第 3 章 语义分析

引言故事：

在编译器的工作流程中，语义分析阶段是通过形式化的方法分析语义，并将抽象语法转换成适合于中间代码生成的表示的过程。我国文字的发展过程与之类似。我国的文字发展不是直接形成了“字正方圆”的风格，而是由多种象形文字逐渐演变而来的[1]。象形文字具有“形”（形式化）和“意”（语义）相统一的特点。例如甲骨文是商代时期的一种文字，主要出现在龟甲和兽骨上。甲骨文的特点是形状独特、纹路清晰、刀刻粗犷[2]。甲骨文中的象形文字数量很多，如“人”“马”“虎”“鱼”等。这些象形文字具有明显的形象化特点，可以直观地表达事物的外形和属性。金文是西周时期的一种文字，主要出现在青铜器上[3]。金文的特点是笔划粗犷、铸造工艺精湛、字形复杂。金文中的象形文字数量逐渐减少，但是代表性的象形文字如“日”“月”“水”“火”“山”“川”等仍然存在。小篆是秦汉时期的一种文字，是中国历史上第一种统一的规范化文字[4]。小篆的特点是笔画流畅、字形规整、形制完美。虽然小篆中的象形文字数量进一步减少，但是代表性的象形文字如“口”“手”“目”“田”“木”等仍然保留。随着时间的推移，汉字不断发展和演变。在汉朝时期，出现了许多新的字形，如“日”和“月”组成的“明”等。这些新的字形通过外形组合表达语义，逐渐丰富了汉字的表达能力，同时也反映了汉语语言的演变。

从汉字的演变过程中可以看到，汉字的演变和编译器的语义分析过程有一定的相似性。在汉字的演变过程中，由于人们对事物的认识不断深入，对汉字的需求也不断提高，汉字的形态和含义也在不断变化。这与编译器中的语义分析过程有些类似。汉字的演变过程也反映了人们对事物的认识和表达方式的不断改进和提高。汉字从最初的象形文字逐渐演化为具有抽象含义的文字，这为人们的交流和表达提供了更加灵活和精准的工具。

---

[1] 洪飏,杜超月.古文字字形理据重构分类考论[J].辽宁师范大学学报(社会科学版),2020,43(06):122-128.DOI:10.16216/j.cnki.lsxbwk.202006122.

[2] 吴盛亚.甲骨文字构形理论与系统的建构——一百二十年来甲骨文字构形研究述评[J].出土文献,2023,No.14(02):60-76+156.

[3] 邓凯.金文字形构件断代法初探[J].殷都学刊,2015,36(01):103-108.

[4] 朱明月.小篆形成及演变初探[J].中国民族博览,2015,(06):58-60.

同样地，编译器的语义分析算法的不断改进也为程序的编写和解析提供了更加高效和准确的手段。语义分析过程由语法分析过程驱动，通过形式上的检查，达到理解源代码语义并将源代码翻译为中间代码的目的。编译器在对程序进行语义分析的过程中，可以识别一些语义上的错误，比如类型不匹配、变量未定义等，这些问题都是通过语法分析的形式发现的；同时，一些特定形式触发特定的动作，这些动作完成记录符号表、生成中间代码等操作。

本章要点：

语义分析是编译器在词法分析和语法分析之后的关键步骤。在该步骤中，将标识符的含义与使用相关联，从而检查每一个表达式是否拥有正确的类型，进而将抽象语法转换成更简单的、适合于生成中间代码的表示。本章介绍了形式语义学中较为常见的三种理论方法：通过上下文无关文法、属性和语义规则相结合的属性文法；在属性文法基础上增加副作用的语法制导的定义；语法制导的定义进一步增加语义动作得到的语法制导的翻译方案。语法制导的定义在属性文法和语法制导的翻译方案之间提供了一个平衡点，既能够支持属性文法的无副作用性，也能够支持翻译方案的顺序求值和任意程序片段的语义动作。

此外，本章中也讨论了对应的技术实践内容。因为实现语法制导的翻译方案，还需要有一些辅助的容器和模型：符号表和类型系统。语法制导翻译本质上是把程序转化为产生式和语义规则，语义规则的产物之一就是符号表。符号表中一个重要的内容是类型，而类型系统用于实现类型检查，类型检查是编译器尝试发现程序语义中类型错误的关键过程。

思维导图：


第三章
语义分析
主要任务
输入：语法分析树
输出：中间代码
理论方法
属性文法
语法制导的定义（SDD）
语法制导的翻译方案（SDT）
实现技术
符号表
类型系统
属性文法
上下文无关文法、属性和语义规则的结合
属性分类
综合属性 — 从子结点的属性值计算而来
继承属性 — 由父结点和兄弟结点的属性值计算而来
处理方法
依赖图 — 描述属性间的依赖关系
求值算法
环
拓扑顺序
S属性和L属性文法
S属性文法 — 每个属性都是综合属性
L属性文法 — 每个属性要么是综合属性，要么满足一定条件的继承属性
语法制导的定义
基于属性文法，增加了副作用
副作用 — 语义规则的目的是产生副作用，如打印结果等
语法制导的翻译方案
基于属性文法，对SDD的一种补充
后缀SDT
语义动作在产生式的最后
基本文法使用LR分析，且SDD是S属性的
产生式内部带有语义动作的SDT
计算继承属性的动作插入到出现前的位置
计算综合属性的动作放置在最右端
基本文法使用LL分析，且SDD是L属性的
SDT左递归的消除 — 消除带有SDD动作的左递归文法
符号表
语义规则的产物，也是实现语法制导的翻译方案所需要的辅助工具
常见数据结构
线性链表 — 符号表里所有的符号都用链表串联
平衡二叉树 — 符号用二叉树链接
散列表 — 根据关键码值直接进行访问
支持多层作用域的符号表
函数式符号表
命令式风格符号表
类型系统
类型检查 — 编译器尝试发现程序语义中类型错误的过程
类型系统分类
类型检查发生的时间 — 动/静态类型系统
类型转换的支持方式 — 强/弱类型系统

3.1 语义分析的理论方法

代码通过语法分析判断为符合语法规范之后，还需要通过进一步的语义分析来明确语言各个符号的含义。编译器在语义分析阶段的主要任务是将标识符的含义与具体使用含义相关联，从而检查每一个表达式是否拥有正确的类型，进而将抽象语法转换成更简单的、适合于生成中间代码的表示。虽然**形式语义学**（如指称语义学、公理语义学、操作语义学等）的研究已取得了许多重大的进展，但目前在实际应用中比较常见的语义描述和语义处理方法主要还是**属性文法（Attribute Grammar）**和**语法制导的翻译方案（Syntax-Directed Translation， SDT）**。

3.1.1 属性文法

属性文法也称为**属性翻译文法**，是一个上下文无关文法、属性和规则的结合体。属性文法是由 Knuth 在 1968 年首先提出的，是一种形式化方法，用于描述形式语言结构。属性文法通过与文法产生式相关联的语义规则来描述属性值，是对上下文无关文法的推广：将每个文法符号与一个语义属性的集合相关联；将每个产生式与一组语义规则相关联，这些规则用于计算该产生式中各文法符号的属性值。

属性文法设计的核心思想是，为上下文无关文法中的每一个终结符与非终结符赋予一个或多个属性值。举个例子，假设 X 是一个符号，a 是 X 的一个属性，那么我们用 $X.a$ 来表示 a 在某个标号为 X 的语法分析树节点上的值。如果我们使用对象来实现这个语法分析树的节点，那么 X 的属性可以被实现为代表 X 的节点的对象的一个数据字段。这些属性可以代表符号所包含的各种信息，如数值、布尔值和字符串等。属性中的字符串可能会变得非常长，例如，当它们表示编译器使用的中间语言代码序列时。属性就像变量一样，可以进行计算和传递，而属性计算的过程就是语义处理的过程。

简单来说，属性文法是一种用于计算语法结构属性值的形式化工具，其中每个产生式都带有一组**语义规则**，用于计算产生式中各个符号的属性值，并且这个计算过程依赖于产生式右部符号的属性值。

属性可以分成不相交的两类：**综合属性（Synthesized Attribute）**和**继承属性（Inherited Attribute）**。在语法分析树上，一个节点 *N* 的综合属性值是由其自身的属性和子节点的属性值计算而来的，用于“自下而上”传递信息；而综合属性的语法规则，则是根据产生式右部符号的属性计算左部被定义符号的综合属性。

考虑如下产生式：

产生式	语义规则
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

根据语义规则可以看出，*E* 的 *val* 属性值是由 E_1 和 *T* 的 *val* 值来定义的，因此 *val* 是 *E* 的综合属性。注意，在上述产生式和语义规则中，我们在其左部与右部均用到了同一个非终结符 *E*，为了指代与表述清晰，我们通过添加下标的形式，以表示不同位置的非终结符 *E*。例如，上述表示中，*E.val* 表示非终结符 *E* 在产生式左边时的属性 *val*，而 $E_1.val$ 表示非终结符 *E* 在产生式右边的时候的属性 *val*。本章及后续内容中均遵循这一表示方式。

类似的，在语法分析树上，一个节点 *N* 的继承属性值则是由该节点的父节点和兄弟节点的属性值计算而来的，用于“自上而下”传递信息；而继承属性的语法规则，则是根据产生式右部符号自身的属性和产生式左部被定义符号的属性，计算产生式右部符号的继承属性。

考虑如下产生式：

产生式	语义规则
$D \rightarrow T\,L$	$L.inh = T.type$

根据语义规则可以看出，*L* 的 *inh* 属性是由其左侧的兄弟节点 *T* 的 *type* 属性获得的，因此 *inh* 是 *L* 的继承属性。

终结符只有综合属性没有继承属性，其综合属性值是由词法分析器提供的词法值，终结符没有继承属性是因为终结符在语法树是叶子节点，不存在向下传递语义信息的需求。非终结符既可有综合属性也可有继承属性，文法开始符号的所有继承属性作为属性计算前的初始值。

3.1.2 基于属性文法的处理方式

基于属性文法的处理过程一般分为两个阶段：语法分析和语义计算。在语法分析阶段，程序会根据输入字符串构建一棵语法分析树，树中的每个节点表示一个语法单元。在语义计算阶段，

程序会对语法分析树进行遍历，并在每个节点上按照语义规则进行计算。处理流程可以概括如图 3.1 所示：

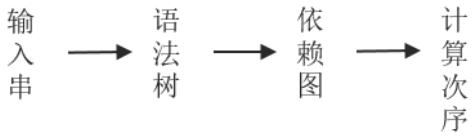

图 3.1 基于属性文法的处理过程

依赖图

在属性文法中，对于输入的字符串，我们需要使用语义规则来计算对应语法分析树上每个节点的属性值。语义规则定义了属性之间的依赖关系，因此在计算节点的属性值之前，必须先计算它所依赖的所有属性值，以确保计算的正确性和一致性。

