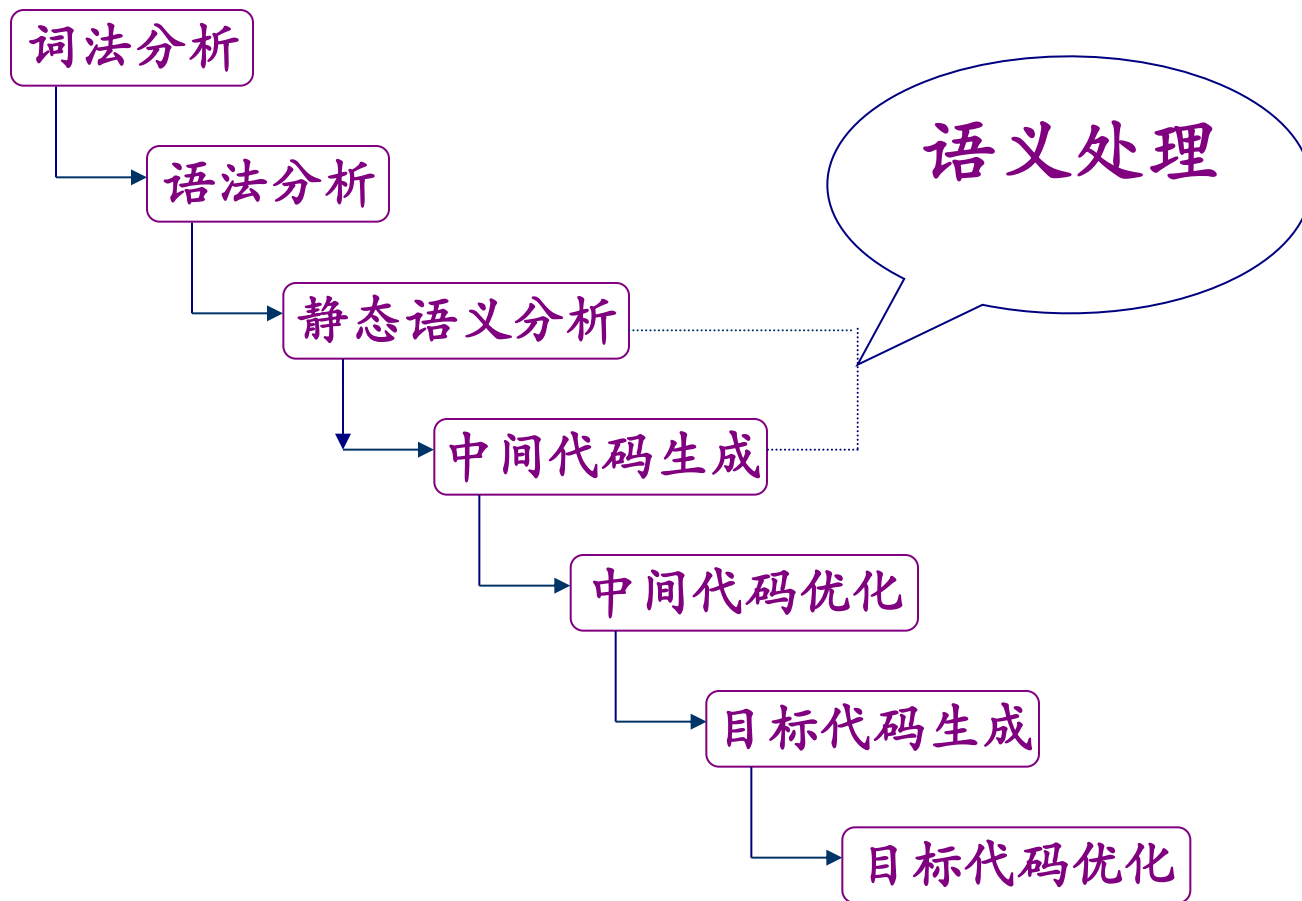


◇ 静态语义分析与中间代码生成

◇ 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四讲专门讨论)

◇ 借助简单语言进行核心内容的讲解

— 一个简单语言（与 MiniDecaf 语言相似）

• 抽象语法

$$P \rightarrow F P \mid D P \mid \varepsilon$$
$$F \rightarrow \text{int } \underline{\text{id}} (L) S$$
$$L \rightarrow L , \text{int } \underline{\text{id}} \mid \varepsilon$$
$$D \rightarrow T \underline{\text{id}} \mid T \underline{\text{id}} = E$$
$$T \rightarrow \text{int} \mid T [\underline{\text{int}}]$$
$$S \rightarrow \text{return } E \mid E \mid \text{if} (E) S S \mid \text{for} (E ? ; E ? ; E ?) S \mid \text{for} (D ; E ? ; E ?) S \\ \mid \text{while} (E) S \mid \text{do } S \text{ while} (E) \mid \text{break} \mid \text{continue} \mid D \mid \{ S \} \mid S S \mid \varepsilon$$
$$E \rightarrow \underline{\text{int}} \mid \underline{\text{id}} \mid \underline{\text{uop}} E \mid E \underline{\text{bop}} E \mid \underline{\text{top}} (E E E) \mid E [E] \mid \underline{\text{id}} (A)$$
$$A \rightarrow A , E \mid \varepsilon$$

◇ 借助翻译模式对实现算法进行规范描述

— 例：借助翻译模式描述类型检查的实现算法

- 翻译模式中对应于 while 语句类型检查的产生式

$$S \rightarrow \text{while} (E) \{ S_1.inloop := 1 \} S_1$$
$$\{ S.type := \text{if } E.type = \text{int} \text{ then } S_1.type \text{ else type_error} \}$$

- 直接对应到如下类型检查程序（伪代码）

```
type CheckStm ( int inloop, stm S )
{
    switch ( S ) {
        .....
        case “ while ( E ) S1 ” :
            type_of_E := CheckExp ( E ) ;
            S1_inloop := 1 ;
            type_of_S1 := CheckStm ( S1_inloop , S1 ) ;
            type_of_S := if type_of_E = int then type_of_S1 else type_error ;
            break;
        default :
            type_of_S := type_error ;
            break;
    }
    return type_of_S ;
}
```

静态语义分析与中间代码生成



清华大学

《编译原理》

◇ 静态语义分析（或语义分析）

◇ 中间代码生成

◇ 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)

◇ 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* 语句必须有合法的循环语句包围它)
- 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
-

◇ 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（选讲）

— 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（选讲）

— 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（选讲）

— 类型系统示例（面向类 MiniDecaf 小语言）

- 类型表达式

基本类型表达式: *int*

数组类型表达式: *array(τ)* 其中, τ 是基本数据类型表达式或数组类型表达式。*array(τ)* 表示元素类型是 τ 的数组类型。

函数类型表达式: *fun(n)* 其中, $n \geq 0$, 表示函数的参数个数。在类 MiniDecaf 小语言中, 函数的参数和返回值都只能为 *int*。

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（选讲）

— 类型系统示例（面向类 MiniDecaf 小语言）

- 类型环境

类型环境是一个从程序的部分标识符集到类型表达式集合的函数，用于记录一些标识符在某个上下文中被赋予的类型表达式

针对类 MiniDecaf 的小语言，类型环境的值域仅包含 *int*、*array(τ)* 和 *fun(n)* 三种类型表达式

◇ 类型系统简介（选讲）

— 类型系统示例（面向类 MiniDecaf 小语言）

• 类型环境（续）

记号：设类型环境为 Γ ，用 $[\underline{id} \mapsto t]$ 表示对类型环境的更新，这样 $\Gamma[\underline{id} \mapsto t]$ 就表示一个以 $\text{dom}(\Gamma) \cup \{\underline{id}\}$ 为定义域的类型环境（ \underline{id} 为一个变量名， t 为一个类型表达式），其含义为：对于任意标识符 $x \in \text{dom}(\Gamma) \cup \{\underline{id}\}$ ，若 $\underline{id} = x$ ，则 $\Gamma[\underline{id} \mapsto t](x) = t$ ；否则有 $\Gamma[\underline{id} \mapsto t](x) = \Gamma(x)$

