Lex 和 Yacc 简明教程

作者 :Thomas Niemann
翻译： 傅惠忠

目　录

序言

本书将教会你如何使用 lex 和 yacc 构造一个编译器。lex 和 yacc 是两个用来生成词汇分析器和剖析器的工具。我假设你能够运用 C 语言编程，并且理解数据结构的含义，例如“链表”和“树”。

导言部分描写了构建编译器所需的基本部分，以及 lex 和 yacc 之间的互动关系。后面两章更加详细的描写了 lex 和 yacc。以此为背景，我们构建了一个经典的计算器程序。这个计算器支持常用的算术符号和控制结构，例如实现了像 if-else 和 while 这样的控制结构。经过小小的修改，我们就把这个计算器转换成一个可以运行在基于栈的计算机上的编译器。后面的间节讨论了在编写编译器是经常发生的问题。本书中使用的例程的源代码可以从下面列出的网站上下载到。

THOMAS NIEMANN

波特兰，俄勒冈州

网站：epaperpress.com

译者序 :

找不到好的中文资料，所以自己翻译了一个，如发现错误，请不吝赐教。

电子邮件：fuhuizn@hotmail.com

傅惠忠

导言

在 1975 年之前，编写编译器一直是一个非常费时间的工作。这一年 Lesk[1975] 和 Johson [1975] 发表了关于 lex 和 yacc 的论文。这些工具极大地简化了编写编译器的工作。在 Aho[1986] 中描写了关于如何具体实现 lex 和 yacc 的细节。从下列地方可以等到 lex 和 yacc 工具：

- Mortice Kern System (MKS)，在 www.mks.com，
- GNU flex 和 bison，在 www.gnu.org，
- Ming，在 www.mingw.org，
- Cygwin，在 www.cygwin.org，还有
- 我的 Lex 和 Yacc 版本，在 epaperpress.com

MKS 的版本是一个高质量的商业产品，零售价大约是 300 美元。GNU 软件是免费的。flex 的输出数据也可以被商业版本所用，对应的商业版本号是 1.24，bison 也一样。Ming 和 Cygwin 是 GNU 软件在某些 32 位 Windows 系统上的移植产物。事实上 Cygwin 是 UNIX 操作系统在 Windows 上的一个模拟接口，其中包括了 gcc 和 g++ 编译器。

我的版本是基于 Ming 的，但量是使用 Visual C++ 编译的并且包含一个改善文件操作程序的小补丁。如果你下载我的版本，请记住解压缩的时候一定要保留文件夹结构。

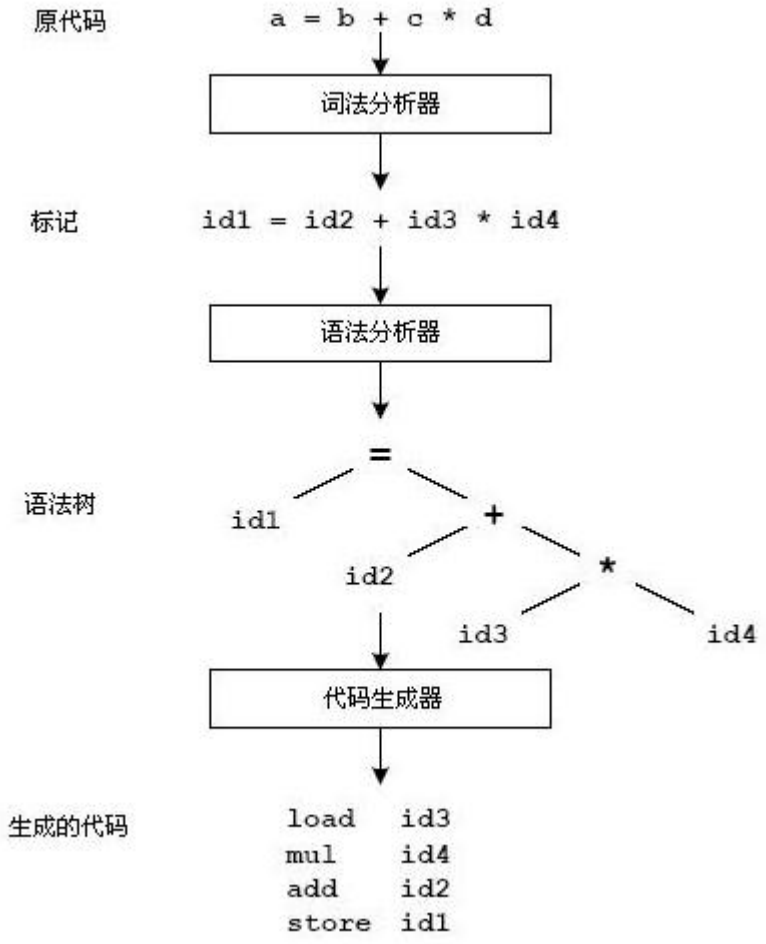


图 1：编译顺序

Lex 为词法分析器或扫描器生成 C 程序代码。它使正则表达式匹配输入的字符串并且把它们转

换成对应的标记。标记通常是代表字符串或简单过程的数值。图 1 说明了这一点。

当 Lex 发现了输入流中的特定标记，就会把它们输入一个特定的符号表中。这个符号表也会包含其它的信息，例如数据类型（整数或实数）和变量在内存中的位置。所有标记的实例都代表符号表中的一个适当的索引值。

Yacc 为语法分析器或剖析器生成 C 程序代码。Yacc 使用特定的语法规则以便解释从 Lex 得到的标记并且生成一棵语法树。语法树把各种标记当作分级结构。例如，操作符的优先级和相互关系在语法树中是很明显的。下一步，生成编译器原代码，对语法树进行一次第一深度历遍以便生成原代码。有一些编译器直接生成机器码，更多的，例如上图所示，输出汇编程序。

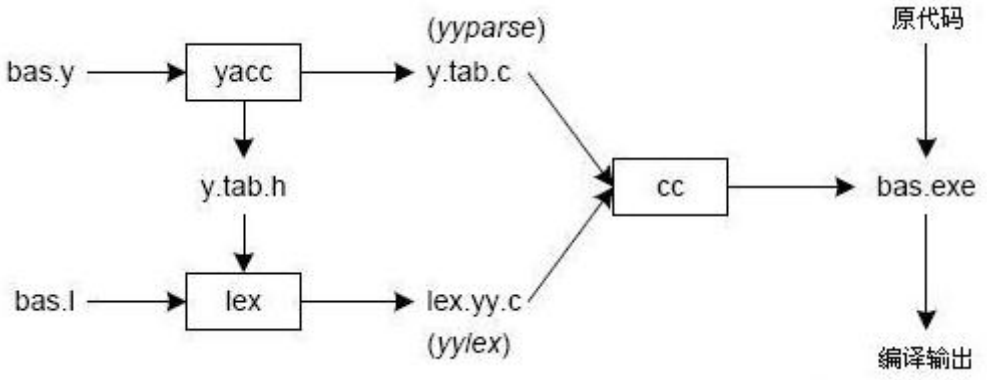


图 2：用 Lex/Yacc 构建一个编译器

图 2 显示了 Lex 和 Yacc 使用的命名约定。我们首先要说明我们的目标是编写一个 BASIC 编译器。首先我们要指定 Lex 的所有的模式匹配规则（bas.l）和 Yacc 的全部语法规则（bas.y）。下面列举了产生我们的编译器，bas.exe，的命令。

```
yacc -d bas.y                  # 生成 y.tab.h, y.tab.c
lex bas.l                      # 生成 lex.yy.c
cc lex.yy.c y.tab.c -o bas.exe     # 编译／连接
```

