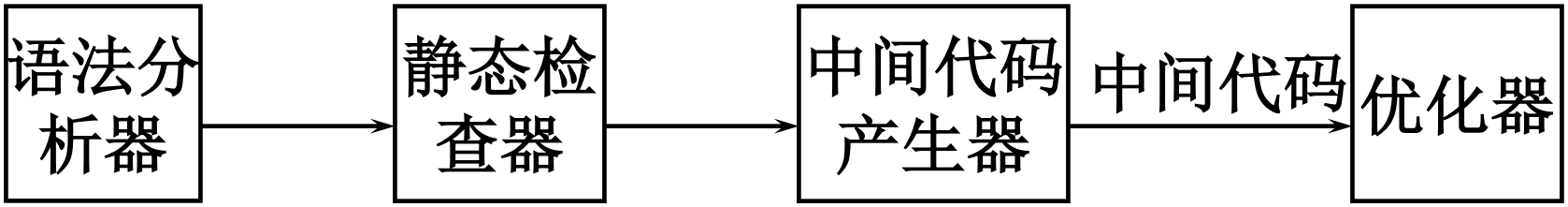
第七章 语义分析和中间代码产生

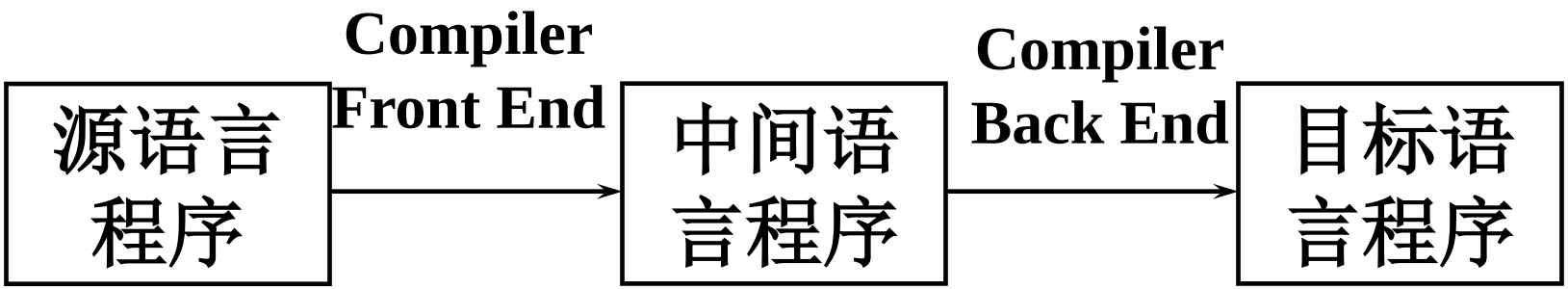
同济大学计算机系

概述

- 静态语义检查
  - 类型检查
  - 控制流检查
  - 一致性检查
  - 相关名字检查
  - 名字的作用域分析



- 中间语言（复杂性界于源语言和目标语言之间）的好处：
  - 便于进行与机器无关的代码优化工作
  - 易于移植
  - 使编译程序的结构在逻辑上更为简单明确



内容线索

- 中间语言
- 说明语句
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

中间语言

- 常用的中间语言
  - 后缀式，逆波兰表示
  - 图表示
    - DAG
    - 抽象语法树
  - 三地址代码
    - 三元式
    - 四元式
    - 间接三元式

后缀式

- **后缀式**表示法：Lukasiewicz发明的一种表示表达式的方法，又称**逆波兰**表示法。
- 一个表达式E的后缀形式可以如下定义：
  1. 如果E是一个变量或常量，则E的后缀式是E自身。
  2. 如果E是E_1 op E_2形式的表达式，其中op是任何二元操作符，则E的**后缀式为**E_1' E_2' op，其中E_1' 和E_2' 分别为E_1 和E_2的后缀式。
  3. 如果E是(E_1)形式的表达式，则E_1的后缀式就是E的后缀式。

- 逆波兰表示法不用括号。只要知道每个算符的目数，对于后缀式，不论从哪一端进行扫描，都能对它进行唯一分解。
- 后缀式的计算
  - 用一个栈实现。
  - 一般的计算过程是：自左至右扫描后缀式，每碰到运算量就把它推进栈。每碰到k目运算符就把它作用于栈顶的k个项，并用运算结果代替这k个项。

把表达式翻译成后缀式的语义规则描述

产生式	语义规则
$E \rightarrow E^{(1)}\ op\ E^{(2)}$	$E.code := E^{(1)}.code \ \|\| \ E^{(2)}.code \ \|\| op$
$E \rightarrow (E^{(1)})$	$E.code := E^{(1)}.code$
$E \rightarrow id$	$E.code := id$

- E.code表示E后缀形式
- op表示任意二元操作符
- “||”表示后缀形式的连接

$E \rightarrow E^{(1)} op\ E^{(2)}$	$E.code := E^{(1)}.code \parallel E^{(2)}.code \parallel op$
$E \rightarrow (E^{(1)})$	$E.code := E^{(1)}.code$
$E \rightarrow id$	$E.code := id$

- 数组POST存放后缀式：k为下标，初值为1
- 上述语义动作可实现为：

产生式	程序段
$E \rightarrow E^{(1)} op\ E^{(2)}$	{POST[k]:=op;k:=k+1}
$E \rightarrow (E^{(1)})$	{}
$E \rightarrow i$	{POST[k]:=i;k:=k+1}

- 例：输入串a+b+c的分析和翻译

POST:

1	2	3	4	5	
a	b	+	c	+	…

图表示法

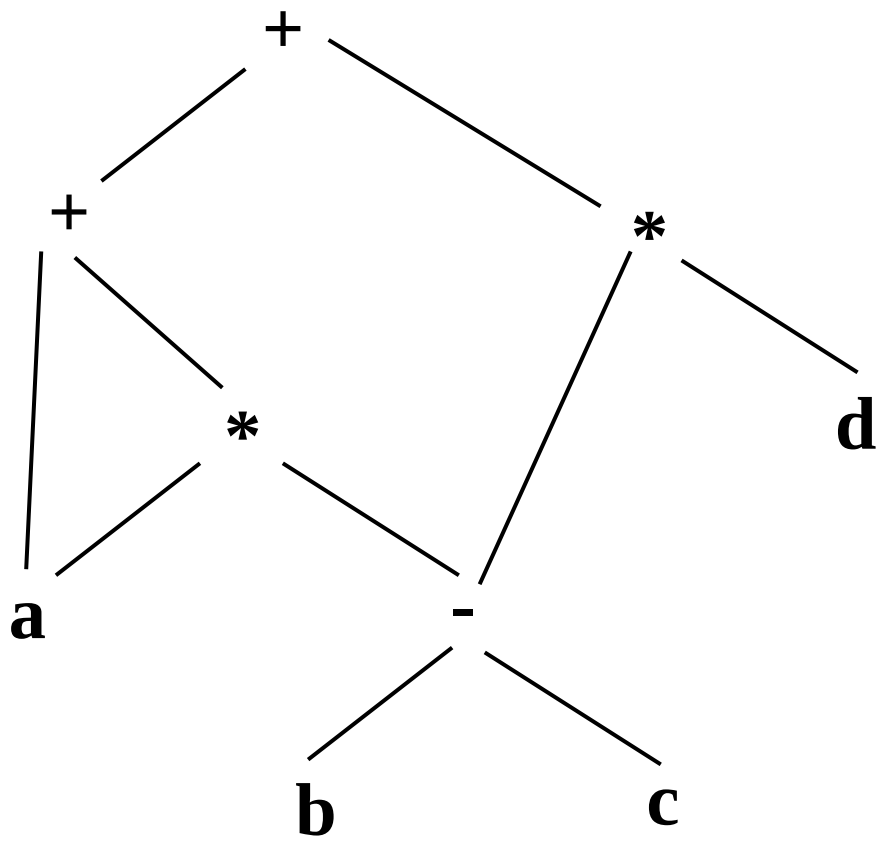
- DAG
- 抽象语法树

DAG

- 有向无循环图(Directed Acyclic Graph，简称DAG)
  - 对表达式中的每个子表达式，DAG中都有一个结点
  - 一个内部结点代表一个操作符，它的孩子代表操作数
  - 在一个DAG中代表公共子表达式的结点具有多个父结点

a+a*(b-c)+(b-c)*d的图表示法

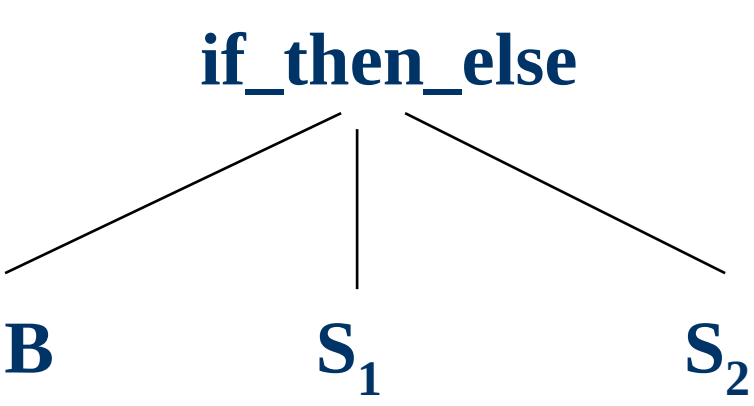
后缀表达式：aabc-*+bc- d *+



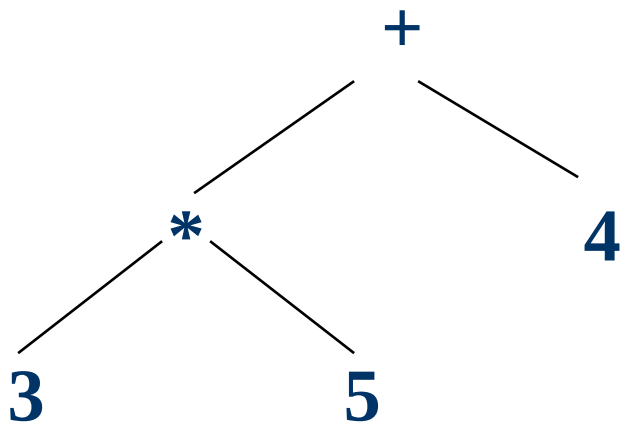
抽象语法树

- 在语法树中去掉那些对翻译不必要的信息，从而获得更有效的源程序中间表示。这种经变换后的语法树称之为**抽象语法树**(Abstract Syntax Tree)

