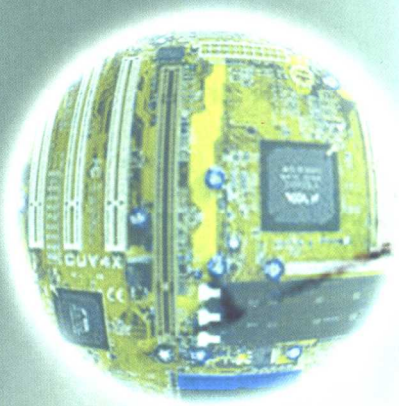
高等教育系列教材(计算机与信息管理类)

编译原理

总主编 彭 波
本书主编 孙总参
总策划 张 玲



南海出版公司

高等教育系列教材(计算机与信息管理类)

编译原理

总主编　彭　波
本书主编　孙总参
总策划　张　玲

南海出版公司
2003·海口

图书在版编目(CIP)数据

编译原理/孙总参主编. —海口:南海出版公司,2003.3

高等教育系列教材(计算机与信息管理类)

ISBN 7-5442-2369-8

Ⅰ.编… Ⅱ.孙… Ⅲ.编译程序-程序设计-高等学校-教材 Ⅳ.TP314

中国版本图书馆 CIP 数据核字(2003)第 099262 号

BIANYI YUANLI

编译原理

主　　编　孙总参
责任编辑　张　辉
装帧设计　时　代
出版发行　南海出版公司　**电话**(0898)65350227
社　　址　海口市蓝天路友利园大厦 B 座 3 楼　**邮编**　570203
经　　销　新华书店
印　　刷　北京昌平前进印刷厂
开　　本　787×960　1/16
印　　张　20.25
字　　数　386 千
版　　次　2003 年 3 月第 1 版　2003 年 3 月第 1 次印刷
印　　数　1—10000 册
书　　号　ISBN 7-5442-2369-8
定　　价　28.80 元

南海版图书　版权所有　盗版必究

编审说明

编译原理是国内高等院校计算机科学与技术专业必修专业课程之一，是一门理论与实践并重的课程，对引导学生进行科学思维和提高学生解决实际问题的能力，具有十分重要的作用。

综观已出版的编译原理教材，要么是讲解高深的理论，要么是介绍具体编译器的实现，都不太适合我国高校教学的特点。基于这一点，本书充分考虑我国高等教育的现状，力求将基本概念、基本原理和实现方法的思路阐述清楚，重视理论联系实际，力求达到条理清晰，通俗易懂，使读者能够很快掌握编译原理的基本内容。同时为了帮助学生掌握每章的重点和难点，本书各章均附习题，以便于学生复习掌握。

全书主要介绍编译系统的一般构造原理、基本实现技术和一些自动构造工具，同时也包含了面向对象语言等当前较新语言的编译技术。主要内容包括语言基础知识、词法分析、语法分析、中间代码生成、代码优化、目标代码生成、符号表的构造和运行时存储空间的组织等。此外，还介绍了近年来在编译程序的自动生成工具研制方面所取得的一些成果，并引入了 LEX、YACC 的使用方法与实例。各校可根据教学需要，有选择地进行学习。

实践中并非每个人都能够构造或维护编译器，但是学生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，学生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语言的编译器，掌握一些软件设计技巧，肯定会受益匪浅。编者希望学生通过这门课程的学习，不仅能够提高编程技巧，掌握软件设计新技术，而且对计算机系统软件有一比较清楚

的了解，对今后进一步深造起到铺垫作用。

本套计算机类系列教材由中国农业大学计算机系主任彭波教授担任总主编，由北京时代科发教材研究中心计算机事业部主任张玲担任总策划。本书由孙总参主编，彭波教授审阅了全书，并提出了许多宝贵的建议，为本书的编写和出版做了大量工作。此外，还有其他为本书的编写和出版付出了辛勤劳动的同志们，编者在此一并表示诚挚的谢意。

编者力求反映编译及其相关领域的基础知识与发展方向，并且力求用通俗的语言讲述抽象的原理，但由于编者水平有限，书中难免存在缺点或错误之处，希望广大师生和有关专家教授不吝批评指正，以便不断修订完善。