Yacc 读入 bas.y 中的语法描述而后生成一个剖析器，即 y.tab.c 中的函数 yyparse。bas.y 中包含的是一系列的标记声明。“-d”选项使 Yacc 生成标记声明并且把它们保存在 y.tab.c 中。Lex 读入 bas.l 中的正则表达式的说明，包含文件 y.tab.h，然后生成词汇解释器，即文件 lex.yy.c 中的函数 yylex。

最终，这个解释器和剖析器被连接到一起，而组成一个可执行程序，bas.exe。我们从 main 函数中调用 yyparse 来运行这个编译器。函数 yyparse 自动调用 yylex 以便获取每一个标志。

Lex

理论

在第一阶段，编译器读入源代码然后把字符串转换成对应的标记。使用正则表达式，我们可以为 Lex 设计特定的表达式以便从输入代码中扫描和匹配字符串。在 Lex 上每一个字符串对应一个动作。通常一个动作返回一个代表被匹配的字符串的标记给后面的剖析器使用。作为学习的开始，在此我们不返回标记而是简单的打印被匹配的字符串。我们要使用下面这个正则表达式扫描标识符。

letter(letter|digit)*

这个问题表达式所匹配的字符串以一个简单字符开头，后面跟随着零个或更多个字符或数字。这个例子很好的显示了正则表达式中所允许的操作。

重复，用“*”表示

交替，用“|”表示

串联

每一个正则表达式代表一个有限状态自动机 (FSA)。我们可以用状态和状态之间的转换来代表一个 FSA。其中包括一个开始状态以及一个或多个结束状态或接受状态。

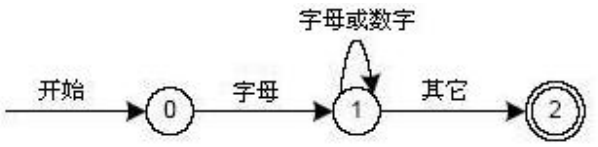


图 3：有限状态自动机

在图 3 中状态 0 是开始状态，而状态 2 是接受状态。当读入字符时，我们就进行状态转换。当读入第一个字母时，程序转换到状态 1。如果后面读入的也是字母或数字，程序就继续保持在状态 1。如果读入的字符不是字母或数字，程序就转换到状态 2，即接受状态。每一个 FSA 都表现为一个计算机程序。例如，我们这个 3 状态机器是比较容易实现的：

```
start: goto state0
state0: read c
        if  c = letter goto state1
    goto state0
state1: read c
    if  c = letter goto state1
    if  c = digit goto state1
    goto state2
state2: accept string
```

这就是 lex 所使用的技术。lex 把正则表达式翻译成模拟 FSA 的一个计算机程序。通过搜索计算机生成的状态表，很容易使用下一个输入字符和当前状态来判定下一个状态。

现在我们应该可以容易理解 lex 的一些使用限制了。例如 lex 不能用于识别像括号这样的外壳结构。识别外壳结构需要使用一个混合堆栈。当我们遇到“（”时，就把它压入栈中。当遇到“）”

时，我们就在栈的顶部匹配它，并且弹出栈。然而，lex 只有状态和状态转换能力。由于它没有堆栈，它不适合用于剖析外壳结构。yacc 给 FSA 增加了一个堆栈，并且能够轻易处理像括号这样的结构。重要的是对特定工作要使用合适的工具。lex 善长于模式匹配。yacc 适用于更具挑战性的工作。

练习

表意字符	匹配字符
.	除换行外的所有字符
\n	换行
*	0 次或无限次重复前面的表达式
+	1 次或更多次重复前面的表达式
?	0 次或 1 次出现前面的表达式
^	行的开始
$	行的结尾
a|b	a 或者 b
(ab)+	1 次或玩多次重复 ab
"a+b"	字符串 a+b 本身（C 中的特殊字符仍然有效）
[]	字符类

表 1：简单模式匹配

表达式	匹配字符
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	a, b, c 中的一个
[a-z]	从 a 到 z 中的任意字符
[a\-z]	a, -, z 中的一个
[-az]	-, a, z 中的一个
[A-Za-z0-9]+	一个或更多个字母或数字
[\t\n]+	空白区
[^ab]	除 a,b 外的任意字符
[a^b]	a, ^, b 中的一个
[a|b]	a, |, b 中的一个
a|b	a, b 中的一个

表 2：模式匹配举例

lex 中的正则表达式由表意字符（表 1）组成。表 2 中列举了模式匹配的例子。在方括号（[]）中，通常的操作失去了本来含意。方括号中只保留两个操作，连字号（“-”）和抑扬号（“^”）。当把连字号用于两个字符中间时，表示字符的范围。当把抑扬号用在开始位置时，表示对后面的表达式取反。如果两个范式匹配相同的字符串，就会使用匹配长度最长的范式。如果两者匹配的长度相同，就会选用第一个列出的范式。

```
... 定义 ...
%%
... 规则 ...
%%
... 子程序 ...
```

lex 的输入文件分成三个段，段间用 %% 来分隔。上例很好的表明了这个意思。第一个例子是最短的可用 lex 文件：

```
%%
```

输入字符将被一个字符一个字符直接输出。由于必须存在一个规则段，第一个 %% 总是要求存在的。然而，如果我们不指定任何规则，默认动作就是匹配任意字符然后直接输出到输出文件。默认的输入文件和输出文件分别是 stdin 和 stdout。下面是效果完全相同的例子，显式表达了默认代码：

```
%%
 /* 匹配除换行外的任意字符 */
. ECHO;
 /* 匹配换行符 */
\n ECHO;
%%
int yywrap(void) {
    return 1;
}
int main(void) {
    yylex();
    return 0;
}
```

上面规则段中指定了两个范式。每一个范式必须从第一列开始。紧跟后面的必须是空白区（空格，TAB 或换行），以及对应的任意动作。动作可以是单行的 C 代码，也可以是括在花括号中的多行 C 代码。任何不是从第一列开始的字符串都会被逐字拷贝进所生成的 C 文件中。我们可以利用这个特殊行为在我们的 lex 文件中增加注释。在上例中有“.”和“\n”两个范式，对应的动作都是 ECHO。lex 预先字义了一些宏和变量。ECHO 就是一个用于直接输出范式所匹配的字符的宏。这也是对任何未匹配字符的默认动作。通常 ECHO 是这样定义的：

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

变量 yytext 是指向所匹配的字符串的指针（以 NULL 结尾），而 yyleng 是这个字符串的长度。变量

yyout 是输出文件，默认状态下是 stdout。当 lex 读完输入文件之后就会调用函数 yywrap。如果返回 1 表示程序的工作已经完成了，否则返回 0。每一个 C 程序都要求一个 main 函数。在本例中我们只是简单地调用 yylex，lex 扫描器的入口。有些 lex 实现的库中包含了 main 和 yywrap。这就是为什么我们的第一个例子，最短的 lex 程序，能够正确运行。

名称	功能
int yylex(void)	调用扫描器，返回标记
char *yytext	指针，指向所匹配的字符串
yyleng	所匹配的字符串的长度
yylval	与标记相对应的值
int yywrap(void)	约束，如果返回 1 表示扫描完成后程序就结束了，否则返回 0
FILE *yyout	输出文件
FILE *yyin	输入文件
INITIAL	初始化开始环境
BEGIN	按条件转换开始环境
ECHO	输出所匹配的字符串

