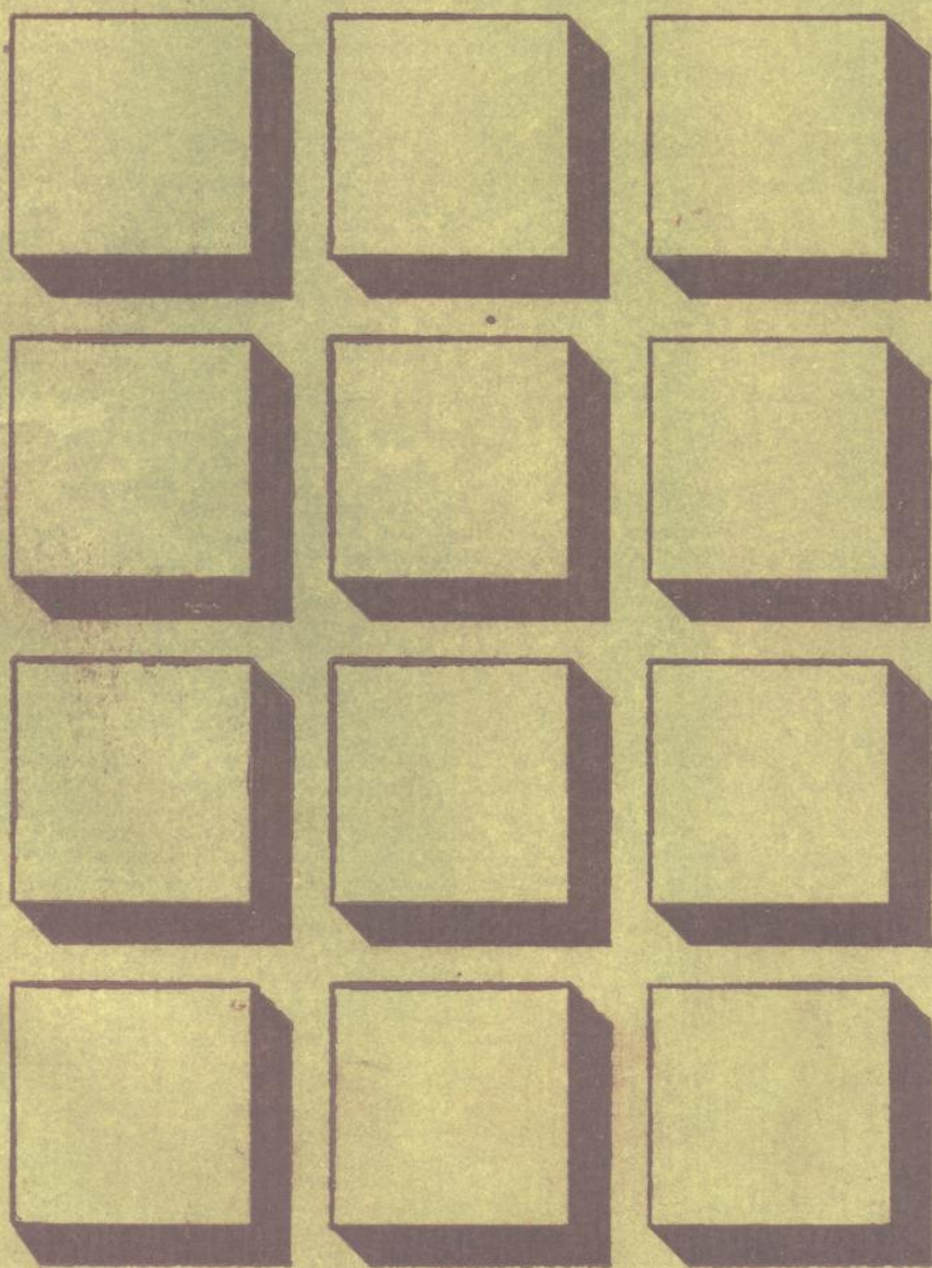
陈意云　马万里　编译　● 中国科学技术大学出版社

编译原理和技术



编译原理和技术

陈意云　马万里　　编译

中国科学技术大学出版社

1988 · 合肥

内 容 简 介

本书介绍了编译器构造的一般原理和基本实现方法，反映了直至八十年代前期的一些最重要的成果．其内容包括词法分析、语法分析、中间代码生成、优化和目标代码生成等．作为原理性书，旨在介绍基本的理论和方法，不偏向于某种源语言或目标机器．全书内容充实，图文并茂，各章节之间循序渐近，并在各章之后附有习题，对读者有一定的借鉴和启迪作用。

