第 1 章 编译概观

1

本章概述

编译器是一种计算机程序，负责将一种语言编写的程序转换为另一种语言编写的程序。同时，编译器也是一种大型软件系统，包括许多内部组件和算法及其之间复杂的交互。因而，学习编译器构建也就是学习用于转换和改进程序的技术，同时也是一项软件工程实践。本章从概念上概述现代编译器的所有主要组件。

关键词：编译器；解释器；自动转换

1.1 简介

计算机在日常生活中的作用逐年俱增。随着互联网的崛起，计算机及运行于其上的软件提供了通信、新闻、娱乐和安全。嵌入式计算机改变了我们制造汽车、飞机、电话、电视和无线电的方法。从视频游戏到社交网络，计算已经建立了全新的活动范畴。超级计算机预测每日的天气和暴风雨的发展过程。嵌入式计算机可以指挥红绿灯的同步，可以向你的掌上电脑发送电子邮件。

所有这些计算机的应用都依赖软件计算机程序，软件程序基于硬件提供的底层抽象建立了虚拟的工具。几乎所有的软件都是通过称为编译器的工具转换而来。编译器也只是一个计算机程序，它转换其他的计算机程序，并使之准备好执行。本书讲述了用于建立编译器的自动转换的基本技术。本书还

描述了编译器构建中出现的许多挑战，以及编译器编写者用于解决这些问题的算法。

编译器
用于转换其他计算机程序的计算机程序。

1. 概念路线图

编译器是一种工具，将一种语言编写的软件转换为另一种语言。为将文本从一种语言转换为另一种语言，该工具必须理解输入语言的形式和内容，或者说语法和语义。它还需要理解输出语言中支配语法和语义的规则。最后，它需要一种方案，以便将内容从源语言映射到目标语言。

典型编译器的结构，通常即衍生于这些简单的观察。编译器有一个前端，用于处理源语言。它还有一个后端，用于处理目标语言。为将前端和后端连接起来，编译器有一种形式化的结构，它用一种中间形式来表示程序，中间形式的语言很大程度上独立于源语言和目标语言。为改进转换，编译器通常包括一个优化器，来分析并重写中间形式。

2. 概述

计算机程序只是用一种程序设计语言编写的抽象操作的序列，程序设计语言是设计用来表示计算的形式语言。不同于自然语言（如中文或葡萄牙文），程序设计语言有着精确的性质和语义。程序设计语言被设计得富有表达力、简洁且清晰，而自然语言允许二义性。程序设计语言应该避免二义性，二义的程序没有语义。程序设计语言应该能指定计算，即可以记录下执行某些任务或产生某些结果所需的行为序列。

一般来说，程序设计语言设计成能允许人类将计算表达为操作的序列。而计算机处理器，下文称为处理器、微处理器或机器，则设计成能执行操作序列。与程序设计语言规定的操作相比，处理器实现的操作在很大程度上是在一个低得多的抽象层次上。例如，程序设计语言通常包括一种将数字输出

到文件的简洁方法。在这个单一的程序设计语言语句能够执行前，它必须被逐字地转换为数百个机器操作。

执行这种转换的工具被称为编译器。编译器将以某种语言编写的程序作为输入，产生一个等价的程序作为输出。对于经典意义上的编译器来说，输出程序用某个特定处理器上可用的操作表示，输出程序所针对的处理器通常称为目标机。如果当做一个黑盒子，编译器看起来可以是这样：

通常的“源”语言可能是C、C++、FORTRAN、Java或ML。“目标”语言通常是某种处理器的指令集。

指令集

处理器支持的操作的集合，指令集的总体设计通常称为指令集系统结构（Instruction Set Architecture，ISA）。

一些编译器产生的目标程序是某种面向人类的程序设计语言，而非某种计算机的汇编语言。这些编译器产生的程序需要更进一步的转换，才能在计算机上直接执行。许多研究性编译器产生C程序作为其输出。因为在大多数计算机上都有C编译器可用，这使得目标程序可以在所有这些系统上执行，代价是需要一次额外的编译才能得到最终的目标程序。目标为程序设计语言而非计算机指令集的编译器，通常称为源到源的转换器。

许多其他系统也可视为编译器。例如，可以认为产生PostScript的排版程序是一个编译器。它将文档在印刷纸面上如何布局的规格作为输入，产生PostScript文件作为输出。PostScript只是一种描述图像的语言。因为排版程序的输入是一种可执行的规格，并生成另一种可执行的规格，因此它是一个

编译器。

将PostScript转换为像素的代码通常是一个解释器，而不是编译器。解释器将一种可执行规格作为输入，产生的输出是执行该规格的结果。

一些语言，如Perl、Scheme和APL，更多是用解释器实现，而不是编译器。

一些语言采用的转换方案，既包括编译，也包括解释。Java从源代码编译为一种称为字节码的形式，这是一种紧凑的表示，意在减少Java应用程序的下载时间。Java应用程序是通过在对应的Java虚拟机（JVM）上运行字节码来执行的，JVM是一种字节码的解释器。许多JVM的实现包括了一个运行时执行的编译器，有时称为JIT（just-in-time）编译器，它将频繁使用的字节码序列转换为底层计算机的本机码，这使得上文描述的图景进一步复杂化。

虚拟机

虚拟机是针对某种处理器的模拟器，它是针对该机器指令集的解释器。

解释器和编译器有许多共同之处，它们执行许多同样的任务。二者都要分析输入程序，并判定它是否是有效的程序。二者都会建立一个内部模型，表示输入程序的结构和语义。二者都要确定执行期间在何处存储值。然而，解释代码来产生结果，与输出转换后可以执行的目标程序来产生结果，二者有很大不同。本书专注于构建编译器过程中可能出现的问题。但这里讲述的大部分题材，解释器的实现者可能也会发现有用。

