第七讲　静态语义分析与中间代码生成

2020-11

第六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。在这一讲里，我们将以常见的静态语义分析工作以及常见语言成分的翻译为例，讨论基于属性文法或翻译模式的语义计算在静态语义分析和中间代码生成中的应用。虽然我们主要介绍的是语法制导的实现方法，但相关技术对比如基于抽象语法树（AST）的静态语义分析和中间代码生成也有指导意义。对 AST 进行多遍扫描时的每一遍均类似于语法制导的单遍语义计算。

1. 语法制导的静态语义分析

1.1　静态语义分析

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），则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）的一致性或完整性，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，名字的出现在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。

我们在第二阶段实验项目中已经实践了许多关于静态语义检查的任务，多数工作都可以在充分理解源语言的语义规则基础上完成。在这一讲里，我们将不再进一步讨论。

类型检查或许是语义分析阶段最重要的工作。理论上，以上提到各种检查都可以划归为类型检查。注：本课所涉及的类型检查工作都是指静态类型检查。

静态语义分析的工作中有许多可以较方便地采用语法制导的方法来实现，但有一些并不容易，需要借助于多遍的方法来处理。然而，无论采取单遍还是多遍的实现方案，采用属性文法/翻译模式进行设计阶段的描述都是很有意义的。在随后的小节里，我们重点讨论借助语法制导的方法来实现一个简单语言的类型检查。

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，可参考第五讲。

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（如过程、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

类型检查程序负责类型检查工作，主要包括：

- 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

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 一个简单语言

为示范类型系统的定义，图 1 描述了一个简单语言语法的上下文无关文法 *G*[*P*]。其中 num，id，int，以及 real 分别对应数字，标识符，整型数，以及实型数的单词符号；op 以及 rop 对应算术运算符以及关系运算符，为简化讨论暂未指定具体的运算；array [num] of *T* 和 *E*[*E*]分别为数组声明和数组元素访问；^T 声明指针类型，而 *E*^ 表示对指针所指对象的访问。

由文法 *G*[*P*]可知，一个程序由声明部分（*D*）和语句部分（*S*）构成。声明部分包括变量声明（*V*）和过程声明（*F*）。变量声明可以分段，每一段声明某一类型（*T*）的多个变量（*L*）。过程声明可以声明一系列无参过程（无嵌套声明），每个无参过程包含过程名（id）、形参声明（*V*）

以及过程体语句部分（*S*），不含局部变量声明。语句部分由一系列语句构成，包括赋值、条件、while 循环、顺序复合、break 以及过程调用，等等。

G[*P*]所描述的是该简单语言的抽象语法（*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。

$P \rightarrow D\ ;\ S$

$D \rightarrow V\ ;\ F$

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

$T \rightarrow$ boolean | integer | real | array [num] of T | ^T

$L \rightarrow L\ ,$ id | id

$S \rightarrow$ id := E | if E then S | if E then S else S | while E then S | $S\ ;\ S$ | break | call id (A)

$E \rightarrow$ true | false | int | real | id | E op E | E rop E | E[E] | E^

$F \rightarrow F\ ;$ id (V) $S \mid \varepsilon$

$A \rightarrow A\ ,\ E \mid \varepsilon$

图 1 一个简单语言的语法

这一简单语言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，针对上述简单语言，我们定义如下类型表达式的集合：

- 基本数据类型表达式：*bool*，*int*，*real*。
- 有界数组类型表达式：*array*(*I*,*T*)。其中，*T* 是基本数据类型表达式，*I* 代表一个整数区间（如 1..10 表示从 1 到 10 的整数集合）。*array*(*I*,*T*) 表示元素类型是 *T*，下标集合是 *I* 的数组类型。
- 指针数据类型表达式：*pointer*(*T*)。其中，*T* 是基本数据类型表达式。*pointer*(*T*) 表示指向类型为 *T* 的对象的指针类型。
- 积类型表达式：< T_1，T_2，…，T_n >。其中，T_1，T_2，…，T_n（$n \geq 0$）取自上述 3 种数据类型表达式；若 $n=0$，则表示为 < >。
- 过程类型表达式：*fun*（*T*）。其中，*T* 是上述积类型表达式。
- 类型表达式 *type_error* 专用于有类型错误的程序单元。
- 类型表达式 *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，若语言中定义了函数，而不是过程，则很容易将以上过程类型表达式修改为某种函数类型表达式。

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

一个类型环境记录一类特定标识符在某个上下文或作用域中被赋予的类型表达式。

对于图 1 的简单语言，我们用 G 表示全局类型环境，用于记录全局变量标识符以及函数标识符的类型表达式。这里，我们没有区分变量标识符与函数（过程）标识符的环境，如果允许二者使用不同的命名空间，则需要定义不同的类型环境。

同时，由于存在函数（过程）作用域（无嵌套声明），所以引入 F 表示函数（过程）内部的局部类型环境，由于不含局部变量声明，所以仅记录形参变量的类型表达式。如果允许局部声明变量，一般需要与形参变量共享同一个类型环境。虽然有些语言需要定义独立的形参类型环境，但通常局部声明的变量不允许与形参变量同名。

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，除了常规断言，还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：

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

其中，e 代表各类不同的程序单元，如表达式，语句，以及各类声明，等等。为简化描述，在下面所定义的类型规则中，用 ξ 表示既可以是环境 'G' 也可以是环境 'G, F'。