另外，定义一个特殊函数 $\text{Update}(\Gamma, s)$ ，其中 Γ 是一个类型环境， s 是一个语句序列， Update 函数返回的结果是对 Γ 更新了 s 最外层作用域的所有声明后所得的类型环境，也就是说，假设 s 最外层作用域声明了类型为 t_1 的变量 \underline{id}_1 、类型为 t_2 的变量 \underline{id}_2 ……类型为 t_n 的变量 \underline{id}_n ，则 $\text{Update}(\Gamma, s) = \Gamma[\underline{id}_1 \mapsto t_1][\underline{id}_2 \mapsto t_2] \dots [\underline{id}_n \mapsto t_n]$

◇ 类型系统简介（选讲）

— 类型系统示例（面向类 MiniDecaf 小语言）

- 类型规则

引入下列断言形式 (judgements) :

$$\vdash t: A$$
$$\Gamma \vdash e: A$$
$$\Gamma \vdash_l s: A$$

这里, t 代表类型声明, e 代表除了类型声明和语句序列之外的程序单元, s 代表一个语句序列, A 代表一个类型表达式, $e: A$ 读作“ e 的类型为 A ”; 为方便, 我们引入了一个元变量 l , 用来表明所考虑的语句序列是否在一个循环体中, 有两种可能的取值: in (在某个循环体中) 和 out (不在任何一个循环体中)

◇ 类型系统简介（选讲）

— 类型系统示例（面向类 MiniDecaf 小语言）

- 类型规则（针对部分表达式）

$$\frac{}{\Gamma \vdash \text{int} : \text{int}} \text{(E-int)}$$

$$\frac{\Gamma(\text{id}) = \tau, \text{ where } \tau \text{ is } \text{int}, \text{ array or function type}}{\Gamma \vdash \text{id} : \tau} \text{(E-id)}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ \underline{bop} } e_2 : \text{int}} \text{(E-bop)} \quad \frac{\Gamma \vdash e_1 : \text{array}(\tau) \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : \tau} \text{(E-access)}$$

$$\frac{\Gamma \vdash \text{id} : \text{fun}(n) \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \dots \quad \Gamma \vdash e_n : \text{int}}{\Gamma \vdash \text{id}(e_1, e_2, \dots, e_n) : \text{int}}$$

◇ 类型系统简介（选讲）

— 类型系统示例（面向类 MiniDecaf 小语言）

• 类型规则（针对部分语句）

$$\frac{\Gamma \vdash e : int \quad \Gamma \vdash_l s_1 : ok \quad \Gamma \vdash_l s_2 : ok}{\Gamma \vdash_l \text{if}(e) s_1 s_2 : ok} \text{ (S-if)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash_l e : ok} \text{ (S-exp)}$$

$$\frac{\Gamma \vdash e : int \quad \Gamma \vdash_{in} s : ok}{\Gamma \vdash_l \text{while}(e) s : ok} \text{ (S-while)} \qquad \frac{}{\Gamma \vdash_{in} \text{break} : ok} \text{ (S-break)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : int \quad \Gamma \vdash e_3 : \tau_2 \quad \Gamma \vdash_{in} s : ok \quad e_1, e_2, \text{ or } e_3 \text{ may be empty}}{\Gamma \vdash_l \text{for}(e_1; e_2; e_3) s : ok} \text{ (S-fore)}$$

$$\frac{\Gamma \vdash_l s_1 : ok \quad \text{Update}(\Gamma, s_1) \vdash_l s_2 : ok}{\Gamma \vdash_l s_1 s_2 : ok} \text{ (S-S)} \qquad \frac{\Gamma \vdash e : int}{\Gamma \vdash_l \text{return } e : ok} \text{ (S-return)}$$

◇ 类型系统简介（选讲）

— 类型系统示例（面向类 MiniDecaf 小语言）

- 类型规则（针对类型声明或函数定义）

$$\frac{}{\vdash \text{int} : \text{int}} (\text{T-int}) \qquad \frac{\vdash t : \tau \quad \text{int} > 0}{\vdash t [\text{int}] : \text{array}(\tau)} (\text{E-array})$$

$$\frac{\Gamma[\underline{\text{id}} \mapsto \text{fun}(n)][\underline{\text{id}}_1 \mapsto \text{int}][\underline{\text{id}}_2 \mapsto \text{int}] \dots [\underline{\text{id}}_n \mapsto \text{int}] \vdash_{\text{out}} s : \text{ok}}{\Gamma \vdash \text{int } \underline{\text{id}} (\text{int } \underline{\text{id}}_1, \text{int } \underline{\text{id}}_2, \dots, \text{int } \underline{\text{id}}_n) s : \text{int}} (\text{F-def})$$

◇ 类型系统简介（选讲）

— 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

— 借助翻译模式描述类型检查的实现算法

- 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

◇ 借助翻译模式描述类型检查算法——举例

— 处理声明的翻译模式片段

$F \rightarrow \text{int } \underline{\text{id}} (L) S \quad \{ \text{addtype}(\underline{\text{id}}.\text{entry}, \text{fun}(L.\text{num})) ;$
 $\quad \quad \quad F.\text{type} := \text{if } S.\text{type} = \text{ok then ok else type_error} \} \quad // \text{规则(F-def)}$

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

$L \rightarrow \varepsilon \quad \{ L.\text{num} := 0 \}$

$D \rightarrow T \underline{\text{id}} \quad \{ \text{addtype}(\underline{\text{id}}.\text{entry}, T.\text{type}) ; D.\text{type} := \text{ok} \}$

$D \rightarrow T \underline{\text{id}} = E \quad \{ \text{addtype}(\underline{\text{id}}.\text{entry}, T.\text{type}) ;$
 $\quad \quad \quad D.\text{type} := \text{if } T.\text{type} = E.\text{type} \text{ then ok else type_error} \}$

$T \rightarrow \text{int} \quad \{ T.\text{type} := \text{int} \} \quad // \text{规则(T-int)}$

$T \rightarrow T_1 [\underline{\text{int}}] \quad \{ T.\text{type} := \text{if } \underline{\text{int}}.\text{lexval} > 0 \text{ then array}(T_1.\text{type})$
 $\quad \quad \quad \text{else type_error} \} \quad // \text{规则(T-array)}$

◇ 借助翻译模式描述类型检查算法——举例

— 处理表达式的翻译模式片段（续）

$E \rightarrow E_1 [E_2] \quad \{ E.type := \text{if } E_2.type = \text{int and } E_1.type = \text{array}(\tau) \\ \text{then } \tau \text{ else type_error} \} \quad // \text{规则(E-access)}$

$E \rightarrow \text{id} (A) \quad \{ E.type := \text{if lookup_type}(\text{id.name}) = \text{fun}(n) \\ \text{and } A.type = \text{ok and } A.num = n \\ \text{then int else type_error} \} \quad // \text{规则(E-call)}$

$A \rightarrow A_1 , E \quad \{ A.type := \text{if } A_1.type = \text{ok and } E.type := \text{int} \\ \text{then ok else type_error}; \\ A.num := A_1.num + 1 \}$

$A \rightarrow \varepsilon \quad \{ A.type := \text{ok}; A.num := 0 \}$

☆ 借助翻译模式描述类型检查算法——举例

— 处理语句的翻译模式片段

$$S \rightarrow \text{if} (E) S_1 S_2 \{ S.type := \text{if } E.type = \text{int and } S_1.type = \text{ok and } S_2.type = \text{ok then ok else type_error} \} \quad // \text{规则(S-if)}$$
$$S \rightarrow \text{while} (E) S_1 \{ S.type := \text{if } E.type = \text{int then } S_1.type \text{ else type_error} \} \quad // \text{规则(S-while)}$$
$$S \rightarrow \text{do } S_1 \text{ while} (E) \{ S.type := \text{if } E.type = \text{int then } S_1.type \text{ else type_error} \} \quad // \text{规则(S-do)}$$
$$S \rightarrow E \{ S.type := \text{if } E.type = \tau \text{ then ok else type_error} \} \quad // \text{规则(S-exp)}$$
$$S \rightarrow \text{return } E \{ S.type := \text{if } E.type = \text{int then ok else type_error} \} \quad // \text{规则(S-return)}$$
$$S \rightarrow D \{ S.type := D.type \} \quad // \text{规则 (S-D)}$$
$$S \rightarrow \{ S_1 \} \{ S.type := S_1.type \} \quad // \text{规则 (S-{\})}$$