表 3：lex 中预定义变量

下面是一个根本什么都不干的程序。所有输入字符都被匹配了，但是所有范式都没有定义对应的动作，所以没有任何输出。

```
%%
.
\n
```

下面的例子在文件的每一行前面插入行号。有些 lex 实现预先定义和计算了 yylineno 变量。输入文件是 yyin，默认指向 stdin。

```
%{
    int yylineno;
%}
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

定义段由替代式，C 代码，和开始状态构成。定义段中的 C 代码被简单地原样复制到生成的 C 文件的顶部，而且必须用 %{ 和 %} 括起来。替代式简化了正则表达式规则。例如，我们可以定义数字

和字母：

```
digit [0-9]
letter [A-Za-z]
%{
    int count;
%}
%%
/* match identifier */
{letter}({letter}|{digit})* count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

范式和对应的表达式必须用空白区分隔开。在规则段中，替代式要用花括号括起来（如 {letter}）以便和其字面意思区分开来。每当匹配到规则段中的一个范式，与之相对应的 C 代码就会被运行。

下面是一个扫描器，用于计算一个文件中的字符数，单词数和行数（类似 Unix 中的 wc 程序）：

```
%{
int nchar, nword, nline;
%}
%%
\n { nline++; nchar++; }
[^ \t\n]+ { nword++, nchar += yyleng; }
. { nchar++; }
%%
int main(void) {
yylex();
printf("%d\t%d\t%d\n", nchar, nword, nline);
return 0;
}
```

YACC

理论

yacc 的文法由一个使用 BNF 文法（Backus-Naur form）的变量描述。BNF 文法规则最初由 John Backus 和 Peter Naur 发明，并且用于描述 Algol60 语言。BNF 能够用于表达上下文无关语言。现代程序语言中的大多数结构可以用 BNF 文法来表达。例如，数值相乘和相加的文法是：

E -> E + E

E -> E * E

E -> id

上面举了三个例子，代表三条规则（依次为 r1，r2，r3）。像 E（表达式）这样出现在左边的结构叫非终结符（nonterminal）。像 id（标识符）这样的结构叫终结符（terminal，由 lex 返回的标记），它们只出现在右边。这段文法表示，一个表达式可以是两个表达式的和、乘积，或者是一个标识符。我们可以用这种文法来构造下面的表达式：

E -> E * E (r2)

-> E * z (r3)

-> E + E * z (r1)

-> E + y * z (r3)

-> x + y * z (r3)

每一步我们都扩展了一个语法结构，用对应的右式替换了左式。右面的数字表示应用了哪条规则。为了剖析一个表达式，我们实际上需要进行倒序操作。不是从一个简单的非终结符开始，根据语法生成一个表达式，而是把一个表达式逐步简化成一个非终结符。这叫做“自底向上”或者“移进 - 归约”分析法，这需要一个堆栈来保存信息。下面就是用相反的顺序细述了和上例相同的语法：

1 . x + y * z 移进

2 x . + y * z 归约 (r3)

3 E . + y * z 移进

4 E + . y * z 移进

5 E + y . * z 归约 (r3)

6 E + E . * z 移进

7 E + E * . z 移进

8 E + E * z . 归约 (r3)

9 E + E * E . 归约 (r2) 进行乘法运算

10 E + E . 归约 (r1) 进行加法运算

11 E . 接受

点左面的结构在堆栈中，而点右面的是剩余的输入信息。我们以把标记移入堆栈开始。当堆栈顶部和右式要求的记号匹配时，我们就用左式取代所匹配的标记。概念上，匹配右式的标记被弹出堆栈，而左式被压入堆栈。我们把所匹配的标记认为是一个句柄，而我们所做的就是把句柄向左式归约。这个过程一直持续到把所有输入都压入堆栈中，而最终堆栈中只剩下最初的非终结符。在第 1

步中我们把 x 压入堆栈中。第 2 步对堆栈应用规则 r3，把 x 转换成 E。然后继续压入和归约，直到堆栈中只剩下一个单独的非终结符，开始符号。在第 9 步中，我们应用规则 r2，执行乘法指令。同样，在第 10 步中执行加法指令。这种情况下，乘法就比加法拥有了更高的优先级。

考虑一下，如果我们在第 6 步时不是继续压入，而是马上应用规则 r1 进行归约。这将导致加法比乘法拥有更高的优先级。这叫做“移进 - 归约”冲突（shift-reduce conflict）。我们的语法模糊不清，对一个表达式可以引用一条以上的适用规则。在这种情况下，操作符优先级就可以起作用了。

举另一个例子，可以想像在这样的规则中

E -> E + E

是模糊不清的，因为我们既可以从左面又可以人右面递归。为了挽救这个危机，我们可以重写语法规则，或者给 yacc 提供指示以明确操作符的优先顺序。后面的方法比较简单，我们将在练习段中进行示范。

下面的语法存在“归约 - 归约”冲突 (reduce-reduce conflict)。当堆栈中存在 id 是，我们既可以归约为 T，也可以归约为 E。

E -> T

E -> id

T -> id

当存在冲突时，yacc 将执行默认动作。当存在“移进 - 归约”冲突时，yacc 将进行移进。当存在“归约 - 归约”冲突时，yacc 将执行列出的第一条规则。对于任何冲突，它都会显示警告信息。只有通过书写明确的语法规则，才能消灭警告信息。后面的章节中我们将会介绍一些消除模糊性的方法。

练习，第一部分

... 定义 ...

%%

... 规则 ...

%%

... 子程序 ...

yacc 的输入文件分成三段。“定义”段由一组标记声明和括在“%{”和“%}”之间的 C 代码组成。BNF 语法定义放在“规则”段中，而用户子程序添加在“子程序”段中。

构造一个小型的加减法计算器可以最好的说明这个意思。我们要以检验 lex 和 yacc 之间的联系开始我们的学习。下面是 yacc 输入文件的定义段：

```
%token INTEGER
```

上面的定义声明了一个 INTEGER 标记。当我们运行 yacc 时，它会在 y.tab.c 中生成一个剖析器，同时会产生一个包含文件 y.tab.h：

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
```

```
#define INTEGER 258
extern YYSTYPE yylval;
```

lex 文件要包含这个头文件，并且使用其中对标记值的定义。为了获得标记，yacc 会调用 yylex。yylex 的返回值类型是整型，可以用于返回标记。而在变量 yylval 中保存着与返回的标记相对应的值。例如，

```
[0-9]+    {    yylval = atoi(yytext);
                return INTEGER;    }
```

将把整数的值保存在 yylval 中，同时向 yacc 返回标记 INTEGER。yylval 的类型由 YYSTYPE 决定。由于它的默认类型是整型，所以在这个例子中程序运行正常。0-255 之间的标记值约定为字符值。例如，如果你有这样一条规则

