【附录 C】语法分析程序自动产生器 YACC 的使用方法

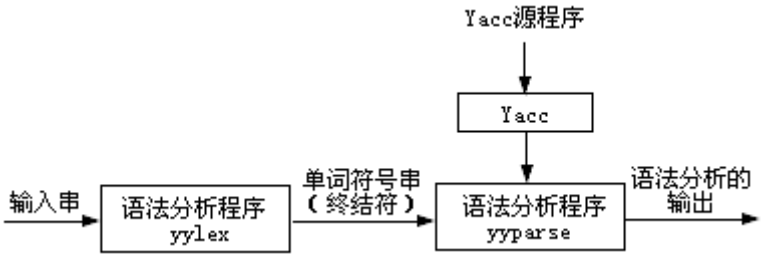
C．1 YACC 概述

形式语言都有严格定义的语法结构，我们对它们进行处理时首先要分析其语法结构。YACC 是一个语法分析程序的自动产生器，严格地说 LEX 也是一个形式语言的语法分析程序的自动产生器。不过 LEX 所能处理的语言仅限于正规语言，而高级语言的词法结构恰好可用正规式表示，因此 LEX 只是一个词法分析程序的产生器。YACC 可以处理能用 LALR（1）文法表示的上下文无关语言。而且我们将会看到 YACC 具有一定的解决语法的二义性的功能。

YACC 的用途很广，但主要用于程序设计语言的编译程序的自动构造上。例如，可移植的 C 语言的编译程序就是用 YACC 来写的。还有许多数据库查询语言是用 YACC 实现的。因此 YACC 又叫做"编译程序的编译程序"（A　Compiler Compiler）。

YACC 的工作示意图如图 C.1。

图 C.1 YACC 示意图



在图 C.1 中，"YACC 源程序"是用户用 YACC 提供的一种类似 BNF 的语言写的要处理的语言的语法描述。YACC 会自动地将这个源程序转换成用 LR 方法进行语法分析的语法分析程序 YYparse，同 LEX 一样，YACC 的宿主语言也是 C，因此 YYparse 是一个 C 语言的程序，用户在主程序中通过调用 YYparse 进行语法分析。

语法分析必须建立在词法分析的基础之上，所以生成的语法分析程序还需要有一个词法分析程序与它配合工作。YYparse 要求这个词法分析程序的名字为 YYlex。用户写 YYlex 时可以借助于 LEX。因为 LEX 产生的词法分析程序的名字正好是 YYlex，所以 LEX 与 YACC 配合使用是很方便的，这将在 C．5 的 C．5．3 中详细介绍，请注意词法分析程序也是可以包含在 YACC 源程序中的。

在 YACC 源程序中除了语法规则外，还要包括当这些语法规则被识别出来时，即用它们进行归约时要完成的语义动作，语义动作是用 C 语言写的程序段。

语法分析的输出可能是一棵语法树，或生成的目标代码，或者就是关于输入串是否符合语法的信息。需要什么样的输出都是由语义动作和程序部分的程序段来实现的。

下面分节介绍 YACC 源程序的写法以及在 UNIX 系统中使用 YACC 的有关命令。

C．2 YACC 源程序的一般格式

一个 YACC 源程序一般包括三部分：说明部分；语法规则部分；程序段部分。这三部分内容依次按下面的格式组织在一起：

说明部分

％％

语法规则部分

％％

程序段部分上述三部分中说明部分和程序段部分不必要时可省去，当没有程序段部分时，第二个％％

也可以省去。但是第一个％％是必须有的。下面详细介绍各部分的组成及写法。

C.3 YACC 源程序说明部分的写法

YACC 源程序的说明部分定义语法规则中要用的终结符号，语义动作中使用的数据类型。变量、语义值的联合类型以及语法规则中运算符的优先级等。这些内容的组织方式如下：

%{

头文件表宏定义

数据类型定义

全局变量定义

%}

语法开始符定义

语义值类型定义

终结符定义

运算符优先级及结合性定义

C.3.1 头文件表

YACC 直接把这部分定义抄到所生成的 C 语言程序 y.tab.c 中去的，所以要按 C 语言的语法规定来写。头文件表是一系列 C 语言的#include 语句，要从每行的第一列开始写，例如

%{#include <stdio.h>

#include <math.h>

#include <ctype.h>

$include ″header.h″

…

%}

C.3.2 宏定义

这部分用 C 语言的#define 语句定义程序中要用的宏。例如

%{

…

#define EOF O

#dffine max(x，y)((x>y)。x：y)

