LEX和YACC指南

作者：THOMAS NIEMANN

e-papers
Information Online

前言

本文解释了如何使用lex和yacc构建一个编译器。Lex和yacc是用来生成词法分析器和解析器的工具。我假设你能用C语言编程，并理解数据结构，如链接列表和树。

引言描述了编译器的基本构件，并解释了lex和yacc之间的互动。接下来的两节将更详细地描述lex和yacc。有了这些背景，我们就可以构建一个复杂的计算器。传统的算术运算和控制语句，如*if-else*和*while*，都已实现。在稍作改动后，我们将计算器转换为一个基于堆栈的机器的编译器。其余部分讨论了在编译器编写中经常出现的问题。例子的源代码可以从下面列出的网站上下载。

THOMAS NIEMANN
波特兰，俄勒冈州

niemannt@yahoo.com
http://members.xoom.com/thomasn

内容

1. 简介

直到1975年，编写一个编译器是一个非常耗时的过程。然后Lesk [1975] 和Johnson [1975]
发表了关于lex和yacc的论文。这些工具大大简化了编译器的编写。lex和yacc的实现细节可以在Aho [1986]中找到。Lex和yacc可以从

- Mortice Kern系统（MKS），在http://www.mks.com。
- GNU flex和bison，在http://www.gnu.org。
- 明，在http://agnes.dida.physik.uni-essen.de/~janjaap/mingw32。
- 天鹅座，在http://www.cygnus.com/misc/gnu-win32，和
- 我，在 http://members.xoom.com/thomasn/y_gnu.zip （可执行文件），以及http://members.xoom.com/thomasn/y_gnus.zip（y_gnu.zip的源代码）。

来自MKS的版本是一种高质量的商业产品，售价约为100美元。
300US美元。GNU软件是免费的。flex的输出可以用于商业产品，从1.24版开始，bison也是如此。Ming和Cygnus是GNU软件的32位Windows-
95/NT端口。我的版本是基于Ming的，但用Visual
C++编译，并包括文件处理例程中的一个小错误修正。如果你下载了我的版本，在解压时要注意保留目录结构。

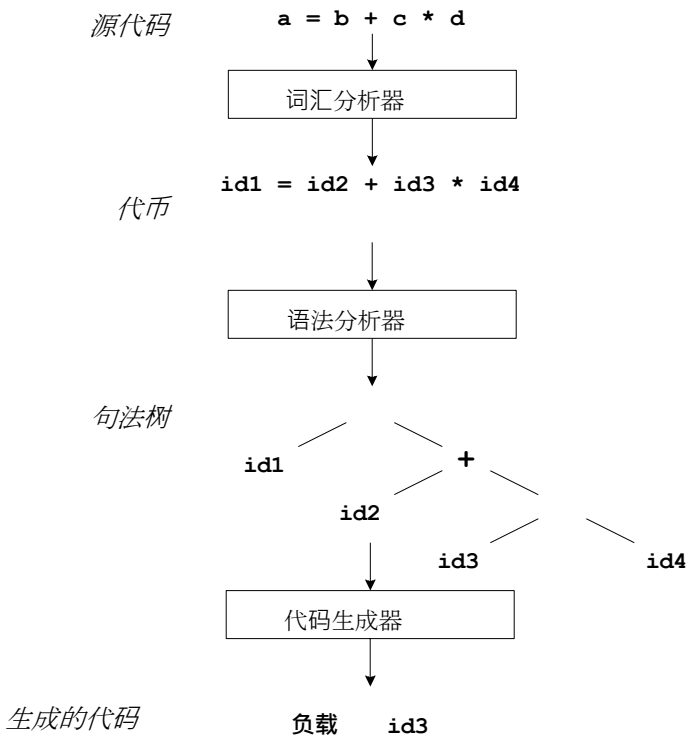


图1-1: 编译顺序

Lex为一个词法分析器或扫描器生成C代码。它使用与输入的字符串相匹配的模式，并将字符串转换为*代币*。标记是字符串的数字表示，可以简化处理。这在图1-1中有所说明。

当lex在输入流中找到标识符时，它将它们输入到一个*符号表中*。符号表还可能包含其他信息，如数据类型（*整数*或*实数*）和内存中变量的位置。所有随后对标识符的引用都是指适当的符号表索引。

Yacc为语法分析器（或称解析器）生成C代码。Yacc使用语法规则，使其能够分析lex中的标记并创建一个语法树。语法树对标记施加了一个层次结构。例如，在语法树中，运算符的优先级和关联性是很明显的。下一步，代码生成，对语法树进行深度优先的行走以生成代码。一些编译器产生机器码，而另一些编译器，如上图所示，输出汇编。

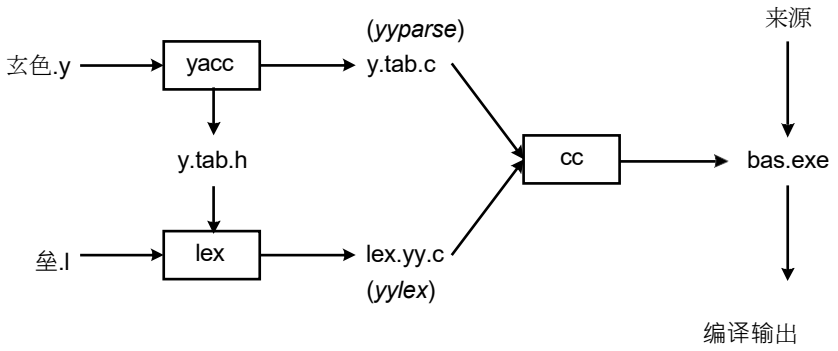


图1-2：用Lex/Yacc建立一个编译器

图1-2说明了lex和yacc使用的文件命名规则。我们假设我们的目标是编写一个BASIC编译器。首先，我们需要指定lex的所有模式匹配规则（**bas.l**）和yacc的语法规则（**bas.y**）。创建我们的编译器**bas.exe**的命令列在下面。

```
yacc -d bas.y# 创建y.tab.h, y.tab.c
                                    lex bas.l#create
                                    lex.yy.c cc
lex.yy.c y.tab.c -obas.exe#compile/link
```

Yacc读取**bas.y**中的语法描述，并在文件**y.tab.c**中生成一个解析器，即函数**yyparse**。在文件**bas.y**中包括标记声明。这些被yacc转换为常量定义并放在文件**y.tab.h** 中。Lex读取**bas.l**中的模式描述，包括文件**y.tab.h**，并在文件**lex.yy.c**中产生一个词法分析器，即函数**yylex**。

最后，词法分析器和解析器被编译并连接在一起，形成可执行文件**bas.exe**。在**main**中，我们调用**yyparse**来运行编译器。函数**yyparse**自动调用**yylex**来获得

每个标记。

2. 莱克斯

2.1 理论

编译器的第一阶段是读取输入源，并将源中的字符串转换为标记。使用正则表达式，我们可以向lex指定模式，让它扫描和匹配输入的字符串。lex中的每个模式都有一个相关的动作。通常，一个动作会返回一个标记，代表匹配的字符串，供解析器随后使用。然而，在开始时，我们将简单地打印匹配的字符串，而不是返回一个标记值。我们可以使用正则表达式来扫描标识符

字母 (字母 | 数字) *

这个模式匹配以单个字母开头，后面是零个或多个字母或数字的字符串。这个例子很好地说明了正则表达式中允许的操作。

- 重复，用 "*****"运算符表示
- 交替，用"**|**"运算符表示
- 串联

*任何*正则表达式都可以被表达为有限状态自动机（FSA）。我们可以用状态和状态之间的转换来表示一个FSA。有一个起始状态，以及一个或多个最终状态或接受状态。

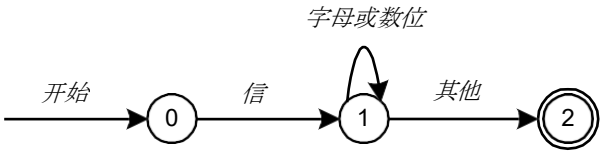


图2-1:有限状态自动机

在图2-
1中，状态0是开始状态，而状态2是接受状态。随着字符被读取，我们从一个状态过渡到另一个状态。当读到第一个字母时，我们过渡到状态1。当更多的字母或数字被读取时，我们保持在状态1。当我们读到一个非字母或数字的字符时，我们过渡到状态2，即接受状态。*任何*FSA*都*可以被表达为一个计算机程序。例如，我们的3状态机器就很容易编程。

```
开始：转入状态0

状态0：读c
        如果c = 信，则转入状态1，转入
        状态0

状态1：读取c
        如果c=字母，则转入状态1 如果c=
        数字，则转入状态1 转入状态2

状态2：接受字符串
```

这就是lex使用的技术。正则表达式被lex翻译成一个模仿FSA的计算机程序。使用下一个*输入*字符和*当前状态*，通过对计算机生成的状态表进行索引，很容易确定下一个状态。

现在我们可以很容易地理解lex的一些限制。例如，lex不能用来识别嵌套结构，如括号。嵌套结构是通过加入一个堆栈来处理的。每当我们遇到一个"`(`"，我们就把它推到堆栈上；当遇到一个"`)`"，我们就把它与堆栈的顶部相匹配，然后弹出堆栈。然而，Lex只有状态和状态之间的转换。由于它没有堆栈，所以它不太适合解析嵌套结构。Yacc用堆栈来增强FSA，可以轻松地处理诸如括号之类的结构。重要的是要使用正确的工具来完成工作。Lex擅长模式匹配。Yacc适合于更具挑战性的任务。

2.2 实践

元气质	*匹配*
`.`	除换行外的任何字符
`\n`	换行
`*`	前面表达式的零个或多个副本
`+`	一个或多个前述表达的副本
`?`	前述表达式的零或一份副本
`^`	行首
`$`	行末
`a|b`	a或b
`(ab)+`	一个或多个ab的副本（分组）。
`"a+b"`	文字 "a+b"（C语言转义仍然有效）。
`[]`	文字类

表2-1:模式匹配基元

表达方式	*匹配*
abc	**abc**
abc*ab	**, abc, abcc, abccc, ...。**
abc+abc	**, abcc, abccc, ...**
a(bc)+abc	**, abcbc, abcbcbc, ...。**
a(bc)?	**a, abc**
[abc]	**a, b, c**
[a-z]	任何字母，从**a**到**z**
[a/z]	**a, -, z**
[-az]	**-, a, z**
[A-Za-z0-9]+	一个或多个字母数字字符
[\t\n]+	空白
[^ab]	任何东西，除了：**A, B**
[a^b]	**a, ^, b**
[a|b]	**a, |, b**
a|ba或**b**	

表2-2：模式匹配的例子

lex中的正则表达式是由*元字符*组成的（表2-1）。模式匹配的例子显示在表2-2中。在一个字符类中，普通的运算符失去了它们的意义。在一个字符类中允许的两个运算符是连字符（"**-**"）和环形符号（"**^**"）。当在两个字符之间使用时，连字符表示一个字符的范围。环形符号，当作为第一个字符使用时，否定了表达式。如果两个模式匹配相同的字符串，最长的匹配者获胜。如果两个匹配都是相同的长度，则使用列出的第一个模式。

对Lex的输入被分为三个部分，用**%%**来划分各部分。这一点最好用例子来说明。第一个例子是最短的lex文件：

```
%%
```

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

输入被复制到输出，一次一个字符。第一个**%%**总是必须的，因为必须有一个规则部分。然而，如果我们没有指定任何规则，那么默认的动作是匹配所有的内容并复制到输出。输入和输出的默认值分别为**stdin**和**stdout**。下面是同样的例子，明确编码了默认值。

```
%%
    /* 匹配除换行之外的所有内容 */
.   ECHO。
    /* 匹配换行 */
\EN ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

在规则部分规定了两种模式。每个模式必须在第一栏开始。后面是*空白*（空格、制表符或换行），以及与该模式相关的可选动作。这个动作可以是一个单一的C语句，或者是用大括号括起来的多个C语句。任何不在第一列开始的内容都会被逐字复制到生成的C文件中。我们可以利用这一行为在我们的lex文件中指定注释。在这个例子中，有两个模式，"."
"和"\n"，每个模式都有一个**ECHO**动作。一些宏和变量是由lex预定义的。**ECHO**是一个写出与模式匹配的代码的宏。这是对任何未匹配的字符串的默认动作。通常情况下，**ECHO**被定义为。

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

变量**yytext**是指向匹配字符串的指针（以**NULL**为结尾），**yyleng**是匹配字符串的长度。变量**yyout**是输出文件，默认为**stdout**。当输入用尽时，lex会调用函数**yywrap**。如果完成了就返回1，如果需要更多处理就返回0。每个C程序都需要一个主函数。在这种情况下，我们简单地调用**yyylex**，这是lex的主入口。有些lex的实现在库中包含了**main**和**yyyrap**的副本，不需要对它们进行明确的编码。这就是为什么我们的第一个例子，最短的lex程序能够正常运行。

名称	*功能*
int yylex(void)	调用lexer，返回token **char**
***yytextpointer**	to matched string **yyleng**
	匹配字符串的长度
yylval	与令牌相关的值
int yywrap(void)	包裹，如果完成返回1，如果没有完成返回0
文件 *yyout	输出文件
文件 *yyin	输入文件
初始化	最初的启动条件
BEGIN条件	开关启动条件
ECHO	写入匹配的字符串

表2-3：Lex预定义变量

这里有一个根本不做任何事情的程序。所有的输入都被匹配，但没有任何行动与任何模式相关联，所以不会有输出。

```
%%
.
\n
```

下面的例子在文件中的每一行都预置了行号。lex的一些实现预先定义并计算**yylineno**。lex的输入文件是**yyin**，默认为**stdin**。

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

定义部分由替换、代码和起始*状态*组成。定义部分的代码被简单地原样复制到生成的C文件的顶部，并且必须用"**%{**"和"**%}**"标记的括号。替换可以简化模式匹配规则。例如，我们可以定义数字和字母。

```
数字      [0-9]字
母[A-Za-z]
%{
    int count;
%}
%%
    /* 匹配标识符 */
{字母}({字母}|{数字})*。            count++。
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

定义性术语和相关表达式之间必须有空格。在规则部分对替换的引用用大括号（**{letter}**）包围，以区分它们与字面意义。当我们在规则部分有一个匹配，相关的C代码就会被执行。下面是一个扫描器，它计算文件中的字符数、单词数和行数（类似于Unix的*wc*）。

```
%{
    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;
}
```

3. Yacc

3.1 理论

yacc的语法是用Backus Naur Form（BNF）的变体来描述的。这种技术是由John Backus和Peter
Naur开创的，并用于描述ALGOL60。BNF语法可以用来表达*无语境*语言。现代编程语言中的大多数结构都可以用BNF表示。例如，一个表达式的乘法和加法的语法是

```
1    E -> E + E
2    E -> E * E
3    E -> id
```

已经指定了三个产品。出现在生产的左侧（lhs）的术语，如E（表达式）是非终端。像**id**（标识符）这样的术语是终端（由lex返回的标记），只出现在生产的右侧（rhs）。这个语法规定，一个表达式可以是两个表达式的和，两个表达式的乘积，或者是一个标识符。我们可以使用这个语法来生成表达式。

```
E->E*E              (r2)
  -> E * z          (r3)
  -> E + E * z      (r1)
  -> E + y * z      (r3)
  -> x + y * z      (r3)
```

在每一步，我们扩展一个术语，用相应的rhs替换生产的lhs。右边的数字表示哪个规则适用。为了解析一个表达式，我们实际上需要做相反的操作。我们不是从一个非终端（起始符号）开始并从语法中生成一个表达式，而是需要*把*一个表达式*减少*到一个非终端。这被称为*自下而上*或*移位还原*解析，并使用堆栈来存储术语。下面是同样的推导，但顺序相反。

```
 1    . x + y * z      移位
 2    x . + y * z      reduce(r3)
 3    E . + y * z      移位
 4    E + . y * z      移位
 5    E + y . * z      reduce(r3)
 6    E + E . * z      移位
 7    E + E   . z      移位
 8    E + E   z .      reduce(r3)
 9    E + E   E .      减少(r2)          发出乘法
10    E + E .          reduce(r1)        发出添加
11    E .              接受
```

点左侧的术语在堆栈中，而剩余的输入则在点的右侧。我们首先将令牌移到堆栈上。当堆栈的顶部与生产的rhs相匹配时，我们用生产的lhs替换堆栈中的匹配代币。从概念上讲，rhs的匹配代币被从堆栈中弹出，而生产的lhs被推到堆栈中。匹配的代币被称为*句柄*，我们将句柄*减少*到生产的lhs上。这个过程一直持续到我们把所有的输入都转移到栈上，栈上只剩下起始非终端。在第1步中，我们将**x**转移到栈中。第2步将规则r3应用于堆栈，将**x**改为**E**。我们继续移位和减少，直到堆栈中只剩下一个非终端，即起始符号。在第9步，当我们减少规则r2时，我们发出了乘法指令。同样地，在第10步中发出了加法指令。因此，乘法比加法有更高的优先权。

然而，考虑一下第6步的转移。与其说是移位，不如说是减少，应用规则r1。这将导致加法比乘法有更高的优先权。这就是所谓的*移位-还原*冲突。我们的语法是*模棱两可的*，因为有不止一个可能的推导会产生表达式。在这种情况下，运算符的优先级会受到影响。作为另一个例子，规则中的关联性

```
E -> E + E
```

是不明确的，因为我们可以在左边或右边进行递归。为了纠正这种情况，我们可以重写语法，或者给yacc提供指示，说明哪个运算符有优先权。后一种方法更简单，我们将在*练习*部分进行演示。

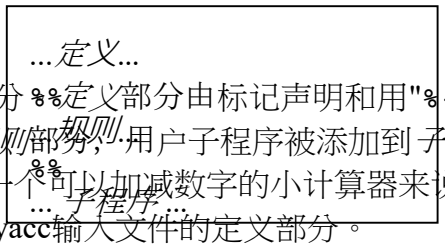
下面的语法有一个*reduce-reduce*冲突。栈上有一个**id**，我们可以减少到**T**，或者减少到**E**。

```
E -> T
E -> id
T -> id
```

当有冲突时，Yacc会采取默认动作。对于shift-reduce冲突，Yacc会进行移位。对于reduce-reduce冲突，它将使用列表中的第一条规则。每当有冲突存在时，它也会发出警告信息。可以通过以下方式抑制警告

语法是不含糊的。在随后的章节中，将介绍几种消除歧义的方法。

3.2 实践，第一部分



yacc的输入被分为三个部分 **%%***定义*部分由标记声明和用"**%{**"和"**%}**"括起来的C代码组成。BNF语法被放在*规则*部分，用户子程序被添加到*子程序*部分。

这一点最好通过构建一个可以加减数字的小计算器来说明。我们先来看看lex和yacc之间的联系。这里是yacc输入文件的定义部分。

```
%token INTEGER
```

这个定义声明了一个**INTEGER**标记。当我们运行yacc时，它在文件**y.tab.c**中生成了一个解析器，并且还创建了一个包含文件**y.tab.h**。

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex包括这个文件，并利用这些定义来获取令牌值。为了获得令牌，yacc调用**yylex**。函数**yylex**的返回类型为**int**，并返回令牌值。与令牌相关的值由lex在变量**yylval**中返回。比如说

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

将把整数的值存储在**yylval**中，并向yacc返回令牌**INTEGER**。**yylval**的类型是由**YYSTYPE**决定的。由于默认的类型是整数，所以在这种情况下效果很好。令牌值0-255是为字符值保留的。例如，如果你有一个规则，如

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

将返回减号或加号的字符值。请注意，我们把减号放在前面，这样它就不会被误认为是一个范围代号。生成的令牌值通常从258左右开始，因为lex为文件结束和错误处理保留了几个值。下面是我们的计算器的完整lex输入规范。

```
%{
#include "y.tab.h"
%}

%%

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

[-+\n]       返回*yytext。

[\t]         ; /* 跳过空格 */

.            yyerror("无效字符")。

%%

int yywrap(void) {
    return 1;
}
```

