语法分析器生成工具 YACC

WANG Hanfei

School of Computer
Wuhan University

October 14, 2019

语法分析器生成工具

将含有附注的上下文无关文法转换以某种程序设计语言识别为输出的识别该文法语法分析器源程序.

Examples

1. yacc(Yet Another Compiler-Compiler): 1975 年由贝尔实验室 Mike Lesk & Eric Schmidt 开发，UNIX 标准实用工具 (utility).
2. byacc: Berkeley YACC: Robert Corbett, 1989 年, yacc compatible, in Free BSD distribution.
3. bison: Robert Corbett & Richard Stallmen, 1988 年，yacc compatible, in Linux distribution, 最新版本：2.4. 支持 GLR(Generalized LR) 文法，http://www.gnu.org/software/bison/.
4. ANTLR(前生 PCCTS): 混合词法和语法的生成器，基于 LL(k)(new LL(*)) 文法，支持多种语言输出 (Java, C, C++, C#, Python, Perl and Ruby etc). current version: 4.5. http://www.antlr.org/.
5. CUP: LALR Parser Generator in Java, current version 0.11a, 对应的词法分析器生成工具为：JFLex(http://jflex.de/), http://www2.cs.tum.edu/projects/cup/.
6. 本讲义主要对 bison(version 2.1) 和 byacc 进行讨论.

BISON 的安装

1. Linux: bison 包含在所有的 Linux 发行版中，安装相关选项即可；也可以下载源码 (http://ftp.gnu.org/gnu/bison/), 编译后安装.
2. Windows: GnuWin32 project 把 GNU 软件制作成 Windows 下可直接安装使用的工具，并且不需要任何的 Linux 仿真器的支持 (如：Cygwin). 在 http://gnuwin32.sourceforge.net/packages/bison.htm下载 bison 安装程序，执行安装程序即可自动安装 bison 且设置其运行环境, 用户可直接在命令行输入命令 “bison”.
3. DOS: 我的compiler_cd目录下有 DOS 版本的 bison(v1.25), 由于 DOS 不支持多后缀名 (xxx.tab.c)，其输出的分析器源程名为 “xxx_tab.c” 和 “xxx_tab.h”(其中 “xxx” 为此法分析源文件的前缀). 用户在 DOS 命令行使用时需要设置运行环境path, 如：
   c:>/path=*d:\path_of_bison*;%path%.
4. DOS 版的 byacc(见compiler_cd) 输出的语法分析源文件名为 “y_tab.c” 和 “y_tab.h”

Example: 逆波兰表达式计算器 1/3

```
$ cat rpcalc.y
%{  /* Reverse polish notation calculator.  */
#define YYSTYPE int
#include <ctype.h>
#include <stdio.h> void yyerror (char *);
%}

%token NUM

%% /* Grammar rules and actions follow.  */

input:    /* empty */
   | input line
   ;
line:'\n'
   | exp '\n' { printf ("\%.d\n", $1); }
   ;
exp: NUM { $$ = $1; }
   | exp exp '+'   { $$ = $1 + $2; }
   | exp exp '-'   { $$ = $1 - $2; }
   | exp exp '*'   { $$ = $1 * $2; }
   | exp exp '/'   { $$ = $1 / $2; }
   | exp 'n'  { $$ = -$1;}/* Unary minus    */
   ;
%%
```

Example: 逆波兰表达式计算器 2/3

```
int yylex () /* The lexical analyzer  */ {
  int c;
    /* Skip white space.  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
    /* Process numbers.  */
  if (isdigit (c)) {
    ungetc (c, stdin);
    scanf ("%d", &yylval);
    return NUM;
    }
  if (c == EOF)  return 0; /* Return end-of-input.  */

  return c;  /* Return a single char.  */
}

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

void yyerror (char *s) { /* 必须提供 */
  printf("%s\n", s);
}
```

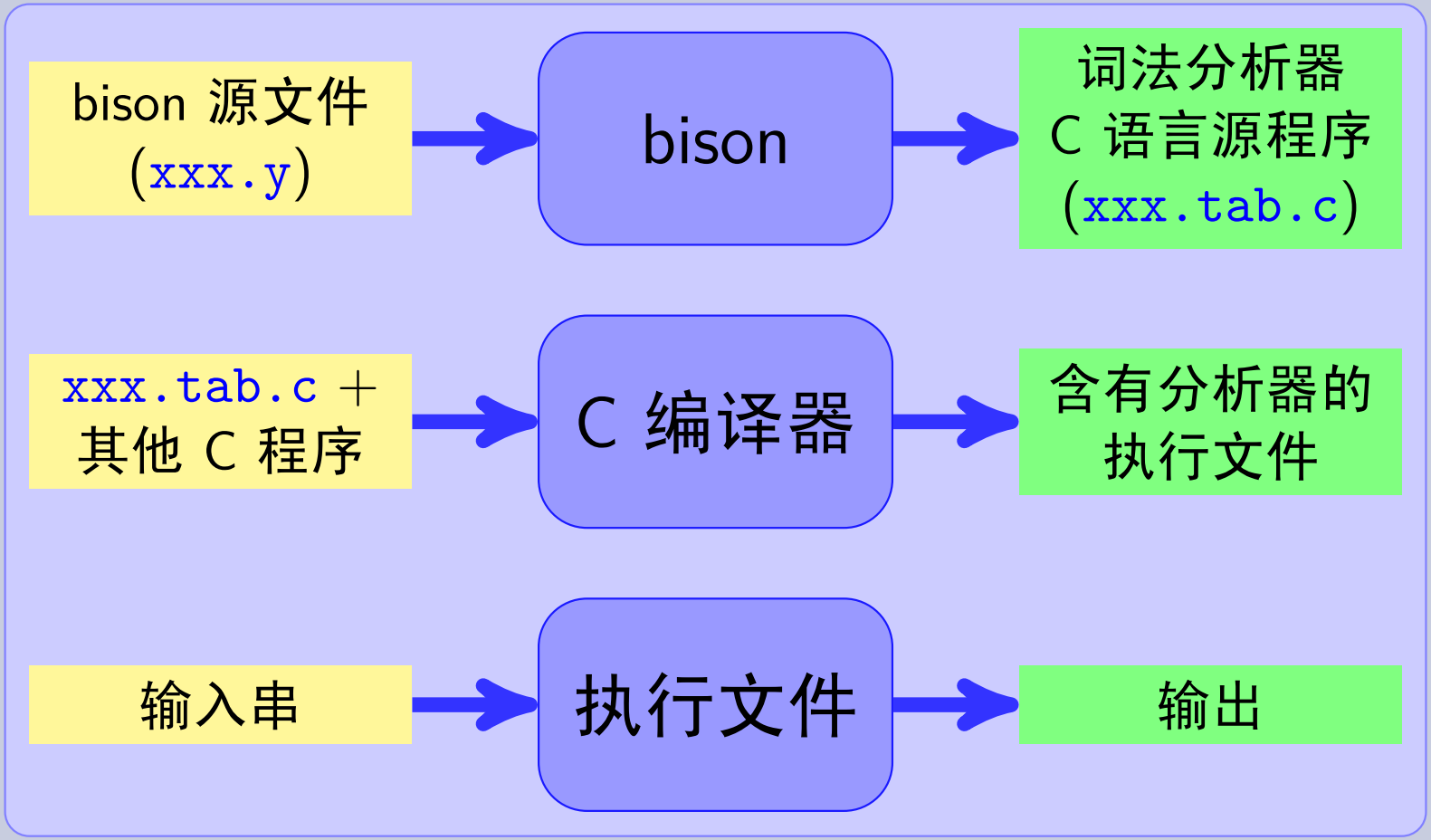
Example: 逆波兰表达式计算器 3/3

```
$ bison rpcalc.y↵
$ ls -l rpcalc.*↵
-rw-r--r--    1 hfwang   teacher   37889 Apr  2 12:32 rpcalc.tab.c
-rw-r--r--    1 hfwang   teacher    1021 Apr  2 12:23 rpcalc.y
$ cat rpcalc.tab.c↵
      ......
#else /* ! YYPARSE_PARAM */ int yyparse() #endif {
  int yystate;
  int yyn;
  int yyresult;
      ......
$ gcc  -o rpcalc rpcalc.tab.c↵
$./rpcalc↵
 2 3 * 5 +↵
        11
 1 + 2↵
syntax error
$
```

