第1章 引 论

程序设计语言是向人以及计算机描述计算过程的记号。如我们所知，这个世界依赖于程序设计语言，因为在所有计算机上运行的所有软件都是用某种程序设计语言编写的。但是，在一个程序可以运行之前，它首先需要被翻译成一种能够被计算机执行的形式。

完成这项翻译工作的软件系统称为编译器(compiler)。

本书介绍的是设计和实现编译器的方法。我们将介绍用于构建面向多种语言和机器的翻译器的一些基本思想。编译器设计的原理和技术还可以用于编译器设计之外的众多领域。因此，这些原理和技术通常会在一个计算机科学家的职业生涯中多次被用到。研究编译器的编写将涉及程序设计语言、计算机体系结构、形式语言理论、算法和软件工程。

在本章中，我们将介绍语言翻译器的不同形式，在高层次上概述一个典型编译器的结构，并讨论了程序设计语言和硬件体系结构的发展趋势。这些趋势将影响编译器的形式。我们还将介绍关于编译器设计和计算机科学理论的关系的一些事实，并给出编译技术在编译领域之外的一些应用。最后，我们将简单论述在我们研究编译器时需要用到的重要的程序设计语言概念。

1.1 语言处理器

简单地说，一个编译器就是一个程序，它可以阅读以某一种语言(源语言)编写的程序，并把该程序翻译成为一个等价的、用另一种语言(目标语言)编写的程序，参见图1-1。编译器的重要任务之一是报告它在翻译过程中发现的源程序中的错误。

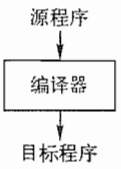

图1-1 一个编译器

如果目标程序是一个可执行的机器语言程序，那么它就可以被用户调用，处理输入并产生输出。参见图1-2。

解释器(interpreter)是另一种常见的语言处理器。它并不通过翻译的方式生成目标程序。从用户的角度看，解释器直接利用用户提供的输入执行源程序中指定的操作。参见图1-3。

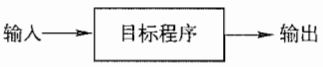

图1-2 运行目标程序

在把用户输入映射成为输出的过程中，由一个编译器产生的机器语言目标程序通常比一个解释器快很多。然而，解释器的错误诊断效果通常比编译器更好，因为它逐个语句地执行源程序。

例1.1 Java语言处理器结合了编译和解释过程，如图1-4所示。一个Java源程序首先被编译成一个称为字节码(bytecode)的中间表示形式。然后由一个虚拟机对得到的字节码加以解释执行。这样安排的好处之一是在一台机器上编译得到的字节码可以在另一台机器上解释执行。通过网络就可以完成机器之间的迁移。

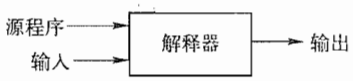

图1-3 一个解释器

为了更快地完成输入到输出的处理，有些被称为即时(just in time)编译器的Java编译器在运行中间程序处理输入的前一刻首先把字节码翻译成为机器语言，然后再执行程序。 □

如图1-5所示，除了编译器之外，创建一个可执行的目标程序还需要一些其他程序。一个源

程序可能被分割成为多个模块，并存放于独立的文件中。把源程序聚合在一起的任务有时会由一个被称为预处理器(preprocessor)的程序独立完成。预处理器还负责把那些称为宏的缩写形式转换为源语言的语句。

然后，将经过预处理的源程序作为输入传递给一个编译器。编译器可能产生一个汇编语言程序作为其输出，因为汇编语言比较容易输出和调试。接着，这个汇编语言程序由称为汇编器(assembler)的程序进行处理，并生成可重定位的机器代码。

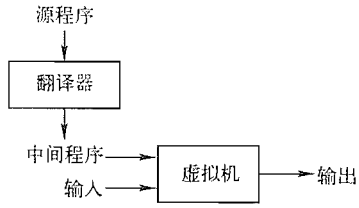

图 1-4 一个混合编译器

大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件连接到一起，形成真正在机器上运行的代码。一个文件中的代码可能指向另一个文件中的位置，而链接器(linker)能够解决外部内存地址的问题。最后，加载器(loader)把所有的可执行目标文件放到内存中执行。

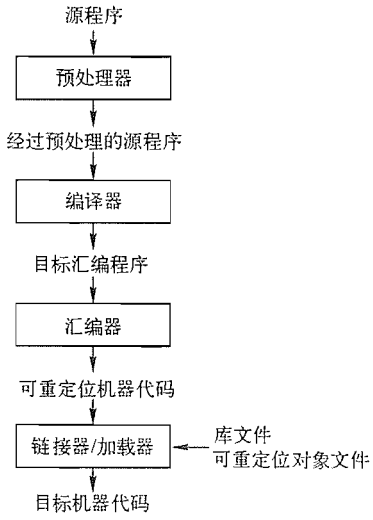

图 1-5 一个语言处理系统

1.1 节的练习

练习 1.1.1：编译器和解释器之间的区别是什么？

练习 1.1.2：编译器相对于解释器的优点是什么？解释器相对于编译器的优点是什么？

练习 1.1.3：在一个语言处理系统中，编译器产生汇编语言而不是机器语言的好处是什么？

练习 1.1.4：把一种高级语言翻译成为另一种高级语言的编译器称为源到源(source-to-source)的翻译器。编译器使用 C 语言作为目标语言有什么好处？

练习 1.1.5：描述一下汇编器所要完成的一些任务。

1.2 一个编译器的结构

到现在为止，我们把编译器看作一个黑盒子，它能够把源程序映射为在语义上等价的目标程序。如果把这个盒子稍微打开一点，我们就会看到这个映射过程由两个部分组成：分析部分和综合部分。

分析(analysis)部分把源程序分解成为多个组成要素，并在这些要素之上加上语法结构。然后，它使用这个结构来创建该源程序的一个中间表示。如果分析部分检查出源程序没有按照正确的语法构成，或者语义上不一致，它就必须提供有用的信息，使得用户可以按此进行改正。分析部分还会收集有关源程序的信息，并把信息存放在一个称为符号表(symbol table)的数据结构中。符号表将和中间表示形式一起传送给综合部分。

综合(synthesis)部分根据中间表示和符号表中的信息来构造用户期待的目标程序。分析部分经常被称为编译器的前端(front end)，而综合部分称为后端(back end)。

如果我们更加详细地研究编译过程，会发现它顺序执行了一组步骤(phase)。每个步骤把源程序的一种表示方式转换成另一种表示方式。一个典型的把编译程序分解成为多个步骤的方式如图 1-6 所示。在实践中，多个步骤可能被组合在一起，而这些组合在一起的步骤之间的中间表示不需要被明确地构造出来。存放整个源程序的信息的符号表可由编译器的各个步骤使用。

有些编译器在前端和后端之间有一个与机器无关的优化步骤。这个优化步骤的目的是在中

间表示之上进行转换，以便后端程序能够生成更好的目标程序。如果基于未经过此优化步骤的中间表示来生成代码，则代码的质量会受到影响。因为优化是可选的，所以图 1-6 中所示的两个优化步骤之一可以被省略。

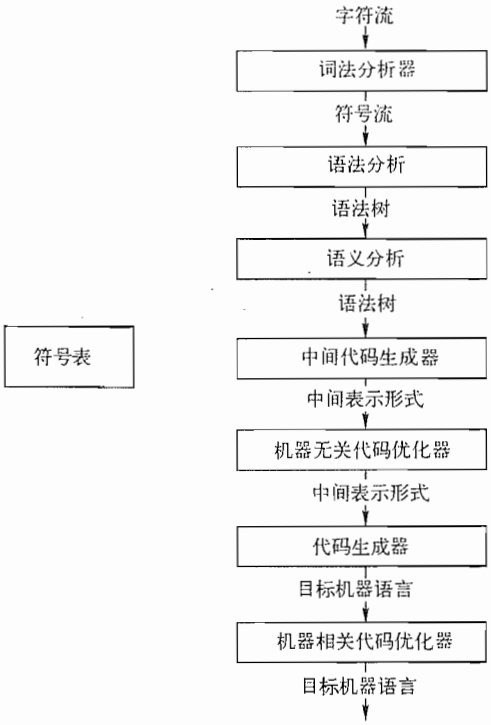

图 1-6　一个编译器的各个步骤

1.2.1　词法分析

编译器的第一个步骤称为*词法分析*(lexical analysis)或*扫描*(scanning)。词法分析器读入组成源程序的字符流，并且将它们组织成为有意义的*词素*(lexeme)的序列。对于每个词素，词法分析器产生如下形式的*词法单元*(token)作为输出：

