Chase Brown

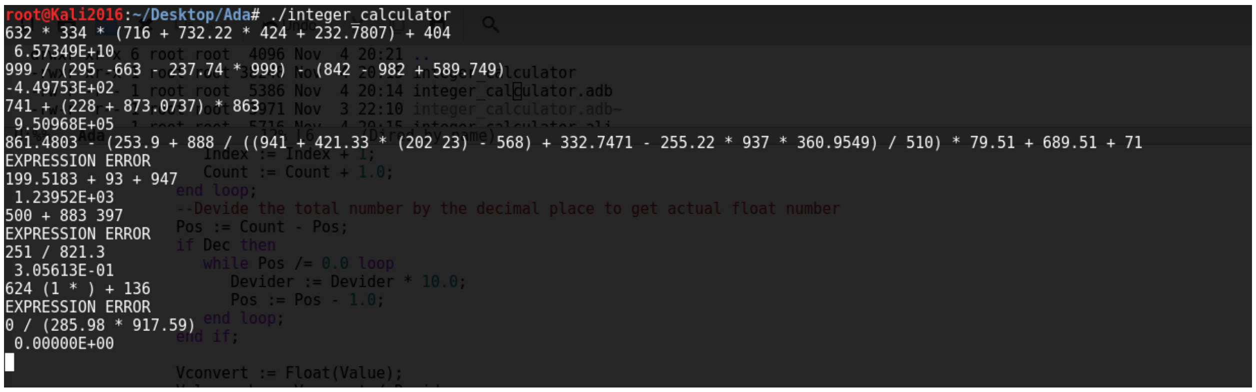
CSCI-3415-001

Program 3 Ada

The purpose of this assignment is to effectively use the concept of a stack. The program is making a simple calculator program in Ada using two stacks to push and pop operands and operators from a string of characters onto two different stacks. The operand stack takes characters that are numbers and the operator stack take the operators '+, -, *, / '. This was later updated to push floats into the operand stack and push parenthesis into the operator stack to create another level of order of operations. The 'calculator' then pops off two operands and an operator and performs an operation based on what the operator is. The product is then pushed into the operand stack and the process repeats until the operator stack is empty and a product is produced.