◇ 借助翻译模式描述类型检查算法——举例

— 处理语句的翻译模式片段（续）

$S \rightarrow S_1 S_2 \{ S.type := \text{if } S_1.type = \text{ok and } S_2.type = \text{ok then ok} \\ \text{else type_error} \} \text{ // 规则(S-S)}$

$S \rightarrow \text{break} \{ S.type := \text{ok} \} \text{ // 规则(S-break, 未检查是否在循环体内)}$

$S \rightarrow \text{continue} \{ S.type := \text{ok} \} \text{ // 规则(S-continue, 未检查是否在循环体内)}$

$S \rightarrow \text{for} (R_1 ; R_2 ; R_3) S_1 \{ S.type := \text{if } R_2.type = \text{int and } R_1.type = \tau \\ \text{and } R_3.type = \tau \text{ then } S_1.type \text{ else type_error} \} \text{ // 规则(S-fore)}$

$S \rightarrow \text{for} (D ; R_1 ; R_2) S_1 \{ S.type := \text{if } D.type = \text{ok and } R_1.type = \text{int} \\ \text{and } R_2.type = \tau \text{ then } S_1.type \text{ else type_error} \} \\ \text{// 规则(S-ford, S-fordi)}$

$R \rightarrow E \quad \{ R.type := E.type \}$

$R \rightarrow \varepsilon \quad \{ R.type := \text{int} \}$

◇ 借助翻译模式描述类型检查算法——举例

— 处理程序外层定义的翻译模式片段

$$P \rightarrow F P_1 \{ P.type := \text{if } F.type = \text{ok and } P_1.type = \text{ok} \\ \text{then ok} \\ \text{else type_error} \} \text{ // 规则(P-F)}$$
$$P \rightarrow D P_1 \{ P.type := \text{if } D.type = \text{ok and } P_1.type = \text{ok} \\ \text{then ok} \\ \text{else type_error} \} \text{ // 规则(P-D)}$$
$$P \rightarrow \varepsilon \quad \{ P.type := \text{ok} \} \quad \text{// 规则(P-eps)}$$

☆ 语法制导的类型检查程序——举例

– 增加: *break* 和 *continue* 只能在循环体内部

$F \rightarrow \text{int } \underline{\text{id}} (L) \{ S.inloop := 0 \} S \{ \text{addtype}(\underline{\text{id}}.entry, \text{fun}(L.num)) ;$
 $F.type := \text{if } S.type = \text{ok} \text{ then ok else type_error} \}$ // 规则(F-def)

$S \rightarrow \text{if } (E) \{ S_1.inloop := S.inloop \} S_1 \{ S_2.inloop := S.inloop \} S_2$
 $\{ S.type := \text{if } E.type = \text{int} \text{ and } S_1.type = \text{ok} \text{ and } S_2.type = \text{ok}$
 $\text{then ok else type_error} \}$ // 规则(S-if)

$S \rightarrow \text{while } (E) \{ S_1.inloop := 1 \} S_1 \{ S.type := \text{if } E.type = \text{int} \text{ then}$
 $S_1.type \text{ else type_error} \}$ // 规则(S-while)

$S \rightarrow \text{do } \{ S_1.inloop := 1 \} S_1 \text{ while } (E) \{ S.type := \text{if } E.type = \text{int}$
 $\text{then } S_1.type \text{ else type_error} \}$ // 规则(S-do)

$S \rightarrow \{ S_1.inloop := S.inloop \} S_1 \{ S_2.inloop := S.inloop \} S_2$
 $\{ S.type := \text{if } S_1.type = \text{ok} \text{ and } S_2.type = \text{ok} \text{ then ok}$
 $\text{else type_error} \}$ // 规则(S-S)

☆ 语法制导的类型检查程序——举例

— 增加: *break* 和 *continue* 只能在循环体内部 (续)

$$S \rightarrow \text{'{' } \{ S_1.inloop := S.inloop \} S_1 \text{'}} \{ S.type := S_1.type \}$$
$$S \rightarrow \text{for (} R_1 ; R_2 ; R_3 \text{) } \{ S_1.inloop := 1 \} S_1$$
$$\{ S.type := \text{if } R_2.type = \text{int and } R_1.type = \tau \text{ and } R_3.type = \tau$$
$$\text{then } S_1.type \text{ else type_error} \} \quad // \text{ 规则(S-fore)}$$
$$S \rightarrow \text{for (} D ; R_1 ; R_2 \text{) } \{ S_1.inloop := 1 \} S_1$$
$$\{ S.type := \text{if } D.type = \text{ok and } R_1.type = \text{int and } R_2.type = \tau$$
$$\text{then } S_1.type \text{ else type_error} \} \quad // \text{ 规则(S-ford,S-fordi)}$$
$$S \rightarrow \text{break} \quad \{ S.type := \text{if } S.inloop = 1 \text{ then ok else type_error} \}$$
$$// \text{ 规则(S-break)}$$
$$S \rightarrow \text{continue} \quad \{ S.type := \text{if } S.inloop = 1 \text{ then ok else type_error} \}$$
$$// \text{ 规则(S-break)}$$

◇ 作用域分析

- 静态作用域
 - 通过符号表实现
(参见第四讲)
- 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

- 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 *Pascal* 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)

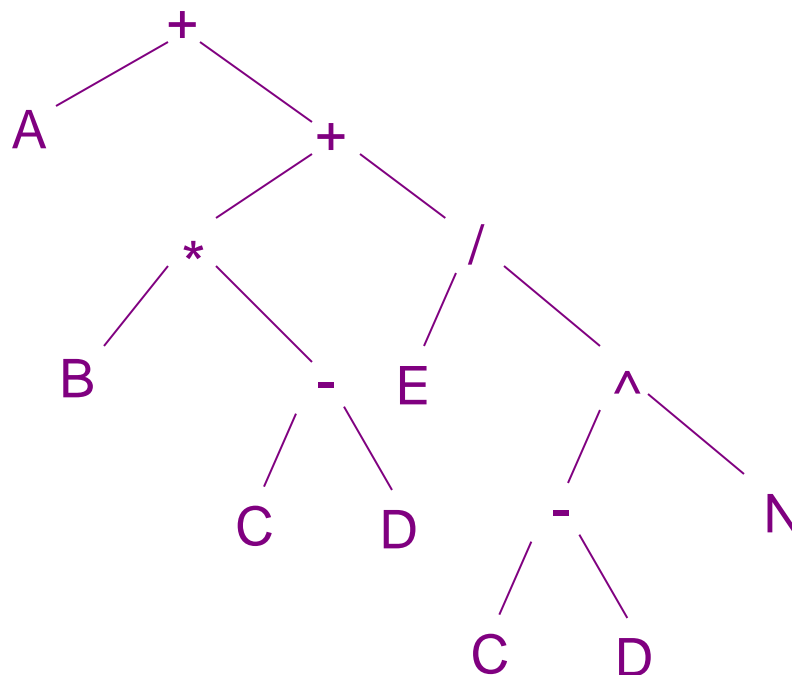
$T6 := E / T5$

(7) (+ T3 T6 T7)

