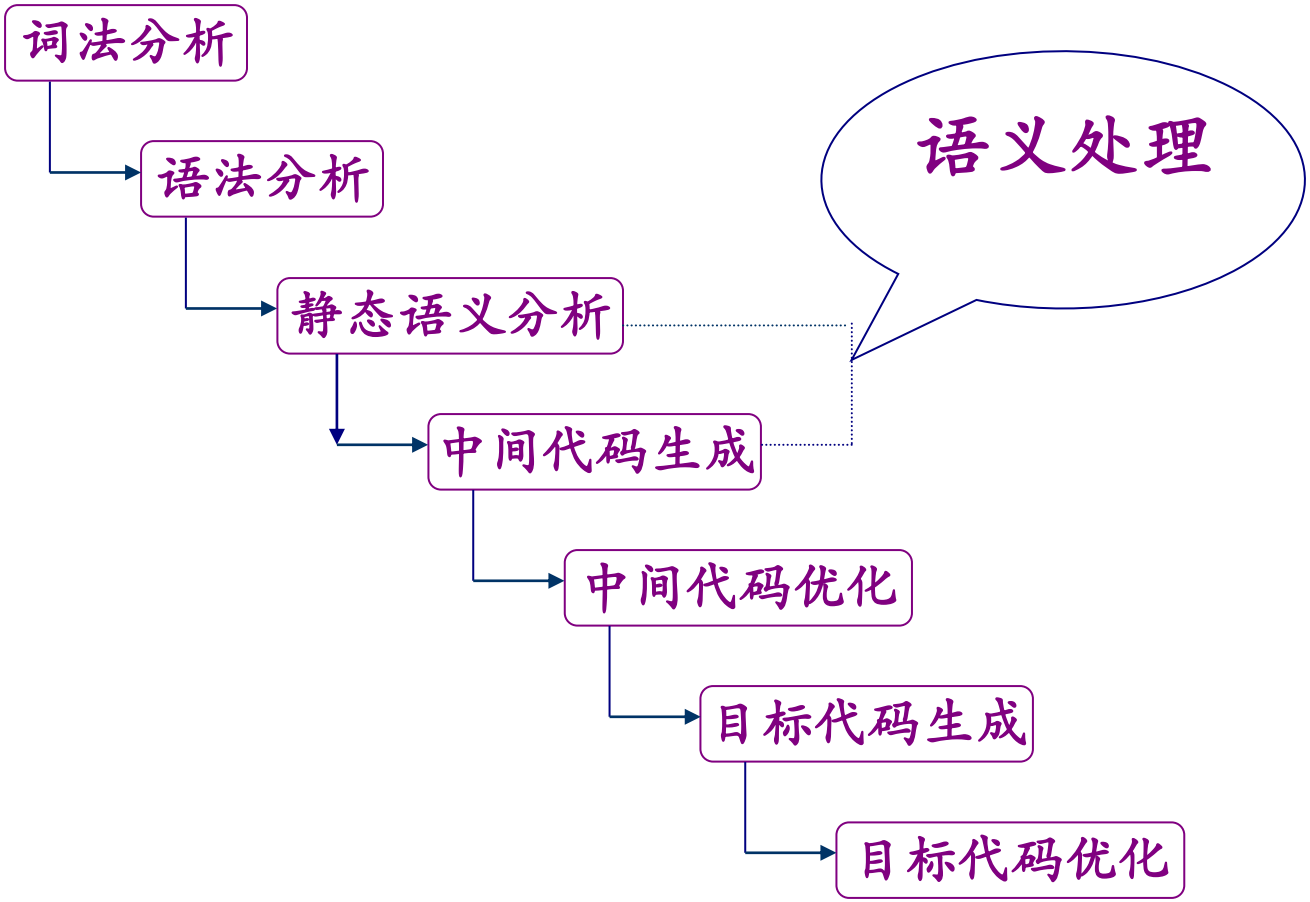
✧ 静态语义分析与中间代码生成

- 静态语义分析和中间代码生成在编译程序中的逻辑位置



✧ 重要数据结构

– 符号表（*symbol tables*）

- 名字信息建立后加入/更改符号表
  名字信息如：种类，类型，偏移地址，占用空间等
- 需要获取名字信息时，查找符号表
- 符号表的组织可以体现名字作用域规则

（符号表的组织已在第六讲专门讨论）

静态语义分析与中间代码生成

- 静态语义分析（或语义分析）
- 中间代码生成

语义分析

✧ 与语义分析相关的工作

- 静态语义检查
  - 编译期间所进行的语义检查
- 动态语义检查
  - 所生成的代码在运行期间进行的语义检查
- 收集语义信息
  - 为语义检查收集程序的语义信息
  - 为代码生成等后续阶段收集程序的语义信息

  有些内容合并到“中间代码生成”部分讨论
  （如过程、数组声明的语义处理）

语义分析

✧ 静态语义检查

– 代码生成前程序合法性检查的最后阶段

- 静态类型检查（*type checks*）
  检查每个操作是否遵守语言类型系统的定义
- 名字的作用域（*scope*）分析
  建立名字的定义和使用之间联系
- 控制流检查（*flow-of-control checks*）
  控制流语句必须使控制转移到合法的地方（如 *break* 语句必须有合法的语句包围它）
- 唯一性检查（*uniqueness checks*） 很多场合要求对象只能被定义一次（如枚举类型的元素不能重复出现）
- 名字的上下文相关性检查（*name-related checks*） 某些名字的多次出现之间应该满足一定的上下文相关性
- ……

语义分析

✧ 类型检查

- 类型检查程序（*type checker*）负责类型检查
  - 验证语言结构是否匹配上下文所期望的类型
  - 为相关阶段搜集及建立必要的类型信息
  - 实现某个类型系统（*type system*）
- 静态类型检查
  - 编译期间进行的类型检查
- 动态类型检查
  - 目标程序运行期间进行的类型检查

✧ 类型系统（简介）

– 作用

- 维护程序中变量,表达式及其他单元的类型信息
- 刻画程序的行为是否良好/安全可靠
- 规范类型检查过程的实现

✧ 类型系统（简介）

– 类型系统的定义

- 语法范畴

  定义合法的程序单元

- 语义范畴

  定义类型表达式

- 类型环境

  定义标识符作用域，维护程序中变量的类型

- 类型规则

  为程序单元定义类型表达式

✧ 类型系统（简介）

– 类型系统示例

- 一个简单语言

$P \to D\ ;\ S$

$D \to V\ ;\ F$

$V \to V\ ;\ T\ L \mid \varepsilon$

$T \to$ boolean $\mid$ integer $\mid$ real $\mid$ array [num] of T $\mid$ ^T

$L \to L\ ,\ \underline{\text{id}} \mid \underline{\text{id}}$

$S \to \underline{\text{id}} := E \mid$ if E then $S \mid$ if E then S else $S \mid$ while E then S

