

# 第十讲 静态语义分析与中间代码生成

2021-11

前面（第九讲）介绍了语法制导的语义计算的基本原理，其重点内容是关于在实践中应用广泛的基于属性文法或翻译模式的方法。这一内容起着承上启下作用，从语法分析过渡到语义分析。首先，语法制导的语义计算是许多编译-编译工具如 Yacc 的基础，这些工具所自动生成的编译代码既包含语法分析功能，也包含语义计算功能。如，在 MiniDecaf 实验中，我们使用 Yacc 类工具实现语法分析和抽象语法树（AST）生成（简单的语义计算功能）。另一方面，在这一讲（第十讲）里，我们将对其应用范围加以扩展，不限于进行语法制导的语义计算，而是也将其用于规范描述抽象语法层面的工作，如基于抽象语法树（AST）的静态语义分析和中间代码生成。对于后者，下面我们用具体的例子予以说明，也可看作是本讲的导引。

MiniDecaf 实验中，我们有语句的（抽象）语法定义：

$$S \rightarrow \dots \mid \text{while} (E) S \mid \dots$$

其中包含  $\text{while} (E) S$  语句。

在 1.1.2 节，我们借助翻译模式描述类型检查的实现算法，其中  $\text{while} (E) S$  语句的类型检查对应于翻译模式中的如下产生式：

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

先不必去理解其确切含义，仅注意  $S$  含 *inloop* 和 *type* 两个属性（前者是继承属性，后者是综合属性）， $E$  含一个综合属性 *type*，根据前面第九讲，我们知道该产生式符合 L-翻译模式的限定，同样大家将会了解到 1.1.2 节翻译模式的其他产生式也都满足这一限定。在第九讲，我们学习过从 L-翻译模式可以直接对应到一份伪代码，可用作实现语义计算的递归下降翻译程序。这样，我们可以将描述语句（ $S$ ）类型检查算法的翻译模式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 other-statement-1 :
            /* code to check other-statement-1 */
            break;
        case other-statement-2 :
            /* code to check other-statement-2 */
            break;
        ...
        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 ;
}

```

值得注意的是，与第九讲语法制导的递归下降语义计算程序相比，上边这段伪代码不包含对源程序中某些终结符（如左、右括号等）的扫描，这些符号与抽象语法树节点无关。或者，我们可以将其称之为“抽象”语法制导的递归下降语义计算程序，相应的翻译模式也可以称之为面向抽象语法的翻译模式。

在 1.1.2 节之前，我们还给出了针对一个小语言（本学期 MiniDecaf 语言）的（半）形式化定义的类型系统，其中的类型规则可以作为类型检查算法的高层抽象规范。与形式化定义的类型规则相比，翻译模式距离代码实现更近。

类似地，在本讲第 2 节，我们同样借助这种面向抽象语法的翻译模式，来讲解用于中间代码生成的相关内容、常见方法与技巧。

## 1. 静态语义分析

### 1.1 静态语义分析简介

程序的语义是指在为程序单元赋予一定含义时程序应该满足的性质。通常，语义是多方面的，并且相比词法和语法来说更加难以定义。静态语义刻画程序在静态一致性或完整性方面的特征，而动态语义刻画程序执行时的行为。编译器根据语言的静态语义规则完成静态语义分析。静态语义分析过程中若发现程序有不符合静态语义规则之处，则报告语义错误；若没有语义错误，则称该程序通过了静态语义检查。仅当程序已通过静态语义检查，编译器才进一步根据语言的动态语义完成后续的中间代码或目标代码生成。若要求对程序在运行时的行为进行一定的检查（称为动态语义检查，如避免除零、数组越界等），多数情况均需要生成相应的代码。

静态语义检查的工作是多方面的，取决于不同的语言和不同的实现。最基本的工作就是检查程序结构（控制结构和数据结构）的一致性 or 完整性，例如：

- **控制流检查。**控制流语句必须使控制转移到合法的地方。例如，一个跳转语句会使控制转移到一个由标号指明的后续语句，如果标号没有对应到语句，那么就出现一个语义错误；另外，这一后续语句通常必须出现在和跳转语句相同的块中；又如，*break* 语句必须有合法的循环语句包围它；等等。
- **唯一性检查。**某些对象，如标识符、枚举类型的元素等，在源程序的一个指定上下文范围内只允许定义一次，因此，语义分析要确保它们的定义是唯一的。
- **名字的上下文相关性检查。**在源程序中，名字的出现遵循作用域与可见性前提下应该满足一定的上下文相关性，如果不满足就需要报告语义错误或警告信息。比如，变量在使用前必须经过声明，在外部不能访问私有变量，类声明和类实现之间需要规定相应的匹配关系，向对象发送消息时所调用的方法必须是在该对象的类中合法定义或继承的方法，等等。

- **类型检查。**例如，运算的类型检查需要搞清楚运算数是否与给定运算兼容，如果不兼容，它就要采取适当的动作来处理这种不兼容性，或者是指出错误，或者是进行自动类型转换；又如，源程序中使用的标识符是否已声明过，或者是否与已声明的类型相矛盾，这也可以看作是名字上下文相关性的一种约束条件；等等。

本学期 MiniDecaf 实验框架中也涉及到一些关于静态语义检查的任务，多数都可以在源语言语义规则的文档基础上并结合框架中的测例进行理解或完成。

类型检查或许是语义分析阶段最重要的工作。广义来讲，如果基于足够广泛的类型安全性质，所有静态语义检查工作均可以看作是静态类型检查。静态类型检查过程用于确保程序具有类型安全行为，剔除掉那些不具有类型安全性的程序，避免其进入运行状态。

静态语义分析的工作中有许多可以较方便地采用翻译模式来描述其实现规范，但有一些并不容易采用单遍扫描的方式实现，需要借助于多遍扫描的方法来处理，这种情况下需要用到多个翻译模式。在随后的小节里，我们重点讨论借助翻译模式来描述基于本学期 MiniDecaf 语言的类型系统如何实现相应的类型检查。

另外，在静态语义检查中经常会访问符号表信息。有关符号表的内容，可参考第四讲。

语义分析的另外一项工作是收集语义信息，这些信息服务于语义检查或后续的代码生成。这一节里，我们在讨论语义检查时会涉及到一部分内容，而另外一些内容（如函数、数组声明的处理）将合并到第 2 节“中间代码生成”部分进行讨论。

## 1.2 类型检查

实现类型检查过程的程序称为**类型检查程序**，主要功能包括：

- 验证程序的结构是否匹配上下文所期望的类型。
- 为代码生成阶段搜集及建立必要的类型信息。
- 实现某个类型系统。

类型检查程序，通常基于所定义的类型系统来实现。类型系统可维护程序中变量、表达式以及其他程序单元的类型信息，用于刻画程序的行为是否（类型）良好/安全可靠。在编译期间可以完成的类型检查即为静态类型检查。

下面先从类型系统说起。

### 1.2.1 类型系统简介<sup>1</sup>（选讲）

与程序语言类型系统相关的知识十分丰富，有兴趣的同学可以参考其他材料进行系统学习，如 Robert Harper 的“Practical Foundations for Programming Languages”[2] 以及 Benjamin C. Pierce 的“Types and Programming Languages”[3]。对于多数同学而言，可先了解类型系统的基本定义方法，能够在实现类型检查过程中实际应用即可。为此，我们推荐 Luca Cardelli 写的一份关于类型系统的材料[1]，可供大家参阅，作为入门。本节我们仅以一个简单语言为例对类型系统的概念进行初步介绍。

#### 1.2.1.1 对 MiniDecaf 语言的抽象

---

<sup>1</sup> 本学期课程的类型系统简介内容与 MiniDecaf 实验框架相关，由谢兴宇助教为主设计与整理。

为示范类型系统的定义，图 1 采用文法  $G[P]$  描述了一个 MiniDecaf 语言的抽象语法。其中  $\underline{id}$  和  $\underline{int}$  分别对应标识符和整型字面量； $\underline{uop}$ 、 $\underline{bop}$  和  $\underline{top}$  分别对应于单目运算符、（除了数组元素访问之外的）双目运算符以及三目运算符（即条件运算符），为简化讨论未指定具体的运算； $T[\underline{int}]$  和  $E[E]$  分别为数组声明和数组元素访问。

由文法  $G[P]$  可知，一个程序由若干全局变量和函数的定义所构成。简单起见，我们仅支持参数和返回值的类型均为  $\underline{int}$  的函数。函数体由一个语句序列构成，一条语句包括局部变量声明及定义、赋值、if 分支、while 循环、顺序复合、break、continue、return、单表达式语句，等等。

$G[P]$  所描述的是 MiniDecaf 语言的抽象语法（*abstract syntax*），并未包含源程序中一些必要的语法成分之间的界符/分隔符（为了方便说明，文法中还是保留了部分此类符号）。在后面的讨论中，大家应注意这一点。

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

图 1 MiniDecaf 语言的抽象语法

这一简单语言（MiniDecaf）是贯穿本讲所使用的小语言例子。

### 1.2.1.2 类型表达式

在设计类型检查程序时，首先需要为程序单元赋予类型的含义，即使用类型表达式对其进行解释。**类型表达式**是由基本类型，类型名字，类型变量，及类型构造子通过归纳定义得到的表达式。

例如，针对 MiniDecaf 语言，我们定义如下类型表达式的集合：

- 基本数据类型表达式： $\underline{int}$
- 数组类型表达式： $\text{array}(\tau)$ 。其中， $\tau$  是基本数据类型表达式或数组类型表达式。 $\text{array}(\tau)$  表示元素类型是  $\tau$  的数组类型。
- 函数类型表达式： $\text{fun}(n)$ 。其中， $n \geq 0$ ，表示函数的参数个数。在 MiniDecaf 中，函数的参数和返回值都只能为  $\underline{int}$ 。
- 类型表达式  $\text{type\_error}$  专用于有类型错误的程序单元。
- 类型表达式  $\text{ok}$  专用于没有类型错误的程序单元。

根据需要，可以修改或扩充类型表达式的种类。比如，若语言中定义了过程，而不是函数，则很容易将以上函数类型表达式修改为某种过程类型表达式。

这里，我们并没有涉及更复杂的类型表达式，如递归定义的类型，高阶函数类型，等等。同时，为简化讨论，我们也没有引入类型名字，类型变量，子类型，以及动态类型等内容。

### 1.2.1.3 类型环境

一个**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，记录了一些标识符在某个上下文中被赋予的类型表达式。这里，针对 MiniDecaf 语言，类型环境的值域仅包含 *int*、*array*( $\tau$ ) 和 *fun*( $n$ ) 三种类型表达式。

对于图 1 的 MiniDecaf 语言，我们用  $\Gamma$  表示类型环境。

我们用  $[\underline{id} \mapsto \tau]$  来表示对类型环境的更新，其中  $\underline{id}$  为一个变量名， $\tau$  为一个类型表达式，例如  $\Gamma[\underline{id} \mapsto \tau]$  表示一个以  $\text{dom}(\Gamma) \cup \{\underline{id}\}$  为定义域的类型环境，其可被如下定义：对于任意标识符  $x \in \text{dom}(\Gamma) \cup \{\underline{id}\}$ ，若  $\underline{id} = x$ ，则  $\Gamma[\underline{id} \mapsto \tau](x) = \tau$ ，否则有  $\Gamma[\underline{id} \mapsto \tau](x) = \Gamma(x)$ 。

我们还会定义一个特殊的函数  $\text{Update}(\Gamma, s)$ ，其中  $\Gamma$  是一个类型环境， $s$  是一个语句序列， $\text{Update}$  函数返回的结果是对  $\Gamma$  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  $\tau_1$  的变量  $\underline{id}_1$ 、类型为  $\tau_2$  的变量  $\underline{id}_2$ ……类型为  $\tau_n$  的变量  $\underline{id}_n$ ，则  $\text{Update}(\Gamma, s) = \Gamma[\underline{id}_1 \mapsto \tau_1][\underline{id}_2 \mapsto \tau_2] \dots [\underline{id}_n \mapsto \tau_n]$ 。

### 1.2.1.4 类型规则

将类型表达式赋给程序各个部分的规则集合就构成一个**类型系统**。下面，我们给出描述上述简单语言的一个类型系统的**类型规则**集合。类型规则与在特定类型环境（或作用域）有关，因此除了常规断言，需要用到与类型环境相关的断言，在我们的例子中，引入下列断言形式：

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

其中， $t$  代表类型声明， $e$  代表除了语句序列之外的程序单元， $s$  代表一个语句序列，等等， $A$  代表一个类型表达式， $e: A$  读作“ $e$  的类型为  $A$ ”。为了方便，这里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，其有两种可能的取值 **in**（在某个循环体中）和 **out**（不在任何一个循环体中）。

MiniDecaf 的类型系统可以定义如下：

（注：如无特殊说明，我们使用  $\tau$  来指代 *int* 或数组类型表达式）

- 针对表达式的类型规则

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

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

$$\Gamma \vdash \underline{\text{id}} : \tau$$

$$\frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash \underline{\text{uop}} e : \text{int}} \quad (\text{E-uop})$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \underline{\text{bop}} e_2 : \text{int}} \quad (\text{E-bop})$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_3 : \text{int}}{\Gamma \vdash \underline{\text{top}} (e_1 e_2 e_3) : \text{int}} \quad (\text{E-top})$$

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

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

- 针对语句的类型规则

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

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

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

$$\Gamma \vdash e_1 : \tau_l \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_3 : \tau_2 \quad \Gamma \vdash_{\text{in}} s : \text{ok} \quad e_1, e_2, \text{ or } e_3 \text{ may be empty}$$

---


$$\Gamma \vdash_l \text{for } (e_1; e_2; e_3) s : ok \quad (\text{S-fore})$$

$$\frac{\begin{array}{c} \vdash t : \tau \quad \Gamma[\underline{\text{id}} \mapsto \tau] \vdash_{e_1} : int \quad \Gamma[\underline{\text{id}} \mapsto \tau] \vdash_{e_2} : \tau_2 \\ \Gamma[\underline{\text{id}} \mapsto \tau] \vdash_{\text{in } s} : ok \quad e_1 \text{ or } e_2 \text{ may be empty} \end{array}}{\Gamma \vdash_l \text{for } (t \underline{\text{id}}; e_1; e_2) s : ok} \quad (\text{S-ford})$$

$$\frac{\begin{array}{c} \vdash t : \tau \quad \Gamma[\underline{\text{id}} \mapsto \tau] \vdash_{e_1} : \tau \quad \Gamma[\underline{\text{id}} \mapsto \tau] \vdash_{e_2} : int \quad \Gamma[\underline{\text{id}} \mapsto \tau] \vdash_{e_3} : \tau_3 \\ \Gamma[\underline{\text{id}} \mapsto \tau] \vdash_{\text{in } s} : ok \quad e_2 \text{ or } e_3 \text{ may be empty} \end{array}}{\Gamma \vdash_l \text{for } (t \underline{\text{id}} = e_1; e_2; e_3) s : ok} \quad (\text{S-fordi})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash_l e : ok} \quad (\text{S-exp})$$

$$\frac{\Gamma \vdash e : int}{\Gamma \vdash_l \text{return } e : ok} \quad (\text{S-return})$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash_l t \underline{\text{id}} : ok} \quad (\text{S-D})$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma[\underline{\text{id}} \mapsto \tau] \vdash e : \tau}{\Gamma \vdash_l t \underline{\text{id}} = e : ok} \quad (\text{S-Di})$$

$$\frac{\Gamma \vdash_l s : ok}{\Gamma \vdash_l \{s\} : ok} \quad (\text{S-}\{\})$$

$$\frac{\Gamma \vdash_l s_1 : ok \quad \text{Update}(\Gamma, s) \vdash_l s_2 : ok}{\Gamma \vdash_l s_1 s_2 : ok} \quad (\text{S-S})$$

$$\frac{}{\Gamma \vdash_{\text{in}} \text{break} : \text{ok}} \quad (\text{S-break})$$

$$\frac{}{\Gamma \vdash_{\text{in}} \text{continue} : \text{ok}} \quad (\text{S-continue})$$

$$\frac{}{\Gamma \vdash_l \varepsilon : \text{ok}} \quad (\text{S-eps})$$

- 针对类型声明的规则

$$\frac{}{\vdash \text{int} : \text{int}} \quad (\text{T-int})$$

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

- 针对函数定义的类型规则

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

- 针对顶层程序单元的类型规则

$$\frac{\Gamma \vdash \text{int } \underline{\text{id}} (\text{int } \underline{\text{id}}_1, \dots, \text{int } \underline{\text{id}}_n) s : \text{ok} \quad \Gamma[\underline{\text{id}} \mapsto \text{fun}(n)] \vdash p : \text{ok}}{\Gamma \vdash \text{int } \underline{\text{id}} (\text{int } \underline{\text{id}}_1, \dots, \text{int } \underline{\text{id}}_n) s p : \text{ok}} \quad (\text{P-F})$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma[\underline{\text{id}} \mapsto \tau] \vdash p : \text{ok}}{\Gamma \vdash t \underline{\text{id}} p : \text{ok}} \quad (\text{P-D})$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash e : \tau \quad \Gamma[\underline{\text{id}} \mapsto \tau] \vdash p : \text{ok}}{} \quad (\text{P-Di})$$



$$\Gamma \vdash t \text{ id} = e p : ok$$

$$\frac{}{\Gamma \vdash \varepsilon : ok} \text{ (P-eps)}$$

(当一个程序单元的类型无法通过这些类型规则被推导出时，我们记其类型为 *type\_error*)

### 1.2.1.5 例：类型检查

$$\begin{array}{c} \frac{\{foo \mapsto fun(2), x \mapsto int, y \mapsto int\}(x)=int \quad \{foo \mapsto fun(2), x \mapsto int, y \mapsto int\}(y) = int}{\text{ (E-id) }} \\ \frac{\{foo \mapsto fun(2), x \mapsto int, y \mapsto int\} \vdash x : int \quad \{foo \mapsto fun(2), x \mapsto int, y \mapsto int\} \vdash y : int}{\text{ (E-bop) }} \\ \frac{\{foo \mapsto fun(2), x \mapsto int, y \mapsto int\} \vdash x + y : int}{\text{ (S-return) }} \\ \frac{\{foo \mapsto fun(2), x \mapsto int, y \mapsto int\} \vdash_{out} \text{return } x + y : ok}{\text{ (F-def) }} \quad \text{ (P-eps)} \\ \frac{\emptyset \vdash \text{int foo (int x, int y) return } x + y : ok \quad \{foo \mapsto fun(2)\} \vdash \varepsilon : ok}{\text{ (P-F) }} \\ \emptyset \vdash \text{int foo (int x, int y) return } x + y \varepsilon : ok \\ \\ \frac{\{foo \mapsto fun(0)\} \vdash x : type\_error}{\text{ (S-return) }} \\ \frac{\{foo \mapsto fun(0)\} \vdash_{out} \text{return } x : type\_error}{\text{ (F-def) }} \quad \text{ (P-eps)} \\ \frac{\emptyset \vdash \text{int foo () return } x : type\_error \quad \{foo \mapsto fun(0)\} \vdash \varepsilon : ok}{\text{ (P-F) }} \\ \emptyset \vdash \text{int foo () return } x \varepsilon : type\_error \end{array}$$

### 1.2.1.6 类型系统相关话题

- 类型等价 (equivalence)。结构等价 (structural equivalence)，名字等价 (by-name equivalence)，合一 (unification) 算法。
- 类型推导 (inference)。类型推导问题的核心是根据类型规则推出 (derive) 程序项 (term) 或程序单元的类型，决定了类型检查算法的构造过程，其可解性 (decidability) 和复杂度 (decidability) 取决于类型系统的定义。静态/动态类型推导。

- 子类型（subtyping）关系。用于考虑类型转换（conversion），类型兼容（compatibility）以及多态（polymorphism）、重载（overloading）等问题。
- 类型合理性/可靠性（Soundness）。由于涉及到程序的行为安全性，因此需结合动态语义来定义，如指称语义（denotational semantics）和操作语义（operational semantics）。

.....

这些话题大多超出本课范围，有兴趣的同学可参考有关书籍和文献。

### 1.2.2 借助翻译模式描述类型检查的实现算法

类型系统是由类型检查程序实现的，实现类型检查程序的算法通常可以借助于翻译模式进行规范/描述。相比 1.2.1.4 所示的形式化定义的类型规则，翻译模式离代码实现更近。如本讲前言部分所述，可以将 L-翻译模式直接对应于递归下降翻译程序。

下面以图 1 的简单语言（MiniDecaf 抽象语言）为例，讨论实现类型检查的翻译模式设计，主要工作是将类型表达式作为属性值赋给程序各个部分，实现相应语言的一个类型系统。为简化讨论，对于 1.2.1.4 中类型规则的实现并不完整，可能会忽略某些方面。

以下是与声明相关的翻译模式片断，其作用是计算变量声明相关语法单位的类型信息，并保存标识符的类型信息至符号表：