属性之间的依赖关系可以表示为语义规则之间的**依赖图**。依赖图描述语法分析树中的属性实例之间的信息流，它可以确定一棵给定的语法分析树中各个属性实例的求值顺序。依赖图中的节点表示属性，边表示语义规则所定义的计算约束，从一个属性节点到另一个属性节点的边表示计算后一个属性节点时需要前一个属性节点的值。语法分析树中每个标号为 X 的节点的每个属性 a 都对应依赖图中的一个节点。如果属性 $X.a$ 的值依赖于属性 $Y.b$ 的值，则依赖图中有一条从 $Y.b$ 的节点指向 $X.a$ 的节点的有向边。

考虑如表 3.1 所示的产生式和语义规则[1]：

表 3.1 简单类型声明的语法制导定义

产生式	语义规则
$D \rightarrow TL$	$L.inh = T.type$
$T \rightarrow$ int	$T.type =$ **int**
$T \rightarrow$ real	$T.type =$ **real**
$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ *Addtype*(*id.lexeme*, *L.inh*)
$L \rightarrow$ **id**	*Addtype*(*id.lexeme*, *L.inh*)

[1] 该例子来源于: 《编译原理》，Alfred V. Aho等著，赵建华、郑滔和戴新宇译，机械工业出版社，第202页，2009年。

结合语法制导定义，我们可以在符号属性的计算过程中，在语法分析的基础上，对语法分析树上的各个节点进行属性值标注，我们称标注了属性值的语法分析树为**注释分析树（或注释语法分析树）**。图 3. 2 所示为输入语句 int x, y, z 在表 3. 1 的语法制导定义中产生的注释分析树所对应的依赖图。在这个依赖图中，注释分析树中的每一个属性，都对应着依赖图中的一个节点。我们将继承属性放在语法分析树中对应节点的左侧，将综合属性放在右侧。例如，*type* 是 *T* 的一个综合属性，所以放在 *T* 节点的右侧，*inh* 是 *L* 的继承属性，放在 *L* 节点的左侧。根节点表示对第一个产生式的应用，根据语义规则，*L* 的 *inh* 属性依赖于 *T* 的 *type* 属性，因此在依赖图中有一条从 *T* 的 *type* 属性指向 *L* 的 *inh* 属性的边。节点⑤表示的是对表 3. 1 中第四个产生式的第一个语义规则的应用。根据产生式的第一条语义规则可以看出子节点的 *inh* 属性依赖于父节点的 *inh* 属性。因此在依赖图中，有一条从父节点的 *inh* 属性指向子节点的 *inh* 属性的有向边。第四条产生式的第二个语义规则是副作用，即在**符号表（Symbol Table）**中将 *id.lexeme* 所代表的标识符的类型设置为 *L.inh* 所代表的类型。我们可以将其看作用来定义产生式左部的非终结符 *L* 的一个虚综合属性的规则。因此，我们在依赖图中为 *L* 设置一个虚节点⑥，*L* 的这个虚综合属性用到了子节点的 *lexeme* 属性和它自身的 *inh* 属性。因此，在依赖图中，分别从 *L* 的 *inh* 属性节点和 **id** 的 *lexeme* 属性节点引出了一条指向 *L* 的这个虚属性节点的有向边。

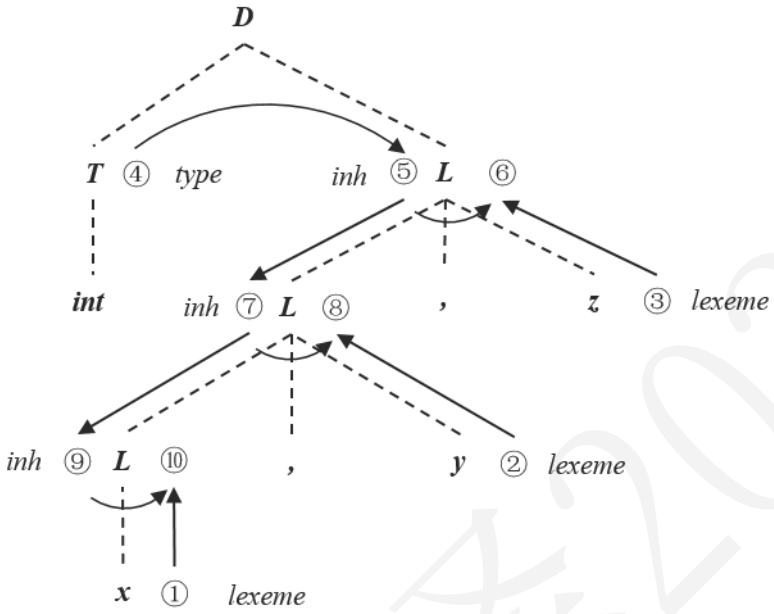

图 3.2 int x, y, z 的依赖图

属性求值的顺序

有了依赖图我们就可以设计属性值的计算顺序，如果依赖图中有一条从节点 M 到节点 N 的边，那么要先对 M 对应的属性求值，然后再对 N 对应的属性求值。因此，所有可行的求值顺序就是满足下列条件的节点顺序 N_1、N_2、…、N_k：如果有一条从节点 N_i 到 N_j 的依赖图的边，那么 $i<j$。这样的排序就将一个有向图变成了一个线性排序，这个线性排序称为这个图的**拓扑排序（Topological Sort）**。

如图 3.2 所示，依赖图中的标号①到⑩就是一个拓扑排序。首先可以计算①、②、③和④这四个节点对应的属性，然后⑤的属性依赖于④的属性，⑥的属性依赖于⑤和③的属性，⑦的属性依赖于⑤的属性，⑧的属性依赖于⑦和②的属性，⑨的属性依赖于⑦的属性，⑩的属性依赖于⑨和①的属性。依赖者的序号都大于被依赖者的序号。也就是说，在计算依赖者属性的时候，被依赖者的属性值都需要先被计算出来。

在拓扑排序中，通常会存在多种可能的节点排序方式，具体取决于节点之间的依赖关系。除了已经提到的拓扑排序以外，还可以出现以下情况：比如①、②、③和④这四个节点互不依赖，因此这四个节点的顺序可以任意调换。另外⑥和⑦的节点互不依赖，它们的顺序可以调换。⑧和

⑨的节点互不依赖，它们的顺序也可以任意调换。在实际应用中，不同的排序方式可能会对性能产生影响。因此，在进行拓扑排序时，还需要结合实际应用场景考虑，选择出最优的排序方式。

对于只具有综合属性的文法，可以按照任何自底向上的顺序计算它们的值。对于同时具有继承属性和综合属性的文法，不能保证存在一个顺序来对各个节点上的属性进行求值。比如，考虑非终结符号 *X* 和 *Y*，它们分别具有综合属性 *X.s* 和继承属性 *Y.i*。同时它们的产生式和规则如下：

产生式	语义规则
$X \rightarrow Y$	$X.s = Y.i + 1$ $Y.i = X.s - 2$

在该产生式中，由第一条语义规则可得 *X* 的 *s* 属性依赖于 *Y* 的 *i* 属性，*Y* 的 *i* 属性依赖于 *X* 的 *s* 属性，其依赖图存在一个环，两个属性是循环定义的。不可能首先求出节点 *N* 上的 *X.s* 或 *N* 的子节点上的 *Y.i* 中的任何一个值，然后再求出另一个值。

如果构建的依赖图中存在环路，那么这个图就不能进行拓扑排序。也就是说，在这种情况下，无法使用拓扑排序在语法分析树上对相应的属性求值。相反，如果依赖图中没有环路，那么总是至少存在一个拓扑排序。这是因为如果没有环路，那么必然存在至少一个没有入边的节点。假设不存在这样的节点，那么我们可以从一个前驱节点一直走到另一个前驱节点，直到回到已经访问过的节点，形成一个环路。所以我们可以将没有入边的节点作为拓扑排序的第一个节点，并从依赖图中删除这个节点，重复上述过程，直到形成一个拓扑排序。因此，在图 3. 2 中，由于依赖图没有环路，因此总是存在至少一个拓扑排序。

3.1.3 S 属性文法和 L 属性文法

文法的具体实现过程不一定完全按图 3. 1 的流程，在某些情况下可用一遍扫描实现属性文法的语义规则计算，即在语法分析的同时完成语义规则的计算，无须显式地构造语法树或构造属性之间的依赖图。一遍扫描的实现能够达到较高的编译效率，这种方法是在语法分析的同时计算属性值，而不是构造语法分析之后进行属性的计算，所以采用一遍扫描的方法可以不构造实际的语法树，且当一个属性值不再用于计算其它属性值时，编译程序可以不再保留这个属性值。

一遍扫描处理方法的实现与所采用的语法分析方法和属性的计算次序密切相关。

接下来介绍两种可用一遍扫描实现的文法：**S 属性文法（S-Attribute Grammar）**可用于一遍扫描的自下而上分析，而 **L 属性文法（L-Attribute Grammar）**可用于一遍扫描的自上而下分析。

S 属性文法

只含有综合属性的文法称为 **S 属性文法**。如果一个文法是 S 属性的，我们可以按照语法分析树节点的任何自底向上的顺序来计算它的各个属性值。对语法分析树进行后序遍历并对属性求值常常会非常简单，当遍历最后一次离开某个节点 N 时，计算出 N 的各个属性值。

S 属性的定义可以在自底向上语法分析的过程中实现，因为一个自底向上的语法分析过程对应于一次后序遍历。特别地，后序顺序精确地对应于一个 LR 分析器将一个产生式体归约成为产生式头的过程，这个过程不会显式地创建语法分析树的节点。

表 3.2 是一个加法计算的 S 属性文法的例子。其中的每个属性都是综合属性。

表 3.2 加法计算的 S 属性文法

产生式	语义规则
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow (E)$	$T.val = E.val$
$T \rightarrow$ **digit**	$T.val =$ **digit**$.lexval$

L 属性文法

L 属性文法的直观含义：在一个产生式所关联的各属性之间，依赖图的边可以从左到右，但不能从右到左（因此称为 L 属性文法，即 Left 属性文法）。

一个文法是 L 属性的当且仅当它的每个属性要么是一个综合属性，要么是满足如下条件的继承属性。假设存在一个产生式 $A \rightarrow X_1 X_2 \ldots X_n$，其右部符号 X_i $(1 \leqslant i \leqslant n)$的继承属性仅依赖于下列属性：(1) 父节点 A 的继承属性；(2) 产生式中 X_i 左边的符号 $X_1 X_2 \ldots X_{i-1}$ 的属性；(3) X_i 本身的属性，但 X_i 的全部属性不能在依赖图中形成环。

由 L 属性文法的定义我们可以得出结论：每个 S 属性文法都是 L 属性文法。

表 3.3 是一个加法计算的 L 属性文法的例子。

表 3.3 加法计算的 L 属性文法

产生式	语义规则
$E \rightarrow T\,R$	$R.inh = T.val$ $E.val = R.syn$
$R \rightarrow +\,T\,R_1$	$R_1.inh = R.inh + T.val$ $R.syn = R_1.syn$
$R \rightarrow \varepsilon$	$R.syn = R.inh$
$T \rightarrow (\,E\,)$	$T.val = E.val$
$T \rightarrow$ **digit**	$T.val =$ **digit**$.lexval$

