Outline

- ✧ 运行时存储组织
  - ✓ 活动记录
  - ✓ 静态链*
  - ✓ 动态链
- ✧ 中间代码生成*
  - ✓ 中间代码
  - ✓ 声明语句
  - ✓ 赋值语句
  - ✓ 布尔表达式
  - ✓ 控制语句**
  - ✓ 拉链与代码回填**
- ✧ 代码生成*
- ✧ 代码优化基础

✧ 活动记录

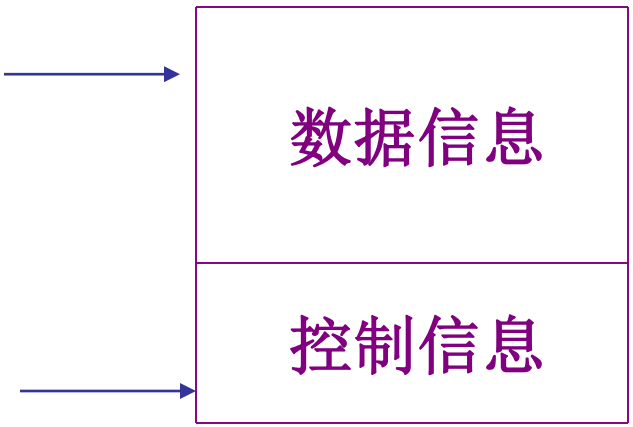
– 过程活动记录

- 函数/过程调用或返回时，在运行栈上创建或从运行栈上消去的栈帧（*frame*）

  包含函数实参、局部变量、临时值（用于表达式计算的中间单元）等数据信息以及必要的控制信息

某个数据对象的地址=
活动记录起始地址
+ 偏移地址(*offset*)

活动记录起始地址
（通常存于某寄存器中）



✧ 活动记录

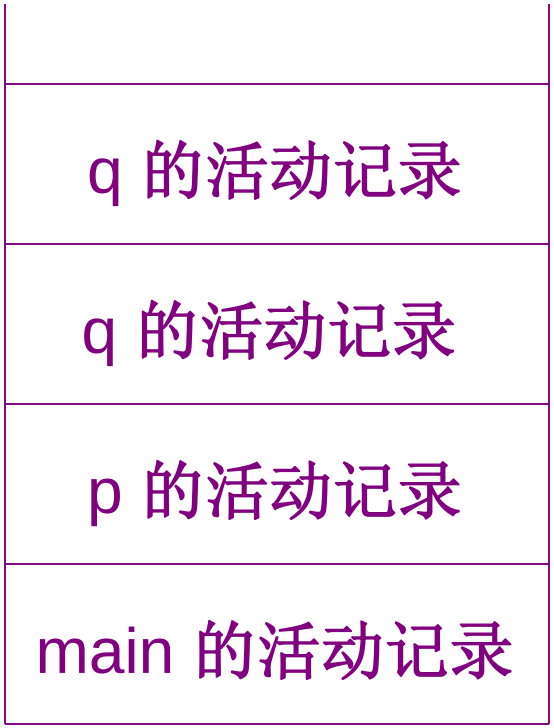
– 过程活动记录的栈式分配举例

```
void p( ) {
   …
   q( );
}

void q( ) {
   …
   q( );
}

int main {
   p( );
}
```

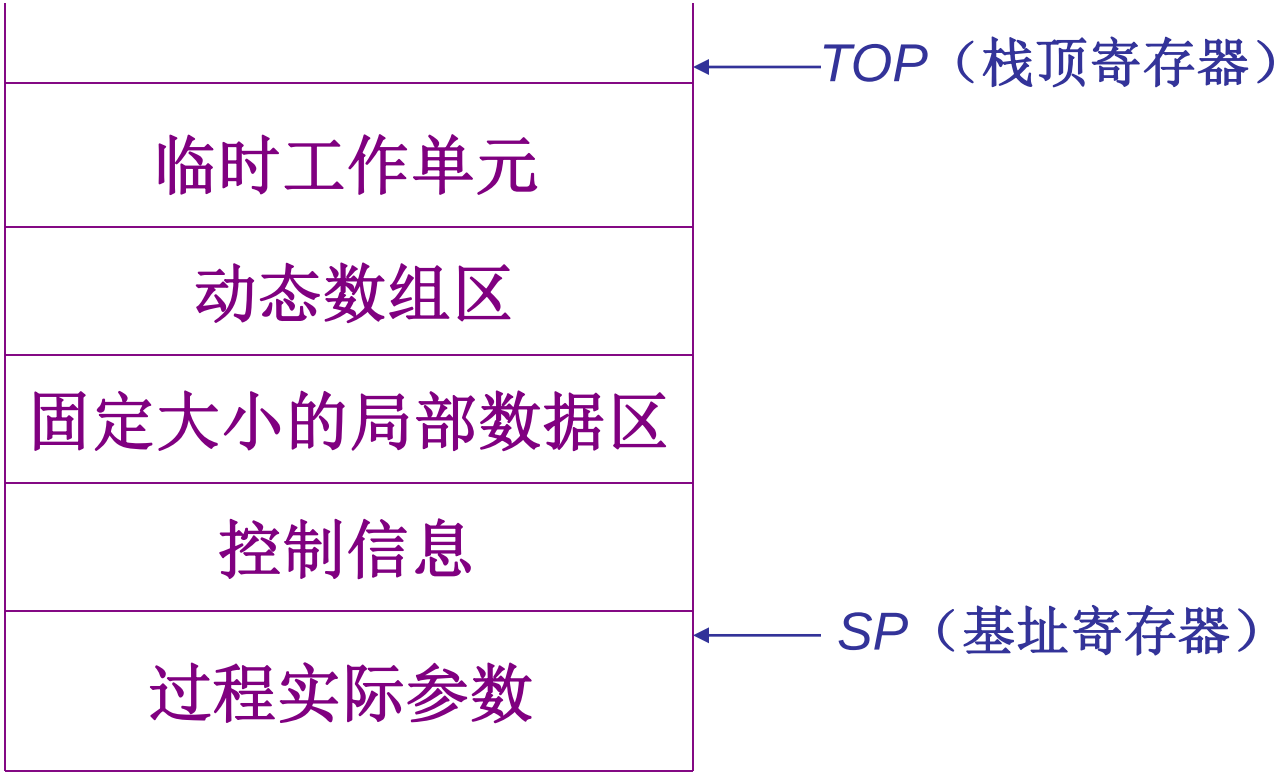
函数 q 被第二次激活时运行栈上活动记录分配情况



运行时存储组织

✧ 活动记录

– 典型的过程活动记录形式



✧ 活动记录

– 嵌套过程语言的栈式分配

- 主要问题

  解决对非局部量的引用（存取）

- 解决方案

  采用 Display 寄存器表

  为活动记录增加静态链域

✧ 活动记录

– 嵌套过程语言的栈式分配

- 采用 Display 寄存器表

  Display 寄存器表（简称 Display 表） 记录各嵌套层当前过程的活动记录在运行栈上的起始位置（基地址）

  当前激活过程的层次为K（主程序的层次设为*0*），则对应的 Display 表含有 K+1 个单元，依次存放着现行层，直接外层...直至最外层的每一过程的最新活动记录的基地址

  嵌套作用域规则确保每一时刻Display 表内容的唯一性

  Display 表的大小（即最多嵌套的层数）取决于实现

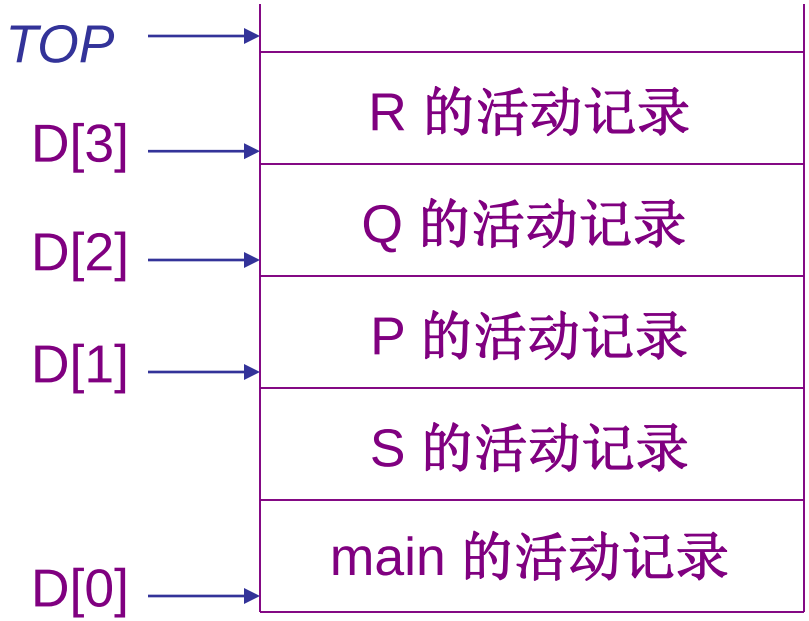
运行时存储组织

✧ 活动记录

– 嵌套过程语言的栈式分配

- Display 表方案举例

过程 R 被第一次激活后运行栈和 Display 寄存器 D[i] 的情况




```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
R     begin
        …    /*here*/
Q     end;   /*R*/
      begin
        … R; …
P     end;   /*Q*/
    begin
      … Q; …
    end;   /*P*/
procedure S;
    begin
S     … P; …
    end;   /*S*/
begin
  …  S; …
end.   /*main*/
```

