现代编译原理——ML 语言描述

作者：Andrew Appel

目录

第一部分　编译基本原理 **4**

1　绪论 **5**

1.1　模块与接口 5
1.2　工具和软件 6
1.3　树语言的数据结构 8
1.4　程序设计：直线式程序解释器 10
1.5　习题 12

2　词法分析 **13**

2.1　词法标记 13
2.2　正则表达式 15
2.3　有限自动机 17
2.4　非确定性有限自动机 20
2.4.1　将正则表达式转换为 NFA 21
2.4.2　将 NFA 转换为 DFA 22
2.5　ML-Lex：词法分析器的生成器 25
2.6　程序设计：词法分析 28
2.7　推荐阅读 29
2.8　习题 29

3　语法分析 **33**

3.1　上下文无关文法 35
3.1.1　推导 35
3.1.2　语法分析树 36
3.1.3　二义性文法 36
3.1.4　文件结束符 38
3.2　预测分析 39
3.2.1　FIRST 集合和 FOLLOW 集合 40
3.2.2　构造预测分析器 43
3.3　LR 分析 44
3.4　使用分析器的生成器 44
3.5　错误恢复 45
3.6　程序设计：语法分析 45
3.7　推荐阅读 45

3.8 习题 45

4 抽象语法 46

4.1 语义动作 46

4.1.1 递归下降 46

4.1.2 ML-Yacc 生成的语法分析器 47

5 语义分析 50

5.1 符号表 50

5.1.1 多个符号表 51

5.1.2 高效的命令式风格符号表 52

5.1.3 高效的函数式符号表 53

5.1.4 Tiger 编译器的符号 55

5.1.5 命令式风格的符号表 57

5.2 Tiger 编译器的绑定 57

5.3 表达式的类型检查 60

5.4 声明的类型检查 62

5.4.1 变量声明 62

5.4.2 类型声明 63

5.4.3 函数声明 63

5.4.4 递归声明 64

5.5 程序设计：类型检查 65

5.6 习题 65

6 活动记录 67

7 翻译成中间代码 68

8 基本块和轨迹 69

9 指令选择 70

10 活跃分析 71

11 寄存器分配 72

12 整合为一体 73

第二部分 高级主题 74

13 垃圾收集 75

14 面向对象的语言 **76**

15 函数式程序设计语言 **77**

16 多态类型 **78**

17 数据流分析 **79**

18 循环优化 **80**

19 静态单赋值形式 **81**

20 流水线化和调度 **82**

21 存储层次 **83**

21.1 cache 的组织结构 83

前言

近十余年来，编译器的构建方法出现了一些新的变化。一些新的程序设计语言得到应用，例如，具有动态方法的面向对象语言、具有嵌套作用域和一等函数闭包（first-class function closure）的函数式语言等。这些语言中有许多都需要垃圾收集技术的支持。另一方面，新的计算机都有较大的寄存器集合，且存储器访问成为了影响性能的主要因素。这类机器在具有指令调度能力并能对指令和数据高速缓存（cache）进行局部性优化的编译器辅助下，常常能运行得更快。

本书可作为一到两个学期编译课程的教材。学生将看到编译器不同部分中隐含的理论，学习到将这些理论付诸实现时使用的程序设计技术和以模块化方式实现该编译器时使用的接口。为了清晰具体地给出这些接口和程序设计的例子，我使用 ML 语言来编写它们。本（序列）书还有使用 C 和 Java 语言的另外两种版本。

实现项目。我在书中概述了一个“学生项目编译器”，它相当简单，而且其安排方式也便于说明现在常用的一些重要技术。这些技术包括避免语法和语义相互纠缠的抽象语法树，独立于寄存器分配的指令选择，能使编译器前期阶段有更多灵活性的复写传播，以及防止从属于特定目标机的方法。与其他许多教材中的“学生编译器”不同，本书中采用的编译器有一个简单而完整的后端，它允许在指令选择之后进行寄存器分配。

本书第一部分中，每一章都有一个与编译器的某个模块对应的程序设计习题。在

http://www.cs.princeton.edu/~appel/modern/ml

中可找到对这些习题有帮助的一些软件。

习题。每一章都有一些书面习题：标有一个星号的习题有点挑战性，标有两个星号的习题较难但仍可解决，偶尔出现的标有三个星号的习题是一些尚未找到解决方法的问题。

授课顺序。图1展示了各章相互之间的依赖关系。

- 一学期的课程可包含第一部分的所有章节（第 1~12 章），同时让学生实现项目编译器（多半按项目组的方式进行）。另外，授课内容中还可以包含从第二部分中选择的一些主题。
- 高级课程或研究生课程可包含第二部分的内容，以及另外一些来自其他文献的主题。第二部分中有许多章节和第一部分无关，因此，对于那些在初始课程中使用不同教材的学生而言，仍然可以给他们讲授高级课程。
- 若按两个半个学期来安排教学，则前半学期可包含第 1~8 章，后半学期包括第 9~12 章和第二部分的某些章。

致谢。对于本书，许多人给我提出了富有建设性的意见，或在其他方面给我提供了帮助。我要感谢这些人，他们是 Leonor Abraido-Fandino，Scott Ananian，Stephen Bailey，Maia Ginsburg，Max Hailperin，David Hanson，Jeffrey Hsu，David MacQueen，Torben Mogensen，Doug Morgan，Robert Netzer，Elma Lee Noah，Mikael Petterson，Todd Proebsting，Anne Rogers，Barbara Ryder，Amr Sabry，Mooly Sagiv，Zhong Shao，Mary Lou Soffa，Andrew

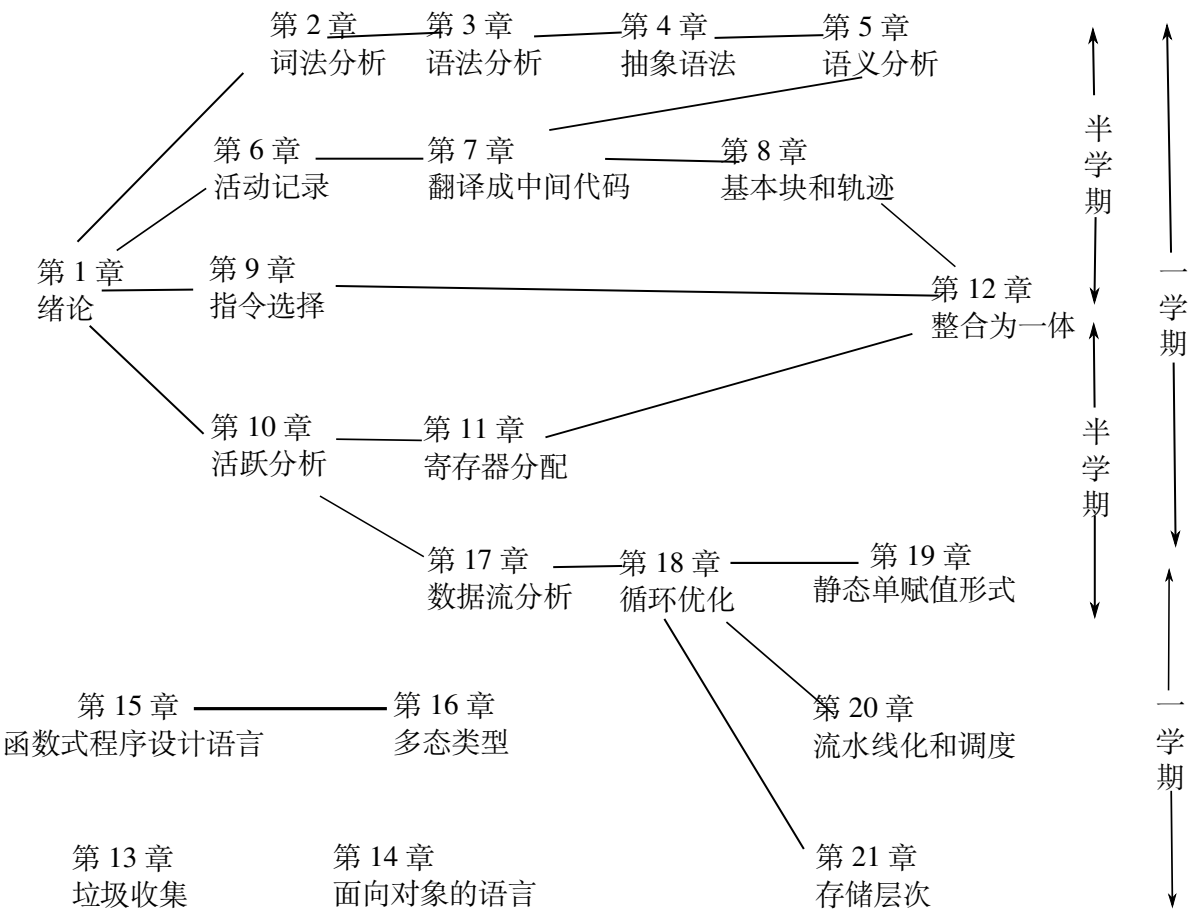

第 2 章
词法分析
第 3 章
语法分析
第 4 章
抽象语法
第 5 章
语义分析
第 6 章
活动记录
第 7 章
翻译成中间代码
第 8 章
基本块和轨迹
第 1 章
绪论
第 9 章
指令选择
第 12 章
整合为一体
第 10 章
活跃分析
第 11 章
寄存器分配
第 17 章
数据流分析
第 18 章
循环优化
第 19 章
静态单赋值形式
第 15 章
函数式程序设计语言
第 16 章
多态类型
第 20 章
流水线化和调度
第 13 章
垃圾收集
第 14 章
面向对象的语言
第 21 章
存储层次
半学期
半学期
一学期
一学期

图 1: 内容结构图

Tolmach，Kwangkeun Yi 和 Kenneth Zadeck。

第一部分

编译基本原理

第一章 绪论

编译器（compiler）：原指一种将各个子程序装配组合到一起的程序 [连接-装配器]。当 1954 年出现了（确切地说是误用了）复合术语“代数编译器”（algebraic compiler）之后，这个术语的意思变成了现在的含义。

Bauer 和 Eickel[1975]

本书讲述将程序设计语言转换成可执行代码时使用的技术、数据结构和算法。现代编译器常常由多个阶段组成，每一阶段处理不同的抽象“语言”。本书的章节按照编译器的结构来组织，每一章循序渐进地论及编译器的一个阶段。

为了阐明编译真实的程序设计语言时遇到的问题，本书以 Tiger 语言为例来说明如何编译一种语言。Tiger 语言是一种类 Algol 的语言，它有嵌套的作用域和在堆中分配存储空间的记录，虽简单却并不平凡。每一章的程序设计练习都要求实现相应的编译阶段；如果学生实现了本书第一部分讲述的所有阶段，便能够得到一个可以运行的编译器。将 Tiger 修改成*函数式的*或*面向对象的*（或同时满足两者的）语言并不难，第二部分中的习题说明了如何进行这种修改。第二部分的其他章节讨论了有关程序优化的高级技术。附录描述了 Tiger 语言。

编译器各模块之间的接口几乎和模块内部的算法同等重要。为了具体描述这些接口，较好的做法是用真正的程序设计语言来编写它们，本书使用的是 ML 语言——一种严格的，具有模块系统的，静态类型的函数式编程语言。ML 语言适合用来编写很多类型的应用程序。但如果使用 ML 语言来实现编译器，似乎能最大限度的利用 ML 语言中的一些强大特性，同时无需使用 ML 语言的一些缺陷特性。使用 ML 语言来实现一个编译器，是一个很愉快的过程。而且，对于一本完备的编译器教材来讲，书中需要引入一些现代编程语言设计的教学内容。

1.1 模块与接口

对于任何大型软件系统，如果设计者注意到了该系统的基本抽象和接口，那么对这个系统的理解和实现就要容易得多。图1.1展示了一个典型的编译器的各个阶段，每个阶段由一至多个软件模块来实现。

将编译器分解成这样的多个阶段是为了能够重用它的各种构件。例如，当要改变此编译器所生成的机器语言的目标机时，只要改变栈帧布局（Frame Layout）模块和指令选择（Instruction Selection）模块就够了。当要改变被编译的源语言时，则至多只需改变翻译（Translate）模块之前的模块就可以了，该编译器也可以在*抽象语法*（Abstract Syntax）接口处与面向对象的语法编辑器相连。

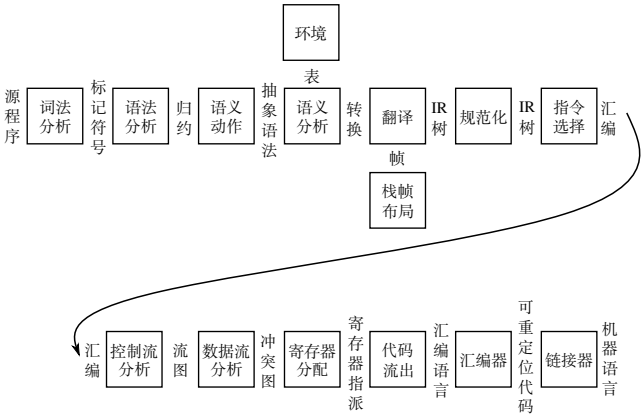


图 1.1: 编译器的各个阶段及其之间的接口

每个学生都不应缺少反复多次“思考-实现-重新设计”，从而获得正确的抽象这样一种学习经历。但是，想要学生在一个学期内实现一个编译器是不现实的。因此，我在书中给出了一个项目框架，其中的模块和接口都经过深思熟虑，而且尽可能地使之既精巧又通用。

抽象语法（Abstract Syntax）、IR 树（IR Tree）和汇编（Assem）之类的接口是数据结构的形式，例如语法分析动作阶段建立抽象语法数据结构，并将它传递给语义分析阶段。另一些接口是抽象数据类型：翻译接口是一组可由语义分析阶段调用的函数；标记符号（Token）接口是函数形式，分析器通过调用它而得到输入程序中的下一个标记符号。

各个阶段的描述

第一部分的每一章各描述编译器的一个阶段，具体如表1.2所示。

这种模块化设计是很多真实编译器的典型设计。但是，也有一些编译器把语法分析、语义分析、翻译和规范化合并成一个阶段，还有一些编译器将指令选择安排在更后一些的位置，并且将它与代码流出合并在一起。简单的编译器通常没有专门的控制流分析、数据流分析和寄存器分配阶段。

我在设计本书的编译器时尽可能地进行了简化，但并不意味着它是一个简单的编译器。具体而言，虽然为简化设计而去掉了一些细枝末节，但该编译器的结构仍然可以允许增加更多的优化或语义而不会违背现存的接口。

1.2 工具和软件

现代编译器中使用的两种最有用的抽象是上下文无关文法（context-free grammar）和正则表达式（regular expression）。上下文无关文法用于语法分析，正则表达式用于词法分析。为了更好地利用这两种抽象，较好的做法是借助一些专门的工具，例如 Yacc（它将文法转换成语法分析器）和 Lex（它将一个说明性的规范转换成一个词法分析器）。幸运的是，ML 语言提供了这些工具的比较好的版本，所以本书中的项目 ML 提供的工具来描述。

本书中的编程项目可以使用 Standard ML of New Jersey 系统来编译，这个系统中还包含了像 ML-Yacc、ML-Lex 以及 Standard ML of New Jersey Software Library。所有这些

表 1.2: 编译器的各阶段

章号	阶段	描述
2	词法分析	将源文件分解成一个个独立的*标记符号*
3	语法分析	分析程序的短语结构
4	语义动作	建立每个短语对应的*抽象语法树*
5	语义分析	确定每个短语的含义，建立变量和其声明的关联，检查表达式的类型，翻译每个短语
6	栈帧布局	按机器要求的方式将变量、函数参数等分配于活动记录（即栈帧）内
7	翻译	生成*中间表示树*（IR 树），这是一种与任何特定程序设计语言和目标机体系结构无关的表示
8	规范化	提取表达式中的副作用，整理条件分支，以方便下一阶段的处理
9	指令选择	将 IR 树结点组合成与目标机指令的动作相对应的块
10	控制流分析	分析指令的顺序并建立*控制流图*，此图表示程序执行时可能流经的所有控制流
10	数据流分析	收集程序变量的数据流信息。例如，*活跃分析*（liveness analysis）计算每一个变量仍需使用其值的地点（即它的*活跃点*）
11	寄存器分配	为程序中的每一个变量和临时数据选择一个寄存器，不在同一时间活跃的两个变量可以共享同一个寄存器
12	代码流出	用机器寄存器替代每一条机器指令中出现的临时变量名

工具都可以在因特网上免费获取；具体信息可以查看网页：

http://www.cs.princeton.edu/~appel/modern/ml。

Tiger 编译器中某些模块的源代码、某些程序设计习题的框架源代码和支持代码、Tiger 程序的例子以及其他一些有用的文件都可以从该网址中找到。本书的程序设计习题中，当提及特定子目录或文件所在的某个目录时，指的是目录 $TIGER/。

1.3 树语言的数据结构

编译器中使用的许多重要数据结构都是被编译程序的中间表示。这些表示常常采用树的形式，树的结点有若干类型，每一种类型都有一些不同的属性。这种树可以作为图 1-1 所示的许多阶段的接口。

树表示可以用文法来描述，就像程序设计语言一样。为了介绍有关概念，我将给出一种简单的程序设计语言，该语言有语句和表达式，但是没有循环或 if 语句 [这种语言称为直线式程序（straight-line program）语言]。

该语言的语法在文法1.3中给出。

Stm	→	*Stm* ; *Stm*	(CompoundStm)
Stm	→	id := *Exp*	(AssignStm)
Stm	→	print (*ExpList*)	(PrintStm)
Exp	→	id	(IdExp)
Exp	→	num	(NumExp)
Exp	→	*Exp Binop Exp*	(OpExp)
Exp	→	(*Stm* , *Exp*)	(EseqExp)
ExpList	→	*Exp* , *ExpList*	(PairExpList)
ExpList	→	*Exp*	(LastExpList)
Binop	→	+	(Plus)
Binop	→	−	(Minus)
Binop	→	×	(Times)
Binop	→	/	(Div)

文法 1.3: 直线式程序设计语言

这个语言的非形式语义如下。每一个 *Stm* 是一个语句，每一个 *Exp* 是一个表达式。$s_1; s_2$ 表示先执行语句 s_1，再执行语句 s_2。$i := e$ 表示先计算表达式 e 的值，然后把计算结果赋给变量 i。print$(e_1, e_2, \cdots, e_n)$ 表示从左到右输出所有表达式的值，这些值之间用空格分开并以换行符结束。

标识符表达式，例如 i，表示变量 i 的当前内容。数按命名它的整数计值。运算符表达式 e_1 op e_2 表示先计算 e_1 再计算 e_2，然后按给定的二元运算符计算表达式结果。表达式序列 (s, e) 的行为类似于 C 语言中的逗号运算符，在计算表达式 e（并返回其结果）之前先计算语句 s 的副作用。

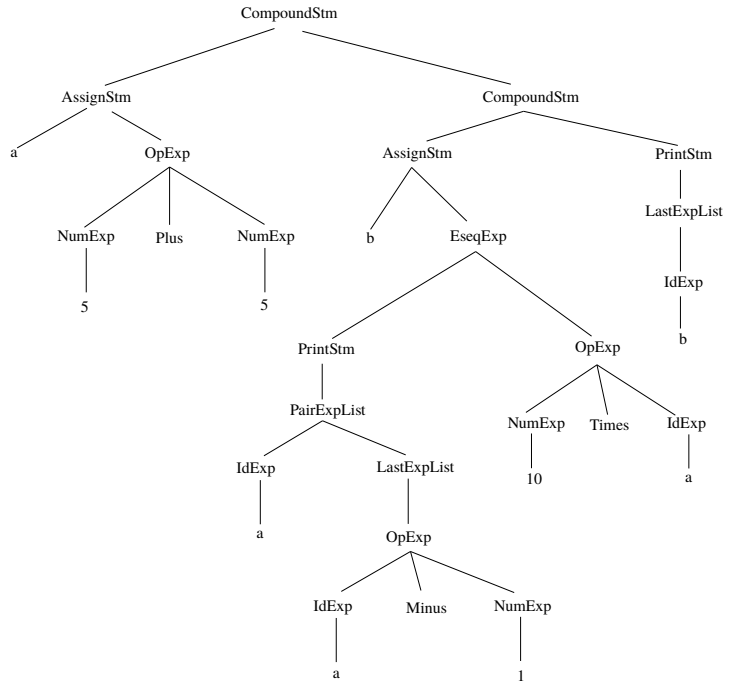
例如，执行下面这段程序：

```
a := 5+3; b := (print(a, a-1), 10*a); print(b);
```

