PSET 6 Solutions

Problem 1

Part 1

Add two 3x3 matrices

Preconditions: two non-empty 3x3 matrices of integer/ real / complex type
Postconditions: A new 3x3 matrix of the same type with the elements added

Pseudo-code:

1. Let the matrices be A, B be the input matrices
2. Let the matrix holding the sum be called Sum.
3. For I in 1 .. 3 loop
    i. For J in 1.. 3 loop
        1. Sum(I,J) := A(I,J) + B(I,J)
4. Return matrix Sum

Multiply two 3x3 matrices

Suppose that A and B are two matrices and that A is an m x n matrix (m rows and n columns) and that B is a p x q matrix. To be able to multiply A and B together, A must have the same number of columns as B has rows (ie. n=p). The product will be a matrix with m rows and q columns. To find the entry in row r and column c of the new matrix we take the "dot product" of row r of matrix A and column c of matrix B (pair up the elements of row r with column c, multiply these pairs together individually, and then add their products).

Mathematically,

$$C(I,J) = \sum_{K=1}^{n} A(I,K)\, B(K,J)$$

Where

- I ranges from 1.. m
- J ranges from 1 .. q
- K ranges from 1 .. n = p

Preconditions: two non-empty 3x3 matrices of integer/ real / complex type
Postconditions: A new 3x3 matrix of the same type with the product of the matrices

Pseudo-code:

1. Let the matrices be A, B be the input matrices
2. Let the matrix holding the product be called Product.

3. Use a local variable sum to store the intermediate value of product.
4. For I in 1 .. 3 loop
    i. For J in 1.. 3 loop
        1. Sum := 0;
        2. For K in 1 .. 3 loop
            a. Sum := Sum + A(I,K) * B(K,J);
        3. End K loop
        4. Product(I,J) := Sum;
    ii. End J loop
5. End I loop
6. Return matrix Product

Transpose a 3x3 matrix

Preconditions: A non-empty 3x3 matrix
Postconditions: A new 3x3 matrix of the same type with the elements in rows and columns exchanged

Pseudo-code:

1. Let the input matrix be A
2. Let the matrix holding the transpose be called Transpose.
3. For I in 1 .. 3 loop
    i. For J in 1.. 3 loop
        1. Transpose(I,J) := A(J,I)
4. Return matrix Transpose

Inverse of a 3x3 matrix

The inverse of a 3×3 matrix is given by:

$$A^{-1} = \frac{\mathrm{adj} A}{\det A}$$

We use *cofactors* to determine the adjoint of a matrix.

The *cofactor* of an element in a matrix is the value obtained by evaluating the determinant formed by the elements not in that particular row or column.

We find the *adjoint matrix* by replacing each element in the matrix with its cofactor and applying a + or - sign as follows:

$$\begin{pmatrix} + & - & + \\ - & + & - \\ + & - & + \end{pmatrix}$$

and then finding the *transpose* of the resulting matrix

The *determinant* of an *n*-by-*n* matrix *A*, denoted det *A* or |*A*|, is a number whose value can be defined recursively as follows. If *n*=1, i.e., if *A* consists of a single element a_{11}, det *A* is equal to a_{11}; for *n* > 1, det *A* is computed by the recursive formula

$$\det A = \sum_{j=1}^{n} s_j a_{1j} \det A_j,$$

where s_j is +1 if *j* is odd and.1 if *j* is even, a_{1j} is the element in row 1 and column *j* , and A_j is the *n* . 1-by-*n* . 1 matrix obtained from matrix *A* by deleting its row 1 and column *j* .

For a 3x3 matrix, the formula can be determined as:

$$\begin{aligned}
&\det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \\
&= a_{11} \det \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} - a_{12} \det \begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix} + a_{13} \det \begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \\
&= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13} - a_{31}a_{22}a_{13} - a_{21}a_{12}a_{33} - a_{32}a_{23}a_{11}.
\end{aligned}$$

Preconditions: A 3x3 invertible matrix
Postconditions: A new 3x3 matrix which is the inverse of the input matrix

Pseudocode:

1. Let the input matrix be A
2. Let the cofactor matrix be Cofactor
3. Let Determinant be the variable used to store the determinant
4. For I in 1.. 3 loop
    a. Compute the indices of the elements to compute the determinant using the formula:
        i. I1 := (I + 1) Rem 3. If I1 = 0 then I1 = 3
        ii. I2 := (I + 2) Rem 3. If I2 = 0 then I2 = 3
    b. For J in 1..3 loop
        i. Compute the indices of the elements to compute the determinant using the formula:
            1. J1 := (J + 1) Rem 3. If J1 = 0 then J1 = 3
            2. J2 := (J + 2) Rem 3. If J2 = 0 then J2 = 3

        ii. Cofactor(I,J):=A(I1,J1)*A(I2,J2)- A(I1,J2) * A(I2,J1)

5. Compute determinant as
   Determinant := A(1,1)*Cofactor(1,1) + A(1,2)*Cofactor (1,2)+ A(1,3)*Cofactor (1,3)

6. Compute the transpose of the Cofactor matrix
7. For I in 1.. 3
   a. For J in 1..3
      i. Inverse(I,J) := Cofactor(I,J) / Determinant
8. Return Inverse

Note: The method for computing the cofactor automatically generates the required signs in the cofactor matrix.

Part 2

Matrix Specification

GNAT 3.13p (20000509) Copyright 1992-2000 Free Software Foundation, Inc.

