CSE340 Fall 2014 - Project 3: Semantic Analysis
Simple Hindley-Milner Type Checking

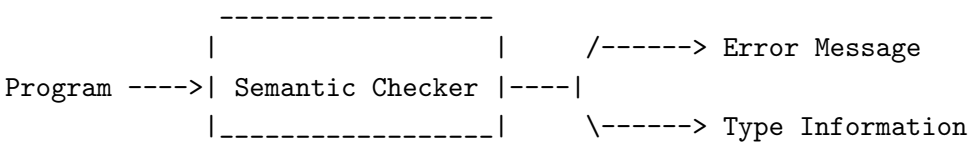
Due: Nov. 3, 2014 by 11:59 PM

Abstract

The goal of this project is to finish an incomplete parser and write a *semantic checker* for a language. The input to the semantic checker will be a program and the output will be information about the types of the variables in the program or an error message if there is type mismatch or other semantic error. This project is conceptually more involved than previous projects. It will take around 20 hours to finish this project.

1 Overview

The goal of this project is to finish an incomplete parser and write a semantic checker for a language. The input to the semantic checker will be a program and the output will be information about the types of the variables in the program or an error message if there is type mismatch or other semantic error. This is an illustration:



The semantic checker enforces semantic rules on the input program. The rules constrain variable usage; depending on the type of a variable, some operations on the variable are allowed or disallowed.

You are provided with an incomplete parser for a language and asked to complete it and enforce the semantic rules of the language. The language is specified by a grammar (see below) and the parser we provide already parses all rules except `while_stmt` and `condition`. Completing the parser means adding code to parse `while_stmt` and `condition`. **You are not allowed to modify other parts of the provided parser. In particular, you should not modify how expressions are parsed.**

The provided code also builds a parse tree. While you do not need to use a parse tree for your solution, looking at the implementation can be useful. The parse tree building is also not complete! It does not handle `while_stmt` and `condition`.

The provided parser also rewrites the input program so that all expressions are changed to prefix notation. This functionality is also not needed for your implementation but it can be useful as an example.

The language and the rules to be enforced are described in the following sections.

2 Grammar

The grammar of the language is the following:

```
(1)  program             -> decl  body
(2)  decl                -> type_decl_section  var_decl_section
(3)  type_decl_section   -> TYPE type_decl_list
(4)  type_decl_section   ->                              // epsilon
(5)  type_decl_list      -> type_decl type_decl_list
(6)  type_decl_list      -> type_decl
(7)  type_decl           -> id_list COLON type_name SEMICOLON
(8)  type_name           -> REAL
(9)  type_name           -> INT
(10) type_name           -> BOOLEAN
(11) type_name           -> STRING
(12) type_name           -> ID
(13) var_decl_section    -> VAR var_decl_list
(14) var_decl_section    ->                             // epsilon
(15) var_decl_list       -> var_decl var_decl_list
(16) var_decl_list       -> var_decl
(17) var_decl            -> id_list COLON type_name SEMICOLON
(18) id_list             -> ID COMMA id_list
(19) id_list             -> ID
(20) body                -> LBRACE stmt_list RBRACE
(21) stmt_list           -> stmt stmt_list
(22) stmt_list           -> stmt
(23) stmt                -> while_stmt
(24) stmt                -> assign_stmt
(25) while_stmt          -> WHILE condition body
(26) assign_stmt         -> ID EQUAL expr SEMICOLON
(27) expr                -> term PLUS expr
(28) expr                -> term MINUS expr
(29) expr                -> term
(30) term                -> factor MULT term
(31) term                -> factor DIV term
(32) term                -> factor
(33) factor              -> LPAREN expr RPAREN
(34) factor              -> NUM
```

```
(35) factor             -> REALNUM
(36) factor             -> ID
(37) condition          -> ID
(38) condition          -> primary relop primary
(39) primary            -> ID
(40) primary            -> NUM
(41) primary            -> REALNUM
(42) relop              -> GREATER
(43) relop              -> GTEQ
(44) relop              -> LESS
(45) relop              -> NOTEQUAL
(46) relop              -> LTEQ
```

The following sections will go into the details of semantic checking that needs to be done.

3 Program

A program consists of two sections: a declaration section (`decl`) that contains type and variable declarations and a body section (`body`) that contains statements of the program. The top-level grammar rules for a program are:

```
(1) program             -> decl  body
(2) decl                -> type_decl_section  var_decl_section
```

In what follows we elaborate on each program section.