将打印出：

```
8 7
80
```

那么，这段程序在编译器内部是如何表示的呢？一种表示是源代码形式，即程序员所编写的字符，但这种表示不易处理。较为方便的表示是树数据结构。每一条语句（Stm）和每一个表达式（Exp）都有一个树结点。图1.4给出了这个程序的树表示，其中结点都用文法1.3中产生式的标识加以标记，并且每个结点的子节点数量与相应文法产生式右边的符号个数相同。



a := 5 + 3 ; b := (print (a , a - 1) , 10 * a) ; print (b)

图 1.4: 直线式程序的树形表示

我们可以将这个文法直接翻译成数据结构定义，如程序1.5所示。每个文法符号对应于这些数据结构中的一个 type。

每一项文法规则都有一个构造器（constructor），隶属于规则左部符号的类型（type）。ML 语言的 datatype 声明语法可以非常漂亮地表达这些树形结构。这些构造器的名字在文法1.3各项右部的括号内。

ML 程序的模块化规则。编译器是一个很大的程序，仔细地设计模块和接口能避免混乱。在用 ML 语言编写一个编译器时，我们将使用如下一些规则。

1. 编译器的每个阶段或者模块都应该归入各自的 structure。
2. 我们将不会使用 open 声明。如果一个 ML 文件以如下开头：

   ```
   open A.F; open A.G; open B; open C;
   ```

   那么你（一个人类读者）将必须查看一下这个文件之外的代码来确定 X.put() 表达式中的 X 是在哪一个 structure 中定义的。

   structure 的缩略形式将会是一个比较好的解决方案。如果一个模块以如下开头：

```
type id = string

datatype binop = Plus | Minus | Times | Div

datatype stm = CompoundStm of stm * stm
             | AssignStm of id * stm
             | PrintStm of exp list

     and exp = IdExp of id
             | NumExp of int
             | OpExp of exp * binop * exp
             | EseqExp of stm * exp
```

程序 1.5: 直线式程序的表示

```
structure W=A.F.W and X=A.G.X and Y=B.Y and Z=C.Z
```

那么你无需查看这个文件外的代码就可以确定 X 来自 A.G。

1.4 程序设计：直线式程序解释器

为直线程序设计语言实现一个简单的程序分析器和解释器。对环境（即符号表，它将变量名映射到这些变量相关的信息）、抽象语法（表示程序的短语结构的数据结构）、树数据结构的递归性（它对于编译器中很多部分都是非常有用的）以及无赋值语句的函数式风格程序设计，这可作为入门练习。

这个练习也可以作为 ML 语言程序设计的热身。熟悉其他语言但对 ML 语言陌生的程序员应该也能完成这个习题，只是需要有关 ML 语言的辅助资料（如教材）的帮助。

需要进行解释的程序已经被分析为抽象语法，这种抽象语法如程序 1-5 中的数据类型所示。

但是，我们并不希望涉及该语言的具体分析细节，因此利用了相应数据的构造器来编写该程序：

```
val prog =
 CompoundStm(AssignStm("a",OpExp(NumExp 5, Plus, NumExp 3)),
  CompoundStm(AssignStm("b",
      EseqExp(PrintStm[IdExp "a",OpExp(IdExp "a", Minus,
                                       NumExp 1)],
           OpExp(NumExp 10, Times, IdExp "a"))),
   PrintStm[IdExp "b"]))
```

在目录 $TIGER/chap1 中可以找到包含树的数据类型声明的文件以及这个样板程序。

编写没有副作用（即更新变量和数据结构的赋值语句）的解释器是理解指称语义（denotational semantic）和属性文法（attribute grammar）的好方法，后两者都是描述程序设

计语言做什么的方法。对编写编译器而言，它也常常是很有用的技术，因为编译器也需要知道程序设计语言做的是什么。

因此，在实现这些程序时，不要使用引用变量，数组或者赋值表达式等 ML 语言的语法特性。

1. 写一个函数 (maxargs : stm → int)，告知给定语句中所有子表达式内的 print 语句中包含最大参数数量的 print 语句的参数个数。例如，maxargs(prog) 是 2。
2. 写一个函数 interp : stm → unit，对一个用这种直线式程序语言写的程序进行“解释”。使用“函数式”的风格来编写这个函数——不使用赋值（:=）或者数组特性——维护一个（变量，整型）偶对[1]所组成的列表，然后再解释每个 AssignStm 时，产生这个列表的新版本。

对于第一个程序，要记住 print 语句可能会包含一些表达式，而这些表达式中又包含了其他的 print 语句。

对于第二个程序，编写两个互相递归调用的函数 interpStm 和 interpExp。构造一个“表”，将标识符映射到赋值给标识符的整型数值，“表”使用 id × int 偶对所组成的列表来实现。那么 interpStm 的类型是：stm × table → table，如果表 t_1 作为参数的话，那么返回值将会是一个新的表 t_2，t_2 和 t_1 基本相同。不同的是，作为语句的执行结果，一些标识符被映射到了一些不同的整型数值。

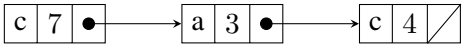
例如，表 t_1 中 a 映射到了 3，c 映射到了 4，我们将 t_1 写成 $\{a \mapsto 3, c \mapsto 4\}$ 这样的数学符号，还可以将 t_1 写成链表的形式 

，写成 ML 代码是`("a",3)::("c",4)::nil`。

现在，令表 t_2 就像表 t_1，不同的是，c 映射到了 7 而不是 4。我们可以将这个过程写为以下数学形式：

$t_2 = update(t_1, c, 7)$

其中函数 update 返回一个新表 $\{a \mapsto 3, c \mapsto 7\}$。

在计算机中，只要我们假设在链表中 c 的第一次出现优先于它较后的任何出现，就可以通过在表头插入一个新元素来实现新表 t_2 

。

因此，update 函数很容易实现，而与之相应的 lookup 函数

```
val lookup : table * id -> int
```

则只要沿着链表从头向后搜索即可。

表达式的解释要比语句的解释复杂一些，因为表达式返回整型数值且有副作用。我们希望解释器本身在模拟直线程序设计语言的赋值语句时不产生任何副作用（但是 print 语句将有解释器的副作用来实现）。实现它的方法是将 interpExp 的类型设计成 exp × table → int × table。用表 t_1 解释表达式 e_1 的结果是得到一个整型数值 i 和一个新表 t_2。当解释一个含有两个子表达式的表达式（例如 OpExp）时，由第一个子表达式得到的表 t_2 可以继续用于处理第二个子表达式。

[1]pair

1.5 习题

1. 下面这个简单的程序实现了一种持久化（persistent）函数式二叉搜索树，使得如果`tree2 = insert(x, tree1)`，则当使用 tree2 时，tree1 仍然可以继续用于查找。

```
type key = string
datatype tree = LEAF | TREE of tree * key * tree

val empty = LEAF

fun insert(key, LEAF) = TREE(LEAF, key, LEAF)
  | insert(key, TREE(l, k, r)) =
                if key < k
                  then TREE(insert(key, l), k, r)
                else if key > k
                  then TREE(l, k, insert(key, r))
                else TREE(l, key, r)
```

   (a). 实现函数 member，若查找到了相应项，返回 true，否则返回 false。

   (b). 扩充这个程序使其不仅包含成员关系，而且还包含了键值（key）到绑定的映射。

```
datatype 'a tree = ...
insert : 'a tree * key * 'a -> 'a tree
lookup : 'a tree * key -> 'a
```

   (c). 这个程序构造的树是不平衡的；用下述插入顺序说明树的形成过程：

   I. t s p i p f b s t

   II. a b c d e f g h i

   (*d). 研究 Sedgewick[1997] 中讨论过的平衡搜索树，并为函数式符号表推荐一种平衡树数据结构。**提示：**为了保持函数式风格，算法应该在插入时而不是在查找时保持树的平衡，因此，不适合使用类似于伸展树（splay tree）这样的数据结构。

第二章 词法分析

词法的（lex-i-cal）：与语言的单词或词汇有关，但有别于语言的文法和结构。

韦氏词典

为了将一个程序从一种语言翻译成另一种语言，编译器必须首先把程序的各种成分拆开，并搞清其结构和含义，然后再用另一种方式把这些成分组合起来。编译器的前端执行分析，后端进行合成。

分析一般分为以下三种。

- **词法分析：**将输入分解成一个个独立的词法符号，即“标记符号”（token），简称标记。
- **语法分析：**分析程序的短语结构。
- **语义分析：**推算程序的含义。

词法分析器以字符流作为输入，生成一系列的名字、关键字和标点符号，同时抛弃标记之间的空白符和注释。程序中每个地方都有可能出现空白符和注释，如果让语法分析器来处理它们就会使得语法分析过于复杂，这便是将词法分析从语法分析中分离出去的主要原因。

词法分析并不太复杂，但是我们却使用能力强大的形式化方法和工具来实现它，因为类似的形式化方法对语法分析研究很有帮助，并且类似的工具还可以应用于编译器以外的其他领域。

2.1 词法标记

词法标记是字符组成的序列，可以将其看作程序设计语言的文法单位。程序设计语言的词法标记可以归类为有限的几组标记类型。例如，典型程序设计语言的一些标记类型为：

类型	例子
ID	foo n14 last
NUM	73 0 00 515 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

IF、VOID、RETURN 等由字母字符组成的标记成为保留字（reserved word），在多数语言中，它们不能作为标识符使用。

不是标记的例子有：

注释	/* try again */
预处理命令	#include <stdio.h>
预处理命令	#define NUMS 5 , 6
宏	NUMS
空格符、制表符和换行符	

在能力较弱而需要宏预处理器的语言中，由预处理器处理源程序的字符流，并生成另外的字符流，然后由词法分析器读入这个新产生的字符流。这种宏处理过程也可以与词法分析集成到一起。

对于下面一段程序：

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
  return 0.;
}
```

词法分析器将返回下列标记流：

```
FLOAT    ID(match0)    LPAREN    CHAR    STAR    ID(s)    RPAREN
LBRACE    IF  LPAREN    BANG    ID(strncmp)    LPAREN    ID(s)
COMMA    STRING(0.0)    COMMA    NUM(3)    RPAREN    RPAREN
RETURN    REAL(0.0)    SEMI  RBRACE    EOF
```

其中报告了每个标记的标记类型。这些标记中的一些（如标识符和字面量）有语义值与之相连，因此，词法分析器还给出了除标记类型之外的附加信息。

应当如何描述程序设计语言的词法规则？词法分析器又应当用什么样的语言来编写呢？

我们可以用自然语言来描述一种语言的词法标记。例如，下面是对 C 或 Java 中标识符的一种描述：

> 标识符是字母和数字组成的序列，第一个字符必须是字母。下划线“_”视为字母。大小写字母不同。如果经过若干标记分析后输入流已到达一个给定的字符，则下一个标记将由有可能组成一个标记的最长字符串所组成。其中的空格符、制表符、换行符和注释都将被忽略，除非它们作为独立的一类标记。另外需要有某种空白符来分隔相邻的标识符、关键字和常数。

任何合理的程序设计语言都可以用来实现特定的词法分析器。我们将用正则表达式的形式语言来指明词法标记，用确定的有限自动机来实现词法分析器，并用数学的方法将两者联系起来。这样将得到一个简单且可读性更好的词法分析器。

2.2 正则表达式

我们说语言（language）是字符串组成的集合，字符串是符号（symbol）的有限序列。符号本身来自有限字母表（alphabet）。

Pascal 语言是所有组成合法 Pascal 程序的字符串的集合，素数语言是构成素数的所有十进制数字字符串的集合，C 语言保留字是 C 程序设计语言中不能作为标识符使用的所有字母数字字符串组成的集合。这 3 种语言中，前两种是无限集合，后一种是有限集合。在这 3 种语言中，字母表都是 ASCII 字符集。

以这种方式谈论语言时，我们并没有给其中的字符串赋予任何含义，而只是企图确定每个字符串是否属于其语言。

为了用有限的描述来指明这类（很可能是无限的）语言，我们将使用正则表达式（regular expression）表示法。每个正则表达式代表一个字符串集合。

- **符号**（symbol）：对于语言字母表中的每个符号 **a**，正则表达式 **a** 表示仅包含字符串 a 的语言。
- **或**（alternation）：对于给定的两个正则表达式 **M** 和 **N**，或运算符（|）形成一个新的正则表达式 **M | N**。如果一个字符串属于语言 **M** 或者语言 **N**，则它属于语言 **M | N**。因此，**a | b** 组成的语言包含 a 和 b 这两个字符串。
- **联结**（concatenation）：对于给定的两个正则表达式 **M** 和 **N**，联结运算符（·）形成一个新的正则表达式 **M** · **N**。如果一个字符串是任意两个字符串 α 和 β 的联结，且 α 属于语言 **M**，β 属于语言 **N**，则该字符串属于 **M** · **N** 组成的语言。因此，正则表达式 (**a**|**b**) · **a** 定义了包含两个字符串 aa 和 ba 的语言。
- ϵ（epsilon）：正则表达式 ϵ 表示仅含一个空字符串的语言。因此，(**a** · **b**)|ϵ 表示语言 {"", "ab"}。
- **重复**（repetition）：对于给定的正则表达式 **M**，它的克林闭包（Kleene closure）是 **M***。如果一个字符串是由 M 中的字符串经零至多次联结运算的结果，则该字符串属于 **M***。因此，((**a**|**b**) · **a**)* 表示无穷集合 {"", "aa", "ba", "aaaa", "baaa", "aaba", "baba", "aaaaaa", · · · }。

通过使用符号、或、联结、ϵ 和克林闭包，我们可以规定与程序设计语言词法单词相对应的 ASCII 字符集。首先，考虑若干例子：

(**0**|**1**)* · **0**	由 2 的倍数组成的二进制数。
b*(**abb***)*(**a**|ϵ)	由 a 和 b 组成，但 a 不连续出现的字符串。
(**a**|**b**)***aa**(**a**|**b**)*	由 a 和 b 组成，且有连续出现 a 的字符串。

在书写正则表达式时，我们有时会省略联结运算符或 ϵ 符号，并假定克林闭包的优先级高于联结运算，联结运算的优先级高于或运算，所以 **ab**|**c** 表示 (**a** · **b**)|**c**，而 (**a**|) 表示 (**a**|ϵ)。

还有一些更为简洁的缩写形式：**[abcd]** 表示 (**a**|**b**|**c**|**d**)，**[b-g]** 表示 **[bcdefg]**，**[b-gM-Qkr]** 表示 **[bcdefgMNOPQkr]**，**M**? 表示 (**M**|ϵ)，**M**$^+$ 表示 **M** · **M***。这些扩充很方便，但

它们并没有扩充正则表达式的描述能力：任何可以用这些简写形式描述的字符串集合都可以用基本运算符集合来描述。图2.1概括了所有这些运算符。

a	一个表示字符本身的原始字符
ϵ	空字符串
	空字符串的另一种写法
M | **N**	或运算符，在 **M** 和 **N** 之间选择
M · **N**	联结，**M** 之后跟随 **N**
MN	联结的另一种写法
$\mathbf{M}^*$	重复（0 次或 0 次以上）
$\mathbf{M}^+$	重复（1 次或 1 次以上）
M?	选择，**M** 的 0 次或 1 次的出现
[a-zA-Z]	字符集
.	句点表示除换行符之外的任意单个字符
" a. + * "	引号，引号中的字符串表示文字字符串本身

图 2.1: 正则表达式表示符号

使用这种语言，我们便可以指明程序设计语言的词法标记（见图2.2）。对于每一个标记，我们提供一段 ML 代码，报告识别的是哪种标记类型。

```
if                                  (IF);
[a-z][a-z0-9]*                      (ID);
[0-9]+                              (NUM);
([0-9]+"."[0-9]*) | ([0-9]*"."[0-9]+)  (REAL);
("–"[a-z]*"\n") | (" " | "\n" | "\t")+  (continue());
.                                   (error(); continue());
```

图 2.2: 某些标记的正则表达式

图2.2第 5 行的描述虽然识别注释或空白，但是不提交给语法分析器，而是忽略它们并重新开始词法分析。这个分析器识别的注释以两个短横线开始，且只包含字母字符，并以换行符结束。

最后，词法规范应当是完整的，它应当总是能与输入中的某些初始子串相匹配；使用一个可以与任意字符相匹配的规则，我们便总能做到这一点（在这种情况下，将打印出“illegal character”错误信息，然后再继续进行）。

图2.2中的规则存在着二义性。例如，对于 if8，应当将它看成一个标识符，还是两个标记 if 和 8？字符串 if 89 是以一个标识符开头还是以一个保留字开头？Lex，ML-Lex 以及其他类似的词法分析器使用了两条消除二义性的重要规则。

- **最长匹配**：初始输入子串中，取可与任何正则表达式匹配的那个最长的字符串作为下一个标记。
- **规则优先**：对于一个特定的最长初始子串，第一个与之匹配的正则表达式决定了这个子串的标记类型。也就是说，正则表达式规则的书写顺序有意义。

因此，依据最长匹配规则，if8 是一个标识符；根据规则优先，if 是一个保留字。

2.3 有限自动机

用正则表达式可以很方便地指明词法标记，但我们还需要一种用计算机程序来实现的形式化方法。可以使用有限自动机达到此目的。有限自动机有一个有限状态集合和一些从一个状态通向另一个状态的边，每条边上标记有一个符号；其中一个状态是初态，某些状态是终态。

图2.1给出了一些有限自动机的例子。为了方便讨论，我们给每个状态一个编号。每个例子中的初态都是编号为 1 的状态。标有多个字符的边是多条平行边的缩写形式；因此，在机器 ID 中，实际上有 26 条边从状态 1 通向状态 2，每条边用不同的字母标记。

在确定的有限自动机（DFA）中，不会有从同一状态出发的两条边标记为相同的符号。DFA 以如下方式接收或拒绝一个字符串：从初始状态出发，对于输入字符串中的每个字符，自动机都将沿着一条确定的边到达另一状态，这条边必须是标有输入字符的边。对 n 个字符的字符串进行了 n 次状态转换后，如果自动机到达了终态，自动机将接收该字符串。若到达的不是终态，或者找不到与输入字符相匹配的边，那么自动机将拒绝接收这个字符串。由一个自动机识别的语言是该自动机接收的字符串集合。

例如，显然，在由自动机 ID 识别的语言中，任何字符串都必须以字母开头。任何单字母都能通至状态 2，因此单字母字符串是可被接收的字符串。从状态 2 出发，任何字母和数字都将重新回到状态 2，因此一个后跟任意个数字母和数字的字母也将被接收。

事实上，图2.1所示的自动机接收的语言与图2.2给出的正则表达式相同。

图2.1中是 6 个独立的自动机，如何将它们合并为一个可作为词法分析器的自动机呢？我们将在下一章学习合并它们的形式化方法；在这里只给出合并它们后得到的机器，如图2.2所示。机器中的每个终态都必须标明它所接收的标记类型。在这个自动机中，状态 2 是自动机 IF 的状态 2 和自动机 ID 的状态 2 的合并；由于状态 2 是自动机 ID 的终态，因此这个合并的状态也必须是终态。状态 3 与自动机 IF 的状态 3 和自动机 ID 的状态 2 相同，因为这两者都是终态，故我们使用消除二义性的规则优先原则将状态 3 的接收标记类型标为 IF。之所以使用规则优先原则是因为我们希望这一标记被识别为保留字，而不是标识符。

这个自动机可用一个转换矩阵来表示。转换矩阵是一个二维数组（一个元素为向量的向量），数组的下标是状态编号和输入字符。其中有一个停滞状态（状态 0），这个状态对于任何输入字符都返回到自身，我们用它来表示不存在的边。

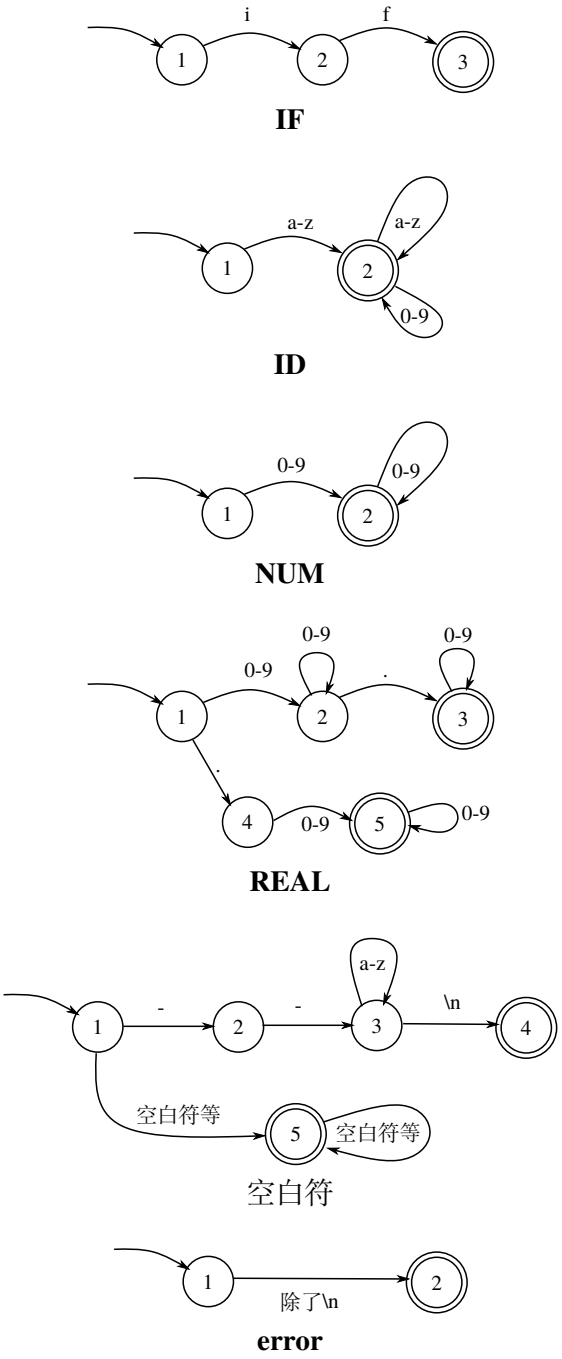


图 2.1: 词法标记的有限自动机。圆圈表示状态，双圆圈表示终态。初态是进入边没有来源的状态。标有多个字符的边是多条平行边的缩写