  - $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$



  - 3*5+4



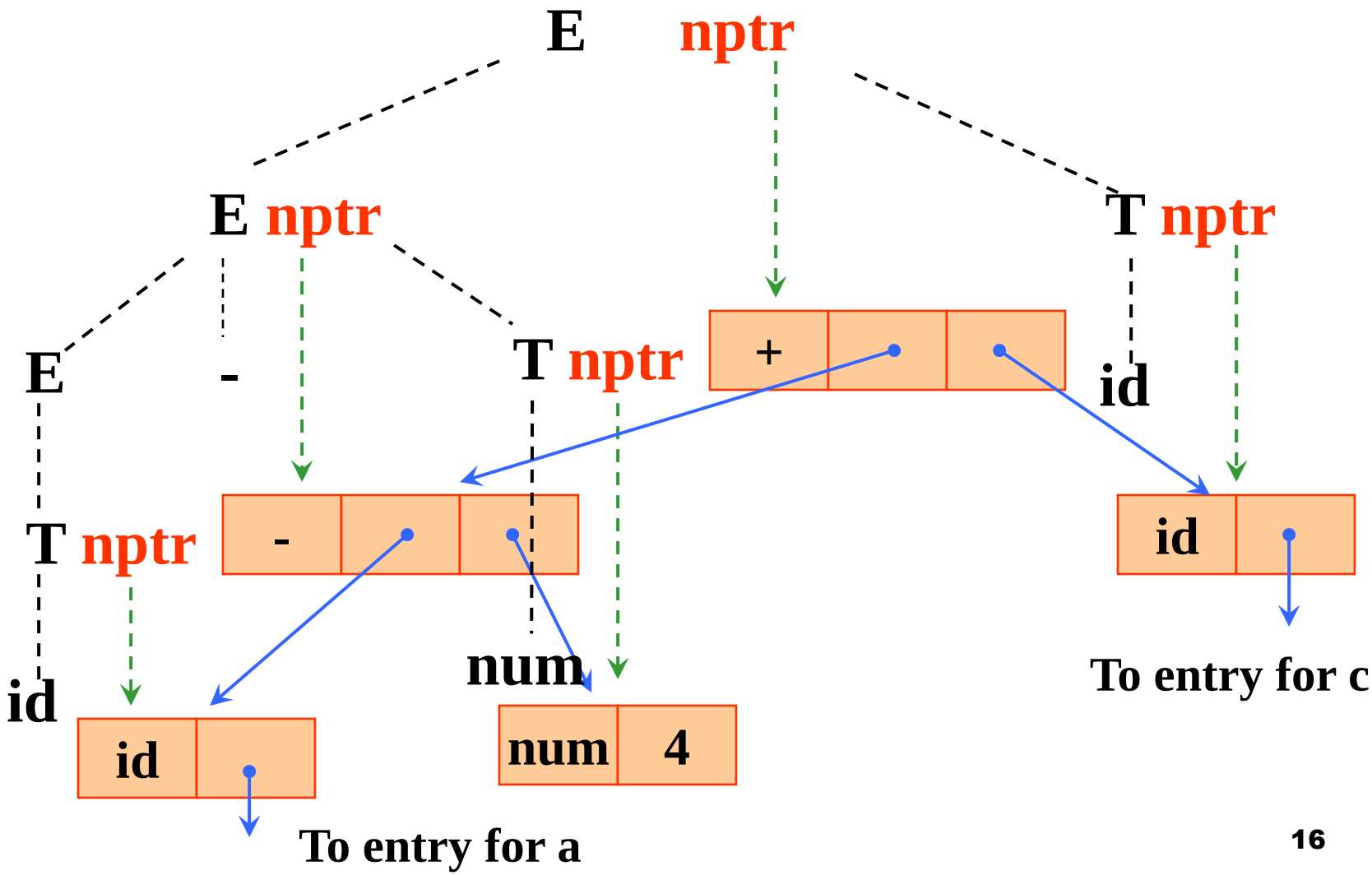
建立表达式的抽象语法树

- **mknode (op,left,right)** 建立一个运算符号结点，标号是op，两个域left和right分别指向左子树和右子树。
- **mkleaf (id,entry)** 建立一个标识符结点，标号为id，一个域entry指向标识符在符号表中的入口。
- **mkleaf (num,val)** 建立一个数结点，标号为num，一个域val用于存放数的值。

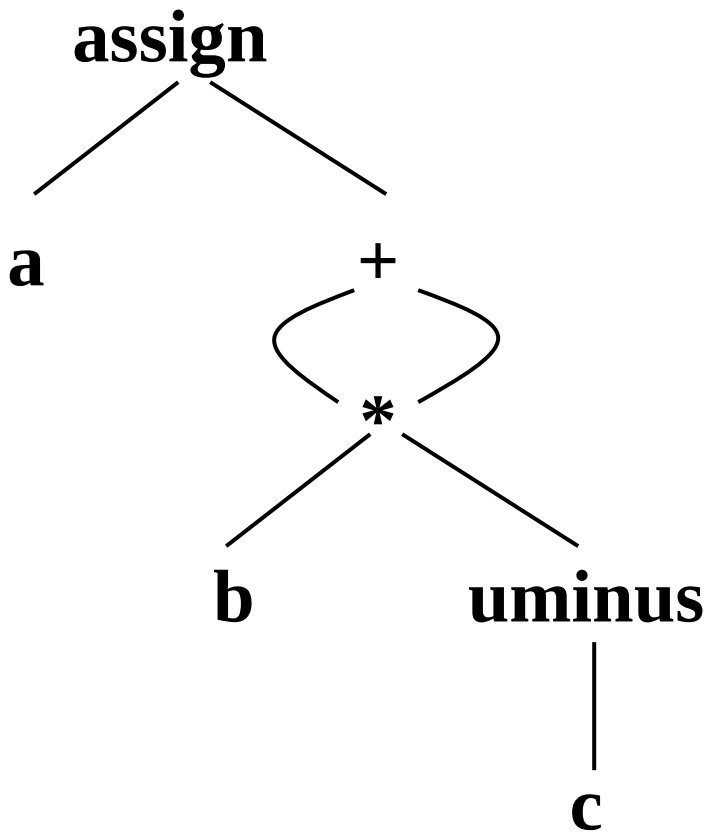
建立抽象语法树的语义规则

产生式	语义规则
$E \rightarrow E_1+T$	E.nptr := mknode (‘+’ , E_1.nptr, T.nptr)
$E \rightarrow E_1-T$	E.nptr := mknode (‘-’ , E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow (E)$	T.nptr := E.nptr
$T \rightarrow id$	T.nptr := mkleaf (id, id.entry)
$T \rightarrow num$	T.nptr := mkleaf (num, num.val)

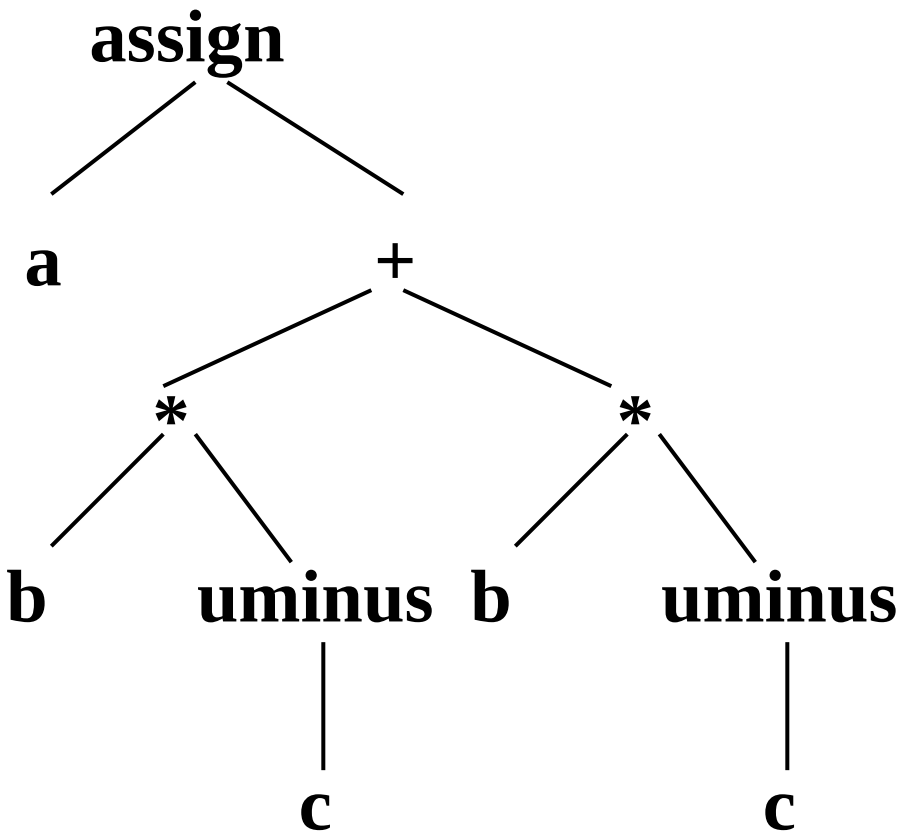
a－4＋c的抽象语法树


E
nptr
E
nptr
T
nptr
E
-
T
nptr
+
id
T
nptr
-
id
id
num
id
num
4
To entry for c
To entry for a

a:=b*(-c)+b*(-c)的图表示法



DAG



抽象语法树

产生赋值语句抽象语法树的属性文法

产生式	语义规则
$S \rightarrow id:=E$	S.nptr:=mknode(‘assign’, mkleaf(id,id.place),E.nptr)
$E \rightarrow E_1+E_2$	E.nptr:=mknode(‘+’,E_1.nptr,E_2.nptr)
$E \rightarrow E_1*E_2$	E.nptr:=mknode(‘*’,E_1.nptr,E_2.nptr)
$E \rightarrow -E_1$	E.nptr:=mknode(‘uminus’,E_1.nptr)
$E \rightarrow (E_1)$	E.nptr:=E_1.nptr
$E \rightarrow id$	E.nptr:=mkleaf(id,id.place)