其中的第一条规则定义继承属性 *R.inh* 时只使用了 *T.val*，且 *T* 在相应产生式中出现在 *R* 的左部，因此满足 *L* 属性的要求。第二条规则定义 $R_1.inh$ 时使用了与产生式头部相关联的继承属性 *R.inh* 及 *T.val,* 其中 *T* 在这个产生式中出现在 R_1 的左边。后面三个语义规则则是对综合属性的计算，符合 L 属性文法的要求。同时，从语法分析树的角度看，在每一种情况中，当这些规则被应用于某个节点时，它所使用的信息都来自于“上边或左边”的语法树节点，因此满足 L 属性文法的要求。

3.1.4 语法制导的定义

语法制导的定义（Syntax-Directed Definition，SDD）也是一种语义分析技术，用于在编译器或解释器中将源代码转换为目标代码或执行程序。SDD 在属性文法和语法制导的翻译方案之间提供了一个平衡点，既能够支持属性文法的无副作用性，也能够支持翻译方案的顺序求值和任意程序片段的语义动作。

具体而言，属性文法是一种规定了语法规则和相关属性计算的形式化语言，它没有副作用，并支持与依赖图一致的求值顺序。而语法制导的翻译方案则是一种在产生式体中嵌入了语义动作的上下文无关文法。

将源代码转换为目标代码或执行程序的过程，要求按照从左到右的顺序求值，并允许语义动作包含任何程序片段。

SDD 将属性文法和翻译方案结合起来，实现了对源代码的分析和转换。通过在属性文法中引入语义动作，并在分析过程中按照翻译方案中的顺序求值，SDD 可以生成目标代码或执行程序。SDD 的主要优点是能够提高编译器或解释器的性能和可维护性，同时还可以在语法分析和代码生成之间建立一个桥梁，实现更加灵活和高效的编程语言处理。

在 SDD 的使用过程中，需要进行一般属性值计算外的操作，以提升计算效率或完成某些检查等功能。我们把这些一般属性值计算外的操作称为 SDD 中的副作用（**Side Effect**）。一些语义规则的设计目的是产生**副作用**。例如，一个桌上计算器可能需要打印出计算结果，或者一个代码生成器需要将一个标识符的类型加入到符号表中。没有副作用的 SDD 我们也称为**属性文法**。

为了控制 SDD 中的副作用，我们可以采用以下方法之一：

（1）支持那些不会对属性求值产生约束的附带副作用。换向话说，如果按照依赖图的任何拓扑顺序进行属性求值时都可以产生正确的翻译结果，我们就允许这样的副作用存在。

（2）对允许的求值顺序添加约束，使得以任何允许的顺序求值都会产生相同的翻译结果。这些约束可以被看作隐含加入到依赖图中的边。

如下为一个带有副作用的例子，实现打印计算的结果：

产生式	语义规则
$L \rightarrow E$ n	*print*(*E.val*)

像 *print*(*E.val*)这样的语义规则的设计目的就是执行它的副作用。这个经过修改的 SDD 在任何拓扑顺序下都能产生相同的值，因为这个打印语句在结果被计算到 *E.val* 中之后才会被执行。

3.1.5 语法制导的翻译方案

属性文法衍生出一种非常强大的翻译模式，我们称之为**语法制导的翻译方案（Syntax - Directed Translation，SDT）**。语法制导的翻译方案 SDT 是 SDD 的一种重要补充。在 SDT 中，把属性文法中的属性文法规则用计算属性值的语义动作来表示，并用花括号“{”和“}”括起来，它们可被插入到产生式右部的任何合适的位置上，这是一种语法分析和语义动作交错的表示法。

SDT 可以看作是 SDD 的具体实施方案。在这里我们主要关注如何使用 SDT 来实现两类重要的 SDD，因为在这两种情况下，SDT 可以在语法分析过程中实现。具体而言，主要分为如下两种情况：

（1）基本文法可以用 LR 技术分析，且是 S 属性的。

将一个 S 属性文法转换为 SDT 的方法：将每个语义动作都放在产生式的最后。所有动作都在产生式最右端的 SDT 称为**后缀 SDT**。如表 3.2 所示的加法计算的 S 属性文法中，所有的语义规则都是用来计算综合属性的，因为此 SDD 的基本文法是 LR 的，并且这个 SDD 是 S 属性的，所以

这些动作可以与语法分析器的归约步骤一起正确地执行。在这种情况下，我们把所有的语义规则直接放到产生式右部的末尾即可，表 3.4 为表 3.2 所示的加法计算对应的后缀 SDT。

表 3.4 加法计算的后缀 SDT

$E \rightarrow E_1 + T$ { $E.val = E_1.val + T.val$; }
$E \rightarrow T$ { $E.val = T.val$; }
$T \rightarrow (E)$ { $T.val = E.val$; }
$T \rightarrow$ **digit** { $T.val =$ **digit**.*lexval*; }

如果一个 S 属性文法的基本文法可以使用 LR 分析技术，那么它的 SDT 可以在 LR 语法分析的过程中实现。

（2）基本文法可以用 LL 技术分析，且是 L 属性的。

将一个 L 属性文法转换为 SDT 的方法：将计算某个非终结符号 A 的继承属性的动作插入到产生式右部中紧靠在 A 的本次出现之前的位置上，将计算一个产生式左部符号的综合属性的动作放置在这个产生式右部的最右端。如表 3.3 所示的加法计算的 L 属性文法，以第一条产生式为例，我们将其转换为 SDT：第一个语义规则是计算 R 的 *inh* 属性，R 是产生式右部的符号，因此 *inh* 是 R 的继承属性，所以计算这个继承属性的语义规则应该放在 R 即将出现的位置，即 R 和 T 之间的位置；第二个语义规则是计算 T 的 *val* 属性，T 是产生式左部的非终结符，所以 *val* 是 T 的综合属性，根据转换规则，计算综合属性值的语义动作应该追加到产生式右部末尾的位置，即 T 的右侧位置。其它产生式的转换方法同第一条产生式，最终我们可以得到如表 3.5 所示的产生式内部带有语义动作的 SDT。

表 3.5 加法计算的产生式内部带有语义动作的 SDT

$E \rightarrow T$ { $R.inh = T.val$; } R { $E.val = R.syn$; }
$R \rightarrow +T$ { $R_1.inh = R.inh + T.val$; } R_1 { $R.syn = R_1.syn$; }
$R \rightarrow \varepsilon$ { $R.syn = R.inh$; }
$T \rightarrow (E)$ { $T.val = E.val$; }
$T \rightarrow$ **digit** { $T.val =$ **digit**.*lexval*; }

如果一个 L 属性文法的基本文法可以用 LL 分析技术，那么它的 SDT 可以在 LL 或 LR 语法分析的过程中实现。

理论上，语义动作可以放置在产生式的任何位置上。当一个动作左边的所有符号都被处理过后，则立即执行该语义动作。因此，如果我们有一个产生式 $B \rightarrow X\{a\}Y$，那么当我们识别到 X（如

果 *X* 是终结符号）或者所有从 *X* 推导出的终结符号（如果 *X* 是非终结符号）之后，动作 a 就会被执行。更准确地讲，如果语法分析过程是自底向上的，那么当 *X* 的此次出现位于语法分析栈的栈顶时，我们立刻执行动作 a。如果语法分析过程是自顶向下的，那么我们在试图展开 *Y* 的本次出现（如果 *Y* 是非终结符号）或者在输入中检测 *Y*（如果 *Y* 是终结符号）之前执行语义动作 a。

此外，任何 SDT 都可以按照如下的通用方法实现：

（1）忽略语义动作，对输入进行语法分析，并产生一棵语法分析树。

（2）然后检查语法分析树的每个内部节点 *N*，假设它的产生式是 *A*→*α*。将 *α* 中的各个动作当作 *N* 的附加子节点加入，使得 *N* 的子节点从左到右与 *α* 中的符号和动作完全一致。

（3）最后对这棵语法分析树进行前序遍历，并且当访问到一个以某个动作为标号的节点时立刻执行这个动作。

3.1.6 SDT 中左递归的消除

在自上而下的翻译中，语义动作是在处于相同位置上的符号被展开或是被匹配的时候执行的。为了构造没有回溯的自顶向下语法分析，必须消除文法中的左递归。在前面我们已经学习了把左递归的上下文无关文法变成非左递归的，当消除一个 SDT 的左递归时，不仅要考虑基本文法的左递归消除，同时还要考虑语义动作的处理。

只有综合属性的情况

首先考虑简单的情况，只有综合属性的情况， 表 3.4 所示是一个只计算综合属性的带有左递归的 SDT，该 SDT 用于计算表达式的值，对其消除左递归之后得到如表 3.5 所示的不含左递归的 SDT。具体改写过程为。

按照已经学习过的知识，首先我们可以把 SDT 的文法部分消除左递归，然后要把原来的文法中的语义动作变成新文法中的语义动作并保持等价性，为此我们对消除文法左递归时引入的非终结符 *R* 定义两个属性：继承属性 *inh* 和综合属性 *syn*。*R.inh* 存放的是自上而下分析中在 *R* 分析之前已有的部分表达式的值，*R.syn* 存放的是 *R* 在分析完之后完成的表达式的值。

首先看表 3.5 第一行文法 *E*→*TR* 对应的语义规则：把 *T* 扩展完之后 *T* 的值就存放在 *T.val* 里面，在扩展 *R* 之前，需要把 *T.val* 赋值给 *R* 的继承属性 *inh*，接下来就可以扩展 *R*，当 *R* 匹配完了

之后 R 的综合属性 *syn* 就有整个表达式计算的结果，即 *R.syn* 应该是 E 的值；所以在语句结束的时候 *E.val* 被赋上 *R.syn* 的值。接下来我们看第二行文法 $R \rightarrow +TR_1$ 对应的语义规则：首先匹配+，然后匹配 T，T 扩展完了之后，把父节点 R 的继承属性 *inh* 和 *T.val* 的值做加法运算，运算结果送到 R_1.*inh* 里作为 R_1 匹配之前的继承属性。当 R_1 匹配完之后同第一行的表达式，R_1 的综合属性值就是整个表达式计算之后的结果，即把 R_1.*syn* 赋值给 *R.syn*。以此类推即可构建不含左递归的 SDT。

普遍情况的推广

接下来我们将情况推广到多个递归、非递归产生式的情况。假设有一个左递归的 SDT[1]，我们可以将其抽象成如表 3.6 的形式：

表 3.6 左递归的 SDT

$A \rightarrow A_1\, Y\ \{\ A.a = g(A_1.a,\ Y.y)\ \}$
$A \rightarrow X\ \{\ A.a = f(X.x)\ \}$

这里 *A.a* 是左递归非终结符号 A 的综合属性，而 X 和 Y 是单个文法符号，分别有综合属性 *X.x* 和 *Y.y*。因为这个方案在递归的产生式中用任意的函数 g 来计算 *A.a*，而在第二个产生式中用任意函数 f 来计算 *A.a* 的值，所以这两个符号可以代表由多个文法符号组成的串，每个符号都有自己的属性。在每种情况下，f 和 g 可以把它们能够访问的属性当作它们的参数，只要这个 SDD 是 S 属性的。