BISON 的工作原理

- 对输入文件的中的形式文法构架 LALR 分析表，并生成基于该分析表的语法分析器 C 语言源程序.tab.c，其语法分析函数原型为int yyparse().
- 该语法分析器通过调用用户提供的词法分析函数int yylex()对输入串进行扫描得到的终结符进行语法分析，如果分析成功，则函数返回 0，否则返回非 0.
- 其分析方法采用自底向上的移进/归约分析法，在完成归约时，yyparse()将执行 bison 源文件用户在当前归约产生式后附注的 C 语言代码，bison 称之为语义动作 (Semantic Action).
- 在为该语法分析器提供必要的环境后 (int yylex(), yyerror 和main()), （这些 C 源码也可直接附加在 bison 源文件的语法规则之后），即可用 C 语言编译器编译链接为一个对任意的输入串进行语法分析并完成相关语义动作的执行文件.

由 BISON 源文件到语法分析器的过程



BISON 源文件格式

```
%{
C语言说明部分:
   /* 定义在语义动作中需要的全局变量，语义值的类型，及需要包含的头文件等 */
%}

bison声明部分:
   /* 通过bison提供的指令定义单词（终结符）非终结符，规定非终结符语义值的
      类型，规定运算的优先级别以及文法的形态等 */

%%

语法规则部分:
    /* 形式文法产生式规则及其对应的C语言语义动作附注等. */

%%

附加C语言代码:
   /* 该部分同C语言说明部分一样，bison直接拷贝到输出文件的尾部，用户在此
      可定义分析器需要的接口函数，也可以为空，在其他C文件中定义，
      最后通过连接形成执行文件. */
```

字符，单词和非终结符

- 为了便于 bison 辨别输入文件中语法规则的终结符和非终结符并在生成输出分析器设定其对应的编码, bison 将终结符分为两类: 字符(Symbol) 和单词(Token).
- 字符如果作为终结符出现在语法规则中可以直接用 C 语言字符常量的形式表达，而不需要在bison 声明部分定义. 如：

```
line : exp '\n'
```

- 在需要定义其语义值类型或规定优先级别和结合左右顺序时，字符可以在声明部分用 bison 提供的指令加以申明，如:

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

- 非字符形式单词可以用任意的标识符表示，但必须在 bison 声明部分中用“%token”申明，为了区别非终结符，单词一般用大写字母组成的标识符表示：如：

```
%token NUM
```

- bison 将把语法规则中出现的没有申明的标识符当非终结符处理，如果需要对非终结符的语义值的数据类型进行说明，可在声明部分用“%type”申明该非终结符，非终结符一般小写字母组成的标识符表示.

常见错误

- 语法规则中出现的字符形式的单词未加单引号' '，如产生式之间的分割符号“;”和终结符';';
- 语法规则中出现的单词没有在声明部分用“%token”申明，导致 bison 在编译时将其当非终结符处理，而由于没有对应于该非终结符的产生式而报错.

Example

```
$ cat rpcalc.y
  ......
/* forget declare %token NUM */
  ......
%% /* Grammar rules and actions follow.  */
  ......
exp: NUM { $$ = $1; }
   | exp exp + /* forget quote '+' */  { $$ = $1 + $2; }
  ......
$ bison rpcalc.y
rpcalc.y:19.6-8: symbol NUM is used, but is not defined
  as a token and has no rules
rpcalc.y:16.11-14: syntax error, unexpected identifier
```

终结符编码及其词法分析器接口

- bison 输出的语法分析器是在终结符和非终结符编码基础上进行的，yyparse()函数是通过用户提供的词法分析函数yylex()的返回值来实现向前查看终结符，如：上例中非终结符NUM的编码为258, 为了使得yylex()能协同yyparse()工作，要求yylex()在识别连续的数字组成的字符串后返回258.
- bison 对终结符的编码规则如下：
  - 0或负数：文件结束标记；
  - 字符单词：编码为其对应的 ASCII 码，如：return '+';
  - bison 保留的单词error(出错处理使用) 为 256；
  - 非字符单词从 258 开始顺序编码.
- 为了方便yylex()使用非字符单词的编码，在 bison 输出的语法分析源程序中提供了单词名与其编码的宏定义，这样yylex()函数可以在返回对应单词编码时可直接使用作为宏定义的单词名，如：return NUM; 如果用户提供的词法分析模块是一个独立的文件，bison 可通过命令选项“-d” 输出该宏定义到后缀名为“.tab.h” 的头文件中，这样词法分析模块可通过包含该头文件使用作为宏定义的单词名.

Example

Example

```
$ bison -d rpcalc.y
$ cat rpcalc.tab.h
  ......
/* Tokens.  */
#define NUM 258
  ......

#if ! defined (YYSTYPE) && ! defined (YYSTYPE_IS_DECLARED)
typedef int YYSTYPE;
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
# define YYSTYPE_IS_TRIVIAL 1
#endif

extern YYSTYPE yylval;
  /* 引用定义终结符语义值全局变量 */
```

语义值及其类型

- bison 输出的语法分析器是以 LALR 分析表为驱动，在分析栈中进行移进/归约的自底向上的分析程序，为了提高分析效率，其分析栈采用静态栈结构，分析栈除了状态和文法符号外，还装有当前文法符号对应的语义值.
- 分析器在移进一个终结符进栈的同时也将记录当前终结符语义值的全局变量`yylval`拷贝进栈.
- 分析器匹配某一产生式进行归约时，首先用户执行该产生式规则后附注的 C 语言语义动作，再弹出句柄成分，最后将产生式左边的非终结符和语义动作所计算的语义值拷贝进栈.
- bison 输出的语法分析器语义值的数据类型是宏名`YYSTYPE`，如果用户没有指定语义值的数据类型，bison 将其定义为`int`, 用户可以根据需要通过下述两方式定义自己所需的语义值数据类型.