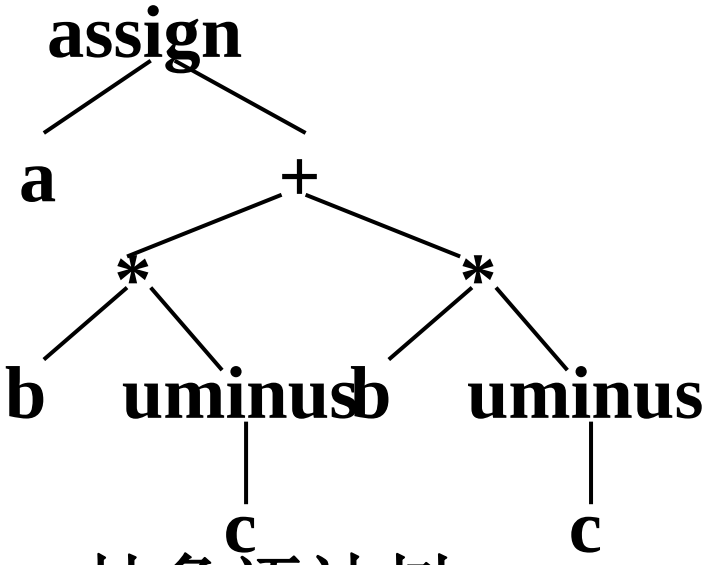
三地址代码

- 三地址代码

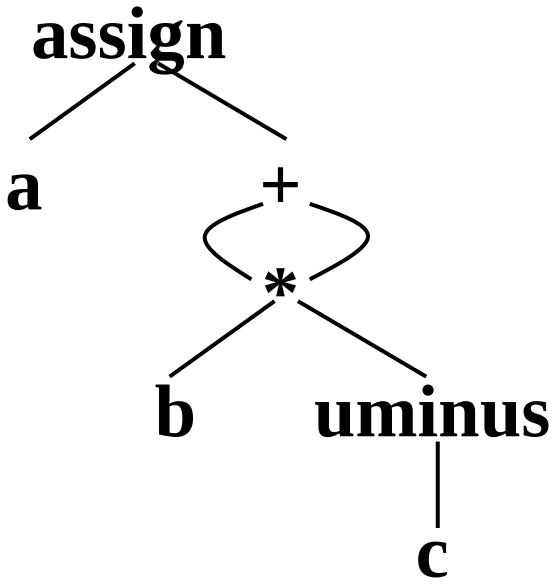
  x:=y op z

- 三地址代码可以看成是抽象语法树或DAG的一种线性表示

a:=b*(-c)+b*(-c)的图表示法



抽象语法树



DAG

抽象语法树对应的代码：

T_1:=-c
T_2:=b*T_1
T_3:=-c
T_4:=b*T_3
T_5:=T_2+T_4
a:=T_5

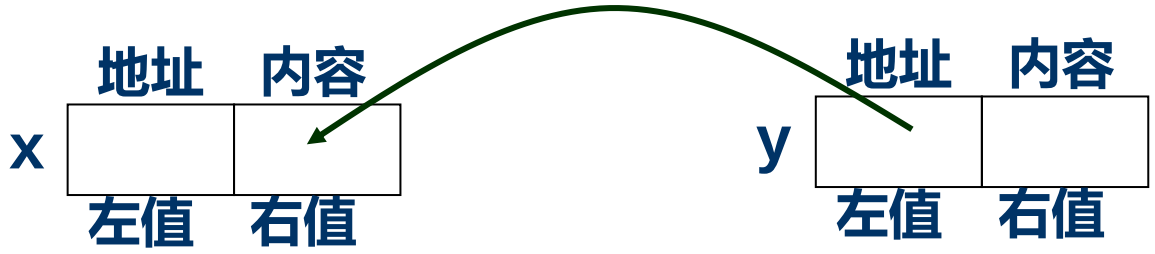
DAG对应的代码：

T_1:=-c
T_2:=b*T_1
T_5:=T_2+T_2
a:=T_5

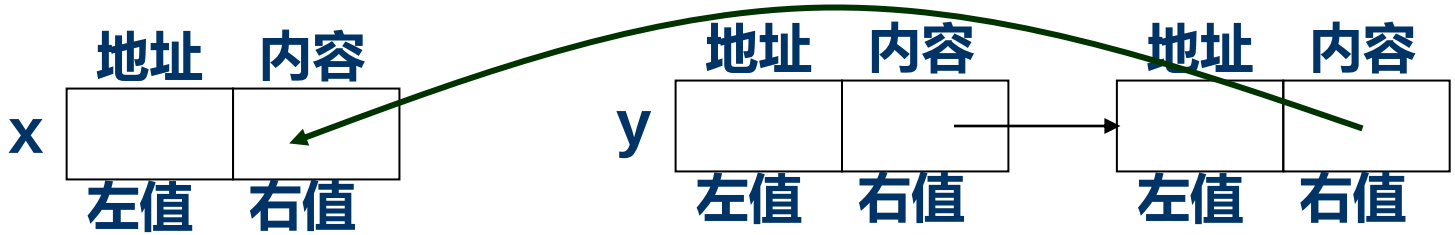
三地址语句的种类

- x:=y op z
- x:=op y
- x:=y
- goto L
- if x relop y goto L或if a goto L
- param x和call p,n，以及返回语句return y
- x:=y[i]及x[i]:=y的索引赋值
- x:=&y, x:=*y和*x:=y的地址和指针赋值

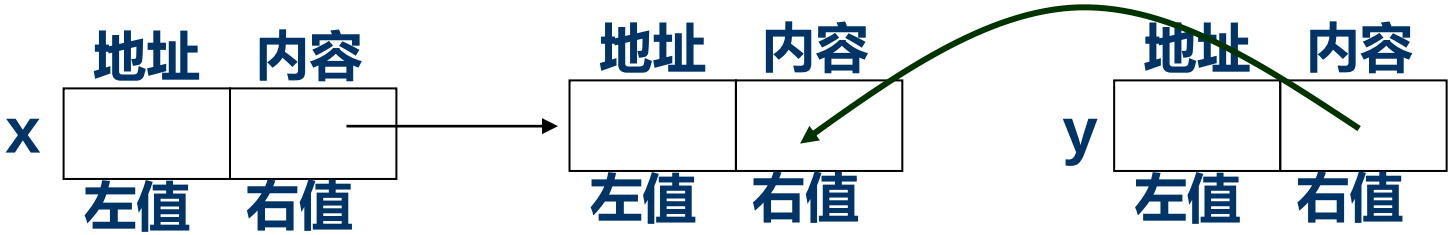
- x:=&y 把y的地址赋给x，其中y是一个名字，或者是一个临时变量，代表一个具有左值的表达式



- x:=*y 把y所指示的地址单元里存放的内容赋给x，其中y是一个指针或者是一个其右值为地址的临时变量。



- *x:=y 把y的右值赋给x所指向的对象的右值



语句的三地址代码

- 生成三地址代码时，临时变量的名字对应抽象语法树的内部结点
  - 产生式E →E1+E2
    - E的值放到一个新的临时变量T中
  - 赋值语句id:=E的三地址代码包括：
    （1）对表达式E求值并置于变量T中
    （2）进行赋值 id.place:=T

赋值语句的文法

产生式

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow id$

- 非终结符号S有综合属性S.code，它代表赋值语句S的三地址代码。
- 非终结符号E有如下两个属性：
  - E.place表示存放E值的名字。
  - E.code表示对E求值的三地址语句序列。
  - 函数newtemp的功能是，每次调用它时，将返回一个不同临时变量名字,如$T_1, T_2, \cdots$。
  - gen表示生成三地址语句
- 三地址语句序列往往是被存放到一个输出文件中

S-属性文法定义

产生式	语义规则
$S \rightarrow id := E$	S.code:=E.code || gen(id.place ‘:=’ E.place)
$E \rightarrow E_1 + E_2$	E.place:=newtemp; $E.code := E_1.code \,\|\|\, E_2.code \,\|\|$ gen(E.place ‘:=’ E_1.place ‘+’ E_2.place)
$E \rightarrow E_1 * E_2$	E.place:=newtemp; $E.code := E_1.code \,\|\|\, E_2.code \,\|\|$ gen(E.place ‘:=’ E_1.place ‘*’ E_2.place)
$E \rightarrow -E_1$	E.place:=newtemp; $E.code := E_1.code \,\|\|$ gen(E.place ‘:=’ ‘uminus’ E_1.place)
$E \rightarrow (E_1)$	$E.place := E_1.place$; $E.code := E_1.code$
$E \rightarrow id$	E.place:=id.place; E.code= ‘ ’

三地址语句

a:=b*(-c)+b*(-c)

- 四元式
  - 一个带有四个域的记录结构，这四个域分别称为op, arg1, arg2及result

	op	arg1	arg2	result
(0)	uminus	c		T_1
(1)	*	b	T_1	T_2
(2)	uminus	c		T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	:=	T_5		a

  - 四元式之间的联系通过临时变量实现。
  - 单目运算只用arg1域，转移语句将目标标号放入result域。
  - arg1,arg2,result通常为指针，指向有关名字的符号表入口，且临时变量填入符号表。

三地址语句

a:=b*(-c)+b*(-c)

- 三元式
  - 通过计算临时变量值的语句的位置来引用这个临时变量
  - 三个域：op、arg1和arg2

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

三地址语句

- 多目运算的三元式

例. x[i]:=y

	op	arg1	arg2
(0)	[ ] =	x	i
(1)	assign	(0)	y

例. x:=y[i]

	op	arg1	arg2
(0)	= [ ]	y	i
(1)	assign	x	(0)

三地址语句

- 间接三元式
  - 为了便于代码优化，用三元式表+间接码表表示中间代码
    - 间接码表:一张指示器表，按运算的先后次序列出有关三元式在三元式表中的位置。
  - 优点: 方便优化，节省空间

- 例如，语句

X:=(A+B)*C;

Y:=D↑(A+B)

的间接三元式表示如下表所示。

间接代码
(1)
(2)
(3)
(1)
(4)
(5)

三元式表

	OP	ARG1	ARG2
(1)	+	A	B
(2)	*	(1)	C
(3)	:=	X	(2)
(4)	↑	D	(1)
(5)	:=	Y	(4)

四元式、三元式和间接三元式比较

- 三元式中使用了指向三元式的指针，优化时修改较难。
- 间接三元式优化只需要更改间接码表，并节省三元式表存储空间。
- 修改四元式表也较容易，只是临时变量要填入符号表，占据一定存储空间。

内容线索

- ✓ 中间语言
- ■ **说明语句的翻译**
- ■ 赋值语句的翻译
- ■ 布尔表达式的翻译
- ■ 控制语句的翻译
- ■ 过程调用的处理

概述

- 说明部分中把定义性出现的标识符与类型等属性相关联,从而确定它们在计算机内部的表示法、取值范围及可对其进行的运算。
- 为了产生有效地可执行目标代码，对于说明部分的翻译，不仅仅把与标识符相关联的类型等属性填入符号表中，还必须考虑到标识符所标记的对象的存储分配问题。

常量定义的翻译

- C++语言中有常量定义，以关键字const为标志，如

  const float pi=3.1416， one=1.0

- 对每个常量定义的处理应包括下列工作：

  （1）把等号右边的常量值登录入常量表中（若之前尚未登录）并回送常量表序号；

  （2）然后为等号左边的标识符建立符号表新条目，在该条目中填入常量标志、相应类型及常量表序号。

- 假定常量定义中不指明类型，类型直接由常量值本身确定，且等号右边仅整数或常量标识符。
- 常量定义及相应翻译方案

CONSTEDF→const CDT

CDT →CDT，CD

CDT →CD

CD →id=num

{ num.ord:=lookCT(num.lexval)
id.ord:=num.ord; id.type:=integer;
id.kind:=CONSTANT
add(id.entry, id.kind, id.type, id.ord)}

CD → id=id_1

{id.kind:=CONSTANT; id.type:=id_1.type;
id.ord:=id_1.ord
add(id.entry, id.kind, id.type, id.ord)}

lookCT(c)将在常量表中查找常量c，若查不到，则把给常量值登录入常量表。最终回送常量c值在常量表中的序号

过程add是对过程addtype的扩充，它把种类、类型与序号三者填入符号表相应条目中

变量说明的翻译

- 变量说明部分由一组变量说明语句组成，这里假定每个变量说明语句中仅包含一个标识符。
- 变量说明部分的语法定义

  $P \rightarrow D;$

  $D \rightarrow D;\ D$

  $D \rightarrow id:\ T$

  $T \rightarrow integer$

  $T \rightarrow real$

  $T \rightarrow array[num]\ of\ T_1$

  $T \rightarrow \uparrow T_1$

- **变量说明部分的翻译**

$P \rightarrow D;$ {offset:=0}

$D \rightarrow D; D$

$D \rightarrow$ id: T
{enter(id.name, T.type, offset);
offset:=offset+T.width}

$T \rightarrow$ integer
{T.type:=integer; T.width:=4}

$T \rightarrow$ real
{T.type:=real; T.width:=8}

$T \rightarrow$ array[num] of T_1
{T.type:=array(num.val, T_1.type);
T.width:=num.val*T_1.width}

$T \rightarrow \uparrow T_1$
{T.type=pointer(T_1.type); T.width:=4}

文法转换

- P → D;　{offset:=0}

  在处理第一条说明语句之前，需要对offset赋初值；

  为了使得offset赋初值更明显

  变更为 P → {offset:=0} D;

  也可采用增加非终结符号及产生式来实现

  P → M D

  M → ε {offset:=0}

保留作用域信息

- 过程（函数）定义的语法

  $P \rightarrow D;$

  $D \rightarrow D; D$

  $D \rightarrow id: T$

  $D \rightarrow proc\ id; D; S$

$T \rightarrow integer$
$T \rightarrow real$
$T \rightarrow array[num]\ of\ T_1$
$T \rightarrow \uparrow T_1$