Checking: c:/docume~2/jk/desktop/16070c~1/lab4~1/matrix.ads (source file time stamp: 2003-03-30 15:08:38)

```
1. ------------------------------------------------------------------
2. -- Package to perform the following operations
3. --   - Create a 3x3 matrix
4. --   - Display a 3x3 matrix
5. --   - Add two 3x3 matrices
6. --   - Multiply two 3x3 matrices
7. --   - Transpose a 3x3 matrix
8. --   - Invert a 3x3 matrix
9. -- Specifier : Jayakanth Srinivasan
10. -- Date Last Modified : 15 march 2003
11. ------------------------------------------------------------------
12.
13. package Matrix is
14.    type A3X3matrix is array (1..3, 1..3) of Float;
15.
16.    -- function to create the 3X3 matrix
17.    -- preconditions: invoked with the matrix to be created on the LHS
18.    -- postconditions: matrix with the elements filled
19.    function Create return A3X3matrix;
20.
21.    -- Procedure to display the 3X3 matrix
22.    -- preconditions: existing 3X3 matrix
23.    -- postconditions: matrix displayed to user on screen
24.    procedure Display (Matrix_A : in A3X3matrix);
25.
26.    -- function to add two 3X3 matrices
27.    -- Preconditions: two 3X3 matrices with elements of type float
28.    --                the added matrix in the LHS of the call
29.    -- Postconditions: a new 3X3 matrix with the elements added
```

```
30.    function Add(Matrix_A : A3x3matrix; Matrix_B : A3x3matrix) return A3x3matrix;
31.
32.    -- function to multiply two 3X3 matrices
33.    -- Preconditions: two 3X3 matrices with elements of type float
34.    --                the resultant matrix in the LHS of the call
35.    -- Postconditions: a new 3X3 matrix with the matrices multiplied
36.    function Multiply(Matrix_A : A3x3matrix; Matrix_B : A3x3matrix) return A3x3matrix;
37.
38.    -- function to transpose a 3x3 matrix
39.    -- Preconditions : Existing 3X3 matrix
40.    -- Postconditions : functions returns a transposed matrix
41.    function Transpose(Matrix_A : A3x3matrix) return A3x3matrix;
42.
43.    -- function to inverse a 3x3 matrix
44.    -- Precondtions : invertible 3X3 matrix
45.    -- Postconditions : inverted matrix in the LHS of function call
46.    function Invert(Matrix_A : A3x3matrix) return A3x3matrix;
47.
48. end Matrix;
49.
```

49 lines: No errors

Matrix Implementation

GNAT 3.13p (20000509) Copyright 1992-2000 Free Software Foundation, Inc.

Compiling: c:/docume~2/jk/desktop/16070c~1/lab4~1/matrix.adb (source file time stamp: 2003-03-30 15:59:14)

```
1. ----------------------------------------------------------------
2. -- Implementation of the Matrix package
3. -- Implementer: Jayakanth Srinivasan
4. -- Date Last Modified : 15 march 2003
5. ----------------------------------------------------------------
6. with Ada.Text_Io;
7. with Ada.Float_Text_Io;
8. with Ada.Integer_Text_IO;
9.
10. use Ada.Text_Io;
11. use Ada.Float_Text_Io;
12. use Ada.Integer_Text_Io;
13.
14. package body Matrix is
15.
16.    -- function to create the 3X3 matrix
17.    -- preconditions: invoked with the matrix to be created on the LHS
18.    -- postconditions: matrix with the elements filled
19.    function Create return A3x3matrix is
20.       Matrix_A : A3x3matrix;
21.       begin
22.       for I in 1..3 loop
23.          for J in 1..3 loop
```

```
24.            Put("Please Enter the element in ");
25.            Put(I);
26.            Put(",");
27.            Put(J);
28.            Put(": ");
29.              Get(Matrix_A(I,J));
30.              Skip_Line;
31.              New_Line;
32.            end loop;
33.          end loop;
34.          return Matrix_A;
35.       end Create;
36.
37.
38.    -- Procedure to display the 3X3 matrix
39.    -- preconditions: existing 3X3 matrix
40.    -- postconditions: matrix displayed to user on screen
41.       procedure Display (Matrix_A : in A3x3matrix) is
42.       begin
43.          for I in 1..3 loop
44.            for J in 1..3 loop
45.              Put(Matrix_A(I,J));
46.            Put("  ");
47.            end loop;
48.            New_Line;
49.          end loop;
50.       end Display;
51.
52.
53.    -- function to add two 3X3 matrices
54.    -- Preconditions: two 3X3 matrices with elements of type float
55.    --               the added matrix in the LHS of the call
56.    -- Postconditions: a new 3X3 matrix with the elements added
57.       function Add(Matrix_A : A3x3matrix; Matrix_B : A3x3matrix) return A3x3matrix is
58.          Matrix_C : A3x3matrix;
59.       begin
60.          for I in 1..3 loop
61.            for J in 1..3 loop
62.              Matrix_C(I,J) := Matrix_A(I,J) +  Matrix_B(I,J);
63.            end loop;
64.          end loop;
65.          return Matrix_C;
66.       end Add;
67.
68.    -- function to multiply two 3X3 matrices
69.    -- Preconditions: two 3X3 matrices with elements of type float
70.    --               the resultant matrix in the LHS of the call
71.    -- Postconditions: a new 3X3 matrix with the matrices multiplied
72.    function Multiply(Matrix_A : A3x3matrix; Matrix_B : A3x3matrix) return A3x3matrix is
73.       Matrix_C : A3x3matrix;
74.       Sum : Float;
75.
76.       begin
77.          for I in 1..3 loop
78.            for J in 1..3 loop
79.              Sum :=0.0;
```

```
80.              for K in 1..3 loop
81.              Sum := Sum + Matrix_A(I,K) * Matrix_B(K,J);
82.              end loop;
83.              Matrix_C(I,J):=Sum;
84.            end loop;
85.          end loop;
86.          return Matrix_C;
87.       end Multiply;
88.
89.
90.    -- function to transpose a 3x3 matrix
91.    -- Preconditions : Existing 3X3 matrix
92.    -- Postconditions : functions returns a transposed matrix
93.       function Transpose(Matrix_A : A3x3matrix) return A3x3matrix is
94.          Matrix_A_Transpose : A3x3matrix;
95.       begin
96.          for I in 1..3 loop
97.             for J in 1..3 loop
98.                Matrix_A_Transpose(I,J) := Matrix_A(J,I);
99.             end loop;
100.         end loop;
101.         return Matrix_A_Transpose;
102.      end Transpose;
103.
104.
105.   -- function to inverse a 3x3 matrix
106.   -- Precondtions : invertible 3X3 matrix
107.   -- Postconditions : inverted matrix in the LHS of function call
108.   function Invert(Matrix_A : A3x3matrix) return A3x3matrix is
109.      Matrix_A_Inv : A3x3matrix;
110.      Cofactor_Matrix : A3x3matrix;
111.      I1,I2,J1,J2 : Integer;
112.       determinant : float;
113.   begin
114.
115.      for I in 1.. 3 loop
116.         I1 := Integer(I+1) REM 3;
117.         if (I1 = 0) then
118.            I1 :=3;
119.         end if;
120.
121.            I2 := Integer(I+2) REM 3;
122.         if (I2 = 0) then
123.            I2 :=3;
124.         end if;
125.
126.         for J in 1..3 loop
127.            J1 := Integer(J+1) rem 3;
128.            J2 := Integer(J+2) REM 3;
129.            if (J1 = 0) then
130.               J1 :=3;
131.            end if;
132.            if (J2 = 0) then
133.               J2 :=3;
134.            end if;
```

```
135.             Cofactor_Matrix(I,J):= Matrix_A(I1,J1)*Matrix_A(I2,J2) -
Matrix_A(I1,J2)*Matrix_A(I2,J1);
136.          end loop;
137.       end loop;
138.
139.       Determinant := Matrix_A(1,1)*Cofactor_Matrix(1,1) +
Matrix_A(1,2)*Cofactor_Matrix(1,2)+ Matrix_A(1,3)*Cofactor_Matrix(1,3);
140.       Cofactor_Matrix := Transpose(Cofactor_Matrix);
141.
142.       for I in 1 .. 3 loop
143.          for J in 1.. 3 loop
144.             Matrix_A_Inv(I,J):= Cofactor_Matrix(I,J) / Determinant;
145.          end loop;
146.       end loop;
147.
148.       return Matrix_A_Inv;
149.    end Invert;
150. end Matrix;
```

