CS323 Project1

Lexical Analysis & Syntax Analysis

Team Members: Li Yuanzhao(11812420), Xu Xinyu(11811536), Jiang Yuchen(11812419)

I. Overview

In this project, we are required to implement lexical analysis and syntax analysis with lexer, syntaxer and other useful files in order to output a parser tree or possible lexical and syntax error information for given SPL(Sustech Programming Language) code. Our files can be run successfully with GCC version 7.4.0, GNU Flex version 2.6.4 and GNU Bison version 3.0.4 .

II. Design and Implementation

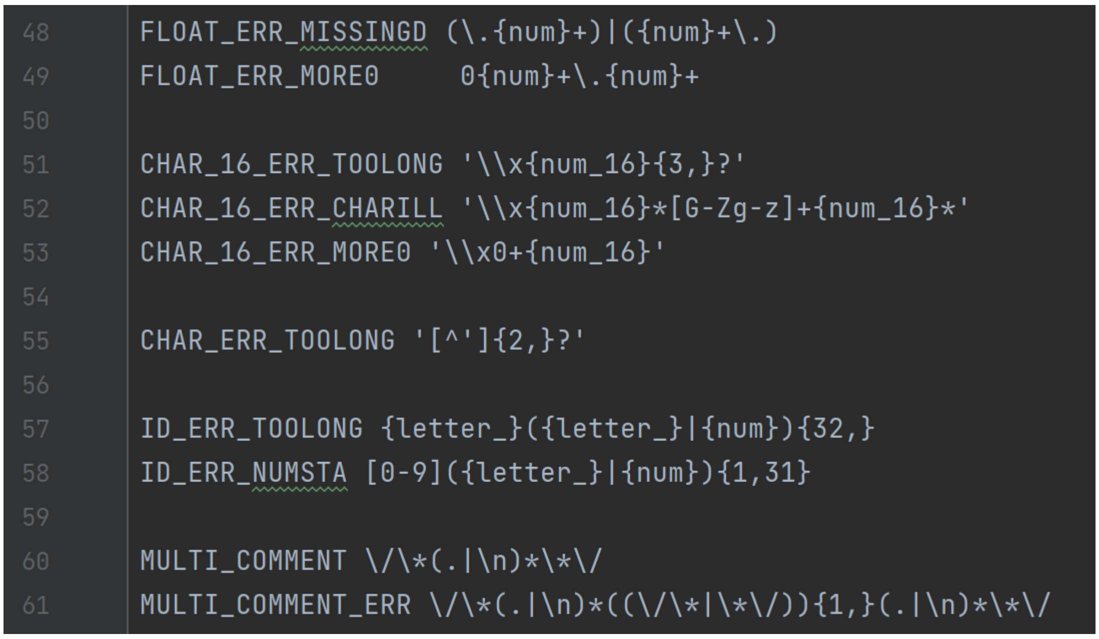
A. Lexer

In lexer part, we define a new class named `Node` which record the information of matched token for syntax analysis and final output.

