编译原理第一次实验报告

组长：161220055 金亦凡

组员：161220033 刁含顺

一、实现功能

1. 词法分析

词法分析，使用词法分析工具 GNU Flex 实现，Flex 的主要任务是将输入文件中的字符流组织成为词法单元流，具体实现为编写 Flex 源代码，在 Flex 源代码文件的定义部分按照 name definition 格式定义了 int,float,id,空格等基本的词法单元，然后在规则部分，通过一些正则表达式和相应的响应函数返回了词法单元。对于在附录中没有定义的词法单元也即正则表达式无法匹配的字符流组织作为非法的字符打印出来，相应的函数如下：

```
printf("Error type A at Line %d: Mysterious characters
    \'%s\'\n",yylineno, yytext);
```

打印出错误类型 A(词法错误)以及错误出现的行号和非法的字符。

例如，如下

```
int main()
{
    int i=1;
    int j=~i;
}
```

因为正则表达式中并没有定义'~'，报错信息如下：

```
./parser ../Test/test3.cmm
Error type A at Line 4: Mysterious characters '~'
dhs@debian:~/compile/compilers-master/Code$
```

注：关于浮点数正则表达式的匹配，采取了附录中所述小数点前后必须有数字出现，没有规定浮点数的整数部分如同 int 类型一样，只能匹配 0 或者最高位不为 0 的正整数。

词法单元FLOAT表示的是所有（无符号）浮点型常数。一个浮点数由一串数字与一个小数点组成，小数点的前后必须有数字出现。例如，下面几个串都是浮点数：0.7、

2. 语法分析

在词法分析基础上实现语法分析，使用 Bison 工具实现。主要任务是按照附录 1 定义文法，按照附录 1 的表格设置了各个 token 的结合方式和优先级来排除移入规约冲突；同时用 LOWER_THAN_ELSE 解决了悬空 else 带来的移入规约问题。最后，在文法部分位置加入错误处理，利用 error 符号判断错误并打印行号和错误信息。

关于语法树的建立：对于词法和语法正确的文件，生成语法树并打印；对于语法错误的文件，也一并生成语法树以便判断接下来的更多错误，同时设置初始值为 true 的标志符 signal，在判断出任意类型的 error 后置为 false，打印语法树时根据 signal 判断是否打

印。语法树结构如下图：（语法树相关定义均位于自定义头文件 syntaxtree.h 中）

```
struct TreeNode {
        enum Type type;
        char unit[20];
        int lineno;
        float val;
        char name[20];
        struct TreeNode* child;
        struct TreeNode* next;
        struct TreeNode* parent;
};
```

定义了枚举类型 Type 用于判断 token 类型；二叉树结构为左节点是子树，右节点是同父节点的下一颗子树。打印语法树时从根节点往下遍历，一个一个打印。

语法树的自底向上形成如下图所示：

```
%%
/* High-level Definitions */
Program: ExtDefList { $$ = newNode(yylineno, 27, 1, $1); root = $$; }
;
ExtDefList: ExtDef ExtDefList { $$ = newNode(yylineno, 28, 2, $1, $2); }
| { $$ = newNode(yylineno, 28, 0); }
;
ExtDef: Specifier ExtDecList SEMI { $$ = newNode(yylineno, 29, 3, $1, $2, $3); }
| Specifier SEMI { $$ = newNode(yylineno, 29, 2, $1, $2); }
| Specifier FunDec CompSt { $$ = newNode(yylineno, 29, 3, $1, $2, $3); }
| Specifier error { $$ = newNode(yylineno, 29, 2, $1, $2); errorInfo(yytext); }
```

该图包含了所有三种可能的情况：

1. 当判断出一个正确的产生式时，执行后面{}中的函数。struct TreeNode* newNode()函数定义于 syntaxtree.h 中，用于建立语法树节点，各个参数分别为行号、token 类型（将转化为 enum Type）以及子节点的个数（可以为 0），根据子节点个数读取若干个语法树节点构成该节点的子树。由于是自底向上，这些参数表示的节点都已经定义完毕。在 lexical.l 中，词法单元对应的叶子结点也如此初始化。

2. 当判断出 error 时，也一并建立语法树，同时调用自定义的 errorInfo()函数报告错误信息。

（注：由于 yyerror(char* msg)函数总会默认自己调用一次，故将其屏蔽改为使用自己定义的报错函数。）

3. 在产生式 Program->ExtDefList 中，程序自底向上分析结束，将语法树根节点设置为 Program 对应的节点。

在定义 token 时，将所有 token 的类型改为了 struct TreeNode*。

运行如下两个程序：（对 test3.cmm 进行修改）

```
1 int main() {
2         float a = 1.2e-3;
3         a = a + 2;
4         return a;
5 }
```

```
1 int main() {
2         a[5,3] = 9;
3         if(a == 1) i = 0 else a = 2;
4         return;
5 }
```

第一个程序没有错误，打印语法树：

第二个程序有三个错误：数组括号中的逗号、else 之前未加;以及没有 return 对象（在 C—中文法规定 return 得返回一个 Exp），打印错误信息：

```
jyf@ubuntu:~/github/compilers/Code$ ./parser ../Test/test3.cmm
Error type B at Line 2: syntax error, before "]"
Error type B at Line 3: syntax error, before "else"
Error type B at Line 4: syntax error, before ";"
```

二、编译方式

(1)修改 Makefile 中的

test:

./parser ../Test/test1.cmm

部分，目前默认测试文件为 test1.cmm，可以将待测试代码放入这个文件中。

(2)make clean

(3)make

(4)make test

即可完成词法分析与语法分析，得出结果。