本书可作为高等院校计算机科学专业的教材，也可作为有关软件工程技术人员的参考书。

编 译 原 理 和 技 术

陈意云　马万里　编　译

责任编辑：李华友　封面设计：王瑞荣

中国科学技术大学出版社出版
（安徽省合肥市金寨路96号）
中国科学技术大学印刷厂印刷
安徽省新华书店发行

开本：787×1092/16　印张：29.25 字数：708 千字
1989年12月第1版　1989年12月第1次印刷
印数：3000
ISBN 7-312-00124-6/TP·7　定价：5.80元

前　　言

本书是在V·Aho，Ravi Scthi 和Jeffrey D·man的Compilers Principles，Techniqnes and Tools（1986年）的基础上作了部分增删而改编的．原著不仅包含了最经典、最广泛应用的基本编译技术，还反映了直至八十年代前期的一些最重要的新成果，这是本书区别于其它有关这方面著作的显著特点，因此国外很多大学都以此为教材．编者有志于编写一本适合国内大学的编译课程教材，希望本书的出版会对“编译原理和技术”的教学产生积极的影响，这也是编者最大的愿望．

虽然只有少数人要去构造或维护程序设计语言的编译器，但是读者可以把本书讨论的概念和技术应用于一般的软件设计之中．例如，建立词法分析器的串匹配技术已用于正文编辑器、信息检索系统和模式识别程序；上下文无关文法和语法制导定义已用于创建许多诸如排版、绘图系统的小语言；代码优化技术已用于程序验证器和从非结构化的程序产生结构化程序的编程之中．无疑，读者会从本书中学到许多新的软件设计方法，尤其是通过自己亲手编写一个小语言（比如PL/0）的编译器，掌握一些软件设计技巧，肯定会受益非浅．

本书注重讨论在设计语言翻译器时普遍遇到的问题，而不偏向某种源语言或目标机器．

第一章介绍编译器的基本结构，它是本书其余部分的基础；第二章涉及词法分析器、正规式、有限自动机和扫描器生成器．该章内容被广泛用于正文处理；第三章主要讲述了各种分析技术，这些技术包括从适于手工实现的递归下降方法到已用于分析器生成器的更为精致的LR技术；第四章介绍语法制导翻译的主要概念．本书其余各章都会应用到这些概念来描述和实现翻译；第五章提出了完成静态语义检查的主要思想，并详细讨论了类型检查和合一问题；第六章讨论了支持程序运行环境的存储组织问题；第七章从讨论中间语言开始，然后说明了如何把一般的程序设计语言结构翻译成中间代码；第八章涉及目标代码生成．包括简单的代码生成方法，产生表达式的优化代码的方法，同时也讨论了窥孔优化和代码生成器；第九章是代码优化的综合论述，详细探讨数据流分析和全局优化的主要方针；最后一章讨论实现编译器的一些编程问题．软件工程和测试在构造编译器时显得尤其重要．

编者在中国科学技术大学讲述《编译原理和技术》课程时，就以本书内容为教材．广大同学普遍反映良好，认为通过这门课的学习不仅提高了他们的编程技巧，掌握了软件设计新技术，而且对计算机系统软件有了一个比较清晰的了解，对今后进一步的学习和研究起到了登堂入室的作用．当然由于编者水平有限，书中难免还存在一些缺点和错误，恳请广大读者批评指正．

B1

目　录

第一章　引论 …………………………………………（ 1 ）

1.1　翻译和解释 …………………………………………（ 1 ）
1.2　源程序的分析 …………………………………………（ 3 ）
1.3　编译的阶段 …………………………………………（ 6 ）
1.4　阶段的分组 …………………………………………（ 10 ）
1.5　编译器的伙伴 …………………………………………（ 11 ）
1.6　构造编译器的工具 …………………………………………（ 14 ）

第二章　词法分析 …………………………………………（ 16 ）

2.1　词法分析器的作用 …………………………………………（ 16 ）
2.2　输入缓冲区 …………………………………………（ 19 ）
2.3　记号的说明 …………………………………………（ 22 ）
2.4　记号的识别 …………………………………………（ 26 ）
2.5　词法分析器的说明语言 …………………………………………（ 32 ）
2.46 有限自动机 …………………………………………（ 36 ）
2.7　从正规式到NFA …………………………………………（ 42 ）
2.8　DFA 的化简 …………………………………………（ 46 ）

第三章　语法分析 …………………………………………（ 52 ）

3.1　分析器的作用 …………………………………………（ 52 ）
3.2　上下文无关文法 …………………………………………（ 56 ）
3.3　语言和文法 …………………………………………（ 60 ）
3.4　自上而下分析 …………………………………………（ 69 ）
3.5　自下而上分析 …………………………………………（ 81 ）
3.6　算符优先分析 …………………………………………（ 87 ）
3.7　LR分析器 …………………………………………（ 95 ）
3.8　二义文法的应用 …………………………………………（ 119 ）
3.9　分析器的生成器 …………………………………………（ 128 ）