〈token-name，attribute-value〉

这个词法单元被传送给下一个步骤，即语法分析。在这个词法单元中，第一个分量 token-name 是一个由语法分析步骤使用的抽象符号，而第二个分量 attribute-value 指向符号表中关于这个词法单元的条目。符号表条目的信息会被语义分析和代码生成步骤使用。

比如，假设一个源程序包含如下的赋值语句

```
position = initial + rate * 60        (1.1)
```

这个赋值语句中的字符可以组合成如下词素，并映射成为如下词法单元。这些词法单元将被传递给语法分析阶段。

1）`position` 是一个词素，被映射成词法单元〈**id**，1〉，其中 **id** 是表示标识符(identifier)的抽象符号，而 1 指向符号表中 `position` 对应的条目。一个标识符对应的符号表条目存放该标识符有关的信息，比如它的名字和类型。

2）赋值符号 = 是一个词素，被映射成词法单元〈=〉。因为这个词法单元不需要属性值，所以我们省略了第二个分量。也可以使用 **assign** 这样的抽象符号作为词法单元的名字，但是为了标记上的方便，我们选择使用词素本身作为抽象符号的名字。

3）`initial` 是一个词素，被映射成词法单元〈**id**，2〉，其中 2 指向 `initial` 对应的符号表条目。

4）+ 是一个词素，被映射成词法单元〈+〉。

5）`rate` 是一个词素，被映射成词法单元〈**id**，3〉，其中 3 指向 `rate` 对应的符号表条目。

6）* 是一个词素，被映射成词法单元〈*〉。

7）60 是一个词素，被映射成词法单元〈60〉[⊖]。

分隔词素的空格会被词法分析器忽略掉。

图 1-7 给出经过词法分析之后，赋值语句 1.1 被表示成如下的词法单元序列：

〈**id**，1〉〈=〉〈**id**，2〉〈+〉〈**id**，3〉〈*〉〈60〉　　(1.2)

在这个表示中，词法单元名 =、+ 和 * 分别是表示赋值、加法运算符、乘法运算符的抽象符号。

⊖ 从技术上讲，我们应该为语法单元 60 建立一个形如〈**number**，4〉的词法单元，其中 4 指向符号表中对应于整数 60 的条目。但是我们要到第 2 章中才讨论数字的词法单元。第 3 章将讨论建立词法分析器的技术。

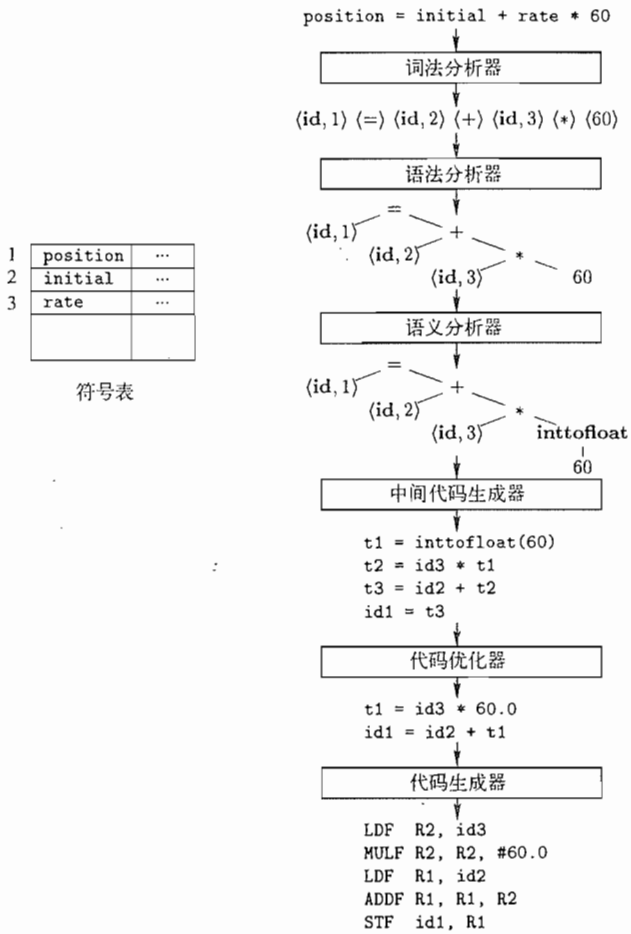

图 1-7 一个赋值语句的翻译

1.2.2 语法分析

编译器的第 2 个步骤称为*语法分析*(syntax analysis)或*解析*(parsing)。语法分析器使用由词法分析器生成的各个词法单元的第一个分量来创建树形的中间表示。该中间表示给出了词法分析产生的词法单元流的语法结构。一个常用的表示方法是*语法树*(syntax tree),树中的每个内部结点表示一个运算,而该结点的子结点表示该运算的分量。在图 1-7 中,词法单元流(1.2)对应的语法树被显示为语法分析器的输出。

这棵树显示了赋值语句

```
position = initial + rate * 60
```

中各个运算的执行顺序。这棵树有一个标号为 * 的内部结点,< **id**, 3 > 是它的左子结点,整数 60 是它的右子结点。结点 < **id**, 3 > 表示标识符 `rate`。标号为 * 的结点指明了我们必须首先把 `rate` 的值与 60 相乘。标号为 + 的结点表明我们必须把相乘的结果和 `initial` 的值相加。这棵树的根结点的标号为 =,它表明我们必须把相加的结果存储到标识符 `position` 对应的位置上去。这个运算顺序和通常的算术规则相同,即乘法的优先级高于加法,因此乘法应该在加法之前计算。

编译器的后续步骤使用这个语法结构来帮助分析源程序,并生成目标程序。在第 4 章,我们将使用上下文无关文法来描述程序设计语言的语法结构,并讨论为某些类型的语法自动构造高

效语法分析器的算法。在第 2 章和第 5 章，我们将看到，语法制导的定义将有助于描述对程序设计语言结构的翻译。

1.2.3　语义分析

语义分析器(semantic analyzer)使用语法树和符号表中的信息来检查源程序是否和语言定义的语义一致。它同时也收集类型信息，并把这些信息存放在语法树或符号表中，以便在随后的中间代码生成过程中使用。

语义分析的一个重要部分是*类型检查*(type checking)。编译器检查每个运算符是否具有匹配的运算分量。比如，很多程序设计语言的定义中要求一个数组的下标必须是整数。如果用一个浮点数作为数组下标，编译器就必须报告错误。

程序设计语言可能允许某些类型转换，这被称为*自动类型转换*(coercion)。比如，一个二元算术运算符可以应用于一对整数或者一对浮点数。如果这个运算符应用于一个浮点数和一个整数，那么编译器可以把该整数转换(或者说自动类型转换)成为一个浮点数。

图 1-7 中显示了一个这样的自动类型转换。假设 `position`、`initial` 和 `rate` 已被声明为浮点数类型，而词素 60 本身形成一个整数。图 1-7 中的语义分析器的类型检查程序发现运算符 * 被用于一个浮点数 `rate` 和一个整数 60。在这种情况下，这个整数可以被转换成为一个浮点数。请注意，在图 1-7 中，语义分析器输出中有一个关于运算符 **inttofloat** 的额外结点。**inttofloat** 明确地把它的整数参数转换为一个浮点数。类型检查和语义分析将在第 6 章中讨论。

1.2.4　中间代码生成

在把一个源程序翻译成目标代码的过程中，一个编译器可能构造出一个或多个中间表示。这些中间表示可以有多种形式。语法树是一种中间表示形式，它们通常在语法分析和语义分析中使用。

在源程序的语法分析和语义分析完成之后，很多编译器生成一个明确的低级的或类机器语言的中间表示。我们可以把这个表示看作是某个抽象机器的程序。该中间表示应该具有两个重要的性质：它应该易于生成，且能够被轻松地翻译为目标机器上的语言。