150 lines: No errors

Test Program

GNAT 3.13p (20000509) Copyright 1992-2000 Free Software Foundation, Inc.

Compiling: c:/docume~2/jk/desktop/16070c~1/lab4~1/matrix_test.adb (source file time stamp: 2003-03-30 18:30:50)

```
1. ---------------------------------------------------------------
2. -- Program to test the package by
3. --   - Creating two 3X3 matrices
4. --   - Display the contents to the user
5. --   - Add the matrices
6. --   - Display the sum to the user
7. --   - Multiply the matrices
8. --   - Display the product to the user
9. --   - Transpose the product
10. --   - Display the transpose to the user
11. --   - Invert the product matrix
12. --   - Display it to the user
13. --   Programmer : Jayakanth Srinivasan
14. --   Date Last Modified : March 15 2003
15. ---------------------------------------------------------------
16. with Matrix;
17. with Ada.Text_Io;
18.
19. use Matrix;
20. use Ada.Text_Io;
21.
22. procedure Matrix_Test is
23.    Matrix_A, Matrix_B, Matrix_Sum, Matrix_Product, Matrix_Transpose : A3x3matrix;
24. begin
25.    -- create the matrices
26.    Matrix_A := Create;
```

```
27.    Matrix_B := Create;
28.
29.    -- display the matrices
30.    Put_Line("Matrix A");
31.    Display(Matrix_A);
32.    New_Line;
33.
34.    Put_Line("Matrix B");
35.    Display (Matrix_B);
36.    New_Line;
37.    -- add the matrices and display it
38.    Put_Line("Sum of Matrix A and Matrix B");
39.    Matrix_Sum := Add(Matrix_A, Matrix_B);
40.    Display (Matrix_Sum);
41.    New_Line;
42.
43.    -- multiply the matrices and display it to the user
44.    Matrix_Product:= Multiply(Matrix_A, Matrix_B);
45.    Put_Line("Product of Matrix A and Matrix B");
46.    Display(Matrix_Product);
47.    New_Line;
48.
49.    -- transpose the product matrix and display it to the user
50.    Matrix_Transpose := Transpose(Matrix_Product);
51.    Put_Line("Transpose of the Product Matrix");
52.    Display (Matrix_Transpose);
53.    New_Line;
54.
55.    -- invert the product matrix and display it to the user
56.    Matrix_Inverse := Inverse(Matrix_Product);
57.    Put_Line("The inverted matrix is");
58.    Display (Matrix_Inverse);
59.    New_Line;
60.
61. end Matrix_Test;
```

61 lines: No errors

Part 3

a. Create

```
19.   function Create return A3x3matrix is
20.      Matrix_A : A3x3matrix;
21.      begin
22.      for I in 1..3 loop                                  ┐
23.         for J in 1..3 loop                               ┐
24.            Put("Please Enter the element in ");    1     │
25.            Put(I);                                 1     │
26.            Put(",");                               1     │
27.            Put(J);                                 1     │  N^2   N
28.            Put(": ");                              1     │
29.               Get(Matrix_A(I,J));                  1     │
30.               Skip_Line;                           1     │
31.               New_Line;                            1     │
32.            end loop;                                      ┘
                                                              ┘
```

```
33.       end loop;
34.       return Matrix_A;                          1
35.     end Create;
```