```

 $F \rightarrow \text{int } \underline{id} (L) S$       {  $\text{addtype}(\underline{id}.\text{entry}, \text{fun}(L.\text{num}))$  };
                                $F.\text{type} := \text{if } S.\text{type} = \text{ok} \text{ then } \text{ok} \text{ else } \text{type\_error}$  } // 规则 (F-def)
 $L \rightarrow L_1 , \text{int } \underline{id}$     {  $\text{addtype}(\underline{id}.\text{entry}, \text{int})$  ;  $L.\text{num} := L_1.\text{num} + 1$  }
 $L \rightarrow \varepsilon$                 {  $L.\text{num} := 0$  }
 $D \rightarrow T \underline{id}$             {  $\text{addtype}(\underline{id}.\text{entry}, T.\text{type})$  ;  $D.\text{type} := \text{ok}$  }
 $D \rightarrow T \underline{id} = E$         {  $\text{addtype}(\underline{id}.\text{entry}, T.\text{type})$  ;
                                $D.\text{type} := \text{if } T.\text{type} = E.\text{type} \text{ then } \text{ok} \text{ else } \text{type\_error}$  }
 $T \rightarrow \text{int}$                 {  $T.\text{type} := \text{int}$  } // 规则 (T-int)
 $T \rightarrow T_1 [ \underline{int} ]$       {  $T.\text{type} := \text{if } \underline{int}.\text{lexval} > 0 \text{ then } \text{array}(T_1.\text{type}) \text{ else } \text{type\_error}$  }
                               // 规则 (T-array)

```

其中， $\underline{int}.\text{lexval}$  为词法分析返回的单词属性值， $\underline{id}.\text{entry}$  指向当前标识符对应于符号表中的表项，语义函数  $\text{addtype}(\underline{id}.\text{entry}, \tau)$  表示将类型属性值  $\tau$  填入当前标识符在符号表表项中的  $\text{type}$  域（记录标识符的类型）。

以下是与表达式相关的翻译模式片断，其作用是计算表达式相关语法单位的类型信息，同时检查表达式中运算数类型与给定运算是否兼容：

```

 $E \rightarrow \underline{int}$               {  $E.\text{type} := \text{int}$  } // 规则 (E-int)
 $E \rightarrow \underline{id}$               {  $E.\text{type} := \text{if } \text{lookup\_type}(\underline{id}.\text{name}) = \text{nil}$ 
                               then  $\text{type\_error}$  else  $\text{lookup\_type}(\underline{id}.\text{name})$  } // 规则 (E-id)
 $E \rightarrow \underline{\text{uop}} E_1$         {  $E.\text{type} := \text{if } E_1.\text{type} = \text{int} \text{ then } \text{int} \text{ else } \text{type\_error}$  } // 规则 (E-uop)
 $E \rightarrow E_1 \underline{\text{bop}} E_2$     {  $E.\text{type} := \text{if } E_1.\text{type} = \text{int} \text{ and } E_2.\text{type} = \text{int}$ 
                               then  $\text{int}$  else  $\text{type\_error}$  } // 规则 (E-bop)
 $E \rightarrow \underline{\text{top}} (E_1 E_2 E_3)$  {  $E.\text{type} := \text{if } E_1.\text{type} = \text{int} \text{ and } E_2.\text{type} = \text{int} \text{ and } E_3.\text{type} = \text{int}$ 

```

$$\begin{array}{ll}
& \text{then } int \text{ else } type\_error \} \quad // \text{ 规则 (E-top)} \\
E \rightarrow E_1 [E_2] & \{ E.type := \text{if } E_2.type = int \text{ and } E_1.type = array(\tau) \\
& \text{then } \tau \text{ else } type\_error \} \quad // \text{ 规则 (E-access)} \\
E \rightarrow \underline{id} (A) & \{ E.type := \text{if } lookup\_type(\underline{id}.name) = fun(n) \text{ and } A.type = ok \text{ and } A.num = n \\
& \text{then } int \text{ else } type\_error \} \quad // \text{ 规则 (E-call)} \\
A \rightarrow A_1, E & \{ A.type := \text{if } A_1.type = ok \text{ and } E.type := int \text{ then } ok \text{ else } type\_error; \\
& A.num := A_1.num + 1 \} \\
A \rightarrow \varepsilon & \{ A.type := ok ; A.num := 0 \}
\end{array}$$

其中,  $\underline{id}.name$  为当前标识符的名字; 语义函数  $lookup\_type(\underline{id}.name)$  从符号表中查找名字为  $\underline{id}.name$  的标识符所对应的表项中  $type$  域的内容, 若未查到该表项或表项中的  $type$  域无定义, 则返回  $nil$ 。

以下是与语句相关的类型检查的翻译模式片断:

$$\begin{array}{ll}
S \rightarrow \text{if} (E) S_1 S_2 & \{ S.type := \text{if } E.type = int \text{ and } S_1.type = ok \text{ and } S_2.type = ok \\
& \text{then } ok \text{ else } type\_error \} \quad // \text{ 规则 (S-if)} \\
S \rightarrow \text{while} (E) S_1 & \{ S.type := \text{if } E.type = int \text{ then } S_1.type \text{ else } type\_error \} \quad // \text{ 规则 (S-while)} \\
S \rightarrow \text{do } S_1 \text{ while} (E) & \{ S.type := \text{if } E.type = int \text{ then } S_1.type \text{ else } type\_error \} \quad // \text{ 规则 (S-do)} \\
S \rightarrow \text{for} (R_1; R_2; R_3) S_1 & \{ S.type := \text{if } R_2.type = int \text{ and } R_1.type = \tau \text{ and } R_3.type = \tau \\
& \text{then } S_1.type \text{ else } type\_error \} \quad // \text{ 规则 (S-fore)} \\
S \rightarrow \text{for} (D; R_1; R_2) S_1 & \{ S.type := \text{if } D.type = ok \text{ and } R_1.type = int \text{ and } R_2.type = \tau \\
& \text{then } S_1.type \text{ else } type\_error \} \quad // \text{ 规则 (S-ford, S-fordi)} \\
S \rightarrow E & \{ S.type := \text{if } E.type = \tau \text{ then } ok \text{ else } type\_error \} \quad // \text{ 规则 (S-exp)} \\
S \rightarrow \text{return } E & \{ S.type := \text{if } E.type = int \text{ then } ok \text{ else } type\_error \} \quad // \text{ 规则 (S-return)} \\
S \rightarrow D & \{ S.type := D.type \} \quad // \text{ 规则 (S-D)} \\
S \rightarrow \{ S_1 \} & \{ S.type := S_1.type \} \quad // \text{ 规则 (S-{} )} \\
S \rightarrow S_1 S_2 & \{ S.type := \text{if } S_1.type = ok \text{ and } S_2.type = ok \text{ then } ok \text{ else } type\_error \} \quad // \text{ 规则 (S-S)} \\
S \rightarrow \text{break} & \{ S.type := ok \} \quad // \text{ 规则 (S-break, 未检查是否在循环体内)} \\
S \rightarrow \text{continue} & \{ S.type := ok \} \quad // \text{ 规则 (S-continue, 未检查是否在循环体内)} \\
S \rightarrow \varepsilon & \{ S.type := ok \} \quad // \text{ 规则 (S-eps)} \\
R \rightarrow E & \{ R.type := E.type \} \\
R \rightarrow \varepsilon & \{ R.type := int \}
\end{array}$$

最后, 我们补充如下翻译模式片断:

$$\begin{array}{ll}
P \rightarrow F P_1 & \{ P.type := \text{if } F.type = ok \text{ and } P_1.type = ok \text{ then } ok \text{ else } type\_error \} \quad // \text{ 规则 (P-F)} \\
P \rightarrow D P_1 & \{ P.type := \text{if } D.type = ok \text{ and } P_1.type = ok \text{ then } ok \text{ else } type\_error \} \quad // \text{ 规则 (P-D)} \\
P \rightarrow \varepsilon & \{ P.type := ok \} \quad // \text{ 规则 (P-eps)}
\end{array}$$

容易理解, 如果  $P.type$  的计算结果为  $ok$ , 则对应的输入程序即通过了类型检查。

读者可能已经注意到, 上述翻译模式没有对  $break$  和  $continue$  语句进行是否循环体内部。通过引入继承属性  $S.inloop$  可以解决这一问题, 以下仅列出有变化的产生式:

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