在第 6 章，我们将考虑一种称为*三地址代码*(three-address code)的中间表示形式。这种中间表示由一组类似于汇编语言的指令组成，每个指令具有三个运算分量。每个运算分量都像一个寄存器。图 1-7 中的中间代码生成器的输出是如下的三地址代码序列：

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```
(1.3)

关于三地址指令，有几点是值得专门指出的。首先，每个三地址赋值指令的右部最多只有一个运算符。因此这些指令确定了运算完成的顺序。在源程序 1.1 中，乘法应该在加法之前完成。第二，编译器应该生成一个临时名字以存放一个三地址指令计算得到的值。第三，有些三地址指令的运算分量的少于三个(比如上面的序列 1.3 中的第一个和最后一个指令)。

在第 6 章，我们将讨论在不同编译器中用到的主要中间表示形式。第 5 章将介绍语法制导翻译技术。这些技术在第 6 章中被用于处理典型程序设计语言构造进行类型检查和中间代码生成。这些程序设计语言构造包括：表达式、控制流构造和过程调用。

1.2.5　代码优化

机器无关的代码优化步骤试图改进中间代码，以便生成更好的目标代码。“更好”通常意味着更快，但是也可能会有其他目标，如更短的或能耗更低的目标代码。比如，一个简单直接的算法会生成中间代码(1.3)。它为由语义分析器得到的树形中间表示中的每个运算符都使用一个指令。

使用一个简单的中间代码生成算法，然后再进行代码优化步骤是生成优质目标代码的一个合理方法。优化器可以得出结论：把 60 从整数转换为浮点数的运算可以在编译时刻一劳永逸地完成。因此，用浮点数 60.0 来替代整数 60 就可以消除相应的 **inttofloat** 运算。而且，t3 仅被使用一次，用来把它的值传递给 id1。因此，优化器可以把序列(1.3)转换为更短的指令序列

```
t1 = id3 * 60.0
id1 = id2 + t1
```
(1.4)

不同的编译器所做的代码优化工作量相差很大。那些优化工作做得最多的编译器，即所谓的“优化编译器”，会在优化阶段花相当多的时间。有些简单的优化方法可以极大地提高目标程序的运行效率而不会过多降低编译的速度。从第 8 章开始，将详细讨论机器无关和机器相关的优化。

1.2.6 代码生成

代码生成器以源程序的中间表示形式作为输入，并把它映射到目标语言。如果目标语言是机器代码，那么就必须为程序使用的每个变量选择寄存器或内存位置。然后，中间指令被翻译成为能够完成相同任务的机器指令序列。代码生成的一个至关重要的方面是合理分配寄存器以存放变量的值。

比如，使用寄存器 R1 和 R2，(1.4)中的中间代码可以被翻译成为如下的机器代码：

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```
(1.5)

每个指令的第一个运算分量指定了一个目标地址。各个指令中的 F 告诉我们它处理的是浮点数。代码(1.5)把地址 id3 中的内容加载到寄存器 R2 中，然后将其与浮点常数 60.0 相乘。井号“#”表示 60.0 应该作为一个立即数处理。第三个指令把 id2 移动到寄存器 R1 中，而第四个指令把前面计算得到并存放在 R2 中的值加到 R1 上。最后，在寄存器 R1 中的值被存放到 id1 的地址中去。这样，这些代码正确地实现了赋值语句(1.1)。第 8 章将讨论代码生成。

上面对代码生成的讨论忽略了对源程序中的标识符进行存储分配的重要问题。我们将在第 7 章中看到，运行时刻的存储组织方法依赖于被编译的语言。编译器在中间代码生成或代码生成阶段做出有关存储分配的决定。

1.2.7 符号表管理

编译器的重要功能之一是记录源程序中使用的变量的名字，并收集和每个名字的各种属性有关的信息。这些属性可以提供一个名字的存储分配、它的类型、作用域(即在程序的哪些地方可以使用这个名字的值)等信息。对于过程名字，这些信息还包括：它的参数数量和类型、每个参数的传递方法(比如传值或传引用)以及返回类型。

符号表数据结构为每个变量名字创建了一个记录条目。记录的字段就是名字的各个属性。这个数据结构应该允许编译器迅速查找到每个名字的记录，并向记录中快速存放和获取记录中的数据。符号表在第 2 章中讨论。

1.2.8 将多个步骤组合成趟

前面关于步骤的讨论讲的是一个编译器的逻辑组织方式。在一个特定的实现中，多个步骤的活动可以被组合成一趟(pass)。每趟读入一个输入文件并产生一个输出文件。比如，前端步骤中的词法分析、语法分析、语义分析，以及中间代码生成可以被组合在一起成为一趟。代码优化可以作为一个可选的趟。然后可以有一个为特定目标机生成代码的后端趟。

有些编译器集合是围绕一组精心设计的中间表示形式而创建的，这些中间表示形式使得我们可以把特定语言的前端和特定目标机的后端相结合。使用这些集合，我们可以把不同的前端

和某个目标机的后端结合起来，为不同的源语言建立该目标机上的编译器。类似地，我们可以把一个前端和不同的目标机后端结合，建立针对不同目标机的编译器。

1.2.9　编译器构造工具

和任何软件开发者一样，写编译器的人可以充分利用现代的软件开发环境。这些环境中包含了诸如语言编辑器、调试器、版本管理、程序描述器、测试管理等工具。除了这些通用的软件开发工具，人们还创建了一些更加专业的工具来实现编译器的不同阶段。

这些工具使用专用的语言来描述和实现特定的组件，其中的很多工具使用了相当复杂的算法。其中最成功的工具都能够隐藏生成算法的细节，并且它们生成的组件易于和编译器的其他部分相集成。一些常用的编译器构造工具包括：

1）*语法分析器的生成器*：可以根据一个程序设计语言的语法描述自动生成语法分析器。

2）*扫描器的生成器*：可以根据一个语言的语法单元的正则表达式描述生成词法分析器。

3）*语法制导的翻译引擎*：可以生成一组用于遍历分析树并生成中间代码的例程。

4）*代码生成器的生成器*：依据一组关于如何把中间语言的每个运算翻译成为目标机上的机器语言的规则，生成一个代码生成器。

5）*数据流分析引擎*：可以帮助收集数据流信息，即程序中的值如何从程序的一个部分传递到另一部分。数据流分析是代码优化的一个重要部分。

6）*编译器构造工具集*：提供了可用于构造编译器的不同阶段的例程的完整集合。

在本书中，我们将讨论多个这类工具的例子。

1.3　程序设计语言的发展历程

第一台电子计算机出现在20世纪40年代。它使用由0、1序列组成的机器语言编程，这个序列明确地告诉计算机以什么样的顺序执行哪些运算。运算本身也是很低层的：把数据从一个位置移动到另一个位置，把两个寄存器中的值相加，比较两个值，等等。不用说，这种编程速度慢且枯燥，而且容易出错。写出的程序也是难以理解和修改的。

1.3.1　走向高级程序设计语言

走向更加人类友好的程序设计语言的第一步是20世纪50年代早期人们对助记汇编语言的开发。一开始，一个汇编语言中的指令仅仅是机器指令的助记表示。后来，宏指令被加入到汇编语言中。这样，程序员就可以通过宏指令为频繁使用的机器指令序列定义带有参数的缩写。

走向高级程序设计语言的重大一步发生在20世纪50年代的后五年。其间，用于科学计算的Fortran语言，用于商业数据处理的Cobol语言和用于符号计算的Lisp语言被开发出来。在这些语言的基本原理是设计高层次表示方法，使得程序员可以更加容易地写出数值计算、商业应用和符号处理程序。这些语言取得了很大的成功，至今仍然有人使用它们。

在接下来的几十年里，很多带有新特性的程序设计语言被陆续开发出来。它们使得编程更加容易、自然，功能也更强大。我们将在本章的后面部分讨论很多现代程序设计语言所共有的一些关键特征。

当前有几千种程序设计语言。可以通过不同的方式对这些语言进行分类。方式之一是通过语言的代来分类。*第一代语言*是机器语言，*第二代语言*是汇编语言，而*第三代语言*是Fortran、Cobol、Lisp、C、C++、C#及Java这样的高级程序设计语言。*第四代语言*是为特定应用设计的语言，比如用于生成报告的NOMAD，用于数据库查询的SQL和用于文本排版的Postscript。术语*第五代语言*指的是基于逻辑和约束的语言，比如Prolog和OPS5。

