Compiler Project 1

11611006 邓韵杰

This project is divided into two part. The first part is to implement a lexical analyzer, and the second part is to do syntax analysis.

1. Lexical Analysis

This part requires us design a lexical analyzer to obtain the token in code. By using Flex, we can define several regular expression in **.l** file to recognize the token that we defined in **token.txt**. Notice that the order of regular expression is important.

2. Syntax Analysis

In this part, we will use the token generated by lexical analyzer and syntax rules are defined in **syntax.txt** to build a syntax tree.

We define a struct to represent all nodes in syntax tree.

```
struct treeNode
{
    char *value;
    int childNum;
    struct treeNode *child[10];
};
```

For convenient, we use a treeNode* array to record each child node and childNum to record the number of child. According to the rules in **syntax.txt**, the childNum can not exceed 10 (for-loop has the most number of child nodes who only has 9 child nodes), which means the treeNode* array with capacity 10 is enough.

Each token is terminal and is represented by a leaf node(set childNum to 0) in syntax tree, while nonterminal is represented by a non-leaf node.

A. Required features

- **Lexical error (error type A) when there are undefined characters or tokens in the SPL program, or identifier starting with digits**

  Simple regular expression can implement this feature.

  ```
  . { printf("Error type A at Line %d: Unknown characters %s\n", yylineno, yytext); }
  [^ \n\r\t\(\)\.\{\};,!&\*\[\]\|\<>=\+-]* { printf("Error type A at line %d: Unknown lexeme \"%s\" \n", yylineno, yytext); }
  ```

- Syntax error (error type B) when the program has an illegal structure, such as missing closing symbol.

  This feature can be implemented by adding `error` token in proper position.

  For example, an return statement is `Stmt: RETURN Exp SEMI`. If we want to detect a error about missing `SEMI` token, we can add an option grammar `RETURN Exp error`.

B. Optional features

- Supporting single-line and multi-line comments

  Obviously, regular expressions can recognize comments. Sometime, only use regular expressions to implement this feature is a little complex. To simplify the regular expression, this feature is implemented by using some functions in Flex Library.

  single-line comments //

  Code:

  ```
  \/\/[^\r\n]* {  }
  ```

  Alternatively,

  Code:

  ```
  "//" { char c; while((c=input()) != '\n'); }
  ```

  multi-line comments /* */

  Code:

  ```
  C1 [/]
  C2 [*]
  C3 [^*/]
  MULTILINE_COMMENT "/*"{C1}*({C3}{C1}*|{C2}|{C3})*"*/"
  ```

  Alternatively,

  Code:

  ```
  "/*" { char c; while (c=input()) { if (c == '*') { c=input(); if (c=='/') break; else
  unput(c);} } }
  ```

  After the start of comment, analyzer will only read the comments character until the first end of multi-line comment(`*/`) occurs.

- Supporting hexadecimal representation of integers such as 0x12. You should be able to detect illegal form of hex-int, like 0x5gg, and report lexical errors.

  ```
  INT_HEX -?(0[xX][0-9a-fA-F]{1,8})
  ```

  Hex-form integers always start with `0X` or `0x`. Each digit can be [0-9a-f]. And the number of digit in hex form integer is at least 1 but can not exceed 8.

- Supporting hex-form characters, such as `\x90`, also, you need to detect its illegal form like `\xt0` and report lexical errors.

  ```
  CHAR_HEX 0[xX][0-9a-fA-F]{1,2}
  ```

Hex-form char is similar with hex-form integers. The difference is that hex-form char surround by `''` and the number of digit is 1 or 2.

- Detecting nested multi-line comments, and report a syntax error

  Nested multi-line comments can be `/* c1 /* c2 */ c3 */`. `c3` part is illegal comment due to missing `/*`. But `/* c2` is legal because all `/*` after first `/*` are regarded as comment. Hence, this case can be detected by finding whether single `*/` after complete multi-line comment(`/* ... */`).

```
C1 [/]
C2 [*]
C3 [^*/]
C4 [^/]
MULTILINE_COMMENT "/*"{C1}*({C3}{C1}*|{C2}|{C3})*"*/"
ERROR_MULTILINE_COMMENT {MULTILINE_COMMENT}({C4}{C2}*|({C1}*{C3})|{C3})*"*/"
```

C. Bonus features

- for-statement

  In C language, the structure of for-statement is `for ( s1 ; s2 ; s3 ) {...}`. `s1` can be the definition of variable(s)(such like `int i = 0, j = 0`) or Exp list(such like `a = 0, b = 0`) or empty. `s2` and `s3` can be Exp list or empty.

  1. First of all, **lex.l** should be able to obtain `for` token in code. We add a simple regular expression to recognize it.

     ```
     for { yylval = createLeaf("FOR"); return FOR; }
     ```

  2. Then we add for-statement grammar in **syntax.y**.

     ```
     Stmt : ...
         | FOR LP Def ExpListEx SEMI ExpListEx RP Stmt `
         | FOR LP ExpListEx SEMI ExpListEx SEMI ExpListEx RP Stmt `
         ;
     ```

     Here, ExpListEx can be empty or a list of Exp. In code,

     ```
     ExpListEx: ExpList
         |
         ;
     ExpList: Exp
         | Exp COMMA ExpList
         ;
     ```

     Notice that `Def` already has `SEMI` in the end. So we should not add a redundant `SEMI` token in grammar.

  3. Finally, we can test the for-statement in following test case. This test case is in **test/for_test.spl**.

     ```
     int for_test()
     {
     ```

```
    int i = 0;
    for (int j = 0 ; j < 10 ; j = j + 1) {  }
    for ( ; ; ) {  }
    for ( ; k < 10 ; k = k + 1)
    {
        k = k * 100;
    }
    for (i = 1 ; i < 2 ; i = i + 1)
    {
        i = i * 10;
    }
    return i;
}
```

4. Error for-statement may miss `(` or `)` or `;`. We can set corresponding grammar to detect the error.

Note

- How to run splc?

  Use command `./bin/splc < test_file_path > output_file_path.out`