在内部，yacc在内存中维护两个堆栈；一个是解析堆栈，一个是值堆栈。解析栈包含终端和非终端，并代表当前的解析状态。值栈是一个**YYSTYPE**元素的数组，并将一个值与解析栈中的每个元素联系起来。例如，当lex返回一个**INTEGER**标记时，yacc将这个标记转移到解析栈中。同时，相应的**yylval**被转移到值栈中。解析栈和值栈总是同步的，所以在栈上找到一个与令牌相关的值是很容易完成的。下面是我们的计算器的yacc输入规范。

```
%token INTEGER

%%

方案。
        程序expr '\n'                { printf("%d\n", $2); }
        |
        ;

阐述。
        整数{ $$ = 1; }
        | expr '+' expr{ $$ = 1 + 3; }
        | expr '-' expr{ $$ = 1 - 3; }
        ;

%%

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

int main(void) {
    yyparse();
    return 0;
}
```

规则部分类似于前面讨论的BNF语法。生产的左侧，或非终端，被输入左对齐，后面是冒号。随后是生产的右侧。与一个规则相关的动作用大括号输入。

通过利用左旋，我们指定一个程序由零个或多个表达式组成。每个表达式都以一个换行来结束。当检测到换行时，我们打印表达式的值。当我们应用规则

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

我们将解析堆栈中生产的右侧替换为相同生产的左侧。在这种情况下，我们弹出"**expr '+' expr**"并推入"**expr**"。我们通过从堆栈中弹出三个术语，并推回一个术语来减少堆栈。在我们的C语言代码中，我们可以通过指定"**$1**"为生产右侧的第一个术语，"**$2**"为第二个术语，以此类推来引用值栈中的位置。"**$$**"指的是缩减后的堆栈顶部。上述动作增加了与两个表达式相关的值，从值栈中弹出了三个项，并推回了一个和。因此，解析栈和值栈保持同步。

当我们从**INTEGER**减为NUMBER时，数字值最初被输入到堆栈中。**expr**。在**INTEGER**被移到堆栈后，我们应用规则

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

INTEGER标记被从解析堆栈中弹出，接着推送**expr**。对于值堆栈，我们从堆栈中弹出整数值，然后再把它推回去。换句话说，我们什么都不做。事实上，这是默认动作，不需要指定。最后，当遇到换行时，与**expr**相关的值被打印出来。

在出现语法错误时，yacc会调用用户提供的函数**yyerror**。如果你需要修改**yyerror**的接口，你可以修改yacc包含的*罐头*文件，以适应你的需要。我们的yacc规范中的最后一个函数是**main**
...如果你想知道它在哪里的话。这个例子仍然有一个模棱两可的语法。Yacc会发出shift-reduce的警告，但仍会使用shift作为默认操作来处理这个语法。

3.3 实践，第二部分

在本节中，我们将对上一节的计算器进行扩展，加入一些新的功能。新功能包括算术运算符乘法和除法。括号可以用来覆盖运算符的优先级，单字符变量可以在赋值语句中指定。下面说明了输入和计算器输出的例子。

用户。**3 * (4 + 5)**
计算：**27**
用户：**x = 3 * (5 + 4)**
用户：**Y = 5**
用户：**X**
计算：**27**
用户：**Y**
计算：**5** *用户*：**X**
+ 2*Y *计算*：
37**37**

词法分析器返回**VARIABLE**和**INTEGER**标记。对于变量，**yylval**指定一个索引到**sym**，我们的符号表。在这个程序中，**sym**只是保存相关变量的值。当**INTEGER**标记被返回时，**yylval**包含所扫描的数字。下面是lex的输入规范。

```
%{
    #include "y.tab.h"
%}

%%

    /* 变量 */
[a-z]       {
                yylval = *yytext - 'a';
                return VARIABLE;
            }

    /* 整数 */ [0-
9]+         {
                yylval = atoi(yytext);
                返回INTEGER。
            }

    /* 运营商 */
[-+()=/*/n] { 返回 *yytext; }

    /* 跳过空白 */ [ `t] ;

    /*其他的都是错误 */
.               yyerror("无效字符")。

%%

int yywrap(void) {
    return 1;
}
```

yacc的输入规范如下。**INTEGER**和**VARIABLE**的标记被yacc用来在**y.tab.h**中创建**#defines**，以便在lex中使用。接下来是算术运算符的定义。我们可以指定**%left**，表示左联运算，或
%right，表示右联想。最后列出的定义具有最高的优先权。因此，乘法和除法比加法和减法的优先级高。所有四个运算符都是左联动的。使用这个简单的技术，我们就能够消除我们的语法。

```
%token INTEGER VARIABLE
左边的'+''-'。
%left '*' '/'

%{
    int sym[26];
%}
```

```
%%
```

方案。

```
        程序语句' (n) '
        |
        ;
```

声明。

```
        expr{ printf("%d\n", $1); }
        | VARIABLE '=' expr{ sym[1] = 3; }
        ;
```

阐述。

```
        INTEGER
        | VARIABLE{ $$ = sym[1]; }
        | expr '+' expr{ $$ = 1 + 3; }
        | expr '-' expr{ $$ = 1 - 3; }
        | expr '*' expr{ $$ = 1 * 3; }
        | expr '/' expr{ $$ = 1 / 3; }
        | '(' expr ')'            { $$ = 2; }
        ;
```

```
%%

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

int main(void) {
    yyparse();
    return 0;
}
```

4. 计算器

4.1 描述

这个版本的计算器比以前的版本要复杂得多。主要变化包括控制结构，如*if-else*和*while*。此外，在解析过程中还构建了一个语法树。在解析之后，我们会在语法树上行走以产生输出。我们提供了两个版本的树形行走程序。

- 一个解释器，在树状行走过程中执行语句，以及
- 一个为假设的基于堆栈的机器生成代码的编译器。

为了使事情更加具体，这里有一个示例程序。

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

与解释性版本的输出。

```
0
1
2
```

和输出的编译器版本。

```
    推动    0
    啪啪啪  x
L000。
    推动    x
    推      3
    compLT
    jz L001
    推      x
    打印 推 x
    推动    1
    添加
    啪啪啪  x
    jmp     L000
L001:
```

include文件包含语法树和符号表的声明。符号表，**sym**，允许使用单字符的变量名。语法树中的一个节点可以容纳一个常数（**conNodeType**）、一个标识符（**idNodeType**），或者一个带有操作符的内部节点（**OPRNodeType**）。联盟**nodeType**封装了所有这三种变体，**nodeType.type**被用来确定我们有哪种结构。

lex输入文件包含**VARIABLE**和**INTEGER**标记的模式。此外，还为**EQ**和**NE**等双字符运算符定义了标记。单字符运算符只是作为其本身返回。

yacc输入文件定义**YYSTYPE**，即**yylval**的类型，为

```
%union {
    int iValue;             /* 整数值 */
    char sIndex;            /* 符号表索引 */
    nodeType nPtr;          /* 节点指针 */
};
```

这导致在**y.tab.h**中生成以下内容： **typedef union**

```
{
    int iValue;             /* 整数值 */
    char sIndex;            /* 符号表索引 */
    nodeType nPtr;          /* 节点指针 */
YYSTYPE;
外来的YYSTYPE yylval。
```

常量、变量和节点可以在解析器的值栈中用**ylval**表示。注意类型定义

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

这将**expr**与**nPtr**绑定，并将**INTEGER**与**YYSTYPE**联盟中的**iValue**绑定。这是必须的，这样yacc才能生成正确的代码。例如，规则

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

应该产生以下代码。注意**yyvsp[0]**的地址是值堆栈的顶部，或与**INTEGER**相关的值。

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

单元减号运算符比二元运算符有更高的优先权，如下所示。

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

%nonassoc表示不隐含关联性。它经常与**%prec**一起使用，以指定规则的优先级。因此，我们有

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

表示该规则的优先级与令牌**UMINUS**的优先级相同。而且，按照上面的定义，**UMINUS**的优先级比其他运算符高。类似的技术被用来消除与if-else语句相关的歧义（参见If-Else歧义，第35页）。

语法树是自下而上构建的，当变量和整数被减少时，分配叶子节点。当遇到运算符时，会分配一个节点，并将以前分配的节点的指针作为操作数输入。随着语句的减少，**ex**被调用以对语法树进行深度优先的行走。由于语法树是自下而上构建的，深度优先的行走是按照节点最初被分配的顺序访问的。这导致运算符的应用顺序与解析过程中遇到的顺序一致。包括两个版本的**ex**，一个解释版本和一个编译版本。

4.2 列入文件

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* 常量 */ typedef
struct {
    nodeEnum type;              /* 节点的类型 */
    int value;                  /* 常数的值 */
} conNodeType;

/* 标识符 */ typedef
struct {
    nodeEnum type;              /* 节点的类型 */
    int i;                      /*下标到身份数组 */
} idNodeType;

/* 运营商 */ typedef
struct {
    nodeEnum type;              /* 节点的类型 */
    int oper;                   /* operator */
                                   int nops;/*操作数 */
    union nodeTypeTag *op[1];/*操作数（可扩展） */
} oprNodeType。

typeedef union nodeTypeTag {
    nodeEnum type;              /* 节点的类型 */
    conNodeType con;            /* 常量 */
    idNodeType id;              /*标识符 */
    oprNodeType opr;            /* 运营商 */
} nodeType;
```

外来的`int sym[26]`。

4.3 Lex输入

```
%{
#include <stdlib.h>
#include
"calc3.h"#include
"y.tab.h"
%}

%%

[a-z]        {
                yylval.sIndex = *yytext - 'a';
                return VARIABLE;
             }

[0-9]+       {
                yylval.iValue = atoi(yytext); 返回
                INTEGER。
             }

[-()<>=+*/;{}.] {
                返回 *yytext;
              }

">="             返回GE。
"<="             返回LE。
"=="             返回EQ。
"！="            返回NE。
"while "         返回WHILE。
"如果 "          返回IF。
"else "          返回ELSE。
"打印 "          返回PRINT。

[ \t\n+]         ;       /* 忽略空白 */

.                yyerror("未知字符")。
%%
int yywrap(void) {
    return 1;
}
```

4.4 Yacc输入

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calc3.h"

/*原型 */
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i)。
nodeType *con(int value);
void freeNode(nodeType *p)。

void yyerror(char *s);
int sym[26];                    /* 符号表 */
%}

%union {
    int iValue;                 /* 整数值 */
    char sIndex;                /* 符号表索引 */
    nodeType *nPtr;             /* 节点指针 */
};

%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE

%left GE LE EQ NE '>' '<'。
左边的'+''-'。
%left '*' '/'
%nonassoc UMINUS

%type <nPtr> stmt expr stmt_list

%%
```

方案。

```
        函数'.'                 { exit(0); }
        ;
```

功能。

```
          函数 stmt{ ex(2美元); freeNode(2美元); }
        | /* 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。
          讲话                  { $$   $1; }
        | stmt_list 讲话        { $$   opr(';', 2, $1, $2); }
        ;
```

阐述。

```
          INTEGER               { $$   con(1美元);
                                       }
        | 变量                  { $$   id($1); }
        | '-' expr %prec 姆尼乌 { $$   opr(UMINUS  1, $2); }
                         斯             。
        | expr '+' expr         { $$   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         { $$   opr('>', 2, $1, $3); }
        | 符号化的GE符号化      { $$   opr(GE, 2, 1, 3); } }
          的EXR
        | 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; }
        ;

%%
```

```
nodeType *con(int value) {
    nodeType *p;

    /* 分配节点 */
    如果((p = malloc(sizeof(conNodeType))) == NULL)
        yyerror("out of memory")。

    /* 复制信息 */ p->type
    = typeCon;
    p->con.value = value。

    返回p。
}

nodeType *id(int i) {
    nodeType *p;

    /* 分配节点 */
    如果((p = malloc(sizeof(idNodeType))) == NULL)
        yyerror("out of memory")。

    /* 复制信息 */ p->type
    = typeId。
    p->id.i = i。

    返回p。
}

nodeType *opr(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    size_t size;
    int i。

    /* 分配节点 */
    size = sizeof(OPRNodeType) + (nops - 1) * sizeof(nodeType*);
    if ((p = malloc(size)) == NULL)
        yyerror("out of memory")。

    /* 复制信息 */ 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);
    返回p。
}
```

```
void freeNode(nodeType *p) {
    int i;

    if (!p) return;
    如果(p->type == typeOpr) {
        for (i = 0; i < p->opr.nops; i++)
            freeNode（p->opr.op[i]）。
    }
    免费(p)。
}

void yyerror(char *s) {
    fprintf(stdout, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

4.5 翻译人员

```
#include <stdio.h>
#include
"calc3.h"#include
"y.tab.h"

int ex(nodeType *p) {
    if (!p) return 0;
    switch(p->type) {
    caseCon:  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]);
                    返回0。
        case IF: 如果(ex(p->opr.op[0]))
                        ex(p->opr.op[1]);
                    else if (p->opr.nops >
                    2)
                        ex(p->opr.op[2]);
                    返回0。
        case PRINT: printf("%d\n", ex(p->opr.op[0]));
                    return 0;
        case ';':    ex(p->opr.op[0])。
                    返回 ex(p->opr.op[1])。
        case '=':    return sym[p->opr.op[0]->id.i] =
                        ex(p->opr.op[1]);
        case UMINUS: 返回 -ex(p->opr.op[0])。
        case '+':    return ex(p->opr.op[0]) + ex(p->opr.op[1]) 。
        case '-':    返回 ex(p->opr.op[0]) - ex(p->opr.op[1])。
        case '*':    返回 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]) 。
        op[0]) <=    ex(     p->opr.    op[   1]); case NE:   返
        回 ex(p->opr.op[0])!
        }
    }
}
```

4.6 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])。
            如果（p->opr.nops > 2） {
                /* 如果其他 */
                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 {
                /* 如果 */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
                ex（p->opr.op[1]）。
                printf("L%03d:\n", lbl1)。
            }
            突破。
```

```
        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;
        默认情况下。
            ex(p->opr.op[0])。
            ex(p->opr.op[1]);
            switch(p->opr.oper) {
            案例  '+':   printf("\tadd\n"); break;
            案例  '-':   printf("\tsub\n"); break;
            案例  '*':   printf("\tmul\n"); break;
            案例  '/':   printf("\tdiv\n"); break;
            案例  '<':   printf("\tcompLT\n"); break;
            案例  '>':   printf("\tcompGT\n"); break;
            案例  通用电  printf("\tcompGE\n"); break;
                  气。
            案例  LE。    printf("\tcompLE\n"); break;
            案例  NE。    printf("\tcompNE\n"); break;
            案例  EQ。    printf("\tcompEQ\n"); break;
            }
        }
    }
}
```

5. 更多莱克斯

5.1 弦乐

带引号的字符串经常出现在编程语言中。下面是在lex中匹配字符串的一种方法。

```
%{
    char *yylval;
    #include
    <string.h>.
%}
%%

          yylval = strdup(yytext+1);
          if (yylval[yyleng-2] != '"'
          )
              warning("字符串终止不当"); else
              yylval[yyleng-2] = 0;
          printf("found '%s'\n", yylval);
      }
```

上面的例子确保了字符串不跨越行的边界，并删除了包围的引号。如果我们希望添加转义序列，如 \n 或 \"，开始状态会简化问题。

```
%{
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; }
```

独有的开始状态**STRING**是在定义部分定义的。当扫描器检测到一个引号时，**BEGIN**宏将lex转移到**STRING**状态。Lex保持在**STRING**状态，只识别以**<STRING>**开始的模式，直到另一个**BEGIN**

被执行。因此，我们有一个扫描字符串的小环境。当尾部的引号被识别时，我们就切换回状态0，即初始状态。

5.2 保留字词

如果你的程序有大量的保留字集合，让lex简单地匹配一个字符串，并在自己的代码中确定它是一个变量还是保留字，会更有效率。例如，与其编码

```
"如果 "          返回IF。
"then "          返回THEN。
"else "          返回ELSE。

{letter}({letter}|{digit})* { yylval.id
         = symLookup(yytext); return
         IDENTIFIER;
     }
```

其中**symLookup**返回一个符号表的索引，最好是同时检测保留字和标识符，如下所示。

```
{字母}({letter}|{digit})* { int
         i;

         如果((i = resWord(yytext)) != 0) 返回
             (i)。
         yylval.id = symLookup(yytext);
         return (IDENTIFIER);
     }
```

这种技术大大减少了所需的状态数量，并使扫描器表更小。

5.3 调试Lex

Lex有启用调试的设施。这个功能可能随lex的不同版本而变化，所以你应该查阅文档以了解细节。lex在文件**lex.yy.c**中生成的代码包括调试语句，通过指定命令行选项"**-d**
"启用。调试输出可以通过设置**yy_flex_debug**来切换开和关。输出包括应用的规则和相应的匹配文本。如果你同时运行lex和yacc，在你的yacc输入文件中指定以下内容。

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

另外，你也可以通过定义函数来编写自己的调试代码，显示令牌值和**yylval**联盟的每个变量的信息。这在下面的例子中有所说明。当定义**DEBUG**时，调试函数生效，并显示令牌和相关值的跟踪。

```
%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) 。
            }

"如果"       RETURN(IF)。
"else"      RETURN(ELSE)。
```

6. 更多Yacc

6.1 递归

当指定一个列表时，我们可以使用左递归来进行。

```
列表。
        项目
        | 列表 ',' 项目
        ;
```

或右侧递归。

```
列表。
        项目
        | 项目 ',' 列表
```

如果使用右递归，列表上的所有项目都被推到堆栈上。在最后一个项目被推入后，我们开始减少。使用左递归，我们的堆栈上永远不会有超过三个项，因为我们是边走边减。由于这个原因，使用左递归是很有利的。

6.2 If-Else歧义

一个经常发生的shift-reduce冲突涉及*if-else*结构。假设我们有以下规则。

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

和以下状态。

```
IF expr stmt IF expr stmt .ELSE stmt
```

我们需要决定是转移**ELSE**，还是减少栈顶的**IF expr stmt**。如果我们转移，那么我们有

```
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 .
```

其中第二个**ELSE**与第二个**IF**配对。如果我们减少，我们有

```
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 .
```

其中第二个**ELSE**是与第一个**IF**配对的。现代编程语言将一个**ELSE**与最近的未配对的**IF 配 对**，所以前者的行为是可以预期的。这在yacc中很有效，因为当遇到shift-reduce冲突时，默认行为是shift。

尽管yacc做了正确的事情，但它也发出了shift-reduce警告信息。要删除该信息，给**IF-ELSE**一个比简单的**IF**语句更高的优先级。

```
%nonassoc IFX
%nonassoc ELSE

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

6.3 错误信息

一个好的编译器会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信息。

```
语法错误
```

如果我们在lex中跟踪行号，那么我们至少可以给用户一个行号。

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

当yacc发现一个解析错误时，默认动作是调用**yyerror**，然后从**yylex**返回，返回值为1。一个更优雅的动作是将输入流冲到一个语句定界符上，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 '}'
        | 错误 ';'
        |错误 '}'
        ;
```

错 误 标记是yacc的一个特殊功能，它将匹配所有的输入，直到找到 **错 误** 后的标记。在这个例子中，当yacc检测到语句中的错误时，它将调用**yyerror**，刷新输入到下一个分号或大括号，然后继续扫描。

6.4 继承的属性

到目前为止的例子都使用了*合成的*属性。在语法树的任何一点，我们都可以根据一个节点的子节点的属性来确定该节点的属性。考虑一下规则

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

由于我们是自下而上的解析，两个操作数的值都是可用的，我们可以确定与左侧相关的值。一个节点的*继承*属性取决于一个父节点或同级节点的值。下面的语法定义了一个C语言的变量声明。

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

下面是一个解析的样本。

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

当我们把**VAR**减少到**varlist** **时**，我们应该在符号表上注解变量的类型。然而，该类型被埋在堆栈中。这个问题可以通过索引回到栈中来解决。回顾一下，**1** **美** **元**指定了右手边的第一个项。我们可以向后索引，使用**$0**、**$-1**，以此类推。在这种情况下，**$0**就可以了。如果你需要指定一个符号类型，语法是**$<tokentype>0**，包括角括号。在这个特殊的例子中，必须注意确保 **类 型** *总是*在**varlist**之前。

6.5 嵌入行动

yacc中的规则可能包含嵌入式动作。

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

注意，行动在堆栈中占用一个槽，所以**do_item2**必须用**3美元**来引用**项2**。实际上，这个语法被yacc转换为以下内容。

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

6.6 调试Yacc

Yacc有能够进行调试的设施。这个功能可能随yacc的不同版本而变化，所以你应该查阅文档以了解细节。yacc在文件**y.tab.c**中生成的代码包括调试语句，通过定义**YYDEBUG**并将其设置为非零值来启用。这也可以通过指定命令行选项"**-t**"来实现。正确设置**YYDEBUG**后，可以通过设置**ydebug**来切换调试输出。输出包括扫描的令牌和shift/reduce动作。

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

此外，你可以通过指定命令行选项"**-v**"来转储解析状态。状态被转储到文件**y.output**中，在调试语法时通常很有用。另外，你也可以通过定义**TRACE**宏来编写自己的调试代码，如下图所示。当定义了**DEBUG**时，就会按行号显示减少的痕迹。

```
%{
#ifdef DEBUG
#define TRACE printf("reduce at line %d\n", 线）。
#else
#define TRACE
#endif
%}

%%

statement_list。
              声明
                  { 追踪$$=1; }
        | 声明_列表 声明
                  { TRACE $$ = newNode('; ', 2, $1, $2); }
        ;
```

7. 书目

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman [1986].编译器，原理，技术和工具。Addison-Wesley, Reading, Massachusetts.