```

 $S \rightarrow \text{if } (E) \{ S_1.inloop := S.inloop \} S_1 \{ S_2.inloop := S.inloop \} S_2$ 
 $\{ S.type := \text{if } E.type = \text{int and } S_1.type = \text{ok and } S_2.type = \text{ok}$ 
 $\text{then ok else type\_error} \}$  // 规则 (S-if)
 $S \rightarrow \text{while } (E) \{ S_1.inloop := 1 \} S_1 \{ S.type := \text{if } E.type = \text{int then } S_1.type \text{ else type\_error} \}$ 
// 规则 (S-while)
 $S \rightarrow \text{do } \{ S_1.inloop := 1 \} S_1 \text{ while } (E) \{ S.type := \text{if } E.type = \text{int then } S_1.type \text{ else type\_error} \}$ 
// 规则 (S-do)
 $S \rightarrow \text{for } (R_1; R_2; R_3) \{ S_1.inloop := 1 \} S_1$ 
 $\{ S.type := \text{if } R_2.type = \text{int and } R_1.type = \tau \text{ and } R_3.type = \tau$ 
 $\text{then } S_1.type \text{ else type\_error} \}$  // 规则 (S-fore)
 $S \rightarrow \text{for } (D; R_1; R_2) \{ S_1.inloop := 1 \} S_1$ 
 $\{ S.type := \text{if } D.type = \text{ok and } R_1.type = \text{int and } R_2.type = \tau$ 
 $\text{then } S_1.type \text{ else type\_error} \}$  // 规则 (S-ford, S-fordi)
 $S \rightarrow \{ S_1.inloop := S.inloop \} S_1 \{ S_2.inloop := S.inloop \} S_2$ 
 $\{ S.type := \text{if } S_1.type = \text{ok and } S_2.type = \text{ok then ok else type\_error} \}$  // 规则 (S-S)
 $S \rightarrow \{ \{ S_1.inloop := S.inloop \} S_1 \} \{ S.type := S_1.type \}$  // 注: '{'和'}'加引号以区别元符号
 $S \rightarrow \text{break} \{ S.type := \text{if } S.inloop = 1 \text{ then ok else type\_error} \}$  // 规则 (S-break)
 $S \rightarrow \text{continue} \{ S.type := \text{if } S.inloop = 1 \text{ then ok else type\_error} \}$  // 规则 (S-continue)

```

## 2. 中间代码生成

中间代码是源程序的不同表示形式，也称为**中间表示**，其作用包括：

- 用于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。

如果源程序的词法、语法和语义正确，编译程序通常会将这个源程序翻译到机器无关的中间表示形式。在实现一个语言时，可能会用到不同层次的多种中间表示形式，称为**多级中间表示**。由源程序到第一级中间表示的翻译，再翻译到后面一级中间表示。最后一级中间表示将被翻译为机器相关的目标代码。

### 2.1 常见的中间表示形式

中间表示形式有不同层次不同目的之分。下面列举几种中间表示形式：

- *AST* (*Abstract syntax tree*, **抽象语法树**, 简称**语法树**), 及其改进形式 *DAG* (*Directed Acyclic Graph*, **有向无圈图**)。
- *TAC* (*Three-address code*, **三地址码或三元式**)。
- *P-code* (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)

例如, 算术表达式  $A + B * (C - D) + E / (C - D) \wedge N$  的一种 AST 和 DAG 表示分别如图 2 (a) 和 (b) 所示。抽象语法树中每一个子树的根结点都对应一种动作或运算, 它的所有子结点对应该动作或运算的参数或运算数。参数或运算数也可以是另一子树, 代表另一动作或运算。有向无圈图在语法树的基础上, 对某些执行同样动作或运算的子树进行了合并。

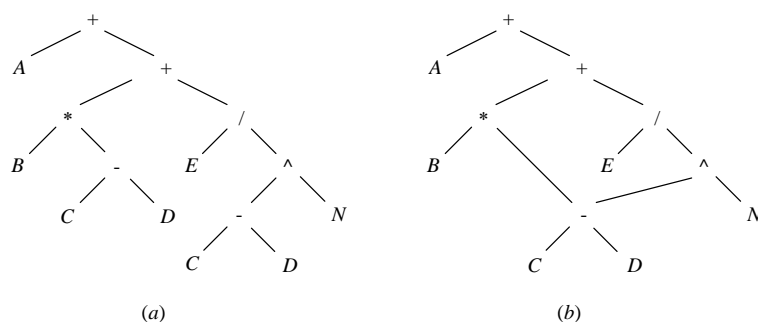


图 2 抽象语法树和有向无圈图

该表达式的一种 TAC 表示如图 3 所示。TAC 是一组顺序执行的语句序列, 其语句可以表示为如下形式:

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

其中,  $op$  为运算符,  $y$  和  $z$  为运算数,  $x$  为运算结果。语句或可采用四元式形式表示为:

$$(op \quad y \quad z \quad x)$$

|     |   |                       |
|-----|---|-----------------------|
| (1) | (- C D T <sub>1</sub> )                           | $T_1 := C - D$        |
| (2) | (* B T <sub>1</sub> T <sub>2</sub> )              | $T_2 := B * T_1$      |
| (3) | (+ A T <sub>2</sub> T <sub>3</sub> )              | $T_3 := A + T_2$      |
| (4) | (- C D T <sub>4</sub> )                           | 或 $T_4 := C - D$      |
| (5) | (^ T <sub>4</sub> N T <sub>5</sub> )              | $T_5 := T_4 \wedge N$ |
| (6) | (/ E T <sub>5</sub> T <sub>6</sub> )              | $T_6 := E / T_5$      |
| (7) | (+ T <sub>3</sub> T <sub>6</sub> T <sub>7</sub> ) | $T_7 := T_3 + T_6$    |

图 3 三地址码 / 四元式

注: TAC 语句中的  $x, y, z$  通常对应变量的地址信息 (立即数除外), 因而称为 TAC。然而,  $x, y, z$  中的每个位置都有可能为空。

P-code 和 Bytecode 是具体程序设计语言专用的中间代码形式, 有需要的读者可参考相关的技术手册。

静态单赋值 (SSA) 形式借鉴了纯函数式语言的特性。“单赋值”的含义是: 程序中的名字仅有一次赋值。在 SSA 形式中, 在使用一个名字时仅关联于唯一的“定值点”。此一特性使得沿着 DU 链 (参见第 14 讲) 的程序分析信息可以进行代数替换, 因此十分有利于程序分析和优化。

获得 SSA 形式需要两个步骤：

- 第一步是对程序的“定值点”进行“重命名”。比如，对于图 4 左边的程序，我们将  $x$  的两个定值点的名字分别重命名为  $x_1$  和  $x_2$ ,  $y$  的两个定值点的名字分别重命名为  $y_1$  和  $y_2$ , 以及  $w$  的两个定值点的名字分别重命名为  $w_1$  和  $w_2$ 。对于没有分支的程序，通过重命名足以获得 SSA 形式。
- 第二步是插入  $\phi$  函数。对于有分支的情形，需要通过插入所谓的“ $\phi$  函数”来解决同一名字的多个定值点的合流问题。例如，图 4 程序中条件语句之后的  $y$  的定值点是  $y_1$  还是  $y_2$  呢？如图 3 右边的代码所示，在条件语句之后插入  $\phi$  函数  $\phi(y_1, y_2)$ , 并赋值给  $y_3$ 。在条件语句之后使用的  $y$  是  $y_3$ 。 $\phi(y_1, y_2)$  的含义是：程序若执行 **then** 分支时取定值点为  $y_1$ , 若执行 **else** 分支时取定值点为  $y_2$ 。“ $\phi$  函数”仅作为特殊标志供编译时使用，当相应的分析和优化工作结束后，在寄存器分配和代码生成过程中将根据代码原有的语义被解除。

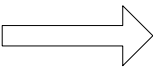
|  |   |  |
|--|---|--|
| <pre>x ← 5 x ← x - 3 if x &lt; 3 then   y ← x * 2   w ← y else   y ← x - 3   w ← x - y   z ← x + y</pre> |  | <pre>x<sub>1</sub> ← 5 x<sub>2</sub> ← x<sub>1</sub> - 3 if x<sub>2</sub> &lt; 3 then   y<sub>1</sub> ← x<sub>2</sub> * 2   w<sub>1</sub> ← y<sub>1</sub> else   y<sub>2</sub> ← x<sub>2</sub> - 3   y<sub>3</sub> ← <math>\phi(y_1, y_2)</math>   w<sub>2</sub> ← x<sub>2</sub> - y<sub>3</sub>   z ← x<sub>2</sub> + y<sub>3</sub></pre> |
|--|---|--|

图 4 静态单赋值形式

在现行的编译程序中，*AST* 是较常用的高级中间表示形式，而 *TAC* 是较常用的低级中间表示形式。许多编译程序都是将源程序首先翻译成等价的 *AST* 形式，然后再从 *AST* 表示得到对应的 *TAC* 形式。这个过程中也伴随着基于 *AST* 和 *TAC* 进行的代码优化工作。*SSA* 是很受重视的专用于分析和优化的中间表示形式，但有关 *SSA* 更多的内容超出本课范围，有需要的同学可参考相关书籍。

在随后的两个小节里，我们将以语法制导的方法为依托，以常见语言成分的翻译为例介绍中间代码生成的一些常用技术，涉及到两类重要的中间表示形式，即 *AST* 和 *TAC*。

## 2.2 生成抽象语法树

抽象语法树 (*AST*) 是一种非常接近源代码的中间表示，它的特点是：(1) 不含我们不关心的终结符（例如逗号），而只含像标识符、常量等之类的终结符；(2) 不具体体现语法分析的细节步骤，例如对于  $A \rightarrow AE \mid \varepsilon$  这样的规则，按照语法分析的细节步骤来记录的话应该是一棵二叉树，但是在 *AST* 中我们可以将其表示成同类节点的一个链表<sup>2</sup>，这样更便于后续处理；(3) 能够完整体现源程序的语法结构，使后续过程可以反复利用。合理定义抽象语法树的节点类型，是编译器设计人员的重要责任之一。

生成抽象语法树的工作一般是在语法分析的同时进行的，如在 *MiniDecaf* 实验框架中，我

---

<sup>2</sup> 注：本讲中的翻译模式片段中，多数产生式规则均可适用于抽象语法树，然而，为了方便讲解，我们并没有设计直接针对同类节点链表进行操作的语义函数，而是还保留了  $A \rightarrow AE \mid \varepsilon$  这样的规则，可以少引入一些相关的语义处理，如链表上的迭代处理函数（通过某些高阶算子，比如 *map*）。在具体实现时，这样做是有必要的，更加高效，复用性也更好。

们借助于 yacc 工具所完成的工作，这相当于是第九讲所介绍的语法制导的语义计算。使用 yacc 工具的核心是写 yacc 规范描述，相当于基于 MiniDecaf 源语言语法规则来写翻译模式。比如，通过如下翻译模式描述的算法在语法分析的过程中将部分简单语句和表达式翻译至一种 AST（语法制导的方法）：

|   |   |
|---|---|
| $S \rightarrow \underline{id} = E$                | $\{ S.ptr := mknnode('assign', mkleaf(\underline{id}.entry), E.ptr) \}$ |
| $S \rightarrow \text{if } E \text{ then } S_1$    | $\{ S.ptr := mknnode('if\_then', E.ptr, S_1.ptr) \}$                    |
| $S \rightarrow \text{while } E \text{ then } S_1$ | $\{ S.ptr := mknnode('while\_do', E.ptr, S_1.ptr) \}$                   |
| $S \rightarrow S_1 S_2$                           | $\{ S.ptr := mknnode('seq', S_1.ptr, S_2.ptr) \}$                       |
| $E \rightarrow \underline{id}$                    | $\{ E.ptr := mkleaf(\underline{id}.entry) \}$                           |
| $E \rightarrow E_1 + E_2$                         | $\{ E.ptr := mknnode('add', E_1.ptr, E_2.ptr) \}$                       |
| $E \rightarrow E_1 * E_2$                         | $\{ E.ptr := mknnode('mul', E_1.ptr, E_2.ptr) \}$                       |
| $E \rightarrow (E_1)$                             | $\{ E.ptr := E_1.ptr \}$  |
| .....   | .....   |

其中，*mknnode* 为构造 AST 内部结点的语义函数，它的第一个参数标识该结点相应的动作或运算，其余参数代表各个子结点对应的运算数或运算数指针（子结点个数对应运算的元数）；*mkleaf* 为构造 AST 叶结点的语义函数，叶结点对应常量或变量运算数；*id.entry* 为指向当前标识符对应于符号表中表项的指针；*int.val* 和 *real.val* 均为词法分析得到的常量值。语义函数 *mknnode* 和 *mkleaf* 都将返回相应结点的一个指针。文法符号 *S*、*E* 的综合属性 *S.ptr*、*E.ptr* 分别对应 AST 中某个结点的指针。从上述翻译模式片断中不难看出各个结点所对应动作或运算的含义。

上述翻译模式所处理的源语言语法以及抽象语法树结点的定义或许与 MiniDecaf 实验框架有个别非本质的出入，不会影响大家的理解。

## 2.3 生成三地址码

三地址码（TAC），是一种比较接近汇编语言的表示方式。

与生成 AST 类似，我们同样可以给出生成 TAC 的翻译模式。然而，TAC 是较低级的中间表示，因而技术层面上需要考虑更加复杂和细致一些问题。

在随后的例子中，我们可能用到的 TAC 语句类型如下（与 MiniDecaf 实验框架的 TAC 语句类型大致吻合）：

- 赋值语句  $x := y \text{ op } z$  （*op* 代表二元运算）。
- 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 *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*）。
- 参数语句 *param a* （变量 *a* 作为调用语句的参数传递）。
- 间接调用语句  $x := \text{call } a$  （取出变量 *a* 中函数地址，并调用，结果赋值给 *x*）。

- 函数返回语句 `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  $i$  个存储单元的值赋给  $x$ ）。
- `STORE` 语句 `x[i] := y` （将  $y$  的值保存到  $x$  的存储位置起第  $i$  个存储单元）。
- `ALLOC` 语句 `x := alloc y` （分配  $y$  个字节内存，起始位置赋值给  $x$ ）。
- `MEMO` 语句 `memo 'XXX'` （含形参变量及其大小，专供栈帧存储分配使用）

注：这里，*TAC* 语句中的临时变量对应一个存储位置。实际上，在 *TAC* 层次，变量名字所对应的存储位置信息（相对基地址的偏移量）总是可以从符号表中得到。换句话说，变量的取值即为其名字对应的存储位置上存储单元的内容。另外，标号  $L$  对应一个全局的行号。

### 2.3.1 声明语句的翻译

源程序中标识符的许多信息在 *TAC* 中不复存在，许多重要信息需要保存在符号表中。这些信息如类型，偏移地址等。在 1.2.2 的示例中，为实现类型检查，我们设计了处理变量声明的翻译模式片断，可以将变量标识符的类型保存至符号表。下面我们设计新的翻译模式片断，以使变量标识符的偏移地址信息可以保存至符号表。这些翻译模式可以在类型检查的翻译模式基础上扩充，以使变量标识符的类型以及偏移地址可以同时保存至符号表，即可以通过一遍扫描来完成。还可以设计单独的翻译模式，即通过单独扫描来计算变量标识符的偏移地址信息，并保存至符号表。为清晰易读同时方便讲解，我们选择后者。

全局变量将分配全局存储空间，需要计算其在全局空间中的偏移量，并存入全局符号表。如下是与全局声明语句相关的翻译模式片段：

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

其中，文法符号的属性值具有如下含义： $\underline{\text{int}}.\text{lexval}$  为词法分析返回的单词属性值， $\underline{\text{id}}.\text{name}$  为  $\underline{\text{id}}$  的词法名字；综合属性  $T.\text{width}$  表示所申明类型所占的字节数；综合属性  $D.\text{width}$  和  $P.\text{width}$  表示所申明变量所占的字节数；继承属性  $P.\text{offset}$  和  $D.\text{offset}$  表示相应程序单元起始位置在全局空间中的偏移量；继承属性  $F.\text{offset}$  与全局空间的偏移量无关，这里被置为 0 意味着重置函数局部存储空间的起始位置。

语义函数  $\text{enter}(\underline{\text{id}}.\text{name}, o)$  的含义为：将符号表中  $\underline{\text{id}}.\text{name}$  所对应表项的  $\text{offset}$  域置为  $o$ 。上面的翻译模式片段是针对全局声明语句，因此这里对应的是全局符号表。

另外，在这个翻译模式中，我们假设了基础数据类型的宽度（字节数）：整型为 4。



下面我们处理函数内部的局部变量声明（含函数参数），需要计算各个变量在函数局部空间中的偏移量，并存入该函数的局部符号表。设变量在局部存储空间中的偏移量按照声明的先后递增，且参数在前。此外，上面的翻译模式片段中针对语法单元  $D$  和  $T$  的部分仍然适用。如下是相应于剩余语法单元的翻译模式片段：

$$\begin{aligned}
 F &\rightarrow \text{int } \underline{id} \{ \{ L.offset := F.offset \} L \} \{ S.offset := F.offset + L.width \} S^3 \\
 L &\rightarrow \{ L_1.offset := L.offset \} L_1, \text{int } \underline{id} \\
 &\quad \{ \text{enter}(\underline{id.name}, L.offset + L_1.width) ; L.width := L_1.width + 4 \} \\
 L &\rightarrow \varepsilon \quad \{ L.width := 0 \} \\
 S &\rightarrow \text{if} (E) \{ S_1.offset := S.offset \} S_1 \{ S_2.offset := S.offset \} S_2 \{ S.width := 0 \} \\
 S &\rightarrow \text{while} (E) \{ S_1.offset := S.offset \} S_1 \{ S.width := 0 \} \\
 S &\rightarrow \text{do} \{ S_1.offset := S.offset \} S_1 \text{while} (E) \{ S.width := 0 \} \\
 S &\rightarrow \text{for} (R_1; R_2; R_3) \{ S_1.offset := S.offset \} S_1 \{ S.width := 0 \} \\
