编译原理实验报告

Lab1 词法分析与语法分析

学号：151220069
姓名：罗义力
邮箱：luoyl233@foxmail.com

一、已完成功能

1. 词法分析：使用 GNU Flex 分析输入文件中的字符流，并返回词法单元。
2. 语法分析：使用 GNU Bison 对词法分析返回的词法单元进行处理并生成语法树。
3. 错误检测/提示：对源代码中出现的词法错误或者语法错误定位并输出提示信息。
4. 语法树的构建与输出：在语法分析过程中构建一棵语法树，在没有出现任何错误的情况下输出语法树的前序遍历结果。
5. 识别八进制数和十六进制数（要求 1.1）。
6. 识别指数形式的浮点数（要求 1.2）。
7. 识别“//” 和“/*…*/”形式的注释（要求 1.3）。

二、实验环境

GNU Linux Release: Ubuntu 12.04, kernel version 4.4.0-66
GCC version 5.4.0
GNU Flex version 2.6.0
GNU Bison version 3.0.4

三、使用说明

make: 编译生成分析器 parser。
make clean: 清除所有的中间文件和目标文件。
make test: 对 Makefile 文件中规定的样例测试，可一次传入多个参数从而统计多个文件。
注意：由于 Bison 遇到语法错误时会自动调用 yyerror(“syntax error”)，某些文件可能会输出 1 条错误语句，以前面的”#数字”代表第几个错误。

四、实验过程

1. 词法分析

词法分析使用 GNU Flex 工具将源代码中的字符流组织成为词法单元流，处理过程中遇到不属于定义范围内的符号会报错。

lexical.l 文件基本是跟着实验文档指导写的，需要注意的是，在完成规则部分时，必须注意把标识符的定义置于关键字之前，否则将无法匹配所有的关键字。另外一点是部分符号需要转义字符转义，也可以直接写在一对双引号里。

2. 语法分析

语法分析根据词法分析返回的词法单元匹配相应的语法结构，构建一棵语法树。Bison 采用的是自底向上的分析技术，避免了对左递归等问题的处理，但仍然需要对二义性和冲突进行处理。正确地定义符号的结合性，可以解决移入/归约冲突。

3. 错误检测/提示

词法错误检测比较简单，在词法分析的最后添加一条正则表达式如下：

```
168 .       {
169             printf("Error type A at line %d: Mysterious characters \'%s\'\n",
170             yylineno, yytext);
171             nError ++;
172         }
```

“.”表示匹配所有符号，由于正则表达式匹配是从前到后依次进行的，所有未定义的字符都会在最后被检测到。

语法错误需要在语法定义里指定 error 符号进行错误恢复。Bison 会在检测到语法错误时自动调用 yyerror()函数输出错误信息，但系统的 yyerrro()函数没有提供对错误行号的输出，因此，需要自己重新定义。这里正好可以利用 Flex 里面的 yylineno 变量确定错误发生的行号。

```
495 void yyerror(char *pstr, ...)
496 {
497     printf("Error type B at line %d: ", yylineno);
498     va_list varList;
499     va_start(varList, pstr);
500     vprintf(pstr, varList);
501     va_end(varList);
502     printf(".\n");
503 }
```

4. 语法树的构建与输出

另外定义了两个文件 TreeNode.h 和 TreeNode.c，分别定义了树节点结构体和操作函数。

```
22 typedef struct TreeNode {
23     int lineno;          //line no
24     ValueType nType;            //to explain which type value is,interger or float or id
25     union {
26         int iValue;
27         float fValue;
28         char *ptr;
29     };
30     char *info;          //other information
31     int nChild;          //childs no
32     struct TreeNode *childs[8]; //childs pointers
33 } TreeNode;
```

```
35 struct TreeNode *newNode();
36 struct TreeNode *createNode(char *pstr, int lineno);
37 void addChild(struct TreeNode *parent, struct TreeNode *child);
38 void printTree(struct TreeNode *root);
```

在语法分析的时候根据语法结构，调用 addChild()函数添加子节点生成语法树，其中根节点 root 在 Program 的语法里面定义。处理完整个文件之后，main()函数里负责调用 printTree()函数，以前序遍历的方式按照文档要求打印整棵语法树。

5. 识别八进制数和十六进制数

如果把 8、10、16 进制数的匹配全写在同一个正则表达式里，会增加识别和处理的难度。八进制数和十六进制数的识别和转换不同于十进制数，因此可以另外写两个正则表达式来分别匹配并转换成十进制数（使用函数 strtol()）。

```
43 0[0-7]+      {
44             yylval.pNode = createNode("INT", yylineno);
45             yylval.pNode->nType = Int;
46             yylval.pNode->iValue = strtol(yytext, NULL, 8);
47             return INT;}
```

```
38 0[xX][A-Fa-f0-9]+    {
39             yylval.pNode = createNode("INT", yylineno);
40             yylval.pNode->nType = Int;
41             yylval.pNode->iValue = strtol(yytext, NULL, 16);
42             return INT;}
```

对于 09 和 0x3G 这种错误的 8 进制和 16 进制数，在规则的末尾添加如下两个正则表达

式，Flex 总是优先选择最先匹配的正则表达式，不会影响到正确的 8 进制和 16 进制数。

```
0[xX][0-9a-zA-Z]+   {
            printf("Error type A at line %d: Illegal hexadecimal number \'%s\'\n",
            yylineno, yytext);
            nError ++;  return INT;}
0[0-9a-zA_Z]+       {
            printf("Error type A at line %d: Illegal octal number \'%s\'\n",
            yylineno, yytext);
            nError ++; return INT;}
```

6. 识别指数形式的浮点数

难点在于省略整数部分的的浮点数，如.5E03，与省略小数部分的浮点数，如 01.E12，这两种浮点数的正则表达式很难写在一起，因为两部分都可以省略，但又不能同时省略。按照讲义上浮点数的写法会匹配错误的数 09（本应该被判断为错误的八进制数）。最后，我自己写了两个规则如下：

```
[+-]?{digits}\.({digits})?([Ee][+-]?{digits})?  {
        yylval.pNode = createNode("FLOAT", yylineno);
        yylval.pNode->nType = Float;
        yylval.pNode->fValue = atof(yytext);
        return FLOAT;}
[+-]?\.{digits}([Ee][+-]?{digits})? {
        yylval.pNode = createNode("FLOAT", yylineno);
        yylval.pNode->nType = Float;
        yylval.pNode->fValue = atof(yytext);
        return FLOAT;}
```

7. 识别“//” 和“/*…*/”形式的注释

“//“类型的注释只需要用 input()函数一直读入字符，直到出现换行符”\n”。“/*…*/”类型的注释需要检测到”*/”时才停止抛弃字符串。由于“/*…*/”注释是不支持嵌套的，所以“/*…*/”里面的注释会被忽略。需要注意的是对文件结束符 EOF 的判断。

```
"//"    {char c = input(); while (c != EOF && c != '\n') c = input();}
"/*"    {   char c = input();
            while (c != EOF) {
                if(c == '*') {
                    c = input();
                    if(c == '/')
                        break;
                }
                else
                    c = input();
            }
        }
```

五、总结反思

1.测试了一些样例，发现了许多问题，比如全局变量无法在定义时初始化，函数内的变量必须在前面先集中定义，无法随用随定义等等，这些应该是为了简化语法结构。由此可见，为了让程序员能够随心所欲地写代码，编译器需要做许多额外的工作。

2.yyerror()会被自动调用，无论如何也无法取消掉，最后无奈加上数字代替表示错误号。