Gardner, Jim, Chris Retterath and Eric Gisin [1988].MKS Lex & Yacc.Mortice Kern系统公司，加拿大安大略省滑铁卢市。

Johnson, Stephen C. [1975].Yacc: Yet Another Compiler Compiler.计算科学技术报告第32号，贝尔实验室，Murray hill，新泽西。

Lesk, M. E. and E. Schmidt [1975].Lex - 词汇分析器生成器。计算科学技术报告第39号，贝尔实验室，默里山，新泽西。

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

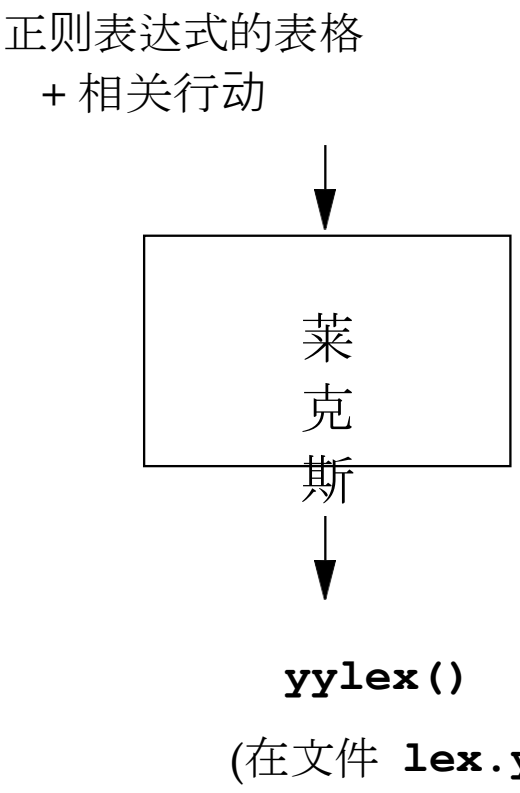
Lex和Yacc：一个轻快的教程

Saumya K. Debray 计算机科学系

亚利桑那大学图森分校, AZ 85721

莱克斯。一个扫描器发生器

- 帮助编写程序，其控制流是由输入流中的正则表达式指导的。



- `yylex()` 。
  - 将输入流与提供的正则表达式表进行匹配。

- 当发现匹配时，执行相关动作。

Lex规范文件的结构

定义

%%

规则

%%

用户子程序

红色：需要
蓝色：可选的

规则：以线为导向。

(*reg. exp*) (*whitespace*) (*action*)

(*reg. exp*)：从行首开始，一直到第一个未转义的空白处。

(*行动*)：一个单一的C语句
(多个语句：用大括号{ }括起来)。

未匹配的输入字符：复制到stdout。

Lex正则表达式

与egrep相似。

- *运算符*：" [] ^ - ?* | ( ) $ / { } % < >
- 字母和数字自己匹配
- 句号'.'匹配任何字符（除换行外）。
- 大括号[ ]中包含了一连串的字符，被称为 *字符类*。这符合。
  - 序列中的任何字符
  - 字符类中的'-'表示一个包容性的范围，例如：。[0-9] 匹配任何数字。
  - 开头的 "^"表示*否定*。[^0-9] 匹配任何非数字的字符。
- 带引号的字符""与该字符相匹配。操作符可以

  通过\转义。
- \n, 匹配新行，tab。

- 小括号 () 分组栏
  | 替代品
  棕鸟 零次或多次出现
  + 一次或多次发生
  ? 零次或一次发生

Lex规则的例子

- `intprintf ("keyword: INTEGER\n")`。
- `[0-9]+ printf("number\n")`。
- `"-"? [0-9]+("."[0-9]+)? printf("number\n")`。

在不同的可能匹配之间进行选择。

当有一个以上的模式可以匹配输入时，lex会按以下方式进行选择。

1. *最长的*匹配是首选。
2. 在匹配相同数量的字符的规则中，在列表中最早出现的规则是首选。

举例说明：模式

```
"/""*"(.|\n)*"*""/"
```

(旨在匹配多行注释)可能会消耗所有的输入!

与用户程序进行交流

`yytext` ：一个字符数组，包含匹配模式的实际字符串。

`yyleng` ：匹配的字符数。

例子：

- `[a-z][a-z0-9_]* printf("ident: %s\n", yytext)`。
- 计算文件中的字数和它们的总大小。

  `[a-zA-Z]+{nwords += 1; size += yyleng;}`。

Lex来源的定义

- 任何没有被lex拦截的来源都被复制到生成的程序中。
  - 不属于词条规则或动作的一行，以空白或制表符开头的，按上述方法复制出来（对全局声明等很有用）。
  - 任何包含在只包含%{和%}如上所述被复制出来（例如，对于必须以col.1开头的预处理器语句很有用）
  - 在第二个%%分界符之后的任何内容都会在lex输出中被复制出来（对本地函数定义很有用）。

- 用于lex的定义在第一个%%之前给出。本节中任何没有以空白或制表符开始的行，或者没有被%{...%}括起来的行，都被认为是在定义一个lex替换字符串，其形式为

  名称　翻译

  例如：.....。

  字母　　[a-zA-Z]

一个例子

```
%{
#include
"tokdefs.h"#include
<strings.h>
static int id_or_keywd(char *s);
%}

字母            [a-zA-Z]
数字            [0-9]
阿尔法          [a-zA-Z0-9_]
whitesp         [ \t\n] 。
%%
{whitesp}*            ;
{pos(192,240)}评论    ;
{字母}{alfa}REPORT    (id_or_keywd(yytext), yytext)。
   ...
%%
静态结构 { char
  *name; int
  val;
} keywd_entry,
  keywd_table[] = {
    "char",           CHAR。
    "int",            INT。
    "while",          WHILE,
        ...
   };
```

```
static int id_or_keywd(s)
char *s;
{
  ...
}
```

左边的上下文敏感度。开始条件

*启动条件*是一种有条件地激活补丁的机制。这对于处理

- 在概念上不同的输入成分；或
- 在lex默认值（如 "最长可能的匹配"）不能很好工作的情况下，如注释或带引号的字符串。

基本理念。

- 用以下方法声明一组*起始条件的名称*

  %起始*名称*$_1$ *名称*$_2$ . . .

- 如果*scn*是一个起始条件名称，那么一个前缀为 < *scn*>只有在扫描仪处于启动状态*scn*时才会被激活。
- 扫描器从起始条件INITIAL开始，所有非< *scn*>前缀的规则都是其成员。
- 像这样的开始条件是*包容性的*：即在该开始条件中，会将适

当的前缀规则添加到活动规则集。

flex也允许*排他性*启动条件（用%x），这有时更方便。

使用启动条件的例子

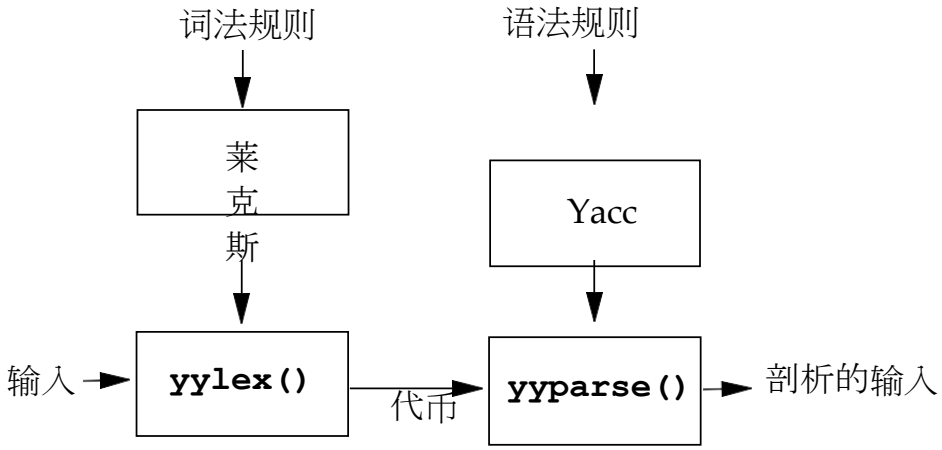
```
%开始评论0评论1
%{
#include "tokens.h"
%}
whitespace      [\t\n] 。
数字            [0-9]
符号            {位数}+
漂泊            {digit}+"."{digit}+
start_comment "/""*"

%%
<INITIAL>{start_comment}。   BEGIN(comment0)。
<comment0>"*"                BEGIN(comment1);
<评论0>[^*]。                ;
<comment1>"*"                ;
<comment1>"/"                begin(initial);
<comment1>[^*/]              BEGIN(comment0)。

{pos(192.}                   return(INTCON)。
{floatcon}.                  return(FLOATCON)。
  ...
```

%%

. . .

Yacc：一个解析器生成器

- 接受一个CFG的规范，产生一个LALR解析器。



- yacc规范文件的形式。

申报

%%

语法规则

%%

节目

红色：需要

蓝色：可选的

Yacc: 语法规则

终端(tokens):必须声明名称。

%token *名称*$_1$ *名称*$_2$ . . .

任何没有在声明部分被声明为标记的名字都被认为是一个非终端。

启动符号:

- 可以通过以下方式声明：%开始*名称*
- 如果没有明确声明，则默认为列出的第一条语法规则的LHS上的非终端。

生产。一个语法生产$A \rightarrow B_1\ B_2$ - - - B_n，写成：

$a : b_1\ b_2$ - - b_n 。

注意：出于效率的考虑，左重传比右重传更受欢迎。

例子。

```
stmt : KEYWD_IF '(' expr ')' stmt ;
```

扫描仪和分析器之间的通信

- 用户必须提供一个整数值的函数 `yylex()`
  实现词法分析器（扫描器）。

- 如果有一个与令牌相关的值，它应该被分配给外部变量 `yylval`。

- 错误令牌是为错误处理保留的。

- *代号*：如果需要的话，可以由用户选择。默认情况是。
  - 由yacc选择[在文件`y.tab.h`中]
  - 一个字的标记号是其ASCII值
  - 其他代币的编号从257开始。
  - 该标记的数字必须为零或负数。

- 使用'`yacc -d`'生成`y.tab.h`

使用Yacc

- 假设语法规范在一个文件foo.y中，那么。
  - 命令'yacc foo.y'产生一个文件y.tab.c，其中包含yacc构建的分析器。
  - 命令'yacc -d foo.y'构建了一个文件y.tab.h，它可以被#include到lex生成的扫描器中。
  - 命令'yacc -v foo.y'会额外构建一个包含解析器描述的文件y.output（对调试很有用）。
- 用户需要提供一个函数main()给驱动程序，以及一个函数yyerror()，如果输入中出现错误，该函数将被分析器调用。

矛盾和含糊不清

- 冲突可能是*移位/减法*或*减法/减法*。
  - 在shift/reduce冲突中，默认是shift。
  - 在reduce/reduce冲突中，默认是*使用第一个适用的语法规则进行还原*。
- *算术运算符*：可以指定关联性和优先级。

  关联性：使用 `%left`, `%right`, `%nonassoc`

  优先级（二进制运算符）。
  - 使用`%left`等指定关联性。
  - 一个组内的操作符具有相同的优先权。在组与组之间，优先级往下递增。

  优先级（单数运算符）：使用`%prec`关键字。这将改变一个规则的优先级，使其成为以下标记的优先级。

  例子。

```
%left '+' '-'
%left '*' '/'
    ...
```

```
expr : expr '+' expr
     | expr '*' expr
     | '-' expr        %prec '*'
     | 身份证
```

Yacc: 错误处理

- 错误标记是为错误处理保留的。这可以在语法规则中使用，以指示可能发生错误的地方并进行恢复。

- *当检测到一个错误时*:
  - 如果指定了一个错误标记，解析器会弹出其堆栈，直到找到一个错误标记合法的状态。

    然后它的行为就像`error`是当前的lookahead标记一样，并执行遇到的动作。
  - 如果没有使用错误标记的规则，当遇到错误时，处理将停止。

- 为了防止串联错误信息，解析器在检测到错误后一直处于*错误状态*，直到有3个标记被成功读取和移位。

  如果在错误状态下遇到错误，则不给出错误信息，并丢弃输入令牌。

Yacc错误处理：（续）。

- 一个形式的规则

  ```
  stmt : 错误
  ```

  意味着在语法错误时，解析器将试图跳过违规的语句，寻找可以合法地跟在stmt后面的3个标记。

- 一个形式的规则

  ```
  stmt : 错误 ';'
  ```

  导致解析器跳到stmt之后的下一个';'：所有中间的标记被删除。

- 行动可能与这些特殊的错误规则有关：这些行动可能试图（重新）初始化表，回收空间，关闭代码生成，等等。

添加错误符号

他们的安置是由以下（相互冲突的）目标指导的。

- 尽可能地接近语法的起始符号

  (以便在不丢弃整个项目的情况下进行恢复)

- 尽可能地靠近每个终端符号

  (只允许少量的输入因错误而被取消)

- 在不引入冲突的情况下（这

  可能是困难的。

  如果shift/reduce冲突有助于延长字符串，即延迟错误的报告，则可以接受)

错误信息

使用应该提供一个函数yyerror()，当检测到语法错误时被调用。

```
yyerror(s)
char *s; /* s: 一个包含错误味精的字符串 */
{/*通常是 "语法错误" */
  ...
}
```

更多信息性的错误信息：

– 源程序中的行号。 yylineno
– 导致错误的令牌号。 yychar

例子：

```
外部int yylineno, yychar; yyerror(s)
char *s;
{
```

```
  fprintf(stderr,
          "%s: token %d on line %d\n"。
             /*       ^^ Ugh: 内部令牌          不！？
          s, yychar, yylineno) 。
}
```

控制错误行动

有时我们可能想停止丢弃令牌，如果看到某个（同步）令牌：为此，附加一个动作{yyerrok; }。

例子：

```
id list : id list ',' ID { yyerrok; }
        | 身份证
        | 错误
```

特殊用途的错误处理。

- 设置一个全局标志，以表明该问题。
- 在yyerror()中使用这个标志，可以给出更好的错误信息。

例子：

```
compd_stmt : '{' stmt_list '}' 。
```

```
    | '{' stmt_list error {errno = NO_RBRACE; }
    | '{' 错误 '}'
   ...
yyerror(s)
{
    如果 (errno == NO_RBRACE)  printf("missing }\n");
     else ...
}
```

向Yacc规范添加语义动作

语义行动可与每个规则相关联。

一个动作是由一个或多个语句指定的，用大括号{ }括起来。

例子：

```
ident decl : ID     {install symtab(id name); }

type decl : type { tval = ...} id list
```

注意：行动可能发生在生产的RHS内部。

(合成的）非终结者的属性

每个非终端可以返回一个值。

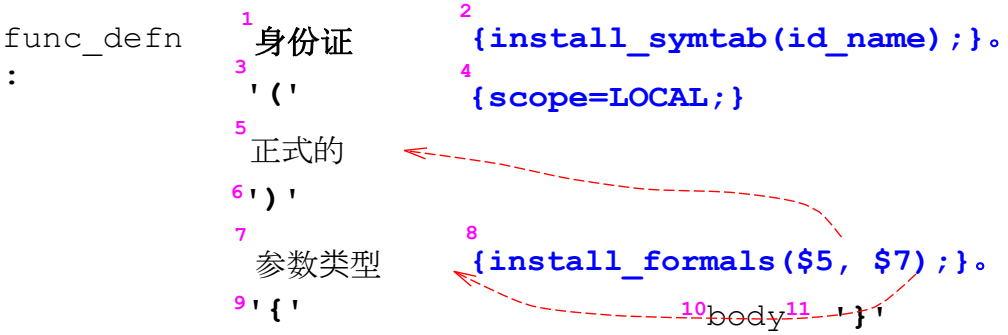
- 要访问规则主体中i^{th} 符号返回的值，使用`$i`。

  如果一个动作发生在一个规则的中间，它被算作一个可以返回一个值的 "符号"。

- 要设置一个规则要返回的值，分配给**$$**。

  默认情况下，一个规则的值是其中第一个元素的值（1美元）。

例子：



```
12
                {scope = GLOBAL;
               symtab_cleanup();}
```

例子

```
var_decl
    : 标识符 opt_dimension
        { 如果( Seen(1)>0 ){ errmsg(MULTI_DECL,
            Ident(1))。
          }
          否则 {
            Seen($1) = DECL;
            如果（2美元<0） {
              BaseType(1美元) =
              tval;
            }
            否则 {
              BaseType(1) = ARRAY;
              ArrayEltType(1) = tval;
              ArrayDim(1) = 2;
            }
          }
        }
    ;

opt_dimension
```

```
: '[' INTCON ']'    { $$ = Value(yylval); }
|/* epsilon */ { $$ = -1; }
;
```

合成的属性。类型

默认情况下，动作和词法分析器返回的值是整数。

一般来说，动作可能需要返回其他类型的值，例如，指向符号表或语法树节点的指针。为此，我们需要

1. 声明可能被返回的各种值的联合。例如。

```
%union {
    symtab_ptr    st_ptr;
    id_list_ptr list_of_ids;
    tree_node     st_node; int
                  值。
}
```

2. 指定一个特定的非终端将返回哪个联盟成员。

```
%token <value> INTCON, CHARCON;
```
}终端

```
%type <st_ptr>标识符。
%type <list of ids>
formals;
```
非终结者

例子

```
%union {
    记忆中的 "小 "字。
    id列表ptr           id的列表。
    expr 节点 ptr       expr;
    stmt节点ptr         stmt;
    价值                值。
}

%token 身份证。                                  /* 标识符 */
%token <值> INTCON CHARCON                       /* 常数 */
%token CHAR INT VOID                             /* 类型 */
...
左边                    和或
%nonassocLE GE EQ NEQ '<' '>' 。
%left                   '+' '-'
...
%type <值               选择尺寸、类型
%type <st ptr>          标识符。
%type <列表中的id>      格式化
...

%开始程序

%{
st rec ptrid ptr;       /* globals */
id列表ptr               fparms。
```

```
...
%}
%%
...
```

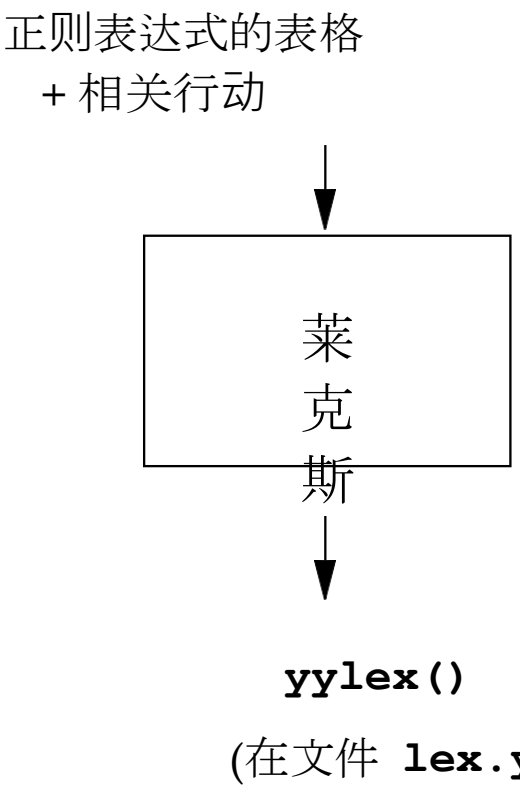
Lex和Yacc：一个轻快的教程

Saumya K. Debray 计算机科学系

亚利桑那大学图森分校, AZ 85721

莱克斯。一个扫描器发生器

- 帮助编写程序，其控制流是由输入流中的正则表达式指导的。



- `yylex()` 。
  - 将输入流与提供的正则表达式表进行匹配。

- 当发现匹配时，执行相关动作。

Lex规范文件的结构

定义

%%

规则

%%

用户子程序

红色：需要
蓝色：可选的

规则：以线为导向。

(*reg. exp*) (*whitespace*) (*action*)

(*reg. exp*)：从行首开始，一直到第一个未转义的空白处。

(*行动*)：一个单一的C语句
(多个语句：用大括号{ }括起来)。

未匹配的输入字符：复制到stdout。

Lex正则表达式

与egrep相似。

- *运算符*：" [] ^ - ?* | ( ) $ / { } % < >
- 字母和数字自己匹配
- 句号'.'匹配任何字符（除换行外）。
- 括号[ ]中包含了一连串的字符，称为一个*字符类*。这符合。
  - 序列中的任何字符
  - 字符类中的'-'表示一个包容性的范围，如：。[0-9]匹配任何数字。
  - 开头的 "^"表示*否定*。[^0-9]匹配任何非数字的字符。
- 带引号的字符""与该字符相匹配。操作符可以

  通过\转义。
- \n, 匹配新行，tab。

- 小括号 () 分组栏
  | 替代品
  棕鸟 零次或多次出现
  + 一次或多次发生
  ? 零次或一次发生

Lex规则的例子

- `intprintf  ("keyword: INTEGER\n")`。
- `[0-9]+ printf("number\n")`。
- `"-"? [0-9]+("."[0-9]+)? printf("number\n")`。

在不同的可能匹配之间进行选择。

当有一个以上的模式可以匹配输入时，lex会按以下方式选择。

1. *最长的*匹配是首选。
2. 在匹配相同数量的字符的规则中，在列表中最早出现的规则是首选。

举例说明：模式

```
"/""*"(.|\n)*"*""/"
```

(旨在匹配多行注释)可能会消耗所有的输入!

