5.5 Yacc：一个LALR(1)分析程序的生成器

分析程序生成器(parser generator)是一个指定某个格式中的一种语言的语法作为它的输入，并为该种语言产生分析过程以作为它的输出的程序。在历史上，分析程序生成器被称作编译 - 编译程序(compiler-compiler)，这是由于按照规律可将所有的编译步骤作为包含在分析程序中的动作来执行。现在的观点是将分析程序仅考虑为编译处理的一个部分，所以这个术语也就有些过时了。合并 LALR(1)分析算法是一种常用的分析生成器，它被称作 Yacc(yet another compiler-compiler)。本节将给出Yacc的概貌来，下一节将使用 Yacc为TINY语言开发一个分析程序。由于Yacc有许多不同的实现以及存在着通常称作Bison的若干个公共领域版本⊖，所以在它的运算细节中有许多变化，而这又可能与这里所用的版本有些不同 ⊜。

5.5.1 Yacc基础

Yacc取到一个说明文件(通常带有一个**.y**后缀)并产生一个由分析程序的C源代码组成的输出文件(通常是在一个称作**y.tab.c**或**ytab.c**或更新一些的<文件名>**.tab.c**的文件中，而<文件名>**.y**则是输入文件)。Yacc说明文件具有基本格式

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

因此就有3个被包含了双百分号的行分隔开来的部分——定义部分、规则部分和辅助程序部分。

定义部分包括了Yacc需要用来建立分析程序的有关记号、数据类型以及文法规则的信息。它还包括了必须在它的开始时直接进入输出文件的任何 C代码 (主要是其他源代码文件的**#include**指示)。说明文件的这个部分可以是空的。

⊖ 一个流行版本——Gnu Bison——是由Free Software Foundation发布的Gnu软件的一个部分，请参见“注意与参考”部分。

⊜ 实际上，我们已使用了若干个不同的版本来生成后面的示例。

China-pub.com 下载

规则部分包括修改的BNF格式中的文法规则以及将在识别出相关的文法规则时被执行的 C代码中的动作(即：根据LALR(1)分析算法，在归约中使用)。文法规则中使用的元符号惯例如下：通常，竖线被用作替换(也可分别写出替换项)。用来分隔文法规则的左右两边的箭头符号→在Yacc中被一个冒号取代了，而且必须用分号来结束每个文法规则。

第3点：辅助程序部分包括了过程和函数声明，除非通过 **#include**文件，否则它们会不适用，此外还需要被用来完成分析程序和 /或编译程序。这个部分也可为空，此时第 2个双百分号元符号可从说明文件中省略掉。因此，最小的 Yacc说明文件可仅由后面带有文法规则和动作(若仅是要分析文法也可省掉动作，本节稍后再讲到它)的%%组成。

Yacc还允许将C-风格的注解插入到说明文件的任何不妨碍基本格式的地方。

利用一个简单的示例，我们可将 Yacc说明文件的内容解释得更详细一些。这个示例是带有文法

exp → *exp addop term* | *term*
addop→ **+** | **-**
term → *term mulop factor* | *factor*
mulop → ***
factor → **(** *exp* **)** | ***number***

的简单整型算术表达式的计算器。这个文法在前面的章节中已用得很多了。在 4.1.2节中，我们为这个文法开发了一个递归下降计算器程序。程序清单 5-1给出了完全等价的Yacc说明。下面按顺序讨论这个说明中3个部分的内容。

程序清单5-1的定义部分有两个项目。第1个项目由要在Yacc输出的开始处插入的代码组成。这个代码是由两个典型的**#include**指示组成，且由在其前后的分隔符 **%{**和**%}**将它与这个部分中的其他Yacc说明分开(请注意百分号在括号之前)。定义部分的第2个项目是记号**NUMBER**的声明，它代表数字的一个序列。

程序清单5-1　一个简单计算器程序的Yacc定义

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%%

command : exp      { printf("%d\n",$1);}
        ; /* allows printing of the result */

exp     : exp '+' term   {$$ = $1 + $3;}
        | exp '-' term   {$$ = $1 - $3;}
        | term    {$$ = $1;}
        ;

term    : term '*' factor {$$ = $1 * $3;}
        | factor  {$$ = $1;}
        ;
```

China-pub.com
下载


