编译实验一实验报告

——词法分析器 LEX

王曦　10300240014
陈济凡　10300240076
承沐南　10300240003
田应涛　10302010029

2013-04-07

1.LEX & YACC

1.1 LEX & YACC 概述

Lex 是 LEXical compiler 的缩写，是 Unix 环境下非常著名的工具,主要功能是生成一个词法分析器(scanner)的 C 源码,描述规则采用正则表达式(regular expression)。描述词法分析器的文件*.l，经过 lex 编译后，生成一个 lex.yy.c 的文件，然后由 C 编译器编译生成一个词法分析器。词法分析器，简单来说，其任务就是将输入的各种符号，转化成相应的标识符(token)，转化后的标识符 很容易被后续阶段处理。其过程如图所示：

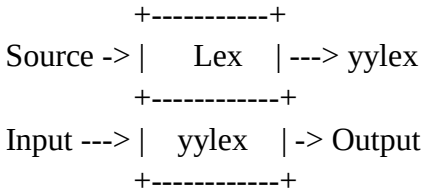

Figure 1 An overview of Lex

YACC(Yet Another Compiler Compiler)，是一个经典的生成语法分析器的工具。它是 Unix/Linux 上一个用来生成编译器的编译器（编译器代码生成器）。YACC 生成的编译器主要是用 C 语言写成的语法解析器（Parser），需要与词法解析器 Lex 一起使用，再把两部份产生出来的 C 程序一并编译，并生成相应的语言分析过程。YACC 和 LEX 结合使用的过程如下图所示：

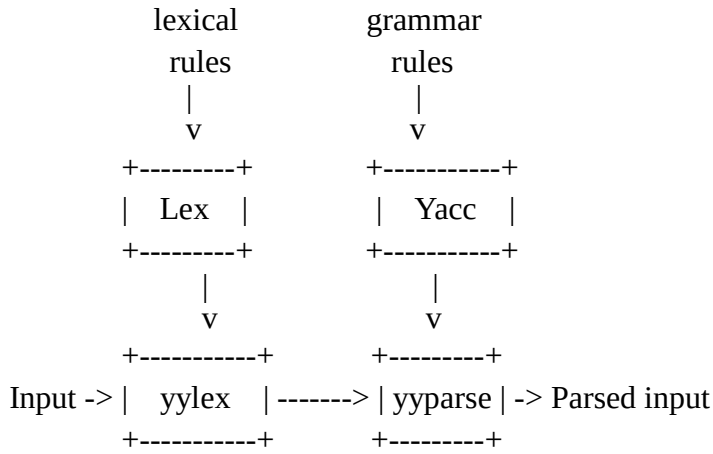

Figure 2 Lex with Yacc

1.2 LEX 文件结构

LEX 文件结构分为三个部分，分别为<定义部分><规则部分><用户子程序部分>。其中规则部分是必须的，定义和用户子程序部分是任选的。

Definition section
%%
Rules section
%%

C code section

(1) 定义部分

定义部分起始于 %{ 符号，终止于 %} 符号，其间可以是包括 include 语句、声明语句在内的 C 语句。这部分跟普通 C 程序开头没什么区别。

(2) 规则部分

规则部分起始于"%%"符号，终止于"%%"符号，其间则是词法规则。词法规则由模式和动作两部分组成。模式部分可以由任意的正则表达式组成，动作部分是由 C 语言语句组成，这些语句用来对所匹配的模式进行相应处理。需要注意的是，lex 将识别出来的单词存放在 yytext[]字符数据中，长度为 yyleng。因此该数组的内容就代表了所识别出来的单词的内容。动作部分如果有多行执行语句，也可以用{}括起来。

(3) 用户子程序部分

最后一个%%后面的内容是用户子程序部分，可以包含用 C 语言编写的子程序，而这些子程序可以用在前面的动作中，这样就可以达到简化编程的目的。

1.3 正则表达式

规则部分是 Lex 描述文件中最为复杂的一部分，主要由正则表达式描述。下面列出一些模式部分的正则表达式字：

字符	含义
A-Z, 0-9, a-z	构成了部分模式的字符和数字。
.	匹配任意字符，除了 \n。
-	用来指定范围。例如：A-Z 指从 A 到 Z 之间的所有字符。
[ ]	一个字符集合。匹配括号内的 *任意* 字符。如果第一个字符是 **^** 那么它表示否定模式。例如: [abC] 匹配 a, b, 和 C 中的任何一个。
*****	匹配 *0 个*或者多个上述的模式。
+	匹配 *1 个*或者多个上述模式。
?	匹配 *0 个或 1 个*上述模式。
$	作为模式的最后一个字符匹配一行的结尾。
{ }	指出一个模式可能出现的次数。 例如: A{1,3} 表示 A 可能出现 1 次或 3 次。
****	用来转义元字符。同样用来覆盖字符在此表中定义的特殊意义，只取字符的本意。
^	否定。

|	表达式间的逻辑或。
"<一些符号>"	字符的字面含义。元字符具有。
/	向前匹配。如果在匹配的模版中的“/”后跟有后续表达式，只匹配模版中“/”前 面的部分。如：如果输入 A01，那么在模版 A0/1 中的 A0 是匹配的。
( )	将一系列常规表达式分组。