3. 为何研究编译器的构建

编译器是一个庞大、复杂的程序。编译器通常包括数十万行代码（即使没有上百万行），组织为

多个子系统和组件。编译器的各个部分以复杂的方式进行交互。对编译器一部分作出的设计决策，对其他部分有着重要的影响。因而，编译器的设计和实现，是软件工程中一项很有分量的实践活动。

一个好的编译器是自成天地的，包含了整个计算机科学的一个映像。编译器实际运用了贪心算法（寄存器分配）、启发式搜索技术（表调度）、图算法（死代码消除）、动态规划（指令选择）、有限自动机和下推自动机（词法分析和语法分析）以及不动点算法（数据流分析）。它处理诸如动态分配、同步、命名、局部性、分级存储结构管理和流水线调度等问题。很少有软件系统能汇集同样多且复杂的组件。处理编译器的内部设计和实现在软件工程方面所获取的难得的实践经验，是那些规模较小、复杂度较低的系统所无法提供的。

在计算机科学的主要活动中，编译器发挥着根本的作用：为通过计算机解决问题而做好准备。大多数软件都需要通过编译，该过程的正确性和结果代码的效率，对我们构建大型系统的能力有着直接影响。大多数学生无法满足于仅仅通过阅读而了解这些思想，其中的许多东西都必须得亲自实现才能肯定其价值。因而，学习编译器构建是计算机科学教育的一个重要部分。

编译器是成功将理论应用到实际问题的范例。自动产生词法分析器和语法分析器的工具应用了形式语言理论的结果。这些工具同样可用于文本搜索、网站过滤、文字处理和命令行语言解释器。类型检查和静态分析应用了格理论、数论和其他数学分支的结果，以理解并改进程序。代码生成器使用了树模式匹配、语法分析、动态规划和文本匹配的算法，来自动化指令选择的过程。

但编译器构建领域出现的一些问题仍然未解决，即当前的最佳解决方案仍有改进的余地。尝试设计高级的通用中间表示的努力因复杂性而宣告失败。用于指令调度的主导算法是一个贪心算法，其中包含了几层不相上下的启发式逻辑。编译器显然应该利用交换性和结合性来改进代码，但大多数试图

这样做的编译器都只是将表达式重排为某种规范次序。

构建成功的编译器需要精通算法、工程和计划。好的编译器会对困难问题近似求解，它们强调效率，无论是自身的实现还是生成的代码。它们的内部数据结构和知识表示暴露的细节不多不少，既足够进行强有力的优化，而又不会使编译器沉湎于细节。编译器构建汇集了整个计算机科学领域中的思想和技术，将其应用到受限的环境下，以解决某些真正困难的问题。

4. 编译的基本原则

编译器是庞大、复杂、细致工程化的对象。虽然编译器设计领域中的许多问题有多种解决方案和解释，但编译器编写者必须始终铭记两个基本原则。第一个原则是不能违反的：

编译器必须保持被编译程序的语义。

正确性是程序设计中的一个根本问题。编译器必须忠实地实现其输入程序的“语义”，以保持其正确性。这个原则是编译器编写者和编译器用户之间社会契约的核心。如果编译器可以随意处理语义，那么何不直接生成一个`nop`或`return`呢？如果不正确的转换是可接受的，那么为何要费力做对呢？

编译器必须遵守的第二个原则是实用：

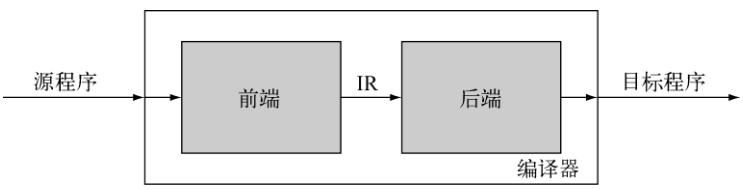
编译器必须以某种可觉察的方式改进输入程序。

传统的编译器通过使输入程序可以在某种目标机上直接执行来进行改进。其他“编译器”对输入程序的改进各有不同的方法。例如，tpic是一个程序，以作图语言pic编写的绘图规格作为输入，将其转换为LATEX，这里的“改进”在于LATEX具有更好的可用性和一般性。而C语言的源到源转换器生成的代码在一定程度上要优于输入程序，否则，谁还会调用它呢？

1.2 编译器结构

编译器是一个庞大、复杂的软件系统。编译器社区自1955年以来一直在构建编译器，多年以来，关于如何设计编译器的结构，我们已经学习了许多经验教训。早期，我们将编译器描述为一个简单的盒子，能够将源程序转换为目标程序即可。当然，现实比这个简单的图景更为复杂。

单盒模型指出，编译器必须理解输入源程序并将其功能映射到目标机。这两项任务截然不同的性质暗示着一种可能的任务划分，并最终导致了一种将编译分解为两个主要部分的设计：前端和后端。



前端专注于理解源语言程序。后端专注于将程序映射到目标机。对于编译器的设计和实现，这种关注点的分离隐含着几个重要结论。

前端必须将其对源程序的认识编码到某种结构中，以供后端稍后使用。中间表示（IR）成为了编译器对所转换代码的权威表示。在编译过程中的每个点，编译器都有一个权威表示。实际上，随着编译过程的进展，可以使用几种不同的IR，但在每个点上，都只有一种表示会成为权威的IR。我们可以将权威IR看做编译器各个独立阶段之间所传递程序的版本，就像是前述图中从前端传递到后端的IR一样。

IR

编译器使用一些数据结构来表示它处理的代码，这种形式称为中间表示（Intermediate Representation，IR）。

在一个两阶段编译器中，前端必须确保源程序是良构的，而且必须将输入的代码映射到IR。后端必须将IR程序映射到目标机的指令集和有限的资源上。由于后端仅处理前端生成的IR，因此它可以认为IR不包括任何语法和语义错误。