单类型语义值

所有的语法符号的语义值类型相同

- 在 C 语言说明部分通过YYSTYPE的宏定义规定语义值的数据类型，如：下述宏定义定于语义值的数据类型为double

```
#define YYSTYPE double
```

- 由于在移进/归约时语义值是拷贝进栈，所以语义值的类型不能是数组，如：为了记录输入的逆波兰表达式对应的中缀表达式，定于语义值的类型为字符数组:

```
typedef char INFIX [100];
#define YYSTYPE INFIX
......
exp : exp exp '+' {
    sprintf($$, "(%s+%s)", $1, $2); }
```

输出的分析器 C 语言源程序编译时将会报错:

```
rpcalc.tab.c:807: error: incompatible types in
assignment }
```

多类型语义值

不同的语法符号语义值的类型不同

- 为了在同一个语义栈中对不同的语法符号使用不同类型，bison 通过%union指令在声明部分将YYSTYPE定义为 C 语言的union结构，并通过 “%type <union 分量名 > 非终结符名” 和 “%token <union 分量名 > 终结符名” 来规定在语义动作中以何种方式访问语义值对应的类型. 如在mcalc.y中：

```
%union {
  double val; /* For returning numbers. */
  symrec *tptr; /* For symbol-table pointers */
} /* without ';' for ending %union */
%token <val> NUM
%token <tptr> VAR FNCT
%type <val> exp
```

经过 bison 编译后，在mcalc.tab.c中YYSTYPE将定义如下：

```
typedef union {
  double val; /* For returning numbers. */
  symrec *tptr; /* For symbol-table pointers */
} YYSTYPE ;
```

语义值的生命周期

- 一个语义值的生命周期是指该语义值在语义栈中存在的时间. 如rpcalc.y中的NUM的对应的yylval生命周期是指在yylex()函数返回NUM前通过语句 "scanf ("%d", &yylval);" 创建，语法分析器在移进NUM进栈的同时，将yylval拷贝进栈，当分析器使用产生式 "exp: NUM" 进行归约前首先执行该产生式对应的语义动作 "{ $$ = $1; }" 计算出归约后非终结符exp的语义值, 再将NUM及其语义值弹出栈顶，此时NUM的语义值生命周期结束，分析器将exp和其语义值$$压栈，从而开始exp生命，该生命将延续到对其进行归约后结束.
- 在使用指针类型语义值时特别要注意:
  - 所指的地址一定是非局部变量, 如所指地址是yylex()的局部变量，但是在其生命周期内函数yylex() 已经返回，从而产生 dangling reference.
  - 所指的变量在其生命周期内被修改，如语义值指指向字符缓冲区的某个位置，但是在其生命周期内该缓冲区被重新装载，因此在引用该语义值时发生错误.
  - 安全的做法是: 生命周期开始时通过动态申请内存将所指的值保存，语义值指向所申请的内存，在其生命周期结束时释放该内存.

Example: 逆波兰表达式到中缀表达式的转换

```
$ cat rp2inf.y
......
#define YYSTYPE char * /* in C declaration part */
  ......
exp : NUM { $$ = $1; }  /* in grammar rule part */
    | exp exp '+' {
        $$ = malloc (strlen($1) + strlen($2) +5);
        sprintf($$, "(%s+%s)", $1, $2);
        free($1), free($2);
      }
......
int yylex (void) /* in additional C source part */
{
  int c, tmp;
   /* Skip white space. */
  while ((c = getchar ()) == ' ' || c == '\t')
  ;
  if (isdigit (c)) {
    ungetc (c, stdin);
    scanf ("%d", &tmp);
    yylval = malloc (sizeof(char) * 20);
    sprintf( yylval, "%d", tmp);
    return NUM;
  }
  ......
```

文法规则及语义动作

格式

- 同一个非终结符的n个产生式按下述格式在bison 语法规则部分输入：

```
非终结符     :     产生式 1    { 语义动作 1 }
             |     产生式 2    { 语义动作 2 }
            ......
             |     产生式 n    { 语义动作 n }
             ;
```

- 每个产生式 i是以白字符 (空格、横向跳格和换行) 和 C 语言注释分割符的终结符和非终结符中间列表，如果产生式 i没有任何语法符号，表示是空产生式.

Example

```
input: /* empty */
    | input line
    ;
exp: NUM { $$ = $1; }
    | exp exp '+' { $$ = $1 + $2; }
    | exp exp '-' { $$ = $1 - $2; }
    ;
```

附注在产生式尾部的语义动作

- 附注在产生式尾部的语义动作在发现该产生式句柄已经在栈顶形成，需要对该产生式进行归约时进行，这时产生式右边的文法符号的语义值已在栈中，语义动作可以是完成归约后的非终结符的语义值的计算 (综合属性)，也可以是其他有副作用的操作，如: 打印输出或保存符号表等. 在附注 C 语言动作中可以使用 "$m" 访问在语义栈中产生式右边文法符号的语义值，用 "$$" 记录归约后非终结符对应的语义值. 其对应关系如下：

  非终结符 : X_1 X_2 ⋯ X_n

  ↑ ↑ ↑ ↑

  $$ $1 $2 $n

  如: `exp : exp exp '+' { $$ = $1 + $2; }`

- bison 在处理语义动作时将 "$m" 转换为 "yyvsp[-n+m]"(n为产生式右边字符串的长度，yyvsp为当前栈顶指针)，如上述语义动作翻译为：

  ```
  (yyval) = (yyvsp[-2]) + (yyvsp[-1]);
  ```

  输出的语法分析器在用该产生式归约前执行该段 C 语言代码，归约后将yyval拷贝到语义栈顶.

- 在执行附注 C 语言动作前，分析器首先执行 "yyval = yyvsp[-n+1];", 即 "$$ = $1;"，称之为缺省语义动作, 这样如果产生式没有附注或者附注中没有任何对$$的赋值，归约后的非终结符的语义值是其第一个儿子的语义值.

继承属性与产生式中部的语义动作 (1/4)

- 附注在产生式末端的语义动作仅能计算综合属性，而对继承属性无能为力. 为此，bison 允许在产生式中间附注语义动作以求解 L-属性 (同翻译规程一样).
- 中部的语义动作相当于在产生式中间引入一个新的非终结符，该非终结符有一个空产生式：

$$A \to X_1\ X_2\{Action1\}\ X_3\ \cdots\ X_m\ \{Action2\}$$
$$\Leftrightarrow \begin{cases} A \to X_1\ X_2 Action1\ X_3\ \cdots X_m\ \{Action2\} \\ Action1 \to \varepsilon\ \{Action1\} \end{cases}$$

即当分析器进入到项目集 "$\{A \to X_1 X_2 \bullet X_3 \cdots X_m, Action1 \to \bullet\}$" 所在的状态，用虚拟产生式 $Action1 \to \varepsilon$归约，即不弹出任何符号，压虚拟非终结符 $Action1$入栈，同时完成 $Action1$ 语义动作，压对应的语义值入栈. 因此 $Action1$ 可以完成 X_3 继承属性的计算，或通过该语义动作的副作用，设定后续语义值计算的环境.

