编译原理实验一：词法分析与语法分析

郭松 2015301500205

November 16, 2017

1 系统环境

Ubuntu 17.10 (Kernel 4.13.0-16) GCC 7.2.0 Flex 2.6.1 Bison (GNU Bison 3.0.4)

2 功能简介

对于有一定错误的程序，也尝试生成抽象语法树。基本完成了必做样例和选做样例的全部内容。

构建抽象语法树，并给出每一个节点对应的源代码(的范围)。

在附件中准备了很多.c文件，是我自己用来测试功能的，涵盖了大部分语法点。

另外，Makefile文件是非常简单的Make脚本，直接调用make即可编译。如果使用make debug编译，可以生成含调试输出的程序，能输出更详细的错误信息。

当程序遇到了词法错误，会产生类似于下面的输出：

```
line 4: <<Error Type A.2>> Nya! I cannot recognize ``~'', wtf!
line 5: <<Error Type A.1>> Nya! I can recognize ``1e'', but why?
line 6: <<Error Type A.0>> Nya? ``0x1G'' might be a wrong hex integer.
line 7: <<Error Type A.0>> Nya? ``09'' might be a wrong oct integer.
```

当程序遇到了语法错误，会产生类似于下面的输出：

```
line 8: syntax error
line 8: <<Error Type B.1>> Meow! Valid argument expression required.
line 9: syntax error
line 9: <<Error Type B.1>> Meow! Valid expression required.
line 10: syntax error
line 10: <<Error Type B.0>> Meow? ``;'' is expected
```

和要求的输出格式有一些不同，我的输出格式为：

```
[  1:  1]->[ 21:  1]Program
[  1:  1]->[ 21:  1]  ExtDefList
[  1:  1]->[  4:  1]    ExtDef
[  1:  1]->[  1:  5]      Specifier
[  1:  1]->[  1:  5]        TYPE: float
[  1:  7]->[  1: 20]      FunDec
[  1:  7]->[  1: 11]        ID: sqr_f
[  1: 12]->[  1: 12]        (
[  1: 13]->[  1: 19]        VarList
[  1: 13]->[  1: 19]          ParamDec
[  1: 13]->[  1: 17]            Specifier
(...)
```

每一行的开头表示了语法树节点对应的源代码中的位置。同时对于部分终结符也只显示其文本（如各种括号，一些运算符等）

3 功能测试

调用make编译完成之后，会产生一个叫”ejq_cc”的程序，即最终的可执行文件。
我在附件的test文件夹中准备了诸多用于测试的样例。
ac.c是一个简单的程序，包含了基本的语法元素
ac_numbers.c是一个仅包含数值字面量的程序，展示了对于数字的词法分析
normal.c和dinic.c是两个由实际环境下的程序修改得到的程序，展示了在一般情况下的表现基本覆盖了所有语言点。
error*.c是一些简短的，包含了常见错误的C语言程序。
相应的*.output或者*.error是我本机测试时，程序的输出。

4 词法分析

4.1 十进制整数

十进制整数不含有前导零，即如果这个数非零，那么它首位不为0,否则其为0,根据这个定义，可以写出:

```
([1-9][0-9]*)|0
```

其前半部分表示正数，后半部分表示0,对于负数的表示，我们将其表示为符号后跟一个正数，这也是大部分C编译器的实现。

4.2 八进制整数

八进制整数以0开头，不包含有大于7的数码，可以有前导零，因而可以写为

```
OCTINT 0[0-7]+
ERROCT 0[0-9]+
```

这里，Erroct表示了猜测为错误的八进制整数的情况。

4.3 十六进制整数

十六进制整数以0x开头，包含0-9和a-f，不区分大小写，因而可以写为

```
HEXINT 0[xX][0-9a-fA-F]+
ERRHEX 0[xX][0-9a-zA-Z_]+
```

这里Errhex表示了猜测为错误的十六进制整数的情况。

4.4 十进制浮点数

十进制浮点数有两种表示方法，其一为普通的表示的方法，其二为科学计数法。对于普通的表示方法，小数点前后至少一部分不为空，则可以分情况考虑为

```
{INT}.[0-9]* | .[0-9]+
```

对于第二种表示方法，可以认为其是简单地在后加上了指数部分，那么指数部分可以表示为

```
[eE][+-]?{INT}
```

同时也要注意到前半部分也可以是一个整数。因此综合上述两种情况可以表示为:

```
(({INT}(\.{DIGIT}*)?|\.{DIGIT}+)([eE][+-]?{DIGIT}+)|({INT}?\.{DIGIT}+)|({INT}\.{DIGIT}*))
```

4.5 行末注释的表示

行末注释可以表示为”//”.*”\n”，Flex使用的正则表达式的”.”不包含换行符，十分方便。

4.6 块注释的表示

因为注释不组成任何一个语法符号，所以考虑使用词法分析完成块注释的隔离。Flex提供了“状态”这一概念，可以给自动机加入指定的状态，因此，在读入”/*”之后，我们进入comment状态，直到读到第一个”*/”为止，返回INITAL状态。这样的处理方式天然解决了嵌套注释的问题。

对于其余的正则表达式，十分简单，不再赘述。

4.7 错误处理

按照词法定义，不包含在词法定义中的单词都被认为是错误的单词，在正则表达式中用

```
ERRWORD ([a-zA-Z0-9_]+)
%%
{ERRWORD} {
    char buf[1024];
    sprintf(buf, "<<Error Type A.1>> Nya! I can recognize ``%s'', but why?", yytext);
    yyerror(buf);
    sprintf(buf, "ERRWORD: %s", yytext);
    yylval = nnewnode(buf, lineno, charno, lineno, charno + strlen(yytext));
    charno += strlen(yytext);
    return ID;
}
```

来表示。因为yyerror的签名不能格式化，所以需要用sprintf来格式化错误信息。在这一种情况下，我将其认定为一个ID

5 文法分析

对于基本的文法分析，照着要求的附件翻译一下即可。对于错误处理，也只需稍加修改，在可能的地方加上error标记，然后处理即可，这里只介绍构建语法树的方法。

定义语法树结构体:

```
typedef struct node{
    char *desc;
    int soncnt;
    int start_lineno, start_pos, end_lineno, end_pos;
    struct node** son;
}node;
```

desc起到了描述的功能，在日后进行修改的时候，如要添加访问标志符表等需求，只需稍加修改如添加item项，即可实现。 start_lineno、start_pos、end_lineno和end_pos是为了描述语法树节点对应的源代码的位置。