语法分析器

实验环境

- 操作系统：Ubuntu 20.04.2 LTS
- CPU：Intel® Core™ i7-9750H CPU @ 2.60GHz
- 内存：32GB 2400MHz

文件内容

- $./code$：实现语法分析器的代码集合
    - $grammar.y$：基于$bison$的语法分析器
    - $morphology.l$：基于$flex$的此法分析器
    - $test.sy$：测试程序
    - $Detail.txt$：生成的规约规则
    - $Tree.dot$：基于$Graphviz$生成抽象语法树的文件
    - $Tree.png$：抽象语法树可视化
    - $run.sh$：样例测试脚本
    - $error.png$：错误恢复测试截图
- $./report.md$：实验报告的md版本
- $./report.pdf$：实验报告的pdf版本

指令简介

- 编译并运行

```
(bash) ~/$ flex ./morphology.l
(bash) ~/$ yacc -d ./grammar.y
(bash) ~/$ gcc y.tab.c lex.yy.c -o mc -O2 -w
(bash) ~/$ ./mc PATH_TO_CODE
```

- 语法树可视化

```
(bash) ~/$ dot -Tpng -o Tree.png Tree.dot
```

实现细节

- 词法分析器对于不同的终结符返回不同的符号进行区分，对终结符进行标号。
- 设置终结符，如下图所示：

```
%token NOELSE
%token INT MAIN RETURN IF BREAK CONTINUE WHILE CONST ELSE VOID
%token IDENT
%token BC BS BLB BRB BLM BRM BLL BRL
%token OE OP OS OM OD OMOD OEE ONE OL OG OLE OGE ON OAND OOR
%token NUMBER
%token ERROR
```

- 对于每一个语法，按照产生式进行匹配，同时尽量使用左递归。然后将所有可以规约的元素当成当前节点的儿子节点，使用一个结构体进行维护。下面的代码为一个例子：

```
CompUnit:
    /* empty */ {$$ = 0;}  /* 可以匹配到空 */
    | CompUnit oCompUnit {  /* 可以匹配一个编译单元 */
        fprintf (fd, "CompUnit -> CompUnit {CompUnit}\n");
        /* 输出规约规则 */
        $$ = ++cnt; a[$$].size = 0; /* 记录父亲节点的编号 */
        strcpy(Labe[$$], "CompUnit"); /* 标记父亲节点的名称 */
        if ($1) a[$$].child[a[$$].size++] = $1;
        if ($2) a[$$].child[a[$$].size++] = $2;
        /* 标记两个儿子 */
    }
;
```

- 分析结束后使用深度优先遍历构建语法树，语法树的可视化基于$Graphviz$实现。
- 对于$if-else$和if之间的移进-规约冲突，使用$\%left$和$\%prec$定义二者之间的优先级关系，前者移进优先级更高，避免冲突。
- 根据产生式可以发现，加减乘除等运算，可以避免之间的冲突，无需定义这些运算的优先级。

错误恢复

- 对于常数定义、变量定义、函数定义、$Block$内容的编译错误，使用$bison$自带的终结符$error$进行判断。当发生意料中错误，可以跳过这个错误，对后续的程序进行编译。