试读结束：需要全本请在线购买： www.ertongbook.com

第四章 语法制导的翻译 …………………………………………………… (142)

4.1 语法制导的定义 ………………………………………………………… (142)
4.2 语法树的构造 …………………………………………………………… (148)
4.3 L-属性定义 ……………………………………………………………… (152)
4.4 S-属性的自下而上计算 ……………………………………………… (156)
4.5 自上而下翻译 …………………………………………………………… (159)
4.6 继承属性的自下而上计算 …………………………………………… (164)
4.7 递归计算 ………………………………………………………………… (171)
4.8 编译时属性值的空间指派 …………………………………………… (173)
4.9 编译器构造时的空间指派 …………………………………………… (176)
4.10 语法制导定义的分析 ………………………………………………… (180)

第五章 类型检查 ……………………………………………………………… (189)

5.1 类型体制 ………………………………………………………………… (190)
5.2 简单类型检查器的说明 ……………………………………………… (193)
5.3 类型表达式的等价 …………………………………………………… (196)
5.4 类型转换 ………………………………………………………………… (201)
5.5 函数和算符的超载 …………………………………………………… (203)
5.6 多型函数 ………………………………………………………………… (206)
5.7 合一算法 ………………………………………………………………… (215)

第六章 运行环境 ……………………………………………………………… (224)

6.1 源语言问题 ……………………………………………………………… (224)
6.2 存储组织 ………………………………………………………………… (229)
6.3 存储分配策略 …………………………………………………………… (233)
6.4 访问非局部名字 ………………………………………………………… (241)
6.5 参数传递 ………………………………………………………………… (251)
6.6 符号表 …………………………………………………………………… (255)
6.7 动态存储分配的语言功能 …………………………………………… (262)
6.8 动态存储分配技术 ……………………………………………………… (265)

第七章 中间代码生成 ………………………………………………………… (273)

7.1 中间语言 ………………………………………………………………… (273)

7.2 声明 …………………………………………………………………… (280)
7.3 赋值语句 ……………………………………………………………… (284)
7.4 布尔表达式 …………………………………………………………… (292)
7.5 分情况语句 …………………………………………………………… (298)
7.6 回填 …………………………………………………………………… (301)
7.7 过程调用 ……………………………………………………………… (306)

第八章 代码生成 ………………………………………………………… (311)

8.1 代码生成器设计中的问题 …………………………………………… (311)
8.2 目标机器 ……………………………………………………………… (315)
8.3 运行时的存储管理 …………………………………………………… (317)
8.4 基本块和流图 ………………………………………………………… (323)
8.5 下次引用信息 ………………………………………………………… (327)
8.6 一个简单的代码生成器 ……………………………………………… (328)
8.7 寄存器分配和指派 …………………………………………………… (333)
8.8 基本块的 DAG 表示 ………………………………………………… (336)
8.9 窥孔优化 ……………………………………………………………… (341)
8.10 从 DAG 生成代码 …………………………………………………… (344)
8.11 动态规划的代码生成算法 …………………………………………… (353)
8.12 代码生成器的生成器 ………………………………………………… (356)

第九章 代码优化 ………………………………………………………… (366)

9.1 引言 …………………………………………………………………… (366)
9.2 优化的主要种类 ……………………………………………………… (370)
9.3 基本块的优化 ………………………………………………………… (375)
9.4 流图中的循环 ………………………………………………………… (377)
9.5 全局数据流分析介绍 ………………………………………………… (381)
9.6 数据流方程的迭代求解 ……………………………………………… (392)
9.7 代码改进变换 ………………………………………………………… (399)
9.8 别名的处理 …………………………………………………………… (410)
9.9 结构化流图的数据流分析 …………………………………………… (418)
9.10 快速数据流算法 ……………………………………………………… (426)
9.11 类型估计 ……………………………………………………………… (432)
9.12 优化代码的符号调试 ………………………………………………… (438)

第十章 如何编写编译器 ……………………………………………（449）

10.1 编译器的规划……………………………………………………（449）
10.2 开发编译器的途径………………………………………………（450）
10.3 编译器的开发环境………………………………………………（453）
10.4 测试和维护………………………………………………………（454）
10.5 几个编译器简介…………………………………………………（455）

第一章 引 论

从理论上说，构造专用计算机来直接执行某种高级语言写的程序是可能的，但是，目前的机器能直接执行的是非常低级的语言，即机器语言．那么，一个基本的问题是：高级语言最终是怎样在计算机上执行的．