```
[-+]        return *yytext;                /* 返回操作符 */
```

减号和加号的字符值将会被返回。注意我们必须把减号放在第一位心避免出现范围指定错误。由于 lex 还保留了像“文件结束”和“错误过程”这样的标记值，生成的标记值通常从 258 左右开始。下面是为我们的计算器设计的完整的 lex 输入文件：

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}
%%
[0-9]+    {
                yylval = atoi(yytext);
                return INTEGER;
          }
[-+\n]    return *yytext;
[ \t]     ;       /* skip whitespace */
.         yyerror("invalid character");
%%
int yywrap(void) {
    return 1;
}
```

yacc 在内部维护着两个堆栈；一个分析栈和一个内容栈。分析栈中保存着终结符和非终结符，并且代表当前剖析状态。内容栈是一个 YYSTYPE 元素的数组，对于分析栈中的每一个元素都保存着一个对应的值。例如，当 yylex 返回一个 INTEGER 标记时，yacc 把这个标记移入分析栈。同时，相应的 yylval 值将会被移入内容栈中。分析栈和内容栈的内容总是同步的，因此从栈中找到对应于一个标记的值是很容易实现的。下面是为我的计算器设计的 yacc 输入文件：

```
%{
```

```
int yylex(void);
void yyerror(char *);
%}
%token INTEGER
%%
program:
        program expr '\n'                { printf("%d\n", $2); }
       |
       ;
expr:
        INTEGER                          { $$ = $1; }
       | expr '+' expr                   { $$ = $1 + $3; }
       | expr '-' expr                   { $$ = $1 - $3; }
       ;
%%
void yyerror(char *s) {
      fprintf(stderr, "%s\n", s);
      return 0;
}
int main(void) {
     yyparse();
     return 0;
}
```

规则段的方法类似前面讨论过的 BNF 文法。规则第一条叫 command 规则。其中的左式，或都称为非终结符，从最左而开始，后面紧跟着一个自己的克隆。后面跟着的是右式。与规则相应的动作写在后面的花括号中。

通过利用左递归，我们已经指定一个程序由 0 个或更多个表达式构成。每一个表达式由换行结束。当探测到换行符时，程序就会打印出表达式的结果。当程序应用下面这个规则时

```
expr: expr '+' expr                  { $$ = $1 + $3; }
```

在分析栈中我们其实用左式替代了右式。在本例中，我们弹出“expr '+' expr”然后压入“expr”。我们通过弹出三个成员，压入一个成员缩小的堆栈。在我们的 C 代码中可以用通过相对地址访问内容栈中的值，“$1”代表右式中的第一个成员，“$2”代表第二个，后面的以此类推。“$$”表示缩小后的堆栈的顶部。在上面的动作中，把对应两个表达式的值相加，弹出内容栈中的三个成员，然后把造得到的和压入堆栈中。这样，分析栈和内容栈中的内容依然是同步的。

当我们把 INTEGER 归约到 expr 时，数字值开始被输入内容栈中。当 NTEGER 被移分析栈中之后，我们会就应用这条规则

```
expr: INTEGER                        { $$ = $1; }
```

INTEGER 标记被弹出分析栈，然后压入一个 expr。对于内容栈，我们弹出整数值，然后又把它压回去。也可以说，我们什么都没做。事实上，这就是默认动作，不需要专门指定。当遇到换行符时，与 expr 相对应的值就会被打印出来。当遇到语法错误时，yacc 会调用用户提供的 yyerror 函数。如果你需要修改对 yyerror 的调用界面，改变 yacc 包含的外壳文件以适应你的需求。你的 yacc 文件中的最后的函数是 main ... 万一你奇怪它在哪里的话。这个例子仍旧有二义性的语法。yacc 会显示“移进 - 归约”警告，但是依然能够用默认的移进操作处理语法。

练习，第二部分

在本段中我们要扩展前一段中的计算器以便加入一些新功能。新特性包括算术操作乘法和除法。圆括号可以用于改变操作的优先顺序，并且可以在外部定义单字符变量的值。下面举例说明了输入量和计算器的输出：

```
user: 3 * (4 + 5)
calc: 27
user: x = 3 * (4 + 5)
user: y = 5
user: x
calc: 27
user: y
calc: 5
user: x + 2*y
calc: 37
```

词汇解释器将返回**VARIABLE**和**INTEGER**标志。对于变量，yylval 指定一个到我们的符号表**sym**中的索引。对于这个程序，**sym**仅仅保存对应变量的值。当返回**INTEGER**标志时，yylval 保存扫描到的数值。这里是 lex 输入文件：

```
%{
    #include <stdlib.h>
    void yyerror(char *);
    #include "y.tab.h"
%}
%%
    /* variables */
[a-z]   {
            yylval = *yytext - 'a';
            return VARIABLE;
        }
    /* integers */
[0-9]+  {
```

```
            yylval = atoi(yytext);
            return INTEGER;
        }
        /* operators */
[-+()=/*\n]     { return *yytext; }
        /* skip whitespace */
[ \t]           ;
        /* anything else is an error */
.       yyerror("invalid character");
%%
int yywrap(void) {
        return 1;
}
```

接下来是 yacc 的输入文件。yacc 利用 **INTEGER**和 **VARIABLE**的标记在 **y.tab.h**中生成 **#defines** 以便在 lex 中使用。这跟在算术操作符定义之后。我们可以指定**%left**，表示左结合，或者用**%right** 表示右结合。最后列出的定义拥有最高的优先权。因此乘法和除法拥有比加法和减法更高的优先权。所有这四个算术符都是左结合的。运用这个简单的技术，我们可以消除文法的歧义。

```
%token      INTEGER VARIABLE
%left       '+' '-'
%left       '*' '/'
%{
        void yyerror(char *);
        int yylex(void);
        int sym[26];
%}
%%
program:
            program statement '\n'
            |
            ;
statement:
            expr { printf("%d\n", $1); }
            | VARIABLE '=' expr { sym[$1] = $3; }
            ;
expr:
            INTEGER
            | VARIABLE { $$ = sym[$1]; }
```

```
        | expr '+' expr { $$ = $1 + $3; }
        | expr '-' expr { $$ = $1 - $3; }
        | expr '*' expr { $$ = $1 * $3; }
        | expr '/' expr { $$ = $1 / $3; }
        | '(' expr ')' { $$ = $2; }
        ;

%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
int main(void) {
    yyparse();
    return 0;
}
```

计算器

描述

这个版本的计算器的复杂度大大超过了前一个版本。主要改变包括像**if-else** 和**while**这样的控制结构。另外，在剖析过程中还构造了一个语法树。剖析完成之后，我们历遍语法树来生成输出。此处提供了两个版本的树历遍程序：

- 一个在历遍树的过程中执行树中的声明的解释器，以及
- 一个为基于堆栈的计算机生成代码的编译器。

为了使我们的解释更加形象，这里有一个例程，

```
x = 0;
while (x < 3) {
    print x;
    x = x + 1;
}
```

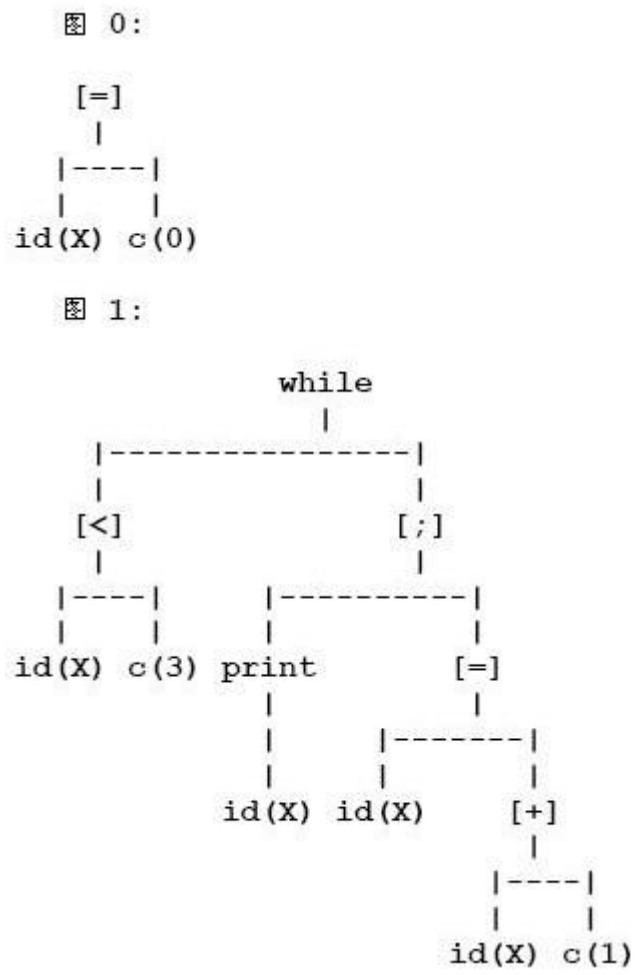
以及解释版本的输出数据，

```
0
1
2
```

和编译版本的输出数据，和

```
    push 0
    pop x
L000:
    push x
    push 3
    compLT
    jz L001
    push x
    print
    push x
    push 1
    add
    pop x
    jmp L000
L001:
```