…

%}

C.3.3 数据类型定义

这部分定义语义动作中或程序段部分中要用到的数据类型，例如

%{

…

typedef struct interval {

double lo，hi;

}INTERVAL;

…

%}

C.3.4 全局变量定义

外部变量(external variable)和 YACC 源程序中要用到的全局变量都在这部分定义，例如

```
%{
  …
extern int nfg;
double dreg[26];
INTERVAL Vreg[26];
  …
%}
```

另外非整型函数的类型声明也包含在这部分中，请参看本附录 C.6 例 2。

重申一遍，上述四部分括在%{和%}之间的内容是由 YACC 原样照抄到 y.tab.c 中去，所以必须完全符合 C 语言文法，另外，界符%{和%}最好各自独占一行，即最好不要写成:

```
%{ int x; %}
```

C.3.5 语法开始符定义

上下文无关文法的开始符是一个特殊的非终结符，所有的推导都从这个非终结符开始。在 YACC 中，语法开始符定义语句是：

```
%start 非终结符……
```

如果没有上面的说明，YACC 自动将语法规则部分中第一条语法规则左部的非终结符作为语法开始符。

C.3.6 语义值类型定义

YACC 生成的语法分析程序 yyparse 用的是 LR 分析方法，它在作语法分析时除了有一个状态栈外，还有一个语义值栈，存放它所分析到的非终结
符和终结符的语义值，这些语义值有的是从词法分析程序传回的，有的是在语义动作中赋与的，这些在介绍语义动作时再详细说明。如果没有对语义值的类型做定义，那么 YACC 认为它是整型(int)的，即所有语法符号如果赋与了语义值，则必须是整型的，否则会出类型错，但是用户经常会希望语义值的类型比较复杂，如双精度浮点数，字符串或树结点的指针。这时就可以用语义值类型定义进行说明。因为不同的语法符号的语义值类型可能不同，所以语义值类型说明就是将语义值的类型定义为一个联合(union)，这个联合包括所有可能用到的类型(各自对应一个成员名)，为了使用户不必在存取语义值时每次都指出成员名，在语义值类型定义部分还要求用户说明每一个语法符号(终结符和非终结符)的语义值是哪一个联合成员类型。下面举例说明并请参看本附录 C.6 例 2。

```
%union {
        int ival;
        double dval;
        INTERVAL VVal;
        }
%token <ival> DREG VREG
%token <dval> CONST
%type <dval> dexp
%type <vval> vexp
…
```

在上述定义中，以%union 开始的行定义了语义值的联合类型，共有三个成员类型分别取名为 ival，dval，vval。

以%token 开始的行定义的是终结符(见本附录 C.3.7)所以 DREG，VREG 和 CONST 都是终结符，尖括号中的名字就是这些终结符的语义值的具体
类型。如 DREG 和 VREG 这两个终结符的语义值将是整型(int)的，成员名是 ival。

以%type 开始的行是说明非终结符语义值的类型。如非终结符 dexp 的语义值将是双精度浮点类型，请注意，在 YACC 中非终结符不必特别声明，
但是当说明部分有对语义值类型的定义，而且某非终结符的语义值将被存取，就必须用上面的方法定义它的类型。

C.3.7 终结符定义

在 YACC 源程序语法规则部分出现的所有终结符(文字字符 literal 除外)必须在这部分定义，定义方法如下例：

%token DIGIT LETTER

每个终结符定义行以%token 开头，注意%与 token 之间没有空格，一行中可以定义多个终结符，它们之间用空格分开，终结符名可以由字母，数
字，下划线组成，但必须用字母开头。非终结符名的组成规则与此相同。终结符定义行可多于一个。

YACC 规定每个终结符都有一个唯一的编号(token number)。当我们用上面的方式定义终结符时，终结符的编号由 YACC 内部决定，其编号规则是从 257 开始依次递增，每次加 1。但这个规则不适用于文字字符(literal)的终结符。例如在下面的语法规则中，'+'，';'就是文字字符终结符：

stats:stats㶲;㶲stat;

expr:expr　㶲+㶲expr;

文字字符终结符在规则中出现时用单引号括起来。它们不需要用%token 语句定义，YACC 对它们的编号就采用该字符在其字符集(如 ASCII)中的值。注意上面两条语法规则末尾的分号是 YACC 元语言的标点符号，不是文字字符终结符。