3.1 Types and Variables

3.1.1 Types

The language has a number of built-in types. These are: `INT`, `REAL`, `BOOLEAN`, and `STRING`. In addition to the built-in types, the language allows the declaration of user-defined types. User-defined types are declared in a `type_decl_section` (type declaration section). The syntax for type declarations is given by the following grammar rules:

```
(3)  type_decl_section -> TYPE type_decl_list
(4)  type_decl_section ->                              // epsilon
(5)  type_decl_list    -> type_decl type_decl_list
(6)  type_decl_list    -> type_decl
(7)  type_decl         -> id_list COLON type_name SEMICOLON
(8)  type_name         -> REAL
(9)  type_name         -> INT
(10) type_name         -> BOOLEAN
```

```
(11) type_name          -> STRING
(12) type_name          -> ID
```

The rules for `id_list` are rules (18) and (19) above. They are also repeated below with variable declarations.

User-declared types can be explicit or implicit:

- ⋆ **Explicit user-declared types** are names that are not built-in types and that have their first appearance in the program as part of `id_list` of a declaration according to rule (7).
- ⋆ **Implicit user-declared types** are not built-in types and not explicit user-declared types. Implicit user-declared types have their first appearance as `type_name` in a declaration according to rules (7) or (17) (see below also).

Example: The following is an example program. We added line numbers to more easily refer to the declarations and statement. Line numbers are not part of the input.

In the example, types `at` (first appearance in line 02) and `bt` (first appearance in line 03) are explicitly declared. `at` is explicitly declared because it is not the name of a built-in type and it first appears in `at : INT;` (line 02). In the parsing of the program `at : INT;` is parsed as rule (7) and `at` is part of the `id_list`.

Type `ct` (first appearance in line 07) is an implicit user-declared type because it has its first appearance as `type_name` in `bv : ct;` which is parsed according to rule (17) for variable declarations.

```
01:   TYPE
02:     at : INT;
03:     bt : at;
04:
05:   VAR
06:     av : at;
07:     bv : ct;
08:   {
09:     av = 1;
10:     bv = av + lv;
11:     while jv < iv
12:     {
13:        av = av+1;
14:     }
15:   }
```

Constraints for type declaration section

(C1) Declaration of a built-in type should result in a syntax error. This is already handled by the parser that I provide.

(C2) A user-declared type name should not be declared more than once in the `TYPE` section. The following should hold:

- ⋆ An explicit declared type should not appear in two different `id_list`(s) (in the `TYPE` section) or more than once in the same `id_list` (in the `TYPE` section).
- ⋆ An implicit user-declared type should not appear in an `id_list` of a declaration according to (7).

Example In the following example `at` appears in two different `id_list`(s) in the `TYPE` section, thereby rule (C2) is violated. Also, `et` appears twice in the same `id_list` in the `TYPE` section, a violation of rule (C2). Built-in type `REAL` is explicitly declared (line 04), thereby rule (C1) is violated. Finally, implicitly user-declared type `bt` appears in an `id_list` of a declaration according to (7) (line 05) which is a violation of rule (C2).

```
01:   TYPE
02:     at,et,et : INT;
03:     at       : bt;
04:     REAL     : at;
05:     bt       : dt;
06:
07:   VAR
08:     av : at;
09:     bv : ct;
09:   {
10:     av = 1;
11:   }
```

3.2 Variables

The language allows the explicit declaration of variables. This is done in a `var_decl_section` (variable declaration section) whose syntax is defined as follows:

```
(13) var_decl_section  -> VAR var_decl_list
(14) var_decl_section  ->                          // epsilon
(15) var_decl_list     -> var_decl var_decl_list
(16) var_decl_list     -> var_decl
(17) var_decl          -> id_list COLON type_name SEMICOLON
(18) id_list           -> ID COMMA id_list
(19) id_list           -> ID
```

Variables are declared explicitly or implicitly: A variable is declared explicitly if it appears in an `id_list` in a declaration according to (17). A variable is declared implicitly if it is not declared explicitly but appears in the program body (see below).