$T7 := T3 + T6$

◇ 中间代码举例

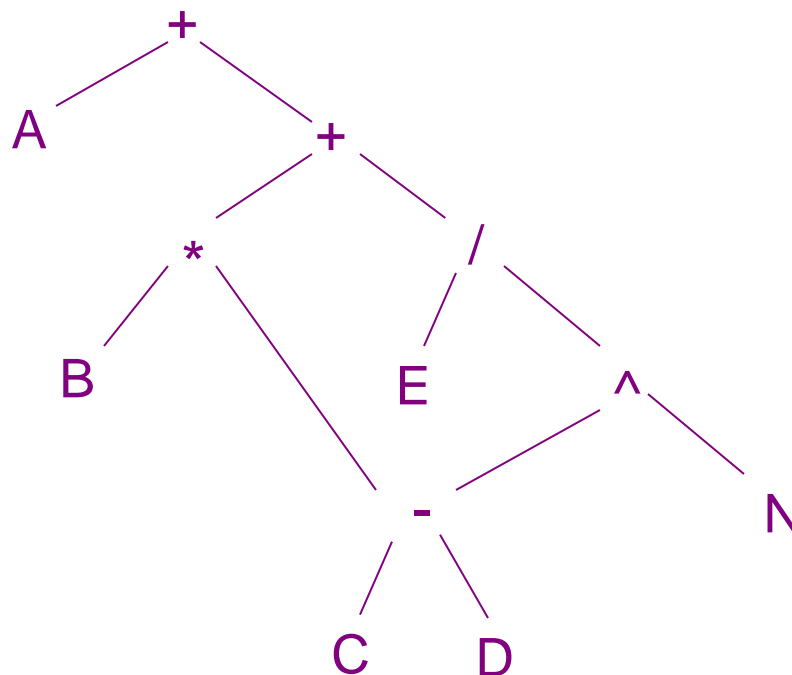
- 算术表达式 $A + B * (C - D) + E / (C - D)^N$
 - AST (抽象语法树) 表示



◇ 中间代码举例

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

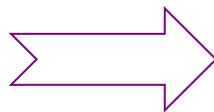
- DAG (Directed Acyclic Graph, 有向无圈图, 改进型 AST)



◇ 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

— 语法制导的方法 (如yacc)

- 例：生成抽象语法树

mknnode: 构造内部结点
Mkleaf: 构造叶子结点

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

✧ 三地址码 TAC

- 顺序的语句序列 其语句一般具有如下形式

$$x := y \text{ op } z$$

(op 为操作符, y 和 z 为操作数, x 为结果)

◇ 课程后续部分用到的 TAC 语句类型

- 赋值语句 $x := y \text{ op } z$ (op 代表二元算术/逻辑运算)
- 赋值语句 $x := \text{op } y$ (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 $\text{goto } L$ (无条件跳转至标号 L)
- 条件跳转语句 $\text{if } x \text{ rop } y \text{ goto } L$ (rop 代表关系运算)
- 标号语句 $L:$ (定义标号 L)
- 参数语句 $\text{param } x$
- 函数调用语句 $\text{call } p$
- 函数调用语句序列 $\text{param } x_1 \dots \text{param } x_n \text{ call } p$
- 函数返回语句 $\text{return } c$ (结束函数并把 c 的值作为返回值返回)
- LOAD 语句 $x := y[i]$ (将 y 的存储位置起第 i 个单元的值赋给 x)
- STORE 语句 $x[i] := y$ (将 y 的值保存到 x 的存储位置起第 i 个单元)
- ALLOC 语句 $x := \text{alloc } y$ (分配 y 个字节内存, 起始位置赋给 x)
- MEMO 语句 memo 'XXX' (栈帧中形参存储信息)

◇ 声明语句的翻译

— 语义属性

$\underline{id.name}$: \underline{id} 的词法名字 (符号表中的名字)

$T.width, D.width, \dots$: 数据宽度 (字节数)

$P.offset, S.offset, \dots$: 不同存储区中变量的偏移地址

— 语义函数/过程

$enter(\underline{id.name}, o)$: 将符号表中 $\underline{id.name}$ 所对应表项的 $offset$ 域置为 o , 具体哪个符号表与上下文相关

☆ 声明语句的翻译

– 全局声明语句相关的翻译模式片段

$$Prog \rightarrow \{ P.offset := 0 \} P$$
$$P \rightarrow \{ F.offset := 0 \} F \{ P_1.offset := P.offset \} P_1 \{ P.width := P_1.width \}$$
$$P \rightarrow \{ D.offset := P.offset \} D \{ P_1.offset := P.offset + D.width \} P_1 \\ \{ P.width := D.width + P_1.width \}$$
$$P \rightarrow \varepsilon \quad \{ P.width := 0 \}$$
$$D \rightarrow T \underline{id} \quad \{ enter(\underline{id}.name, D.offset) ; D.width := T.width \}$$
$$D \rightarrow T \underline{id} = E \quad \{ enter(\underline{id}.name, D.offset) ; D.width := T.width \}$$
$$T \rightarrow int \quad \{ T.width := 4 \}$$
$$T \rightarrow T_1 [\underline{int}] \quad \{ T.width := \underline{int}.lexval \times T_1.width \}$$

◇ 声明语句的翻译

— 局部声明语句相关的翻译模式片段

$F \rightarrow \text{int } \underline{\text{id}} \{ L.\text{offset} := F.\text{offset} \} L \{ S.\text{offset} := F.\text{offset} + L.\text{width} \} S$
 $L \rightarrow \{ L_1.\text{offset} := L.\text{offset} \} L_1, \text{int } \underline{\text{id}}$
 $\quad \{ \text{enter}(\underline{\text{id}}.\text{name}, L.\text{offset} + L_1.\text{width}); L.\text{width} := L_1.\text{width} + 4 \}$
 $L \rightarrow \varepsilon \quad \{ L.\text{width} := 0 \}$
 $S \rightarrow \text{if} (E) \{ S_1.\text{offset} := S.\text{offset} \} S_1$
 $\quad \{ S_2.\text{offset} := S.\text{offset} \} S_2 \{ S.\text{width} := 0 \}$
 $S \rightarrow \text{while} (E) \{ S_1.\text{offset} := S.\text{offset} \} S_1 \{ S.\text{width} := 0 \}$
 $S \rightarrow \text{do} \{ S_1.\text{offset} := S.\text{offset} \} S_1 \text{while} (E) \{ S.\text{width} := 0 \}$
 $S \rightarrow \text{for} (R_1; R_2; R_3) \{ S_1.\text{offset} := S.\text{offset} \} S_1 \{ S.\text{width} := 0 \}$
 $S \rightarrow \text{for} (D; R_1; R_2) \{ S_1.\text{offset} := S.\text{offset} + D.\text{width} \} S_1 \{ S.\text{width} := 0 \}$
 $S \rightarrow D \{ S.\text{width} := D.\text{width} \}$
 $S \rightarrow \{ \{ S_1.\text{offset} := S.\text{offset} \} S_1 \} \{ S.\text{width} := 0 \}$
 $S \rightarrow \{ S_1.\text{offset} := S.\text{offset} \} S_1 \{ S_2.\text{offset} := S.\text{offset} + S_1.\text{width} \} S_2$
 $\quad \{ S.\text{width} := S_1.\text{width} + S_2.\text{width} \}$

◇ 数组说明和数组元素引用的翻译

— 数组说明和数组元素引用（类型检查和语义信息收集）

将前面有关数组声明和数组元素引用相关处理的翻译模式片段进行合并，得到如下翻译模式：

$$D \rightarrow T \ \underline{id} \quad \{ \text{enter}(\underline{id}.name, T.type, D.offset) ; D.type := ok ; \\ D.width := T.width \}$$
$$D \rightarrow T \ \underline{id} = E \quad \{ \text{enter}(\underline{id}.name, T.type, D.offset) ; \\ D.type := \text{if } T.type = E.type \text{ then } ok \text{ else } type_error ; \\ D.width := T.width \}$$
$$T \rightarrow \text{int} \quad \{ T.type := \text{int} ; T.width := 4 \}$$
$$T \rightarrow T_1 [\underline{int}] \quad \{ T.type := \text{if } \underline{int}.lexval > 0 \text{ then } array(T_1.type) \\ \text{else } type_error ; T.width := \underline{int}.lexval \times T_1.width \}$$
$$E \rightarrow E_1 [E_2] \quad \{ E.type := \text{if } E_2.type = \text{int and } E_1.type = array(\tau) \\ \text{then } \tau \text{ else } type_error \}$$