生在这个时代真好

就编译器的设计和实现而言，这是一个令人激动的时代。在20世纪80年代，几乎所有的编译器都是庞大的单块式系统。它们将一种语言作为输入，产生针对某种特定计算机的汇编代码。生成的汇编代码与其他编译过程产生的代码（包括系统库和应用程序库）粘贴在一起，最终形成一个可执行文件。这个可执行文件存储在磁盘上，最终的可执行代码在适当的时间从磁盘移动到内存并执行。

今天，编译器技术应用于各种不同环境下。随着计算机在各种不同的场合获得应用，编译器必须处理新的不同的限制。速度不再是用于判断编译后代码的唯一标准。当今，判断代码质量的规则很可能是代码“占地”有多小，代码执行时耗能多少，代码能压缩到多小，代码执行时产生多少个缺页异常，等等。

同时，编译技术已经脱离了80年代单块式系统的窠臼，它们出现在许多新场合。Java编译器采用部分编译的程序（Java“字节码”格式）作为输入，并将其转换为目标机的本机码。在这种环境中，成功意味着编译时间加上运行时间的总和必须小于解释执行的代价。分析整个程序的技术正在从编译时转向链接时，链接器可以分析整个应用程序的汇编代码并利用该认识来改进程序。最后，可以在运行时调用编译器生成定制的代码，以利用仅能从运行时得知的事实获利。如果编译花费的时间保持在比较短的水准上，而生成定制代码的获利较大，那么这种策略可以产生显著的改进。

在输出目标程序之前，编译器可以在代码的IR形式上进行多趟迭代。多遍迭代可能会生成更好的代码，因为编译器实际上可以在一个阶段中研究代码并记录相关细节。那么在后续阶段中，编译器可以利用这些记下的知识来提高转换的质量。这种策略要求第一趟获得的知识记录在IR中，后续各趟可以查找并利用这些知识。

最后，两阶段结构可以简化编译器重定目标的过程。我们可以轻易地想象到为单个前端构建多个后端，这样即可产生输入同一语言但输出针对不同目标机器的编译器。类似地，我们可以想象针对不同语言的前端生成同样的IR并使用共同的后端。这两种场景，都假定一种IR可以服务于几种源和目标

的组合，实际上，特定于语言和特定于机器的细节通常都会进入到IR。

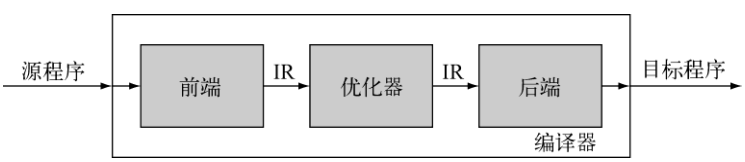
重定目标

改变编译器使之针对新处理器生成代码的任务，通常称为将该编译器重定目标。

引入IR使得可以向编译增加更多阶段。编译器编写者可以在前端和后端之间插入第三个阶段。这个中间部分，也称为优化器，以IR程序作为其输入，产生一个语义上等价的IR程序作为其输出。通过使用IR作为接口，编译器编写者插入第三个阶段时，可以将对前端和后端的破坏降到最低。这就形成了下述的编译器结构，称为三阶段编译器。

优化器

编译器的中间部分称为优化器，负责分析并转换IR，以改进IR。



优化器是一个IR到IR的转换器，试图在某些方面改进IR程序。（请注意，依据我们在1.1节中的定义，这些转换器本身就是编译器。）优化器可以对IR处理一遍或多遍，分析IR并重写IR。优化器重写IR可以使后端生成一个可能更快速或更小的目标程序。优化器还可能有其他目标，诸如产生更少缺页异常或耗能较少的程序。

概念上，三阶段结构表示了经典的优化编译器。实际上，每个阶段内部都划分为若干趟。前端由两趟或三趟组成，处理识别有效源语言程序的各种细节，并产生该程序的初始IR形式。中间部分包含执行不同优化的各趟处理。这些趟的数目和目的因编译器而异。后端由若干趟处理组成，每一趟都将输入的IR程序进一步处理，使之更接近目标机的指令集。编译器的三个阶段和其中的各趟处理共享了

同一个基础设施，这些结构如图1-1所示。

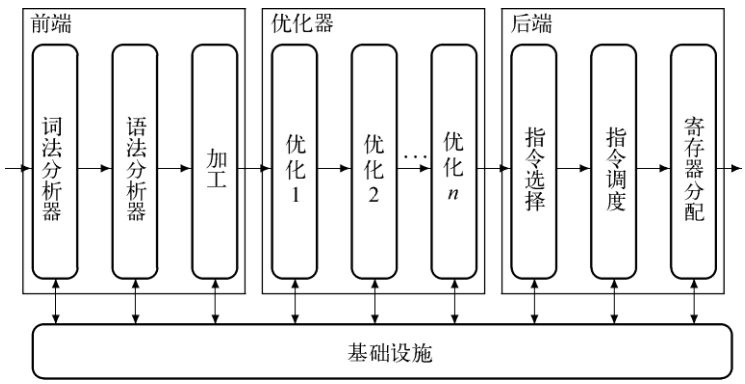


图1-1　典型编译器的结构

将编译器在概念上划分为前端、中间部分（优化器）以及后端在实践中是很有用的做法。这些阶段解决的问题各有不同。前端的工作涉及理解源程序并将其分析结果以IR形式记录下来；优化器部分专注于改进IR形式；后端必须将转换过的IR程序映射到目标机的有限资源上，从而能够有效利用那些资源。

在这三个阶段中，优化器的描述最为模糊不清。优化这个术语，暗含编译器需要找到某个问题的最优解的意思。优化过程中发生的问题是如此之错综复杂，因而这些问题实际上是得不到最优解的。需要进一步澄清的是，编译后代码的实际性能取决于优化器和后端两个阶段中应用的所有技术之间的相互影响。因而，即使单个的某项技术可以被证明是最优的，但它与其他技术的交互可能产生次优的结果。因此，相对于未优化的代码版本而言，良好的优化编译器可以改进代码的质量。但优化编译器通常无法产生最优代码。

