Lexical Analysis & Syntax Analysis

小组成员姓名学号：李昕（12012138）、廖铭骞（12012919）

FLEX部分

我们检测了正确格式的token和错误格式的token。正确的token可以帮助bison部分做语法分析，错误的token也可以帮助bison做错误恢复，将整个程序分析完。我们通过一个int类型的共享变量has_error作为判断词法分析部分是否出错。如果出错则将`has_error`变量置为1，最后将不会输出语法分析树。

BISON部分

在这一部分我们通过构建符合`spl`的语法规则，并且在相应的规约上面加上错误处理。

不管token是终结符还是非终结符，我们认定它的类型都是`Node*`。

```
%union{
    Node* node_ptr;
}
```

对于各种运算符，按照优先级从低到高，依此从上定义到下。通过`%prec`指定优先级以消除部分二义性。

对于每一条生产式，我们调用`Node`构造器，构建起一层一层的语法分析树。当递归到Program时，判断词法分析和语法分析是否中途存在错误。如果中途不存在错误，则输出语法分析树。

MAKEFILE部分

由于在语法分析树部分使用了queue，所以我们的程序需要使用g++编译。

并且我们使用`extern "C" int yylex()`来解决链接问题。

my_error报错模块

为了更加全面地处理报错信息，我们专门建立了一个`my_error`报错模块。在这个模块当中，我们使用枚举类型来表示各种错误信息，并且分别对其进行报错输出处理。在`syntax.y`文件当中，在发现错误时，通过向函数中传递错误的类型以及行号，即可在不中断程序的情况下对错误进行处理，并且能够较为准确地定位到错误对应的行号，方便用户调试错误。

构建语法分析树

我们设计了枚举类`NodeType`，来表示语法分析树上不同节点的类型。根据不同的节点类型，在输出的时候采用不同的策略。

```
typedef enum NodeType{
    Type,
    Int,
    Char,
    Float,
    Id,
    TERMINAL,
    NONTERMINAL
} NodeType;
```

我们设计了类`Node`，来表示语法分析树上的每个节点。通过Node中`nodetype`来标志节点类型，`name`表示该节点的名字，`nodes_num`表示该节点子节点的个数，`int_value`，`char_value`，`float_value`相应节点类型的值，`line_num`表示该节点所对应的行数。我们设计了五种构造器用于不同的节点类型。

通过`explicit Node(NodeType nodetype, string name, int nodes_num, int line_num, ...)`将生产式右边的的子节点作为可变参数传入构造器函数。在构造器中，我们将子节点push进入队列nodes_queue。

```
class Node{
public:
    NodeType nodetype;
    string name;
    union{
        int line_num;
        int int_value;
        char* char_value;
        float float_value;
    };
    int nodes_num = 0;
    queue<Node*> nodes_queue;

    // 用于 bison
    explicit Node(NodeType nodetype, string name, int nodes_num, int line_num, ...);

    // 用于 lex
    // 用于 TERMINAL
    explicit Node(string name);

    // 用于 CHAR, Id
    explicit Node(NodeType nodetype, char* char_value);

    // 用于INT
    explicit Node(NodeType nodetype, string name, int int_value);

    // 用于 FLOAT
    explicit Node(NodeType nodetype, string name, float float_value);
};
```

我们通过以下两个函数完成对语法分析树的输出

```
void printTree(Node* root, int space=0);

void print(Node* node, int space);
```

`printTree`和`print`结合不断递归地输出整棵语法分析树。如果判断该节点是NONTERMINAL，则调用`printTree`继续往下递归；如果是TERMINAL，则调用`print`，停止递归。

BONUS部分

对于bonus部分，我们做了单行注释和多行注释。测试用例分别对应`test-ex/comment_1.spl`和`test-ex/comment_2.spl`。

实现的代码如下：

通过`COMMENT "//".*$`匹配单行注释。

通过`MULTIPLE_COMMENT "/*"((("*"[^/])?)|[^*])*"*/"`匹配多行注释。

对于测试的文件`comment_1.spl`我们在代码当中嵌入单行注释

```
int t01_12012138(int a, int b, int c)
{
  int d = 0;
  if(d < a || (b == c || a == 0)){
    d = a;
  }
  // comment
}
```

而在测试文件`comment_2.spl`中，我们在代码中嵌入多行注释

```
int t02_12012138(int a, int b, int c)
{
/*
multiple line comment
----------------------
*/
  int d = 0;
  if(d < a || (b == c || a == 0)){
    d = a;
  }
}
```

两个文件都能够正确地解析出语法树结构，可以证明我们的注释支持是正确的。相关的测试文件见代码部分。