```
val edges =
 vector[              (* ···0 1 2···-···e f g h i j···*)
(* state 0 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 1 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 2 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 3 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 4 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 5 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 6 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 7 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
  et cetera
      ]
```

另外还需要有一个“终结”（finality）数组，它的作用是将状态编号映射至动作。例如，终态 2 映射到动作 ID，等等。

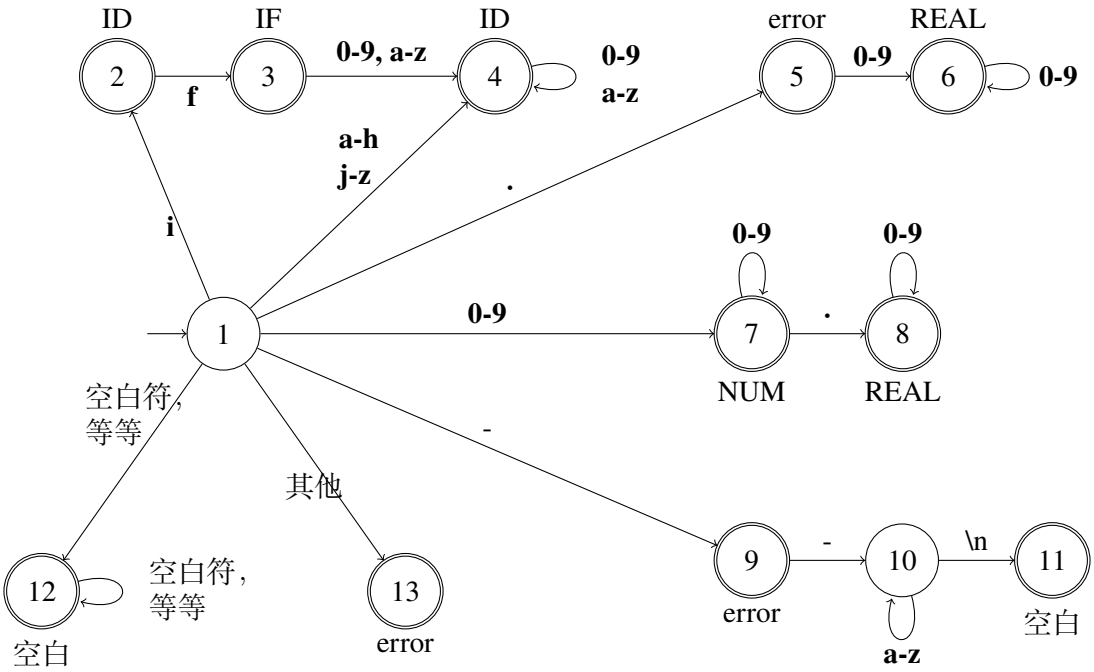


图 2.2: 合并后的有限自动机

识别最长的匹配

很容易看出如何使用转换矩阵来识别一个字符串是否会被接收，但是词法分析器的任务是要找到最长的匹配，因为输入中最长的初始子串才是合法的标记。在进行转换的过程中，词法分析器要一直追踪迄今见到的最长匹配以及这个最长匹配的位置。

追踪最长匹配意味着需要用变量 Last-Final（最近遇到的终态的编号）和 Input-Position-at-Last-Final 来记住自动机最后一次处于终态时的时机。每次进入一个终态时，词法分析

器都要更新这两个变量，当到达停滞状态（无出口转换的非终态状态）时，从这两个变量便能得知所匹配的标记和它的结束位置。

图2.3说明了词法分析器识别最长匹配的操作过程。注意，当前输入位置可能相距识别器最近到达终态时的位置已很远。

最后的终态	当前状态	当前输入	接收动作
0	1	|if --not-a-com	
2	2	i|f --not-a-com	
3	3	if| --not-a-com	
3	0	if| --not-a-com	返回 IF
0	1	if| --not-a-com	
12	12	if| -not-a-com	
12	0	if| not-a-com	找到空白；重新开始
0	1	if |-not-a-com	
9	9	if -|not-a-com	
9	10	if -n|ot-a-com	
9	10	if -no|t-a-com	
9	10	if -not|-a-com	
9	10	if -not-|a-com	
9	0	if -not-a|-com	错误；非法标记 “-”；重新开始
0	1	if -|not-a-com	
9	9	if --|not-a-com	
9	0	if --n|ot-a-com	错误；非法标记 “-”；重新开始

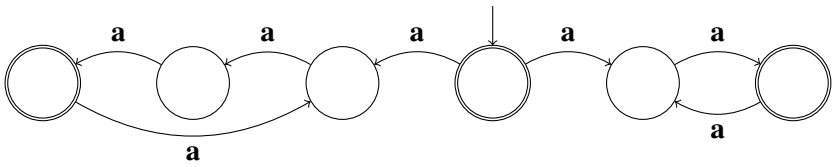
图 2.3: 图2.2中自动机识别的几个单词。符号 “|” 指出每次调用词法分析器时的输入位置，符号 “” 指出自动机的当前位置，符号 “” 指出自动机最近一次处于终态时的位置

2.4 非确定性有限自动机

非确定性有限自动机（NFA）是一种需要对从一个状态出发的多条标有相同符号的边进行选择的自动机。它也可能存有标有 ϵ（希腊字母）的边，这种边可以在不接收输入字符的情况下进行状态转换。

下面是一个 NFA 的例子：

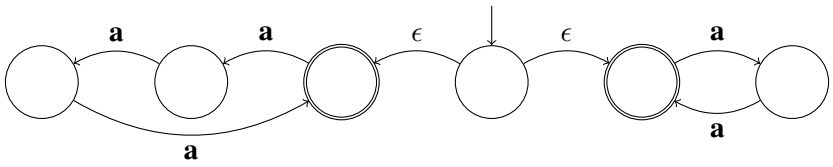
在初态时，根据输入字母 a，自动机既可向左转换，也可向右转换。若选择了向左转换，则接收的是长度为 3 的倍数的字符串；若选择了向右转换，则接收的是长度为偶数



的字符串。因此，这个 NFA 识别的语言是长度为 2 的倍数或 3 的倍数的所有由字母 a 组成的字符串的集合。

在第一次转换时，这个自动机必须选择走哪条路。如果存在着任何导致该字符串被接收的可选择路径，那么自动机就必须接收该字符串。因此，自动机必须进行“猜测”，并且必须总是作出正确的猜测。

标有 ϵ 的边可以不使用输入中的字符。下面是接收同样语言的另一个 NFA：

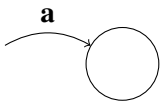


同样地，这个自动机必须决定选取哪一条 ϵ 边。若存在一个状态既有一些 ϵ 边，又有一些标有符号的边，则自动机可以选择接收一个输入符号（并沿着标有对应符号的边前进），或者选择沿着 ϵ 边前进。

2.4.1 将正则表达式转换为 NFA

非确定性的自动机是一个很有用的概念，因为它很容易将一个（静态的、说明性的）正则表达式转换成一个（可模拟的、准可执行的）NFA。

转换算法可以将任何一个正则表达式转换为有一个尾巴和一个脑袋的 NFA。它的尾巴即开始边，简称为*尾*；脑袋即末端状态，简称为*头*。例如，单个符号的正则表达式 **a** 转换成的 NFA 为：

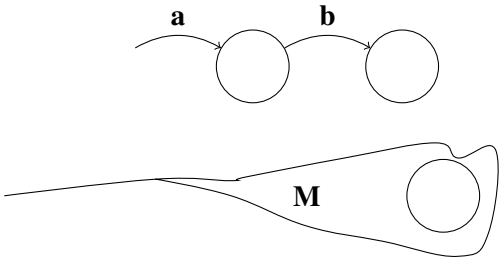


由 **a** 和 **b** 经联结运算而形成的正则表达式 **ab** 对应的 NFA 是由两个 NFA 组合而成的，即将 **a** 的头与 **b** 的尾连接起来。由此得到的自动机有一个用 **a** 标记的尾和一个从 **b** 边进入的头。

一般而言，任何一个正则表达式 **M** 都有一个具有尾和头的 NFA：

我们可以归纳地定义正则表达式到 NFA 的转换。一个正则表达式或者是原语（单个符号或 ϵ），或者是由多个较小的表达式组合而成。

图2.4展示了将正则表达式转换至 NFA 的规则。我们用图2.2中关于单词 IF、ID、NUM 以及 error 的一些表达式来举例说明这种转换算法。每个表达式都转换成了一个 NFA，每



个 NFA 的头是用不同标记类型标记的终态结点，并且每一个表达式的尾汇合成一个新的初始结点。由此得到的结果（在合并了某些等价的 NFA 状态之后）如图2.5所示。

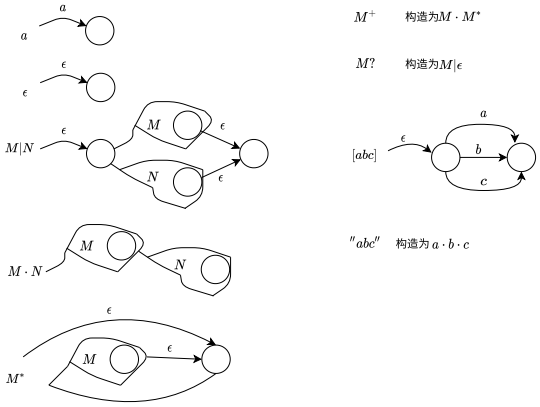


图 2.4: 正则表达式至 NFA 的转换

2.4.2 将 NFA 转换为 DFA

如在 2.3 节看到的，用计算机程序实现确定的有限自动机（DFA）较容易。但实现 NFA 则要困难一些，因为大多数计算机都没有足够好的可以进行“猜测”的硬件。

通过一次同时尝试所有可能的路径，可以避免这种猜测。我们用字符串 in 来模拟图2.5的 NFA。首先从状态 1 开始。现在，替代猜测应采用哪个 ϵ 转换，我们只是说此时 NFA 可能选择它们中的任何一个，因此，它是状态 {1,4,9,14} 当中的任何一个，即我们需要计算 {1} 的 ϵ 闭包。显然，不接收输入中的第一个字符，就不可能到达其他状态。

现在要根据字符 i 来进行转换。从状态 1 可以到达状态 2，从状态 4 可以到达状态 5，从状态 9 则无处可去，而从状态 14 则可以到达状态 15，由此得到状态集合 {2,5,15}。但是，我们还必须计算 ϵ 闭包：从状态 5 有一个 ϵ 转换至状态 8，从状态 8 有一个 ϵ 转换至状态 6。因此这个 NFA 一定属于状态集合 {2,5,6,8,15}。

对于下一个输入字符 n，我们从状态 6 可到达状态 7，但状态 2、5、8 和 15 都无相应的转换。因此得到状态集合 {7}，它的 ϵ 闭包是 {6,7,8}。

现在我们已经到达了字符串 in 的末尾，那么，这个 NFA 是否已经到达了终态呢？在我们得到的可能状态集合中，状态 8 是终态，因此 in 是一个 ID 标记。

我们形式化地定义 ϵ 闭包如下。令 **edge**(s, c) 是从状态 s 沿着标有 c 的一条边可以到达的所有 NFA 状态的集合。对于状态集合 S，**closure**(S) 是从 S 中的状态出发，无需接收任何字符，即只通过 ϵ 边便可以到达的状态组成的集合。这种经过 ϵ 边的概念可用数

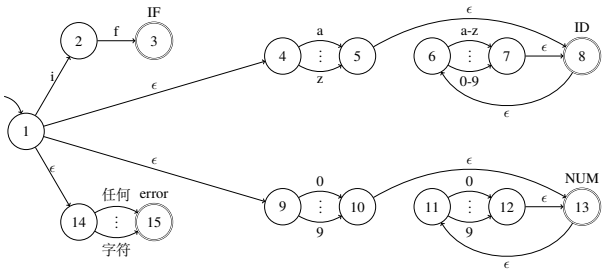


图 2.5: 由 4 个正则表达式转换成的一个 NFA

学方式表述，即 **closure**(S) 是满足如下条件的最小集合 T：

$$T = S \cup (\bigcup_{s \in T} \textbf{edge}(s, \epsilon))$$

我们可以用迭代法来算出 T：

$$\begin{aligned}
&T \leftarrow S \\
&\textbf{repeat } T' \leftarrow T \\
&\qquad T \leftarrow T' \cup (\bigcup_{s \in T'} \textbf{edge}(s, \epsilon)) \\
&\textbf{until } T = T'
\end{aligned}$$

这个算法为什么是正确的？因为 T 只可能在迭代中扩大，所以最终的 T 一定包含 S。如果在一次迭代之后有 $T = T'$，则 T 也一定包含 $\bigcup_{s \in T'} \textbf{edge}(s, \epsilon)$。因为在 NFA 中只有有限个不同的状态，所以算法一定会终止。

现在，当用前面描述的方法来模拟一个 NFA 时，假设我们位于由 NFA 状态 s_i、s_k、s_l 组成的集合 $d = \{s_i, s_k, s_l\}$ 中。从 d 中的状态出发，并吃进输入符号 c，将到达 NFA 的一个新的状态集合；我们称这个集合为 **DFAedge**(d, c)：

$$\textbf{DFAedge}(d, c) = \textbf{closure}(\bigcup_{s \in d} \textbf{edge}(s, c))$$

利用 **DFAedge** 能够更加形式化地写出 NFA 模拟算法。如果 NFA 的初态是 s_1，输入字符串中的字符是 $c_1, \cdots, c_k$，则算法为：

$$\begin{aligned}
&d \leftarrow \textbf{closure}(\{s_1\}) \\
&\textbf{for } i \leftarrow 1 \textbf{ to } k \\
&\qquad d \leftarrow \textbf{DFAedge}(d, c_i)
\end{aligned}$$

状态集合运算是代价很高的运算——对进行词法分析的源程序中的每一个字符都做这种运算几乎是不现实的。但是，预先计算出所有的状态集合却是有可能的。我们可以由 NFA 构造一个 DFA，使得 NFA 的每一个状态集合都对应于 DFA 的一个状态。因为 NFA 的状态个数有限（n 个），所以这个 DFA 的状态个数也是有限的（至多为 2^n 个）。

一旦有了 **closure** 和 **DFAedge** 的算法，就很容易构造出 DFA。DFA 的状态 d_1 就是 **closure**(s_1)，这同 NFA 模拟算法一样。抽象而言，如果 $d_j =$ **DFAedge**(d_i, c)，则存在着一条从 d_i 到 d_j 的标记为 c 的边。令 Σ 是字母表。

$states[0] \leftarrow \{\}$;　　$states[1] \leftarrow$ **closure**$(\{s_1\})$
$p \leftarrow 1$;　　$j \leftarrow 0$
while $j \leq p$
　foreach $c \in \Sigma$
　　$e \leftarrow$ **DFAedge**$(states[j], c)$
　　if $e = states[i]$ *for some* $i \leq p$
　　then $trans[j, c] \leftarrow i$
　　else $p \leftarrow p + 1$
　　$states[p] \leftarrow e$
　　$trans[j, c] \leftarrow p$
　$j \leftarrow j + 1$

这个算法不访问 DFA 的不可到达状态。这一点特别重要，因为原则上 DFA 有 2^n 个状态，但实际上一般只能找到约 n 个状态是从初始状态可以到达的。这一点对避免 DFA 解释器的转换表出现指数级的膨胀很重要，因为这个转换表是编译器的一部分。

只要 $states[d]$ 中有任何状态是其 NFA 中的接受状态，状态 d 就是 DFA 的*接受状态*。仅仅标志一个状态为*接受状态*是不够的，我们还必须告知它识别的是什么标记，并且 $states[d]$ 中还可能有多个状态是这个 NFA 的接受状态。在这种情况下，我们用一个适当的标记类型来标识 d，这个适当的标记类型即组成词法规则的正则表达式中最先出现的那个标记类型。这就是*规则优先*的实现方法。

构造了 DFA 之后便可以删除“状态”数组，只保留“转换”数组用于词法分析。

对图2.5的 NFA 应用这个 DFA 构造算法得到了图2.6的自动机。

这个自动机还不是最理想的，也就是说，它不是识别相同语言的最小自动机。一般而言，我们称两个状态 s_1 和 s_2 是等价的，如果开始于 s_1 的机器接收字符串 σ，则它从状态 s_2 开始也一定接收 σ，反之亦然。图2.6中，标为 [5,6,8,15] 的状态和标为 [6,7,8] 的状态等价。标为 [10,11,13,15] 的状态与标为 [11,12,13] 的状态等价。若自动机存在两个等价状态 s_1 和 s_2，则我们可以使得所有进入 s_2 的边都指向 s_1 而删除 s_2。

那么如何才能找出所有等价的状态呢？若 s_1 和 s_2 同为接受状态或同为非接受状态，且对于任意符号 c，$trans[s_1, c] = trans[s_2, c]$，则显然它们两者等价。容易看出 [10,11,13,15] 和 [11,12,13] 满足这个判别条件。但是这个条件的普遍性还不够充分，考虑下面的自动机：

其中状态 2 和 4 等价，但是 $trans[2, a] \neq trans[4, a]$。

在构造出一个 DFA 后，用一个算法来找出它的等价状态，并将之最小化是很有好处

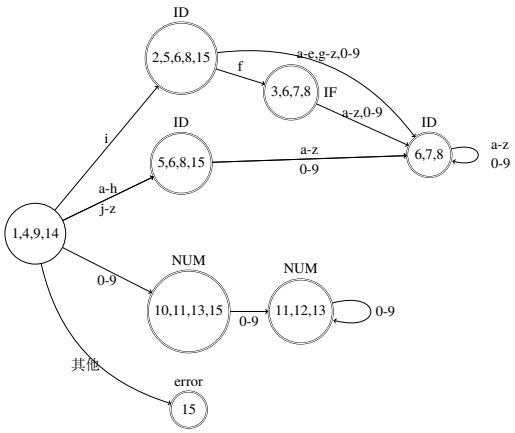
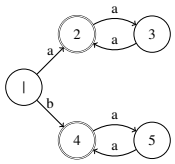


图 2.6: NFA 被转化为 DFA



的；见习题 2.6。

2.5 ML-Lex：词法分析器的生成器

构造 DFA 是一种机械性的工作，很容易由计算机来实现，因此一种有意义的做法是，用*词法分析器的自动生成器*来将正则表达式转换为 DFA。

ML-Lex 就是这样的一个词法分析器的生成器，它由*词法规范*生成一个 ML 程序。对于要进行分析的程序设计语言中的每一种标记类型，该规范包含一个正则表达式和一个*动作*。这个动作将标记类型（可能和其他信息一起）传给编译器的下一个处理阶段。

ML-Lex 的输出是一个 ML 程序，即一个词法分析器。该分析器使用 2.3 节介绍的算法来解释 DFA，并根据每一种匹配执行一段动作代码，这段动作代码是用于返回标记类型的 ML 语句。

图2.2描述的标记类型在 ML-Lex 中的规范如程序2.1所示。

程序 2.1: 图2.2描述的标记的 ML-Lex 规范

