CSci 340, Spring 2003, Project 0

This project is due *Friday, January 31* at 11:20am. To submit your final solution, run the "`handincs 340 1`" command and submit a paper copy of your well-documented, well-written code for evaluation.

To begin work on this assignment, run the command "`getcs 340 1`". This will place many useful files into your `CS340/Lab1` directory.

`Makefile`	data explaining how to make the program
`scan.ad[bs]`	Token_Type type; `Next_Token` and `Token_Error` procedures
`parse_tree.ad[bs]`	Parse_Node type; `New_Parse_???` functions; `Print_Parse_Tree` procedure
`variables.ad[bs]`	Variable_Type and Variable_Set types; `Empty_Variable_Set` and `Find_Variable` functions; `Add_Variable` procedure
`parse.ad[bs]`	`Parse_File` function (*you will complete this file*)
`typecheck.ad[bs]`	`Type_Check` function (*you will complete this file*)
`main.adb`	`Main` procedure

Your job is to complete `parse.adb` and `typecheck.adb`. While you are welcome to modify the other files, none of this code will be considered part of your assignment solution.

Given an input file, your "compiler" either should print the program's abstract syntax tree if the program passes compilation and an error message if the input program is not in this subset language or has a type error. Printing the abstract syntax tree is not a particularly useful thing to do; in the second project, we'll complete the project to do something more useful with the syntax tree.

The distributed code already contains code for handling the ***program***, ***vardecs***, and ***factor*** metavariables. Your job in this project is to write functions for the other metavariables to complete the job of building the parse tree. You will also complete the `Type_Check` function of `typecheck.adb` to verify that the generated parse tree uses types correctly.

In this project, you will build a parser for the subset of Ada described by the following EBNF grammar.

program	→	`procedure` *identifier* `is` `begin` ***vardecs*** `begin` ***stmts*** `end` *identifier semicolon*
vardecs	→	*identifier colon* (`integer` | `boolean`) *semicolon* ***vardecs*** | ε
stmts	→	***stmt stmts*** | ***stmt***
stmt	→	`while` ***expr*** `loop` ***stmts*** `end` `loop` *semicolon*
	|	`if` ***expr*** `then` ***stmts*** `end` `if` *semicolon*
	|	*identifier assignment* ***expr*** *semicolon*
	|	`get` *left_parenthesis identifier right_parenthesis semicolon*
	|	`put` *left_parenthesis* ***expr*** *right_parenthesis semicolon*
expr	→	***arith*** [(*is_equal* | *is_not_equal* | *is_less_than* | *is_greater_than*) ***arith***]
arith	→	***term*** { (*plus* | *minus*) ***term*** }
term	→	***factor*** { (*star* | *slash*) ***factor*** }
factor	→	*number* | *identifier* | `true` | `false` | *left_parenthesis* ***expr*** *right_parenthesis*

There are many things that this language does not include, and so your compiler doesn't need to consider them either. Among these are nested procedures, `else` clauses, and the negation operator.

This language supports only Integer and Boolean types for its variables. A program using types correctly obeys the following rules.

- The `+`, `-`, `*`, `/`, `<`, and `>` can operate only on integers.
- The `=` and `/=` operators should have the same type on both sides (Integer or Boolean).

- In an assignment statement, the type of the expression must match the assigned variable's type.
- The Get and Put statements must work only with integers.
- The expressions after `if` and `while` must be Boolean expressions.

The following is an example of compiling and running the compiler.

```
unix% cat test.ada
procedure Test is
    I : Integer;
begin
    I := 0;
    while I < 10 loop
        I := I + 1;
        Put(I);
    end loop;
end Test;
unix% make
unix% ./main test.ada
SEQ
  ASSIGNMENT_STMT(I)
    INTEGER_VALUE( 0)
  WHILE_STMT
    ISLT_OP
      IDENTIFIER(I)
      INTEGER_VALUE( 10)
    SEQ
      ASSIGNMENT_STMT(I)
        ADD_OP
          IDENTIFIER(I)
          INTEGER_VALUE( 1)
      PUT_STMT
        IDENTIFIER(I)
```

This output represents the following abstract syntax tree.