Example: In the following example `av` and `bv` are declared explicitly. Variable `ev` is declared implicitly.

```
01:   VAR
02:     av : at;
03:     bv : ct;
04:   {
05:     av = 1;
06:     ev = 2;
07:   }
```

Constraints for variable declaration section

(C3) Types declared in TYPE section should not be re-declared as a variable. A type name that is introduced in a `type_decl` (implicit or explicit) should not appear as part of an `id_list` of rule (17)

(C4) Multiple declarations of the same variable name not allowed. The same variable name should not appear in two different `var_decl` or more than once in the same `var_decl`.

(C5) Types declared in VAR section should not be re-declared as a variable. A name that first appears as `type_name` in rule (17) should not be declared as a variable (note that a name can appear as a `type_name` without being explicitly or implicitly declared as a type in the `type_decl_section`. This

rule is the same as rule (C3) but applied to implicitly declared types that are first introduced in the `var_decl_section`).

(C6) Variables should not be re-declared as type names. A name that first appears in an `id_list` of rule (17) should not appear as a `type_name` in rule (17).

Example: In the following example the re-declaration of the explicitly-declared type `bt` (first introduced in line 03) as a variable in line 06 violates rule (C3). The re-declaration of the implicitly declared type `it` in line 05 is also a violation of rule (C3).

In the example, the two declarations of `av` in line 05 is a violation of rule (C4). The re-declaration of `bv` (first declared in line 05) in line 06 is also a violation of rule (C4). The re-declaration of implicit type `dt` (first introduced in line 05) as a variable in line 06 is a violation of rule (C5). The re-declaration of variable `x` (first introduced in line 07) as a type name in line 08 is a violation of rule (C6).

```
01:   TYPE
02:     at : INT;
03:     bt : it;
04:   VAR
05:     it, av, av, bv : dt;
06:     bt, bv, dt    : INT;
07:     x, y : REAL;
08:     z : x;
09:   {
10:     av = 1;
11:     ev = 2;
12:   }
```

4 Program Body

The body of the program consists of a list of statements between two braces. The grammar rules are:

```
(20) body              -> LBRACE stmt_list RBRACE
(21) stmt_list         -> stmt stmt_list
(22) stmt_list         -> stmt
```

5 Statements

A statement can be either an assignment (`assign_stmt`) or a while statement (`while_stmt`). An `assign_stmt` assigns an expression to a variable. A while statement has 3 parts:

(1) the `WHILE` keyword, (2) a `condition`, and (3) a `body` (this is a recursive definition). The grammar rules for `stmt` are the following:

```
(23) stmt             -> while_stmt
(24) stmt             -> assign_stmt
(25) while_stmt       -> WHILE condition body
(26) assign_stmt      -> ID EQUAL expr SEMICOLON
(27) expr             -> term PLUS expr
(28) expr             -> term MINUS expr
(29) expr             -> term
(30) term             -> factor MULT term
(31) term             -> factor DIV term
(32) term             -> factor
(33) factor           -> LPAREN expr RPAREN
(34) factor           -> NUM
(35) factor           -> REALNUM
(36) factor           -> ID
(37) condition        -> ID
(38) condition        -> primary relop primary
(39) primary          -> ID
(40) primary          -> NUM
(41) primary          -> REALNUM
(42) relop            -> GREATER
(43) relop            -> GTEQ
(44) relop            -> LESS
(45) relop            -> NOTEQUAL
(46) relop            -> LTEQ
```

Constraints on statements and expressions

(C7) Assignment cannot be made between variables of different types

(C8) Operations (`PLUS`, `MINUS`, `MULT`, `DIV`) cannot be applied to operands of different types (but can be applied to operands of same type including `STRING` and `BOOLEAN`)

(C9) Relational operators (`relop`) cannot be applied to operands of different types (but can be applied to operands of same type including `STRING` and `BOOLEAN`)

(C10) `NUM` constants are of type `INT`

(C11) `REALNUM` constants are of type `REAL`

(C12) The result of `primary relop primary` is of type `BOOLEAN` (both primaries have to have the same type)

(C13) `condition` should be of type `BOOLEAN` (this is automatically satisfied for rule (38) when both primaries have the same type)

(C14) The type of an expression is the same as the type of its operands (which must have the same type by (C8))

(C15) Names that are explicitly or implicitly declared as types should not appear in the program body. This error is considered a type name re-declared as variable.

6 Requirements

The goal is to complete the parser and write a type checker. The type checker will enforce all the constraints above and the constraints described in this section. In this section, I will just describe the constraints. Examples are given in the next section.