高等教育系列教材编审指导委员会

2003 年 3 月

此为试读，需要完整PDF请访问：www.ertongbook.com

目　录

第 1 章　编译程序概论

§1.1　引　论

使用过计算机的人都知道，大多数用户都是应用高级语言来实现他们所需要的计算的。编译程序是现代计算机系统的基本组成部分之一，也是用户最关心的工具之一。多数计算机系统都含有不止一个高级语言的编译程序，对有些高级语言甚至配置了几个不同性能的编译程序，供用户按照不同的需要进行选择。

一个高级语言程序在计算机上的执行过程可以分为两步：第一步，用一个编译程序把高级语言翻译成机器语言程序；第二步，运行所得到的机器语言程序来求得计算结果。

从功能上看，一个编译程序就是一个语言翻译程序。通常所说的编译程序就是指能把某一种语言程序（*称为源语言程序*）改造成另一种语言程序（*称为目标语言程序*），而且这两者在逻辑上是等价的。例如，汇编程序是一个翻译程序，它把汇编语言程序翻译成机器语言程序。若源语言是诸如 Fortran、Pascal、Algol 或 C 等这样的高级语言，而目标语言是诸如汇编语言或机器语言之类的低级语言，则这样的翻译程序就称为**编译程序**。

编译程序所执行的功能，如图 1-1 所示。

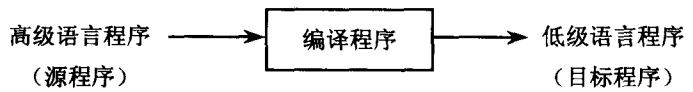


图 1-1　编译程序的功能

编译程序的重要性体现在它使得大多数的计算机用户不必考虑与机器有关的繁琐细节，即机器对用户是透明的，使得程序员专心于程序设计而不必考虑机器的差异。这对于当今机器的种类和数量不断增长的年代尤为重要。

§1.2　编译器简介

何谓编译器？编译器就是将一种语言翻译成另一种语言的计算机程序。编译器将源程序（Source Language）编写的程序作为输入，产生用目标语言（target language）编写的等价程序。通常源程序为高级语言（High-level Language），如 C 或 C++，而目标语言则是目标机器的目标代码（Object Code），有时也称作机器代码（Machine Code），也就是写在计算机机器指令中的用于运行的代码。这一过程如图 1-2 所示。

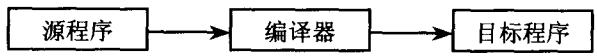


图 1-2　编译器的作用

编译器是一种相当复杂的程序，其代码的长度可从10 000行到1 000 000行不等。编写甚至读懂这样一个程序绝非易事，大多数的计算机科学家和专业人员也从来没有编写过一个完整的编译器。但是，几乎所有形式的计算均要用到编译器，而且任何一个与计算机打交道的专业人员都应掌握编译器的基本结构和操作。除此之外，计算机应用程序中经常遇到的一个任务就是命令解释程序和界面程序的开发，这比编译器要小，但使用的却是相同的技术。因此，掌握这一技术具有非常大的实际意义。

1.2.1　编译器的历史及其发展

在20世纪40年代，由于冯·诺伊曼在存储一程序计算机方面的先锋作用，编写一串代码或者程序已经成为必要，这样计算机就可以执行所需的计算。这些程序最初都是用机器语言（Machine Language）编写的。机器语言就是表示机器实际操作的数字代码，例如：

C7 06 0000 0002

表示在IBM PC上使用的Intel 8x86处理器将数字2移至地址0000（16进制）的指令。当然，编写这样的代码是十分费时和乏味的，这种代码形式很快就被汇编语言（Assembly Language）代替了。在汇编语言中，都是以符号形式给出指令和存储地址的。例如，汇编语言指令

MOV X, 2

就与前面的机器指令等价（假设符号存储地址X是0000）。汇编程序（Assembler）将汇编语言的符号代码和存储地址翻译成与机器语言相对应的数字代码。汇编语言大大提高了编程的速度和准确度，人们至今仍在使用它，在编码需要极快

的速度和极高的简洁程度时尤为如此。但是，汇编语言也有许多缺点：编写起来也不容易，阅读和理解很难；而且汇编语言的编写严格依赖于特定的机器，所以为一台计算机编写的代码在应用于另一台计算机时必须完全重写。很明显，发展编程技术的下一个重要步骤就是以一个更类似于数学定义或自然语言的简洁形式来编写程序的操作，它应与任何机器都无关，而且也可由一个程序翻译为可执行的代码。例如，前面的汇编语言代码可以写成一个简洁的与机器无关的形式**x=2**，起初人们担心这是不可能的，或者即使可能，目标代码也会因效率不高而没有多大用处。

