

熟悉 Yacc 的使用

一、实验目的：

熟悉语法分析器生成工具 Yacc 的使用，并学会在 cygwin 下使用 bison 工具编译 Yacc 文法说明文件。学习如何使用 lex 和 yacc 合作进行语法分析。

二、实验内容：

根据给出的 calculator 例子 (calculator0, calculator1, calculator2, calculator3) 完成下面题目：用 lex 和 yacc 写一个计算布尔表达式真值的计算器。

三、实验要求：

1. 输入为一个布尔表达式，以换行结束。输出为这个布尔表达式的真值 (true 或 false)。
2. 必须用二义文法实现。布尔表达式二义文法为： $S \rightarrow S \text{ or } S \mid S \text{ and } S \mid \text{not } S \mid (S) \mid \text{true} \mid \text{false}$ ，其中优先级 $\text{or} < \text{and} < \text{not}$ ，or 和 and 左结合，not 右结合。
3. 用非二义文法实现作为选作内容，非二义文法请参照表达式非二义文法自己写出来。
4. 在 cygwin 下用 flex, bison 和 gcc 工具将实验调试通过，并写出测试例测试正确性。

四、具体实现

Yacc 文件

1. 定义段部分：

```
1 %{\n2 #include <ctype.h>\n3 #include <stdio.h>\n4 int yylex();\n5 int yyerror();\n6 #define YYSTYPE double\n7 %}\n8\n9 %token LPAREN RPAREN ENTER\n10 %token NUMBER\n11 %left GT LT EQ\n12 %left ADD SUB\n13 %left MUL DIV\n14 %left OR\n15 %left AND\n16 %right NOT
```

定义段分两部分：

- 1、以 C 语法写的一些定义和声明：例如，文件包含，宏定义，全局变量定义，函数声明等。
- 2、对文法的终结符和非终结符做一些相关声明。

定义和声明

- int yylex() 是词法分析程序，它返回记号。语法分析驱动程序 yyparse() 将会调用 yylex() 获取记号。如果不使用 lex 生成这个函数，则必须在辅助函数段用 C 语言写这个程序。
- yyerror() 函数用于输出错误信息。
- Yacc 将属性值栈的栈内元素的类型定义为 YYSTYPE。yylval 的类型与属性值栈元素的类型相同，即，默认状态下，yylval 为 int 类型，在此使用 #define YYSTYPE double 将属性值栈元素定义为 double 类型，则 yylval 就是 double 类型。
- 输入的是整数，之所以用 double 类型是因为作除法后可能会产生浮点数。

文法的终结符号和非终结符

- `%token` 定义文法中使用了哪些终结符，根据题目的文法，在此有左右括号，回车符，数字。
- 优先级: `%left` 和 `%right` 定义文法中使用的终结符，`left` 和 `right` 分别代表他们的结合性，有大于、小于、等于三个比较运算符，加、减、乘、除四则运算符，与、或、非三种逻辑运算符。除了非运算，其他都是左结合，非运算是单目运算符，是右结合性。
- 结合性: 由定义出现的顺序决定的，先定义的优先级低，最后定义的优先级最高，同时定义的优先级相同。所以 `n` 非运算优先级最高，逻辑运算 > 乘除 > 加减 > 比较运算，符合要求。

2. 规则段:

```

20 prog      : prog exprp
21           | exprp
22           ;
23 exprp     : expr ENTER {
24           if($1) printf("true\n");
25           else printf("false\n");
26           ;
27 expr      : expr LT expr {if($1 < $3)$2 = 1;else $2 = 0;}
28           | expr GT expr {if($1 > $3)$2 = 1;else $2 = 0;}
29           | expr EQ expr {if($1 == $3)$2 = 1;else $2 = 0;}
30           | expr ADD expr {$2 = $1 + $3; }
31           | expr SUB expr {$2 = $1 - $3; }
32           | expr MUL expr {$2 = $1 * $3; }
33           | expr DIV expr {$2 = $1 / $3; }
34           | expr AND expr {$2 = $1 && $3;}
35           | expr OR expr  {$2 = $1 || $3;}
36           | NOT expr %prec NOT{$2 = ! $2;}
37           | LPAREN expr RPAREN {$2 = $2;}
38           | NUMBER {$2 = $1;}
39           ;
40 %%

```

- 直接使用题目给的文法，具有二义性，通过对优先级和结合性消除二义。
- 紧接着文法的时候语义动作，引用存放在属性值栈中的文法符号的属性值，模拟移进-规约过程。
- 第二条文法是为了识别输入是否结束，其语义动作是判断输出输出式的布尔值。
- 用 `%prec NOT` 强制定义了其优先级与结合性跟 `NOT` 相同，使得非运算优先级最高。

