

编译原理实验一：词法分析与语法分析

郭松 2015301500205

November 23, 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

基本完成了必做样例和选做样例的全部内容。可以构建抽象语法树，并给出每一个节点对应的源代码(的范围)。如果机房快要关门了或者急着去约会，那么看完第四节后就可以去看代码了，如果编译不了，就直接看我的样例输出，第二面中间往后的内容是为了文档完整写的，没啥新东西，跟别人的都一样。

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：

```
1 error type A on line 4: Nya! ``~'': wtf!
2 error type A on line 5: Nya! ``1e'': is this an ID?
3 error type A on line 6: Nya? ``0x1G'' might be a wrong hex integer.
4 error type A on line 7: 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：

```
1 error type B on line 9: syntax error, unexpected ID, expecting RP
2 error type B on line 10: syntax error, unexpected RP
3 error type B on line 11: syntax error, unexpected DIV
```

和要求的输出格式有一些不同，我的输出格式为：

```
1 [ 1: 1]->[ 21: 1]Program
2 [ 1: 1]->[ 21: 1] ExtDefList
3 [ 1: 1]->[ 4: 1] ExtDef
4 [ 1: 1]->[ 1: 5] Specifier
5 [ 1: 1]->[ 1: 5] TYPE: float
6 [ 1: 7]->[ 1: 20] FunDec
7 [ 1: 7]->[ 1: 11] ID: sqr_f
8 [ 1: 12]->[ 1: 12] (
9 [ 1: 13]->[ 1: 19] VarList
10 [ 1: 13]->[ 1: 19] ParamDec
11 [ 1: 13]->[ 1: 17] Specifier
12 (...)
```

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 特点

原来的数组定义我觉得挺反人类的，不利于后面的类型计算，重新定义了变量定义的文法：

$$\begin{array}{lcl} VarDec & \rightarrow & ID \\ & | & ID\ VarDimList \\ VarDimList & \rightarrow & [\text{int}] \\ & | & [\text{int}]\ VarDimList \end{array}$$

4 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。

5 词法分析

5.1 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 ([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。

5.2 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

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

这里，Erroct 表示了猜测为错误的八进制整数的情况。

5.3 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

```
1 HEXINT 0[xX][0-9a-fA-F]+
2 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。

5.4 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

```
1 {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

```
1 [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：

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

5.5 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。

5.6 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 INITIAL 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。

5.7 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

```
1 ERRWORD ([a-zA-Z0-9_]+)
2 %%
3 {ERRWORD} {
4     char buf[1024];
5     sprintf(buf, "<<Error Type A.1>> Nya! I can recognize ``%s'', but why?", yytext);
6     perror(buf);
7     sprintf(buf, "ERRWORD: %s", yytext);
8     yylval = nnewnode(buf, lineno, charno, lineno, charno + strlen(yytext));
9     charno += strlen(yytext);
10    return ID;
11 }
```

来表示。用 sprintf 来格式化错误信息，同时定义了 perror 用于输出 A 类错误。在这一种情况下，我将其认定为一个 ID

6 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：

```
1 typedef struct node{
2     char *desc;
3     int soncnt;
4     int start_lineno, start_pos, end_lineno, end_pos;
5     struct node** son;
6 }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。