```
(* ML Declarations: *)
  type lexresult = Tokens.token
  fun eof() = Tokens.EOF(0,0)
  %%
(* Lex Definitions: *)
  digits = [0-9]+
  %%
(* Regular Expressions and Actions *)
  if                    => (Tokens.IF(yypos,yypos+2));
```

```
[a-z][a-z0-9]*        => (Tokens.ID(yytext,yypos,yypos+size yytext));
{digits}              => (Tokens.NUM(Int.fromString yytext,
                                     yypos,yypos+size yytext));
({digits}"."[0-9]*)|([0-9]*"."{digits})
                      => (Tokens.REAL(Real.fromString yytext,
                                      yypos,yypos+size yytext));
("--"[a-z]*"\n")|(" ""\n""\t")+
                      => (continue());
                      => (ErrorMsg.error yypos "illegal character";
                          continue());
```

该规范的第一部分，即位于第一个“%%”标志上面的部分，包含了 ML 编写的函数和类型。在这一部分里面，必须包含 lexresult 类型，这个类型是每次调用词法分析函数所产生的结果的类型；而 eof 函数，将会在词法分析引擎碰到文件结尾时被调用。这一部分也可以包含一些功能函数，作为第三部分的语义动作使用。

这个规范的第二部分包含正则表达式的简写形式和状态说明。例如，在这一部分中的说明 digits[0-9]+ 允许用名字 {digits} 代表正则表达式中非空的数字序列。

第三部分包含正则表达式和动作。这些动作是一段原始的 ML 代码。每一个动作必须返回一个 lexresult 类型的值。在这个约定里面，lexresult 是 Tokens 结构中的一个标记。

动作代码中可以使用几个特殊的变量。由正则表达式匹配的字符串是 yytext。匹配到的字符串的开始位置在文件中的位置是 yypos。函数 continue() 递归地调用词法分析器。

在这个特定的例子中，每种标记都是一个数据构造器，构造器接受两个整型参数来表示位置——输入文件中的位置——标记开始的位置和结束的位置。

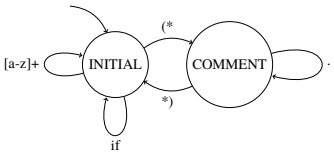
```
structure Tokens =
struct
   type pos = int
   datatype token = EOF of pos * pos
                  | IF of pos * pos
                  | ID of string * pos * pos
                  | NUM of int * pos * pos
                  | REAL of real * pos * pos
                  ⋮
end
```

所以，我们将 yypos 和 yypos + size(yytext) 这两个值作为参数传递给构造器就可以了。有一些标记关联有语义值。例如，ID 的语义值是组成标识符的字符串，NUM 的语义值是一个整数，而 IF 则没有语义值（因为每一个 IF 都有别于其他标记）。所以，ID 构造器和 NUM 构造器都有一个额外的参数来作为语义值，而这个语义值可以从 yytext 变量计算出来。

初始状态

正则表达式是静态的和说明性的，自动机是动态的和命令式的；也就是说，你不必用一个算法来模拟就能看到正则表达式的成分和结构，但是理解自动机常常需要你在自己的头脑中来“执行”它。因此，正则表达式一般更适合于用来知名程序设计语言标记的词法结构。

有时候一步一步地模拟自动机的状态转换过程也是一种合适的做法。ML-Lex 有一种将状态和正则表达式混合到一起的机制。你可以声明一组*初始状态*，每个正则表达式的前面可以有一组对它而言是合法的初始状态作为其前缀。动作代码可以明显地改变初态。这相当于我们有这样的一种有限自动机，其边标记的不是符号而是正则表达式。下面的例子给出了一种只由简单标识符、单词 if 和以“(* ”和“*)”作为界定符的注释所组成的语言。



尽管有可能写出与整个注释相匹配的单个正则表达式，但是随着注释变得越来越复杂，特别是在允许注释嵌套的情况下，这种正则表达式也会越来越复杂，甚至变得不可能。

与这个机器对应的 ML-Lex 的规范为：

```
        规范一般以如下代码开头...
%%
%s COMMENT
%%
<INITIAL>if          => (Tokens.IF(yypos,yypos+2));
<INITIAL>[a-z]+      => (Tokens.ID(yytext,yypos,
                                   yypos+size(yytext)));
<INITIAL>"(*"        => (YYBEGIN COMMENT; continue());
<COMMENT>"*)"        => (YYBEGIN INITIAL; continue());
<COMMENT>.           => (continue();)
```

其中 INITIAL 是“任何注释之外”的状态。最后一个规则是一种调整，其用途是使得 ML-Lex 进入此状态。任何不以 <STATE> 为前缀的正则表达式在所有状态中都能工作，这种特征很少有用处。

利用一个全局变量，并在语义动作中适当增减此全局变量的值，这个例子便很容易扩充成可以处理嵌套的注释。

2.6 程序设计：词法分析

用 ML-Lex 写出一个 Tiger 语言的词法分析器。附录中描述了 Tiger 的词法标记。

本章未对词法分析器应当如何初始化以及它应当如何与编译器的其他部分通信作出说明。你可以从 ML-Lex 使用手册中得到这些内容，而在 $TIGER/chap2 目录中有一个最基本的“脚手架”文件可以帮助你入门。

你应当在连同 tiger.lex 文件一起提交的文档中描述清楚以下问题：

- 你是怎样处理注释的。
- 你是怎样处理字符串的。
- 错误处理。
- 文本结束处理。
- 你的词法分析器的其他令人感兴趣的特征。

在 $TIGER/chap2 中有如下一些可用的支持文件。

- tokens.sig，Tokens 结构的签名。
- tokens.sml，Tokens 结构，包含 token 类型和构造器，你的词法分析器可以用它们来创建 token 类型的实例。以这种方式来完成词法分析非常重要，因为当把“真正的”语法分析器链接到这个词法分析器的后面，以及使用“真正的”Tokens 结构时，所有的代码仍然可以运行。
- errormsg.sml，ErrorMsg 结构可以用来产生带有文件名和行号的错误信息。
- driver.sml，一个运行你的词法分析器来分析输入文件的测试平台。
- tiger.lex，tiger.lex 文件的初始代码。
- sources.cm，ML 编译管理器的“makefile”。

在阅读附录（Tiger 语言参考手册）时，要特别注意以**标识符**（Identifier）、**注释**（Comment）、**整型字面量**（Integer literal）和**字符串字面量**（String literal）作为标题的段落。

Tiger 语言的保留字是：while、for、to、break、let、in、end、function、var、type、array、if、then、else、do、of、nil。

Tiger 语言使用的符号是：

, : ; ( ) [ ] { } . + - * / = <> < <= > >= & | :=

对于字符串字面量，你的词法分析器返回的字符串值应当包含所有已转换到其含义的转义字符。

没有负整型字面量。对于带负号的整型字面量，例如-32，要返回两个标记。

检测没有闭合的注释（在文件末尾）和没有闭合的字符串。

目录 $TIGER/testcases 中含有几个简单的 Tiger 样例程序。

开始时：首先创建一个目录，并复制 $TIGER/chap2 中的内容到此目录。用 Tiger 语言编写一个小程序保存于文件 test.tig 中。然后，键入 sml 并输入命令 CM.make()；CM（Compilation Manager，编译管理器）将会使得系统运行 ml-lex 命令，如果需要的话，还会编译和链接需要的 ML 源文件。

最后，Parse.parse "test.tig"; 命令将会利用一个测试台对该文件进行词法分析。

2.7 推荐阅读

Lex 是第一个基于正则表达式的词法分析器的生成器 [Lesk 1975]，它现在仍被广泛使用。

将那种还未对它的边进行过 ϵ 转换检查的状态保存在一个队列或栈中，可以更高效地计算闭包 [Aho et al. 1986]。正则表达式可以直接转换成 DFA 而不需经过 NFA[McNaughton and Yamada1960;Aho et al. 1986]。

DFA 转换表可能非常大，而且很稀疏。若用一个二维矩阵（状态 × 符号）来表示这张表则会需要太多的存储空间。在实际中，这个表是经过压缩的。这样做减少了存储空间需求，但却增加了寻找下一状态需要的时间 [Aho et al. 1986]。

词法分析器，无论是自动生成的还是手工书写的，都必须有效地处理其输入。当然，输人可以放在缓冲区中，从而一次可以获取成批的字符，然后词法分析器可以每次处理缓冲区中的一个字符。每次读取字符时，词法分析器都必须检查是否已到达缓冲区的末尾。通过在缓冲区末尾放置一个敏感标记（sentinel），即一个不属于任何标记的字符，词法分析器就有可能只对每个标记进行一次检查，而不是对每个字符都进行检查 [Aho et al. 1986]。Gray[1988] 使用的一种设计可以只需每行检查一次，而不是每个标记检查一次，但它不能适合那种包含行结束字符的标记。Bumbulis 和 Cowan[1993] 的方法只需对 DFA 中的每一次循环检查一次；当 DFA 中存在很长的路径时，这可减少检查的次数（相对每个字符一次）。

自动生成的词法分析器常常受到速度太慢的批评。从原理上而言，有限自动机的操作非常简单，因而应该是高效的，但是通过转换表进行解释增加了开销。Gray[1988] 指出直接将 DFA 转换为可执行代码（将状态作为 case 语句来实现），其速度可以和手工编写的词法分析器一样快。例如，Flex(fast lexical analyzer generator)[Paxson 1995] 的速度就比 Lex 要快许多。

2.8 习题

1. 写出下面每一种标记的正则表达式。
   (a). 字母表 {a,b,c} 上满足后面条件的字符串：首次出现的 a 位于首次出现的 b 之前。
   (b). 字母表 {a,b,c} 上由偶数个 a 组成的字符串。
   (c). 是 4 的倍数的二进制数。
   (d). 大于 101001 的二进制数。
   (e). 字母表 {a,b,c} 上不包含连续子串 baa 的字符串。
   (f). C 语言中非负整常数组成的语言，其中以 0 开头的数是八进制常数，其他数是十进制常数。
   (g). 使得方程 $a^n + b^n = c^n$ 存在着整数解的二进制整数 n。
2. 对于下列描述，试解释为什么不存在对应的正则表达式。

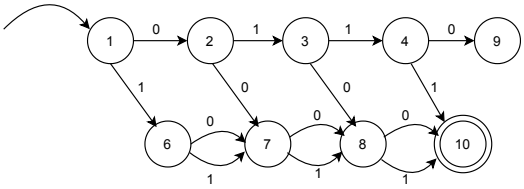
(a). 由 a 和 b 组成的字符串，其中 a 的个数要多于 b。

(b). 由 a 和 b 组成的回文字符串（顺读与倒读相同）。

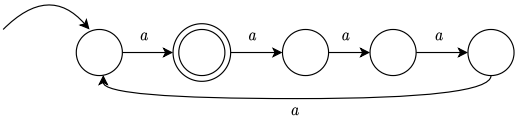
(c). 语法上正确的 C 程序。

3. 用自然语言描述下述有限状态自动机识别的语言。

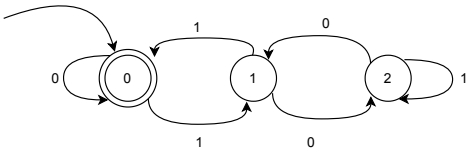
(a).



(b).



(c).



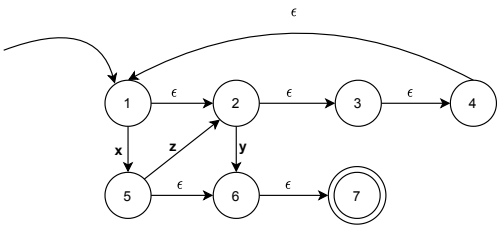
4. 将下面两个正则表达式转换为非确定性有限自动机。

(a). **(if|then|else)**

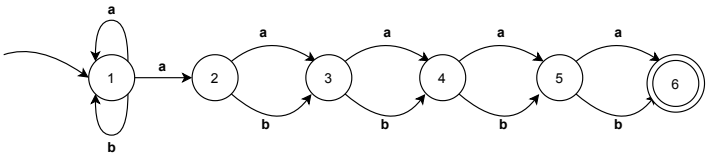
(b). **a((b|a*c)x)*|x*a**

5. 将下面的 NFA 转换为确定性有限自动机。

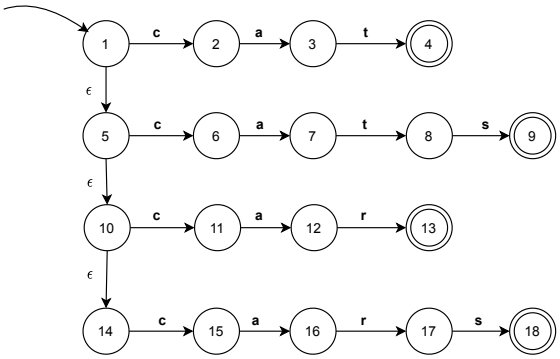
(a).



(b).

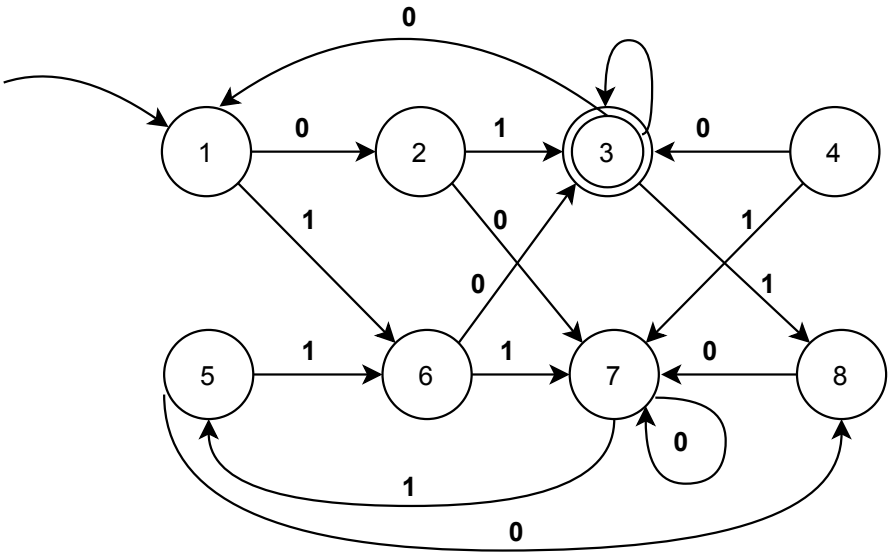


(c).



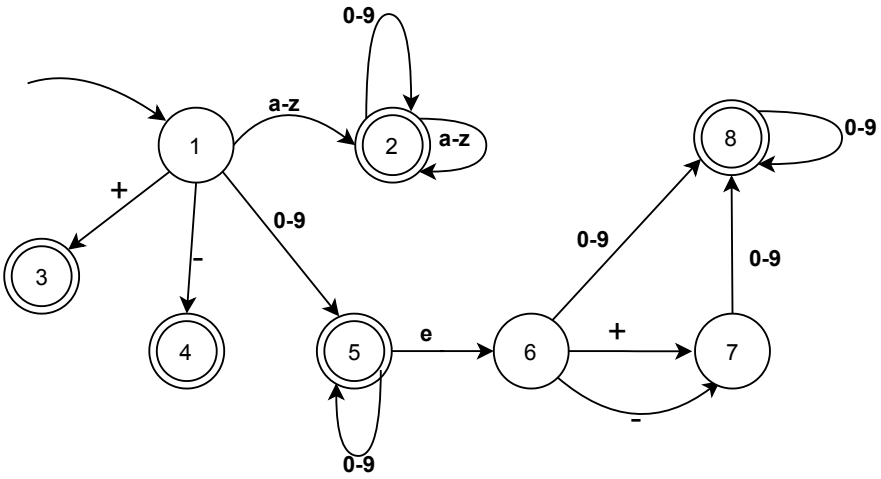
6. 在下面这个自动机中找出两个等价的状态，并合并它们产生一个识别相同语言且较小的自动机。重复这个过程直到没有等价状态为止。

实际上，最小化有限自动机的通用算法是以相反的思路来工作的。它首先要找出的是所有不等价的状态偶对。若 X 是终结符而 Y 不是，或者（通过迭代）$X \to X'$ 且 Y-Y 但 X 和 Y 不等价，则状态 X 和 Y 不等价。用这种迭代方式寻找新的不等价



状态偶对且由于没有更多的不等价状态而停止后，如果 X、Y 仍不是不等价偶对，则它们就是等价状态。参见 Hopcroft 和 Ullman[1979] 中的定理 3.10。

7. 任何接收至少一个字符串的 DFA 都能转换为一个正则表达式。将习题 2.3c 的 DFA 转换为正则表达式。提示：首先假装状态 1 是初态。然后，编写一个通到状态 2 并返回到状态 1 的正则表达式和一个类似的通到状态 0 并返回到状态 1 的正则表达式。或者查看 Hopcroft 和 Ullman[1979] 一书中定理 2.4 关于此算法的论述。
8. 假设 Lex 使用下面这个 DFA 来找输入文件中的单词：



(a). Lex 在匹配一个单词之前，必须在该单词之后再检测多少个字符?

(b). 设你对问题 a 的答案为 k，写出一个至少包含两个单词的输入文件，使得 Lex 的第一次调用在返回第一个单词前需要检测该单词末尾之后的 k 个字符。若对问题 a 的答案为 0，则写出一个包含至少两个单词的输人文件，并指出每个单词的结束点。

9. 一个基于 DFA 的解释型词法分析器使用以下两张表：
   - edges 以状态和输人符号为索引产生一个状态号。

- final 以状态为索引，返回 0 一个动作号。

从下面这个词法规范开始：

