第6章 语义分析和中间代码生成(参考书第6、8章)

- 分析和综合
  - 词法、语法与语义分析是对源程序的分析
  - 中间代码生成、代码优化、目标代码的生成则属于对源程序的综合
- 语法分析结果存储——符号表、语法树
- 语义分析是如何实现含义的理解?

从一个问题说起...

编写一个能计算简单算术表示式值的程序.

例如: 3+4*5

方案：递归动作分析法

解决步骤：

(1)先写出简单算术表达式的文法规则：

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

(2)写出递归动作分析算法——先消除左递归

规则： factor → (exp) | n

```
void  factor( )
{
  switch( token )
  {
     case '(':
            match('(') ;
            exp( );
            match(')') ;
            break;
     case n :
            match (n) ;
            break;
     default:
           error( ) ;
  } // switch
} // factor
```

规则：　　exp → term { addop term }

```
void  exp( )
  {
    term( );
    while ((token == '+') || (token == '-' ))
    {
       match (token) ;
       term( );
    } // while
  } // exp
```

- 为了算法更简洁，用addop→+|-规则做了替换。

规则： term → factor { mulop factor }

```
void term( )
{
  factor( ) ;
  while ((token == '*' )||(token == '/' ))
  {
     match (token) ;
     factor( );
  } //while
} // term
```

- （3）如何在递归下降分析程序中增加计算功能
  - 遇到运算对象则返回
  - 遇到运算符号就将对应运算对象进行计算
  - 遇到括号就返回括号中表达式的计算结果
  - 因此，需要为每个递归函数加上返回当前计算结果

```
int exp( )
{
   int temp;
   temp = term( );
   while ((token ==’+’)||(token ==’-‘))
    {
        switch (token )
         {
            case ‘+’ : match (‘+’) ;
                       temp = temp + term( );
                        break;
            case ‘-‘:  match (‘-’) ;
                       temp = temp – term( );
                       break;
         }  //switch
   } while
  return temp ;
} //exp
```

```
int  term( )
{
   int temp;
   temp = factor( );
   while ((token ==’*’)||(token ==’/‘))
    {
       switch (token )
        {
            case ‘* ’: match (‘*’) ;
                       temp = temp * factor( );
                       break;
            case ‘/‘:  match (‘/’) ;
                       temp = temp / factor( );
                       break;
        }  //switch
   } while
  return temp ;
} //term
```

```
Int   factor( )
{
  switch( token )
  {
     case '(':
              match('(') ;
              temp=exp( );
              match(')') ;
              break;
     case n :
              match (n) ;
              temp=n;
              break;
     default:
             error( ) ;
  } // switch
 return temp;
} // factor
```

main函数的安排

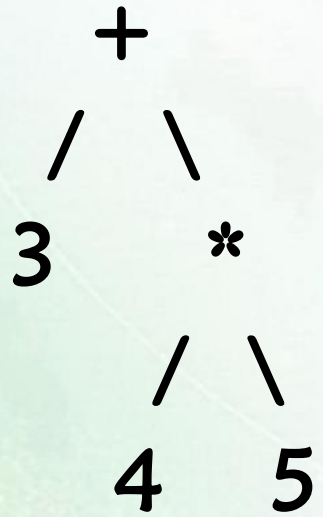
```
main()
{
   getToken( );
   cout<<Exp( );
}
```

如：3+4*5

另一个问题...

编写一个能生成简单算术表示式对应语法树的程序.

例如: 3+4*5



解决方法：

- （1）写出文法规则
- （2）写出递归下降分析程序
- （3）在递归下降分析程序中增加语法树生成功能
  - 遇到运算对象生成叶子结点并返回
  - 遇到运算符号就将对应运算对象进行新树根的构造
  - 遇到括号就返回括号中表达式对应语法树返回
  - 因此，需要为每个递归函数加上返回当前所生成的语法树树根指针

```
BTreeNode  * exp( )
{   BTreeNode * temp, *newtemp;
    temp = term( ) ;
    while ( token == '+' || token == '-' )
    {  switch ( token)
        {
         case '+' :  match ('+') ;
                     newtemp = new BTreeNode ;
                     newtemp ->data= '+' ;
                     newtemp->lchild = temp ;
                     newtemp->rchild= term( ) ;
                     temp = newtemp ;
          case  '-' :  match ('-') ;
                     newtemp = new BTreeNode ;
                     newtemp ->data= '-' ;
                     newtemp->lchild = temp ;
                     newtemp->rchild= term( ) ;
                     temp = newtemp ;
        }
    }
  return temp ;
}// exp
```

进一步合并代码...

```
BTreeNode  * exp( )
{
   BTreeNode * temp, *newtemp;
    temp = term( ) ;
    while ( token == '+' || token == '-' )
    {
         newtemp = new BTreeNode ;
         newtemp ->data= token ;
         match (token) ;
         newtemp->lchild = temp ;
         newtemp->rchild= term( ) ;
         temp = newtemp ;
    }
 return temp ;
}// exp
```

```
BTreeNode  * term( )
{
   BTreeNode * temp, *newtemp;
    temp = factor( ) ;
    while ( token == '*' || token == '/'  )
    {
        newtemp = new BTreeNode ;
         newtemp ->data= token ;
         match (token) ;
         newtemp->lchild = temp ;
         newtemp->rchild= factor( ) ;
         temp = newtemp ;
    }
 return temp ;
}// term
```

```
BTreeNode   * factor( )
{
 BTreeNode *temp;
  switch( token )
  {
     case '(':
            match('(') ;
            temp= exp( );
            match(')') ;
            break;
     case n :
            match (n) ;
           temp = new BTreeNode ;
           temp ->data= n ;
           temp->lchild = NULL ;
           temp->rchild=NULL;
           break;
     default:
           error( ) ;
  }  // switch
 return temp;
} // factor
```

main函数的安排

```
main( )
{
   BTreeNode *root;

   getToken( );
   root=Exp( );
}
```

如：3+4*5

一个新问题…

编写一个能生成简单算术表示式对应汇编代码。

例如: 3+4*5

```
Ldc 3
Ldc 4
Ldc 5
MPI
Adl
```

与栈打交道的汇编语言

所有操作都依赖与栈来完成

汇编指令的介绍

Ldc n　把常数n压入栈
Mpi　取出栈顶与次栈顶元素做乘法运算，结果入栈
Adi　取出栈顶与次栈顶元素做加法运算，结果入栈
Sbi　取出栈顶与次栈顶元素做减法运算，结果入栈
Dvi　取出栈顶与次栈顶元素做除法运算，结果入栈

解决方法：

- （1）写出文法规则
- （2）写出递归下降分析程序
- （3）在递归下降分析程序中增加汇编代码生成功能
  - 遇到运算对象则产生Ldc指令
  - 遇到运算符号就生成相应运算的指令
  - 遇到括号则不做指令生成处理

```
void exp( )
{
   term( );
   while ((token ==’+’)||(token ==’-‘))
    {
        switch (token )
        {
           case ‘+’ :  match (token) ;
                       term( );
                       Gen(Adi)
                       break;
           case ‘-‘:   match (token) ;
                       term( );
                       Gen(Sbi)
                       break;
        }  //switch
   } while
} //exp
```

```
void term( )
{
   factor( );
   while ((token ==’*’)||(token ==’/‘))
    {
       switch (token )
        {
            case ‘* ’:  match (token) ;
                        factor( );
                        Gen(Mpi);
                        break;
             case ‘/‘:  match (token) ;
                        factor( );
                        Gen(Dvi);
                        break;
        }  //switch
   } while
} //term
```

```
void  factor( )
{
  switch( token )
  {
     case '(':
            match('(') ;
            exp( );
            match(')') ;
            break;
     case n :
            match (n) ;
            Gen( Ldc, n )
            break;
     default:
           error( ) ;
  }  // switch
} // factor
```

main函数的安排

```
main()
{
   getToken( );
   Exp( );
}
```

如：3+4*5

结 论

处理过程：

（1）先进行形式化的描述——文法规则

（2）递归下降分析程序——其它分析方法是否可以？

（3）根据问题的含义或要求在递归下降分析程序中增加相应的处理功能——语义动作

改变处理方法...

G[E]:

$E \rightarrow E+n \mid n$

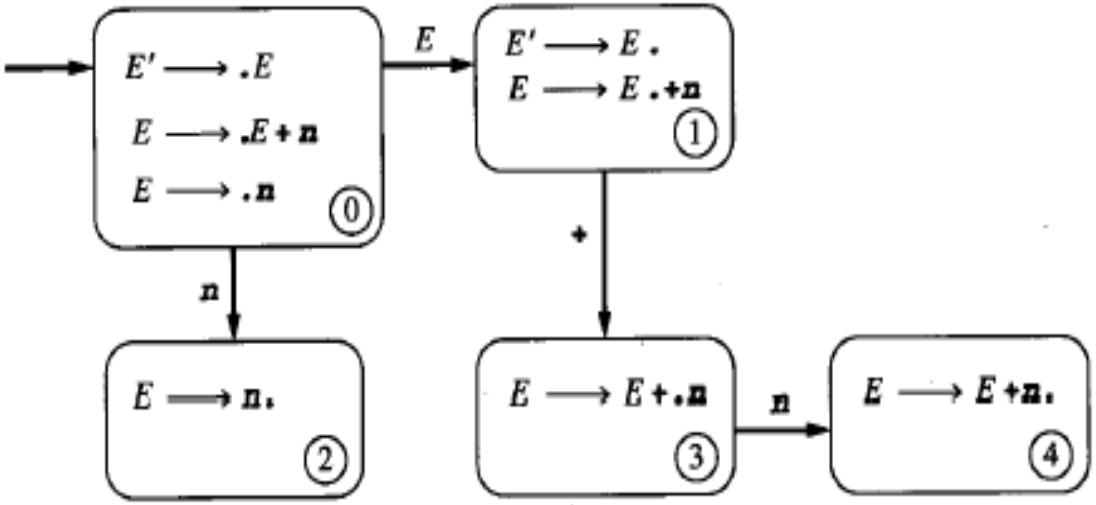
试用自底向上分析方法计算 3+4+5的值。

(1) 文法的扩充

$E' \rightarrow E$

$E \rightarrow E+n \mid n$

(2)构造LR(0)DFA



(3)构造SLR(1)分析表

状　态	输　入			Goto
	n	+	$	*E*
0	s2			1
1		s3	接受	
2		r(*E*→n)	r(*E*→n)	
3	s4			
4		r(*E*→*E*+n)	r(*E*→*E*+n)	

状　态	输　入			Goto
	n	+	$	E
0	s2			1
1		s3	接受	
2		r(E→n) 添加语义动作	r(E→n) 添加语义动作	
3	s4			
4		r(E→E+n) 添加语义动作	r(E→E+n) 添加语义动作	

步骤	分析栈	输入	动作
1	$ 0	n + n + n $	移进2
2	$ 0 n 2	+ n + n $	用E→n归约
3	$ 0 E 1	+ n + n $	移进3
4	$ 0 E 1 + 3	n + n $	移进4
5	$ 0 E 1 + 3 n 4	+ n $	用E→E + n 归约
6	$ 0 E 1	+ n $	移进3
7	$ 0 E 1 + 3	n $	移进4
8	$ 0 E 1 + 3 n 4	$	用E→E + n 归约
9	$ 0 E 1	$	接受

每条规则的语义动作分别为：

规则：$E \rightarrow n$　　{ E.val=n }

规则：$E \rightarrow E + n$　　{ E.val=E.val+n }

改变功能...