术语编译代表从面向人的源语言表示的算法到面向硬件的目标语言表示的算法的一个等价变换。本章将通过描述编译器的各个组成部分，编译器完成它工作的环境和使得编译器易于构造的软件工具来介绍编译这个课题．该课题涉及程序设计语言、机器结构、语言理论、算法和软件工程等方面．

§ 1.1 翻译和解释

程序设计语言是构造有限的计算（算法）的形式描述工具，每个计算包含一系列操作，这些操作把给定的初始状态转变为某个终止状态．程序设计语言本质上提供了三种成分来描述这样的计算：

1.数据类型、对象和值，以及定义在它们上面的运算．

2.确定这些运算的计算顺序的规则．

3.确定程序静态结构规则．

这些成分一起形成一种抽象，借助这种抽象，我们可以用这种语言表达算法．

在计算过程中的某一时刻，所存在的所有对象构成这个计算在那个时刻的状态 s ．用这种语言所能表达的所有状态的集合 S，称为这种语言的状态空间．算法是一个(部分定义的)函数 $f: S \to S$，它把初始状态转变为终止状态．

图1.1 说明状态的概念，图1.1(a) 是一段Pascal 程序，其中的 i 和 j 是整型变量．在这段程序执行前，i 和 j 的值构成初始状态，执行停止后，它们的值构成终止状态．图1.1(b) 说明从特定的初始状态开始，执行这段程序所引起的状态变化．

```
While i ≠ j do
   if i > j then i := i - j else j := j - i ;
```

a) 一个算法

```
Initial : i = 36  j = 24
          i = 12  j = 24
Final :   i = 12  j = 12
```

b) 一个状态序列

图 1.1 算法和状态

令 f 是某个算法 A 的状态变换函数，在把此算法翻译成另一种语言的算法时，如果

要保持A的含义不变，那么后一个算法的状态转换函数f'必须在某种程度上和f“一致”．因为目标语言的状态空间S'可以和源语言的状态空间不一样,因此首先要确定函数M，它把状态$s \in S$映射到S'的子集$M(s)$．于是，如果对于所有允许的初始状态$s \in S$，$f'(M(s))$是$M(f(s))$的子集，那么函数f'就保持了f的含议．

例如，考虑有一个累加器和两个数据单元（分别称为I和J）的简单计算机．假定M把图1.1(a)算法的状态映射到此机器的状态集，其中I含变量i的值，J含变量j的值，累加器含任意值．图1.2(a)给出图1.1(a)到这个机器的变换，图1.2(b)给出部分状态序列．显然，图1.2(a)算法保持了图1.1(a)算法的含义．

```
LOOP  LOAD   I
      SUB    J
      JZERO  EXIT
      JNEG   SUBI
      STORE  I
      JUMP   LOOP
SUBI  LOAD   J
      SUB    I
      STORE  J
      JUMP   LOOP
EXIT
```

a）一个算法

```
Initial:  I = 36   J = 24   ACC = ?
          I = 36   J = 24   ACC = 36
          I = 36   J = 24   ACC = 12
          .        .
          .        .        :
          .        .        .
Final:    I = 12   J = 12   ACC = 0
```

b）对应图1.1(b)的状态序列

图1.2　图1.1的一个翻译

在确定图1.1(b)的状态序列时，我们仅用了Pascal语言定义给出的概念．对任何一种程序设计语言PL，可以定义一种抽象机，PL的运算、数据结构和控制结构是这种机器的存储元素和指令．在这样的机器上执行用PL写的算法称为解释．通常，这种抽象机是由软件实现的，即程序设计语言和机器语言之间的差距由软件来填补，这样的软件叫做解释器．

编译器以另一种方式处理这个差距，它通过翻译序列(SL, L_1)，(L_1, L_2)，…，(L_k, TL)来得到和源程序等价的TL语言程序，这里SL是源语言，TL是目标语言，$L_i(1 \leq i \leq k)$叫做中间语言得到的目标语言程序由机器硬件解释执行，或由其它软件进一步处理．在分解编译的任务时，中间语言是概念上的工具，在设计编译器时要决定哪个中间语言（如果有的话）作为具体的正文或数据结构的真正出现．

能够完成一种语言到另一种语言变换的软件称为翻译器。编译器是其中一类，它的特点是目标语言比源语言低级。

§ 1.2 源程序的分析

任何编译都可分两大部分:

1.**分析** 揭示源程序的结构和基本数据，决定它们的含义，建立源程序的中间表示.

2.**综合** 从源程序的中间表示建立和源程序等价的目标程序. 本节非形式地说明分析，下节指述编译器目标代码的生成方法.

编译的分析包括三个阶段:

1.**线性分析** 从左到右地读构成源程序的字符流，并组成一个个记号，记号是有确定含义的字符序列.

2.**层次分析** 把记号按语言的语法结构层次地分组.

