**Project 4**

Due: **11/7/16** on or before 11:59:59pm MST

Your goal is to finish an incomplete parser and write a type checker for a given language. The input to your project will be a program and the output will be either error messages if there is a type mismatch or lists of equivalent types if there is no type mismatch. Your type checker will enforce semantic checks on the input program, and will be described in the following.

**Grammar Description**

program → decl body

decl → type\_decl\_section var\_decl\_section

type\_decl\_section → TYPE type\_decl\_list

type\_decl\_section → ε

type\_decl\_list → type\_decl type\_decl\_list

type\_decl\_list → type\_decl

type\_decl → id\_list COLON type\_name SEMICOLON

type\_name → REAL

type\_name → INT

type\_name → BOOLEAN

type\_name → STRING

type\_name → LONG

type\_name → ID

var\_decl\_section → VAR var\_decl\_list

var\_decl\_section → ε

var\_decl\_list → var\_decl var\_decl\_list

var\_decl\_list → var\_decl

var\_decl → id\_list COLON type\_name SEMICOLON

id\_list → ID COMMA id\_list

id\_list → ID

body → LBRACE stmt\_list RBRACE

stmt\_list → stmt stmt\_list

stmt\_list → stmt

stmt → while\_stmt

stmt → assign\_stmt

stmt → do\_stmt

stmt → switch\_stmt

while\_stmt → WHILE condition body

assign\_stmt → ID EQUAL expr SEMICOLON

do\_stmt → DO body WHILE condition SEMICOLON

switch\_stmt → SWITCH ID LBRACE case\_list RBRACE

case\_list → case case\_list

case\_list → case

case → CASE NUM COLON body

expr → term PLUS expr

expr → term MINUS expr

expr → term

term → factor MULT term

term → factor DIV term

term → factor

factor → LPAREN expr RPAREN

factor → NUM

factor → REALNUM

factor → ID

condition → ID

condition → primary relop primary

primary → ID

primary → NUM

primary → REALNUM

relop → GREATER

relop → GTEQ

relop → LESS

relop → NOTEQUAL

relop → LTEQ

The tokens used in the grammar description are:

TYPE = TYPE

COLON = :

SEMICOLON = ;

REAL = REAL

INT = INT

BOOLEAN = BOOLEAN

STRING = STRING

LONG = LONG

VAR = VAR

COMMA = ,

LBRACE = {

RBRACE = }

WHILE = WHILE

EQUAL = =

DO = DO

SWITCH = SWITCH

CASE = CASE

PLUS = +

MINUS = -

MULT = \*

DIV = /

LPAREN = (

RPAREN = )

GREATER = >

GTEQ = >=

LESS = <

LTEQ = <=

NOTEQUAL = <>

ID = letter(letter | digit)\*

NUM = 0 | (digit digit\*)

REALNUM = NUM \. digit\*

**Language Semantics**

As can be seen from the grammar, in this language types are first declared, then variables are declared, then the body of the program follows.

**Types**

The language has five built-in types: INT, REAL, BOOLEAN, STRING, and LONG.

Programmers can declare types either explicitly or implicitly.

* Explicit types are names that are not built-in types and that have their first appearance in the program as part of the id\_list of a type\_decl.
* Implicit types are not built-in types and not explicit programmer-declared types. Implicit types have their first appearance as a type\_name in a var\_decl or a type\_decl.

**Example**

Consider the following program written in our language:

TYPE

a : INT;

b : a;

VAR

x : b;

y : c;

{

y = x;

}

There are three types declared by the programmer in this example, a, b, and c, where a and b are explicit types and c is an implicit type.

**Variables**

Programmers can declare variables either explicitly or implicitly.

* Explicit variables are declared in an id\_list of a var\_decl.
* A variable is declared implicitly if it is not declared explicitly but it appears in the program body.

**Example**

Consider the following program written in our language:

TYPE

a : INT;

b : a;

VAR

x : b;

y : c;

{

y = x;

z = 10;

w = z \* 5;

}

This program has four variables declared: x, y, z, and w, with x and y explicitly declared and z and w implicitly declared. Note that the implicitly declared variables z and w also have an implicitly declared type.

