现代编译原理

ML 语言描述

[美] Andrew W.Appel　Maia Ginsburg 著

2022

目录

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

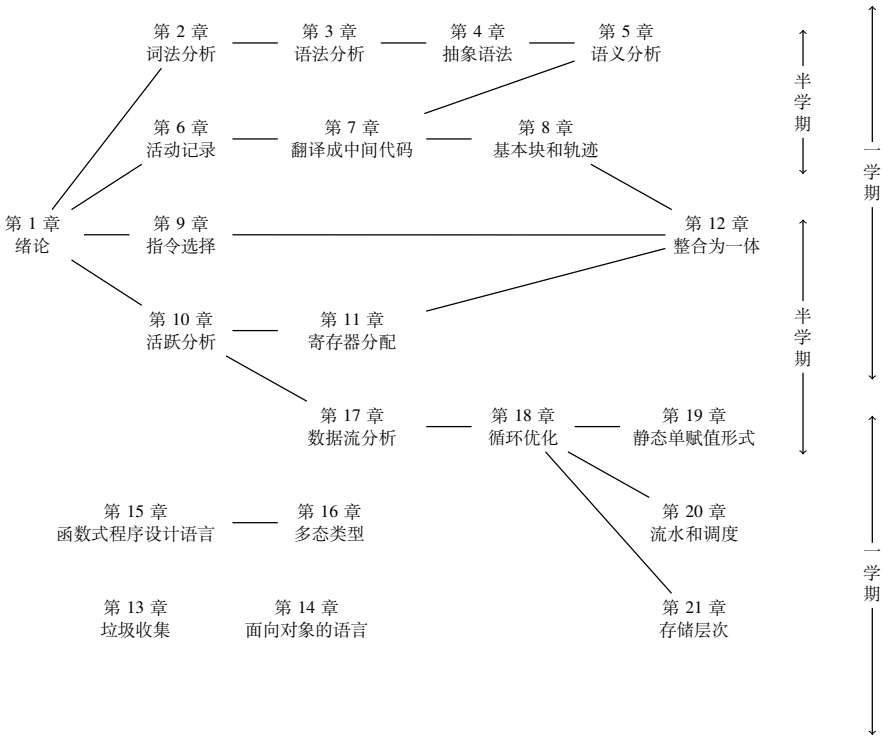


图 1: 内容结构图

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。

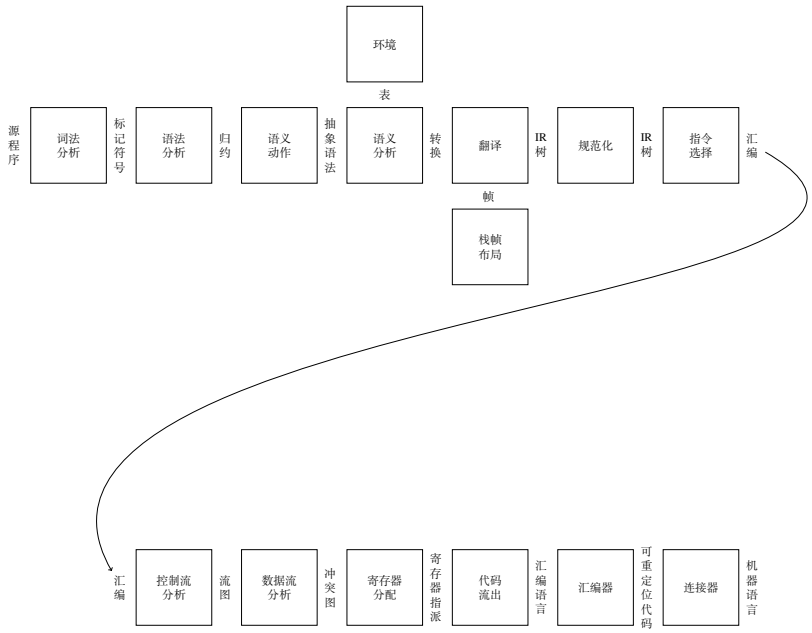


图 1.1: 编译器的各个阶段及其之间的接口

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1.1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