另一种语言分类方式把程序中指明如何完成一个计算任务的语言的称为强制式(imperative)语言，而把程序中指明要进行哪些计算的语言称为声明式(declarative)语言。诸如 C、C ++ 、C#和 Java 等语言都是强制式语言。所有强制式语言中都有用于表示程序状态和语句的表示方法，这些语句可以改变程序状态。像 ML、Haskell 这样的函数式语言和 Prolog 这样的约束逻辑语言通常被认为是声明式语言。

术语冯·诺伊曼语言(von Neumann language)是指以冯·诺伊曼计算机体系结构为计算模型的程序设计语言。今天的很多语言(比如 Fortran 和 C)都是冯·诺伊曼语言。

面向对象语言(object-oriented language)指的是支持面向对象编程的语言，面向对象编程是指用一组相互作用的对象组成程序的编程风格。Simula 67 和 Smalltalk 是早期的主流面向对象语言。C ++ 、C#、Java 和 Ruby 是现在常用的面向对象语言。

脚本语言(scripting language)是具有高层次运算符的解释型语言，它通常被用于把多个计算过程"粘合"在一起。这些计算过程被称为脚本。Awk、JavaScript、Perl、PHP、Python、Ruby 和 Tcl 是常见的脚本语言。使用脚本语言编写的程序通常要比用其他语言(比如 C)写的等价的程序短很多。

1.3.2 对编译器的影响

因为程序设计语言的设计和编译器是紧密相关的，程序设计语言的发展向编译器的设计者提出了新的要求。他们必须设计相应的算法和表示方式来翻译和支持新的语言特征。从 20 世纪 40 年代以来，计算机体系结构也有了很大的发展。编译器的设计者不仅需要跟踪新的语言特征，还需要设计出新的翻译算法，以便尽可能地利用新硬件的能力。

通过降低用高级语言程序的执行开销，编译器还可以推动这些高级语言的使用。要使得高性能计算机体系结构能够高效运行用户应用，编译器也是至关重要的。实际上，计算机系统的性能是非常依赖于编译技术的，以至于在构建一个计算机之前，编译器会被用作评价一个体系结构概念的工具。

编写编译器是很有挑战性的。编译器本身就是一个大程序。而且，很多现代语言处理系统在同一个框架内处理多种源语言和目标机。也就是说，这些系统可以被当做一组编译器来使用，可能包含几百万行代码。因此，好的软件工程技术对于创建和发展现代的语言处理器是非常重要的。

编译器必须能够正确翻译用源语言书写的所有程序。这样的程序的集合通常是无穷的。为一个源程序生成最佳目标代码的问题一般来说是不可判定的。因此，编译器的设计者必须作出折衷处理，确定解决哪些问题，使用哪些启发式信息，以便解决高效代码生成的问题。

我们将在 1.4 节看到，有关编译器的研究也是有关如何使用理论来解决实践问题的研究。

本书的目的是教授编译器设计中使用的根本思想和方法论。本书并不想让读者学习建立一个最新的语言处理系统时可能用到的所有算法和技术。但是，本书的读者将获得必要的基础知识和理解，学会建立一个相对简单的编译器。

1.3.3 1.3 节的练习

练习 1.3.1：指出下面的术语：

1) 强制式的　2) 声明式的　3) 冯·诺伊曼式的　4) 面向对象的
5) 函数式的　6) 第三代　7) 第四代　8) 脚本语言

可以被用于描述下面的哪些语言：

1) C　2) C ++　3) Cobol　4) Fortran　5) Java
6) Lisp　7) ML　8) Perl　9) Python　10) VB

1.4 构建一个编译器的相关科学

编译器的设计中有很多通过数学方法抽象出问题本质从而解决现实世界中复杂问题的完美

例子。这些例子可以被用来说明如何使用抽象方法来解决问题：接受一个问题，写出抓住了问题的关键特性的数学抽象表示，并用数学技术来解决它。问题的表达必须根植于对计算机程序特性的深入理解，而解决方法必须使用经验来验证和精化。

编译器必须接受所有遵循语言规范的源程序。源程序的集合是无穷的，而程序可能大到包含几百万行代码。在翻译一个源程序的过程中，编译器所做的任何翻译工作都不能改变被编译源程序的含义。因此，编译器设计者的工作不仅会影响到他们创建的编译器，还会影响到他们所创建的编译器所编译的全部程序。这种杠杆作用使得编译器设计的回报丰厚，但也使得编译器的开发工作具有挑战性。

1.4.1　编译器设计和实现中的建模

对编译器的研究主要是有关如何设计正确的数学模型和选择正确算法的研究。设计和选择时，还需要考虑到对通用性及功能的要求与简单性及有效性之间的平衡。

最基本的数学模型是我们将在第 3 章介绍的有穷状态自动机和正则表达式。这些模型可以用于描述程序的词法单位（关键字、标识符等）以及描述被编译器用来识别这些单位的算法。最基本的模型中还包括上下文无关文法，它用于描述程序设计语言的语法结构，比如嵌套的括号和控制结构。我们将在第 4 章研究文法。类似地，树形结构是表示程序结构以及程序到目标代码的翻译方法的重要模型。我们将在第 5 章介绍这一概念。

1.4.2　代码优化的科学

在编译器设计中，术语“优化”是指编译器为了生成比浅显直观的代码更加高效的代码而做的工作。“优化”这个词并不恰当，因为没有办法保证一个编译器生成的代码比完成相同任务的任何其他代码更快，或至少一样快。

现在，编译器所作的代码优化变得更加重要，而且更加复杂。之所以变得更加复杂，是因为处理器体系结构变得更加复杂，也有了更多改进代码执行方式的机会。之所以变得更加重要，是因为巨型并发计算机要求实质性的优化，否则它们的性能将会呈数量级地下降。随着多核计算机（这些计算机上的芯片拥有多个处理器）日益流行，所有的编译器都将面临充分利用多处理器计算机的优势的问题。

即使有可能通过随意的方法来建造一个健壮的编译器，实现起来也是非常困难的。因此，人们已经围绕代码优化建立了一套广泛且有用的理论。应用严格的数学基础，使得我们可以证明一个优化是正确的，并且它对所有可能的输入都产生预期的效果。从第 9 章开始，我们将会看到，如果想使得编译器产生经过良好优化的代码，图、矩阵和线性规划之类的模型是必不可少的。

从另一方面来说，只有理论是不够的。很多现实世界中的问题都没有完美的答案。实际上，我们在编译器优化中提出的很多问题都是不可判定的。在编译器设计中，最重要的技能之一是明确描述出真正要解决的问题的能力。我们在一开始需要对程序的行为有充分的了解，并且需要通过充分的试验和评价来验证我们的直觉。

编译器优化必须满足下面的设计目标：

- 优化必须是正确的，也就是说，不能改变被编译程序的含义。
- 优化必须能够改善很多程序的性能。
- 优化所需的时间必须保持在合理的范围内。
- 所需要的工程方面的工作必须是可管理的。

对正确性的强调是无论如何不会过分的。不管设计得到的编译器能够生成运行速度多么快的代码，只要生成的代码不正确，这个设计就是毫无意义的。正确设计优化编译器是如此困难，我们敢说没

有一个优化编译器是完全无错的！因此，设计一个编译器时最重要的目标是使它正确。

第二个目标是编译器应该有效提高很多输入程序的性能。性能通常意味着程序执行的速度。我们也希望能够尽可能降低生成代码的大小，在嵌入式系统中更是如此。而对于移动设备的情况，尽量降低代码的能耗也是我们期待的。在通常情况下，提高执行效率的优化也能够节约能耗。除了性能，错误报告和调试等的可用性方面也是很重要的。

第三，我们需要使编译时间保持在较短的范围内，以支持快速的开发和调试周期。当机器变得越来越快，这个要求会越来越容易达到。开始时，一个程序经常在没有进行优化的情况下开发和调试。这么做不仅可以降低编译时间，更重要的是未经优化的程序比较容易调试。这是因为编译器引入的优化经常使得源代码和目标代码之间的关系变得模糊。在编译器中开启优化有时会暴露出源程序中的新问题，因此需要对经过优化的代码再次进行测试。因为可能需要额外的测试工作，有时会阻止人们在应用中使用优化技术，当应用的性能不很重要的时候更是如此。