 S &\rightarrow \text{for} (D; R_1; R_2) \{ S_1.offset := S.offset + D.width \} S_1 \{ S.width := 0 \} \\
 S &\rightarrow D \{ S.width := D.width \} \\
 S &\rightarrow \{ \{ S_1.offset := S.offset \} S_1 \} \{ S.width := 0 \} \\
 S &\rightarrow \{ S_1.offset := S.offset \} S_1 \{ S_2.offset := S.offset + S_1.width \} S_2 \\
 &\quad \{ S.width := S_1.width + S_2.width \}
 \end{aligned}$$

其中，文法符号的属性 *offset* 和 *width* 与前面类似。值得注意以下几点：（1）由于当前是局部环境，因此在处理语法单元  $D$  时，所调用的语义函数 *enter* (*id.name*, *o*) 是在当前函数的局部符号表相应表项的 *offset* 域添加偏移量信息；（2）对于语句块或者循环内部声明的变量，其可见性仅限于该语句块或相应循环体，退出后所占空间将释放（请注意属性 *S.width* 的计算方式）。

在声明数组类型的变量时，需要根据具体实现时所采用的存储组织方式来计算相关语法单元的编译量信息。若需在堆中分配空间，还需要生成相应的代码。后续我们会对数组相关的内容进行专门的讨论。

另外，在处理声明  $T \underline{id} = E$  时，因为有赋值，所以应该生成相应的 Tac 语句。对此，我们合并到随后的小节一并考虑。

### 2.3.2 数组声明和数组元素引用的翻译

在 1.2.2 和 2.3.1 的翻译模式中，已经包含了有关数组声明和数组元素引用相关的处理。为方便进一步深入讨论，我们将相关翻译模式片段进行合并，得到如下翻译模式：

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

<sup>3</sup> 注：如果需要参数的与函数体内局部变量分开寻址（比如处于基址不同方向的偏移位置），则可将  $S.offset := F.offset + L.width$  改为  $S.offset := F.offset$ ，而参数偏移量可通过 2.3.5 节中提到的 *memo* 信息记录，其设计取决于函数调用的具体实现方案。

这里，语义函数  $enter(\underline{id.name}, t, o)$  合并了之前分别添加类型和偏移信息两项功能，其含义为：将符号表中  $\underline{id.name}$  所对应表项的  $type$  域置为  $t$ ， $offset$  域置为  $o$ 。

我们认真分析一下上面翻译模式中数组类型和变量相关的几个重要属性。首先，数组类型的  $width$  属性，可以准确地表示相应数组变量需要分配存储空间的大小（字节数），如果这一信息会多次用到，最好也单独保存一份（通过符号表可以访问），以免需要时再通过类型属性  $type$  重新计算。

再看数组变量的类型信息  $type$ ，一般会保存在符号表中。实际上，数组类型信息一般不只是单个信息，而是一组信息。通过这些信息，可以计算由数组变量加下标的数组元素的引用位置，即在数组首元素基地址基础上的偏移量。此外，这些信息的存在，对数组的安全访问也由重要影响。

我们先抛开 MiniDecaf 语言，在编程语言中，最一般的情况下，对于  $n$  维静态数组可用  $A[l_1:u_1, l_2:u_2, \dots, l_n:u_n]$  声明，其中， $l_1, u_1, l_2, u_2, \dots, l_n, u_n$ ：  $l_i$  和  $u_i$  ( $1 \leq i \leq n$ ) 分别为第  $i$  维的下界和上界。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：

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

重新整理后可得： $D = a - C + V$ ，其中

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

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

这里， $C$  为常量，在生成数组元素地址时不用重复计算。当然，若是换作 MiniDecaf， $D$  和  $C$  的计算要简化很多，这里我们不去讨论，大家在完成 MiniDecaf 实验时一定会搞清楚的。

另外，上下界 ( $l_1, u_1, l_2, u_2, \dots, l_n, u_n$ ) 对于生成程序分析以及生成数组安全访问（如不越界）动态保护代码起到关键作用。

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；对于动态可变数组，将在运行时建立相应的内情向量。例如，对于上面的  $n$  维静态数组说明  $A[l_1:u_1, l_2:u_2, \dots, l_n:u_n]$ ，可以考虑在符号表中建立如下形式的内情向量：

- $l_1, u_1, l_2, u_2, \dots, l_n, u_n$ ：  $l_i$  和  $u_i$  ( $1 \leq i \leq n$ ) 分别为第  $i$  维的下界和上界。
- $type$ ：数组元素的类型。
- $a$ ：数组首元素的地址。
- $n$ ：数组维数。
- $size$ ：数组存储空间大小（字节数）。
- $C$ ：计算元素偏移地址时不变的部分，见随后的解释。

上述翻译模式中，还有一个属性  $offset$ （如  $D.offset$ ），是相对于当前上下文（全局存储空间或某函数的局部存储空间）基地址的偏移量。假设在这个位置声明了一个数组变量，如果是在全局静态区或者函数栈区分配空间，那么这个位置就是该数组首元素的地址，从这个地址开始分配大小为  $size$  的连续空间。前面的翻译模式中，都是假定了这种分配方式。在实际中，数组

变量往往是需要堆存储区动态分配空间，这种情况下，就需要生成动态申请内存的代码，可以通过调用相应的内置库函数或者系统库函数来完成。

在本学期 MiniDecaf 实验框架中，助教给出一种可能的实现方案（参见实验指导书），即添加一种申请分配内存的中间代码语句（TAC 语句），只要在处理声明数组变量时生成这个 TAC 语句即可，将真实用于调用存储分配函数的指令生成留给后端的目标代码生成阶段。大家回到 2.3 节的开始，我们设计了一条 Tac 语句 ALLOC，形如  $x := \text{alloc } y$ ，其含义是分配  $y$  个字节的存储空间，其起始存储位置值赋给  $x$ 。有了这条 Tac 语句，大家做实验时是比较省事了。但这一做法其实也并未回避掉需要选择什么样的存储分配方案，待方案明确后，我们才能正确描述中间代码生成的算法，即写出相应的翻译模式。这里有不少问题值得大家探究、讨论，本学期 MiniDecaf 实验框架中也仅仅是提供了一种可行的实现方案而已。

现在，假设我们的数组存储分配方案为：在堆区申请空间；在栈区使用两个单元，一个用于记录数组首元素地址，另一个用于记录数组存储空间大小。当然，根据需要可以增加栈区单元数目。这样的话，在上面的翻译模式中，我们需要修改  $D.width$  的计算方式。当所声明的变量是数组类型时，需要将原来的  $D.width := T.width$  修改为  $D.width := 2$ 。是否数组，可通过  $T.type$  来判别。

同时，我们也需要考虑数组表达式  $E_1[E_2]$  的中间代码生成。这样，我们用如下翻译模式来描述数组声明和数组元素引用的 TAC 代码生成算法：

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

其中， $\underline{id}.place$  表示相应的名字对应的存储位置；综合属性  $E.place$  表示存放  $E$  的值的存储位置；综合属性  $E.code$  表示对  $E$  进行求值的 TAC 语句序列；综合属性  $D.code$  表示对应于  $D$  的 TAC 语句序列。

语义函数  $\text{gen}$  的结果是生成一条 TAC 语句；语义函数  $\text{newtemp}$  的作用是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； $\parallel$  是 TAC 语句序列之间的链接运算。

### 2.3.3 表达式的翻译

以下翻译模式片断，可以产生相应于表达式（数组元素引用表达式  $E_1[E_2]$  在前面的小节已处理过）的 TAC 语句序列：

$$\begin{aligned}
E \rightarrow \underline{\text{int}} & \quad \{ E.place := \text{newtemp}; \quad E.code := \text{gen}(E.place \text{ ':=' } \underline{\text{int}}.val) \} \\
E \rightarrow \underline{\text{id}} & \quad \{ E.place := \underline{\text{id}}.place; \quad E.code := "" \} \\
E \rightarrow \underline{\text{uop}} E_1 & \quad \{ E.place := \text{newtemp}; \\
& \quad E.code := E_1.code \parallel \text{gen}(E.place \text{ ':=' } \underline{\text{uop}}.op E_1.place) \} \\
E \rightarrow E_1 \underline{\text{bop}} E_2 & \quad \{ E.place := \text{newtemp}; \\
& \quad E.code := E_1.code \parallel E_2.code \parallel \text{gen}(E.place \text{ ':=' } E_1.place \underline{\text{bop}}.op E_2.place) \} \\
E \rightarrow \underline{\text{top}}(E_1 E_2 E_3) & \quad \{ E.place := \text{newtemp}; L := \text{newlabel}; M := \text{newlabel}; \\
& \quad E.code := E_1.code \parallel \text{gen}(\text{'if' } E_1.place \text{ '<=' '0' 'goto' } L) \parallel \\
& \quad E_2.code \parallel \text{gen}(E.place \text{ ':=' } E_2.place) \parallel \text{gen}(\text{'goto' } M) \parallel \\
& \quad \text{gen}(L \text{ ':'}) \parallel E_3.code \parallel \text{gen}(E.place \text{ ':=' } E_2.place) \parallel \text{gen}(M \text{ ':'}) \} \\
E \rightarrow \underline{\text{id}}(A) & \quad \{ E.code := A.code; \\
& \quad \text{for } A.arglist \text{ 中的每一项 } d \text{ do} \\
& \quad \quad E.code := E.code \parallel \text{gen}(\text{'param' } d); \\
& \quad E.code := E.code \parallel \text{gen}(\text{'call' } \underline{\text{id}}.place) \} \\
A \rightarrow A_1, E & \quad \{ A.n := A_1.n + 1; A.arglist := \text{append}(A_1.arglist, \text{makelist}(E.place)); \}; \\
& \quad A.code := A_1.code \parallel E.code \} \\
A \rightarrow \varepsilon & \quad \{ A.n := 0; A.arglist := ""; A.code := "" \} \\
E \rightarrow \underline{\text{id}} = E_1 & \quad \{ E.code := E_1.code \parallel \text{gen}(\underline{\text{id}}.place \text{ ':=' } E_1.place) \} \\
E \rightarrow E_1 [E_2] = E_3 & \quad \{ E.code := E_1.code \parallel E_2.code \parallel E_3.code \parallel \\
& \quad \text{gen}(E_1.place \text{ '[' } E_2.place \text{ ']' ':=' } E_3.place) \}
\end{aligned}$$

其中,  $\underline{\text{id}}.place$ ,  $E.place$ ,  $E.code$ ,  $A.code$ , 语义函数  $gen$ , 以及运算  $\parallel$  的含义同前。语义函数  $\text{newlabel}$  将返回一个新的语句标号。性  $A.n$  记录参数个数; 属性  $A.arglist$  代表实参地址的列表; 语义函数  $\text{makelist}$  表示创建一个实参地址的结点; 语义函数  $\text{append}$  表示在已有实参地址列表中添加一个结点。语义动作中,  $\underline{\text{uop}}.op$  和  $\underline{\text{bop}}.op$  分别表示具体的一元和二元运算。

注: 在上述翻译模式中,  $\underline{\text{bop}}$  未包括赋值运算符, 对于赋值表达式, 这里是单独考虑的, 参加上面最后两条规则。

### 2.3.4 布尔表达式的翻译

对于布尔表达式的翻译, 一种方法是可以直接对布尔表达式进行求值。比如, 我们可以用数值“1”表示 true, 用数值“0”表示 false, 设计如下的 S-翻译模式片断:

$$\begin{aligned}
E \rightarrow E_1 \vee E_2 & \quad \{ E.place := \text{newtemp}; \\
& \quad E.code := E_1.code \parallel E_2.code \parallel \text{gen}(E.place \text{ ':=' } E_1.place \text{ 'or' } E_2.place) \} \\
E \rightarrow E_1 \wedge E_2 & \quad \{ E.place := \text{newtemp}; \\
& \quad E.code := E_1.code \parallel E_2.code \parallel \text{gen}(E.place \text{ ':=' } E_1.place \text{ 'and' } E_2.place) \} \\
E \rightarrow \neg E_1 & \quad \{ E.place := \text{newtemp}; \\
& \quad E.code := E_1.code \parallel \text{gen}(E.place \text{ ':=' 'not' } E_1.place) \} \\
E \rightarrow (E_1) & \quad \{ E.place := E_1.place; E.code := E_1.code \} \\
E \rightarrow \underline{\text{id}}_1 \underline{\text{rop}} \underline{\text{id}}_2 & \quad \{ E.place := \text{newtemp}; \\
& \quad E.code := \text{gen}(\text{'if' } \underline{\text{id}}_1.place \underline{\text{rop}}.op \underline{\text{id}}_2.place \text{ 'goto' } \text{nextstat}+3) \parallel \\
& \quad \text{gen}(E.place \text{ ':=' '0'}) \parallel \text{gen}(\text{'goto' } \text{nextstat}+2) \parallel \text{gen}(E.place \text{ ':=' '1'}) \} \\
E \rightarrow \text{true} & \quad \{ E.place := \text{newtemp}; \quad E.code := \text{gen}(E.place \text{ ':=' '1'}) \} \\
E \rightarrow \text{false} & \quad \{ E.place := \text{newtemp}; \quad E.code := \text{gen}(E.place \text{ ':=' '0'}) \}
\end{aligned}$$

其中,综合属性综合属性  $E.place$  表示存放  $E$  的值的存储位置;综合属性  $E.code$  表示对  $E$  进行求值的  $TAC$  语句序列,语义函数  $gen$ 、 $newtemp$ ,以及运算  $//$  的含义同 2.3.1 小节。语义函数  $nextstat$  返回输出代码序列中下一条  $TAC$  语句的下标。 $id1.place$  和  $id2.place$  表示相应的名字对应的存储位置; $rop.op$  表示相应关系运算符;下同。

以上这种做法与 MiniDecaf 以及 C 语言是可以对应的,即用“1”表示 true,用“0”表示 false。当然,如果有宏定义支持,我们完全可以定义 true 为“1”, false 为“1”,在正式代码中就可以像许多强类型语言中使用 boolean 类型那样使用 tru 和 false 了。这样,下面所介绍的与上面这种直接求值完全不同的方法也照样是能适用的。

翻译布尔表达式的另一种方法是通过控制流体现布尔表达式的语义,即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。这种方法的一个优点是可以方便实现控制流语句中布尔表达式的翻译,通常还可以得到**短路代码**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。考虑下列翻译模式片断:

$$\begin{aligned}
 E &\rightarrow \{ E_1.true := E.true; E_1.false := newlabel \} E_1 \vee \\
 &\quad \{ E_2.true := E.true; E_2.false := E.false \} E_2 \\
 &\quad \{ E.code := E_1.code // gen (E_1.false ':') // E_2.code \} \\
 E &\rightarrow \{ E_1.false := E.false; E_1.true := newlabel \} E_1 \wedge \\
 &\quad \{ E_2.false := E.false; E_2.true := E.true \} E_2 \\
 &\quad \{ E.code := E_1.code // gen (E_1.true ':') // E_2.code \} \\
 E &\rightarrow \neg \{ E_1.true := E.false; E_1.false := E.true \} E_1 \{ E.code := E_1.code \} \\
 E &\rightarrow ( \{ E_1.true := E.true; E_1.false := E.false \} E_1 ) \{ E.code := E_1.code \} \\
 E &\rightarrow id_1 \text{ rop } id_2 \quad \{ E.code := gen ('if' id_1.place \text{ rop } id_2.place 'goto' E.true) // \\
 &\quad \quad \quad gen ('goto' E.false) \} \\
 E &\rightarrow true \quad \{ E.code := gen ('goto' E.true) \} \\
 E &\rightarrow false \quad \{ E.code := gen ('goto' E.false) \}
 \end{aligned}$$

其中,综合属性  $E.code$ ,语义函数  $gen$ ,以及运算  $//$  的含义同前。调用语义函数  $newlabel$  将返回一个新的语句标号。继承属性  $E.true$  和  $E.false$  分别代表  $E$  为真和假时控制要转移到的程序位置,即标号。

这是一个 L-翻译模式。若规定运算  $\wedge$  优先于  $\vee$ ,且都为左结合,则可以基于 LR 分析构造一个翻译程序。若以布尔表达式  $E = a < b \vee c < d \wedge e < f$  为输入,那么可能的翻译结果形如:

```

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

```

其中,  $E.true$  和  $E.false$  会在表达式  $E$  的上下文中确定,参见下一小节。

### 2.3.5 语句的翻译

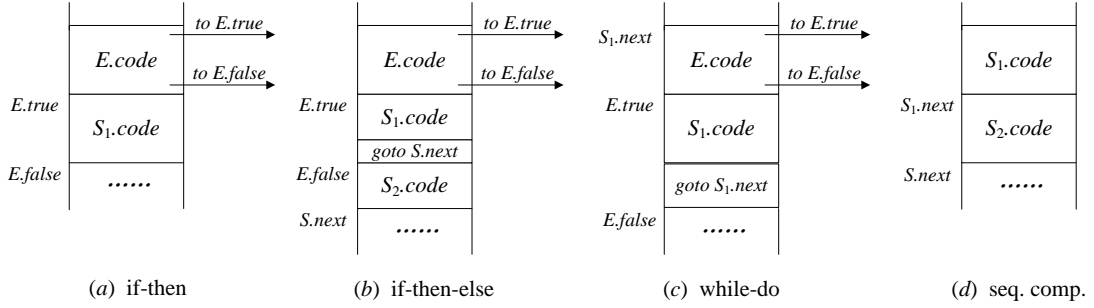
以下是一个  $L$ -翻译模式片断，可以产生语句（为简洁，先不考虑 *break* 和 *continue* 语句）的 TAC 语句序列：

$$\begin{aligned}
 S &\rightarrow \text{if } ( \{ E.true := \text{newlabel}; E.false := S.next \} E ) \\
 &\quad \{ S_1.next := S.next \} S_1 \{ S.code := E.code // \text{gen}(E.true ':') // S_1.code \}^4 \\
 S &\rightarrow \text{if } ( \{ E.true := \text{newlabel}; E.false := \text{newlabel} \} E ) \quad \{ S_1.next := S.next \} \quad S_1 \\
 &\quad \{ S_2.next := S.next \} \quad S_2 \quad \{ S.code := E.code // \text{gen}(E.true ':') // S_1.code // \\
 &\quad \quad \text{gen}('goto' S.next) // \text{gen}(E.false ':') // S_2.code \} \\
 S &\rightarrow \text{while } ( \{ E.true := \text{newlabel}; E.false := S.next \} E ) \{ S_1.next := \text{newlabel} \} S_1 \\
 &\quad \{ S.code := \text{gen}(S_1.next ':') // E.code // \text{gen}(E.true ':') // S_1.code // \text{gen}('goto' S_1.next) \} \\
 S &\rightarrow \text{do } \{ S_1.next := \text{newlabel} \} S_1 \text{ while } ( \{ E.true := \text{newlabel}; E.false := S.next \} E ) \\
 &\quad \{ S.code := \text{gen}(E.true ':') // S_1.code // \text{gen}(S_1.next ':') // E.code \} \\
 S &\rightarrow \text{for } ( \{ R_1.next := \text{newlabel} \} \quad R_1 ; \\
 &\quad \{ E.true := \text{newlabel}; E.false := S.next \} \quad E ; \\
 &\quad \{ R_2.next := R_1.next \} \quad R_2 ) \\
 &\quad \{ S_1.next := \text{newlabel} \} \quad S_1 \\
 &\quad \{ S.code := R_1.code // \text{gen}(R_1.next ':') // E.code // \text{gen}(E.true ':') \\
 &\quad \quad // S_1.code // \text{gen}(S_1.next ':') // R_2.code // \text{gen}('goto' R_1.next) \} \\
 S &\rightarrow \text{for } ( \{ D.next := \text{newlabel} \} \quad D ; \\
 &\quad \{ E.true := \text{newlabel}; E.false := S.next \} \quad E ; \\
 &\quad \{ R.next := D.next \} \quad R ) \\
 &\quad \{ S_1.next := \text{newlabel} \} \quad S_1 \\
 &\quad \{ S.code := D.code // \text{gen}(D.next ':') // E.code // \text{gen}(E.true ':') \\
 &\quad \quad // S_1.code // \text{gen}(S_1.next ':') // R.code // \text{gen}('goto' D.next) \} \\
 R &\rightarrow E \quad \{ R.code := E.code \} \\
 R &\rightarrow \varepsilon \quad \{ R.code := \varepsilon \} \\
 S &\rightarrow E \quad \{ S.code := E.code \} \\
 S &\rightarrow \text{return } E \quad \{ \text{gen}('return' E.place) \} \\
 S &\rightarrow D \quad \{ S.code := D.code \} \\
 S &\rightarrow \{ \} \{ S_1.next := S.next \} S_1 \{ \} \quad \{ S.code := S_1.code \} \\
 S &\rightarrow \{ S_1.next := \text{newlabel} \} S_1 \{ S_2.next := S.next \} S_2 \\
 &\quad \{ S.code := S_1.code // \text{gen}(S_1.next ':') // S_2.code \} \\
 S &\rightarrow \varepsilon \quad \{ S.code := \varepsilon \}
 \end{aligned}$$

其中，综合属性  $E.code$  和  $S.code$ ，继承属性  $E.true$  和  $E.false$ ，语义函数  $gen$ ， $newlabel$ ，以及运算  $//$  的含义同前。继承属性  $S.next$  代表退出  $S$  时控制要转移到的语句标号。

上述翻译模式片断中，控制语句设计思路略微复杂，需要是先想清楚实现方案。图 5 中我们选择了 4 中典型的控制语句，刻画了由  $E.code$ ， $S.code$ ， $E.true$ ， $E.false$  和  $S.next$  等属性刻画其基本语义以及翻译算法。其余控制语句的情形可类比。

<sup>4</sup> 注：图 1 描述的简单语言不含 If-Else 结构的条件语句，但这一小节还保留了以前课件的这一内容，也是出于尽可能保持一定程度的完整性。



**图 5 控制语句的翻译**

最后，我们补充如下翻译模式片断，以生成完整程序的 TAC 序列：

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

这里，语义函数 *addplace* 将标识符存储位置填入符号表中相应表项的 *place* 域。另外，处理每个函数时，会生成一条指导（directive）信息，memo ‘XXX’，可看作特殊的 TAC 语句，其中字符串信息‘XXX’记录有该函数所有参数距离栈帧基址的偏移量信息，在 MiniDecaf 实验框架中被用于确认函数调用时参数在栈上的位置。语义函数 *output* 用于输出或者返回所生成的 TAC 语句序列。

下列翻译模式片断考虑了对 *break* 语句<sup>5</sup>的处理，仅列出了有变化的产生式：

$$\begin{aligned}
 F &\rightarrow \text{int } \underline{\text{id}} ( L ) \{ S.\text{next} := \text{newlabel}; S.\text{break} := \text{newlabel} \} S \\
 &\quad \{ S.\text{code} := \text{gen}(\text{'memo' } ''L.\text{code} '') \parallel S.\text{code} \parallel \text{gen}(S.\text{next} ':') \} \\
 S &\rightarrow \text{if} ( \{ E.\text{true} := \text{newlabel}; E.\text{false} := S.\text{next} \} E ) \\
 &\quad \{ S_1.\text{next} := S.\text{next}; S_1.\text{break} := S.\text{break} \} S_1 \\
 &\quad \{ S.\text{code} := E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \}^6 \\
 S &\rightarrow \text{if} ( \{ E.\text{true} := \text{newlabel}; E.\text{false} := \text{newlabel} \} E ) \\
 &\quad \{ S_1.\text{next} := S.\text{next}; S_1.\text{break} := S.\text{break} \} S_1 \\
 &\quad \{ S_2.\text{next} := S.\text{next}; S_2.\text{break} := S.\text{break} \} S_2 \\
 &\quad \{ S.\text{code} := E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \parallel \\
 &\quad \quad \text{gen}(\text{'goto' } S.\text{next}) \parallel \text{gen}(E.\text{false} ':') \parallel S_2.\text{code} \} \\
 S &\rightarrow \text{while} ( \{ E.\text{true} := \text{newlabel}; E.\text{false} := S.\text{next} \} E ) \\
 &\quad \{ S_1.\text{next} := \text{newlabel}; S_1.\text{break} := S.\text{next} \} S_1 \\
 &\quad \{ S.\text{code} := \text{gen}(S_1.\text{next} ':') \parallel E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \parallel \\
 &\quad \quad \text{gen}(\text{'goto' } S_1.\text{next}) \}
 \end{aligned}$$

<sup>5</sup> 可参照实现对 *continue* 语句的处理，留作练习。

<sup>6</sup> 注：图 1 描述的简单语言不含 If-Else 结构的条件语句，但这一小节还保留了以前课件的这一内容，也是出于尽可能保持一定程度的完整性。

