

CS323-Compilers First Report

XUJIA ZHANG, SID: 12011122

QI ZHANG, SID: 12010123

CHENGHUAN HUANG, SID: 11912908

requirement:

"Report directory contains a pdf file that illustrates your design and implementation, you should focus on the optional/bonus features you have realized, since the required part is rather simple and straight-forward. Your report should not exceed 4 pages, we suggest you to use 11pt font size and single-line spacing for main content."

- In CS323 course project, we will write a compiler for a toy programming language called SPL, the abbreviation for SUSTech Programming Language. SPL is a C-like programming.
- In the first phase, we use GNU Flex to build the lexical analyzer: (**lex.l**) and GNU Bison to build the syntax analyzer: (**syntax.y**).
- We pass all the test cases that provided by prof. Liu and write 2 python scripts (**test.py** and **test-ex.py**) to do automated test for our project.
- To confirm the accuracy of this project, we print both the tokens and the abstract syntax tree if we input a correct .spl file to the executable file, and error messages for an incorrect .spl file input.
- For further expansion, we use **g++** instead of **gcc** to compile lex.yy.c, syntax.tab.c and the cpp file we write. With the C++ grammar and library, we can implement our SPL compiler in other phases easier.

1 ENVIRONMENT

Tool	Version
OS	WSL2(Ubuntu 20.04)
IDE	VScode 1.72.2
Bison	3.5.1
Flex	2.6.4
Python	3.8.10
g++	9.4.0
makefile	4.2.1

2 DETAIL

2.1 Util

To print the abstract syntax tree, we write a cpp class called **Node** to store the relevant information of tokens.

1. Node.hpp

```
1 #include <string >
2 #include <iostream >
3 #include <vector >
4 #include <initializer_list >
```

```

5
6 enum class TYPE
7 {
8     MEDIAN,    // 1 exp: StmtList (4)
9     OTHER,     // 2 exp: SEMI
10    TYPE,      // 3 exp: TYPE: int
11    INT,       // 4 exp: INT: 30
12    CHAR,      // 5 exp: CHAR: 'c'
13    FLOAT,     // 6 exp: FLOAT: 30.5
14    ID,        // 7 exp: ID: b
15    NOTHING,   // 0 exp: /*empty here*/
16 };
17
18 class Node
19 {
20 public:
21     enum TYPE type;
22     std::string name;           // AST: node's name
23     std::string content;       // possible cotent: 'c',30.0,2193,abcd
24     int line_num;             // the line in the context
25     std::vector<Node *> child; // the children of the node
26
27 public:
28     explicit Node(enum TYPE type, std::string name, const char *content, int
        line);
29     ~Node() = default;
30     void addChild(std::initializer_list<Node *> childs);
31     static void print(Node *node, long depth);
32 };

```

With the help of this class, we can store the line number, type and value of a matching lexeme into `yylval`, which allows us to print the AST and debug easily.

2. lex.l Example

```

1 "struct" {yylval.node = new Node(TYPE::OTHER, "STRUCT", "struct", yylineno); return
    STRUCT;}
2 "if"     {yylval.node = new Node(TYPE::OTHER, "IF", "if", yylineno); return IF;}
3 "else"   {yylval.node = new Node(TYPE::OTHER, "ELSE", "else", yylineno); return
    ELSE;}
4 "while"  {yylval.node = new Node(TYPE::OTHER, "WHILE", "while", yylineno); return
    WHILE;}

```

```

5 "return">{yyval.node = new Node(TYPE::OTHER, "RETURN", "return", yylineno); return
    RETURN;}

```

2.2 Lexical Analyzer

2.2.1 *basic part.* We implement INT, FLOAT, CHAR and ID four special token and other keyword token.

- INT represents an unsigned-integer. It contains of two part, hex-form(base 16) and decimal form(base 10). Decimal form is a consecutive sequence of digits(0-9), and it cannot start with “0”, except 0 itself, hex-form starts with "0x" or "0X", followed by a sequence of hex-digits(a-f,0-9).
- FLOAT represents an unsigned floating-point5 number.A valid floating-point number contains a dot(.), the left side of dot is decimal form of INT, the right side of dot is a consecutive sequence of digits(0-9)
- CHAR represents a single character.It contains of two part,normal char and hex-form char.Normal char contains a pair of single-quotes and one char between them.Hex-form char contains a pair of single-quotes and hex-form int between them.
- ID stands for identifier. A valid identifier cannot start with digit(0-9) and consists of 3 types of characters: the underscore (_), digits (0-9), and letters (A-Z and a-z).
- Other known token represents keyword. For any other keyword, We will deal with it as in section 2.1 lex.l example.
- If encounter illegal or unknown token, we will output error message and return UNKNOWN token.

```

1 letter_      [a-zA-Z_]
2 hex          (0|[1-9a-fA-F][0-9a-fA-F]*)
3 digit        [0-9]
4 id           {letter_}({letter_}|{digit})*
5 decimal_uint {digit}+
6 hexadecimal_uint 0(x|X){hex}
7 int          {decimal_uint}|{hexadecimal_uint}
8 normal_char  ('').('')
9 hex_form_char ('')(\x){hex}('')
10 char        {normal_char}|{hex_form_char}
11 float       (0|[1-9][0-9]*)\.(\.){digit}+
12 %%
13 ...
14 .           {printf("Error type A at Line %d: unknown lexeme %s\n",
    yylineno,yytext);return UNKNOWN;}\\error
15 ...