- 语义值的对应坐标如下所示：

$$\underbrace{A}_{\$\$} \to \underbrace{X_1}_{\$1}\ \underbrace{X_2}_{\$2}\ \underbrace{\{\$\$ = f(\$1, \$2)\}}_{\$3}\ \underbrace{X_3}_{\$4}\ \cdots\ \underbrace{X_k}_{\$(k+1)}\ \{Action2\}$$

继承属性与产生式中部的语义动作 (2/4)

- 在同一产生式的后续语义动作中可以通过 “ $m”(语义值为单一类型) 或 “ $<comp_name>m”(语义值类型用 “%union” 定义, 其中comp_name是union结构的分量名)，m 为该中部语义动作所在的位置.
- 如果作为后续语法符号的继承属性使用，必须用 “$0” 或 “$ 负数” 引用栈中句柄前的语义值的方式实现. 如：设有产生式 “$X_3 \to BC\{Action3\}$”, 分析器进入规约项目 $X_3 \to BC\bullet$所在的状态时，栈的格局如下：

bottom	$\cdots$	X_1	X_2	$Action1$	B	C	top
	$\cdots$	$-2	$-1	$0	$1	$2	

这样在 $Action3$中可以通过 “$0” 访问 $Action1$定义的语义值.

- 该访问模式要求出现在产生式右边的 X_3之前全都有中部语义动作，有一个例外就导致 $Action3$无法正确工作，如：设有无中部语义动作的产生式 $Y \to Y_1\ X_3$，分析器在吃进前缀 “$\cdots Y_1 BC$后到达 $X_3 \to BC\bullet$所在的状态，栈的格局如下:

bottom	$\cdots$	Y_1	B	C	top
	$\cdots$	$0	$1	$2	

这样在 $Action3$中访问的 “$0” 不是希望的继承属性. 增加中部语义动作 $Y \to Y_1\ \{Action4\}\ X_3$ 即可保证此时 “$0” 能正确取到 X_3的继承属性.

继承属性与产生式中部的语义动作 (3/4)

增加中部语义动作相当于在原文法中增加了许多空产生式，可能导致在用 bison 编译时产生归约/归约冲突. 如:

```
compound : { prepare_for_local_variables (); }
             '{' declarations statements '}'
         | '{' statements '}'
         ;
```

该文法相当于进行如下的变换:

$$\left\{\begin{array}{lcl} A & \to & aBCb \\ & | & aCb \end{array}\right. \Longleftrightarrow \left\{\begin{array}{lcl} A & \to & A'aBCb \\ & | & aCb \\ A' & \to & \varepsilon \end{array}\right.$$

这样其 LR(0) 项目集中有项目 $\{A \to \bullet A'aBCb, A \to \bullet aCb, A' \to \bullet\}$, 即面对 a 可以移进到下一个状态也可用 $A' \to \varepsilon$ 归约，产生移进/归约冲突.

继承属性与产生式中部的语义动作 (4/4)

如果 B 和 C 的 First 集合交为空，则将 A' 放置在 a 之后就可避免上述的移进/归约冲突：

$$\left\{\begin{array}{lcl} A & \rightarrow & aBCb \\ & | & aCb \end{array}\right. \Longleftrightarrow \left\{\begin{array}{lcl} A & \rightarrow & aA'BCb \\ & | & aCb \\ A' & \rightarrow & \varepsilon \end{array}\right.$$

这样将中部语义动作移到`{'之后就可避免上述移进/归约冲突：

```
compound : '{' { prepare_for_local_variables (); }
             declarations statements '}'
         | '{' statements '}'
         ;
```

如果 B 和 C 的 First 集合交不为空，这样做还有 S-R 冲突. 必须对文法进行变换，提取 B 和 C 的公因子，在公因子后加上中部动作从而避免冲突.

注意：并不是所有的 L-属性都能在不破坏 LALR 文法的前提下通过中部语义动作求解.

终结符的语义值的计算

- 分析器在移进一个终结符到分析栈顶的同时将全局变量YYSTYPE yylval的值拷贝进语义栈；
- 用户提供的词法分析函数yylex()在返回一个单词前负责完成该单词的语义值的计算，并将计算结果保存到yylval
- 如果某一单词的语义值不被 bison 中的语义动作引用，则yylex()返回该单词时，不必计算yylval；
- 如果yylex()在另一个独立于 bison 源文件的 C 源程序中定义，该 C 源程序必须：
  - #include "xxx.tab.h"

  该文件包含了所有的单词宏定义和全局变量yylval的引用声明.

Example: 逆波兰表达式计算器的词法分析函数

```
int yylex () /* The lexical analyzer  */
{
  int c;
    /* Skip white space.  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
    /* Process numbers.  */
  if (isdigit (c)) {
    ungetc (c, stdin);
    scanf ("%d", & yylval);
    return NUM;
    }
  if (c == EOF)  return 0; /* Return end-of-input. */

  return c;  /* Return a single char
                without yylval computation. */
}
```

移进/归约和归约/归约冲突

bison 输出的语法分析器是以 LALR(1) 分析表为推动的分析器，bison 对输入的文法构造 LALR(1) 分析表时，如果发现了移进/归约和归约/归约冲突，将采取以下措施：

- 报警输出冲突的统计信息，在输出的 LALR(1) 分析表文件.output(“bison -t” 选项打开时) 指出冲突所在；
- 如果是移进/归约冲突，缺省选择移进解销冲突；
- 如果是归约/归约冲突，缺省选择输入文件中排列在前的产生式解销冲突；
- 输出按照上述解消冲突后的 LALR 分析表工作的语法分析 C 语言源程序.

如果 bison 有冲突报警，表示输入的文法不是 LALR(1) 文法，因此解消冲突后的分析程序不能保证能识别该文法的所有语句，因此有冲突发生时，一定要文法进行缜密的分析，保证解消冲突后的分析程序能正确地工作. bison 也提供一些解决冲突的方法.

