实验一　词法分析和语法分析

131220128 杨帆

一、实验环境

操作系统：　GNU Linux Release: Ubuntu 12.04
软件版本：　GNU Flex version 2.5.35；　GNU Bison version 2.5

二、实验内容

编写一个程序对使用C语言书写的源代码进行词法分析和语法分析。若有错误则输出相关信息，否则输出语法树。实验要求使用词法分析工具GNU Flex和语法分析工具GNU Bison，并使用C语言来完成。

三、实验过程

整体思路：

由于编译器的工作特性是先将源程序进行词法分析，然后将识别出的词法单元送入语法分析器，再根据文法分析句子。故必须先完成词法分析的任务。将提取出的token作为参数逐个传入语法分析程序，根据定义的文法规则自底向上进行分析匹配。

为了最后打印语法分析树，必须对终结符、非终结符做建树操作，将每条产生式的左部作父节点，将右部每个符号做子节点（作为兄弟节点用链表连接），打印时对树进行前序遍历即可

完成的功能点：

必做功能
　　识别词法错误、识别语法错误
选做功能
　　识别八制与十六数、识别指数形式的浮点数(如1.05e-4)、
　　识别“//”与“/*…*/”形式的注释

程序结构与编译过程：

main.c 程序入口。打开文件，启动yystart()分析程序，若无错误则打印语法树。
lexical.l 词法分析程序。定义了词法单元的正则表达式和识别出词法单元后的动作；将终结符号的词法单元建立树节点。
syntax.y 语法分析程序。定义终结符号的优先级，定义合法的文法，建立语法分析树，实现了打印树的过程，打印错误信息。
lex.yy.c 由flex编译lexical.l得到 进行词法分析，被syntax.y调用。
syntax.tab.c　syntax.tab.h 由bison编译syntax.y得到，进行语法分析。

```
bison –d syntax.y　　flex lexical.l
gcc main.c syntax.tab.c –lfl –ly –o parser
```

主要功能及其实现方法：

1. 识别 8 进制、16 进制数和无符号浮点数。正则定义如下：

```
INT [0]|[1-9][0-9]*
INT8 [0][1-7][0-7]*
INT16 0[xX][a-fA-F0-9]+
FLOAT1 ([0-9]*(\.[0-9]+)?)([eE][+-]?[0-9]*)
FLOAT2 ([0-9]+\.[0-9]+)
```

在 flex 程序的响应函数中，三种整数类型匹配到后都返回到语法分析程序中的 INT 类型。而 FLOAT1、FLOAT2(区分普通浮点数和科学计数法)返回到 FLOAT 类型。

2. 实现识别两种风格的注释

利用 flex 库函数 input(),可以从当前的输入文件中读入一个字符.讲义中已将识别“//”注释行的写 1 法给出。匹配“/*…*/”时 while 循环直到文件末尾，匹配到一对“/*”和“*/”时结束。

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

3. 打印更详细错误信息

使用 bison 提供的宏 YYERROR_VERBOSE 可以在错误恢复后打印错误信息时先出更详细的内容。例如“unexpected …”“expected …”.
亦可在 bison 文件定义部分加上%error-verbose.

4. 在 flex 程序中返回给 bison 程序的内部参数是 yylval，表示当前词法单元所对应的属性值，它的默认属性是 int 类型。而传入 bison 的数据类型种类很多。我们在定义部分加入 union 类型，定义为 yylval 的类型。

```
%union{struct Node *node;}
```

5. 语法树的建立与打印

考虑一个节点：每个节点对应待分析程序中的一个符号。节点应该有自己的属性值、属性类型、子节点指针、兄弟节点指针等属性，由于打印时需要打印非终结符号的行号，故还需要保存一个节点的行号。

```
struct Node{
    int terminal;
    int line;
    char text[30];
    char type[30];
    struct Node *arity;
    struct Node *nextsibling;
};
```

对于终结符来说，它们是整个语法树的叶子节点，故没有子节点。现阶段的终结符节点的建立都在 flex 程序中识别 token 后的动作中完成。

对于非终结符，在 bison 程序中规约到一个非终结符后，执行动作 addNode 函数，作用是将这个非终结符插入语法树中，并定义这个节点的每个属性。
这里 addNode 函数，由于每条文法的参数不固定，我们使用 c 语言提供的库函数 stdarg 中的 va_list()方法实现不固定多参数的传入，从而实现多叉树的建立。

```
struct Node *addNode(char* type, int arg_num, ...)
```

参数 type 代表需定义节点的类型、arg_num 代表传入参数的个数

```
p->terminal = 0;
arity = va_arg(vl, struct Node*);
p->line = arity->line;
p->arity = arity;
for (i = 1; i < arg_num; i++)
{
    arity->nextsibling  = va_arg(vl, struct Node*);
    if (arity->nextsibling != NULL)
        {
            arity = arity->nextsibling;
        }
}
```

定义了节点 p 是否为终结符、第一子节点指针、子节点的下一个兄弟节点指针

打印树只需要从深度为 0 的节点，若遇到的节点是非终结符号节点，打印属性值后继续向下做深度优先遍历。遇到终结符时只需打印其属性即可。由于保存在节点中的属性值是 char*类型（flex 内部变量 yytext 是 char*类型），按实验要求，我们需要将识别出的 10 进制、8 进制、16 进制以及浮点数进行字符串到数值类型的转化。此时，常用的 atoi()与 atof()功能就不够了。通过查阅 C 手册，使用以下两个转化函数可以实现其功能：

```
long int strtol(const char *nptr,char **endptr,int base);
float strtof(const char *nptr, char **endptr);
```

6. 错误恢复

错误恢复的原理在这里就不再赘述了。通过思考和不断尝试，编写测试样例，在消除 shift/reduce conflicts 的前提下，在 bison 定义产生式的部分加入 error 语句。并重写 yyerror（）函数，在分析程序时遇到文法无法匹配的句子时调用它，并打印错误。设置全局变量 err，一旦发现一处语句文法无法匹配，就取消打印语法树的过程。

四、实验中的一些注意点

1. 在 flex 程序中 %%rules%% 部分中，定义的顺序决定了分析程序识别 token 属性的优先级。所以 ID 的定义必须放在 TYPE(int|float)、if、else 等正则表达式的后面，否则 if、else 这样的终结符都会被识别成 ID。

bison 程序中对于操作符的优先级、左右结合规则参考了实验手册的附录，同时也加入了对悬空 else 问题、负号和减号的二义性问题的处理。

2. char*类型的变量无法直接做比较和赋值操作，需要用到 string.h 库函数：

```
int strcmp ( const char * str1, const char * str2 );
char * strcpy ( char * destination, const char * source );
```

3. 由于本实验是在 linux 环境下进行的实验，由于编码不同的问题，就无法正确识别其他系统下的文件内容，例如 windows 下的换行符‘\r’在 linux 系统下为‘\n’。故在词法分析 DELIM 的定义中，加入了对‘\r’的识别。

五、实验反思与收获

通过本次实验，我对词法分析以及自底向上的语法分析有了更深层次的理解；提高了对 c 语言下的工程性代码的编写能力；也为后面的编译器语义分析等部分打下了基础。错误恢复部分仍有不理解和不全面的地方，有待调整和修改。