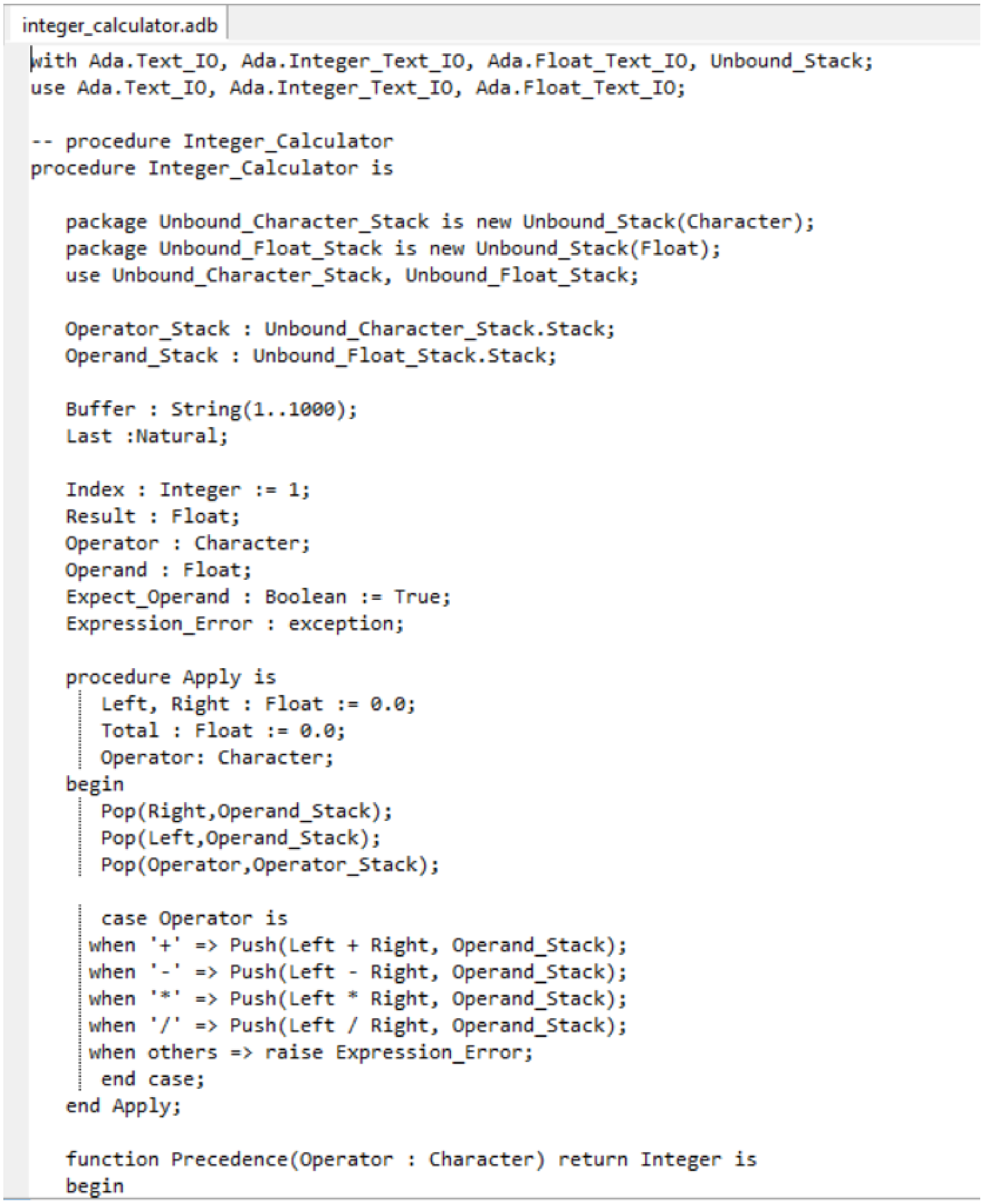
SOURCE CODE BELOW—

To implement this program, I first started with the hint example that was provided. I have never worked with Ada before, so this was very helpful to start understanding the syntax of the language and how to implement it. I began by getting the program to work with integers and simple operators. This was not difficult and simply took a few extra lines of code to get working. I also implemented another function to the stack so that I could look at the top of the operator stack and see if the operator that was pushed onto it was of higher precedence then the operator that was currently being read. This was simply so that I could decide the order of precedence. From there I decided that since we need to implement floating point and parenthesis, I figured that parenthesis would be the harder of the two. I ended up adding a precedence to the parenthesis operators and then added a few when statements to happen if the program comes across parenthesis in the string. It ended up being easier to implement and could move onto the floating-point problem. The swap of most of the integer to float values was simple. I decided that since the reading in was working for integers I simply added some statements. If it finds a decimal skip it but then keep track of the position in the number where the decimal was. I then did a simple float conversion on the integer and divided it by a multiple of 10 based on the position of the decimal to get the correct float number. For example, 20.1 the program reads in 201 and then divides the number 201 by 10 to get the number $2.01E^1$.