```

 $S \rightarrow \text{do } \{ S_1.\text{next} := \text{newlabel}; S_1.\text{break} := S.\text{next} \} S_1 \text{ while } ($ 
 $\quad \{ E.\text{true} := \text{newlabel}; E.\text{false} := S.\text{next} \} E)$ 
 $\quad \{ S.\text{code} := \text{gen}(E.\text{true} ':') // S_1.\text{code} // \text{gen}(S_1.\text{next} ':') // E.\text{code} \}$ 
 $S \rightarrow \text{for } ( \{ R_1.\text{next} := \text{newlabel} \} R_1 ;$ 
 $\quad \{ E.\text{true} := \text{newlabel}; E.\text{false} := S.\text{next} \} E ;$ 
 $\quad \{ R_2.\text{next} := R_1.\text{next} \} R_2 )$ 
 $\quad \{ S_1.\text{next} := \text{newlabel}; S_1.\text{break} := S.\text{next} \} S_1$ 
 $\quad \{ S.\text{code} := R_1.\text{code} // \text{gen}(R_1.\text{next} ':') // E.\text{code} // \text{gen}(E.\text{true} ':')$ 
 $\quad \quad // S_1.\text{code} // \text{gen}(S_1.\text{next} ':') // R_3.\text{code} // \text{gen}(\text{'goto' } R_1.\text{next}) \}$ 
 $S \rightarrow \text{for } ( \{ D.\text{next} := \text{newlabel} \} D ;$ 
 $\quad \{ E.\text{true} := \text{newlabel}; E.\text{false} := S.\text{next} \} E ;$ 
 $\quad \{ R.\text{next} := D.\text{next} \} R )$ 
 $\quad \{ S_1.\text{next} := \text{newlabel}; S_1.\text{break} := S.\text{next} \} S_1$ 
 $\quad \{ S.\text{code} := D.\text{code} // \text{gen}(D.\text{next} ':') // E.\text{code} // \text{gen}(E.\text{true} ':')$ 
 $\quad \quad // S_1.\text{code} // \text{gen}(S_1.\text{next} ':') // R.\text{code} // \text{gen}(\text{'goto' } D.\text{next}) \}$ 
 $S \rightarrow \text{'}' \{ S_1.\text{next} := S.\text{next}; S_1.\text{break} := S.\text{break} \} S_1 \text{'}' \quad \{ S.\text{code} := S_1.\text{code} \}$ 
 $S \rightarrow \{ S_1.\text{next} := \text{newlabel}; S_1.\text{break} := S.\text{break} \} S_1$ 
 $\quad \{ S_2.\text{next} := S.\text{next}; S_2.\text{break} := S.\text{break} \} S_2$ 
 $\quad \{ S.\text{code} := S_1.\text{code} // \text{gen}(S_1.\text{next} ':') // S_2.\text{code} \}$ 
 $S \rightarrow \text{break} \quad \{ S.\text{code} := \text{gen}(\text{'goto' } S.\text{break}) \}$ 

```

注：对于不被 while 包围的语句  $S$ ， $S.\text{break}$  可取任意标号（因为已经过静态语义检查，故  $S$  不可能是 break 语句）。

### 2.3.6 拉链与代码回填（选讲）

前面两小节里，我们设计了将布尔表达式和控制语句翻译为 TAC 语句序列的 L-翻译模式片断（通过控制流体现布尔表达式的语义）。这一小节里，我们介绍一种可处理同样问题的 S-翻译模式。这一翻译模式用到下列属性和语义函数：

- 综合属性  $E.\text{truelist}$ （真链）：表示一系列跳转语句的地址，这些跳转语句的目标语句标号是体现布尔表达式  $E$  为“真”的标号。
- 综合属性  $E.\text{falselist}$ （假链）：表示一系列跳转语句的地址，这些跳转语句的目标语句标号是体现布尔表达式  $E$  为“假”的标号。
- 综合属性  $S.\text{nextlist}$ （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语句标号是在执行序列中紧跟在  $S$  之后的下条 TAC 语句的标号。综合属性  $N.\text{nextlist}$  是仅含一个语句地址的链表，对应于处理到  $N$  时的跳转语句。
- 综合属性  $S.\text{breaklist}$ （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语句标号是跳出直接包围  $S$  的 while 语句后的下条 TAC 语句的标号。
- 综合属性  $M.\text{gotostm}$  中记录处理到  $M$  时下一条待生成语句的标号。
- 语义函数  $\text{makelist}(i)$ ：创建只有一个结点  $i$  的表，对应于一条跳转语句的地址。
- 语义函数  $\text{merge}(p_1, p_2)$ ：链接两个链表  $p_1$  和  $p_2$ ，返回结果链表。



- 语义函数  $backpatch(p, i)$ : 将链表  $p$  中每个元素所指向的跳转语句的标号置为  $i$ 。
- 语义函数  $nextstm$ : 返回下一条 TAC 语句的地址。
- 语义函数  $emit(...)$ : 输出一条 TAC 语句, 并使  $nextstm$  加 1。

我们先来看处理布尔表达式的 S-翻译模式片段:

$$\begin{aligned}
 E \rightarrow E_1 \vee M E_2 & \quad \{ backpatch(E_1.falselist, M.gotostm); \\
 & \quad E.truelist := merge(E_1.truelist, E_2.truelist); E.falselist := E_2.falselist \} \\
 E \rightarrow E_1 \wedge M E_2 & \quad \{ backpatch(E_1.truelist, M.gotostm); \\
 & \quad E.falselist := merge(E_1.falselist, E_2.falselist); E.truelist := E_2.truelist \} \\
 E \rightarrow \neg E_1 & \quad \{ E.truelist := E_1.falselist; E.falselist := E_1.truelist \} \\
 E \rightarrow (E_1) & \quad \{ E.truelist := E_1.truelist; E.falselist := E_1.falselist \} \\
 E \rightarrow \underline{id_1} \text{ rop } \underline{id_2} & \quad \{ E.truelist := makelist(nextstm); E.falselist := makelist(nextstm+1); \\
 & \quad emit('if' \underline{id_1}.place \text{ rop.op } \underline{id_2}.place \text{ 'goto' } \_); emit('goto \_') \} \\
 E \rightarrow \text{true} & \quad \{ E.truelist := makelist(nextstm); emit('goto \_') \} \\
 E \rightarrow \text{false} & \quad \{ E.falselist := makelist(nextstm); emit('goto \_') \} \\
 M \rightarrow \varepsilon & \quad \{ M.gotostm := nextstm \}
 \end{aligned}$$

这个翻译模式使用了所谓的**代码回填**技术: 当处理到某一步, 生成的转移语句不能确定目标语句标号时, 先将目标语句标号的位置用 ‘\_’ 表示, 并将该转移语句的地址加入到某个链表(真链、假链、*next* 链)中; 当这个目标语句标号可以确定之时, 再将其回填至 ‘\_’ 处。例如, 对于产生式  $E \rightarrow E_1 \vee M E_2$ , 在产生  $E_1$  部分的代码时,  $E_1$  求值为 true 或 false 时转移语句的目标语句标号不能确定, 所以将转移语句的地址加入到  $E_1.truelist$  或  $E_1.falselist$  之中; 在处理到  $M$  时, 当前得到的综合属性值  $M.gotostm$  正是  $E_1$  求值为 false 时应该转移到的目标语句标号, 因此执行语义动作  $backpatch(E_1.falselist, M.gotostm)$  将  $M.gotostm$  回填至  $E_1.falselist$  中的所有转移语句; 另外, 需要将  $E_1.truelist$  中的所有转移语句地址合并到  $E.truelist$  之中, 待将来  $E$  求值为 true 时应该转移到的目标语句标号确定后, 再回填给这些转移语句。

我们来看一个简单的例子。若以布尔表达式  $E = a < b \vee c < d \wedge e < f$  为输入, 那么基于这个翻译模式的翻译过程和翻译结果如图 6 所示(规定运算  $\wedge$  优先于  $\vee$ )。

在归约  $a < b$  时生成语句 (0) 和 (1), 归约  $c < d$  时生成语句 (2) 和 (3), 归约  $e < f$  时生成语句 (4) 和 (5), 但都不能确定目标语句标号。在按照产生式  $E \rightarrow E_1 \wedge M E_2$  进行归约时, 将当前  $M.gotostm$  中记录的语句标号 (4) 回填至当前  $E_1.truelist$  中记录的所有转移语句, 结果使得语句 (2) 中的目标语句标号被替换为 (4)。同理, 在按照产生式  $E \rightarrow E_1 \vee M E_2$  进行归约时, 使得语句 (1) 中的目标语句标号被替换为 (2)。

在处理完整个表达式  $E$  之后, 语句 (0), (3), (4) 和 (5) 的目标语句标号仍未确定, 但这些语句已被记录在  $E$  的真链和假链之中:  $E.truelist = \{0, 4\}$ ,  $E.falselist = \{3, 5\}$ 。

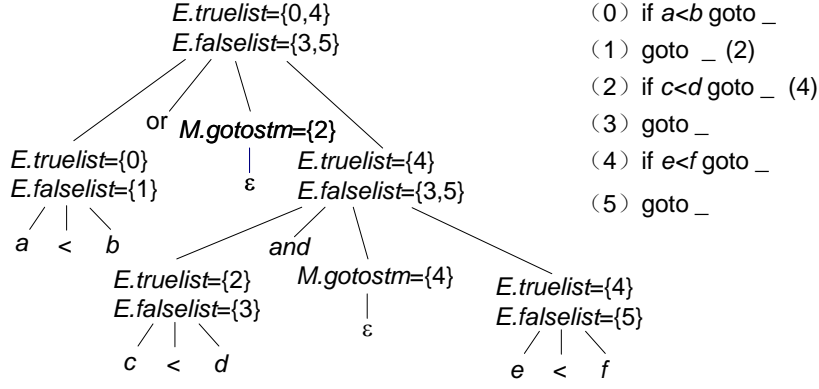


图 6 拉链与代码回填

我们再来看处理语句（为简洁，我们仅选择部分语句，控制语句仅保留图 5 中的 4 种，并且先不考虑 *break* 语句）的 *S*-翻译模式片段：

$$\begin{aligned}
 F &\rightarrow \text{int id} ( L ) S M \quad \{ \text{backpatch}(S.\text{nextlist}, M.\text{gotostm}) \} \\
 S &\rightarrow \text{if} ( E ) M S_1 \quad \{ \text{backpatch}(E.\text{truelist}, M.\text{gotostm}) ; \\
 &\quad S.\text{nextlist} := \text{merge}(E.\text{falselist}, S_1.\text{nextlist}) \} \\
 S &\rightarrow \text{if} ( E ) M_1 S_1 N M_2 S_2 \quad \{ \text{backpatch}(E.\text{truelist}, M_1.\text{gotostm}) ; \\
 &\quad \text{backpatch}(E.\text{falselist}, M_2.\text{gotostm}) ; \\
 &\quad S.\text{nextlist} := \text{merge}(S_1.\text{nextlist}, \text{merge}(N.\text{nextlist}, S_2.\text{nextlist})) \} \\
 S &\rightarrow \text{while} ( M_1 E ) M_2 S_1 \quad \{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{gotostm}) ; \\
 &\quad \text{backpatch}(E.\text{truelist}, M_2.\text{gotostm}) ; \\
 &\quad S.\text{nextlist} := E.\text{falselist}; \text{emit}('goto', M_1.\text{gotostm}) \} \\
 S &\rightarrow \{ S_1 \} \quad \{ S.\text{nextlist} := S_1.\text{nextlist} \} \\
 S &\rightarrow B \quad \{ S.\text{nextlist} := B.\text{nextlist} \} \quad // B \text{ 表示赋值和算术表达式 (并不排除关系和逻辑运算)} \\
 S &\rightarrow S_1 M S_2 \quad \{ \text{backpatch}(S_1.\text{nextlist}, M.\text{gotostm}) ; S.\text{nextlist} := S_2.\text{nextlist} \} \\
 M &\rightarrow \epsilon \quad \{ M.\text{gotostm} := \text{nextstm} \} \\
 N &\rightarrow \epsilon \quad \{ N.\text{nextlist} := \text{makelist}(\text{nextstm}); \text{emit}('goto _') \}
 \end{aligned}$$

再对这一 *S*-翻译模式片段增加 *break* 语句<sup>7</sup>的处理，仅列出有变化的产生式：

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

<sup>7</sup> 可参照实现对 *continue* 语句的处理，留作练习。

$$\begin{aligned}
& \text{emit('goto', } M_1.\text{gotostm} \text{)} \} \\
S \rightarrow \{ S_1 \} & \quad \{ S.\text{nextlist} := S_1.\text{nextlist} ; S.\text{breaklist} := S_1.\text{breaklist} \} \\
S \rightarrow B & \quad \{ S.\text{nextlist} := B.\text{nextlist} ; S.\text{breaklist} := \varepsilon \} \\
S \rightarrow S_1 M S_2 & \quad \{ \text{backpatch}(S_1.\text{nextlist}, M.\text{gotostm}) ; \\
& \quad S.\text{nextlist} := S_2.\text{nextlist} ; S.\text{breaklist} := \text{merge}(S_1.\text{breaklist}, S_2.\text{breaklist}) \} \\
S \rightarrow \text{break} & \quad \{ S.\text{breaklist} := \text{makelist}(\text{nextstm}) ; S.\text{nextlist} := \varepsilon ; \text{emit('goto _')} \} \\
M \rightarrow \varepsilon & \quad \{ M.\text{gotostm} := \text{nextstm} \} \\
N \rightarrow \varepsilon & \quad \{ N.\text{nextlist} := \text{makelist}(\text{nextstm}) ; \text{emit('goto _')} \}
\end{aligned}$$

最后，我们需要调整一下关于赋值和算术表达式的翻译模式的设计风格：

$$\begin{aligned}
B \rightarrow \underline{\text{id}} = B_1 & \quad \{ \text{emit}(\underline{\text{id}}.\text{place} \text{ '}' B_1.\text{place}) ; B.\text{nextlist} := \varepsilon \} \\
B \rightarrow \underline{\text{int}} & \quad \{ B.\text{place} := \text{newtemp} ; \text{emit}(B.\text{place} \text{ '}' \underline{\text{int}}.\text{val}) ; B.\text{nextlist} := \varepsilon \} \\
B \rightarrow \underline{\text{id}} & \quad \{ B.\text{place} := \underline{\text{id}}.\text{place} ; B.\text{nextlist} := \varepsilon \} \\
B \rightarrow \underline{\text{uop}} B_1 & \quad \{ B.\text{place} := \text{newtemp} ; \text{emit}(B.\text{place} \text{ '}' \underline{\text{uop}}.\text{op } B_1.\text{place}) ; B.\text{nextlist} := \varepsilon \} \\
B \rightarrow B_1 \underline{\text{bop}} B_2 & \quad \{ B.\text{place} := \text{newtemp} ; \text{emit}(B.\text{place} \text{ '}' B_1.\text{place } \underline{\text{bop}}.\text{op } B_2.\text{place}) ; \\
& \quad B.\text{nextlist} := \varepsilon \} \\
B \rightarrow \underline{\text{top}} ( E M_1 B_1 N M_2 B_2 ) & \quad \{ \text{backpatch}(E.\text{truelist}, M_1.\text{gotostm}) ; \\
& \quad \text{backpatch}(E.\text{falselist}, M_2.\text{gotostm}) ; B.\text{nextlist} := N.\text{nextlist} \} \\
M \rightarrow \varepsilon & \quad \{ M.\text{gotostm} := \text{nextstm} \} \\
N \rightarrow \varepsilon & \quad \{ N.\text{nextlist} := \text{makelist}(\text{nextstm}) ; \text{emit('goto _')} \}
\end{aligned}$$

注：这里， $\underline{\text{uop}}.\text{op}$  和  $\underline{\text{bop}}.\text{op}$  并不排除关系和逻辑运算。

## 练习

- 1 参考 2.3.4 节采用短路代码进行布尔表达式翻译的  $L$ -翻译模式片断及所用到的语义函数。若在基础文法中增加产生式  $E \rightarrow E \uparrow E$ ，试给出相应产生式的语义动作集合。其中，“ $\uparrow$ ”代表“与非”逻辑算符，其语义可用其它逻辑运算定义为  $P \uparrow Q \equiv \text{not}(P \text{ and } Q)$ 。
- 2 参考 2.3.5 节进行语句（不含 **break**）翻译的  $L$ -翻译模式片断及所用到的语义函数。若在基础文法中增加产生式  $S \rightarrow \text{repeat } S \text{ until } E$ ，试给出相应产生式的语义动作集合。

注：控制语句 **repeat** <循环体> **until** <布尔表达式> 的语义为：至少执行 <循环体> 一次，直到 <布尔表达式> 成真时结束循环。

- 3 参考 2.3.6 节采用拉链与代码回填技术进行布尔表达式和语句（不含 **break**）翻译的  $S$ -翻译模式片断及所用到的语义函数，重复题 1 和题 2 的工作。
- 4 参考 2.3.6 节采用拉链与代码回填技术进行布尔表达式和语句（不含 **break**）翻译的  $S$ -翻译模式片断及所用到的语义函数，以及参考 2.3.5 节对 **do-while** 循环以及两种 **for**-循环语句的处理，为 2.3.6 节的翻译模式增加对 **do-while** 循环以及两种 **for**-循环语句的处理。
- 5 对于第 4 题的问题，再增加 **break** 语句的处理。
- 6 参考 2.3.6 节采用拉链与代码回填技术进行布尔表达式翻译的  $S$ -翻译模式片断及所用到的语义函数。若在基础文法中增加产生式

$$E \rightarrow \Delta ( E, E, E )$$

其中“ $\Delta$ ”代表一个三元逻辑运算符。逻辑表达式  $\Delta ( E_1, E_2, E_3 )$  的语义可由下表定义：

| $E_1$ | $E_2$ | $E_3$ | $\Delta ( E_1, E_2, E_3 )$ |
|-------|-------|-------|----------------------------|
| false | false | false | false                      |
| false | false | true  | false                      |
| false | true  | false | true                       |
| false | true  | true  | false                      |
| true  | false | false | false                      |
| true  | false | true  | false                      |
| true  | true  | false | false                      |
| true  | true  | true  | false                      |

试给出相应产生式的语义处理部分（必要时增加文法符号，类似 2.3.6 节示例中的符号  $M$  和  $N$ ），不改变 S-属性文法（翻译模式）的特征。

- 7 重复上题的工作，若逻辑表达式  $\Delta ( E_1, E_2, E_3 )$  的语义由下表定义：

| $E_1$ | $E_2$ | $E_3$ | $\Delta ( E_1, E_2, E_3 )$ |
|-------|-------|-------|----------------------------|
| false | false | false | false                      |
| false | false | true  | false                      |
| false | true  | false | true                       |
| false | true  | true  | true                       |
| true  | false | false | true                       |
| true  | false | true  | false                      |
| true  | true  | false | true                       |
| true  | true  | true  | false                      |

- 8 设有开关语句