与用户程序进行交流

`yytext` ：一个字符数组，包含匹配模式的实际字符串。

`yyleng` ：匹配的字符数。

例子：

- `[a-z][a-z0-9_]* printf("ident: %s\n", yytext)`。
- 计算文件中的字数和它们的总大小。

  `[a-zA-Z]+{nwords += 1; size += yyleng;}`。

Lex来源的定义

- 任何没有被lex拦截的来源都被复制到生成的程序中。
  - 不属于词条规则或动作的行，以空白或制表符开头的，按上述方法复制出来（对全局声明等有用）。
  - 任何包含在只包含%{和%}如上所述被复制出来（例如，对于必须以col.1开头的预处理器语句很有用）
  - 在第二个%%分隔符之后的任何内容都会在lex输出中被复制出来（对本地函数定义很有用）。

- 用于lex的定义在第一个%%之前给出。本节中任何没有以空白或制表符开始的行，或者没有被%{...%}括起来的行，都被认为是在定义一个lex替换字符串，其形式为

  名称　翻译

  例如：.....。

  字母　　[a-zA-Z]

一个例子

```
%{
#include
"tokdefs.h"#include
<strings.h>
static int id_or_keywd(char *s);
%}

字母            [a-zA-Z]
数字            [0-9]
阿尔法          [a-zA-Z0-9_]
whitesp         [ \t\n] 。
%%
{whitesp}*          ;
{pos(192,240)}评论   ;
{字母}{alfa}REPORT   (id_or_keywd(yytext), yytext)。
   ...
%%
静态结构 { char
  *name; int
  val;
} keywd_entry,
  keywd_table[] = {
    "char",           CHAR。
    "int",            INT。
    "while",          WHILE,
        ...
   };
```

```
static int id_or_keywd(s)
char *s;
{
  ...
}
```

左边的上下文敏感度。开始条件

*启动条件*是一种有条件地激活补丁的机制。这对于处理

- 在概念上不同的输入成分；或
- 在lex默认值（如 "最长可能的匹配"）不能很好工作的情况下，如注释或带引号的字符串。

基本理念。

- 用以下方法声明一组*起始条件的名称*

  %起始*名称*$_1$ *名称*$_2$ . . .

- 如果*scn*是一个起始条件名称，那么一个前缀为 < *scn*>只有在扫描仪处于启动状态*scn*时才会被激活。
- 扫描器从起始条件INITIAL开始，所有非< *scn*>前缀的规则都是其成员。
- 像这样的开始条件是*包容性的*：即在该开始条件中，会将适

当的前缀规则添加到活动规则集。

flex也允许*排他性*启动条件（用%x），这有时更方便。

使用启动条件的例子

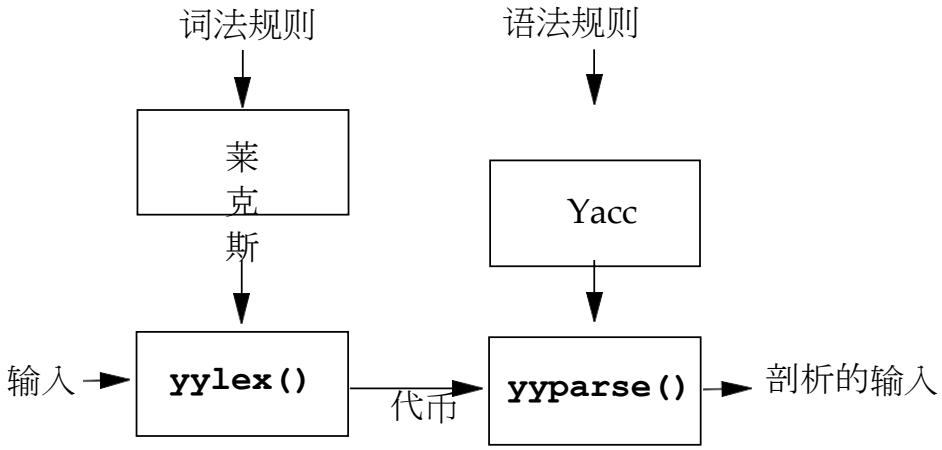
```
%开始评论0评论1
%{
#include "tokens.h"
%}
whitespace      [\t\n] 。
数字            [0-9]
符号            {位数}+
漂泊            {digit}+"."{digit}+
start_comment "/""*"

%%
<INITIAL>{start_comment}。   BEGIN(comment0)。
<comment0>"*"                BEGIN(comment1);
<评论0>[^*]。                ;
<comment1>"*"                ;
<comment1>"/"                begin(initial);
<comment1>[^*/]              BEGIN(comment0)。

{pos(192.}                   return(INTCON)。
{floatcon}.                  return(FLOATCON)。
  ...
```

%%

. . .

Yacc：一个解析器生成器

- 接受一个CFG的规范，产生一个LALR解析器。



- yacc规范文件的形式。

申报

%%

语法规则

%%

节目

红色：需要

蓝色：可选的

Yacc: 语法规则

终端(tokens):必须声明名称。

%token *名称*$_1$ *名称*$_2$ . . .

任何没有在声明部分被声明为标记的名字都被认为是一个非终端。

启动符号:

- 可以通过以下方式声明：%开始*名称*
- 如果没有明确声明，则默认为列出的第一条语法规则的LHS上的非终端。

生产。一个语法生产$A \rightarrow B_1\ B_2$ - - - B_n 写成：

$a : b_1\ b_2$ - - b_n 。

注意：出于效率的考虑，左重传比右重传更受欢迎。

例子。

```
stmt : KEYWD_IF '(' expr ')' stmt ;
```

扫描仪和分析器之间的通信

- 用户必须提供一个整数值的函数 `yylex()`
  实现词法分析器（扫描器）。

- 如果有一个与令牌相关的值，它应该被分配给外部变量 `yylval`。

- 错误令牌是为错误处理保留的。

- *代号*：如果需要的话，可以由用户选择。默认情况是。
  - 由yacc选择[在文件`y.tab.h`中]
  - 一个字的标记号是它的ASCII值
  - 其他代币的编号从257开始。
  - 该标记的数字必须为零或负数。

- 使用'`yacc -d`'生成`y.tab.h`

使用Yacc

- 假设语法规范在一个文件foo.y中，那么。
  - 命令'yacc foo.y'产生一个文件y.tab.c，其中包含yacc构建的分析器。
  - 命令'yacc -d foo.y'构建了一个文件y.tab.h，它可以被#include到lex生成的扫描器中。
  - 命令'yacc -v foo.y'会额外构建一个包含解析器描述的文件y.output（对调试有用）。
- 用户需要提供一个函数main()给驱动程序，以及一个函数yyerror()，如果输入中出现错误，该函数将被分析器调用。

矛盾和含糊不清

- 冲突可能是*移位/减法*或*减法/减法*。
  - 在shift/reduce冲突中，默认是shift。
  - 在reduce/reduce冲突中，默认是*使用第一个适用的语法规则进行还原*。
- *算术运算符*：可以指定关联性和优先级。

  关联性：使用 `%left`, `%right`, `%nonassoc`

  优先级（二进制运算符）。
  - 使用`%left`等指定关联性。
  - 一个组内的操作符具有相同的优先权。在组与组之间，优先级往下递增。

  优先级（单数运算符）：使用`%prec`关键字。这将改变一个规则的优先级，使其成为以下标记的优先级。

  例子。

```
%left '+' '-'
%left '*' '/'
    ...
```

```
expr : expr '+' expr
     | expr '*' expr
     | '-' expr        %prec '*'
     | 身份证
```

Yacc: 错误处理

- 错误标记是为错误处理保留的。这可以在语法规则中使用，以指示可能发生错误的地方并进行恢复。

- *当检测到一个错误时*:
  - 如果指定了一个错误标记，解析器会弹出其堆栈，直到找到一个错误标记合法的状态。

    然后它的行为就像`error`是当前的lookahead标记一样，并执行遇到的动作。
  - 如果没有使用错误标记的规则，当遇到错误时，处理将停止。

- 为了防止错误信息的级联，解析器在检测到错误后一直处于*错误状态*，直到有3个标记被成功读取和移位。

  如果在错误状态下遇到错误，则不给出错误信息，并丢弃输入令牌。

Yacc错误处理：（续）。

- 一个形式的规则

  ```
  stmt : 错误
  ```

  意味着在语法错误时，解析器将试图跳过违规的语句，寻找可以合法地跟在stmt后面的3个标记物。

- 一个形式的规则

  ```
  stmt : 错误 ';'
  ```

  导致解析器跳到stmt之后的下一个';'：所有中间的标记被删除。

- 行动可能与这些特殊的错误规则有关：这些行动可能试图（重新）初始化表，回收空间，关闭代码生成，等等。

添加错误符号

他们的安置是由以下（相互冲突的）目标指导的。

- 尽可能地接近语法的起始符号

  (以便在不丢弃整个项目的情况下进行恢复)

- 尽可能地靠近每个终端符号

  (只允许少量的输入因错误而被取消)

- 在不引入冲突的情况下（这

  可能是困难的。

  如果shift/reduce冲突有助于延长字符串，即延迟错误的报告，则可以接受)

错误信息

使用应该提供一个函数yyerror()，当检测到语法错误时被调用。

```
yyerror(s)
char *s; /* s: 一个包含错误味精的字符串 */
{/*通常是 "语法错误" */
  ...
}
```

更多信息性的错误信息：

– 源程序中的行号。 yylineno
– 导致错误的令牌号。 yychar

例子：

```
外部int yylineno, yychar; yyerror(s)
char *s;
{
```

```
  fprintf(stderr,
          "%s: token %d on line %d\n"。
             /*       ^^ Ugh: 内部令牌          不！？
          s, yychar, yylineno) 。
}
```

控制错误行动

有时我们可能想停止丢弃令牌，如果看到某个（同步）令牌：为此，附加一个动作{yyerrok; }。

例子：

```
id list : id list ',' ID { yyerrok; }
        | 身份证
        | 错误
```

特殊用途的错误处理。

- 设置一个全局标志，以表明该问题。
- 在yyerror()中使用这个标志，可以给出更好的错误信息。

例子：

```
compd_stmt : '{' stmt_list '}' 。
```

```
    | '{' stmt_list error {errno = NO_RBRACE; }
    | '{' 错误 '}'
   ...
yyerror(s)
{
    如果 (errno == NO_RBRACE)  printf("missing }\n");
     else ...
}
```

向Yacc规范添加语义动作

语义行动可与每个规则相关联。

一个动作是由一个或多个语句指定的，用大括号{ }括起来。

例子：

```
ident decl : ID     {install symtab(id name); }

type decl : type { tval = ...} id list
```

注意：行动可能发生在生产的RHS内部。

(合成的）非终结者的属性

每个非终端可以返回一个值。

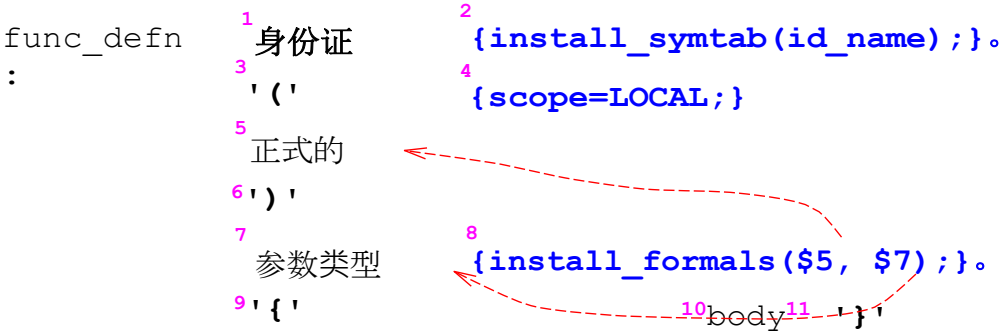
- 要访问规则主体中i^{th} 符号返回的值，使用`$i`。

  如果一个动作发生在一个规则的中间，它被算作一个可以返回一个值的 "符号"。

- 要设置一个规则要返回的值，分配给**$$**。

  默认情况下，一个规则的值是其中第一个元素的值（1美元）。

例子：



```
12
                {scope = GLOBAL;
               symtab_cleanup();}
```

例子

```
var_decl
    : 标识符 opt_dimension
        { 如果( Seen(1)>0 ){ errmsg(MULTI_DECL,
            Ident(1))。
          }
          否则 {
            Seen($1) = DECL;
            如果（2美元<0） {
              BaseType(1美元) =
              tval;
            }
            否则 {
              BaseType(1) = ARRAY;
              ArrayEltType(1) = tval;
              ArrayDim(1) = 2;
            }
          }
        }
    ;

opt_dimension
```

```
: '[' INTCON ']'    { $$ = Value(yylval); }
|/* epsilon */ { $$ = -1; }
;
```

合成的属性。类型

默认情况下，动作和词法分析器返回的值是整数。

一般来说，动作可能需要返回其他类型的值，例如，指向符号表或语法树节点的指针。为此，我们需要

1. 声明可能被返回的各种值的联合。例如。

```
%union {
    symtab_ptr    st_ptr;
    id_list_ptr list_of_ids;
    tree_node     st_node; int
                  值。
}
```

2. 指定一个特定的非终端将返回哪个联盟成员。

```
%token <value> INTCON, CHARCON;
```
}终端

```
%type <st_ptr>标识符。
%type <list of ids>
formals;
```
非终结者

例子

```
%union {
    记忆中的 "小 "字。
    id列表ptr           id的列表。
    expr 节点 ptr       expr;
    stmt节点ptr         stmt;
    价值                值。
}

%token 身份证。                                  /* 标识符 */
%token <值> INTCON CHARCON                       /* 常数 */
%token CHAR INT VOID                             /* 类型 */
...
左边                    和或
%nonassocLE GE EQ NEQ '<' '>' 。
%left                   '+' '-'
...
%type <值               选择尺寸、类型
%type <st ptr>          标识符。
%type <列表中的id>      格式化
...

%开始程序

%{
st rec ptrid ptr;       /* globals */
id列表ptr               fparms。
```

```
...
%}
%%
...
```

并非每一个固件开发工具最初都是为了这个目的而设计的。例如，Unix的lex和yacc，可以用来自动生成一些代码。

Lex和yacc是免费的开发工具，可以帮助你用C或C++编写解释或转换结构化输入的软件。解释结构化输入的应用可以从简单的文本搜索到完整的编译器，但也包括嵌入式应用，如串行协议和消息解释器。如果你正在开发一个操作ASCII数据的应用程序，lex和yacc可以为你的开发省去许多麻烦。本文将告诉你何时以及如何使用这些工具。

在一个解释结构化ASCII输入的程序中，通常要重复执行两项任务。首先，必须将输入从单个字符流转换为有意义的符号序列。这一操作被称为**扫描**。Lex是一个生成扫描器的有用工具。一旦你掌握了使用lex所需的基本知识，你会发现词法扫描器所需的规则输入比同等的C代码更容易编写。Lex还可以自动检查你的扫描器规格，以发现各种常见的错误。

解释结构化输入的第二步是识别扫描器发现的标记序列中的模式。这个过程被称为**解析**。Yacc可以生成解析器来执行这种模式匹配。(在70年代，不乏编写解析器生成工具的团体，这有助于解释yacc这个名字的由来，它代表着"又一个编译器的编译器"）。就像扫描仪一样，解析器一直都是手动生成的，但是yacc生成的解析器可以带来更强大的代码，更少的调试，以及易于阅读和维护的输入文件（yacc语法）。

图1显示了一个典型的扫描器/解析器应用中的控制和数据流。数据流被扫描器还原为标记，标记在解析器中积累，解析器识别标记的有效组合。

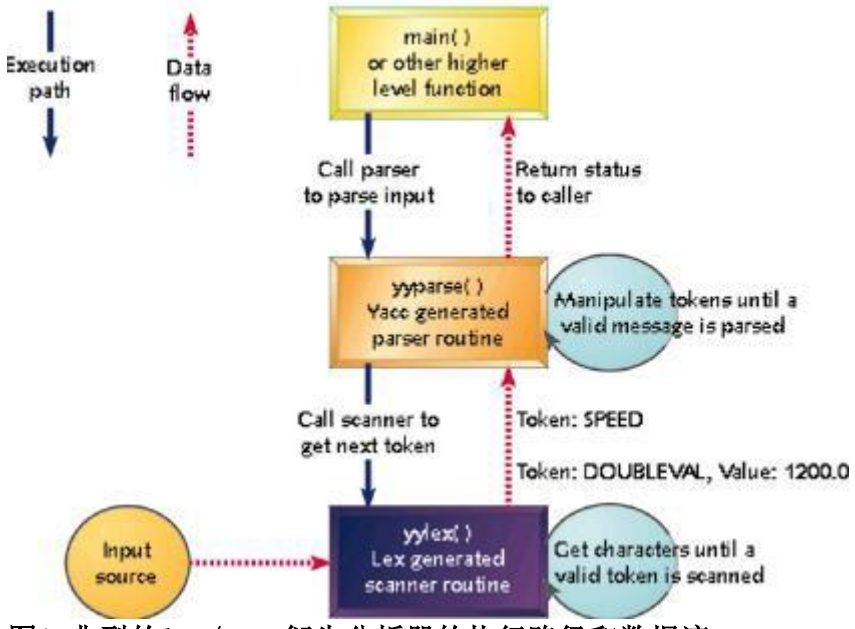


图1:典型的lex/yacc衍生分析器的执行路径和数据流

图2说明了基于lex-/yacc的应用程序的开发过程。许多简单的应用，如文本搜索，只需要在输入流中寻找标记。这种应用可以单独使用lex来创建。同样，如果输入流中的标记排列很简单，一个简单的手工编码的分析器可能就足够了。然而，随着标记的可能排列和组合的增加，我们通常需要将扫描器的输出（标记流）指向一个单独的yacc生成的分析器的输入。

在学习如何实际使用lex和yacc之前，我们先问一下我们想用它们做什么，为什么？正如我前面提到的，词法扫描器和分析器的嵌入式应用种类繁多。我们可以把这些可能性分成两大领域：开发工具和目标代码。

我所说的开发工具是指驻留在开发主机上的自定义应用程序，它帮助你产生运行时的代码，就像你可能使用的任何其他编译器或代码生成工具一样。目标代码指的是C或C++代码，我们最终将其与应用程序的其他部分编译和链接，以便在我们的目标硬件上执行。本文的重点是使用lex和yacc来生成目标代码。