Total = $8* N^2 + N + 1 = O(N^2)$

b. Display

```
41.     procedure Display (Matrix_A : in A3x3matrix) is
42.     begin
43.       for I in 1..3 loop
44.         for J in 1..3 loop
45.           Put(Matrix_A(I,J));          1
46.         Put(" ");                      1          N^2          N
47.         end loop;
48.         New_Line;                      1
49.       end loop;
50.     end Display;
```

Total = $2* N^2 + N = O(N^2)$

c. Add two matrices $O(N^2)$ as the structure is similar to the display program

d. Multiply

```
72.   function Multiply(Matrix_A : A3x3matrix; Matrix_B : A3x3matrix) return A3x3matrix is
73.     Matrix_C : A3x3matrix;
74.     Sum : Float;
75.
76.     begin
77.       for I in 1..3 loop
78.         for J in 1..3 loop
79.           Sum :=0.0;                                  1
80.           for K in 1..3 loop
81.           Sum := Sum + Matrix_A(I,K) * Matrix_B(K,J);  1      N*N*(N+2)   N*(N+2)
82.           end loop;                                                          N
83.           Matrix_C(I,J):=Sum;                         1
84.         end loop;
85.       end loop;
86.       return Matrix_C;
87.     end Multiply;
```

Total = $N*N*(N+2) = N^3 + N^2 + 2N = O(N^3)$

e. Transpose a matrix is $O(N^2)$ as the structure is similar to the display program

f. Invert

```
108.   function Invert(Matrix_A : A3x3matrix) return A3x3matrix is
109.     Matrix_A_Inv : A3x3matrix;
110.     Cofactor_Matrix : A3x3matrix;
111.     I1,I2,J1,J2 : Integer;
112.      determinant : float;
113.   begin
114.
```

```
115.      for I in 1.. 3 loop                                                    6N
116.        I1 := Integer(I+1) REM 3;
117.        if (I1 = 0) then
118.          I1 :=3;
119.        end if;
120.
121.          I2 := Integer(I+2) REM 3;
122.        if (I2 = 0) then
123.          I2 :=3;
124.        end if;
125.
126.        for J in 1..3 loop                                                   5N^2
127.          J1 := Integer(J+1) rem 3;
128.          J2 := Integer(J+2) REM 3;
129.          if (J1 = 0) then
130.            J1 :=3;
131.          end if;
132.          if (J2 = 0) then
133.            J2 :=3;
134.          end if;
135.          Cofactor_Matrix(I,J):= Matrix_A(I1,J1)*Matrix_A(I2,J2) -
Matrix_A(I1,J2)*Matrix_A(I2,J1);
136.        end loop;
137.      end loop;
138.
139.      Determinant := Matrix_A(1,1)*Cofactor_Matrix(1,1) +
Matrix_A(1,2)*Cofactor_Matrix(1,2)+ Matrix_A(1,3)*Cofactor_Matrix(1,3);      (1)
140.      Cofactor_Matrix := Transpose(Cofactor_Matrix);                       O(N^2)
141.
142.      for I in 1 .. 3 loop
143.        for J in 1.. 3 loop
144.          Matrix_A_Inv(I,J):= Cofactor_Matrix(I,J) / Determinant;          (N^2)
145.        end loop;
146.      end loop;
147.
148.      return Matrix_A_Inv;                                                 (1)
149.    end Invert;
```

$$\text{Total} = 6N + 5N^2 + N^2 + O(N^2) + 2 = O(N^2)$$

Problem 2

Strings are arrays of characters. They can be treated as arrays in that they can be indexed and individual elements can be treated as characters.

The predefined Ada language environment includes string handling packages to encourage portability. They support different categories of strings: fixed length, bounded length, and unbounded length. They also support subprograms for string construction, concatenation, copying, selection, ordering, searching, pattern matching, and string transformation.

Ada.Strings.Maps

The package Strings.Maps defines the types, operations, and other entities needed for character sets and character-to-character mappings.

```
function Is_In (Element : in Character;
        Set     : in Character_Set);
return Boolean
```

Determines if the element is in the string.

```
function Is_Subset (Elements : in Character_Set;
        Set      : in Character_Set)
return Boolean;
```

Determines if set of elements belongs to the string.

```
function To_Mapping (From, To : in Character_Sequence) return Character_Mapping;
```

To_Mapping produces a Character_Mapping such that each element of From maps to the corresponding element of To, and each other character maps to itself. If From'Length /= To'Length, or if some character is repeated in From, then Translation_Error is propagated.

For Example: To_Mapping("ABCD", "ZZAB") returns a Character_Mapping that maps 'A' and 'B' to 'Z', 'C' to 'A', 'D' to 'B', and each other Character to itself.

Ada.Strings.Fixed

The language-defined package Strings.Fixed provides string-handling subprograms for fixed-length strings; that is, for values of type Standard.String. Several of these subprograms are procedures that modify the contents of a String that is passed as an out

or an in out parameter; each has additional parameters to control the effect when the logical length of the result differs from the parameter's length.

```
function Element (Source : in Bounded_String;
                  Index  : in Positive)
   return Character;
```

Returns the character at position Index in the string represented by Source; propagates Index_Error if Index > Length(Source).

```
procedure Replace_Element (Source : in out Bounded_String;
                           Index  : in Positive;
                           By     : in Character);
```

Updates Source such that the character at position Index in the string represented by Source is By; propagates Index_Error if Index > Length(Source).

```
function Slice (Source : in Bounded_String;
                Low    : in Positive;
                High   : in Natural)
   return String;
```

Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1.

```
procedure Move (Source  : in  String;
                Target  : out String;
                Drop    : in  Truncation := Error;
                Justify : in  Alignment  := Left;
                Pad     : in  Character  := Space);
```

The Move procedure copies characters from Source to Target. If Source has the same length as Target, then the effect is to assign Source to Target. If Source is shorter than Target then:

- If Justify=Left, then Source is copied into the first Source'Length characters of Target.
- If Justify=Right, then Source is copied into the last Source'Length characters of Target.