$\mid S\ ;\ S \mid$ break $\mid$ call $\underline{\text{id}}$ (A)

$E \to$ true $\mid$ false $\mid \underline{\text{literal}} \mid \underline{\text{int}} \mid \underline{\text{real}} \mid \underline{\text{id}} \mid E\ \underline{\text{op}}\ E \mid E\ \underline{\text{rop}}\ E \mid E[E] \mid E\text{^}$

$F \to F\ ;\ \underline{\text{id}}\ (\ V\ )\ S \mid \varepsilon$

$A \to A\ ,\ E \mid \varepsilon$

✧ 类型系统（简介）

– 类型系统示例

- **类型表达式**

纯量类型表达式：*bool, int, real*

有界数组类型表达式：*array(I,T)* ，其中，$T \in \{ bool, int, real \}$；*I* 代表一个整数区间，如 *1..10*

指针数据类型表达式：*pointer(T)*，$T \in \{ bool, int, real \}$

积类型表达式：$<T_1, T_2, \ldots, T_n>$ ，其中，$T_1, T_2, \ldots, T_n$ 为上述数据类型表达式；若$n=0$，则表示为 < >

过程类型表达式：*fun*（*T*），*T* 是上述积类型表达式

type_error　专用于有类型错误的程序单元

ok　专用于没有类型错误的程序单元

✧ 类型系统（简介）

– 类型系统示例

- 类型环境

全局类型环境 G：

记录全局变量标识符以及函数标识符的类型表达式

局部类型环境 F:

记录函数（过程）形参变量的类型表达式

✧ 类型系统（简介）

– 类型系统示例

- 类型规则

引入下列断言形式（judgements）：

$$G \vdash e : A$$

$$G, F \vdash e : A$$

$$\vdash e : A$$

✧ 类型系统（简介）

– 类型系统示例

- 类型规则（针对部分表达式）

$$\frac{}{\xi \vdash \underline{\text{int}} : int}$$

（ξ 表示‘G’或者‘$G,\ F$’，下同）

$$\frac{(\underline{\text{id}} : \tau) \in G}{G \vdash \underline{\text{id}} : \tau} \qquad \frac{(\underline{\text{id}} : \tau) \in G \cup F}{G,\ F \vdash \underline{\text{id}} : \tau}$$

$$\frac{\xi \vdash e_1 : \tau \quad \xi \vdash e_2 : \tau}{\xi \vdash e_1\ \underline{\text{op}}\ e_2 : \tau}$$

$$\frac{\xi \vdash e_1 : array(I, \tau) \quad \xi \vdash e_2 : int}{\xi \vdash e_1[e_2] : \tau}$$

✧ 类型系统（简介）

– 类型系统示例

- 类型规则（针对部分语句）

$$\frac{\xi \vdash \text{id} : \tau \quad \xi \vdash e : \tau}{\xi \vdash \text{id} := e \ : \ ok}$$

$$\frac{\xi \vdash e : bool \quad \xi \vdash s_1 : ok \quad \xi \vdash s_2 : ok}{\xi \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 \ : \ ok}$$

$$\frac{\xi \vdash \text{while } e \text{ then } s_1 : ok \quad \xi \vdash \text{while } e \text{ then } s_2 : ok \quad s_1 \text{ or } s_2 \text{ may be empty}}{\xi \vdash \text{while } e \text{ then } s_1 \text{ ; break; } s_2 \ : \ ok}$$

$$\frac{G \vdash \text{id} : fun(<\tau_1, \tau_2, \ldots, \tau_n>) \quad \xi \vdash e_1 : \tau_1 \quad \ldots \quad \xi \vdash en : \tau_n}{\xi \vdash \text{call id } (e_1, e_2, \ldots, e_n) \ : \ ok}$$

✧ 类型系统（简介）

– 类型系统示例

- 类型规则（针对类型声明或声明语句

$$\frac{}{\vdash \text{integer} : int} \qquad \frac{\vdash t : \tau}{\vdash \hat{}t : pointer(\tau)}$$

$$\frac{G \vdash v : < \tau_1, \ldots, \tau_m > \quad \vdash t : \tau \quad l = x_1, \ldots, x_n \quad \{ x_1, \ldots, x_n \} \cap \mathrm{dom}(G) = \phi}{G \cup \{ (x_1, \tau) \ldots, (x_n, \tau) \} \vdash v ; t\ l : < \tau_1, \ldots, \tau_m, \tau, \ldots, \tau > \ (n \text{个} \tau)}$$

（Here，we assume ‘ε ; *t l* ‘ to be ‘*t l* ‘ and omit the rule: $\vdash \varepsilon : < >$. dom(*G*) is the set of identifiers in the domain of *G*.）

✧ 类型系统（简介）

– 类型系统相关话题

- 类型等价（equivalence）

  结构等价，名字等价，合一算法

- 类型推导（inference）

  静态/动态类型推导

- 子类型（subtyping）关系

  类型转换，类型兼容，多态，重载

- 类型合理性/可靠性（Soundness）

  类型良定（well-typed）的程序是行为安全的

- ……

✧ 类型检查程序的设计

– 语法制导的方法

- 将类型表达式作为属性值赋给程序各个部分
- 设计恰当的翻译模式
- 可实现相应语言的一个类型系统

✧ 语法制导的类型检查程序 —— 举例

– 处理声明的翻译模式

$V \rightarrow V_1 \; ; \; T \quad \{ L.in := T.type \}$
$\quad L \quad \{ V.type := make\_product\_3 (V_1.type, T.type, L.num) \}$

$V \rightarrow \varepsilon \quad \{ V.type := <> \}$

$T \rightarrow$ boolean $\quad \{ T.type := bool \}$

$T \rightarrow$ integer $\quad \{ T.type := int \}$

$T \rightarrow$ real $\quad \{ T.type := real \}$

$T \rightarrow$ array [num] of $T_1 \quad \{ T.type := array(1..\ num.lexval, T1.type) \}$

$T \rightarrow \uparrow T_1 \quad \{ T.type := pointer(T1.type) \}$

$L \rightarrow \{ L_1.in := L.in \}\ L_1,$ id $\{ addtype(\text{id}.entry, L.in) \; ; \; L.num := L_1.num + 1 \}$