最后，编译器是一个复杂的系统，我们必须使系统保持简单以保证编译器的设计和维护费用是可管理的。我们可以实现的优化技术有无穷多种，而创建一个正确有效的优化过程需要相当大的工作量。我们必须划分不同优化技术的优先级别，只实现那些可以对实践中遇到的源程序带来最大好处的技术。

因此，我们在研究编译器时不仅要学习如何构造一个编译器，还要学习解决复杂和开放性问题的一般方法学。在编译器开发中用到的方法涉及理论和实验。在开始的时候，我们通常根据直觉确定有哪些重要的问题并把它们明确描述出来。

1.5 编译技术的应用

编译器设计并不只是关于编译器的。很多人用到了在学校里研究编译器时学到的技术，但是严格地说，它们从没有为一个主流的程序设计语言编写过一个编译器(甚至其中的一部分)。编译器技术还有其他重要用途。另外，编译器设计影响了计算机科学中的其他领域。在本节，我们将回顾和编译技术有关的最重要的互动和应用。

1.5.1 高级程序设计语言的实现

一个高级程序设计语言定义了一个编程抽象：程序员使用这个语言表达算法，而编译器必须把这个程序翻译成目标语言。总的来说，用高级程序设计语言编程比较容易，但是比较低效，也就是说，目标程序运行较慢。使用低级程序设计语言的程序员能够更多地控制一个计算过程，因此从原则上讲，可以产生更加高效的代码。遗憾的是，低级程序比较难编写，而且更糟糕的是可移植性较差，更容易出错，而且更加难以维护。优化编译器包括了提高所生成代码性能的技术，因此弥补了因高层次抽象而引入的低效率。

例 1.2 C 语言中的关键字 **register** 是编译器技术和语言发展互动的一个较早的例子。当 C 语言在 20 世纪 70 年代中期被创立时，人们认为有必要让程序员来控制哪个程序变量应该存放在寄存器中。当有效的寄存器分配技术出现后，这个控制变得没有必要了，大多数现代的程序不再使用这个语言特征。

实际上，使用关键字 **register** 的程序还可能损失效率，因为寄存器分配是一类很低层次的问题，程序员常常不是最好的判断这类问题的人选。寄存器分配的最优选择很大程度上取决于一个机器的体系结构的特点。把低层次资源管理的决策，比如寄存器分配，写死在程序中反而有可能损害性能。当运行程序的计算机有别于当初所设定的目标机时更是如此。 □

对于程序设计语言的选择的变化与不断提高抽象层次的方向是一致的。C 语言是在 20 世纪

80 年代主流的系统程序设计语言；20 世纪 90 年代开始的很多项目则选择 C ++；在 1995 年推出的 Java 很快在 20 世纪 90 年代后期流行起来。在每一轮中引入的新的程序设计语言特征都会推动对于编译器优化的新研究。接下来，我们将给出一个关于主要语言特征的概览，这些特征曾经推动了编译器技术的重要发展。

在实践中，所有的通用程序设计语言，包括 C、Fortran 和 Cobol，都支持用户定义的聚合类型（如数组和结构）和高级控制流（比如循环和过程调用）。如果我们仅仅把每个高级结构和数据存取运算直接翻译成为机器代码，得到的代码将会非常低效。编译器优化的一个组成部分称为*数据流优化*，它可以对程序的数据流进行分析，并消除这些构造之间的冗余。它们很有效，生成的代码和一个熟练的低级语言程序员所写的代码类似。

面向对象概念首先于 1967 年在 Simula 中引入，并被集成到 Smalltalk、C ++、C#和 Java 这样的语言中。面向对象的主要思想是

1）数据抽象

2）特性的继承

人们发现这两者都可以使得程序更加模块化和易于维护。面向对象程序和用很多其他语言编写的程序之间的不同在于它们由多得多的（但是较小）过程（在面向对象术语中称为方法（method））组成。因此，编译器优化技术必须能够很好地跨越源程序中的过程边界进行优化。过程内联技术（即把一个过程调用替换为相应过程体）在这里是非常有用的。人们还开发了可以加速虚拟方法分发的优化技术。

Java 有很多特征可以使编程变得更容易，其中的很多特征之前已经在别的语言中引入。Java 语言是类型安全的；也就是说，一个对象不能被当作另一个无关类型的对象来使用。所有的数组访问运算都会被检查以保证它们在数组的界限之内。Java 没有指针，也不允许指针运算。它具有一个内建的垃圾收集机制来自动释放那些不再使用的变量所占用的内存。虽然所有这些特征使得编程变得更加容易，但它们也会引起运行时刻的开销。人们开发了相应的编译优化技术来降低这个开销。比如，消除不必要的下标范围检查，以及把那些在过程之外不可访问的对象分配在栈里而不是堆里。此外，人们还开发了高效的算法来尽量降低垃圾收集的开销。

除此之外，Java 用来支持可移植和可移动的代码。程序以 Java 字节码的方式分发。这些字节码要么被解释执行，要么被动态地（即在运行时刻）编译为本地代码。动态编译也曾经在其他上下文环境中被研究过。在那里，信息在运行时刻被动态地抽取出来，并用来生成更加优化的代码。在动态编译中，尽可能降低编译时间是很重要的，因为编译时间也是运行开销的一部分。一个常用的技术是只编译和优化那些经常运行的程序片断。

1.5.2　针对计算机体系结构的优化

计算机体系结构的快速发展也对新编译器技术提出了越来越多的需求。几乎所有的高性能系统都利用了两种技术：*并行*（parallelism）和*内存层次结构*（memory hierarchy）。并行可以出现在多个层次上：在指令层次上，多个运算可以被同时执行；在处理器层次上，同一个应用的多个不同线程在不同的处理器上运行。内存层次结构是应对下述局限性的方法：我们可以制造非常快的内存，或者非常大的内存，但是无法制造非常大又非常快的内存。

并行性

所有的现代微处理器都采用了指令级并行性。但是，这种并行性可以对程序员隐藏起来。程序员写程序的时候就好像所有指令都是顺序执行的。硬件动态地检测顺序指令流之间的依赖关系，并且在可能的时候并行地发出指令。在有些情况下，机器包含一个硬件调度器。该调度器可以改变指令的顺序以提高程序的并行性。不管硬件是否对指令进行重新排序，编译器都可以

重新安排指令，以使得指令级并行更加有效。

指令级的并行也显式地出现在指令集中。VLIW(Very Long Instruction Word，非常长指令字)机器拥有可并行执行多个运算的指令。Intel IA64 是这种体系结构的一个有名的例子。所有的高性能通用微处理器还包含了可以同时对一个向量中的所有数据进行运算的指令。人们已经开发出了相应的编译器技术，从顺序程序出发为这样的机器自动生成代码。

多处理器也已经日益流行，即使个人计算机也拥有多个处理器。程序员可以为多处理器编写多线程的代码，也可以通过编译器从传统的顺序程序自动生成并行代码。这样的编译器对程序员隐藏了一些细节，包括如何在程序中找到并行性，如何在机器中分发计算任务，以及如何最小化处理器之间的同步和通信。很多科学计算和工程性应用需要进行高强度的计算，因此可以从并行处理中得到很大的好处。人们已经开发了并行技术以便自动地把顺序的科学计算程序翻译成为多处理器代码。

内存层次结构

一个内存层次结构由几层具有不同速度和大小的存储器组成。离处理器最近的层速度最快但是容量最小。如果一个程序的大部分内存访问都能够由层次结构中最快的层满足，那么程序的平均内存访问时间就会降低。并行性和内存层次结构的存在都会提高一个机器的潜在性能。但是，它们必须被编译器有效利用才能够真正为一个应用提供高性能计算。

内存层次结构可以在所有的机器中找到。一个处理器通常有少量的几百个字节的寄存器，几层包含了几 K 到几兆字节的高速缓存，包含了几兆到几 G 字节的物理寄存器，最后还包括多个几 G 字节的外部存储器。相应地；层次结构中相邻层次间的存取速度会有两到三个数量级上的差异。系统性能经常受到内存子系统的性能(而不是处理器的性能)的限制。虽然一般来说编译器注重优化处理器的执行，现在人们更多地强调如何使得内存层次结构更加高效。