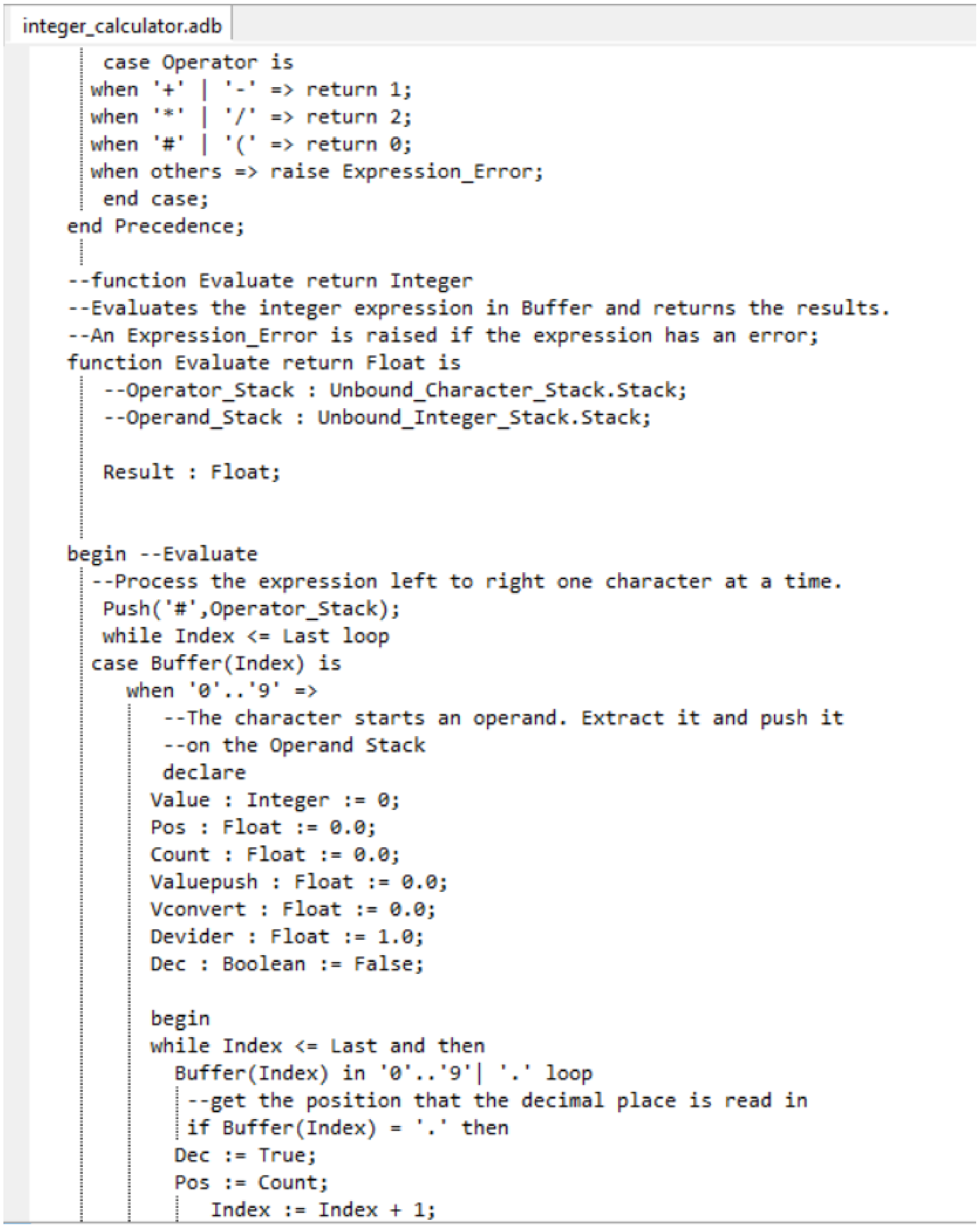


```
root@Kali2016:~/Desktop/Ada# ./integer_calculator
632 * 334 * (716 + 732.22 * 424 + 232.7807) + 404
 6.57349E+10
999 / (295 -663 - 237.74 * 999) - (842 - 982 + 589.749)
-4.49753E+02
741 + (228 + 873.0737) * 863
 9.50968E+05
861.4803 - (253.9 + 888 / ((941 + 421.33 * (202 23) - 568) + 332.7471 - 255.22 * 937 * 360.9549) / 510) * 79.51 + 689.51 + 71
EXPRESSION ERROR
199.5183 + 93 + 947
 1.23952E+03
500 + 883 397
EXPRESSION ERROR
251 / 821.3
 3.05613E-01
624 (1 * ) + 136
EXPRESSION ERROR
0 / (285.98 * 917.59)
 0.00000E+00
```