$L \rightarrow$ id $\quad \{ addtype(\text{id}.entry, L.in); \; L.num := 1 \}$

✧ 语法制导的类型检查程序 —— 举例

– 处理表达式的翻译模式

$E \rightarrow true$ { *E.type := bool* }

$E \rightarrow false$ { *E.type := bool* }

$E \rightarrow \underline{int}$ { *E.type := int* }

$E \rightarrow \underline{real}$ { *E.type := real* }

$E \rightarrow \underline{id}$ { *E.type := if lookup_type(id.name) = nil*
then type_error
else lookup_type(id.name }

✧ 语法制导的类型检查程序——举例

– 处理表达式的翻译模式

$E \rightarrow E_1$ op E_2 { *E.type* := if E_1.*type*=*real* and E_2.*type*=*real* then *real*
else if E_1.*type*=*int* and E_2.*type*=*int* then *int*
else *type_error* }

$E \rightarrow E_1$ rop E_2 { *E.type* := if E_1.*type*=*real* and E_2.*type*=*real* then *bool*
else if E_1.*type*=*int* and E_2.*type*=*int* then *bool*
else *type_error* }

$E \rightarrow E_1 [E_2]$ { *E.type* := if E_2.*type*= *int* and E_1.*type*=*array*(*s*, *t*) then *t*
else *type_error* }

$E \rightarrow E_1$^ { *E.type* := *if* E_1.*type*= *pointer*(*t*) then *t*
else *type_error* }

✧ 语法制导的类型检查程序——举例

– 处理语句、过程声明及程序的翻译模式

$S \rightarrow$ id := E { *S.type* := if *lookup_type* (id.*entry*) = *E.type* then *ok* else *type_error* }

$S \rightarrow$ if E then S_1 { *S.type* := if *E.type=bool* then S_1.*type* else *type_error* }

$S \rightarrow$ if E then S_1 else S_2
{ *S.type* := if *E.type=bool* and S_1.*type* = *ok* and S_2.*type* = *ok* then *ok* else *type_error* }

$S \rightarrow$ while E then S_1 { *S.type* := if *E.type=bool* then S_1.*type* else *type_error* }

$S \rightarrow S_1 ; S_2$ { *S.type* := if S_1.*type* = *ok* and S_2.*type* = *ok* then *ok* else *type_error* }

$S \rightarrow$ break { *S.type* := *ok* }

✧ 语法制导的类型检查程序——举例

– 处理语句、过程声明及程序的翻译模式（续）

$S \rightarrow call\ \underline{id}\ (\ A\ )$
{ *S.type* := if *match* (*lookup_type* (*id.name*), *A.type*)
then *ok else type_error* }

$F \rightarrow F_1\ ;\ \underline{id}\ (\ V\ )\ S$ { *addtype*(*id.entry*, *fun* (*V.type*));
F.type := if F_1*.type* = *ok and S.type* = *ok*
then *ok else type_error* }

$F \rightarrow \varepsilon$ { *F.type* := *ok* }

$A \rightarrow A_1\ ,\ E$ { *A.type* := *make_product_2* (A_1*.type*, *E.type*) }

$A \rightarrow \varepsilon$ { *A.type* := <> }

$P \rightarrow D\ ;\ S$ { *P.type* := if *D.type* = *ok* and *S.type* = *ok*
then *ok* else *type_error* }

$D \rightarrow V\ ;\ F$ { *D.type* := *F.type* }

✧ 语法制导的类型检查程序 — 举例

– 增加处理：*break* 只能在某个循环语句内部

$P \rightarrow D$; { $S.break$:= 0 }
S { $P.type$:= if $D.type = ok$ and $S.type = ok$
then ok else $type\_error$ }

$S \rightarrow$ if E then { $S_1.break := S.break$ }
S_1 { $S.type$:= if $E.type = bool$ then $S_1.type$ else $type\_error$ }

$S \rightarrow$ if E then { $S_1.break := S.break$ } S_1
else { $S_2.break := S.break$ } S_2
{ $S.type$:= if $E.type = bool$ and $S_1.type = ok$ and $S_2.type = ok$
then ok else $type\_error$ }

✧ 语法制导的类型检查程序 — 举例

– 增加处理：*break* 只能在某个循环语句内部（续）

$S \rightarrow$ while E then { S_1.*break* := 1 } S_1
{ *S.type* := if *E.type*=*bool* then S_1.*type* else *type_error* }

$S \rightarrow$ { S_1 .*break* := *S.break* } S_1 ; { S_2 .*break* := *S.break* } S_2
{ *S.type* := if S_1.*type* = *ok* and S_2.*type* = *ok*
then *ok* else *type_error* }

$S \rightarrow$ break { *S.type* := if *S.break* = 1
then *ok* else *type_error* }

$F \rightarrow F_1$; id (V) { *S.break* := 0 }
S { *addtype*(id.*entry*, *fun* (*V.type*));
F.type := if F_1.*type* = *ok* and *S.type* = *ok*
then *ok* else *type_error* }

✧ 作用域分析

- 静态作用域
  - 通过符号表实现
    （参见第五讲）
- 动态作用域
  - 通过运行时活动记录实现
    （参见第八讲）

中间代码生成

✧ 中间代码

- 源程序的不同表示形式
- 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

中间代码生成

✧ 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）
  - *TAC*（*Three-address code*,三地址码，四元式）
  - *P-code* （特别用于 *Pasal* 语言实现）
  - *Bytecode*（*Java* 编译器的输出, *Java* 虚拟机的输入）
  - *SSA*（*Static single assignment form*，静态单赋值形式）

中间代码生成

✧ 中间代码举例

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

- *TAC*（三地址码）表示

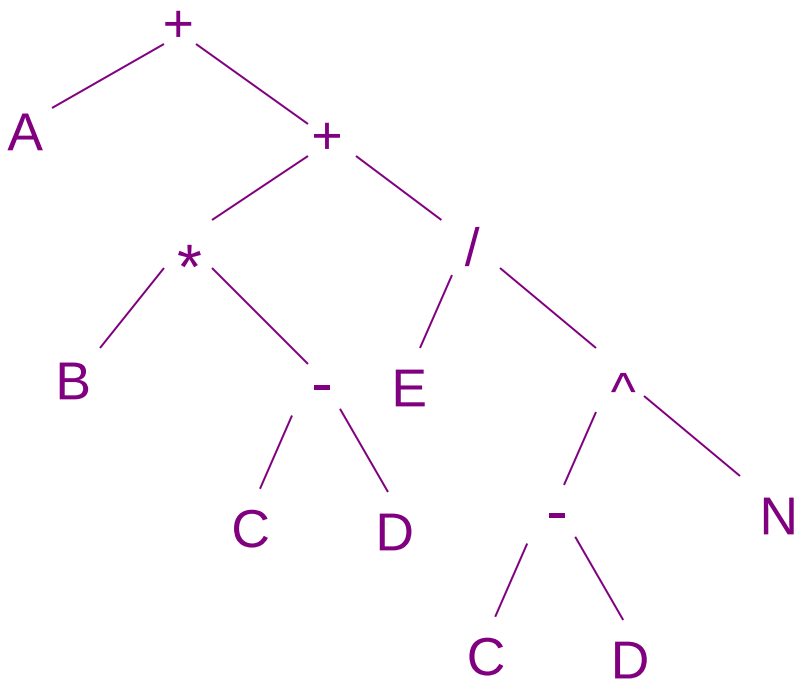
(1)	(-	C	D	T1)		T1 := C - D
(2)	(*	B	T1	T2)		T2 := B * T1
(3)	(+	A	T2	T3)		T3 := A + T2
(4)	(-	C	D	T4)	或	T4 := C - D
(5)	(^	T4	N	T5)		T5 := T4 ^ N
(6)	(/	E	T5	T6)		T6 := E / T5
(7)	(+	T3	T6	T7)		T7 := T3 + T6

