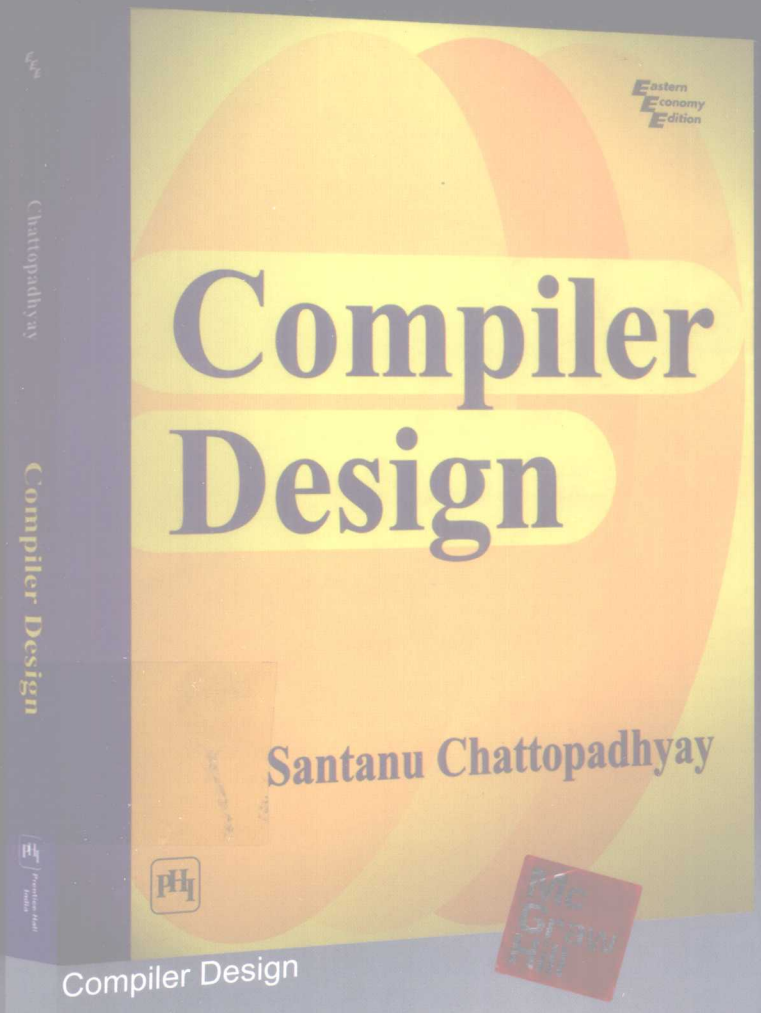
国外计算机科学经典教材

编译器设计

(印度) Santanu Chattopadhyay 著　　徐骁栋 王海涛 译



清华大学出版社

国外计算机科学经典教材

编译器设计

(印度) Santanu Chattopadhyay 著

徐骁栋　王海涛　　译

清华大学出版社

北　京

Compiler Design, EISBN: 81-203-2725-X
By Santanu Chattopadhyay

Originally published in India by Prentice Hall of India
Chinese(in simplified character only) translation rights arranged with Prentice Hall of India through McGraw-Hill Education(Asia)

北京市版权局著作权合同登记号　图字：01-2007-3123

图书在版编目(CIP)数据

编译器设计/(印)查托帕答雅(Chattopadhyay, S.)著；徐骁栋，王海涛 译.
—北京：清华大学出版社，2009.1
(国外计算机科学经典教材)
书名原文：Compiler Design

ISBN 978-7-302-18865-0

I. 编… II. ①查… ②徐… ③王… 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(2008)第 173615 号

责任编辑：王　军　徐燕萍
装帧设计：孔祥丰
责任校对：成凤进
责任印制：李红英
出版发行：清华大学出版社　　地　　址：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　邮　　编：100084
社　总　机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：北京市清华园胶印厂
装 订 者：三河市溧源装订厂
经　　销：全国新华书店
开　　本：185×230　印　张：13.25　字　数：273 千字
版　　次：2009 年 1 月第 1 版　　印　　次：2009 年 1 月第1 次印刷
印　　数：1～4000
定　　价：28.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177 转 3103　　产品编号：025484-01

