CSE340 Spring 2018 Project 3: Parsing and Type Checking

Due: **Friday March 30 , 2018** on or before 11:59 pm MST

In this project, you are asked to write a predictive parser and a type checker for a small language. The parser checks that the input is syntactically correct and the type checker enforces the semantic rules of the language. The semantic rules that your program will enforce relate to declarations and types.

The input to your code will be a program and the output will be:

- syntax error if the input program is not syntactically correct
- error messages if there is a declaration error or a type mismatch in the input program
- information about the types of the symbols declared in the input program if there is no error.

In what follows, I describe the language syntax and semantic rules.

1. Language Syntax

1.1. Grammar Description

program	→ scope
scope	→ LBRACE scope_list RBRACE
scope_list	→ scope
scope_list	→ var_decl
scope_list	→ stmt
scope_list	→ scope scope_list
scope_list	→ var_decl scope_list
scope_list	→ stmt scope_list
var_decl	→ id_list COLON type_name SEMICOLON
id_list	→ ID
id_list	→ ID COMMA id_list
type_name	→ REAL | INT | BOOLEAN | STRING
stmt_list	→ stmt
stmt_list	→ stmt stmt_list
stmt	→ assign_stmt
stmt	→ while_stmt
assign_stmt	→ ID EQUAL expr SEMICOLON
while_stmt	→ WHILE condition LBRACE stmt_list RBRACE
while_stmt	→ WHILE condition stmt
expr	→ arithmetic_operator expr expr
expr	→ boolean_operator expr expr
expr	→ relational_operator expr expr
expr	→ NOT expr
expr	→ primary
arithmetic_operator	→ PLUS | MINUS | MULT | DIV
boolean_operator	→ AND | OR | XOR
relational_operator	→ GREATER | GTEQ | LESS | NOTEQUAL | LTEQ
primary	→ ID | NUM | REALNUM | STRING_CONSTANT | bool_const
bool_const	→ TRUE | FALSE
condition	→ LPAREN expr RPAREN

Notice that expressions are in prefix notation in this language.

1.2. Tokens

The tokens used in the grammar description are:

```
LBRACE              =   {
RBRACE              =   }
COLON               =   :
SEMICOLON           =   ;
REAL                =   REAL
INT                 =   INT
BOOLEAN             =   BOOLEAN
STRING              =   STRING
WHILE               =   WHILE
COMMA               =   ,
LPAREN              =   '('
RPAREN              =   ')'
EQUAL               =   =
PLUS                =   '+'
MULT                =   *
DIV                 =   /
AND                 =   ^
OR                  =   '|'
XOR                 =   &
NOT                 =   ~
GREATER             =   >
GTEQ                =   >=
LESS                =   <
LTEQ                =   <=
NOTEQUAL            =   <>
TRUE                =   TRUE
FALSE               =   FALSE
STRING\_CONSTANT    =   " (letter | digit)* "
ID                  =   letter (letter | digit)*
NUM                 =   0 | (pdigit digit*)
REALNUM             =   NUM . digit digit*
```

where

letter = a | b | c | ... | z | A | B | C | ... | Z
digit = 0 | 1 | 2 | ... | 9
pdigit = 1 | 2 | 3 | ... | 9

2. Language Semantics

2.1. Scoping Rules

Static scoping is used. I have already covered static scoping in class, so you should know the semantics and how references should be resolved. Every `scope` defines a scope.

2.2. Types

The language has four basic types: `INT` , `REAL` , `BOOLEAN` , and `STRING` .

2.3. Variables

Programmers declare variables explicitly in `var_decl`. The names of the declared variables appear in the `id_list` and their type is the `type_name`.
Example Consider the following program written in our language:

```
{
    x : INT;
    y : INT;
    y = x;
}
```

This program has two declared variables: `x` and `y`.

2.4. Declaration and Reference

Any appearance of a name in the program is either a **declaration** or a **reference**. Any appearance of a name in the `id_list` part of a `var_decl` is a declaration of the name. Any other appearance of a name is considered a reference to that name. Note that the above definitions exclude the built-in type names, which are predeclared as part of the language definition.