✧ 活动记录

– 嵌套过程语言的栈式分配

- Display 表的维护（过程被调用和返回时的保存和恢复）

方法一 极端的方法是把整个 Display 表存入活动记录
若过程为第 *n* 层，则需要保存 D[0] ~D[*n*] 。

一个过程被调用时，从调用过程的 Display 表中自下向上抄录 *n* 个 SP 值，再加上本层的 SP 值

方法二 只在活动记录保存一个 Display 表项

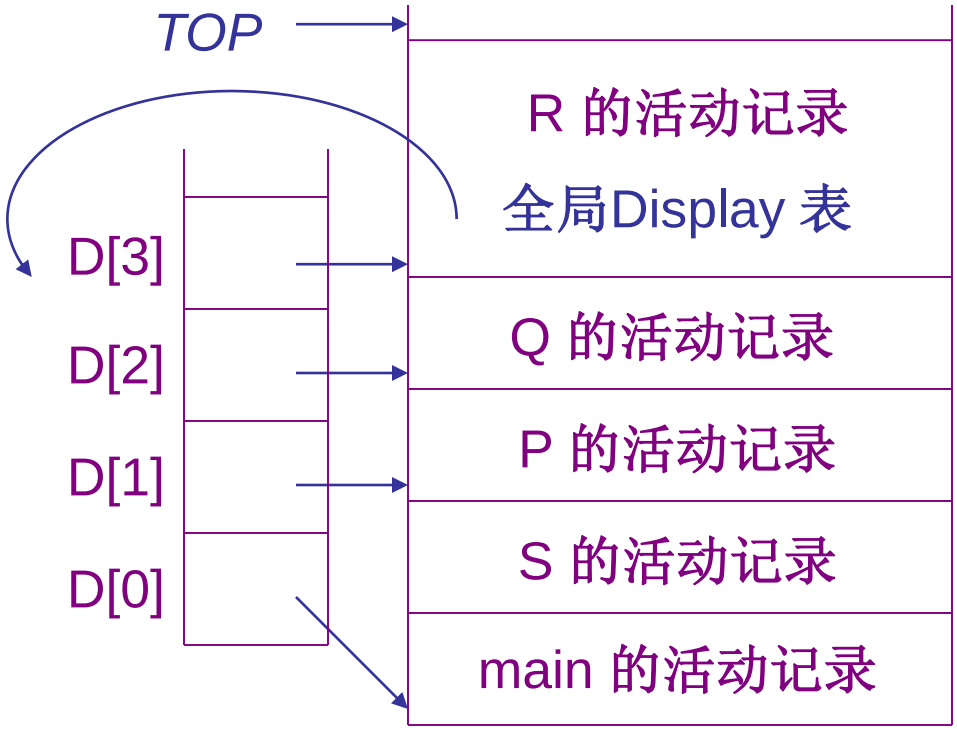
运行时存储组织

✧ 活动记录

– 嵌套过程语言的栈式分配

- Display 表的维护举例

  保存完整的全局Display 表




```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
R     begin
        …   /*here*/
Q     end;  /*R*/
      begin
        … R; …
P     end;  /*Q*/
  begin
    … Q; …
  end;  /*P*/
procedure S;
  begin
S     … P; …
  end;  /*S*/
begin
  …  S; …
end.  /*main*/
```

✧ 活动记录

– 嵌套过程语言的栈式分配

- 采用静态链（*static link*）

  Display 表的方法要用到多个寄存器，有时并不情愿这样做（寄存器资源很宝贵），一种可选的方法是采用静态链，只保留一个寄存器（即SP）指向当前 AR

  所有活动记录都增加一个静态链（如在offset 为 0 处）的域，指向定义该过程的直接外过程（或主程序）运行时最新的活动记录

  在过程返回时当前 AR 要被撤销。为回卷（*unwind*）到调用过程的AR（恢复SP），还需增加一个动态链

  （**静态链/**访问链：SL，**动态链/**控制链：DL ）

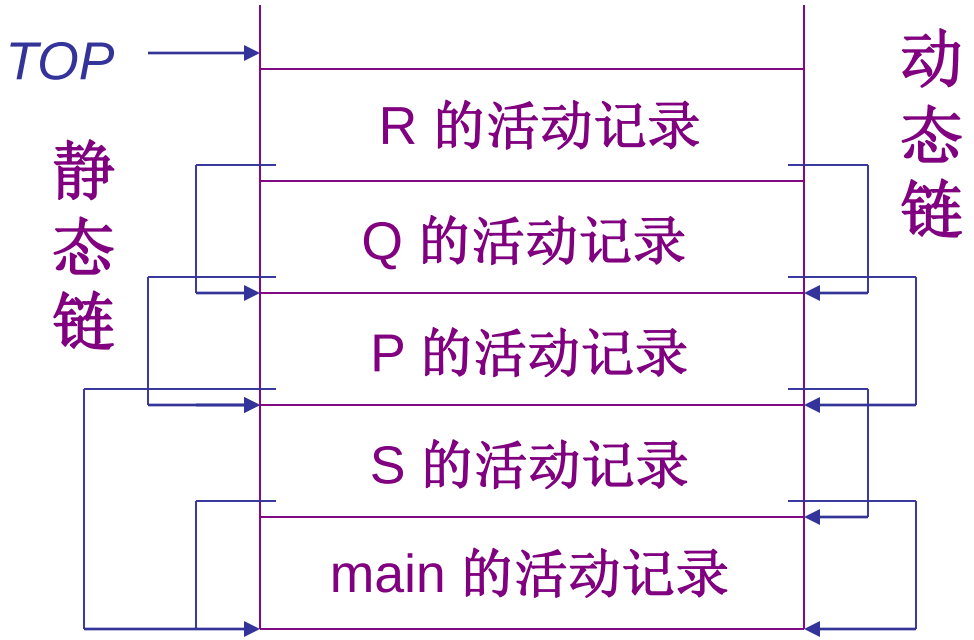
运行时存储组织

✧ 活动记录

– 嵌套过程语言的栈式分配

- 采用静态链的方法举例

第一次运行至/*here*/时的栈状态




```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        …   /*here*/
      end;  /*R*/
    begin
      … R; …
    end;  /*Q*/
  begin
    … Q; …
  end;  /*P*/
procedure S;
  begin
    … P; …
  end;  /*S*/
begin
  …  S; …
end.  /*main*/
```