- 过程及函数定义的翻译必然涉及标识符作用域问题
- 基本思想：每个过程有一张独立的符号表

- 过程示例

```
(1) Program sort(input, output)
(2)  var a: array[0..10] of integer;
(3)        x: integer;
(4)        procedure readarray;
(5)             var i: integer;
(6)             begin …a … end {readarray}
(7)        procedure exchange(i, j:integer)
(8)             begin
(9)                  x:=a[i]; a[i]:=a[j]; a[j]:=x
(10)            end {exchange}
(11)       procedure quicksort (m, n:integer);
(12)            var k, v: integer;
(13)            function partition (y,z: integer): integer;
(14)                 var i,j: integer;
(15)                 begin …a …
(16)                           …v …
(17)                           …exchange(i, j); …
(18)                 end {partition}
(19)             begin … end {quicksort};
(20)  begin … end {sort}.
```

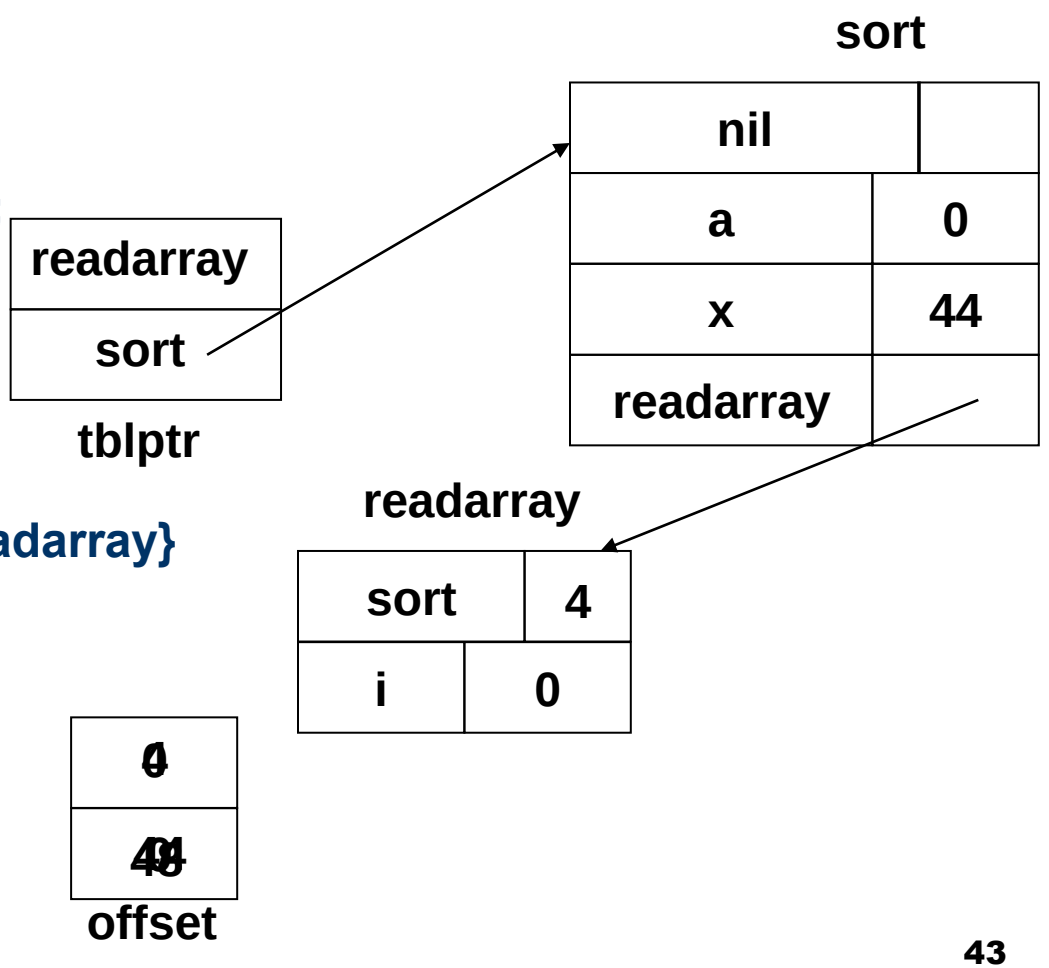
- 语义规则中的操作
  - mktable(previous)：创建一张新符号表，并返回指向新表的一个指针。
    - 参数previous指向一张先前创建的符号表，其值放在新符号表表头
  - enter(table, name, type, offset)：在指针table指示的符号表中为名字name建立一个新项，并把类型type、相对地址offset填入到该项中。
  - addwidth(table, with)：指针table指示的符号表表头中记录下该表中所有名字占用的总宽度
  - enterproc(table, name, newtable)：在指针table指示的符号表中为名字name的过程建立一个新项。
    - 参数newtable指向过程name的符号表

- 翻译

$P \rightarrow M\ D;$	{addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset)}
$M \rightarrow \varepsilon$	{t:=mktable(nil); push(t, tblptr); push(0, offset)}
$D \rightarrow D_1;\ D2$	
$D \rightarrow$ proc id; $N\ D_1;\ S$	{t:=top(tblptr); addwidth(t,top(offset)); pop(tblptr); pop(offset) enterproc(top(tblptr), id.name, t)}
$D \rightarrow$ id: T	{enter(top(tblptr), id.name, T.type, top(offset)); top(offset):=top(offset)+T.width}
$N \rightarrow \varepsilon$	{t:=mktable(top(tblptr)); push(t, tblptr); push(0, offset)}

- 栈tblptr 保存各外层过程的符号表指针
- 栈offset 存放各嵌套过程的当前相对地址
  - 栈顶元素为当前被处理过程的下一个名字的相对地址

```
(1) Program sort(input, output)
(2)  var a: array[0..10] of integer;
(3)        x: integer;
(4)        procedure readarray;
(5)             var i: integer;
(6)             begin …a … end {readarray}
          ……
```



内容线索

- ✓ 中间语言
- ✓ 说明语句的翻译
- ■ 赋值语句的翻译
- ■ 布尔表达式的翻译
- ■ 控制语句的翻译
- ■ 过程调用的处理

赋值语句的翻译

- 简单算术表达式及赋值语句
  - 简单算术表达式及赋值语句翻译为三地址代码的翻译模式
    - 属性id.name 表示id所代表的名字本身
    - 过程lookup(id.name)检查是否在符号表中存在相应此名字的入口。如果有，则返回一个指向该表项的指针，否则，返回nil表示没有找到
    - 过程emit将生成的三地址语句发送到输出文件中

产生赋值语句三地址代码的翻译模式

S→id:=E　S.code:=E.code || gen(id.place ‘:=’ E.place)

$E \rightarrow E_1+E_2$　E.place:=newtemp;
E.code:=E_1.code || E_2.code ||gen(E.place ‘:=’ E_1.place ‘+’ E_2.place)

$E \rightarrow E_1*E_2$　E.place:=newtemp;
E.code:=E_1.code || E_2.code || gen(E.place ‘:=’ E_1.place ‘*’ E_2.place)

S→id:=E　{ p:=lookup(id.name);
if p≠nil then
emit(p ‘:=’ E.place)
else error }

$E \rightarrow E_1+E_2$　{ E.place:=newtemp;
emit(E.place ‘:=’ E_1.place ‘+’ E_2.place)}

$E \rightarrow E_1*E_2$　{ E.place:=newtemp;
emit(E.place ‘:=’ E_1.place ‘*’ E_2.place)}

$E \rightarrow -E_1$	E.place:=newtemp; E.code:=E_1.code || gen(E.place ‘:=’ ‘uminus’ E_1.place)
$E \rightarrow (E_1)$	E.place:=E_1.place; E.code:=E_1.code
$E \rightarrow id$	E.place:=id.place; E.code= ‘ ’

$E \rightarrow -E_1$ { E.place:=newtemp;
emit(E.place‘:=’‘uminus’E_1.place)}

$E \rightarrow (E_1)$ { E.place:=E_1.place}

$E \rightarrow id$ { p:=lookup(id.name);
if p≠nil then
E.place:=p
else error }

类型转换

- 用E.type表示非终结符E的类型属性
- 对应产生式$E \rightarrow E_1$ op E_2的语义动作中关于E.type的语义规则可定义为：

  { if E_1.type=integer and E_2.type=integer
  E.type:=integer
  else E.type:=real }
- 进行类型转换的三地址代码

  x:= inttoreal y
- 算符区分为整型算符int op和实型算符real op，

例. x:=y + i*j

其中x、y为实型；i、j为整型。这个赋值句产生的三地址代码为：

T_1:=i int* j
T_3:=inttoreal T_1
T_2:=y real+ T_3
x:=T_2

关于产生式$E \rightarrow E_1 + E_2$的语义动作

{ E.place:=newtemp;
  if E_1.type=integer and E_2.type=integer then begin
    emit (E.place ‘:=’ E_1.place ‘int+’ E_2.place);
    E.type:=integer
  end
  else if E_1.type=real and E_2.type=real then begin
    emit (E.place ‘:=’ E_1.place ‘real+’ E_2.place);
    E.type:=real
  end

else if E_1.type=integer and E_2.type=real then begin
    u:=newtemp;
    emit (u ‘:=’ ‘inttoreal’ E_1.place);
    emit (E.place ‘:=’ u ‘real+’ E_2.palce);
    E.type:=real
end
else if E_1.type=real and E_2.type=integer then begin
    u:=newtemp;
    emit (u ‘:=’ ‘inttoreal’ E_2.place);
    emit (E.place ‘:=’ E_1.place ‘real+’ u);
    E.type:=real
end
 else E.type:=type_error}

内容线索

- ✓ 中间语言
- ✓ 说明语句的翻译
- ✓ 赋值语句的翻译
- ■ 布尔表达式的翻译
- ■ 控制语句的翻译
- ■ 过程调用的处理

布尔表达式的翻译

- **布尔表达式：用布尔运算符把布尔量、关系表达式联结起来的式子。**
  - 布尔运算符：and, or, not；
  - 关系运算符 < ,≤, = , ≠, > ,≥
- **布尔表达式的两个基本作用:**
  - 用于逻辑演算,计算逻辑值;
  - 用于控制语句的条件式.
- **产生布尔表达式的文法:**
  - E→E or E | E andE | ¬ E | (E) | id rop id | id
- **运算符优先级：布尔运算由高到低:not and or，同级左结合**
  关系运算符同级，且高于布尔运算符

- 计算布尔表达式通常采用两种方法:

(1) 如同计算算术表达式一样,一步步算

1 or (not 0 and 0) or 0

=1 or (1 and 0) or 0

=1 or 0 or 0

=1 or 0

=1

(2) 采用优化措施

把A or B解释成　　if A then true else B

把A and B解释成　　if A then B else false

把¬ A解释成　　if A then false else true

对应的，有两种布尔表达式翻译方法

数值表示法

- a or b and not c 翻译成

  T_1:=not c

  T_2:=b and T_1

  T_3:=a or T_2

- a<b的关系表达式可等价地写成

  if a<b then 1 else 0， 翻译成

```
100:    if a<b goto 103
101:    T:=0
102:    goto 104
103:    T:=1
104:
```

数值表示法的翻译模式

- 过程emit将三地址代码送到输出文件中
- nextstat：给出输出序列中下一条三地址语句的地址索引
- 每产生一条三地址语句后，过程emit便把nextstat加1