3. 辅助函数段:

```

42 int main()
43 {
44     yyparse();
45     return 0;
46 }

```

- `yyparse()` 是语法分析驱动程序，它会调用 `yylex()` 获取记号。

Lex 文件

```

1  %{
2  #include "cal.tab.h"
3  int yywrap(void){ return 1; }
4  %}
5
6  delim      [ \t]
7  ws         {delim}+
8  digit      [0-9]
9  number     {digit}+
10
11  %%
12  {number}   {sscanf(yytext, "%lf", &yylval); return NUMBER;}
13  not        {return NOT;}
14  "||"       {return OR;}
15  "&&"       {return AND;}
16  "+"        {return ADD;}
17  "-"        {return SUB;}
18  "*"        {return MUL;}
19  "/"        {return DIV;}
20  ">"        {return GT;}
21  "<"        {return LT;}
22  "=="       {return EQ;}
23  "("        {return LPAREN;}
24  ")"        {return RPAREN;}
25  {ws}       {;}
26  "\n"       {return ENTER;}
27
28  %%

```

- 用 yacc 编译器对 cal.y 文件进行编译，编译时带上参数-d，此时编译器除生成 cal.tab.c 以外，还将生成名为 cal.tab.h 的头文件。该头文件中包含 cal.y 中定义的所有终结符的常量定义，属性值栈的类型定义，以及变量 yylval 的外部引用定义。用 Lex 写的词法分析规则文件为 cal.l，则在 cal.l 的声明部分应包含头文件 cal.tab.h，即，在 cal.l 声明部分应包含如下语句：#include "cal.tab.h"。并且，cal.l 文件中凡涉及返回记号名的部分，都返回 cal.y 中定义的终结符名；而用 yylval 返回记号属性值。
- 其他的和之前的 Lex 文件是一样的。
- Sscanf 函数的作用是把 yytext，也就是记录当前词法单元的属性值，以给出格式赋给 yylval 变量，因为在移进规约是需要对表达式计算和比较，数字才可以比较，而且这也会成为属性值栈里的元素，属性栈定义的是 double 型，所以以 lf 格式输入。
- 测试程序是否正确时，非不用感叹号表示，而是用 not 表示。

Makefile 文件

1. 在 cygwin 下用 LEX 定义词法分析器并把它和 YACC 写的语法分析器链接起来的命令相对较多，使用 bison 和 flex 联合写一个语法分析器时，需要的编译步骤稍显复杂，用 makefile 可以将这个复杂的编译步骤简化。
2. Makefile 告诉我们如何对一个包含若干源文件的工程进行编译，比如，先编译什么，后编译什么，怎样链接等。Makefile 文件直接使用例子给出的 makefile 文件作出部分修改：

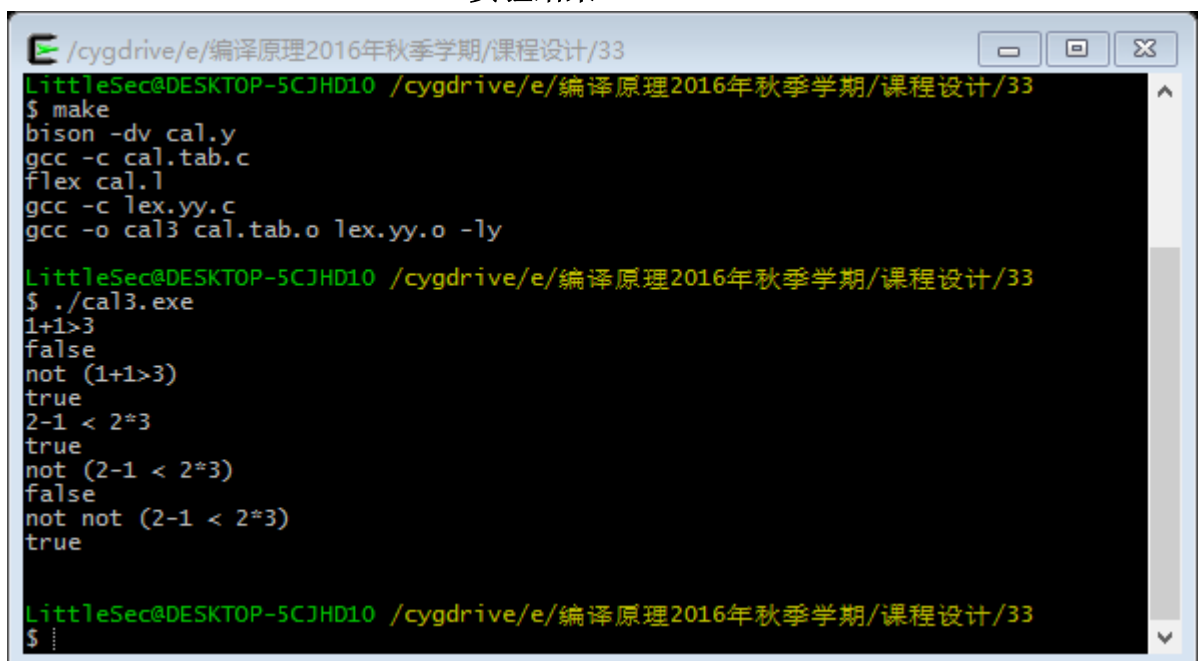
```

1  cal3: cal.tab.o lex.yy.o
2      gcc -o cal3 cal.tab.o lex.yy.o -ly
3
4  lex.yy.o: lex.yy.c cal.tab.h
5      gcc -c lex.yy.c
6
7  cal.tab.o: cal.tab.c
8      gcc -c cal.tab.c
9
10 lex.yy.c: cal.l
11     flex cal.l
12
13 cal.tab.c: cal.y
14     bison -dv cal.y
15
16 cal.tab.h: cal.y
17     echo "cal.tab.h was created at the same time as cal.tab.c."
18
19 clean:
20     rm -f cal3.exe lex.yy.o cal.tab.o lex.yy.c cal.tab.c cal.tab.h cal3.exe.stackdump cal.output