中间部分可以是单块式的一趟处理，其中应用一个或多个优化技术来改进代码，也可以从结构上设计为若干趟规模较小的处理过程，其中每趟都读写IR。单体式结构可能更为高效；多趟结构有助于

降低实现的复杂度，同时在调试编译器时也相对简单。此外还提高了灵活性，可以在不同情况下采用不同的优化技术组合。这两种方法之间的选择，取决于构建编译器时和编译器实际运行时的约束条件。

1.3 转换概述

在编译器将某种程序设计语言编写的代码转换为适合于在某种目标机上执行的代码时，要运行许多步骤。

符号表示法

本质上，讲解编译器的书是关于符号表示法的。毕竟，编译器是将一种符号表示法编写的程序转换为另一种符号表示法编写的等价程序。在你阅读本书过程中，可能有若干符号表示法方面的问题。有时候，这些问题将直接影响你对书中内容的理解。

阐述算法　我们试图保持算法的简洁。算法的描述是在一个比较高的层次上进行的，假定读者可以提供缺失的实现细节。算法的描述文本以斜体无衬线字体印刷。其中的缩进是作者有意进行，同时具有重要的含义，在`if-then-else`控制结构中，缩进用处最大。`then`或`else`之后缩进的代码形成一个块语句。在下列代码片段中

```
if Action [s,word] = "shift si" then
   push word
   push si
   word ← NextWord()
else if ···
```

`then`和`else`之间所有的语句都是`if-then-else`结构中`then`子句的一部分。当`if-then-else`结构中的一个子句仅包含一个语句时，我们将关键字`then`或`else`与该语句写在同一行。

编写代码　在某些例子中，我们给出以某种选定的语言编写的实际程序文本，以说明特定的要点。实际程序文本以等宽字体印刷。

算术运算符　最终，除了实际程序以外，我们放弃了用*代替×和/代替÷的做法。对读者来说，相关运算符的含义应该是清楚的。

为使这个抽象过程更为具体，考虑为下列表达式生成可执行代码需要的步骤：

$$a \leftarrow a \times 2 \times b \times c \times d$$

其中a、b、c和d都是变量，← 表示赋值操作，×运算符表示乘法。在以下各小节中，我们会追溯编译

器将这个简单表达式转换为可执行代码所经历的路径。

1.3.1　前端

在编译器能够将表达式转换为可执行的目标机代码之前，它必须理解表达式的形式和内容，或语法和语义。前端根据语法和语义，判断输入代码是否是良构的。如果前端发现代码是有效的，它会以编译器的IR格式来建立该代码的一个表示，否则，它向用户回报诊断错误信息，以标识该代码的问题。

1. 检查语法

为检查输入程序的语法，编译器必须将程序的结构与语言的定义进行比较。这需要一种适当的形式化定义，一种检测输入是否满足该定义的高效机制，和如何继续处理无效输入的相关规划。

数学上，源语言是一个字符串的集合，通常是无限集，由某种规则的有限集定义，后者称为语法。前端中有两趟独立的处理，分别称为词法分析器和语法分析器，来判断输入代码实际上是否属于语法定义的有效程序集合。

程序设计语言语法通常基于词类来引用单词，词类有时称为语法范畴。将语法规则基于词类划分使得单一规则能够描述许多句子。例如在英语中，许多句子有下述形式：

Sentence（句子）→ *Subject*（主语）`verb`（动词）*Object*（宾语）`endmark`（结束标点符号）

其中`verb`和`endmark`是词类，而*Sentence*、*Subject*和*Object*是语法变量。*Sentence*表示任何具有该规则所描述形式的字符串。符号→读做“导出”，意味着右侧的一个实例可以抽象为左侧的语法变量。

考虑形如“Compilers are engineered objects.”的句子。理解这个句子的语法时，第一步是标识输入程序中的各个单词，并将每个单词归入对应的词类。在编译器中，该项任务属于称为词法分析器的

一趟处理过程。词法分析器以字符流为输入，并将其转换为已归类单词的流，已归类的单词是形如(*p*, *s*)的对，其中*p*是单词的词类，而*s*是单词的拼写。对应上述例句，词法分析器会将其转换为下述已归类单词的流：

(noun,"Compilers"), (verb,"are"), (adjective,"engineered"),
(noun,"objects"), (endmark,".")

词法分析器

编译器中的一趟，将字符构成的串转换为单词构成的流。

实际上，单词的实际拼写可能保存在散列表中，而(*p*, *s*)对中的*s*可以用一个整数索引表示，以简化相等性测试。第2章将探讨词法分析器构建的理论和实践。

在下一步中，编译器试图根据指定了输入语言语法的规则，来匹配已分类单词的流。例如，实际英语知识可能包括以下语法规则：

1	*Sentence*	→	*Subject* verb *Object* endmark
2	*Subject*	→	noun
3	*Subject*	→	*Modifier* noun
4	*Object*	→	noun
5	*Object*	→	*Modifier* noun
6	*Modifier*	→	adjective
	...		