数值表示法的翻译模式

$E \rightarrow E_1$ or E_2 {E.place:=newtemp;
emit(E.place ‘:=’ E_1.place ‘or’ E_2.place)}

$E \rightarrow E_1$ and E_2 {E.place:=newtemp;
emit(E.place ‘:=’ E_1.place ‘and’ E_2.place)}

$E \rightarrow$ not E_1 {E.place:=newtemp;
emit(E.place ‘:=’ ‘not’ E_1.place)}

$E \rightarrow (E_1)$ {E.place:=E_1.place}

数值表示法的翻译模式

a<b 翻译成

```
100:  if a<b goto 103
101:  T:=0
102:  goto 104
103:  T:=1
104:
```

$E \rightarrow id_1$ relop id_2 { E.place:=newtemp;
emit(‘if’ id_1.place relop. op
id_2. place ‘goto’ nextstat+3);
emit(E.place ‘:=’ ‘0’);
emit(‘goto’ nextstat+2);
emit(E.place ‘:=’ ‘1’) }

$E \rightarrow id$ { E.place:=id.place }

布尔表达式a<b or c<d and e<f的翻译结果

100: if a<b goto 103
101: T_1:=0
102: goto 104
103: T_1:=1
104: if c<d goto 107
105: T_2:=0
106: goto 108
107: T_2:=1
108: if e<f goto 111
109: T_3:=0
110: goto 112
111: T_3:=1
112: T_4:=T_2 and T_3
113: T_5:=T_1 or T_4

$E \rightarrow id_1$ relop id_2
{ E.place:=newtemp;
emit(‘if’ id_1.place relop. op id_2. place
‘goto’ nextstat+3);
emit(E.place ‘:=’ ‘0’);
emit(‘goto’ nextstat+2);
emit(E.place ‘:=’ ‘1’) }

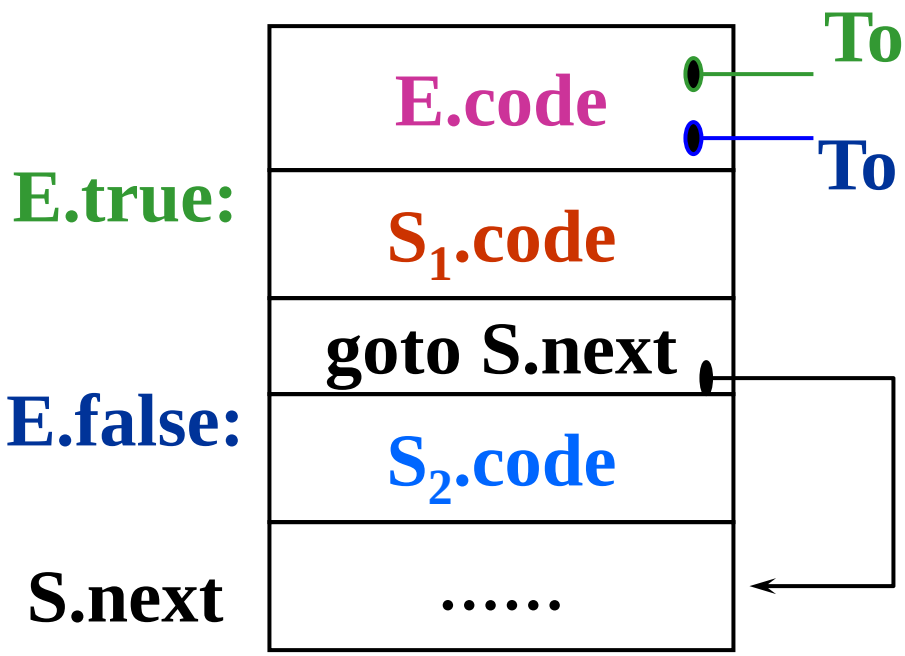
$E \rightarrow id$
{ E.place:=id.place }

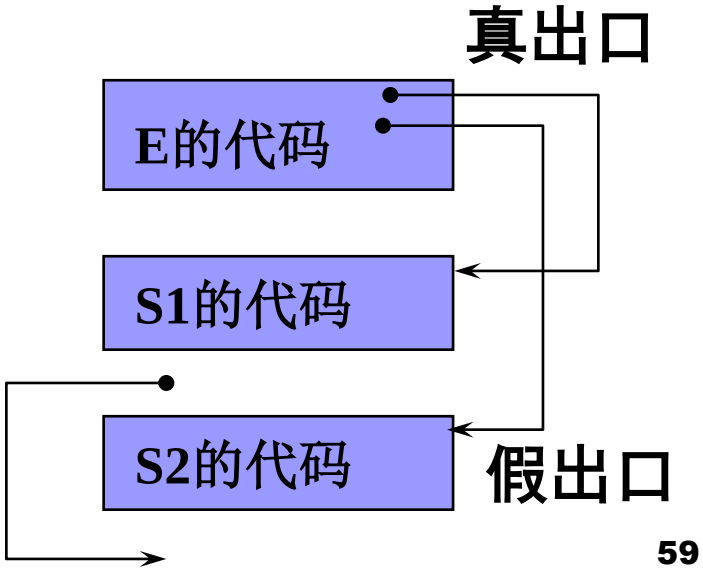
$E \rightarrow E_1$ or E_2
{ E.place:=newtemp;
emit(E.place ‘:=’ E_1.place ‘or’ E_2.place)}

$E \rightarrow E_1$ and E_2
{ E.place:=newtemp;
emit(E.place ‘:=’ E_1.place ‘and’ E_2.place) }

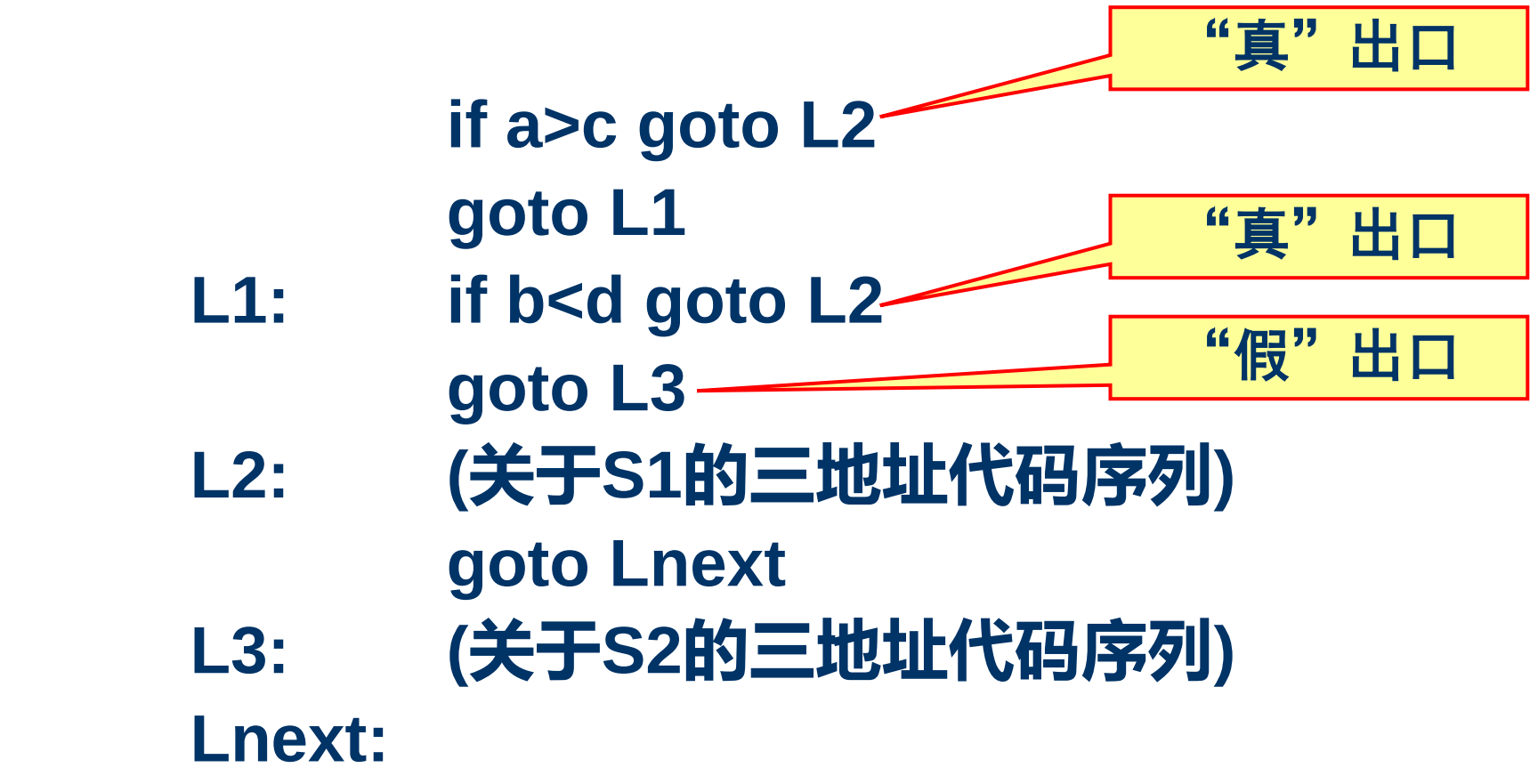
作为条件控制的布尔式翻译

- 条件语句 if E then S_1 else S_2
  赋予 E 两种出口:一真一假





例. 把语句: if a>c or b <d then S_1 else S_2 翻译成如下的一串三地址代码



产生布尔表达式三地址代码的语义规则

产生式

$E \rightarrow E_1$ or E_2

语义规则

每次调用函数newlabel后都返回一个新的符号标号

E_1.true:=E.true;

E_1.false:=newlabel;

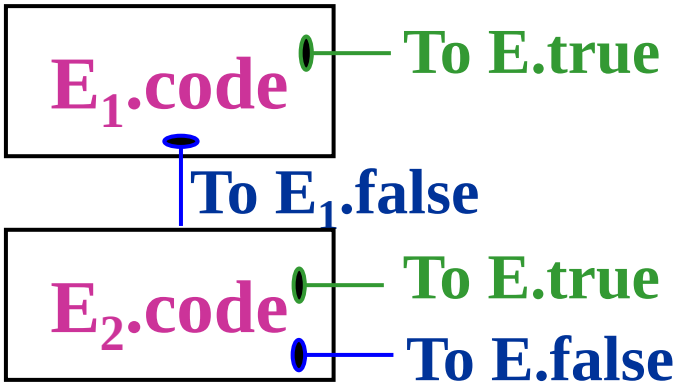
E_2.true:=E.true;

E_2.false:=E.false;

E.code:=E_1.code || gen(E_1.false ‘:’) ||E_2.code

E.true是E为‘真’时控制流转向的标号

E.false是E为‘假’时控制流转向的标号

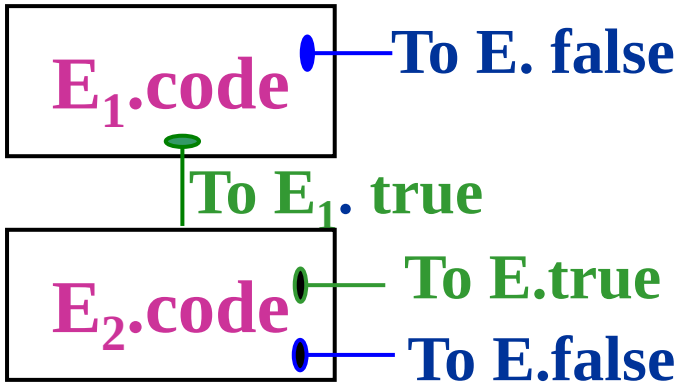


产生布尔表达式三地址代码的语义规则

产生式 $E \rightarrow E_1$ and E_2

语义规则

E_1.true:=newlabel;
E_1.false:=E.false;
E_2.true:=E.true;
E_2.false:=E.fasle;
E.code:=E_1.code || gen(E_1.true ‘:’) || E_2.code