```
(aba)+      (action 1);
(a(b*)a)    (action 2);
(a|b)       (action 3);
```

为一个词法分析器生成 edges 和 final 表。

然后给出该词法分析器分析字符串 abaabbaba 的每一步。注意，一定要给出此词法分析器重要的内部变量的值。该词法分析器将被反复调用以获得后继的单词。

10. 词法分析器 Lex 有一个超前查看操作符 “/”，它使得正则表达式 abc/def 只有在 abc 之后跟有 def 时，才能匹配 abc（但是 def 并不是所匹配字符串的一部分，而是下一个或几个单词的一部分)。Aho 等人 [1986] 描述了一种不正确的实现超前查看的算法，并且 Lex[Lesk 1975] 中也使用了这种算法（对于 (alab)/ba，当输入为 aba 时，该算法不能进行正确的识别。它在应当匹配 a 的地方匹配了 ab)。Flex[Paxson1995] 使用了一种更好的机制，这种机制对于 (alab)/ba 能正确工作，但对 zx*/xy* 却不能（但能打印出警告信息)。

请设计出一种更好的超前查看机制。

第三章 语法分析

语法（syn-tax）：组合单词以形成词组、从句或句子的方法。

韦氏词典

ML-Lex 中用一个符号替代某个正则表达式的缩写机制非常方便，这使我们想到用下面的方法来表示一个正则表达式：

$$digits = [0-9]+$$
$$sum = (digits\text{“}+\text{”})^* digits$$

这两个正则表达式定义了形如 28 + 301 + 9 的求和表达式。

但是，考虑下面的定义：

$$digits = [0-9]+$$
$$sum = expr\text{“}+\text{”}expr$$
$$expr = \text{“(”}sum\text{“)”}|digits$$

它们定义的是如下形式的表达式：

(109 + 23)

61

(1 + (250 + 3))

其中的所有括号都是配对的。可是有限自动机却不能识别出这种括号配对的情况（因为一个状态数为 N 的自动机无法记忆嵌套深度大于 N 的括号），因此，sum 和 $expr$ 显然不能是正则表达式。

那么，词法分析器 ML-Lex 怎样实现类似于 $digits$ 这种缩写形式的正则表达式呢？答案是在将正则表达式翻译成有限自动机之前，简单地用 $digits$ 右部的式子（[0-9]+）替代正则表达式中出现的所有 $digits$。

但这种方法对于前面给出的那种 $sum\text{-}expr$ 语言却行不通；我们虽然可以首先将 $expr$ 中的 sum 替换掉，得到：

$$expr = \text{“(”}expr\text{“}+\text{”}expr\text{“)”} \mid digits$$

但是若再用 $expr$ 右部的表达式替换 $expr$ 自身，则得到

$$expr = \text{“(”“(”}expr\text{“}+\text{”}expr\text{“)”}|digits)\text{“}+\text{”}expr\text{“)”} \mid digits$$

右部现在仍然同以前一样出现有 $expr$，且事实上，$expr$ 的出现次数不但没有减少反

而还增加了!

因此，仅仅这种形式的缩写表示并不能增强正则表达式的语言描述能力（它并没有定义额外的语言）。除非这种缩写形式是递归的（或者是相互递归的，如 sum 和 $expr$ 的情形），才能增强正则表达式的语言描述能力。

由这种递归而获得的额外的表达能力正好是语法分析需要的。另外，一旦有了递归的缩写形式，则除了在表达式的顶层之外，可以不再需要可选操作。因为定义

$$expr = ab(c|d)e$$

可通过一个辅助定义重写为:

$$aux = c \mid d$$
$$expr = a\ b\ aux\ e$$

事实上，可以完全不使用可选符号而写出同一个符号的多个可接受的扩展:

$$aux = c$$
$$aux = d$$
$$expr = a\ b\ aux\ e$$

克林闭包也不再是必需的，我们可以将

$$expr = (a\ b\ c)*$$

重写为

$$expr = (a\ b\ c)expr$$
$$expr = \epsilon$$

至此我们得到了一种非常简单的表示法，称为上下文无关文法（context-free grammar）。正如正则表达式以一种静态的、声明式的方式来定义词法结构一样，文法以声明式的方式来定义语法结构。但是我们需要比有限自动机更强大的方法来分析文法所描述的语言。

事实上，文法也可用来描述词法标记的结构[1]；但对于此目的，使用正则表达式要更为适合，也更为简练。

[1]译者注：也就是可以使用上下文无关文法来描述正则表达式语法。

3.1 上下文无关文法

与前面类似，我们认为语言是由字符串组成的集合，每个字符串是由有限字母表中的符号组成的有限序列。对于语法分析而言，字符串是源程序，符号是词法标记（token），字母表是词法分析器返回的标记类型集合。

一个上下文无关文法描述一种语言。文法有如下形式的产生式（production）集合:

$$symbol \to symbol\ symbol \cdots symbol$$

其中，产生式的右部有 0 至多个符号。每一个符号或者是终结符（terminal）——来自该语言字符串字母表中的标记，或者是非终结符（nonterminal）——出现在某个产生式的左部。标记决不会出现在产生式的左部。最后，有一个区别对待的非终结符，称为文法的开始符号（start symbol）。

1	S	$\to$	S ; S
2	S	$\to$	id := E
3	S	$\to$	print (L)
4	E	$\to$	id
5	E	$\to$	num
6	E	$\to$	$E + E$
7	E	$\to$	(S, E)
8	L	$\to$	E
9	L	$\to$	L, E

文法 3.1: 直线式程序的文法

文法3.1是一个直线式程序的文法例子。它的开始符号是 S（当未明确给出开始符号时，约定第一个产生式左部的非终结符为开始符号）。此例中的终结符为:

id print num . + ( ) := ;

非终结符是 S、E 和 L。属于这个文法语言的一个句子为:

id := num; id := id + (id := num + num, id)

与它对应的源程序（在词法分析之前的）可以是:

a := 7;

b := c + (d := 5 + 6, d)

标记（终结符）的类型为 id、num、:= 等。名字（a、b、c、d）和数字（7、5、6）是与其中一些标记关联的语义值（semantic value）。

3.1.1 推导

为了证明这个句子属于该文法的语言，我们可以进行推导（derivation）：从开始符号出发，对其右部的每一个非终结符，都用此非终结符对应的产生式中的任一个右部来替换，如推导3.2所示。

$\underline{S}$
S ; $\underline{S}$
$\underline{S}$; id := E
id := $\underline{E}$; id := E
id := num ; id := $E + \underline{E}$
id := num ; id := $\underline{E} + (S, E)$
id := num ; id := id + $(\underline{S}, E)$
id := num ; id := id + (id := $\underline{E}, E$)
id := num ; id := id + (id := $E + E, \underline{E}$)
id := num ; id := id + (id := $\underline{E} + E$,id)
id := num ; id := id + (id := num + $\underline{E}$,id)
id := num ; id := id + (id := num + num,id)

推导 3.2

同一个句子可以存在多种不同的推导。*最左推导*（leftmost derivation）是一种总是扩展最左边非终结符的推导；在*最右推导*（rightmost derivation）中，下一个要扩展的非终结符总是最右边的非终结符。

推导3.2既不是最左推导，也不是最右推导，因为这个句子的最左推导应当以下述推导开始：

$$
\begin{aligned}
&\underline{S}\\
&\underline{S}\ ;\ S\\
&\text{id} := \underline{E}\ ;\ S\\
&\text{id} := \text{num}\ ;\ \underline{S}\\
&\text{id} := \text{num}\ ;\ \text{id} := \underline{E}\\
&\text{id} := \text{num}\ ;\ \text{id} := \underline{E} + E\\
&\quad\vdots
\end{aligned}
$$

3.1.2 语法分析树

语法分析树（parse tree，也简称为语法树或分析树）是将一个推导中的各个符号连接到从它推导出来的符号而形成的，如图3.3所示。两种不同的推导可以有相同的语法树。

3.1.3 二义性文法

如果一个文法能够推导出具有两棵不同语法树的句子，则该文法有*二义性*（ambiguous）。文法3.1是有二义性的，因为句子 id := id + id + id 有两棵语法分析树（图3.4）。

文法3.5也是有二义性的。图3.6给出了句子 $1-2-3$ 的两棵语法分析树，图3.7则给出了 $1+2*3$ 的两棵语法树。显然，如果我们用这些语法分析树来解释这两个表达式的含义，$1-2-3$ 的两棵语法分析树则有两种不同的含义，分别为 $(1-2)-3=4$ 和 $1-(2-3)=2$。同样，$(1+2)\times 3$ 也不同于 $1+(2\times 3)$。而且编译器正是利用语法分析树来推导语义的。

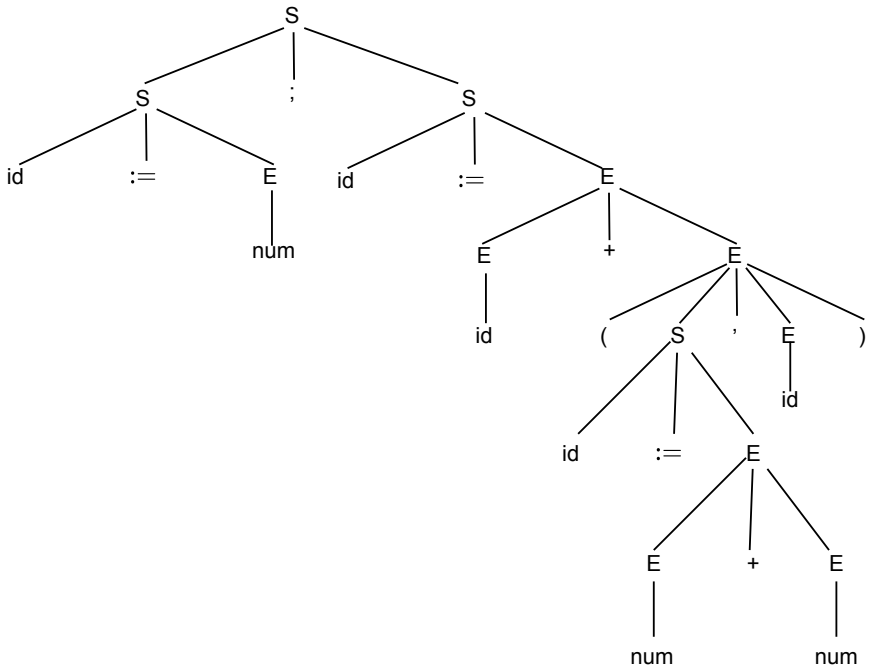


图 3.3: 语法分析树

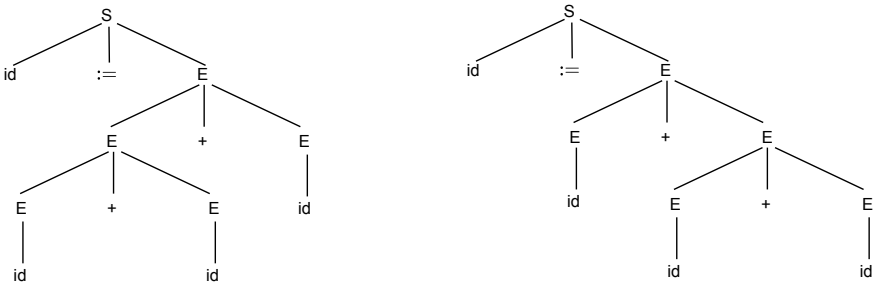


图 3.4: 文法3.1的同一个句子的两颗语法分析树

E	$\rightarrow$	id
E	$\rightarrow$	num
E	$\rightarrow$	$E * E$
E	$\rightarrow$	E/E
E	$\rightarrow$	$E + E$
E	$\rightarrow$	$E - E$
E	$\rightarrow$	(E)

文法 3.5

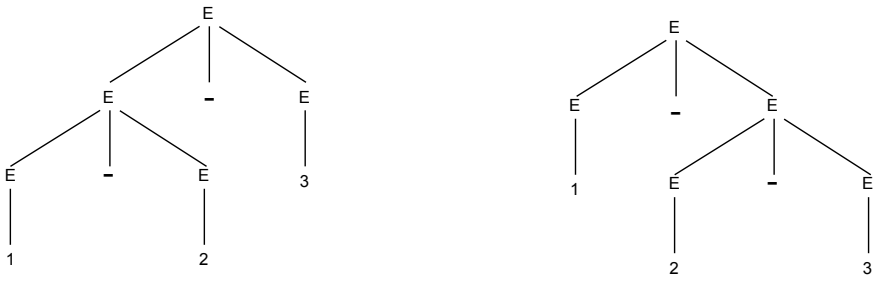


图 3.6: 文法3.5的句子 $1-2-3$ 的两颗语法分析树

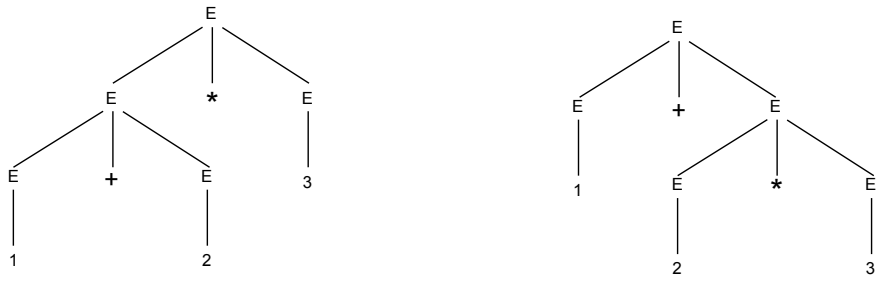


图 3.7: 文法3.5的句子 $1+2*3$ 的两颗语法分析树

因此，二义性文法会给编译带来问题，通常我们希望文法是无二义性的。幸运的是，二义性文法常常可以转换为无二义性的文法。

让我们来找出文法3.5的另一种文法，它接收的语言与文法3.5相同，但却是无二义性的。首先，假定 * 比 + 具有更紧密的约束，或换言之，* 具有较高的优先级。其次，假定每一种操作符都是左结合的，于是我们得到 $(1-2)-3$ 而不是 $1-(2-3)$。通过引入一个新的非终结符得到文法3.8，我们就可达到此目的。

$E \to E+T$	$T \to T*F$	$F \to id$
$E \to E-T$	$T \to T/F$	$F \to num$
$E \to T$	$T \to F$	$F \to (E)$

文法 3.8

文法3.8中，符号 E、T 和 F 分别代表表达式 (expression)、项 (term) 和因子 (factor)。习惯上，因子是可以相乘的语法实体，项是可以相加的语法实体。

这个文法接收的句子集合与原二义性文法接收的相同，但是现在每一个句子都只有一棵语法分析树。文法3.8决不会产生图3.9所示的两棵语法分析树（见习题 3.17）。

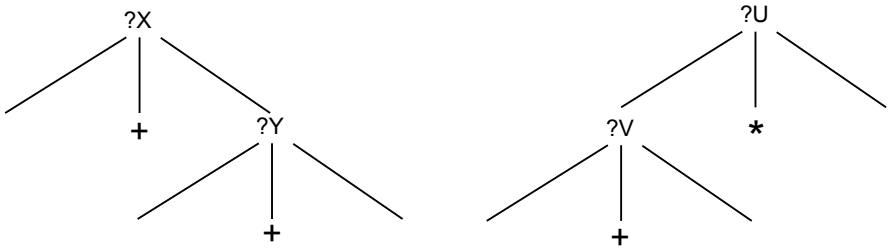


图 3.9: 文法3.8决不会产生的两颗语法分析树

如果我们想让 * 是右结合的，则可将产生式改写为 $T \to F*T$。

我们一般通过文法转换来消除文法的二义性。但是一些语言（即字符串集合）只有有二义性的文法，而没有无二义性的文法。这种语言作为程序设计语言会有问题，因为语法上的二义性会导致程序编写和理解上的问题。

3.1.4 文件结束符

语法分析器读入的不仅仅是 +、-、num 这样的终结符，而且会读入文件结束标志。我们用 $ 符号来表示文件结束。

设 S 是一文法的开始符号。为了指明 \$ 必须出现在一个完整的 S 词组之后，需要引入一个新的开始符号 S' 以及一个新的产生式 $S' \to S\$$。

在文法3.8中，E 是开始符号，修改后的文法为文法3.10。

$S \to E\$$		
	$T \to T * F$	$F \to \text{id}$
$E \to E + T$	$T \to T/F$	$F \to \text{num}$
$E \to E - T$	$T \to F$	$F \to (E)$
$E \to T$		

文法 3.10

3.2 预测分析

有一些文法使用一种称为*递归下降*（recursive descent）的简单算法就很容易进行分析。这种算法的实质是将每一个文法产生式转变成递归函数中的一个子句。为了举例说明这种算法，我们来为文法3.11写一个递归下降语法分析器。

	$L \to \text{end}$
$S \to \text{if } E \text{ then } S \text{ else } S$	$L \to ; S\ L$
$S \to \text{begin } S\ L$	
$S \to \text{print } E$	$E \to \text{num} = \text{num}$

文法 3.11

这个语言的递归下降语法分析器对每个非终结符有一个函数，非终结符的每个产生式对应一个子句。

```
datatype token = IF | THEN | ELSE | BEGIN | END | PRINT
               | SEMI | NUM | EQ

val tok = ref (getToken())
fun advance() = tok := getToken()
fun eat(t) = if (!tok=t) then advance() else error()

fun S() = case !tok
           of IF => (eat(IF); E(); eat(THEN); S();
                     eat(ELSE); S())
            | BEGIN => (eat(BEGIN); S(); L())
            | PRINT => (eat(PRINT); E())
and L() = case !tok
           of END => (eat(END))
            | SEMI => (eat(SEMI); S(); L())
```

```
and E() = (eat(NUM); eat(EQ); eat(NUM))
```

若恰当地定义 error 和 getToken，这个程序就能很好地对文法3.11进行分析。

这种简单方法的成功给了我们一种鼓励，让我们再用它尝试文法3.10：