通过观察，我们可以为前述例句找到下列推导（derivation）：

规则	原型句子
—	*Sentence*
1	*Subject* verb *Object* endmark
2	noun verb *Object* endmark
5	noun verb *Modifier* noun endmark
6	noun verb adjective noun endmark

推导从语法变量*Sentence*开始。在每一步，它都重写原型句子中的一项，将其替换为可以从规则推导出的某个右侧项。第一步使用规则1替换*Sentence*。第二步使用规则2替换*Subject*。第三步使用规

则5替换*Object*，而最后一步根据规则6将*Modifier*重写为`adjective`。此时，通过推导生成的原型句子，可以与词法分析器产生的已分类单词流匹配。

上述推导证明了句子“Compilers are engineered objects.”属于由规则1到6描述的语言。该句子的语法是正确的。自动查找推导的过程称为解析（或语法分析，parsing）。第3章给出了编译器用于解析输入程序的技术。

语法分析器
　　编译器中的一趟，判断输入流是否是源语言的一个句子。

语法正确的句子可能是无意义的。例如，句子“Rocks are green vegetables.”与“Compilers are engineered objects.”按相同的顺序使用了同样的词类，但前一个句子没有合理的语义。若要理解这两个句子之间的差别，需要关于软件系统、岩石和蔬菜的上下文知识。

编译器用于推断程序设计语言的语义模型比理解自然语言所需的模型要简单。编译器会构建数学模型来检测程序中各种不一致之处。编译器会检查类型一致性，例如，下述表达式

$$a \leftarrow a \times 2 \times b \times c \times d$$

在语法上可能是良构的，但如果`b`和`d`是字符串，这个语句仍然可能是无效的。编译器在特定的情况下也检查数字的一致性，例如，引用数组时应该使用与数组声明的阶/秩（rank）相一致的维数，而过程调用指定的参数数目应该与过程的定义一致。第4章探讨了基于编译器的类型检查和语义加工过程中可能发生的一些问题。

类型检查
　　编译器中的一趟，检查输入程序中对名字的使用在类型方面是否一致。

2. 中间表示

$$
\begin{aligned}
t_0 &\leftarrow a \times 2 \\
t_1 &\leftarrow t_0 \times b \\
t_2 &\leftarrow t_1 \times c \\
t_3 &\leftarrow t_2 \times d \\
a &\leftarrow t_3
\end{aligned}
$$

编译器前端处理的最后一个问题是生成代码的IR形式。编译器可以使用各种不同种类的IR，这取决于源语言、目标语言和编译器应用的各种特定的转换。一些IR将程序表示为图。其他的类似于有序的汇编代码程序。右侧的代码给出了我们的示例表达式在底层的顺序IR中可能的表示。第5章概述了编译器使用的各种IR。

对于源语言中的每一种结构，编译器都需要一种策略，指定如何用代码的IR形式实现该结构。具体的选择可能会影响到编译器转换和改进代码的能力。因而，我们将花费两章的篇幅来探讨为源代码结构生成IR过程中发生的各种问题。过程链接既是造成最终代码低效的“罪魁祸首”，也是将不同源文件拼接为应用程序的“黏合剂”。我们在第6章讨论围绕过程调用的一些问题。第7章阐述了对应大多数其他程序设计语言结构的实现策略。

术 语

细心的读者会注意到，书中许多处使用了“代码”这个词，在相应的上下文中，使用“程序”或“过程”也比较适合。我们调用编译器来转换代码的片段，“片段”的范围很广泛，既可以是单个的引用，也可以是很多程序构成的整个系统。我们将沿用意义较模糊但更通用的术语“代码”，而不会为编译设定范围。

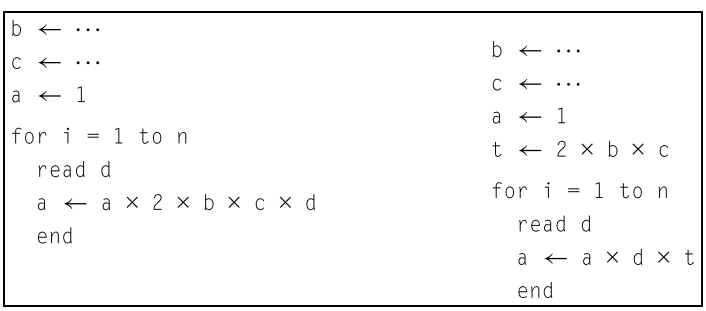
1.3.2 优化器

在前端为输入程序产生IR表示时，它按语句在源代码中的顺序，每次处理一个语句。因而，初始IR程序包含了通用的实现策略，在编译器可能产生的任何上下文中都可以工作。但在运行时，代码将在更为受限、也更可预测的上下文中执行。优化器分析代码的IR形式，以发现有关上下文的事实，并利用此项上下文相关知识来重写代码，使之能够以更有效的方式来算得同样的答案。

效率可以有许多含义。优化的经典观念是减少应用程序的运行时间。在其他上下文中，优化器可

能试图减少编译后的代码长度，或致力于其他性质的改进，如处理器执行代码的耗能情况。所有这些策略都以效率为目标。

返回到我们的例子，在图1-2a所示的上下文中考虑它。该语句发生在循环内部，在它使用的值中，只有a和d在循环内部改变。2、b和c的值在循环中是不变的。如果优化器发现这个事实，它可以如图1-2b所示重写代码。在这个版本中，乘法操作的数目从4·n下降到2·n + 2。对于n>1来说，重写的循环应该执行得更快速。这种优化将在第8章、第9章和第10章讨论。



(a) 上下文中的原始代码　　(b) 改进过的代码

图1-2　上下文的作用

1. 分析

大多数优化都包括分析和转换两个过程。分析判断编译器可以在何处安全地应用优化技术且有利可图。编译器使用几种分析技术来支持转换。数据流分析在编译时推断运行时值的流动。数据流分析器通常需要解一个联立方程组，该方程组是根据被转换代码的结构得出的。相关性分析（dependence analysis）使用数论中的测试方法来推断下标表达式的可能值。它用于消除引用数组元素时的歧义。第9章详细地考察了数据流分析及其应用，以及静态单赋值形式的构建，静态单赋值形式是一种IR，可以将值和控制的流动信息直接编码在IR中。

数据流分析

编译时一种对运行时数据值流动的推断。

2. 转换

为改进代码，编译器不能只分析代码。编译器还必须使用分析的结果来将代码重写为一种更高效的形式。此前已经发明了无数的转换技术，用于改进可执行代码的时间或空间需求。其中一些，例如找到循环中不变的计算并将其移动到不那么频繁执行的位置，可以改进程序的运行时间。其他的转换可以使代码更为紧凑。转换的效果、它们能够运作的范围以及支持转换所需的分析技术，都各有不同。有关转换的文献资料十分丰富，主题十分广泛和深入，足以写一本或几本单独的书来进行探讨。第10章涵盖了标量转换的主题，即意在改进单处理器上代码性能的转换。其中给出了一个组织该主题的分类法，并提供了相关的例子，以充实分类法的内容。