1954~1957年，IBM John Backus带领的一个研究小组对FORTRAN语言及其编译器的开发，解除了上述担忧。但是，由于当时处理中所涉及到的大多数程序设计语言的翻译并不为常人所掌握，所以这个项目的成功也伴随着巨大的辛劳。几乎与此同时，人们也在开发着第一个编译器，Noam Chomsky开始了他的自然语言结构的研究。他的发现最终使得编译器结构异常简单，甚至还带有了一些自动化。Chomsky的研究导致了根据语言文法（Grammar，指定其结构的规则）的难易程度以及识别它们所需的算法来为语言分类。正如现在所称的——与乔姆斯基分类结构（Chomsky hierarchy）一样——包括了文法的4个层次：0型、1型、2型和3型文法，且其中的每一个都是其前者的专门化。2型（或上下文无关文法，Context-free Grammar）被证明是程序设计语言中最有用的，而且今天它已代表着程序设计语言结构的标准方式。分析问题（Parsing Problem，用于限定上下文无关语言的识别的有效算法）的研究是在20世纪60年代和70年代，它相当完善地解决了这一问题，现在它已是编译理论的一个标准部分。有穷自动机（Finite Automata）和正则表达式（Regular Expression）同上下文无关文法紧密相关，它们与乔姆斯基的3型文法相对应。对它们的研究与乔姆斯基的研究几乎同时开始，并且引出了表示程序设计语言的单词（或称为记号）的符号方式。人们接着又深化了生成有效的目标代码的方法，这就是最初的编译器，它们被一直使用至今。人们通常将其误称为优化技术（Optimization Technique），但因其从未真正地得到过被优化了的目标代码而仅仅改进了它的有效性，因此实际上应称作代码改进技术（Code Improvement Technique）。当分析问题变得好懂时，人们就在开发程序上花费了很大的功夫来研究这一部分的编译器的自动构造。这些程序最初被称为编译程序——编译器，但更确切地应称为分析程序生成器（Parser Generator），这是因为它们仅仅能够自动处理编译的一部分。这些程序中最著名的是Yacc（Yet Another Compiler-compiler），它是由Steve Johnson在1975年为Unix系统编写的。类似地，有穷自动机的研究也发展了另一种称为扫描程序生成器（Scanner Generator）的工具，Lex（与Yacc同时，由Mike Lesk为Unix系统开发的）是其中的佼佼者。

此为试读，需要完整PDF请访问：www.ertongbook.com

在20世纪70年代后期和80年代早期，大量的项目都关注于编译器其他部分的生成自动化，这其中就包括了代码生成。这些尝试并未取得多少成功，这大概是因为操作太复杂而人们又对其不甚了解，在此不再赘述。

就最近的发展来说：首先，编译器包括了更为复杂的算法的应用程序，它用于推断或简化程序中的信息；这又与更为复杂的程序设计语言（可允许此类分析）的发展结合在一起。其中典型的有用于函数语言编译的Hindley-Milner类型检查的统一算法。其次，编译器已越来越成为基于窗口的交互开发环境（Interactive Development Environment，IDE）的一部分，它包括了编辑器、链接程序、调试程序以及项目管理程序。这样的IDE的标准并没有多少，但是已沿着这一方向对标准的窗口环境进行开发了。尽管近年来对此进行了大量的研究，但是基本的编译器设计在近20年中都没有多大的改变，而且它们正迅速地成为计算机科学课程中的中心一环。

1.2.2 与编译器相关的程序描述

1. 解释程序（Interpreter）

解释程序是如同编译器的一种语言翻译程序。它与编译器的不同之处在于：它立即执行源程序而不是生成在翻译完成之后才执行的目标代码。从原理上讲，任何程序设计语言都可被解释或被编译，但是根据所使用的语言和翻译情况，很可能会选用解释程序而不用编译器。例如，我们经常解释BASIC语言而不是去编译它。类似地，诸如LISP的函数语言也常常是被解释的。解释程序也经常用于教育和软件的开发，此处的程序很有可能被翻译若干次。而另一方面，当执行的速度是最为重要的因素时就使用编译器，这是因为被编译的目标代码比被解释的源代码要快得多，有时要快10倍或更多。但是，解释程序具有许多与编译器共享的操作，而两者之间也有一些混合之处。