3.**语义分析** 完成一些检查，以保证程序的各部分能有意义地结合在一起.

1.2.1 词法分析

编译器的线性分析叫做词法分析或扫描。例如，在词法分析时，赋值语句

position: = initial + rate*60

的字符将被组成下列记号:

1.标识符position
2.赋值号: =
3.标识符initial
4.加号
5.标识符rate
6.乘号
7.数60

分隔记号的空格通常在词法分析时被删去.

1.2.2 语法分析

层次分析叫做语法分析或简称分析. 它把源程序的记号分组形成语法短语，编译器使用这些短语去完成综合. 源程序的语法短语常用分析树表示，图1.3便是一例.

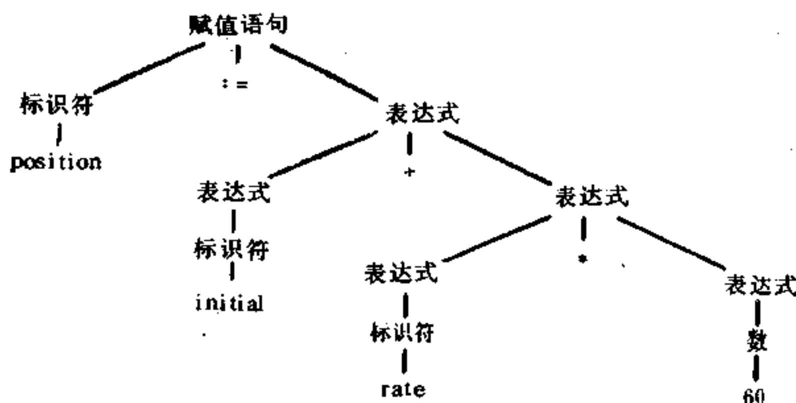


图 1.3 position: = initial +rate*60的分析树

试读结束：需要全本请在线购买： www.ertongbook.com

在表达式initial+rate* 60中，短语rate* 60是一个逻辑单位，因为按一般的算术表达达式的习惯，乘比加先完成；由于表达式initial+rate 后面是乘号，所以它不能组成一个短语.

程序的层次结构通常由递归的规则表示，例如，可以用如下的规则作为表达式定义的一部分：

1.任何一个标识符是表达式；

2.任何一个数是表达式；

3.如果$e1$和$e2$都是表达式，那么

$e1+e2$

$e1*e2$

$(e1)$

也都是表达式．规则(1)和(2)是（非递归）基本规则，而规则(3)是把算符作用于其它表达式来定义表达式的，于是，根据规则(1)，initial 和rate 都是表达式；由(2)，60是表达式；由(3)，首先可推出rate * 60是表达式，然后initial + rate * 60是表达式.

同样地，许多语言用类似如下的规则递归地定义语句：

1.如果identifier 是标识符，expression 是表达式，那么

inentifier : = expression

是语句.

2.如果expression 是表达式，statement 是语句，那么

while(expression)**do** statement

if(expression)**then** statement

也都是语句.

词法和语法分析的划分是有点任意的，通常选择可以简化整个分析任务的划分．决定这种划分的一个因素是源语言的结构是否允许递归，词法结构一般不需要递归，而语法结构通常需要．上下文无关文法是递归规则的一种形式化，它可以用来指导语法分析，在第三章中将进一步研究它.

例如，递归无需用于识别标识符．标识符是由字母开头的字母和数字串．识别标识符时，我们简单地扫描输入串，直到发现一个既不是字母，也不是数字的字符为止，然后把已看见的所有字母和数字组成一个标识符记号，并把这些字符记录在符号表中，再把它们从输入串中移开，下一个记号的处理便可以开始.

这种线性扫描不足以分析表达式和语句．比方说，不在输入中加上某种层次或嵌套的结构，我们不能正确地匹配表达式的括号或语句的**begin** 和**end** .

图1.3 的分析树描绘了输入的语法结构，这种语法结构更常见的内部表示由图1.4(a)的语法树给出。语法树是分析树的浓缩表示,其中算符作为内部结点出现,它的运算对象作为它的后代。图1.4(a) 这样的树结构在4.2 节讨论．在第四章的语法制导翻译中,我们将详细讨论编译器如何利用输入所含的层次结构来产生输出.

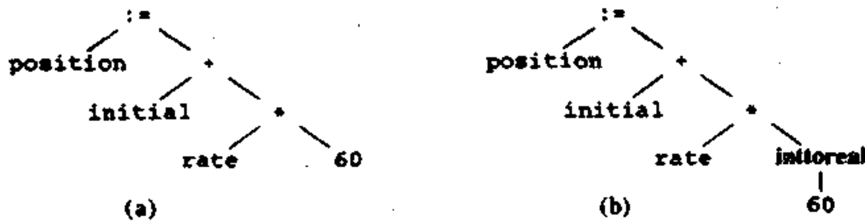


图 1.4 语义分析插入整型到实型的转换

1.2.3 语义分析

语义分析阶段检查源程序的语义错误，并为以后的代码生成阶段收集类型信息，它使用语法分析阶段确定的层次结构来标识表达式和语句的算符和运算对象.