一个生成语法树的版本。



包含文件中包括了对语法树和符号表的定义。符号表 sym 允许使用单个字符表示变量名。语法树中的每个节点保存一个常量（conNodeType），标识符（idNodeType），或者一个带算子（oprNodeType）的内部节点。所有这三种变量压缩在一个 union 结构中，而节点的具体类型跟据其内部所拥有的结构来判断。

lex 输入文件中包含有返回 VARIABLE 和 INTEGER 标志的正则表达式。另外，也定义了像 EQ 和 NE 这样的双字符算子的标志。对于单字符算子，只需简单地返回其本身。

yacc 输入文件中定义了 YYSTYPE，yylval 的类型，定义如下

```
%union {
    int iValue;                 /* integer value */
    char sIndex;                /* symbol table index */
    nodeType *nPtr;             /* node pointer */
};
```

这将导致在 y.tab.h 中生成如下代码：

```
typedef union {
    int iValue; /* integer value */
```

```
        char sIndex; /* symbol table index */
        nodeType *nPtr; /* node pointer */
    } YYSTYPE;
    extern YYSTYPE yylval;
```

在剖析器的内容栈中，常量、变量和节点都可以由 yylval 表示。注意下面的定义

```
    %token <iValue> INTEGER
    %type <nPtr> expr
```

这把 expr 和 INTEGER 分别绑定到 union 结构 YYSTYPE 中的 nPtr 和 iValue 成员。这是必须的，只有这样 yacc 才能生成正确的代码。例如，这个规则

```
    expr: INTEGER { $$ = con($1); }
```

可以生成下面的代码。注意，yyvsp[0] 表示内容栈的顶部，或者表示对应于 INTEGER 的值。

```
    yylval.nPtr = con(yyvsp[0].iValue);
```

一元算子的优先级比二元算子要高，如下所示

```
    %left GE LE EQ NE '>' '<'
    %left '+' '-'
    %left '*' '/'
    %nonassoc UMINUS
```

%nonassoc 意味着没有依赖关系。它经常在连接词中和 %prec 一起使用，用于指定一个规则的优先级。因此，我们可以这样

```
    expr: '-' expr %prec UMINUS { $$ = node(UMINUS, 1, $2); }
```

表示这条规则的优先级和标志 UMINUS 相同。而且，如同上面所定义的，UMINUS 的优先级比其它所有算子都高。类似的技术也用于消除 if-else 结构中的二义性（请看 if-else 二义性）。

语法树是从底向上构造的，当变量和整数减少时才分配叶节点。当遇到算子时，就需要分配一个节点，并且把上一个分配的节点作为操作数记录在其中。

构造完语法树之后，调用函数 ex 对此语法树进行第一深度历遍。第一深度历遍按照原先节点分配的顺序访问各节点。

这将导致各算子按照剖析期间的访问顺序被使用。此处含有三个版本的 ex 函数：一个解释版本，一个编译版本，一个用于生成语法树的版本。

包含文件

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;
/* constants */
typedef struct {
    int value; /* value of constant */
} conNodeType;
/* identifiers */
typedef struct {
```

```
    int i; /* subscript to sym array */
} idNodeType;
/* operators */
typedef struct {
    int oper; /* operator */
    int nops; /* number of operands */
    struct nodeTypeTag *op[1]; /* operands (expandable) */
} oprNodeType;
typedef struct nodeTypeTag {
    nodeEnum type; /* type of node */
    /* union must be last entry in nodeType */
    /* because operNodeType may dynamically increase */
    union {
        conNodeType con; /* constants */
        idNodeType id; /* identifiers */
        oprNodeType opr; /* operators */
    };
} nodeType;
extern int sym[26];
```

Lex 输入文件

```
%{
#include <stdlib.h>
#include "calc3.h"
#include "y.tab.h"
void yyerror(char *);
%}
%%
[a-z] {
    yylval.sIndex = *yytext - 'a';
    return VARIABLE;
    }
[0-9]+ {
    yylval.iValue = atoi(yytext);
    return INTEGER;
    }
[-()<>=+*/;{}.] {
```

```
        return *yytext;
        }
">="    return GE;
"<="    return LE;
"=="    return EQ;
"!="    return NE;
"while"    return WHILE;
"if"    return IF;
"else"    return ELSE;
"print"    return PRINT;
[ \t\n]+    ;          /* ignore whitespace */
.              yyerror("Unknown character");
%%
int yywrap(void) {
        return 1;
}
```

Yacc 输入文件

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calc3.h"
/* prototypes */
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);
void yyerror(char *s);
int sym[26]; /* symbol table */
%}
%union {
        int iValue; /* integer value */
        char sIndex; /* symbol table index */
        nodeType *nPtr; /* node pointer */
```

```
    };
%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%type <nPtr> stmt expr stmt_list
%%
program:
        function                { exit(0); }
        ;
function:
        function stmt           { ex($2); freeNode($2); }
        | /* NULL */
        ;
stmt:
        ';'                             { $$ = opr(';', 2, NULL, NULL); }
        | expr ';'                      { $$ = $1; }
        | PRINT expr ';'                { $$ = opr(PRINT, 1, $2); }
        | VARIABLE '=' expr ';'         { $$ = opr('=', 2, id($1), $3); }
        | WHILE '(' expr ')' stmt       { $$ = opr(WHILE, 2, $3, $5); }
        | IF '(' expr ')' stmt %prec IFX    { $$ = opr(IF, 2, $3, $5); }
        | IF '(' expr ')' stmt ELSE stmt    { $$ = opr(IF, 3, $3, $5, $7); }
        | '{' stmt_list '}' { $$ = $2; }
        ;
stmt_list:
        stmt                    { $$ = $1; }
        | stmt_list stmt        { $$ = opr(';', 2, $1, $2); }
        ;
expr:
        INTEGER                 { $$ = con($1); }
        | VARIABLE              { $$ = id($1); }
        | '-' expr %prec UMINUS         { $$ = opr(UMINUS, 1, $2); }
```

```
        | expr '+' expr          { $$ = opr('+', 2, $1, $3); }
        | expr '-' expr          { $$ = opr('-', 2, $1, $3); }
        | expr '*' expr          { $$ = opr('*', 2, $1, $3); }
        | expr '/' expr          { $$ = opr('/', 2, $1, $3); }
        | expr '<' expr          { $$ = opr('<', 2, $1, $3); }
        | expr '>' expr          { $$ = opr('>', 2, $1, $3); }
        | expr GE expr           { $$ = opr(GE, 2, $1, $3); }
        | expr LE expr           { $$ = opr(LE, 2, $1, $3); }
        | expr NE expr           { $$ = opr(NE, 2, $1, $3); }
        | expr EQ expr           { $$ = opr(EQ, 2, $1, $3); }
        | '(' expr ')'           { $$ = $2; }
        ;
%%
#define SIZEOF_NODETYPE ((char *)&p->con - (char *)p)
nodeType *con(int value) {
    nodeType *p;
    size_t nodeSize;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(conNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");
    /* copy information */
    p->type = typeCon;
    p->con.value = value;
    return p;
}
nodeType *id(int i) {
    nodeType *p;
    size_t nodeSize;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(idNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");
    /* copy information */
    p->type = typeId;
    p->id.i = i;
    return p;
```

```
}
nodeType *opr(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    size_t nodeSize;
    int i;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(oprNodeType) +
    (nops - 1) * sizeof(nodeType*);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");
    /* copy information */
    p->type = typeOpr;
    p->opr.oper = oper;
    p->opr.nops = nops;
    va_start(ap, nops);
    for (i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
    va_end(ap);
    return p;
}
void freeNode(nodeType *p) {
    int i;
    if (!p) return;
    if (p->type == typeOpr) {
        for (i = 0; i < p->opr.nops; i++)
            freeNode(p->opr.op[i]);
    }
    free (p);
}
void yyerror(char *s) {
    fprintf(stdout, "%s\n", s);
}
int main(void) {
    yyparse();
    return 0;
}
```