G[E]:

$E \rightarrow E+n \mid n$

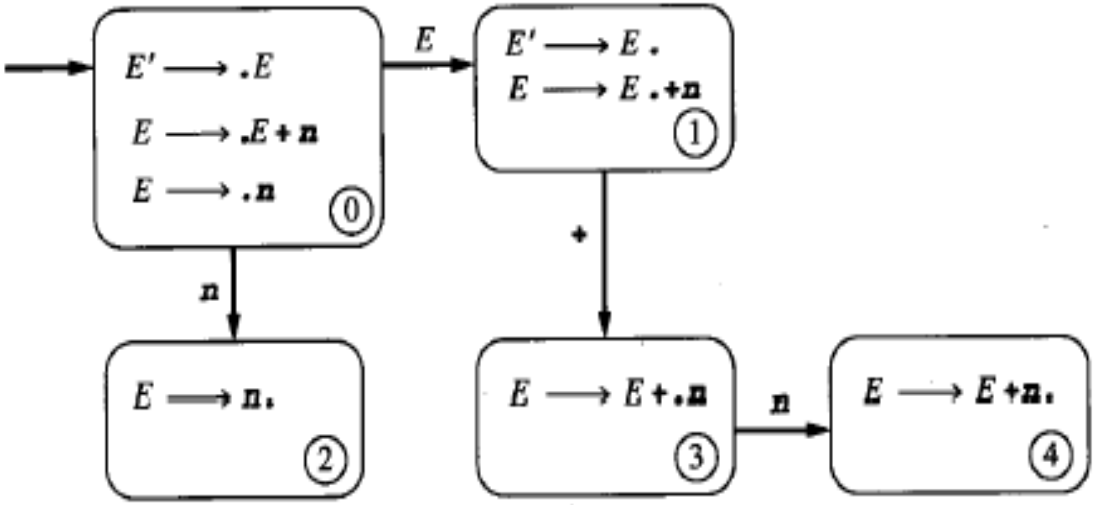
试用自底向上分析方法生成3+4+5 语法树。

(1)文法的扩充

$E' \rightarrow E$

$E \rightarrow E+n \mid n$

(2)构造LR(0)DFA



(3)构造SLR(1)分析表

状　态	输　入			Goto
	n	+	$	*E*
0	s2			1
1		s3	接受	
2		r(*E*→n)	r(*E*→n)	
3	s4			
4		r(*E*→*E*+n)	r(*E*→*E*+n)	

状　态	输　入			Goto
	n	+	$	E
0	s2			1
1		s3	接受	
2		r(E→n) 添加语义动作	r(E→n) 添加语义动作	
3	s4			
4		r(E→E+n) 添加语义动作	r(E→E+n) 添加语义动作	

步骤	分析栈	输入	动作
1	$ 0	n + n + n $	移进2
2	$ 0 n 2	+ n + n $	用E→n归约
3	$ 0 E 1	+ n + n $	移进3
4	$ 0 E 1 + 3	n + n $	移进4
5	$ 0 E 1 + 3 n 4	+ n $	用E→E + n 归约
6	$ 0 E 1	+ n $	移进3
7	$ 0 E 1 + 3	n $	移进4
8	$ 0 E 1 + 3 n 4	$	用E→E + n 归约
9	$ 0 E 1	$	接受

每条规则的语义动作分别为：

规则：E → n　　{ p=new BTreeNode;p->data=n;
p->lchild=p->rchild=0;
E.root=p;
}

规则：$E_2 \rightarrow E_1 + n$ { p=new BTreeNode; p->data=n;
p->lchild=p->rchild=0;
nr=new BTreeNode; nr->data='+';
nr->lchild= E_1 .root;
nr->rchild=p;
E_2.root=nr;
}

再一个例子…

G[E]:

E → E + n | n

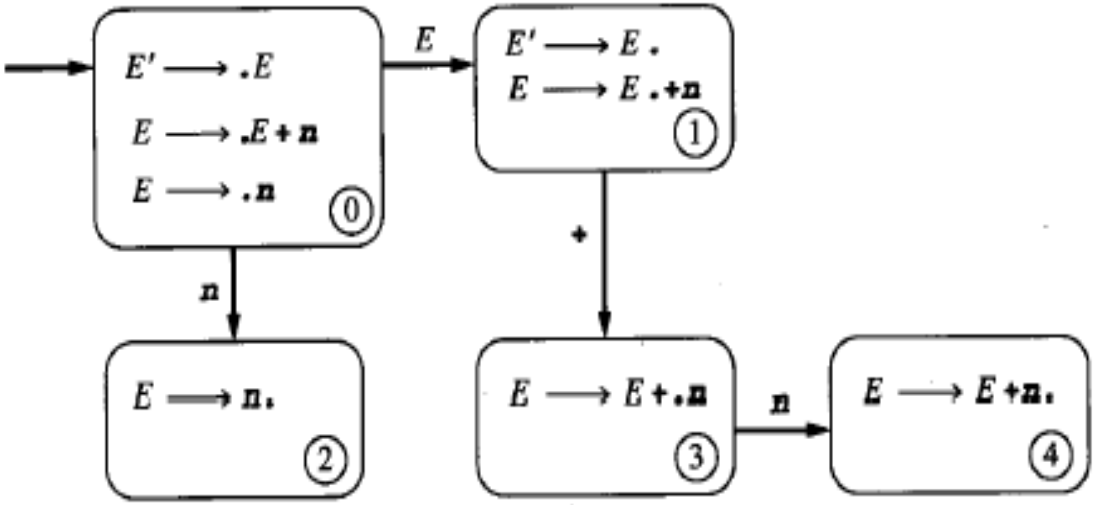
试用自底向上分析方法生成n+n+n的汇编代码。

(1) 文法的扩充

E’→ E

E → E + n | n

(2)构造LR(0)DFA



(3)构造SLR(1)分析表

状　态	输　入			Goto
	n	+	$	*E*
0	s2			1
1		s3	接受	
2		r(*E*→n)	r(*E*→n)	
3	s4			
4		r(*E*→*E*+n)	r(*E*→*E*+n)	

状　态	输　入			Goto
	n	+	$	E
0	s2			1
1		s3	接受	
2		r(E→n) 添加语义动作	r(E→n) 添加语义动作	
3	s4			
4		r(E→E+n) 添加语义动作	r(E→E+n) 添加语义动作	

步骤	分析栈	输入	动作
1	$ 0	n + n + n $	移进2
2	$ 0 n 2	+ n + n $	用E→n归约
3	$ 0 E 1	+ n + n $	移进3
4	$ 0 E 1 + 3	n + n $	移进4
5	$ 0 E 1 + 3 n 4	+ n $	用E→E + n 归约
6	$ 0 E 1	+ n $	移进3
7	$ 0 E 1 + 3	n $	移进4
8	$ 0 E 1 + 3 n 4	$	用E→E + n 归约
9	$ 0 E 1	$	接受

G[E]:

$E \rightarrow E + n \mid n$

自底向上分析中的语义动作——规约时执行的动作

——即需要为每一条规则定义语义动作

每条规则的语义动作分别为：

规则：$E \rightarrow n$ { Gen(Ldc n) }

规则：$E \rightarrow E + n$ { Gen(Ldc n), Gen(Adi) }

语义动作定义方法的简化

为了简单起见，我们以后一般采用如下的规约过程，就可以完成对各规则语义动作的定义。

如：

G[E]:

$E \rightarrow E + n \mid n$

定义生成汇编代码的语义动作的分析过程为：

3+4+5

新 结 论

处理过程：

（1）先进行形式化的描述——文法规则

（2）语法分析法

（3）根据问题的含义或要求在语法分析法的基础上增加相应的处理功能——语义动作

结 论

- 语法分析基本思想：在语法分析过程中，根据语言的语义定义随时分析并翻译已识别的那部分语法成分的全部含义。
  - 翻译需要是通过调用为该语法成分事先编好的语义动作实现的。
- 常用语法分析翻译方法：
  - 方法一：在确定的递归下降语法分析程序中，利用隐含堆栈存储各递归下降函数内的局部变量所表示的语义信息。

- 方法二：在自底向上语法分析程序中使用和语法分析栈同步操作的语义栈进行语法分析翻译。

- 方法三：在LL(1)语法分析程序中，利用翻译文法实施语法分析翻译。
  - 翻译文法是在描述语言的文法(即源文法或输入文法)中加入语义动作符号而形成的。

- 方法四：利用属性文法进行语法分析翻译。
  - 属性文法也是一种翻译文法
  - 其符号(文法符号和动作符号)都扩展为带有语义属性和同一规则内各属性间的运算规则。

属性文法(Attribute Grammar)

- 属性

  对文法的每一个符号，引进一些属性，这些属性代表与文法符号相关的信息，如类型、值、存储位置等。

- 语义规则

  为文法的每一个产生式配备的计算属性的计算规则，称为语义规则。

- 属性文法是带属性的一种文法

  它的主要思想：

➢ 首先对于每个文法符号引进相关的属性符号；

➢ 其次对于每个产生式写出计算属性值的语义规则

- 属性文法的形式定义

  一个属性文法是一个三元组,A = (G, V, F)

  - G是一个上下文无关文法；
  - V是属性的有穷集；
  - F是关于属性的断言的有穷集。

说明：

1. 每个**属性**与文法符号相联，N.t表示文法符号N的属性t。**属性值**又称**语义值**。存储属性值的变量又称**语义变量**。
2. 每个断言与文法的某个产生式相联，写在{ }内。属性的断言又称**语义规则**，它所描述的工作可以包括属性计算、静态语义检查、符号表的操作、代码生成等，有时写成函数或过程段。

属性文法的例子

对于文法：

G[E]：

$E \rightarrow n$

$E \rightarrow E + n$

属性文法的例子

表达式值的计算的属性文法：

规则：$E \rightarrow n$　　语义动作 { E.val=n }

规则：$E \rightarrow E+n$ 语义动作 { E.val=E.val+n }

属性文法的例子

语法树生成的属性文法：

规则：$E \rightarrow n$ { p=new BTreeNode;p->data=n;
p->lchild=p->rchild=0;
E.root=p;
}

规则：$E_2 \rightarrow E_1 + n$ { p=new BTreeNode; p->data=n;
p->lchild=p->rchild=0;
nr=new BTreeNode; nr->data='+';
nr->lchild= E_1 .root;
nr->rchild=p;
E_2.root=nr;
}

属性文法的例子

汇编代码生成的属性文法：

规则：E → n　　{　Gen(Ldc　n)　}

规则：E → E + n　　{　Gen(Ldc　n)，　Gen(Adi)　}

练习：定义算术表达式计算的属性文法

（1）分析C语言算术表达式的书写格式：

3+4*5　　a*(b+c)

（2）按书写格式规律写出对应的C语言算术表达式文法

G[E′]

E′→ E
$E^{(1)}$→ $E^{(2)}$+T | $E^{(2)}$-T | T
$T^{(1)}$→ $T^{(2)}$*F | $T^{(2)}$/F | F
F→(E) | i

3+4*5

G[E']

E'→E
$E^{(1)}$→ $E^{(2)}$+T
$E^{(1)}$→ $E^{(2)}$-T
$E^{(1)}$→ T
$T^{(1)}$→ $T^{(2)}$*F
$T^{(1)}$→ $T^{(2)}$/F
$T^{(1)}$ → F
F→ (E)
F→ i

（3）定义属性文法：

规则	语义动作
1） E′→E	{ E′.val=E.val }
2） $E^{(1)}$→ $E^{(2)}$+T	{ $E^{(1)}$.val= $E^{(2)}$.val +T.val }
3） $E^{(1)}$→ $E^{(2)}$-T	{ $E^{(1)}$.val= $E^{(2)}$.val -T.val }
4） $E^{(1)}$→ T	{ $E^{(1)}$.val=T.val }
5） $T^{(1)}$→ $T^{(2)}$*F	{ $T^{(1)}$.val= $T^{(2)}$.val*F.val }
6） $T^{(1)}$→ $T^{(2)}$/F	{ $T^{(1)}$.val= $T^{(2)}$.val/F.val }
7） $T^{(1)}$→ F	{ $T^{(1)}$.val=F.val }
8） F→(E)	{ F.val=E.val }
9） F→ i	{ F.val=i }

有关语义分析的基本概念

- 语义分析的实施方式
  - 语法分析程序通过直接调用语义分析函数进行语义分析；
  - 先生成相应的语法树→再作语义分析。
- 语义分析的功能通常包括两个方面：
  - 检查语法结构的静态语义,即分析句子的含义是否有意义,实际工作多为作用域分析、类型的分析。
  - 将合乎语义的句子进行翻译，并生成某一种中间形式。

- 语句的种类
  - 说明语句：用于定义各种名字的属性；
  - 可执行语句：用于完成指定功能。

- 语义分析的任务种类
  - 分析说明语句——需要把所定义名字的各种属性登记到符号表，以便在分析到可执行语句时使用；
  - 分析可执行语句——首先根据各语句的语法结构和语义设计出相应目标代码结构,然后再给出从源语法结构到翻译过程图示的变换方法,语义分析程序则根据这些变换方法进行分析并生成中间代码。

简单例子

- 说明语句

int i,j,k;

name	type	kind	val	……
i	int	var		
j	int	var		
k	int	var		

符号表

- 可执行语句

i=2;

i=i+1;

1. (= , 2 , , i)

2. (+ , i , 1 , T)

3. (= , T , , i)

中间代码

符号表

标识符定义实体

实体属性保存在符号表

符号表的形式

每个名字对应一个表项

一个表项包括名字域和属性信息域

名字	属性信息

符号表

属性信息域

多个子域及标志位

类型、值、存储大小、相对地址

形参标志、说明标志、赋值标志

符号表的操作分析

- **说明语句**

  int i,j,k;

name	type	kind	val	……
i	int	var		
j	int	var		
k	int	var		

符号表

- **可执行语句**

  i=2;

  i=i+1;