序言

本书源于笔者多年来对编译器的学习和教学经验，并以在该学科领域与学生的交流以及他们的反馈和研究作为依据。编译器的设计过程是将自动机理论、数据结构、算法、计算机体系机构和操作系统等领域知识进行综合的过程。编译器设计者必须要有较好的计算机科学理论的背景。同时，还应能掌握各种可用的软硬件平台。因此，任何编译器的书籍都应强调这些领域的作用，并向读者介绍编译领域中的难点和技术。同时，书籍应足够精炼，以便能在一个学期内完成课程。另外，还需要介绍编译器设计中各种可用的工具。本书涵盖了编译器中所有应掌握的主题，一共分为 9 章。

第 1 章介绍了编译器及其应用，并着重介绍了大量的应用领域，以使读者能理解该领域的范围。然后介绍了编译器的各个阶段，列举了各个阶段的任务。依据机器中软/硬件的功能和用户的需求，详细介绍了编译器设计的难点。本章以一个示例结束。

第 2 章展示了编译器中词法分析阶段的理论和设计。在介绍完正式的表示法如正则表达式、非确定和确定的有限自动机后，本书详细描述了使用正则表达式开始表示语言中单词规范的方式，并依据正则表达式构造识别器来标识输入流中语言的合法单词。最后，介绍了轻松生成词法分析器的自动化工具——Lex。

第 3 章介绍了语法分析。首先明确了在指定的语言中文法的作用、文法的符号约定以及相关定义。本章详细描述了各种不同类型的解析器，首先介绍了最简单的自上而下分析，如递归下降和 LL。然后介绍了先进的自下而上分析，如 LR、规范 LR 和 LALR。因为大多数编译器使用 LR 解析器，所以主要强调了 LR 分析，并介绍了工具 yacc，用于自动生成 LALR 解析器。本章最后介绍了使用解析器来执行输入流的各种语法制导翻译的方式。

第 4 章介绍了类型检查，并介绍了类型检查的相关问题，以及使用这些理论检查类型等价的方式。最后介绍了类型转换，用于展示编译器自动推断必要的类型转换的方式。

第 5 章介绍了编译器中符号表的用法。根据语言的作用域规则，介绍了各种不同的数据结构来存储标识符，因此，编译器能有效地使用这些信息。

第 6 章介绍了编译器在程序执行时维持运行环境的职责，然后介绍了活动记录，并

展示了使用不同而有效的数据结构维护活动记录的方式。

第 7 章详细介绍了中间代码的生成。这是编译器中的可选阶段，有助于将代码重定向到不同的体系结构中。首先介绍中间代码不同的表现方式，然后使用语法制导模式将常用的编程语言结构翻译成中间代码的形式。

第 8 章列举了目标代码的生成技术。在讨论完影响代码生成的因素后，介绍了使用基本的代码块表示中间语言程序的方式，然后从基本的代码块生成目标代码。同时也讨论了寄存器分配、缓存管理的问题及相应的解决策略。

第 9 章是代码优化阶段。在讨论完优化阶段必要且相关的问题后，介绍了优化的分类。然后介绍了不同的优化变换，如编译时计算、公共子表达式消除等。接下来介绍了全局优化和数据流问题。最后根据建立和求解数据流等式的技术解决一些数据流问题。