```
factor      : NUMBER          {$$ = $1;}
            | '(' exp ')'     {$$ = $2;}
            ;
%%

main()
{ return yyparse();
}

int yylex(void)
{ int c;
  while((c = getchar()) == ' ');
  /* eliminates blanks */
  if ( isdigit(c) ) {
    ungetc(c,stdin);
    scanf("%d",&yylval);
    return(NUMBER);
  }
  if (c == '\n') return 0;
  /* makes the parse stop */
  return(c);
}

void yyerror(char * s)
{ fprintf(stderr,"%s\n",s);
} /* allows for printing of an error message */
```

Yacc用两种方法来识别记号。首先，文法规则的单引号中的任何字符都可被识别为它本身。因此，单字符记号就可直接被包含在这个风格的文法规则中，正如程序清单 5-1中的运算符记号**+**、**-**和***** (以及括号记号)。其次，可在Yacc的%记号(**%token**)中声明符号记号，如程序清单5-1中的记号**NUMBER**。这样的记号被Yacc赋予了不会与任何字符值相冲突的数字值。典型地，Yacc开始用数字258给记号赋值。Yacc将这些记号定义作为**#define**语句插入到输入代码中。因此，在输出文件中就可能会找到行

```
#define NUMBER 258
```

作为Yacc对说明文件中的**%token NUMBER**声明的对应。Yacc坚持定义所有的符号记号本身，而不是从别的地方引入一个定义。但是却有可能通过在记号声明中的记号名之后书写一个值来指定将赋给记号的数字值。例如，写出

```
%token NUMBER 18
```

就将给**NUMBER**赋值18 (不是258)。

在程序清单5-1的规则部分中，我们看到非终结符exp、term和factor的规则。由于还需要打印出一个表达式的值，所以还有另外一个称为 command的规则，而且将其与打印动作相结合。因为首先列出了command的规则，所以command则被作为文法的开始符号。若不这样，我们还可在定义部分中包括行

```
%start command
```

此时就不必将*command*的规则放在开头了。

Yacc中的动作是由在每个文法规则中将其写作真正的 C代码(在花括号中)来实现的。通常，尽管也有可能在一个选择中写出嵌入动作 (embedded action)(稍后将讨论它)，但动作代码仍是放在每个文法规则选择的末尾 (但在竖线或分号之前)。在书写动作时，可以享受到 Yacc伪变量

China-pub.com 下载

(pseudovariable)的好处。当识别一个文法规则时，规则中的每个符号都拥有一个值，除非它被参数改变了，该值将被认为是一个整型 (稍后将会看到这种情况)。这些值由Yacc保存在一个与分析栈保持平行的值栈 (value stack)中。每个在栈中的符号值都可通过使用以 $开始的伪变量来引用。**$$**代表刚才被识别出来的非终结符的值，也就是在文法规则左边的符号。伪变量**$1**、**$2**、**$3**等等都代表了文法规则右边的每个连续的符号。因此在程序清单 5-1中，文法规则和动作

```
exp    : exp '+' term { $$ = $1 + $3; }
```

就意味着当识别规则*exp*→*exp* **+** *term*时，就将左边的*exp* 的值作为*exp* 的值与右边的*term* 的值之和。

所有的非终结符都是通过这样的用户提供的动作来得到它们的值。记号也可被赋值，但这是在扫描过程中实现的。 Yacc假设记号的值被赋给了由 Yacc内部定义的变量**yylval**，且在识别记号时必须给**yylval**赋值。因此，在文法和动作

```
factor    : NUMBER    { $$ = $1;}
```

中，值**$1**指的是当识别记号时已在前面被赋值为**yylval**的**NUMBER**记号的值。

程序清单5-1的第3个部分(辅助程序部分)包括了3个过程的定义。第1个是**main**的定义，之所以包含它是因为 Yacc输出的结果可被直接编译为可执行的程序。过程 **main**调用**yyparse**，**yyparse**是Yacc给它所产生的分析过程起的名称。这个过程被声明是返回一个整型值。当分析成功时，该值总为 0；当分析失败时，该值为 1(即发生一个错误，且还没有执行错误恢复)。Yacc生成的**yyparse**过程接着又调用一个扫描程序过程，该过程为了与 Lex扫描程序生成器相兼容，所以就假设叫作**yylex** (参见第2章)。因此，程序清单5-1中的Yacc说明还包括了**yylex**的定义。在这个特定的情况下， **yylex** 过程非常简单。它所需要做的只有返回下一个非空字符；但若这个字符是一个数字，此时就必须识别单个元字符记号 **NUMBER**并返回它在变量**yylval**中的值。这里有一个例外：由于假设一行中输入了一个表达式，所以当扫描程序已到达了输入的末尾时，输入的末尾将由一个新行字符（在 C中的‘ **\n** ’）指出。Yacc希望输入的末尾通过**yylex**由空值0标出(这也是Lex所共有的一个惯例)。最后就定义了一个**yyerror**过程。当在分析时遇到错误时， Yacc就使用这个过程打印出一个错误信息 (典型地， Yacc打印串“ 语法错误 ”，但这个行为可由用户改变)。

5.5.2 Yacc选项

除了**yylex**和**yyerror**之外， Yacc通常需要访问许多辅助过程，而且它们经常是被放在外置的文件之中而非直接放在 Yacc说明文件中。通过写出恰当的头文件以及将 **#include**指示放在Yacc说明的定义部分中，就可以很容易地使 Yacc访问到这些过程。将 Yacc特定的定义应用到其他文件上就要复杂一些了，在记号定义时尤为如此。此时正如前面所讲的， Yacc坚持自己生成(而不是引入)，但是它又必须适用于编译程序的许多其他部分 (尤其是扫描程序)。正是由于这个原因， Yacc就有一个可用的选项，自动产生包含了该信息的头文件，而这个头文件将被包括在需要定义的任何其他文件中。这个头文件通常叫作 **y.tab.h**或**ytab.h**，并且它与**-d**选项(用于heaDer文件)一起生成。

例如，若文件**calc.y**包括了程序清单5-1中的Yacc说明，则命令