高效使用寄存器可能是优化一个程序时要处理的最重要的问题。和寄存器必须由软件明确管理不同，高速缓存和物理内存是对指令集合隐藏的，并由硬件管理。人们发现，由硬件实现的高速缓存管理策略有时并不高效。当处理具有大型数据结构(通常是数组)的科学计算代码时更是如此。我们可以改变数据的布局或数据访问代码的顺序来提高内存层次结构的效率。我们也可以通过改变代码的布局来提高指令高速缓存的效率。

1.5.3 新计算机体系结构的设计

在计算机体系结构设计的早期，编译器是在机器建造好之后再开发的。现在，这种情况已经有所改变。因为使用高级程序设计语言是一种规范，决定一个计算机系统性能的不是它的原始速度，还包括编译器能够以何种程度利用其特征。因此，在现代计算机体系结构的开发中，编译器在处理器设计阶段就进行开发，然后编译得到代码并运行于模拟器上。这些代码被用来评价提议的体系结构特征。

RISC

有关编译器如何影响计算机体系结构设计的最有名的例子之一是 RISC(Reduced Instruction-Set Computer，精简指令集计算机)的发明。在发明 RISC 之前，趋势是开发的指令集越来越复杂，以使得汇编编程变得更容易。这些体系结构称为 CISC(Complex Instruction-Set Computer，复杂指令集计算机)。比如，CISC 指令集包含了复杂的内存寻址模式来支持对数据结构的访问，还包含了过程调用指令来保存寄存器和向栈中传递参数。

编译器优化经常能够消除复杂指令之间的冗余，把这些指令削减为少量较简单的运算。因此，人们期望设计出简单指令集。编译器可以有效地使用它们，而硬件也更容易进行优化。

大部分通用处理器体系结构，包括 PowerPC、SPARC、MIPS、Alpha 和 PA-RISC，都是基于

RISC 概念的。虽然 x86 体系结构(最流行的微处理器)具有 CISC 指令集，但在这个处理器本身的实现中使用了很多为 RISC 机器发展得到的思想。不仅如此，使用高性能 x86 机器的最有效的方法是仅使用它的简单指令。

专用体系结构

在过去的 30 年中，提出了很多的体系结构概念。其中包括：数据流机器、向量机、VLIW(非常长指令字)机器、SIMD(单指令，多数据)处理器阵列、心动阵列(systolic array)、共享内存的多处理器、分布式内存的多处理器。每种体系结构概念的发展都伴随着相应编译器技术的研究和发展。

这些思想中的一部分已经应用到嵌入式机器的设计中。因为整个系统都可以放到一个芯片里面，所以处理器不再是预包装的商品。人们可以针对特定应用进行裁剪以获得更好的费效比。由于规模经济效用，通用处理器的体系结构具有趋同性。而专用应用的处理器则与此相反，体现出了计算机体系结构的多样性。人们不仅需要编译器技术来为这些体系结构编程提供支持，也需要用它们来评价拟议中的体系结构设计。

1.5.4　程序翻译

我们通常把编译看作是从一个高级语言到机器语言的翻译过程。同样的技术也可以应用到不同种类的语言之间的翻译。下面是程序翻译技术的一些重要应用。

二进制翻译

编译器技术可以用于把一个机器的二进制代码翻译成另一个机器的二进制代码，使得可以在一个机器上运行原本为另一个指令集编译的程序。二进制翻译技术已经被不同的计算机公司用来增加它们的机器上的可用软件。特别地，因为 x86 在个人计算机市场上的主导地位，很多软件都是以 x86 二进制代码的形式提供的。人们开发了二进制代码翻译器，把 x86 代码转换成 Alpha和 Sparc 的代码。Transmeta 公司也在他们的 x86 指令集实现中使用了二进制转换。他们没有直接在硬件上运行复杂的 x86 指令集，他们的 Ttransmeta Crusoe 处理器是一个 VLIW 处理器，它依赖于二进制翻译器来把 x86 代码转换成为本地的 VLIW 代码。

二进制翻译也可以被用来提供向后兼容性。1994 年，当 Apple Macintosh 中的处理器从 Motorola MC68040 变为 PowerPC 的时候，便使用二进制翻译来支持 PowerPC 处理器运行遗留下来的 MC68040 代码。

硬件合成

不仅仅大部分软件是用高级语言描述的，连大部分硬件设计也是使用高级硬件描述语言描述的，这些语言有 Verilog 和 VHDL(Very high-speed integrated circuit Hardware Description Language，甚高速集成电路硬件描述语言)。硬件设计通常是在寄存器传输层(Register Transfer Level，RTL)上描述的。在这个层中，变量代表寄存器，而表达式代表组合逻辑。硬件合成工具把 RTL 描述自动翻译成为门电路，而门电路再被翻译成为晶体管，最后生成一个物理布局。和程序设计语言的编译器不同，这些工具经常会花费几个小时来优化门电路。还存在一些用来翻译更高层次(比如行为和函数层次)的设计描述的技术。

数据查询解释器

除了描述软件和硬件，语言在很多应用中都是有用的。比如，查询语言(特别是 SQL 语言(Structured Query Language，结构化查询语言)被用来搜索数据库。数据库查询由包含了关系和布尔运算符的断言组成。它们可以被解释，也可以编译为代码，以便在一个数据库中搜索满足这个断言的记录。

编译然后模拟

模拟是在很多科学和工程领域内使用的通用技术。它用来理解一个现象或者验证一个设计。

模拟器的输入通常包括设计描述和某次特定模拟运行的具体输入参数。模拟可能会非常昂贵。我们通常需要在不同的输入集合中模拟很多可能的设计选择。而每个实验可能需要在高性能计算机上花费几天时间才能完成。另一个方法不需要写一个模拟器来解释这些设计。它对设计进行编译并生成能够在机器上直接模拟特定设计的机器代码。后者的运行更加快。经过编译的模拟运行可以比基于解释器的方法快几个数量级。在那些可以模拟用 Verilog 或 VHDL 描述的设计的最新工具中，人们都使用了编译后模拟的技术。

1.5.5 软件生产率工具

程序可以说是人类迄今为止生产出的最复杂的工程制品，它们包含了很多很多的细节。要使得程序能够完全正确运行，每个细节都必须是正确的。结果是程序中的错误很是猖獗。错误可以使一个系统崩溃，产生错误的输出，使得系统容易受到安全性攻击，在关键系统中甚至会引起灾难性的运行错误。测试是对系统中的错误进行定位的主要技术。

一个很有意思且很有前景的辅助性方法是通过数据流分析技术静态地(即在程序运行之前)定位错误。数据流分析可以在所有可能的执行路径上找到错误，而不是像程序测试的时候所做的那样，仅仅是在那些由输入数据组合执行的路径上找错误。很多原本为编译器优化所开发的数据流分析技术可以用来创建相应的工具，帮助程序员完成他们的软件工程任务。

找到程序的所有错误是不可判定问题。可以设计一个数据流分析方法来找出所有可能带有某种错误的语句，对程序员发出警告。但是如果这些警告中的大部分都是误报，用户将不会使用这个工具。因此，实用的错误检测器经常既不是健全的也不是完全的。也就是说，它们不可能找出程序中的所有错误，也不能保证报告的所有错误都真正是错误。虽然如此，人们仍然开发了很多种静态分析工具，这些工具能够在实际程序中有效地找到错误，比如释放空指针或已释放过的指针。错误探测器可以是不健全的。这个事实使得它们和编译器的优化有着显著不同。优化器必须是保守的，在任何情况下都不能改变程序的语义。

在本节中，我们将提到使用程序分析技术来提高软件生产效率的几个已有途径。这些分析是在原本为编译器代码优化而开发的技术的基础上建立的。其中静态探测一个程序是否具有安全漏洞的技术是极为重要的。