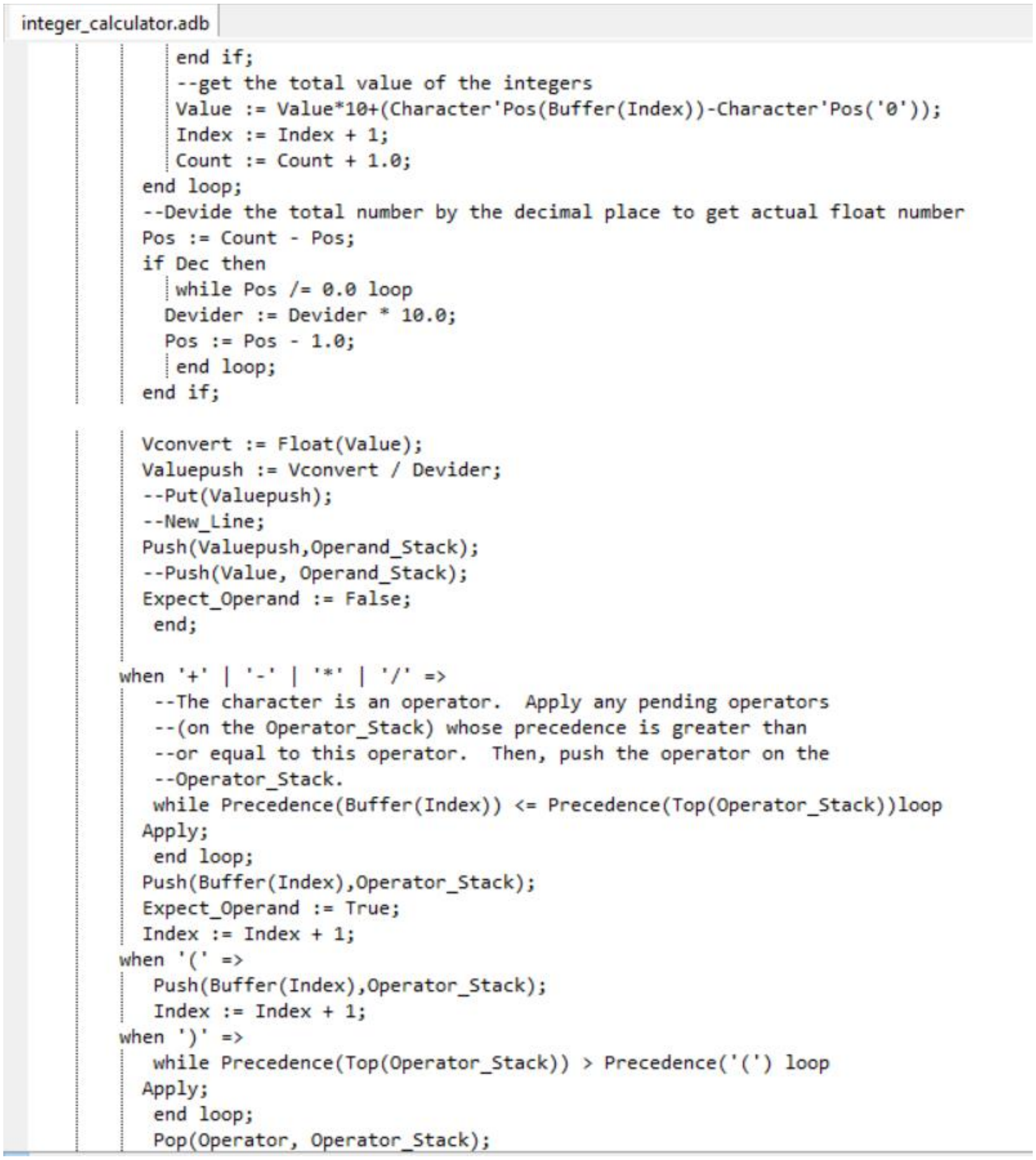
Ada turned out to be a little bit of a frustration for me to work with. I think anytime you must learn a new language there are barriers to get over. I found that the language was not difficult to read or understand but found parts of the language frustrating. The programming was simple and straight forward till I got to the float conversion. While this is usually achieved very easily in most languages for some reason I found it frustrating with Ada. I am not sure if it is just because of how the program syntax works with begins and ends etc. This program was like a C++ calculator program that I did a few years ago so I could remember how to go about completing this assignment. The hint code gave me a good starting point to work with so that I was not stuck figuring out syntax as much. I used a site www.adahome.come/rm95/ for reference material related to Ada for any syntax problems that I ran into and https://en.wikibooks.org/wiki/Ada_Programming/Mathematical_calculations which actually goes over creating a calculator using arrays for referencing ideas on how to complete the program.