消除左递归后的基础文法如表 3.7 所示。

表 3.7 消除左递归后的基础文法

$A \rightarrow X\,R$
$R \rightarrow Y\,R \mid \varepsilon$

图 3.3 指出了在新文法上的 SDT 必须做的事情。在图 3.3(a)中，我们看到的是原文法之上的后缀 SDT 的运行效果。我们将 f 应用一次，该次应用对应于产生式 $A \rightarrow X$ 的使用。然后我们应用函数 g，应用的次数与我们使用产生式 $A \rightarrow AY$ 的次数一样。因为 R 生成了 Y 的一个余部，它的翻译依赖于它左边的串，即一个形如 $XYY\ldots Y$ 的串。对产生式 $R \rightarrow YR$ 的每次使用都导致对 g 的一次应用。对于 R，我们使用一个继承属性 *R.i* 来累计从 *A.a* 的值开始不断应用 g 所得到的结果。

[1] 该例子来源于: 《编译原理》，Alfred V. Aho等著，赵建华、郑滔和戴新宇译，机械工业出版社，第211页，2009年。

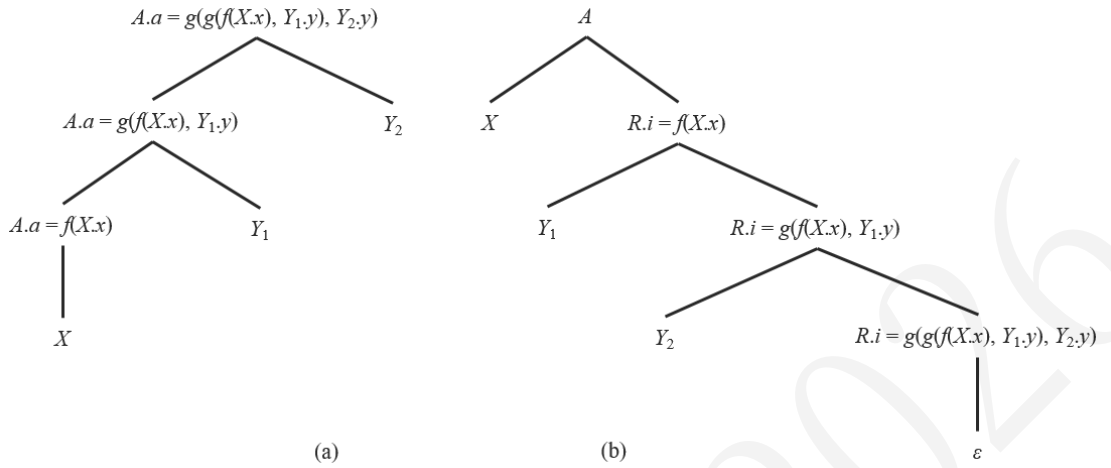

图 3. 3 消除一个后缀 SDT 中的左递归

除此之外，*R* 还有一个没有在图 3. 3 中显示的综合属性 *R.s*。当 *R* 不再生成文法符号 *Y* 时才开始计算这个属性的值，这个时间点是以产生式 $R \rightarrow \varepsilon$ 的使用为标志的。然后 *R.s* 沿着树向上复制，最后它就可以变成对应于整个表达式 *XYY…Y* 的 *A.a* 的值。从 *A* 生成 *XYY* 的情况显示在图 3. 3 中，我们看到在图 3. 3a 中的根节点上的 A.a 的值使用了两次 *g* 函数，而在图 3. 3b 的底部的 *R.i* 也使用了两次 *g* 函数，而正是这个节点上的 *R.s* 的值被沿着树向上复制。

为了完成这个翻译，我们使用如表 3.8 所示的 SDT：

表 3. 8 消除左递归后的 SDT

$A \rightarrow X \{ R.i = f(X.x) \} R \{ A.a = R.s \}$
$R \rightarrow Y \{ R_1.i = g(R.i, Y.y) \} R_1 \{ R.s = R_1.s \}$
$R \rightarrow \varepsilon \{ R.s = R.i \}$

请注意，继承属性 *R.i* 在产生式中 *R* 的使用之前完成求值，而综合属性 *A.a* 和 *R.s* 在产生式的结尾完成求值。因此，计算这些属性时需要的任何值都已经在左边计算完成，变成了可用的值。

3.1.7 类型检查

在语义分析中，另外一项重要的任务，是确保程序符合语义规范，而语言的类型系统则从根本上定义了程序中变量函数等的行为约束。在计算机程序设计语言中，“类型”包含两个要素：一组值，以及在这组值上的一系列操作。当我们在某组值上尝试去执行其不支持的操作时，类型

错误就产生了。编译器尝试去发现输入程序语义中的类型错误的过程被称为是**类型检查（Type Checking）**。为了进行类型检查，我们首先需要介绍**类型系统（Type System）**。

类型系统

一个典型程序设计语言的类型系统应该包含如下四个部分：

（1）一组基本类型。C-- 语言包括 int 和 float 两种基本类型。

（2）从一组已有类型构造新类型的规则。在 C—语言中，可以通过定义数组和结构体来构造新的类型。

（3）判断两个类型是否等价的机制。在 C—语言中，默认要求实现名等价，如果程序需要完成后续实践内容的要求 3.3，则需实现结构等价。

（4）从变量的类型推断表达式类型的规则。

根据不同的标准，程序设计语言的类型系统根据类型检查发生的时间可以分为两类，即**静态类型系统（Static Type System）**和**动态类型系统（Dynamic Type System）**。静态类型系统是指在编译时就确定了变量的类型，并且在运行时不会发生改变。在静态类型系统中，编译器会检查代码中变量的类型是否匹配，从而避免一些常见的类型错误。静态类型系统要求变量在声明时必须指定其类型，而且一旦指定类型，就不能再改变它的类型。常见的采用静态类型系统的程序设计语言包括 Java、C++、C#、Rust、Go 等。**动态类型系统**是指在运行时才确定变量的类型，变量的类型也可以随时发生改变。在动态类型系统中，通常不需要声明变量的类型，而是通过赋值来确定变量的类型。动态类型系统的优点是代码更加灵活，但也容易出现一些类型错误。常见的采用动态类型系统的程序设计语言包括 Python、Ruby、JavaScript、PHP 等。

另外，根据类型检查的严格程度，可以将类型系统分为**强类型系统**和**弱类型系统**。强类型系统要求变量在使用之前必须被明确定义，并且不能进行隐式类型转换。在强类型系统中，必须显式地进行类型转换才能将一个类型转换为另一个类型。强类型系统可以在编译时发现类型错误，可以避免一些常见的类型错误，但也会增加代码的复杂性。常见的强类型语言包括 Python、Java、C++、C#、Rust、Go 等。弱类型系统允许变量在使用时可以自动转换，不需要显式地进行类型转换。在弱类型系统中，变量的类型可以隐式转换为另一种类型，而且变量的类型也可以在运行时

动态地改变，这增加了代码的灵活性，但也会增加出错的风险。常见的弱类型语言包括 JavaScript、Ruby、PHP 等。

有了类型信息，语义分析就可以执行类型检查，检查每个运算符是否具有匹配的运算分量。类型检查验证一种结构的类型是否匹配其上下文要求。例如，下标只能用于数组；调用用户定义的函数或过程时，实参的个数和类型要与形参一致等。

类型检查分类

与类型系统的分类对应的，我们可以把程序设计语言的类型检查进行分类：根据类型检查发生的时间可以分为两类，即**静态类型检查（Static Type Checking）**和**动态类型检查（Dynamic Type Checking）**。静态类型检查在编译时进行，通过对程序中的变量和表达式进行类型分析，检查是否符合类型规则，如果存在类型错误，则编译器会在编译时报告错误。静态类型检查可以提前发现类型错误，减少程序运行时出错的可能性，但也可能会增加编程的难度和开发周期。动态类型检查是在程序运行时进行类型检查，通过对程序中的变量和表达式进行类型分析，检查是否符合类型规则。动态类型检查可以在程序运行时发现类型错误，并提供更好的灵活性，但也可能会导致运行时性能下降和更难调试。

根据是否允许隐式类型转换可以分为两类，即**强类型检查（Strong Type Checking）**和**弱类型检查（Weak Type Checking）**。强类型检查要求在进行类型转换时必须显式地进行类型转换，从而保证类型安全。在强类型检查语言中，不同类型之间不能隐式转换，如果需要进行类型转换，必须进行显式转换。弱类型检查允许变量在使用时可以自动进行类型转换，不需要显式地进行类型转换。在弱类型检查语言中，变量的类型可以隐式转换为另一种类型，而且变量的类型也可以在运行时动态地改变。

对于什么类型系统更好，学界进行了长期、激烈而又没有结果的争论。动态类型检查语言更适合快速开发和构建程序原型（因为这类语言往往不需要指定变量的类型[1]），而使用静态类型检

[1] 对于那些对变量没有类型限制的语言，有一种生动形象的说法是，这类语言采用了**鸭子类型系统**（duck typing）：如果一个东西走起来像一只鸭子、叫起来也像一只鸭子，那么它就是一只鸭子（if it walks like a duck and quacks like a duck, it's a duck）。

查语言写出来的程序通常错误更少（因为这类语言往往不允许多态）。强类型系统语言更加健壮，而弱类型系统语言在编程开发方面更加高效。总之，不同的类型系统特点不一，目前还没有哪种选择在所有情况下都比其它选择来得更好。

3.2 语义分析的实践技术

本节是本书讨论的编译器构造技术的第二部分实践内容。完成实践内容二需要在词法分析和语法分析程序的基础上编写一个程序，对C—源代码进行语义分析和类型检查，并打印分析结果。与实践内容一不同的是，实践内容二不再借助已有的工具，所有的任务都必须手写代码来完成。另外，虽然语义分析在整个编译器实现中并不是难度最大的任务，但却是最细致、琐碎的任务。因此需要用心地设计诸如符号表、变量类型等数据结构的实现细节，从而正确、高效地实现语义分析的各种功能。

3.2.1 语义分析实现思想概述

除了词法和语法分析之外，编译器前端所要进行的另一项工作就是对输入程序进行语义分析。进行语义分析的原因很简单：一段语法上正确的源代码仍可能包含严重的逻辑错误，这些逻辑错误可能会对编译器后续阶段的工作产生影响。首先，我们在语法分析阶段所借助的理论工具是上下文无关文法，从名字上就可以看出上下文无关文法没有办法处理一些与输入程序上下文相关的内容（例如变量在使用之前是否已经被定义过，一个函数内部定义的变量在另一个函数中是否允许使用等）。这些与上下文相关的内容都会在语义分析阶段得到处理，因此也有人将这一阶段叫做**上下文相关分析（Context-sensitive Analysis）**。其次，现代程序设计语言一般都会引入类型系统，很多语言甚至是强类型的。引入类型系统可以为程序设计语言带来很多好处，例如它可以提高代码在运行时刻的安全性，增强语言的表达力，还可以使编译器为其生成更高效的目标代码。对于一个具有类型系统的语言来说，编译器必须要有能力检查输入程序中的各种行为是否都是类型安全的，因为类型不安全的代码出现逻辑错误的可能性很高。最后，为了之后的阶段能够顺利