2. 汇编程序（Assembler）

汇编程序是用于特定计算机上的汇编语言的翻译程序。正如前面所提到的，汇编语言是计算机的机器语言的符号形式，它极易翻译。有时，编译器会生成汇编语言以作为其目标语言，然后再由一个汇编程序将它翻译成目标代码。

3. 连接程序（Linker）

编译器和汇编程序都经常依赖于连接程序，它将分别在不同的目标文件中编译或汇编的代码收集到一个可直接执行的文件中。在这种情况下，目标代码，即还未被连接的机器代码，与可执行的机器代码之间就有了区别。连接程序还连接目标程序和用于标准库函数的代码，以及连接目标程序和由计算机的操作系统提供的资源（例如，存储分配程序及输入与输出设备）。有趣的是，连接程序现在正在完成编译器最早的一个主要活动（这也是“编译”一词的用法，

即通过收集不同的来源来构造）。因为连接过程对操作系统和处理器具有极大的依赖性，本书也就不研究它了。我们也就不细分连接的目标代码和可执行的代码，这是因为对于编译技术而言，这个区别并不重要。

4. 装入程序（Loader）

编译器、汇编程序或连接程序生成的代码经常还不完全适用或不能执行，但是它们的主要存储器访问却可以在存储器的任何位置中且与一个不确定的起始位置相关。这样的代码被称为是**可重定位的**（Relocatable），而装入程序可处理所有的与指定的基地址或起始地址有关的可重定位的地址。装入程序使得可执行代码更加灵活，但是装入处理通常是在后台（作为操作环境的一部分）或与连接相联合时才发生，装入程序极少会是实际的独立程序。

5. 预处理器（Preprocessor）

预处理器是在真正的翻译开始之前由编译器调用的独立程序。预处理器可以删除注释、包含其他文件以及执行宏（宏Macro是一段重复文字的简短描写）替代。预处理器可由语言（如C）要求或以后作为提供额外功能（诸如为FORTRAN提供Ratfor预处理器）的附加软件。

6. 编辑器（Editor）

编译器通常接受由任何生成标准文件（例如ASCII文件）的编辑器编写的源程序。最近，编译器已与另一个编辑器和其他程序捆绑进一个交互的开发环境——IDE中。此时，尽管编辑器仍然生成标准文件，但会转向正被讨论的程序设计语言的格式或结构。这样的编辑器称为**基于结构的**（Structure Based），且它早已包括了编译器的某些操作；因此，程序员就会在程序的编写时而不是在编译时就得知错误了。从编辑器中也可调用编译器以及与它共用的程序，这样程序员无需离开编辑器就可执行程序。

7. 调试程序（Debugger）

调试程序是可在被编译了的程序中判定执行错误的程序，它也经常与编译器一起放在IDE中。运行一个带有调试程序的程序与直接执行不同，这是因为调试程序保存着所有的或大多数源代码信息（诸如行数、变量名和过程）。它还可以在预先指定的位置（称为断点，Breakpoint）暂停执行，并提供有关已调用的函数以及变量的当前值的信息。为了执行这些函数，编译器必须为调试程序提供恰当的符号信息，而这有时却相当困难，尤其是在一个要优化目标代码的编译器中。

8. 描述器（Profiler）

描述器是在执行中搜集目标程序行为统计的程序。程序员特别感兴趣的统计是每一个过程的调用次数和每一个过程执行时间所占的百分比。这样的统计对于帮助程序员提高程序的执行速度极为有用。有时编译器也甚至无需程序员

的干涉就可利用描述器的输出来自动改进目标代码。

9. 项目管理程序（Project Manager）

现在的软件项目通常大到需要由一组程序员来完成，这时对那些由不同人员操作的文件进行整理就非常重要了，而这正是项目管理程序的任务。例如，项目管理程序应将由不同的程序员制作的文件的各个独立版本整理在一起，它还应保存一组文件的更改历史，这样就能维持一个正在开发的程序的连贯版本了（这对那些由单个程序员管理的项目也很有用）。项目管理程序的编写可与语言无关，但当其与编译器捆绑在一起时，它就可以保持有关特定的编译器和建立一个完整的可执行程序的链接程序操作的信息。在Unix系统中有两个流行的项目管理程序：sccs（Source Code Control System ）和rcs（Revision Control System ）。