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

```
01 {
02      a : INT;
03      b : REAL;
04      b = x;
05 }
```

We can categorize all appearances of names as **declaration** or **reference**:

- Line 2, the appearance of name `a` is a declaration
- Line 3, the appearance of name `b` is a declaration
- Line 4, the appearance of name `b` is a reference
- Line 4, the appearance of name `x` is a reference

2.5. Type System

We describe which assignments are valid (type compatibility) and how to determine the types of expressions (type inference).

2.5.1. Type Compatibility Rules

Type compatibility rules specify which assignments are valid. The Rules are the following.

- **C1:** If the types of the lefthand side of an assignment is `INT`, `BOOLEAN`, or `STRING`, the righthand side of the assignment should have the same type.
- **C2:** If the type of the lefthand side of an assignment is `REAL`, the type of the righthand side of the assignment should be `INT` or `REAL`.
- **M1:** Assignments that do not satisfy **C1** or **C2** are not valid. In this case, we say that there is a type mismatch.

2.5.2. Type Inference Rules

- **C3:** The types of the operands of an arithmetic operator should be `REAL` or `INT`
- **C4:** The type of the operands of a `boolean_operator` should be `BOOLEAN`.
- **C5:** If the type of one the operands of a `relational_operator` are not `INT` or `REAL`, the types of the two operands of the `relational_operator` should be the same. In this case, both types can be STRING or both types can be BOOLEAN.
- **C6:** If the type of on operand of a `relational_operator` is `INT` or `REAL`, the type of the other operand should be `INT` or `REAL`. In this case, the two operands can have different types.
- **C7:** The type of a `condition` should be `BOOLEAN`.
- **I1:** If **C1** is satisfied, and the type of one of the operands of an arithmetic operator is `REAL`, the type of the resulting expression is `REAL`.
- **I2:** For the `PLUS`, `MINUS`, and `MULT` operators, if the types of the two operands are `INT`, the type of the resulting expression is `INT`.
- **I3:** For the `DIV` operator, if the types of the two operands are `INT`, the type of the resulting expression is `REAL`.
- **I4:** If the operands of a `boolean_operator` are `BOOLEAN`, the type of the resulting expression is `BOOLEAN`.
- **I5:** If **C5** and **C6** are satisfied, the type of a `relational_operator expr expr` is `BOOLEAN`.
- **I6:** The type of `NUM` constants is `INT`
- **I7:** The type of `REALNUM` constants is `REAL`
- **I8:** The type of `bool_const` constants is `BOOLEAN`
- **I9:** The type of `STRING_CONSTANT` constants is `STRING`
- **M2:** If **C3**, **C4**, **C5**, **C6**, or **C7** are not satisfied, the type of the expressions is ERROR. In this case, we say that there is a type mismatch.

3. Parser

You must write a parser for the grammar, If your parser detects a syntax error in the input, it should output the following message and exit:

```
Syntax Error
```

You should start coding by writing the parser first and make sure that your parser generates a syntax error message if the input program is syntactically incorrect. **There will be no partial credit given for Task 1**

The grammar of the language is not LL(1) i.e. it does not satisfy the conditions for predictive parser with one token lookahead, however, it is still possible to write a recursive descent parser with no backtracking. We can do this by looking ahead at more than one token in some cases and left-factoring some rules. Two rules like

stmt_list $\rightarrow$ stmt
stmt_list $\rightarrow$ stmt stmt_list

can be parsed by first parsing `stmt` and then checking if the next token is the beginning of `stmt_list` or in the FOLLOW of `stmt_list`. In fact the two rules are equivalent to

stmt_list $\rightarrow$ stmt stmt_list1
stmt_list1 $\rightarrow$ stmt_list $\mid \epsilon$

but you do not need to explicitly left-factor the rules by introducing `stmt_list1` to parse `stmt_list`.

4. Semantic Checking

Your program will check for the following semantic errors and output the correct message when it encounters that error.