符号表的存储结构

- 需要查找效率高
- 散列存储结构

符号表的散列存储方法

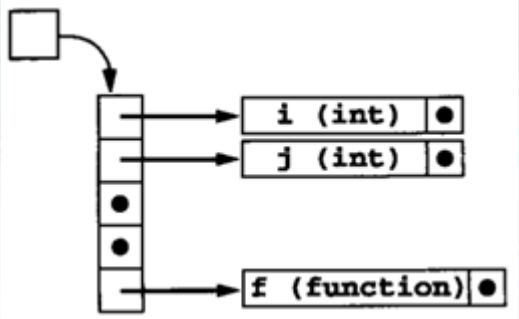
- 说明语句

int i,j,k;

name	type	kind	val	……
i	int	var		
j	int	var		
k	int	var		

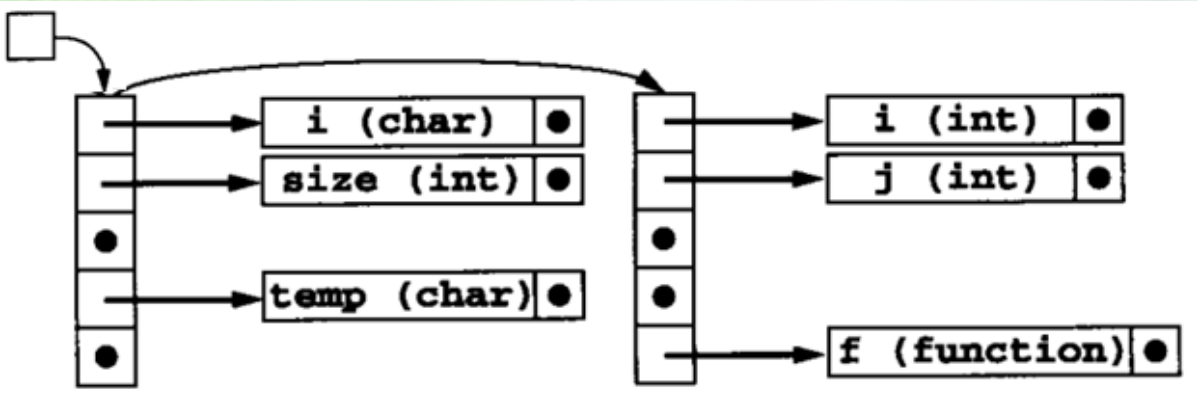
符号表如何解决作用域的问题

```
int  i,j;
int f()
{  char *j;    …   }
void main( )
{  int  size;
   char i,temp;
   f();
 }
```



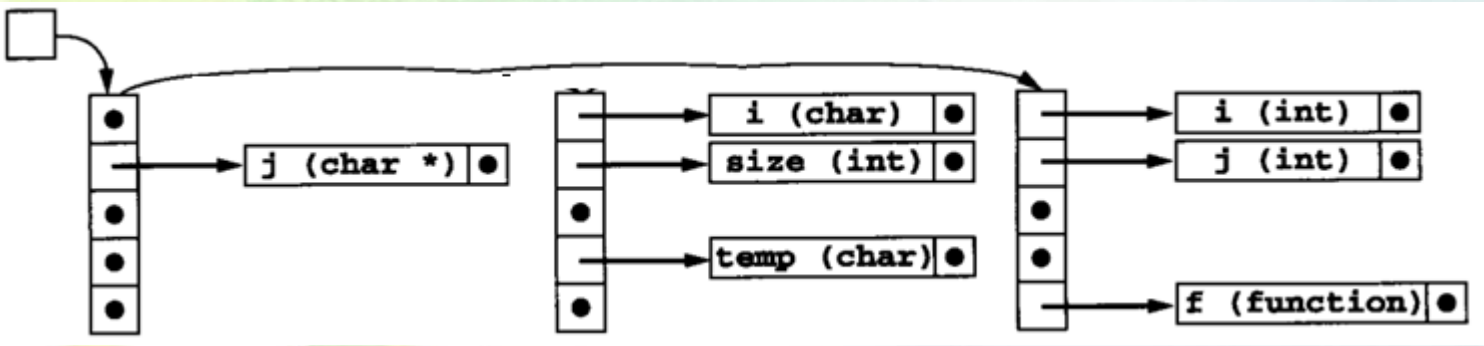
符号表如何解决作用域的问题

```
int  i,j;
int f()
{  char *j;    …   }
void main( )
{  int  size;
   char i,temp;
   f();
 }
```



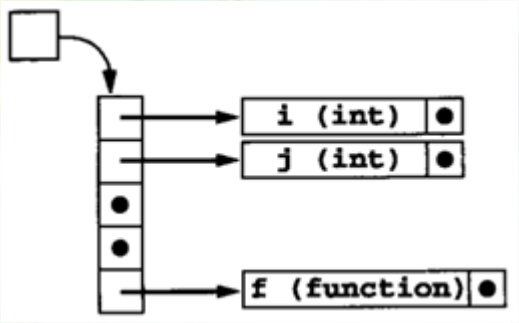
符号表如何解决作用域的问题

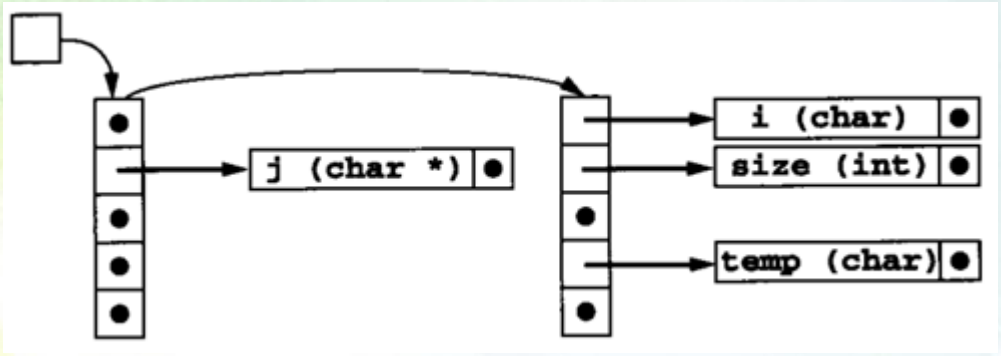
```
int  i,j;
int f()
{  char *j;    …   }
void main( )
{  int  size;
   char i,temp;
   f();
 }
```



符号表在程序运行时的作用

```
int  i,j;
int f()
{  char *j;    …   }
void main( )
{  int  size;
   char i,temp;
   f();
 }
```





符号表操作的函数

- 1)、LOOKUP(NAME)：以符号名NAME(标识符)查符号表，若表中已存在该标识符，则返回其在表中的位置(序号)，否则返回NULL。

- 2)、ENTER(NAME)：在符号表中新登记一名字为NAME的项，并返回该项在表中的位置(序号)。

- 3).ENTRY(NAME):查、填符号表的语义函数：

```
Pointer  ENTRY(NAME)
{
    ENTRYNo=LOOKUP(NAME);
    if(ENTRYNo==NULL)
        return(ENTER(NAME);
}
```

- 4）Fill(符号表位置，类型)属性填写函数

常见的中间代码形式

- 在语义函数的设计中，
  - 一方面要依赖于相应语法结构中相应规则的各个量的语义；
  - 另一方面也取决于要产生什么形式的中间代码。
- 常见的中间代码有哪些？
  - 树、后缀表示、三元组、四元组、P代码

(1) 中缀表示

- 运算符号位于两个运算对象中间，如a+b
- 不利于表达式的计算及目标代码的产生。

(2) 后缀表示——逆波兰表示

- 将运算符放在运算对象的后面，如 a b +
- 表达式中各运算符的出现顺序决定其计算的先后顺序，因此无括号。
- 后缀表达式与相应的中缀表达式中的运算对象的出现顺序是一致的。

- 问题1：如何将中缀表示转换成相应的后缀表示。
- 问题2：如何计算一个后缀表达式的值。
- 问题3：如何在原文法规则的基础上添加相应的语义函数，以便在做语法分析的同时调用相应的语义动作来实现中缀表示转换成相应的后缀表示。
  - 方案一:采用自顶向下分析——递归下降分析法
  - 方案二:采用自底向上分析——LR分析法

方案一 递归下降分析法

解决步骤:

(1) 写出进行简单算术运算的文法

[为了简单起见,只做+ * 括号运算]

E→E+T | T

T→T*F | F

F→ (E) | n

(2) 由于采用递归下降分析法,因此要先用EBNF改造文法.

文法改造结果:

E→T{+T}

T→F{*F }

F→(E)|n

(3)写出递归下降分析程序.

规则：　　E → T { + T } 的代码为：

```
void  E( )
  {
     T( );
      while (token == '+')
      {
          match (token) ;
          T( );
      } // while
  } // E
```

规则：　　T → F { * F } 的代码为：

```
void T()
{
  F();
  while (token == '*')
  {
     match (token);
     F();
  } //while
} // T
```

规则：F → (E) | n的递归动作：

```
void F()
{
 switch( token )
 {
   case '(':
          match('(') ;
          E();
          match(')') ;
          break;
   case number :
          match (n) ;
          break;
   default:
         error( ) ;
 } // switch
} // F
```

(4)根据要求进行语义动作的添加

- 后缀表示中运算对象的出现顺序与中缀表示一致
  - 意味着在分析到 n 的时候要把 n放入结果数组中.
  - 因此,在 match(n)后面增加这个语义动作即可.
- 后缀表示中运算符号的出现顺序就是计算的优先顺序
  - 由于文法规则本身已能反映运算符号的优先关系,因此递归动作同样具有运算符号优先关系的处理功能.
  - 因此运算符号的保存应该在所对应的运算对象都已保存之后.

规则：F → (E) | n的递归动作：

```
void  F( )
{
  switch( token )
  {
     case '(':
            match('(') ;
            E( );
            match(')') ;
            break;
     case number :
            match (n) ;
            post[p]=n; p++;     //post与p均为全局变量
            post[p]='@';  p++;
            break;
     default:
           error( ) ;
  } // switch
}// F
```

规则：　　T→F{*F}的代码为：

```
void T()
{
  F();
  while (token == '*')
  {
     match (token);
     F();
     post[p]='*'; p++;
  }//while
}//T
```

规则：　　E → T { + T } 的代码为：

```
void  E( )
  {
     T( );
      while (token == ‘+’)
      {
          match (token) ;
          T( );
          post[p]='+';  p++;
      } // while
  } // E
```

main()的安排

```
{
   p=0;
   getToken();
   E();
}
```

- 例子：3*(5+4)
- 运行结果为:3 5 4 + *

方案二：采用自底向上分析——LR法

解决步骤：

（1）写出进行简单算术运算的文法

[为了简单起见，只做+ * 括号运算]

E'→E

E→E+T | T

T→T*F | F

F→(E) | i

（2）画出对应的LR(0)DFA图——采用SLR(1)分析法

（3）构造规则的语义动作

各规则语义动作的定义过程

- 例子：3+4*5
- 用SLR(1)语法分析法进行语义分析并翻译生成后缀表达式的过程。
- 图示：

规则语义动作的构造方法：

- 根据后缀表示与前缀表示运算对象出现顺序的一致性——遇到运算对象则保存——栈
- 规则规约时相应的含义

因此

- 规则F→i 添加的语义动作为：i入栈
- 规则E→E+T 添加的语义动作为：+入栈
- 规则T→T*F 添加的语义动作为：*入栈

因此，对应规则的语义函数为：

规则	语义函数
E'→E	{ }
E→E+T	{ Post[p]='+'; p=p+1 }
E→T	{ }
T→T*F	{ }
T→F	{ Post[p]='*'; p=p+1 }
F→(E)	{ }
F→i	{ }
	{ Post[p]=i; p=p+1 }

- 问题1：如何将中缀表示转换成相应的后缀表示。
- 问题2：如何计算一个后缀表达式的值。

扩充的后缀表示

- 赋值语句

  a=e 则其对应的后缀表示为ae=

  如a=b*(c+b) 则后缀表示为 a b c b+*=

- 数组

  a[e] 则后缀表示为 e a SUBS

  其中SUBS 表示计算数组下标的运算。

- 条件语句

if (u) S1 else S2

后缀表示：

u L1 BZ S1 L2 BR S2

- BZ为双目运算符，表示当u不成立(为零)时转向标号L1部分继续执行
- L1表示语句S2开始执行的位置
- BR为一个单目运算符，表示无条件转向L2部分继续执行
- L2表示该条件语句下一个语句开始执行的位置。

一个例子

if (m > n) k=1 ； else m=0;

后缀表示：

m n > 10 BZ k1= 20 BR

10: m0=

20:

- While循环语句

while (u)

S1

后缀表示：

L2

u L1 BZ S1 L2 BR

L1

一个例子

```
while ( m > n )
    k=1 ;
```

后缀表示：

```
10:
m n >  20 BZ  k 1 =  10 BR
20:
```

- 其他循环语句

- for循环、do~while 循环、repeat ~ until 循环

- 可以根据其计算机的执行顺序分别转换为后缀表示即可。

2.四元组和三元组

2.1四元组的表示方法

(OP，P1，P2，T)

其中OP为运算符，OP1、OP2为运算对象，T为计算结果的临时暂存变量。

- 注意:在按语法制导翻译实际产生的四元组中，OP用一个整数码表示，它除了标识运算符的种类之外，还附带地表示其它一些语义特性。

注意事项：

- P1、P2、T 可代表的含义
  - 或是一个指向符号表某一登记的入口位置
  - 或是表示一个临时变量的整数码。

- 即意味着在产生中间代码的过程中，也相应地进行查造符号表的工作——与符号表打交道

一个例子：

对于表达式 a*(b+c)对应的四元组为：

(+, b, c, T1)
(*, a, T1, T2)

一个问题：　如何将一个中缀表达式表示成四元组

——手动方法

——语法分析加语义动作的方法

- 2.2三元组的表示方法

  (OP，P1，P2)

  其中OP为运算符，P1、P2为运算对象.