1.2.3 翻译步骤

编译器内部包括了许多步骤或称为阶段（phase），它们执行不同的逻辑操作。将这些阶段设想为编译器中一个个单独的片断是很有用的，尽管在应用中它们是经常组合在一起的，但它们确实是作为单独的代码操作来编写的。图1-3是编译器中的阶段和与以下阶段（文字表、符号表和错误处理器）或其中的一部分交互的3个辅助部件。在此只简要地描述一下每个阶段，今后大家还将会更详细地学习。

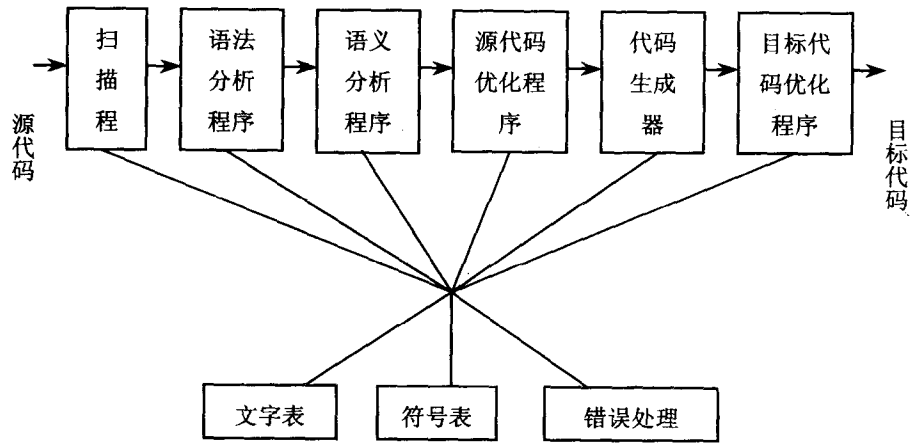


图1-3 编译器的阶段

1. 扫描程序（Scanner）

在这个阶段编译器实际阅读源程序（通常以字符流的形式表示）。扫描程序执行词法分析（Lexical analysis）：它将字符序列收集到称作记号（token）的

有意义单元中，记号同自然语言，如英语中的字词相似。因此可以认为扫描程序执行与拼写相似的任务。例如在下面的代码行（它可以是C程序的一部分）中：

a [index] = 4 + 2

这个代码包括了12个非空字符，但只有8个记号：

a	标识符
[	左括号
index	标识符
]	右括号
=	赋值
4	数字
+	加号
2	数字

每一个记号均由一个或多个字符组成，在进一步处理之前它已被收集在一个单元中。扫描程序还可完成与识别记号一起执行的其他操作。例如，它可将标识符输入到符号表中，将文字（literal）输入到文字表中（文字包括诸如3.1415926535的数字常量，以及诸如“Hello, world!”的引用字符串）。

2. 语法分析程序（Parser）

语法分析程序从扫描程序中获取记号形式的源代码，并完成定义程序结构的语法分析（syntax analysis），这与自然语言中句子的语法分析类似。语法分析定义了程序的结构元素及其关系。通常将语法分析的结果表示为分析树（parse tree）或语法树（syntax tree ）。分析树对于显示程序的语法或程序元素很有帮助，但是对于表示该结构却显得力不从心了。分析程序更趋向于生成语法树，语法树是分析树中所含信息的浓缩（有时因为语法树表示从分析树中的进一步抽取，所以也被称为**抽象的语法树**（abstract syntax tree））。

3. 语义分析程序（Semantic analyzer）

程序的语义就是语义分析的“意思”，它与语法或结构不同。程序的语义确定程序的运行，但是大多数的程序设计语言都具有在执行之前被确定而不易由语法表示和由分析程序分析的特征。这些特征被称作静态语义（static semantic），而语义分析程序的任务就是分析这样的语义（程序的“动态”语义具有只有在程序执行时才能确定的特性，由于编译器不能执行程序，所以它不能由编译器来确定）。一般的程序设计语言的典型静态语义包括声明和类型检查。由语义分析程序计算的额外信息（诸如数据类型）被称为属性（attribute），它们通常是作为注释或“装饰”增加到树中（还可将属性添加到符号表中）。

在正运行的C表达式：

a [index] = 4 + 2

中，该行分析之前收集的典型类型信息可能是：**a**是一个整型值的数组，它带有来自整型子范围的下标；**index**则是一个整型变量。接着，语义分析程序将用所有的子表达式类型来标注语法树，并检查赋值是否使这些类型有意义了，如若没有，则声明一个类型匹配错误。在上例中，所有的类型均有意义，有关语法树的语义分析结果可用注释了的树来表示。