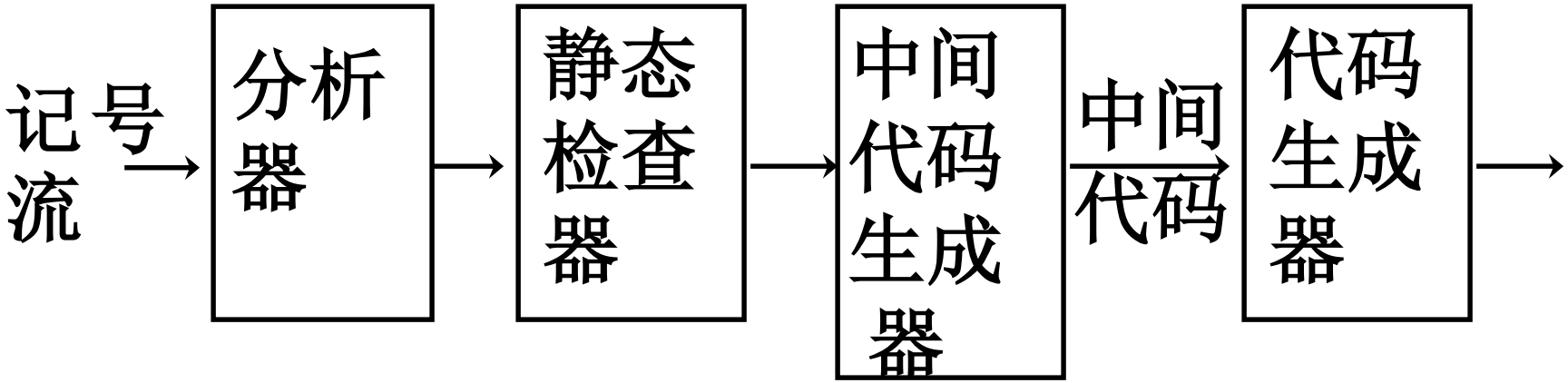
R

Q

P

S

Chapter 7. 中间代码生成及优化



– 用语法制导定义和翻译方案的方法来展示程序设计语言的结构怎样被翻译成中间形式

✧ 中间代码

- 源程序的一种内部表示形式
- 作用
  - 源语言和目标语言之间的桥梁，避开二者之间较大的语义跨度，使编译程序的逻辑结构更加简单明确
  - 利于编译程序的重定向
  - 利于进行与目标机无关的优化

✧ 中间代码的形式

- 有不同层次不同目的之分
- 中间代码举例
  - *AST*（*Abstract syntax tree*，抽象语法树）
  - *DAG*（*Directed Acyclic Graph*, 有向无圈图）
  - *Postfix*（后缀式，逆波兰式）
  - *TAC*（ *Three-address code*, 三地址码，四元式）
  - *P-code* （特别用于 *Pascal* 语言实现）

✧ 三地址码*TAC*

– 顺序的语句序列 其语句一般具有如下形式

$x := y\ op\ z$

(*op* 为操作码，*y* 和 *z* 为操作符， *x* 为结果)

– 表达式 $x + y * z$ 翻译成的三地址语句序列是

$t_1 := y * z$

$t_2 := x + t_1$

✧ 中间代码举例

– 算术表达式 A + B * (C - D) + E / (C - D) ^N （中缀形式）

- *TAC*（ *Three-address code*, 三地址码，四元式）

(1)	(-	C	D	T1)	T1 := C - D
(2)	(*	B	T1	T2)	T2 := B * T1
(3)	(+	A	T2	T3)	T3 := A + T2
(4)	(-	C	D	T4)	或 T4 := C - D
(5)	(^	T4	N	T5)	T5 := T4 ^ N
(6)	(/	E	T5	T6)	T6 := E / T5
(7)	(+	T3	T6	T7)	T7 := T3 + T6

声明语句

- 为局部名字建立符号表条目
- 为它分配存储单元
- 符号表中包含名字的类型和分配给它的存储单元的相对地址等信息

✧ 过程中说明语句的语法制导翻译

– 语义属性

id.*name* : id 的词法名字（符号表中的名字）

T.type : 类型属性

T.width : 数据宽度（字节数）

offset : 相对于过程数据区基址的下一个可用的相对偏移地址（参考运行时存储组织）

– 语义函数/过程

put* (id.*name* ,*T.type, offset*)** : 将符号表中 **id.*name 所对应表项的 ***type*** 域置为 ***T.type,*** 填入其在过程数据区中的相对地址***offset***。

✧ 过程中说明语句的语法制导翻译

– 翻译方案

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

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow \underline{\text{id}}\ :\ T$ { *top.put* (id.*name* ,*T.type*, *offset*);
offset := offset+T.width }

$T \rightarrow$ char { *T.type := char ; T.width := 1* }

$T \rightarrow$ 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* $\times$ T_1.*width* }

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

$S \rightarrow \ldots$

计算被声明名字的类型和相对地址

✧ 赋值语句的语法制导翻译

– 语义属性

id.*name* : id 的词法名字（符号表中的名字）

E.*code* : 求值 *E* 的 *TAC* 语句序列

E.*place* : 用来存放 *E* 的值的名字

– 语义函数/过程

get (id.*name*) : 从符号表中查找名字为 id.*name* 的项，返回存放相应值的指针，若无该项，则返回*nil*

gen : 生成一条 *TAC* 语句

newtempt : 返回一个未使用过的名字

赋值语句

✧ 赋值语句的语法制导定义（三地址码）

产生式	语义规则
$S \rightarrow \underline{\text{id}} := E$	*S.code := E.code || gen (top.get(*$\underline{\text{id}}$*.name) ':=' 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.place := newtemp;* $E.code := E_1.code \,\|\|$
	gen (E.place ':=' 'uminus' E_1*.place)*
$E \rightarrow (E_1)$	$E.place := E_1.place$; $E.code := E_1.code$
$E \rightarrow \underline{\text{id}}$	*E.place := top.get(*$\underline{\text{id}}$*.name);*
	E.code := ''

✧ 符号表中的名字

$S \rightarrow$ id := E {*p* := *top.get*(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*) }

赋值语句

$E \rightarrow -E_1$ {*E.place* := *newtemp*;

emit (*E.place*, ':=', 'uminus', E_1.*place*) }

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

$E \rightarrow$ id {*p* := *top.get*(id.*name*);

if $p \neq nil$ then *E.place* := *p* else *error* }

✧ 数组元素的地址计算

一维数组A的第*i*个元素的地址计算

$$base + (i - low) \times w$$

重写成

$$i \times w + (base - low \times w)$$

减少了运行时的计算

二维数组

- 列为主

  A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

- 行为主

  A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

$base + ((i_1 - low_1) \times n_2 + (i_2 - low_2)) \times w$

（其中$n_2 = high_2 - low_2 + 1$）

$((i_1 \times n_2) + i_2) \times w +$

$(base - ((low_1 \times n_2) + low_2) \times w)$

多维数组

A[i_1, i_2, ..., i_k]的地址表达式

$$((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$$
$$+ base - ((\ldots ((low_1 \times n_2 + low_2) \times n_3 + low_3) \ldots) \times n_k + low_k) \times w$$

– 数组的内情向量（*dope vector*）

在处理数组时，通常会将数组的有关信息记录在一些单元中，称为“内情向量”. 对于静态数组，内情向量可放在符号表中；对于可变数组，运行时建立相应的内情向量

例： 对于静态数组说明 $A[l_1{:}u_1,l_2{:}u_2,\ldots,l_n{:}u_n]$，可以在符号表中建立如下形式的内情向量:

l_1	u_1
l_2	u_2
…	…
l_n	u_n
type	*a*
n	*C*

l_i: 第 *i* 维的下界
u_i: 第 *i* 维的上界
type: 数组元素的类型
a: 数组首元素的地址
n: 数组维数
C: 随后解释

– 数组元素的地址计算

例：对于静态数组 $A[l_1:u_1,l_2:u_2,\ldots,l_n:u_n]$，若数组布局采用行优先的连续布局，数组首元素的地址为 **a**，则数组元素$A[i_1,i_2,\ldots,i_n]$ 的地址 D 可以如下计算:

$$\begin{aligned} D = a &+ (i_1-l_1)(u_2-l_2)(u_3-l_3)\ldots(u_n-l_n) \\ &+ (i_2-l_2)(u_3-l_3)(u_4-l_4)\ldots(u_n-l_n) \\ &+\ldots+ (i_{n-1}-l_{n-1})(u_n-l_n) + (i_n-l_n) \end{aligned}$$

重新整理后得: $D = a - C + V$，其中

$$C = (\ldots(\boldsymbol{l_1}(u_2-l_2)+\boldsymbol{l_2})(u_3-l_3)+\boldsymbol{l_3})(u_4-l_4)+\ldots+\boldsymbol{l_{n-1}})(u_n-l_n)+\boldsymbol{l_n}$$

$$V = (\ldots((\boldsymbol{i_1}(u_2-l_2)+\boldsymbol{i_2})(u_3-l_3)+\boldsymbol{i_3})(u_4-l_4)+\ldots\ldots+\boldsymbol{i_{n-1}})(u_n-l_n)+\boldsymbol{i_n}$$

（这里的 C 即为前页内情向量中的 C）

✧ 类型转换

x := y + i * j
（x和y的类型是real，i和j的类型是integer）

中间代码
t_1 := i int× j
t_2 := inttoreal t_1
t_3 := y real+ t_2
x := t_3

赋值语句

$E \rightarrow E_1 + E_2$

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* = *integer* and E_2.*type* = *real* then begin

    u := *newtemp*;

    emit (*u*, ':=', 'inttoreal', E_1.*place*);

    emit (*E*.place, ':=', *u*, 'real+', E_2.*place*);

    E.type := *real*

end

. . .