比如，我们可以定义一个如下类型规则集合：

- 针对表达式的类型规则

$$\frac{}{\xi \vdash \text{true} : bool}\ (\text{E-true}) \qquad \frac{}{\xi \vdash \text{false} : bool}\ (\text{E-false})$$

$$\frac{}{\xi \vdash \underline{\text{int}} : int}\ (\text{E-int}) \qquad \frac{}{\xi \vdash \underline{\text{real}} : real}\ (\text{E-real})$$

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

$$\frac{\xi \vdash e_1 : \tau \qquad \xi \vdash e_2 : \tau}{\xi \vdash e_1 \ \underline{\text{op}}\ e_2 : \tau}\ (\text{E-alop})$$

$$\frac{\xi \vdash e_1 : \tau \qquad \xi \vdash e_2 : \tau \qquad \tau = int \text{ or } real}{\xi \vdash e_1 \ \underline{rop} \ e_2 : bool} \text{(E-rop)}$$

$$\frac{\xi \vdash e_1 : array(I, \tau) \qquad \xi \vdash e_2 : int}{\xi \vdash e_1[e_2] : \tau} \text{(E-arrayacc)}$$

$$\frac{\xi \vdash e : \tau}{\xi \vdash e\text{^} : pointer(\tau)} \text{(E-pointeracc)}$$

- 针对语句的类型规则

$$\frac{\xi \vdash \underline{id} : \tau \qquad \xi \vdash e : \tau}{\xi \vdash \underline{id} := e \ : \ ok} \text{(S-assin)}$$

$$\frac{\xi \vdash e : bool \qquad \xi \vdash s : ok}{\xi \vdash \text{if } e \text{ then } s \ : \ ok} \text{(S-ift)}$$

$$\frac{\xi \vdash e : bool \qquad \xi \vdash s_1 : ok \qquad \xi \vdash s_2 : ok}{\xi \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 \ : \ ok} \text{(S-ifte)}$$

$$\frac{\xi \vdash e : bool \qquad \xi \vdash s : ok}{\xi \vdash \text{while } e \text{ then } s \ : \ ok} \text{(S-while)}$$

$$\frac{\xi \vdash s_1 : ok \qquad \xi \vdash s_2 : ok}{\xi \vdash s_1 ; s_2 \ : \ ok} \text{(S-com)}$$

$$\frac{\xi \vdash \text{while } e \text{ then } s_1 : ok \qquad \xi \vdash \text{while } e \text{ then } s_2 : ok \qquad s_1 \text{ or } s_2 \text{ may be empty}}{\xi \vdash \text{while } e \text{ then } s_1 \text{ ; break; } s_2 \ : \ ok} \text{(S-break)}$$

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

- 针对类型声明的规则

$$\frac{}{\vdash \text{boolean} : bool} \text{(T-bool)} \qquad \frac{}{\vdash \text{integer} : int} \text{(T-int)}$$

$$\frac{}{\vdash \text{real} : real} \text{(T-real)} \qquad \frac{\vdash t : \tau}{\vdash \hat{}t : pointer(\tau)} \text{(T-pointer)}$$

$$\frac{\vdash t : \tau \qquad n \text{ is the value of } \underline{num}}{\vdash \text{array } [\underline{num}] \text{ of } t : array(0..(n\text{-}1)\ ,\ \tau)} \text{(T-array)}$$

- 针对声明语句的类型规则（初始化类型环境）

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

（Here，we assume ‘ε ; *t l*’ to be ‘*t l* ‘ and omit the rule: $\vdash \varepsilon : <>$. dom(*G*) is the set of identifiers in the domain of *G.*）

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

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

$$\frac{\begin{array}{c} G \vdash f : ok \qquad G，\ \{ (x_1, \tau_1) \ldots, (x_m, \tau_m) \} \vdash v : <\tau_1, \ldots, \tau_m> \\ G，\ \{ (x_1, \tau_1) \ldots, (x_m, \tau_m) \} \vdash s : ok \qquad \{ x_1, \ldots, , \tau_m \} = \text{ids}(v) \qquad \underline{\text{id}} \notin \text{dom}(G) \end{array}}{G \cup \{ (\underline{\text{id}}, fun(<\tau_1, \ldots, \tau_m>)) \} \ \vdash f\ ;\ \underline{\text{id}}\ (v)\ s \ : \ ok} \text{(D-fun)}$$

（Here，we assume ‘ε ; $\underline{\text{id}}$ (*v*) *s* ‘ to be ‘$\underline{\text{id}}$ (*v*) *s* ‘ and omit the rule: $G \vdash \varepsilon : ok$. ids(*v*) is the set of all identifiers in *v.*）

- 其他语法单元的类型规则

$$\frac{G \vdash v : <\tau_1, \ldots, \tau_m> \quad G \vdash f : ok}{G \vdash v\,;f : ok}(\text{D-vf}) \qquad \frac{G \vdash d : ok \quad G \vdash s : ok}{G \vdash d\,;s : ok}(\text{P-ds})$$

1.2.1.5 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 语法制导的类型检查

通常，类型系统是由类型检查程序实现的。下面以图 1 的简单语言为例，讨论实现类型检查的属性文法/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，1.2.1.4 中某些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，其作用是计算变量申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：

$V \rightarrow V_1\ ;\ T\ \{\ L.in := T.type\ \}\ L\ \{\ V.type := make\_product\_3\ (V_1.type, T.type, L.num)\ \}$	
$V \rightarrow \varepsilon$	{ *V.type* := <> }
$T \rightarrow$ boolean	{ *T.type* := *bool* }
$T \rightarrow$ integer	{ *T.type* := *int* }
$T \rightarrow$ real	{ *T.type* := *real*}
$T \rightarrow$ array [num] of T_1	{ *T.type* := *array*(1.. *num*.*lexval*, T_1.*type*) }
$T \rightarrow \hat{}T_1$	{ *T.type* := *pointer*(T_1.*type*) }
$L \rightarrow \{\ L_1.in := L.in\ \}\ L_1$, id	{ *addtype*(id.*entry*, *L.in*) ; *L.num* := L_1.*num* +1 }
$L \rightarrow$ id	{ *addtype*(id.*entry*, *L.in*); *L.num* := 1 }

