CS323 Project Phase1 Report

小组成员：12011136 陈茜，12010339 王思懿，12012438 杨可芸

Compiler Design and Implementation

Error

这一部分我们利用 my_error.hpp, my_error.cpp 以及共享变量 has_error 来处理异常。my_error.hpp, my_error.cpp 用于处理语法异常，has_error=1 用于表示当前代码存在错误，不打印语法树。

```C++
typedef enum {
    MISS_SEMI,  //缺失;
    MISS_PAREMTHESIS,  //缺失()
    MISS_BRACKET,  //缺失[/]
    MISS_CURLY_BRACE,  //缺失{/}
    MISS_SPEC  //缺失Specifier
} MY_ERROR_TYPE;
void my_error(MY_ERROR_TYPE type, int line_num,char content=' ');
```

我们小组对定义了以上五类 syntax error，当在语法分析匹配到含有 error 的语法规则的时候进入错误恢复，调用 my_error 方法打印 syntax error 错误信息，并且把 has_error 设置为 1。

Lexical Analysis

这一部分我们通过 flex 对 spl 定义词法规则，并在 spl 词法的定义和规则之上对 spl 代码进行词法分析和错误处理。在 lex.l 中定义了正确 token 和错误 token 以及相对应的规则。在检测到错误 token 的时候，进入错误恢复，将 has_error 设为 1 并打印对应的 lexical error 错误信息。

Syntax Analysis

这一部分我们通过 bison 对 spl 定义语法规则，并在 spl 语法的定义和规则之上对 spl 代码进行语法分析和错误处理。

```C++
/* declared types */
%union{
    Node* node_ptr;
}
/* declared tokens */
%nonassoc <node_ptr> ILLEGAL_TOKEN
%nonassoc LOWER_THAN_ELSE
%nonassoc <node_ptr> ELSE
%token <node_ptr> TYPE INT CHAR FLOAT STRUCT ID
%token <node_ptr> IF WHILE RETURN FOR
%token<node_ptr> COMMA
%right <node_ptr> ASSIGN
%left <node_ptr> OR
%left <node_ptr> AND
%left <node_ptr> LT LE GT GE NE EQ
%left <node_ptr> PLUS MINUS
%left <node_ptr> MUL DIV
%right <node_ptr> NOT LP RP LB RB
%left <node_ptr> DOT
%token <node_ptr> SEMI LC RC
/*ERROR*/
%token <node_ptr> INVALID_CHAR WRONG_ID UNKNOWN_CHAR INVALID_NUMBER
/* declared non-terminal */
%type <node_ptr> Program ExtDefList ExtDef ExtDecList
%type <node_ptr> Specifier StructSpecifier
%type <node_ptr> VarDec FunDec VarList ParamDec
%type <node_ptr> CompSt StmtList Stmt
%type <node_ptr> DefList Def DecList Dec
%type <node_ptr> Exp Args
```

在 syntax.y 中我们定义 type 都是我们自定义的 Node*，为了方便后续打印 Parse Tree 。

我们通过 %left，%right，%nonassoc 来规定运算符的结合性，按照运算符的位置定义运算符之间的优先级，并且通过 %prec 指定优先级以消除 if 和 if else 的二义性。我们在每一条语法对应的语义动作中，调用 Node 构造器，构建 Parse Tree，当递归到