中间代码生成

✧ 中间代码举例

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

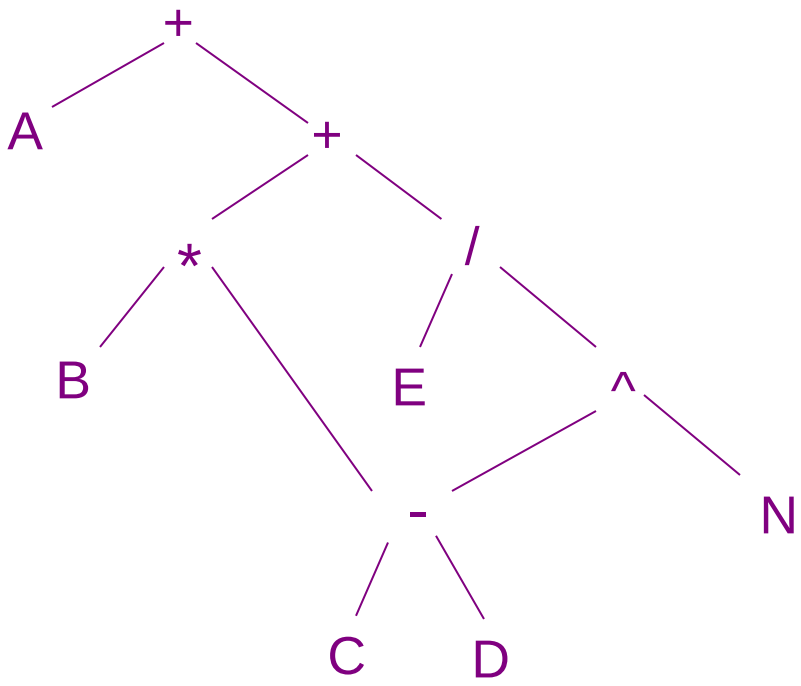
• *AST*（抽象语法树）表示



✧ 中间代码举例

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

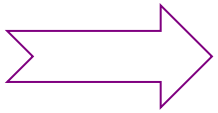
- *DAG*（*Directed Acyclic Graph*,有向无圈图，改进型 *AST*）



✧ 中间代码举例

– 静态单赋值形式

```
x ← 5
x ← x - 3
if x < 3
then
    y ← x * 2
    w ← y
else
    y ← x - 3
w ← x - y
z ← x + y
```



```
x1 ← 5
x2 ← x1 - 3
if x2 < 3
then
    y1 ← x2 * 2
    w1 ← y1
else
    y2 ← x2 - 3
y3 ← ϕ(y1 , y2)
w2 ← x2 - y3
z ← x2 + y3
```

中间代码生成

✧ 中间代码生成

– 语法制导的方法

- 例：生成抽象语法树

mknode: 构造内部结点
Mkleaf : 构造叶子结点

$S \rightarrow \underline{\text{id}} := E$	{ *S.ptr* := *mknode*(‘assign’, *mkleaf*(id.*entry*), *E.ptr*) }
$S \rightarrow$ if E then S_1	{ *S.ptr* := *mknode*(‘if_then’, *E.ptr*, S_1.*ptr*) }
$S \rightarrow$ while E do S_1	{ *S.ptr* := *mknode*(‘while_do’, *E.ptr*, S_1.*ptr*) }
$S \rightarrow S_1 ; S_2$	{ *S.ptr* := *mknode*(‘seq’ , S_1.*ptr* , S_2.*ptr*) }
$E \rightarrow \underline{\text{id}}$	{ *E.ptr* := *mkleaf*(id.*entry*) }
$E \rightarrow E_1 + E_2$	{ *E.ptr* := *mknode*(‘+’ , E_1.*ptr* , E_2.*ptr*)}
$E \rightarrow E_1 * E_2$	{ *E.ptr* := *mknode*(‘*’ , E_1.*ptr* , E_2.*ptr*)}
$E \rightarrow (E_1)$	{ *E.ptr* := E_1.*ptr* }
……	……

✧ 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 为操作符，y 和 z 为操作数， x 为结果)

中间代码生成

✧ 课程后续部分用到的 TAC 语句类型

- 赋值语句 *x := y op z*　（*op* 代表二元算术/逻辑运算）
- 赋值语句 *x := op y*　（*op* 代表一元运算）
- 复写语句 *x := y*　（*y* 的值赋值给 *x*）
- 无条件跳转语句 *goto L*　（无条件跳转至标号 *L*）
- 条件跳转语句 *if x rop y goto L*　（*rop* 代表关系运算）
- 标号语句 *L*：（定义标号 *L*）
- 过程调用语句序列 *param* x_1 … *param* x_n *call p,n*
- 过程返回语句 *return y*（*y* 可选，存放返回值）
- 下标赋值语句 *x := y[i]* 和 *x[i] := y*（前者表示将地址 *y* 起第 *i* 个存储单元的值赋给 *x*，后者类似）
- 指针赋值语句 *x := *y* 和 **x := y*

中间代码生成

✧ 赋值语句及算数表达式的语法制导翻译

– 语义属性

id.*place* : id 对应的存储位置

E.place : 用来存放 *E* 的值的存储位置

E.code : *E* 求值的 *TAC* 语句序列

S.code : 对应于 *S* 的 *TAC* 语句序列

– 语义函数/过程

gen : 生成一条 *TAC* 语句

newtemp : 在符号表中新建一个从未使用过的名字，并返回该名字的存储位置

|| 是*TAC* 语句序列之间的链接运算

✧ 赋值语句及算数表达式的语法制导翻译

– 翻译模式

$S \rightarrow$ id := E { $S.code$:= $E.code$ || gen(id $.place$ ':=' $E.place$) }