产生布尔表达式三地址代码的语义规则

产生式	语义规则
$E \rightarrow \text{not } E_1$	$E_1.true := E.false$; $E_1.false := E.true$; $E.code := E_1.code$
$E \rightarrow (E_1)$	$E_1.true := E.true$; $E_1.false := E.false$; $E.code := E_1.code$

产生布尔表达式三地址代码的语义规则

产生式	语义规则
$E \rightarrow id_1$ relop id_2	E.code:=gen(‘if’ id_1.place relop.op id_2.place ‘goto’ E.true) || gen(‘goto’ E.false)
E→true	E.code:=gen(‘goto’ E.true)
E→false	E.code:=gen(‘goto’ E.false)

考虑如下表达式：

a<b or c<d and e<f

假定整个表达式的真假出口已分别置为Ltrue和Lfalse，则按定义将生成如下的代码：

```
        if a<b goto Ltrue
        goto L1
L1:     if c<d goto L2
        goto Lfalse
L2:     if e<f goto Ltrue
        goto Lfalse
```

布尔表达式的翻译

- 两遍扫描
  - 为给定的输入串构造一棵语法树；
  - 对语法树进行深度优先遍历，进行语义规则中规定的翻译。
- 一遍扫描

一遍扫描实现布尔表达式的翻译

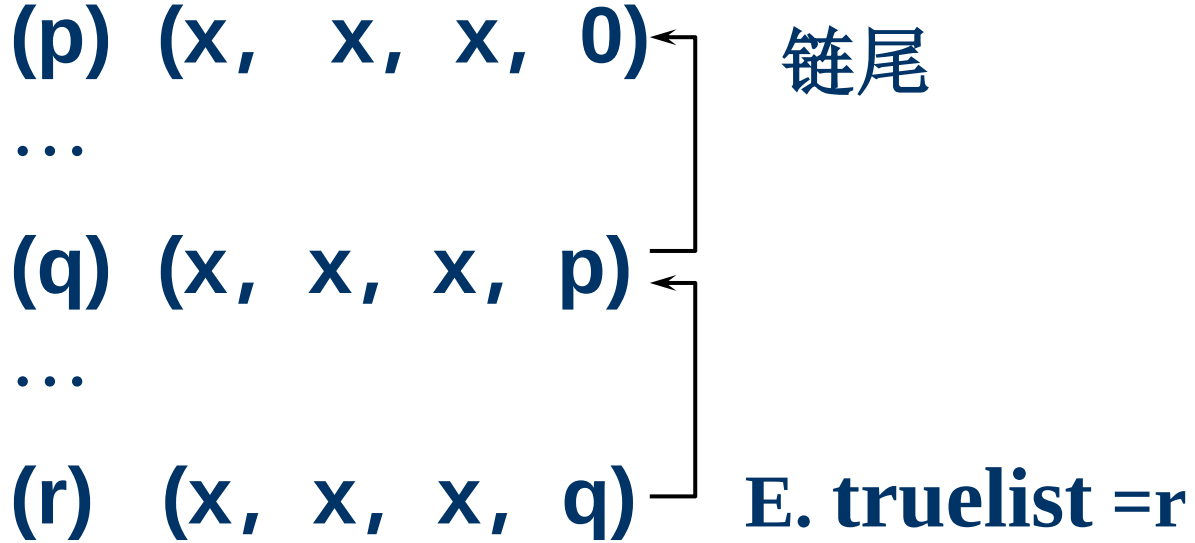
- 采用四元式形式
- 把四元式存入一个数组中，数组下标就代表四元式的标号
- 约定

  四元式(jnz, a, -, p)　表示　if a goto p

  四元式(jrop, x, y, p)表示　if x rop y goto p

  四元式(j, -, -, p)　　表示　goto p
- 有时,四元式转移地址无法立即知道,我们只好把这个未完成的四元式地址作为E的语义值保存,待机"回填"。

- 为非终结符E赋予两个综合属性E.truelist和E.falselist。它们分别记录布尔表达式E所对应的四元式中需回填“真”、“假”出口的四元式的标号所构成的链表
- 例如:假定E的四元式中需要回填"真"出口的p，q，r三个四元式，则E.truelist为下列链:



- 为了处理E.truelist和E.falselist ，引入下列语义变量和过程:
  - 变量nextquad，它指向下一条将要产生但尚未形成的四元式的地址(标号)。nextquad的初值为1，每当执行一次emit之后，nextquad将自动增1。
  - 函数makelist(i)，它将创建一个仅含i的新链表，其中i是四元式数组的一个下标(标号)；函数返回指向这个链的指针。
  - 函数merge(p_1,p_2)，把以p_1和p_2为链首的两条链合并为一，作为函数值，回送合并后的链首。
  - 过程backpatch(p, t)，其功能是完成“回填”，把p所链接的每个四元式的第四区段都填为t。

布尔表达式的文法

(1) $E \rightarrow E_1$ or M E_2

(2) | E_1 and M E_2

(3) | not E_1

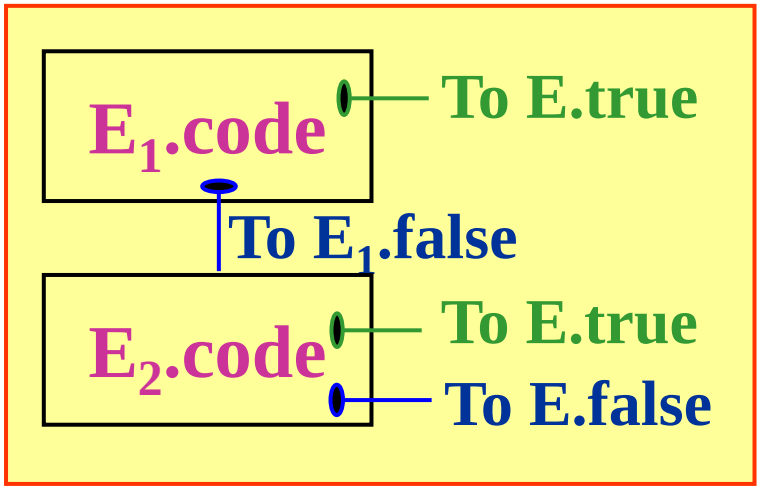
(4) | (E_1)

(5) | id_1 relop id_2

(6) | id

(7) $M \rightarrow \varepsilon$

布尔表达式的翻译模式



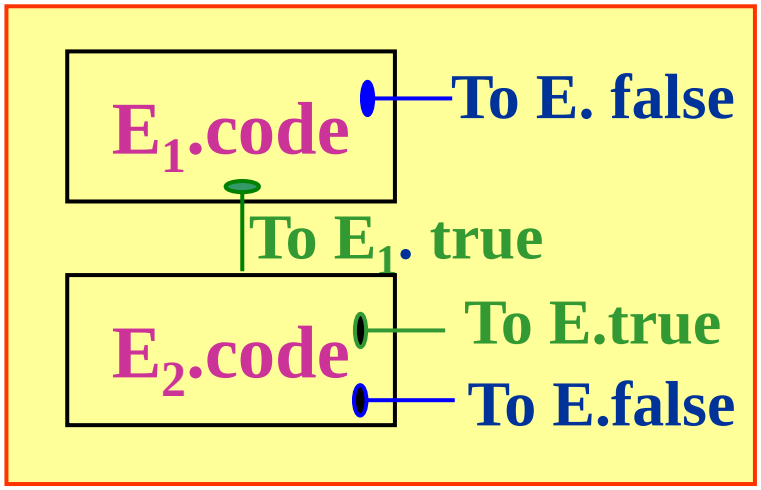
(1) $E \rightarrow E_1$ or M E_2

{ backpatch(E_1.falselist, M.quad);

E.truelist:=merge(E_1.truelist, E_2.truelist);

E.falselist:=E_2.falselist }

布尔表达式的翻译模式



(2) $E \rightarrow E_1$ and M E_2

{ backpatch(E_1.truelist, M.quad);

E.truelist:=E_2.truelist;

E.falselist:=merge(E_1.falselist,E_2.falselist) }

布尔表达式的翻译模式

(3) $E \rightarrow \text{not } E_1$

{ E.truelist:=E_1.falselist;

E.falselist:=E_1.truelist}

(4) $E \rightarrow (E_1)$

{ E.truelist:=E_1.truelist;

E.falselist:=E_1. falselist}

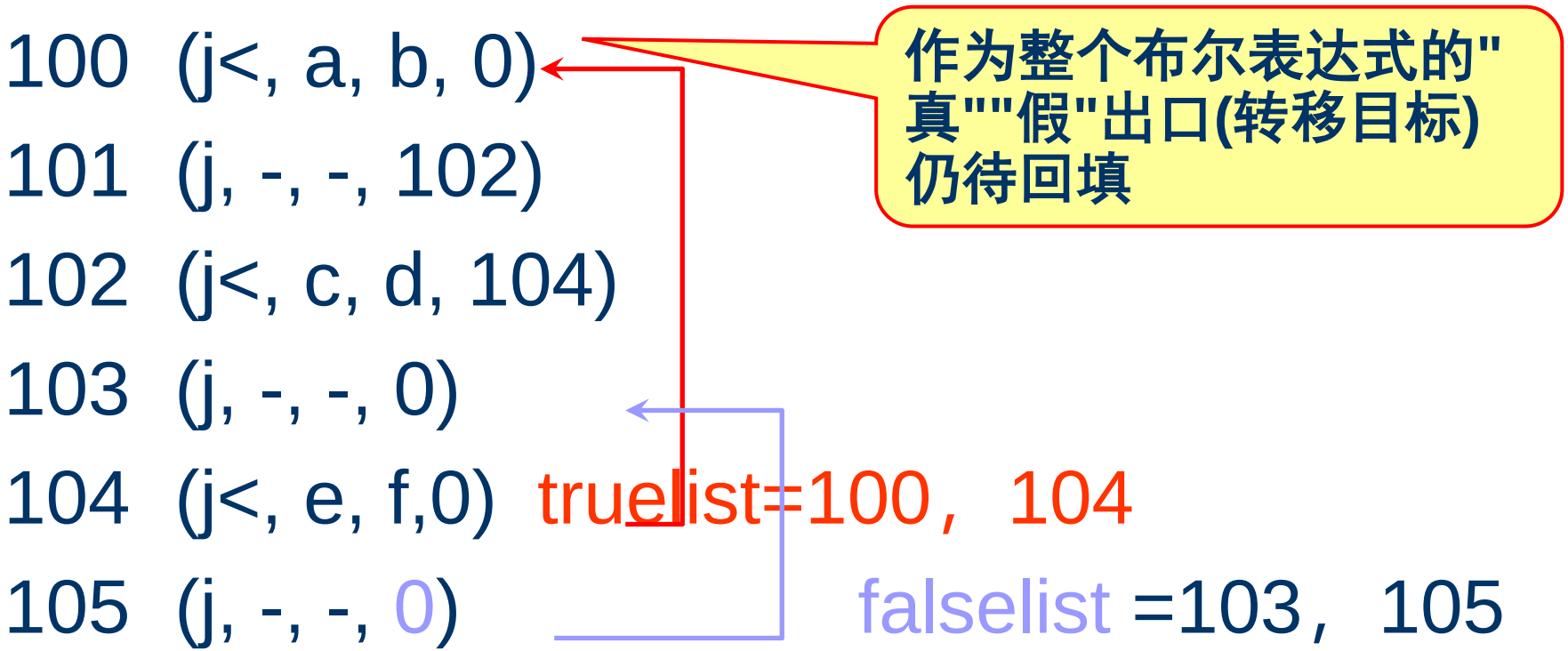
布尔表达式的翻译模式

(5) $E \rightarrow id_1$ relop id_2 { E.truelist:=makelist(nextquad);
E.falselist:=makelist(nextquad+1);
emit(‘j’ relop.op ‘,’ id_1.place ‘,’ id_2.place ‘,’ ‘0’);
emit(‘j, - , - , 0’) }