- If Justify=Center, then Source is copied into the middle Source'Length characters of Target. In this case, if the difference in length between Target and Source is odd, then the extra Pad character is on the right.

- Pad is copied to each Target character not otherwise assigned.

  If Source is longer than Target, then the effect is based on Drop.

- If Drop=Left, then the rightmost Target'Length characters of Source are copied into Target.

- If Drop=Right, then the leftmost Target'Length characters of Source are copied into Target.

- If Drop=Error, then the effect depends on the value of the Justify parameter and also on whether any characters in Source other than Pad would fail to be copied:

  - If Justify=Left, and if each of the rightmost Source'Length-Target'Length characters in Source is Pad, then the leftmost Target'Length characters of Source are copied to Target.

  - If Justify=Right, and if each of the leftmost Source'Length-Target'Length characters in Source is Pad, then the rightmost Target'Length characters of Source are copied to Target.

  - Otherwise, Length_Error is propagated.

```
function Index (Source   : in String;
                Pattern  : in String;
                Going    : in Direction := Forward;
                Mapping  : in Maps.Character_Mapping_Function)
    return Natural;
```

Each Index function searches for a slice of Source, with length Pattern'Length, that matches Pattern with respect to Mapping; the parameter Going indicates the direction of the lookup. If Going = Forward, then Index returns the smallest index I such that the slice of Source starting at I matches Pattern. If Going = Backward, then Index returns the largest index I such that the slice of Source starting at I matches Pattern. If there is no such slice, then 0 is returned. If Pattern is the null string then Pattern_Error is propagated.

```
function Count (Source   : in String;
                Pattern  : in String;
```

```
            Mapping  : in Maps.Character_Mapping_Function)
    return Natural;
```

Returns the maximum number of nonoverlapping slices of Source that match Pattern with respect to Mapping. If Pattern is the null string then Pattern_Error is propagated.

```
procedure Find_Token (Source : in String;
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

Find_Token returns in First and Last the indices of the beginning and end of the first slice of Source all of whose elements satisfy the Test condition, and such that the elements (if any) immediately before and after the slice do not satisfy the Test condition. If no such slice exists, then the value returned for Last is zero, and the value returned for First is Source'First.

```
function Translate (Source  : in String;
                    Mapping : in Maps.Character_Mapping_Function)
  return String;
```

Returns the string S whose length is Source'Length and such that S(I) is the character to which Mapping maps the corresponding element of Source, for I in 1..Source'Length.

```
function Replace_Slice (Source   : in String;
                        Low      : in Positive;
                        High     : in Natural;
                        By       : in String)
    return String;
```

If Low > Source'Last+1, or High < Source'First-1, then Index_Error is propagated. Otherwise, if High >= Low then the returned string comprises Source(Source'First..Low-1) & By & Source(High+1..Source'Last), and if High < Low then the returned string is Insert(Source, Before=>Low, New_Item=>By).

```
function Insert (Source   : in String;
                 Before   : in Positive;
                 New_Item : in String)
  return String;
```

Propagates Index_Error if Before is not in Source'First .. Source'Last+1; otherwise returns Source(Source'First..Before-1) & New_Item & Source(Before..Source'Last), but with lower bound 1.

```
function Overwrite (Source   : in String;
                    Position : in Positive;
                    New_Item : in String)
      return String;
```

Propagates Index_Error if Position is not in Source'First .. Source'Last+1; otherwise returns the string obtained from Source by consecutively replacing characters starting at Position with corresponding characters from New_Item. If the end of Source is reached before the characters in New_Item are exhausted, the remaining characters from New_Item are appended to the string.

```
function Delete (Source  : in String;
                 From    : in Positive;
                 Through : in Natural)
      return String;
```

If From <= Through, the returned string is Replace_Slice(Source, From, Through, ""), otherwise it is Source.

```
    function Trim (Source : in String;
                   Side   : in Trim_End)
      return String;
```

Returns the string obtained by removing from Source all leading Space characters (if Side = Left), all trailing Space characters (if Side = Right), or all leading and trailing Space characters (if Side = Both).

```
      function Head (Source : in String;
                     Count  : in Natural;
                     Pad    : in Character := Space)
      return String;
```

Returns a string of length Count. If Count <= Source'Length, the string comprises the first Count characters of Source. Otherwise its contents are Source concatenated with Count-Source'Length Pad characters.

```
      function Tail (Source : in String;
                     Count  : in Natural;
```

```
            Pad    : in Character := Space)
    return String;
```

Returns a string of length Count. If Count <= Source'Length, the string comprises the last Count characters of Source. Otherwise its contents are Count-Source'Length Pad characters concatenated with Source.

Ada.Strings.Bounded

The language-defined package Strings.Bounded provides a generic package each of whose instances yields a private type Bounded_String and a set of operations. An object of a particular Bounded_String type represents a String whose low bound is 1 and whose length can vary conceptually between 0 and a maximum size established at the generic instantiation. The subprograms for fixed-length string handling are either overloaded directly for Bounded_String, or are modified as needed to reflect the variability in length. Additionally, since the Bounded_String type is private, appropriate constructor and selector operations are provided.

The functions are extensions of that defined in Ada.Strings.Maps and Ada.Strings.Fixed.

```
function Length (Source : in Bounded_String) return Length_Range;
```

The Length function returns the length of the string represented by Source.