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

$E \rightarrow$ int { $E.place$:= $newtemp$; $E.code$:= gen ($E.place$ ':=' int $.val$) }

$E \rightarrow$ real { $E.place$:= $newtemp$; $E.code$:= gen ($E.place$ ':=' real $.val$) }

$E \rightarrow E_1 + E_2$ { $E.place$:= $newtemp$; $E.code$:= $E_1.code$ || $E_2.code$ || 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 ($E.place$ ':=' '$uminus$' $E_1.place$) }

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

中间代码生成

✧ 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 （综合属性）

  T.width，*V.width* : 数据宽度（字节数）

  L.offset : 列表中第一个变量的偏移地址

  L.type : 变量列表被申明的类型 （继承属性）

  L.num : 变量列表中变量的个数

- 语义函数/过程

  enter (id.*name*, *t*, *o*) : 将符号表中 id.*name* 所对应表项的 *type* 域置为 *t*，*offset* 域置为 *o*

✧ 说明语句的语法制导翻译

– 翻译模式

$V \rightarrow V_1$; T { $L.type := T.type$; $L.offset := V_1.width$; $L.width := T.width$ }
L { $V.type := make\_product\_3\ (V_1.type, T.type, L.num)$;
$V.width := V_1.width + L.num \times T.width$ }

$V \rightarrow \varepsilon$ { $V.type :=$ <>; $V.width := 0$ }

$T \rightarrow$ boolean { $T.type := bool$; $T.width := 1$ }

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

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

$T \rightarrow$ array [num] of T_1 { $T.type := array(1..\ num.lexval, T_1.type)$;
$T.width := num.val \times T_1.width$ }

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

$L \rightarrow$ { $L_1.\ type := L.\ type$; $L_1.\ offset := L.\ offset$; $L_1.\ width := L.\ width$; }
L_1 , id { $enter\ (id.name, L.\ type, L.\ offset + L_1.num \times L.\ width)$;
$L.num := L_1.num + 1$ }

$L \rightarrow$ id { $enter\ (id.name, L.\ type, L.\ offset)$; $L.num := 1$}

✧ 数组说明和数组元素引用的语法制导翻译

– 数组说明

参考前页的翻译模式，可了解（一维）数组说明的翻译思想. 至于符号表中一般情况下是如何组织数组说明信息的，随后将会讨论.

……

$T \rightarrow$ array [num] of T_1 { $T.type$:= $array(1..\ num.lexval, T_1.type)$;
$T.width := num.val \times T_1.width$ }

.....

$L \rightarrow$ { $L_1.\ type := L.\ type$; $L_1.\ offset := L.\ offset$; $L_1.\ width := L.\ width$; }
L_1 , id { $enter(id.name, L.\ type, L.\ offset + L_1.num \times L.\ width)$;
$L.num := L_1.num + 1$ }

$L \rightarrow$ id { $enter(id.name, L.\ type, L.\ offset)$; $L.num := 1$}

✧ 数组说明和数组元素引用的语法制导翻译

– 数组引用

$S \rightarrow E_1[E_2] := E_3$ { $S.code := E_2.code \,||\, E_3.code \,||$ $gen(E_1.place$ '[' $E_2.place$ ']' ':=' $E_3.place)$ }

$E \rightarrow E_1[E_2]$ { $E.place := newtemp;$
$E.code := E_2.code \,||$
$gen(E.place$ ':=' $E_1.place$ '[' $E_2.place$ ']') }

✧ 数组说明和数组元素引用的语法制导翻译

– 数组的内情向量（*dove 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

✧ 数组说明和数组元素引用的语法制导翻译

– 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）

✧ 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，如：在已知 E_1 为真时，不必再对 $E_1 \vee E_2$ 中的 E_2 进行求值；同样，在已知 E_1 为假时，不必再对 $E_1 \wedge E_2$ 中的 E_2 进行求值

中间代码生成

✧ 布尔表达式的语法制导翻译

– 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

$E \to E_1 \vee E_2$ { *E.place := newtemp; E.code := E_1.code || E_2.code || gen (E.place ':=' E_1.place 'or' E_2.place)* }

$E \to E_1 \wedge E_2$ { *E.place := newtemp; E.code := E_1.code || E_2.code || gen (E.place ':=' E_1.place 'and' E_2.place)* }

$E \to \neg E_1$ { *E.place := newtemp; E.code := E_1.code || gen (E.place ':=' 'not' E_1.palce)* }

$E \to (E_1)$ { *E.place := E_1.place ; E.code := E_1.code* }

$E \to \underline{id_1}\ \underline{rop}\ \underline{id_2}$ { *E.place := newtemp; E.code := gen ('if' $\underline{id_1}$.place* $\underline{rop}$*.op* $\underline{id_2}$*.place 'goto' nextstat+3) || gen (E.place ':=' '0') || gen ('goto' nextstat+2) || gen (E.place ':=' '1')* }

$E \to$ true { *E.place := newtemp; E.code := gen(E.place ':=' '1')* }

$E \to$ false { *E.place := newtemp; E.code := gen(E.place ':=' '0')* }

✧ 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）：

```
    if a<b goto E.true
    goto  label1
label1:
    if c<d goto label2
    goto E.false
label2:
    if e<f goto E.true
    goto E.false
```

✧ 布尔表达式的语法制导翻译

– 翻译布尔表达式至短路代码（L-翻译模式）

$E \rightarrow \{ E_1.true := E.true;\ E_1.false := newlabel \}\ E_1 \vee$
$\{ E_2.true := E.true;\ E_2.false := E.false \}\ E_2$
$\{ E.code := E_1.code \parallel gen(E_1.false\ ':') \parallel E_2.code \}$

$E \rightarrow \{ E_1.false := E.false;\ E_1.true := newlabel \}\ E_1 \wedge$
$\{ E_2.false := E.false;\ E_2.true := E.true \}\ E_2$
$\{ E.code := E_1.code \parallel gen(E_1.true\ ':') \parallel E_2.code \}$

$E \rightarrow \neg \{ E_1.true := E.false;\ E_1.false := E.true \}\ E_1 \{ E.code := E_1.code \}$

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

$E \rightarrow \underline{id_1}\ \underline{rop}\ \underline{id_2}$ $\{ E.code := gen(\text{'if'}\ \underline{id_1}.place\ \underline{rop}.op\ \underline{id_2}.place\ \text{'goto'}\ E.true) \parallel gen(\text{'goto'}\ E.false) \}$