```

2.3 Syntax Analyzer

2.3.1 *basic part.* For token, we define tokens in syntax.y follows operation priority in SPL.

```

1 %nonassoc UELSE
2 %nonassoc ELSE
3 %nonassoc UMINUS
4 %right ASSIGN
5 %left OR AND
6 %left LT LE GT GE NE EQ
7 %left PLUS MINUS
8 %left MUL DIV
9 %right NOT
10 %left LP RP LB RB DOT

```

For grammar, we follow Grammar Specification, and also do error recovery in syntax analyzer.

If the file follows grammar rule, we will treat token as node and put them into a tree. If the file contains lexical and syntax error, we will read the whole file and output the error message. Type A error represent lexical error, type B error represent syntax error

```

1 ExtDef: error ExtDecList SEMI {printf("Error type B at Line %d:Missing
    specifier\n",@$.first_line); type_B_error=1;}
2 | Specifier ExtDecList SEMI    {$$=new Node(TYPE::MEDIAN,"ExtDef","",@$.
    first_line); $$->addChild({$1,$2,$3});}
3 | Specifier SEMI              {$$=new Node(TYPE::MEDIAN,"ExtDef","",@$.
    first_line); $$->addChild({$1,$2});}
4 | Specifier FunDec CompSt     {$$=new Node(TYPE::MEDIAN,"ExtDef","",@$.
    first_line); $$->addChild({$1,$2,$3});}
5 ...
6 Exp: UNKNOWN {type_A_error = 1;}|....

```

2.4 bonus

2.4.1 comment. We have implement both C-style single-line and multi-line comments.

Single line comment starts with // and all symbols follows until the end of the line would be ignored.

Multi-line comment starts with /* and ends with */ and all symbols between them would be ignored.

```

1 "/*" {
2     char c;
3     while(1){
4         while ((c = yyinput()) != '/'){};
5         if(yytext[yytextlen-1] == '*') {
6             break;
7         }
8     }
9 }

```

```

10  "/" {
11  he following chars until seeing a newline character
12      while((c = yyinput()) != '\n');
13      unput(c);
14  }

```

2.4.2 *for-loop*. We add a rule **Stmt:FOR LP Exp SEMI Exp SEMI Exp RP Stmt** to implement for-loop

2.4.3 *an example for bonus*. This is an example to show the implementation for all bonus.

```

1  //for loop
2  int test_1(int n, int i){
3      /*
4      all of the symbols between multi-line comment would be ignored.
5      int qwq = 19260817;
6      */
7      int res = 0;
8      for (i = 0; i < n; i = i + 1){
9          res = res + i;
10     }
11 }

```

Output

```

1
2 Program (2)
3   ExtDefList (2)
4     ExtDef (2)
5       Specifier (2)
6         TYPE: int
7       FunDec (2)
8         ID: test_1
9         LP
10        VarList (2)
11          ParamDec (2)
12            Specifier (2)
13              TYPE: int
14            VarDec (2)
15              ID: n
16          COMMA
17        VarList (2)
18          ParamDec (2)

```

```

19         Specifier (2)
20         TYPE: int
21         VarDec (2)
22         ID: i
23     RP
24 CompSt (2)
25     LC
26     DefList (7)
27     Def (7)
28         Specifier (7)
29         TYPE: int
30         DecList (7)
31         Dec (7)
32         VarDec (7)
33         ID: res
34         ASSIGN
35         Exp (7)
36         INT: 0
37     SEMI
38 StmtList (8)
39     Stmt (8)
40     FOR
41     LP
42     Exp (8)
43         Exp (8)
44         ID: i
45         ASSIGN
46         Exp (8)
47         INT: 0
48     SEMI
49     Exp (8)
50         Exp (8)
51         ID: i
52     LT
53     Exp (8)
54         ID: n
55     SEMI
56     Exp (8)
57         Exp (8)
58         ID: i

```

```
59      ASSIGN
60      Exp (8)
61      Exp (8)
62      ID: i
63      PLUS
64      Exp (8)
65      INT: 1
66  RP
67  Stmt (8)
68  CompSt (8)
69  LC
70  StmtList (9)
71  Stmt (9)
72  Exp (9)
73  Exp (9)
74  ID: res
75  ASSIGN
76  Exp (9)
77  Exp (9)
78  ID: res
79  PLUS
80  Exp (9)
81  ID: i
82  SEMI
83  RC
84  RC
```

REFERENCES