运算优先级和结合次序制定

文法中运算优先级和结合次序产生的二义性，在 LALR(1) 分析表中表现移进/归约冲突，选择任何一种解消冲突的方法将不会破坏分析器的识别能力，但是该选择将表现为运算的优先级和结合次序. bison 可以通过声明部分的指令 “%left”(左结合), “%right”(右结合) 和 “%nonassoc”(不能结合) 规定结合次序和优先级别来解消二义性. 如：

```
%left '+' '-'
%left '*' '/'
%left UMINUS     %%
exp : exp '+' exp | exp '-' exp
    |  exp '*' exp | exp '/' exp
    |  '-' exp %prec UMINUS
```

bison 将按如下优先级和结合次序解消冲突：

- '+'和二元减'-', '*'和'/'，及虚拟终结符'UMINUS'均为左结合.
- 出现在同一个 “%left” 后的单词其优先级别相同；如，'+'和二元减'-'.
- 出现在不同 “%left” 的终结符，其运算优先级别按照 “%left” 在源文件中出现的先后次序由低到高排列, 即：UMINUS的优先级最高，乘法和除法次之，加法和减法最低.
- 由于一元减和二元减的运算符相同，在语法中出现一元减的产生式通过%prec UNIMUS设定一元减和UMINUS具有相同的优先级.

Example 1/3: if-then-else 结构

```
$ cat if_else.y
/* if-then-else ambiguity test */
%left 'e'
%left 't' /* 'then' precedence greater than 'else', means in the
             state {s -> 'i' s 't' ., s -> 'i' s 't' s. 'e' s},
             and lookahead 'e', the parser will perform reduce,
             so the parser can never recognize 'else' part */

%%
line: '\n'
   | s '\n'  printf ("parser success!\n");
   ;

s: 'a'
   | 'i' s 't' s 'e' s
   | 'i' s 't' s
   ;
%%
......
$ bison  if_else.y
$ gcc -o if_else if_else.tab.c
$ ./if_else
i a t a e a
  syntax error
```

Example 2/3: 归约/归约冲突 (1/3)

```
$ cat sequenc.y
%token WORD

%% /* Grammar rules and actions follow. */

sequence: /* empty */
        { printf ("empty sequence\n"); }
  |  maybeword
  |  sequence WORD
        { printf ("added word %s\n", $2); }
  ;
maybeword: /* empty */
        { printf ("empty maybeword\n"); }
  |  WORD { printf ("single word %s\n", $1); }
  ;
%%
$ bison -v  sequence.y
sequence.y: conflicts: 1 shift/reduce, 2 reduce/reduce
sequence.y:11.5-37: warning: rule never reduced because of
conflicts: maybeword: /* empty */
```

Example 2/3: 归约/归约冲突 (2/3)

```
$ cat sequence.output
Grammar
    0 $accept: sequence $end
    1 sequence: /* empty */
    2         | maybeword
    3         | sequence WORD
    4 maybeword: /* empty */
    5          | WORD
.......
state 0
    0 $accept: . sequence $end

    WORD  shift, and go to state 1
     /* 面对WORD可移进到状态1，也可用产生式1或4归约，
        缺省选择移进 */
    $end      reduce using rule 1 (sequence)
    $end      [reduce using rule 4 (maybeword)]
     /* 面对$可用产生式1或4归约，缺省选择1归约 */
    WORD      [reduce using rule 1 (sequence)]
    WORD      [reduce using rule 4 (maybeword)]
    $default  reduce using rule 1 (sequence)
    sequence   go to state 2
    maybeword  go to state 3
```

Example 2/3: 归约/归约冲突 (3/3)

```
$ cat sequenc1.y
%token WORD

%% /* Grammar rules and actions follow. */

sequence: /* empty */
        { printf ("empty sequence\n"); }
  |  sequence WORD
        { printf ("added word %s\n", $2); }
  ;
  /* delete redundant grammar rules which cause
     reduce/reduce conflicts */
%%
$ bison -v  sequence1.y
```

Example 3/3: 归约/归约冲突 (1/4)

```
$ cat define.y
/* 该文法为LR(1),但是不是LALR(1),因为识别成分return_spec的
   一个项目{<type: ID ., ','>, <name : ID ., ':'>}与识别
   param_spec的项目{<type: ID ., 'ID'>, <name : ID ., ','>}
   合并为LALR(1)项目时发生归约/归约冲突. */
%token ID
%%
def: param_spec return_spec ','
   ;
param_spec:  type
   | name_list ':' type
   ;
return_spec: type
   | name ':' type
   ;
type: ID
   ;
name: ID
   ;
name_list: name
   | name ',' name_list
   ;
%%
......
$ bison -v define.y
define.y: conflicts: 1 reduce/reduce
```

Example 3/3: 归约/归约冲突 (2/4)

```
$ cat define.output
Grammar
    0 $accept: def $end
    1 def: param_spec return_spec ','
    2 param_spec: type
    3           | name_list ':' type
    4 return_spec: type
    5            | name ':' type
    6 type: ID
    7 name: ID
    8 name_list: name
    9          | name ',' name_list
.......
state 1
    6 type: ID .
    7 name: ID .

    ','       reduce using rule 6 (type)
    ','       [reduce using rule 7 (name)]
      /* 面对','可用产生式6或7归约, 缺省选择6归约 */
    ':'       reduce using rule 7 (name)
    $default  reduce using rule 6 (type)
```

Example 3/3: 归约/归约冲突 (3/4)

```
$ cat define1.y
/* 修改文法，让上述两个项目不能合并. 为此引入新终结符BOGUS, 新产生式
   return_spec: ID BOGUS, 这样识别return_spec的项目集为:
   {<type: ID., 'ID'>, <name: ID., ','>, <return_spec: ID. BOGUS, ','>},
   不能再与前者合并，从而解销冲突. 只要词法分析器永远都不返回BOGUS, 其识
   别能力与未修改的一样.  */
%token ID  BOGUS
%%
def: param_spec return_spec ','
  ;
param_spec:  type
  | name_list ':' type
  ;
return_spec: type
  | name ':' type
    /* this rule never used */
  | ID BOGUS
  ;
type: ID
  ;
name: ID
  ;
name_list: name
  | name ',' name_list
  ;
%%
$ bison define1.y
```

Example 3/3: 归约/归约冲突 (4/4)

```
$ cat define1.output
Grammar
    0 $accept: def $end
    1 def: param_spec return_spec ','
    2 param_spec: type
    3           | name_list ':' type
    4 return_spec: type
    5            | name ':' type
    6            | ID BOGUS
    7 type: ID
    8 name: ID
    9 name_list: name
   10          | name ',' name_list
.......
state 1
    7 type: ID .
    8 name: ID .

    ID        reduce using rule 7 (type)
    $default  reduce using rule 8 (name)
```

Example: 中缀表达式的翻译 calc.h

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <alloc.h>
/* 符号表的数据类型，每个符号元素表将保存函数名和变量名和
   对应的类型及其数值或函数指针，符号表以链表方式实现  */
struct symrec {
  char *name;  /* 符号名 */
  int type;    /* 符号类型 */
  union {
    double var;           /* 变量的数值 */
    double (*fnctptr)();  /* 函数指针 */
    } value;
    struct symrec *next;    /* 指向下一个符号单元 */
  };
typedef struct symrec SYMREC;

/* 符号表的应用定义*/
extern SYMREC *sym_table;

SYMREC *putsym (); /* 插入函数 */
SYMREC *getsym (); /* 查表函数 */
```

Example: 中缀表达式的翻译 main.c (1/2)