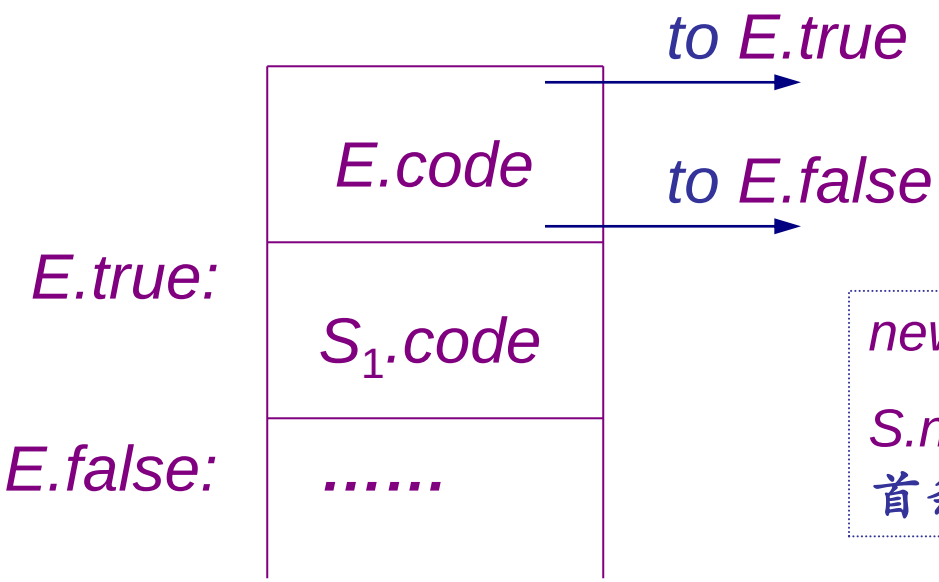
$E \rightarrow true$ $\{ E.code := gen(\text{'goto'}\ E.true) \}$

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

✧ 条件语句的语法制导翻译

– if-then 语句（*L* 翻译模式）

$S \rightarrow$ if { *E.true* := *newlabel* ; *E.false* := *S.next* } *E*
then { S_1.*next* := *S.next* } S_1
{ *S.code* := *E.code* || *gen(E.true* ':' *)* || S_1.*code* }

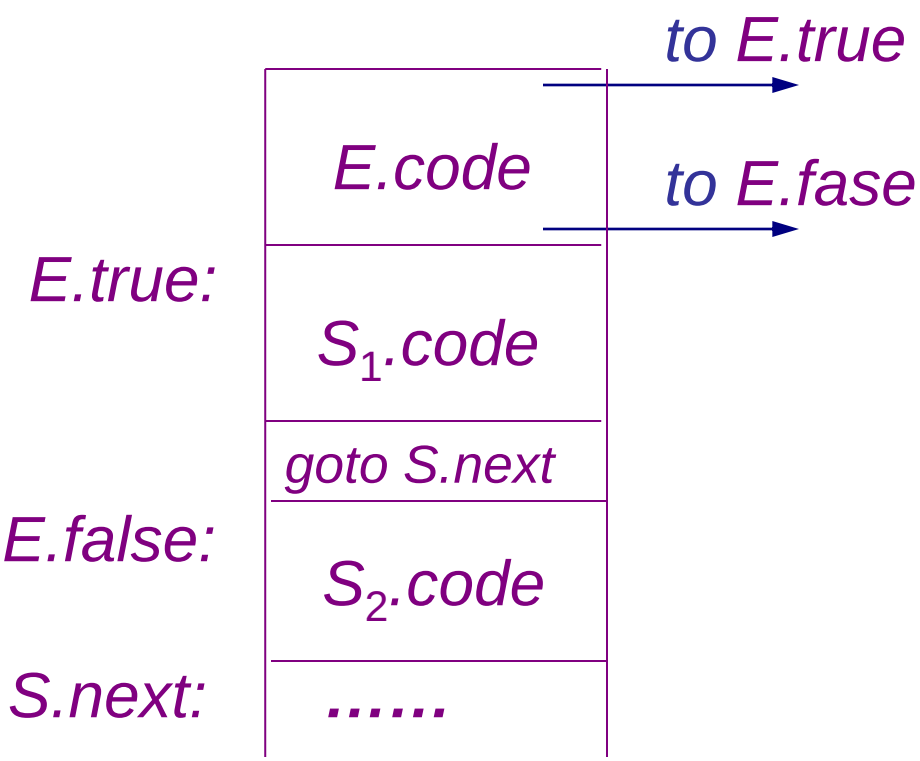


newlabel 返回一个新的语句标号

S.next 属性表示 *S* 之后要执行的首条 *TAC* 语句的标号

✧ 条件语句的语法制导翻译

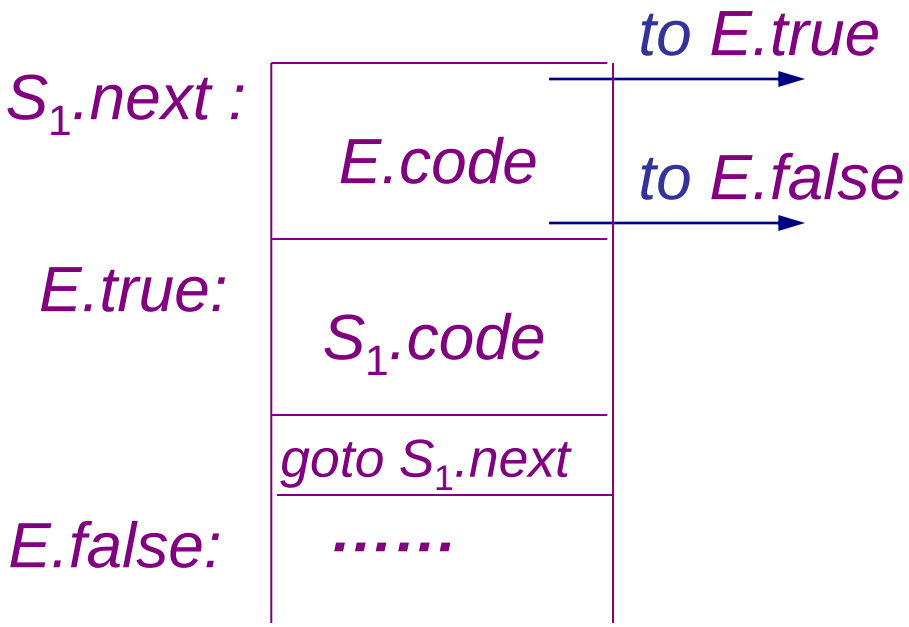
– if-then-else 语句（*L* 翻译模式）



$S \rightarrow$ if { *E.true* := *newlabel*;
E.false := *newlabel* }
E then
{ S_1.*next* := *S.next* }
S_1 else
{ S_2.*next* := *S.next* }
S_2
{ *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翻译