4.1. Declaration Errors

- **Variable declared more than once** (error code **1.1**)
  A variable is declared more than once if appears more than once in the same `id_list` of a `var_decl` or if it appears in two `id_list` of two different `var_decl`.
- **Undeclared variable** (error code **1.2**)
  If resolving a variable name that appears in a statement other than a declaration returns no declaration, the variable is undeclared.
- **Variable Declared but not used** (error code **1.3**)
  If a variable is declared but is not referenced, we say that the variable is declared but not used. We say that a declaration referenced if some reference to the variable resolves to the declaration. If a declaration is not referenced, we have error code 1.3.
- **Redeclaration or reference as a variable for of a built-in type name** If a built-in type name appears in `id_list` of a variable declaration or appears in a statement other than a declaration statement, your parser should output syntax error.

For each error involving declarations, your type checker 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 variable that caused the error. Since the test cases will have at most one error each, the order in which these error messages are printed does not matter.

Note that there will only be at most one declaration error per test case.

4.2. Type Mismatch

If there are no declaration errors and 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**, **C2**, **C3**, **C4**, **C5**, **C6**, or **C7**. (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.

If there are declaration errors and type mismatch errors, only the output for the declaration errors should be produced.

Note that there will only be at most one type mismatch error per test case.

4.3. Use of Uninitialized Variables

For this part, your program should identify variables that are used before they are defined. We say that a *variable is used if it appears in an expression*. We say that a *variable is defined if it appears on the lefthand side of an assignment*. A variable is used before it is defined if there is an execution path in which there is a use of the variable that is not preceded by a definition of the variable. Compilers typically call this "use of an uninitialized variable". In general determining use of uninitialized variables is involved and requires building a control flow graph of the program. Fortunately, for the language of this project, determining uninitialized uses is relatively easy. First, I will describe the output format, then I will give a more detailed description of how determine uninitialized uses.

The output format for this task is the following

```
UNINITIALIZED <name_reference_1> <line_no_reference_1>
UNINITIALIZED <name_reference_2> <line_no_reference_2>
...
```

Where `<name_reference_i>` is the name of `i`'th uninitialized variable and `<line_no_reference_i>` is the line number in which the `i`'th uninitialized reference appears.

In what follows, I will explain how to determine uses of uninitialized variables. Consider the program below (the line numbers are not part of the input but are added for ease of reference):

```
01 {
02     x1, x2, x3, x4 , x5 : INT;
03
```

```
04      x3 = 2;
05
06      WHILE ( < x1 10 ) {
07           x1 = + x1 1;
08           x2 = 1;
09           x4 = + x2 1;
10           x1 = + x3 1;
11           WHILE ( < x1 10 ) {
12               x5 = + x1 + x3 x5 ;
13           }
14           x1 = x5;
15           x5 = 1;
16      }
17
18      x2 = + x2 1 ;
19 }
```

- The use of `x1` in the expression `< x1 10` is a use of an uninitialized variable. The first time the expression is evaluated, there is no previous definition (assignment) to `x1`. Note that later evaluations of `< x1 10` happen after the assignment `x1 = + x1 1;` in the body of the loop, but use still counts as uninitialized use because the first use is uninitialized.
- The use of `x2` on line `09` is initialized because it is always preceded by the definition (assignment to) of `x2` on line `08`.
- The use of `x3` on line `10` is initialized because it is always preceded by the definition (assignment to) `x3` on line `04`.
- The use of `x1` on line `11` is initialized because it is always preceded by the definition (assignment to) `x1` on line `10` (also on line `07`).
- The use of `x1` on line `12` is initialized because it is always preceded by the definition (assignment to) `x1` on line `10` (also on line `07`).
- The use of `x3` on line `12` is initialized because it is always preceded by the definition (assignment to) `x3` on line `04`.
- The use of `x5` on line `12` is not initialized because it is not preceded by a definition (assignment to) `x5`.
- The use of `x5` on line `14` is not initialized because it is not preceded by a definition (assignment to) `x5`. Even though there is a definition (assignment to) of `x5` on line `12`, we cannot determine (in general) if the body of the loop will be executed, so we assume that the preceding loop did not execute. Note that this situation is different from the use of `x1` on line `12` which is guaranteed to be preceded by the definition (assignment to) of `x1` on line `10`.
- The use of `x2` on line `18` is not initialized because it is not preceded by a definition (assignment to) `x2`. Even though `x2` is defined (assigned a value) on line `08`, we cannot determine (in general) if the body of the loop is executed, so we have to assume that it is not executed.