```
#include "calc.h"
#include "y_tab.h" /* bison为mcalc.tab.h */
extern int yyparse();
struct init {
  char *fname;
  double (*fnct)();
};
struct init arith_fncts[] = {
  "sin", sin, "cos", cos, "atan", atan, "ln", log,
  "exp", exp, "sqrt", sqrt, 0, 0
};
  /* 定义符号表表头指针，其初值为空指针 */
SYMREC *sym_table = (SYMREC *)0;
SYMREC * putsym (char *sym_name, int sym_type)
{  SYMREC *ptr;
   ptr = (SYMREC *) malloc (sizeof (SYMREC));
   ptr->name = (char *) malloc (strlen (sym_name) + 1);
   strcpy (ptr->name,sym_name);
   ptr->type = sym_type;
   ptr->value.var = 0; /* set value to 0 even if fctn.  */
   ptr->next = (SYMREC *)sym_table;
   sym_table = ptr;
   return ptr;
}
```

Example: 中缀表达式的翻译 main.c (2/2)

```
SYMREC * getsym (char *sym_name)
{
  SYMREC *ptr;
  for (ptr = sym_table; ptr != (SYMREC *) 0;
       ptr = (SYMREC *)ptr->next )
    if (strcmp (ptr->name,sym_name) == 0)
      return ptr;
  return 0;
}

void init_table ()  /* 将数学函数预置于符号表中 */
{
  int i;
  SYMREC *ptr;
  for (i = 0; arith_fncts[i].fname != 0; i++) {
    ptr = putsym (arith_fncts[i].fname, FNCT);
    ptr->value.fctptr = arith_fncts[i].fnct;
  }
}
main()
{
  init_table();
  yyparse();
}
```

Example: 中缀表达式的翻译 mcalc.y (1/2)

```
%{
#include "calc.h"  /* 包含系统定义头文件*/
#define YYDEBUG 1 /* 激活调试代码 */
#define YYDEBUG_LEXER_TEXT lexeme /* 调试时输出的向前查看的词形 */
char lexeme[80];
%}

%union {
  double     val;  /* 表达式的语义值 */
  SYMREC  *tptr;   /* 变量和函数的语义值是符号表元素指针  */
  }

%token <val>  NUM        /* 语义值的类型是双精度浮点数  */
%token <tptr> VAR FNCT   /* 语义值的类型是符号表元素指针 */
%type  <val>  exp
     /* 定义结合次序和优先级别 */
%right '='
%left '-' '+'
%left '*' '/'
%left NEG     /* 虚拟词汇，和一元减同级 */
%right '^'    /* 指数函数  */
```

Example: 中缀表达式的翻译 mcalc.y (2/2)

```
%% /* 语法规则部分 */
input:    /* 空串 */
        | input line
        ;
line:     '\n'
        | exp '\'    { printf ("\t%.10g\n", $1); }
        | error '\n' { yyerrok; }
        ;
exp:      NUM                { $$ = $1; }
        | VAR                { $$ = $1->value.var; }
        | VAR '=' exp        { $$ = $3; $1->value.var = $3; }
        | FNCT '(' exp ')'   { $$ = (*($1->value.fnctptr))($3); }
        | exp '+' exp        { $$ = $1 + $3; }
        | exp '-' exp        { $$ = $1 - $3; }
        | exp '*' exp        { $$ = $1 * $3; }
        | exp '/' exp        { $$ = $1 / $3;  }
        | '-' exp  %prec NEG { $$ = -$2;  }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2; }
        ;
%%
void yyerror(char const *s)
{ fprintf(stderr, "%s", s); }   /* 必须提供 */
```

Example: 中缀表达式的翻译 scanner.c (1/2)

```
#include  "calc.h"
#include  "y_tab.h" /* DOS的byacc输出的头文件，bison为mcalc.tab.h */
extern YYSTYPE yylval;
extern char lexeme[];

int yylex ()
{
  int c;
     /* 跳过白字符 */
  while ((c = getchar ()) == ' ' || c == '\t');

  if (c == EOF) return 0;

    /* 读取数值常量 */
  if (c == '.' || isdigit (c)) {
      ungetc (c, stdin);
      scanf ("%lf", &yylval.val);
      sprintf(lexeme, "%lf", yylval.val); /* 记录词形 */
      return NUM;
      }
  if (isalpha (c)) {/* 读取标识符  */
      SYMREC *s;
      static char *symbuf = 0;
      static int length = 0;
      int i;
       /* 动态申请一个长度为41的字符数组保存词形 */
```

Example: 中缀表达式的翻译 scanner.c (2/2)

```
    if (length == 0)
        length = 40, symbuf = (char *)malloc length + 1);
    i = 0;
    do { /* 如果词形的长度超过40，重新分配双倍的内存 */
      if (i == length) {
        length *= 2;
        symbuf = (char *)realloc(symbuf, length + 1);
        }
      symbuf[i++] = c; /* 保存当前字符到数组symbuf中 */
      c = getchar (); /* 读取下一个字符 */
    } while (c != EOF && isalnum (c));
    ungetc (c, stdin); /* 回退最后一个字符到输入 */
    symbuf[i] = '\0';
      /* 查看符号表，如果不在其中表示新定义变量 */
    s = getsym (symbuf);
    if (s == 0) s = putsym (symbuf, VAR);
    yylval.tptr = s;
    sprintf(lexeme, s->name,79); /* 返回词形 */
    lexeme[79] = 0;
    return s->type;
    }
   /* 对其他的字符返回其ASCII码  */
lexeme[0] = c; lexeme[1] = 0;
return c;
}
```

Example: TC 的 Makefile mcalc.mak

```
CFLAGS  = -v -O
INCLUDE = -I\tc\include;.
LIB = -L\tc\lib
CC  = \tc\tcc
MODEL  = -mc

all: mcalc.exe

.c.obj:
     $(CC) $(INCLUDE) -c $(CFLAGS) $(MODEL) $*.c
#-------------------------------------------------------
y_tab.c y_tab.h:  mcalc.y
    byacc -tdv mcalc.y
y_tab.obj: y_tab.c y_tab.h
scanner.obj: y_tab.h
#-------------------------------------------------------
mcalc.exe:   y_tab.obj scanner.obj main.obj
    $(CC) -emcalc.exe $(LIB) $(MODEL) y_tab.obj scanner.obj main.obj
# ------------------------------------------------------
clean:
    del mcalc.exe
    del y_tab.c
    del y_tab.h
    del y_tab.obj
    del main.obj
    del scanner.obj
```

Example: DOS 下编译及调试

```
E:\mcalc>set path=e:\tc;%path%
E:\mcalc>make
MAKE  Version 2.0   Copyright (c) 1987, 1988 Borland ...