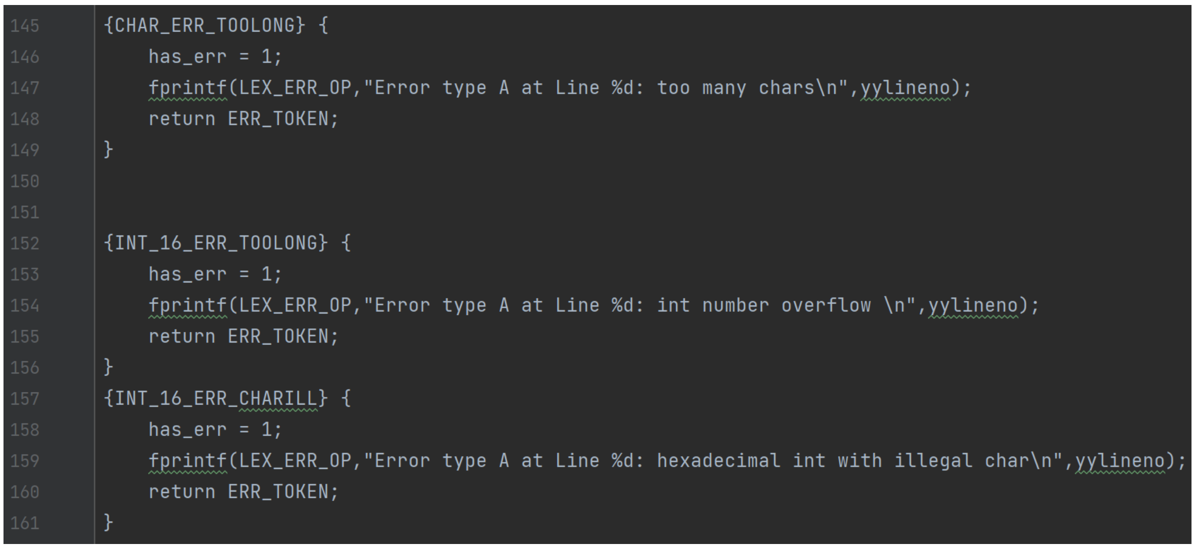
```
enum class Node_TYPE{
    INT,
    FLOAT,
    CHAR,
    NAME,   //IF ELSE ASSIGN etc
    ID,     //ALL IDENTIFIERS
    DTYPE,  //DATATYPE, INCLUDING INT,FLOAT & CHAR
    LINE
};

class Node{
private:
    string name;
    //string id_name;// == string_value
    Node_TYPE TYPE;
    union{
        int lineno;
        int int_value;
        float float_value;
        char char_value;
    };
    vector <Node*> child;


public:

    Node();

    void print(int depth);
    Node(int val);
    Node(float val);
    Node(char val);

    Node(string name);
    Node(string name,Node_TYPE type);
    Node(string name,int line_no);
    Node(string name,int line_no, vector<Node*>& child);

    void set_child(vector<Node*>&);

    void show(int depth);

};
```

We define the variable `has_err` to record whether there exists possible lexical and syntax error for final output. We declare the illegal and error situations in detail so that we can handle all possible errors. All lexical error will be reported here with line number.

Bonus: We also support single line comment symbol `//` and multiple line comment symbol `/* */` together with nest error besides given matching rules. This part will be tested in `./test-ex/` folder.



```
FLOAT_ERR_MISSINGD (\.{num}+)|({num}+\.)
FLOAT_ERR_MORE0     0{num}+\.{num}+

CHAR_16_ERR_TOOLONG '\\x{num_16}{3,}?'
CHAR_16_ERR_CHARILL '\\x{num_16}*[G-Zg-z]+{num_16}*'
CHAR_16_ERR_MORE0 '\\x0+{num_16}'

CHAR_ERR_TOOLONG '[^']{2,}?'

ID_ERR_TOOLONG {letter_}({letter_}|{num}){32,}
ID_ERR_NUMSTA [0-9]({letter_}|{num}){1,31}

MULTI_COMMENT \/\*(.|\n)*\*\/
MULTI_COMMENT_ERR \/\*(.|\n)*((\/\*|\*\/)){1,}(.|\n)*\*\/
```