**Declaration vs. Use**

Any appearance of a name (type or variable) in the program is either a **declaration** or a **use**.

The following lists all possible **declarations** of a name:

1. Any appearance of a name in the (left of COLON) id\_list part of a type\_decl is a type declaration
2. Any appearance of a name in the (left of COLON) id\_list part of a var\_decl is a variable declaration
3. The first appearance of a name in the entire program, if the name appears as type\_name in a type\_decl
4. The first appearance of a name in the entire program, if the name appears as type\_name in a var\_decl
5. The first appearance of a name inside the body of a program is a variable declaration

Any other appearance of a name is considered a **use** of that name.

Note that the above definitions exclude the built-in type names.

Given the following example (the line numbers are not part of the input):

1 TYPE

2 a : INT;

3 b : a;

4 VAR

5 x : b;

6 y : c;

7 {

8 y = x;

9 z = 10;

10 w = z \* 5;

11 }

We can categorize all appearances of names as declaration or use:

* Line 2, the appearance of name a is a declaration
* Line 3, the appearance of name b is a declaration
* Line 3, the appearance of name a is a use
* Line 5, the appearance of name x is a declaration
* Line 5, the appearance of name b is a use
* Line 6, the appearance of name y is a declaration
* Line 6, the appearance of name c is a declaration
* Line 8, the appearance of name y is a use
* Line 8, the appearance of name x is a use
* Line 9, the appearance of name z is a declaration
* Line 10, the appearance of name w is a declaration
* Line 10, the appearance of name z is a use

**Type System**

Our language uses structural equivalence for checking type equivalence.

Implicit types (in variable declarations or on implicitly declared variables) will be inferred from the usage (in a simplified form of Hindley-Milner type inference).

Here are all the type rules/constraints that your type checker will enforce (constraints are labeled from **C1** to **C5** for reference):

* **C1:** The left hand side of an assignment should have the same type as the right hand side of that assignment
* **C2:** The operands of an operation (PLUS, MINUS, MULT, and DIV) should have the same type (it can be any type, including STRING and BOOLEAN)
* **C3:** The operands of a relational operator (see relop in grammar) should have the same type (it can be any type, including STRINGand BOOLEAN)
* **C4:** condition should be of type BOOLEAN
* **C5:** The variable that follows the SWITCH keyword in switch\_stmt should be of type INT
* The type of an expr is the same as the type of its operands
* The result of p1 relop p2 is of type BOOLEAN (assuming that p1 and p2 have the same type)
* NUM constants are of type INT
* REALNUM constants are of type REAL
* If two types cannot be determined to be the same according to the above rules, the two types are different

**Incomplete Parser**

The parser given on the submission site is incomplete, as it is missing an implementation for while\_stmt, condition, do\_stmt, switch\_stmt, case\_list, and case.

As described in the evaluation section, you must implement parsing for while\_stmt, condition, and do\_stmt, while switch\_stmt, case\_list, and case are extra credit (though note that to receive full extra credit, you must implement all of the type checks in addition to the parsing cases.

It is strongly recommended that you finish the incomplete parser *before* implementing the type checking part. You should make sure that your parser generates a syntax error message if the input program does not follow the proper syntax. We recommend that you check your code on the submission website to make sure it passes all the syntax error test cases before moving on to implementing the type checking part.

**Output**

Your program will check for the following semantic errors and output the correct message when it encounters that error. Note that there will only be at most one error per test case.

**Duplication Errors**

* Errors involving programmer-defined types:
  + Programmer-defined type declared more than once:
    - **Explicit type redeclared explicitly** (error code **1.1**)  
      An explicitly declared type can be declared again explicitly by appearing as part of an id\_list in a type declaration.
    - **Implicit type redeclared explicitly** (error code **1.2**)  
      An implicitly declared type can be declared again explicitly by appearing as part of an id\_list in a type declaration.

Note that a previously declared type name (either implicit or explicit) cannot be declared again *implicitly*. Since it has already been introduced, the new reference to the name (as type\_name in a type\_decl or var\_decl) would be a *use* and not a *declaration*.