integer_calculator.adb

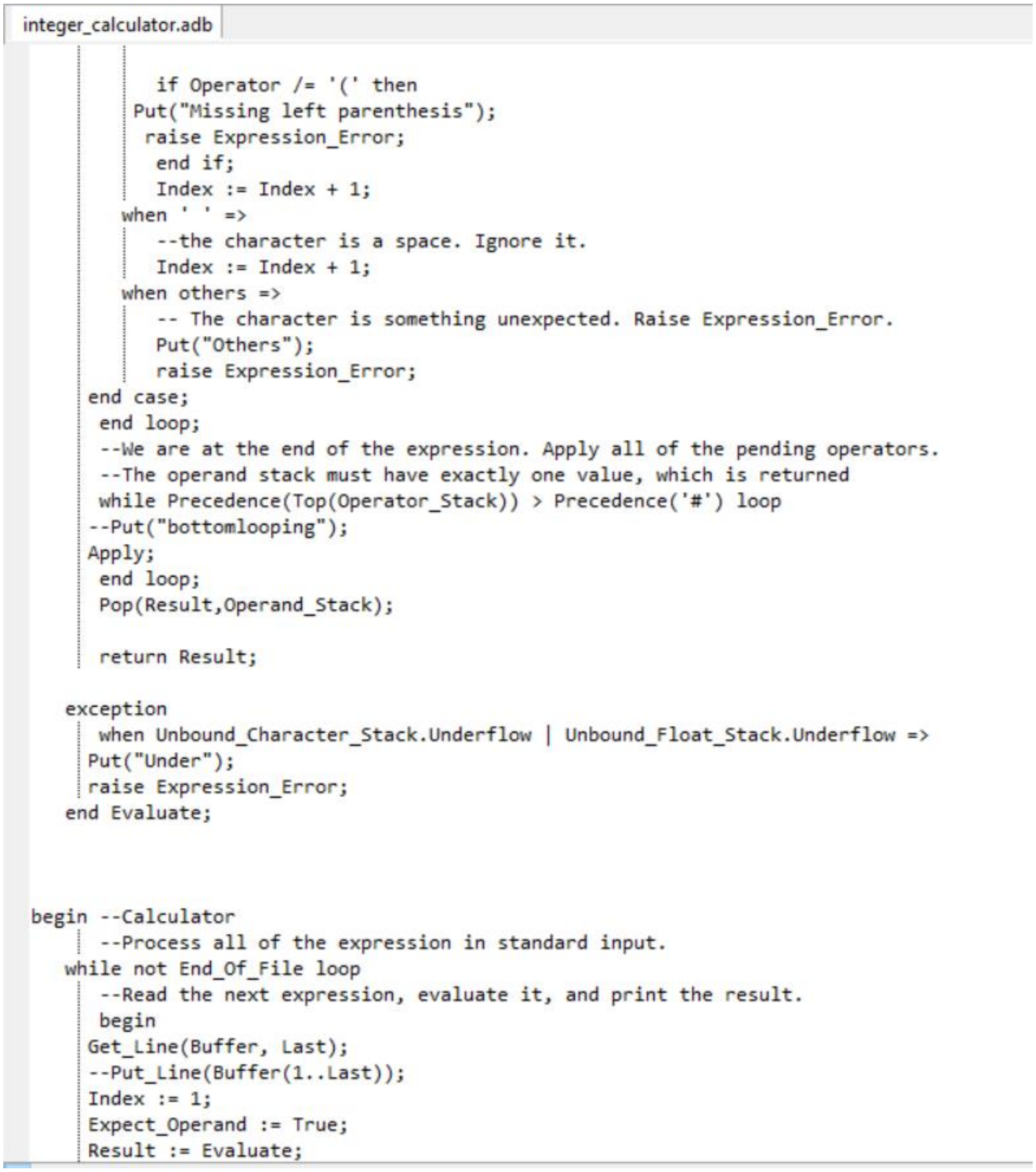
```
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO, Unbound_Stack;
use Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;

-- procedure Integer_Calculator
procedure Integer_Calculator is

   package Unbound_Character_Stack is new Unbound_Stack(Character);
   package Unbound_Float_Stack is new Unbound_Stack(Float);
   use Unbound_Character_Stack, Unbound_Float_Stack;

   Operator_Stack : Unbound_Character_Stack.Stack;
   Operand_Stack : Unbound_Float_Stack.Stack;

   Buffer : String(1..1000);
   Last :Natural;

   Index : Integer := 1;
   Result : Float;
   Operator : Character;
   Operand : Float;
   Expect_Operand : Boolean := True;
   Expression_Error : exception;

   procedure Apply is
      Left, Right : Float := 0.0;
      Total : Float := 0.0;
      Operator: Character;
   begin
      Pop(Right,Operand_Stack);
      Pop(Left,Operand_Stack);
      Pop(Operator,Operator_Stack);

      case Operator is
     when '+' => Push(Left + Right, Operand_Stack);
     when '-' => Push(Left - Right, Operand_Stack);
     when '*' => Push(Left * Right, Operand_Stack);
     when '/' => Push(Left / Right, Operand_Stack);
     when others => raise Expression_Error;
      end case;
   end Apply;

   function Precedence(Operator : Character) return Integer is
   begin
```


