编译器（SysY）—实验报告

实验环境

- 操作系统：Ubuntu20.04（wsl）
- 处理器：Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- 内存：32GB 2400MHz
- 编译器：gcc.exe (MinGW.org GCC-8.2.0-5) 8.2.0

文件内容

- ./编译器（SysY）—实验报告.pdf：本实验报告
- ./编译器（SysY）—实验报告.md：本实验报告的markdown版本
- ./code：代码文件夹
    - ./grammar.y：语法分析和语义分析的源代码
    - ./morphology.l：词法分析的源代码
    - ./test.sy：测试程序1
    - ./test1.sy：测试程序2
    - ./run.sh：快速运行程序的脚本
    - ./assemble.s：测试程序1转化成汇编代码的结果

词法分析

运行方式

```
(bash) ~/$ flex mycompiler.l
(bash) ~/$ gcc lex.yy.c -o MyCompiler
(bash) ~/$ ./MyCompiler ./test.sy > ans.out
```

- 使用$flex$对代码进行转化。
- 使用gcc对生成的$lex.yy.c$文件进行编译，
- 程序接受一个命令行参数，指明需要进行词法分析文件所在的路径。

程序内容

1）处理注释

对于注释的处理尽量放到开头以提高优先级，避免注释的内容被其他的规则匹配，导致出现错误。

- 匹配单行注释。

    ```
    "//".*
    ```

    最开头匹配两个斜杠，然后匹配任意多个字符。由于 . 不会匹配回车，所以这样能保证匹配的为单行的注释。

- 匹配多行注释

```
"/*"([^*]|[\r\n]|("*"+([^*/]|[\r\n])))*"*"+"/"
```

先匹配注释的开头，为了避免**多个注释被匹配到一起**的情况，对注释的内容需要进行特殊处理。

首先是不能有*，如果出现*号，那么后面必须不能是/，防止将多个注释作为同一个注释一起匹配而导致错误。

2）处理算符

```
"="|"+"|"-"|"*"|"/"|"%"|"=="|"!="|"<"|">"|"
<="|">="|"!"|"&&"|"||"|"&"|"|"|"^"
```

使用双引号对符号进行转义，所有的符号之间是或的关系，任意一个都能匹配上。

3）处理界符

```
","|";"|"{"|"}"|"["|"]"|"("|")"
```

使用双引号对符号进行转义，所有的符号之间是或的关系，任意一个都能匹配上。

4）处理关键字

```
"int"|"main"|"return"|"if"|"break"|"continue"|"while"|"const"|"else
"
```

使用双引号对关键字进行转义，所有的关键字之间是或的关系，任意一个都能匹配上。

5）处理标识符

先进行辅助定义：

```
digit [0-9]
letter [A-Za-z_]
```

然后借助辅助定义对标识符进行匹配：

```
{letter}({letter}|{digit})*
```

最开始一定是一个字母或者下划线，接下来可以没有后续，也可以有一个或多个字母、数字或下划线。

6）处理常量

先进行辅助定义：

```
numberD (-?)([1-9][0-9]+)|([0-9])
numberO (-?)(0[0-7]+)
numberH (-?)("0x"|"0X")([0-9a-fA-F])+
```

然后通过辅助定义进行匹配：

```
{numberD}|{numberO}|{numberH}
```

- *numberD*用于匹配十进制整数，多位数的情况下第一位不能是0，一位数的情况下第一位可以是任何数。
- *numberO*用于匹配八进制整数，一定以0开头，接下来可以是任何数。
- *numberH*用于匹配十六进制整数，一定以“0x”或“0X”开头，接下来可以是任何数。

匹配到常量的时候，需要对该常量的位数进行判定，如果该常量的长度大于10位，需要报*Warning*，但是该常量不会被认为是错误的，仍存将其标识为*C*。

7）处理空格

```
[ \t\r]
```

匹配空格，同时对\r进行处理，防止系统不同带来的差异。

8）处理其他符号

在最后通过 . 对其余无法识别的字符进行匹配。

语法分析

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。

语义分析

运行方式

先使用自己的编译器（mc）对测试代码进行转换，将其从sysY转化为x86汇编，然后再使用gcc编译器将汇编代码转化为可执行文件。

```
(bash) ~/$ flex ./morphology.l
(bash) ~/$ yacc -d ./grammar.y
(bash) ~/$ g++ y.tab.c lex.yy.c -o mc -O2 -w
(bash) ~/$ ./mc ./test1.sy
(bash) ~/$ gcc assemble.s -o assemble
(bash) ~/$ ./assemble
```