- 用元组编号来来代表结果保存的位置。
- 对于表达式 a*(b+c)对应的三元组为：
  (1) (+，b，c)
  (2) (*，a，(1))

- 问题：如何将一个中缀表达式表示成三元组。

  ——手动方法

  ——语法分析加语义动作的方法

2.3四元组的扩充表示

- 赋值语句　　a=e

先将e转换为四元组，在将 = 运算转换为四元组。

如a=b+c,则四元组表示为：

(+, b , c , T1)

(= ,T1 , ,a)

- 关系比较语句

  a rop b （rop 表示<、>、<>、>=、<=、=等关系比较运算符）

- 由于**关系运算通常都会作为条件语句或循环语句的条件**，因此通常将关系运算**转换为减法运算**，后面跟着**条件成立与否转移的四元组**。

如：a > b 则表示为四元组为

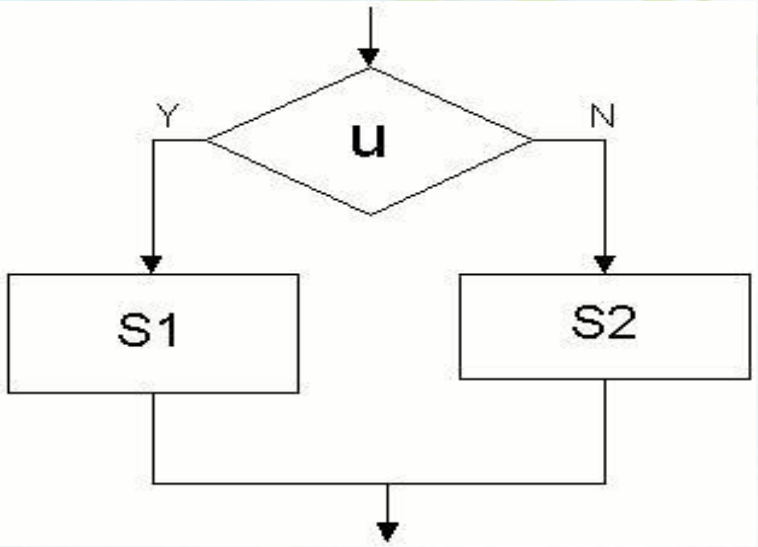
(—, a, b, T)　　**或**　　(J>, a, b, ?)

(BMZ, ?, , T)　　(J, , , ?)

注意：这只是其中的一种处理方法而已，在下面的章节中，我们将介绍其它的处理方案。

- 条件语句

  if (u) S1 else S2



- 按照计算机执行的顺序将其翻译成四元组，生成步骤如下
  - 将u转换为四元组。
  - 条件不成立转移，转换为四元组。
  - 语句S1转换为四元组。
  - 无条件转移转换为四元组。
  - 语句S2转换为四元组。

例子： if (m > n)

{ k=0; S=1; } else m=0;

四元组表示:

(1) (—, m , n , T1)

(2) (BMZ, 6 , , T1)

(3) (=, 0, , k)

(4) (=, 1, , S)

(5) (BR, 7 , ,)

(6) (=, 0, , m)

(7)

BMZ表示小于等于0转
BR表示无条件转
BZ 等于0
BLZ 大于等于0转

或

例子： if (m > n)

{ k=0; S=1; } else m=0；

四元组表示:

(1) (J>, m , n , 3)

(2) (J , , , 6)

(3) (=, 0, , k)

(4) (=, 1, , S)

(5) (J, , , 7)

(6) (=, 0, , m)

(7)

例，对语句 if (A<B && C>D) x=1; else x=0；进行翻译

(1)(j<，A，B，3)

(2)(j，_，_，7)

(3)(j>，C，D，5)

(4)(j，_，_，7)

(5) (=,1，，x)

(6)(j，，，?)

(7) (=,0，，x)

(8)

例，对语句 if (A<B || C>D) x=1; else x=0；进行翻译

(1)(j<, A, B, ?)

(2)(j, _, _, ?)

(3)(j>, C, D, ?)

(4)(j, _, _, ?)

(5) (=, 1, , x)

(6)(j, , , ?)

(7) (=, 0, , x)

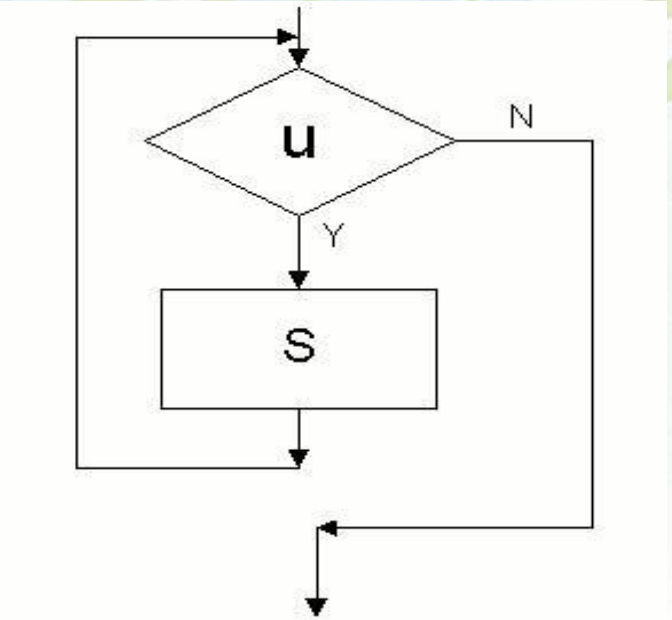
(8)

- 循环语句

下面以WHILE 循环为例来说

while (u)

S;



- 同样，按照计算机执行的顺序将其翻译成四元组，生成步骤如下
  - (1)将u转换为四元组。
  - (2)条件不成立转移，转换为四元组。
  - (3)语句S转换为四元组。
  - (4)无条件转移转换为四元组。

例子：while (m > n)

{ k=1; S=0; }

四元组表示：

(1)(—，m，n，T1)

(2)(BMZ，6，，T1)

(3)(=，1，，k)

(4)(=，0，，S)

(5)(BR，1，，)

(6)

其它循环表示形式如for，do while，请自行完成。

或

例子：while (m > n)

{ k=1; S=0; }

四元组表示：

(1)(J>，m，n，3)

(2)(J，，，6)

(3)(=，1，，k)

(4)(=，0，，S)

(5)(J，，，1)

(6)

- 2.4 三元组扩充表示

类似于四元组的相应部分。

注意：BMZ,BZ,BR,BLZ 等跳转三元组的表示格式：

(BMZ或BZ或BLZ，转移入口位置编号，判断值)

(BR，，转移的入口位置编号)

练习：算术表达式转换为三元组的属性文法

（1）分析C语言算术表达式的书写格式：

3+4*5　　a*(b+c)

（2）按书写格式规律写出对应的C语言算术表达式文法

G[E′]

$E' \rightarrow E$

$E^{(1)} \rightarrow E^{(2)}+T \mid E^{(2)}-T \mid T$

$T^{(1)} \rightarrow T^{(2)}*F \mid T^{(2)}/F \mid F$

$F \rightarrow (E) \mid i$

产生算术表达式的三元组

G[E′]

E′→E
$E^{(1)}$→ $E^{(2)}$+T
$E^{(1)}$→ $E^{(2)}$-T
$E^{(1)}$→ T
$T^{(1)}$→ $T^{(1)}$→ $T^{(2)}$*F
$T^{(1)}$→ $T^{(2)}$/F
$T^{(1)}$→ F
F→ (E)
F→ i

3+4*5

产生算术表达式的三元组

各规则语义函数的构造过程：

3+4*5

产生算术表达式的三元组

(1) 将计算的过程改为三元组的生成

(2) 语义动作定义如下：

规则	语义动作
(1) E′::=E	E′.val:=E.val
(2) $E^{(1)}::=E^{(2)}+T$	$E^{(1)}$.val:= GEN(+ ,$E^{(2)}$.val , T.val)
(3) $E^{(1)}::=E^{(2)}-T$	$E^{(1)}$.val:= GEN(– ,$E^{(2)}$.val ,T.val)
(4) E::=T	E.val:=T.val
(5) E::= –T	E.val:=GEN(Θ,T.val,)
(6) $T^{(1)}::=T^{(2)}*F$	$T^{(1)}$.val:= GEN(* ,$T^{(2)}$.val ,F.val)
(7) $T^{(1)}::=T^{(2)}/F$	$T^{(1)}$.val:= GEN(/ ,$T^{(2)}$.val ,F.val)
(8) T::= F	T.val:= F.val
(9) F::=(E)	F.val:=E.val
(10) F::= i	F.val=ENTRY(i)

说明：GEN()函数是产生三元组或四元组的函数

- Tiny语言源代码：

```
{ Sample program
  in TINY language--
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

- Tiny语言源代码

```
{ Sample program
  in TINY language--
  computes factorial
}
read x; { input an int
if 0 < x then { don't
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output
end
```

(1)(rd, , , x)
(2)(j<, 0, x, ?)
(3)(j, , , ?)
(4)(:=, 1, _, fact)
(5) (*, fact , x , T1)
(6)(:= ,T1 , , fact)
(7) (-, x ,1 , T2)
(8) (:=, T2 , , x)
(9)(j= ,x , 0 , 11)
(10)(j , , , 5)
(11)(WR, fact , ,)
(12)

（2）三地址（四元组）的处理过程

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

三地址（四元组表示）

```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

（3）三地址（四元组表示）的存储结构：

```
typedef enum {rd,gt,if_f,asn,lab,mul,
                    sub,eq,wri,halt,. . .} OpKind;
typedef enum {Empty,IntConst,String} AddrKind;
typedef struct
            { AddrKind kind;
              union
              { int val;
                char * name;
              } contents;
            } Address;
typedef struct
            { OpKind op;
              Address addr1,addr2,addr3;
            } Quad;
```

（4）三元组表示

```
(0)     (rd,x,_)
(1)     (gt,x,0)
(2)     (if_f,(1),(11))
(3)     (asn,1,fact)
(4)     (mul,fact,x)
(5)     (asn,(4),fact)
(6)     (sub,x,1)
(7)     (asn,(6),x)
(8)     (eq,x,0)
(9)     (if_f,(8),(4))
(10)    (wri,fact,_)
(11)    (halt,_,_)
```

- 3.伪代码——P代码
- 引入目的：增强移植性

- Tiny语言编译器中使用的P代码

```
2*a+(b-3)
```

这个表达式的P-代码版本如下：

```
ldc 2          ; load constant 2
lod a          ; load value of variable a
mpi            ; integer multiplication
lod b          ; load value of variable b
ldc 3          ; load constant 3
sbi            ; integer substraction
adi            ; integer addition
```

- Tiny Machine指令集(P347-P357) 请自行阅读

```
x := y + 1
```

对应于如下的P-代码指令：

```
lda x          ; load address of x
lod y          ; load value of y
ldc 1          ; load constant 1
adi            ; add
sto            ; store top to address
               ; below top & pop both
```

- Tiny语言源代码：

```
{ Sample program
  in TINY language--
  computes factorial
}
read x; { input an int
if 0 < x then { don't
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output
end
```

```
lda x       ; load address of x
rdi         ; read an integer, store to
            ; address on top of stack (& pop it)
lod x       ; load the value of x
ldc 0       ; load constant 0
grt         ; pop and compare top two values
            ; push Boolean result
fjp L1      ; pop Boolean value, jump to L1 if false
lda fact    ; load address of fact
ldc 1       ; load constant 1
sto         ; pop two values, storing first to
            ; address represented by second
lab L2      ; definition of label L2
lda fact    ; load address of fact
lod fact    ; load value of fact
lod x       ; load value of x
mpi         ; multiply
sto         ; store top to address of second & pop
lda x       ; load address of x
lod x       ; load value of x
ldc 1       ; load constant 1
sbi         ; subtract
sto         ; store (as before)
lod x       ; load value of x
ldc 0       ; load constant 0
equ         ; test for equality
fjp L2      ; jump to L2 if false
lod fact    ; load value of fact
wri         ; write top of stack & pop
lab L1      ; definition of label L1
stp
```

- 将该代码翻译成P代码为：

```
lda x          ; load address of x
rdi            ; read an integer, store to
               ; address on top of stack (& pop it)
lod x          ; load the value of x
ldc 0          ; load constant 0
grt            ; pop and compare top two values
               ; push Boolean result
fjp L1         ; pop Boolean value, jump to L1 if false
lda fact       ; load address of fact
ldc 1          ; load constant 1
sto            ; pop two values, storing first to
               ; address represented by second
lab L2         ; definition of label L2
lda fact       ; load address of fact
lod fact       ; load value of fact
lod x          ; load value of x
mpi            ; multiply
sto            ; store top to address of second & pop
lda x          ; load address of x
lod x          ; load value of x
ldc 1          ; load constant 1
sbi            ; subtract
sto            ; store (as before)
lod x          ; load value of x
ldc 0          ; load constant 0
equ            ; test for equality
fjp L2         ; jump to L2 if false
lod fact       ; load value of fact
wri            ; write top of stack & pop
lab L1         ; definition of label L1
stp
```

几种常用语句的翻译方法

- （1）算术表达式
- （2）说明语句
- （3）赋值语句
- （4）逻辑表达式
- （5）条件判断语句——if语句
- （6）循环语句——while、repeat、for语句

1.算术表达式的计算

(1) 分析C语言算术表达式的书写格式：

3+4*5　　a*(b+c)

(2) 按书写格式规律写出对应的C语言算术表达式文法

G[E′]

$E' \rightarrow E$

$E^{(1)} \rightarrow E^{(2)}+T \mid E^{(2)}-T \mid T \mid -T$

$T^{(1)} \rightarrow T^{(2)}*F \mid T^{(2)}/F \mid F$

$F \rightarrow (E) \mid i$

3+4*5

G[E']