The goal of the type checker is to determine which types and variables have the same "type". We will express the type of a type name or a variable name as an integer number. Built-in types have predefined numbers. The goal of the type checker is to assign type numbers to user defined types and to variables so that names that have the same "type" also have the same type number.

We have the following main rules (MR) to be enforced:

(MR1) Structural equivalence is enforced.

(MR2) Each type is assigned an integer value (type number).

(MR3) Each variable is assigned an integer type number.

(MR4) Each variable (whether or not it is explicitly declared) has a fixed "type" that does not change. This means that there should be one type number that is assigned to the variable in the whole program and that is consistent with the declaration of the variable and all uses of the variable.

(MR5) Each type name (whether or not it is explicitly declared) has a fixed "type" that does not change. This means that there should be one type number that can be assigned to the type name and that is consistent with the declaration of the type name and all the uses of the type name.

(MR6) If `a : b;` is a type declaration, then `#a = #b`, where `#x` is the integer value assigned to `x`.

(MR7) If `a : t;` is a variable declaration, then `#a = #t`, where `#x` is the number assigned to `x`.

7 Implementation Suggestions

As the declaration sections are parsed, type numbers are assigned to type names and variable names in a way that satisfy the constraints above.

(IS1) For built-in types, you can use fixed numbers. We suggest the following numbers: `INT` is assigned 10, `REAL` is assigned 11, `STRING` is assigned 12, and `BOOLEAN` is assigned 13.

(IS2) Implicit types are assigned unique numbers when they are introduced. The numbers should be greater than 13.

(IS3) Variables that are implicitly declared are assigned unique numbers when they are introduced. The numbers should be greater than 13.

When an assignment statement is parsed, the numbers of the left hand side and the right hand side should be the same. If the two sides have the same number, nothing needs to be done (constraint (C7) is satisfied). Otherwise, due to rule (C7), we need to make sure they get the same number. There are three cases to consider:

(IS4) The two numbers are numbers of two different built-in types: type mismatch error, since two built-in types cannot have the same type number.

(IS5) One number is the number of a built-in type and the other number is the number of an implicit type: The implicit type number should be changed to match that of the built-in type.

(IS6) The two numbers are the numbers of implicit types: One of the implicit type numbers should be changed to match the other one.

Similar changes are done when processing conditions or expressions. This is illustrated in a detailed example below.

8 WHAT YOUR PROGRAM SHOULD DO?

Analyze the input program and either:

1. Print out semantic error message if there is an error

   OR

2. For every type and variable, print all the variables and types with the same type number together. The detailed format is given below

If there is a semantic error, you should not print any variables or types. As in all projects in this class, your program should read input from standard input, and write output to standard output.

OUTPUT

Error codes:

- ⋆ type name declared more than once: `ERROR CODE 0`
- ⋆ type re-declared as variable: `ERROR CODE 1`
- ⋆ variable declared more than once: `ERROR CODE 2`
- ⋆ type mismatch: `ERROR CODE 3`
- ⋆ variable re-declared as a type name: `ERROR CODE 4`

Non-error output:

- ⋆ For every type name that is built-in, explicitly or implicitly declared, the semantic checker should output, in the order they are introduced in the program, all type names (built-in first, then explicit, then implicit) and variable names (explicit first then implicit) that have the same integer value (type number). The format is:

```
typename_1 : typename_2 typename_3 ... typename_n variablename_1
             variablename_2  ... variablename_k #
```

  Note that if built-in types appear in a list, they always appear to the left of a colon.

  Your should read this rule very carefully. It is not obvious. Read exactly what it says.

- ⋆ For built-in types the order is according to their type numbers, e.g. 10, 11,... So you would list `INT` before `REAL` in the output.
- ⋆ If a type name already appears in a list do not create a list for it
- ⋆ For every variable name that is implicitly declared, output, in the order they are introduced in the program all variable names that have the same integer number. Do not create a list for an implicitly declared variable if it already appears in an earlier list. The format is:

```
variablename_1 : variablename_2 variablename_3 ... variablenam_l #
```

  Note that explicitly-declared variables will already appear in one of the "type" lists and therefore do not need separate lists.