其中，*num*.*lexval* 为词法分析返回的单词属性值，id.*entry* 指向当前标识符对应于符号表中的表项，语义函数 *addtype*(id.*entry*,*L.in*) 表示将属性值 *L.in* 填入当前标识符在符号表表项中的 *type* 域（记录标识符的类型），语义函数 *make_product_3*（< t_1，t_2，…，t_m>，$type_2$，*n*) 生成积

类型表达式 $< t_1, t_2, \ldots, t_m, type_2, \ldots, type_2 >$（含 n 个 $type_2$）。在这个翻译模式片断中 $L.in$ 为继承属性，$T.type$ 和 $V.type$ 为综合属性。

为方便，过程声明部分相关的翻译模式片断随语句部分一起给出。

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：

$E \rightarrow$ true	{ $E.type$:= $bool$ }
$E \rightarrow$ false	{ $E.type$:= $bool$ }
$E \rightarrow$ int	{ $E.type$:= int }
$E \rightarrow$ real	{ $E.type$:= $real$ }
$E \rightarrow$ id	{ $E.type$:= if $lookup\_type(id.name)$ = nil then $type\_error$ else $lookup\_type(id.name)$ }
$E \rightarrow E_1$ op E_2	{ $E.type$:= if $E_1.type=real$ and $E_2.type=real$ then $real$ else if $E_1.type=int$ and $E_2.type=int$ then int else $type\_error$ }
$E \rightarrow E_1$ rop E_2	{ $E.type$:= if $E_1.type=real$ and $E_2.type=real$ then $bool$ else if $E_1.type=int$ and $E_2.type=int$ then $bool$ else $type\_error$ }
$E \rightarrow E_1 [E_2]$	{ $E.type$:= if $E_2.type= int$ and $E_1.type=array(s, t)$ then t else $type\_error$ }
$E \rightarrow E_1$^	{ $E.type$:= if $E_1.type= pointer(t)$ then t else $type\_error$ }

其中，$id.name$ 为当前标识符的名字；语义函数 $lookup\_type(id.name)$ 从符号表中查找名字为 $id.name$ 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。

以下是与语句及过程声明相关的类型检查的翻译模式片断：

$S \rightarrow$ id := E	{ $S.type$:= if $lookup\_type$ (id.$entry$) = $E.type$ then ok else $type\_error$ }
$S \rightarrow$ if E then S_1	{ $S.type$:= if $E.type=bool$ then $S_1.type$ else $type\_error$ }
$S \rightarrow$ if E then S_1 else S_2	{ $S.type$:= if $E.type=bool$ and $S_1.type = ok$ and $S_2.type = ok$ then ok else $type\_error$ }
$S \rightarrow$ while E then S_1	{ $S.type$:= if $E.type=bool$ then $S_1.type$ else $type\_error$ }
$S \rightarrow S_1 ; S_2$	{ $S.type$:= if $S_1.type = ok$ and $S_2.type = ok$ then ok else $type\_error$ }
$S \rightarrow$ break	{ $S.type$:= ok }
$S \rightarrow$ call id (A)	{ $S.type$:= if $match$ ($lookup\_type$ ($id.name$), $A.type$) then ok else $type\_error$ }
$F \rightarrow F_1$; id (V) S	{ $addtype$(id.$entry$, fun ($V.type$)); $F.type$:= if $F_1.type = ok$ and $S.type = ok$ then ok else $type\_error$ }
$F \rightarrow \varepsilon$	{ $F.type$:= ok }
$A \rightarrow A_1 , E$	{ $A.type$:= $make\_product\_2$ ($A_1.type$, $E.type$) }

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

其中，语义函数 *make_product_2*（< t_1，t_2，…，t_m>，$type_2$) 生成积类型表达式 < t_1，t_2，…，t_m，$type_2$>；语义函数 *match*（fun ($type_1$)，$type_2$) 返回 true，当且仅当 $type_1$，$type_2$ 都是完全相同的积类型表达式（即二者有同样多的分量，且每个分量都相同）。

最后，我们补充如下翻译模式片断：

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

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

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* 语句进行如下检查：*break* 语句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。可以通过引入继承属性 *S.break* 来解决这一问题，以下仅列出有变化的产生式：

$P \to D$; { *S.break* := 0 } S
{ *P.type* := if *D.type* = *ok* and *S.type* = *ok* then *ok* else *type_error* }

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

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

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

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

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

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

我们简要讨论一下上述翻译模式片断的语义计算问题。单独看这一小节出现的几个翻译模式片断，第一个翻译模式片断（用于计算变量申明相关语法单位的类型信息）是 *L*-翻译模式，其基础文法是 *LR* 文法，且嵌入在产生式中间的语义动作只有复写规则，因此可以采用自下而上方式进行语义计算。第二个翻译模式片断（与表达式相关的类型检查）和第三个翻译模式片断（与语句及过程声明相关的类型检查）是 *S*-翻译模式，虽然其基础文法不是 *LR* 文法，但可以通过规定优先级、结合性和最近嵌套匹配等方法构造出适当的 *LR* 分析表，因此可以采用自下而上方式进行语义计算。最后一个翻译模式片断（在前面翻译模式基础上增加 *break* 语句相关的处理）是 *L*-翻译模式，类似于第三个翻译模式片断，针对其基础文法也可以构造适当的 *LR* 分析表；虽然嵌入在产生式中间的语义动作含有非复写规则（*S.break* := 0），但只要对翻译模式片断稍加修改（引入新的文法符号，添加相应的ε-产生式）就可以变换为适合自下而上语义计算的翻译模式。因此，将这些翻译模式片断整合起来的翻译模式（或稍加变换）能够满足语法制导语义计算的要求。对于本讲后续部分的翻译模式片断，设计时都考虑到了其语法制导语义计算的可行性问题，届时将不再重复解释了。