YACC 也允许用户自己定义终结符的编号。如果这样，那么终结符定义的格式就是：

%token 终结符名　　整数

其中"终结符名"就是要定义的终结符，"整数"就是该终结符的编号，每一个这样的行定义一个终结符。特别注意不同终结符的编号不能相同。例如

%token BEGIN100

%token END101

%token IF105

%token THEN200

…

在本附录 C.3.6 中我们说过如果用户定义了语义值的类型，那么那些具有有意义的语义值的终结符其语义值的类型要用 union 中的成员名来说明，除了在 C.3.6 段中介绍的定义方法外，还可以把对终结符的定义和其语义值的类型说明分开，例如

%token DREG VREG CONST

%type <ival> DREG VREG

%type <dval> CONST

C.3.8 运算符优先级及结合性定义

请看下面的关于表达式的文法：

%token NAME

expr：expr '+' expr

　　|expr '-' expr

```
        |expr '*' expr
        |NAME
        ;
```

这个文法有二义性，例如句子

```
a+b-c
```

可以解释成(a+b)-c 也可以解释成 a+(b-c)，虽然这两种解释都合理但造成了二义性，如果将句子

```
a+b*c
```

解释为(a+b)*c 就在语义上错了。

YACC 允许用户规定运算符的优先级和结合性，这样就可以消除上述文法的二义性。例如规定'+"-'具有相同的优先级，而且都是左结合的，这样 a+b-c 就唯一地解释为(a+b)-c。再规定'*'的优先级大于'+"-'，则 a+b*c 就正确地解释为 a+(b*c)了，因此上述文法的正确形式应是：

```
%token NAME
%left '+' '-' 塿
%left '*'
%%
    expr：expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | NAME
        ;
```

在说明部分中以%left 开头的行就是定义算符的结合性的行。%left 表示其后的算符是遵循左结合的;%right 表示右结合性，而%nonassoc 则表示其后的算符没有结合性。优先级是隐含的，在说明部分中，排在前面行的算符较后面行的算符的优先级低；排在同一行的算符优先级相同，因此在上述文法中，'+'和'-'优先级相同，而它们的优先级都小于'*'，三个算符都是左结合的。

在表达式中有时要用到一元运算符，而且它可能与某个二元运算符是同一个符号，例如一元运算符负号'-'就与减号'-'相同，显然一元运算符的优先级应该比相应的二元运算符的优先级高。至少应该与'*'的优先级相同，这可以用 YACC 的%prec 子句来定义，请看下面的文法：

```
%token NAME
%left '-"+'
%left '*"/'
塿%%
expr：expr   '+' expr
    | expr   '-' expr
    | expr   '*' expr
    | expr   '/' expr
    |    '-' expr %prec   '*' 塿
    | NAME
    ;
```

在上述文法中，为使一元'-'的优先级与'*'相同，我们使用了子句

```
%prec 塿   塿
```

它说明它所在的语法规则中最右边的运算符或终结符的优先级与%prec 后面的符号的优先级相同，注意%prec 子句必须出现在某语法规则结尾处分

号之前，%prec 子句并不改变'-'作为二元运算符时的优先级。

上面介绍的 8 项定义，没有必要的部分都可以省去。

C.4 YACC 源程序中语法规则部分的写法

语法规则部分是 YACC 源程序的核心部分，这一部分定义了要处理的语言的语法及要采用的语义动作。下面介绍语法规则的书写格式。语义动作的写法以及 YACC 解决二义性和冲突的具体措施。最后介绍错误处理。

C.4.1 语法规则的书写格式

每条语法规则包括一个左部和一个右部，左右部之间用冒号'：'来分隔，规则结尾处要用分号'；'标记，所以一条语法规则的格式如下：

```
nonterminal ： BODY；
或
nonterminal ： BODY
             ；
```

其中 nonterminal 是一个非终结符，右部的 BODY 是一个由终结符和非终结符组成的串，可以为空，请看几个例子：

```
stat： WHILE bexp DO stat
     ；
stat： IF bexp THEN stat
     ；
stat： /* empty */
     ；
```

上面的第三条语法规则的右部为空，用'/*'和'*/'括起来的部分是注解。

可以把左部非终结符相同的语法规则集中在一起，规则间用短竖线'|'分隔，最后一条规则之后才用分号，例如

```
stat： WHILE bexp DO stat
     | IF bexp THEN stat
     | /* empty */；
```