– while 语句（*L* 翻译模式）

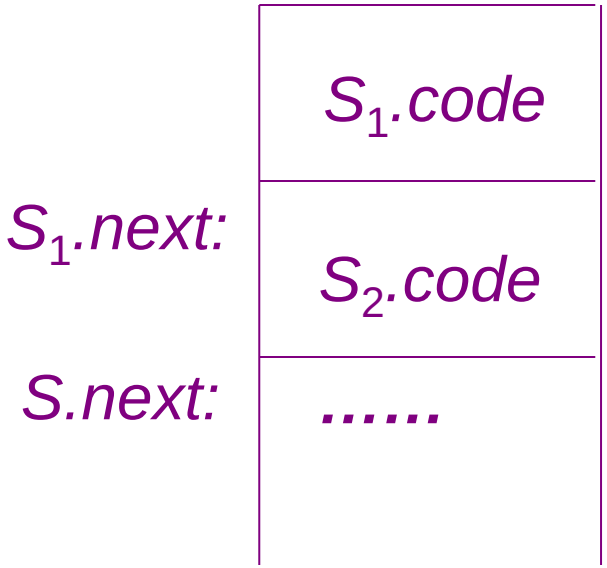


$S \rightarrow$ while
{ *E.true* := *newlabel*;
E.false := *S.next* }
E do
{ S_1.*next* := *newlabel*}
S_1
{ *S.code* := *gen*(S_1.*next* ':')
|| *E.code*
|| *gen*(*E.true* ':')
|| S_1.*code*
|| *gen*('goto' S_1.*next*)
}

✧ 复合语句的语法制导翻译

– 顺序复合语句（*L* 翻译模式）

$S \rightarrow$ { $S_1.next$:= *newlabel* } S_1 ;
{ $S_2.next$:= $S.next$ } S_2
{ $S.code$:= $S_1.code$
|| *gen*($S_1.next$ ':')
|| $S_2.code$
}



✧ 含 *break* 语句的语法制导翻译

– 翻译模式

$P \rightarrow D\ ;\ \{ S.next := newlabel;\ S.break := newlabel \}\ S$
$\{ gen(S.next\ ':') \}$

$S \rightarrow if\ \{ E.true := newlabel;\ E.false := S.next \}\ E\ then$
$\{ S_1.next := S.next;\ S_1.break := S.break \}\ S_1$
$\{ S.code := E.code \| gen(E.\ true\ ':') \| S_1.code \}$

$S \rightarrow if\ \{ E.true := newlabel;\ E.false := newlabel \}\ E\ then$
$\{ S_1.next := S.next;\ S_1.break := S.break \}\ S_1\ else$
$\{ S_2.next := S.next;\ S_2.break := S.break \}\ S_2$
$\{ S.code := E.code \| gen(E.true\ ':') \| S_1.code \|$
$gen('goto'\ S.next) \| gen(E.false\ ':') \| S_2.code \}$

✧ 含 *break* 语句的语法制导翻译

– 翻译模式 （续）

$S \rightarrow$ *while* { *E.true* := *newlabel*; *E.false* := *S.next* } *E do*
{ S_1.*next* := *newlabel*; S_1.*break* := *S.next* } S_1
{ *S.code* := *gen*(S_1.*next* ':') ||
E.code || *gen*(*E.true* ':') || S_1.*code* ||
gen('goto' S_1.*next*) }

$S \rightarrow$ { S_1.*next* := *newlabel*; S_1.*break* := *S.break* } S_1 ;
{ S_2.*next* := *S.next*; S_2.*break* := *S.break* } S_2
{ *S.code* := S_1.*code* || *gen*(S_1.*next* ':') || S_2.*code* }

$S \rightarrow$ *break* ; { *S.code* := *gen*('goto' *S.break*) }

✧ 拉链与代码回填 (*backpatching*)

– 另一种控制流中间代码生成技术

比较：前面的方法采用 L-属性文法/翻译模式

下面的方法采用 S-属性文法/翻译模式

✧ 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 ***S*** 之后的下条*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模式

$E \to E_1 \vee M\ E_2$ { *backpatch*(E_1.*falselist*,*M.gotostm*) ;
E.truelist := *merge*(E_1.*truelist*, E_2.*truelist*) ;
E.falselist := E_2.*falselist* }

$E \to E_1 \wedge M\ E_2$ { *backpatch*(E_1.*truelist*,*M.gotostm*) ;
E.falselist := *merge*(E_1.*falselist*, E_2.*falselist*) ;
E.truelist := E_2.*truelist* }

$E \to \neg\ E_1$ { *E.truelist* := E_1.*falselist* ;
E.falselist := E_1.*truelist* }