Available memory 546569 bytes
        byacc -tdv mcalc.y

        \tc\tcc -I\tc\include;. -c -v -O  -mc y_tab.c
Turbo C  Version 2.01  Copyright (c) 1987, 1988 Borland ...
y_tab.c:
......
E:\mcalc>mcalc
x= 4*atan(1)
        3.141592654
sin(x) + cos(x)
        -1
sin(x)^2 + cos(x)^2
        1
Ctrl+Z
E:\mcalc>
```

出错处理

- 如果在 bison 输入文件中没有任何提供任何出错处理，输出分析器在工作时，如果发现语法错误 (即在当前状态下面对当前的终结符不能进行任何移进或归约操作), yyparse()在传参 “syntax error” 调用用户提供的void yyerror(char const *)函数后，返回1退出yyparse(). 通常yyerror可定义如下：

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

- bison 通过预留的终结符 error 提供从出错状态恢复分析的机制, 如：

```
stmnts: /* empty string */
      | stmnts '\n'
      | stmnts exp '\n'
      | stmnts error '\n'
```

- 在产生式右边有保留终结符 error 的规则称为错误恢复规则，输出分析器在一个含有错误恢复规则的状态出错时，首先从分析栈中弹出语法符号直到能将 error 移进，在栈顶形成错误恢复规则中 error 前的句型为止，如上例中在分析exp 时出错，将弹出所有的已分析的表达式成分，将 error 压栈到项目{stmnts: stnmts error . '\n'}; 然后，跳过所有的输入直到遇到错误恢复规则规中error的后随符号为止. 如上例中将跳过所有的输入直到'\n' 才恢复分析.

yyerrok 和 yyclearin

- 在错误恢复规则中可以像其他规则一样有语义动作，但是不能引用终结符error的语义值.
- 在某一错误恢复规则起作用时，为了防止分析器频繁报错，输出分析器规定只有在连续移进三个单词后才恢复报错，在错误恢复规则语义动作中，可以使用 bison 提供的 C 语言宏 “yyerrok;” 解消上述限定，即刻恢复报错机制.
- 在激活错误恢复规则后，当前向前查看的单词将会继续作为向前查看符号进行分析，如果用户希望跳过该单词，可在语义动作中用 bison 提供的宏 “yyclearin;” 跳过当前单词. 但是由于该操作是在归约出错产生式时才起作用，因此如果错误恢复规则中error 之后还有终结符，只有移进该终结符才能有语义动作，因此 “yyclearin;” 将无任何效果. “yyclearin;” 有效当且仅当它是在紧随error之后的动作中.

Example: 出错处理 (1/4)

```
E:\mcalc>type mcalc.y
input:   /* 空串 */
       | input line
       ;
line:
         '\n'
       | exp '\n'   { printf ("\t%.10g\n", $1); }
       | error { yyclearin; /* valid */ }
       ;
exp:     NUM                { $$ = $1; }
       | VAR                { $$ = $1->value.var; }
       | VAR '=' exp        { $$ = $3; $1->value.var = $3; }
       | FNCT '(' exp ')'   { $$ = (*($1->value.fnctptr))($3); }
       | exp '+' exp        { $$ = $1 + $3; }
       | exp '-' exp        { $$ = $1 - $3; }
       | exp '*' exp        { $$ = $1 * $3; }
       | exp '/' exp        { $$ = $1 / $3;  }
       | '-' exp  %prec NEG { $$ = -$2;  }
       | exp '^' exp        { $$ = pow ($1, $3); }
       | '(' exp ')'        { $$ = $2; }
       ;
%%
```

Example: 出错处理 (2/4)

```
E:\mcalc>set YYDEBUG=6↵ /* 激活跟踪调试 */
E:\mcalc>mcalc
$ mcalc↵
a + ↵
Ctrl + Z
input
|                 .... 向前查看VAR   `a'
|       VAR <-- `a'
|       |                 .... 向前查看'+'   `+'
|       exp
|       |       '+' <-- `+'
|       |       |                 .... 向前查看'\n'   `\n'
syntax error
|       +-------+  discarding state
|       |
+-------+  discarding state
|
|       error
|       line
input
|               .... 向前查看end-of-file   `' /* '\n' is discarded */
```

Example: 出错处理 (3/4)

```
E:\mcalc>type mcalc.y
input:   /* 空串 */
       | input line
       ;
line:
         '\n'
       | exp '\n'   { printf ("\t%.10g\n", $1); }
       | error '\n' { yyclearin; /* invalid */ }
       ;
exp:     NUM                { $$ = $1; }
       | VAR                { $$ = $1->value.var; }
       | VAR '=' exp        { $$ = $3; $1->value.var = $3; }
       | FNCT '(' exp ')'   { $$ = (*($1->value.fnctptr))($3); }
       | exp '+' exp        { $$ = $1 + $3; }
       | exp '-' exp        { $$ = $1 - $3; }
       | exp '*' exp        { $$ = $1 * $3; }
       | exp '/' exp        { $$ = $1 / $3;  }
       | '-' exp  %prec NEG { $$ = -$2;  }
       | exp '^' exp        { $$ = pow ($1, $3); }
       | '(' exp ')'        { $$ = $2; }
       ;
%%
```

Example: 出错处理 (4/4)

```
E:\mcalc>mcalc
a +
Ctrl + Z
input
|                  .... 向前查看VAR   `a'
|       VAR <-- `a'
|       |                .... 向前查看'+'   `+'
|       exp
|       |       '+' <-- `+'
|       |       |                .... 向前查看'\n'   `\n'
syntax error
|       +-------+  discarding state
|       |
+-------+  discarding state
|
|       error
|       |       '\n' <-- `\n' /* no discarded */
|       line
input
|                .... 向前查看end-of-file   `'
```

BISON 与 FLEX 的接口

- bison 通过`int yylex()`的返回值获得单词的编码；通过全局变量`YYSTYPE yylval`获得当前单词的语义值.
- flex 输出的单词扫描程序正是`int yylex()`, 为了让 bison 输出的语法分析源程序能与 flex 输出的词法分析源程序协同工作，必须保证单词编码的一致，以及语义值数据类型的一致, 这要求在 flex 源程序中:
  1. 定义部分包含 bison 输出的头文件`.tab.h`
  2. 根据语法规则中的单词设计匹配单词的正规表达式，及需要过滤的白字符 (white space) 和注释, 如: '`\n`' 如果是语义规则所需的单词，就不能冒然过滤掉.
  3. 在识别单词的 C 语言动作中，如果该单词有语义值，应先计算其语义值并赋值到全局变量`yylval`，如果是字符单词，`return yytext[0];`，否则`return 单词名;`. `.tab.h`的单词名的宏定义保证词法与语法分析单词编码的一致性;
  4. 为了保证词法分析能处理所有的字符，对语法分析不支持的字符的的动作是报错并过滤.
- 注意：bison 使用的单词名不能与 flex 中自带的宏名有冲突. 如: `BEGIN`不能作为单词名, 否则`return BEGIN;`将调用 flex 预置的宏`BEGIN`, 而不能返回期望的单词编码, 可修改单词名为`SBEGIN`来避免冲突.

Example: xml.l