嵌入使用

串行协议是嵌入式软件系统的一个常见组成部分。也许一个系统必须通过一个串行端口在两个独立的微控制器之间提供通信。使用双向串行协议需要在接口的两端都有一个协议解释器，这可以用lex和yacc来创建。

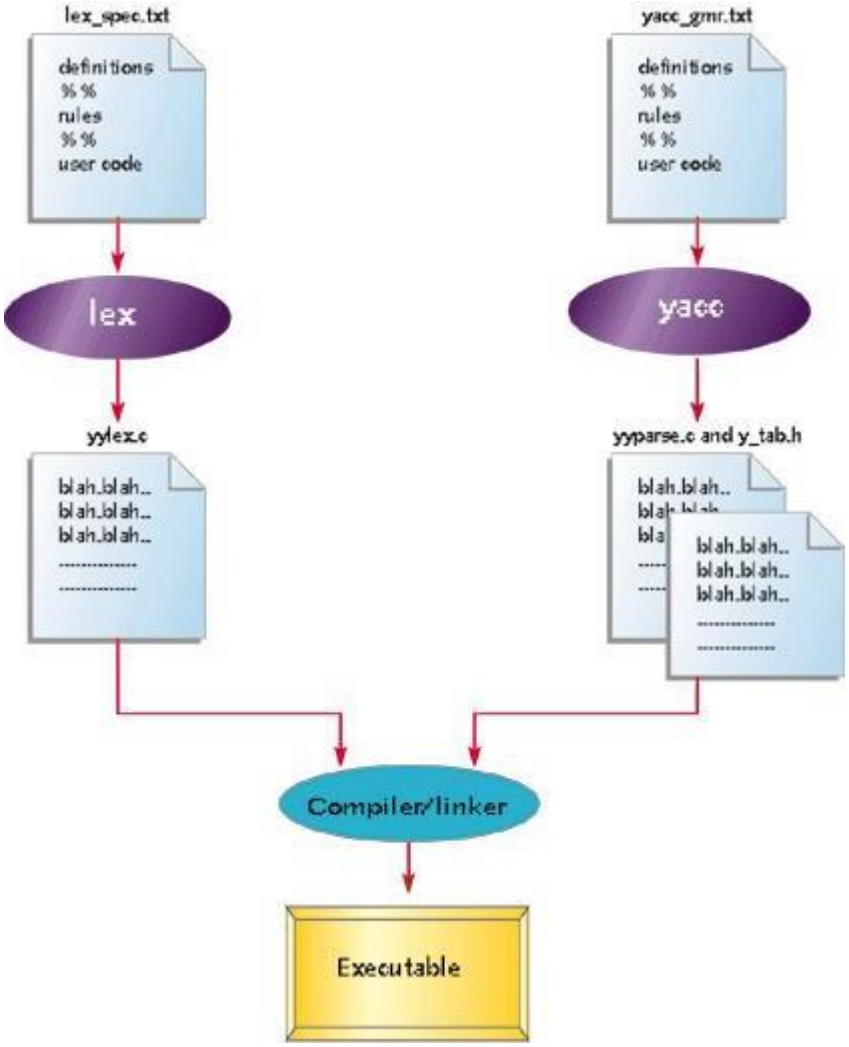


图2：使用lex和yacc时的开发顺序

我们可以使用lex和yacc的另一个地方是在两个软件组件之间解释结构化消息。结构化消息在多任务环境中很常见，其中一个数据块必须通过某种操作系统服务（如消息队列）从一个任务传递到另一个任务。

考虑两个独立任务希望发送和接收数据块的情况。如果这两个任务位于相同的硬件上，并且使用相同的编译器（具有相同的设置）进行编译，那么简单地将一个结构体或对象塞入一个消息（换句话说，一个内存块）并将这个消息传递给接收任务是很方便的。假设接收者知道要期待什么，它可以简单地使用适当类型的指针访问数据。

但在许多情况下，任务并不想假设接收者以发送者的方式存储数据结构。这可能是由于可移植性的要求，也可能是由于任务是属于一个

分布式应用并在不同的硬件上运行。在这种情况下，通常会提供一个消息，以商定的格式描述数据，而不是传递二进制。在这种情况下，你需要在接收端对描述进行解析。

许多嵌入式系统需要配置数据，这些数据通常存储在闪存文件系统中。这些配置数据通常存储在一个文本文件中，该文件在运行时被解析，并被用来填充一个数据库，其中的数值会影响应用程序的后续行为。我在最宽泛的意义上使用数据库这个术语。数据存储可能包括任何东西，从几个变量到一组大型复杂的数据结构。让我们看几个例子，看看如何做到这一点。有许多版本的lex和yacc可用于不同的平台。在下面的例子中，我使用了GNU的flex 2.5和bison 1.25。

字符串处理

图3显示了一个由三个关键部件组成的电机控制系统。

- 一种用于精确控制被动轴速度和位置的电动伺服电机
- 一个数字伺服电机控制系统（驱动系统）来执行电机的模拟控制
- 一个微控制器，用于监督控制驱动装置的电子元件和从驱动装置获取数据

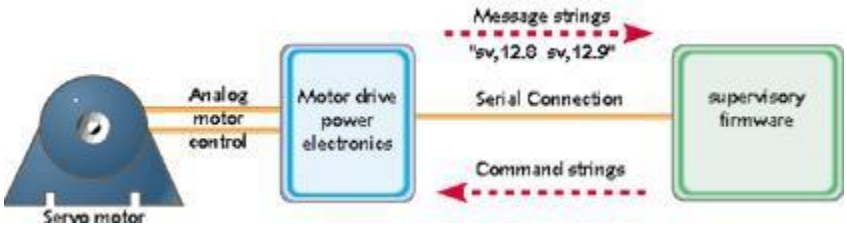


图3：电机控制器和监控微控制器配置驱动器

运行在微控制器上的固件通过RS-232电缆向驱动系统发送命令并接收响应。我们在这里只关心这个数据流的一个组成部分：从驱动器传来的数据。这些数据以字符流的形式出现；一个lex生成的扫描器对其进行解释，并生成可在应用程序内部使用的信息。我们假设我们只是在寻找一条信息。

"sv,"

它表示目前的轴旋转速度，以任意单位表示。它可以根据一些微控制器的输出而产生，也可以由驱动系统异步产生。这并不影响扫描仪对数据的解释。由于其简单性，这个应用不需要使用yacc。我们将在我们的扫描器函数中完成所有的工作，并为应用程序生成消息，而不是将标记返回给分析器，制作我们的扫描器的过程如下。

- 写一个lex规范文件，描述我们希望我们的扫描仪做什么。
- 使用lex编译我们的规范。
- 编译并将我们生成的C或C++扫描器模块与我们的固件的其他部分连接起来。

我们希望我们的扫描器做什么？我们的扫描器函数将根据应用程序的要求被调用（也许是通过轮询或在有数据可供处理时产生一个事件）。它将重复地从其输入中读取数据，直到它。

- 遇到一个内部错误（例如，坏数据）。
- 返回一个有效的消息

尽管lex规范的格式从根本上说是简单的，但像许多工具一样，它有足够的选项，如果你想的话，可以让你花很长的时间。清单1中的规范被分为以下三个部分。

定义%%规则%%用户代码

清单1 简单的串行协议解释器的裸体lex规范

```
/* 定义 */

数字[0-9]
双倍值
[+-]?{digit}{1,10}”.”{digit}{1,10}([eE][+-]?{digit}{1,2})?

%%

/*规则 */ sv,{MessageStruct.msg=SHAFT_VELOCITY;}
{DoubleValue}{MessageStruct.Value=strtod (yytext,NULL) 。
```

```
returnVALID_INPUT;}
.{returnINVALID_INPUT;}。

%%

/* 用户代码 */

/*Implementationofinputfunction*/
intpipe_yyinput(char*buf,intmax_size)
{
/*读取串行端口获取输入的数据*/。
/*重新接收的字符数*/。
}
/* ------------- */
```

每一节都由一个单行分开，只包含
string **%%** .在清单1中，定义部分的唯一项目是一个复杂正则表达式的别名。你可以使用别名来避免在整个规则部分重复一个复杂的表达式。DoubleValue别名描述了一个双精度小数值的可接受格式。Lex支持一种大型的正则表达式语法，在配套的手册中有所记载。第二部分是规则部分。这是lex规范的业务部分，描述了扫描仪的大部分功能。规则部分是你 猜到的，是一个规则的列表。每条规则由一个要匹配的输入模式和一个要采取的行动组成，如果它被匹配的话。输入模式可以是字面字符，也可以是我们定义部分的别名（用大括号括起来），或者两者的组合。

用户代码包含任何我们想在生成的源文件底部插入的额外代码。在这个例子中，我包含了一个输入函数，用于从串行端口读取数据。要启用这个函数需要一些额外的定义，但为了清楚起见我省略了这些定义。

默认情况下，一个lex生成的扫描器使用一个全局文件指针获得其输入，这不适合于文件I/O不可用或不必要的应用。这就是为什么我在这里重新定义了输入函数。执行的过程如下。

- 扫描器从调用输入函数开始。
- 如果没有收到输入，它将返回0（文件结束），否则它将尝试将输入与它的模式相匹配。
- 如果 ″sv，″被匹配，则执行相应的动作。这个动作在一个消息**结构**中存储一个枚举类型，以表示收到的消息。

- 扫描器将VALID_INPUT（一个预定义的常数）返回给调用函数。调用函数现在可以解释消息结构的内容。
- 当收到双精度值时，它被复制到消息结构中。扫描器将当前匹配的模式暂时存储在一个数组yytext中，用来从输入流中获取数值并将其复制到消息结构中。

如果在任何阶段收到了一个意外的输入（不能匹配的东西），它将被默认的规则所匹配，它由句号（...）表示。在这种情况下，我们把输入无效的事实返回给调用函数。扫描器将重复调用输入函数，直到没有数据可用。如果输入函数返回0（例如，因为通过串行连接没有进一步的数据可用），扫描器将返回0给调用函数，但将保持其内部状态。当它随后被调用时，它将简单地再一次尝试从输入函数中获得额外的数据。

如前所述，如果在任何阶段收到的数据与预期的模式不一致，就会匹配默认的规则并返回一个错误。通过使用不同的选项（在lex规范和命令行中）可以改变lex扫描器行为的许多方面，所产生的扫描器可能是一个非常不同的野兽。要看这个例子中需要的所有具体选项，请从ftp://ftp.embedded.com/pub/2003/03power下载代码和注释。

现在我们有了一个lex规范，我们可以用lex来编译它并产生C/C++扫描器的源代码。

该例子的命令行是。

```
flex lex_spec.txt
```

为一个名为lex_spec.txt的描述文件。要在你的程序中使用扫描器，根据需要调用扫描器函数int yylex(void)。

配置文件解析

图4说明了对一个简单的配置文件的解析。该配置文件被设计为容纳一个设备GUI的可变数量的图像描述的列表。这些图像描述定义了。

- 图像文件名
- 在我们设备屏幕上的应用窗口中显示图像的x和y坐标。

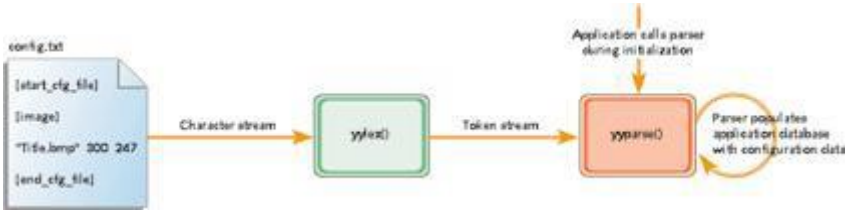


图4：配置文件解析器应用中的数据流 点击这里查看大图

当固件初始化时，我们希望它能从闪存中读取配置文件，对其进行解析，生成一个简单的表格或数据库，其中包含的数据，并显示图像。这些标记将传递给yacc生成的解析器进行解释。生成扫描器和解析器的过程是

- 写一个lex规范文件和一个yacc语法文件（这就是我们所说的yacc规范），描述我们希望我们的扫描仪和分析器做什么。
- 使用lex和yacc编译输入规范。
- 编译并将生成的C或C++模块与应用程序的其他部分连接起来。

我们已经看到了如何写一个词条描述，所以我们在这里只关心这个特殊应用所引入的yacc语法。清单2显示了lex的说明，现在应该可以理解了。清单3显示了yacc语法文件。

清单2：一个简单的配置文件扫描器的Lex规范

```
/* 定义 */

%{
#include "y_tab.h"
/* 定义循环计数器 */ int
iCount;
%}

/* 定义一些别名，以便在下面使用 */ whitespace[
tr] 。
数字[0-9]
longVal[+-]?{digit}{1,10}
stringVal[_./-a-zA-Z0-9]{1,50}

%%
```

```
/* 语法规则 */

[image]{returnDEFINE_IMAGE;}。[start_cfg_file]{returnSTART_CONFIGURATI
ON;}。[end_cfg_file]{returnEND_CONFIGURATION;}"{stringVal}"{for(iCount
=0;iCount<>)
{
yylval.stringValue[iCount]=yytext[iCount]。
}
yylval.stringValue[yyleng]='';
returnSTRING;
}
{longVal}{
yylval.longValue=strtol(yext,NULL,10);
returnLONG;
}
{whitespace}{;/*DoNothing*/}
.{returnBAD_INPUT;}//Error, nome aningfulinput

>{yyterminate();}。

%%

/* 用户代码 */
```

清单3：一个简单配置文件解析器的yacc语法文件

```
/* 定义 */

%{
/* 词库函数的原型 */ int
yylex(void);
/*错误处理函数的原型 */ int yyerror(char *
pErrorMsg);
%}

/* 终端令牌（从扫描仪收到的令牌）的类型定义 */
团结的百分比
{
charstringValue[51];
longlongValue;
}
```

```
/* 终端 TOKENS 的定义, 由词典提供 */
%token DEFINE_IMAGE
%token START_CONFIGURATION
%token END_CONFIGURATION
%token BAD_INPUT
%token STRING
%token LONG

%%

/* 语法规则 */

配置：START_CONFIGURATION
ListOfImages
结束配置
{
return0。
/*ConfigurationFileParsingComplete.*/
}

ListOfImages:ImageDefinition|ListOfImagesImageDefinition
{
/*这里不要做什么。这个规则是为了允许变量的长度而提供的*/。
/*Listofimagestobedefined*/
}

ImageDefinition:DEFINE_IMAGE
STRING/*Filename*//。
LONG/*xco-ordinate*/
LONG/*yco-ordinate*/
{
/*Storedata*/
}

BadInput:BAD_INPUT{return1;}。

%%

/* 用户代码 */

intyyerror(char*pErrorMsg)
{
return1;/*Recoverynotattempted*/。
}
```

yacc的语法格式类似于lex的规范。(实际上，yacc先出现，lex借用了这种格式。)语法分为三个部分。

定义
%%
语法规则
%%
用户代码

yacc语法的声明部分的第一项是包含在%{和%}模式内的C代码体。这段代码被逐字复制到生成的分析器模块的顶部。

yyerror()函数在遇到内部错误时被解析器调用；这个函数必须由开发者提供。一个典型的错误是扫描器提供了一个有效的标记，但是这个标记在标记流中不在一个有效的位置上，不能被匹配到一个规则。

在声明部分，还有终端标记的声明。终端标记（大写）是由扫描器返回给分析器的标记。其中一些令牌只有一个令牌类型，如例子中的令牌DEFINE_IMAGE。但是考虑到令牌LONG .这个标记有一个类型和一个相关的语义值；如果它没有一个值，那就没有用。终端标记的语义值是由扫描器使用yylval联盟传递给解析器的。令牌语义值的可能数据类型是用%union语句定义的。

第二部分是语法规则部分。这一部分包含各种合法的标记组合。不仅使用终端令牌，而且还使用额外的非终端令牌（小写）来产生完整的语法。非终端令牌是分析器内部的，不为扫描器所知。与扫描器一样，当语法规则被匹配时，相应的动作和其中的C代码被执行。

解析器/扫描器组合的执行情况如下。

- 应用程序使用适当的系统调用打开配置文件进行读取，扫描器的全局文件指针yyin被分配到配置文件的文件指针上。然后应用程序调用解析器函数yyparse()。
- 解析器反复调用扫描器的yylex()函数，直到遇到错误、到达文件末尾，或者解析器的

语法规则是匹配的。扫描器将终端标记和它们的语义值返回给分析器。语义值从扫描器的yytext数组中获得，并放在适当的yylval联盟成员中。

例如，扫描器在收到longVal模式后将执行相关的扫描器动作。这个动作将模式字符串（yytext）转换为一个长整数变量，并将这个值存储在yylval联盟中供解析器使用。然后扫描器返回LONG终端令牌。收到LONG令牌后，解析器知道这个令牌有一个与之相关的语义值，以及要访问哪个联盟成员（由于令牌声明）。该令牌被推到解析器的内部堆栈中，然后再次调用扫描器以获得下一个令牌。

当解析器匹配图像定义规则时，相关的动作被调用，图像数据被动作代码添加到内部应用数据库中（未显示）。解析器也匹配递归的ListOfImages规则，因为这个规则可以匹配一个或多个图像定义。这允许解析器在收到END_CONFIGURATION标记之前匹配数量不定的图像定义。

当收到END_CONFIGURATION标记时，解析器与配置规则匹配并返回给调用者。配置文件的处理就完成了。

如果在任何阶段，扫描器返回BAD_INPUT标记（作为对某些扫描器无法标记的输入的回应），BadInput规则被解析器匹配，解析器就会中止（返回1）。如前所述，收到不匹配任何语法规则的有效标记，解析器会调用 yyerror() 并传递一个描述所遇到的错误的字符串。

要编译yacc语法并产生C语言分析器的源代码，命令行是。

```
bison -y -d yacc_gmr.txt
```

为一个名为yacc_gmr.txt的语法文件。像lex一样，yacc有许多命令行选项。

尽管lex产生的扫描器通常比手工编码的等价物更快，但yacc产生的分析器通常比设计良好的手工编码的等价物更慢。因此，如果应用程序需要极快的性能，你可能想避免使用yacc。考虑到嵌入式领域包含了大量不同的硬件类型和

能力，有支持和反对使用自动生成的扫描器和解析器的情况。虽然lex和yacc是成熟的工具，但它们没有经过任何特定质量标准的验证。由于这个原因，在我看来，它们不适合用于安全关键系统。在一天结束的时候，是否适合的问题只有个别开发者可以回答。

莱克斯和雅克的味道

虽然lex和yacc通常被认为是UNIX的实用工具，但也有适用于各种平台的版本。在不太可能的情况下，如果你的特定开发操作系统不存在一个版本，你的情况仍然不坏。用lex和yacc进行交叉编译是很简单的。这两个工具输出的源代码可以用任何ANSI C或C++编译器进行编译，所以没有必要在与你的编译器相同的平台上使用lex和yacc。在我所使用的各种编译器下，生成的输出从未出现过任何编译问题。

事实上，输出通常比输入文件更具可移植性。例如，如果你试图将你的flex规范移植到AT&T lex（众多UNIX lex版本之一），由于两者之间的差异，可能会出现一些问题。然而，Yacc语法通常很容易在不同的Yacc版本之间移植。

如前所述，我一直使用flex 2.5和bison 1.25作为我的首选工具。像软件开发的许多领域一样，使用哪种lex和yacc变体的问题可能是一个感性的问题。下面列出了我选择的主要原因，但请记住，许多其他版本可能同样适合你的需要。

- Flex和bison提供了从Windows下的DOS盒子中运行的便利（至少对我来说是这样）（尽管它们并不是唯一可以这样做的版本）。
- Flex被认为是一个特别稳定的lex版本，而且生成的代码也比其他一些版本快。
- 如果需要，可以用flex来生成C++扫描器类。
- Bison v. 1.24及以上版本不限制将bison生成的解析器纳入商业软件的发行中。(以前的版本是这样的。)同样地，对flex生成的扫描器的发行也没有限制。

bison的一个限制是它不能生成C++输出。这对我来说不是一个大问题，因为在一个C++应用程序中隐藏一两个C模块并不困难。顺便提一下，现在有更多的flex和bison的最新版本可以生成C++代码。它们被称为flex++和bison++。你可以下载flex和bison（以及大量的