◇ 数组说明和数组元素引用的翻译

— 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 $A[l_1:u_1, l_2:u_2, \dots, l_n:u_n]$ ，可以在符号表中建立如下形式的内情向量：

 $l_1 \quad u_1$ $l_2 \quad u_2$ $\dots \quad \dots$ $l_n \quad u_n$ $type \quad a$ $n \quad C$ 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 $A[l_1:u_1, l_2:u_2, \dots, l_n:u_n]$ ，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 $A[i_1, i_2, \dots, i_n]$ 的地址 D 可以如下计算：

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

重新整理后得： $D = a - C + V$ ，其中

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

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

(这里的 C 即为前页内情向量中的 C)

◇ 数组说明和数组元素引用的翻译

— 数组说明和数组元素引用 (TAC 中间代码生成)

$$D \rightarrow T \ \underline{id} \quad \{ \begin{array}{l} x := newtemp; \ y := newtemp; \\ \text{if } T.type = int \text{ then } D.code := \varepsilon \text{ else} \\ D.code := gen (y := D.offset) \parallel gen (x := 'alloc' \ T.width) \parallel \\ gen (y['0'] := x) \parallel gen (y['1'] := T.width) \end{array} \}$$
$$D \rightarrow T \ \underline{id} = E \quad \{ \begin{array}{l} x := newtemp; \ y := newtemp; \\ \text{if } T.type = int \text{ then } D.code := E.code \parallel gen(\underline{id}.place := E.place) \\ \text{else } D.code := gen (E.code \parallel y := D.offset) \parallel \\ gen (y['0'] := E.place) \parallel gen (y['1'] := T.width) \end{array} \}$$
$$E \rightarrow E_1 [E_2] \quad \{ \begin{array}{l} E.place := newtemp; \\ E.code := E_1.code \parallel E_2.code \parallel \\ gen (E.place := E_1.place [' E_2.place ']) \end{array} \}$$

◇ 表达式的翻译

— 语义属性

$\underline{id.place}$: \underline{id} 对应的存储位置

$\underline{E.place}$: 用来存放 E 的值的存储位置

$\underline{E.code}$, $\underline{A.code}$: 相应的 TAC 语句序列

— 语义函数/过程

\underline{gen} : 生成一条 TAC 语句

$\underline{newtemp}$: 新建一个未用过的名字, 并返回存储位置

$\underline{newlabel}$: 返回一个新的语句标号

$\underline{\parallel}$ 是 TAC 语句序列之间的链接运算

$\underline{A.arglist}$ 代表实参地址的列表; 语义函数 $\underline{makelist}$ 表示创建一个实参地址的结点; 语义函数 \underline{append} 表示在已有实参地址列表中添加一个结点

◇ 表达式的翻译

— 翻译模式

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

$E \rightarrow \underline{\text{id}}$ { $E.place := \underline{\text{id}}.place$; $E.code := ""$ }

$E \rightarrow \underline{\text{uop}} E_1$ { $E.place := \text{newtemp}$;
 $E.code := E_1.code \parallel \text{gen}(E.place := \underline{\text{uop}}.op E_1.place)$ }

$E \rightarrow E_1 \underline{\text{bop}} E_2$ { $E.place := \text{newtemp}$; $E.code := E_1.code \parallel E_2.code$
 $\parallel \text{gen}(E.place := E_1.place \underline{\text{bop}}.op E_2.place)$ }

$E \rightarrow \underline{\text{top}} (E_1 E_2 E_3)$ { $E.place := \text{newtemp}$; $L := \text{newlabel}$;
 $M := \text{newlabel}$; $E.code := E_1.code \parallel \text{gen}('if' E_1.place <= '0' 'goto' L)$
 $\parallel E_2.code \parallel \text{gen}(E.place := E_2.place) \parallel \text{gen}('goto' M)$
 $\parallel \text{gen}(L ':') \parallel E_3.code \parallel \text{gen}(E.place := E_3.place) \parallel \text{gen}(M ':')$ }

◇ 表达式的翻译

— 翻译模式 (续)

$$E \rightarrow \underline{\text{id}} (A) \quad \{ E.\text{code} := A.\text{code};$$

for $A.\text{arglist}$ 中的每一项 d do

$$E.\text{code} := E.\text{code} \parallel \text{gen}(\text{'param' } d);$$
$$E.\text{code} := E.\text{code} \parallel \text{gen}(\text{'call' } \underline{\text{id}}.\text{place}) \}$$
$$A \rightarrow A_1 , E \quad \{ A.n := A_1.n + 1;$$

$A.\text{arglist} := \text{append}(A_1.\text{arglist}, \text{makelist}(E.\text{place}));$ $\}$

$$A.\text{code} := A_1.\text{code} \parallel E.\text{code} \}$$
$$A \rightarrow \varepsilon \quad \{ A.n := 0; A.\text{arglist} := ""; A.\text{code} := "" \}$$
$$E \rightarrow \underline{\text{id}} = E_1 \quad \{ E.\text{code} := E_1.\text{code} \parallel \text{gen}(\underline{\text{id}}.\text{place} \text{' := ' } E_1.\text{place}) \}$$
$$E \rightarrow E_1 [E_2] = E_3 \quad \{ E.\text{code} := E_1.\text{code} \parallel E_2.\text{code} \parallel E_3.\text{code} \parallel$$

$\text{gen}(E_1.\text{place} \text{' [' } E_2.\text{place} \text{']' ' := ' } E_3.\text{place}) \}$

◇ 布尔表达式的翻译

— 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

◇ 布尔表达式的翻译

— 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

$E \rightarrow E_1 \vee E_2$ { $E.place := newtemp$; $E.code := E_1.code \parallel E_2.code$
 $\parallel gen(E.place := 'E_1.place \text{ or } E_2.place)$ }

$E \rightarrow E_1 \wedge E_2$ { $E.place := newtemp$; $E.code := E_1.code \parallel E_2.code$
 $\parallel gen(E.place := 'E_1.place \text{ and } E_2.place)$ }

$E \rightarrow \neg E_1$ { $E.place := newtemp$; $E.code := E_1.code \parallel$
 $gen(E.place := 'not E_1.place)$ }

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

$E \rightarrow \underline{id}_1 \text{ rop } \underline{id}_2$ { $E.place := newtemp$; $E.code := gen('if \underline{id}_1.place$
 $\text{ rop.op } \underline{id}_2.place \text{ goto } nextstat+3) \parallel$
 $gen(E.place := '0') \parallel gen('goto' nextstat+2)$
 $\parallel gen(E.place := '1')$ }