语义分析的一个重要部分是类型检查，编译器检查每个算符的运算对象，看它们的类型是否合法. 例如，当实数作为数组的下标时，许多语言的定义要求编译器报告错误；当然也有些语言允许运算对象类型隐式转换，如二元的算术算符作用于一个整数和一个实数时. 类型检查和语义分析在第五章讨论.

例1.1 在机器内部，整数的二进制表示和实数的二进制表示是有区别的，即使它们有相同的值. 例如，在图1.4 中，所有的标识符声明为实型，由60本身可知它是整数对图1.4(a)进行类型检查会发现 * 作用于一个实型变量rate 和一个整数60，通常的办法是把整数转变为实数，可以建立一个额外的算符结点**inttoreal** （见图1.4(b)），它显式地把整数转变为实数. 由于**inttoreal** 的运算对象是常数，编译器也可能是用等价的实常数来代替这个整常数. □

许多处理源程序的软件工具首先都要完成某类分析，这样的例子有：

1. **结构编辑器** 结构编辑器取一串命令作为输入来建立源程序。结构编辑器不仅象普通的正文编辑器那样完成正文的建立和修改，而且分析程序正文，把恰当的层次结构加在源程序上. 这样，结构编辑器能够完成一些对准备程序来说很有用的额外事情，例如，它能够检查输入是否能正确构成程序，能够自动提供关键字（如用户键入while，编辑器自动提供匹配的do，并提醒用户，在这两个关键字之间必须有一个条件）和能够从begin 或左括号跳到匹配的end 或右括号. 而且，这种编辑器的输出往往和编译器分析阶段的输出类似.

2. **格式打印器** 格式打印器分析源程序，以程序的结构清晰可见的方式输出程序. 例如，注解可以用特殊的字体出现，语句可以按它们嵌套的层次阶梯式地显示出来.

3. **静态检查器** 静态检查器读入程序，分析它，并试图不运行程序而发现一些潜在的错误。它的分析部分和第九章优化编译器的分析部分类似. 例如，它可以检查出源程序的某些部分决不会执行，或者某个变量在赋值前可能被引用. 此外，使用第五章讨论的类型检查技术，它还能捕捉一些逻辑错误，如企图把实型变量当作指针.

4.**解释器** 纯解释器在执行源程序指令时，都需分析构成该指令的字符串，以便识别和执行它指定的计算．如果给定的指令仅执行一次，纯解释是所有方法中代价最小的方法，因此它常用于作业控制语言和交互语言的“立即命令”．如果指令重复执行，较好途径是分析源程序的字符流仅一次，用一串更适于解释的符号序列或其它形式来代替它．因此解释器往往也做某种程度的翻译．例如，对于赋值语句，解释器可能建立象图1.4（a）那样的树，在遍历树时执行结点的操作．在根，它发现必须完成赋值，因此调用子例程来计算右部表达式的值，然后把值存到和标识符position 有关的存储单元．在根的右边，该子例程发现必须计算两个表达式的和，它递归地调用自己来计算表达式rate * 60的值，然后再加上变量initial 的值．

各虚拟层解释的例子都可以找到．一个极端是仅把常数改成内部形式，确定标识符的含义，可能还把中缀形式翻成后缀形式，APL 和SNOBOL 4 通常用这种方法实现．另一个极端是系统的硬件及一个小的运行系统形成解释器，这已是编译方式了，FORTRAN 和Pascal 总是用这种策略实现．

§1.3 编译的阶段

前面已经提到，编译器的工作可以分成若干阶段，每个阶段把源程序从一种表示变换成另一种表示。编译器的一种典型分解见图1.5．实际上，若干阶段可以组合在一起，如1.4 节将要提到的那样，各阶段组之间的中间表示也无需显式构造．