✧ 布尔表达式的语法制导翻译

– 直接对布尔表达式求值

例如：可以用数值“1”表示 true; 用数值“0”表示 false; 采用与算术表达式类似的方法对布尔表达式进行求值

– 通过控制流体现布尔表达式的语义

方法：通过转移到程序中的某个位置来表示布尔表达式的求值结果

优点：方便实现控制流语句中布尔表达式的翻译

常可以得到短路（*short-circuit*）代码，而避免不必要的求值

E_1 or E_2 定义成 **if E_1 then true else E_2**

E_1 and E_2 定义成 **if E_1 then E_2 else false**

✧ 布尔表达式的翻译

$E \rightarrow E$ or E | E and E | not E | (E)
| id relop id | true | false

a < b的翻译
100: if a < b goto 103
101: t := 0
102: goto 104
103: t := 1
104:

✧ 布尔表达式的翻译方案

– 直接对布尔表达式求值

nextstat 返回输出代码序列中下一条 *TAC* 语句的下标

$E \rightarrow E_1$ 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* }
$E \rightarrow \underline{id_1}$ relop $\underline{id_2}$	{ *E.place := newtemp;* *emit (* 'if' $\underline{id_1}$.place relop.op $\underline{id_2}$.place 'goto' *nextstat+3)*; *emit (E.place ':=' '0')*; *emit (*'goto' *nextstat+2)*; *emit (E.place ':=' '1')* }
$E \rightarrow$ true	{ *E.place := newtemp; emit(E.place ':=' '1')* }
$E \rightarrow$ false	{ *E.place := newtemp; emit(E.place ':=' '0')* }

✧ 布尔表达式的控制流翻译

如果*E*是*a* < *b*的形式，
那么代码是：
if *a* < *b* goto *E.true*
goto *E.false*

✧ 布尔表达式的语法制导翻译

– 通过控制流体现布尔表达式的语义

例：布尔表达式 *E* = *a*<*b* **or** *c*<*d* **and** *e*<*f* 可能翻译为如下*TAC* 语句序列（采用短路代码，*E.true* 和*E.false* 分别代表 *E* 为真和假时对应于程序中的位置，可用标号体现）：

（1）if *a*<*b* goto *E.true*
（2）goto (3)
（3）if *c*<*d* goto (5)
（4）goto *E.false*
（5）if *e*<*f* goto *E.true*
（6）goto *E.false*

✧ 布尔表达式的语法制导定义

– 三地址码形式（短路代码）

$E \rightarrow E_1$ or E_2
{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* }

$E \rightarrow$ not E_1
{E_1.*true* := *E.false*;
E_1.*false* := *E.true*;
E.code := E_1.*code* }

布尔表达式和控制流语句

$E \rightarrow E_1$ and E_2
{E_1.*true* := *newlabel*;
E_1.*false* := *E.false*;
E_2.*true* := *E.true*;
E_2.*false* := *E.false*;
E.code := E_1.*code* || *gen*(E_1.*true*, ':') || E_2.*code* }

$E \rightarrow (E_1)$
{E_1.*true* := *E.true*;
E_1.*false* := *E.false*;
E.code := E_1.*code* }

布尔表达式和控制流语句

$E \rightarrow \mathrm{id}_1\ \mathrm{relop}\ \mathrm{id}_2$
{*E.code* := *gen*('if', id_1.*place*, relop.*op*, id_2.*place*,
'goto', *E.true*) ||
gen('goto', *E.false*) }

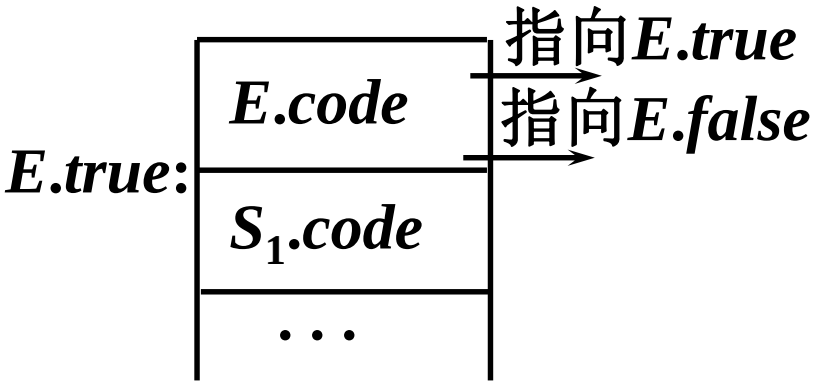
$E \rightarrow$ true
{*E.code* := *gen*('goto', *E.true*)}

$E \rightarrow$ false
{*E.code* := *gen*('goto', *E.false*)}

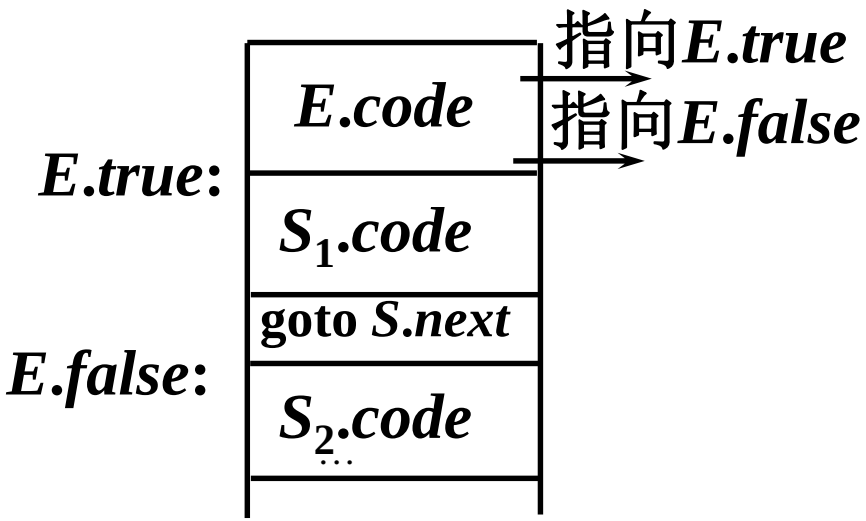
✧ 控制流语句的翻译

$S \rightarrow$ if E then S_1
| if E then S_1 else S_2
| while E do S_1
| S_1; S_2

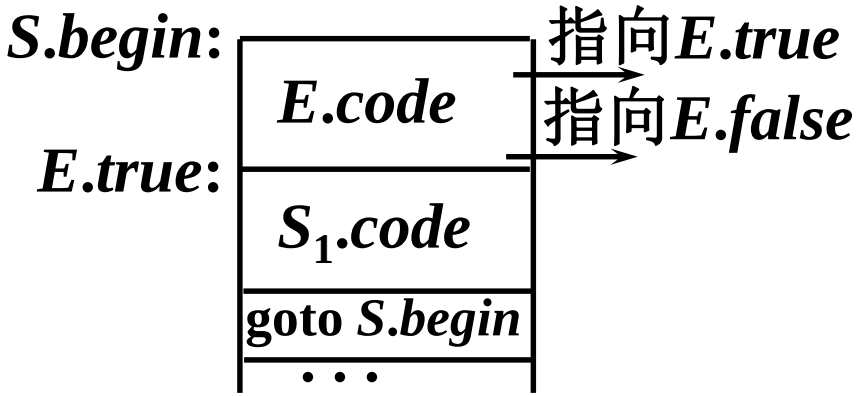
布尔表达式和控制流语句



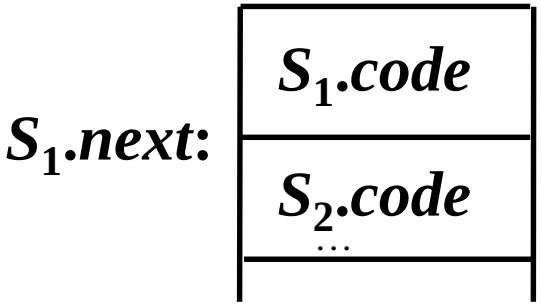
(a) if-then



(b) if-then-else



(c) while-do



(d) S_1; S_2

✧ 条件语句的语法制导定义

– if-then 语句

$S \rightarrow$ if E then S_1 {

E.true := newlabel;

