CSE340 Fall 2017 Project 3: Parsing and Type Checking

Due: **Monday, November 6, 2017** on or before 11:59 pm MST

Your goal is to write a predictive parser and write a type checker for a given language. The parser checks the syntax of the input and the type checker enforces the semantic rules of the language. The semantic rules that we are interested in specify which assignments are valid and which expressions are valid.

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

- an error message if there is a type mismatch or syntax error or
- lists of symbols of different types if there is no error.

1. Grammar Description

program	→ scope
scope	→ LBRACE scope_list RBRACE
scope_list	→ stmt
scope_list	→ declaration
scope_list	→ stmt scope_list
scope_list	→ declaration scope_list
declaration	→ type_decl | var_decl
type_decl	→ TYPE id_list COLON type_name SEMICOLON
type_name	→ REAL | INT | BOOLEAN | STRING | LONG | ID
var_decl	→ VAR id_list COLON type_name SEMICOLON
id_list	→ ID
id_list	→ ID COMMA id_list
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
expr	→ term
expr	→ term PLUS expr
term	→ factor
term	→ factor MULT term
~~term~~	→ ~~factor DIV term~~
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

To be added

scope_list → scope
scope_list → scope scope_list

To be deleted

The tokens used in the grammar description are:

```
TYPE        = TYPE
VAR         = VAR
REAL        = REAL
INT         = INT
BOOLEAN     = BOOLEAN
STRING      = STRING
LONG        = LONG
WHILE       = WHILE
COMMA       = ,
COLON       = :
SEMICOLON   = ;
LBRACE      = {
RBRACE      = }
LPAREN      = (
RPAREN      = )
EQUAL       = =
PLUS        = +
MULT        = *
DIV         = /
GREATER     = >
GTEQ        = >=
LESS        = <
LTEQ        = <=
NOTEQUAL    = <>
ID          = letter (letter + digit)*
NUM         = 0 + (pdigit digit*)
REALNUM     = NUM \. digit digit*
```

2. Language Semantics

2.1. Scoping Rules

Lexical scoping is used. Every `scope` defines a scope.

2.2. Types

The language has five built-in types: `INT` , `REAL` , `BOOLEAN` , `STRING` , and `LONG` . Programmers can also declare new types with a `type_decl` which can appear anywhere in the program, except in the statement list of a `while_stmt` .

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:

```
{
    TYPE  a : INT;
    TYPE b : a;
```

```
    VAR x : b;
    VAR y : c;
    y = x;
}
```

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

2.4. 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 `id_list` part of a `type_decl`
2. Any appearance of a name in the `id_list` part of a `var_decl`

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):

```
01 {
02     TYPE  a : INT;
03     TYPE b : a;
04     VAR x : b;
05     VAR y : c;
06     y = x;
07 }
```

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 4, the appearance of name `x` is a declaration
- Line 4, the appearance of name `b` is a use
- Line 5, the appearance of name `y` is a declaration
- Line 5, the appearance of name `c` is a declaration
- Line 6, the appearance of name `y` is a use
- Line 6, the appearance of name `x` is a use

2.5. Type System

Our language uses structural equivalence for checking type equivalence.

Here are all the type rules/constraints that your type checker will enforce (constraints are labeled from **C1** to **C4** 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 `STRING` and `BOOLEAN`)
- **C4:** `condition` should be of type `BOOLEAN`
- 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

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 can start coding by writing the parser first and then move on to 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 test cases in the parsing category before moving on to implementing the type checking part.

Our grammar 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. In particular, parsing `condition` requires more than one token lookahead. Also, two rules like

```
stmt_list  → stmt
stmt_list  → 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   → stmt stmt_list1
stmt_list1  → stmt_list | ϵ
```

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

4. 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.