对语法规则部分的书写有几条建议：

1) 用小写字母串表示非终结符，用大写字母串表示终结符。

2) 将左部相同的产生式集中在一起，像上例一样。

3) 各条规则的右部尽量对齐，例如都从第一个 tab 处开始。

按这样的风格写 Yacc 源程序，清晰可读性强而且易修改和检查错误。

4) 如果产生式(语法规则)需要递归，尽可能使用左递归方式。例如

```
seq ： item
    | seq 塿，塿 item
    ；
```

因为用左递归方式可以使语法分析器尽可能早地进行归约，不致使状态栈溢出。

C.4.2 语义动作

当语法分析程序识别出某个句型时，它即用相应的语法规则进行归约，YACC 在进行归约之前，先完成用户提供的语义动作，这些语义动作可以是返回语法符号的语义值，也可以是求某些语法符号的语义值，或者是其它适当的动作如建立语法树，产生目标代码，打印有关信息等。

终结符的语义值是通过词法分析程序返回的，这个值由全局变量(YACC 自动定义的)yylval 带回，如果

用户在词法分析程序识别出某终结符时，给 yylval 赋与相应的值，这个值就自动地作为该终结符的语义值。当语义值的类型不是 int 时，要注意赋与 yylval 的值的类型需与相应的终结符的语义值类型一致。

语义动作是用 C 语言的语句写成的，跟在相应的语法规则后面，用花括号括起来。例如

```
A：垜(垜B垜)垜
                { hello (l，"abc");}
XXX ： YYY ZZZ
          { printf ("amessage＼n");
            flag=25;
          }
     ;
```

要存取语法符号的语义值，用户要在语义动作中使用以$开头的伪变量，这些伪变量是 YACC 内部提供的，用户不用定义。

伪变量$$代表产生式左部非终结符的语义值，产生式右部各语法符号的语义值按从左到右的次序为$1，$2，…。

例如在下面的产生式中：

```
A ：B C D
  ;
```

A 的语义值为$$，B。C。D 的语义值依次为$1，$2，$3。

为说明伪变量的作用，请看下例：有产生式

```
expr： '(' expr ')'
     ;
```

左边的 expr 的值应该等于右边的 expr 的值，表示这个要求的语义动作为，

```
expr： '(' expr ')'
       {$$=$2;}
      ;
```

如果在产生式后面的语义动作中没有为伪变量$$赋值，YACC 自动把它置为产生式右部第一个语法符号的值(即$1)。

有较复杂的应用中，往往需要在产生式右部的语法符号之间插入语义动作。这意味着使语法分析器在识别出句柄的一部分时就完成这些动作。请看下例：

```
A ： B
       {$$=1;}
     C
       {x=$2; y=$3;}
  ;
```

例中 x 的值最后为 1 而 y 的值置为符号 C 的语义值，注意 B 后面的语义动作$$=1 并非将符号 A 的语义值置为 1，这是因为上面的例子是按下面的方式实
现的。

```
$ACT ：/* empty */
          {$$=1;}
       ;
A      ： B $ACT C
          {x=$2; y=$3;}
```

;

即 YACC 自动设置一个非终结符$ACT 及一个空产生式用以完成上述语义动作。

关于语义动作的实例请读者详细阅读本附录 C.6 中的两个例子。

C.4.3 YACC 解决二义性和冲突的方法

在附录 C.3.8 中已涉及到二义性和冲突的问题，这里再集中介绍一下，这在写 YACC 源程序时会经常碰到。

二义性会带来冲突。在附录 C.3.8 中我们介绍了 YACC 可以用为算符确定优先级和结合规则解决由二义性造成的冲突，但是有一些由二义性造成的冲突不易通过优先级方法解决，如有名的例子：

```
stat ： IF bexp THEN stat
      | IF bexp THEN stat ELSE stat
      ;
```

对于这样的二义性造成的冲突和一些不是由二义性造成的冲突，YACC 提供了下面两条消除二义性的规则：

1) 出现移进/归约冲突时，进行移进;

2) 出现归约/归约冲突时，按照产生式在 YACC 源程序中出现的次序，用先出现的产生式归约。

我们可以看出用这两条规则解决上面的 IF 语句二义性问题是合乎我们需要的。所以用户不必将上述文法改造成无二义性的。当 YACC 用上述两条规则消除了二义性，它将给出相应信息。