Program 时，判断词法分析和语法分析中途是否存在错误(has_error=1)，如果不存在，则输出语法分析树。

Parse Tree

这一部分我们利用 tree.hpp, tree.cpp 来管理节点构建语法树。

C++

```
typedef enum NodeType{
    Type,
    Int,
    Char,
    Float,
    Id,
    TERMINAL,
    NONTERMINAL
} NodeType;
class Node{
public:
    NodeType nodetype;  //节点类型
    string name;  //节点名字 eg. Program,Specifier……
    int line_num;  //位于代码文件的第几行
    /*union: Members share the same memory*/
    union{
        int int_value;  //储存INT对应的值
        char* char_value;  //储存CHAR对应的字符
        float float_value;  //储存FLOAT对应的值
    };
    int nodes_num = 0;  //子节点个数
    queue<Node*> children;  //子节点队列
    //语法分析时的node构造器
    explicit Node(NodeType nodetype, string name, int nodes_num, int l
    //词法分析时terminal的node构造器
    explicit Node(string name);
    //词法分析时char,type,id的构造器
    explicit Node(NodeType nodetype, char* char_value);
    //词法分析时int的构造器
    explicit Node(NodeType nodetype, string name, int int_value);
    //词法分析时float的构造器
    explicit Node(NodeType nodetype, string name, float float_value);
};
//打印语法树
```

```
void printTree(Node* root, int whiteSpace=0);
//打印node节点
void print(Node* node, int whiteSpace);
```

我们设计了枚举类 NodeType 来表示语法树上不同的节点类型，根据不同的 NodeType 在输出的时候采取不同的策略。接着我们设计了 Node 类，来表示语法树上的不同节点。printTree 方法和 print 方法递归地输出整棵语法分析树，如果节点是 NONTERMINAL，则调用 printTree 继续递归；如果是 TERMINAL，则调用 print，结束递归。

Extended Features

Single- and Multi-Line Comment

在 lex.l 中，我们通过单行注释和多行注释的词法定义和规则来进行匹配，对应 test_ex/test_1.spl。

C++

```
COMMENT "//".*$
MULTIPLE_COMMENT "/*"((("*"[^/])?)|[^*])*"*/"
%%
{COMMENT} {
    fprintf(stdout,"Comment at Line %d: %s\n",
              yylineno, yytext);
}
{MULTIPLE_COMMENT} {
    fprintf(stdout,"Multiple comment at Line %d: %s\n",
              yylineno, yytext);
}
```

File Inclusion

在 lex.l 中，我们通过 file inclusion 和 library inclusion 的词法定义和规则来进行匹配，对应 test_ex/test_2.spl。

C++

```
INCLUDE_FILE #include[ \t]+"\""[^\n]*"\""
INCLUDE_LIB #include[ \t]+"<"[^\n]*">"
%%
{INCLUDE_FILE} {
```

```
    fprintf(stdout,"File inclusions at Line %d: %s\n",
                yylineno, yytext);
}
{INCLUDE_LIB} {
    fprintf(stdout,"Lib inclusions at Line %d: %s\n",
                yylineno, yytext);
}
```

For Statements

在 Flex 中，我们通过 FOR 的词法定义和规则来进行匹配

C++

```
FOR "for"
%%
{FOR} {yylval.node_ptr = new Node("FOR"); return FOR;}
```

在 Bison 中，通过定义 FOR token 和 for 循环语句的产生式来匹配 for statement，并且在语法规则中添加 error 符号来对 for 循环语句进行异常处理，错误恢复。

C++

```
/* declared tokens */
%nonassoc <node_ptr> ILLEGAL_TOKEN
%nonassoc LOWER_THAN_ELSE
%nonassoc <node_ptr> ELSE

%token <node_ptr> TYPE INT CHAR FLOAT STRUCT ID
%token <node_ptr> IF WHILE RETURN FOR
%%
Stmt: FOR LP Exp SEMI Exp SEMI Exp RP Stmt
    {
        $$=new Node(NONTERMINAL, "Stmt", 9, @$.first_line, $1, $2, $3,
    }
    | FOR error Exp SEMI Exp SEMI Exp RP Stmt
    {
        my_error(MISS_PAREMTHESIS, @$.first_line,'('); has_error=1;
    }
    | FOR LP Exp SEMI Exp SEMI Exp error Stmt
    {
        my_error(MISS_PAREMTHESIS, @$.first_line,')'); has_error=1;
```

```
    }
;
```

对应 test_ex/test_1.spl。