```
yacc -d calc.y
```

将产生(除了文件**y.tab.c**文件之外)内容各异的文件**y.tab.h**(或相似的名称)，但却总是包括

China-pub.com
下载

下示的内容：

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

(稍后将详细地描述**YYSTYPE**的含义)。这个文件可被用来通过插入行

```
#include y.tab.h
```

到文件中而将**yylex**的代码放在另一个文件中⊖。

Yacc的另一个且极为有用的选项是详细选项(verbose option)，它是由命令行中的**-v**标志激活。这个选项也产生另一个名字为**y.output**(或类似名称的)的文件。这个文件包括了被分析程序使用的LALR(1)分析表的文本描述。阅读这个文件将使得用户可以准确地判断出 Yacc生成的分析程序在任何情况下将会采取的动作，而且这是处理文法中的二义性和不准确性的极为有效的方法。在向说明添加动作或辅助过程之前， Yacc与这个选项一起在文法上运行以确保 Yacc生成的分析程序将确实如希望的那样执行的确是一个好办法。

例如，程序清单5-2中的Yacc说明。这是程序清单5-1中Yacc说明的基本版本。当同时使用Yacc和详细选项：

```
yacc -v calc.y
```

程序清单5-2 使用-V选项的Yacc说明提纲

```
%token NUMBER
%%
command     : exp
            ;
exp         : exp '+' term
            | exp '-' term
            | term
            ;
term        : term '*' factor
            | factor
            ;
factor      : NUMBER
            | '(' exp ')'
            ;
```

时，这两个说明生成相同的输出文件。程序清单 5-3是该文法完整的典型**y.output**文件⊜。下面的段落将讨论如何解释这个文件。

Yacc输出文件列出了 DFA中的所有状态，此外还有内部统计的小结。状态由 0开始编号。输出文件在每个状态的下面列出了核心项目 (并未列出闭包项目)，其次是与各个先行对应的动作，最后则是各个非终结符的 goto动作。Yacc尤其使用一个下划线字符 _来标出项目中的显著

⊖ Yacc的早期版本可能只将记号的定义 (但没有**yylval**的定义)放置在**y.tab.h**中。这可能将要求一个共同的工作区或重新安排代码。

⊜ Bison的较新版本在输出文件中产生了一个根本不同的格式，但是内容却基本相同。

China-pub.com 下载

的位置，用它来代替本章所用到的句点，却用句点来指明缺省，或“不在意”每个状态列表的动作部分中的先行记号。

程序清单5-3 为程序清单5-1中的Yacc说明使用详细选项生成的典型y.output 文件

```
state 0
        $accept : _command $end

        NUMBER  shift 5
        (  shift 6
        .  error

        command  goto 1
        exp  goto 2
        term  goto 3
        factor  goto 4

state 1
        $accept :  command_$end

        $end  accept
        .  error

state 2
        command :  exp_    (1)
        exp :  exp_+ term
        exp :  exp_- term

        +  shift 7
        -  shift 8
        .  reduce 1

state 3
        exp :  term_    (4)
        term :  term_* factor

        *  shift 9
        .  reduce 4

state 4
        term :  factor_    (6)

        .  reduce 6

state 5
        factor :  NUMBER_    (7)

        .  reduce 7

state 6
        factor :  (_exp )