E'→E

$E^{(1)}$→ $E^{(2)}$+T

$E^{(1)}$→ $E^{(2)}$-T

$E^{(1)}$→ T

$E^{(1)}$→ –T

$T^{(1)}$→ $T^{(2)}$*F

$T^{(1)}$→ $T^{(2)}$/F

$T^{(1)}$ → F

F→ (E)

F→ i

（3）分析算术表达式执行时的具体含义，以便观察是否需要进行文法规则的改造以及写出对应规则规约时需要执行的语义动作

1）$E' \rightarrow E$	{ $E'.val = E.val$ }
2）$E^{(1)} \rightarrow E^{(2)} + T$	{ $E^{(1)}.val = E^{(2)}.val + T.val$ }
3）$E^{(1)} \rightarrow E^{(2)} - T$	{ $E^{(1)}.val = E^{(2)}.val - T.val$ }
4）$E^{(1)} \rightarrow T$	{ $E^{(1)}.val = T.val$ }
5）$E^{(1)} \rightarrow -T$	{ $E^{(1)}.val = -T.val$ }
6）$T^{(1)} \rightarrow T^{(2)} * F$	{ $T^{(1)}.val = T^{(2)}.val * F.val$ }
7）$T^{(1)} \rightarrow T^{(2)} / F$	{ $T^{(1)}.val = T^{(2)}.val / F.val$ }
8）$T^{(1)} \rightarrow F$	{ $T^{(1)}.val = F.val$ }
9）$F \rightarrow (E)$	{ $F.val = E.val$ }
10）$F \rightarrow i$	{ $F.val = i$ }

产生算术表达式的三元组

G[E′]

E′→E

$E^{(1)}\rightarrow E^{(2)}+T$

$E^{(1)}\rightarrow E^{(2)}-T$

$E^{(1)}\rightarrow T$

$E^{(1)}\rightarrow -T$

$T^{(1)}\rightarrow T^{(1)}\rightarrow T^{(2)}*F$

$T^{(1)}\rightarrow T^{(2)}/F$

$T^{(1)}\rightarrow F$

F→ (E)

F→ i

3+4*5

产生算术表达式的三元组

各规则语义函数的构造过程：

3+4*5

产生算术表达式的三元组

(1) 将计算的过程改为三元组的生成

(2) 语义动作定义如下：

规则	语义动作
(1) E′::=E	E′.val:=E.val
(2) $E^{(1)}::=E^{(2)}+T$	$E^{(1)}$.val:= GEN(+ ,$E^{(2)}$.val , T.val)
(3) $E^{(1)}::=E^{(2)}-T$	$E^{(1)}$.val:= GEN(– ,$E^{(2)}$.val ,T.val)
(4) E::=T	E.val:=T.val
(5) E::= –T	E.val:=GEN(Θ,T.val,)
(6) $T^{(1)}::=T^{(2)}*F$	$T^{(1)}$.val:= GEN(* ,$T^{(2)}$.val ,F.val)
(7) $T^{(1)}::=T^{(2)}/F$	$T^{(1)}$.val:= GEN(/ ,$T^{(2)}$.val ,F.val)
(8) T::= F	T.val:= F.val
(9) F::=(E)	F.val:=E.val
(10) F::= i	F.val=ENTRY(i)

说明：GEN()函数是产生三元组或四元组的函数

问题：如果需要进行数据类型的检查，如何实现呢？

——增加变量类型属性：如用E.type表示E的类型信息

规则	语义动作

(2) $E^{(1)} := E^{(2)} + T$	if $E^{(2)}$.type=int and T.type =int then
	{ $E^{(1)}$.val:= GEN(+ ,$E^{(2)}$.val , T.val)
	$E^{(1)}$.type:=int
	}
	if $E^{(2)}$.type=real and T.type =real then
	{ $E^{(1)}$.val:= GEN(+ ,$E^{(2)}$.val , T.val)
	$E^{(1)}$.type:=real
	}
	if $E^{(2)}$.type=int and T .type=real then
	{ $E^{(2)}$.val :=GEN(ItoR, , $E^{(2)}$.val)
	$E^{(1)}$.val= GEN(+ ,$E^{(2)}$.val , T.val)
	} $E^{(1)}$.type=real
	if $E^{(2)}$.type=real and T.type =int then
	{ T.val :=GEN(ItoR, , T.val)
	$E^{(1)}$.val= GEN(+ ,$E^{(2)}$.val , T.val)
	} $E^{(1)}$.type=real

2.说明语句的翻译

- 语句的种类
  - 说明语句：用于定义各种名字的属性；
  - 可执行语句：用于完成指定功能。

- 语义分析的任务种类
  - 分析说明语句——需要把所定义名字的各种属性登记到符号表，以便在分析到可执行语句时使用；

简单例子

- 说明语句

int i,j,k;

name	type	kind	val	……
i	int	var		
j	int	var		
k	int	var		

符号表

- 可执行语句

i=2;

i=i+1;

1. (= , 2 , , i)

2. (+ , i , 1 , T)

3. (= , T , , i)

中间代码

符号表

标识符定义实体

实体属性保存在符号表

符号表的形式

每个名字对应一个表项

一个表项包括名字域和属性信息域

名字	属性信息

符号表

属性信息域

多个子域及标志位

类型、值、存储大小、相对地址

形参标志、说明标志、赋值标志

符号表的操作分析

- **说明语句**

  int i,j,k;

name	type	kind	val	……
i	int	var		
j	int	var		
k	int	var		

符号表

- **可执行语句**

  i=2;

  i=i+1;

符号表的存储结构

- 需要查找效率高
- 散列存储结构

符号表的散列存储方法

- 说明语句

int i,j,k;

name	type	kind	val	……
i	int	var		
j	int	var		
k	int	var		

符号表操作的函数

- 1)、LOOKUP(NAME)：以符号名NAME(标识符)查符号表，若表中已存在该标识符，则返回其在表中的位置(序号)，否则返回NULL。

- 2)、ENTER(NAME)：在符号表中新登记一名字为NAME的项，并返回该项在表中的位置(序号)。

- 3).ENTRY(NAME):查、填符号表的语义函数：

```
Pointer  ENTRY(NAME)
{
    ENTRYNo=LOOKUP(NAME);
    if(ENTRYNo==NULL)
        return(ENTER(NAME);
}
```

- 4）Fill(符号表位置，类型)属性填写函数

2.说明语句的翻译

- 对于象Pascal、C语言之类的强类型语言，要求在变量使用之前必须说明其类型。
- 说明语句的作用之一就是告知编译程序该变量的类型信息。
- 因此，在翻译说明语句时，语义动作函数只需要把类型信息填入符号表中。

符号表与语义分析的关系

说明语句：

int i,j,k;

name	type	kind	val	……
i	int	var		
j	int	var		
k	int	var		

C语言的说明语句

（1）分析C语言说明语句的书写格式：

```
int j;
int k,m,n;
float x;
```

（2）按书写格式规律写出对应的C语言说明语句文法

G[S]

S→T V；

V→V,i

V→i

T→int | char | float |double

方法一

(3分析说明语句执行时的具体含义，以便观察是否需要进行文法规则的改造以及写出对应规则规约时需要执行的语义动作

如： int k,m,n;

S→T V;
V→V,i
V→i
T→int | char | float |double

问题的分析：规约问题与语义动作问题。

语义函数的定义过程分析：根据文法对下面说明语句进行规约处理。

int k,m,n;

（4）为每条规则定义对应的语义动作

1）S→T V；　{ 将T.type填入所有V保存位置的type中 }

2）$V^{(1)}$→V,i　{ entry(i)；V保存的位置与i的位置
均保存起来，并传递给$V^{(1)}$ }

3）V→i　{ entry(i)；并把i的位置传递给V；}

4）T→int　{ T.type=int }

5）T→ char　{ T.type=char }

6）T→ float　{ T.type=float }

7）T→ double　{ T.type=double }

分析问题所在

G[S]

S→T V;

V→V,i

V→i

T→int | char | float |double

语义函数的定义过程分析：根据文法对下面说明语句进行规约处理。

int k,m,n;

方法二（解决方法）

(3分析说明语句执行时的具体含义，以便观察是否需要进行文法规则的改造以及写出对应规则规约时需要执行的语义动作

如： int k,m,n;

S→ V;
V→V,i
V→T i
T→int | char | float |double

（4）为每条规则定义对应的语义动作

1）S→V；　　{ }

2）$V^{(1)}$→V,i　{ fill(entry(i)，V.type); $V^{(1)}$.type=V.type }

3）V→T i　　{ fill(entry(i),T.type); V.type=T.type }

4）T→int　　{ T.type=int }

5）T→ char　　{ T.type=char }

6）T→ float　　{ T.type=float }

7）T→ double　{ T.type=double }

几种常用语句的翻译方法

- （1）赋值语句
- （2）逻辑表达式
- （3）条件判断语句——if语句
- （4）循环语句——while、repeat、for语句

1.赋值语句的翻译

（1）分析C语言赋值语句的书写格式：

x=3;　y=a+b*c;　z=(3+4)*2

（2）按书写格式规律写出对应的C语言赋值语句文法

S→id=E　　其中E简单算术表达式

（3）分析并理解其执行过程，以便写出语义动作

——将算术表达式E的值填入符号表中id的值属性中

——需要先查符号表

——还需要进行数据类型的检查

1.赋值语句的翻译

S→id=E的语义动作：

```
{
    p=LookUp(id);
    if (p!=NULL)
        Gen(=,E.Val,_,p.Val) ;
    else  ERROR( );
}
```

问题：如果需要进行数据类型的检查，如何实现呢？

——增加变量类型属性：

如用E.type表示E的类型信息，

做法类似于：

规则	语义动作

(2) $E^{(1)} := E^{(2)}+T$	if $E^{(2)}$.type=int and T.type =int then
	{ $E^{(1)}$.val:= GEN(+ ,$E^{(2)}$.val , T.val)
	$E^{(1)}$.type:=int
	}
	if $E^{(2)}$.type=real and T.type =real then
	{ $E^{(1)}$.val:= GEN(+ ,$E^{(2)}$.val , T.val)
	$E^{(1)}$.type:=real
	}
	if $E^{(2)}$.type=int and T .type=real then
	{ $E^{(2)}$.val :=GEN(ItoR, , $E^{(2)}$.val)
	$E^{(1)}$.val= GEN(+ ,$E^{(2)}$.val , T.val)
	} $E^{(1)}$.type=real
	if $E^{(2)}$.type=real and T.type =int then
	{ T.val :=GEN(ItoR, , T.val)
	$E^{(1)}$.val= GEN(+ ,$E^{(2)}$.val , T.val)
	} $E^{(1)}$.type=real

2.逻辑表达式的翻译

- C语言中,逻辑表达式可出现在

(1)X=(a>b) && (c<d)　　逻辑计算表达式

(2)if ((a>b) && (c<d))　x=0; else x=1;　条件

- 因此,逻辑表达式的作用有两个:
  - 用作控制语句中的条件表达式;
  - 用作逻辑赋值语句中的逻辑计算表达式

翻译方法及过程

（1）分析逻辑表达式的书写格式：

(a>b) && (c<d) || !(a==c)

运算符号有： 逻辑运算符 关系运算符

- 逻辑运算符号 ！ && ||

优先级高到低为!,&&,||；&&和||服从左结合

- 关系运算符号 rop（有<、<=、=、>=、>等）

  形如 E1 rop E2　　E1和E2是算术表达式

- 各关系运算符的优先级都相同，并且高于所有逻辑运算符，低于任何算术运算符

- 根据运算符号的优先级,逻辑表达式的文法规则可以表示为:

逻辑表达式的文法规则

1. BE→BE || BT
2. BE→BT
3. BT→BT && BF
4. BT→BF
5. BF→! BF
6. BF→(BE)
7. BF→E rop E
8. BF→i rop i
9. BF→i

G[E']

$E' \rightarrow E$

$E^{(1)} \rightarrow E^{(2)}+T$

$E^{(1)} \rightarrow E^{(2)}-T$

$E^{(1)} \rightarrow T$

$E^{(1)} \rightarrow -T$

$T^{(1)} \rightarrow T^{(2)}*F$

$T^{(1)} \rightarrow T^{(2)}/F$

$T^{(1)} \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow i$

(1)用作逻辑计算运算表达式的翻译

- 采用类似于算术表达式的语义动作定义方法

如　(1>2) && (3<4) || (5>3)

- 每条文法规则的语义动作定义如下

算术表达式值的计算对应的语义动作

1) $E' \rightarrow E$	{ $E'.val = E.val$ }
2) $E^{(1)} \rightarrow E^{(2)} + T$	{ $E^{(1)}.val = E^{(2)}.val + T.val$ }
3) $E^{(1)} \rightarrow E^{(2)} - T$	{ $E^{(1)}.val = E^{(2)}.val - T.val$ }
4) $E^{(1)} \rightarrow T$	{ $E^{(1)}.val = T.val$ }
5) $E^{(1)} \rightarrow -T$	{ $E^{(1)}.val = -T.val$ }
6) $T^{(1)} \rightarrow T^{(2)} * F$	{ $T^{(1)}.val = T^{(2)}.val * F.val$ }
7) $T^{(1)} \rightarrow T^{(2)} / F$	{ $T^{(1)}.val = T^{(2)}.val / F.val$ }
8) $T^{(1)} \rightarrow F$	{ $T^{(1)}.val = F.val$ }
9) $F \rightarrow (E)$	{ $F.val = E.val$ }
10) $F \rightarrow i$	{ $F.val = i$ }

1) $BE' \rightarrow BE$ { $BE'.val = BE.val$ }

2) $BE^{(1)} \rightarrow BE^{(2)} || BT$ { $BE^{(1)}.val = BE^{(2)}.val \ || \ BT.val$ }

3) $BE^{(1)} \rightarrow BT$ { $BE^{(1)}.val = BT.val$ }

4) $BT^{(1)} \rightarrow BT^{(2)} \&\& BF$ { $BT^{(1)}.val = BT^{(2)}.val \ \&\& \ BF.val$ }

5) $BT^{(1)} \rightarrow BF$ { $BT^{(1)}.val = BF.val$ }

6) $BF \rightarrow ! \ BF$ { $BF.val = ! \ BF.val$ }

7) $BF \rightarrow (\ BE \ )$ { $BF.val = BE.val$ }

8) $BF \rightarrow E \ rop \ E$ { $BF.val = E.val \ rop \ E.val$ }

9) $BF \rightarrow i \ rop \ i$ { $BF.val = i \ rop \ i$ }

10) $BF \rightarrow i$ { $BF.val = i$ }

- (2)用做控制语句中条件表达式的翻译
  - 其作用是用来选择下一个执行点
- 例如1：if ((a>b) && (c<d)) x=0; else x=1;

如果 a>b 的条件成立，

继续执行c<d,

如果c<d成立才执行x=0，否则执行x=1

如果 a>b 的条件不成立,则转到执行x=1

- 例如2：while ((a>b) || (c<d))
  x=1;

如果 a>b 的条件成立,则转到执行循环体 x=1

否则,则继续执行c<d,

如果c<d成立则转到执行循环体x=1，

否则转到执行循环语句的下一个语句

翻译方法及过程

- 出现在控制语句中条件表达式要根据逻辑运算符号进行分析.