(6) E→id
{ E.truelist:=makelist(nextquad);
E.falselist:=makelist(nextquad+1);
emit(‘jnz’ ‘,’ id .place ‘,’ ‘ - ’ ‘,’ ‘0’);
emit(‘ j, -, -, 0’) }

(7) M→ε { M.quad:=nextquad }

a<b or c<d and e<f



内容线索

- ✓ 中间语言
- ✓ 说明语句的翻译
- ✓ 赋值语句的翻译
- ✓ 布尔表达式的翻译
- ■ 控制语句的翻译
- ■ 过程调用的处理

控制语句的翻译

- 考虑下列产生式所定义的语句

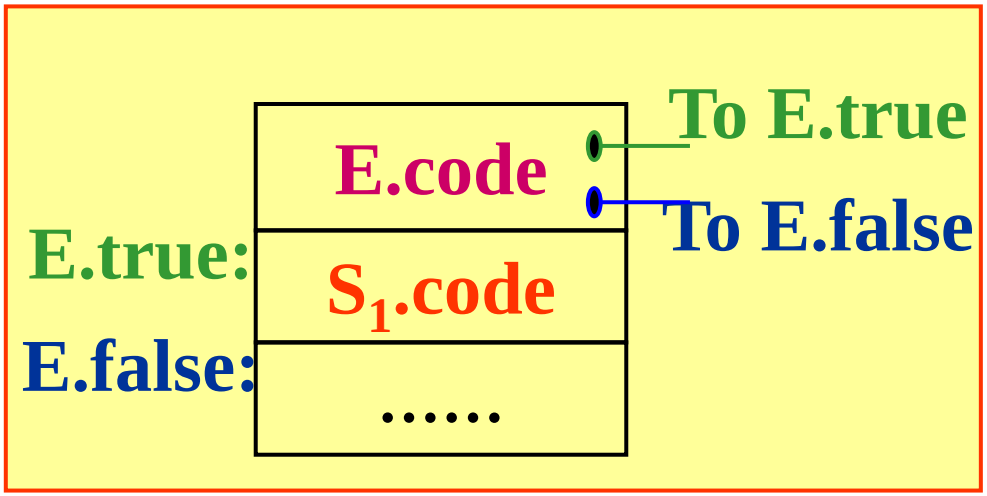
$S \rightarrow$ if E then S_1

| if E then S_1 else S_2

| while E do S_1

其中E为布尔表达式。

if-then语句的属性文法



产生式	语义规则
$S \rightarrow$ if E then S_1	E.true:=newlabel; E.false:=S.next; S_1.next:=S.next S.code:=E.code || gen(E.true ‘:’) || S_1.code

S.next之值是一个标号，它指出继S的代码之后将被执行的第一条三地址指令

if-then-else语句的属性文法

产生式

$S \rightarrow$ if E then S_1 else S_2

语义规则

E.true:=newlabel;

E.false:=newlabel;

S_1.next:=S.next

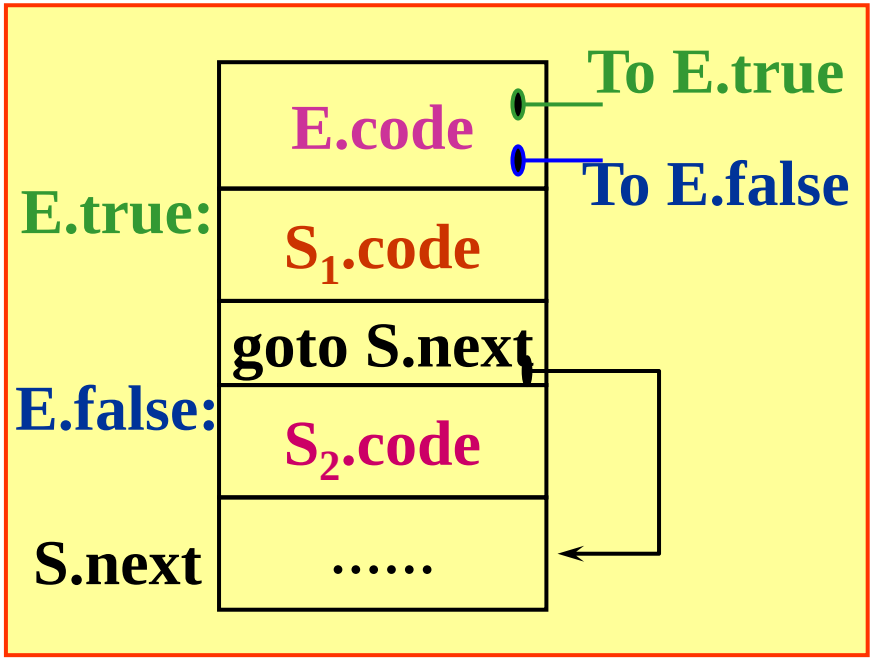
S_2.next:=S.next;

S.code:=E.code ||

gen(E.true ‘:’) || S_1.code

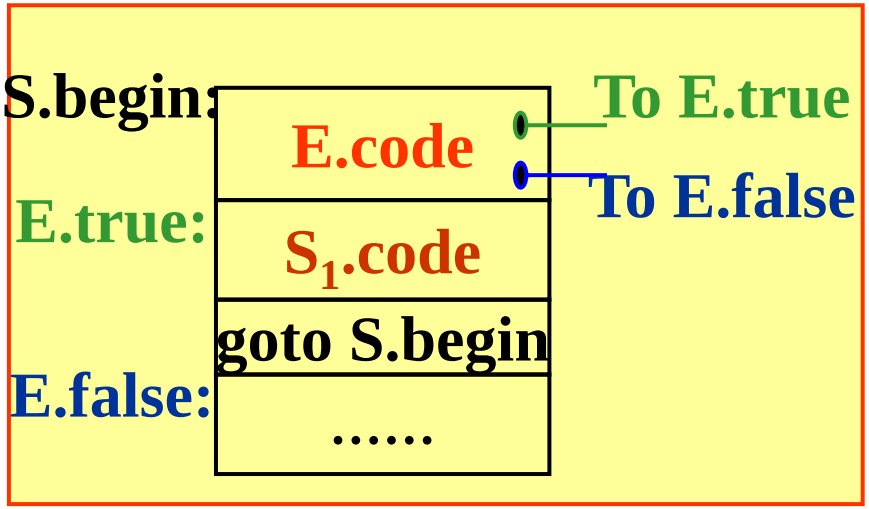
gen(‘goto’ S.next) ||

gen(E.false ‘:’) || S_2.code



while-do语句的属性文法

产生式	语义规则
$S \rightarrow$ while E do S_1	S.begin:=newlabel; E.true:=newlabel; E.false:=S.next; S_1.next:=S.begin; S.code:=gen(S.begin ‘:’) ‖ E.code ‖ gen(E.true ‘:’) ‖ S_1.code ‖ gen(‘goto’ S.begin)



考虑如下语句：

```
while a<b do
        if c<d then  x:=y+z
              else   x:=y-z
```

- 生成下列代码：

L_1: if a<b goto L_2
goto Lnext
L_2: if c<d goto L_3
goto L_4
L_3: T_1:=y+z
x:=T_1
goto L_1
L_4: T_2:=y-z
x:=T_2
goto L_1
Lnext:

E→id1 relop id2
E.code:=gen('if ' id1.place relop.op id2.place 'goto' E.true) ||gen('goto' E.false)

一遍扫描翻译控制流语句

- 考虑下列产生式所定义的语句:

  (1) S→if E then S

  (2) | if E then S else S

  (3) | while E do S

  (4) | begin L end

  (5) | A

  (6) L→L;S

  (7) | S

- S表示语句，　L表示语句表，

  A为赋值语句，E为一个布尔表达式

if 语句的翻译

相关产生式

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

| if E then $S^{(1)}$ else $S^{(2)}$

改写后产生式

$S \rightarrow$ if E then M S_1

$S \rightarrow$ if E then M_1 S_1 N else M_2 S_2

$M \rightarrow \varepsilon$

$N \rightarrow \varepsilon$

翻译模式:

1. $S \rightarrow$ if E then M S_1
 { backpatch(E.truelist, M.quad);
 S.nextlist:=merge(E.falselist, S_1.nextlist) }

2. $S \rightarrow$ if E then M_1 S_1 N else M_2 S_2
 { backpatch(E.truelist, M_1.quad);
 backpatch(E.falselist, M_2.quad);
 S.nextlist:=merge(S_1.nextlist, N.nextlist, S_2.nextlist) }

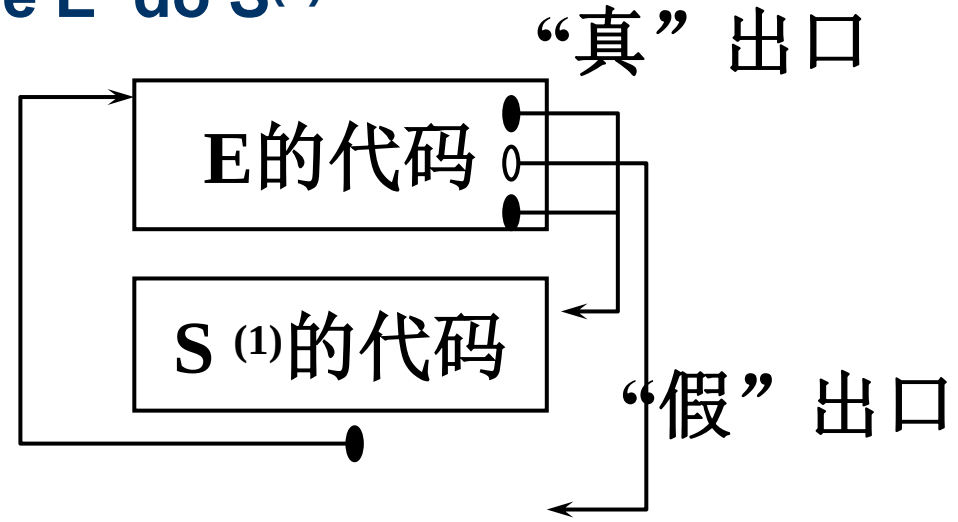
3. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

4. $N \rightarrow \varepsilon$ { N.nextlist:=makelist(nextquad);
 emit(‘j, - , - , - ’) }

while 语句的翻译

相关产生式

S→ while E do S(1)