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

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

◇ 布尔表达式的翻译

— 通过控制流体现布尔表达式的语义

例：布尔表达式 $E = a < b \text{ or } c < d \text{ and } e < f$ 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  $E.false$ 
```

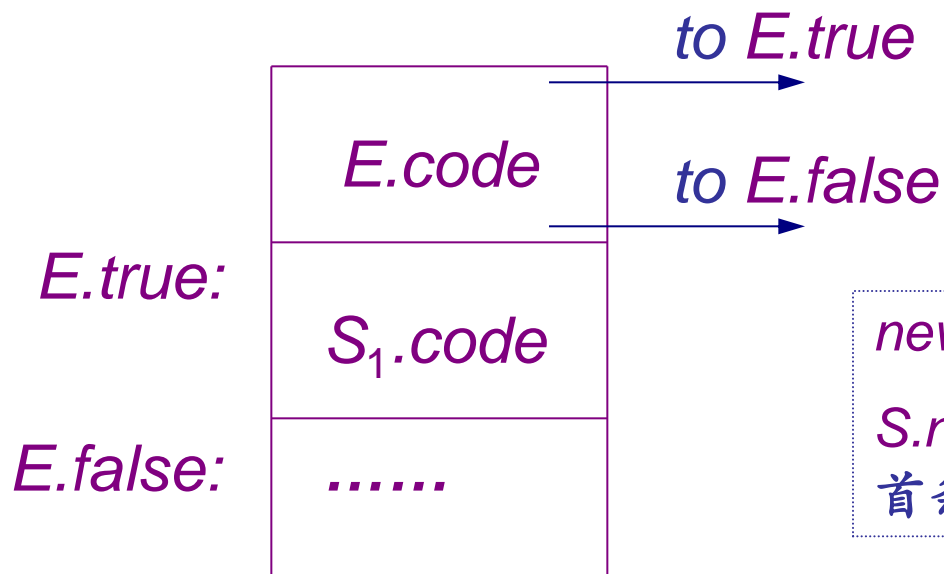
◇ 布尔表达式的翻译

— 翻译布尔表达式至短路代码 (L-翻译模式)

$$E \rightarrow \{ E_1.true := E.true; E_1.false := newlabel \} E_1 \vee$$
$$\{ E_2.true := E.true; E_2.false := E.false \} E_2$$
$$\{ E.code := E_1.code \parallel gen(E_1.false ':') \parallel E_2.code \}$$
$$E \rightarrow \{ E_1.false := E.false; E_1.true := newlabel \} E_1 \wedge$$
$$\{ E_2.false := E.false; E_2.true := E.true \} E_2$$
$$\{ E.code := E_1.code \parallel gen(E_1.true ':') \parallel E_2.code \}$$
$$E \rightarrow \neg \{ E_1.true := E.false; E_1.false := E.true \} E_1 \{ E.code := E_1.code \}$$
$$E \rightarrow (\{ E_1.true := E.true; E_1.false := E.false \} E_1) \{ E.code := E_1.code \}$$
$$E \rightarrow \underline{id_1} \ \underline{rop} \ \underline{id_2} \ \{ E.code := gen('if' \ \underline{id_1}.place \ \underline{rop}.op \ \underline{id_2}.place \ 'goto'$$
$$E.true) \parallel gen('goto' \ E.false) \}$$
$$E \rightarrow true \ \{ E.code := gen('goto' \ E.true) \}$$
$$E \rightarrow false \ \{ E.code := gen('goto' \ E.false) \}$$

◇ 条件语句的翻译

– if-then 语句 (L 翻译模式)

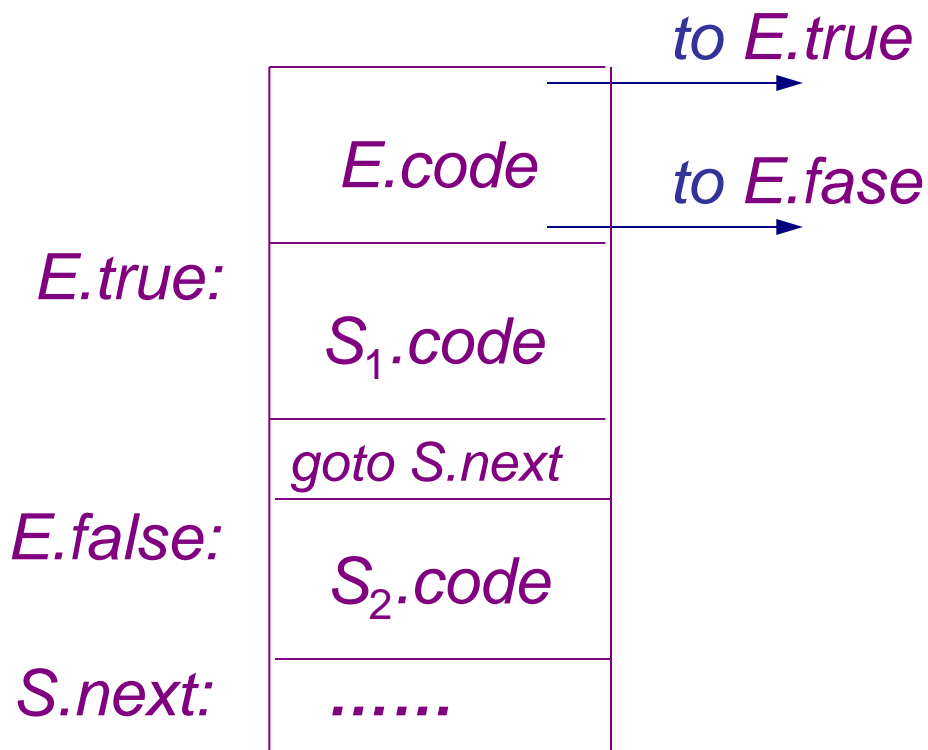
$$S \rightarrow \text{if } (\{ E.true := \text{newlabel}; E.false := S.next \} E)$$
$$\quad \{ S_1.next := S.next \} \quad S_1$$
$$\quad \{ S.code := E.code \parallel \text{gen}(E.true ':') \parallel S_1.code \}$$


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

◇ 条件语句的翻译

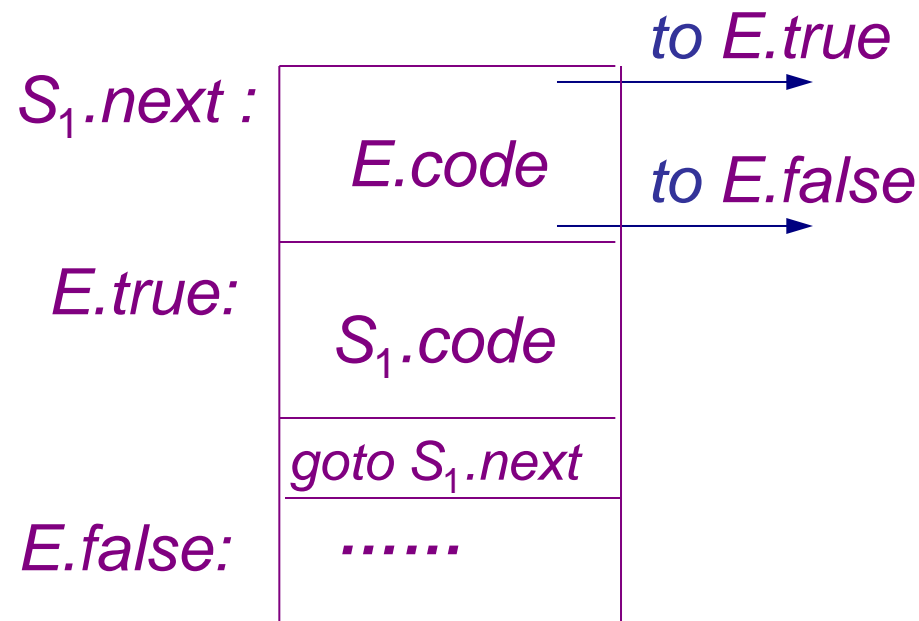
— if-then-else 语句 (*L* 翻译模式)



```
S → if ( { E.true := newlabel;  
          E.false := newlabel }  
        E )  
    { S1.next := S.next }   S1  
    { S2.next := S.next }   S2  
    { S.code := E.code ||  
      gen(E.true ':') ||  
      S1.code ||  
      gen('goto' S.next) ||  
      gen(E.false ':') ||  
      S2.code  
    }
```

◇ 循环语句的翻译

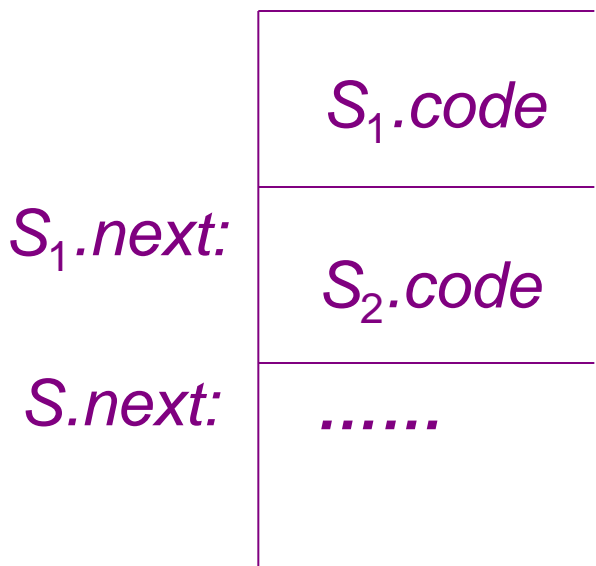
— while 语句 (L 翻译模式)