- 每一个关系比较表达式都有条件成立和条件不成立的转向

  翻译方法：每个关系比较表达式都翻译出两个转向指令，一个成立时转向——真出口（TC）

  一个不成立转向——假出口（FC）

即对于规则

7.BF→E rop E　　产生两条指令

8.BF→i rop i　　产生两条指令

9.BF→i　　产生两条指令

逻辑表达式的文法规则

1. BE→BE || BT
2. BE→BT
3. BT→BT && BF
4. BT→BF
5. BF→! BF
6. BF→(BE)
7. BF→E rop E
8. BF→i rop i
9. BF→i

文法规则的语义动作的定义

7.BF→E rop E

8.BF→i rop i

9.BF→i

翻译方法

- 例：if (a>b)　x=0; else x=1;

- (a>b)翻译为：

1. (j>,a ,b , ?)

2. (j , , , ?)

8.BF→i1 rop i2　　语义动作：

(1)(jrop，i1，i2，p):真出口

(2)(j，_，_，q)：假出口

- 例：if (c) x=0; else x=1;

- 逻辑值 c 翻译为：

1. (jnz, c, , ?)

2. (j , , , ?)

9.BF→i　　语义动作：

(1)(jnz，i，_，p1):真出口

(2)(j，_，_，p2)：假出口

- 例：if (e+f > g+h)　x=0; else x=1;

- e+f>g+h 翻译为：

(1)　(+,e ,f , t1)

(2)　(+,g ,h ,t2)

(3)　(j>, t1 , t2 , ?)

(4)　(j ， ， ， ?)

7.BF→E1 rop E2　　语义动作：

(1)(jrop，E1.val，E2.val，p)：真出口

(2)(j，_，_，q)：假出口，

逻辑运算符的翻译

- 例如1：if (！ (a>b)) x=0; else x=1;

- (a>b)翻译为：

  1. (j>,a ,b , ?)
  2. (j , , , ?)

- ！ 翻译 —— 不成立转向变成成立时转向

  即真假出口进行交换

逻辑表达式的文法规则

1. BE→BE || BT

3.BT→BT && BF

7.BF→E rop E
8.BF→i rop i
9.BF→i

逻辑运算符的翻译

- 例如1：if ((a>b) && (c<d)) x=0; else x=1;
- (a>b)翻译为：
  1. (j>,a ,b , ?)
  2. (j , , , ?)
- (c<d) 翻译为：
  3. (j<,c ,d , ?)
  4. (j , , , ?)
- && 翻译 —— (1) 填写成立时转向的入口点
  (2) 填写不成立时转向的入口点

假出口链

例，对语句 if (A && B && C>D) 进行翻译

(1)(jnz，A，_，3)　　真出口

(2)(j，_，_，0)　　假出口

(3)(jnz，B，_，5)　　真出口

(4)(j，_，_，2)　　假出口

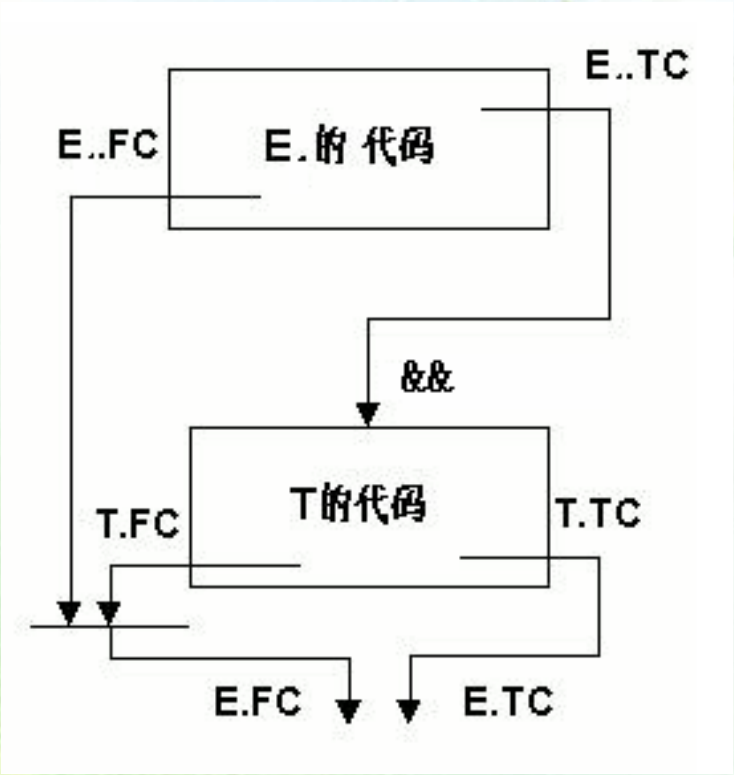
(5)(j>，C，D，7)　　真出口

(6)(j，_，_，4)　　假出口

(7)

生成了一条假出口链表

&& 翻译方法图示



E && T

定义E&&T的语义动作

从E&&T翻译图示,发现要对规则BT → BT && BF进行改造：因为遇到&&的时候要做相应的处理

$BT^{and} \rightarrow BT$ &&

$BT \rightarrow BT^{and}$ BF

- 当使用规则$BT^{and} \rightarrow BT$ && 归约时，就可及时后一个表达式的第一个四元组位置回填给BT的真出口链。

(5)BT^{and} → BT &&

{

BackPatch(BT.TC,NextStat);

BT^{and}.FC=BT.FC;

}

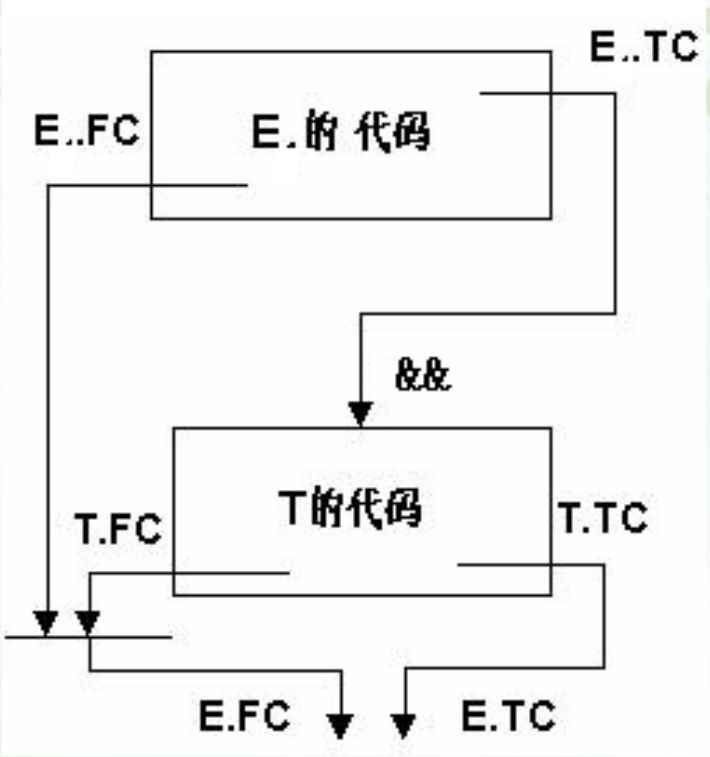
(4)BT → BT^{and} BF

{

BT.TC=BF.TC;

BT.FC=Merge(BT^{and}.FC ,BF.FC);

}



逻辑表达式的文法规则

1. BE→BE || BT

3.BT→BT && BF

7.BF→E rop E
8.BF→i rop i
9.BF→i

逻辑运算符的翻译

- 例如1：if ((a>b) || (c<d)) x=0; else x=1;
- (a>b)翻译为：

  1. (j>,a ,b , ?)
  2. (j , , , ?)

- (c<d) 翻译为：

  3. (j<,c ,d , ?)
  4. (j , , , ?)

- || 翻译——(1) 填写不成立时转向的入口点
  (2) 填写成立时转向的入口点

真出口链

例，对语句 if (A || B || C>D) 进行翻译

(1)(jnz，A，_，7)　　真出口

(2)(j，_，_，3)　　假出口

(3)(jnz，B，_，7)　　真出口

(4)(j，_，_，5)　　假出口

(5)(j>，C，D，7)　　真出口

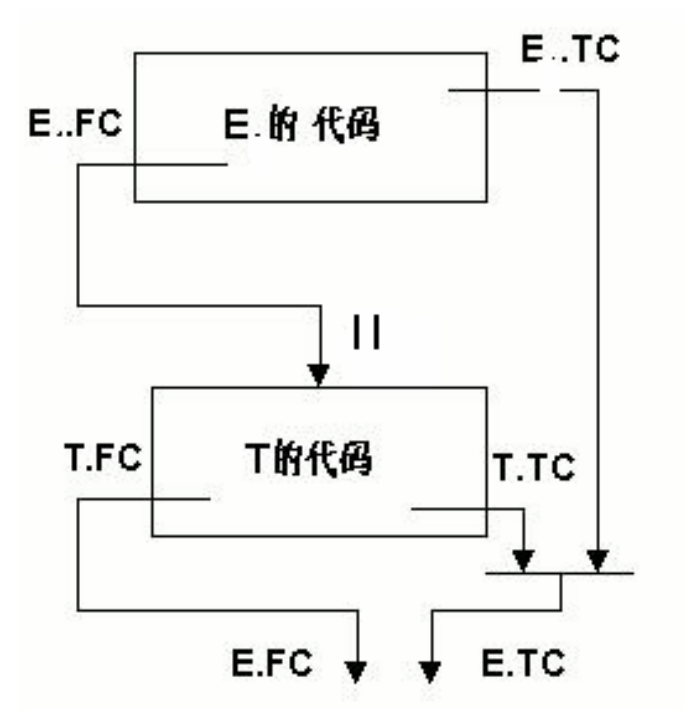
(6)(j，_，_，0)　　假出口

(7)

回填

生成了一条真出口链表

|| 翻译方法图示



E || T

定义E||T的语义动作

- E||T的翻译图示可以知道：遇到||就要做相应的处理
- 因此，有必将文法规则 BE→BE || BT 进行改造

$$BE^{or} \rightarrow BE \ ||$$
$$BE \rightarrow BE^{or} \ BT$$

- 当使用规则$BE^{or} \rightarrow BE \ ||$归约时，就可立即执行回填BE的假出口。

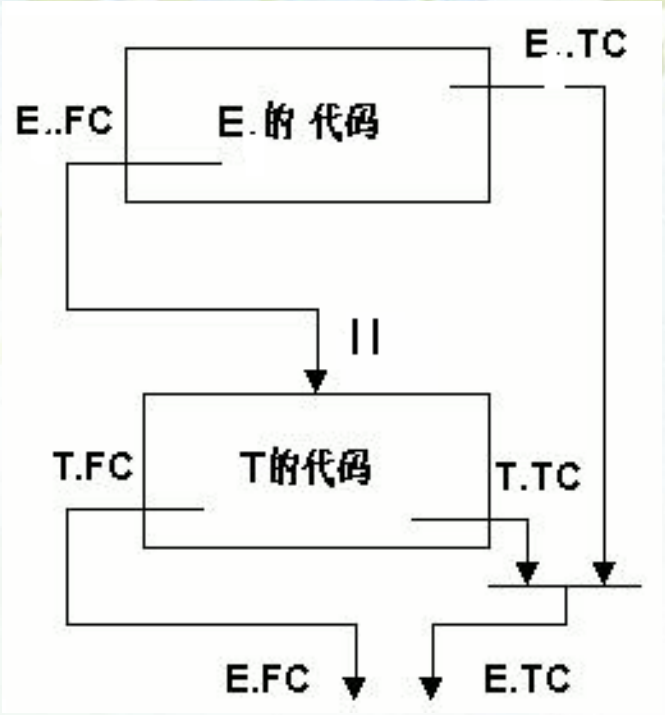
(2)BE^{or} → BE ||

{

BackPatch(BE.FC,NextStat);