为了便于"回填"，改写产生式为:

S→while M1 E do M2 S1

M→ε

翻译模式:

1. $S \rightarrow$ while M_1 E do M_2 S_1

{backpatch(S_1.nextlist, M_1.quad);

backpatch(E.truelist, M_2.quad);

S.nextlist:=E.falselist

emit(‘j, - , - ,’ M_1.quad) }

2. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

语句L→L;S的翻译

产生式

$L \rightarrow L;S$

改写为:

$L \rightarrow L_1; M\ S$

$M \rightarrow \varepsilon$

翻译模式:

1. $L \rightarrow L_1; M\ S$ { backpatch(L_1.nextlist, M.quad); L.nextlist:=S.nextlist }
2. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

其它几个语句的翻译

S→begin L end
{ S.nextlist:=L.nextlist }

S→A
{ S.nextlist:=makelist( ) }

L→S
{ L.nextlist:=S.nextlist }

翻译语句

while (a<b) do
if (c<d) then x:=y+z;

P195

S→if E then M S_1
{ backpatch(E.truelist, M.quad);
S.nextlist:=merge(E.falselist, S_1.nextlist) }

M→ε { M.quad:=nextquad }

S→A { S.nextlist:=makelist( ) }

S→id:=E { p:=lookup(id.nam
if p≠nil then
emit(p ':=
else error }

E→E_1+E_2 { E.place:=newtemp
emit(E.place '

extquad);

id $_2$.place ',' '0');

P195

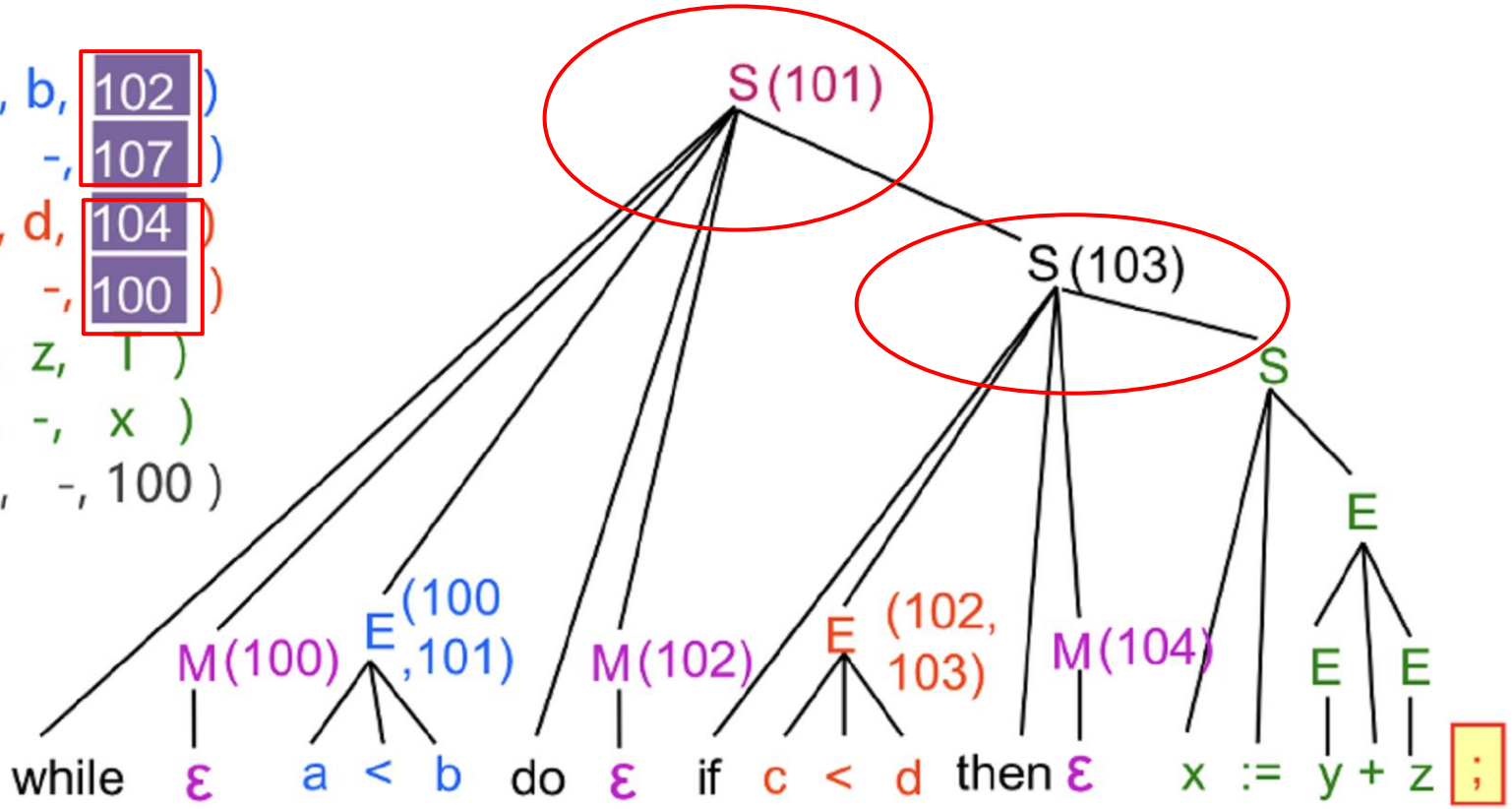
S→while M_1 E do M_2 S_1
{ backpatch(S_1.nextlist, M_1.quad);
backpatch(E.truelist, M_2.quad);
S.nextlist:=E.falselist
emit('j, - , - ,' M_1.quad) }

M→ε { M.quad:=nextquad }

翻译语句

while (a<b) do
if (c<d) then x:=y+z;

100 (j<, a, b, 102)
101 (j, -, -, 107)
102 (j<, c, d, 104)
103 (j, -, -, 100)
104 (+, y, z, T)
105 (:=, T, -, x)
106 (j, -, -, 100)
107



标号与goto语句

- 标号定义形式

L: S;

当这种语句被处理之后，标号L称为“定义了”的。即在符号表中，标号L的“地址”栏将登记上语句S的第一个四元式的地址。

- 标号引用

goto L;

向后转移：

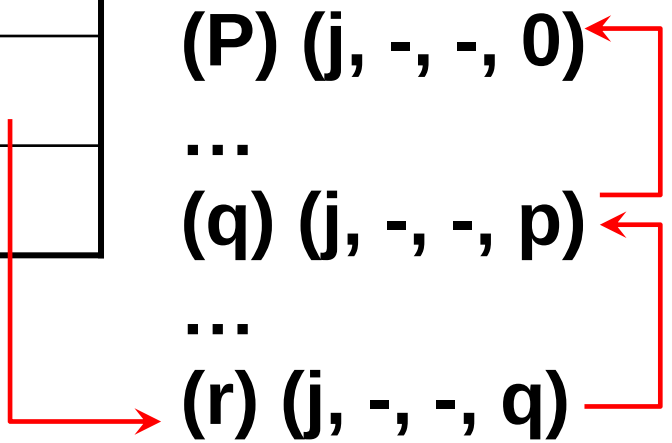
```
L1:   ……
      ……
      goto L1;
```

向前转移：

```
      goto L1;
      ……
L1:   ……
```

符号表信息

名字	类型	…	定义否	地址
…	…	…	…	…
L	标号		未	r



产生式S′→goto L的语义动作:

```
{  查找符号表;
   IF  L在符号表中且"定义否"栏为"已"
     THEN GEN(J，-，-，P)
     ELSE IF  L不在符号表中
          THEN BEGIN
                  把L填入表中;
                  置"定义否"为"未"，"地址"栏为NXQ;
                  GEN(J，-，-，0)   END
          ELSE BEGIN
                  Q:=L的地址栏中的编号;
                  置地址栏编号为NXQ;
                  GEN(J，-，-，Q)
          END
}
```

- 带标号语句的产生式:

  S→label S　　label → i:

- label → i: 对应的语义动作:

1. 若i所指的标识符(假定为L)不在符号表中，则把它填入，置"类型"为"标号"，定义否为"已"，"地址"为nextquad ；
2. 若L已在符号表中但"类型"不为标号或"定义否"为"已"，则报告出错;
3. 若L已在符号表中，则把标号"未"改为"已"，然后，把地址栏中的链头(记为q)取出，同时把nextquad填在其中，最后，执行BACKPATCH(q，nextquad)。

CASE语句的翻译

- 语句结构

case E of

C_1: S_1;

C_2: S_2;

…

C_{n-1}: S_{n-1};

otherwise: S_n

end

E是一个表达式，称为选择子。E通常是一个整型表达式或字符型变量。

- 翻译法(一):

T:=E

L_1: if T≠C_1 goto L_2

S_1的代码

goto next

L_2: if T≠C_2 goto L_3

S_2的代码

goto next

L_3:

…

L_{n-1}: if T≠C_{n-1} goto L_n

S_{n-1}的代码

goto next

L_n: S_n的代码

next:

- 改进:

C_1	S_1 的地址
C_2	S_2 的地址
⋮	⋮
E	S_n 的地址

- 翻译法(二):

计算E并放入T中
goto test
L_1: 关于S_1的中间码
goto next
…
L_{n-1}: 关于S_{n-1}的中间码
goto next
L_n: 关于S_n的中间码
goto next
test: if T=C_1 goto L_1
if T=C_2 goto L_2
…
if T=C_{n-1} goto L_{n-1}
goto L_n
next:

L_1	L_1 的四元式首地址
L_2	L_2 的四元式首地址
⋮	⋮
L_{n-1}	L_{n-1} 的四元式首地址
L_n	L_n 的四元式首地址

C_1	P_1
C_2	P_2
⋮	⋮
C_{n-1}	P_{n-1}

P_i是L_i在符号表中的位置

(case, C_1, P_1, -)
(case, C_2, P_2, -)
…
(case, C_{n-1}, P_{n-1}, -)
(case, T, P_n, -)
(label, NEXT, -, -)

内容线索

- ✓ 中间语言
- ✓ 说明语句的翻译
- ✓ 赋值语句的翻译
- ✓ 布尔表达式的翻译
- ✓ 控制语句的翻译
- ■ 过程调用的处理

过程调用的处理

- 过程调用主要对应两种事:
  - 传递参数
  - 转子（过程）
- 传地址:把实在参数的地址传递给相应的形式参数
  - 调用段预先把实在参数的地址传递到被调用段可以拿到的地方;
  - 程序控制转入被调用段之后，被调用段首先把实在参数的地址抄进自己相应的形式单元中;
  - 过程体对形式参数的引用与赋值被处理成对形式单元的间接访问。

过程调用的文法

- 过程调用文法:
  - (1) S → call id (Elist)
  - (2) Elist → Elist, E
  - (3) Elist → E
- 参数的地址存放在一个队列中
- 最后对队列中的每一项生成一条par语句

过程调用的翻译

- 翻译方法：把实参的地址逐一放在转子指令的前面.

例如， CALL S(A，X+Y) 翻译为:

中间代码：计算X+Y，置于T中

```
par A     /*第一个参数的地址*/
par T     /*第二个参数的地址*/
call S      /*转子*/
```

翻译模式

3. Elist→E

{ 初始化queue仅包含E.place }

2. Elist→Elist, E

{ 将E.place加入到queue的队尾 }

1. S→call id (Elist)

{ for 队列queue中的每一项p do

emit(‘param’ p);

emit(‘call’ id.place) }