本书提供了编译器设计领域中的所有要点，适合作为本科生一个学期的学习内容。如果能有效地使用本书，将会得到很大的收获。同时欢迎读者给出任何对内容有改进的建设性意见。

Santanu Chattopadhyay

目录

第 1 章 引 言

编译器已成为现代计算机系统中不可或缺的一部分。它们负责将用户的计算需求转变成一段程序，从而让底层机器理解。因此，这些工具可以作为两种不同领域的实体——人类和计算机之间的接口。该转换的实际过程其实很复杂，其理论基础是自动机理论讲座中经常涉及到的语言的正式定义和识别程序。这为我们能在编译器设计的各个阶段设计出不同类型的自动化工具提供了坚实的基础。这些工具将在后面章节详细介绍。另一方面，随着计算机体系结构的发展，内存管理和新的操作系统为编译器的设计者生成优化的代码提供了大量的选择。在该方向的持续研究和努力反映了该现状。编译器设计课程主要用于解决所有这些问题，从理论基础到自动化工具的体系结构。本章将介绍编译工作及其应用领域、编译器的各个阶段以及编译器设计者需要处理的难点。最后以一个示例来演示这一过程。

1.1 编译器的定义

编译器是一个系统软件，用于将源语言程序转换为相应的目标语言程序。它还需要验证输入程序是否符合源语言的规范。如果有违反规范的，例如，源语言规定每个语句必须以分号(；)结束，如果程序中违反了该规范，则编译器将向用户显示错误消息。对于少数不重要的违规，可能会显示警告。图 1-1 展示了整个编译过程。

早期的计算机系统中不存在编译器的概念。早期的程序都是用汇编语言编写的，通过手工将其直接转换成机器代码交给计算机处理，而需要处理的程序也很小。20 世纪 50 年代，第一个计算机高级语言 Fortran 的引入推动了编译器的设计。在那个时候，编写编译器被认为是非常难的任务。但是由于有自动机理论的背景，这在很大程度上加速了编译器的

发展，产生了很多用于编译器设计的自动化工具。现在，设计一个中等规模语言的编译器至多只能算学生一个学期的项目。

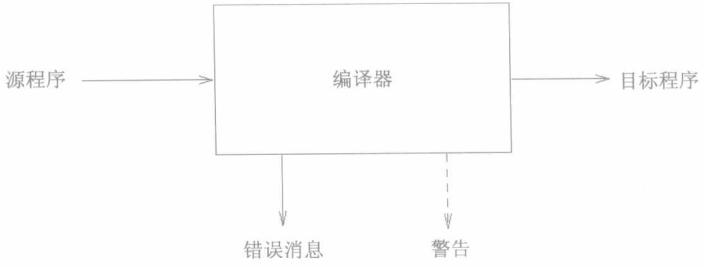


图 1-1　编译器的输入和输出

由于目前出现了大量知名度较小且可能不出名的计算机语言，以及大量经过测试的软硬件平台，因此到目前为止，几乎不可能确定已设计出来的编译器的数目。但是，有效的编译器设计是任何常用语言所必需的。在早期费力地编写函数式编程语言 LISP 的翻译程序时，恰好验证了这一点。和用对应的命令式语言如 Fortran、Pascal 等编写的程序相比，翻译程序的低效率使得用 LISP 编写的程序运行得非常慢。最近在内存管理策略尤其是垃圾收集方面的改进，为加快翻译程序的实现和这些语言的复兴铺平了道路。

1.2　编译器应用

编译器应用主要集中于将程序从一种语言转换为另一种语言的基本功能。不过，编译器应用主要涵盖以下几个领域。

(1) 机器代码生成。如前面所述，编译器主要用于将源语言程序转换成机器可理解的代码。因此，一方面需要注意不同结构的源语言的语义。与此同时，也需要考虑到目标机器的局限性和特定的功能。应当指出，虽然自动机理论有助于进行语法检查(即语法构造)，但它并不能说明如何产生语法正确的代码。因此，编译的理论也包括有效的机器代码生成。

(2) 格式转换器。这些工具通常需要两个或更多个软件程序包之间的接口。由于软件来自不同的供应商，而在许多情况下需要将一种工具的输出作为下一个工具的输入。对于这些应用，输入输出格式的兼容性是一个非常重要的要求。在缺乏兼容性的情况下，需要使用格式转换器将前一种软件的输出转换成后一种软件所能接受的格式。同样地，代码转换器能将一些老式语言(如 COBOL)编写的常用程序转换成如 C、C++等新式语言编写的程序。