进行，编译器在面对一段输入程序时不得不从语法之外的角度进行理解。比如，假设输入程序中有一个变量或函数 *x*，那么编译器必须要提前确定：

（1）如果 *x* 是一个变量，那么变量 *x* 中存储的内容是什么？是一个整数值、浮点数值，还是一组整数值或其它自定义结构的值？

（2）如果 *x* 是一个变量，那么变量 *x* 在内存中需要占用多少字节的空间？

（3）如果 *x* 是一个变量，那么变量 x 的值在程序的运行过程中会保留多长时间？什么时候应当创建 *x*，而什么时候它又应该消亡？

（4）如果 *x* 是一个变量，那么谁该负责为 *x* 分配存储空间？是用户显式地进行空间分配，还是由编译器生成专门的代码来隐式地完成这件事？

（5）如果 *x* 是一个函数，那么这个函数要返回什么类型的值？它需要接受多少个参数，这些参数又都是什么类型？

以上这些与变量或函数 *x* 有关的信息几乎所有都无法在词法或语法分析过程中获得，即输入程序能为编译器提供的信息要远超过词法和语法分析能从中挖掘出的信息。

从编程实现的角度看，语义分析可以作为编译器里单独的一个模块，也可以并入前面的语法分析模块或者并入后面的中间代码生成模块。不过，由于其牵扯到的内容较多而且较为繁杂，我们还是将语义分析单独作为一块内容。下面我们对 C—编译中的符号表和类型表示这两大重点内容进行讨论，然后提出帮助顺利完成实践内容二的一些建议。

3.2.2 符号表的设计与实现

在编译过程中，编译器使用符号表来记录源程序中各种标识符在编译过程中特性信息。“标识符”包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等。“特性信息”包括：上述标识符的种类、具体类型、维数、参数个数、数值及目标地址（存储单元地址）等编译过程中的关键信息。

符号表上的操作包括**填表（Fill）**和**查表（Lookup）**。当分析到程序中的说明或定义语句时，应将说明或定义的名字，以及与之有关的特性信息填入符号表中，这便是填表操作。查表操作的使用场景则更加丰富，例如，填表前查表，包括检查在输入程序的同一作用域内名字是否被重复

定义，对于那些类型要求更强的语言，则要检查表达式中各变量的类型是否一致等；此外生成目标指令时，也需要查表以取得所需要的地址或者寄存器编号等。符号表有多种组织方式，可以将程序中出现的所有符号组织成一张表，也可以将不同种类的符号组织成不同的表（例如，所有变量名组织成一张表，所有函数名组织成一张表，所有临时变量组织成一张表，所有结构体定义组织成一张表，等等）。可以针对每个语句块、每个结构体都新建一张表，也可以将所有语句块中出现的符号全部插入到同一张表中。符号表可以仅支持插入操作而不支持删除操作（此时如果要实现作用域则需要将符号表组织成层次结构），也可以组织一张既可以插入又可以删除的、支持动态更新的表。不同的组织方式各有利弊，可仔细思考并为实践内容二做出决定。

语法制导翻译方案的实现，需要有一些辅助的容器和模型。符号表是其中最重要的容器，因为语法制导翻译方案本质上是把程序转化为产生式和语义规则，语义规则实现的核心媒介之一就是符号表。**符号表**也称为**环境（Environment）**，其核心作用是通过标识符映射获得其在编译过程中所需要记录的类型与存储位置等信息。在处理类型、变量和函数的声明时，标识符便与其在符号表中的编译信息相绑定。每当发现标识符的使用（即非声明性出现）时，便可以在符号表中查看它们在对应作用域中的相关信息。

符号表有各种各样的数据结构实现形式，不同的数据结构有不同的时间和空间复杂度，我们下面讨论几种最常见的选择：

线性链表：

线性链表（Linear linked list）由一系列节点组成，每个节点包含两个成员：数据和指向下一个节点的指针。节点的数据可以是任意类型的，指针则指向下一个节点，最后一个节点的指针为空（null）。由于每个节点都只有一个指针，所以这种数据结构称为线性链表。

用线性链表实现的符号表，表里所有的符号（假设有 n 个，下同）都用一条链表串起来，插入一个新的符号只需将该符号放在链表的表头，其时间复杂度是 O(1)。在链表中查找一个符号需要对其进行遍历，时间复杂度是 O(n)。删除一个符号只需要将该符号从链表里摘下来，不过在摘之前由于我们必须要执行一次查找操作以找到待删除的节点，因此时间复杂度也是 O(n)。

链表的最大问题是它的查找和删除效率太低，一旦符号表中的符号数量较大，查表操作将变得十分耗时。不过，使用链表的好处也是显而易见：它的结构简单，编程容易，可以被快速实现。如果能够确定表中的符号数目较少（例如，在结构体定义中或在面向对象语言的一些短方法中），链表是一个非常不错的选择。

平衡二叉树：

平衡二叉树 （Balanced Binary Tree）是一种二叉搜索树，它的左子树和右子树的高度差不超过 1。也就是说，它是一种保证在最坏情况下，树的高度为 O(log *n*)的二叉搜索树。相对于只能执行线性查找的链表而言，在平衡二叉树上进行查找就是二分查找。在一个典型的平衡二叉树实现（如 AVL 树、红黑树或伸展树[1]等）上查找一个符号的时间复杂度是 O(log*n*)。插入一个符号需要找到插入的位置，相当于进行一次失败的查找，时间复杂度也是 O(log*n*)。删除一个符号可能需要做更多的维护操作，但其时间复杂度仍然维持在 O(log*n*)的级别。

平衡二叉树相对于其它数据结构而言具有很多优势，例如较高的搜索效率（在绝大多数应用中 O(log*n*)的搜索效率已经完全可以接受）以及较好的空间效率（它所占用的空间随树中节点的增多而增长）。平衡二叉树的缺点是编程难度高，成功写完并调试出一个较好的红黑树需要较多的时间。

散列表：

散列表也称为**哈希表（Hash Table）**，是根据**关键码值对 (Key Value)**而直接进行访问的数据结构。散列表通过哈希函数将键映射到一个桶中，每个桶中存储一个或多个键值对。当需要访问一个键值对时，先通过哈希函数计算该键所对应的桶，然后在桶中查找对应的值。一个好的散列表实现可以让插入、查找和删除的平均时间复杂度都达到 O(1)。同时，与红黑树等操作复杂的数据结构不同，散列表在代码实现上也很简单：申请一个大数组，计算一个散列函数的值，然后根据该值将对应的符号放到数组相应下标的位置即可。对于符号表来说，一个最简单的散列函数（即

[1] 《数据结构与算法分析——C语言描述》，Mark Allen Weiss著，冯舜玺译，机械工业出版社，第80、351和89页，2004年。

hash 函数）可以把符号名中的所有字符相加，然后对符号表的大小取模。也可以寻找更好的 hash 函数，这里我们提供一个不错的选择，由 P.J. Weinberger[1]所提出：

```
1  unsignedinthash_pjw(char* name)
2  {
3    unsignedint val = 0, i;
4    for (; *name; ++name)
5    {
6      val = (val << 2) + *name;
7      if (i = val & ~0x3fff) val = (val ^ (i >> 12)) & 0x3fff;
8    }
9    return val;
10  }
```

需要注意的是，此处代码第 7 行的常数（0x3fff）确定了符号表的大小（即 16384），我们在实现中可根据实际需要调整此常数以获得大小合适的符号表。如果散列表出现冲突，则可以通过在相应数组元素下面挂一个链表的方法[即闭合地址法（Closed Addressing），也 称为链接法（Chaining）] 来解决问题。这种方法在哈希表中为每个桶分配一个链表，当出现哈希冲突时，新的元素可以插入到相应桶的链表末尾。也可以通过再次计算散列函数的值而为当前符号寻找另一个槽的方式 [即开放地址法（Open Addressing），也称为探测法（Probing）]来解决冲突问题。这种方法在哈希表中为每个桶分配一个探测序列（Probing Sequence），当出现哈希冲突时，新的元素可以通过探测序列的方式依次尝试放入相邻的位置，直到找到一个空的位置为止。或使用一些更酷的技术，使散列表中的元素分布更加平均一些，如：**乘法哈希函数（Multiplicative Hash Function）**，其基本思想是将关键字乘以一个常数因子，然后取结果的小数部分作为哈希值；通用**哈希函数（Universal Hash Function）**其设计原则是，对于任意给定的关键字集合，都应该能够设计出一个哈希函数族，使得每个哈希函数在这个关键字集合上的哈希冲突概率都很小。散列表在搜索效率和编程难度上的优异表现，使它成为符号表的实现中最常被采用的数据结构。

至于符号表里应该填些什么，这与不同程序设计语言的特性相关，更取决于编译器的设计者本身。只要是为了实现更好地生成目标代码，可以向符号表里填任何内容，因为符号表就是为了支持编写编译器而设置的。就实践内容二而言，对于变量，符号表至少要记录变量名及其类型；对于函数，符号表至少要记录其返回类型、参数个数以及参数类型这些信息。

[1] http://en.wikipedia.org/wiki/Peter_J._Weinberger。

3.2.3 支持多层作用域的符号表

如果编译器不需要支持变量的作用域（即不需要实现后续实践内容中的要求 3.2），那可以跳过本节内容，不会对实践内容二的完成产生负面的影响。如果编译器需要支持变量的作用域，在现实中有函数式风格和命令式风格两种选择。考虑下面这段代码：

```
1  …
2  int f()
3  {
4    int a, b, c;
5    …
6    a = a + b;
7    if (b > 0)
8    {
9      int a = c * 2;
10      b = b - a;
11    }
12    …
13  }
14  …
```

函数 f 中定义了变量 a，在 if 语句中也定义了一个变量 a。如果要支持作用域，那么第一，编译器不能在“`int a = c * 2;`”这个地方报错；第二，语句“`a = a + b;`”中的 a 的值应该取外层定义中 a 的值，语句“`b = b - a;`”中的 a 的值应该是 if 语句内部定义的 a 的值，而这两个语句中 b 的值都应该取外层定义中“`int a, b, c;`”中 b 的值。为了使符号表支持这样的行为，可采用以下的方法。

函数式风格 Functional Style：

第一种方法是维护一个符号表栈。假设当前函数 f 有一个符号表，表里有 a、b、c 这三个变量的定义。当编译器发现函数中出现了一个被“{”和“}”包含的语句块（在 C—语言中就相当于发现了 CompSt 语法单元）时，它会将 f 的符号表压栈，然后新建一个符号表，这个符号表里只有变量 a 的定义。当语句块中出现任何表达式使用到某个变量时，编译器先查找当前的符号表，如果找到就使用这个符号表里的该变量，如果找不到则顺着符号表栈向下逐个符号表进行查找，使用第一个查找成功的符号表里的相应变量。如果查遍所有的符号表都找不到这个变量，则报告当前语句出现了变量未定义的错误。每当编译器离开某个语句块时，会先销毁当前的符号表，然后从栈中弹一个符号表出来作为当前的符号表。这种符号表的维护风格被称为**函数式风格**。该维护风格最多会申请 *d* 个符号表，其中 *d* 为语句块的最大嵌套层数。这种风格比较适合于采用链表