        NUMBER  shift 5
        (  shift 6
        .  error

        exp  goto 10
        term  goto 3
        factor  goto 4

state 7
        exp :  exp +_term

        NUMBER  shift 5
        (  shift 6
        .  error

        term  goto 11
        factor  goto 4

state 8
        exp :  exp -_term

        NUMBER  shift 5
        (  shift 6
        .  error

        term  goto 12
        factor  goto 4

state 9
        term :  term *_factor

        NUMBER  shift 5
        (  shift 6
        .  error

        factor  goto 13

state 10
     exp :  exp_+ term
     exp :  exp_- term
     factor :  ( exp_)

     +  shift 7
     -  shift 8
     )  shift 14
     .  error

state 11
     exp :  exp + term_    (2)
     term :  term_* factor

     *  shift 9
     .  reduce 2
```

China-pub.com
下载


```
state 12
     exp :  exp - term_    (3)
     term :  term_* factor

     *  shift 9
     .  reduce 3

state 13
     term :  term * factor_   (5)

     .  reduce 5

state 14
     factor :  ( exp )_    (8)

       .  reduce 8

8/127 terminals, 4/600 nonterminals
9/300 grammar rules, 15/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
9/601 working sets used
memory: states, etc. 36/2000, parser 11/4000
9/601 distinct lookahead sets
6 extra closures
18 shift entries, 1 exceptions
8 goto entries
4 entries saved by goto default
Optimizer space used: input 50/2000, output 218/4000
218 table entries, 202 zero
maximum spread: 257, maximum offset: 43
```

Yacc通过列出扩充产生式的初始项目而从状态 0开始，而这通常在DFA的开始状态中只有核心项目。在上面示例的输出文件中，这个项目写作

```
$accept :_command $end
```

它与我们自己的术语中的项目 *command'→.command* 对应。Yacc为扩充的非终结符提供的名字为**$accept**。它还将输入结尾的伪记号显式地列为**$end**。

首先大致地看一下状态0的动作部分，它后面是核心项目的列表：

```
NUMBER shift 5
( shift 6
. error
command goto 1
exp goto 2
term goto 3
factor goto 4
```

上面的列表指出了DFA移进到先行记号**NUMBER**的状态5中、移进到先行记号 **(**的状态6中，并且说明其他所有先行记号中的错误。此外为了在归约到所给出的非终结符中使用还列出了 4个goto转换。这些动作与用这章中的方法手工构造分析表中的内容完全一样。

再看看状态2，它有输出列表

```
state 2
command : exp_ (1)
exp : exp_+ term
exp : exp_- term
+ shift 7
- shift 8
. reduce 1
```

这里的核心项目是一个完整的项目，所以在动作部分中有一个用相关的产生式选择实现的归约。为了提醒我们在归约中所使用的产生式的编号， Yacc在完整的项目之后列出了编号。在这种情

China-pub.com 下载

况下，产生式编号就是1，而且有一个表示用产生式 *command* → *exp*实现的**reduce 1**动作。Yacc总是按照它们在说明文件中所列的顺序为产生式编号。在我们的示例中，有 8个产生式(*command*的一个，*exp*的3个，*term*的两个以及*factor*的两个)。

请注意，在这个状态中的归约动作是一个缺省动作：一个将在任何不是 **+**或**-**的先行之上的归约。这里的Yacc与一个单纯的LALR(1)分析程序(而且甚至是SLR(1)分析程序)的不同在于它并不试着去检查归约上的合法先行(而是在若干个归约中决定)。Yacc分析程序将在最终声明错误之前在错误之上作出多个归约(这将是在任何的更多的移进发生之前的最后行为)。这就意味着错误信息可能不是像它们应该的那样有信息价值，但是分析表却会变得十分简单，这是因为情况发生得更少了(这点将在5.7节中再次讨论到)。

在这个示例最后，我们从Yacc输出文件中构造出一个分析表，该表与本章早些时候手工写出的完全一样。表5-11就是这个分析表。

表5-11 与程序清单5-3的Yacc输出对应的分析表

状 态	输 入							Goto			
	NUMBER	**(**	**+**	**-**	*****	**)**	$	*command*	*exp*	*term*	*factor*
0	s5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			s7	s8		S14					
11	r2	r2	r2	r2	s9	r2	r2				
12	r3	r3	r3	r3	s9	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				
14	r8	r8	r8	r8	r8	r8	r8				

5.5.3 分析冲突与消除二义性的规则

详细选项的一个重要用处是Yacc将在**y.output**文件中报告调查的分析冲突。Yacc在其中建立了消除二义性的规则，该规则将允许它甚至在分析冲突发生时产生一个分析程序 (因此，甚至这也是用于二义性文法的)。这些消除二义性的规则通常总能做对，但有时也会出错。对**y.output**文件的检查使用户判断出分析冲突是什么，以及由Yacc产生的分析程序是否可正确地解决问题。

China-pub.com
下载

程序清单5-1的示例中并没有分析冲突，在输出文件结尾的小结信息中， Yacc将它报告为