(3) 硅编译。该过程使用如 VHDL、Verilog 等行为描述语言自动合成电路。由于电路的复杂性和产品上市时间的缩短，对自动化工具的使用迅猛增长。和程序代码生成相比，硅编译的优化标准也不同。这里最重要的问题是面积要求、延迟以及功耗等。

(4) 查询解释。这属于数据库查询处理的领域。在数据库中，最重要的优化准则是减少查找时间。此外，在这类查询中经常存在多个求值序列，这些选择的代价多数取决于如表的相对大小、索引的可用性等实际情况。这里编译器负责生成合适的运算序列，用于在最短的时间内响应查询。

(5) 文本格式化。接受普通的文本文件连同嵌入在其中的格式化命令作为输入的文本格式化软件属于这一范畴。这些系统中比较典型的有 troff、nroff 和 LaTex 等。

1.3 编译器的各个阶段

为了简化开发过程和便于理解，可将编译器在概念上划分为几个阶段，即：

- 词法分析
- 语法分析
- 语义分析
- 中间代码生成
- 目标代码生成
- 代码优化
- 符号表管理
- 错误处理和恢复

可以发现，在这些模块间不存在任何严格的分界，而是相互交织在一起合作完成任务。图 1-2 显示了各模块间的关系。下面简要列举各阶段的任务。

(1) 词法分析。这是编译器和外部世界的接口。该阶段的任务是扫描输入的源程序，标识其中的合法单词。另外，词法分析还需要删除些多余的符号，如程序中额外的空格、用户添加的注释等，并需要扩展用户自定义的宏命令，如 C 中的#include 和#define，以及报告出现的不属于源程序语言的外来词。例如，词法分析还可能执行从小写字母到大写字母的转换。词法分析器通常将称为记号(token)的一串整数传递给语法分析阶段。这样，每个程序关键字、变量、数字和符号都转换成了整型量，编译器的后续阶段就不需要担心程序的文本。词法分析阶段通常使用有限自动机实现。