或红黑树数据结构的符号表实现。假如符号表采用的是散列表数据结构，申请多个符号表无疑会占用大量的空间。

命令式风格 Imperative Style：

命令式风格不会申请多个符号表，而是自始至终在单个符号表上进行动态维护。假设编译器在处理到当前函数 f 时符号表里有 a、b、c 三个变量的定义。当编译器发现函数中出现了一个被“{”和“}”包含的语句块，而在这个语句块中又有新的变量定义时，它会将该变量插入 f 的符号表里。当语句块中出现任何表达式使用某个变量时，编译器就查找 f 的符号表。如果查找失败，则报告一个变量未定义的错误；如果查表成功，则返回查到的变量定义；如果出现了变量既在外层又在内层被定义的情况，则要求符号表返回最近的那个定义。每当编译器离开某个语句块时，会将这个语句块中定义的变量全部从表中删除。

命令式风格对符号表的数据结构有一定的要求。图 3. 4 是一个满足要求的基于十字链表和 open hashing 散列表的 Imperative Style 的符号表设计。这种设计的初衷很简单：除了散列表本身为了解决冲突问题所引入的链表之外，它从另一维度也引入链表将符号表中属于同一层作用域的所有变量都串起来。在图中，a、x 同属最外层定义的变量，i、j、var 同属中间一层定义的变量，i、j 同属最内层定义的变量。其中 i、j 这两个变量有同名情况，被分配到散列表的同一个槽内。每次向散列表中插入元素时，总是将新插入的元素放到该槽下挂的链表以及该层所对应的链表的表头。每次查表时如果定位到某个槽，则按顺序遍历这个槽下挂的链表并返回这个槽中符合条件的第一个变量，如此一来便可以保证：如果出现了变量既在外层又在内层被定义的情况，符号表能够返回最内层的那个定义（当然最内层的定义不一定在当前这一层，因此我们还需要符号表能够为每个变量记录一个深度信息）。每次进入一个语句块，需要为这一层语句块新建一个链表用来串联该层中新定义的变量；每次离开一个语句块，则需要顺着代表该层语句块的链表将所有本层定义变量全部删除。

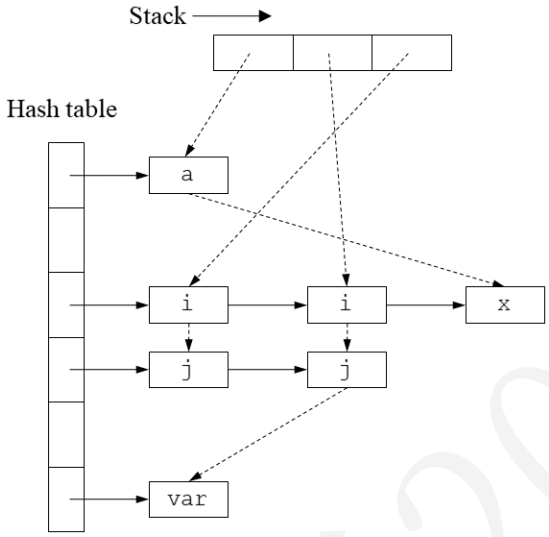

图 3.4 基于十字链表和 open hashing 散列表的符号表

如何处理作用域是语义分析的一大重点也是难点。考虑到实现难度，实践内容二并没有对作用域做过多要求，但现实世界中的动态作用域将更难实现，某些与作用域相关的问题甚至涉及代码生成与运行时刻环境。

3.2.4 类型表示

如果整个语言中只有基本类型，那么类型的表示将会极其简单：我们只需用不同的常数代表不同的类型即可。但是，在引入了数组（尤其是多维数组）以及结构体之后，类型的表示就不那么简单了。

如果某个数组的每一个元素都是结构体类型，而这个结构体中又有某个域是多维数组，那么最简单的表示方法还是链表。多维数组的每一维都可以作为一个链表节点，每个链表节点存两个内容：数组元素的类型，以及数组的大小。例如，`int a[10][3]`可以表示为图 3.5 所示的形式。

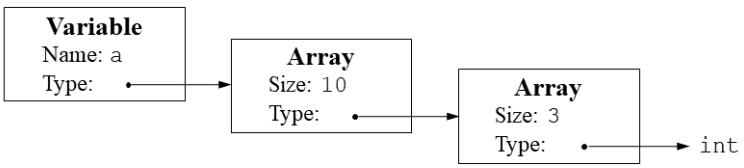

图 3.5 多维数组的链表表示示例

结构体同样也可以使用链表保存。例如，结构体 `struct SomeStruct { float f; float array[5]; int array2[10][10]; }` 可以表示为图 3.6 所示的形式。

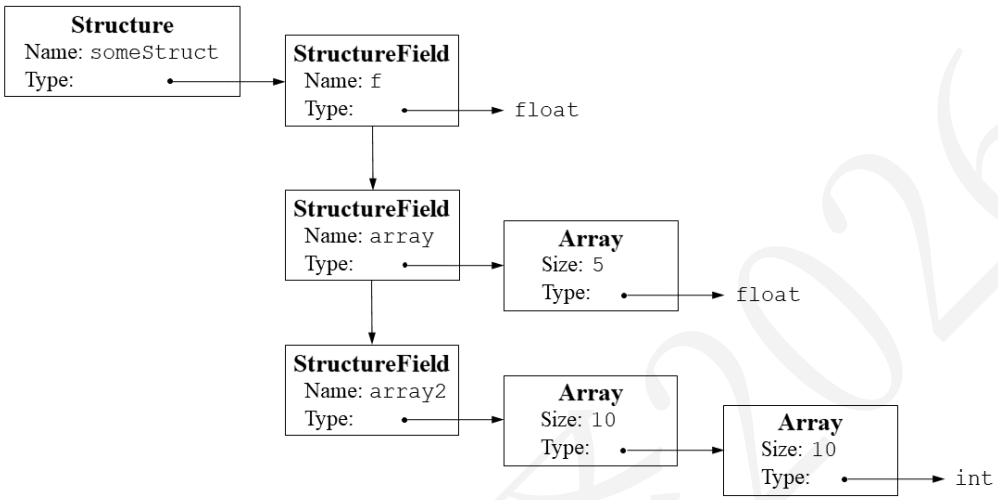

图 3.6 结构体的链表表示示例

在代码实现上，可以使用如下定义的 Type 结构来表示 C—语言中的类型：

```
typedef struct Type_* Type;
typedef struct FieldList_* FieldList;

struct Type_
{
  enum { BASIC, ARRAY, STRUCTURE } kind;
  union
  {
    // 基本类型
    int basic;
    // 数组类型信息包括元素类型与数组大小构成
    struct { Type elem; int size; } array;
    // 结构体类型信息是一个链表
    FieldList structure;
  } u;
};

struct FieldList_
{
  char* name;  // 域的名字
  Type type;  // 域的类型
  FieldList tail;  // 下一个域
};
```

同作用域一样，类型系统也是语义分析的一个重要的组成部分。C—语言属于强类型系统，并且进行静态类型检查。当我们尝试着向 C—语言中添加更多的性质，例如引入指针、面向对象、显式/隐式类型转换、类型推断等机制时，会发现实现编译器的复杂程度会陡然上升。一个严谨的

类型检查机制需要通过将类型规则转化为形式系统，并在这个形式系统上进行逻辑推理。为了控制实践内容的难度，我们不要求完成这些复杂的机制，但需要明白的是，实用的编译器内部的类型检查要复杂的多。

3.2.5 语义分析实践的额外提示

后续的实践内容二需要在实践内容一的基础上完成，特别是需要在实践内容一所构建的语法树上完成。实践内容二仍然需要对语法树进行遍历以进行符号表的相关操作以及类型的构造与检查。可以模仿 SDT 在 Bison 代码中插入语义分析的代码，但我们更推荐的做法是，Bison 代码只用于构造语法树，而把和语义分析相关的代码都放到一个单独的文件中去。如果采用前一种做法，所有语法节点的属性值请尽量使用综合属性；如果采用后一种做法，就没有这些限制。

每当遇到语法单元 ExtDef 或者 Def，就说明该节点的子节点们包含了变量或者函数的定义信息，这时候应当将这些信息通过对子节点们的遍历提炼出来并插入到符号表里。每当遇到语法单元 Exp，说明该节点及其子节点们会对变量或者函数进行使用，这个时候应当查符号表以确认这些变量或者函数是否存在以及它们的类型是什么。具体如何进行插入与查表，取决于符号表和类型系统的实现。实践内容二要求检查的错误类型较多，因此代码需要处理的内容也较复杂，请仔细完成。还有一点值得注意，在发现一个语义错误之后不要立即退出程序，因为实践内容要求中有说明需要程序有能力查出输入程序中的多个错误。

实践内容要求的必做内容共有 17 种语义错误需要检查，大部分只涉及到查表与类型操作，不过有一个错误例外，那就是有关左值的错误。简单地说，左值代表地址，它可以出现在赋值号的左边或者右边；右值代表数值，它只能出现在赋值号的右边。变量、数组访问以及结构体访问一般既有左值又有右值，但常数、表达式和函数调用一般只有右值而没有左值。例如，赋值表达式 `x=3` 是合法的，但 `3=x` 是不合法的；`y=x+3` 是合法的，但 `x+3=y` 是不合法的。简单起见，可以只从语法层面来检查左值错误：赋值号左边能出现的只有 ID、Exp LB、Exp RB 以及 Exp DOT ID，而不能是其它形式的语法单元组合。最后 5 种语义错误都与结构体有关，如前所述，结构体可以使用链表来表示。

实践要求 3.1 与函数声明有关，函数声明需要在语法中添加产生式，并在符号表中记录每个函数当前的状态：是被实现了，还是只被声明未被实现。实践要求 3.2 涉及作用域，作用域的实现方法前文已经讨论过。实践要求 3.3 为实现结构等价，对于结构等价来说，只需要在判断两个类型是否相等时不是直接去比较类型名，而是针对结构体中的每个域逐个进行类型比较即可。

3.3 语义分析的实践内容

3.3.1 实践要求

在本次实践内容中，我们对 C—语言做如下假设，可以认为这些就是 C—语言的特性（注意，假设 3、4、5 可能因后面的不同选做要求而有所改变）：

（1）假设 1：**整型（int）**变量不能与**浮点型（float）**变量相互赋值或者相互运算。

（2）假设 2：仅有 int 型变量才能进行逻辑运算或者作为 if 和 while 语句的条件；仅有 int 型和 float 型变量才能参与算术运算。

（3）假设 3：任何函数只进行一次定义，无法进行函数声明。

（4）假设 4：所有变量（包括函数的形参）的作用域都是全局的，即程序中所有变量均不能重名。

（5）假设 5：结构体间的类型等价机制采用**名等价（Name Equivalence）**的方式。

（6）假设 6：函数无法进行嵌套定义。

（7）假设 7：结构体中的域不与变量重名，并且不同结构体中的域互不重名。

以上七个假设也可视为要求，违反即会导致各种语义错误，不过我们只对后面讨论的 17 种错误类型进行考察。此外，可以安全地假设输入文件中不包含注释、八进制数、十六进制数、以及指数形式的浮点数，也不包含任何词法或语法错误（除了特别说明的针对选做要求的测试）。