```
0 shift / reduce, 0 reduce / reduce conflicts reported
```

例5.12中的二义性悬挂else文法是一个更为有趣的示例。在表5-9中，我们为这个文法给出了SLR(1)分析表，在其中通过选取移进而不是归约将状态5中的移进-归约冲突消除了(它与最近嵌套消除二义性的规则对应)。Yacc以完全相同的术语报告了二义性，并且通过相同的消除二义性的规则解决了二义性。除了Yacc插入到错误项中的缺省归约之外，由Yacc报告的分析表确实与表5-9相同。例如， Yacc将状态5的动作在**y.output**文件中报告如下(记号在说明文件中的定义用小写字母写出，这样就避免了与C的保留字相冲突)：

```
5: shift / reduce conflict ( shift 6, red'n 3) on ELSE
state 5
      I : IF S_(3)
      I : IF S_ELSE S
      ELSE shift 6
      . reduce 3
```

在小结信息中， Yacc还报告了一个移进-归约冲突：

```
1 shift / reduce, 0 reduce / reduce conflicts reported
```

在归约-归约冲突的情况下， Yacc 通过执行由文法规则在说明文件中首先列出的归约来消除二义性。尽管它也会带来正确的分析程序，这仍与在文法中的错误很相似。下面是一个简单的示例。

例5.18 考虑以下的文法：

$$S \rightarrow A \mid B$$
$$A \rightarrow a$$
$$B \rightarrow a$$

由于单个合法串a有两个派生：$S \Rightarrow A \Rightarrow a$和$S \Rightarrow B \Rightarrow a$，所以这是一个有二义性的文法。程序清单5-4是这个文法完整的**y.output**文件。请注意，在状态4中的归约-归约冲突，它由在规则$B \rightarrow a$之前执行规则$A \rightarrow a$来解决。这样就导致了后面的这个规则永远不会用在归约之中(它很明确地指出文法的一个问题)。Yacc在最后报告了这项情况以及行

```
Rule not reduced : B : a
```

程序清单5-4 例5.18中文法的Yacc输出文件

```
   Rule not reduced:  B : a

state 0
      $accept : _S $end

      a  shift 4
      .  error

      S  goto 1
      A  goto 2
      B  goto 3

state 1
      $accept :  S_$end
```

China-pub.com 下载

```
        $end   accept
        .   error

state 2
        S :   A_     (1)

        .   reduce 1

state 3
        S :   B_     (2)

        .   reduce 2

4: reduce/reduce conflict (red'ns 3 and 4 ) on $end
state 4
        A :   a_     (3)
        B :   a_     (4)

        .   reduce 3

Rule not reduced:    B :   a

3/127 terminals, 3/600 nonterminals
5/300 grammar rules, 5/1000 states
0 shift/reduce, 1 reduce/reduce conflicts reported
...
```

除了前面已提到过的消除二义性的规则之外，Yacc为指定与一个有二义性的文法相分隔开的算符优先及结合性还具有特别的机制。它具有一些优点。首先，文法无需包括指定了结合性和优先权的显式构造，而这就意味着文法可以短一些和简单一些了。其次，相结合的分析表也可小一点且得出的分析程序更有效。

例如程序清单5-5中的Yacc说明。在那个图中，文法是用没有算符的优先权和结合性的具有二义性的格式书写。相反地，算符的优先权和结合性通过写出行

```
%left '+' '-'
%left '*'
```

在定义部分中给出来。这些行向 Yacc指出算符+和-具有优先权且是左结合的，而且运算符*是左结合且有比+和-更高的优先权(因为在说明中，它是列在这些算符之后)。在Yacc中，其他可能的算符说明是%right 和%nonassoc ("nonassoc" 意味着重复的算符不允许出现在相同的层次上)。

程序清单5-5　带有二义性文法和算符的优先权及结合性规则的简单计算器的 Yacc说明

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%left '+' '-'
%left '*'
```

China-pub.com
下载


```
%%

command : exp       { printf("%d\n",$1);}
        ;

exp     : NUMBER        {$$ = $1;}
        | exp '+' exp   {$$ = $1 + $3;}
        | exp '-' exp   {$$ = $1 - $3;}
        | exp '*' exp   {$$ = $1 * $3;}
        | '(' exp ')'   {$$ = $2;}
        ;

%%
/* auxiliary procedure declarations as in Figure 5.10 */
```