下面再稍微严格地介绍一下 YACC 如何利用优先级和结合性来解决冲突的。

YACC 源程序中的产生式也有一个优先级和结合性。这个优先级和结合性就是该产生式右部最后一个终结符或文字字符的优先级和结合性，当使用了%prec 子句时，该产生式的优先级和结合性由%prec 子句决定。当然如果产生式右部最后一个终结符或文字字符没有优先级或结合性，则该产生式也没有优先级或结合性。

根据终结符(或文字字符)和产生式的优先级和结合性，YACC 又有两个解决冲突的规则：

(1) 当出现移进/归约冲突或归约/归约冲突，而当时输入符号和语法规则(产生式)均没有优先级和结合性，就用 1)和 2)来解决这些冲突。

(2) 当出现移进/归约冲突时，如果输入符号和语法规则(产生式)都有优先级和结合性，那么如果输入符号的优先级大于产生式的优先级就移进；如果输入符号的优先级小于产生式的优先级就归约。如果二者优先级相等，则由结合性决定动作，左结合则归约，右结合则移进，无结合性则出错。

用优先级和结合性能解决的冲突，YACC 不报告给用户。

C.4.4 语法分析中的错误处理

当进行语法分析时发现输入串有语法错误，最好能在报告出错信息以后继续进行语法分析，以便发现更多的错误。

YACC 处理错误的方法是:当发现语法错误时，YACC 丢掉那些导致错误的符号适当调整状态栈。然后从出错处的后一个符号处或跳过若干符号直到遇到用户指定的某个符号时开始继续分析。

YACC 内部有一个保留的终结符 error，把它写在某个产生式的右部，则 YACC 就认为这个地方可能发生错误，当语法分析的确在这里发生错误时，YACC 就用上面介绍的方法处理，如果没有用到 error 的产生式，则 YACC 打印出"Syntax error"，就终止语法分析。

下面看两个使用 error 的简单例子

1) 下面的产生式

```
stat：error
    ;
```

使 YACC 在分析 stat 推导出的句型时，遇到语法错误时跳过出错的部分，继续分析(也会打印语法错信息)

2) 下面的产生式

```
stat：error ';'
    ;
```

使 YACC 碰到语法错时，跳过输入串直到碰到下一个分号才继续开始语法分析。

如果语法分析的输入串是从键盘上输入的(即交互式)，那么某一行出错后，希望重新输入这一行，并使 YACC 立即开始继续分析，这只要在语义动作中使用语句 yyerror 即可，如下例

```
input：error '\n'
            { yyerror;
                printf(″Reenter last line：″);}
        input
            {$$=$4;}
        ;
```

关于错误处理请参看本附录 C.6 的例子。

C.5 程序段部分

程序段部分主要包括以下内容:主程序 main();错误信息执行程序 yyerror(s);词法分析程序 yylex();用户在语义动作中用到的子程序，下面分别介绍。

C.5.1 主程序

主程序的主要作用是调用语法分析程序 yyparse()，yyparse()是 YACC 从用户写的 YACC 源程序自动生成的，在调用语法分析程序 yyparse()之前或之后用户往往需要做一些其它处理，这些也在 main()中完成，如果用户只需要在 main()中调用 yyparse()，则也可以使用 Unix 的 YACC 库(-1y)中提供的 main()而不必自己写。库里的 main()如下:

```
main(){
    return(yyparse());
    }
```

C.5.2 错误信息报告程序

YACC 的库也提供了一个错误信息报告程序，其源程序如下:

```
#include <stdio.h>
  yyerror(s) char * s {
        fprintf(stderr，"%s＼n"，s);
        }
```

如果用户觉得这个 yyerror(s)太简单。也可以自己提供一个，如在其中记住输入串的行号并当 yyerror(s)被调用时，可以报告出错行号。

C.5.3 词法分析程序

词法分析程序必须由用户提供，其名字必须是 yylex，词法分析程序向语法分析程序提供当前输入的单词符号。yylex 提供给 yyparse 的不
是终结符本身，而是终结符的编号，即 token number，如果当前的终结符有语义值，yylex 必须把它赋给 yylval。下面是一个词法分析程序例
子的一部分：

```
yylex () {
        extern int yylval;
        int c;
        …
        c=getchar ();
```

```
…
switch (c) {
                    …
case 垜0垜:
case 垜1垜:
…
case 垜9垜:
        yylval=c-垜0垜;
        return (DIGIT);
        …
        }
…
```