1.3.3 后端

编译器的后端会遍历代码的IR形式并针对目标机输出代码。对于每个IR操作，后端都会选择对应的目标机操作来实现它。同时后端会选择一种次序，使得操作能够高效执行。后端还会确定哪些值能够驻留在寄存器中，哪些值需要放置到内存中，并插入代码来实施相应的决策。

关于ILOC

在整本书中，相对底层的例子都是以一种称为ILOC的符号表示法写成的，ILOC是“Intermediate Language for an Optimizing Compiler”的首字母缩写词。多年以来，该符号表示法经历了许多变更。本书中使用的版本在附录A中有详述。

可以将ILOC看做某种简单RISC机器的汇编语言。它有一组标准的操作，大多数操作以寄存器为参数。内存操作如`load`和`store`，是在内存和寄存器之间传输数据值。为简化文中的阐述，大多数例子假定所有数据都是整数。

每个操作都有一组操作数和一个结果。操作分为五部分：操作名、操作数列表、分隔符、结果列表和可选的注释。因而，对寄存器1和2作加法，将结果置于寄存器3中，程序员可以这样写：

```
add r1,r2 ⇒ r3  // 示例指令
```

分隔符⇒，先于结果列表。它是一个可视化的提示符，暗示信息从左侧流动到右侧。特别地，这样做消除了阅读汇编代码文本时容易混淆操作数和结果的情况。（参见下表中的loadAI和storeAI。）

图1-3中的例子只使用了四个ILOC操作。

ILOC操作		语　义
loadAI	$r_1, c_2 \Rightarrow r_3$	Memory(r_1+c_2) → r_3
loadI	$c_1 \Rightarrow r_2$	$c_1 \rightarrow r_2$
mult	$r_1, r_2 \Rightarrow r_3$	$r_1 \times r_2 \rightarrow r_3$
storeAI	$r_1 \Rightarrow r_2, c_3$	r_1 → Memory(r_2+c_3)

附录A包含了对ILOC更为详细的描述。各个例子都将r_{arp}看做一个包含了当前过程数据存储起始地址的寄存器，也称为**活动记录指针**（activation record pointer）。

1. 指令选择

代码生成的第一阶段会将IR操作重写为目标机操作，这个过程称为指令选择（instruction selection）。指令选择将每个IR操作在各自的上下文中映射为一个或多个目标机操作。考虑重写我们的示例表达式a ← a × 2 × b × c × d，将其表示为ILOC虚拟机的代码以说明此过程。（在整本书中我们都将使用ILOC。）右侧再次给出了表达式的IR形式。编译器可以选择如图1-3所示的操作。该代码假定a、b、c和d分别位于从寄存器r_{arp}指定的地址开始，偏移量分别为@a、@b、@c和@d之处。

$t_0 \leftarrow a \times 2$
$t_1 \leftarrow t_0 \times b$
$t_2 \leftarrow t_1 \times c$
$t_3 \leftarrow t_2 \times d$
$a \leftarrow t_3$

```
loadAI  rarp, @a ⇒ ra       // 加载'a'
loadI   2        ⇒ r2       // 加载常数2到r2
loadAI  rarp, @b ⇒ rb       // 加载'b'
loadAI  rarp, @c ⇒ rc       // 加载'c'
loadAI  rarp, @d ⇒ rd       // 加载'd'
mult    ra, r2   ⇒ ra       // ra ← a × 2
mult    ra, rb   ⇒ ra       // ra ← (a × 2) × b
mult    ra, rc   ⇒ ra       // ra ← (a × 2 × b) × c
mult    ra, rd   ⇒ ra       // ra ← (a × 2 × b × c) × d
storeAI ra       ⇒ rarp,@a  // 写回ra到'a'
```

图1-3　对应于a ← a × 2 × b × c × d表达式的ILOC代码

编译器选择了一个很简单的操作序列。它将所有相关的值加载到寄存器中，按顺序执行乘法，并将结果存储到a对应的内存地址。这里假定寄存器的数目是不受限制的，并用符号名来命名寄存器，如r_a包含了a，r_{arp}包含了一个地址，我们例子中所有有名字的值在内存中的数据存储都自该地址开始。指令选择器隐式依赖于寄存器分配器，以便将这些符号寄存器名，或称为虚拟寄存器，映射到目标机的实际寄存器。

虚拟寄存器

一个符号寄存器名，编译器用其表示某个值可以保存在寄存器中。

指令选择器可以利用目标机提供的特殊操作。例如，如果有立即数乘法操作（multI）可用，编译器会将$mult\ r_a, r_2 \Rightarrow r_a$替换为$multI\ r_a, 2 \Rightarrow r_a$，这样就不必要进行$loadI\ 2 \Rightarrow r_2$操作了，而且减少了对寄存器的使用。如果加法比乘法快速，编译器可以将$mult\ r_a, r_2 \Rightarrow r_a$替换为$add\ r_a, r_a \Rightarrow r_a$，避免对$r_2$的loadI操作和使用，而且还可以将mult替换为较快速的add。第11章阐述了两种使用模式匹配进行指令选择的技术，为IR操作选择高效的实现。