9 Examples

Here are some example input programs and detailed explanations of type checking procedure. Note that gray boxes are input programs and brown boxes are output boxes.

```
{
  a = b;
}
```

In this example:

- ⋆ type of a = 14
- ⋆ type of b = 15

⇒ we change type of `a` to 15 because of the assignment statement.
Output:

```
a : b #
```

Note that no built-in types are listed in the output because there are no variables or types that have the same type numbers as built-in types.

```
VAR
  a : INT;
  b : REAL;
{
  a = b;
}
```

In this example:

- ⋆ type of a = 10
- ⋆ type of b = 11

⇒ type mismatch (assignment statement)

Output:

```
ERROR CODE 3
```

```
{
   while a
   {
     b = b+1;
   }
}
```

In this example:

- ⋆ type of a = 13 (BOOLEAN) since it is used as a condition
- ⋆ type of b = 10 (INT) since it is used in an expr of type INT

⇒ type of a = 13 and type of b = 10

Output:

```
INT : b #
BOOLEAN : a #
```

Note that in the above example there is no output for REAL or STRING because there are no variables or types that have the same type number as REAL or STRING.

```
{
  a = 1;
  b = a;
  while b
  {
     c = a;
  }
}
```

In this example:

- ⋆ a is INT (first statement)
- ⋆ b is INT (from second statement)
- ⋆ b is BOOLEAN (from the condition of the while)

⇒ type mismatch

Output:

```
ERROR CODE 3
```

```
{
  a = b+c;
  b = 1;
  c = 1.5;
}
```

In this example:

- ⋆ a, b, and c are the same type (first statement)
- ⋆ b is INT (second statement) this also makes a and c INT
- ⋆ c is REAL (from the third statement), type mismatch between INT and REAL

⇒ type mismatch

Output:

```
ERROR CODE 3
```

Detailed example

Note that line numbers are for reference only and not part of the input.

```
01:   TYPE
02:     at : INT;
03:     bt : at;
04:     ct : dt;
05:     et : BOOLEAN;
06:     ft : STRING;
07:     gt : REAL;
08:
09:   VAR
10:     av : at;
11:     bv : bt;
12:     cv : ct;
13:     ev : et;
14:     fv : ft;
15:     gv : gt;
16:     hv : t1;
17:     iv : t1;
18:     jv : t2
19:     kv : t2;
20:     lv : t3;
21:
22:   {
23:     av = 1;
24:     bv = av + lv;
25:     while jv < iv
26:     {
27:        av = av+1;
28:     }
29:
30:     while jv
31:     {
32:        av = av+1;
33:     }
34:
35:     kv = kv+1;
36:     bv = av;
37:     cv = 1.0;
38:     cv = bv;
39:     mv = nv;
40:   }
```