实现细节

1）变量定义

- 对于常量的定义，先判断初始化的值是否为常数，如果是常数则直接计算得到，否则报错。对于普通变量的定义，几乎没有限制。
- 全局变量在程序开头进行定义，储存在栈的特殊位置，而局部变量，在运行的时候将其放到栈上即可。
- 当一个变量声明的时候，先查看同层的符号表中是否有相同的符号，如果有则报错，如果没有则将新定义的变量放入到符号表中。
- 符号表使用以下数据结构进行存储：

```
vector < map <string, var> *>
```

最外层的vector起到一个类似栈的作用，每进入一个作用域，就新new一个map加入到vector的尾部。当离开一个作用域时，就将vector的尾部删除。

var中储存了一个变量的有用的信息，包括该变量的类型（常量、数组、整形变量、函数等）、该变量的值（当且仅当为常量时，该值有效）、该变量在栈上的偏移量、该变量的维数（当且仅当该变量为数组）等，具体结构如下所示：

```
struct var {
    var() {}
    var(varType __type, int __val, int __offset) {
        type = __type;
        val = __val;
        offset = __offset;
    }
    var(varType __type, int __val, int __offset, vector
<int> __dim) {
        type = __type;
        val = __val;
        offset = __offset;
        dim = __dim;
    }
    var(varType __type, int __val) {
        type = __type;
        val = __val;
    }
    varType type;
    int val;
    int offset;
    vector <int> dim;
```

```
    int num_para;
};
```

使用变量和生命变量时，都在符号表中进行查询。

2）四则运算

- 处理四则运算时，先将两个操作数移动到寄存器上，然后再对两个寄存器进行计算，最后把计算结果存会到栈上。
- 对操作数的移动，需要注意这个操作数的类型（数组、常量或变量），对于不同的类型需要使用不同的移动方式。
- sysY的文法保证了乘法、除法和求余的优先级高于加法和减法。

3）控制流语句

- 对于if语句，在条件判断的时候处理出truelist和falselist，然后在if的分支中打标记，使用回填的方法和处理truelist和falselist中指令的跳转位置。

  由于汇编代码需要保证栈中内容的对齐，所以在进入分支之前需要匹配一个空字符串，其语义动作将栈中的内容对其，在离开分支的时候，需要将栈中的内容进行恢复，恢复到进入循环之前的状态。
- 对于while语句，由于其判断语句可能会执行多次，所以在进入判断语句之间就要对栈中的内容进行维护和对齐，退出时要恢复到原来的样子。
- 对于break和continue语句，需要要求其在while内才能使用。对于这两条语句，需要先对栈中内容进行恢复，才能进行跳转。否则最后程序结束时栈顶指针无法恢复到原来的状态。
- 进入while和if时，进入了一个新的scope，需要对符号表进行相应的处理。
- 对于while和if的语法进行了一定程序上的修改，修改之后如下所示：

```
IF BLL Cond BRL SetLabel Enter Stmt Exit SetLabel
IF BLL Cond BRL SetLabel Enter Stmt Exit SetLabel OutIF
SetLabel Enter Stmt
WHILE BLL SetWhileLabel Enter Cond SetOutLabel BRL SetLabel
Stmt SetLabel Exit
```

其中，SetLabel为设置一个标签，用于程序的跳转，Enter为进入时备份先前的状态，Exit为退出时恢复原来的状态。对于while语句，由于其比较特别，所以有细微区别，各个终结符的作用与if语句的大致相同。

这些新增加的非终结符都是匹配空字符串的。

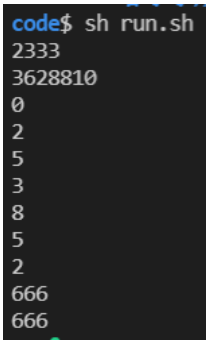
4）函数调用

- 对于无参的函数调用，直接使用call进行调用即可。
- 对于有参的函数调用，在LALR匹配的过程中，记录下每一个参数，然后将这些参数储存到栈上，使用call调用函数。进入新的函数之后，再从栈中将这些变量取出来，取到当前的段上。
- 进入一个函数前，需要对栈中的内容进行对其操作。

运行结果展示

1）test.sy

该程序测试了几乎所有的项目，其中第10行需要手动输入。



2）test1.sy

第一行输入n和m，表示两个多项式的长度，第二行和第三行分别输入两个多项式的系数。程序输出两个多项式相乘的结果。