integer_calculator.adb

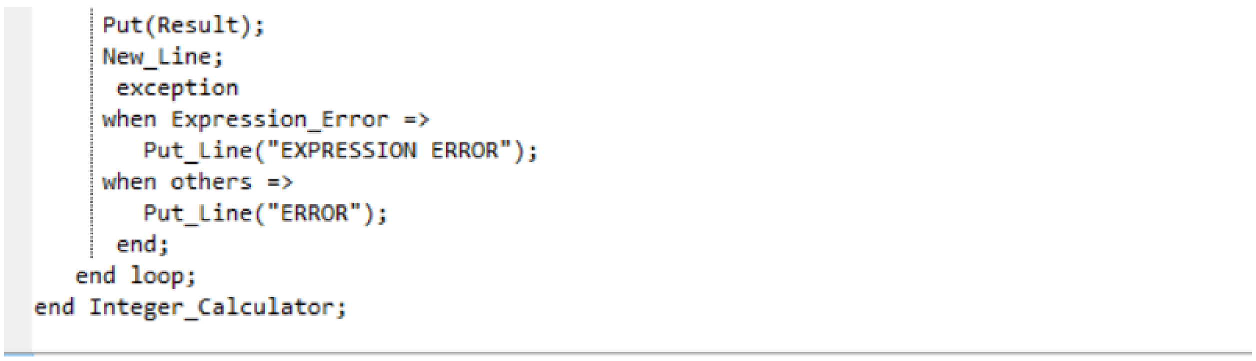
```
      case Operator is
     when '+' | '-' => return 1;
     when '*' | '/' => return 2;
     when '#' | '(' => return 0;
     when others => raise Expression_Error;
      end case;
   end Precedence;

   --function Evaluate return Integer
   --Evaluates the integer expression in Buffer and returns the results.
   --An Expression_Error is raised if the expression has an error;
   function Evaluate return Float is
      --Operator_Stack : Unbound_Character_Stack.Stack;
      --Operand_Stack : Unbound_Integer_Stack.Stack;

      Result : Float;


   begin --Evaluate
     --Process the expression left to right one character at a time.
      Push('#',Operator_Stack);
      while Index <= Last loop
     case Buffer(Index) is
        when '0'..'9' =>
           --The character starts an operand. Extract it and push it
           --on the Operand Stack
           declare
          Value : Integer := 0;
          Pos : Float := 0.0;
          Count : Float := 0.0;
          Valuepush : Float := 0.0;
          Vconvert : Float := 0.0;
          Devider : Float := 1.0;
          Dec : Boolean := False;

          begin
          while Index <= Last and then
            Buffer(Index) in '0'..'9'| '.' loop
             --get the position that the decimal place is read in
             if Buffer(Index) = '.' then
            Dec := True;
            Pos := Count;
               Index := Index + 1;
```