```
function To_Bounded_String (Source : in String;
                            Drop   : in Truncation := Error)
  return Bounded_String;
```

If Source'Length <= Max_Length then this function returns a Bounded_String that represents Source. Otherwise the effect depends on the value of Drop:

- If Drop=Left, then the result is a Bounded_String that represents the string comprising the rightmost Max_Length characters of Source.
- If Drop=Right, then the result is a Bounded_String that represents the string comprising the leftmost Max_Length characters of Source.
- If Drop=Error, then Strings.Length_Error is propagated.

Problem 3

Part 1

Preconditions: A non-empty input string containing the expression in infix form
Postconditions: A string in postfix form that is equivalent to the infix expression

Pseudocode:

1. Create a user Stack
2. Get the infix expression from the user as a string, say Infix
3. Check if the paranthesis are balanced as follows:
   For I in 1.. length(Infix) do
      i. If Infix(I) = '(' then Push onto the Stack
      ii. If Infix(I) = ')' then Pop one element form the Stack
4. If Stack is non-empty
   a. Display "non-balanced expression"
   b. Goto 2
5. Create a new string Postfix
6. Set Postfix_Index to 1
7. For I in 1 .. Length(Infix)
   a. If Infix(I) is an operand, append it to postfix string as follows:
      i. Postfix(Postfix_Index) := Infix(I);
      ii. Postfix_Index:=Postfix_Index + 1;

   b. If the Infix(I) is an operator, process operator as follows
      1. Set done to false
      2. Repeat
         a. If Stack is empty or Infix(I) is '(' then
            i. push Infix(I) onto stack
            ii. set done to true
         b. Else if precedence(Infix(I)) > precedence(top operator)
            i. Push Infix(I) onto the stack (ensures higher precedence operators evaluated first)
            ii. set done to true
         c. Else
            i. Pop the operator stack
            ii. If operator popped is '(', set done to true
            iii. Else append operator popped to postfix string
      3. Until done
8. While Stack is not empty
   a. Pop operator
   b. Append it to the postfix string
9. Return Postfix

Preconditions: A non-empty input string containing the expression in infix form
Postconditions: A string in prefix form that is equivalent to the infix expression

Pseudocode:
Pseudocode:

1. Create a user Stack
2. Get the infix expression from the user as a string, say Infix
3. Check if the paranthesis are balanced as follows:
   For I in 1 .. length(Infix) do
      i. If Infix(I) = '(' then Push onto the Stack
      ii. If Infix(I) = ')' then Pop one element form the Stack
4. If Stack is non-empty
   b. Display "non-balanced expression"
   c. Goto 2
5. Create a new string Prefix
6. Set Prefix_Index to 1
7. For I in Length(Infix) down to 1
   d. If Infix(I) is an operand, append it to Prefix string as follows:
      i. Prefix(Prefix_Index) := Infix(I);
      ii. Prefix_Index:=Prefix_Index + 1;

   e. If the Infix(I) is an operator, process operator as follows
      4. Set done to false
      5. Repeat
         a. If Stack is empty or Infix(I) is ')' then
            i. push Infix(I) onto stack
            ii. set done to true
         b. Else if precedence(Infix(I)) > precedence(top operator)
            i. Push Infix(I) onto the stack (ensures higher precedence operators evaluated first)
            ii. set done to true
         c. Else
            i. Pop the operator stack
            ii. If operator popped is ')', set done to true
            iii. Else append operator popped to Prefix string
      6. Until done
8. While Stack is not empty
   f. Pop operator
   g. Append it to the Prefix string
9. Reverse Prefix string as follows
   a. For I in 1.. Length(Prefix) do
      i. Push Prefix(I) onto stack
   b. For I in 1.. Length(Prefix) do
      i. Pop one element form stack
      ii. Prefix(I) := element
10. Return Prefix

Problem Statement

Write a program in Ada95 to:

- Get an input expression in infix form from the user.
- Convert the expression into postfix
- Display it to the user
- Convert the expression into prefix notation
- Display it to the user.

Analysis

Input : Input expression in infix form
Output : Two expressions in prefix and postfix form
Constraints: The program is restricted to expressions of Stack_Size length

Design

PseudoCode:

1. Get the input from the user
2. Check if the input is balanced. If not, goto step 1
3. Convert into postfix using the algorithm detailed in Part 1
4. Display the converted expression to the user
5. Convert into prefix using the algorithm detailed in Part 2
6. Display the converted expression to the user

Test Plan

Test Case	Input	Expected Output
1	a+b	ab+, +ab
2	(a+b	Loop until user enters the correct input
3	(a+b)-c	ab+c-, -+abc
4	3+5*6 – 7 * (8+5)	356*+785+*- +3-*56*7+85
5	a + b	ab+, +ab

Code Listing

Converter Test

GNAT 3.13p (20000509) Copyright 1992-2000 Free Software Foundation, Inc.

Compiling: c:/docume~2/jk/desktop/16070c~1/lab4~1/converter_test.adb (source file time stamp: 2003-03-30 23:08:04)

```
1. ------------------------------------------------------------
2. -- Procedure to test the my_expression_converter package
3. -- Programmer : Jayakanth Srinivasan
```

```
4. -- Date Last Modified : Mar 20, 2003
5. ------------------------------------------------------------
6.
7. with My_Expression_Converter;
8. with Ada.Text_IO;
9.
10. use My_Expression_Converter;
11. use Ada.Text_IO;
12.
13. procedure Converter_Test is
14.    Infix, postfix, prefix : String(1.. Stack_Size);
15.    Length : Integer;
16. begin
17.    loop
18.       Put_Line("Please enter a valid expression in infix form");
19.       Get_Line(Infix, Length);
20.       exit when Check_Balance(Infix, length);
21.    end loop;
22.    -- convert into postfix and display it
23.    Postfix:=Infix_To_Postfix(Infix, Length);
24.    Put_Line(Postfix);
25.    -- convert into prefix and display it
26.    Prefix:=Infix_To_Prefix(Infix, Length);
27.    Put_Line(Prefix);
28.
29. end Converter_Test;
30.
31.
```

31 lines: No errors

My_Expression_Converter Specification

GNAT 3.13p (20000509) Copyright 1992-2000 Free Software Foundation, Inc.

Checking: c:/docume~2/jk/desktop/16070c~1/lab4~1/my_expression_converter.ads (source file time stamp: 2003-03-30 23:10:28)