程序需要对输入文件进行语义分析（输入文件中可能包含函数、结构体、一维和高维数组）并检查如下类型的错误：

（1）错误类型 1：变量在使用时未经定义。

（2）错误类型 2：函数在调用时未经定义。

（3）错误类型 3：变量出现重复定义，或变量与前面定义过的结构体名字重复。

（4）错误类型 4：函数出现重复定义（即同样的函数名出现了不止一次定义）。

（5）错误类型 5：赋值号两边的表达式类型不匹配。

（6）错误类型 6：赋值号左边出现一个只有右值的表达式。

（7）错误类型 7：操作数类型不匹配或操作数类型与操作符不匹配（例如整型变量与数组变量相加减，或数组（或结构体）变量与数组（或结构体）变量相加减）。

（8）错误类型 8：return 语句的返回类型与函数定义的返回类型不匹配。

（9）错误类型 9：函数调用时实参与形参的数目或类型不匹配。

（10）错误类型 10：对非数组型变量使用“[…]”（数组访问）操作符。

（11）错误类型 11：对普通变量使用“(…)”或“()”（函数调用）操作符。

（12）错误类型 12：数组访问操作符“[…]”中出现非整数（例如 a[1.5]）。

（13）错误类型 13：对非结构体型变量使用“.”操作符。

（14）错误类型 14：访问结构体中未定义过的域。

（15）错误类型 15：结构体中域名重复定义（指同一结构体中），或在定义时对域进行初始化（例如 struct A {int a=0;}）。

（16）错误类型 16：结构体的名字与前面定义过的结构体或变量的名字重复。

（17）错误类型 17：直接使用未定义过的结构体来定义变量。

其中，要注意三点：一是关于数组类型的等价机制，同 C 语言一样，只要数组的基类型和维数相同我们即认为类型是匹配的，例如 int a[10][2]和 int b[5][3]即属于同一类型；二是我们允许类型等价的结构体变量之间的直接赋值（见后面的测试样例），这时的语义是，对应的域相应赋值(数组域也如此,按相对地址赋值直至所有数组元素赋值完毕或目标数组域已经填满)；三是对于结构体类型等价的判定，每个匿名的结构体类型我们认为均具有一个独有的隐藏名字，以此进行名等价判定。

除此之外，程序可以选择完成以下部分或全部的要求：

（1）*要求 3.1*：修改前面的 C—语言假设 3，使其变为“函数除了在定义之外还可以进行声明”。函数的定义仍然不可以重复出现，但函数的声明在相互一致的情况下可以重复出现。任一函数无论声明与否，其定义必须在源文件中出现。在新的假设 3 下，程序还需要检查两类新的错误和增加新的产生式：

（a）*错误类型 18*：函数进行了声明，但没有被定义。

（b）*错误类型 19*：函数的多次声明互相冲突（即函数名一致，但返回类型、形参数量或者形参类型不一致），或者声明与定义之间互相冲突。

（c）由于 C—语言文法中并没有与函数声明相关的产生式，因此需要先对该文法进行适当修改。对于函数声明来说，我们并不要求支持像“`int foo(int, float)`”这样省略参数名的函数声明。在修改的时候要留意，改动应该以不影响其它错误类型的检查为原则。

（2）*要求 3.2*：修改前面的 C—语言假设 4，使其变为“变量的定义受可嵌套作用域的影响，外层语句块中定义的变量可在内层语句块中重复定义（但此时在内层语句块中就无法访问到外层语句块的同名变量），内层语句块中定义的变量到了外层语句块中就会消亡，不同函数体内定义的局部变量可以相互重名”。在新的假设 4 下，完成错误类型 1 至 17 的检查。

（3）*要求 3.3*：修改前面的 C—语言假设 5，将结构体间的类型等价机制由名等价改为**结构等价**（Structural Equivalence）。例如，虽然名称不同，但两个结构体类型 `struct a {int x; float y;}`和 `struct b {int y; float z;}`仍然是等价的类型。注意，在结构等价时不要将数组展开来判断，例如 `struct A {int a; struct {float f; int i;} b[10];}`和 `struct B {struct {int i; float f;} b[10]; int b;}`是不等价的。在新的假设 5 下，完成错误类型 1 至 17 的检查。

3.3.2 输入格式

程序的输入是一个包含 C—源代码的文本文件，该源代码中可能会有语义错误。程序需要能够接收一个输入文件名作为参数。例如，假设程序名为 cc、输入文件名为 test1、程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行 `./cc test1` 即可获得以 test1 作为输入文件的输出结果。

3.3.3 输出格式

实践内容二要求通过标准输出打印程序的运行结果。对于那些没有语义错误的输入文件，程序不需要输出任何内容。对于那些存在语义错误的输入文件，程序应当输出相应的错误信息，这些信息包括错误类型、出错的行号以及说明文字，其格式为：

```
Error type [错误类型] at Line [行号]: [说明文字].
```

说明文字的内容没有具体要求，但是错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准。请严格遵守实践内容要求中给定的错误分类，否则将影响实践内容评分。

输入文件中可能包含一个或者多个错误（但每行最多只有一个错误），程序需要将它们全部检查出来。当然，有些时候输入文件中的一个错误会产生连锁反应，导致别的地方出现多个错误（例如，一个未定义的变量在使用时由于无法确定其类型，会使所有包含该变量的表达式产生类型错误），我们只会去考察程序是否报告了本质错误（如果难以确定哪个错误是本质错误，建议报告所有发现的错误）。但是，如果源程序里有错而程序没有报错或报告的错误类型不对，又或者源程序里没有错但程序却报错，都会影响实践内容评分。

3.3.4 验证环境

程序将在如下环境中被编译并运行（同实践内容一）：

（1）GNU Linux Release: Ubuntu 20.04, kernel version 5.13.0-44-generic；

（2）GCC version 7.5.0；

（3）GNU Flex version 2.6.4；

（4）GNU Bison version 3.5.1。

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，也基本上不会出现兼容性方面的问题。注意，实践内容二的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在程序中使用它们。

3.3.5 提交要求

实践内容二要求提交如下内容（同实践内容一）：

（1）Flex、Bison 以及 C 语言的可被正确编译运行的源程序。

（2）一份 PDF 格式的实践内容报告，内容包括：

（a）程序实现了哪些功能？简要说明如何实现这些功能。清晰的说明有助于助教对程序所实现的功能进行合理的测试。

（b）程序应该如何被编译？可以使用脚本、makefile 或逐条输入命令进行编译，请详细说明应该如何编译程序。无法顺利编译将导致助教无法对程序所实现的功能进行任何测试，从而丢失相应的分数。

（c）实践内容报告的长度不得超过三页！所以实践内容报告中需要重点描述的是程序中的亮点，是提交者认为最个性化、最具独创性的内容，而相对简单的、任何人都可以做的内容则可不提或简单地提一下，尤其要避免大段地向报告里贴代码。实践内容报告中所出现的最小字号不得小于 5 号字（或英文 11 号字）。

3.3.6 样例（必做部分）

实践内容二的样例包括必做内容样例与选做要求样例两部分，分别对应于实践内容要求中的必做内容和选做要求。请仔细阅读样例，以加深对实践内容要求以及输出格式要求的理解。这节列举必做内容样例。

样例 1：

输入：

```
1  int main()
2  {
3    int i = 0;
4    j = i + 1;
5  }
```

输出：

样例输入中变量“j”未定义，因此程序可以输出如下的错误提示信息：

```
Error type 1 at Line 4: Undefined variable "j".
```

样例 2：

输入：

```
1  int main()
2  {
3    int i = 0;
4    inc(i);
5  }
```

输出：

样例输入中函数“inc”未定义，因此程序可以输出如下的错误提示信息：

```
Error type 2 at Line 4: Undefined function "inc".
```

样例 3：

输入：

```
1  int main()
2  {
3    int i, j;
4    int i;
5  }
```

输出：

样例输入中变量“i”被重复定义，因此程序可以输出如下的错误提示信息：

```
Error type 3 at Line 4: Redefined variable "i".
```

样例 4：

输入：

```
 1  int func(int i)
 2  {
 3    return i;
 4  }
 5
 6  int func()
 7  {
 8    return 0;
 9  }
10
11  int main()
12  {
13  }
```

输出：

样例输入中函数“func”被重复定义，因此程序可以输出如下的错误提示信息：

```
Error type 4 at Line 6: Redefined function "func".
```

样例 5：

输入：

```
1  int main()
2  {
3    int i;
4    i = 3.7;
5  }
```

输出：

样例输入中错将一个浮点常数赋值给一个整型变量，因此程序可以输出如下的错误提示信息：

```
Error type 5 at Line 4: Type mismatched for assignment.
```

样例 6：

输入：

```
1  int main()
2  {
3    int i;
4    10 = i;
5  }
```

输出：

样例输入中整数“10”出现在了赋值号的左边，因此程序可以输出如下的错误提示信息：

```
Error type 6 at Line 4: The left-hand side of an assignment must be a variable.
```

样例 7：

输入：

```
1  int main()
2  {
3    float j;
4    10 + j;
5  }
```

输出：

样例输入中表达式“10+j”的两个操作数的类型不匹配，因此程序可以输出如下的错误提示信息：

```
Error type 7 at Line 4: Type mismatched for operands.
```

样例 8：

输入：

```
1  int main()
2  {
3    float j = 1.7;
4    return j;
```

```
5  }
```

输出：

样例输入中“main”函数返回值的类型不正确，因此程序可以输出如下的错误提示信息：

```
Error type 8 at Line 4: Type mismatched for return.
```

样例 9：

输入：

```
1  int func(int i)
2  {
3    return i;
4  }
5
6  int main()
7  {
8    func(1, 2);
9  }
```

输出：

样例输入中调用函数“func”时实参数目不正确，因此程序可以输出如下的错误提示信息：

```
Error type 9 at Line 8: Function "func(int)" is not applicable for arguments

"(int, int)".
```

样例 10：

输入：

```
1  int main()
2  {
3    int i;
4    i[0];
5  }
```

输出：

样例输入中变量“i”不是数组型变量，因此程序可以输出如下的错误提示信息：

```
Error type 10 at Line 4: "i" is not an array.
```

样例 11：

输入：

```
1  int main()
2  {
3    int i;
4    i(10);
5  }
```

输出：

样例输入中变量“i”不是函数，因此程序可以输出如下的错误提示信息：

```
Error type 11 at Line 4: "i" is not a function.
```

样例 12：

输入：

```
1  int main()
2  {
3    int i[10];
4    i[1.5] = 10;
5  }
```

输出：

样例输入中数组访问符中出现了非整型常数“1.5”，因此程序可以输出如下的错误提示信息：

```
Error type 12 at Line 4: "1.5" is not an integer.
```

样例 13：

输入：

```
 1  struct Position
 2  {
 3    float x, y;
 4  };
 5
 6  int main()
 7  {
 8    int i;
 9    i.x;
10  }
```