```
fun S() = (E(); eat(EOF))
and E() = case !tok
           of ? => (E(); eat(PLUS); T())
            | ? => (E(); eat(MINUS); T())
            | ? => (T())
and T() = case !tok
           of ? => (T(); eat(TIMES); F())
            | ? => (T(); eat(DIV); F())
            | ? => (F())
and F() = case !tok
           of ID => (eat(ID))
            | NUM => (eat(NUM))
            | LPAREN => (eat(LPAREN); E(); eat(RPAREN))
```

这时我们遇到了一个冲突：函数 E 不知道该使用哪个子句。考虑标记串 $(1*2-3)+4$ 和 $(1*2-3)$ 初次调用 E 时，对于前者，应使用产生式 $E \to E+T$；而对于后者，则应该使用 $E \to T$。

递归下降分析也称为预测 (predictive) 分析，它只适合于每个子表达式的第一个终结符号能够为产生式的选择提供足够信息的那种文法。为了便于理解，我们将形式化 FIRST 集合的概念，然后用一个简单的算法导出无冲突的递归下降语法分析器。

就像从正则表达式可以构造出词法分析器一样，也存在语法分析器的生成器之类的工具，可以用来构造预测分析器。但是如果我们打算使用工具的话，可能同时会需要用到基于更强大的 LR(1) 分析算法的工具，3.3 节将讲述 LR(1) 分析算法。

有时使用语法分析器生成工具并不方便，或者说不可能。预测分析器的优点就在于其算法简单，我们可以用它手工构造分析器，而无需自动构造工具。

3.2.1 FIRST 集合和 FOLLOW 集合

给定一个由终结符和非终结符组成的字符串 γ，FIRST(γ) 是从 γ 可以推导出的任意字符串中的开头终结符组成的集合。例如，令 $\gamma = T*F$。任何可从 γ 推导出的由终结符组成的字符串都必定以 id、num 或 (开始。因此有

FIRST($T*F$)= {id,num,(}

如果两个不同的产生式 $X \to \gamma_1$ 和 $X \to \gamma_2$ 具有相同的左部符号 (X)，并且它们的右部有重叠的 FIRST 集合，则这个文法不能用预测分析法来分析。因为如果存在某个终

结符 I，它既在 FIRST(γ_1) 中，又在 FIRST(γ_2) 中，则当输入标记为 I 时，递归下降分析器中与对应的函数将不知道该怎样做。

FIRST 集合的计算似乎很简单。若 $\gamma = XYZ$，则好像只要忽略 Y 和 Z，只需计算 FIRST(X) 就可以了。但是考虑文法3.12就可以看出情况并不如此。因为 Y 可能产生空串，所以也可能产生空串，于是我们发现 FIRST(XYZ) 一定包含 FIRST(Z)。因此，在计算 FIRST 集合时，我们必须跟踪能产生空串的符号；这种符号称为可空（nullable）符号。同时我们还必须跟踪有可能跟随在可空符号之后的其他符号。

$Z \to d$	$Y \to$	$X \to Y$
$Z \to X\ Y\ Z$	$Y \to c$	$X \to a$

文法 3.12

对于一个特定的文法，当给定由终结符和非终结符组成的字符串 γ 时，下述结论成立。

- 若 X 可以导出空串，那么 nullable(X) 为真。
- FIRST(γ) 是可从 γ 推导出的字符串的开头终结符的集合。
- FOLLOW(X) 是可直接跟随于 X 之后的终结符集合。也就是说，如果存在着任一推导包含 Xt，则 t ∈FOLLOW(X)。当推导包含 $XYZt$，其中 Y 和 Z 都推导出 ϵ 时，也有 t ∈FOLLOW(X)。

可将 FIRST、FOLLOW 和 nullable 精确地定义为满足如下属性的最小集合：

对于每个终结符 Z，FIRST[Z]={Z}

for 每个产生式 $X \to Y_1Y_2\cdots Y_k$

  if $Y_1 ... Y_k$ 都是可为空的（或者如果 $k = 0$）

    then nullable[X]=true

  for 每个 i 从 1 到 k，每个 j 从 $i+1$ 到 k，

    if $Y_1 \cdots Y_{i-1}$ 都是可为空的（或者如果 $i = 1$）

      then FIRST[X]=FIRST[X] ∪ FIRST[Y_i]

    if $Y_{i+1} \cdots Y_k$ 都是可为空的（或者如果 $i = k$）

      then FOLLOW[Y_i]=FOLLOW[Y_i] ∪ FOLLOW[X]

    if $Y_{i+1} \cdots Y_{j-1}$ 都是可为空的（或者如果 $i + 1 = j$）

      then FOLLOW[Y_i]=FOLLOW[Y_i] ∪ FIRST[Y_j]

计算 FIRST、FOLLOW 和 nullable 的算法**??**遵循的正式上述事实。我们只需要简单地用一个赋值语句替代每一个方程并进行迭代，就可以计算出每个字符串的 FIRST、FOLLOW 和 nullable。

计算 FIRST、FOLLOW 和 nullable 的算法

将所有的 FIRST 和 FOLLOW 初始化为空集合，将所有的 nullable 初始化为 false。

for 每个终结符 Z

FIRST[Z]←{Z}

repeat

for 每个产生式 $X \to Y_1 Y_2 \cdots Y_k$

if $Y_1 ... Y_k$ 都是可为空的（或者如果 $k = 0$）

then nullable[X]←true

for 每个 i 从 1 到 k，每个 j 从 $i+1$ 到 k，

if $Y_1 \cdots Y_{i-1}$ 都是可为空的（或者如果 $i = 1$）

then FIRST[X]←FIRST[X] ∪ FIRST[Y_i]

if $Y_{i+1} \cdots Y_k$ 都是可为空的（或者如果 $i = k$）

then FOLLOW[Y_i]←FOLLOW[Y_i] ∪ FOLLOW[X]

if $Y_{i+1} \cdots Y_{j-1}$ 都是可为空的（或者如果 $i + 1 = j$）

then FOLLOW[Y_i]←FOLLOW[Y_i] ∪ FIRST[Y_j]

until FIRST、FOLLOW 和 nullable 在此轮迭代中没有改变

当然，按正确的顺序考察产生式会有助于提高这个算法的效率，具体见 17.4 节。此外，这 3 个关系式不必同时计算，可单独计算 nullable，然后计算 FIRST，最后计算 FOLLOW。

关于集合的一组方程变成了计算这些集合的算法，这并不是第一次遇到：在 2.4.2 节计算 ϵ 闭包的算法中我们也遇到了这种情形。这也不会是最后一次；这种迭代到不动点的技术也适用于编译器后端优化使用的数据流分析。

我们来将这一算法应用于文法3.12。一开始，我们有

	nullable	FIRST	FOLLOW
X	no		
Y	no		
Z	no		

在第一次迭代中，我们发现 $a \in$ FIRST[X]，Y 是可为空的，$c \in$ FIRST[Y]，$d \in$ FIRST[Z]，$d \in$ FOLLOW[X]，$d \in$ FOLLOW[X]，$d \in$ FOLLOW[Y]。因此有

	nullable	FIRST	FOLLOW
X	no	a	c d
Y	yes	c	d
Z	no	d	

在第二次迭代中，我们发现 X 是可为空的，$c \in$ FIRST[X]，$\{a, c\} \subseteq$ FIRST[Z]，$\{a, c, d\} \subseteq$ FOLLOW[X]，$\{a, c, d, \} \subseteq$ FOLLOW[Y]，因此有

	nullable	FIRST	FOLLOW
X	yes	a c	a c d
Y	yes	c	a c d
Z	no	a c d	

第三次迭代没有发现新的信息，于是算法终止。

也可将 FIRST 关系推广到符号串：

$$\begin{aligned} &\text{FIRST}(X\gamma) = \text{FIRST}[X] && \text{若 !nullable}[X] \\ &\text{FIRST}(X\gamma) = \text{FIRST}[X] \cup \text{FIRST}(\gamma) && \text{若 nullable}[X] \end{aligned}$$

并且类似地，如果 γ 中的每个符号都是可为空的，则称符号串 γ 是可为空的。

3.2.2 构造预测分析器

考虑一个递归下降分析器。非终结符 X 的分析函数对 X 的每个产生式都有一个子句，因此，该函数必须根据下一个输入标记 T 来选择其中的一个子句。如果能够为每一个 (X,T) 选择出正确的产生式，就能够写出这个递归下降分析器。我们需要的所有信息可以用一张关于产生式的二维表来表示，此表以文法的非终结符 X 和终结符 T 作为索引。这张表称为预测分析表（predictive parsing table）。

为了构造这张表，对每个 $T \in \text{FIRST}(\gamma)$，在表的第 X 行第 T 列，填入产生式 $X \to \gamma$。此外，如果 γ 是可为空的，则对每个 $T \in \text{FOLLOW}(X)$，在表的第 X 行第 T 列，也填入该产生式。

图3.1给出了文法3.12的预测分析表。但是其中有些项中的产生式不止一个！出现这种多重定义的项意味着不能对文法3.12进行预测分析。

	a	c	d
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$

图 3.1: 文法3.12的预测分析表

如果我们仔细地检查这一文法，就能发现它具有二义性。句子 d 有多个语法树，包括：

3.3 LR 分析

3.4 使用分析器的生成器

文法 3.1: 直线式程序的表示

```
%%
%term ID | WHILE | BEGIN | END | DO | IF | THEN | ELSE | SEMI | ASSIGN | EOF

%nonterm prog | stm | stmlist

%pos int
%verbose
%start prog
%eop EOF %noshift EOF

%%

prog: stmlist                    ()

stm : ID ASSIGN ID               ()
    | WHILE ID DO stm            ()
    | BEGIN stmlist END          ()
    | IF ID THEN stm             ()
    | IF ID THEN stm ELSE stm    ()

stmlist : stm                    ()
        | stmlist SEMI stm       ()
```

文法 3.2: 直线式程序的表示

```
%%
%term INT | PLUS | MINUS | TIMES | UMINUS | EOF
%nonterm exp
%start exp
%eop EOF

%left PLUS MINUS
%left TIMES
%left UMINUS
%%

exp : INT                    ()
    | exp PLUS exp           ()
    | exp MINUS exp          ()
    | exp TIMES exp          ()
```

```
    | MINUS exp %prec UMINUS ()
```

3.5 错误恢复

3.6 程序设计：语法分析

3.7 推荐阅读

3.8 习题

第四章 抽象语法

抽象的 (ab-stract)：从所有具体实例中提取出来的。

韦氏词典

编译器的工作不仅是识别一个句子是否是属于某一个文法的语言，它还必须对那个句子做更多的事情。语法分析器中的语义动作（semantic action）能够对所分析的短语做一些有用的事情。

在递归下降语法分析器中，语义动作代码分散在实现语法分析的控制流中。在用 ML-Yacc 说明的语法分析器中，语义动作是一段附带在文法产生式中的 ML 程序代码。

4.1 语义动作

每个终结符和非终结符都可关联一个语义值类型。例如，在使用文法3.2定义的一个简单计算器中，与 exp 和 INT 关联的类型可能是 int；而其他单词则不需携带值。当然，与单词关联的类型必须与词法分析器随同这个单词一起返回的类型相匹配。

对于规则 $A \to B\ C\ D$ 语义动作返回的值的类型必须是与非终结符 A 关联的类型。但它可以由所匹配的终结符和非终结符 B, C, D 的相关值来建立这个返回值。

4.1.1 递归下降

在递归下降语法分析器中，语义行为是语法分析函数所返回的值，或者是这些函数产生的副作用，或者两者兼而有之。对于每个终结符和非终结符，我们给它关联一种语义值类型（来自于实现该编译器的语言），其中语义值所表示的是由那个符号导出的短语。

Listing 4.1: 文法**??**的递归下降解释器

```
datatype token = ID of string | NUM of int | PLUS | MINUS

fun F() = case !tok
           of ID s  => (advance(); lookup(s))
            | NUM i => (advance(), i)
            | LPAREN => (eat(LPAREN);
                         let i = E()
                          in eatOrSkipTo(RPAREN,[PLUS,TIMES,EOF]);
                             i
                         end)
            | EOF    => (print("expected␣factor"); 0)
            | _      => (print("expected␣ID,␣NUM,␣or␣(");
```

```
                        skipto[PLUS,TIMES,RPAREN,EOF]; 0)

and T() = case !tok
           of ID    => T'(F())
            | NUM   => T'(F())
            | LPAREN => T'(F())
            | _     => (print("expected␣ID,␣NUM,␣or␣(");
                        skipto[PLUS,TIMES,RPAREN,EOF]; 0)

and T'(a) = case !tok
             of PLUS  => a
              | TIMES => (eat(TIMES); T'(a*F()))
              | RPAREN => a
              | EOF   => a

and eatOrSkipTo(expected,stop) =
     if !tok = expected then eat(expected)
     else (print(...); skipto(stop))
```

程序4.1是文法**??**的递归下降语法分析器。它指出了单词 ID 和 NUM 分别必须携带 string 类型和 int 类型的值。我们假定存在一张将标识符映射至整数的查找表。与 E, T, F 等关联的类型是 int，它们的语义动作很容易实现。

但对于像 T' 这样人为引入的符号（为消除左递归而引入的）其语义动作则要稍为棘手一点。原来的产生式是 $T \to T * F$，它的语义动作原本是：

```
let val a = T()
    val _ = eat(TIMES)
    val b = F()
 in a*b
end
```

但将文法改写后，产生式 $T' \to *FT'$ 中的“*”没有了左操作数。解决这个问题的一种方法是，将这个左操作数作为参数由 T 传递给 T'，如程序4.1所示。

4.1.2 ML-Yacc 生成的语法分析器

语法分析器的 ML-Yacc 规范由一组文法规则组成，每个规则标注有一个语义动作，此语义动作是一个 ML 表达式。ML-Yacc 生成的语法分析器每当用一条规则进行归约时，便执行对应的语义动作代码。

程序4.2以文法3.2为例说明了其方法。语义动作可以使用它们的名字来引用右部符号的语义值。如果符号重复出现（例如 exp 在 PLUS 规则中），那么这些重复出现的符号将使用数字后缀来区分（例如 exp1 和 exp2）。exp 的每个语义表达式所产生的值都必须和 exp 非终结符的类型相匹配，这里 exp 的类型是 int。

Listing 4.2: 文法3.2的 ML-Yacc 版本

```
%%
%term INT of int | PLUS | MINUS | TIMES | UMINUS | EOF
%nonterm exp of int
%start exp
%eop EOF

%left PLUS MINUS
%left TIMES
%left UMINUS
%%

exp : INT             (INT)
    | exp PLUS exp    (exp1 + exp2)
    | exp MINUS exp   (exp1 - exp2)
    | exp TIMES exp   (exp1 * exp2)
    | MINUS exp   %prec UMINUS   (~exp)
```

在更为真实的例子中，可能会存在着若干非终结符，且每个非终结符各自有不同的类型。

ML-Yacc 生成的语法分析器同时维护着一个状态栈和一个语义值栈，并由此实现对语义值的操作。语法分析栈上的每个符号在语义值栈上都有对应的语义值。语法分析器在执行一个归约时必须执行用 ML 语言写的语义动作；它通过引用栈顶 k 个元素之一（对于具有 k 个右部符号的规则而言）来满足对一个右部语义值的每一个引用。当语法分析器从符号栈弹出顶部的 k 个元素并压入一个非终结符号时，它也同时从语义值栈弹出 k 个值，并压入通过执行语义动作的 ML 代码而得到的值。

Listing 4.3: 直线型程序的解释器

```
fun update(table,id,num) = fn j => if j=id then num else table(j)
val emptytable fn j => raise Fail ("uninitialized var: " ^ j)
%%
%term INT of int | ID of string | PLUS | MINUS | TIMES | DIV |
      ASSIGN | PRINT | LPAREN | RPAREN | COMMA | SEMICOLON | EOF
%nonterm exp of table->int
       | stm of table->table
       | exps of table->unit
       | prog of table
%right SEMICOLON
%left PLUS MINUS
%left TIMES DIV

%start prog
%eop EOF
```

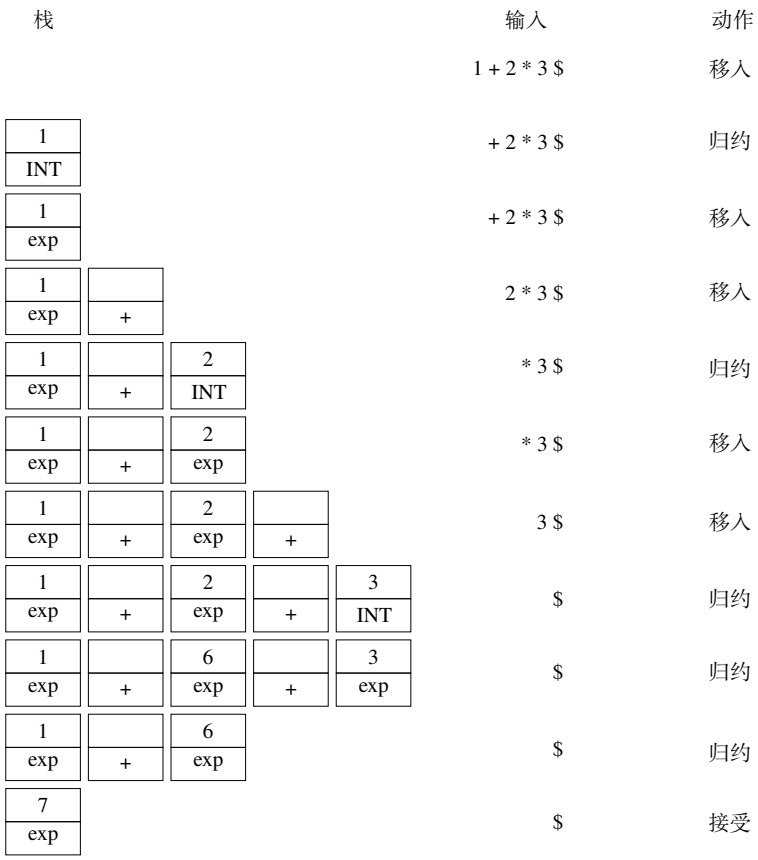


图 4.1: 使用语义栈进行语法分析

```
%pos int
%%
prog : stm                      (stm(emptytable))

stm  : stm SEMICOLON stm        (fn t => stm2(stm1(t)))
stm  : ID ASSIGN exp            (fn t => update(t,ID,exp))
stm  : PRINT LPAREN exps RPAREN (fn t => (exps t; t))

exps : exp                      (fn t => print(exp t))
exps : exps COMMA exp           (fn t => (exps t; print(exp t)))

exp  : INT                      (fn t => INT)
exp  : ID                       (fn t => t(ID))
exp  : exp PLUS exp             (fn t => exp1(t) + exp2(t))
exp  : exp MINUS exp            (fn t => exp1(t) - exp2(t))
exp  : exp TIMES exp            (fn t => exp1(t) * exp2(t))
exp  : exp DIV exp              (fn t => exp1(t) / exp2(t))
exp  : stm COMMA exp            (fn t => exp(stm(t)))
exp  : LPAREN exp RPAREN        (exp)
```

第五章 语义分析

> 语义的（se-man-tic）：与语言表达的含义相关的。
>
> ——韦氏词典

编译器的语义分析（semantic analysis）阶段的任务是：将变量的定义与变量的使用联系起来，检查每一个表达式是否有正确的类型，并将抽象语法转换成更简单的、适合于生成机器代码的表示。

5.1 符号表

语义分析阶段一个主要的工作是符号表的维护。符号表（symbol table）也称为环境(environment)，其作用是将标识符映射到它们的类型和存储位置。在处理类型、变量和函数的声明时，这些标识符便与其在符号表中的“含义”相绑定。每当发现标识符的使用（即非声明性出现）时，便在符号表中查看它们的含义。

程序中的每一个局部变量都有一个作用域（scope），该变量在此作用域中是可见的。例如：在 Tiger 表达式 let D in E end 中，所有在 D 中声明的变量、类型和函数在直到 E 结束为止的范围内都是可见的。当语义分析到达每一个作用域的结束时，所有绑定并局部于此作用域的标识符都将被抛弃。

环境是由一些绑定（binding）构成的集合，所谓绑定指的是标识符与其含义之间的一种映射关系，我们用箭头 $\mapsto$ 来表示绑定。例如，环境 σ_0 包含绑定 {g↦string,a↦int}；这表示标识符 a 是整型变量，g 是字符串变量。

考虑下面这个用 Tiger 语言编写的简单例子：

```
function f(a:int, b:int, c:int) =
   (print_int(a+c);
    let var j := a+b
        var a := "hello"
    in print(a); print_int(j)
    end;
    print_int(b)
   )
```

假设编译这段程序时其环境为 σ_0。第 1 行关于形式参数的声明使我们得到了表 $\sigma_1 = \sigma_0 + \{a \mapsto int, b \mapsto int, c \mapsto int\}$，即，在 σ_0 中加入了 a，b 和 c 的新绑定。在 σ_1 中可查到第 2 行使用的那两个标识符的含义。第 3 行创建了表 $\sigma_2 = \sigma_1 + \{j \mapsto int\}$；第 4 行则创建了表 $\sigma_3 = \sigma_2 + \{a \mapsto string\}$。

当被“加”到一起的两个环境含有同一个符号的不同绑定时，例如，在 σ_2 和 $\{a \mapsto$

$string\}$ 分别将 a 映射为 int 和 string 的情况下，两个表的“+”操作是怎样的呢？为了使得作用域规则按照我们期望的真实程序设计语言的方式工作，我们需要 $\{a \mapsto string\}$ 的优先级更高。因此对于两张表 X 和 Y，我们假定 $X + Y$ 不等于 $Y + X$，并且右边表中的绑定将覆盖左边表中相同符号的绑定。

当到达第 6 行时，我们将抛弃 σ_3 而回到 σ_1，并在 σ_1 中查看第 7 行出现的标识符 b。最后在第 8 行我们将抛弃 σ_1，而回到 σ_0。

应该怎样实现上述过程？在实际中有两种选择。一种是*函数式风格* (functional style)，在这种方式中，当创建 σ_2 和 σ_3 时，保持 σ_1 原来的状态不变。这样，当再次需要 σ_1 时，σ_1 就是已经就绪的了。

另一种是*命令式风格* (imperative style)，在这种方式下，我们修改 σ_1 直到它成为 σ_2。这种*破坏性更新*会“毁坏” σ_1；即，在 σ_2 存在期间，我们不能查看 σ_1 中的符号。但是，当我们完成了 σ_2 中的处理时，可以*撤销*对 σ_1 的更新从而使返回到原来的状态。于是，存在着一个单一的全局环境 σ（它在不同的时间变成 $\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_1, \sigma_0$）和一个“撤销栈”（此栈含有可用来撤销破坏性更新所需要的足够信息）。每当添加一个符号到环境的同时，也将该符号加入到撤销栈中；在作用域的结束点 (例如，在第 6 或第 8 行)，这些符号将从撤销栈中弹出，并且它们的最近一次绑定也将从 σ 中被删除（从而恢复到它们的前一次绑定）。

无论被编译的语言或用于实现编译器的语言是“函数式的”、“命令式的”，还是“面向对象的”，都可以采用函数式的或命令式的环境管理方式。

5.1.1 多个符号表

在有些语言中，可以同时存在若干种活跃的环境：程序中每一个模块、类或者记录都有它自己的符号表 σ。