```

switch A of
    case  $d_1$  :  $S_1$  ;
    case  $d_2$  :  $S_2$  ;
    .....
    case  $d_n$  :  $S_n$  ;
    default  $S_{n+1}$ 
end

```

假设其具有如下执行语义：

- (1) 对算术表达式  $A$  进行求值；
- (2) 若  $A$  的取值为  $d_1$ ，则执行  $S_1$ ，转 (3)；  
否则，若  $A$  的取值为  $d_2$ ，则执行  $S_2$ ，转 (3)；  
.....  
否则，若  $A$  的取值为  $d_n$ ，则执行  $S_n$ ，转 (3)；  
否则，执行  $S_{n+1}$ ，转 (3)；
- (3) 结束该开关语句的执行。

若在基础文法中增加关于开关语句的下列产生式

$$\begin{aligned} S &\rightarrow \text{switch } A \text{ of } L \text{ end} \\ L &\rightarrow \text{case } V : S ; L \\ L &\rightarrow \text{default } S \\ V &\rightarrow d \end{aligned}$$

其中，终结符  $d$  代表常量，其属性值可由词法分析得到，以  $d.lexval$  表示。 $A$  是生成算术表达式的非终结符（对应2.3.1中的  $A$ ）。

试参考 2.3.5节进行控制语句（不含break）翻译的  $L$ -翻译模式片断及所用到的语义函数，给出相应的语义处理部分（不改变  $L$ -翻译模式的特征）。

注：可设计增加新的属性，必要时给出解释。

- 9 参考 2.3.5 节和 2.3.6 节所中关于控制语句（不含 break）翻译的两类翻译模式中的任何一种及所用到的语义函数，给出下列控制语句的一个翻译模式片断：

(a) 在基础文法中增加关于串行条件卫士语句的下列产生式

$$\begin{aligned} S &\rightarrow \text{if } G \text{ fi} \\ G &\rightarrow E : S \square G \\ G &\rightarrow E : S \end{aligned}$$

注：串行条件卫士语句的一般形式如

$$\text{if } E_1 : S_1 \square E_2 : S_2 \square \dots \square E_n : S_n \text{ fi}$$

我们将其语义解释为：

- (1) 依次判断布尔表达式  $E_1$  ,  $E_2$  , ...,  $E_n$  的计算结果。
- (2) 若计算结果为 true 的第一个表达式为  $E_k$  ( $1 \leq k \leq n$ )，则执行语句  $S_k$ ；执行后转 (4)。
- (3) 若  $E_1$  ,  $E_2$  , ...,  $E_n$  的计算结果均为 false，则直接转 (4)。
- (4) 跳出该语句。

(b) 在基础文法中增加关于串行循环卫士语句的下列产生式

$$\begin{aligned}
S &\rightarrow do\ G\ od \\
G &\rightarrow E : S \square G \\
G &\rightarrow E : S
\end{aligned}$$

注：串行循环卫士语句的一般形式如

$$do\ E_1 : S_1 \square E_2 : S_2 \square \dots \square E_n : S_n\ od$$

我们将其语义解释为：

- (1) 依次判断布尔表达式  $E_1$  ,  $E_2$  , ...,  $E_n$  的计算结果。
- (2) 若计算结果为 **true** 的第一个表达式为  $E_k$  ( $1 \leq k \leq n$ )，则执行语句  $S_k$ ；转 (1)。
- (3) 若  $E_1$  ,  $E_2$  , ...,  $E_n$  的计算结果均为 **false**，则跳出循环。

10 以下是语法制导生成 TAC 语句的一个 L-属性文法片断：

```

S → if E then S1
    { E.case := false ;
      E.label := S.next ;
      S1.next := S.next ;
      S.code := E.code // S1.code // gen(S.next ':')
    }

S → if E then S1 else S2
    { E.case := false ;
      E.label := newlabel;
      S1.next := S.next ;
      S2.next := S.next ;
      S.code := E.code // S1.code // gen('goto' S.next) // gen(E.label ':')
                // S2.code // gen(S.next ':')
    }

S → while E do S1
    { E.case := false ;
      E.label := S.next ;
      S1.next := newlabel ;
      S.code := gen(S1.next ':') // E.code // S1.code // gen('goto' S1.next) // gen(S.next ':')
    }

S → S1; S2
    { S1.next := newlabel ;
      S2.next := S.next ;
      S.code := S1.code // S2.code // gen(S.next ':')
    }

E → E1 or E2
    { E2.label := E.label ;

```

```

 $E_2$  .case :=  $E$  .case ;
 $E_1$  .case := true ;
if  $E$  .case {
     $E_1$  .label :=  $E$  .label;
     $E$  .code :=  $E_1$  .code //  $E_2$  .code }
else {
     $E_1$  .label := newlabel ;
     $E$  .code :=  $E_1$  .code //  $E_2$  .code // gen( $E_1$  .label ':' ) }
}

```

```

 $E \rightarrow E_1$  and  $E_2$ 
{  $E_2$  .label :=  $E$  .label ;
   $E_2$  .case :=  $E$  .case ;
   $E_1$  .case := false ;
  if  $E$  .case {
       $E_1$  .label := newlabel ;
       $E$  .code :=  $E_1$  .code //  $E_2$  .code // gen( $E_1$  .label ':' ) }
  else {
       $E_1$  .label :=  $E$  .label;
       $E$  .code :=  $E_1$  .code //  $E_2$  .code }
}

```

```

 $E \rightarrow$  not  $E_1$ 
{  $E_1$  .label :=  $E$  .label;
   $E_1$  .case := not  $E$  .case;
   $E$  .code :=  $E_1$  .code
}

```

```

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

```

```

 $E \rightarrow \underline{id_1}$  rop  $\underline{id_2}$ 
{
  if  $E$  .case {
       $E$  .code := gen('if'   $\underline{id_1}$  .place rop.op  $\underline{id_2}$  .place 'goto'  $E$  .label)  }
  else {
       $E$  .code := gen('if'   $\underline{id_1}$  .place rop.not-op  $\underline{id_2}$  .place 'goto'  $E$  .label)  }
}
// 这里，rop.not-op 是 rop.op 的补运算，例=和≠， < 和 ≥ ， > 和 ≤ 互为补运算

```

```

 $E \rightarrow$  true
{
  if  $E$  .case {

```

```

        E.code := gen( 'goto' E.label)  }
    }
E → false
{
    if not E.case {
        E.code := gen( 'goto' E.label)  }
    }

```

其中，属性  $S.code$ ,  $E.code$ ,  $S.next$ , 语义函数  $newlabel$ ,  $gen$ , 以及所涉及到的TAC 语句与讲稿中一致，“//”表示TAC语句序列的拼接；如下是对属性  $E.case$  和  $E.label$  的简要说明：

$E.case$ ：取逻辑值 **true** 和 **false**之一（**not** 是相应的“非”逻辑运算）

$E.label$ ：布尔表达式  $E$  的求值结果为  $E.case$  时，应该转去的语句标号

(a) 若在基础文法中增加产生式  $E \rightarrow E \uparrow E$ ，其中“ $\uparrow$ ”代表“与非”逻辑运算符，试参考上述布尔表达式的处理方法，给出相应的语义处理部分。

注：“与非”逻辑运算的语义可用其它逻辑运算定义为  $P \uparrow Q \equiv \text{not} (P \text{ and } Q)$

(b) 若在基础文法中增加产生式  $S \rightarrow \text{repeat } S \text{ until } E$ ，试参考上述控制语句的处理方法，给出相应的的语义处理部分。

注： $\text{repeat} \langle \text{循环体} \rangle \text{ until } \langle \text{布尔表达式} \rangle$  至少执行 $\langle \text{循环体} \rangle$ 一次，直到 $\langle \text{布尔表达式} \rangle$ 成真时结束循环

11. (a) 以下是一个L-翻译模式片断，描述了某小语言中语句相关的一种类型检查工作：

```

P → D ; S      { P.type := if D.type = ok and S.type = ok then ok else type_error }
S → if E then S1 { S.type := if E.type=bool then S1.type else type_error }
S → while E then S1 { S.type := if E.type=bool then S1.type else type_error }
S → S1 ; S2    { S.type := if S1.type = ok and S2.type = ok then ok else type_error }

```

其中，**type** 属性以及类型表达式 **ok**, **type\_error**, **bool** 等的含义与讲稿中一致。

（此外，假设在语法制导处理的分析过程中遇到的冲突问题均可以按照某种原则处理，这里不必考虑基础文法是否 **LR** 文法。）

下面叙述本小题的要求：

若在基础文法中增加关于“带条件的继续循环”语句的产生式

$S \rightarrow \text{continue when } E$

语句 **continue when  $E$**  只能出现在某个循环语句内，即至少有一个包围它的 **while**；同时  $E$  必须为 **bool** 类型。

试在上述翻译模式片段基础上增加相应的的语义处理内容（要求是 **L-翻译模式**），以实现针对“带条件的继续循环”语句的这一类型检查任务。（提示：可以引入  $S$  的一个继承属性）



(b) 以下是一个L-翻译模式片断，可以产生控制语句的 TAC 语句序列：

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

其中，属性  $S.code$ ,  $E.code$ ,  $S.next$ ,  $E.true$ ,  $E.false$ , 语义函数  $newlabel$ ,  $gen()$  以及所涉及到的 TAC 语句与讲稿中一致：继承属性  $E.true$  和  $E.false$  分别代表  $E$  为真和假时控制要转移到的程序位置，即标号；综合属性  $E.code$  表示对  $E$  进行求值的 TAC 语句序列；综合属性  $S.code$  表示对应于  $S$  的 TAC 语句序列；继承属性  $S.next$  代表退出  $S$  时控制要转移到的语句标号。语义函数  $gen$  的结果是生成一条 TAC 语句；“ $\parallel$ ”表示 TAC 语句序列的拼接。

（此外，假设在语法制导处理的分析过程中遇到的冲突问题均可以按照某种原则处理，这里不必考虑基础文法是否 LR 文法。）

下面叙述本小题的要求：

若在基础文法中增加关于“带条件的继续循环”语句的产生式

$$S \rightarrow \text{continue when } E$$

语句 **continue when  $E$**  只能出现在某个循环语句内，即至少有一个包围它的 **while** 语句，其执行语义为：当  $E$  为真时，跳出直接包含该语句的 **while** 循环体，并回到 **while** 循环的开始处判断循环条件重新执行该循环；当  $E$  为假时，不做任何事情，转下一条语句。

试在上述 L-翻译模式片段基础上增加针对 **continue when  $E$**  语句的语义处理内容（不改变 L-翻译模式的特征，不考虑“是否出现在while 语句内”的语义检查工作）。

注：可设计引入新的属性或删除旧的属性，必要时给出解释。

(c) 以下是一个 S-翻译模式/属性文法片断，可以产生控制语句（注意：条件语句和上题不同）的 TAC 语句序列：

$$\begin{aligned}
 P &\rightarrow D; S M \quad \{ \text{backpatch}(S.nextlist, M.gotostm) \} \\
 S &\rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2 \quad \{ \text{backpatch}(E.truelist, M_1.gotostm); \\
 &\quad \text{backpatch}(E.falselist, M_2.gotostm); \\
 &\quad S.nextlist := \text{merge}(S_1.nextlist, \text{merge}(N.nextlist, S_2.nextlist)) \} \\
 S &\rightarrow \text{while } M_1 E \text{ then } M_2 S_1 \quad \{ \text{backpatch}(S_1.nextlist, M_1.gotostm); \\
 &\quad \text{backpatch}(E.truelist, M_2.gotostm);
 \end{aligned}$$

$$\begin{array}{ll}
& S.nextlist := E.falselist; \\
& emit('goto', M_1.gotostm) \} \\
S \rightarrow S_1 ; M S_2 & \{ backpatch(S_1.nextlist, M_1.gotostm); \\
& S.nextlist := S_2.nextlist \} \\
M \rightarrow \varepsilon & \{ M.gotostm := nextstm \} \\
N \rightarrow \varepsilon & \{ N.nextlist := makelist(nextstm); emit('goto_') \}
\end{array}$$

其中，所用到的属性和语义函数与讲稿中一致：综合属性  $E.truelist$ （真链），表示一系列跳转语句的地址，这些跳转语句的目标语句标号是体现布尔表达式  $E$  为“真”的标号；综合属性  $E.falselist$ （假链），表示一系列跳转语句的地址，这些跳转语句的目标语句标号是体现布尔表达式  $E$  为“假”的标号；综合属性  $S.nextlist$ （next链），表示一系列跳转语句的地址，这些跳转语句的目标语句标号是在执行序列中紧跟在  $S$  之后的下条 TAC 语句的标号；语义函数  $makelist(i)$ ，用于创建只有一个结点  $i$  的表，对应于一条跳转语句的地址；语义函数  $merge(p_1, p_2)$ ，表示连接两个链表  $p_1$  和  $p_2$ ，返回结果链表；语义函数  $backpatch(p, i)$ ，表示将链表  $p$  中每个元素所指向的跳转语句的标号置为  $i$ ；语义函数  $nextstm$ ，将返回下一条 TAC 语句的地址；语义函数  $emit(...)$ ，将输出一条 TAC 语句，并使  $nextstm$  加1。

（和前面一样，假设在语法制导处理的分析过程中遇到的冲突问题均可以按照某种原则处理，这里不必考虑基础文法是否 LR 文法。）

下面叙述本小题的要求：

若在基础文法中增加关于“带条件的继续循环”语句的产生式

$$S \rightarrow \text{continue when } E$$

语句 **continue when  $E$**  只能出现在某个循环语句内，即至少有一个包围它的 **while** 语句，其执行语义为：当  $E$  为真时，跳出直接包含该语句的 **while** 循环体，并回到 **while** 循环的开始处判断循环条件重新执行该循环；当  $E$  为假时，不做任何事情，转下一条语句。

试在上述  $S$ -翻译模式片段基础上增加针对 **continue when  $E$**  语句的语义处理内容（不改变  $S$ -翻译模式的特征，不考虑“是否出现在 **while** 语句内”的语义检查工作）。

注：可设计引入新的属性或删除旧的属性，必要时给出解释。

12. (a) 以下是一个  $S$ -翻译模式片断，描述了某小语言部分特性的类型检查工作：

$$\begin{array}{l}
P \rightarrow D ; S \quad \{ P.type := \text{if } D.type = ok \text{ and } S.type = ok \text{ then } ok \text{ else } type\_error \} \\
S \rightarrow S' \quad \{ S.type := S'.type \} \\
S \rightarrow \text{if } E \text{ then } S_1 \quad \{ S.type := \text{if } E.type = bool \text{ then } S_1.type \text{ else } type\_error \} \\
S \rightarrow \text{while } E \text{ begin } S_1 \text{ end} \quad \{ S.type := \text{if } E.type = bool \text{ then } S_1.type \text{ else } type\_error \} \\
S \rightarrow \text{for } (S'_1; E; S'_2) \text{ begin } S_1 \text{ end} \\
\quad \{ S.type := \text{if } E.type = bool \text{ and } S'_1.type = ok \text{ and } S'_2.type = ok \\
\quad \text{then } S_1.type \text{ else } type\_error \}
\end{array}$$