解释器

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"
int ex(nodeType *p) {
    if (!p) return 0;
    switch(p->type) {
        case typeCon: return p->con.value;
        case typeId: return sym[p->id.i];
        case typeOpr:
            switch(p->opr.oper) {
                case WHILE: while(ex(p->opr.op[0]))   ex(p->opr.op[1]);
                            return 0;
                case IF:  if (ex(p->opr.op[0]))  ex(p->opr.op[1]);
                            else if (p->opr.nops > 2)  ex(p->opr.op[2]);
                            return 0;
                case PRINT: printf("%d\n", ex(p->opr.op[0]));
                            return 0;
                case ';': ex(p->opr.op[0]);
                            return ex(p->opr.op[1]);
                case '=': return sym[p->opr.op[0]->id.i] = ex(p->opr.op[1]);
                case UMINUS: return -ex(p->opr.op[0]);
                case '+': return ex(p->opr.op[0]) + ex(p->opr.op[1]);
                case '-': return ex(p->opr.op[0]) - ex(p->opr.op[1]);
                case '*': return ex(p->opr.op[0]) * ex(p->opr.op[1]);
                case '/': return ex(p->opr.op[0]) / ex(p->opr.op[1]);
                case '<': return ex(p->opr.op[0]) < ex(p->opr.op[1]);
                case '>': return ex(p->opr.op[0]) > ex(p->opr.op[1]);
                case GE: return ex(p->opr.op[0]) >= ex(p->opr.op[1]);
                case LE: return ex(p->opr.op[0]) <= ex(p->opr.op[1]);
                case NE: return ex(p->opr.op[0]) != ex(p->opr.op[1]);
                case EQ: return ex(p->opr.op[0]) == ex(p->opr.op[1]);
            }
    }
    return 0;
}
```

编译器

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"
static int lbl;
int ex(nodeType *p) {
    int lbl1, lbl2;
    if (!p) return 0;
    switch(p->type) {
        case typeCon: printf("\tpush\t%d\n", p->con.value);
            break;
        case typeId:  printf("\tpush\t%c\n", p->id.i + 'a');
            break;
        case typeOpr:
         switch(p->opr.oper) {
            case WHILE:  printf("L%03d:\n", lbl1 = lbl++);
                ex(p->opr.op[0]);
                printf("\tjz\tL%03d\n", lbl2 = lbl++);
                ex(p->opr.op[1]);
                printf("\tjmp\tL%03d\n", lbl1);
                printf("L%03d:\n", lbl2);
                break;
            case IF:  ex(p->opr.op[0]);
            if (p->opr.nops > 2) {
                /* if else */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
                ex(p->opr.op[1]);
                printf("\tjmp\tL%03d\n", lbl2 = lbl++);
                printf("L%03d:\n", lbl1);
                ex(p->opr.op[2]);
                printf("L%03d:\n", lbl2);
            } else {
                /* if */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
                ex(p->opr.op[1]);
                printf("L%03d:\n", lbl1);
```

```
            }
            break;
        case PRINT:  ex(p->opr.op[0]);
            printf("\tprint\n");
            break;
        case '=':  ex(p->opr.op[1]);
            printf("\tpop\t%c\n", p->opr.op[0]->id.i + 'a');
            break;
        case UMINUS:  ex(p->opr.op[0]);
            printf("\tneg\n");
            break;
        default:  ex(p->opr.op[0]);
            ex(p->opr.op[1]);
            switch(p->opr.oper) {
                case '+': printf("\tadd\n"); break;
                case '-': printf("\tsub\n"); break;
                case '*': printf("\tmul\n"); break;
                case '/': printf("\tdiv\n"); break;
                case '<': printf("\tcompLT\n"); break;
                case '>': printf("\tcompGT\n"); break;
                case GE: printf("\tcompGE\n"); break;
                case LE: printf("\tcompLE\n"); break;
                case NE: printf("\tcompNE\n"); break;
                case EQ: printf("\tcompEQ\n"); break;
            }
        }
    }
    return 0;
}
```