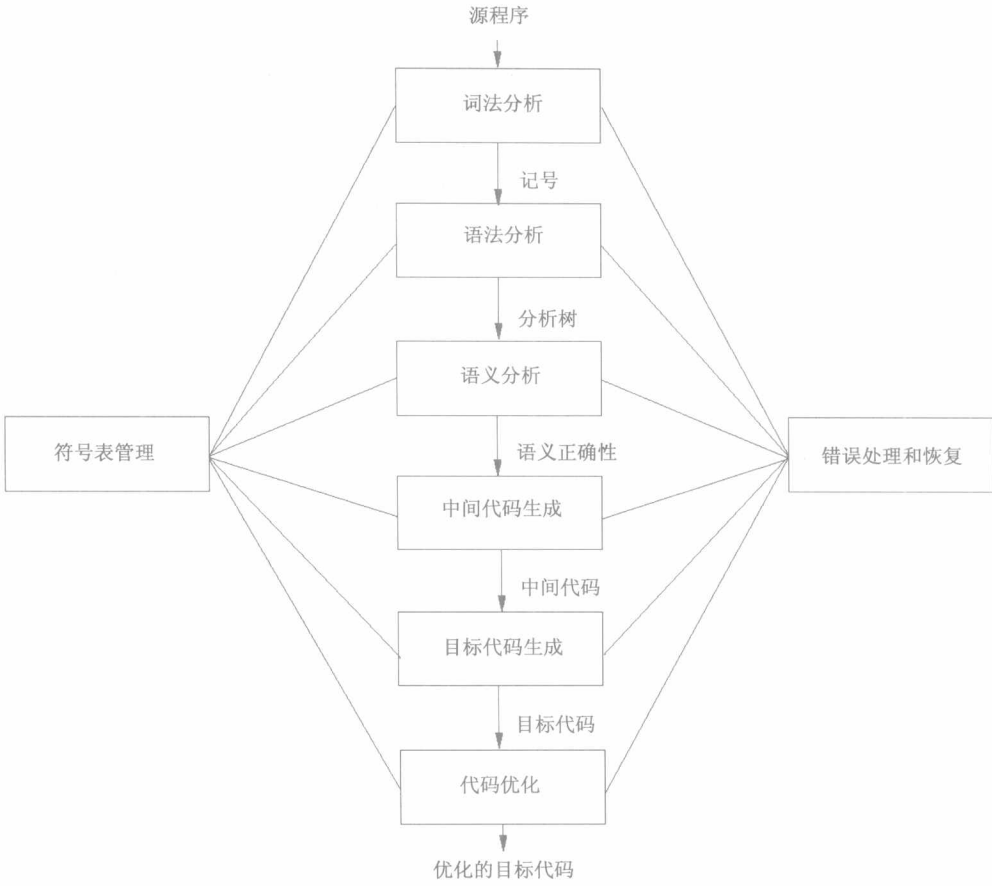


图 1-2　编译器的各个阶段

(2) 语法分析。语法分析阶段从词法分析器中获取单词或记号，然后检查输入程序的语法正确性。如果能标识一个序列的语法规则，并能从语法中特定的起始符推出输入程序，那么语法分析阶段是成功的。在这种情况下会生成树的表现形式，即分析树，该树用于描述将程序中的一系列单词组合并建立语法正确的程序的方式。如果不能构造这样的分析树，则输入的程序在语法上是错误的，并显示适当的错误信息给用户。词法和语法分析阶段是联系在一起的。当语法分析器需要更多的记号时，首先要求词法分析器依次扫描当前位置的程序来标识紧临的下一个记号，并将其返回给语法分析器，因此该阶段也被称为解析阶段。

(3) 语义分析。验证程序语法正确，即语法没有问题后，下一个任务就是检查语义正确性。因为程序的语义独立于语言，所以不可能列举出编译器包含的语义检查的类型。但是在程序中，对变量和表达式类型的检查很常见。强类型语言要求所有变量的类型在使用前声明。另一方面，弱类型或无类型语言则没有这个约束。当推导各类表达式时，通常需要检查两次。第一次是关于操作数运算符的适用性。例如，整数的除法仅当操作数都为整数时才允许，而字符串连接运算仅适用于字符串操作数等。第二次检查需要确定使用的变量的定义。许多编程语言允许相同的变量名称在程序中的多个位置使用。在这种情况下，相应的关于变量名称出现的定义由作用域规则决定。相对于动态作用域语言，在静态作用域语言中更容易实现检查，因为在前者中，正确的定义取决于程序的执行序列。

(4) 中间代码生成。该阶段是目标代码生成中的可选阶段。在该阶段，输入的源语言程序的相应代码依据一些假设的机器指令生成。该过程有助于将编译器生成的代码从一个处理器重定向到另一个处理器。用于中间代码生成的语言很容易被目前的绝大多数处理器支持，同时用该语言的语句也足以表达编程语言的结构。图 1-3 表达了使用中间代码为不同机器设计编译器的思想。需要再次注意的是，由词法分析器、语法分析器、语义分析器和中间代码生成器共同合作生成了源程序对应的中间语言代码。

(5) 目标代码生成。从中间代码到目标代码生成的任务是一种模板替换。对于中间语言的每条语句，预定义的目标语言模板用于生成最终代码。机器指令的可用性、寻址方式以及用于普通和特殊用途的寄存器数量对于设计这些模板起着重要的作用。需要花费些功夫将程序、用户自定义以及一些编译器生成的临时变量包装放入 CPU 寄存器中，这将加速程序的执行。