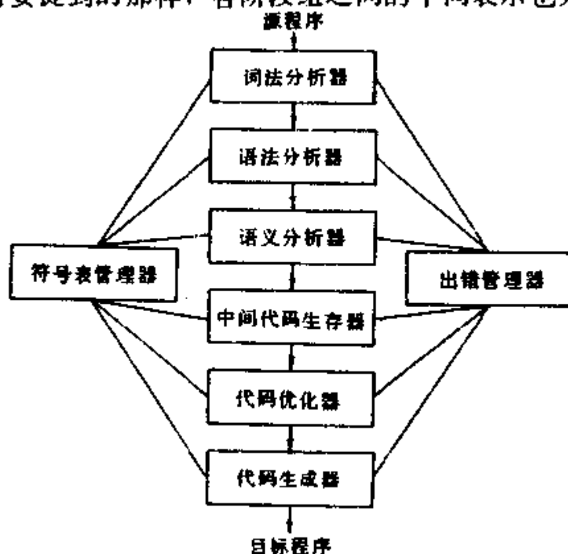


图 1.5 编译的阶段

前三个阶段完成编译器分析部分的大部分工作。另两个活动，符号表管理和出错管理，和词法分析、语法分析、语义分析、中间代码生成、代码优化和代码生成六个阶段相互作用．非正规地，我们把符号表管理和出错管理也称做“阶段”．

1.3.1 符号表管理

编译器的一个重要工作是记录源程序中使用的标识符和收集每个标识符的各种属性．

这些属性提供标识符的存储分配、类型和作用域信息，如果是过程标识符，还有参数的个数和类型，参数传递方式和返回值类型（如果有的话）.

· 符号表是为每个标识符保存一个记录的数据结构，记录的域是标识符的属性. 该数据结构允许我们迅速地找到标识符的记录，在此记录存储和读取数据. 符号表在第六章讨论.

词法分析器发现源程序的标识符时，把该标识符填入符号表. 但是，一般来说，词法分析期间不能确定一个标识符的属性. 例如，象

var position, initial, rate : real ;

这样的Pascal 声明，词法分析器读过position initial 和rate 时，它们的类型还是未知的.

其余的阶段把标识符的信息填入符号表，然后以不同的方式使用这些信息. 例如，语义分析和中间代码生成时，需要知道标识符的类型，才能检查源程序是否以有效的方式使用它们，才能为它们产生适当的操作. 代码生成时需要使用标识符存储分配信息以产生正确的指令.

1.3.2 错误诊断和报告

每个阶段都可能碰到错误. 在发现错误后，该阶段必须处理此错误，使得以后的编译可以继续进行，以便进一步觉察源程序的其它错误. 发现一个错误便停下来的编译器是没有尽到责任的.

语法分析和语义分析通常处理编译器能发现的绝大部分错误. 词法分析阶段可以发现当前被扫描的字符串不能形成语言的记号这类错误，记号流违反了语言的语法规则是由语法分析阶段诊断。语义分析时，编译器试图找出语法正确但对所含的操作来说是无意义的结构，如相加的两个标识符，其一是数组名，另一个是过程名. 我们把每个阶段的出错处理放在与那个阶段有关的章节介绍.

1.3.3 分析阶段

随着翻译的逐步深入，源程序的内部表示也在改变. 我们用语句

$$\text{position} := \text{initial} + \text{rate} * 60 \qquad (1.1)$$

的翻译为例，来说明这些表示.

词法分析读源程序的字符，把它们组成记号流，记号流的每个记号代表逻辑上有内聚力的字符序列，比如标识符、关键字（**if**、**while** 等）、标点符号、或多字符算符，如 := . 形成记号的字符序列叫做该记号的单词（lexeme）.

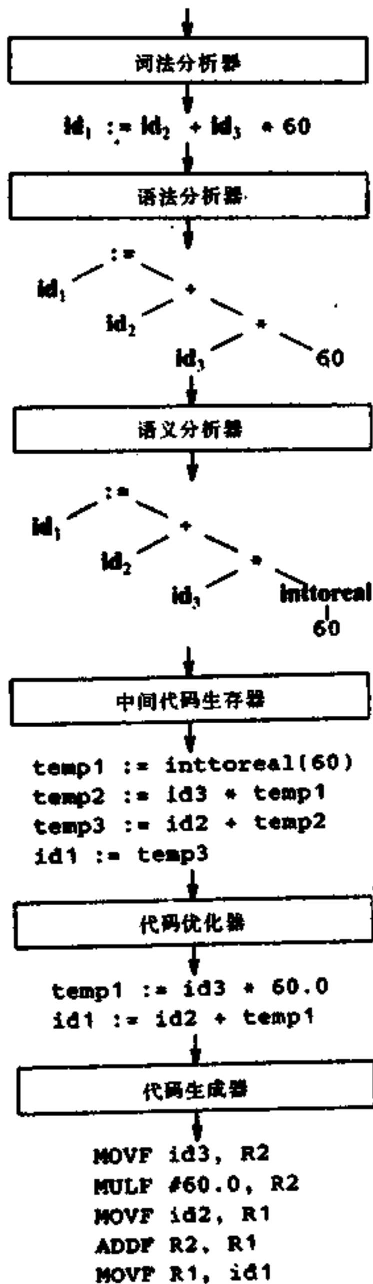
某些记号还增加一个“单词值”. 例如，发现rate 这样的标识符时，词法分析器不仅产生一个记号，如**id**，还把它的单词rate 填入符号表，如果表中还没有它的话. **id** 的这次出现的单词值是符号表中rate 条目的指针.

