CSE340 Summer 2020 Project 2: Parsing

Due: **Thursday, July 30, 2020 by midnight MST**

The goal of this project is to give you experience in writing a top-down recursive descent parser and to get introduced to the basics of symbol tables for nested scopes.

We begin by introducing the grammar of our language. Then we will discuss the semantics of our language that involves lexical scoping rules and name resolution. Finally we will go over a few examples and formalize the expected output.

NOTE: This project is significantly more involved than the first project. You should start on it immediately. For Monday, I expect that you have read the project description completely at least a couple times and maybe started on your parser.

1. Lexical Specification

And here is the list of tokens that your lexical analyzer needs to support:

```
PUBLIC = "public"
PRIVATE = "private"
EQUAL = "="
COLON = ":"
COMMA = ","
SEMICOLON = ";"
LBRACE = "{"
RBRACE = "}"
ID = letter (letter + digit)*
```

Comments and Space

In addition to these tokens, our input programs might have **comments** that should be ignored by the lexical analyzer. A comment starts with `//` and continues until a newline character is encountered. The regular expressions for comments is: `// (any)* \n` in which `any` is defined to be any character except `\n` . Also, like in the first project, your lexical analyzer should skip space between tokens.

2. Grammar

Here is the grammar for our input language:

```
program       -> global_vars scope
global_vars   -> ε
global_vars   -> var_list SEMICOLON
var_list      -> ID
var_list      -> ID COMMA var_list
scope         -> ID LBRACE public_vars private_vars stmt_list RBRACE
public_vars   -> ε
public_vars   -> PUBLIC COLON var_list SEMICOLON
private_vars  -> ε
private_vars  -> PRIVATE COLON var_list SEMICOLON
stmt_list     -> stmt
stmt_list     -> stmt stmt_list
stmt          -> ID EQUAL ID SEMICOLON
stmt          -> scope
```

Here is an example input program with comments:

```
a, b, c;                // These are global variables

test {
    public:
        a, b, hello;    // These are public variables of scope test
    private:
        x, y;           // and these are private variables of scope test

    a = b;              // the body of test starts with this line
    hello = c;
    y = r;
    nested {            // and this is a nested scope
        public:
            b;          // which does not have private variables
        a = b;
        x = hello;
        c = y;
        // We can also have lines that only contain comments like this
    }
}
```

Note that our grammar does not recognize comments, so our parser would not know anything about comments, but our lexical analyzer would deal with comments. This is similar to handling of spaces by the lexer, the lexer skips the spaces. In a similar fashion, your lexer should skip

comments.

We highlight some of the syntactical elements of the language:

- Global variables are optional
- The scopes have optional public and private variables
- Every scope has a body which is a list of statements
- A statement can be either a simple assignment or another scope (a nested scope)

3. Scoping and Resolving References

Here are the scoping rules for our language:

- The public variables of a scope are accessible to its nested scopes
- The private variables of a scope are not accessible to its nested scopes
- Lexical scoping rules are used to resolve name references
- Global variables are accessible to all scopes

Every reference to a variable is resolved to a specific declaration by specifying the variable's defining scope. We will use the following notation to specify declarations:

> If variable `a` is declared in the global variables list, we use `::a` to refer to it
>
> If variable `a` is declared in scope `b`, we use `b.a` to refer to it

And if reference to name `a` cannot be resolved, we denote that by `?.a`

Here is the example program from the previous section, with all name references resolved (look at the comments):

```
a, b, c;

test {
    public:
        a, b, hello;
    private:
        x, y;

    a = b;              // test.a = test.b
    hello = c;          // test.hello = ::c
    y = r;              // test.y = ?.r
    nested {
        public:
            b;
        a = b;          // test.a = nested.b
        x = hello;      // ?.x = test.hello
        c = y;          // ::c = ?.y
    }
}
```

4. Examples

The simplest possible program would be:

```
main {
    a = a;       // ?.a = ?.a
}
```

Let's add a global variable:

```
a;

main {
    a = a;       // ::a = ::a
}
```

Now, let's add a public variable `a`:

```
a;

main {
    public:
        a;
    a = a;              // main.a = main.a
}
```

Or a private `a`:

```
a;

main {
    private:
        a;
    a = a;              // main.a = main.a
}
```

Now, let's see a simple example with nested scopes:

```
a, b;

main {
    nested {
        a = b;    // ::a = ::b
    }
}
```

If we add a private variable in `main`:

```
a, b;

main {
    private: a;
    nested {
        a = b;    // ::a = ::b
    }
}
```

And a public `b`:

```
a, b;

main {
    public: b;
    private: a;
    nested {
        a = b;   // ::a = main.b
    }
}
```

You can find more examples by looking at the test cases and their expected outputs.