integer_calculator.adb

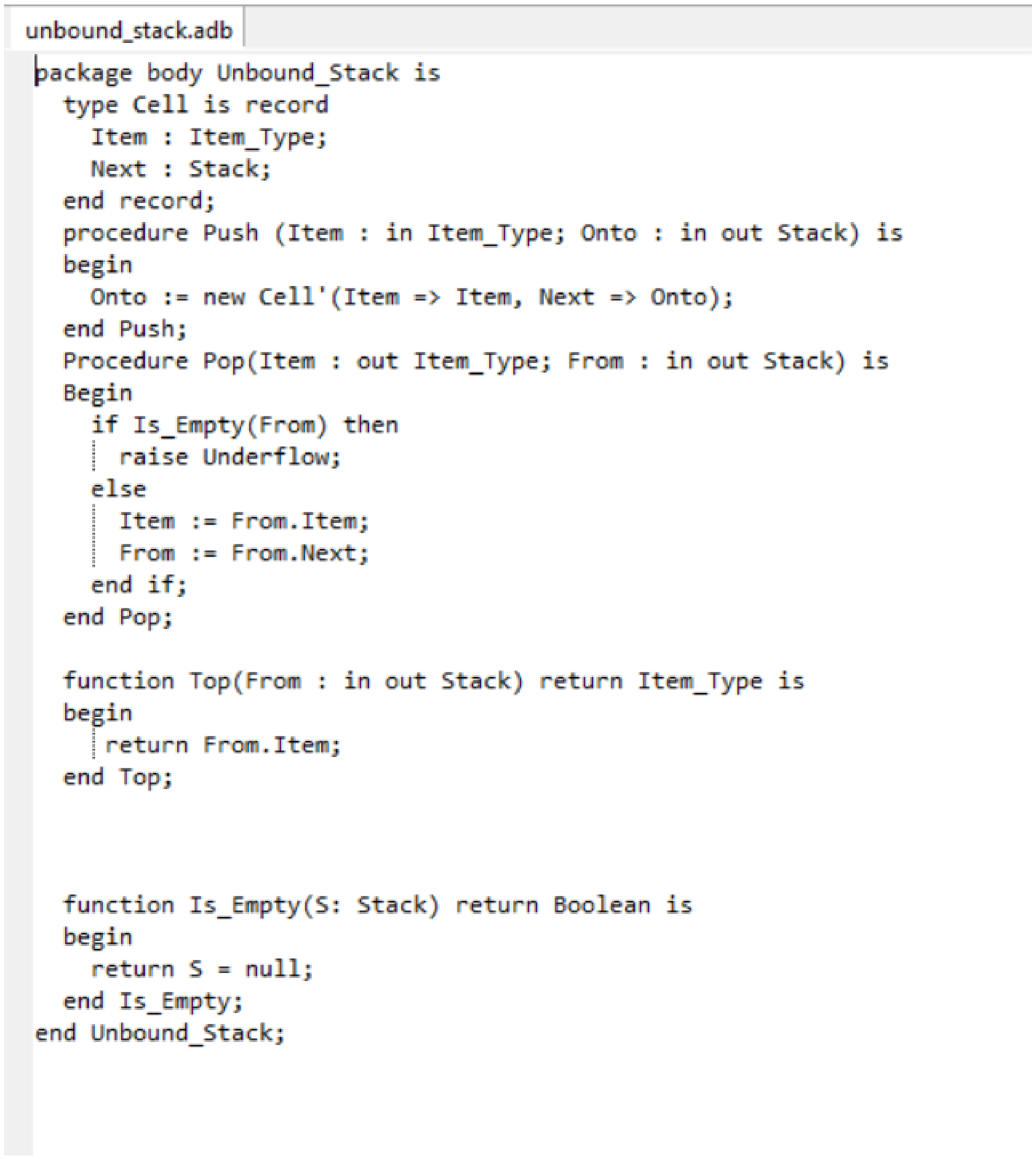
```
          end if;
          --get the total value of the integers
          Value := Value*10+(Character'Pos(Buffer(Index))-Character'Pos('0'));
          Index := Index + 1;
          Count := Count + 1.0;
       end loop;
       --Devide the total number by the decimal place to get actual float number
       Pos := Count - Pos;
       if Dec then
          while Pos /= 0.0 loop
         Devider := Devider * 10.0;
         Pos := Pos - 1.0;
          end loop;
       end if;

       Vconvert := Float(Value);
       Valuepush := Vconvert / Devider;
       --Put(Valuepush);
       --New_Line;
       Push(Valuepush,Operand_Stack);
       --Push(Value, Operand_Stack);
       Expect_Operand := False;
        end;

     when '+' | '-' | '*' | '/' =>
        --The character is an operator.  Apply any pending operators
        --(on the Operator_Stack) whose precedence is greater than
        --or equal to this operator.  Then, push the operator on the
        --Operator_Stack.
        while Precedence(Buffer(Index)) <= Precedence(Top(Operator_Stack))loop
       Apply;
        end loop;
       Push(Buffer(Index),Operator_Stack);
       Expect_Operand := True;
       Index := Index + 1;
     when '(' =>
        Push(Buffer(Index),Operator_Stack);
        Index := Index + 1;
     when ')' =>
        while Precedence(Top(Operator_Stack)) > Precedence('(') loop
       Apply;
        end loop;
        Pop(Operator, Operator_Stack);
```