```
1. ------------------------------------------------------------------
2. -- Package Specification for My_Expression_Converter
3. --
4. -- Specifier : Jayakanth Srinivasan
5. -- Date Last Modified : March 17, 2003
6. ------------------------------------------------------------------
7.
8. package My_Expression_Converter is
9.    Stack_Size : constant Integer := 50;
10.    type User_Stack is array (1 .. Stack_Size) of Character;
11.
12.    -- declare a stack and top index as a record
13.    type My_Stack is
14.       record
15.          Stack : User_Stack;
16.          Top   : Integer;
```

```
17.      end record;
18.
19.   -- create the stack by intializing top to 1;
20.   procedure Create (
21.        Input_Stack : in out My_Stack );
22.
23.   -- push an element onto stack
24.   procedure Push (
25.        Input_Stack : in out My_Stack;
26.        Element     : in     Character);
27.
28.   -- pop an element from the stack
29.   procedure Pop (
30.        Input_Stack : in out My_Stack;
31.        Element     :    out Character);
32.
33.   -- top element in the stack
34.   function Stacktop(Input_Stack : My_Stack) return Character;
35.
36.   -- stack length
37.   function Stacklength (
38.        Input_Stack : My_Stack )
39.    return Integer;
40.
41.   -- is empty stack
42.   function Isempty (
43.        Input_Stack : My_Stack )
44.    return Boolean;
45.
46.   -- is full stack
47.   function Isfull (
48.        Input_Stack : My_Stack )
49.    return Boolean;
50.
51.   -- checks if input is an operator
52.   function Isoperator (
53.        Operator : Character )
54.     return Boolean;
55.
56.   -- returns the precedence of the operator
57.   function Precedence (
58.        Operator : Character )
59.    return Integer;
60.
61.   -- function to convert infix to postfix
62.   function Infix_To_Postfix(Infix : String; Length: Integer) return String;
63.
64.   -- function to convert infix to postfix
65.   function Infix_To_Prefix (Infix : String; Length: Integer) return String;
66.
67.   -- function to check the balance of the input string
68.   function Check_Balance(Infix : String; Length : Integer) return Boolean;
69.
70. end My_Expression_Converter;
```

70 lines: No errors

My_Expression_Converter Body

GNAT 3.13p (20000509) Copyright 1992-2000 Free Software Foundation, Inc.

Compiling: c:/docume~2/jk/desktop/16070c~1/lab4~1/my_expression_converter.adb (source file time stamp: 2003-03-30 23:07:42)

```
1. ---------------------------------------------------------------------
2. -- Package body for My_Expression_Converter
3. --
4. -- Implementer: Jayakanth Srinivasan
5. -- Date Last Modified : March 17, 2003
6. ---------------------------------------------------------------------
7.
8. with Ada.Text_Io;
9.
10. use Ada.Text_Io;
11.
12. package body My_Expression_Converter is
13.
14.    -- create the stack by intializing top to 1;
15.    procedure Create (
16.          Input_Stack : in out My_Stack ) is
17.    begin
18.       Input_Stack.Top :=1;
19.    end Create;
20.
21.    -- push an element onto stack
22.    procedure Push (
23.          Input_Stack : in out My_Stack;
24.          Element     : in     Character) is
25.    begin
26.       if (Isfull(Input_Stack)) then
27.          Put_Line("Cannot Push Onto Stack");
28.       else
29.          Input_Stack.Stack(Input_Stack.Top) := Element;
30.          Input_Stack.Top:= Input_Stack.Top +1;
31.       end if;
32.    end Push;
33.
34.
35.    -- pop an element from the stack
36.    procedure Pop (
37.          Input_Stack : in out My_Stack;
38.          Element     :    out Character) is
39.    begin
40.       if (Isempty(Input_Stack)) then
41.          Put_Line("Cannot Pop an Empty Stack");
42.       else
43.          Input_Stack.Top:= Input_Stack.Top -1;
44.          Element:= (Input_Stack.Stack(Input_Stack.Top));
45.       end if;
```

```
46.    end Pop;
47.
48.
49.    -- stack length
50.    function Stacklength (
51.          Input_Stack : My_Stack )
52.      return Integer is
53.    begin
54.       return(Input_Stack.Top -1);
55.    end Stacklength;
56.
57.
58.    -- is empty stack
59.    function Isempty (
60.          Input_Stack : My_Stack )
61.      return Boolean is
62.    begin
63.       if Input_Stack.Top = 1 then
64.          return True;
65.       else
66.          return False;
67.       end if;
68.    end Isempty;
69.
70. -- function to return the top element in the stack
71.    function Stacktop(Input_Stack : My_Stack) return Character is
72.    begin
73.       if Isempty(Input_Stack) = False then
74.          return( Input_Stack.Stack(Input_STack.Top-1));
75.       else
76.          return('#');
77.       end if;
78.    end Stacktop;
79.
80.
81.    -- is full stack
82.    function Isfull (
83.          Input_Stack : My_Stack )
84.      return Boolean is
85.    begin
86.       if Input_Stack.Top = Stack_Size then
87.          return True;
88.       else
89.          return False;
90.       end if;
91.    end Isfull;
92.
93.
94.
95.    -- checks if input is an operator
96.    function Isoperator (
97.          Operator : Character )
98.      return Boolean is
99.    begin
100.      case Operator is
101.         when '*'| '/'|'+'|'-'|'('|')' =>
```