其他有用的软件）来自www.gnu.org。每个发行版都提供了文档。

我希望这篇文章能让你对lex和yacc产生兴趣。在过去的三年里，我一直在定期使用这些工具，它们无疑省去了许多星期的编码、调试和维护。如果你想得到上述例子的完整规范文件，可以从ftp://ftp.embedded.com/pub/2003/03power。

Liam Power是Embedded Labs Ltd.的高级设计工程师，这是一家位于爱尔兰沃特福德的嵌入式系统开发公司。他专注于汽车应用，可以通过以下方式与他联系。

LEX和YACC教程

作者：**Tom Niemann**

epaperpress.com

内容

前言

本文解释了如何使用lex和yacc构建一个编译器。Lex和yacc是用来生成词法分析器和解析器的工具。我假设你能用C语言编程，并理解数据结构，如链接列表和树。

引言描述了编译器的基本构件，并解释了lex和yacc之间的互动。接下来的两节将更详细地描述lex和yacc。有了这些背景，我们就可以构建一个复杂的计算器。常规的算术运算和控制语句，如**if-else**和**while**，都已实现。只要稍加改动，我们就可以把计算器转换成基于堆栈的机器的编译器。其余部分讨论了在编译器编写中经常出现的问题。例子的源代码可以从下面列出的网站上下载。

汤姆-尼曼 波特兰
，俄勒冈州
epaperpress.com/lexandyacc

简介

在1975年之前，编写一个编译器是一个非常耗时的过程。然后Lesk [1975] 和Johnson [1975] 发表了关于lex和yacc的论文。这些实用工具大大简化了编译器的编写。lex和yacc的实现细节可以在Aho [2006]中找到。Flex和bison是lex和yacc的克隆，可以从GNU和Cygwin免费获得。

Cygwin是GNU软件的一个32位Windows端口。事实上，Cygwin是Unix操作系统在Windows上的一个移植，并带有编译器gcc和g++。安装时只需下载并运行setup可执行文件。在**develop**下安装bison、flex、gcc-g++、gdb和make。**编辑器**下安装vim。最近我一直在Cygwin环境下使用flex和bison。

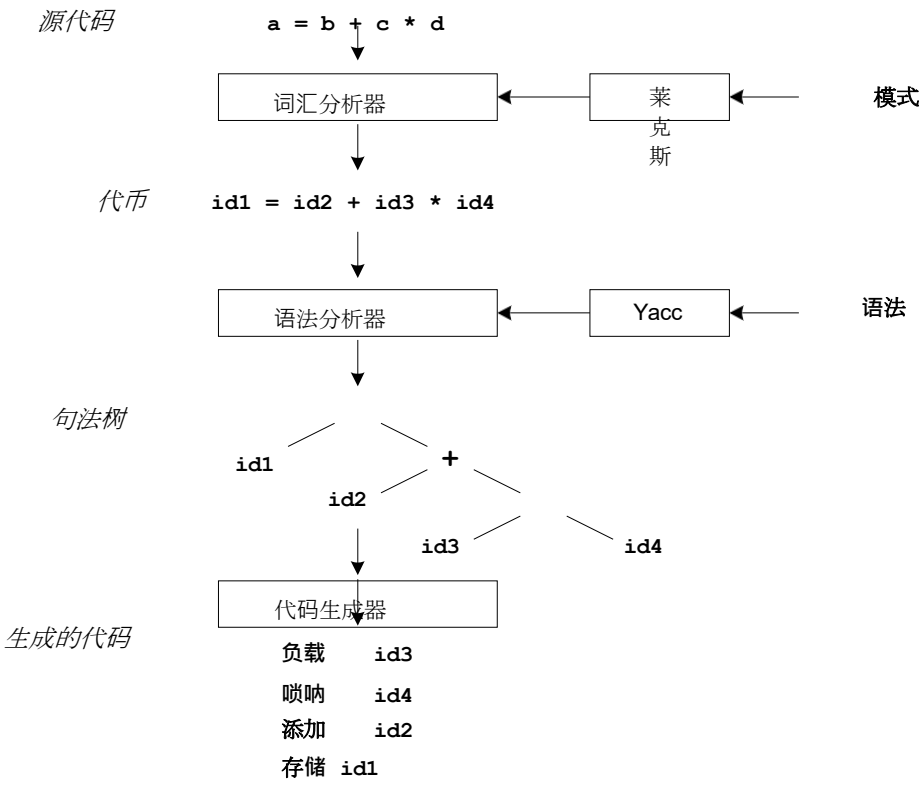


图1: 编译顺序

上图中的**模式**是你用文本编辑器创建的一个文件。Lex将读取你的模式，并为词法分析器或扫描器生成C代码。词法分析器根据你的模式匹配输入的字符串，并将字符串转换为标记。标记是字符串的数字表示，可以简化处理。

当词法分析器在输入流中发现标识符时，它会将它们输入到一个符号表中。符号表还可能包含其他信息，如数据类型（整数或实数）和每个变量在内存中的位置。所有随后对标识符的引用都是指适当的符号表索引。

上图中的**语法**是你用文本编辑器创建的一个文本文件。Yacc会读取你的语法并为语法分析器或解

析器生成C代码。语法分析器使用语法规则，允许它分析来自词法分析器的标记，并创建一个语法树。语法树对标记施加了一个层次结构。例如，运算符优先级

语法树中明显地存在着联想性。下一步，代码生成，对语法树进行第一深度的行走以生成代码。一些编译器生成机器代码，而另一些编译器，如上图所示，则输出汇编语言。

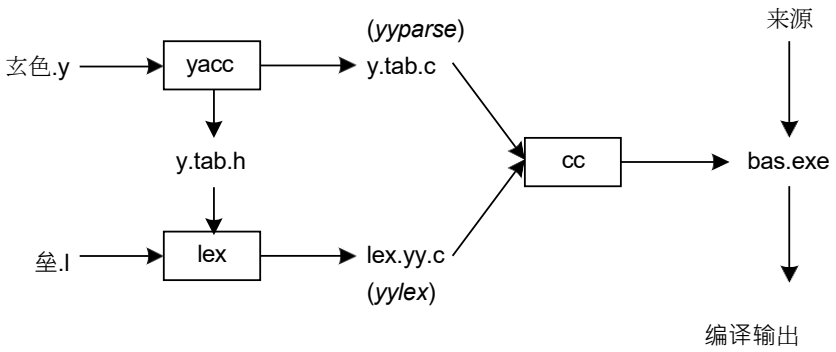


图2：用Lex/Yacc建立一个编译器

图2说明了lex和yacc使用的文件命名约定。我们假设我们的目标是编写一个BASIC编译器。首先，我们需要指定lex的所有模式匹配规则（**bas.l**）和yacc的语法规则（**bas.y**）。创建我们的编译器**bas.exe**的命令列在下面。

```
yacc -d bas.y# 创建y.tab.h, y.tab.c
                                lex bas.l#create
                                lex.yy.c cc
lex.yy.c y.tab.c -obas.exe#compile/link
```

Yacc读取**bas.y**中的语法描述，并在文件**y.tab.c**中生成一个语法分析器（解析器），其中包括函数**yyparse**，包括在文件**bas.y**中的标记声明。**-d**选项使yacc生成标记的定义，并把它们放在文件**y.tab.h**中。Lex读取**bas.l**中的模式描述，包括文件**y.tab.h**，并生成一个词法分析器，包括文件**lex.yy.c**中的函数**yylex**。

最后，词法分析器和解析器被编译并连接在一起，创建可执行的**bas.exe**。在**main**中我们调用**yyparse**来运行编译器。函数**yyparse**自动调用**yylex**来获得每个标记。

莱克斯

理论

在第一阶段，编译器读取输入并将源代码中的字符串转换为标记。通过正则表达式，我们可以向lex指定模式，这样它就可以生成代码，让它扫描和匹配输入中的字符串。在输入到lex的每个模式都有一个相关的动作。通常情况下，一个动作会返回一个代表匹配字符串的标记，供解析器随后使用。最初，我们将简单地打印匹配的字符串，而不是返回一个标记值。

下面表示一个简单的模式，由一个正则表达式组成，用来扫描标识符。Lex将读取这个模式，并为一个扫描标识符的词法分析器生成C代码。

字母 (字母 | 数字) *

这个模式可以匹配以单个字母开头，后跟零个或多个字母或数字的字符串。这个例子很好地说明了正则表达式中允许的操作。

- 重复，用 "*"运算符表示
- 交替，用"|"运算符表示
- 串联

*任何*正则表达式都可以被表达为有限状态自动机（FSA）。我们可以用状态和状态之间的转换来表示一个FSA。有一个起始状态和一个或多个最终或接受状态。

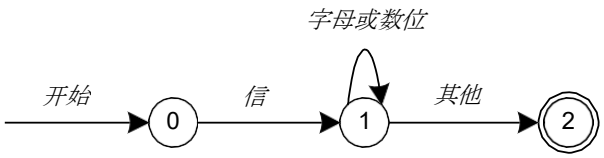


图3：有限状态自动机

在图3中，状态0是开始状态，状态2是接受状态。随着字符被读取，我们从一个状态过渡到另一个状态。当读到第一个字母时，我们过渡到状态1。当更多的字母或数字被读取时，我们保持在状态1。当我们读到一个非字母或数字的字符时，我们过渡到接受状态2。*任何*FSA*都*可以被表达为一个计算机程序。例如，我们的3状态机器很容易被编程。

开始：转入状态0 状态0：读

取c

如果c = 信，则转入状态1，转

入状态0

状态1：读取c

如果c=字母，则转入状态1 如果

c=数字，则转入状态1 转入状态2

状态2：接受字符串

这就是lex使用的技术。正则表达式被lex翻译成一个模仿FSA的计算机程序。使用下一个*输入*字符和*当前状态*，通过对计算机生成的状态表进行索引，可以轻松地确定下一个状态。

现在我们可以很容易地理解lex的一些限制。例如，lex不能用来识别嵌套结构，如括号。嵌套结构是通过加入一个堆栈来处理的。每当我们遇到"**(**"时，我们就把它推到堆栈上；当遇到"**)** "时，我们就把它与堆栈的顶部相匹配并弹出堆栈。然而，lex只有状态和状态之间的转换。因为它没有堆栈，所以不适合解析嵌套结构。Yacc用堆栈增强了FSA，可以轻松地处理诸如括号之类的结构。重要的是要使用正确的工具来完成工作。Lex擅长模式匹配。Yacc适合于更具挑战性的任务。

实践

元老级人物	匹配度
.	除换行外的任何字符
\n	换行
****	前面表达式的零个或多个副本
+	一个或多个前述表达的副本
?	前述表达式的零或一份副本
^	行首
$	行末
a|b	**a**或**b**
(ab)+	一份或多份**ab**（分组）。
"a+b"	字面意思是 "**a+b**"（C语言转义仍然有效）。
[]	字符类

表1：模式匹配基元

表达方式	匹配度
abc	**abc**
abc*	**ab abc abcc abccc ...**
abc+	**abc abcc abccc ...**
a(bc)+	**abc abcbc abcbcbc ...**
a(bc)?	**a abc**
[abc]	其一：**A**、**B**、**C**
[a-z]	任何字母，a-z
[a/z]	其一：**A**, **-**, **Z**
[-az]	其中之一。**-**, **a**, **z**
[A-Za-z0-9]+	一个或多个字母数字字符
[\t\n]+	空白
[^ab]	除了：**A**, **B**
[a^b]	其中之一：**A**, **^**, **B**
[a|b]	其中之一：**A**, **|**, **B**
a|b	其中之一：**A**, **B**

表2：模式匹配的例子

lex中的正则表达式是由元字符组成的（表1）。模式匹配的例子显示在表2中。在一个字符类中，普通运算符失去了它们的意义。

字符类中允许的两个运算符是连字符（"-"）和环形符号（"^"）。当在两个字符之间使用时，连字符表示一个字符的范围。环形符号，当作为第一个字符使用时，否定了表达式。如果两个模式匹配相同的字符串，最长的匹配者获胜。如果两个匹配都是相同的长度，则使用列出的第一个模式。

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

对Lex的输入被分为三个部分，用**%**来划分这些部分。这一点最好用例子来说明。第一个例子是最短的lex文件。

```
%%
```

输入被复制到输出，一次一个字符。第一个**%%**总是必须的，因为必须有一个规则部分。然而，如果我们没有指定任何规则，那么默认的动作是匹配所有的内容并将其复制到输出。输入和输出的默认值分别为**stdin**和**stdout**。下面是同样的例子，明确地编码了默认值。

```
%%
    /* 匹配除换行之外的所有内容 */
.   ECHO。
    /* 匹配换行 */
\EN ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

在规则部分规定了两种模式。每个模式必须在第一栏开始。后面是空白（空格、制表符或换行符）和一个与该模式相关的可选动作。该动作可以是一个单一的C语句，也可以是多个C语句，用大括号括起来。任何不在第一列开始的内容都会被逐字复制到生成的C文件中。我们可以利用这一行为在我们的lex文件中指定注释。在这个例子中，有两个模式，**.** "和"\n"，每个模式都有一个**ECHO**动作。一些宏和变量是由lex预定义的。**ECHO**是一个写出与模式匹配的代码的宏。这是对任何未匹配的字符串的默认动作。通常情况下，**ECHO**被定义为。

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

变量**yytext**是指向匹配字符串的指针（以NULL为结尾），**yleng**是匹配字符串的长度。变量 **yyout**是输出文件，默认为 stdout。当输入用尽时，lex会调用函数**yywrap。**如果完成了就返回1，如果需要更多处理就返回0。每个C程序都需要一个主函数。在这种情况下，我们简单地调用**yyylex**，

这是lex的主入口。一些lex的实现在库中包含了**main**和**yyyrap**的副本，因此不需要对它们进行明确的编码。这就是为什么我们的第一个例子，最短的lex程序能够正常运行。

命名	职能
int yylex(void)	调用词法，返回令牌
char *yytext	指向匹配字符串的指针
yyleng	匹配字符串的长度
yylval	与令牌相关的值
int yywrap(void)	包裹，如果完成则返回1，如果未完成则返回0
文件 *yyout	输出文件
文件 *yyin	输入文件
初始化	初始启动条件
开始	条件开关启动条件
ECHO	写入匹配的字符串

表3：Lex预定义变量

这里有一个根本不做任何事情的程序。所有的输入都被匹配，但没有任何动作与任何模式相关联，所以不会有任何输出。

```
%%
.
\n
```

下面的例子在文件中的每一行都预置了行号。lex的一些实现预先定义并计算**yylineno**。lex的输入文件是**yyin**，默认为**stdin**。

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

定义部分由替换、代码和起始状态组成。定义部分的代码被简单地原样复制到生成的C文件的顶部，并且必须用**"%{"**和**"%}"**标记的括号。替换可以简化模式匹配规则。例如，我们可以定义数字和字母。

```
数字      [0-9]字
母        [A-Za-z]
%{
    int count;
%}
%%
    /* 匹配标识符 */
{字母}({字母}|{数字})*。               count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

定义性术语和相关表达式之间必须有空格。在规则部分对替换的引用用大括号（**{letter}**）包围，以区别于字面意义。当我们在规则部分有一个匹配，相关的C代码就会被执行。下面是一个扫描器，它计算文件中的字符数、字数和行数（类似于Unix的wc）。

```
%{
    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的语法是用Backus Naur Form（BNF）的变体来描述的。这种技术是由John Backus和Peter Naur开创的，被用来描述ALGOL60。BNF语法可以用来表达*无语境*语言。现代编程语言中的大多数构造都可以用BNF表示。例如，一个表达式的乘法和加法的语法是

```
e -> e + e
e -> e * e
E -> id
```

已经指定了三个产品。出现在生产的左侧（lhs）的术语，如E（表达式）是非终端。像**id**（标识符）这样的术语是终端（由lex返回的标记），只出现在生产的右侧（rhs）。这个语法规定，一个表达式可以是两个表达式的和，两个表达式的乘积，或者是一个标识符。我们可以使用这个语法来生成表达式。

```
E -> E * E        (r2)
  -> E * z        (r3)
  -> E + E * z    (r1)
  -> E + y * z    (r3)
  -> x + y * z    (r3)
```

在每一步，我们扩大一个术语，用相应的rhs替换生产的lhs。右边的数字表示哪个规则适用。要解析一个表达式，我们需要做相反的操作。我们需要把表达式还原成一个单一的非终端（起始符号），而不是从语法中生成一个表达式。这被称为*自下而上*或*移位还原*解析，使用堆栈来存储术语。下面是同样的推导，但顺序相反。

```
 1    . x + y * z     移位
 2    x . + y * z     reduce(r3)
 3    E . + y * z     移位
 4    E + . y * z     移位
 5    E + y . * z     reduce(r3)
 6    E + E . * z     移位
 7    E + E   . z     移位
 8    E + E   z .     reduce(r3)
 9    E + E   E .     减少(r2)          发出乘法
10    E + E .         reduce(r1)        发出添加
11    E .             接受
```

点左侧的术语在堆栈中，而剩余的输入则在点的右侧。我们首先将代币移到堆栈中。当堆栈的顶部与生产的rhs相匹配时，我们就用生产的lhs来替换堆栈中的相匹配的代币。换句话说，rhs的匹配代币被从堆栈中弹出，而生产的lhs被推到堆栈中。匹配的代币被称为*句柄*，我们将句柄*减少*到生产的lhs上。这个过程一直持续到我们把所有的输入都转移到栈上，栈上只剩下起始非终端。在第1步中，我们将**x**转移到栈中。第2步将规则r3应用于堆栈，将**x**改为**E**。我们继续移位和减少，直到堆栈中只剩下一个非终端，即起始符号。在第9步，当我们减少规则r2时，我们发出了乘法

指令。同样地，加法指令在第10步发出。因此，乘法的优先级比加法高。

考虑一下第6步的转移。我们可以不进行移位，而是减少并应用规则r1。这将导致加法比乘法有更高的优先权。这就是所谓的*转移-减少*冲突。我们的语法是*模棱两可的*，因为有一个以上的可能推导会产生表达式。在这种情况下，运算符的优先权受到影响。作为另一个例子，规则中的关联性

```
E -> E + E
```

是不明确的，因为我们可以在左边或右边进行递归。为了纠正这种情况，我们可以重写语法，或者给yacc提供指示，说明哪个运算符有优先权。后一种方法比较简单，将在练习部分进行演示。

下面的语法有一个*reduce-reduce*冲突。栈上有一个**id，**我们可以还原到**T**，或者**E**。

```
E -> T
E -> id
T -> id
```

当有冲突时，Yacc会采取默认行动。对于shift-reduce冲突，Yacc会进行移位。对于减少-还原冲突，它将使用列表中的第一条规则。每当有冲突存在时，它也会发出警告信息。警告可以通过使语法不含糊而被抑制。在随后的章节中，将介绍几种消除歧义的方法。

实践，第一部分

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

yacc的输入被分为三个部分。定义部分由标记声明和用**"%{"**和**"%}"**括住的C代码组成。BNF语法被放在规则部分，用户子程序被添加在子程序部分。

这一点最好通过构建一个可以加减数字的小计算器来说明。我们先来看看lex和yacc之间的联系。这里是yacc输入文件的定义部分。

```
%token INTEGER
```

这个定义声明了一个**INTEGER**标记。Yacc在文件**y.tab.c**和include文件**y.tab.h**中生成了一个解析器。

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex包括这个文件，并利用这些定义来获取标记值。为了获得令牌，yacc调用**yylex**。函数**yylex**的返回类型为int，返回一个token。与令牌相关的值由lex在变量**yylval**中返回。例如。

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

将在**yylval**中存储整数的值，并将令牌**INTEGER**返回给yacc。的类型是**yylval**是由**YYSTYPE**决定的。由于默认类型是整数，在这种情况下效果很好。
代号值0-255是为字符值保留的。例如，如果你有一个规则，如

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

将返回减号或加号的字符值。请注意，我们把减号放在前面，这样它就不会被误认为是一个范围代号。生成的令牌值通常从258左右开始，因为lex为文件结束和错误处理保留了几个值。下面是我们的计算器的完整lex输入规范。

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%

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

[-+\n]       返回*yytext。

[\t]         ; /* 跳过空格 */

.            yyerror("无效字符")。

%%

int yywrap(void) {
    return 1;
}
```