(6) 代码优化。这是目标代码生成中最重要的一步，使得代码更有效率。因为编译器的多个阶段是自动化的，会生成很多可以消除的冗余代码。为了进行优化，将代码划分成一些基本块。基本块是有着单独入口和出口的语句序列。当本地优化的操作仅限于单独的基本块时，则需要更多严密的跨越各个基本块边界的全局优化来生成高度优化的代码。最重要的需要进行代码优化的语句是循环语句。因为循环语句需要重复执行。在深度嵌套循环中，甚至减少一条单独语句都能显著地减少程序的运行时间。其他优化方法包括代数化简、存储和加载情况的消除(在这种情况下，前面语句计算出的变量存储在内存中，而接下来的指令是从内存中加载该变量)。在寄存器中保留变量的副本可以避免重新加载。大量其他代码优化的策略将在后面章节介绍。

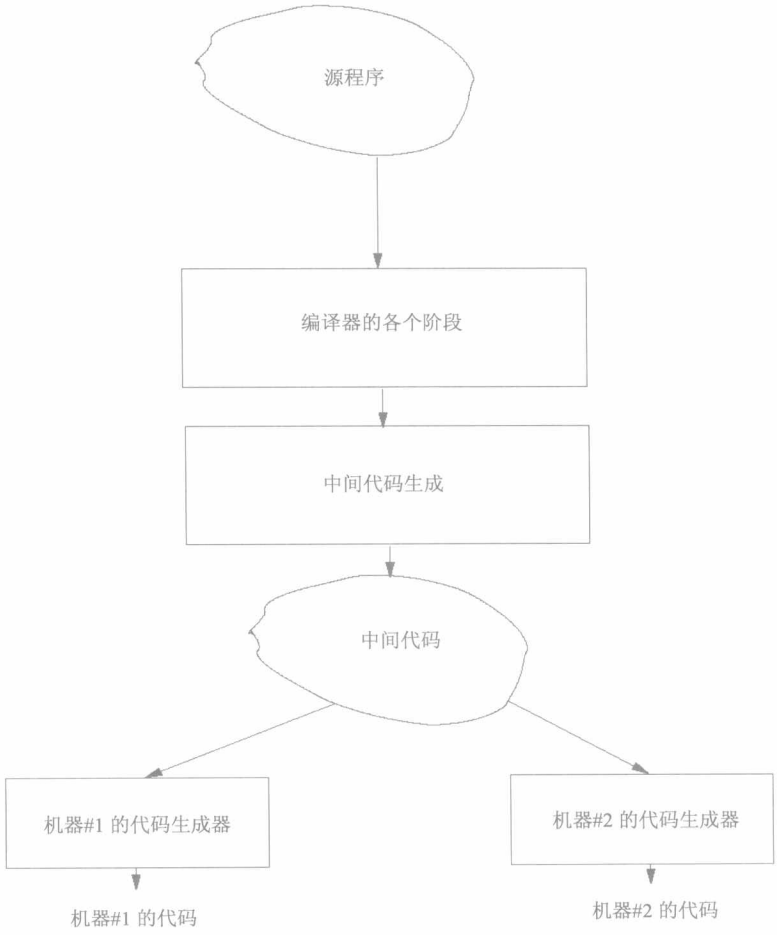


图 1-3　重定向到不同的机器

(7) 符号表管理。符号表是存放源程序中定义的所有符号信息的数据结构。虽然并不是最终生成代码的一部分，但在编译器的各个阶段都需要引用和使用该表。符号表中存储的信息一般包括变量名称、类型、大小以及在程序中的相对偏移等。通常在词法分析和语法分析阶段生成符号表。因为在编译过程中，经常需要在该表中查找变量的定义，良好的数据结构可以将搜索时间减到最少。根据语言的作用域规则，需要将符号表设计成一组多层次的表，而不是一张扁平式的表。如果语言要求先定义变量再使用它，那么存储在符号表中的信息可以用于检查变量是否定义。如果定义了新变量，该变量会被放入符号表，在发现该变量被使用时即会搜索符号表。