```
S → while (  
    {  $E.true := newlabel$ ;  
       $E.false := S.next$  }  
E)  
    {  $S_1.next := newlabel$  }  
S1  
    {  $S.code := gen(S_1.next ':')$   
      ||  $E.code$   
      ||  $gen(E.true ':')$   
      ||  $S_1.code$   
      ||  $gen('goto' S_1.next)$   
    }
```

◇ 复合语句的翻译

— 顺序复合语句 (L 翻译模式)



```
S → {  $S_1.next := newlabel$  }  $S_1$ 
      {  $S_2.next := S.next$  }  $S_2$ 
      {  $S.code := S_1.code$ 
        // gen( $S_1.next$  ':')
        //  $S_2.code$ 
      }
```


◇ 生成完整程序的TAC序列

— 翻译模式

$$Prog \rightarrow P \{ output (P.code) \}$$
$$P \rightarrow \{ F.begin := newlabel \} F P_1 \\ \{ P.code := gen(F.begin ':') \parallel F.code \parallel P_1.code \}$$
$$P \rightarrow D P_1 \{ P.code := D.code \parallel P_1.code \}$$
$$P \rightarrow \varepsilon \quad \{ P.code := \varepsilon \}$$
$$F \rightarrow int \underline{id} (L) S \{ addplace(\underline{id}.entry, F.begin) ; \\ S.code := gen('memo' "" L.code "") \parallel S.code \}$$
$$L \rightarrow L_1 , int \underline{id} \{ L.width := L_1.width + 4; \\ L.code := L_1.code \parallel gen(\underline{id}.name ':' L_1.width + 4) \}$$
$$L \rightarrow \varepsilon \quad \{ L.width := 0; L.code := \varepsilon \}$$

☆ 含 *break* 语句的翻译

— 翻译模式

$$F \rightarrow \text{int } \underline{\text{id}} (L) \{ S.\text{next} := \text{newlabel}; S.\text{break} := \text{newlabel} \} S$$
$$\{ S.\text{code} := \text{gen}(\text{'memo' ' ' } L.\text{code ' '}) \parallel S.\text{code} \parallel \text{gen}(S.\text{next} ':') \}$$
$$S \rightarrow \text{if} (\{ E.\text{true} := \text{newlabel}; E.\text{false} := S.\text{next} \} E)$$
$$\{ S_1.\text{next} := S.\text{next}; S_1.\text{break} := S.\text{break} \} S_1$$
$$\{ S.\text{code} := E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \}$$
$$S \rightarrow \text{if} (\{ E.\text{true} := \text{newlabel}; E.\text{false} := \text{newlabel} \} E)$$
$$\{ S_1.\text{next} := S.\text{next}; S_1.\text{break} := S.\text{break} \} S_1$$
$$\{ S_2.\text{next} := S.\text{next}; S_2.\text{break} := S.\text{break} \} S_2$$
$$\{ S.\text{code} := E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \parallel$$
$$\text{gen}(\text{'goto' } S.\text{next}) \parallel \text{gen}(E.\text{false} ':') \parallel S_2.\text{code} \}$$

☆ 含 *break* 语句的翻译

— 翻译模式 (续)

$$\begin{aligned} S \rightarrow \text{while} (\{ E.\text{true} := \text{newlabel}; E.\text{false} := S.\text{next} \} \quad E) \\ \{ S_1.\text{next} := \text{newlabel}; S_1.\text{break} := S.\text{next} \} \quad S_1 \\ \{ S.\text{code} := \text{gen}(S_1.\text{next} ':') \parallel \\ E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \parallel \\ \text{gen}(\text{'goto' } S_1.\text{next}) \} \end{aligned}$$
$$\begin{aligned} S \rightarrow \{ S_1.\text{next} := \text{newlabel}; S_1.\text{break} := S.\text{break} \} \quad S_1 ; \\ \{ S_2.\text{next} := S.\text{next}; S_2.\text{break} := S.\text{break} \} \quad S_2 \\ \{ S.\text{code} := S_1.\text{code} \parallel \text{gen}(S_1.\text{next} ':') \parallel S_2.\text{code} \} \end{aligned}$$
$$S \rightarrow \text{break} \quad \{ S.\text{code} := \text{gen}(\text{'goto' } S.\text{break}) \}$$

◇ 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

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

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

$E \rightarrow \neg E_1$ { 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模式（续）

$E \rightarrow (E_1)$ $\{ E.truelist := E_1.truelist ;$
 $E.falselist := E_1.falselist \}$

$E \rightarrow \underline{id}_1 \text{ rop } \underline{id}_2$ $\{ E.truelist := makelist (nextstm);$
 $E.falselist := makelist (nextstm+1);$
 $emit (\text{'if' } \underline{id}_1.place \text{ rop.op } \underline{id}_2.place \text{ 'goto _' });$
 $emit (\text{'goto _' }) \}$

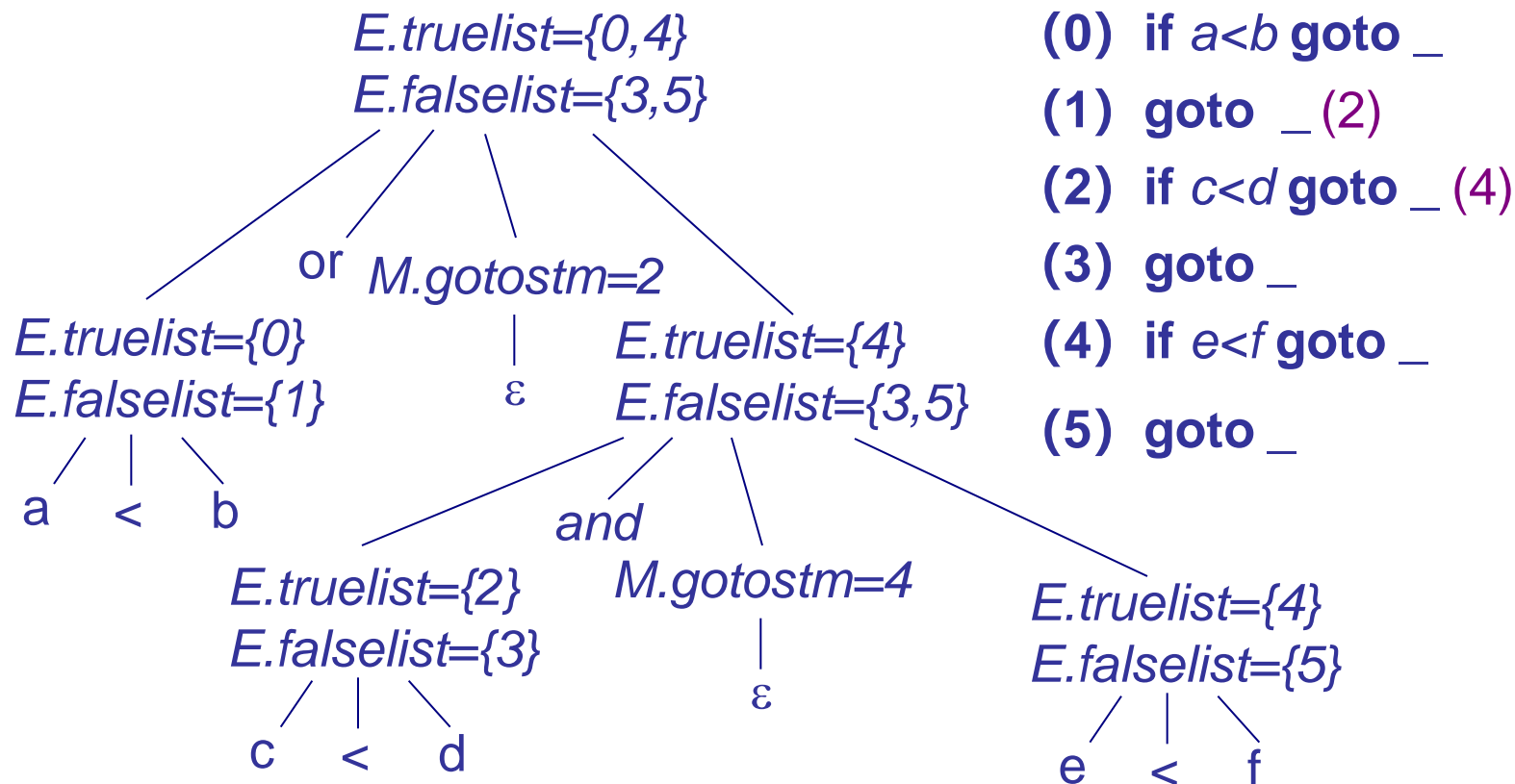
$E \rightarrow \text{true}$ $\{ E.truelist := makelist (nextstm);$
 $emit (\text{'goto _' }) \}$

$E \rightarrow \text{false}$ $\{ E.falselist := makelist (nextstm);$
 $emit (\text{'goto _' }) \}$

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

◇ 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



◇ 拉链与代码回填

— 处理条件语句的翻译模式

$$S \rightarrow \text{if } (E) M S_1$$
$$\{ \text{backpatch}(E.\text{truelist}, M.\text{gotostm}) ;$$
$$S.\text{nextlist} := \text{merge}(E.\text{falselist}, S_1.\text{nextlist}) \}$$
$$S \rightarrow \text{if } (E) M_1 S_1 N \text{ else } M_2 S_2$$
$$\{ \text{backpatch}(E.\text{truelist}, M_1.\text{gotostm}) ;$$
$$\text{backpatch}(E.\text{falselist}, M_2.\text{gotostm}) ;$$
$$S.\text{nextlist} := \text{merge}(S_1.\text{nextlist}, \text{merge}(N.\text{nextlist}, S_2.\text{nextlist})) \}$$
$$M \rightarrow \varepsilon$$
$$\{ M.\text{gotostm} := \text{nextstm} \}$$