在内部，yacc在内存中维护两个堆栈；一个是解析堆栈，一个是值堆栈。解析栈包含代表当前解析状态的终端和非终端。值栈是一个**YYSTYPE**元素的数组，将一个值与解析栈中的每个元素联系起来。例如，当lex返回一个**INTEGER**标记时，yacc将这个标记转移到解析栈中。同时，相应的**yylval**被转移到值栈中。解析栈和值栈总是同步的，所以在栈上找到一个与令牌相关的值是很容易完成的。下面是我们的计算器的yacc输入规范。

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%
```

方案。

```
        程序expr '\n'                { printf("%d\n", $2); }
        |
        ;
```

阐述。

```
        整数{ $$ = 1; }
        | expr '+' expr{ $$ = 1 + 3; }
        | expr '-' expr{ $$ = 1 - 3; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

规则部分类似于前面讨论的BNF语法。产物的左侧，或者说非终端，是以左对齐的方式输入的，后面有一个冒号。随后是生产的右侧。与一个规则相关的动作用大括号输入。

通过左旋，我们指定一个程序由零个或多个表达式组成。每个表达式都以一个换行来结束。当检测到一个换行时，我们会打印表达式的值。当我们应用规则

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

我们将解析堆栈中生产的右侧替换为相同生产的左侧。在这种情况下，我们弹出 "**expr '+' expr** "并推入 "**expr**"。我们通过从堆栈中弹出三个术语并推回一个术语来减少堆栈。在我们的C语言代码中，我们可以通过指定"**$1** "为生产右侧的第一个术语，"**$2** "为第二个术语，以此类推来引用值栈中的位置。"**$$**"指的是缩减后的堆栈顶部。上述动作增加了与两个表达式相关的值，从值堆栈中弹出了三个项，并推回了一个和。因此，解析栈和值栈仍然是同步的。

当我们从**INTEGER**减少到**expr**时，数字值最初被输入到堆栈中。之后
INTEGER被转移到堆栈中，我们应用规则

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

INTEGER标记被从解析栈中弹出，然后推送**expr**。对于值堆栈，我们从堆栈中弹出整数值，然后再把它推回去。换句话说，我们什么都不做。事实上，这是默认动作，不需要指定。最后，当遇到换行时，与**expr**相关的值被打印出来。

在出现语法错误时，yacc会调用用户提供的函数**yyerror**。如果你需要修改**yyerror**的接口，那么就修改yacc包含的罐头文件，以适应你的需要。我们的yacc规范中的最后一个函数是**main ...**如果你想知道它在哪里。这个例子仍然有一个模棱两可的语法。尽管yacc会发出shift-reduce的警告，但它仍然会使用shift作为默认操作来处理语法。

实践，第二部分

在本节中，我们将对上一节的计算器进行扩展，加入一些新的功能。新功能包括算术运算符乘法和除法。圆括号可以用来覆盖运算符的优先级，单字符变量可以在赋值语句中指定。下面说明了输入和计算器输出的例子。

```
用户。3 * (4 + 5)
计算：27
用户：x = 3 * (4 + 5)
用户：y = 5
用户：X
计算：27
用户：Y
计算：5 用户：X
+ 2*Y 计算：
3737
```

词法分析器返回**VARIABLE**和**INTEGER**标记。对于变量，**yylval**指定了符号表**sym**的索引。在这个程序中，**sym**只是保存相关变量的值。当**INTEGER**标记被返回时，**yylval**包含所扫描的数字。下面是lex的输入规范。

```
%{
    #include <stdlib.h>
    void yyerror(char *);
    #include "y.tab.h"
%}

%%

    /* 变量 */
[a-z]       {
                yylval = *yytext - 'a';
                return VARIABLE;
            }

    /* 整数 */ [0-
9]+         {
                yylval = atoi(yytext);
                返回INTEGER。
            }

    /* 运营商 */
[-+()=/*/n] { 返回 *yytext; }

    /* 跳过空白 */ [ `t'
              ;]

    /*其他的都是错误 */
.               yyerror("无效字符")。

%%

int yywrap(void) {
    return 1;
}
```

yacc的输入规范如下。**INTEGER**和**VARIABLE**的标记被yacc用来在**y.tab.h**中创建**#defines**，以便在lex中使用。接下来是算术运算符的定义。我们可以指定**%left**，表示左关联，或者**%right**表示右关联。最后列出的定义具有最高的优先级。因此，乘法和除法比加法和减法的优先级更高。所有四个运算符都是左联动的。使用这个简单的技术，我们就能对我们的语法进行模糊处理。

```
%token INTEGER VARIABLE
左边的'+''-'。
%left '*' '/'

%{
    void yyerror(char *);
    int yylex(void);
    int sym[26];
%}

%%

方案。
        程序语句' (n) '
        |
        ;

声明。
        expr{ printf("%d\n", $1); }
        | VARIABLE '=' expr{ sym[1] = 3; }
        ;

阐述。
        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

这个版本的计算器比以前的版本要复杂得多。主要变化包括控制结构，如**if-else**和**while**。此外，在解析过程中还构建了一个语法树。在解析结束后，我们会在语法树上行走以产生输出。我们提供了两个版本的树形行走程序。

- 一个解释器，在树上行走时执行语句
- 一个为假设的基于堆栈的机器生成代码的编译器
- 一个生成原始程序的语法树的版本 为了使事情更具体，这里

有一个示例程序。

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

与解释性版本的输出。

```
0
1
2
```

和输出的编译器版本，以及

```
    推动   0 啪
           x
L000:
    推动    x
    推动    3
    compLT
    jz      L001
    推动    x
    印刷品
    推动    x
    推动    1
    增加
    流行    x
    脉冲    L000
L001:
```

一个能生成语法树的版本。

图0。

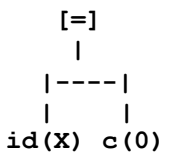
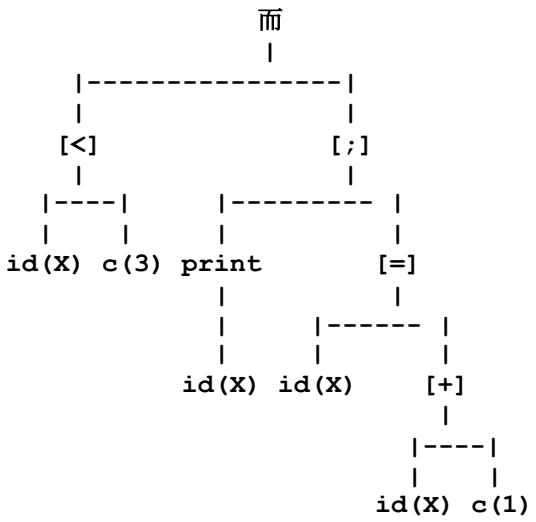


图1：



include文件包含语法树和符号表的声明。符号表（**sym**）允许使用单字符的变量名。语法树中的一个节点可以容纳一个常数（**conNodeType**），一个标识符（**idNodeType**），或者一个带有操作符的内部节点（**OPRNodeType**）。一个联合体封装了所有这三种变体，**nodeType.type**用于确定我们有哪种结构。

lex输入文件包含**VARIABLE**和**INTEGER**标记的模式。此外，还为**EQ**和**NE**等双字符运算符定义了标记。单字符运算符只是作为其本身返回。

yacc输入文件定义**YYSTYPE**，即**yylval**的类型，为

```
%union {
    int iValue;                 /* 整数值 */
    char sIndex;                /* 符号表索引 */
    nodeType *nPtr;             /* 节点指针 */
};
```

这导致在y.tab.h中产生以下内容。

```
typedef union {
    int iValue;                 /* 整数值 */
```

```
    char sIndex;            /* 符号表索引 */
    nodeType *nPtr;         /* 节点指针 */
YYSTYPE;
```
外来的YYSTYPE yylval。

常量、变量和节点可以在解析器的值栈中用**yylval**表示。下面给出了一个更准确的十进制整数的表示方法。这类似于C/C++中以0开头的整数被归类为八进制。

```
0           {
                yylval.iValue = atoi(yytext); 返回
                INTEGER。
            }

[1-9][0-9]* {
                yylval.iValue = atoi(yytext); 返回
                INTEGER。
            }
```

注意类型的定义

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

这将**expr**与**nPtr**绑定，并将**INTEGER**与**YYSTYPE**联盟中的**iValue**绑定。这是必须的，这样yacc才能生成正确的代码。例如，规则

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

应该产生以下代码。注意**yyvsp[0]**的地址是值堆栈的顶部，或与**INTEGER**相关的值。

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

单元减号运算符比二元运算符有更高的优先权，如下所示。

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

%nonassoc表示不隐含关联性。它经常被用于与
%prec来指定一个规则的优先级。因此，我们有

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

表示该规则的优先级与令牌**UMINUS**的优先级相同。而**UMINUS**（如上定义）的优先级高于其他运算符。类似的技术被用来消除与if-else语句相关的歧义（见If-Else歧义）。

语法树是自下而上构建的，当变量和整数被减少时，分配叶子结点。当遇到运算符时，会分配一个节点，并将以前分配的节点的指针作为操作数输入。

语法树建立后，函数**ex**被调用，对语法树进行深度优先的行走。深度优先的行走是按照节点最初被分配的顺序访问这些节点的。这将导致运算符按照解析过程中遇到的顺序被应用。**ex**有三个版本：一个解释版本，一个编译版本，以及一个生成语法树的版本。

列入文件

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* 常量 */ typedef
struct {
    int value;                  /* 常数的值 */
} conNodeType;

/* 标识符 */ typedef
struct {
    int i;                      /*下标到符号阵列 */
} idNodeType;

/* 运营商 */ typedef
struct {
    int oper;                   /* operator */
                                int nops;/*操作数     */
    struct nodeTypeTag **op;/*操作数 */
} oprNodeType。

类型化的结构 nodeTypeTag {
    nodeEnum type;              /* 节点的类型 */

    联盟{
        conNodeType con;        /* 常量 */
        idNodeType id;          /*标识符 */
        oprNodeType opr;        /* 运营商 */
    };
} nodeType;

外来的int sym[26]。
```

Lex输入

```
%{
#include <stdlib.h>
#include
"calc3.h"#include
"y.tab.h" void
yyerror(char *);
%}

%%

[a-z]       {
                yylval.sIndex = *yytext - 'a';
                return VARIABLE;
            }

0           {
                yylval.iValue = atoi(yytext); 返回
                INTEGER。
            }

[1-9][0-9]* {
                yylval.iValue = atoi(yytext); 返回
                INTEGER。
            }


[-()<>=+*/;{}.] {
                返回 *yytext;
             }

">="            返回GE。
"<="            返回LE。
"=="            返回EQ。
"！="           返回NE。
"while "        返回WHILE。
"如果 "         返回IF。
"else "         返回ELSE。
"打印 "         返回PRINT。

[ \t\n]+        ;       /* 忽略空白 */

.               yyerror("未知字符")。
%%
int yywrap(void) {
    return 1;
}
```

Yacc输入

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calc3.h"

/*原型 */
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];                    /* 符号表 */
%}

%union {
    int iValue;                 /* 整数值 */
    char sIndex;                /* 符号表索引 */
    nodeType *nPtr;             /* 节点指针 */
};

%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE

%left GE LE EQ NE '>' '<'。
左边的'+''-'。
%left '*' '/'
%nonassoc UMINUS

%type <nPtr> stmt expr stmt_list
```

```
%%

方案。
  函数{ exit(0); }
  ;

功能。
    函数 stmt{ ex(2美元); freeNode(2美元); }
  | /* 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); }
  ;

阐述。
    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。

    /* 分配节点 */
    如果((p = malloc(sizeof(nodeType))) == NULL)
        yyerror("out of memory")。

    /* 复制信息 */ p->type
    = typeCon;
    p->con.value = value。

    返回p。
}

nodeType *id(int i) {
    nodeType *p;

    /* 分配节点 */
    如果((p = malloc(sizeof(nodeType))) == NULL)
        yyerror("out of memory")。

    /* 复制信息 */ p->type
    = typeId。
    p->id.i = i。

    返回p。
}
```

```
nodeType *opr(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    int i;

    /* 分配节点 */
    如果((p = malloc(sizeof(nodeType))) == NULL)
        yyerror("out of memory")。
    如果((p->opr.op = malloc(nops * sizeof(nodeType))) == NULL)
        yyerror("out of memory")。

    /* 复制信息 */ 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);
    返回p。
}

void freeNode(nodeType *p) {
    int i;

    if (!p) return;
    如果(p->type == typeOpr) {
        for (i = 0; i < p->opr.nops; i++)
            freeNode（p->opr.op[i]）。
        free(p->opr.op)。
    }
    免费(p)。
}

void yyerror(char *s) {
    fprintf(stdout, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

翻译人员

```
#include <stdio.h>
#include
"calc3.h"#include
"y.tab.h"

int ex(nodeType *p) {
    if (!p) return 0;
    switch(p->type) {
    caseCon:   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]);
                        返回0。
      case IF: 如果(ex(p->opr.op[0]))
                          ex(p->opr.op[1]);
                        else if (p->opr.nops >
                        2)
                          ex(p->opr.op[2]); 返
                        回0。
      case PRINT:       printf("%d\n", ex(p->opr.op[0]))。
                        返回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:返回-ex(p-
      >opr.op[0])。
      case '+':         return ex(p->opr.op[0]) + ex(p->opr.op[1]) 。
      case '-':         返回 ex(p->opr.op[0]) - ex(p->opr.op[1])。
      case '*':         返回 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]) 。
      op[0]) <=         ex(  p->opr.     op[   1]); case NE:     返
      回 ex(p->opr.op[0])!
    }
  }
  返回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])。
            如果 (p->opr.nops > 2) {
                /* 如果其他 */
                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 {
                /* 如果 */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
                ex (p->opr.op[1]) 。
                printf("L%03d:\n", lbl1)。
            }
            突破。


        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;
        默认情况下。
            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")
                        printf("\tcompGT\n");
                        break; case
            GE:printf("\tcompGE\n");     ; case
            LE:printf("\tcompLE\n");
                        NE:printf("\tcompNE\n");brea
            k; case EQ: printf("\tcompEQ\n") break。
            }
        }
    }
    返回0。
}
```

图表

```
/* 源代码由Frank Thomas Braun提供 */

#include <stdio.h>
#include <string.h>

#include
"calc3.h"#include
"y.tab.h"

int del = 1; /*图形列的距离 */ int eps = 3; /*
图形线的距离 */

/* 用于绘图的接口（可以用GD或其他方式用 "真正的 "图形代替） */
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) 。

/* 语法树的递归绘制 */
void exNode (nodeType *p, int c, int l, int *ce, int *cm) 。

/*************************************************************/

/* 语法树操作的主入口 */ int ex (nodeType *p) {
    int rte, rtm;

    graphInit () 。
    exNode (p, 0, 0, &rte, &rtm);
    graphFinish();
    返回0。
}

/*c----cm---ce---->                          绘制叶子节点l叶子信息
 */

/*c----------------cm--------------ce---->绘制非叶子节点 l      节点信息
 *                 |
 *    -------------      ...----
 *    |        |                |
 *    v        v                v
 * child1 child2 ...        孩子-n
          che      che               che
 *cs      cs       cs                cs

 */
```

```
空白的exNode
    (   nodeType *p,
        int c, int l,          /*节点的起始列和行 */
        int *ce, int *cm/* 导致节点的末端列和中间列 */)
    )
{
    int w, h;            /* 节点的宽度和高度 */
    char *s;             /* 节点文本 */
    int cbar;            /*节点的 "真实 "起始列（位于中心位置以上
子节点） */
    int k;               /* 子女数 */
                         int che, chm;/*子女的结束列和中间列
                         */ int cs;/*子女的开始列 */
                         char word[20];/*扩展节点


    文本 */ if (!p) return;

    strcpy (word, "???"); /*不应该出现 */ s = word;
    switch(p->type) {
        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){
                 案例 趁着。        s   "而"。     突破。
                 案例 基金会。      s   "如果"。   突破。
                 案例 打印。        s   "打印"。   突破。
                 案例 ';':          s   "[;]";       突破。
                 案例 '=':          s   "[=]";       突破。
                 案例 UMINUS。      s   "[_]";       突破。
                 案例 '+':          s   "[+]";       突破。
                 案例 '-':          s   "[-]";       突破。
                 案例 '*':          s   "[*]";       突破。
                 案例 '/':          s   "[/]";       突破。
                 案例 '<':          s   "[<]";       突破。
                 案例 '>':          s   "[>]";       突破。
                 案例 通用电气。    s   "[>=]";      突破。
                 案例 LE。          s   "[<=]";      突破。
                 案例 NE。          s   "[!=]";      突破。
                 案例 EQ。          s   "[==]";      突破。
            }
            突破。
    }

    /*构建节点文本框 */ graphBox (s,
    &w, &h);
    cbar = c;
    *ce = c + w;
    *cm = c + w / 2。

    /* 节点是叶子 */
    如果 (p->type == typeCon || p->type == typeId || p->opr.nops == 0) {
        graphDrawBox (s, cbar, l);
        返回。
```

```
}
```

```
    /* 节点有子女 */ cs = c;
    for (k = 0; k < p->opr.nops; k++) {
        exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
        cs = che;
    }

    /* 节点总宽度 */ 如果（w <
    che - c） {
        cbar += (che - c - w) / 2;
        *ce = che;
        *cm=（c+che）/2。
    }

    /* 绘制节点 */ graphDrawBox
     （s, cbar, l）。

    /* 绘制箭头（非最佳：孩子们会被第二次绘制） */ 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;
    }
}

/* 用于绘图的接口 */

#define lmax 200
#define cmax 200

char graph[lmax][cmax]; /* 用于ASCII-图形的数组 */ int
graphNumber = 0;

空白GraphTest（int l, int c）。
{int ok; ok
    = 1;
    如果（l < 0）ok = 0。
    如果（l >= lmax）ok = 0；
    如果（c < 0）ok = 0。
    如果（c >= cmax）ok = 0；
    如果（ok）返回。
    printf ("\n+++error: l=%d, c=%d not in drawing rectangle 0, 0 ...。
%d, %d"。
        l, c, lmax, cmax）；
    退出（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;
        如果 (j < cmax-1) graph[i][j+1] = 0。
        如果 (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] = '|'。
}
```

更多莱克斯

弦乐

带引号的字符串经常出现在编程语言中。下面是在lex中匹配字符串的一种方法。

```
%{
    char *yylval;
    #include
    <string.h>.
%}
%%

           yylval = strdup(yytext+1);
           if (yylval[yyleng-2] != '"'
           )
               warning("字符串终止不当"); else
               yylval[yyleng-2] = 0;
           printf("found '%s'\n", yylval);
       }
```

上面的例子确保了字符串不跨越行的边界，并删除了包围的引号。如果我们希望添加转义序列，如 \n 或 \"，开始状态会简化问题。

```
%{
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; }
```

独有的开始状态**STRING**是在定义部分定义的。当扫描器检测到一个引号时，**BEGIN**宏将lex转移到**STRING**状态。Lex保持在**STRING**状态，只识别以**<STRING>**开头的模式，直到另一个**BEGIN**被执行。因此我们有一个扫描字符串的小环境。当识别到尾部的引号时，我们就切换回初始状态0。

保留字词

如果你的程序有大量的保留字集合，让lex简单地匹配一个字符串并在你自己的代码中确定它是一个变量或保留字是更有效的。例如，与其编码

```
"如果 "          返回IF。
"then "          返回THEN。
"else "          返回ELSE。

{letter}({letter}|{digit})* { yylval.id
         = symLookup(yytext); return
         IDENTIFIER;
     }
```

其中**symLookup**返回一个符号表的索引，最好是同时检测保留字和标识符，如下所示。

```
{字母}({letter}|{digit})* {
         int i;

         如果((i = resWord(yytext)) != 0)
             返回(i)。
         yylval.id = symLookup(yytext);
         return (IDENTIFIER);
     }
```

这种技术大大减少了所需的状态数量，并使扫描器表更小。

调试Lex

Lex有启用调试的设施。这个功能可能随lex的不同版本而变化，所以你应该查阅文档以了解细节。lex在文件**lex.yy.c**中生成的代码包括调试语句，可以通过指定命令行选项**"-d "**启用。flex（lex的GNU版本）中的调试输出可以通过设置**yy_flex_debug来**打开或关闭。输出包括应用的规则和相应的匹配文本。如果你同时运行lex和yacc，那么在你的yacc输入文件中指定以下内容。