类型检查

类型检查是一种有效的，且被充分研究的技术，它可以被用于捕捉程序中的不一致性。它可以用来检测一些错误，比如，运算被作用于错误类型的对象上，或者传递给一个过程的参数和该过程的*范型*(signature)不匹配。通过分析程序中的数据流，程序分析还可以做出比检查类型错误更多的工作。比如，一个指针被赋予了 NULL 值，然后又立刻被释放了，这个程序显然是错误的。

这个技术也可以用来捕捉某种安全漏洞。其中，攻击者可以向程序提供一个字符串或者其他数据，而这些数据没有被程序谨慎使用。一个用户提供的字符串可以被加上一个“危险”的标号。如果没有检查这个字符串是否满足特定的格式，那么它仍然是“危险”的。如果这种类型的字符串能够在某个程序点上影响代码的控制流，那么就存在一个潜在的安全漏洞。

边界检查

相对于较高级的程序设计语言而言，用较低级语言编程更加容易犯错。比如，很多系统中的安全漏洞都是因为用 C 语言编写的程序中的缓冲区溢出造成的。因为 C 语言没有数组边界检查，所以必须由用户来保证对数组的访问没有超出边界。因为不能检验用户提供的数据是否可能溢出一个缓冲区，程序可能被欺骗，把一个数据存放到缓冲区之外。攻击者可以巧妙处理这些数据，使得程序做出错误的行为，从而危及系统的安全。人们已经开发了一些技术来寻找程序中的缓冲区溢出，但收效并不显著。

如果程序是用一种包含了自动区间检查的安全的语言编写的，这个问题就不会发生。用来消除程序中的冗余区间检查的数据流分析技术也可以用来定位缓冲区溢出错误。而最大区别在于，没能消除某个区间检查仅仅会导致很小的额外运行时刻开销，而没有指出一个潜在的缓冲区溢出错误却可能危及系统的安全性。因此，虽然使用简单的技术去进行区间检查优化就已经足够了，但在错误探测工具中获得高质量的结果则需要复杂的分析技术，比如在过程之间跟踪指针值的技术。

内存管理工具

垃圾收集机制是在效率和易编程及软件可靠性之间进行折衷处理的另一个极好的例子。自动的内存管理消除了所有的内存管理错误(比如内存泄漏)。这些错误是 C 或 C++ 程序中问题的主要来源之一。人们开发了很多工具来帮助程序员寻找内存管理错误。比如，Purify 是一个能够动态地捕捉内存管理错误的被广泛使用的工具。还有一些能够帮助静态识别部分此类错误的工具也已经被开发出来。

1.6　程序设计语言基础

这一节我们将讨论在程序设计语言的研究中出现的最重要的术语和它们的区别。我们的目标并不是涵盖所有的概念或所有常见的程序设计语言。我们假设读者已经至少熟悉 C、C++、C#或 Java 中的一种语言，并且也可能已经遇到过其他语言。

1.6.1　静态和动态的区别

在为一个语言设计一个编译器时，我们所面对的最重要的问题之一是编译器能够对一个程序做出哪些判定。如果一个语言使用的策略支持编译器静态决定某个问题，那么我们说这个语言使用了一个*静态*(static)策略，或者说这个问题可以在*编译时刻*(compile time)决定。另一方面，一个只允许在运行程序的时候做出决定的策略被称为*动态策略*(dynamic policy)，或者被认为需要在*运行时刻*(run time)做出决定。

我们需要注意的另一个问题是声明的作用域。x 的一个声明的*作用域*(scope)是指程序的一个区域，在其中对 x 的使用都指向这个声明。如果仅通过阅读程序就可以确定一个声明的作用域，那么这个语言使用的是*静态作用域*(static scope)，或者说*词法作用域*(lexical scope)。否则，这个语言使用的是*动态作用域*(dynamic scope)。如果使用动态作用域，当程序运行时，同一个对 x 的使用会指向 x 的几个声明中的某一个。

大部分语言(比如 C 和 Java)使用静态作用域。我们将在 1.6.3 节中讨论静态作用域。

例 1.3　作为静态/动态区别的另一个例子，我们考虑一下 Java 类声明中术语 static 的使用。这个术语作用于数据。在 Java 中，一个变量是用于存放数据值的某个内存位置的名字。这里，“static”指的并不是变量的作用域，而是编译器确定用于存放被声明变量的内存位置的能力。比如声明

```
public static int x;
```

使得 x 成为一个*类变量*(class variable)，也就是说不管创建了多少个这个类的对象，只存在一个 x 的拷贝。此外，编译器可以确定内存中的被用于存放整数 x 的位置。反过来，如果这个声明中省略了“static”，那么这个类的每个对象都会有它自己的用于存放 x 的位置，编译器没有办法在运行程序之前预先确定所有这些位置。　□

1.6.2　环境与状态

我们在讨论程序设计语言时必须了解的另一个重要区别是在程序运行时发生的改变是否会

影响数据元素的值，还是影响了对那个数据的名字的解释。比如，执行像 x = y + 1 这样的赋值语句会改变名字 x 所指的值。更加明确地说，这个赋值改变了 x 所指向的内存位置上的值。

可能下面这一点就不是那么明显了。即 x 所指的位置也可能在运行时刻改变。比如，我们在例 1.3 中讨论过，如果 x 不是一个静态(或者说"类")变量，那么这个类的每一个对象都有它自己的分配给变量 x 的实例的位置。这种情况下，对 x 的赋值可能会改变那些"实例"变量中的某一个变量的值，这取决于包含这个赋值的方法作用于哪个对象。

名字和内存(存储)位置的关联，及之后和值的关联可以用两个映射来描述。这两个映射随着程序的运行而改变(见图 1-8)。

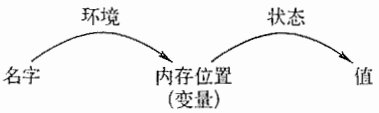

图 1-8 从名字到值的两步映射

1) 环境(environment)是一个从名字到存储位置的映射。因为变量就是指内存位置(即 C 语言中的术语"左值")，我们还可以换一种方法，把环境定义为从名字到变量的映射。

2) 状态(state)是一个从内存位置到它们的值的映射。以 C 语言的术语来说，即状态把左值映射为它们的相应右值。

环境的改变需要遵守语言的作用域规则。

例 1.4 考虑图 1-9 中的 C 程序片断。整数 i 被声明为一个全局变量，同时也被声明为局部于函数 f 的变量。执行 f 时，环境相应地调整，使得名字 i 指向那个为局部于 f 的那个 i 所保留的存储位置，且 i 的所有使用(如图中明确显示的赋值语句 i = 3)都指向这个位置。局部的 i 通常被赋予一个运行时刻栈中的位置。

只要当一个不同于 f 的函数 g 运行时，i 的使用就不能指向那个局部于 f 的 i。在函数 g 中对名字 i 的使用必须位于其他某个对 i 的声明的作用域内。一个例子是图中明确显示的赋值语句 x = i + 1，它位于某个其定义没有在图中显示的过程中。可以假定 $i+1$ 中的 i 指向全局的 i。和大多数语言一样，C 语言中的声明必须先于其使用，因此在全局 i 的声明之前的函数不能指向它。 □

```
...
int i;                    /* 全局i          */
...
void f(···) {
      int i;              /* 局部 i         */
      ...
      i = 3;              /* 对局部i的使用 */
      ...
}
...
      x = i + 1;          /* 对全局i的使用 */
```
图 1-9 名字 i 的两个声明

图 1-8 中的环境和状态映射是动态的，但是也有一些例外。

1) 名字到位置的静态绑定与动态绑定。大部分从名字到位置的绑定是动态的。我们在这一节中讨论了这种绑定的几种方法。某些声明(比如图1-9中的全局变量 i)可以在编译器生成目标代码时一劳永逸地分配一个存储位置。⊖

2) 从位置到值的静态绑定与动态绑定。一般来说，位置到值的绑定(图 1-8 的第二阶段)也是动态的，因为我们无法在运行一个程序之前指出一个位置上的值。被声明的常量是一个例外。比如，C 语言的定义

```
#define ARRAYSIZE 1000
```

把名字 ARRAYSIZE 静态地绑定为值 1000。我们看到这个语句就可以知道这个绑定关系，并且知道在程序运行时刻这个绑定不可能改变。

⊖ 从技术上来讲，C 语言编译器将为全局变量 i 分配一个虚拟内存中的位置，而由程序装载器和操作系统来决定到底把 i 分配在机器的物理地址中的什么地方。但是我们不用担心像这样的"重新分配"问题，因为它对编译过程没有影响。我们按照如下的方式处理地址空间问题：编译器在为它的输出代码使用地址空间时，假设它是在分配物理内存位置。