(8) 错误处理和恢复。虽然该阶段与代码生成的质量没有关系，但错误处理是判断编译器质量的一个标准。在出现语义错误的情况下，解析器依然在处理。但要是出现语法错误，解析器会进入错误状态。这种导致进入错误状态的变化会给出关于错误类型的指示。但是恢复并不是件容易的事，需要撤销解析器已完成的一些处理，或许还要丢弃一些记号来到达能继续处理的状态。如果没有进行恢复，解析器本身会出现一个错误，用户需要去纠正。但是，因为可能会存在很多这样的错误，则用户每次都要花费大量时间去纠正这个单独的错误。因此，错误恢复在整个编译器操作中起着至关重要的作用。

1.4 编译器设计的难点

编译器设计者会面对很多设计的难点，从语言的细节到底层硬件的特点，以及编译器的用户友好性。下面将介绍一些影响编译器设计过程的因素。

(1) 语言的语义。源语言结构的语义对编译器的编写造成了难题，因为其语义结构很复杂。这类结构的典型示例有 case、loop、break 和 next 等。case 语句的语义在不同语言中变化很大。有些语义支持落空(fall-through)语义，有些则不支持。loop 比较难，因为在某些语言中，循环体后的索引变量值取决于执行的序列。对于循环的正常退出，变量根据语言因素可能包含最后的赋值或未定义的值；对于循环的非正常退出，索引值通常保留。像 break 和 next 这样的语句会修改程序的执行序列，这需要根据实际的执行情况而定。编译器设计者必须在生成代码时考虑这些因素。

(2) 硬件平台。底层机器的体系结构也影响着编译过程。例如，基于累加器的机器的代码生成策略不同于基于堆栈的机器，这是因为，基于累加器的体系结构对于所有的运算、其中的操作数以及目标都是累加器，而对于基于堆栈的机器，当操作数弹出堆栈时，运算即执行，结果仍然被压回堆栈中。处理器的指令集也会影响代码生成和优化的过程。对于 CISC(Complex Instruction Set Computer，复杂指令集计算机)体系结构，机器的指令相当复杂，单条指令能做很多工作。而对于 RISC(Reduced Instruction Set Computer，精简指令集计算机)机器，单条指令在大小和执行时间方面都很简单和有规则。在 CISC 机器中，指令的大小是可变的，且需要可变的执行时间。因此编译器编写者更喜欢 RISC 机器，因为规则的小尺寸指令可以作为基本的构建块来构成目标代码。RISC CPU 中存在的大量寄存器有助于将大多数临时变量分配到寄存器中，所以在 RISC 机器中，经常能创建优于 CISC 机器的代码。编译器编写者需要仔细了解处理器的特性来指定代码生成的策略。

(3) 操作系统和系统软件。编译器最终生成的目标代码将在操作系统(Operating System，OS)创建的环境中执行。执行的文件格式是由操作系统或者更具体的加载程序来

描述的。因此编译器的输出应与其相兼容。要完成源语言程序，必须将语言和库例程耦合起来，该过程称为链接。在一个系统中可能存在很多编译器，分别用于不同的语言，但是只会有一个链接程序。因此，所有这些编译器的输出应与链接程序兼容，以使程序的不同模块可以用不同语言编写，并使用不同编译器转换成对象文件，最后链接程序将这些文件链接起来并生成可执行文件。

(4) 错误处理。应当指出，错误处理是编译器中一个最重要的功能。对于编译器设计者来说，这里真正的难点在于向用户显示合适的错误信息，能精确显示错误，同时也不能太冗长而使用户混淆。例如，当在语句结束处缺少分号时，应显示错误消息“<line no.> ; expected”，而不是简单地显示“syntax error”。如果报告变量 x 在某个位置没有定义，那么在接下来引用 x 的行中则没必要重复报告相同的错误，除非 x 定义的作用域改变。我们可以为特定的语言构造一些这样的示例。编译器设计者面临的真正挑战是要想出用户可能遇到的错误类型以及各种情况，然后设计出合适的检测和恢复机制。有些编译器为了纠正错误，甚至需要修改部分源程序。