图

```
/* source code courtesy of Frank Thomas Braun */
#include <stdio.h>
#include <string.h>
#include "calc3.h"
#include "y.tab.h"
int del = 1; /* distance of graph columns */
```

```
int eps = 3; /* distance of graph lines */
/* interface for drawing (can be replaced by "real" graphic using GD or
other) */
void graphInit (void);
void graphFinish();
void graphBox (char *s, int *w, int *h);
void graphDrawBox (char *s, int c, int l);
void graphDrawArrow (int c1, int l1, int c2, int l2);
/* recursive drawing of the syntax tree */
void exNode (nodeType *p, int c, int l, int *ce, int *cm);
/*********************************************************/
/* main entry point of the manipulation of the syntax tree */
int ex (nodeType *p) {
        int rte, rtm;
        graphInit ();
        exNode (p, 0, 0, &rte, &rtm);
        graphFinish();
        return 0;
}
/*c----cm---ce----> drawing of leaf-nodes
l leaf-info
*/
/*c---------------cm--------------ce----> drawing of non-leaf-nodes
l node-info
* |
* ------------ ...----
* | | |
* v v v
* child1 child2 ... child-n
* che che che
*cs cs cs cs
*
*/
void exNode
( nodeType *p,
int c, int l, /* start column and line of node */
int *ce, int *cm /* resulting end column and mid of node */
```

```
)
{
    int w, h; /* node width and height */
    char *s; /* node text */
    int cbar; /* "real" start column of node (centred above
                subnodes) */
    int k; /* child number */
    int che, chm; /* end column and mid of children */
    int cs; /* start column of children */
    char word[20]; /* extended node text */
    if (!p) return;
    strcpy (word, "???"); /* should never appear */
    s = word;
    switch(p->type) {
        case typeCon: sprintf (word, "c(%d)", p->con.value); break;
        case typeId: sprintf (word, "id(%c)", p->id.i + 'A'); break;
        case typeOpr:
        switch(p->opr.oper){
            case WHILE: s = "while"; break;
            case IF: s = "if"; break;
            case PRINT: s = "print"; break;
            case ';': s = "[;]"; break;
            case '=': s = "[=]"; break;
            case UMINUS: s = "[_]"; break;
            case '+': s = "[+]"; break;
            case '-': s = "[-]"; break;
            case '*': s = "[*]"; break;
            case '/': s = "[/]"; break;
            case '<': s = "[<]"; break;
            case '>': s = "[>]"; break;
            case GE: s = "[>=]"; break;
            case LE: s = "[<=]"; break;
            case NE: s = "[!=]"; break;
            case EQ: s = "[==]"; break;
        }
        break;
    }
```

```
        /* construct node text box */
        graphBox (s, &w, &h);
        cbar = c;
        *ce = c + w;
        *cm = c + w / 2;
        /* node is leaf */
        if (p->type == typeCon || p->type == typeId || p->opr.nops == 0) {
            graphDrawBox (s, cbar, l);
            return;
        }
        /* node has children */
        cs = c;
        for (k = 0; k < p->opr.nops; k++) {
            exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
            cs = che;
        }
        /* total node width */
        if (w < che - c) {
            cbar += (che - c - w) / 2;
            *ce = che;
            *cm = (c + che) / 2;
        }
        /* draw node */
        graphDrawBox (s, cbar, l);
        /* draw arrows (not optimal: children are drawn a second time) */
        cs = c;
        for (k = 0; k < p->opr.nops; k++) {
            exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
            graphDrawArrow (*cm, l+h, chm, l+h+eps-1);
            cs = che;
        }
}
/* interface for drawing */
#define lmax 200
#define cmax 200
char graph[lmax][cmax]; /* array for ASCII-Graphic */
int graphNumber = 0;
```

```
void graphTest (int l, int c)
{   int ok;
    ok = 1;
    if (l < 0) ok = 0;
        if (l >= lmax) ok = 0;
    if (c < 0) ok = 0;
    if (c >= cmax) ok = 0;
    if (ok) return;
    printf ("\n+++error: l=%d, c=%d not in drawing rectangle 0, 0 ...%d, %d",l, c, lmax, cmax);
    exit (1);
}
void graphInit (void) {
    int i, j;
    for (i = 0; i < lmax; i++) {
        for (j = 0; j < cmax; j++) {
            graph[i][j] = ' ';
        }
    }
}
void graphFinish() {
    int i, j;
    for (i = 0; i < lmax; i++) {
        for (j = cmax-1; j > 0 && graph[i][j] == ' '; j--);
        graph[i][cmax-1] = 0;
        if (j < cmax-1) graph[i][j+1] = 0;
        if (graph[i][j] == ' ') graph[i][j] = 0;
    }
    for (i = lmax-1; i > 0 && graph[i][0] == 0; i--);
    printf ("\n\nGraph %d:\n", graphNumber++);
    for (j = 0; j <= i; j++) printf ("\n%s", graph[j]);
    printf("\n");
}
void graphBox (char *s, int *w, int *h) {
    *w = strlen (s) + del;
    *h = 1;
}
void graphDrawBox (char *s, int c, int l) {
```

```
        int i;
        graphTest (l, c+strlen(s)-1+del);
        for (i = 0; i < strlen (s); i++) {
                graph[l][c+i+del] = s[i];
        }
}
void graphDrawArrow (int c1, int l1, int c2, int l2) {
        int m;
        graphTest (l1, c1);
        graphTest (l2, c2);
        m = (l1 + l2) / 2;
        while (l1 != m) {
                graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
        }
        while (c1 != c2) {
                graph[l1][c1] = '-'; if (c1 < c2) c1++; else c1--;
        }
        while (l1 != l2) {
                graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
        }
        graph[l1][c1] = '|';
}
```

Lex 进阶

字符串

程序语言中经常出现双引号字符串。这里是在 lex 中匹配双引号字符串的一种方法：

```
%{
char *yylval;
#include <string.h>
%}
%%
\"[^"\n]*["\n]    {
                yylval = strdup(yytext+1);
                if (yylval[yyleng-2] != '"')
                    warning("improperly terminated string");
                else
                    yylval[yyleng-2] = 0;
                printf("found '%s'\n", yylval);
                }
```

上面的例子确保匹配的字符串不超越行边界，并且删除不闭合的双引号字符串。如果我们想增加转义控制序列，例如“\n”或“\"”，运用开始声明可以简化此工作：

```
%{
char buf[100];
char *s;
%}
%x STRING
%%
\"   { BEGIN STRING; s = buf; }
<STRING>\\n    { *s++ = '\n'; }
<STRING>\\t    { *s++ = '\t'; }
<STRING>\\\"   { *s++ = '\"'; }
<STRING>\"     {
                *s = 0;
                BEGIN 0;
                printf("found '%s'\n", buf);
                }
<STRING>\n     { printf("invalid string"); exit(1); }
<STRING>.      { *s++ = *yytext; }
```

在定义段中定义了唯一的开始状态 STRING。当扫描器扫描到一个双引号时，宏 BEGIN 使 lex

转入 STRING 状态。lex 保持在 STRING 状态，只识别以字符串 <STRING> 开始的范式，直到执行另一个 BEGIN。从而，我们获得了一个专门用于扫描字符串的小环境。当探测到结尾的双引号时，程序双转回状态 0，即初始状态。

保留字

如果你的程序有一个庞大的保留字集，为了提高效率，不妨让 lex 简单的匹配字符串，然后在你自己的代码中判断字符串是否是保留字。例如下面这段代码效率较低

```
"if" return IF;
"then" return THEN;
"else" return ELSE;
{letter}({letter}|{digit})* {
        yylval.id = symLookup(yytext);
        return IDENTIFIER;
        }
```

这里的 symLookup 返回符号表中的索引，更好的方法是同时搜索保留字和标志。像下面这样：

```
{letter}({letter}|{digit})*    {
        int i;
        if ((i = resWord(yytext)) != 0)
            return (i);
        yylval.id = symLookup(yytext);
        return (IDENTIFIER);
        }
```

这个技术显著地减少了要求的状态数，因此可以明显减小扫描表。

lex 的调试

Lex 有很多方便调试的工具。对于不同版本的 lex 其特征可能各不相同，因此你最好参考文档以了解其细节。通过指定命令行参数“-d”，lex 会在 lex.yy.c 中生成调试状态。通过设置变量 yy_flex_debug 可以打开或关闭 flex（GNU 版本的 lex）中调试信息的输出。输出信息包括应用的规则和相应的匹配文字。如果你在一起使用 lex 和 yacc，那么需要在你的 yacc 输入文件中增加下面的代码：

```
extern int yy_flex_debug;
int main(void) {
    yy_flex_debug = 1;
    yyparse();
}
```

你也可能选择编写自己的调试代码，定义函数来显示各标志（token）对应的内容，以及联合体

yylval 中每一个成员变量的的值。下面的例程说明了这个方法。当定义了 DEBUG 时，调试函数就能起作用，显示标志（token）和与之相应的值的变化轨迹：

```
%union {
int ivalue;
...
};
%{
#ifdef DEBUG
    int dbgToken(int tok, char *s) {
        printf("token %s\n", s);
        return tok;
    }
    int dbgTokenIvalue(int tok, char *s) {
        printf("token %s (%d)\n", s, yylval.ivalue);
        return tok;
    }
    #define RETURN(x) return dbgToken(x, #x)
    #define RETURN_ivalue(x) return dbgTokenIvalue(x, #x)
#else
    #define RETURN(x) return(x)
    #define RETURN_ivalue(x) return(x)
#endif
%}
%%
[0-9]+  {
        yylval.ivalue = atoi(yytext);
        RETURN_ivalue(INTEGER);
    }
"if"    RETURN(IF);
"else"    RETURN(ELSE);
```