注: 这里可以规定产生式的优先级依次递增来解决冲突问题（下同）

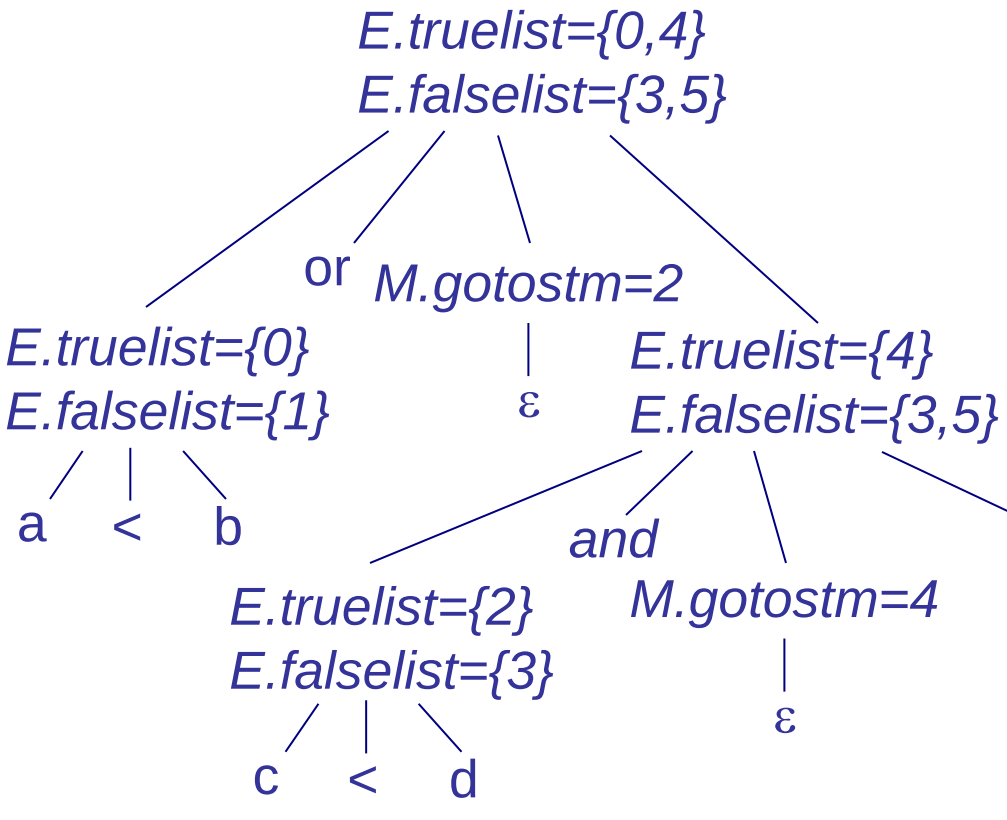
✧ 拉链与代码回填

– 处理布尔表达式的翻译模式 （续）

$E \rightarrow (E_1)$	{ *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 $E = a<b \vee c<d \wedge e<f$ 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模式

$S \rightarrow$ 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、复合的翻译模式

$S \rightarrow$ 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*)}

$S \rightarrow S_1$; M S_2

{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
S.nextlist := S_2.*nextlist* }

$M \rightarrow \varepsilon$

{ *M.gotostm* := *nextstm* }

✧ 拉链与代码回填

– 增加 *break* 语句后控制语句处理的翻译模式

$P \rightarrow D\ ;\ S\ M$ { *backpatch(S.nextlist, M.gotostm) ;*
backpatch(S.breaklist, M.gotostm) }

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

$S \rightarrow$ *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) ;*
$S.nextlist := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist)$;
$S.breaklist := merge(S_1.breaklist, S_2.breaklist)$ }

S. breaklist **：** “*break* 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直接所属 while 语句的结束位置

✧ 拉链与代码回填

– 增加 *break* 语句后控制语句处理的翻译模式（续）

$S \rightarrow$ *while* M_1 *E then* M_2 S_1
{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
backpatch(*E.truelist*, M_2.*gotostm*) ;
S.nextlist := *merge*(*E.falselist*, S_1.*breaklist*);
S.breaklist := “”; *emit*(‘*goto*’, M_1.*gotostm*) }

$S \rightarrow S_1$; M S_2 { *backpatch*(S_1.*nextlist*, *M.gotostm*) ;
S.nextlist := S_2.*nextlist* ;
S.breaklist := *merge*(S_1.*breaklist*, S_2.*breaklist*) }

$S \rightarrow$ *break* ; { *S.breaklist t* := *makelist* (*nextstm*) ; *S.nextlist* := “”;
emit (‘*goto* _’) }

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

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

✧ 拉链与代码回填

- 补充关于赋值语句及算术表达式的翻译模式

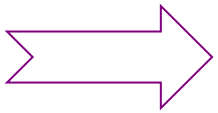
  类似于第31页

  （参见本讲课堂教案）

✧ GOTO 语句的语法制导翻译（选讲）

– 拉链回填技术示例

```
      ……
(10)  goto  L
      ……
(20)  goto  L
      ……
(30)  goto  L
      ……
(40)  L:
      ……
(50)  goto  L
      ……
```

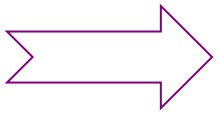


```
      ……
(100)  goto  0
      ……
(200)  goto  100
      ……
(300)  goto  200
      ……
(400)  L:
```

✧ GOTO 语句的语法制导翻译（选讲）

– 拉链回填技术示例

```
      ……                          ……
(10)  goto  L                (100)  goto  400
      ……                          ……
(20)  goto  L                (200)  goto  400
      ……                          ……
(30)  goto  L                (300)  goto  400
      ……                          ……
(40)  L:                     (400)  L:
      ……                          ……
(50)  goto  L                (500)  goto  400
      ……                          ……
```



✧ GOTO 语句的语法制导翻译（选讲）

– 利用标号的符号表项维护拉链

若采用类似 PL0 的符号表结构，可以设计标号表项包括如下域：

name，*kind*，*level* 等，与其它类别的符号一样
defined：表示该标号的说明是否已处理过
add：该标号的说明处理之前用于拉链，处理过后表示该标号的说明翻译后所指向的*TAC* 语句位置

– 语义函数/过程

setlbdefined (id.*name*,*x*)，*getlbdefined* (id.*name*)
setlbadd (id.*name*,*x*)，*getlbadd* (id.*name*)
分别表示设置和获取标号的 *defined*、*add* 值
backpatch (nextstm):
沿拉链反向将所有 *goto* 语句的目标返填为*nextstm*

✧ GOTO 语句的语法制导翻译 （选讲）

– 标号说明和 GOTO 语句的翻译模式

$S \rightarrow$ id : S

{ *p := lookup (*id*.name); if (p=nil) then enter(*id*.name) ;*
*setlbdefined(*id*.name,1);*
*backpatch(nextstm) ; setlbadd(*id*.name, nextstm)*
}

$S \rightarrow$ *goto* id

{ *p := lookup (*id*.name);*
*if (p=nil) then { enter(*id*.name) ;*
*setlbdefined(*id*.name,0) ;*
*setlbadd(*id*.name,0);*
emit('goto', 0)};
*else emit('goto', getlbadd(*id*.name))*
*if getlbdefined (id.name)=0 then setlbadd(*id*.name, nextstm-1)*
}

✧ 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 *p*（*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  p , 2                     // 过程调用语句
```

✧ 过程调用的语法制导翻译

– 简单过程调用的翻译模式

A.n：参数个数
A.arglist：实参地址的列表
makelist：创建实参地址结点
append：在实参表中添加结点

$S \rightarrow$ call id (*A*)
{ *S*.code := *A*.code;
for *A.arglist* 中的每一项 *p* do
S.code := *S.code* || *gen*('param' *p*);
S.code := *S.code* || *gen* ('call' id.*place* , *A.n*) }

$A \rightarrow A_1$, *E*
{ *A.n* := A_1.*n* +1;
A.arglist := *append*(A_1.*arglist* ,*makelist*(*E.place*));
A.code := A_1.*code* || *E.code* }

$A \rightarrow \varepsilon$
{ *A.n* := 0; *A.arglist* := ""; *A.code* := ""}

课后作业

参见网络学堂公告：“第四次书面作业”

That's all for today.

Thank You