5.5.4 描述Yacc分析程序的执行

除了在**y.output**文件中显示分析表的详细选项之外， Yacc生成的分析程序也可能打印出它的执行过程，这其中包括了分析栈的一个描述以及与本章早先给出的描述类似的分析程序动作。这是通过用符号定义的**YYDEBUG**编译**y.tab.c**(例如通过使用**-DYYDEBUG**编译选项)以及在需要描述信息的地方将Yacc整型变量**yydebug**设置为1。例如假设表达式**2+3**是输入，请将下面的行添加到程序清单5-1的**main**过程的开头

```
extern int yydebug;
yydebug = 1;
```

将会使得分析程序产生一个与程序清单 5-6相似的输出。我们希望读者利用表 5-11的分析表手工构造出分析程序的动作描述并将它与这个输出作一对比。

程序清单5-6 假设有输入2 + 3，利用yydebug为由程序清单5-1生成的Yacc分析程序描绘输出

```
Starting parse
Entering state 0
Input: 2+3
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0
Entering state 4
Reducing via rule 6, factor -> term
state stack now 0
Entering state 3
Next token is '+'
Reducing via rule 4, term -> exp
state stack now 0
Entering state 2
Next token is '+'
Shifting token '+', Entering state 7
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0 2 7
Entering state 4
Reducing via rule 6, factor -> term
```

China-pub.com 下载

```
state stack now 0 2 7
Entering state 11
Now at end of input.
Reducing via rule 2, exp '+' term -> exp
state stack now 0
Entering state 2
Now at end of input.
Reducing via rule 1, exp -> command
5
state stack now 0
Entering state 1
Now at end of input.
```

5.5.5 Yacc中的任意值类型

程序清单5-1使用与文法规则中的每个文法符号相关的 Yacc伪变量指出计算器的动作。例如，用写出**$$ = $1 + $3**将一个表达式的值设置为它的两个子表达式值之和(在文法规则exp → exp **+** term的右边的位置1和位置3)。由于这些值的Yacc缺省类型总是整型，所以只要处理的值是整型就可以了。但是若要用浮点值计算一个计算器，那就不合适了。在这种情况下，必须在说明文件中对 Yacc伪变量的值类型进行重定义。这个数据类型总是由 C预处理器符号**YYSTYPE**在Yacc中定义。重定义这个符号会恰当地改变 Yacc值栈的类型。因此若需要一个计算浮点值的计算器，就必须在Yacc说明文件的定义部分的括号**%{...%}**中添加行

```
#define YYSTYPE double
```

在更复杂的情况下，不同的文法规则就有可能需要不同的值。例如，假设希望将算符挑选的识别与用那些算符计算的规则分隔开，如在规则

$$exp \rightarrow exp\ addop\ term \mid term$$
$$addop \rightarrow \mathbf{+} \mid \mathbf{-}$$

(它们实际上是表达式文法的原始规则，我们把它们改变了以直接识别程序清单 5-1中的exp规则的算符)。现在addop必须返回算符(一个字符)，但exp必须返回计算的值(即一个**double**)，但这两个数据类型不同。我们所需要做的是将 **YYSTYPE**定义为**double**与**char**的联合。有两种方法可以办到。其一是在Yacc说明中利用Yacc的**%union**声明来直接声明一个联合：

```
%union { double val;
         char op; }
```

现在Yacc需要被告知每个非终结符的返回类型，这是利用定义部分中的**%type**指示来实现的：

```
%type <val> exp term factor
%type <op> addop mulop
```

请注意，在Yacc的**%type**声明中，联合域的名称是怎样被尖括号括起来的。接着将程序清单5-1的Yacc说明修改成按下开始(我们将详细的写法留在练习中)：

```
...
%token NUMBER

%union { double val;
         char op; }

%type <val> exp term factor NUMBER
%type <op> addop mulop
```

China-pub.com
下载


```
%%
comand : exp      { printf ( " %d\n " , $1 ) ; }
       ;
exp  : exp op term { swithc ( $2 ) {
                     case '+': $$ = $1 + $3; break;
                     case '-': $$ = $1 - $3; break;
                     }
                   }
     | term { $$ = $1; }
     ;
op : '+' { $$ '+'; }
   | '-'{ $$ '-'; }
   ;
```

第2种方法是在另一个包含文件 (例如，一个头文件)中定义一个新的数据类型之后再将YYSTYPE定义为这个类型。接着必须在相关的动作代码中手工地构造出恰当的值来。下一节中的TINY分析程序是它的一个示例。