$$N \rightarrow \varepsilon$$
$$\{ N.\text{nextlist} := \text{makelist}(\text{nextstm}); \text{emit}(\text{'goto _'}) \}$$

◇ 拉链与代码回填

— 处理循环、复合、块、表达式语句的翻译模式

$$\begin{aligned} S \rightarrow \text{while} (M_1 E) M_2 S_1 \\ \{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{gotostm}) ; \\ \text{backpatch}(E.\text{truelist}, M_2.\text{gotostm}) ; \\ S.\text{nextlist} := E.\text{falselist}; \\ \text{emit}(\text{'goto'}, M_1.\text{gotostm}) \} \end{aligned}$$
$$\begin{aligned} S \rightarrow S_1 M S_2 \\ \{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{gotostm}) ; \\ S.\text{nextlist} := S_2.\text{nextlist} \} \end{aligned}$$
$$S \rightarrow \{ S_1 \} \{ S.\text{nextlist} := S_1.\text{nextlist} \}$$
$$S \rightarrow B \{ S.\text{nextlist} := B.\text{nextlist} \}$$
$$M \rightarrow \varepsilon \{ M.\text{gotostm} := \text{nextstm} \}$$

◇ 拉链与代码回填

— 增加 *break* 语句后控制语句处理的翻译模式

$$F \rightarrow \text{int } \underline{\text{id}} (L) S M \quad \{ \text{backpatch}(S.\text{nextlist}, M.\text{gotostm}) ; \\ \text{backpatch}(S.\text{breaklist}, M.\text{gotostm}) \}$$
$$S \rightarrow \text{if} (E) M S_1 \quad \{ \text{backpatch}(E.\text{truelist}, M.\text{gotostm}) ; \\ S.\text{nextlist} := \text{merge}(E.\text{falselist}, S_1.\text{nextlist}) ; \\ S.\text{breaklist} := S_1.\text{breaklist} \}$$
$$S \rightarrow \text{if} (E) M_1 S_1 N \text{ else } M_2 S_2 \quad \{ \text{backpatch}(E.\text{truelist}, M_1.\text{gotostm}) ; \\ \text{backpatch}(E.\text{falselist}, M_2.\text{gotostm}) ; \\ S.\text{nextlist} := \text{merge}(S_1.\text{nextlist}, \text{merge}(N.\text{nextlist}, S_2.\text{nextlist}) ; \\ S.\text{breaklist} := \text{merge}(S_1.\text{breaklist}, S_2.\text{breaklist}) \}$$

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)

$$S \rightarrow \text{while} (M_1 E) M_2 S_1 \quad \{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{gotostm}); \\ \text{backpatch}(E.\text{truelist}, M_2.\text{gotostm}); \\ S.\text{nextlist} := \text{merge}(E.\text{falselist}, S_1.\text{breaklist}); \\ S.\text{breaklist} := \varepsilon; \text{emit}(\text{'goto'}, M_1.\text{gotostm}) \}$$
$$S \rightarrow \{ S_1 \} \quad \{ S.\text{nextlist} := S_1.\text{nextlist}; S.\text{breaklist} := S_1.\text{breaklist} \}$$
$$S \rightarrow B \quad \{ S.\text{nextlist} := B.\text{nextlist}; S.\text{breaklist} := \varepsilon \}$$
$$S \rightarrow S_1 M S_2 \quad \{ \text{backpatch}(S_1.\text{nextlist}, M.\text{gotostm}); S.\text{nextlist} := S_2.\text{nextlist}; \\ S.\text{breaklist} := \text{merge}(S_1.\text{breaklist}, S_2.\text{breaklist}) \}$$
$$S \rightarrow \text{break} \quad \{ S.\text{breaklist} := \text{makelist}(\text{nextstm}); S.\text{nextlist} := \varepsilon; \\ \text{emit}(\text{'goto_'}) \}$$
$$M \rightarrow \varepsilon \quad \{ M.\text{gotostm} := \text{nextstm} \}$$
$$N \rightarrow \varepsilon \quad \{ N.\text{nextlist} := \text{makelist}(\text{nextstm}); \text{emit}(\text{'goto_'}) \}$$

◇ 拉链与代码回填

— 调整赋值和算术表达式翻译模式的设计风格

$B \rightarrow \underline{id} = B_1 \quad \{ \text{emit}(\underline{id}.place \text{ ':=' } B_1.place) ; B.nextlist := \varepsilon \}$

$B \rightarrow \underline{int} \quad \{ B.place := newtemp; \text{emit}(B.place \text{ ':=' } \underline{int}.val) ; B.nextlist := \varepsilon \}$

$B \rightarrow \underline{id} \quad \{ B.place := \underline{id}.place ; B.nextlist := \varepsilon \}$

$B \rightarrow \underline{uop} B_1 \quad \{ B.place := newtemp; \\ \text{emit}(B.place \text{ ':=' } \underline{uop}.op B_1.place) ; B.nextlist := \varepsilon \}$

$B \rightarrow B_1 \underline{bop} B_2 \quad \{ B.place := newtemp; \\ \text{emit}(B.place \text{ ':=' } B_1.place \underline{bop}.op B_2.place) ; B.nextlist := \varepsilon \}$

$B \rightarrow \underline{top} (E M_1 B_1 N M_2 B_2) \quad \{ \text{backpatch}(E.truelist, M_1.gotostm) ; \\ \text{backpatch}(E.falselist, M_2.gotostm) ; B.nextlist := N.nextlist \}$

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

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

☆ GOTO 语句的语法制导翻译

— 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

— 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

— 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

— 标号说明和 GOTO 语句的翻译模式

$S \rightarrow \underline{id} : S$

```
{  $p := \text{lookup}(\underline{id}.name)$ ; if ( $p = \text{nil}$ ) then  $\text{enter}(\underline{id}.name)$  ;  
   $\text{setlbdefined}(\underline{id}.name, 1)$ ;  
   $\text{backpatch}(\text{nextstm})$  ;  $\text{setlbadd}(\underline{id}.name, \text{nextstm})$   
}
```

$S \rightarrow \text{goto } \underline{id}$

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

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