2. 寄存器分配

在指令选择期间，编译器有意忽略了目标机寄存器数目有限的事实。它反而假定有“足够”的寄存器存在，并使用所谓的虚拟寄存器。实际上，编译的前期对寄存器的要求可能高于硬件的能力。寄存器分配器必须将这些虚拟寄存器映射到实际的目标机寄存器。因而，在代码中的每一个点上，寄存器分配器都必须决定哪些值驻留在目标机寄存器中。它接下来重写代码以反映其决策。例如，寄存器分配器将图1-3中的代码重写为如下形式，可以最小化寄存器的使用。

```
loadAI  r_arp, @a ⇒ r1          // 加载'a'
add     r1, r1    ⇒ r1          // r1 ← a × 2
loadAI  r_arp, @b ⇒ r2          // 加载'b'
mult    r1, r2    ⇒ r1          // r1 ← (a × 2) × b
loadAI  r_arp, @c ⇒ r2          // 加载'c'
mult    r1, r2    ⇒ r1          // r1 ← (a × 2 × b) × c
loadAI  r_arp, @d ⇒ r2          // 加载'd'
mult    r1, r2    ⇒ r1          // r1 ← (a × 2 × b × c) × d
storeAI r1        ⇒ r_arp, @a   // 写回ra到'a'
```

这里的指令序列只使用了三个寄存器，而不是原本的六个。

最小化寄存器的使用可能起相反作用。例如，如果已命名的值如a、b、c或d，其中有些已经在寄存器中，代码应该直接引用相应的寄存器。如果所有已命名的值都位于寄存器中，应该在不增加附加寄存器的情况下来实现对应的指令序列。换句话说，如果某些附近的表达式也计算了a×2，应该将该值保留在某个寄存器中，而不是稍后重新计算。这种优化可能会增加对寄存器的需求，但能够消除稍后需要发出的某些多余指令。第13章探讨了寄存器分配过程中发生的问题，以及编译器编写者用于解决相应问题的技术。

3. 指令调度

为产生执行快速的代码，代码生成器可能需要重排操作，以照顾目标机在特定方面的性能约束。不同操作的执行时间可能是不同的。内存访问操作可能需要花费几十甚至数百个CPU周期，但某些算术操作（以除法为例），只需要几个CPU周期。这种延迟较长的操作对编译后代码性能的影响可能是惊人的。

假定目前loadAI或storeAI操作需要三个周期，mult操作需要两个周期，而所有其他操作都只需一个周期。下表说明了在这些假定下，前述代码片段的执行情况。开始列给出了每个操作开始执行的周期，结束列给出了该操作完成的周期。

开始	结束				
1	3	loadAI	r_{arp}, @a	⇒ r_1	// 加载'a'
4	4	add	r_1, r_1	⇒ r_1	// $r_1 \leftarrow a \times 2$
5	7	loadAI	r_{arp}, @b	⇒ r_2	// 加载'b'
8	9	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2) \times b$
10	12	loadAI	r_{arp}, @c	⇒ r_2	// 加载'c'
13	14	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b) \times c$
15	17	loadAI	r_{arp}, @d	⇒ r_2	// 加载'd'
18	19	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b \times c) \times d$
20	22	storeAI	r_1	⇒ r_{arp}, @a	// 写回r_a到'a'

九个操作的指令序列花费了22个周期执行。最小化寄存器的使用并未导致执行变快。

许多处理器都有一种特性，可以在长延迟操作执行期间发起新的操作。只要新操作完成之前不引用长延迟操作的结果，执行都可以正常地进行。但如果某些插入的操作试图过早地读取长延迟操作的结果，处理器将延缓执行需要该值的操作，直至长延迟操作完成。在操作数就绪之前，操作是不能开始执行的，而操作结束之前，其结果也是无法读取的。

指令调度器重排代码中的各个操作。它试图最小化等待操作数所浪费的周期数。当然，调度器必须确保，新指令序列产生的结果与原来的指令序列是相同的。在很多情况下，调度器可以大幅度改进“朴素”代码的性能。对于我们的例子，好的调度器可能产生下列指令序列。

开始	结束				
1	3	loadAI	r_{arp}, @a	⇒ r_1	// 加载'a'
2	4	loadAI	r_{arp}, @b	⇒ r_2	// 加载'b'
3	5	loadAI	r_{arp}, @c	⇒ r_3	// 加载'c'
4	4	add	r_1, r_1	⇒ r_1	// $r_1 \leftarrow a \times 2$
5	6	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2) \times b$
6	8	loadAI	r_{arp}, @d	⇒ r_2	// 加载'd'
7	8	mult	r_1, r_3	⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b) \times c$
9	10	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b \times c) \times d$
11	13	storeAI	r_1	⇒ r_{arp}, @a	// 写回r_a到'a'

编译器构建是工程行为

典型的编译器有许多趟，这些趟共同作用，将某种源语言编写的代码转换为某种目标语言。在转换的过程中，编译器使用许多算法和数据结构。对于该过程中的每一步，编译器编写者都必须选择一种适当的解决方案。

成功的编译器实际执行的次数多到不可想象。考虑GCC编译器总的运行次数。在GCC的生命周期中，即使某些不起眼的低效之处，合计起来也浪费了大量的时间。良好的设计和实现带来的节约，会随着时间而累积。因而，编译器编写者必须注意编译时的成本，如算法的渐近复杂性、实现的实际运行时间和数据结构占用的空间，等等。对于编译器的各个任务都花费多长时间，编译器编写者应该有个预算。

例如，词法分析和语法分析这两个问题有大量的高效算法。词法分析器识别和归类单词所用的时间与输入程序中字符的数目成正比。对于一种典型的程序设计语言，语法分析器构建推导所花费的时间与推导的长度成正比。（程序设计语言的结构限制使高效的语法分析变得可能。）因为词法分析和语法分析存在高效的技术，编译器编写者应该预期仅在这些任务上花费一小部分编译时间。

相形之下，优化和代码生成包含了几个需要更多时间的问题。对于程序分析和优化来说，我们考察的许多算法，其复杂度都超过$\mathbf{O}(n)$。因而，与编译器前端相比，优化器和代码生成器中算法的选择对编译时间有更大的影响。编译器编写者可能需要进行一些折中，以分析准确性和优化有效性的下降来阻止编译时间的上升。进行此类决策时，需要进行明智且审慎的考虑。