1.4 歧义规则

Lex 可以处理有歧义的规定即当多于一个表达式可以匹配当前输入的时候，Lex 按如下来选择:

1. 首选最长匹配。
2. 在匹配相同数目字符的规则中，首选最先给出的规则。

这就要求在给出匹配规则的时候注意它们的先后顺序。并且，首选最长匹配（贪婪匹配）的原理使包含 .* 表达式的规则是危险的。例如 \“[.\n]*\”好象是识别字符串的好方式。但是它诱使程序更加超前的读取，查找最远的双引号。假设输入是

“first” string here, “second” here

上述表达式将匹配

“first” quoted string here, “second”

这显然是一个错误的结果。

匹配字符串应该选用\″[ˆ\″\n]∗\″这个正则表达式。

2.实验实现

2.1 Tokens

本实验要求识别的标示符（Tokens）如下：

```
*     ----- Literals -----------
*
*    IDENTIFIER    for identifiers
*    INTEGERT      for integers
*    REALT     for reals
*    STRINGTfor strings
*
*     ----- Keywords -----------
*
*    PROGRAM, IS, BEGIN, END, VAR, TYPE, PROCEDURE, ARRAY, RECORD,
*    IN, OUT, READ, WRITE, IF, THEN, ELSE, ELSIF, WHILE, DO, LOOP,
*    FOR, EXIT, RETURN, TO, BY, AND, OR, NOT, OF, DIV, MOD
*
*     ----- Operators ----------
```

```
*
*    LPAREN   (
*    RPAREN   )
*    LBRACKET    [
*    RBRACKET    ]
*    LBRACE   {
*    RBRACE   }
*    COLON    :
*    DOT      .
*    SEMICOLON   ;
*    COMMA    ,
*    ASSIGN   :=
*    PLUS     +
*    MINUS    -
*    STAR     *
*    SLASH    /
*    BACKSLASH   \
*    EQ       =
*    NEQ      <>
*    LT       <
*    LE       <=
*    GT       >
*    GE       >=
*    LABRACKET   [<
*    RABRACKET   >]
*
*     ----- Misc ----------
*
*    EOFF      for End Of File
*    ERROR     when lexer error
```

2.2 处理方式

1） 对于标识符，处理方式如下：

[a-zA-Z_][a-zA-Z_0-9]*

2） 对于整数，处理方式如下：

-?[0-9]+

其中需要注意处理溢出的情况，当正整数大于2147483647或者负整数小于-2147483648时，要进行报错并返回错误标识。

3） 对于实数，处理方式如下：

-?([1-9e]\d*\.[0-9e]*|0\.[0-9e]*|0?\.[0-9e]*)

其中需要处理溢出的情况，当正实数大于 3.4e38 或者负整数小于-3.4e38 时，需要报错并返回错误标识。

在实际匹配中还加入了以下两条匹配规则：

-?([1-9]\d*\.[0-9]*|0\.[0-9]*|0?\.[0-9]*)

-?([0-9]+\.)

前一条是为了在处理溢出情况时候的方便，后一条是为了在检测到如 1.这样的情况时，输出 1.00000。

4） 对于字符串，处理方式为：

\"[^\"\n]*\"

并且当字符串长度大于 257 时，需要进行报错并返回错误标识，同样若在字符串中检测到了非法字符同样应该报错并返回。对未结尾的字符串需要用\"[^\"\n]*进行处理。

5） 对于关键字，处理方式为：

关键字的检测非常简单，只需照关键字原样进行检测，若能匹配则返回。如：

"PROGRAM" return PROGRAM;

6） 对于运算符，处理方式为：

运算符的检测同关键字如："(" return LPAREN;

7） 对于注释，处理方式为：

```
"(*" {
  BEGIN(COMMENT);
}
<COMMENT>"*)" {
  BEGIN(INITIAL);
}
<COMMENT><<EOF>> {
  printf("%d:%d Unclosed comment.\n",lineno,columnno-yyleng);
  return EOFF;
}
<COMMENT>. ;
<COMMENT> “\n” {lineno++; columnno=1;}
```

这里涉及到了 LEX 中状态的转移，Lex 提供了一种机制来有条件的启动一个规则。任何规则，如果其模式有前置的"<sc>"，那么都只有在扫描器在起始状态(start condition)sc 时才会启动。状态用尖括号声明，调用 begin(sc)可以进入该状态，调用 begin(INITIAL)可以回到初始状态。在处理中需要注意到当有未完结的注释时应该报错并返回错误标识。