表 1.1: 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

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1.2中给出。

Stm	$\rightarrow$	Stm ; Stm	(CompoundStm)
Stm	$\rightarrow$	id := Exp	(AssignStm)
Stm	$\rightarrow$	print ($ExpList$)	(PrintStm)
Exp	$\rightarrow$	id	(IdExp)
Exp	$\rightarrow$	num	(NumExp)
Exp	$\rightarrow$	Exp $Binop$ Exp	(OpExp)
Exp	$\rightarrow$	(Stm , Exp)	(EseqExp)
$ExpList$	$\rightarrow$	Exp , $ExpList$	(PairExpList)
$ExpList$	$\rightarrow$	Exp	(LastExpList)
$Binop$	$\rightarrow$	$+$	(Plus)
$Binop$	$\rightarrow$	$-$	(Minus)
$Binop$	$\rightarrow$	$\times$	(Times)
$Binop$	$\rightarrow$	$/$	(Div)

文法 1.2: 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。操作符表达式 e_1 op e_2 表示先计算 e_1 再计算 e_2，然后按给定的二元操作符计算表达式结果。表达

式序列 (s, e) 的行为类似于 C 语言中的逗号操作符，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。

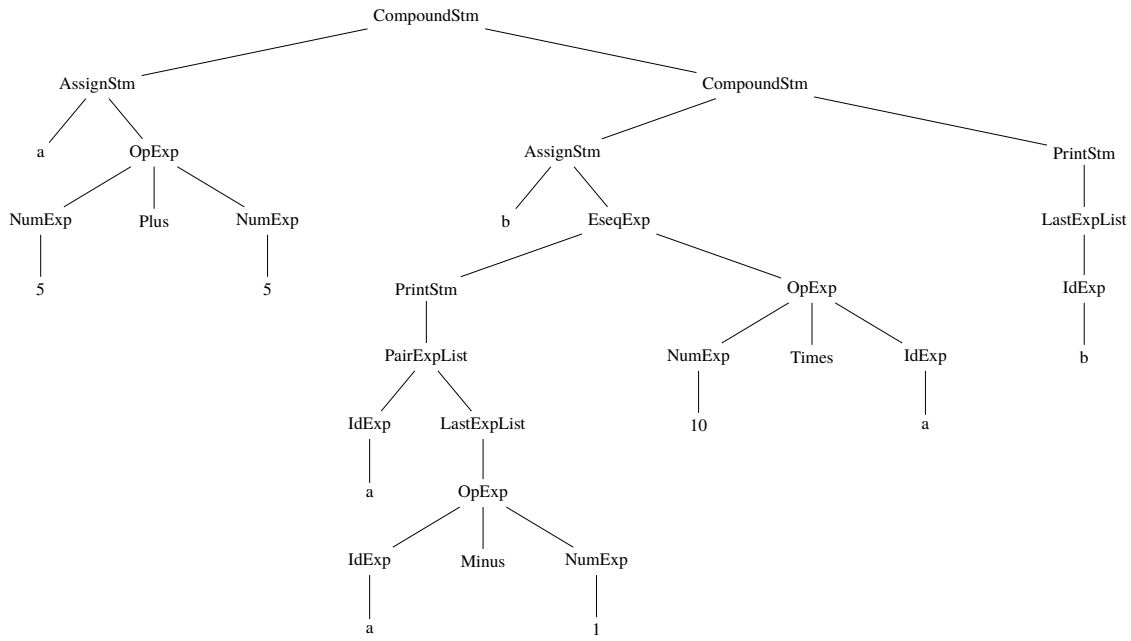
例如，执行下面这段程序：

```
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1.2给出了这个程序的树表示，其中结点都用文法1.2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 1.2: 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1.1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1.2各项右部的括号内。

程序 1.1: 直线式程序的表示

```
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
```

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：

   ```
   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。

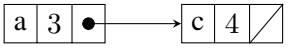
编写没有副作用（即更新变量和数据结构的赋值语句）的解释器是理解指称语义（denotational semantic）和属性文法（attribute grammar）的好方法，后两者都是描述程序设计语言做什么的方法。对编写编译器而言，它也常常是很有用的技术，因为编译器也需要知道程序设计语言做的是什么。

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。

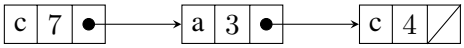
例如，表 t_1 中 a 映射到了 3，c 映射到了 4，我们将 t_1 写成 $\{a \mapsto 3, c \mapsto 4\}$ 这样的数学符号，还可以将 t_1 写成链表的形式 

，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
```

[1] pair

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。

1.5 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

2.3 有限自动机

2.4 非确定性有限自动机

2.4.1 将正则表达式转换为 NFA

2.4.2 将 NFA 转换为 DFA

2.5 ML-Lex：词法分析器的生成器

2.6 程序设计：词法分析

2.7 推荐阅读

2.8 习题