integer_calculator.adb

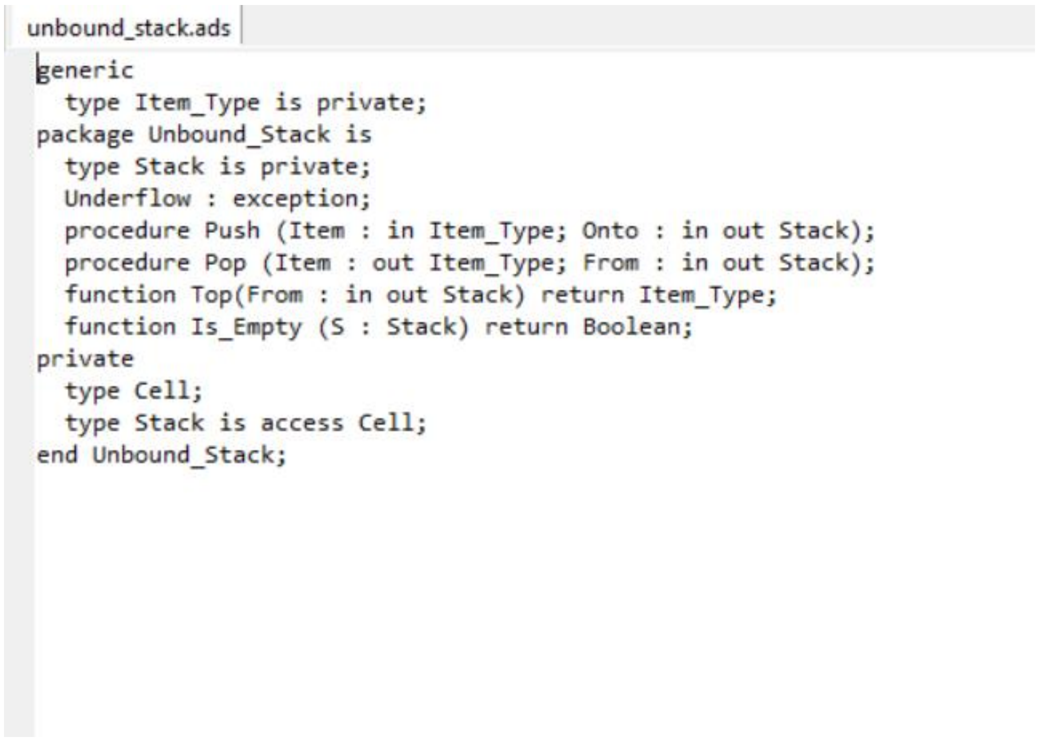
```
         if Operator /= '(' then
       Put("Missing left parenthesis");
        raise Expression_Error;
         end if;
         Index := Index + 1;
      when ' ' =>
         --the character is a space. Ignore it.
         Index := Index + 1;
      when others =>
         -- The character is something unexpected. Raise Expression_Error.
         Put("Others");
         raise Expression_Error;
   end case;
    end loop;
    --We are at the end of the expression. Apply all of the pending operators.
    --The operand stack must have exactly one value, which is returned
    while Precedence(Top(Operator_Stack)) > Precedence('#') loop
   --Put("bottomlooping");
   Apply;
    end loop;
    Pop(Result,Operand_Stack);

    return Result;

  exception
    when Unbound_Character_Stack.Underflow | Unbound_Float_Stack.Underflow =>
   Put("Under");
   raise Expression_Error;
  end Evaluate;



begin --Calculator
    --Process all of the expression in standard input.
  while not End_Of_File loop
    --Read the next expression, evaluate it, and print the result.
    begin
   Get_Line(Buffer, Last);
   --Put_Line(Buffer(1..Last));
   Index := 1;
   Expect_Operand := True;
   Result := Evaluate;
```



```
      Put(Result);
      New_Line;
       exception
      when Expression_Error =>
         Put_Line("EXPRESSION ERROR");
      when others =>
         Put_Line("ERROR");
       end;
   end loop;
end Integer_Calculator;
```



```
package body Unbound_Stack is
  type Cell is record
    Item : Item_Type;
    Next : Stack;
  end record;
  procedure Push (Item : in Item_Type; Onto : in out Stack) is
  begin
    Onto := new Cell'(Item => Item, Next => Onto);
  end Push;
  Procedure Pop(Item : out Item_Type; From : in out Stack) is
  Begin
    if Is_Empty(From) then
      raise Underflow;
    else
      Item := From.Item;
      From := From.Next;
    end if;
  end Pop;

  function Top(From : in out Stack) return Item_Type is
  begin
     return From.Item;
  end Top;



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


unbound_stack.ads

```
generic
  type Item_Type is private;
package Unbound_Stack is
  type Stack is private;
  Underflow : exception;
  procedure Push (Item : in Item_Type; Onto : in out Stack);
  procedure Pop (Item : out Item_Type; From : in out Stack);
  function Top(From : in out Stack) return Item_Type;
  function Is_Empty (S : Stack) return Boolean;
private
  type Cell;
  type Stack is access Cell;
end Unbound_Stack;
```