E.false := S.next;

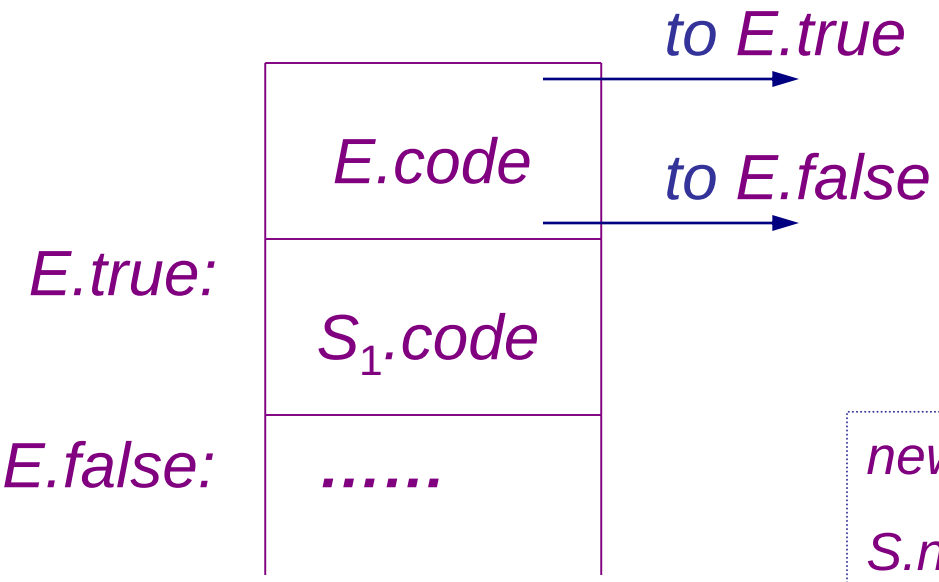
S_1*.next := S.next;*

S.code := E.code ||

gen(E.true ':') ||

S_1*.code*

}



newlabel 返回一个新的语句标号

S.next 属性表示 *S* 之后要执行的首条 *TAC* 语句的标号（这里，未体现对*S.next* 的初始化）

✧ 条件语句的语法制导定义

– if-then-else 语句

$S \rightarrow$ if E then S_1 else S_2 {

E.true := newlabel;

E.false := newlabel;

S_1*.next := S.next;*

S_2*.next := S.next;*

S.code := E.code ||

gen(E.true ':') ||

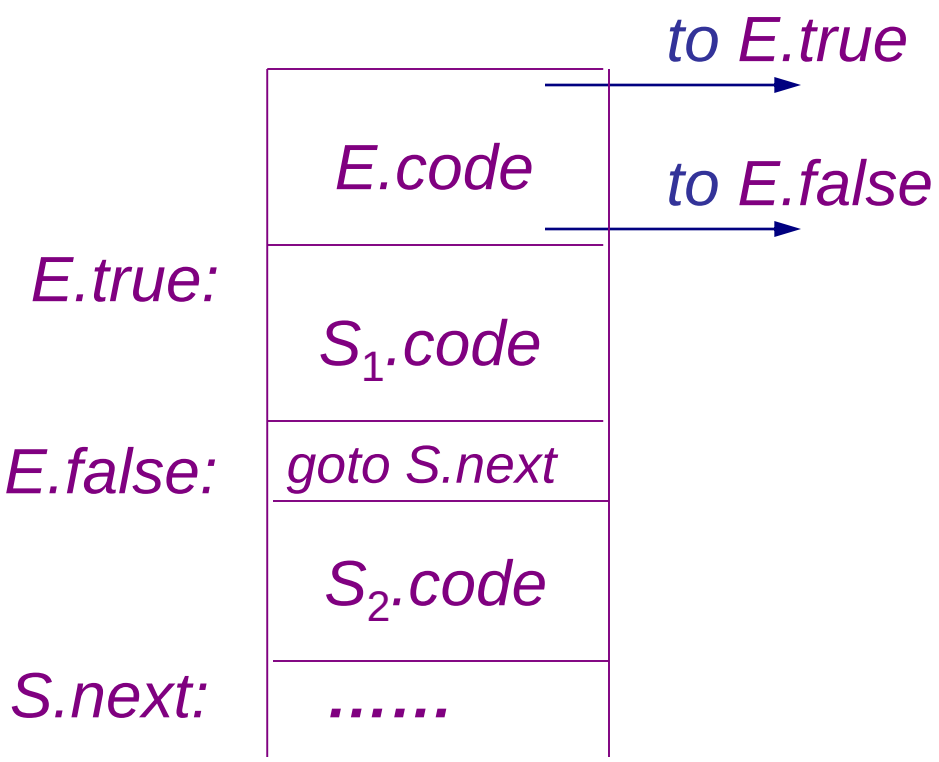
S_1*.code* ||

gen('goto' S.next) ||

gen(E.false ':') ||

S_2*.code*

}



✧ 循环语句的语法制导定义

– while 语句

$S \rightarrow$ 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 ':')

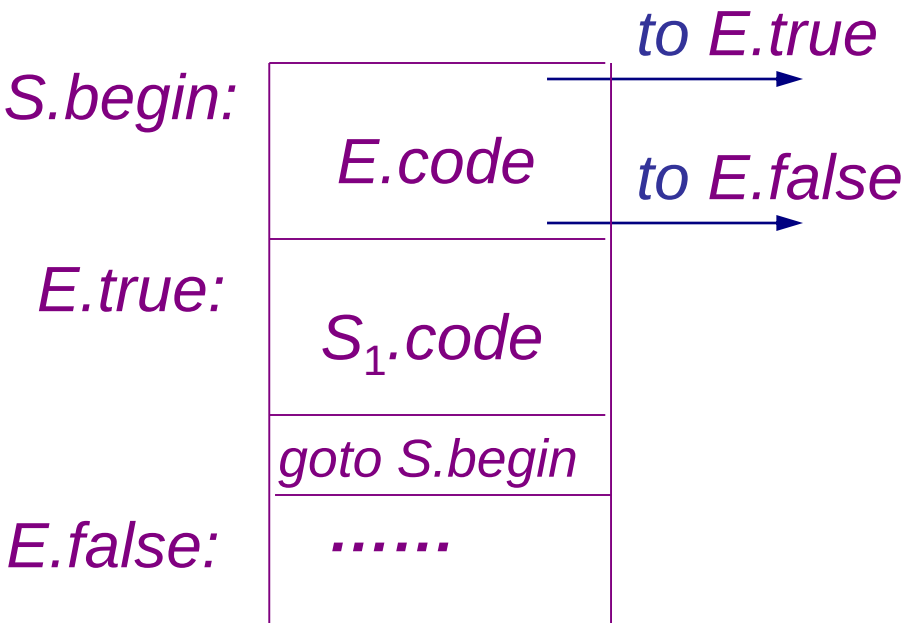
|| *E.code*

|| *gen(E.true ':')*

|| *S_1.code*

|| *gen('goto' S.begin)*

}



– $S_1;S_2$ 语句

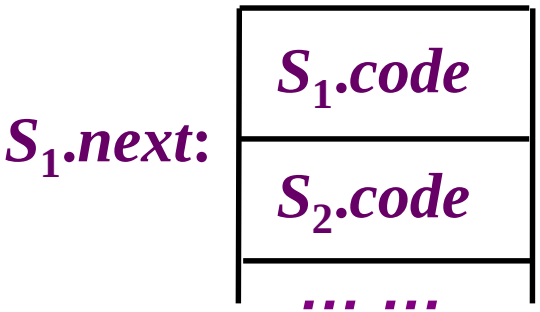
$S \rightarrow S_1; S_2$ {

$S_1.next := newlabel$;

$S_2.next := S.next$;

$S.code := S_1.code \parallel gen(S_1.next, ':') \parallel S_2.code$

}



✧ 开关语句的翻译

```
switch ( E )
   {
        case V1: S1
        case V2: S2
        . . .
        case Vn - 1: Sn – 1
        default: Sn
   }
```

布尔表达式和控制流语句

– 分支数较少时

```
        t := E的代码
        if t ≠ V1 goto L1
        S1的代码
        goto next
L1:     if t ≠ V2 goto L2
        S2的代码
        goto next
L2:     . . .
        . . .
```

```
| Ln-2: if t ≠ Vn-1 goto Ln-1
|          Sn -1的代码
|          goto next
| Ln-1: Sn的代码
| next:
```

– 分支较多时，将分支测试的代码集中在一起，便于生成较好的分支测试代码。

```
        t := E的代码          | Ln:   Sn的代码
        goto test             |       goto next
L1:     S1的代码              |test: if t = V1 goto L1
        goto next             |       if t = V2 goto L2
L2:     S2的代码              |       . . .
        goto next             |       if t = Vn-1 goto Ln-1
        . . .                 |       goto Ln
Ln-1:   Sn -1的代码           | next:
        goto next
```

✧ 过程调用的翻译

$S \rightarrow$ call id (*Elist*)

Elist $\rightarrow$ *Elist*, *E*

Elist $\rightarrow$ *E*

– 过程调用 id(E_1, E_2, …, E_n) 的中间代码结构

E_1.*place* := E_1 的代码

E_2.*place* := E_2 的代码

. . .