5.5.6 Yacc中嵌入的动作

在分析时，有时需要在完整地识别一个文法规则之前先执行某个代码。例如考虑简单声明的情况：

decl → *typevar-list*
type → **int** | **float**
var-list → *var-list* **, id** | **id**

当识别var-list时，用当前类型(整型和浮点)将标签添加于每个变量标识符上。可按如下方法在Yacc中完成：

```
dec1  : type { current_type = $1 ; }
        var_list
      ;
type  : INT { $$ = INT_TYPE ; }
      | FLOAT { $$ = FLOAT_TYPE ; }
      ;
var_list : var_list ';' ID
           { setType (tokenString, current_type);}
         | ID
           { setType (tokenString, current_type);}
         ;
```

请注意，在**decl**规则中识别变量之前，一个嵌入的动作如何设置**current_type**。读者将在后面的章节中看到其他有关嵌入动作的示例。

Yacc将一个嵌入动作

```
A : B { /* embedded action */ } C ;
```

解释为与一个新的占位符非终结符相等价，且与在被归约时，执行嵌入动作的非终结符的 ε- 产生式等价：

```
A : B E C ;
E : { /* embedded action */ } ;
```

China-pub.com 下载

最后，在表5-12中小结了Yacc的定义机制以及已讨论过的一些内置名称。

表5-12 Yacc内置名称和定义机制

Yacc的内置名称	含义／用处
`y.tab.c`	Yacc输出文件名称
`y.tab.h`	Yacc生成的头文件，包含了记号定义
`yyparse`	Yacc分析例程
`yylval`	栈中当前记号的值
`yyerror`	由Yacc使用的用户定义的错误信息打印机
`error`	Yacc错误伪记号
`yyerrok`	在错误之后重置分析程序的过程
`yychar`	包括导致错误的先行记号
`YYSTYPE`	定义分析栈的值类型的预处理器符号
`yydebug`	变量，当由用户设置为1时则导致生成有关分析动作的运行信息
Yacc的定义机制	**含义／用处**
`%token`	定义记号预处理器符号
`%start`	定义开始非终结符符号
`%union`	定义和`YYSTYPE`，允许分析程序栈上的不同类型的值
`%type`	定义由一个符号返回的和类型
`%left %right %nonassoc`	定义算符的结合性和优先权(由位置)

5.6 使用Yacc生成TINY分析程序

TINY的语法已在3.7节中给出，且在4.4节中也已给出了一个手写的分析程序；我们希望读者能掌握这些内容。这里将描述Yacc说明文件`tiny.y`以及对全程定义`globals.h`的修改(我们已采用了一些方法将对其他文件的改变定为最小，这也将讲到)。整个`tiny.y`文件都列在了附录B的第4000行到第4162行中。

首先讨论TINY的Yacc说明中的定义部分。稍后将谈到标志YYPARSER(第4007行)。表示了Yacc在程序中的任何地方都需要的信息有4个`#include`文件(第4009行到第4012行)。定义部分有4个其他声明。第1个(第4014行)是`YYSTYPE`的定义，它定义了通过使Yacc分析过程为指向节点结构的指针返回的值(`TreeNode`本身被定义在`globals.h`中)，这样就允许了Yacc分析程序构造出一个语法树。第2个是全程savedName变量的定义，它被用作暂时储存要被插入到还没构造出的树节点中的标识符串，而此时已能在输入中看到这些串了(在TINY中只有在赋值中才需要)。变量`savedLineNo`也是被用作相同目的，所以那个恰当的源代码行数也将与标识符关联。最后，`savedTree`被用来暂时储存由`yyparse`过程产生的语法树(`yyparse`本身可以仅返回一个整型标志)。

下面讨论一下与TINY的每个文法规则相结合的动作(这些规则与第3章的程序清单3-1中所给出的BNF略有不同)。在绝大多数情况下，这些动作表示与该点上的分析树相对应的语法树的构造。特别地，需要从`util`包调用到`newStmtNode`和`newExpNode`来分配新的节点(这些已在4.4节中讲述过了)，而且也需要指派新树节点的合适的子节点。例如，与TINY的`write_stmt`(第4082ff行)相对应的动作如下所示：

China-pub.com
下载


```
write_stmt : WRITE exp
               { $$ = newStmtNode (WriteK);
                 $$ ->child [0] = $2;
               }
           ;
```