```

3. 编译时，在 cygwin 下进入文件路径，直接输入 make 即可正确编译。

实验结果



```

/cygdrive/e/编译原理2016年秋季学期/课程设计/33
LittleSec@DESKTOP-5CJHD10 /cygdrive/e/编译原理2016年秋季学期/课程设计/33
$ make
bison -dv cal.y
gcc -c cal.tab.c
flex cal.l
gcc -c lex.yy.c
gcc -o cal3 cal.tab.o lex.yy.o -ly

LittleSec@DESKTOP-5CJHD10 /cygdrive/e/编译原理2016年秋季学期/课程设计/33
$ ./cal3.exe
1+1>3
false
not (1+1>3)
true
2-1 < 2*3
true
not (2-1 < 2*3)
false
not not (2-1 < 2*3)
true

LittleSec@DESKTOP-5CJHD10 /cygdrive/e/编译原理2016年秋季学期/课程设计/33
$ !

```

- 输入 make 会自动执行 makefile 文件中的指令，对各个文件进行编译连接。
- 输入表达式，回车后会输出表达式的布尔值。
- 经过多组测试，结果正确。

五、心得与体会

1. 了解和熟悉语法分析器生成工具 Yacc 的使用，并学会在 cygwin 下使用 bison 工具编译 Yacc 文法说明文件。学习了使用 Lex 和 Yacc 合作进行语法分析还有 makefile 文件的使用。
2. Yacc 和 Lex 的链接其实就是通过编译 Yacc 源程序得到的.tab.h 头文件连接的，在 Lex 源程序中包含这个 Yacc 生成的头文件即可。
3. 没有使用非二义文法而是通过优先级和结合性去解决移进-规约冲突，实际上程序还有少许不完

美的地方，例如输入串：“not 2+3>1”，应该输出时 false，但结果是 true，因为 not 的优先级定义为最高，因此实际上是先计算 not 2，在 C 语言里，这个表达式是合法的，结果是 0（大于 0 的整数为真），因此 0+3>1 是 true，结果和我们想不一样，在实验中的文法中应该加括号改变优先级。

4. 可以改写成如下非二义文法，这样在声明符号时就可以不需要考虑结合性和优先级。而且对于上述所述的输入串，留意到 expr3 的文法，not 是对 expr 有效，而不是对 expr3 有效，这样就不回出现上述错误的结果。而对于二义文法是无法改变这一点的。

```

expr: expr LT expr {if($1 < $3)$$ = 1;else $$ = 0;}
    | expr GT expr {if($1 > $3)$$ = 1;else $$ = 0;}
    | expr EQ expr {if($1 == $3)$$ = 1;else $$ = 0;}
    | expr ADD term {$$ = $1 + $3;}
    | expr SUB term {$$ = $1 - $3;}
    ;

term: term MUL factor {$$ = $1 * $3;}
    | term DIV factor {$$ = $1 / $3;}
    | factor
    ;

factor : LPAREN expr RPAREN {$$ = $2;}
       | NUMBER {$$ = $1;}
       | expr1 {$$ = $1;}
       ;

expr1 : expr1 OR expr2 {$$ = $1 || $3;}
       | expr2
       ;

expr2 : expr2 AND expr3 {$$ = $1 && $3;}
       | expr3
       ;

expr3 : | NOT expr {$$ = ! $2;}
       ;

```

5. 无二义文法和带优先级的二义文法的效果大致相同。两者比较来看，带优先级的二义文法的书写更加简便一些，只需为相应的运算符指定优先级就可以了。而改造后的无二义的文法结构更加清晰，容易理解。