2. 语法制导的中间代码生成

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**四元式**）。
- *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。

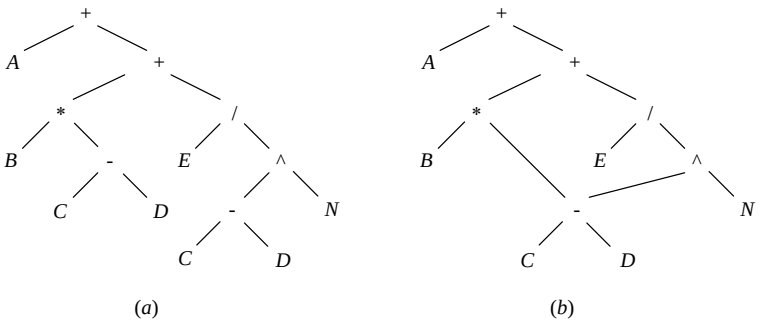


图 2　抽象语法树和有向无圈图

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：

$x := y\ op\ z$

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*　*y*　*z*　*x*）

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

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 ϕ 函数。对于有分支的情形，需要通过插入所谓的“ϕ 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ϕ 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。“ϕ 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。

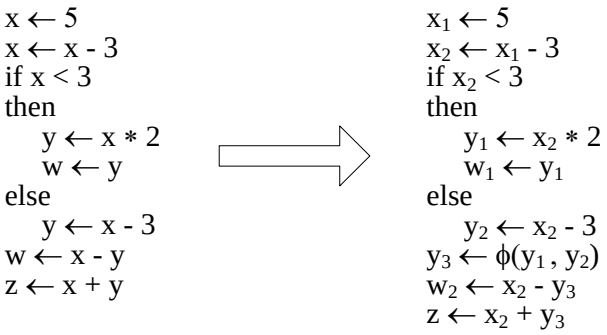


图 4　静态单赋值形式

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 $A \to A\,E \mid \varepsilon$ 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，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。

先看下列翻译模式片断，它可以将简单语句和表达式翻译至一种 *AST*：

$S \to$ id := E	{ *S.ptr* := *mknode*('*assign*', *mkleaf*(id.*entry*), *E.ptr*) }
$S \to$ if E then S_1	{ *S.ptr* := *mknode*('*if_then*', *E.ptr*, S_1.*ptr*) }
$S \to$ if E then S_1 else S_2	{ *S.ptr* := *mknode*('*if_then_else*', *E.ptr*, S_1.*ptr*, S_2.*ptr*) }
$S \to$ while E then S_1	{ *S.ptr* := *mknode*('*while_do*', *E.ptr*, S_1.*ptr*) }
$S \to S_1$; S_2	{ *S.ptr* := *mknode*('*seq*', S_1.*ptr*, S_2.*ptr*) }
$S \to$ break	{ *S.ptr* := *mknode*('*break*') }
$E \to$ id	{ *E.ptr* := *mkleaf* (id .*entry*) }
$E \to$ int	{ *E.ptr* := *mkleaf* (int .*val*) }
$E \to$ real	{ *E.ptr* := *mkleaf* (real .*val*) }
$E \to E_1 + E_2$	{ *E.ptr* := *mknode*('*add*' , E_1.*ptr*, E_2.*ptr*) }
$E \to E_1 * E_2$	{ *E.ptr* := *mknode*('*mul*' , E_1.*ptr*, E_2.*ptr*) }
$E \to$ -E_1	{ *E.ptr* := *mknode*('*uminus*' , E_1.*ptr*) }
$E \to (E_1)$	{ *E.ptr* := E_1.*ptr* }
$E \to$ true	{ *E.ptr* := *mkleaf* ('*true*') }
$E \to$ false	{ *E.ptr* := *mkleaf* ('*false*') }
$E \to E_1 \wedge E_2$	{ *E.ptr* := *mknode*('*and*' , E_1.*ptr*, E_2.*ptr*) }
$E \to E_1 \vee E_2$	{ *E.ptr* := *mknode*('*or*' , E_1.*ptr*, E_2.*ptr*) }

$E \to \neg E_1$	{ *E.ptr* := *mknode*('*not*' , E_1.*ptr*) }
$E \to E_1 [E_2]$	{ *E.ptr* := *mknode*('*array*' , E_1.*ptr*, E_2.*ptr*) }
$E \to E\wedge$	{ *E.ptr* := *mknode*('*pointer*' , E_1.*ptr*) }

其中，*mknode* 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 *mknode* 和 *mkleaf* 都将返回相应结点的一个指针。文法符号 *S*、*E*、*A* 的综合属性 *S.ptr*，*E.ptr*、*A.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。