输出：

样例输入中变量“i”不是结构体类型变量，因此程序可以输出如下的错误提示信息：

```
Error type 13 at Line 9: Illegal use of ".".
```

样例 14：

输入：

```
 1  struct Position
 2  {
 3    float x, y;
 4  };
 5
 6  int main()
 7  {
 8    struct Position p;
 9    if (p.n == 3.7)
10      return 0;
11  }
```

输出：

样例输入中结构体变量“p”访问了未定义的域“n”，因此程序可以输出如下的错误信息：

```
Error type 14 at Line 9: Non-existent field "n".
```

样例 15：

输入：

```
1  struct Position
2  {
3    float x, y;
4    int x;
5  };
6
7  int main()
8  {
9  }
```

输出：

样例输入中结构体的域“x”被重复定义，因此程序可以输出如下的错误信息：

```
Error type 15 at Line 4: Redefined field "x".
```

样例 16：

输入：

```
 1  struct Position
 2  {
 3    float x;
 4  };
 5
 6  struct Position
 7  {
 8    int y;
 9  };
10
11  int main()
12  {
13  }
```

输出：

样例输入中两个结构体的名字重复，因此程序可以输出如下的错误信息：

```
Error type 16 at Line 6: Duplicated name "Position".
```

样例 17：

输入：

```
1  int main()
2  {
```

```
3    struct Position pos;
4  }
```

输出：

样例输入中结构体“Position”未经定义，因此程序可以输出如下的错误信息：

```
Error type 17 at Line 3: Undefined structure "Position".
```

3.3.7 样例（选做部分）

这节列举选做要求样例。

样例 1：

输入：

```
 1  int func(int a);
 2
 3  int func(int a)
 4  {
 5    return 1;
 6  }
 7
 8  int main()
 9  {
10  }
```

输出：

如果程序需要完成要求 3.1，这个样例输入不存在任何词法、语法或语义错误，因此不需要输出错误信息。

如果程序不需要完成要求 3.1，这个样例输入存在语法错误，因此程序可以输出如下的错误提示信息：

```
Error type B at Line 1: Incomplete definition of function "func".
```

样例 2：

输入：

```
 1  struct Position
 2  {
 3    float x,y;
 4  };
 5
 6  int func(int a);
 7
 8  int func(struct Position p);
 9
10  int main()
11  {
```

```
12  }
```

输出：

如果程序需要完成要求 3.1，这个样例输入存在两处语义错误：一是函数“func”的两次声明不一致；二是函数“func”未定义，因此程序可以输出如下的错误提示信息：

```
Error type 19 at Line 8: Inconsistent declaration of function "func".

Error type 18 at Line 6: Undefined function "func".
```

注意，我们对错误提示信息的顺序不做要求。

如果程序不需要完成要求 3.1，这个样例输入存在两处语法错误，因此程序可以输出如下的错误提示信息：

```
Error type B at Line 6: Incomplete definition of function "func".

Error type B at Line 8: Incomplete definition of function "func".
```

样例 3：

输入：

```
 1  int func()
 2  {
 3    int i = 10;
 4    return i;
 5  }
 6
 7  int main()
 8  {
 9    int i;
10    i = func();
11  }
```

输出：

如果程序需要完成要求 3.2，这个样例输入不存在任何词法、语法或语义错误，因此不需要输出错误信息。

如果程序不需要完成要求 3.2，样例输入中的变量“i”被重复定义，因此程序可以输出如下的错误信息：

```
Error type 3 at Line 9: Redefined variable "i".
```

样例 4：

输入：

```
 1  int func()
 2  {
 3    int i = 10;
 4    return i;
 5  }
 6
 7  int main()
 8  {
 9    int i;
10    int i, j;
11    i = func();
12  }
```

输出：

如果程序需要完成要求 3.2，样例输入中的变量“i”被重复定义，因此程序可以输出如下的错误提示信息：

```
Error type 3 at Line 10: Redefined variable "i".
```

如果程序不需要完成要求 3.2，样例输入中的变量“i”被重复定义了两次，因此程序可以输出如下的错误提示信息：

```
Error type 3 at Line 9: Redefined variable "i".
Error type 3 at Line 10: Redefined variable "i".
```

样例 5：

输入：

```
 1  struct Temp1
 2  {
 3    int i;
 4    float j;
 5  };
 6
 7  struct Temp2
 8  {
 9    int x;
10    float y;
11  };
12
13  int main()
14  {
15    struct Temp1 t1;
16    struct Temp2 t2;
17    t1 = t2;
18  }
```

输出：

如果程序需要完成要求 3.3，这个样例输入不存在任何词法、语法或语义错误，因此不需要输出错误信息。

如果程序不需要完成要求 3.3，样例输入中的语句“t1 = t2;”其赋值号两边变量的类型不匹配，因此程序可以输出如下的错误提示信息：

```
Error type 5 at Line 17: Type mismatched for assignment.
```

样例 6：

输入：

```
 1  struct Temp1
 2  {
 3    int i;
 4    float j;
 5  };
 6
 7  struct Temp2
 8  {
 9    int x;
10  };
11
12  int main()
13  {
14    struct Temp1 t1;
15    struct Temp2 t2;
16    t1 = t2;
17  }
```

输出：

如果程序需要完成要求 3.3，样例输入中的语句“t1=t2;”其赋值号两边变量的类型不匹配，因此程序可以输出如下的错误提示信息：

```
Error type 5 at Line 16: Type mismatched for assignment.
```

如果程序不需要完成要求 3.3，应该输出与上述一样的错误提示信息：

```
Error type 5 at Line 16: Type mismatched for assignment.
```

3.4 本章小结

在本章中，我们已经详细讨论了语义分析的理论以及实践技术。语义分析的任务是在代码通过语法分析判断为符合语法规范之后，进一步将标识符的定义与使用相关联，从而检查每一个表达式是否拥有正确的类型，进而将抽象语法转换成更简单的、适合于生成中间代码的表示。

本章首先介绍了形式语义学中目前较流行的三种理论方法：通过上下文无关文法、属性和语义规则相结合的属性文法；在属性文法基础上增加副作用的语法制导的定义；语法制导的定义进

一步增加语义动作得到的语法制导的翻译方案。语法制导的定义在属性文法和语法制导的翻译方案之间提供了一个平衡点，既能够支持属性文法的无副作用性，也能够支持翻译方案的顺序求值和任意程序片段的语义动作。

此外，本章中也讨论了对应的技术实践内容，因为实现语法制导的翻译方案，还需要有一些辅助的容器和模型：符号表和类型系统。符号表是其中最重要的容器，因为语法制导翻译本质上是把程序转化为产生式和语义规则，语义规则的产物之一就是符号表。符号表中一个重要的内容是类型，类型系统是为了实现类型检查，类型检查是编译器尝试发现程序语义中类型错误的关键过程。

习题

3.1 假设有如下文法：

S → *id* := *E* | *if B then S* | *while B do S* | *begin S* ; *S end* | *break*

写出一个翻译方案，其任务是：若发现 break 未出现在循环语句中，则报告错误。

3.2 下面文法产生代表正二进制数的 0 和 1 的串集:：

B → *B 0* | *B 1* | *1*

下面的翻译方案计算这种正二进制数的十进制值：

B → *B10* {*B.val= B1.val*2*} *B* → *B11*{*B.val= B1.val*2+1*} *B* → *1*{*B.val= 1*}

请消除该基础文法的左递归，再重写一个翻译方案,，使得它仍然计算这种正二进制数的十进制值.。

3.3 设有 PASCAL 程序：

```
 2  PROGRAM p;
 3  VAR a,b,c,d,e: real;
 4  PROCEDURE a;
 5  VAR c,e,f,g: real;
 6  BEGIN
 7  …
 8  c;
 9  …
10  END;
11  PROCEDURE b;
12  VAR e, d: integer;
13  BEGIN;
14  …
15  END;
16  PROCEDURE c;
17  VAR h:real;
18  f:ARRAY\[1‥10\] OF integer;
19  BEGIN
20  …
21  END;
22  BEGIN
23  …
```

```
24  END.
```

试给出编译器对此程序建立的可能符号表（假设一个 integer 型变量的长度为 4 个字节，一个 real 型变量的长度为 8 个字节）。

3.4 设有 PASCAL 程序：

```
 1  PROGRAM p;
 2  VAR a,b,c,d,e: real;
 3  PROCEDURE a;
 4  VAR c,e,f,g: real;
 5  PROCEDURE b;
 6  VAR e, d: integer;
 7  BEGIN
 8  …
 9  c;
10  …
11  END;
12  BEGIN;
13  …
14  END;
15  PROCEDURE c;
16  VAR h:real;
17  f:ARRAY\[1··10\] OF integer;
18  BEGIN
19  …
20  END;
21  BEGIN
22  …
23  END.
```

试给出编译器对此程序建立的可能符号表（假设一个 integer 型变量的长度为 4 个字节，一个 real 型变量的长度为 8 个字节）。

3.5 假设有如下的 Pascal 程序段：

```
1  type link=↑cell;
2  var next: link;
3      last: link;
4      r: ↑cell;
5      s,t: ↑cell;
```

程序段中的 5 个名字：next、last、r、s 和 t，哪些是结构等价？哪些是名字等价？

3.6．给定表 3.1 所示的 SDD，图 3.7 是句子 real id, id 的注释分析树的依赖图。图 3.7 中的全部拓扑顺序有哪些？

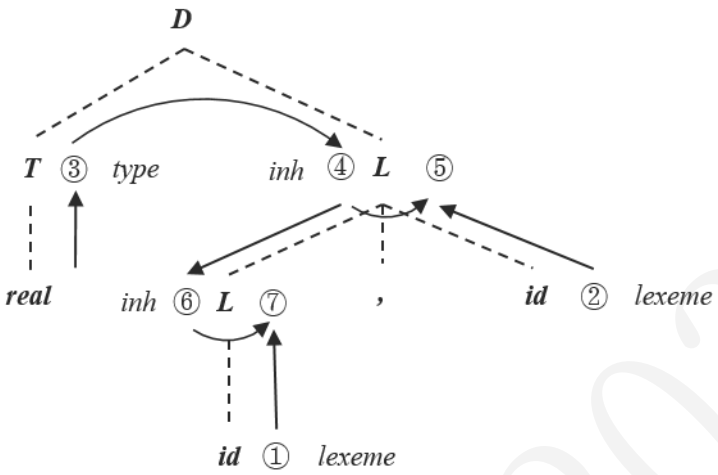

图 3. 7 句子 real id, id 的注释分析树的依赖图

3.7．假设有一个产生式 A →BCD。非终结符 A、B、C、D 各自都有两个属性：一个综合属性 s 和一个继承属性 i。对于下面的每组规则：

(1) $A.s = B.s + D.i$
(2) $A.s = B.i + C.s$
$B.i = C.i + D.s$
(3) $A.s = C.s + D.s$
(4) $A.s = C.s + D.i$
$B.i = A.s$
$C.i = B.s$
$D.i = B.i + C.s$

请指出：

① 这些规则是否满足 S 属性定义的要求？

② 这些规则是否满足 L 属性定义的要求？

③ 是否存在和这些规则一致的求值过程？