BE^{or}.TC =BE.TC;

}



(1)BE → BE^{or} BT

{

BE.FC=BT.FC;

BE.TC=Merge(BE^{or}.TC ,BT.TC);

}

归 纳 小 结

逻辑表达式的翻译，需要先对文法进行改写

(1)BE→BE^{or} BT
(2)BE^{or} → BE ||
(3)BE → BT
(4)BT → BT^{and} BF
(5)BT^{and} →BT &&
(6)BT → BF
(7)BF →(BE)
(8)BF → ！ BF
(9)BF → E rop E
(10)BF → i rop i
(11)BF → i

写出每条规则的语义动作

写出每条规则的语义动作

- (11)BF → i
  {
  BF.TC=NextStat;
  BF.FC=NextStat+1;
  Gen(jnz,ENTRY(i),_,0);
  Gen(j,_,_,0);
  }

(10)BF → $i^{(1)}$ rop $i^{(2)}$ 其中rop 为< ,<=,<>,>=,>

```
{
    BF.TC=NextStat;
    BF.FC=NextStat+1;
    Gen(jrop, i(1) , i(2) ,0);
    Gen(j,_,_,0);
}
```

(9) BF → $E^{(1)}$ rop $E^{(2)}$ 其中E是算术表达式

```
{
    BF.TC=NextStat;
    BF.FC=NextStat+1;
    Gen(jrop, E(1) .Val, E(2).Val,0);
    Gen(j,_,_,0);
}
```

(8)BF → ! $BF^{(1)}$

{

BF.TC → $BF^{(1)}$ .FC ;

BF.FC → $BF^{(1)}$ .TC;

}

(7)BF → (BE)

{

BF.TC → BE.TC ;

BF.FC → BE.FC;

}

(5)BT^{and} → BT &&

{

BackPatch(BT.TC,NextStat);

BT^{and}.FC=BT.FC;

}

(4)BT → BT^{and} BF

{

BT.TC=BF.TC;

BT.FC=Merge(BT^{and}.FC ,BF.FC);

}

(2)BE^{or} → BE ||

{

BackPatch(BE.FC,NextStat);

BE^{or}.TC =BE.TC;

}

(1)BE → BE^{or} BT

{

BE.FC=BT.FC;

BE.TC=Merge(BE^{or}.TC ,BT.TC);

}

结 论

- 为了更好地对逻辑表达式进行处理，需要先做如下的定义：

- (1)指示器NextStat，指向下一个即将形成的四元组编号。NextStat的初值为1,每执行一次产生代码过程Gen后，NextStat将自动增1。

(2)回填函数BackPatch(P,t)：把以P为链首的四元组链上的每个四元组第四分量都填上t。

回填算法如下：

```
void  BackPatch(P , t)
{
   Q=P;
   while (Q!=0)
   {
      S=四元组Q的第四个分量的内容;
      把t填入四元组Q的第四个分量上;
      Q=S;
   }
}
```

- (3)合并——函数Merge(P1,P2)
- 把以P1,P2为链首的两个四元组链合二为一，返回合并后的链首。
- 当P2为空(=0)时，则合并后的链首为P1,否则合并后的链首为P2。

合并算法如下：

```
Pointer  Merge(P1,P2)
{
  if (P2==0) return P1;
   else
  {
      P=P2;
      while (四元组P的第4个分量不为0)
            P=四元组P的第四个分量内容;
       把P1填入四元组P的第四个分量上;
       return P2;
  }
}
```

3、IF语句的翻译

例，对语句 if (A && B && C>D) x=1; else x=0 ;
进行翻译

(1)(jnz，A，_，3)
(2)(j，_，_，9)
(3)(jnz，B，_，5)
(4)(j，_，_，9)
(5)(j>，C，D，7)
(6)(j，_，_，9)
(7) (=，1，，x)
(8)(j，，，10)
(9) (=，0，，x)
(10)

例如：if (A && B && C>D)
if (X<Y) F=1 ; else F=0 ;
else G=1 ; 则其四元组序列为：

(1)(jnz，A，_，3) // A && B && C>D
(2)(j，_，_，13)
(3)(jnz，B，_，5)
(4)(j，_，_，13)
(5)(j>，C，D，7)
(6)(j，_，_，13)
(7)(j<，X，Y，9) // X<Y 的四元组
(8)(j，_，_，11)
(9)(=，1，_，F) // F=1
(10)(j，_，_，14)
(11)(=，0，_，F) // F=0
(12)(j，_，_，14)
(13)(=，1，_，G) // G=1
(14)

注意：上面第(10)个四元组(j，_，_，14),不仅应跳过F=0的计算,而且还应跳过G=1的计算,第(12)个四元组的作用是为跳过外层else部分。

if语句的文法规则

- if语句文法规则为：

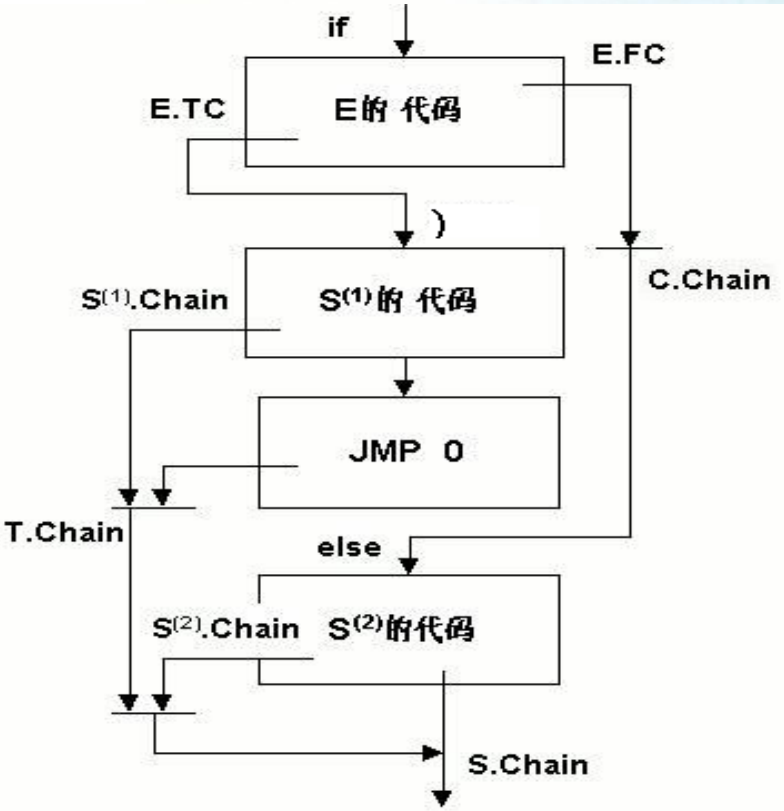
  $S \rightarrow$ if （E） $S^{(1)}$

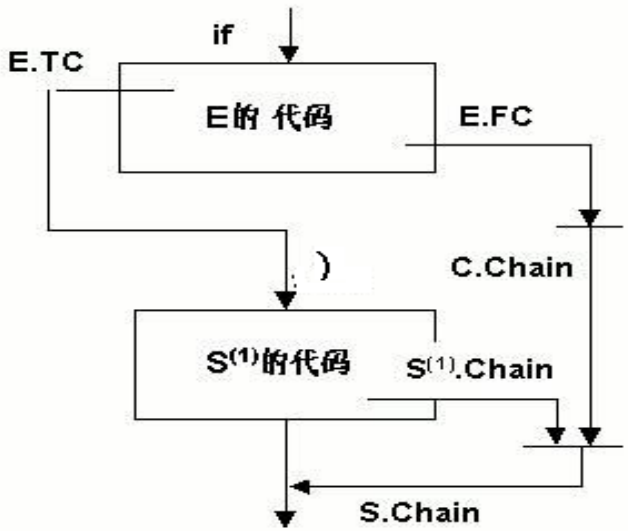
  $S \rightarrow$ if （E） $S^{(1)}$ else $S^{(2)}$

- 由于$S^{(1)}$、$S^{(2)}$本身又可以是if语句或其它语句，所以if语句可以是嵌套的。

- if语句的翻译过程用图示描述为：

if语句翻译过程图示





自底向上分析法只能在使用某条规则进行归约时，才能调用相应的语义动作，因此需要把if语句的文法改造——分别以 ） 和 else 来分割文法

if语句文法规则改造

以）和else作为规则结尾将原文法进行分割改造

$S \rightarrow if\ (E)\ S^{(1)}$

$S \rightarrow if\ (E)\ S^{(1)}\ else\ S^{(2)}$

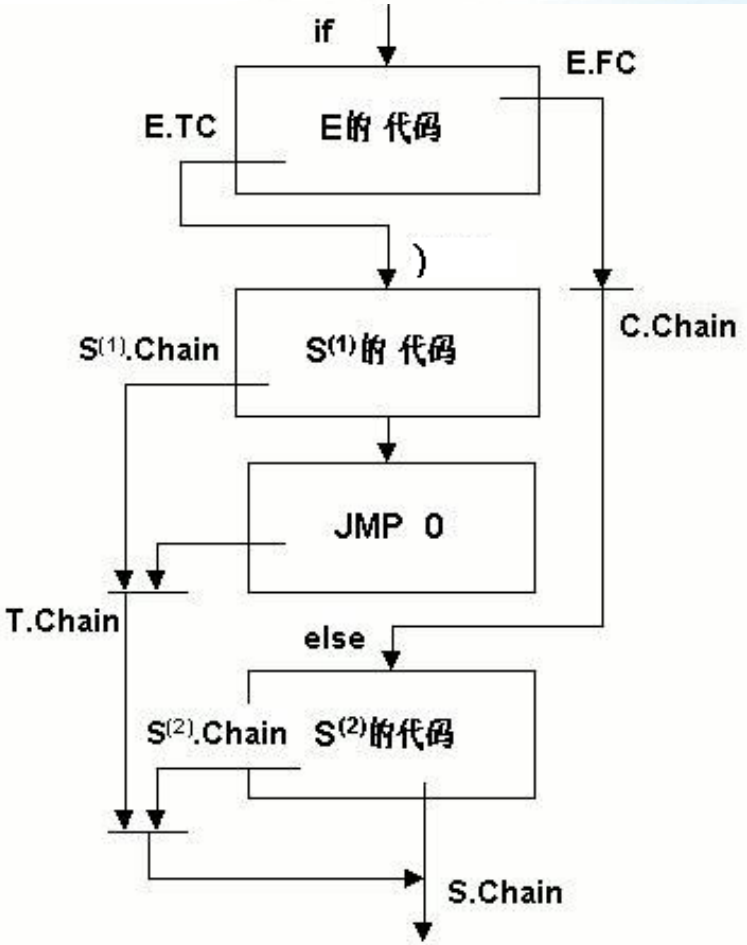
改造为：

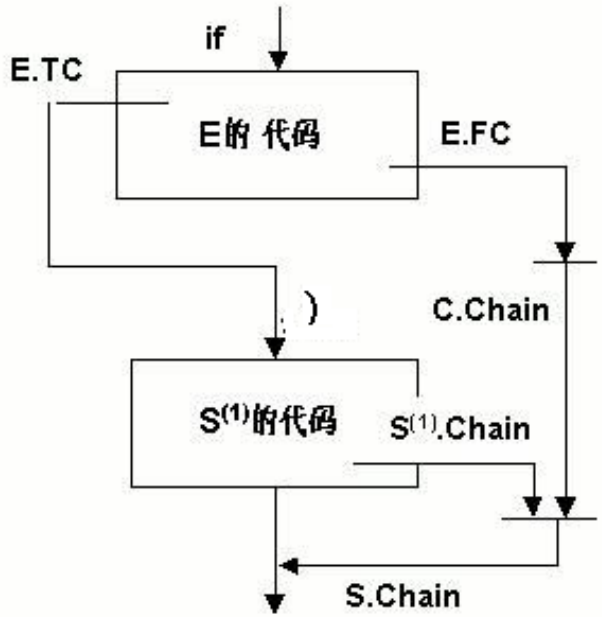
$C \rightarrow if\ (E)$

$T \rightarrow C\ S^{(1)}\ else$

$S \rightarrow T\ S^{(2)}$

$S \rightarrow C\ S^{(1)}$


if
E.FC
E.TC
E的 代码
)
C.Chain
S(1).Chain
S(1)的 代码
JMP 0
T.Chain
else
S(2).Chain
S(2)的代码
S.Chain


if
E.TC
E的 代码
E.FC
)
C.Chain
S(1)的代码
S(1).Chain
S.Chain

定义语义动作

定义if语句每条规则对应的语义动作

C → if （E）
{
BackPatch(E.TC, NextStat);
C.Chain=E.FC;
}

- T → CS(1) else
  { q=NextStat;
     Gen(j, _, _, 0);
     BackPatch(C.Chain , NextStat);
     T.Chain=Merge(S(1).Chain , q);
  }

