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 your 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++ grammer and library, we can implement our SPL compiler in other phases esaier.

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 Utill

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

1. Node.hpp

```
#include <string>
#include <iostream>
#include <vector>
#include <initializer_list>
```

```

enum class TYPE
{
    MEDIAN,    // 1 exp: StmtList (4)
    OTHER,     // 2 exp: SEMI
    TYPE,      // 3 exp: TYPE: int
    INT,       // 4 exp: INT: 30
    CHAR,      // 5 exp: CHAR: 'c'
    FLOAT,     // 6 exp: FLOAT: 30.5
    ID,        // 7 exp: ID: b
    NOTHING,   // 0 exp: /*empty here*/
};

class Node
{
public:
    enum TYPE type;
    std::string name;             // AST: node's name
    std::string content;          // possible cotent: 'c',30.0,2193,abcd
    int line_num;                 // the line in the context
    std::vector<Node *> child;    // the children of the node

public:
    explicit Node(enum TYPE type, std::string name, const char *content, int
        line);
    ~Node() = default;
    void addChild(std::initializer_list<Node *> childs);
    static void print(Node *node, long depth);
};
```

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

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

```
"return"{yylval.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 formal(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.

```
letter_          [a-zA-Z_]
hex              (0|[1-9a-fA-F][0-9a-fA-F]*)
digit            [0-9]
id               {letter_}({letter_}|{digit})*
decimal_uint     {digit}+
hexadecimal_uint 0(x|X){hex}
int              {decimal_uint}|{hexadecimal_uint}
normal_char      (\').(\')
hex_form_char    (\')(\\)x{hex}(\')
char             {normal_char}|{hex_form_char}
float            (0|[1-9][0-9]*)(\.){digit}+
%%
...
.                {printf("Error type A at Line %d: unknown lexeme %s\n",
    yylineno,yytext);return UNKNOWN;}\\error
...
```

2.3 Syntax Analyzer

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

```
%nonassoc UELSE
%nonassoc ELSE
%nonassoc UMINUS
%right ASSIGN
%left OR AND
%left LT LE GT GE NE EQ
%left PLUS MINUS
%left MUL DIV
%right NOT
%left LP RP LB RB DOT
```

For grammer, 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

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

```
"/*" {
    char c;
    while(1){
        while ((c = yyinput()) != '/'){};
        if(yytext[yyleng-1] == '*') {
            break;
            }
    }
}
```

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

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.

```
//for loop
int test_1(int n,int i){
    /*
    all of the symbols between multi-line comment would be ignored.
    int qwq = 19260817;
    */
    int res = 0;
    for (i =0; i < n; i = i + 1){
        res = res + i;
    }
}
```

Output

```

Program (2)
  ExtDefList (2)
    ExtDef (2)
      Specifier (2)
        TYPE: int
      FunDec (2)
        ID: test_1
        LP
        VarList (2)
          ParamDec (2)
            Specifier (2)
              TYPE: int
            VarDec (2)
              ID: n
          COMMA
          VarList (2)
            ParamDec (2)
```

```
            Specifier (2)
              TYPE: int
            VarDec (2)
              ID: i
        RP
      CompSt (2)
        LC
        DefList (7)
          Def (7)
            Specifier (7)
              TYPE: int
            DecList (7)
              Dec (7)
                VarDec (7)
                  ID: res
                ASSIGN
                Exp (7)
                  INT: 0
            SEMI
        StmtList (8)
          Stmt (8)
            FOR
            LP
            Exp (8)
              Exp (8)
                ID: i
              ASSIGN
              Exp (8)
                INT: 0
            SEMI
            Exp (8)
              Exp (8)
                ID: i
              LT
              Exp (8)
                ID: n
            SEMI
            Exp (8)
              Exp (8)
                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