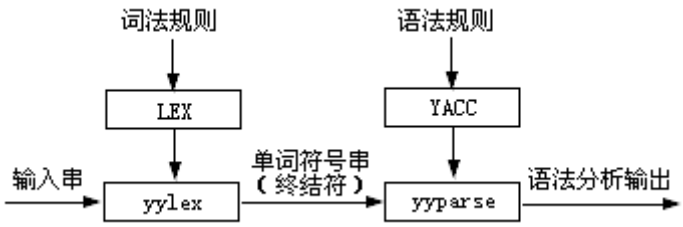
上述词法分析程序碰到数字时将与其相应的数值赋给 yylval，并返回 DIGIT 的终结符编号，注意 DIGIT 代表它的编号(如可以通过宏来定义)。

用户也可以用 LEX 为工具编写词法分析程序，如果这样，在 YACC 源程序的程序段部分就只需要用下面的语句来代替词法分析程序：

```
#include "lex.yy.c"
```

为了清楚 LEX 与 YACC 的关系，我们用图 C.2 表示 LEX 与 YACC 配合使用的情况：

图 C.2 LEX 和 YACC



在 UNIX 系统中，假设 LEX 源程序名叫 plo.l，YACC 源程序名叫 plo.y，则从这些源程序得到可用的词法分析程序和语法分析程序依次使用下述三个命令：

```
lex plo.l
yacc plo.y
cc y.tab.c -ly -ll
```

第一条命令从 LEX 源程序 plo.l 产生词法分析程序，文件名为 lex.yy.c.;第二条命令从 YACC 源程序 plo.y 产生语法分析程序，文件名为 y.tab.c;第三条命令将 y.tab.c 这个 c 语言的程序进行编译得到可运行的目标程序。

第三条命令中的-ll 是调用 LEX 库，-ly 是调用 YACC 库，如果用户在 YACC 源程序的程序段部分自己提供了 main()和 yyerror(s)这两个程序，则不必使用-ly。

另外如果在第二条命令中使用选择项-v，例如

```
yacc -v plo.y
```

则 YACC 除产生 y.tab.c 外，还产生一个名叫 y.output 的文件，其内容是被处理语言的 LR 状态转换表，这个文件对检查语法分析器的工作过程很有用。

C.5.4 其它程序段

语义动作部分可能需要使用一些子程序，这些子程序都必须遵守 C 语言的语法规定，这里不多讲了。

C.6 YACC 源程序例子说明

例 1 用 YACC 描述一个交互式的计算器，该计算器有 26 个寄存器，分别用小写字母 a 到 z 表示，它能接受由运算符+。-。　。/。%(取模)。&(按位求与)。|(按位求或)组成的表达式，能为寄存器赋值，如果计算器接受的是一个赋值语句，就不打印出结果，其它情况下都给出结果，操作数为
整数，若以 0(零)开头，则作为八进制数处理。

例 1 的 YACC 源程序见附录 C.6.1。

读者从例 1 中可以看出用优先关系和二义性文法能使源程序简洁，还可看到错误处理方法，但例 1 不足之处是它的词法分析程序太简单，还有对八进制与十进制数的区分也最好在词法分析中处理。

例 2　这个例子是例 1 的改进，读者能看到语义值联合类型的定义及使用方法和如何模拟语法错误并进行处理，该例也是描述一个交互式的计算器，比例 1 的计算器功能强，它可以处理浮点数和浮点数的区间的运算，它接受浮点常数，以及+、-、*、/、一元-和=(赋值)组成的表达式，它有 26 个浮点变量，用小写字母 a 到 z 表示，浮点数区间用一对浮点数表示：

(x，y)

其中 x 小于或等于 y，该计算器有 26 个浮点数区间变量，用大写字母 A 到 Z 表示。

和例 1 相似，赋值语句不打印出结果，其它表达式均打印出结果，当发生错误时给出相应的信息。

例 2 的 YACC 源程序见附录 C.6.2。

下面简单总结例 2 的一些特点。

1) 语义值联合类型的定义

区间用一个结构表示，其成员给出区间的左右边界点，该结构用 C 语言的 typedef 语句定义，并赋与类型名 INTERVAL。

YACC 的语义值栈经过%union 定义后，可以存放整型。浮点及区间变量的值，还有一些函数(如 hilo，vmul，vdiv)都返回结构类型的值。

2) YACC 的出错处理

源程序中用到了 yyerror 来处理除数区间中含有 0 或除数区间端点次序倒置的错误，当碰到上述错误时，yyerror 使 YACC 调用其错误处理程序，丢掉出错的输入行，继续处理。