```
structure M = struct
  structure E = struct
    val a = 5;
  end
  structure N = struct
    val b = 10
    val a = E.a + b
  end
  structure D = struct
    val d = E.a + N.a
  end
end
```

(a) ML的例子

```
package M;
class E {
  static int a = 5;
}
class N {
  static int b = 10;
  static int a = E.a + b;
}
class D {
  static int d = E.a + N.a;
}
```

(b) Java的例子

图 5.1: 同时存在的若干活跃环境

在对图5.1进行分析时令 σ_0 是包含一些预定义函数的基本环境，并令

$$\sigma_1 = \{a \mapsto int\}$$
$$\sigma_2 = \{E \mapsto \sigma_1\}$$
$$\sigma_3 = \{b \mapsto int, a \mapsto int\}$$
$$\sigma_4 = \{N \mapsto \sigma_3\}$$
$$\sigma_5 = \{d \mapsto int\}$$
$$\sigma_6 = \{D \mapsto \sigma_5\}$$
$$\sigma_7 = \sigma_2 + \sigma_4 + \sigma_6$$

在 ML 语言的情况下 (图5.1a)，编译 N 时查看标识符使用的环境是 $\sigma_0 + \sigma_2$。编译 D 的环境是 $\sigma_0 + \sigma_2 + \sigma_4$ 分析的结果是 $\{M \mapsto \sigma_7\}$。

Java 语言 (图5.1b) 允许向前引用（N 中现表达式 $D.d$ 是合法的），因此，N 和 D 的编译环境都是 σ_7；对于这个程序而言，其结果仍然是 $\{M \mapsto \sigma_7\}$。

5.1.2 高效的命令式风格符号表

大型程序可能含有数千个不同的标识符，因此符号表的组织必须要能够进行高效的查找。

命令式风格的环境通常采用哈希表来实现，哈希表的效率很高。操作 $\sigma' = \sigma + \{a \mapsto \tau\}$ 是通过以 a 作为键值将 τ 插入哈希表来实现的。一个简单的带有外部散列链的哈希表就能够很好地工作，并且很容易实现对作用域的删除操作（当 a 的作用域结束时，我们需要删除 $\{a \mapsto \tau\}$ 以恢复 σ）。

程序5.1实现了一个简单的哈希表。第 i 条散列链（bucket）是由所有这样的元素组成的一张链表，它们的键值的哈希值是：i 模 SIZE。binding 的类型可以是任意类型；在一个真实的程序中，哈希表模块可能是一个函子（functor），或者 table 类型可能是多态的（polymorphic）。

Listing 5.1: 带有外部散列链的哈希表

```
val SIZE = 109 (* 必须是素数 *)
type binding = ...
type bucket = (string * binding) list
type table = bucket Array.array
val t : table = Array.array(SIZE, nil)

fun hash(s: string) : int =
  CharVector.foldl (fn (c,n) => (n*256+ord(c)) mod SIZE) 0 s

fun insert(s: string, b: binding) =
  let val i = hash(s) mod SIZE
   in Array.update(t, i, (s,b)::Array.sub(t,i))
```

```
  end

exception NotFound

fun lookup(s: string) =
  let val i = hash(s) mod SIZE
      fun search((s',b)::rest) = if s=s' then b
                                         else search rest
        | search nil = raise NotFound
   in search(Array.sub(t,i))
  end

fun pop(s: string) =
  let val i = hash(s) mod SIZE
      val (s',b)::rest = Array.sub(t,i)
   in assert(s=s');
      Array.update(t,i,rest)
  end
```

考虑当 σ 已经包含 $a \mapsto \tau_1$ 时，操作 $\sigma + \{a \mapsto \tau_2\}$ 的情况。函数 insert 将 $a \mapsto \tau_1$ 保留在散列链中，并将 $a \mapsto \tau_2$ 插入在散列链的前部。于是在 a 的作用域结束处执行 pop(a) 之后，便恢复了 σ。当然，只有当绑定的插入和弹出都按栈的方式操作时，pop 操作才能正确工作。

在工业级别的编译器中实现这种符号表则还应当在若干方面有所改进，见习题 5.1。

5.1.3 高效的函数式符号表

在函数式风格的实现中，我们希望以这样一种方式来计算 $\sigma' = \sigma + \{a \mapsto \tau\}$，即希望在 σ' 活跃的情况下仍然能够查找 σ 中的标识符。因此，我们不是将一个绑定加入到已存在的表中来“改变”这个表，而是通过计算现有的这个表与一个新的绑定的“和”来创建一个新表。这类似于在计算 7+8 时，不是将 8 加到 7 之上得到的结果覆盖原有的 7，而是创建一个新的值 15——从而 7 仍然可用于其他的计算。

但是，对于非破坏性更新，哈希表的效率不高。图5.2a 给出的是一个实现了映射 m_1 的哈希表。我们可以快速而高效地将 $mouse$ 记录添加到表的第 5 个位置，这只要将 $mouse$ 记录指向第 5 个链表原来的表头，并使第 5 个位置指向该 $mouse$ 记录即可。但这样一来，映射 m_1 将不复存在：我们已破坏了它而使它变成了 m_2。另一种可选的方法是复制散列数组但仍然共享所有老的散列链，如图5.2b 所示。不过这种方法十分低效：哈希表的散列数组可能会相当大，它与元素的个数成正比，因此，对每一个新增添到表中的登记项都复制此数组是不现实的。

通过使用二叉搜索树，我们可以高效地实现对这种搜索树的“函数式的”添加。例如，考虑图5.3的搜索树，它表示了如下映射：

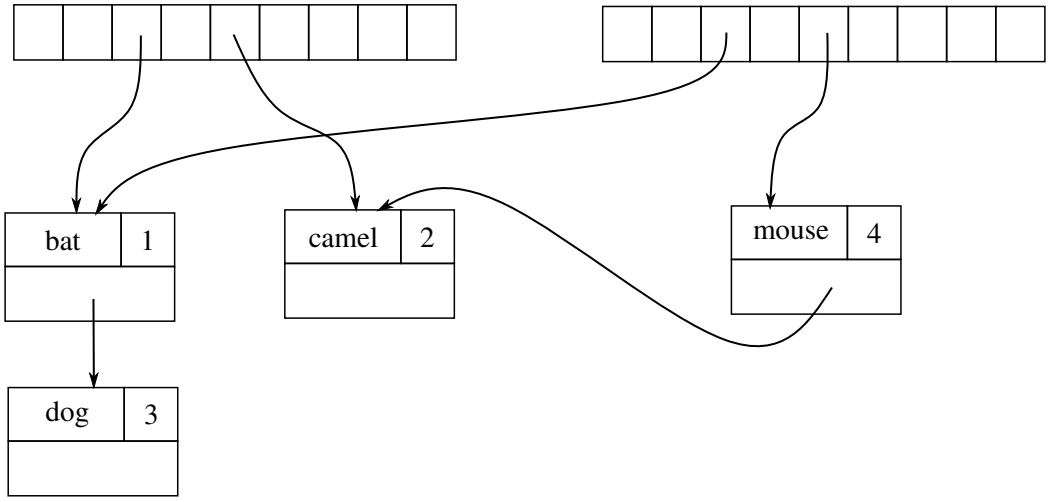


图 5.2: 哈希表

$$m_1 = \{bat \mapsto 1, camel \mapsto 2, dog \mapsto 3\}$$

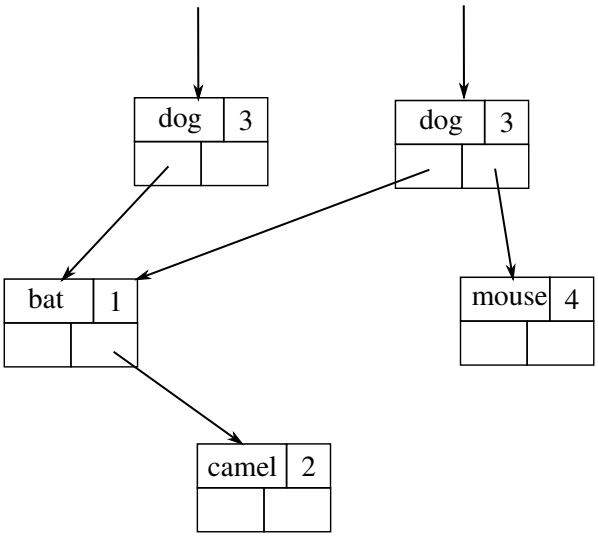


图 5.3: 二叉搜索树

我们可以如图5.3b 那样增加一个绑定 $mouse \mapsto 4$，在不破坏映射 m_1 的情况下创建一个新的映射 m_2。如果想要在树的第 d 层添加一个新的结点，则必须创建 d 个新结点——但不必复制整棵树。因此创建一棵新树（这棵树与原来的树共享一部分结构）的效率与查找一个元素相同；对于棵有 n 个结点的平衡树，时间在 $\log(n)$ 之内。这是使用持久化数据结构（persistent data structure）的一个例子；有一种能保持二叉树平衡的持久化红黑树，这种树可以保证访问一个结点的时间不会超过 $\log(n)$（见习题 1.1c 和第 206 页）。

5.1.4 Tiger 编译器的符号

程序5.1中的哈希表操作需要查看被散列的字符串 s 中的每一个字符，然后将 s 与第 i 个散列链中的字符串逐一进行比较。为了避免不必要的字符串比较，我们可以将每一个字符串转变成一个 symbol 对象，使得任意一个给定字符串的所有不同出现都被转换成同一个符号对象。

Symbol 模块实现这种符号，它有下以几个重要特点：

- 比较两个符号相等的运算非常快（仅仅是指针或者整数比较）。
- 提取一个整型哈希键值的操作非常快（当我们想要哈希表将一个符号映射到其他某种对象时会需要这种操作）。
- 比较两个字符串的“大于”运算（按任意顺序）非常快（当我们想要构造二叉搜索树时会需要这种操作）。

即使想要构造的是一个映射符号到绑定的函数式风格的环境，我们也可以使用破坏-更新式的哈希表来映射字符串到符号：这种做法可以保证使“abc”的第二次出现与它的第一次出现都映射到相同的符号。程序5.2给出了 Symbol 模块的接口。

Listing 5.2: SYMBOL 的签名

```
signature SYMBOL =
sig
  eqtype symbol
  val symbol : string -> symbol
  val name : symbol -> string

  type 'a table
  val empty : 'a table
  val enter : 'a table * symbol * 'a -> 'a table
  val look : 'a table * symbol -> 'a option
end
```

我们使用 table 来实现 Symbol 模块中的环境，table 将 symbol 映射至绑定。在这个编译器中，我们希望不同的用途有不同的 binding 表示——类型绑定用于类型；值绑定用于变量和函数——因此，我们让 table 是一个多态类型。也就是说，无论 α 是一个类型绑定，或是一个值绑定，又或者是其它任意类型的绑定，一个 α table 是一个从 symbol 到 α 的映射。

给定一张表，我们可以使用 enter 函数来将新的绑定添加到表中（创建一张新的表，而不修改旧的表）。如果在旧的表中有相同符号的绑定，那么旧的绑定将被新的绑定所替换。

look(t,s) 函数将寻找符号 s 在表 t 中的绑定 b，返回值是 SOME(b)。如果符号并未绑定在表中，返回值是 NONE。

Symbol 抽象的实现（程序5.3）会使符号表更加的高效。为了使得像 α table 这样的“函数式映射”更加的高效，我们选择使用平衡二叉搜索树。搜索树的实现可以在 *Standard*

ML of New Jersey Library 中的 IntBinaryMap 模块中找到（细节可以参看 Library 手册）。

Listing 5.3: 符号表的实现。HashTable 和 IntBinaryMap 来自 *Standard ML of New Jersey Library*

```
structure Symbol :> SYMBOL =
struct
  type symbol = string * int

  exception Symbol
  val nextsym = ref 0
  val hashtable : (string,int) HashTable.hash_table =
    HashTable.mkTable(HashString.hashString, op = ) (128,Symbol)

  fun symbol name =
    case HashTable.find hashtable name
      of SOME i => (name,i)
       | NONE => let val i = !nextsym
                 in nextsym := i+1;
                    HashTable.insert hashtable (name,i);
                    (name,i)
                 end

  fun name(s,n) = s

  type 'a table = 'a IntBinaryMap.map
  val empty = IntBinaryMap.empty
  fun enter(t: 'a table, (s,n): symbol, a: 'a) = IntBinaryMap.insert(t,n,a)
  fun look(t: 'a table, (s,n): symbol) = IntBinaryMap.look(t,n)
end
```

二叉搜索树需要一个全序（total ordering）函数（“小于”）来比较 symbol 类型。由于字符串拥有全序函数，所以可以用字符串来实现符号，但是字符串的比较函数运行速度并不是很快。

作为替代解决方案，符号包含了整数来用于“小于”比较操作。符号的排序完全是任意的——因为排序基于不同字符串在文件中被看到的顺序——但对于搜索树算法来说无关紧要。符号的内部表示是一个元组 string × int，元组中的 string 部分仅用来实现接口的 name 函数。

不同的字符串必须有不同的整数作为标识。Symbol 模块可以维护一个计数器（count）来对它看到的不同的字符串有多少进行计数。对于任何一个之前没有见过的字符串，都用当前计数器的值作为字符串的整数标识。对于任何一个之前见过的字符串都使用相同的整数进行标识。一个传统的破坏更新式的哈希表可以用来查询某个字符串所对应的整数标识，这是因为 Symbol 模块不再需要之前版本的 string → symbol 映射。Symbol 模块对使用本模块的所有客户端隐藏了破坏更新这样的副作用，而这只需要一个简单的 symbol

接口就可以做到。

5.1.5 命令式风格的符号表

如果我们想要使用破坏更新式的表，那么 SYMBOL 签名中的“table”部分将如下所示：

```
type 'a table
val new : unit -> 'a table
val enter : 'a table * symbol * 'a -> unit
val look : 'a table * symbol -> 'a option

val beginScope : 'a table -> unit
val endScope : 'a table -> unit
```

为了处理破坏性更新的“撤销”要求，接口函数 beginScope 记住表的当前状态；而 endScope 使表恢复到它位于最近一次执行且还未结束的 beginScope 状态。

命令式的表使用哈希表来实现的。当插入绑定 $x \mapsto b$ 时，x 被哈希到索引 i，并且在第 i 条散列链的链首放置一个绑定 $x \mapsto b$。如果这个表已经包含了一个绑定 $x \mapsto b'$，该绑定将仍保留在散列链中且被 $x \mapsto b$ 所隐藏。这一点很重要，因为它将支持撤销操作的实现（beginScope 和 endScope）。

此外，我们还需要一个辅助栈，它给出符号被“压入”到符号表时的次序。当将 $x \mapsto b$ 加入到符号表时，x 便被压入栈中。beginScope 要压入一个特殊的标记至栈。于是，为了实现 endScope，要从栈中弹出符号直至遇到最顶上的一个标记，并将该标记也一并弹出。每当弹出一个符号的同时，也从它的散列链中删除为首的绑定。

5.2 Tiger 编译器的绑定

符号表中应填入什么内容？也即，什么是一个 binding？Tiger 有两个独立的名字空间，一个是类型的名字空间，另一个是函数和变量的名字空间。与类型标识符关联的是 Types.ty。如程序5.4所示，Types 模块描述了表示各种类型的结构。

Listing 5.4: Types 结构

```
structure Types =
struct
  type unique = unit ref

  datatype ty = INT
              | STRING
              | RECORD of (Symbol.symbol * ty) list * unique
              | ARRAY of ty * unique
              | NIL
              | UNIT
```

```
              | NAME of Symbol.symbol * ty option ref
 end
```

Tiger 中的基本类型是 int 和 string；每一种类型或者是基本类型，或者是由其他类型（基本类型，记录，或数组）通过记录或数组构造出来的类型。另外，由于每个“记录类型表达式”都会创建一个新的（且不同的）记录类型，即使它们的各个域都相同，我们依然有一个“唯一的”值来识别它。（你唯一可以做的一件比较有趣的事情就是使用`unit ref`来测试某个记录类型是否和另一个记录类型是否相等；因为每个`ref`都是唯一的。）

记录类型携带有附加信息：各个域的名字和类型。

数组与记录类似：ARRAY 构造器携带有数组元素的类型，还携带了一个“唯一的”值来将这个数组类型和其它数组类型区分开。

如果我们编译的是另外的某种语言，有可能会把下面的程序段视为合法的程序：

```
let type a = {x: int,y: int}
    type b = {x:int, y: int}
    var i : a := ...
    var j : b := ...
 in i := j
end
```

这个程序在 Tiger 中是非法的，但是如果语言支持结构上等价的两个类型可换，这个程序就是合法的。为了在这种语言的编译器中测试类型的等价性，我们需要递归地逐一检查每个域的类型。

不过，下面的 Tiger 程序是合法的，因为类型 c 与类型 a 相同：

```
let type a = {x: int, y: int}
    type c = a
    var i : a := ...
    var j : c := ...
 in i := j
end
```

导致创建一个新的不同类型的并不是类型*声明*，而是类型*表达式*`{x:int,y:int}`。

在 Tiger 中，表达式 nil 属于任何记录类型。我们设计了一个特殊的“nil”类型来处理这种例外的情形。另外，有的表达式没有返回值，因此我们设计了一个类型 unit。

在处理相互递归的类型时，对于那种只知道其名字但还未见到其定义的类型，需要有一个占位符。类型`NAME(sym,ref(SOME(t)))`和类型`t`相同；但`NAME(sym,ref(NONE))`也只是占位符而已。

环境

Symbol 模块中的类型 table 提供从符号到其绑定的映射。这样，我们将有一个类型环境（type enviroment）和一个值环境（value enviroment）。下面的 Tiger 程序说明了只有一个环境是不够的：

```
let type a = int
    var a : a := 5
    var b : a := a
 in b+a
end
```

符号 a 在预期类型标识符的语法上下文中表示类型“a”，在预期变量的语法上下文中表示变量“a”。

对于类型标识符，我们需要记住的只是它代表的类型。因此，类型环境是符号至 Types.ty 的映射——即，它是一张`Types.ty Symbol.table`。如图5.4所示，`Env`模块包含有一个`base_tenv`值——即“基本的”或“预定义的”类型环境。它映射符号 int 到 Ty.INT，映射 string 到 Ty.STRING。

```
signature ENV =
sig
  type access
  type ty
  datatype enventry = VarEntry of {ty: ty}
                    | FunEntry of {formals: ty list, result: ty}
  val base_tenv : ty Symbol.table    (* 预定义类型 *)
  val base_tenv : enventry Symbol.table (* 预定义函数 *)
end
```

图 5.4: 类型检查使用的环境

对于每一个值标识符，我们需要知道它是一个变量还是一个函数；如果是变量，它的类型是什么；如果是函数，它的参数和返回值类型是什么，等等。类型 enventry 如图5.3所示，用于保存所有这些信息；而值环境则是从符号到环境登记项的映射。

值环境将变量映射到一个告知其类型的登记项 VarEntry。当我们查看一个函数时，将得到一个含有下述信息的登记项 FunEntry：

- formals 各个形式参数的类型
- result 该数返回的结果的类型 (或 UNIT)。

对于类型检查，需要的只是 formals 和 result；我们稍后将在中间语言表示中增加转换所需要的其他的域。

`base_venv`环境含有若干预定义的函数 flush，ord，chr，size 等的绑定，这些函数的描述见附录 A。

类型检查阶段需要同时使用类型环境和值环境。

每当遇到类型变量和函数的声明时，类型检查器就会扩大这两个环境；在表达式处理期间（类型检查、中间代码生成）遇到的每一个标识符都需要查阅这两个环境。

5.3 表达式的类型检查

Semant 结构执行抽象语法的语义分析——包括类型检查。此模块包含 4 个语法树上的递归函数：