* + **Programmer-defined type redeclared as variable** (error code **1.3**)  
    If a previously declared type appears again in an id\_list of a variable declaration, the type is redeclared as a variable.
  + **Programmer-defined type used as variable** (error code **1.4**)  
    If a previously declared type appears in the body of the program, the type is used as a variable.
* Errors involving variable declarations:
  + **Variable declared more than once** (error code **2.1**)  
    An explicitly declared variable can be declared again explicitly by appearing as part of an id\_list in a variable declaration.
  + **Variable used as a type** (error code **2.2**)  
    If an explicitly declared variable is used as type\_name in a variable declaration, the variable is used as a type.

Note that an explicitly declared variable cannot be declared again *implicitly*, appearances of the name in the program body are uses. In the same way, an implicitly declared variable cannot be declared again, because all later appearances are uses.

Also note that if a built-in type is redeclared or used in the body of the program, it should result in a syntax error.

**Duplication Error Output Format**

For these errors, you should output one line in the following format:

ERROR CODE <code> <symbol\_name>

in which <code> should be replaced with the proper code (see the error codes listed above) and <symbol\_name> should be replaced with the name of the type or variable related to the error.

**Type Mismatch**

If any of the type constraints (listed in the Type System section above) is violated in the input program, then the output of your program should be:

TYPE MISMATCH <line\_number> <constraint>

Where <line\_number> is replaced with the line number that the violation occurs and <constraint> should be replaced with the label of the violated type constraint (possible values are **C1** through **C5**, see section on Type System for details of each constraint). Note that you can assume that anywhere a violation can occur it will be on a single line.

**No Semantic Errors**

If there are no semantic errors in the program, then your program should output lists of types and variables that are type-equivalent. The symbols should be listed in the order they appear in the program and built-in types should be listed first in the following order: BOOLEAN, INT, LONG, REAL, STRING. Each list must be on a single line of the output and each symbol in the list should be separated by a single space character. Each list must be terminated by a # character.

The following pseudo-code should explain the output format more precisely:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | for each built-in type T:  {  output T  output all names that are type-equivalent with T in order of their appearance  mark outputted names to avoid re-printing them later  output "#\n"  }  if there are unprinted names left:  {  for each unprinted name N in order of appearance:  {  output N  output all other names that are type-equivalent with N in order of their appearance  output "#\n"  }  } |

The phrase in order of appearance in the above pseudo-code means that names that appear before others in the program should be processed first. This order should be easy to maintain since it is the natural order of storing names in your symbol table.

**Examples**

Given the following:

TYPE

a,b,c,b : INT;

VAR

x : a;

{

x = 10;

}

The output will be the following:

ERROR CODE 1.1 b

Given the following:

TYPE

a : INT;

VAR

x : INT;

b, a : STRING;

{

x = 10;

}

The output should be the following:

ERROR CODE 1.3 a

Given the following:

VAR

x1 : INT;

x2, x3, x1 : a;

{

x1 = 0;

}

The output should be the following:

ERROR CODE 2.1 x1

Given the following:

VAR

x, y : STRING;

z : x;

{

y = x;

}

The output should be the following:

ERROR CODE 2.2 x

Given the following:

VAR

x100 : INT;

y : STRING;

{

x100 = y;

}

The output should be the following:

TYPE MISMATCH 5 C1

Given the following:

VAR

x : INT;

{

x = 100;

y = 20.10;

y = x;

}

The output should be the following:

TYPE MISMATCH 6 C1

Given the following:

VAR

x, y : a1;

{

WHILE x <> 10

{

x = x + y;

y = y \* 1.0;

}

}

The output should be the following:

TYPE MISMATCH 7 C2

Given the following:

TYPE

a, b : INT;

c : a;

d : STRING;

VAR

x : e;

y : c;

test : d;

{

a1 = 100;

b1 = a1 + (10 - 50);

foo = b1 / 50;

SWITCH foo

{

CASE 1:

{

foo = 0;

}

CASE 2:

{

test = test \* test;

}

}

h = x;

}

The output should be the following:

BOOLEAN #

INT a b c y a1 b1 foo #

LONG #

REAL #

STRING d test #

x e h #