3) 使用有冲突的文法

如果读者在机器上运行这个例子，就会发现它包含 18 个移进/归约冲突，26 个归约/归约冲突，请看下面两个输入行：

2.5+(3.5-4.0)

2.5+(3.5，4.0)

在第二行中 2.5 用在区间表达式中，所以应把它当作区间处理，即要把它的类型由标量转换成区间量，但 YACC 只有当读到后面的','时才知道是
否应该进行类型转换，此时改变主意为时已晚，当然也可以在读到 2.5 时再向前看几个符号来决定 2.5 的类型，但这样实现较困难，因为 YACC 本身
不支持，该例是通过增加语法规则和充分利用 YACC 内部的二义性消除机构来解决问题的。在上述文法中，每一个区间二元运算都对应两条规则，其中一条左操作数是区间，另一条左操作数是标量，YACC 可以根据上下文自动地进行类型转换。除了这种情况外，还存在着其它要求决定是否进行类型转换的情形，本例将标量表达式的语法规则放在区间表达式语法规则的前面，使运算量的类型先为标量。直到必要时再转换成区间，这样就导致了那些冲突。有兴趣的读者不妨仔细看一看这个源程序和 YACC 处理它时产生的 y.output 文件，分析一下 YACC 解决冲突的具体方法。

要注意上述解决类型问题的方法带有很强的技巧性，对更复杂的问题就难以施展了。

C.6.1 YACC 的源程序例 1

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

int regs[26];
int base;
%}

%start list
%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS/* SUPPLIES PRECEDENCE*/
                    /* FOR UNARY MINUS*/

%%/* beginning of rules section */
? ? ? ? ?

list:/* empty */
        |list stat '\n'
        |list error '\n'
                { yyerrok ; }
        ;
stat:expr
                { printf("%d\ n ",$1); }
        |LETTER '=' expr
                { regs[$1]= $3; }
        ;
expr:'(' expr ')'
                    { $$= $2; }
        |expr '+' expr
                    { $$= $1 + $3; }
        |expr '-' expr
                    { $$= $1 - $3; }
        |expr '*' expr
                    { $$= $1 * $3; }
        |expr '/' expr
                    { $$= $1 / $3; }
        |expr '%' expr
```

```
                { $$= $1 % $3; }
        |expr '&' expr
                { $$= $1 & $3; }
        |expr '|' expr
                { $$= $1 | $3; }
        |'-' expr %prec UMINUS
                { $$= -$2; }
        |LETTER
                { $$= regs[$1]; }
        |number
        ;
number:DIGIT
                { $$= $1; base= ($1==0)?8：10; }
        |number DIGIT
                  { $$= base*$1+$2; }
          ;

%%/* start of programs */

yylex() {
        /* lexcial analysis routine */
        /* return DIGIT for a digit, yylval=0
             through 9 */
        /* return LETTER for a lower case letter,
             yylval=0 through 25 */
        /* all other characters are
             returned immediately */
             int c;
             while ((c=getchar())==' ')
                {/* skip blanks */}
              /* c is now nonblank */
              if ( islower(c) ) {
                        yylval= c-'a';
                        return(LETTER);
                      }
              if ( isdigit(c) ) {
                        yylval= c-'0';
                        return(DIGIT);
                      }
              return(c);
}
```

C.6.2 YACC 的源程序例 2

```
%{

#include <stdio.h>
#include <ctype.h>

typedef struct interval {
        double lo, hi;
        } INTERVAL;
INTERVAL vmul(), vdiv();

double atof();

double dreg[26];
INTERVAL vreg[26];

%}

%start lines

%union{
        int ival;
        double dval;
        INTERVAL vval;
        }

%token <ival> DREG VREG
   /* indices into dreg, vreg arrays */
%token <dval> CONST
   /* floating point constant */
%type<dval> dexp
   /* expression */
%type<vval> vexp
   /* interval expression */

/* precedence information */
/* about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS
   /* supplies precedence for unary minus */
```