```

 $S \rightarrow S_1 ; S_2 \quad \{ S.type := \text{if } S_1.type = ok \text{ and } S_2.type = ok \text{ then } ok \text{ else } type\_error \}$ 
 $S' \rightarrow \underline{id} := E' \quad \{ S'.type := \text{if } lookup\_type(id.entry) = E'.type \text{ then } ok \text{ else } type\_error \}$ 
 $D \rightarrow \dots \quad /*\text{省略与声明语句相关的全部规则}*/$ 
 $E \rightarrow \dots \quad /*\text{省略与布尔表达式相关的全部规则}*/$ 
 $E' \rightarrow \dots \quad /*\text{省略与算术表达式相关的全部规则}*/$ 

```

其中，`type` 属性以及类型表达式 `ok`, `type_error`, `bool`, 以及所涉及到的语义函数（如 `lookup_type`）等的含义与讲稿中一致；加黑的单词为保留字；声明语句、布尔表达式以及算术表达式相关的部分已全部略去。

（此外，假设在语法制导语义处理的分析过程中遇到的冲突问题均可以按照某种原则处理，我们不考虑基础文法是否 *LR* 文法。）

下面叙述本小题的要求：

若在基础文法中增加如下产生式，用于定义针对 `for`-循环的“跳过本次循环体”语句：

**$S \rightarrow \text{leap}$**

语句 `leap` 只能出现在 `for`-循环语句内部，但同时不能直接出现在某个 `while`-循环语句的内部。以下代码片断中，列举了几个合法与不合法使用 `leap` 语句的例子：

```

...          /*不含循环语句*/
leap;        /*此处的leap是不合法的*/
...          /*不含循环语句*/
for (i:=1; i<100; i:=i+1)
begin
  ...
  leap;      /*此处的leap是合法的*/
  ...
  if i=50 then leap;    /*此处的leap是合法的*/
  ...
  while cond
  begin
    ...
    leap;    /*此处的leap是不合法的*/
    ...
    for (...)
    begin
      leap;  /*此处的leap是合法的*/
    end
    ...
  end
end

end
...

```

试在上述翻译模式片段基础上增加相应的语义处理内容（要求是 L-翻译模式），以实现针对 **leap** 语句的类型检查（合法性检查）任务。

（提示：可以引入  $S$  的一个继承属性）

注：未修改的规则可省略不写。每条规则中，可仅给出与变化内容相关的规则。

(b) 以下是一个 L-翻译模式片断，可以产生相应的 TAC 语句序列：

```

 $P \rightarrow D ; \{ S.next := newlabel \} S \{ gen(S.next ':') \}$ 
 $S \rightarrow \{ S'.next := S.next \} S' \{ S.code := S'.code \}$ 
 $S \rightarrow \text{if} \{ E.true := newlabel; E.false := S.next \} E \text{ then}$ 
     $\{ S_1.next := S.next \} S_1 \{ S.code := E.code \parallel gen(E.true ':') \parallel S_1.code \}$ 
 $S \rightarrow \text{while} \{ E.true := newlabel; E.false := S.next \} E$ 
    begin  $\{ S_1.next := newlabel \} S_1$  end
     $\{ S.code := gen(S_1.next ':') \parallel E.code \parallel gen(E.true ':') \parallel$ 
         $S_1.code \parallel gen('goto' S_1.next) \}$ 
 $S \rightarrow \text{for} ( \{ S'_1.next := newlabel \} S'_1;$ 
     $\{ E.true := newlabel; E.false := S.next \} E ;$ 
     $\{ S'_2.next := S'_1.next \} S'_2 )$ 
    begin  $\{ S_1.next := newlabel \} S_1$  end  $\{$ 
         $S.code := S'_1.code \parallel gen(S'_1.next ':') \parallel E.code \parallel$ 
         $gen(E.true ':') \parallel S_1.code \parallel gen(S_1.next ':') \parallel$ 
         $S'_2.code \parallel gen('goto' S'_1.next) \}$ 
 $S \rightarrow \{ S_1.next := newlabel \} S_1 ;$ 
     $\{ S_2.next := S.next \} S_2$ 
     $\{ S.code := S_1.code \parallel gen(S_1.next ':') \parallel S_2.code \}$ 
 $S' \rightarrow \underline{id} := E' \quad \{ S'.code := E'.code \parallel gen(\underline{id}.place ':= E'.place) \}$ 
 $D \rightarrow \dots$  /*省略与声明语句相关的全部规则*/
 $E \rightarrow \dots$  /*省略与布尔表达式相关的全部规则*/
 $E' \rightarrow \dots$  /*省略与算术表达式相关的全部规则*/
```

其中，属性  $S.code$ ,  $S'.code$ ,  $E.code$ ,  $S.next$ ,  $S'.next$ ,  $E.true$ ,  $E.false$ , 语义函数  $newlabel$ ,  $gen()$  以及所涉及到的 TAC 语句与讲稿中一致。语义函数  $newtemp$  的作用是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；语义函数  $gen$  的结果是生成一条 TAC 语句；“ $\parallel$ ”表示 TAC 语句序列的拼接。所有符号的  $place$  综合属性也均与讲稿中一致。

（此外，和前面一样，假设在语法制导语义处理的分析过程中遇到的冲突问题均可以按照某种原则处理，我们不考虑基础文法是否 LR 文法。）

下面叙述本小题的要求：

若在基础文法中增加如下产生式，用于定义针对 for-循环的“跳过本次循环体”语句：

$S \rightarrow \text{leap}$

如第1小题中所述，语句 **leap** 只能出现在 for-循环语句内部，但同时不能直接出现

在某个while-循环语句的内部，其执行语义为：设  $\text{for}(S'_1; E; S'_2) \text{ begin} \dots \text{end}$  是直接包围该 **leap** 语句的 **for**-循环，在执行 **leap** 语句后，控制将跳过该循环体内部剩余的语句，跳转到下一次循环（先执行  $S'_2$ ）。

试在上述 *L*-翻译模式片段基础上增加针对 **leap** 语句的语义处理内容（不改变 *L*-翻译模式的特征，不考虑第1小题中已完成的语义检查工作）。

注：可引入新的属性或删除旧的属性，必要时给出解释。未修改的规则可省略不写。

(c) 以下是一个 *S*-翻译模式片断，可以产生相应的 *TAC* 语句序列：

$$\begin{aligned}
 P &\rightarrow D; S M && \{ \text{backpatch}(S.\text{nextlist}, M.\text{gotostm}) \} \\
 S &\rightarrow S' && \{ S.\text{nextlist} := "" \} \\
 S &\rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2 && \{ \text{backpatch}(E.\text{truelist}, M_1.\text{gotostm}); \\
 &&& \text{backpatch}(E.\text{falselist}, M_2.\text{gotostm}); \\
 &&& S.\text{nextlist} := \text{merge}(S_1.\text{nextlist}, \text{merge}(N.\text{nextlist}, S_2.\text{nextlist})) \} \\
 S &\rightarrow \text{while } M_1 E \text{ begin } M_2 S_1 N \text{ end} && \{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{gotostm}); \\
 &&& \text{backpatch}(E.\text{truelist}, M_2.\text{gotostm}); \\
 &&& \text{backpatch}(N.\text{nextlist}, M_1.\text{gotostm}); \\
 &&& S.\text{nextlist} := E.\text{falselist}; \} \\
 S &\rightarrow \text{for } (S'_1; M_1 E; M_2 S'_2 N_1) \text{ begin } M_3 S_1 N_2 \text{ end} && \{ \text{backpatch}(E.\text{truelist}, M_3.\text{gotostm}); \\
 &&& \text{backpatch}(S_1.\text{nextlist}, M_2.\text{gotostm}); \\
 &&& \text{backpatch}(N_1.\text{nextlist}, M_1.\text{gotostm}); \\
 &&& \text{backpatch}(N_2.\text{nextlist}, M_2.\text{gotostm}); \\
 &&& S.\text{nextlist} := E.\text{falselist} \} \\
 S &\rightarrow S_1; M S_2 && \{ \text{backpatch}(S_1.\text{nextlist}, M.\text{gotostm}); \\
 &&& S.\text{nextlist} := S_2.\text{nextlist} \} \\
 S' &\rightarrow \text{id} := E' && \{ \text{emit}(\text{id}.\text{place} := E'.\text{place}) \} \\
 M &\rightarrow \varepsilon && \{ M.\text{gotostm} := \text{nextstm} \} \\
 N &\rightarrow \varepsilon && \{ N.\text{nextlist} := \text{makelist}(\text{nextstm}); \text{emit}('goto\_') \} \\
 D &\rightarrow \dots && /*省略与声明语句相关的全部规则*/ \\
 E &\rightarrow \dots && /*省略与布尔表达式相关的全部规则*/ \\
 E' &\rightarrow \dots && /*省略与算术表达式相关的全部规则*/
 \end{aligned}$$

其中，所用到的属性和语义函数与讲稿中一致：综合属性 *E.truelist*（真链），表示一系列跳转语句的地址，这些跳转语句的目标语句标号是体现布尔表达式 *E* 为“真”的标号；综合属性 *E.falselist*（假链），表示一系列跳转语句的地址，这些跳转语句的目标语句标号是体现布尔表达式 *E* 为“假”的标号；综合属性 *S.nextlist*（*next*链），表示一系列跳转语句的地址，这些跳转语句的目标语句标号是在执行序列中紧跟在 *S* 之后的下条 *TAC* 语句的标号；语义函数 *makelist*(*i*)，用于创建只有一个结点 *i* 的表，对应于一条跳转语句的地址；语义函数 *merge*(*p*<sub>1</sub>, *p*<sub>2</sub>)，表示连接两个链表 *p*<sub>1</sub> 和 *p*<sub>2</sub>，返回结果链表；语义函数 *backpatch*(*p*, *i*)，表示将链表 *p* 中每个元素所指向的跳转语句的标号置为 *i*；语义函数 *nextstm*，将返回下一条 *TAC* 语句的地址；语义函数 *emit* (...), 将输出一条 *TAC* 语句，并使 *nextstm* 加1。同第2小题，所有符号的 *place* 综合属性也均与讲稿中一致。

（和前面一样，假设在语法制导语义处理的分析过程中遇到的冲突问题均可以按照某种原则处理，这里不必考虑基础文法是否  $LR$  文法。）

下面叙述本小题的要求：

若在基础文法中增加如下产生式，用于定义针对for-循环的“跳过本次循环体”语句：

$$S \rightarrow \text{leap}$$

如（a）小题中所述，语句 `leap` 只能出现在for-循环语句内部，但同时不能直接出现在某个while-循环语句的内部，其执行语义为：设 `for ( $S'_1; E; S'_2$ ) begin ... end` 是直接包围该 `leap` 语句的 for-循环，在执行 `leap` 语句后，控制将跳过该循环体内部剩余的语句，跳转到下一次循环（先执行  $S'_2$ ）。

试在上述  $L$ -翻译模式片段基础上增加针对 `leap` 语句的语义处理内容（不改变  $L$ -翻译模式的特征，不考虑第1小题中已完成的语义检查工作）。

试在上述  $S$ -翻译模式片段基础上增加针对 `leap` 语句的语义处理内容（不改变  $S$ -翻译模式的特征，不考虑第1小题中已完成的语义检查工作）。

注：可引入新的属性或删除旧的属性，必要时给出解释。未修改的规则可省略不写。

## 参考文献

1. Luca Cardelli, Type Systems, Handbook of Computer Science and Engineering, Chapter 103. CRC Press, 1997. Available at : <http://www.lucacardelli.name/Papers/TypeSystems.pdf> .
2. Robert Harper, Practical Foundations for Programming Languages. Cambridge University Press, 2012. 第2版, 2016.
3. Benjamin C. Pierce, Types and Programming Languages, The MIT Press, 2002.