8） 对于制表符，处理方式为：

```
<*>[\t] { columnno += 8 - columnno%8 + 1; }
```

9） 对于换行符，处理方式为：

```
<*>"\n" {
      lineno++;
      columnno=1;
        }
```

10) 对于空格，处理方式为：

```
[ ]+ ;
```

以上规则包括了所有情况的处理

注:上述处理中并为包括输出和错误处理的代码,具体代码见提交中的 PCAT.LEX 文件。

3. 运行结果

下图分别为测试文件 test20.pcat 和测试文件 test02.pcat 的测试结果，图中行号列号均有标识，将扫描到的内容分别输出，不同内容间用不同颜色加以区分。

```
chen@ubuntu:~/pcat-compiler-c-version$ ./lexer tests/test20.pcat
Ln:1 Col:29  Comment: (* an out of range integer *)
 Ln:1 Col:408    Integer out of range.
 Ln:1 Col:408    ERROR
 Ln:3 Col:51  Comment: (* an very large real -- still acceptable though *)
 Ln:4 Col:1    real: 12345678901234567890123456789O.000000
 Ln:4 Col:32    integer: 0
 Ln:6 Col:41  Comment: (* an invalid string with tab in in it *)
 Ln:6 Col:822   Invalid char.
 Ln:6 Col:822    string: "abcde g"
 Ln:8 Col:27  Comment: (* an overly long string *)
 Ln:9 Col:1   String too long.
 Ln:9 Col:1    ERROR
 Ln:11 Col:18  Comment: (* an ok string *)
 Ln:12 Col:1    string: "12345678901234567890123456789012345678901234567890123456
789012345678901234567890123456789012345678901234567890123456789012345678901234
567890123456789012345678901234567890123456789012345678901234567890123456789012
345678901234567890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345"
 Ln:14 Col:23  Comment: (* another ok string *)
 Ln:15 Col:1    string: ""
 Ln:17 Col:31  Comment: (* an overly long identifier *)
 Ln:18 Col:1   Identifier too long.
 Ln:18 Col:1    ERROR
 Ln:20 Col:22  Comment: (* an ok identifier *)
 Ln:21 Col:1    identifier: x12345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890123456789012345678
901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234
 Ln:23 Col:16  Comment: (* an ok line *)
 Ln:24 Col:1    identifier: if
 Ln:24 Col:4    identifier: x
 Ln:24 Col:5    Operators: <
 Ln:24 Col:6    integer: 7
 Ln:24 Col:7    identifier: then
 Ln:24 Col:12    identifier: y
 Ln:24 Col:13    Operators: :=
 Ln:24 Col:15    identifier: y
 Ln:24 Col:16    Operators: +
 Ln:24 Col:17    integer: 17
 Ln:26 Col:30  Comment: (* should lex as two tokens *)
 Ln:27 Col:1    real: 123.000000
 Ln:27 Col:5    identifier: x
 Ln:29 Col:28  Comment: (* a bad character (bell) *)
 Ln:30 Col:1   UnrecogChar.
 Ln:32 Col:1    identifier: xx
 Ln:33 Col:9  Comment: (* an ok comment with an embedded        tab
and cr *)
 Ln:33 Col:350    identifier: yy
```

```
 chen@ubuntu:~/pcat-compiler-c-version$ ./lexer tests/test02.pcat
Ln:1 Col:20  Comment: (* Testing reals. *)
 Ln:3 Col:1    KeyWords: PROGRAM
 Ln:3 Col:9    KeyWords: IS
 Ln:4 Col:5    KeyWords: VAR
 Ln:4 Col:9    identifier: A
 Ln:4 Col:10    Operators: ,
 Ln:4 Col:12    identifier: B
 Ln:4 Col:14    Operators: :
 Ln:4 Col:16    identifier: REAL
 Ln:4 Col:21    Operators: :=
 Ln:4 Col:24    real: 0.0
 Ln:4 Col:27    Operators: ;
 Ln:5 Col:5    identifier: C
 Ln:5 Col:7    Operators: :
 Ln:5 Col:9    identifier: REAL
 Ln:5 Col:14    Operators: :=
 Ln:5 Col:17    real: 0.0
 Ln:5 Col:20    Operators: ;
 Ln:6 Col:1    KeyWords: BEGIN
 Ln:7 Col:5    KeyWords: WRITE
 Ln:7 Col:11    Operators: (
 Ln:7 Col:12    string: "ENTER TWO REALS:"
 Ln:7 Col:30    Operators: )
 Ln:7 Col:31    Operators: ;
 Ln:8 Col:5    KeyWords: READ
 Ln:8 Col:10    Operators: (
 Ln:8 Col:11    identifier: A
 Ln:8 Col:12    Operators: ,
 Ln:8 Col:14    identifier: B
 Ln:8 Col:15    Operators: )
 Ln:8 Col:16    Operators: ;
 Ln:9 Col:5    identifier: C
 Ln:9 Col:7    Operators: :=
 Ln:9 Col:10    real: 8.0
 Ln:9 Col:13    Operators: ;
 Ln:10 Col:5    KeyWords: WRITE
 Ln:10 Col:11    Operators: (
 Ln:10 Col:12    string: "A="
 Ln:10 Col:16    Operators: ,
 Ln:10 Col:18    identifier: A
 Ln:10 Col:19    Operators: ,
 Ln:10 Col:21    string: ", B="
 Ln:10 Col:27    Operators: ,
 Ln:10 Col:29    identifier: B
 Ln:10 Col:30    Operators: ,
 Ln:10 Col:32    string: ", C="
 Ln:10 Col:38    Operators: ,
```