该翻译模式的基础文法可以对应到图 1 中简单语句和表达式的文法定义，不同的是定义了具体的算数和逻辑运算。针对图 1 中其余语法成分，我们设计如下翻译模式片断：

$P \to D$; S	{ *P.ptr* := *mknode*('*toplevel*', *D.ptr*, *S.ptr*) }
$D \to V$; F	{ *D.ptr* := *mknode*('*decl*', *V.v-list*, *F.f-list*) }
$V \to V_1$; T L	{ *V.v-list* := *link_list* (V_1 .*v-list*, *L.v-list*) }
$V \to \varepsilon$	{ *V.v-list* := *make_empty_list* ( ) }
$L \to L_1$, id	{ *L.v-list* := *insert_list* (L_1 .*v-list*, id.*entry*) }
$L \to$ id	{ *L.v-list* := *make_ list* (id.*entry*) }
$F \to F_1$; id (V) S	{ *F.f-list* := *insert_list* (F_1 .*f-list*, id.*entry*) }
$F \to \varepsilon$	{ *F.f-list* := *make_empty_list* ( ) }
$A \to A_1$, E	{ *A.e-list* := *insert_list* (A_1 .*e-list*, *E.ptr*)}
$A \to \varepsilon$	{ *A.e-list* := *make_empty_list* ( ) }
$S \to$ call id (A)	{ *S.ptr* := *mknode*('*call*', *mkleaf*(id.*entry*), *A.e-list*) }

其中，语义函数 *make_empty_list*，*make_ list*，*insert_list*，以及 *merge_list* 分别为创建空表，创建单元素表，在已知表中插入一个新元素，以及两个表的链接。

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的问题。

在随后的例子中，我们将用到下列类型的 *TAC* 语句：

- 赋值语句 $x := y$ op z （op 代表二元算术/逻辑运算）。
- 赋值语句 $x :=$ op y （op 代表一元运算）。
- 复写语句 $x := y$ （y 的值赋值给 x）。
- 无条件跳转语句 goto L （无条件跳转至标号 L）。
- 条件跳转语句 if x rop y goto L （rop 代表关系运算）。
- 标号语句 L :（定义标号 L）。
- 过程调用语句序列 param x_1 … param x_n call p, n，其中包括 n+1 条 *TAC* 语句。

- 过程返回语句 return。
- 下标赋值语句 $x := y[i]$ 和 $x[i] := y$ （前者表示将 y 的存储位置起第 i 个存储单元的值赋给 x，后者表示将 y 的值保存到 x 的存储位置起第 i 个存储单元）。
- 指针赋值语句 $x := *y$ 和 $*x := y$ （前者表示将把 y 的取值作为存储位置所指存储单元的内容赋值给 x，后者表示将 y 的取值保存到 x 的取值作为存储位置所指的存储单元中）。

注：这里，*TAC* 语句中的变量名字对应一个存储位置。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。

2.3.1 赋值语句及算术表达式的翻译

以下是一个 *S*-翻译模式片断，可以产生相应于赋值语句和算术表达式的 *TAC* 语句序列：

$S \to \underline{\text{id}} := A$	{ *S.code* := *A.code* ‖ *gen*(id .*place* ':=' *A.place*) }
$A \to \underline{\text{id}}$	{ *A.place* := id .*place* ; *A.code* := ""}
$A \to \underline{\text{int}}$	{ *A.place* := *newtemp*; *A.code* := *gen* (*A.place* ':=' int .*val*) }
$A \to \underline{\text{real}}$	{ *A.place* := *newtemp*; *A.code* := *gen* (*A.place* ':=' real .*val*) }
$A \to A_1 + A_2$	{ *A.place* := *newtemp*; *A.code* := A_1.*code* ‖ A_2.*code* ‖ *gen* (*A.place* ':=' A_1.*place* '+' A_2.*place*) }
$A \to A_1 * A_2$	{ *A.place* := *newtemp*; *A.code* := A_1.*code* ‖ A_2.*code* ‖ *gen* (*A.place* ':=' A_1.*place* '*' A_2.*place*) }
$A \to -A_1$	{ *A.place* := *newtemp*; *A.code* := A_1.*code* ‖ *gen* (*A.place* ':=' 'uminus' A_1.*place*) }
$A \to (A_1)$	{ *A.place* := A_1.*place* ; *A.code* := A_1.*code* }

其中，id .*place* 表示相应的名字对应的存储位置；综合属性 *A.place* 表示存放 *A* 的值的存储位置；综合属性 *A.code* 表示对 *A* 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。

语义函数 *gen* 的结果是生成一条 *TAC* 语句；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；‖ 是 *TAC* 语句序列之间的链接运算。

2.3.2 说明语句的翻译

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。下面对这个翻译模式片断进行扩充，以使变量标识符的类型以及偏移量可以同时保存至符号表：

$V \to V_1$; *T* { *L.type* := *T.type*; *L.offset* := V_1.*width* ; *L.width* := *T.width* } *L*	
	{ *V.type* := *make_product_3* (V_1.*type*, *T.type*, *L.num*); *V.width* := V_1 .*width* + *L.num* × *T.width* }
$V \to \varepsilon$	{ *V.type* := <>; *V.width* := 0 }

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

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

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

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

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

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

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

其中，文法符号的属性值具有如下含义：$num.lexval$ 为词法分析返回的单词属性值，id.$name$ 为 id 的词法名字；综合属性 $T.type$ 表示所申明的类型；综合属性 $T.width$ 表示所申明类型所占的字节数；继承属性 $L.type$ 表示变量列表被申明的类型；继承属性 $L.width$ 表示变量列表被申明类型所占的字节数；继承属性 $L.offset$ 表示变量列表中第一个变量相对于过程数据区基址的偏移量；综合属性 $L.num$ 表示变量列表中变量的个数；综合属性 $V.width$ 表示申明列表中全部变量所占的字节数。