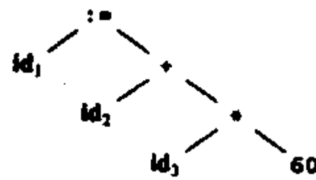
我们用$\mathbf{id}_1$、$\mathbf{id}_2$ 和$\mathbf{id}_3$ 分别表示position、initial 和rate，以强调标识符的内部表示是区别于形成标识符的字符序列的. 于是，（1.1）在词法分析后的表示是

$$\mathbf{id}_1 := \mathbf{id}_2 + \mathbf{id}_3 * 60 \qquad (1.2)$$

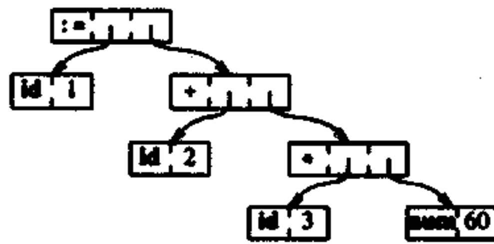
还应该为多字符算符：=和数60构造记号，以反映它们的内部表示，我们把它延迟到第二章词法分析再讨论.

第二和第三阶段，语法和语义分析，也已在1.2节中介绍了.语法分析把层次结构强加于记号流，我们用图1.7（a）的语法树来刻画它.树的一种典型数据结构见图1.7（b），它的内部结点是一个记录，其中一个域是算符，另两个域是指向左子树和右子树的指针.叶子也是记录，有两个或更多的域，一个域标识该叶子代表的记号，其余的记载有关这个记号的信息.可以增加结点记录的域来保存更多的信息.在第三和第五章分别讨论语法和语义分析.





(a)



(b)

图 1.7 树和它的数据结构

符号表

1	position	···
2	initial	···
3	rate	···
4		

图 1.6 一个语句的翻译

1.3.4 中间代码生成

语法和语义分析后，某些编译器产生源程序的显式中间表示，可以认为这种中间表示是一个抽象机的程序．中间表示必须具有两个性质：它容易产生和容易翻译成目标程序．

中间表示有各种形式．在第七章，我们把中间表示看成“三地址代码”，它们象机器的汇编语言，这种机器的每个存储单元的作用类似寄存器．三地址代码由指令序列组成，每条指令最多有三个操作数，（1.1）源程序的三地址代码可以如下：

```
temp 1: = inttoreal (60)
temp 2: = id 3 * temp 1
temp 3: = id 2 + temp 2          (1.3)
id 1: = temp 3
```

这种中间形式有它的特点．首先，除了赋值算符外，每条指令至多只有一个算符，因此，在生成这些指令时，编译器必须决定运算完成的次序，（1.1）源程序的乘优于加．其次，编译器必须产生临时变量名，用以保留每条指令的计算结果．第三，某些“三地址”指令没有三个运算对象，例如（1.3）的第一条和最后一条指令．

在第七章我们叙述编译器用的主要中间表示．通常，除了计算表达式外，这些中间表示还必须做其它事情，它们必须处理控制流结构和过程调用．第四章和第七章提出一些对典型的程序设计语言结构产生中间代码的算法．

1.3.5 代码优化

代码优化阶段试图改进代码，产生执行较快的机器代码．有些优化是简单的，例如，产生中间代码的一个很自然的算法是为语义分析后的树的每个算符产生一条指令，因而得到（1.3）的中间代码．当然，还存在更好的算法，如使用两条指令

```
temp 1: = id 3 * 60.0
id 1: = id 2 + temp 1            (1.4)
```

也可以完成同样的计算．用简单的算法也是可以的，因为这个问题可以在代码优化阶段得以解决，也就是，编译器会推断出，60从整型变为实型表示可以在编译时完成，从而 **inttoreal** 操作可以删去．此外，temp 3 只使用一次，即把它的值传给id 1，所以用id 1代替temp 3是安全的，从而（1.3）的最后一个语句不必存在，这样，得到（1.4）的结果．

不同的编译器完成不同程度的优化，能完成大多数优化的，叫做“优化编译器”，但是编译的相当大一部分时间消耗在优化上．简单的优化也可使目标程序的运行时间大大缩短，而编译速度并没有降低太多．第八章将讨论一般的优化问题，而第九章给出功能更强的优化编译器所使用的技术．

试读结束：需要全本请在线购买： www.ertongbook.com