E_n.*place* := E_n 的代码

param E_1.*place*

param E_2.*place*

. . .

param E_n.*place*

call id.*place*, *n*

布尔表达式和控制流语句

$S \rightarrow$ call id (*Elist*)

{为长度为*n*的队列中的每个 *E.place*，

emit('param', *E.place*);

emit('call', id.*place*, *n*) }

Elist $\rightarrow$ *Elist*, *E*

{把 *E.place* 放入队列末尾}

Elist $\rightarrow$ *E*

{将队列初始化，并让它仅含 *E.place*}

✧ 过程调用的语法制导翻译

– 简单过程调用的翻译

- 示例：过程调用 CALL S(A＋B，A*B)
  将被翻译为：

```
计算A＋B 置于T中的代码      // T:= A＋B
计算A * B 置于Z中的代码     // Z:= A*B
param   T                   // 第一个实参地址
param   Z                   // 第二个实参地址
call  S,2                   // 转子指令
```

✧ 拉链与代码回填

- 重新审查前面几个和控制流有关的语法制导定义，在条件、循环语句及体现布尔表达式语义的控制流中，一系列语句标号的属性值计算时可能需要多遍扫描分析树。
- 下面介绍一种效率较高的拉链与代码回填（*backpatching*）技术处理此问题，相应于上述语法制导定义给出采用此技术的翻译方案。

✧ 拉链与代码回填

– 语义属性

E.truelist : “真链”，链表中的元素表示 一系列跳转语句的地址，这些跳转语句的目标标号是体现布尔表达式 *E* 为“真”的标号

E. falselist : “假链”，链表中的元素表示 一系列跳转语句的地址，这些跳转语句的目标标号是体现布尔表达式 *E* 为“假”的标号

S. nextlist : “*next* 链”，链表中的元素表示 一系列跳转语句的地址，这些跳转语句的目标标号是在执行序列中紧跟在之后的下一条*TAC*语句的标号

✧ 拉链与代码回填

– 语义函数/过程

makelist(i) : 创建只有一个结点 ***i*** 的表，对应存放目标 *TAC* 语句数组的一个下标

merge(p_1,p_2) : 连接两个链表 ***p_1*** 和 ***p_2*** ，返回结果链表

backpatch(p,i) : 将链表 ***p*** 中每个元素所指向的跳转语句的标号置为 ***i***

nextstm : 下一条*TAC* 语句的地址

emit (…) : 输出一条*TAC* 语句，并使 ***nextstm*** 加1

✧ 拉链与代码回填

– 处理布尔表达式的翻译方案

$E \rightarrow E_1$ or $M\ E_2$	{ *backpatch*(E_1.*falselist*,*M.gotostm*) ; *E.truelist* := *merge*(E_1.*truelist*, E_2.*truelist*) ; *E.falselist* := E_2.*falselist* }
$E \rightarrow E_1$ and $M\ E_2$	{ *backpatch*(E_1.*truelist*,*M.gotostm*) ; *E.falselist* := *merge*(E_1.*falselist*, E_2.*falselist*) ; *E.truelist* := E_2.*truelist* }
$E \rightarrow$ not E_1	{ *E.truelist* := E_1.*falselist* ; *E.falselist* := E_1.*truelist* }

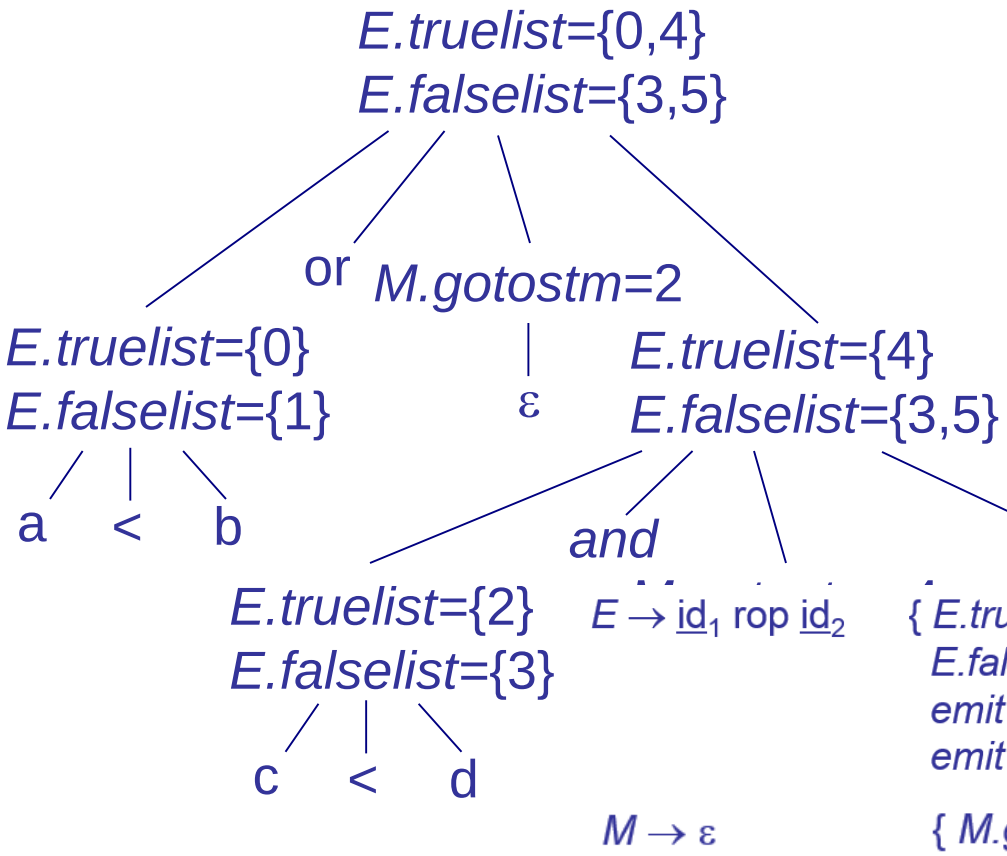
✧ 拉链与代码回填

– 处理布尔表达式的翻译方案

$E \rightarrow (E_1)$	{ *E.truelist := E_1.truelist ;* *E.falselist := E_1.falselist* }
$E \rightarrow \underline{id_1}$ rop $\underline{id_2}$	{ *E.truelist := makelist (nextstm);* *E.falselist := makelist (nextstm+1);* *emit (* ‘if‘ $\underline{id_1}$*.place* rop.op $\underline{id_2}$*.place* ‘goto _’ *);* *emit (*‘goto _’*)* }
$E \rightarrow$ true	{ *E.truelist := makelist (nextstm);* *emit (*‘goto _’*)* }
$E \rightarrow$ false	{ *E.falselist := makelist (nextstm);* *emit (*‘goto _’*)* }
$M \rightarrow \varepsilon$	{ *M.gotostm := nextstm* }

✧ 拉链与代码回填

– 布尔表达式 *E* = *a*<*b* **or** *c*<*d* **and** *e*<*f* 的翻译示意



（0）**if** *a*<*b* **goto** _

（1）**goto** _ (2)

（2）**if** *c*<*d* **goto** _ (4)

（3）**goto** _

（4）**if** *e*<*f* **goto** _

（5）**goto** _

✧ 拉链与代码回填

– 处理条件语句的翻译方案

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

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

$M \rightarrow \varepsilon$
{ *M.gotostm := nextstm* }

$N \rightarrow \varepsilon$
{ *N.nextlist := makelist(nextstm); emit('goto _')* }