语义函数 $enter$ (id.$name$, t, o) 的含义为：将符号表中 id.$name$ 所对应表项的 $type$ 域置为 t，$offset$ 域置为 o。

另外，在这个翻译模式中，我们假设了各数据类型的宽度（字节数）：布尔型和字符型为 1，整型为 4，实型为 8，指针为 4。

不难看出，这是一个 L-翻译模式。

2.3.3 数组说明和数组元素引用的翻译

在 2.3.2 的翻译模式中，已经包含了有关（一维）数组说明的处理。下面的翻译模式片段进一步考虑了数组元素的引用：

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

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

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 n 维静态数组说明 $A[l_1 : u_1, l_2 : u_2, \ldots, l_n : u_n]$，可以考虑在符号表中建立如下形式的内情向量：

- $l_1, u_1, l_2, u_2, \ldots, l_n, u_n$：$l_i$ 和 u_i（$1 \le i \le n$）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。
- C：计算元素偏移地址时不变的部分，见随后的解释。

若数组布局采用行优先的连续布局，数组首元素的地址为 a，则数组元素 $A[i_1, i_2, \ldots, i_n]$ 的地址 D 可以如下计算：

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

重新整理后可得：$D = a - C + V$ ，其中

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

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

这里，C 为常量，即前面内情向量的一部分，在生成数组元素地址时不用重复计算。

在此基础上，我们可以设计处理多维数组说明和数组元素引用的翻译模式。限于篇幅，我们对此不作更多的讨论。

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：

$E \to E_1 \vee E_2$	{ *E.place* := *newtemp*; *E.code* := E_1.*code* || E_2.*code* || *gen* (*E.place* ‘:=’ E_1.*place* ‘or’ E_2.*place*) }
$E \to E_1 \wedge E_2$	{ *E.place* := *newtemp*; *E.code* := E_1.*code* || E_2.*code* || *gen* (*E.place* ‘:=’ E_1.*place* ‘and’ E_2.*place*) }
$E \to \neg E_1$	{ *E.place* := *newtemp*; *E.code* := E_1.*code* || *gen* (*E.place* ‘:=’ ‘not’ E_1.*place*) }
$E \to (E_1)$	{ *E.place* := E_1.*place* ; *E.code* := E_1.*code* }
$E \to \underline{\text{id}}_1\ \underline{\text{rop}}\ \underline{\text{id}}_2$	{ *E.place* := *newtemp*; *E.code* := *gen* (‘if ‘ id1.*place* rop.op id2.*place* ‘goto’ *nextstat*+3) || *gen* (*E.place* ‘:=’ ‘0’) || *gen* (‘goto’ *nextstat*+2) || *gen* (*E.place* ‘:=’ ‘1’) }
$E \to$ true	{ *E.place* := *newtemp*; *E.code* := *gen* (*E.place* ‘:=’ ‘1’) }
$E \to$ false	{ *E.place* := *newtemp*; *E.code* := *gen* (*E.place* ‘:=’ ‘0’) }

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号；下同。

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：

$E \to$ { 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* || *gen* (E_1.*false* ':') || 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* || *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* }

$E \rightarrow$ $\underline{\text{id}}_1$ $\underline{\text{rop}}$ $\underline{\text{id}}_2$ { *E.code* := *gen* ('if ' $\underline{\text{id}}$1.*place* $\underline{\text{rop}}$.op $\underline{\text{id}}$2.*place* 'goto' *E.true*) ||
gen ('goto' *E. false*) }

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

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

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∧优先于∨，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 *label*1
*label*1:
if *c*<*d* goto *label*2
goto *E.false*
*label*2:
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 控制语句的翻译

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

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

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

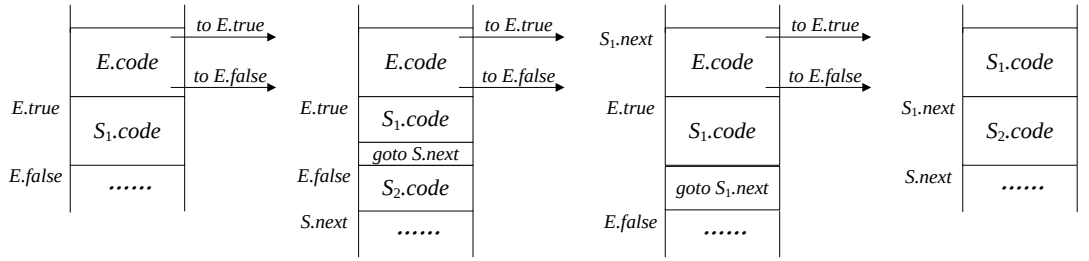
$S \rightarrow$ if { *E.true* := *newlabel*; *E.false* := *newlabel* } *E* then
{ S_1.*next* := *S.next* } S_1 else { S_2.*next* := *S.next* } S_2
{ *S.code* := *E.code* || *gen*(*E. true* ':') || S_1.*code* ||
gen('*goto*' *S.next*) || *gen*(*E.false* ':') || S_2.*code* }

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

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

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。

这一翻译模式片断的设计思路可以参考图 5。



(*a*) if-then　　(*b*) if-then-else　　(*c*) while-do　　(*d*) seq. comp.