4. 源代码优化程序（Source code optimizer）

编译器通常包括许多代码改进或优化步骤。绝大多数最早的优化步骤是在语义分析之后完成的，而此时代码改进可能只依赖于源代码。这种可能性是通过将这一操作提供为编译过程中的单独阶段指出的。每个编译器不论在已完成的优化种类方面还是在优化阶段的定位中都有很大的差异。

在上例中，我们包括了一个源代码层次的优化机会，也就是：表达式**4 + 2**可由编译器计算先得到结果**6**（这种优化称为常量合并，constant folding）。当然，还会有更复杂的情况（有些将在以后的章节中提到）。另一个常见的选择是P-代码（P-code），它常用于Pascal编译器中。

在前面的例子中，原先的C表达式的三元式代码应是：

t = 4 + 2

a [index] = t

（请注意：这里利用了一个额外的临时变量t存放加法的中间值）。这样，优化程序就将这个代码改进为两步。首先计算加法的结果：

t = 6

a [index] = t

接着，将**t**替换为该值以得到三元语句

a [index] = 6

源代码优化程序可以通过将其输出称为**中间代码**（intermediate code）来使用三元式代码。中间代码一直是指一种位于源代码和目标代码（例如三元式代码或类似的线性表示）之间的代码表示形式。但是，我们可以更概括地认为它是编译器使用的源代码的任何一个内部表示。此时，也可将语法树称作中间代码，源代码优化程序则确实能继续在其输出中使用这个表示。有时这个中间代码也称作**中间表示**（intermediate representation ,IR）。

5. 代码生成器（Code generator）

代码生成器得到中间代码（IR），并生成目标机器的代码。尽管大多数编译器直接生成目标代码，但是为了便于理解，本书用汇编语言来编写目标代码。正是在编译的这个阶段中，目标机器的特性成了主要因素。当它存在于目标机器时，使用指令不仅是必须的，而且数据的形式表示也起着十分重要的作用。例如，整型数据类型的变量和浮点数据类型的变量在存储器中所占的字节数或

字数也很重要。

在上述示例中，现在必须决定怎样存储整型数来为数组索引生成代码。例如，下面是所给表达式的一个可能的样本代码序列（在假设的汇编语言中）：

```
MOV R0, index ;   ; value of index -> R0
MUL R0, 2 ;       ; double value in R0
MOV R1, &a ;      ; address of a -> R1
ADD R1, R0 ;      ; add R0 to R1
MOV *R1, 6 ;      ; constant 6 -> address in R1
```

在以上代码中，为编址模式使用了一个类似C 的协定，因此**&a**是**a**的地址（也就是数组的基地址），*** R_1**则意味着间接寄存器地址（因此最后一条指令将值6存放在R_1包含的地址中）。这个代码还假设机器执行字节编址，并且整型数占据存储器的两个字节（所以在第2条指令中用2作为乘数）。

6. 目标代码优化程序（Target code optimizer）

在这个阶段中，编译器尝试着改进由代码生成器生成的目标代码。这种改进包括选择编址模式以提高性能、将速度慢的指令更换成速度快的，以及删除多余的操作。

在上面给出的样本目标代码中，还可以做许多更改：在第2条指令中，利用移位指令替代乘法（通常地，乘法很费时间），还可以使用更有效的编址模式（例如用索引地址来执行数组存储）。使用了这两种优化后，目标代码就变成：

```
MOV R0, index ;   ; value of index -> R0
SHL R0 ;          ; double value in R0
MOV &a[R0], 6 ;   ; constant 6 -> address a+R0
```

到这里，对编译器阶段的简要描述就结束了，但我们还应特别强调这些讲述仅仅是示意性的，也无需表示出正在工作中的编译器的实际结构。编译器在其结构细节上确实差别很大，然而，上面讲到的阶段总会出现在几乎所有的编译器的某个形式上。

1.2.4　编译器的结构

编译器的结构可以从不同的角度来考察，下面分别介绍两种最普遍的观点。

1. 前端和后端

该观点将编译器分成只依赖于源语言（前端（Front End））的操作和只依赖于目标语言（后端（Back End））的操作两部分。这与将编译器分成分析和综合两部分是类似的：扫描程序、分析程序和语义分析程序是前端，代码生成器是后端。但是一些优化分析可以依赖于目标语言，这样就是属于后端了，然而中间代码的综合却经常与目标语言无关，因此也就属于前端了。在理想情况

此为试读，需要完整PDF请访问：www.ertongbook.com