```
%%
lines : /* empty */
        | lines line
        ;

line: dexp '\n'
                { printf("%15.8f\n",$1); }
        | vexp '\n'
                { printf("(%15.8f,%15.8f)\n",
                    $1.lo, $1.hi); }
        | DREG '=' dexp '\n'
                { dreg[$1]= $3; }
        | VREG '=' vexp '\n'
                { vreg[$1]= $3; }
        | error '\n'
                { yyerrok; }
      ;

dexp: CONST
      | DREG
                { $$= dreg[$1]; }
      | dexp '+' dexp
                { $$= $1+$3;}
      | dexp '-' dexp
                { $$= $1-$3;}
      | dexp '*' dexp
                { $$= $1*$3;}
      | dexp '/' dexp
                { $$= $1/$3;}
      | '-' dexp%prec UMINUS
                { $$= -$2; }
      | '(' dexp ')'
                { $$= $2; }
      ;

vexp: dexp
                { $$.hi= $$.lo= $1; }
      | '(' dexp ',' dexp ')'
                {
                  $$.lo= $2;
                  $$.hi= $4;
                  if ($$.lo > $$.hi) {
```

```
                printf("interval out of order\n");
                YYERROR;
                }
            }
        | VREG
            { $$= vreg[$1]; }
        | vexp '+' vexp
            { $$.hi= $1.hi+$3.hi;
              $$.lo= $1.lo+$3.lo; }
        | dexp '+' vexp
            { $$.hi= $1+$3.hi;
              $$.lo= $1+$3.lo; }
        | vexp '-' vexp
            { $$.hi= $1.hi-$3.lo;
              $$.lo= $1.lo-$3.hi; }
        | dexp '-' vexp
            { $$.hi= $1-$3.lo;
              $$.lo= $1-$3.hi; }
        | vexp '*' vexp
            { $$= vmul($1.lo, $1.hi, $3); }
        | dexp '*' vexp
            { $$= vmul($1, $1, $3); }
        | vexp '/' vexp
            { if( dckeck($3) ) YYERROR;
              $$= vdiv($1.lo, $1.hi, $3); }
        | dexp '/' vexp
            { if( dckeck($3) ) YYERROR;
              $$= vdiv($1, $1, $3); }
        | '-' vexp%prec UMINUS
            { $$.hi= -$2.lo;
              $$.lo= -$2.hi; }
        | '(' vexp ')'
            { $$ = $2; }
        ;

    %%
#define BSZ 50
/* buffer size for floating point numbers */

    /* lexcial analysis */

yylex() {
```

```
        register c;

        while((c=getchar())==' ')
        {/* skip over blanks */}
        if (isupper(c)) {
                yylval.ival= c-'A';
                return(VREG);
        }
if (islower(c)) {
                yylval.ival= c-'a';
                return(DREG);
                }

if (isdigit(c) || c=='.') {
    /* gobble up digits, points, exponents */

    char buf[BSZ+1], *cp=buf;
    int dot= 0, exp= 0;

    for (; (cp-buf)<BSZ; ++cp,c=get char() ){

        *cp= c;
          if (isdigit(c)) continue;
      if (c=='.'){
          if (dot++ || exp) return('.');
          /* will cause syntax error */
          continue;
          }
      if (c=='e'){
          if (exp++) return('e');
          /* will cause syntax error */
          continue;
          }

      /* end of number */
      break;
    }

          *cp= '\0';
          if ((cp-buf) >= BSZ)
          printf("constant too long: truncated\n");
          else ungetc(c,stdin);
          /* push back last char read */
```

```
                yylval.dval= atof(buf);
                return(CONST);
                }
        return(c);
        }
#define BSZ 50
/* buffer size for floating point numbers */

        /* lexcial analysis */

yylex() {
            register c;

            while((c=getchar())==' ')
            {/* skip over blanks */}
            if (isupper(c)) {
                        yylval.ival= c-'A';
                        return(VREG);
            }
if (islower(c)) {
                    yylval.ival= c-'a';
                    return(DREG);
                    }

if (isdigit(c) || c=='.') {
        /* gobble up digits, points, exponents */

        char buf[BSZ+1], *cp=buf;
        int dot= 0, exp= 0;

        for (; (cp-buf)<BSZ; ++cp,c=get char() ){

              *cp= c;
                if (isdigit(c)) continue;
           if (c=='.'){
                if (dot++ || exp) return('.');
                /* will cause syntax error */
                continue;
                }
           if (c=='e'){
                if (exp++) return('e');
                /* will cause syntax error */
                continue;
```

```
            }

    /* end of number */
    break;
}

            *cp= '\0';
            if ((cp-buf) >= BSZ)
            printf("constant too long: truncated\n");
            else ungetc(c,stdin);
            /* push back last char read */
            yylval.dval= atof(buf);
            return(CONST);
            }
return(c);
}
```