   function Intersection(S, T: C_Set) return C_Set is
   begin
      return (The_Set => S.The_Set and T.The_Set);
   end Intersection;

   procedure Take(From: in out C_Set;
                                       E: out Element) is
   begin

      E := From.The_Set.First_Element;
      From.The_Set.Delete_First;
   end Take;
end Container_Sets;
```

In this case we have used a wrapper and have to remember that aggregates of one element must be named. We have chosen to use an extended return for Unit – there is no need to initialize R but it helps to emphasize the structure.

If we use a hashed set then Take will need to be rewritten in terms of cursors because First_Element and Delete_First do not exist for hashed sets. Thus

```
   procedure Take(From: in out C_Set;
                                       E: out Element) is
      C: Cursor;
   begin
      C := From.The_Set.First;
      E := Element(C);
      From.The_Set.Delete(C);
   end Take;
```

A more important point is that we can use multiple inheritance and so avoid the wrapper.

```
with Abstract_Sets;
private with Ada.Containers.Ordered_Sets;
package Container_Sets is
   type C_Set is
                  new Abstract_Sets.Set with private;
   function Empty return C_Set;
   function Unit(E: Element) return C_Set;
   function Union(S, T: C_Set) return C_Set;
   function Intersection(S, T: C_Set) return C_Set;
   procedure Take(From: in out C_Set;
                                    E: out Element);
private
   use Ada.Containers;
   package S_Container is
            new Ordered_Sets(Element);
   use S_Container;
   type C_Set is new Set and Abstract_Sets.Set
                                    with null record;
end;

package body Container_Sets is
   function Empty return C_Set is
   begin
      return (Empty_Set with null record);
   end Empty;

   function Unit(E: Element) return C_Set is
   begin
      return R: C_Set do
         R.Insert(E);
      end return;
   end;

   function Union(S, T: C_Set) return C_Set is
   begin
      return S or T;
   end Union;

   function Intersection(S, T: C_Set) return C_Set is
   begin
      return S and T;
   end Intersection;

   procedure Take(From: in out C_Set;
                                    E: out Element) is
   begin

      E := From.First_Element;
      From.Delete_First;
   end Take;
end Container_Sets;
```

Using multiple inheritance makes the body a bit shorter. Note that Union and Intersection are not simply inherited; this is because although C_Set does inherit Union and Intersection from Set nevetheless they get overridden by the new ones we are trying to declare. But luckily the renamings **and** and **or** are also inherited and so we can use them instead. We could equally have used a renaming as body thus

```
   function Union(S, T: C_Set) return C_Set
                                    renames "or";
```

Maybe the wrapper solution is easier to understand.

Exercise 24.10

```
1 function Most(The_Index: Text_Map)
                                    return String is
   Max: Count_Type := 0;
   L: Count_Type;
   Best_One: Indexes.Cursor;

 begin
   for C in The_Index.Iterate loop
      L := Indexes.Element(C).Length;
      if L > Max then
         Best_One := C;
         Max := L;
      end if;
   end loop;
   return Indexes.Element(Best_One);
end Most;
```

Answers 25

Exercise 25.4

1
```
with Ada.Unchecked_Deallocation;  use Ada;
package body Queues is

   procedure Free is
         new Unchecked_Deallocation(Cell, Cell_Ptr);
   ...
   procedure Remove(Q: in out Queue; X: out Item) is
      Old_First: Cell_Ptr := Q.First;
   begin
      if Q.Count = 0 then
         raise Empty;
      end if;
      X := Q.First.Data;
      Q.First := Q.First.Next;
      Q.Count := Q.Count - 1;
      if Q.Count = 0 then
         Q.Last := null;
      end if;
      Free(Old_First);
   end Remove;
   ...
end Queues;
```

Note that we assign **null** to Q.Last if the last item is removed. Otherwise it would continue to refer to the deallocated cell – of course, this should do no harm since it will never be used again but it seems wise not to tempt fate.

Answers 27

Exercise 27.1

1 (a) dynamic – Integer
(b) static – Integer
(c) static – *root_integer*