It should be clear from the example, that to determine uninitialized variables, you can treat `while_stmt` as some kind of a scope. If a use is inside a `while_stmt`, then all definitions preceding it in the `while_stmt` together with all definitions preceding it in enclosing `while_stmt` should be taken into consideration.

4.4. No Semantic Errors

If there are no declaration errors, no type mismatch errors and no use of uninitialized variables, then your program should output for each reference of a programmer declared variable name, in the order

in which the reference appears in the program, the line number in which the reference appears and the line number of the declaration to which the reference of the name resolves. For this part, the format should be the following

```
<name_reference_1> <line_no_reference_1> <line_no_declared_1>
<name_reference_2> <line_no_reference_2> <line_no_declared_2>
...
```

Where `<name_reference_i>` is the name of a variable and corresponds to i'th name reference in the program. `<line_no_reference_i>` is the line number in which the i'th reference appears and `<line_no_declared_i>` is the line number of the declaration corresponding to the i'th reference.

5. More Examples

Example 1. Given the following example (the line numbers are not part of the input):

```
01 {
02     x : INT;
03     y : BOOLEAN;
04     y = x;
05 }
```

The output will be the following:

```
TYPE MISMATCH  4 C1
```

Example 2. Given the following example (the line numbers are not part of the input):

```
01 {
02     x : BOOLEAN;
03     y : REAL;
04     y = x;
05 }
```

The output will be the following:

```
TYPE MISMATCH  4 C2
```

Example 3. Given the following example (the line numbers are not part of the input):

```
01 {
02     x : BOOLEAN;
03     x : BOOLEAN;
04     x = x;
05 }
```

The output will be the following:

```
ERROR CODE 1.1 x
```

Example 4. Given the following example (the line numbers are not part of the input):

```
01 {
02     x : INT;
03     y, x : STRING;
04     y = 10;
05     x = 10;
05 }
```

The output will be the following:

```
ERROR CODE 1.1 x
```

Example 5. Given the following example (the line numbers are not part of the input):

```
01 {
02     x : INT;
03     y : STRING;
04     y = "abc";
05 }
```

The output will be the following:

```
ERROR CODE 1.3 x
```

Example 6. Given the following example (the line numbers are not part of the input):

```
01 {
02     x1 , x2 : INT;
03     y : INT;
04     x1 = y;
05     y = + x1 x2;
06 }
```

The output will be the following:

```
UNINITIALIZED y 4
UNINITIALIZED x2 5
```

Example 7. Given the following example (the line numbers are not part of the input):

```
01  {
02        a : INT;
03        b : INT;
04        a = 1;
05        b = 1;
06        {    a : INT;
07             a = 1;
08             WHILE ( > a b )
09             {
10                   a = - a b;
11             }
12        }
13  }
```

The output will be the following:

```
a 4 2
b 5 3
a 7 6
a 8 6
b 8 3
a 10 6
a 10 6
b 10 3
```

6. Summary of Requirements

1. If there is a syntax error, the program should output syntax error and exit.
2. If there is no syntax error and there is a declaration error, the program should output the error message for the declaration error. You can assume that there can be no more than one declaration error in the test case. If the program outputs a declaration error, the program should output no other error message. Specifically, it should not output any type mismatch or uninitialized variable error message.
3. If there is no syntax error and no declaration error and there is a type mismatch, the program should output the error message for the type mismatch. You can assume that there can be no more than one type mismatch per test case. If the program outputs a type mismatch error message, the program should output no other error message. Specifically, it should not output any uninitialized variable error message.
4. If there is no syntax error, no declaration error and no type mismatch error, the program should output the list of variables used but not initialized in the format specified in Section 4.3.
5. If there is no error of any kind, the program should output for each reference, the line number of the reference and the line number of the declaration to which the reference resolves as explained in Section 4.4 above.

7. Evaluation

Your submission will be graded on passing the automated test cases.

The test cases (there will be multiple test cases in each category, each with equal weight) will be broken down in the following way (out of 100 points):

- Parsing: 35 points
- Errors involving declarations: 20 points
- Type mismatch errors and no semantic error cases: 35 points
- Use of uninitialized variables: 10 points

There is no partial credit for the parsing category, you need to pass all test cases in that category to get the 35 points. All other categories have partial credit.