✧ 拉链与代码回填

– 处理循环、复合及其它语句的翻译方案

$S \rightarrow$ while M_1 E do M_2 S_1
{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
backpatch(*E.truelist*, M_2.*gotostm*) ;
S.nextlist := *E.falselist*;
emit('goto', M_1.*gotostm*)}

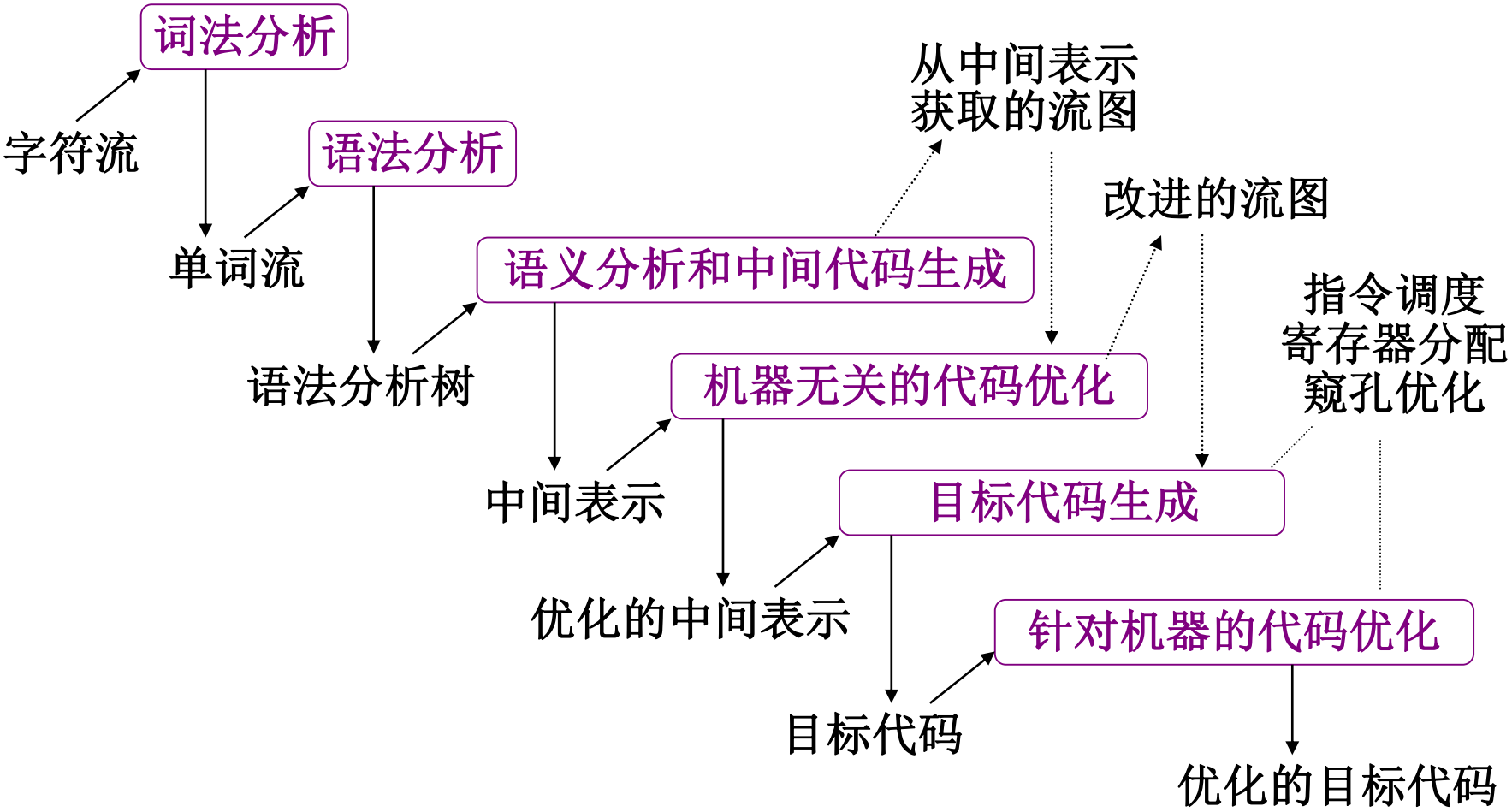
$S \rightarrow$ begin L end { *S.nextlist* := *L.nextlist* }

$S \rightarrow A$ { *S.nextlist* := nil }

$L \rightarrow L_1$; M S
{ *backpatch*(L_1.*nextlist*, *M.gotostm*) ;
L.nextlist := *S.nextlist* }

$L \rightarrow S$ { *L.nextlist* := *S.nextlist* }

✧ 代码生成在编译程序中的逻辑位置



✧ 目标程序

- 可执行目标模块
- 可重定位目标模块
  - 允许程序模块分别编译
  - 调用其它先前编译好的程序模块
- 汇编语言程序
  - 免去编译器重复汇编器的工作
  - 从教学角度，增加可读性

代码生成器的设计中的问题

✧ 代码生成要考虑的主要问题

- 指令选择

  目标机指令集的性质决定指令选择的难易

- 寄存器分配和指派

  尽可能高效地使用寄存器

- 指令排序

  选择好计算的次序，充分利用目标机的特点

✧ 指令选择

– 任务

为每条中间语言语句选择恰当的目标机指令或指令序列

– 原则

- 首先要保证语义的一致性；若目标机指令系统比较完备，为中间语言语句找到语义一致的指令序列模板是很直接的（在不考虑执行效率的情形下）
- 其次要权衡所生成代码的效率（考虑时间/空间代价）

这一点较难做到，因为执行效率往往与该语句的上下文以及目标机体系结构（如流水线）有关

代码生成器的设计中的问题

若不考虑目标程序的效率，指令的选择是直截了当的。

三地址语句 x := y + z（x，y和z都是静态分配）

```
LD      R0,   y          /* 把y装入寄存器R0 */
ADD     R0,   R0,   z    /* z加到R0上 */
ST      x,    R0         /* 把R0存入x中 */
```

其他算术和逻辑运算的*TAC* 语句与此类似，只是选择不同的目标指令，如减运算选择指令SUB，…

代码生成器的设计中的问题

逐个语句地产生代码，常常得到低质量的代码

语句序列　　a := b + c

　　d := a + e

的代码如下

```
LD    R0,  b
ADD   R0,  R0,  c
ST    a,   R0     -- 若a不再使用，第三条也多余
LD    R0,  a      -- 多余的指令
ADD   R0,  R0,  e
ST    d,   R0
```

基本块和流图

怎样为三地址语句序列生成目标代码？

```
begin
    prod := 0;
    i := 1;
    do begin
          prod := prod + a[i] * b[i];
          i := i +1
    end while i <= 20
end
```

```
|(1)   prod := 0
|(2)   i := 1
|(3)   t1 := 4* i
|(4)   t2:= a[t1]
|(5 )  t3 := 4* i
|(6 )  t4 := b[t3]
|(7 )  t5 := t2 * t4
|(8 )  t6 := prod + t5
|(9 )  prod := t6
|(10)  t7 := i +1
|(11)  i := t7
|(12 ) if i <= 20 goto (3)
```

✧ 基本块（*basic block*）

- 概念
  - 基本块是指程序中一个连续的语句序列，控制流从它的开始进入，并从它的末尾离开
- 流图
  - 用有向边表示基本块之间的控制流信息，就能得到程序的流图

✧ 基本块划分算法

- 针对三地址码（TAC）
- 步骤
  - 首先确定所有的入口语句（首指令，leader）
    - 序列的第一个语句是入口语句
    - 能由条件转移语句或无条件转移语句转到的语句是入口语句
    - 紧跟在条件转移语句或无条件转移语句后面的语句是入口语句
  - 每个入口语句到下一个入口语句之前的语句序列构成一个基本块

✧ 基本块划分例 I

(1) prod := 0
(2) i := 1
(3) t_1 := 4∗ i
(4) t_2:= a[t_1]
(5) t_3 := 4∗ i
(6) t_4 := b[t_3]
(7) t_5 := t_2 ∗ t_4
(8) t_6 := prod + t_5
(9) prod := t_6
(10) t_7 := i +1
(11) i := t_7
(12) if i <= 20 goto (3)

(1)prod := 0
(2) i := 1

B_1

(3) t_1 := 4* i
(4) t_2:= a[t_1]
(5) t_3 := 4* i
(6) t_4 := b[t_3]
(7) t_5 := t_2 * t_4
(8) t_6 := prod + t_5
(9) prod := t_6
(10) t_7 := i +1
(11) i := t_7
(12) if i <= 20 goto (3)