图 5　控制语句的翻译

下列翻译模式片断增加了对 *break* 语句的处理：

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

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

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

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

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

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

注：对于不被 while 包围的语句 *S*，*S.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

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.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）：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. 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. 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.gotostm 中记录处理到 M 时下一条待生成语句的标号。
- 语义函数 *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 *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：

$E \to E_1 \vee M\ E_2$	{ *backpatch*(E_1.*falselist*, *M.gotostm*); *E.truelist* := *merge*(E_1.*truelist*, E_2.*truelist*) ; *E.falselist* := E_2.*falselist* }
$E \to E_1 \wedge M\ E_2$	{ *backpatch*(E_1.*truelist*, *M.gotostm*); *E.falselist* := *merge*(E_1.*falselist*, E_2.*falselist*) ; *E.truelist* := E_2.*truelist* }
$E \to \neg\ E_1$	{ *E.truelist* := E_1.*falselist*; *E.falselist* := E_1.*truelist* }
$E \to (E_1)$	{ *E.truelist* := E_1.*truelist*; *E.falselist* := E_1.*falselist* }
$E \to \underline{\text{id}}_1\ \underline{\text{rop}}\ \underline{\text{id}}_2$	{ *E.truelist* := *makelist* (*nextstm*); *E.falselist* := *makelist* (*nextstm*+1); *emit* (‘if ‘ id1.*place* rop.op id2.*place* ‘goto _’); *emit* (‘goto _’) }
$E \to$ true	{ *E.truelist* := *makelist* (*nextstm*); *emit* (‘goto _’) }
$E \to$ false	{ *E.falselist* := *makelist* (*nextstm*); *emit* (‘goto _’) }
$M \to \varepsilon$	{ *M.gotostm* := *nextstm* }

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 $E \to E_1 \vee M\ E_2$，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 $E \to E_1 \wedge M\ E_2$ 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 $E \to E_1 \vee M\ E_2$ 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}。

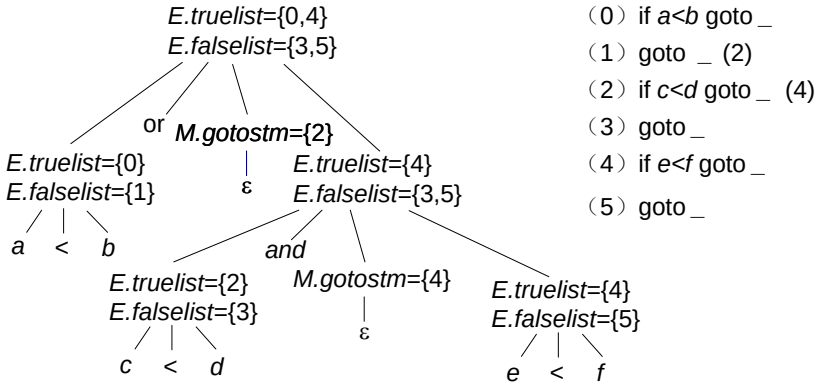


图 6　拉链与代码回填

我们再来看处理控制语句（为简洁，先不考虑 *break* 语句）的 *S*-翻译模式片段：

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

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

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

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

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

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

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

再增加对 *break* 语句的处理：

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

$S \to$ if E then $M\ S_1$　　{ *backpatch*(*E.truelist*, *M.gotostm*) ;

$S.nextlist$:= $merge(E.falselist, S_1.nextlist)$;
$S.breaklist$:= $S_1.breaklist$ }

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

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

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

$S \rightarrow break$; { $S.breaklist$:= $makelist$ ($nextstm$) ; $S.nextlist$:= *“”*;
$emit$ (‘goto _’) }

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

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

最后，我们补充关于赋值语句及算术表达式的翻译（类似于 2.3.1）：

$S \rightarrow$ id := A { $emit$ (id .$place$ ‘:=’ $A.place$); $S.nextlist$:= *“”*; }

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

$A \rightarrow$ int { $A.place$:= $newtemp$; $emit$ ($A.place$ ‘:=’ int .val) }

$A \rightarrow$ real { $A.place$:= $newtemp$; $emit$ ($A.place$ ‘:=’ real .val) }

$A \rightarrow A_1 + A_2$ { $A.place$:= $newtemp$; $emit$ ($A.place$ ‘:=’ $A_1.place$ ‘+’ $A_2.place$) }

$A \rightarrow A_1 * A_2$ { $A.place$:= $newtemp$; $emit$ ($A.place$ ‘:=’ $A_1.place$ ‘*’ $A_2.place$) }

$A \rightarrow -A_1$ { $A.place$:= $newtemp$; $emit$ ($A.place$ ‘:=’ ‘*uminus*’ $A_1.place$) }

$A \rightarrow (A_1)$ { $A.place$:= $A_1.place$ }

2.3.7 过程调用的翻译

这一小节里，我们讨论一个简单过程调用的翻译。例如，对于过程调用

call p（$a + b$，$a * b$）

一种可能的翻译结果形如：

“计算 $a + b$ 结果置于 t 中” 的代码	// $t := a + b$
“计算 $a * b$ 结果置于 z 中”的代码	// $z := a * b$
param t	// 第一个实参地址
param z	// 第二个实参地址
call p , 2	// 过程调用语句

以下是完成此工作的一个 S-翻译模式：

$S \rightarrow$ call id (A)
{ $S.code$:= A .$code$；

for *A.arglist* 中的每一项 *d* do
S.code := *S.code* || *gen*(‘param’ *d*)；
S.code := *S.code* || *gen* (‘call’ id.*place* , *A.n*) }

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

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

其中，属性 *A.code*，*S.code* 和 id.*place*，语义函数 *gen*，以及运算 || 的含义同前。属性 *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 *makelist* 表示创建一个实参地址的结点；语义函数 *append* 表示在已有实参地址列表中添加一个结点。