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

另外，你也可以通过定义函数来编写你自己的调试代码，显示令牌值和**yylval**联盟的每个变量的信息。这在下面的例子中有所说明。当定义了**DEBUG**后，调试函数生效，并显示令牌和相关值的跟踪。

```
%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)。
            }

"如果"       RETURN(IF)。
"else"      RETURN(ELSE)。
```

更多Yacc

递归

一个列表可以用左递归来指定

```
列表。
      项目
      | 列表 ',' 项目
      ;
```

或右侧递归。

```
列表。
      项目
      | 项目 ',' 列表
```

如果使用右递归，那么列表上的所有项目都被推到堆栈上。在最后一个项目被推入后，我们开始减少。使用左递归，我们的堆栈上永远不会有超过三个项，因为我们边走边减。由于这个原因，使用左递归是很有利的。

If-Else歧义

一个经常发生的shift-reduce冲突涉及**if-else**结构。假设我们有以下规则。

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

和以下状态。

```
IF expr stmt IF expr stmt .ELSE stmt
```

我们需要决定是转移**ELSE**还是减少栈顶的**IF expr stmt**。如果我们转移，那么我们有

```
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 .
```

其中第二个**ELSE**与第二个**IF**配对。如果我们减少，我们有

```
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 .
```

其中第二个**ELSE**是与第一个**IF**配对的。现代编程语言将一个**ELSE**与最近的未配对的**IF配对**。因此，前者的行为是预期的。这在yacc中很有效，因为当遇到shift-reduce冲突时，默认行为是shift。

虽然yacc做了正确的事情，但也发出了shift-reduce的警告信息。为了消除该信息，给**IF-ELSE**一个比简单的**IF**语句更高的优先级。

```
%nonassoc IFX
%nonassoc ELSE

   说话。
       IF expr stmt %prec IFX
       | IF expr stmt ELSE stmt
```

错误信息

一个好的编译器会给用户提供有意义的错误信息。例如，下面的信息传达的信息不多。

语法错误

如果我们在lex中跟踪行号，那么我们至少可以给用户一个行号。

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

当yacc发现一个解析错误时，默认动作是调用**yyerror**，然后从yylex返回，返回值为1。一个更优雅的动作是将输入流冲到一个语句定界符上，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 '}'
        | 错误 ';'
        |错误 '}'
        ;
```

错误标记是yacc的一个特殊功能，它将匹配所有的输入，直到找到错误后的标记。在这个例子中，当yacc检测到语句中的错误时，它将调用**yyerror**，刷新输入到下一个分号或大括号，然后继续扫描。

继承的属性

到目前为止的例子都使用了合成的属性。在语法树的任何一点，我们都可以根据一个节点的子节点的属性来确定该节点的属性。考虑一下规则

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

由于我们是自下而上的解析，两个操作数的值都是可用的，我们可以确定与左侧相关的值。一个节点的继承属性取决于一个父节点或同级节点的值。下面的语法定义了一个C语言的变量声明。

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

下面是一个解析的样本。

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

当我们把**VAR**减少到**varlist**时，我们应该在符号表上注解变量的类型。然而，该**类型**被埋在堆栈中。这个问题可以通过索引回到栈中来解决。回顾一下，**1美元**指定了右手边的第一个项。我们可以向后索引，使用**$0**、**$-1**，以此类推。在这种情况下，**$0就可以了**。如果你需要指定一个符号类型，语法是**$<tokentype>0**，包括角括号。在这个特殊的例子中，必须注意确保类型总是在**varlist**之前。

嵌入行动

yacc中的规则可能包含嵌入式动作。

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

注意，这些动作在堆栈中需要一个槽。因此，**do_item2**必须用**3美元**来引用**item2**。在内部，这个语法被yacc转换为以下内容。

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

调试Yacc

Yacc有实现调试的设施。这个功能可能因不同版本的yacc而不同，所以一定要查阅文档以了解细节。yacc在文件**y.tab.c**中生成的代码包括调试语句，通过定义**YYDEBUG**并将其设置为非零值来启用。这也可以通过指定命令行选项**"-t "**来实现。正确设置**YYDEBUG**后，可以通过设置**ydebug来**切换调试输出。输出包括扫描的令牌和shift/reduce动作。

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

此外，你可以通过指定命令行选项**"-v "**来转储解析状态。状态被转储到文件**y.output中**，在调试语法时通常很有用。另外，你也可以通过定义**TRACE**宏来编写自己的调试代码，如下图所示。当**DEBUG**被定义时，将显示按行号计算的减少的痕迹。

```
%{
#ifdef DEBUG
#define TRACE printf("reduce at line %d\n", 线）。
#else
#define TRACE
#endif
%}

%%

statement_list。
          声明
                { 追踪$$=1; }
        | 声明_列表 声明
                { TRACE $$ = newNode('; ', 2, $1, $2); }
        ;
```

书目

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman [2006].Compilers, Prinicples, Techniques and Tools.Addison-Wesley, Reading, Massachusetts.

Gardner, Jim, Chris Retterath and Eric Gisin [1988].MKS Lex & Yacc.Mortice Kern系统公司，加拿大安大略省滑铁卢市。

Johnson, Stephen C. [1975].Yacc: Yet Another Compiler Compiler.计算科学技术报告第32号，贝尔实验室，Murray hill，新泽西。PDF版本可在ePaperPress获得。

Lesk, M. E. and E. Schmidt [1975].Lex - 词汇分析器生成器。计算科学技术报告第39号，贝尔实验室，默里山，新泽西。PDF版本可在ePaperPress获得。

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

目录

术语

词素。词素是一个字符序列，根据标记的匹配模式包含在源程序中。它只不过是一个标记的实例。

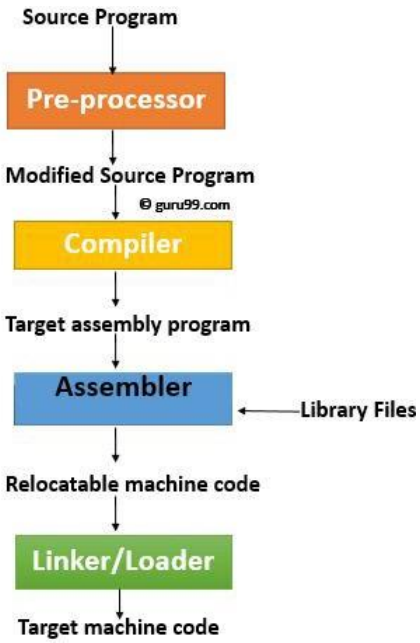
代号：编译器设计中的代号是代表源程序中一个信息单位的字符序列。

模式。模式是一种描述，它被标记所使用。如果是作为标记使用的关键词，模式是一串字符。

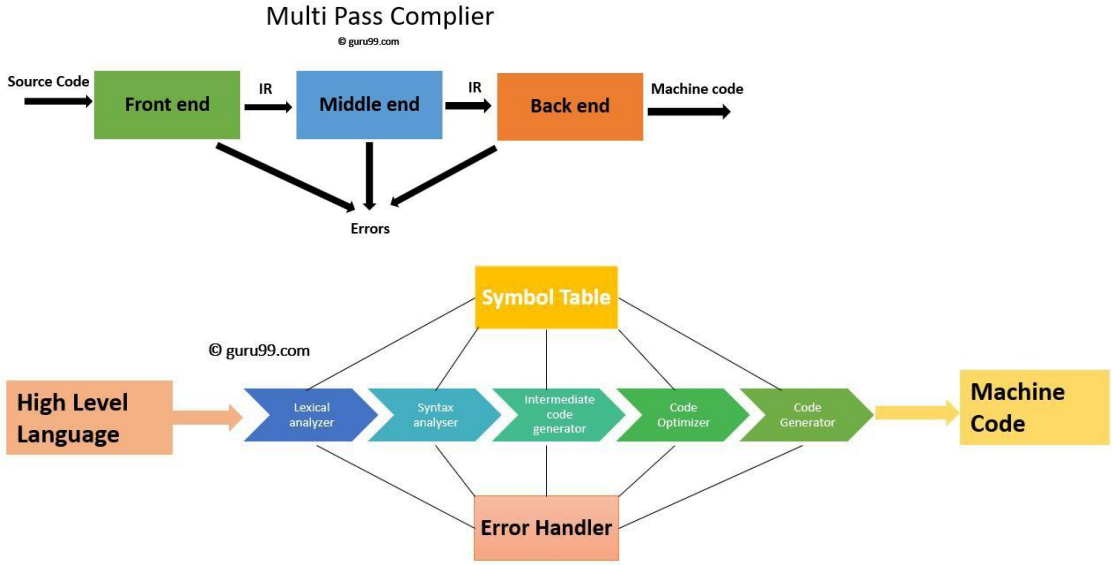
1. 介绍

1.1 编译器实现流程

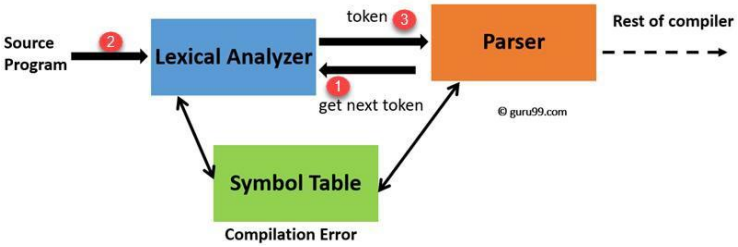
从代码到程序的流程



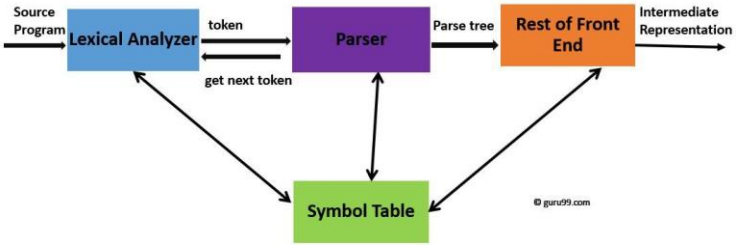
其中的编译器实现的理论模型如下



词法分析架构：



语法分析架构：



1.2 Lex 和 Yacc 命令的实现步骤

Lex和yacc是实现complier的工具，步骤如下：

1. 制定词法（Lexical）的模式的匹配规则，形成 Lex 工具输入文件 bas.l
2. 制定语法（程序员）规则，形成Yacc工具的输入文件bas.y
3. 执行生成编译器（bas.exe）的命令步骤
4. 详细过程描述如下：

```
$yacc -d bas.y                  #创建y.tab.h, y.tab.c
$lex bas.l                      #创建lex.yy.c
$cc lex.yy.c y.tab.c -obas.exe  #compile/link
```

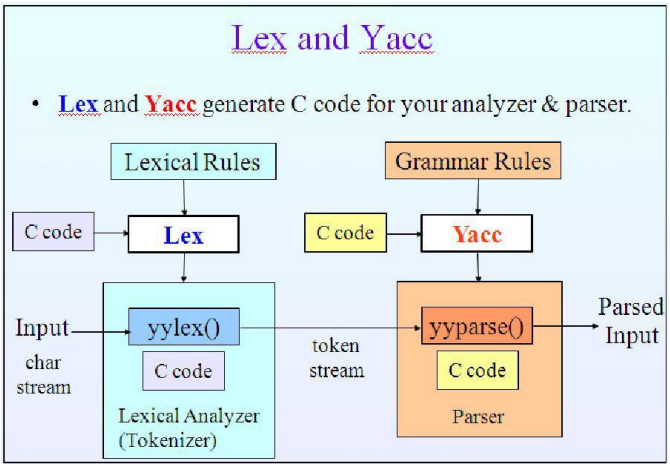
Yacc读取**bas.y**中的语法描述，并在文件y.tab.c中生成一个解析器，即函数yyparse。在文件**bas.y**中包括标记声明。这些被yacc转换为常量定义并放在文件y.tab.h中。

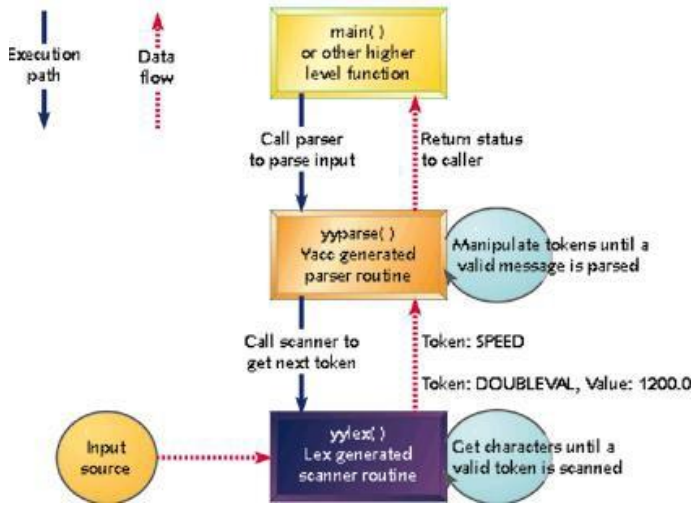
Lex读取**bas.l**中的模式描述，包括文件y.tab.h，并生成了一个词法分析器，函数 yylex，在文件lex.yy.c中。

最后，词法分析器和解析器被编译并连接在一起，形成可执行文件bas.exe。

在main中，我们调用yyparse来运行编译器。函数yyparse自动调用yylex来获得每个标记。

图示如下：





1.4 Lex 和 Yacc 的实际例子

计算器的例子如下：


calc.l


calc.y

执行命令如下

```
$ yacc -d calc.y
whf-2018@DESKTOP-RF1867I /cygdrive/d/HO_CODESVN/Tools/Shell/Lex_yacc/calc
$ lex calc.l

whf-2018@DESKTOP-RF1867I /cygdrive/d/HO_CODESVN/Tools/Shell/Lex_yacc/calc
$ gcc lex.yy.c y.tab.c -o calc.exe
y.tab.c: In function 'yyparse':
y.tab.c:1062:16: warning: implicit declaration of function 'yylex' [-Wimplicit
 1062 |       yychar = yylex ();
      |                ^~~~~

whf-2018@DESKTOP-RF1867I /cygdrive/d/HO_CODESVN/Tools/Shell/Lex_yacc/calc
$ ./calc
576*23/4
3312
```

2、Lex 命令

Unix 系统上的 lex 命令可以产生 C 程序，用于执行字符串输入的词法分析任务。这可以作为语义分析工具 yacc 的输入环节。

Lex 工具产生的 C 程序，符合 ISO C 标准规范，通常用名为 lex.yy.c。

Make工具支持lex，Lex配置源文件以".l "为文件后缀。 Make内部LFLAGS用于

指定Lex选项。

2.1 Lex的输入文件的格式

称定义的声明，以简化 *定义*
%%
扫描仪*规则*规格，以及启动条件的声明，这些将在后面的章节中解释。
%%
定义的形式是：。

名称定义

名称是一个以字母或下划线（`_'）开头的词，后面是零或多个字母、数字、`_'或`-'（破折号）。定义从名字后面的第一个非空格字符开始，一直到行的末端。定义可以

数字 [0-9]
身份证 [a-z][a-z0-9]*

定义 "DIGIT "是一个匹配单个数字的正则表达式，"ID "是一个匹配一个字母后面有零个或多个字母或数字的正则表达式。随后引用的

{位数}+"。"{位数}*。

同于

([0-9])+"."([0-9])*

并匹配一个或多个数字，后面是一个`.'，后面是零个或多个数字。灵活输入法的规则部分包含一系列的规则，其形式为：。

模式 行动

其中模式必须是无缩进的，而且动作必须在同一行开始。

最后，用户代码部分被简单地逐字复制到`lex.yy.c'。它用于调用或被扫描器调用的配套例程。这一部分的存在是可选的；如果缺少它，输入文件中的第二个`%%'也可能被跳过。

在定义和规则部分，任何缩进的文本或用`%{'和`%}'括起来的文本都被逐字复制到输出中（去掉`%{}'）。`%{}'必须单独出现在没有缩进的行中。

在规则部分，出现在第一条规则之前的任何缩进或`%{}'的文本可以用来声明属于扫描例程本地的变量，以及（在声明之后）只要进入扫描例程就要执行的代码。其他缩进或

规则部分的"%{}"文本仍然被复制到输出端，但其含义没有得到很好的定义，而且很可能导致编译时的错误（这个特性是为了符合POSIX标准而存在的；见

以下为其他此类功能）。

最简单的法律柔性输入是。

%%

生成一个扫描器，简单地将其输入（一次一个字符）复制到其输出。

3、Yacc 命令

Yacc (**Yet Another Compiler-Compiler**) 是 Unix 系统下的可执行程序。Yacc 的 GNU 的版本称为 Bison。

- 它是一种工具，将任何一种编程语言的所有语法翻译成针对此种语言的Yacc语 法 解 析 器 。
- 它用巴科斯范式(BNF, Backus Naur Form)来书写。
- 按照惯例，Yacc文件有.y后缀。创

建有四个步骤

通过在语法文件上运行Yacc 生成一个解析器。

说明语法：。

编写一个.y的语法文件（同时说明C在这里要进行的动作）。

编写一个词法分析器来处理输入并将标记传递给解析器。 这可以使用 Lex 来完成。

编写一个函数，通过调用 yyparse() 来开始解析。

编写错误处理例程（如 yyerror()）。

编译Yacc 生成的代码以及其他相关的源文件。

将目标文件链接到适当的可执行解析器库。

附录 1：AWK 开发编译器：

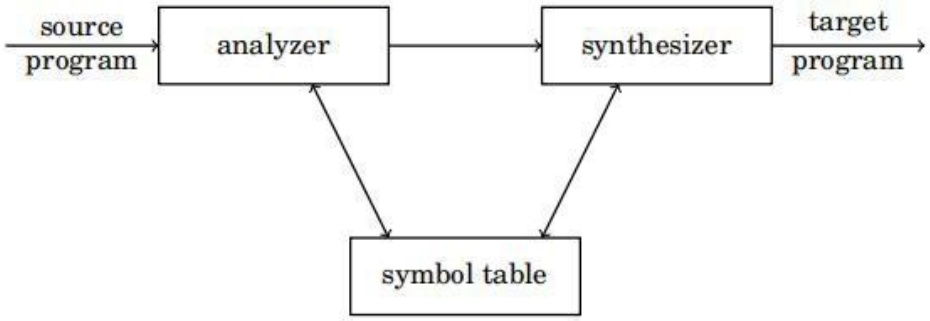
人们经常使用awk开发 "小型语言 "的翻译器("小型语言 "指的是特定于某些应用领域的专用编程语言),开发翻译器的原因主要有三点：

首先,它可以帮助你了解语言处理程序的工作流程。本章的第一个例子是一个汇编程序,虽然只有20来行,但已经包含了汇编过程的核心要素,为了执行汇编程序,我们还要开发一个解释程序。汇编程序与解释程序反映了早期阶段汇编语言与机器架构的关系。

第二个原因是在实际工作中,为了实现一个专用的编程语言,通常需要投入大量的精力与财力,不过在这之前,我们有必要测试一下新语言的语法和语义。作为示例,本章讨论了一个画图语言和一个参数设置语言,后者用于设置排序命令的参数。

最后一点是希望编程语言能够在实际的工作中发挥作用，就比如说本章所开发的计算器。

语言处理程序围绕下面这个模型构造而成:



分析器(analyzer)系统语言处理程序的前端,它负责读取源程序(sourceprogram)并将其切分成一个个词法单元,词法单元包括运算符、操作数等。分析器对源程序进行语法检查,如果源程序含有语法错误,它就会打印一条错误消息。最后,分析器把源程序转换成某种中间形式,并传递给后端(合成器),合成器(synthesizer)再根据中间形式生成目标程序(targetprogram) 。合成器在生成目标程序的过程中需要和符号表(symboltable)通信，而符号表中的内容由分析器收集而来。虽然我们把语言的处理过程描述成多个界限分明的不同步骤,但实际上，这些界限通常很模糊，而且有些步骤有可能被合并成一个。

利用 awk 为实验性语言构造处理程序非常方便，这是因为它支持许多和语言翻译相关的操作。对源程序的分析可以通过字段分割与正则表达式来完成，用关联数组管理符号表，用printf 生成目标代码。