```
$ cat xml.l
%{
......
#include "xml.tab.h"  /* token define header file */
%}
......
%%
......
(" "|\t|\n)+   ; /* skip white space */

([^<\n \t][^<\n]*)|("<?"[^<?]+"?>")|("<![CDATA["([^\]]+"]"|[^\n])*"]]>")  {
  char * att = (char *) malloc (yyleng + 1 );
  strcpy(att, yytext);
  yylval.node = make_node(PCDATA, NULL, att, yylineno);
    /* because semantic value is pointer, the value which
       pointed must be global or allocate by malloc(),
       otherwise dangling reference */
    /* set token semantic value */
  return TEXT; /* return the token name */
  }
......
%%
```

BISON 和 BYACC 主要命令选项

- `-v` 输出分析表文件*name*`.output`(bison), `y.output`(byacc);
- `-d` byacc 输出单词宏定义及`YYSTYPE`类型定义和`yylval`引用说明文件`y.tab.c`;
- `-t` 在输出文件中设置宏`YYDEBUG`为 1，从而打开分析器的调试代码, 使得最后生成的分析器执行文件能对分析过程进行跟踪;

BISON 的设置 1/3

bison 通过输出分析器 C 语言源程序的宏YYDEBUG和全局变量yydebug来激活跟踪分析过程的功能，即在stderr上显示移进/归约的过程. 通过查看 bison 输出的.tab.c文件，可看到跟踪的实现方式.

mcalc.tab.c的代码片段

```
#if YYDEBUG
.......
do {                                    \
  if (yydebug)                              \
    {                                       \
      YYFPRINTF (stderr, "%s ", Title);             \
      yysymprint (stderr,                       \
                  Type, Value); \
      YYFPRINTF (stderr, "\n");                 \
    }                                       \
} while (0)
.....
# endif
```

BISON 的设置 2/3

设置方式

1. 用 bison 编译时加-t选项，或在 bison 源文件的 C 语言说明部分加 “#define YYDEBUG”;
2. 调用yyparse()前，置yydebug非零，如下例所示 linux 版mcalc的main.c

linux的main.c

```
main()
{
  extern int yydebug;
  yydebug = 1;

  init_table();
  yyparse();
}
```

BISON 的设置 3/3

```
$ ./mcalc
1 + 2
Starting parse
Entering state 0
Reducing stack by rule 1 (line 28):
-> $$ = nterm input ()
Stack now 0
Entering state 1
Reading a token: Next token is token NUM ()
Shifting token NUM ()
Entering state 4
Reducing stack by rule 6 (line 38):
   $1 = token NUM ()
-> $$ = nterm exp ()
Stack now 0 1
Entering state 11
Reading a token: Next token is token '+' ()
Shifting token '+' ()
Entering state 18
Reading a token: Next token is token NUM ()
Shifting token NUM ()
.....
```

BYACC 设置 1/3

byacc 通过输出分析器 C 语言源程序的宏YYDEBUG和环境变量YYDEBUG来激活跟踪分析过程的功能. 输出的分析器通过环境变量YYDEBUG的值显示不同的跟踪方式: YYDEBUG=0时不显示分析过程; YYDEBUG<5时文字方式显示; 4<YYDEBUG<10时字符图形方式显示.

mcalc.y用byacc编译输出的y_tab.c片段

```
int yyparse()
{
    register int yym, yyn, yystate;
#if YYDEBUG
    register char *yys;
    extern char *getenv();

    if (yys = getenv("YYDEBUG"))
    {
        yyn = *yys;
        if (yyn >= '0' && yyn <= '9')
            yydebug = yyn - '0';
    }
#endif
```

BYACC 的设置 2/3

```
E:\mcalc>set YYDEBUG=1 /* linux: export YYDEBUG=1 */
E:\mcalc>mcalc
yydebug: state 0, reducing by rule 1 (input :)
yydebug: after reduction, shifting from state 0 to state 1
1+2
yydebug: state 1, reading 257 (NUM)
yydebug: state 1, shifting to state 3
yydebug: state 3, reducing by rule 6 (exp : NUM)
yydebug: after reduction, shifting from state 1 to state 9
yydebug: state 9, reading 43 ('+')
yydebug: state 9, shifting to state 17
yydebug: state 17, reading 257 (NUM)
yydebug: state 17, shifting to state 3
yydebug: state 3, reducing by rule 6 (exp : NUM)
yydebug: after reduction, shifting from state 17 to state 26
yydebug: state 26, reading 10 ('\n')
yydebug: state 26, reducing by rule 10 (exp : exp '+' exp)
yydebug: after reduction, shifting from state 1 to state 9
yydebug: state 9, shifting to state 21
yydebug: state 21, reducing by rule 4 (line : exp '\n')
3
yydebug: after reduction, shifting from state 1 to state 10
yydebug: state 10, reducing by rule 2 (input : input line)
yydebug: after reduction, shifting from state 0 to state 1
```

BYACC 的设置 3/3

```
E:\mcalc>set YYDEBUG=6↵ /*  linux:  export YYDEBUG=6 */
E:\mcalc>mcalc↵
input
1+2↵
|                 .... 向前查看NUM   `1.000000'
|       NUM <-- `1.000000'
|       exp
|       |                   .... 向前查看'+'   `+'
|       |       '+' <-- `+'
|       |       |                   .... 向前查看NUM   `2.000000'
|       |       |       NUM <-- `2.000000'
|       |       |       exp
|       |       |       |                 .... 向前查看'\n'   `\n'
|       exp
|       |       '\n' <-- `\n'
|       line
|       3
input
```

注: 跟踪程序通过宏YYDEBUG_LEXER_TEXT所定义的字符指针类型的变量输出向前查看单词的词形，如mcalc.y的C语言说明部分:

```
#define YYDEBUG_LEXER_TEXT lexeme
char lexeme[80];
```

总结 — 用 YACC 写语法分析器的步骤

1. 文法和词法设计：尽量使用左递归文法，使得分析栈保持较小的规模. 用 YACC 测试文法是否有 S-R 和 R-R 冲突，`-v`选项输出分析表`.output`文件查看冲突所在.
2. 测试没有任何语义操作的分析器: `-t`选项跟踪分析过程，使得移进/归约的操作与所设计的文法一致.
3. 语法制导定义：合成分析、制定属性、编写 SDD(Syntax-Directed Definition) 及验证 SDD 的完备性，一致性 (见 semantics.pdf).
4. 将 SDD 转换为翻译规程：综合属性对应产生式尾部的语义动作，L 属性对应于产生式中部的语义动作. 加入空语句作为中部的语义动作，查看是否导致 R-R 冲突. 如果有，则修改属性或文法.
5. 为每个属性设计数据结构，`struct`统一同一文法符号的不同属性，`union`统一不同文法符号的不同属性.
6. 加入语义动作，生成有语义动作的分析器.
7. 测试：各种可能的语句.

本章小节