第1个指令调用**newStmtNode**并指派返回的值为**write_stmt**的值。接着**exp** (Yacc伪变量**$2**是指向将要被打印的表达式的树节点的指针)前面构造的值为write语句的树节点的第1个孩子。其他的语句和表达式的动作代码十分类似。

program、**stmt_seq**和**assign_stmt**的动作处理与每个这些构造相关的小问题。在**program**的文法规则中，相关的动作(第4029行)是

```
{savedTree = $1;}
```

它将**stmt_seq**构造的树赋给了静态变量**savedTree**。由于它使得语法树可由**parse**过程之后返回，所以这是必要的。

在**assign_stmt**的情况中，我们早已指出需要储存作为赋值目标的变量的标识符串，这样当构造节点时 (以及为了便于今后的描绘，还编制了它的行号)它就是恰当的了。通过使用**savedName**和**saveLineNO**静态变量(第4067行)可以做到这一点：

```
assign_stmt : ID { savedName = copyString ( tokenString ) ;
                   savedLineNo = lineno ; }
              ASSIGN exp
                 { $$ = newStmtNode ( AssignK );
                   $$ ->child [ 0 ] = $4 ;
                   $$ ->attr.name = savedName ;
                   $$ ->lineno = saveLineNo ;
                 }
            ;
```

由于作为被匹配的新记号， **tokenString**和**lineno**的值都被扫描程序改变了，所以标识符串和行号必须作为一个在 **ASSIGN**记号识别之前的一个嵌入动作储存起来。但是只有在识别出**exp**之后才能完全构造出赋值的新节点，因此就需要 **savedName**和**saveLineNo**。(实用程序过程**copyString**的使用确保了这些串没有共享存储。读者还需注意将 **exp**的值认为是**$4**。这是因为 Yacc认为嵌入动作在文法规则的右边是一个额外的位置——参见上一节的讨论)。

在**stmt_seq**(第4031行到第4039行)的情况中，它的问题是：属指针 (而不是孩子指针)将语句在TINY语法树中排在一起。因为人们将语法序列的规则写成左递归的，这也就要求为了在末尾附上当前的语句，代码应找出早先为左子集构造的属列表。这样做的效率并不高而且我们还可以通过将规则重写为右递归来避免它，但是这个解决方法也有它自身的问题：只要处理语句序列，其中的分析栈就会变得很大。

最后， Yacc说明的辅助过程部分 (第4144行到第4162行)包括了3个过程——**yyerror**、**yylex**和**parse**——的定义。**parse**过程是由主程序调用，它将调用 Yacc定义的分析过程**yyparse**并且返回保存的语法树。之所以需要 **yylex**过程是因为Yacc假设这是扫描程序过程的名称，而它又在外部被定义为**getToken**。写出了这个定义就使得Yacc生成的分析程序可在对别的代码文件只作出最小的改变的情况下就可与 TINY编译程序一起工作。有人可能希望对扫描程序作出恰当的改变并省掉这个定义，特别是在使用扫描程序的 Lex版本时。在出现错误时，**yyerror**过程由Yacc调用：它将一定的有用信息(如行号)打印到列表文件上。它使用的是

China-pub.com 下载

Yacc的内置变量**yychar**，**yychar**包含了引起错误的记号的记号号码。

我们还需要描述对TINY分析程序中的其他文件的改变，由于使用了 Yacc来产生分析程序，所以这些改变也是必要的。正如前面已指出的，我们的目标是使这些改变成为最小的，而且将所有的改变都限制为**globals.h**文件的。修改过的文件列在了附录 B的第4200行到第4320行中。其基本问题是由 Yacc生成的包含了记号定义的头文件必须被包括在大多数的其他代码文件中，但它又不能被直接包括在 Yacc生成的分析程序中，因为这样做会重复内置的定义。对上面问题的解决办法是用一个标志 **YYPARSER**(第4007行)的定义作为Yacc说明部分的开头，而**YYPARSER**位于Yacc分析程序中且指出C编译程序何时处于分析程序之中。我们使用那个标志 (第4226行到第 4236行)有选择地将 Yacc生成的头文件 **y.tab.h**包括在**globals.h**中。

第2个问题出现在**ENDFILE**记号中，此时扫描程序要指出输入文件的结尾。 Yacc假设这个记号总是存在着值 0，因此也就提供了这个记号的直接定义 (第4234行)并且将它包括在由**YYPARSER**控制的有选择性的编译部分之中，这是因为Yacc内部并不需要它。

因为所有的Yacc记号都有整型值，所以**globals.h**文件最后的改变是将**TokenType**重定义为int的一个同义字(第4252行)。这样就避免了无必要地替代前面所列的其他文件中的类型**TokenType**。