B_2

✧ 基本块划分例 II

- 针对三地址码（TAC）
- 举例 右边 TAC 程序可划分成 **4** 个基本块

B1
（1）
（2）

B2
（3）
（4）

B3
（5）
（6）
（7）

B4
（8）
（9）

```
*(1)  read  x
  (2)  read y
*(3)  r:=x mod y
  (4)  if r=0 goto (8)
*(5)  x:=y
  (6)  y:=r
  (7)  goto(3)
*(8)  write y
  (9)  halt
```

✧ 流图（*flow graph*）

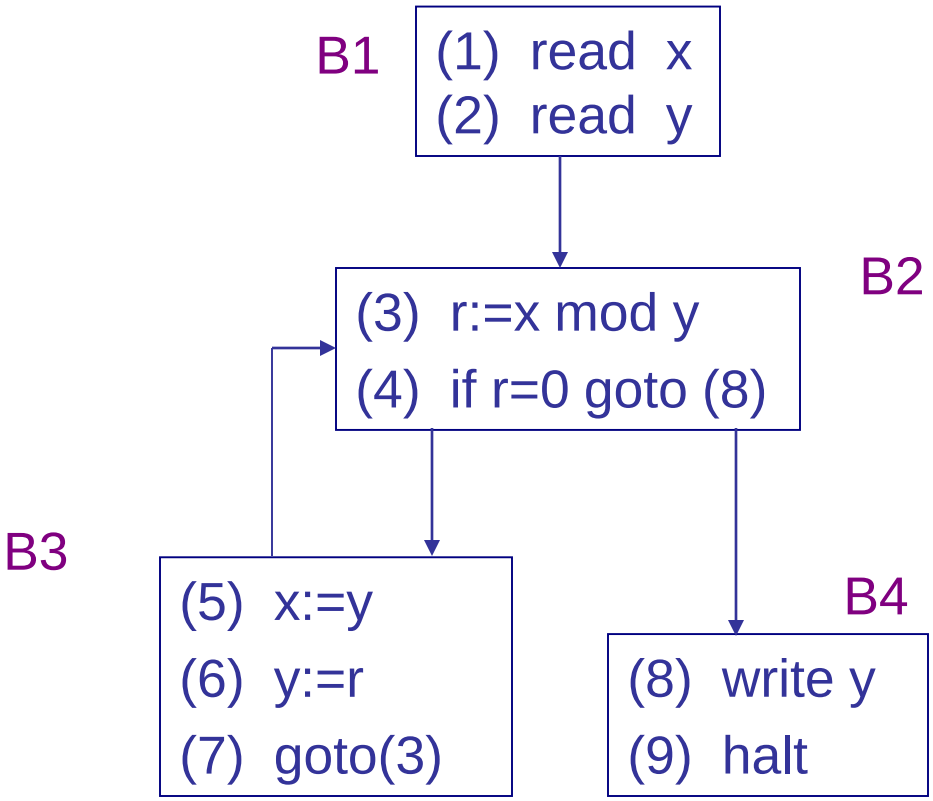
– **概念** 可以为构成一个程序的基本块增加控制流信息，方法是构造一个有向图，称之为**流图**

流图以**基本块**集为**结点**集；**第一个结点**为含有程序第一条语句的基本块；从基本块 *i* 到基本块 *j* 之间存在有向边，当且仅当

- 基本块 *j* 在程序的位置紧跟在 *i* 后,且 *i* 的出口语句**不是**无条件转移或停语句；或者
- *i* 的出口是 *goto(S)* 或 *if goto(S)*, 而 *(S)* 是 *j* 的入口语句

✧ 流图

– 举例




```
*(1)  read  x
  (2)  read y
*(3)  r:=x mod y
  (4)  if r=0 goto (8)
*(5)  x:=y
  (6)  y:=r
  (7)  goto(3)
*(8)  write y
  (9)  halt
```

- ✧ 简单的归类
  - – 依优化范围划分
    - 窥孔优化（*peephole optimization*）

      局部的几条指令范围内的优化
    - 局部优化

      基本块范围内的优化
    - 全局优化

      流图范围内的优化
    - 过程间优化

      整个程序范围内的优化

✧ 简单的归类

– 依优化对象划分

- 目标代码优化

  面向目标代码

- 中间代码优化

  面向程序的中间表示

- 源级优化

  面向源程序

- 简单的归类
  - 依优化侧面划分
    - 指令调度
    - 寄存器分配
    - 存储层次优化
    - 循环优化
    - 控制流优化
    - 过程优化
    - ……

✧ 窥孔优化（*peephole optimization*）

- 工作方式 在目标指令序列上滑动一个包含几条指令的窗口（称为窥孔），发现其中不够优化的指令序列，用一段更短或更有效的指令序列来替代它，使整个代码得到改进
- 虽然窥孔优化的话题主要针对目标代码的优化，但类似的技术也可以应用于中间代码的优化

✧ 窥孔优化（*peephole optimization*）

– 举例

- 删除冗余的加载和保存（*redundant loads and stores*）

  指令序列

  （1） ST a, R0
  （2） LD R0, a

  可优化为

  （1） ST a, R0

✧ 窥孔优化（*peephole optimization*）

– 举例

- 合并已知量（*constants folding*）

  代码序列

  （1） r2:=3*2

  可优化为

  （1） r2:=6

✧ 窥孔优化（*peephole optimization*）

– 举例

- 常量传播（*constants propagating*）

代码序列

（1） r2:=4
（2） r3:=r1+r2

可优化为

（1） r2:=4
（2） r3:=r1+ 4

注：虽然条数未少，但若知道是r2不活跃时，可删除（1）

✧ 窥孔优化（*peephole optimization*）

– 举例

- 代数化简（*algebraic simplification*）

  代码序列

  （1） x:=x+0
  （2） ……
  （n） y:=y*1

  中的（1），（n）可在窥孔优化时删除

✧ 窥孔优化（*peephole optimization*）

– 举例

- 控制流优化（*flow-of-control optimization*）

  代码序列

```
        goto L1
        ……
L1:     goto L2
```

  可替换为

```
        goto L2
        ……
L1:     goto L2
```

✧ 窥孔优化（*peephole optimization*）

– 举例

- 死代码删除（*dead-code elimination*）

代码序列

```
debug := false
if (debug) print …
……
```

可替换为

```
debug := false
……
```

✧ 窥孔优化（*peephole optimization*）

– 举例

- 强度削弱（*reduction in strength*）

  x:=2.0*f　可替换为　x:=f+f

✧ 窥孔优化（*peephole optimization*）

– 举例

- 使用目标机惯用指令（*use of idioms*）

  某个操作数与*1* 相加，通常用“加*1*”指令，而不是用“加”指令

  某个定点数乘以*2*，可以采用“左移”指令；而除以*2*，则可以采用“右移”指令

  …

✧ 基本块内的优化

– 举例

- 例

  合并已知量

  删除多余运算（公共表达式删除）

  删除无用赋值

✧ 全局优化（*global optimization*）

– 借助于针对流图的数据流分析进行的优化

- 例

  全局公共表达式删除

  全局死代码删除（删除从流图入口不能到达的代码）

  ……

✧ 循环优化（*loop optimization*）

– 举例

- 代码外提（*code motion*）

```
while （i< limit/2）{…} // 无改变limit值的语句
```

等价于 t:=limit/2;

```
while （i< t）{…}
```

- 循环不变量（*loop-invariant*）代码可以外提

如对于循环内部的语句 x:=y+z，若 y 和 z 的定值点都在循环外，则 x:=y+z 为循环不变量

Conclusions

- ✧ 运行时存储组织
  - ✓ 活动记录
  - ✓ 静态链
  - ✓ 动态链
- ✧ 中间代码生成
  - ✓ 中间代码
  - ✓ 声明语句
  - ✓ 赋值语句
  - ✓ 布尔表达式
  - ✓ 控制语句
  - ✓ 拉链与代码回填
- ✧ 代码生成
- ✧ 代码优化基础

推荐教学资料

- §6.1-6.4, §6.5.1 §6.6 - 6.8
- §7.1-7.4
- §8.1, §8.4
- §9.1

课外学习建议

- Paakki, J.. Attribute Grammar Paradigms — A High-Level Methodology in Language Implementation. Computing Surveys. 27:2, 196-255. 1995
- Chapter 5, *Modern Compiler Design*, Dick Grune, et al. John Wiley & Sons Press, 2000

Thank you!