语法实验

实验目标

在词法实验的基础上，借助yacc工具实现一个语法分析器，要求编写适当的语义动作，能够按照规约顺序输出需要用到的规约规则，同时绘制SysY代码的语法树。

实验指导

（1）yacc使用方法

```
%{
声明
%}
辅助定义
%%
语法规则
%%
用户子程序
```

- 声明

  该部分用于编写需要使用的头文件、全局变量等，可以为空，代码语法同C。

  我在实验过程中把main函数和其他辅助函数也写在这个部分，也是可行的。

- 辅助定义

  使用$\%token$定义终结符，使用$\%left$定义优先级。定义终结符的时候，两个终结符之间不需要逗号，用空格隔开即可。

  在该实验中，需要修改之前的词法分析程序。在词法分析器中，当匹配到一个终结符时，需要return在语法分析器中对应的终结符。由于此时词法分析器作为语法分析器的一部分，因此词法分析器中不再需要main函数。

- 语法规则

  对于每个语法规则，在匹配空字符串的时候建议加上注释，防止忘记。匹配其他规则时，对于每两个（非）终结符之间用空格隔开。匹配上之后，在大括号内部编写语义动作，即建立语法树。

可以使用$1来访问第一个匹配上的（非）终结符，用$2来访问第二个匹配上的（非）终结符，依次类推。使用$$来设置规约元素的值。终结符的值可以在词法分析器中使用yylval进行传递。一般来说，每一个（非）终结符的值都是一个int。

```
STATEMENT : /* empty */ {
                ...
            }
            | STA STB {
                $$ = $1 + $2
                // 此处为一个例子：
                // 将STATEMENT的值设置为STA的值加上STB的值
            }
            | ...
            ;
```

- 用户子程序

  编写main函数和其他辅助函数。

  需要注意的是，main函数中需要调用yyparse函数进行语法分析，而不用调用yylex。同时，需要将stdin绑定在需要编译的文件上。

编写完程序之后，需要先使用flex生成lex.yy.c，然后再使用yacc生成y.yab.c，然后再编译过程中链接这两个程序，最后生成可执行文件。

```
flex ./FILE_FLEX.l
yacc -d ./FILE_YACC.y
gcc y.tab.c lex.yy.c -o mc -O2 -w
```

（2）graphviz使用方法

该工具仅供参考，可以自己选择其他绘图工具。

- 定义有向图：

  在大括号内部编写结点之间的连边关系。

```
digraph " "{
    ...
}
```

- 设置结点形状：

```
node [shape = record,height=.1]
```

- 定义节点：

  中括号外为结点的名称（不会显示在图片中）。

  中括号中的双引号内部为结点每个部分的名称。尖括号中为该部分的标号，后续的空格为该部分的名称（显示在图片中的）

```
NAME_NODE[label = "<f0> NAME0|<f1> NAME1"];
```

- 结点连边：

  连边可以从一个结点的某个部分联想另一个结点。

  下面是一个从node0结点的f0部分连向node1结点的示例：

```
"node0":f0->"node1";
```

- 在命令行中使用如下指令，将Tree.dot文件转化为Tree.png文件：

```
dot -Tpng -o Tree.png Tree.dot
```

（3）注意事项

- if-else

  注意if-else语句中的移进-规约冲突：

```
if (...) ...
    if (...) ...
    else ...
```

  使用$\%left$和$\%prec$定义语句的优先级，使得if-else的优先级比if的优先级更高。

- 四则运算、条件语句优先级问题？
- 错误恢复（可选）

  错误恢复旨在尽可能多地找出程序中的错误语法，以提高编程效率。

  在yacc中，可以使用自带的终结符$error$通配错误的语法，而不中止编译过程。当匹配到$error$终结符时，会自动调用yyerror函数。

- 改写语法

大部分语法可以直接照抄文档，但是由于yacc不支持正则表达式，在编写规则的时候需要适当的修改语法，使其能够被识别。

通过定义多条类似的语法，实现正则表达式中“出现一次或多次”、“至多出现一次”等功能。

- 尽量使用左递归