```
102.            return True;
103.         when others =>
104.            return False;
105.      end case;
106.   end Isoperator;
107.
108.   -- checks if input is an operator
109.   function Precedence (
110.         Operator : Character )
111.     return Integer is
112.   begin
113.      case Operator is
114.         when '*'| '/'=> return 3;
115.         when '+'|'-' => return 2;
116.         when '('| ')' => return 1;
117.         when others =>
118.            return -1;
119.      end case;
120.   end Precedence;
121.   -- function to check balance
122.   function Check_Balance(Infix : String; Length : Integer) return Boolean is
123.   count : integer :=0;
124.   begin
125.      for I in 1 .. Length loop
126.         if Infix(I) = '(' then
127.            Count := Count +1;
128.         end if;
129.         if Infix(I) = ')' then
130.            Count := Count -1;
131.         end if;
132.      end loop;
133.      if (Count /= 0) then
134.            return False;
135.      else
136.         return True;
137.      end if;
138.   end Check_Balance;
139.
140.
141.   -- function to convert infix to postfix
142.   function Infix_To_Postfix(Infix : String; Length: Integer) return String is
143.      Postfix : String(1.. stack_size);
144.      Postfix_Index : Integer;
145.      Operator_Stack : My_Stack;
146.      Done   : Boolean;
147.      element : character;
148.   begin
149.      Create(Operator_Stack);
150.      Postfix_Index := 1;
151.      for I in 1.. Length loop
152.         -- if it is an operand
153.         if Isoperator(Infix(I)) = False then
154.            -- ignore blank spaces
155.            if (Infix(I) /= ' ') then
156.               Postfix(Postfix_Index):= Infix(I);
157.               Postfix_Index := Postfix_Index + 1;
```

```
158.          end If;
159.
160.        else
161.          -- it is an operator
162.          done:= false;
163.          loop
164.            exit when Done = True;
165.            -- if its a ( or an empty stack then push operator onto stack
166.            if IsEmpty(Operator_Stack) or Infix(I) = '(' then
167.              Push(Operator_Stack, Infix(I));
168.              Done := True;
169.              -- check the precedence of the element on top of the stack and infix(I)
170.            elsif Precedence(Infix(I)) > Precedence(Stacktop(Operator_Stack)) then
171.              Push(Operator_Stack, Infix(I));
172.              Done:= True;
173.            else
174.              -- pop until an operator of smaller precedence appears on the stack
175.              Pop(Operator_Stack, Element);
176.              if Element = '(' then
177.                Done := True;
178.              else
179.                Postfix(Postfix_Index):= element;
180.                Postfix_Index := Postfix_Index + 1;
181.              end if;
182.            end if;
183.          end loop;
184.        end if;
185.    end loop;
186.    -- pop the remaining elements into postfix
187.    while (Isempty(Operator_Stack) = false) loop
188.        Pop(Operator_Stack, Element);
189.              Postfix(Postfix_Index):= Element;
190.              Postfix_Index := Postfix_Index + 1;
191.    end loop;
192.    -- set the remaining spaces to blanks
193.    for I in Postfix_Index .. Stack_Size loop
194.      Postfix(I) := ' ';
195.    end loop;
196.    return Postfix;
197.  end Infix_To_Postfix;
198.
199.  -- function to convert infix to postfix
200.  function Infix_To_Prefix (Infix : String; Length: Integer) return String is
201.  Prefix: String(1.. stack_size);
202.    Prefix_Index : Integer;
203.    Operator_Stack : My_Stack;
204.    Done   : Boolean;
205.    element : character;
206.  begin
207.    Create(Operator_Stack);
208.    Prefix_Index := 1;
209.    for I in reverse 1.. Length loop
210.      -- if it is an operand
211.      if Isoperator(Infix(I)) = False then
212.        -- ignore blank spaces
213.        if (Infix(I) /= ' ') then
```

```
214.              Prefix(Prefix_Index):= Infix(I);
215.              Prefix_Index := Prefix_Index + 1;
216.            end If;
217.
218.          else
219.            -- it is an operator
220.            done:= false;
221.            loop
222.              exit when Done = True;
223.              -- if its a ( or an empty stack then push operator onto stack
224.              if IsEmpty(Operator_Stack) or Infix(I) = ')' then
225.                Push(Operator_Stack, Infix(I));
226.                Done := True;
227.                -- check the precedence of the element on top of the stack and infix(I)
228.              elsif Precedence(Infix(I)) > Precedence(Stacktop(Operator_Stack)) then
229.                Push(Operator_Stack, Infix(I));
230.                Done:= True;
231.              else
232.                -- pop until an operator of smaller precedence appears on the stack
233.                Pop(Operator_Stack, Element);
234.                if Element = ')' then
235.                  Done := True;
236.                else
237.                  Prefix(Prefix_Index):= element;
238.                  Prefix_Index := Prefix_Index + 1;
239.                end if;
240.              end if;
241.            end loop;
242.          end if;
243.      end loop;
244.      -- pop the remaining elements into postfix
245.      while (Isempty(Operator_Stack) = false) loop
246.          Pop(Operator_Stack, Element);
247.                Prefix(Prefix_Index):= Element;
248.                Prefix_Index := Prefix_Index + 1;
249.      end loop;
250.      -- set the remaining spaces to blanks
251.      for I in Prefix_Index .. Stack_Size loop
252.        Prefix(I) := ' ';
253.      end loop;
254.
255.      for I in 1 .. Prefix_Index-1 loop
256.        Push(Operator_Stack, Prefix(I));
257.        end loop;
258.      for I in 1 .. Prefix_Index-1 loop
259.        Pop(Operator_Stack, Prefix(I));
260.        end loop;
261.      return Prefix;
262.    end Infix_To_Prefix;
263.
264. end My_Expression_Converter;
265.
```

265 lines: No errors

Tests

Test Case	Input	Output
1	a+b	ab+, +ab
2	(a+b	Loop until user enters the correct input
3	(a+b)-c	ab+c-, -+abc
4	3+5*6 – 7 * (8+5)	356*+785+*- +3-*56*7+85
5	a + b	ab+, +ab