Yacc 进阶

递归

当设计一个列表时，我们可以这样使用左递归，

```
list:
    item
    | list ',' item
    ;
```

或者右递归：

```
list:
    item
    | item ',' list
```

如果使用了右递归，列表中的所有项都会被压入堆栈。在压入最后一个项之后，才开始简化。当使用左递归时，堆栈中项的数目永远不会超过三个，即一边前进一边简化。因此，使用基递归更好些。

If-Else 歧义

隐藏在**If-Else**结构中的一个“移进 - 归约”冲突（**shift-reduce conflict**）经常出现。假定我们有下面的规则：

```
stmt:
    IF expr stmt
    | IF expr stmt ELSE stmt
    ...
```

以及下面的状态：

IF expr stmt IF expr stmt . ELSE stmt

我们需要确定是否应该移进**ELSE**，或者归约堆栈顶部的**IF expr stmt**。如果移进，那么结果会这样：

IF expr stmt IF expr stmt . ELSE stmt

IF expr stmt IF expr stmt ELSE . stmt

IF expr stmt IF expr stmt ELSE stmt .

IF expr stmt stmt .

第二个**ELS**E 和第二个 **IF**配对。如果归约，又会这样：

IF expr stmt IF expr stmt . ELSE stmt

IF expr stmt stmt . ELSE stmt

IF expr stmt . ELSE stmt

IF expr stmt ELSE . stmt

IF expr stmt ELSE stmt .

第二个**ELS**E 和第一个 **IF**配对。现代程序语言通常把**ELSE**和最近的未配对**IF**配对，所以前面的结果才是我们希望得到的。这个动作是**yacc**的默认行为，当遇到“移进 - 归约”冲突（**shift-reduce conflict**）时，默认行为是“移进”。

尽管 yacc 的动作是正确的，它仍然会显示一个“移进 - 归约”冲突（**shift-reduce conflict**）警告。为了阻止警告信息，可以给**IF-ELSE**结构定义比单独的**IF**更高的优先权：

```
%nonassoc IFX
%nonassoc ELSE
stmt:
    IF expr stmt %prec IFX
    | IF expr stmt ELSE stmt
```

错误信息

一个优秀的编译器应当给用户提供有意义的错误信息。例如，下面的信息没有传达多少有意义的信息：

```
syntax error
```

如果我们在**lex**中跟踪行号，那么至少可以给用户提供出错的行号：

```
void yyerror(char *s) {
    fprintf(stderr, "line %d: %s\n", yylineno, s);
}
```

当**yacc**发现一个解析错误，默认动作是调用**yyerror**，然后从**yylex**中返回一个值或**1**。下面是更优雅的动作，把输入流刷新为一个分隔符，然后继续扫描：

```
stmt:
    ';'
    | expr ';'
    | PRINT expr ';'
    | VARIABLE '=' expr ';
    | WHILE '(' expr ')' stmt
    | IF '(' expr ')' stmt %prec IFX
    | IF '(' expr ')' stmt ELSE stmt
    | '{' stmt_list '}'
    | error ';'
    | error '}'
    ;
```

标志 error 是 yacc 中的一个特殊标志，会匹配任意输入直到碰到紧随其后的错误符号。在这个例子中，当 yacc 遇到错误时它会调用 yyerror，把输入流刷新为下一个分号或者右花括号。然后继续扫描。

继承属性

到目前为止的例程都使用了综合属性。在一棵语法树的任意一点中我们都能基于子节点的属性来判断此节点的属性。观察这个规则

```
expr: expr '+' expr          { $$ = $1 + $3; }
```

由于我们是从底向上剖析的，两个操作数的值都是可用的，并且我们可以计算出对应左边的值。一个节点的继承属性依赖于其父节点或兄弟节点的值。下面的语法定义了一个 C 变量的声明：

```
decl: type varlist
type: INT | FLOAT
varlist:
      VAR                { setType($1, $0); }
      | varlist ',' VAR    { setType($3, $0); }
```

这里是此例程的分析：

```
. INT VAR
INT . VAR
type . VAR
type VAR .
type varlist .
decl .
```

当我们把 VAR 归约到 varlist 去的时候，我们应当为符号表注明此变量的类型。当然，这个类型隐藏在堆栈中。解决这个问题的办法是把堆栈索引向后推。回想一下 $1 指向右边第一项。我们可以把索引值向后推，使用 $0、$-1 等等。既然这样，$0 是可以正确使用的。如果你要指定一个特定类型的标志，语法是这样的 $<tokentype>0，用尖括号括起来。在这个特殊例子中，注意必须总是指向变量列表之前的不确定类型。

内含动作

yacc 中的规则可以包含内含动作：

```
list: item1 { do_item1($1); } item2 { do_item2($3); } item3
```

注意每个动作也在堆栈中占有一个位置，所以 do_item2 中必须用 $3 指向 item2。实际上，这个语法被 yacc 翻译成了下面这样：

```
list: item1 _rule01 item2 _rule02 item3
_rule01: { do_item1($0); }
_rule02: { do_item2($0); }
```

调试 Yacc

Yacc 包含有允许调试的工具。这个特性可能随**yacc**版本的不同而各不相同，所以你应当查看参考文档以了解其细节。通过定义**YYDEBUG**并且把它设置成非零值，**yacc**就会在**y.tab.**c 中生成调试

状态代码。这也需要在命令行指定参数“**-t**”。如果设置了**YYDEBUG**，通过设置**yydebug**可以打开或者关闭调试信息的输出。输出信息包括扫描到的标志和移进 / 归约动作。

```
%{
#define YYDEBUG 1
%}
%%
...
%%
int main(void) {
    #if YYDEBUG
    yydebug = 1;
    #endif
    yylex();
}
```

另外，你可以通过指定命令行参数“-v”保存剖析状态。状态会保存在文件**y.output**中，当调试语法时常常很有用。你还可以选择编写自己的调试代码，方法是定义宏**TRACE**，如同下面的例子。当定义**DEBUG**之后，归约的轨迹会带着行号显示。

```
%{
#ifdef DEBUG
#define TRACE printf("reduce at line %d\n", __LINE__);
#else
#define TRACE
#endif
%}
%%
statement_list:
        statement
            { TRACE $$ = $1; }
        | statement_list statement
            { TRACE $$ = newNode(';', 2, $1, $2); }
        ;
```

参考书目

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman [1986]. Compilers, Prinicples, Techniques and Tools. Addison-Wesley, Reading, Massachusetts.

Gardner, Jim, Chris Retterath and Eric Gisin [1988]. MKS Lex & Yacc. Mortice Kern Systems Inc., Waterloo, Ontario, Canada.

Johnson, Stephen C. [1975]. Yacc: Yet Another Compiler Compiler. Computing Science

Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey. A PDF version is available at ePaperPress.

Lesk, M. E. and E. Schmidt [1975]. Lex – A Lexical Analyzer Generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey. A PDF version is available at ePaperPress.

Levine, John R., Tony Mason and Doug Brown [1992]. Lex & Yacc. O'Reilly & Associates, Inc. Sebastopol, California.