- S → T S(2)
  {
     S.Chain=Merge(T.Chain ,S(2).Chain);
  }

- (4)S → C $S^{(1)}$

  {

  S.Chain=Merge(C.Chain ,$S^{(1)}$.Chain);

  }

- 最后(3)、(4)两个规则的语义动作中，并不会立刻回填T(或C)的Chain，这是考虑到了语句可能嵌套的情况，转移目标暂时还不能确定。

- 因此，最后建立总的S.Chain，留待转移目标明确后(如遇到“；”)，再回填。

4、while语句的翻译

例，对语句 while (A && B && C>D) x=1 ; 进行翻译

(1)(jnz，A，_，3)

(2)(j，_，_，9)

(3)(jnz，B，_，5)

(4)(j，_，_，9)

(5)(j>，C，D，7)

(6)(j，_，_，9)

(7) (=，1，，x)

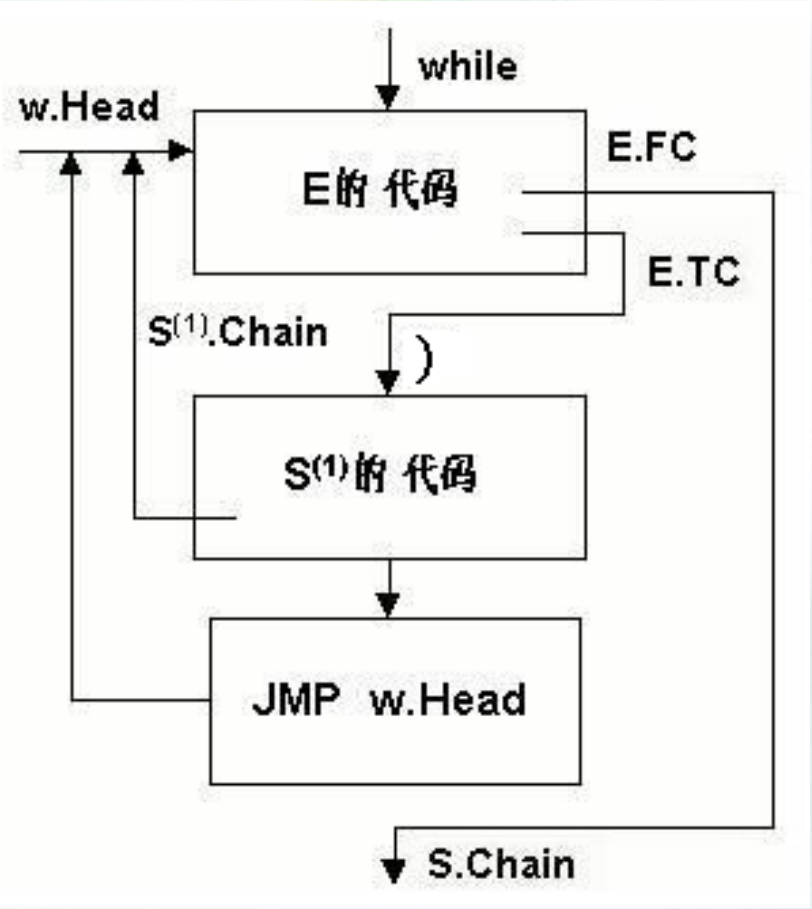
(8)(j，，，1)

(9)

4、while语句的文法规则

文法表示为： $S \rightarrow$ while （E） $S^{(1)}$

while语句的翻译过程图示：



因为遇到while和)都要进行相应的处理，因此

（1）以while分割

（2）以）分割

以便在规约时做这些处理

因此,我们将while语句的文法改进为:

(1)W → while

(2)W^d → W （E）

(3)S → W^d $S^{(1)}$

于是，为每条规则定义相应的语义动作：

- (1)W → while
  {
  W.Head=NextStat;
  }

- (2)$W^d \rightarrow W$ (E)

  { BackPatch(E.TC,NextStat);

  W^d.Chain=E.FC;

  W^d.Head=W.Head;

  }

- (3)$S \rightarrow W^d\ S^{(1)}$

  { BackPatch($S^{(1)}$.Chain , W^d.Head);

  Gen(j,_,_, W^d.Head);

  S.Chain=W^d.Chain;

  }

• 小　结

- 语句或语法成分的翻译方法，解决问题的基本方法：
- （1）研究各种语句的代码结构；
- （2）根据**代码结构的特点**和只有**在规约时**才能调用语义动作这一关系，对原文法进行适当的改造，使得在翻译时能及时调用相应的语义动作，执行某些重要的语义操作。

- 九、一个具体的分析例子

语句： while（A<B）if（C>D）X=Y+Z;

为例，说明按前述语义动作将它翻译为四元组序列的过程。

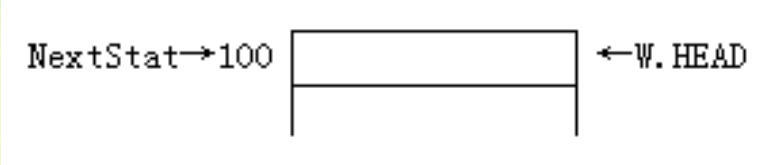
为了确定起见，我们假定所产生的四元组序列从编号100开始(即开始时，NextStat之值为100)。

翻译的步骤如下：

- 第一步：对于所给句子，首先按如下的规则进行归约，并执行相应的语义动作：

W → while {W.Head=NextStat; }

此时W.Head之值为100，且四元组表为：



- 第二步：对第一步所得句型

  W（A<B） if （C>D ） X=Y+Z

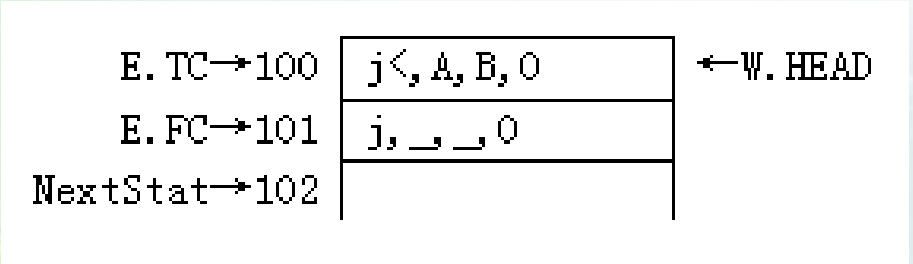
E → $i^{(1)}$ rop $i^{(2)}$

{ E.TC=NextStat; E.FC=NextStat+1;

Gen(jrop, $i^{(1)}$, $i^{(2)}$,0);

Gen(j,_,_,0);

}



- 第三步：对第二步所得的句型

  W（E） if （C>D） X=Y+Z

分别按如下的规则以语义动作进行归约和处理：

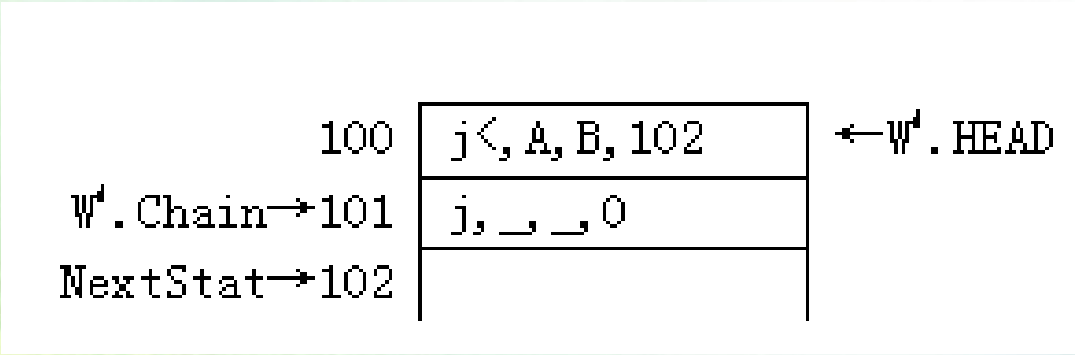
$W^d \rightarrow W$（E）

{ BackPatch(E.TC,NextStat);

W^d.Chain=E.FC; W^d.Head=W.Head;

}

四元组为：



- 第四步：对第三步所得句型

  W^d if （C>D）X=Y+Z

  中的关系表达式C>D，四元组为：

100	j<, A, B, 102	←W'.HEAD
W'.Chain→101	j, , , 0	
$E^{(1)}$.TC→102	j>, C, D, 0	
$E^{(1)}$.FC→103	j, _, _, 0	
NextStat→104		

- 第五步：对第四步所得句型

  W^d if （$E^{(1)}$）X=Y+Z

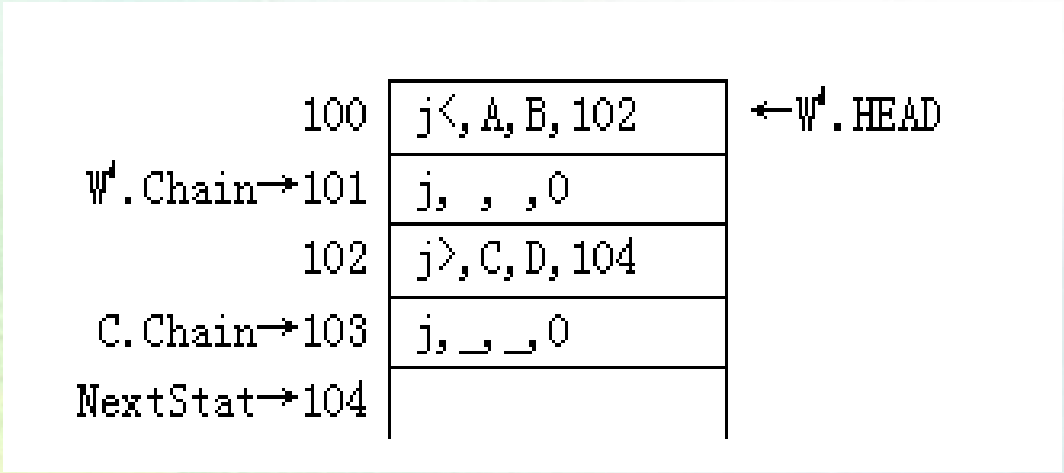
分别按如下的规则及语义动作进行归约和处理：

C→if（E）

{ BackPatch(E.TC, NextStat);

C.Chain=E.FC; }

则四元组为：



- 第六步：对第五步所得句型
  W^d C X=Y+Z

归约和处理其中的赋值语句X=Y+Z，

$E^{(1)}$ → $E^{(2)}$+T　/*加法运算的文法规则 */
{ T1=NewTemp; Gen(+ ,$E^{(2)}$.val , T.val ,T1); }

S → id=E /* 赋值语句的文法规则 */
{
  p=LookUp(id);
  if (p!=NULL) Gen(=,E.Val,_,p.Val)
    else ERROR();
  S.Chain=0;
}

于是,我们可以得到句型：
W^d C S

且此时四元组表为：

$S^{(1)}$.Chain:=0

100	j<,A,B,102	←W'.HEAD
W'.Chain→101	j, , ,0	
102	j>,C,D,104	
C.Chain→103	j,_,_,0	
104	+, Y,Z,T	
105	:=,T,_,X	
NextStat→106		

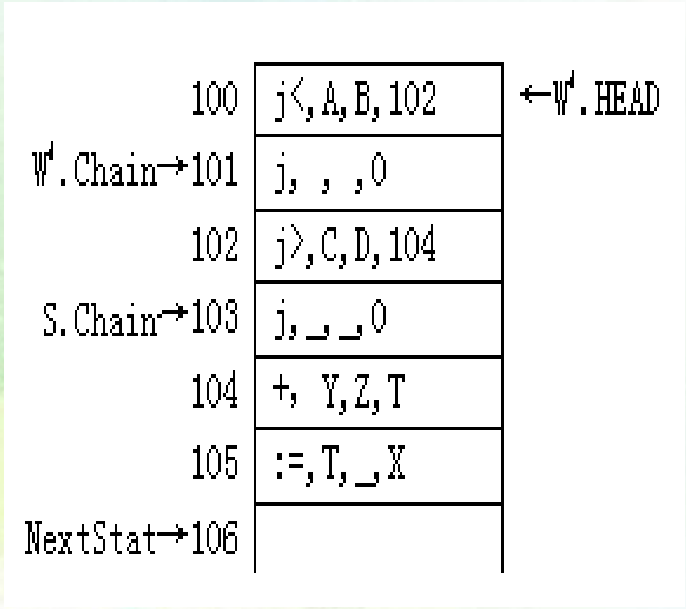
- 第七步：对上述句型，分别按如下规则和语义动作进行归约和处理：

  $S \rightarrow C\ S^{(1)}$

  {

  S.Chain=Merge(C.Chain ,$S^{(1)}$.Chain);

  }



- 第八步：对句型

  W^d S

分别按下面的规则和语义动作进行归约和处理：

S → W^d $S^{(1)}$

{

BackPatch($S^{(1)}$.Chain , W^d.Head);

Gen(j,_,_, W^d.Head);

S.Chain=W^d.Chain;

}

则四元组表为：

100	j<,A,B,102
S.Chain→101	j, , ,0
102	j>,C,D,104
103	j,_,_,100
104	+, Y,Z,T
105	:=,T,_,X
106	j,_,_,100
NextStat→107	