5. Expected Output

There are two cases:

- In case the input does not follow the grammar, the expected output is:

  ```
  Syntax Error
  ```

  NOTE: no extra information is needed here! Also, notice that we need the exact message and it's case-sensitive.

- In case the input follows the grammar:

  For every assignment statement in the input program in order of their appearance in the program, output the following information:

  - The resolved left-hand-side of the assignment
  - The resolved right-hand-side of the assignment

  in the following format:

  ```
  resolved_lhs = resolved_rhs
  ```

  NOTE: You can assume that scopes have unique names and variable names in a single scope (public and private) are not repeated.

For example, given the following input program:

```
a, b, c;

test {
    public:
        a, b, hello;
    private:
        x, y;

    a = b;
    hello = c;
    y = r;
    nested {
        public:
            b;
        a = b;
        x = hello;
        c = y;
    }
}
```

The expected output is:

```
test.a = test.b
test.hello = ::c
test.y = ?.r
test.a = nested.b
?.x = test.hello
::c = ?.y
```

6. Implementation

- Start by modifying the lexical analyzer from previous project to make it recognize the tokens required for parsing this grammar. It should also be able to handle comments (skip them like spaces) . **NOTE:** make sure you **remove the tokens that are not used in this grammar from your lexer**, otherwise you might not be able to pass all test cases. Your `TokenType` type declaration should look like this:

  ```
  typedef enum { END_OF_FILE = 0,
          PUBLIC, PRIVATE,
          EQUAL, COLON, COMMA, SEMICOLON,
          LBRACE, RBRACE, ID, ERROR
  } TokenType;
  ```

- Next, write a parser for the given grammar. You would need one function per each non-terminal of the grammar to handle parsing of that non-terminal. I suggest you use the following signature for these functions:

  ```
  void parse_X()
  ```

  Where `X` would be replaced by the target non-terminal. The lexical analyzer object needs to be accessible to these functions so that they can use the lexer to get and unget tokens. These functions can be member functions of a class, and the lexer object can be a member variable of that class.

  You also need a `syntax_error` function that prints the proper message and terminates the program:

  ```
  void syntax_error()
  {
      cout << "Syntax Error\n";
      exit(1);
  }
  ```

- Test your parser thoroughly. Make sure it can detect any syntactical errors.
- Next, write a symbol table that stores information about scopes and variables. You would also need to store assignments in a list to be accessed after parsing is finished. You need to think about how to organize all this information in a way that is useful for producing the required output.

  Write a function that resolves the left-hand- side and right-hand-side of all assignments and produces the required output. Call this function in your `main()` function after successfully parsing the input.

  NOTE: you might need more time to finish the last step compared to previous steps.

7. Requirements

- You should use C/C++, no other programming languages are allowed.
- You should test your code on Ubuntu linux 19.04 or greater with gcc 4.9 or higher
- You should submit your code on the course submission website, no other submission forms will be accepted.

8. Evaluation

The submissions are evaluated based on the automated test cases on the submission website. Your grade will be proportional to the number of test cases passing. If your code does not compile on the submission website, you will not receive any points.

Here is the breakdown of points for tests in different categories:

- Parsing (including inputs with syntax errors and error-free inputs): **50** points
- Name resolution: **50** points
- Test cases containing comments: **5** points extra credit

The parsing test cases contain cases that are syntactically correct and cases that have syntax errors. If a syntax test case has no syntax error, your program passes the test case if the output is not `Syntax Error`. If a syntax test case has syntax error, your program passes the test case if the output is `Syntax Error`.

Note that if your program prints the syntax error message independently of the input, for example:

```
int main()
{
    cout << "Syntax Error\n";
    return 0;
}
```

It will pass some of the test cases, but you will not receive any points.