4.1. Redeclaration Errors

1. Errors involving programmer-defined types:
   - **Programmer-defined type declared more than once** (error code **1.1**)
     A type is declared more than once if it appears in more than one `id_list` of `type_decl` in the same scope. Declaring the same type name is allowed in non-overlapping scopes and in nested scopes (lexical scoping).
   - **Programmer-defined type redeclared as a variable** (error code **1.2**)
     A type name is redeclared as a variable if the name appears first in an `id_list` of a `type_decl` and appears again in `id_list` of `var_decl` in the same scope. Using the same programmer defined name for a variable and a type is allowed in non-overlapping scopes and in nested scopes (lexical scoping)
   - **Programmer-defined type used as variable** (error code **1.3**)
     If resolving a variable name returns a type declaration, the type is used as a variable.
   - **Undeclared type** (error code **1.4**)
     If resolving a `type_name` returns no declaration, the type is undeclared.
2. 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 and resolving the name at the site of the new declaration returns the first declaration.
   - **Programmer-defined variable redeclared as a type** (error code **2.2**)
     A variable name is redeclared as a type if the name appears first in an `id_list` of a `var_decl` and appears again in `id_list` of `type_decl` in the same scope. Using the same programmer defined name for a variable and a type is allowed in non-overlapping scopes and in nested scopes (lexical scoping)
   - **Variable used as a type** (error code **2.3**)
     If the `type_name` in a variable declaration resolves to a variable declaration, the variable is used as a type.
   - **Undeclared variable** (error code **2.4**)
     If resolving a variable name that appears in a statement other than a declaration returns no declaration, the variable is undeclared.
3. Errors involving built-in type:
   - **Redeclaration or use of a built-in type name** If a built-in type name appears in `id_list` of a variable declaration or a type declaration or appears in a statement other than a declaration statement, your parser should output syntax error.

For each error involving variable declarations and errors involving type 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 type or 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.

4.2. 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 **C4**, 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.

4.3. No Semantic Errors

If there are no semantic errors in the program, then your program should output for each use of a programmer-defined type name and variable name, in the order in which the use appears in the program, the line number in which the use appears and the line number of the declaration to which the use of the name resolves. For this part, the format should be the following

```
<name_used_1> <line_no_use_1> <line_no_declared_1>
<name_used_2> <line_no_use_2> <line_no_declared_2>
...
```

Where `<name_use_i>` is the name of a variable or a type and corresponds to i'th name use in the program. `<line_no_use_i>` is the line number in which the i'th use appears and `<line_no_declared_i>` is the line number of the declaration corresponding to the i'th use.

5. Examples

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

```
01 {
02     TYPE  a, b, c, b : INT;
04     VAR x : b;
05     VAR y : c;
06     y = x;
07 }
```

The output will be the following:

```
ERROR CODE 1.1 b
```

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

```
01 {
02     TYPE  a : INT;
03     VAR x : INT;
04     VAR b, a : STRING;
05     x = 10;
06 }
```

The output will be the following:

```
ERROR CODE 1.2 a
```

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

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

The output will be the following:

```
ERROR CODE 2.1 x
```

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

```
01 {
04     VAR x : INT;
05     TYPE y, x : STRING;
06     x = 10;
07 }
```

The output will be the following:

```
ERROR CODE 2.2 x
```

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

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

The output will be the following:

```
TYPE MISMATCH 4 C1
```

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

```
01 {
02    VAR
03         x : INT;
04    VAR  y : REAL;
05    WHILE x <> 10
06    {
07          x = x + y;
08    }
09 }
```

The output will be the following:

```
TYPE MISMATCH 7 C2
```

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

```
01  {
02        TYPE t : INT;
03        VAR a, b : t;
04        {    VAR a : INT;
05             WHILE  a > b
06              {
07                   a = a + b;
08              }
09        }
10  }
```

The output will be the following:

```
t 3 2
a 5 4
b 5 3
a 7 4
a 7 4
b 7 3
```

6. 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 programmer-defined types (error codes 1.x): 15 points
- Errors involving variable declarations (error codes 2.x): 20 points
- Type mismatch errors and no semantic error cases: 30 points

The parsing category is not partially graded, you need to pass all test cases in that category to get the 35 points. All other categories are partially graded.