代码的这一版本仅需要13个周期执行。与最小数目相比，该代码使用的寄存器多出一个。在这一指令序列中，除8、10、12周期之外，每个周期都开始一个操作。其他等价的调度也是可能的，但与之等长的调度一般需要使用更多的寄存器。第12章阐明了几种广泛应用的调度技术。

4. 代码生成的各组件间的交互

编译中大多数真正困难的问题出现在代码生成期间。而且这些问题相互影响，使得情况更为复杂。例如，指令调度移动`load`操作，使之远离依赖`load`的算术操作。这样做可以增加需要这些值的时间段，但此期间内所需的寄存器的数目也会相应地增加。类似地，将特定的值赋值给特定的寄存器，可以在两个操作之间建立“伪”相关性，从而限制指令调度。（在第一个操作完成之前第二个操作不能调度执行，即使在公用寄存器中的值并无依赖性。重命名该值可以消除这种伪相关性，代价是使用更多的寄存器。）

1.4 小结和展望

编译器构建是一项复杂任务。好的编译器合并了来自形式语言理论、算法研究、人工智能、系统设计、计算机体系结构和程序设计语言理论的思想，并将其应用到程序转换的问题上。编译器汇集了贪心算法、启发式技术、图算法、动态规划、DFA和NFA、不动点算法、同步和局部性、分配和命名，以及流水线管理。编译器面临的许多问题很难给出最优解，因此它使用近似算法、启发式技术和经验规则。这样做产生的复杂交互可能导致令人惊奇的结果，好坏兼有。

为将这些活动置于一个有条理的框架中，大多数编译器组织成三个主要的阶段：前端、优化器和后端。每个阶段都有一组不同的问题要解决，用于解决这些问题的方法也各有不同。前端专注于将源代码转换为某种IR。前端依赖于形式语言理论和类型理论的结果，以及若干健壮的算法和数据结构。中间部分或优化器，会将一个IR程序转换为另一个，其目标是生成执行更高效的IR程序。优化器分析程序得出关于其运行时行为的知识，而后利用此项知识来变换代码并改进其行为。后端将IR程序映射到特定处理器的指令集。对寄存器分配和指令调度方面的困难问题，后端会近似求解，其近似解的质量对于编译后代码的速度和大小有着直接影响。

本书会探讨所有这些阶段。第2章到第4章处理编译器前端所用的算法。第5章到第7章描述讨论优化和代码生成所需的一些背景材料。第8章介绍了代码优化。第9章和第10章更详细地阐述了程序分析和优化，感兴趣的读者可以阅读。最后，第11章到第13章涵盖了后端用于指令选择、调度和寄存器分配的技术。

本章注释

第一个编译器出现在20世纪50年代。这些早期系统显示了令人惊讶的复杂性。原始的FORTRAN编译器是一个多趟系统，包括分离的词法分析器、语法分析器和寄存器分配器以及一些优化[26, 27]。Ershov及其同事构建的Alpha system，可以进行局部优化[139]，并使用图着色算法减少数据项所需的内存数量[140, 141]。

Knuth回忆了有关20世纪60年代早期编译器构建的一些有趣的往事[227]。Randell和Russell描述了Algol 60语言的早期实现工作[293]。Allen描述了IBM公司内部编译器开发的历史，强调了理论和实践的相互影响[14]。

许多有影响力的编译器都构建于20世纪六七十年代。这其中包括经典的优化编译器FORTRANH[252, 307]，Bliss-11和Bliss-32编译器[72, 356]和可移植的BCPL编译器[300]。这些编译器可以为各种CISC机器产生高质量的代码。另一方面，学生使用的编译器专注于快速的编译、良好的诊断信息和纠错[97, 146]。

20世纪80年代RISC体系结构的问世导致了另一代编译器的出现，这些编译器专注于强有力的优化和代码生成技术[24, 81, 89, 204]。这些编译器带有全功能的优化器，其结构如图1-1所示。现代RISC编译器仍然遵循该模型。

在20世纪90年代期间，编译器构建方面的研究专注于对微处理器体系结构中发生的急剧变化作出反应。从Intel的i860处理器开始的十年间，编译器编写者面临了种种挑战，如直接管理流水线和内存延迟。到90年代末，编译器面临的诸多挑战包括（处理器）多功能部件、长的内存延迟和并行代码生成。事实证明，20世纪80年代RISC编译器的结构和组织仍然具有足够的灵活性来应对这些挑战，因此研究人员构建新的趟插入到其编译器的优化器和代码生成器中。

虽然Java系统混合使用了编译和解释[63, 279]，但Java并非第一个采用这种混合的语言。长期以来，Lisp语言系统早已经包含了本机代码编译器和虚拟机实现方案[266, 324]。Smalltalk-80语言系统使用了字节码分发和虚拟机[233]，有几个实现添加了JIT编译器[126]。

习题

(1) 考虑一个简单的Web浏览器，其输入是HTML格式的文本串，输出需要在屏幕上显示输入指定的图形。显示的过程是编译或解释吗？

(2) 在设计编译器的过程中，你可能面临许多折中。作为用户，你认为购买编译器最重要的五个特性是什么？如果你转为编译器编写者，上述特性列表会发生变动吗？关于你要实现的编译器，列表能告诉你什么呢？

(3) 编译器用于许多不同环境中。你认为针对下列应用设计的编译器会有哪些差别？

(a) 一个JIT编译器，用于转换通过网络下载而来的用户界面代码。

(b) 一个编译器，其目标机为移动电话中使用的嵌入式处理器。

(c) 一个编译器，用于中学的介绍性程序设计课程。

(d) 一个编译器，用于联编运行在海量并行处理器上（其中所有处理器都是相同的）的风洞模拟程序。

(e) 一个编译器，用于编译针对大量不同目标机的数字计算密集型程序。