Here is a line-by-line type analysis of the above program. Note that there is a type mismatch at line 35 but the analysis is continued after that line, this is just for information purposes and normally the type checking should be stopped after detecting an error.

```
line 02: at : INT;
         #at = #INT              // by (MR6)
         #INT = 10               // INT is a built-in type. See (IS1)
         #at <- 10

line 03: bt : at;
         #bt = #at               // by (MR6)
         #at = 10                // assigned previously (line 2)
         #bt <- 10

line 04: ct : dt;
         #ct = #dt               // by (MR6)
         #dt <- 14               // dt is an implicit type. See (IS2)
         #ct <- 14

line 05: et : BOOLEAN;
         #et = #BOOLEAN          // by (MR6)
         #BOOLEAN = 13           // BOOLEAN is a built-in type. See (IS1)
         #et <- 13

line 06: ft : STRING;
         #ft = #STRING           // by (MR6)
         #STRING = 12            // STRING is a built-in type. See (IS1)
         #ft <- 12

line 07: gt : REAL;
         #gt = #REAL             // by (MR6)
         #REAL = 11              // REAL is a built-in type. See (IS1)
         #gt <- 11

line 10: av : at;
         #av = #at               // by (MR7)
         #at = 10                // assigned previously (line 2)
         #av <- 10

line 11: bv : bt;
         #bv = #bt               // by (MR7)
         #bt = 10                // assigned previously (line 3)
         #bv <- 10
```

```
line 12: cv : ct;
         #cv = #ct              // by (MR7)
         #ct = 14               // assigned previously (line 4)
         #cv <- 14              // note that the base type is unknown

line 13: ev : et;
         #ev = #et              // by (MR7)
         #et = 13               // assigned previously (line 5)
         #ev <- 13

line 14: fv : ft;
         #fv = #ft              // by (MR7)
         #ft = 12               // assigned previously (line 6)
         #fv <- 12

line 15: gv : gt;
         #gv = #gt              // by (MR7)
         #gt = 11               // assigned previously (line 7)
         #gv <- 11

line 16: hv : t1;
         #hv = #t1              // by (MR7)
         #t1 <- 15              // t1 is an implicit type. See (IS2)
         #hv <- 15

line 17: iv : t1;
         #iv = #t1              // by (MR7)
         #t1 = 15               // assigned previously (line 16)
         #iv <- 15

line 18: jv : t2
         #jv = #t2              // by (MR7)
         #t2 <- 16              // t2 is an implicit type. See (IS2)
         #jv <- 16

line 19: kv : t2;
         #kv = #t2              // by (MR7)
         #t2 = 16               // assigned previously (line 18)
         #kv <- 16

line 20: lv : t3;
         #lv = #t3              // by (MR7)
         #t3 <- 17              // t3 is an implicit type. See (IS2)
         #lv <- 17
```

```
line 23: av = 1;
         #(1) = 10              // by (C10)
         #av = 10               // assigned previously in line 10
         no mismatch            // no rules violated

line 24: bv = av + lv;
         #av = 10               // assigned previously in line 10
         #lv = 17               // assigned previously in line 20
         #av = #lv              // by (C8)
         #lv <- 10              // See (IS5)
         #t3 <- 10              // since #lv = #t3
         #bv = 10               // assigned previously in line 11
         no mismatch            // rule (C7) applies

line 25: while jv < iv
         #jv = 16               // assigned previously in line 18
         #iv = 15               // assigned previously in line 17
         #jv = #iv              // by (C9)
         #jv <- 15              // See (IS6)
         #t2 <- 15              // since #jv = #t2
         #kv <- 15              // since #kv = #t2

line 27: av = av+1;
         #av = 10               // assigned previously in line 10
         #(1) = 10              // by (C10)
         no mismatch            // rules (C7) and (C8) apply

line 30: while jv
         #jv = #BOOLEAN         // by (C13)
         #jv = 15               // assigned previously in line 25
         #BOOLEAN = 13          // BOOLEAN is a built-in type. See (IS1)
         #jv <- 13              // See (IS5)
         #t1 <- 13              // change all 15's to 13's
         #t2 <- 13              // for all types and
         #hv <- 13              // variables
         #iv <- 13
         #kv <- 13

line 32: av = av+1;
         #av = 10               // assigned previously in line 10
         #(1) = 10              // by (C10)
         no mismatch            // rules (C7) and (C8) apply
```

```
line 35: kv = kv+1;
         #kv = 13              // from line 30
         #(1) = 10             // by (C10)
         #kv = #(1)            // by (C8)
         Type mismatch         // type mismatch since 13 and 10 are
                               // built-in type numbers. See (IS4)

line 36: bv = av;
         #av = 10              // assigned previously in line 10
         #bv = 10              // assigned previously in line 11
         no mismatch           // rule (C7) applies

line 37: cv = 1.0;
         #cv = 14              // assigned previously in line 12
         #(1.0) = 11           // by (C11)
         #cv = #(1.0)          // by (C7)
         #cv <- 11             // See (IS5)
         #ct <- 11             // change all 14's to 11's
         #dt <- 11

line 38: cv = bv;
         #cv = 11              // assigned previously in line 37
         #bv = 10              // assigned previously in line 11
         #cv = #bv             // by (C7)
         Type mismatch         // type mismatch since 11 and 10 are
                               // built-in type numbers. See (IS4)

line 39: mv = nv;
         #mv <- 18             // mv is an implicit variable. See (IS3)
         #nv <- 19             // nv is an implicit variable. See (IS3)
         #mv = #nv             // by (C7)
         #nv <- 18             // See (IS6)
```

10 Grading

Your program will be executed on a number of test cases using the shell script provided for project 1. The grade is based on the test cases that your program correctly passes. To pass a test case, the output of the program should **match** the expected output. It is not enough that the right information be in the output that your program generates. The format and order of the output is essential.

11 Submission

1. Only C or C++ can be used for this project
2. You should submit all your code on the course website by 11:59:59 pm on due date.
3. Make sure your submission has no compile problems. If after submission you get a compiler error on the website, fix the problem and submit again. A submission that does not compile will not be given any credit
4. You can submit any number of times you need, but remember that we only grade your last submission
5. DO **NOT** include test scripts or test cases with your submission
6. DO **NOT** use any whitespace characters in your file names