(5) 辅助调试。调试是检测程序中逻辑错误的一项很重要的技术。这包括单步执行程序测试和设置程序的变量值以及在程序中设置断点等。这里最重要的问题是，虽然处理器执行程序使用的是机器语言，但用户需要使用源语言来控制，这就需要编译器生成关于源语言程序语句和相应的机器指令之间通信的额外信息，也同样需要符号表信息以供调试器使用。代码生成过程中额外的修改可能需要在目标代码中嵌入额外的调试指令。因此为了使编译器生成的代码适合调试，编译器设计者必须要采取适当的步骤。

(6) 优化。优化也是一个真正的难题，因为编译器设计者需要找出一套转换机制，使其可能有利于用某一种语言编写大多数程序。这不是简单的任务，因为这取决于该语言的结构、目标机器的体系结构，甚至是指定编译器编译的这种语言所开发的应用程序的类型。这里一个很重要的问题是转换应当安全，即优化转换不应修改未优化的程序所做的计算。在优化程序的工作量和执行时间的改进之间经常要进行权衡。大多数编译器设计成具有几种不同层次的优化以供用户使用。选择调试选项可能会禁用任何在目标文件中修改指令顺序的优化，因为这些会干扰源程序和目标代码间的通信。

(7) 运行环境。运行环境涉及到子程序的处理，它需要为子程序中传递的参数和本地变量创建空间。对于早期的语言如 Fortran，环境是静态的。这意味着对每个本地变量和传递的参数，都由编译器在内存中为其分配了固定位置。因为这类语言不支持递归，所以这就足够了。在任何时间，都只会存在至多一个活动的子程序。而对于支持递归的语言，运行环境的管理非常关键，通常通过压入栈框架(frame)来存放传递的参数和本地变量。每次调用子程序时，都将该框架压入堆栈中，并在返回时弹出。因此，编译器设计者需要查看有关子程序的语言特性，设计出有效处理运行环境的模式。

(8) 编译速度。这是判断用户对编译器接受程度的最重要的一条标准。在程序的开发初期，存在很多语法和逻辑缺陷。因此，程序的逻辑经常需要修改。该阶段的编写者更关注他们代码的正确性，而不是运行的效率。因此尽管编译器能编译得很快，但产生的代码可能效率低下。而在接近程序开发的最后阶段时，开发者需要投入更多精力来使程序最有效率。相应地，编译器应在花费更多的编译时间的代价下做一些好的优化工作。因此，所开发的编译器应能很快地产生未优化的代码，而优化代码生成可能需要一些额外的时间来编译。

1.5 编译过程—— 示例

本节来看一个程序片段的整个编译过程，该程序段是用类 Pascal 语言编写的，非常简单。先考虑以下程序：

```
program
   var X1, X2: integer; { integer variable declaration }
   var XR1: real;       { real variable declaration }
begin
   XR1 := X1 + X2 * 10; {assignment statement }
end.
```

1. 词法分析

词法分析器扫描程序，寻找称为记号的主要词法元素。对于以上示例，词法分析器产生如下序列的记号：

```
program, var, X1, ',', X2, ':', integer, ';', var, XR2, ':', real, ';', begin,
XR1, ':=', X1, '+', X2, '*', 10, ';', end, '.'
```

词法分析器会丢弃和去除空格(空白、制表符和换行)和注释。

2. 语法分析

语法分析阶段分析了程序的语法正确性。对语法正确的程序将产生分析树。假定给出的程序语法正确，则在分析语法规则和程序后所产生的分析树如图 1-4 所示。

3. 代码生成

代码生成过程遍历了分析树，然后生成输入程序的代码。假定使用了面向堆栈的机器，