```
type venv = Env.enventry Symbol.table
type tenv = ty Symbol.table

transVar: venv * tenv * Absyn.var -> expty
transExp: venv * tenv * Absyn.exp -> expty
transDec: venv * tenv * Absyn.dec -> {venv: venv, tenv: tenv}
transTy:        tenv * Absyn.ty -> Types.ty
```

类型检查器是抽象语法树上的一个递归函数。我给它取名为 transExp 是因为稍后还将扩充这个函数，使得它不仅可以进行类型检查，而且还能将表达式转换为中间代码。transExp 的三个参数分别是值环境 venv，类型环境 tenv 和表达式。其返回值是 expty，含有转换后的表达式和该表达式的 Tiger 语言类型：

```
type expty = {exp: Translate.exp, ty: Types.ty}
```

其中，Translate.exp 是已转换为中间代码的表达式，ty 是该表达式的类型。

为了避免在这里讨论中间代码，我们定义一个虚的 Translate 模块：

```
structure Translate = struct type exp = unit end
```

并对每个 exp 值都使用 ()。在第 7 章，我们将充实 Translate.Exp 类型。

现在我们来考虑一个非常简单的加法表达式 $e_1 + e_2$。在 Tiger 中，这两个操作数都必须是整型（类型检查器必须对此进行检查），并且结果是整型（类型检查器将返回这种类型）。

在多数语言中，加法操作符是重载的：即操作符 + 既是整数加法，也是实数加法。如果两个操作数都是整数，其结果也是整数；如果两个操作数都是实数，其结果也是实数。并且，在多数语言中，如果两个操作数中有一个是整数，而另一个是实数，则整数将隐式地转换为实数。且结果为实数。当然，编译器必须使得这种转换在它生成的中间代码中显式地表现出来。

对于 Tiger 的非重载的类型，其类型检查很容易实现：

```
fun transExp(venv,
             tenv,
             Absyn.OpExp{left,oper=Absyn.PlusOp,right,pos}) =
    let val {exp=_, ty=tyleft} = transExp(venv,tenv,left)
        val {exp=_, ty=tyright} = transExp(venv,tenv,right)
     in case tyleft of Types.INT => ()
                     | _ => error pos "integer required";
        case tyright of Types.INT => ()
                     | _ => error pos "integer required";
```

```
        {exp=(), ty=Types.INT}
    end
```

尽管我们还没有写出对其他种类表达式（以及非 + 操作符）的处理，但这段代码已能很好地工作了。也正因为如此，当对 left 和 right 进行递归调用时，有可能会抛出一个 Match 异常。你可以自己来完善其他情形的处理（见 5.4.4 节）。

这样做有一点不够灵活。大部分对 transExp 的递归调用都会传递相同的 venv 和 tenv 参数，所以我们可以使用嵌套函数的方法来将它们抽取出来。检查整型类型的情况很平凡，以至于我们可以立即给出函数定义：checkInt。我们可以使用定义局部结构的方式来给经常用到的结构的名字起一个缩略名，方便我们的使用（例如 Absyn → A）。一个重构过的 transExp 如下：

```
structure A = Absyn

fun checkInt ({exp,ty},pos) = (...)

fun transExp(venv,tenv) =
  let fun trexp (A.OpExp{left,oper=A.PlusOp,right,pos}) =
              (checkInt(trexp left, pos);
               checkInt(trexp right, pos);
               {exp=(),ty=Types.INT})
        | trexp (A.RecordExp ...) ...
```

变量、下标和域的类型检查

trexp 函数递归地遍历 Absyn.exp，trvar 函数递归地遍历 Absyn.var；这两个函数都嵌套在 transExp 函数之内，并通过 transExp 的形式参数访问 venv 和 tenv。只有在极少数的情况下，trexp 会想去改变 venv，那么 trexp 必须调用 transExp 而不是 trexp。

```
   and trvar (A.SimpleVar(id,pos)) =
     (case Symbol.look(venv,id)
        of SOME(E.VarEntry{ty}) => {exp=(),ty=actual_ty ty}
         | NONE => (error pos ("undefined variable " ^ S.name id);
                    exp=(), ty=Types.INT))
     | trvar (A.FieldVar(v,id,pos)) = ...
  in trexp
end
```

对 SimpleVar 进行类型检查的 trvar 子句勾勒出了如何在环境中查找变量的绑定。如果标识符已经在环境中且绑定到了一个 VarEntry（而不是 FunEntry），那么标识符的类型就是 VarEntry 中给出的类型（见图5.4）。

有时 VarEntry 中的类型可能是一个“NAME 类型”（见程序5.4），而由 transExp 返回的所有类型都应当是“实在的”类型（即由其名字追溯到了它们最终的定义），因此一种较好的做法是让一个函数（函数名或许是 actual_ty）跳过所有的 NAME 类型。该函数的

结果是一个非 NAME 类型的 Types.ty，尽管当这个类型是一个记录或者数组时，其成员可能会含有 NAME 类型。

对于函数调用，需要在环境中查看函数的标识符来得到其登记项 FunEntry，此登记项含有一张参数类型表。函数调用表达式中的实参类型必须与参数类型表给出的类型相匹配。FunEntry 也给出了函数的结果类型，它将作为整个函数调用的类型。

每一种表达式都有它自己的类型检查规则，对于我未讲述过的其它所有情形，其规则都可以通过查阅附录 A（Tiger 语言参考手册）推导出来。

5.4 声明的类型检查

环境的创建和扩大是由程序中的声明导致的。在 Tiger 中，声明只出现在 let 表达式中。在用 transDec 翻译声明的过程中，很容易对 let 进行类型检查：

```
| trexp(A.LetExp{decs,body,pos}) =
    let val {venv=venv',tenv=tenv'} =
      transDecs(venv,tenv,decs)
     in transExp(venv',tenv') body
    end
```

其中，transExp 首先调用 beginScope() 记住两个环境 (venvtenv) 的当前“状态”；然后用新的声明调用 transDec 来扩大环境 (venvtenv); 接下来翻译函数体表达式; 最后调用 endscope() 将这两个环境恢复到它们原来的状态。

5.4.1 变量声明

从原理上讲，声明的处理相当简单：声明用一个新的绑定扩大环境，而扩大了的环境则用于后继的声明和表达式的处理。

唯一有问题的是（相互）递归的类型声明和递归的函数声明。因此，我们先从非递归声明的特殊情形开始。

例如，处理一个没有类型约束的变量声明，比如说`var x := exp`，是相当简单的。

```
fun transDec (venv,tenv,A.VarDec{name,typ=NONE,init,...}) =
  let val {exp,ty} = transExp(venv,tenv,init)
   in {tenv=tenv,
       venv=S.enter(venv,name,E.VarEntry{ty=ty})}
  end
```

还有什么情形能比这更简单？实际上，如果出现了 typ，如在下面的声明中：

```
var x : type-id := exp
```

就会需要检查这个类型约束和进行初始化的表达式是否兼容。另外，类型为 NIL 的初始化表达式还必须受 RECORD 类型的约束。

5.4.2 类型声明

非递归类型声明的处理也不太难：

```
| transDec (venv,tenv,A.TypeDec[{name,ty}]) =
    {venv=venv,
     tenv=S.enter(tenv,name,transTy(tenv,ty))}
```

函数 transTy 将抽象语法中的类型表达式（Absyn.ty）翻译为要放入到环境中去的转换后的类型描述（Types.ty）。这种翻译是在结构 Absyn.ty 上递归进行的，它将 Absyn.RecordTy 转换成 Types.RECORD，等等。在转换过程中，transTy 只需查看它在类型环境 tenv 中找到的每一个符号。

上面给出的程序中的模式 [{name,ty}] 的通用性不是很好，因为它只处理长度为 1 的类型声明列表，即单个相互递归类型声明的列表。请读者推广这段代码使之适应任意长度的类型声明列表。

5.4.3 函数声明

函数声明要稍微繁琐点：

```
| transDec(venv,tenv,
          A.FunctionDec[{name,params,body,pos,
                         result=SOME(rt,pos)}]) =
    let val SOME(result_ty) = S.look(tenv,rt)
      fun transparam{name,typ,pos} =
        case S.look(tenv,typ)
          of SOME t => {name=name,ty=t}
      val params' = map transparam params
      val venv'  = S.enter(venv,name,
                           E.FunEntry{formals=map #ty params',
                                      result=result_ty})
      fun enterparam ({name,ty},venv) = S.enter(venv,name,
                                                E.VarEntry{access=(),ty=ty})
      val venv'' = fold enterparam params' venv'
     in transExp(venv'',tenv) body;
        {venv=venv',tenv=tenv}
    end
```

这是一种已剥离得非常彻底的实现：它只处理单个函数的情况，不处理递归函数，只处理有返回结果的函数（是函数，不是过程），不处理诸如未声明的类型标识符之类的程序错误，等等；并且，它不检查函数体表达式的类型是否与声明的结果类型相匹配。

那么，它做了些什么？考虑下面的 Tiger 声明：

```
function f(a: ta, b: tb) : rt = body.
```

transDec 首先在类型环境中查找结果类型标识符 rt。然后调用局部定义的函数 transparam，此函数遍历形式参数列表，并返回一个元素类型为元组的列表，$(a, t_a), (b, t_b)$，其中 t_a 是在环境中查找 ta 所得到的 NAME 类型。现在 transDec 就有了足够的信息来为这个函数构造一个 FunEntry，然后将构造好的 FunEntry 添加到值环境中，并产生一个新的环境 venv'。

接下来，形式参数（作为 VarEntry）被添加到值环境中 venv' 中，并产生 venv''；而这个环境则被（transexp 函数）用来处理函数体。最终，venv'' 被丢弃，然后 (venv',tenv) 就是结果：这个环境将被用来处理那些允许调用函数 f 的表达式。

5.4.4 递归声明

上面的实现不能用于递归类型和递归函数的声明，因为在这两种声明中会遇到未定义的类型或函数的标识符（对于递归记录类型，未定义的类型出现在 transTy 中，或对于递归函数未定义的函数出现在 transExp(body) 中）。

对于一组相互递归的对象（类型或函数）$t_1, ..., t_n$，其解决方法是首先将所有这些对象的 "头" 放入到环境中，得到一个环境 e_1。然后在环境 e_1 下处理所有这些对象的 "体"。在处理这些体的期间，将会需要查找一些最近定义的名字，但是事实上它们已经在环境中——尽管其中有一些可能只是有头而没有体。

那么，"头" 指的是什么？对于如下的类型声明：

```
type list = {first: int, rest: list)
```

头可以认为是 type list =。

为了将这个头送入环境 tenv，我们可以使用一个其绑定为空（ty option）的 NAME 类型：

```
tenv' = S.enter(tenv,name,Types.NAME(name,ref NONE))
```

现在，我们可以根据类型声明的 "体"，即记录表达式`{first: int, rest: list}`来调用 transTy。我们给到 transTy 的环境是 tenv'。

重要的一点是，只要到达任何 NAME 类型，transTy 就应停止。例如，如果 transTy 像 actual_ty 一样企图顺着绑定到标识符 list 的 NAME 类型一直查找，它找到的（在这个例子的情况下）就只有 NONE——此时它的类型肯定还未定义完毕。这个 NONE 只能在整个 {first:int,rest:list} 都被翻译之后才能用一个有效的类型来替代。

然后，transTy 返回的类型可以赋给 NAME 构造器中的引用变量。现在我们有了一个完整的类型环境，在这个环境中，调用 actual_ty 不会有问题。

ref 变量的赋值（在 NAME 类型描述符中）意味着 Translate 不是一个 "纯函数式" 程序。任何对副作用的使用都会增加编写程序和理解程序的难度，但在我们这个情况中有限制的使用副作用还是可以接受的。

在一组相互递归的类型声明中，每一个递归都必须通过记录或数组声明传递一个类型；下面这个声明

```
type a = b
type b = d
type c = a
type d = a
```

含有一个非法的递归 $a \to b \to d \to a$。类型检查器应当检测出这种非法的递归。

处理相互递归的函数与处理递归类型类似。第一遍收集每一个函数的头的信息（函数名、形式参数列表、返回值类型），但不处理函数体。在这一遍中，需要的是形式参数的类型，而不是它们的名字（在函数之外见不到它们的名字）。

第二遍处理相互递归声明中的所有函数的函数体，此时使用的环境是已用所有函数头扩大了的环境。对于每一个函数体，再次处理它的形式参数列表，这一次则参数作为 VarEntry 加入到值环境中。

5.5 程序设计：类型检查

为你的编译器编写类型检查阶段的程序，即一个包含下面的函数的模块 Semantic：

```
transProg : Absyn.exp -> unit
```

它对抽象语法树进行类型检查，并生成适当的关于类型不匹配或未声明的标识符的报错信息。

实现本章描述的模块 Env。构造一个模块 Main 调用语法分析器来生成 Absyn.exp，然后对这个表达式调用 transProg。

你必须完全按照图**??**描述的方式使用 Absyn 接口，但可以自行决定采纳或者忽略本章给出的关于 Semant 模块内部结构的建议。

你会需要用你写的语法分析器来生成抽象语法树。此外，在 $TIGER/chap5 中还包含了下面支持文件：

types.sml 表述了 Tiger 语言的数据类型。

以及其他些与以前相同的文件。必要时要修改第 4 章练习中的 sources.cm 文件。

a. 实现一个简单的类型检查器和声明处理器，这个声明处理器不处理递归数或递归数据类型（不必处理向前引用的函数或类型）。类型检查器不检查每一个 break 语句是否位于 for 语句或 while 语句之内。

b. 扩充你的简单类型检查器，使能处理递归的（和相互递归的）函数、（相互）递归的类型声明，并保证 break 语句的正确嵌套。

5.6 习题

5.1 改进程序5.1的哈希表实现：

a. 当散列链的平均长度大于 2 时将散列数组增大一倍（因此，现在 table 是 ref(array)。为了将数组增大一倍，在分配一个更大的数组时，要重新散列原数组中的内容然后再释

放原数组。

b. 给 insert 和 lookup 增加一个参数以允许使用多个表。

c. 将 table 类型的表示隐藏在一个抽象模块中使得 table 的使用者不会直接修改该数据结构（只通过 insert，lookup 和 pop 操作进行修改）。

*****5.2** 在很多应用中，我们会想要作用于环境的 + 操作符不仅仅是加入一个新的绑定；即不仅仅是 $\sigma' = \sigma + \{a \mapsto \tau\}$，而是 $\sigma' = \sigma_1 + \sigma_2$，其中 σ_1 和 σ_2，是任意的环境（可以是重叠的，在这种情况下，σ_2 中的绑定优先）。

我们希望有一种能高效实现这种环境“加法”的算法和数据结构。平衡树可以高效地实现 o+|a|(时间为 log(N)，其中的大小)，但在和的大小都是 N 时，计算 o+o, 却需要 O(N)。

为了将这个问题抽象化，要解一般的不相交整数集合的并运算问题。此问题的输入是如下形式的命令集合：

$$
\begin{aligned}
&s_1 = \{4\} && \text{(定义单元素集合)}\\
&s_2 = \{7\}\\
&s_3 = s_1 \cup s_2 && \text{(非破坏性联合)}\\
&6 \overset{?}{\in} s_3 && \text{(是否为成员的测试)}\\
&s_4 = s_1 \cup s_3\\
&s_5 = \{9\}\\
&s_6 = s_4 \cup s_5\\
&7 \overset{?}{\in} s_2
\end{aligned}
$$

高效的算法是这样一种算法，它可以处理 N 条命令组成的输入，并且回答任意成员关系查询的时间不超过 o(N)。

***a.** 实现一个算法，该算法对于典型集合并运算 a-bUc，在 b 比 c 小很多的情况下仍是高效的 [Brown 和 Tarjan 1979]。

*****b.** 设计一个即使在最坏情况下也是高效的算法，或证明不可能有这样的算法（参见 Lipton 等 [1997] 关于受限模型下界的论述）。

***5.3** Tiger 语言定义要求，类型定义的每一个递归都必须经过一个记录或数组。但是，如果编译器忘记了检查这类错误，也不会出现特别糟糕的问题。解释这是为什么？

第六章 活动记录

第七章 翻译成中间代码

```
signature TREE =
sig

datatype exp    = CONST of int
                | NAME of Temp.label
                | TEMP of Temp.temp
                | BINOP of binop * exp * exp
                | MEM of exp
                | CALL of exp * exp list
                | ESEQ of stm * exp

     and stm    = MOVE of exp * exp
                | EXP of exp
                | JUMP of exp * Temp.label list
                | CJUMP of relop * exp * exp * Temp.label * Temp.label
                | SEQ of stm * stm
                | LABEL of Temp.label

     and binop = PLUS | MINUS | MUL | DIV
                | AND | OR | LSHIFT | RSHIFT | ARSHIFT | XOR

     and relop = EQ | NE | LT | GT | LE | GE
                | ULT | ULE | UGT | UGE
end
```

第 八 章　基本块和轨迹

第九章 指令选择

第十章 活跃分析

第 十一 章　寄存器分配

第 十二 章　整合为一体

第二部分

高级主题

第 十三 章 垃圾收集

第 十四 章　面向对象的语言

第 十五 章　函数式程序设计语言

第 十六 章　多态类型

第 十七 章　数据流分析

第 十八 章　循环优化

第 十九 章 静态单赋值形式

第 二十 章　流水线化和调度

第二十一章 存储层次

存储器（mem-o-ry）：能够写入和存放信息的设备，当需要时可以从中提取信息。
层次（hi-er-ar-chy）：一个分层或分级的系列。

韦氏字典

理想的*随机存取存储器*（random access memory，RAM）有 N 个以整数作为索引的字，这样，它的每一个字都可以通过整数地址以同样快的速度来存取。硬件设计者既能够构建容量大但速度慢的存储器，也能够构建容量小但速度快的存储器，但是构建既满足容量大又满足速度快的存储器，其价格却高得惊人。另外，提高存储器访问速度的一个办法是使其靠近处理器。但是，在这个问题上无论花费多少钱，大的存储器中总会有一些部分远离处理器。

将一个容量小速度快的*高速缓冲存储器*（cache）与一个容量大速度慢的*主存储器*组合在一起，就几乎能够和一个容量大速度快的存储器相媲美；程序将它频繁使用的数据放在 cache 中，很少使用的数据放在主存储器中，当程序进入需要频繁使用数据 x 的某个阶段时，就将 x 从慢的主存储器中移至快的 cache 存储器中。

由程序员管理多个存储器相当不方便，因此硬件会自动地进行管理。当处理器需要访问在地址 x 处的数据时，处理器首先在 cache 中查找，并且我们希望通常能够在 cache 中找到该数据。如果发生 cache *缺失*（cache miss），即 x 不在 cache 中，则处理器会将 x 从主存储器中取出，并将 x 的一个副本放入到 cache 中，这样，下一次对 x 的引用就会 cache *命中*（cache hit）。将 x 放入到 cache 中意味着需要将另一个数据 y 从 cache 中移出，以便为 x 腾出空间，当然这样便导致了以后访问 y 时又会发生 cache 缺失。

21.1 cache 的组织结构

直接映射的（direct-mapped）cache 按如下方式来组织，以实现快速的存储器管理。cache 被分成 2^m 个块，每个块包含 2^l 个字，每个字 2^w 字节；因此，这个 cache 总共包含 2^{w+l+m} 字节，并且排列成一个数组 $Data[block][word][byte]$。每个块都是主存储器中某个数据的一个副本，并且还存在着一个 tag 数组，用以指明当前内容来自主存储器的什么位置。一般地，字大小 2^w 是 4 字节，块大小 2^{w+l} 是 32 字节，cache 大小可以小到 8KB，也可以大到 2MB。

tag	key	$word$	$byte$
$(n-(m+l+w))$ 位	m 位	l	w

给定地址 x，cache 部件必须能够查找出 x 是否在 cache 中。地址 x 由 n 位组成：

$x_{n-1}x_{n-2}\cdots x_2x_1x_0$（见图 21-1）。在直接映射的 cache 组织中，我们用中间的 m 位作为键值，即 $key = x_{w+l+m-1}x_{w+l+m-2}\cdots x_{w+l}$，并将 x 中的数据保存在 $Data[key]$ 中。