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 $E \rightarrow E \uparrow E$，试给出相应该产生式的语义动作集合。其中，“↑”代表“与非”逻辑算符，其语义可用其它逻辑运算定义为 P ↑ Q ≡ not (P and Q)。

2 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。若在基础文法中增加产生式 $S \rightarrow$ repeat S until E，试给出相应该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控制语句（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 2.3.5 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。设在该翻译模式基础上增加下列两条产生式及相应的语义动作集合：

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

其中，*E’* 是生成算术表达式的非终结符（对应 2.3.1 中的 *A*）。若在基础文法中增加对应 for-循环语句的产生式 $S \rightarrow$ *for* (*S’*; *E* ; *S’*) *S*，试给出相应该产生式的语义动作集合。

注：for-循环语句的控制语义类似 C 语言中的for-循环语句。

5 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控制语句（不含 break）翻译的 *S*-翻译模式片断及所用到的语义函数。设在该翻译模式基础上增加下列两条产生式及相应的语义动作集合：

$S \rightarrow S'$ { *S.nextlist* := *S’.nextlist* }
$S' \rightarrow$ id := *E’* { *S’.nextlist* := “”; *emit* (id .*place* ‘:=’*E’*. *place*) }

其中，*E’* 是生成算术表达式的非终结符（对应 2.3.1 中的 *A*）。若在基础文法中增加对应 for-循环语句的产生式 $S \rightarrow$ *for* (*S’*; *E* ; *S’*) *S*，试给出相应该产生式的语义动作集合。

注：for-循环语句的控制语义类似 C 语言中的for-循环语句。

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“Δ”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

试给出相应该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

$S \rightarrow$ *switch A of L end*
$L \rightarrow$ *case V* : *S* ; *L*
$L \rightarrow$ *default S*
$V \rightarrow d$

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

$S \rightarrow$ *if G fi*
$G \rightarrow E : S \square G$
$G \rightarrow E : S$

注：串行条件卫士语句的一般形式如

if $E_1 : S_1 \square E_2 : S_2 \square \ldots \square E_n : S_n$ *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（$1 \le k \le n$），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

$$S \rightarrow do\ G\ od$$
$$G \rightarrow E : S \square G$$
$$G \rightarrow E : S$$

注：串行循环卫士语句的一般形式如

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

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（$1 \le k \le n$），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：

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

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

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

$S \rightarrow S_1; S_2$
{ S_1 *.next := newlabel ;*
S_2 *.next := S .next ;*

S .code := S_1 .code || S_2 .code || gen(S .next ':')
}

$E \rightarrow E_1$ or E_2
{ E_2 .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)* }

}

$E \rightarrow$ 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“↑”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 $S \rightarrow$ repeat S 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：repeat <循环体> until <布尔表达式> 至少执行<循环体>一次，直到<布尔表达式>成真时结束循环

11 在 2.3.2 的翻译模式中，已经包含了有关指针类型说明的处理。参考 2.3.3 中数组元素引用的处理，试给出指针引用的翻译模式片断。设基础文法中，指针引用相关的产生式包含：

$S \rightarrow *E_1 := E_2$

$S \rightarrow \underline{\text{id}} := *E$

$E \rightarrow E_1\hat{}$

注：指针访问的 TAC 语句 $x := *y$ 和 $*x := y$

12 （*a*）以下是与语句及过程声明相关的类型检查的一个翻译模式片断：

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

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

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

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

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

$F \rightarrow$ F_1 ; $\underline{\text{id}}$ (*V*) *S* { *addtype*($\underline{\text{id}}$.*entry*, *fun* (*V.type*));

F.type := if F_1.*type* = *ok* and *S.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1.2.2中一致。

若在基础文法中增加关于 *continue* 语句的产生式

$$S \rightarrow continue$$

continue 语句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。

试在该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 *continue* 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：

$P \rightarrow D$; { *S.next* := *newlabel* } S { *gen*(*S.next* ':') }
$S \rightarrow$ if { *E.true* := *newlabel*; *E.false* := *S.next* } E then
{ S_1.*next* := *S.next* } S_1 { *S.code* := *E.code* || *gen*(*E. true* ':') || S_1.*code* }
$S \rightarrow$ while { *E.true* := *newlabel*; *E.false* := *S.next* } E do
{ S_1.*next* := *newlabel* } S_1
{ *S.code* := *gen*(S_1.*next* ':') || *E.code*|| *gen*(*E. true* ':') ||
S_1.*code* || *gen*('*goto*' S_1.*next*) }
$S \rightarrow$ { S_1.*next* := *newlabel* } S_1 ;
{ S_2.*next* := *S.next* } S_2
{ *S.code* := S_1.*code* || *gen*(S_1.*next* ':') || S_2.*code* }

其中的属性及语义函数与2.3.5中一致。

若在基础文法中增加关于 *continue* 语句的产生式

$$S \rightarrow continue$$

这里，continue 语句的执行语义为：跳出直接包含该 continue 语句的 while 循环体，并回到 while 循环的开始处重新执行循环体。

试在该*L*-翻译模式片段基础上增加针对 *continue* 语句的语义处理内容（不改变*L*-翻译模式的特征）。

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，可以产生控制语句的 *TAC* 语句序列：

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

其中的属性及语义函数与2.3.6中一致。

若在基础文法中增加关于 *continue* 语句的产生式

$S \rightarrow$ *continue*

这里，*continue* 语句的执行语义如（2）所述。

试在该 *S*-翻译模式片段基础上增加针对 *continue* 语句的语义处理内容（不改变 *S*-翻译模式的特征）。

注：可设计引入新的属性或删除旧的属性，必要时给出解释。

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.