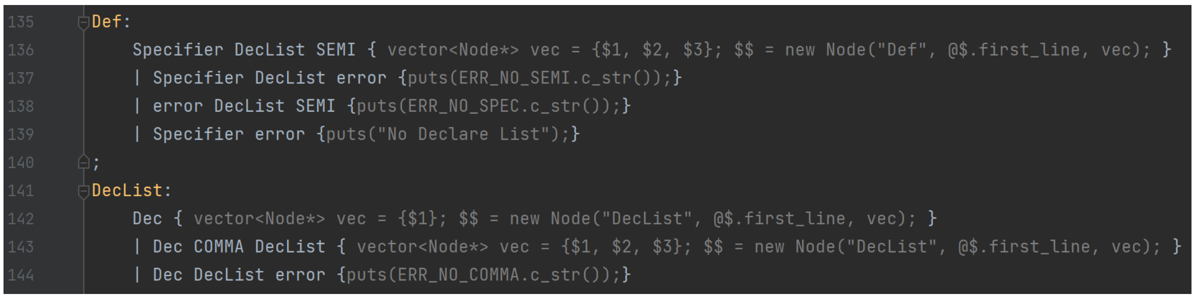
```
{CHAR_ERR_TOOLONG} {
    has_err = 1;
    fprintf(LEX_ERR_OP,"Error type A at Line %d: too many chars\n",yylineno);
    return ERR_TOKEN;
}


{INT_16_ERR_TOOLONG} {
    has_err = 1;
    fprintf(LEX_ERR_OP,"Error type A at Line %d: int number overflow \n",yylineno);
    return ERR_TOKEN;
}
{INT_16_ERR_CHARILL} {
    has_err = 1;
    fprintf(LEX_ERR_OP,"Error type A at Line %d: hexadecimal int with illegal char\n",yylineno);
    return ERR_TOKEN;
}
```

Figure.1 Some matching rules we defined in `lex.l`

B. Syntaxer

As mentioned in Appendix B, we construct our syntaxer which will accept tokens and make actions or report errors(type B). In this part we will take use of class `Node` not only for tokens' information but also for level differentiation. Finally the nodes will form up a tree to record those tokens' information when the program has no lexical and syntax error.

In syntaxer, we construct `vector<Node*> vec` to record the child nodes of the current node. If necessary, we can traverse the tree from root and output the whole parser tree as required.



```
Def:
    Specifier DecList SEMI { vector<Node*> vec = {$1, $2, $3}; $$ = new Node("Def", @$.first_line, vec); }
    | Specifier DecList error {puts(ERR_NO_SEMI.c_str());}
    | error DecList SEMI {puts(ERR_NO_SPEC.c_str());}
    | Specifier error {puts("No Declare List");}
;
DecList:
    Dec { vector<Node*> vec = {$1}; $$ = new Node("DecList", @$.first_line, vec); }
    | Dec COMMA DecList { vector<Node*> vec = {$1, $2, $3}; $$ = new Node("DecList", @$.first_line, vec); }
    | Dec DecList error {puts(ERR_NO_COMMA.c_str());}
```

Figure.2 Syntax Design

C. Other useful files

`spl_node.cpp` and `spl_node.hpp` are used to define the class `Node` and declare the fields and functions about it.

`main.cpp` are main function to start parsing and output the final result.

III. Test Cases

For evaluation purpose, our test cases contain 1 correct code, 2 type A errors and 2 type B errors, including illegal identifier, hexadecimal representation errors, int overflow errors (Type A) and missing or unexpected symbols(Type B) . All of original test cases are saved in `./test/` folder.

For extra test cases, we put them in `./test-ex/` folder which contains two test cases. They are used for testing single-line and multiple-line comment and multiple-line errors.

- Test case with Type A error

```
struct test{

    int valid_int,valid_hexint;
    char valid_hexchar = '\x0',valid_char = '?';
    char invalid_char = 'izfr';

};
int main(){
    int invalid_int_1 = 0295;
    int invalid_int_2 = 5920483965;

    int invalid_hexint_1 = 0x00242;
    int invalid_hexint_2 = 0xadg1u3;
    int invalid_hexint_3 = 0x123456789a;

    struct test test_struct;

    test_struct.valid_int = 24235;
    test_struct.valid_hexint = 0xabcdef;


}
```

- Ex-Test case with nested multiple error

```
int test_2()
{
  int a = 0;
  /*
  test multi-line comment nest
  /*  /* sdadas */ s ad a */
  sdadasdasdas
  cS3To3
  */
}
```

IV. Instructions

Change directory to the root path and using `make splc` to create `splc` in `./bin` root for spl codes' parsing. Then using `./bin/splc ./test/<file_name>` to create output parsing tree.