1.6.3 静态作用域和块结构

包括 C 语言和它的同类语言在内的大多数语言使用静态作用域。C 语言的作用域规则是基于程序结构的，一个声明的作用域由该声明在程序中出现的位置隐含地决定。稍后出现的语言，比如 C ++ 、Java 和 C#，也通过诸如 **public**、**private** 和 **protected** 等关键字的使用，提供了对作用域的明确控制。

在本节中，我们将考虑块结构语言的静态作用域规则，其中块(block)是声明和语句的一个组合。C 使用括号{和}来界定一个块。另一种为同一目的使用 **begin** 和 **end** 的方法可以追溯到 Algol。

名字、标识符和变量

虽然术语“名字”和“变量”通常指的是同一个事物，我们还是要很小心地使用它们，以便区别编译时刻的名字和名字在运行时刻所指的内存位置。

标识符(identifier)是一个字符串，通常由字母和数字组成。它用来指向(标记)一个实体，比如一个数据对象、过程、类，或者类型。所有的标识符都是名字，但并不是所有的名字都是标识符。名字也可以是一个表达式。比如名字 $x.y$ 可以表示 x 所指的一个结构中的字段 y。这里，x 和 y 是标识符，而 $x.y$ 是一个名字。像 $x.y$ 这样的复合名字称为*受限名字*(qualified name)。

*变量*指向存储中的某个特定的位置。同一个标识符被多次声明是很常见的事情，每一个这样的声明引入一个新的变量。即使每个标识符只被声明一次，一个递归过程中的局部标识符将在不同的时刻指向不同的存储位置。

例 1.5 C 语言的静态作用域策略可以概述如下：

1）一个 C 程序由一个顶层的变量和函数声明的序列组成。

2）函数内部可以声明变量，变量包括局部变量和参数。每个这样的声明的作用域被限制在它们所出现的那个函数内。

3）名字 x 的一个顶层声明的作用域包括其后的所有程序。但是如果一个函数中也有一个 x 的声明，那么函数中的那些语句就不在这个顶层声明的作用域内。

还有一些关于 C 语言的静态作用域策略的细节用来处理语句中的变量声明。我们将在接下来的内容中，以及在例 1.6 中查看这样的声明。 □

过程、函数和方法

为了避免总是说“过程、函数或方法”，每次我们要讨论一个可以被调用的子程序时，我们通常把它们统称为“过程”。但是当明确地讨论某个语言(比如 C)的程序时有一个例外。因为 C 语言只有函数，所以我们把它们称为“函数”。或者，如果我们讨论像 Java 这样的只有“方法”的语言时，我们就使用这个术语。

一个函数通常返回某个类型(即“返回类型”)的值，而一个过程不返回任何值。C 和类似的语言只有函数，因此它们把过程当作是具有特殊返回类型“void”的函数来处理。“void”表示没有返回值。像 Java 和 C ++ 这样的面向对象语言使用术语“方法”。这些方法可以像函数或者过程一样运行，但是总是和某个特定的类相关联。

在 C 语言中，有关块的语法如下：

1）块是一种语句。块可以出现在其他类型的语句(比如赋值语句)所能够出现的任何地方。

2）一个块包含了一个声明的序列，然后再跟着一个语句序列。这些声明和语句用一对括号包围起来。

注意，这个语法允许一个块嵌套在另一个块内。这个嵌套特性称为*块结构*(block structure)。C 族语言都具有块结构，但是不能在一个函数内部定义另一个函数。

如果块 B 是包含声明 D 的最内层的块，那么我们说 D 属于 B。也就是说，D 在 B 中，且不在嵌套于 B 中的任何其他块中。

在一个块结构语言中，关于变量声明的静态作用域规则如下。如果名字 x 的声明 D 属于块 B，那么 D 的作用域包括整个 B，但是以任意深度嵌套在 B 中、重新声明了 x 的所有块 B'不在此作用域中。这里，x 在 B'中重新声明是指存在另一个属于 B'的对相同名字 x 的声明 D'。

另一个等价的表达这个规则的方法着眼于名字 x 的一次使用。设 B_1，B_2，…，B_k 是所有的包含了 x 的该次使用的块。其中，B_k 嵌套在 B_{k-1}中，B_{k-1}嵌套在 B_{k-2}中，…，依此类推。寻找最大的满足下面条件的 i：存在一个属于 B_i 的 x 的声明。本次对 x 的使用就是指向 B_i 中对 x 的声明。换句话说，x 的本次使用在 B_i 中的这个声明的作用域内。

例 1.6 在图 1-10 中的 C++ 程序有四个块，其中包含了变量 a 和 b 的几个定义。为了帮助记忆，每个声明把其变量初始化为它所属于的那个块的编号。

比如，考虑块 B_1 中的声明 `int a = 1;`。它的作用域包括整个 B_1，当然那些(可能很深地)嵌套在 B_1 中并且有它自己的对 a 的声明的块除外。直接嵌套在 B_1 中的 B_2 没有 a 的声明，而 B_3 就有。B_4 没有 a 的声明。因此块 B_3 是整个程序中唯一位于名字 a 在 B_1 中的声明的作用域之外的地方。也就是说，这个作用域包括 B_4 和 B_2 中除了 B_3 之外的所有部分。关于程序中的全部五个声明的作用域的总结见图 1-11。

```
main() {
    int a = 1;                                   B1
    int b = 1;
    {
        int b = 2;                               B2
        {
            int a = 3;                           B3
            cout << a << b;
        }
        {
            int b = 4;                           B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

图 1-10 一个 C++ 程序中的块结构

声　明	作用域
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4;`	B_4

图 1-11 例 1.6 中的声明的作用域

从另一个角度看，让我们考虑块 B_4 中的输出语句，并把那里使用的变量 a 和 b 和适当的声明绑定。包含该语句的块的列表从小到大是 B_4、B_2、B_1。请注意，B_3 没有包含问题中所提到的点。B_4 有一个 b 的声明，因此该语句中对 b 的使用被绑定到这个声明，因此打印出来的 b 的值是 4。然而，B_4 没有 a 的声明，因此我们接着看 B_2。这个块也没有 a 的声明，因此我们继续看 B_1。幸运的是，这个块有一个声明 `int a = 1;`。因此，打印出来的 a 的值是 1。如果没有这个声明，程序就是错误的。 □

1.6.4 显式访问控制

类和结构为它们的成员引入了新的作用域。如果 p 是一个具有字段(成员)x 的类的对象，那么在 $p.x$ 中对 x 的使用指的是这个类定义中的字段 x。和块结构类似，类 C 中的一个成员声明 x

的作用域可以扩展到所有的子类 C'，除非 C'有一个本地的对同一名字 x 的声明。

通过 **public**、**private** 和 **protected** 这样的关键字的使用，像 C ++ 或 Java 这样的面向对象语言提供了对超类中的成员名字的显式访问控制。这些关键字通过限制访问来支持封装(encapsulation)。因此，私有(private)名字被有意地限定了作用域，这个作用域仅仅包含了该类和“友类”(C ++ 的术语)相关的方法声明和定义。被保护的(protected)名字可以由子类访问，而公共的(public)名字可以从类外访问。

在 C ++ 中，一个类的定义可能和它的部分或全部方法的定义分离。因此对于一个和类 C 相关联的名字 x，可能存在一个在它作用域之外的代码区域，然后又跟着一个在它作用域内的代码区域(一个方法定义)。实际上，在这个作用域之内和之外的代码区域可能相互交替，直到所有的方法都被定义完毕。

声明和定义

程序设计语言概念中的两个看起来相似的术语“声明”和“定义”实际上有着很大的不同。声明告诉我们事物的类型，而定义告诉我们它们的值。因此，`int i` 是一个 i 的声明，而 `i =1` 是 i 的一个定义(定值)。

当我们处理方法或者其他过程时，这个区别就更加明显。在 C ++ 中，通过给出了方法的参数及结果的类型(通常称为该方法的范型)，在类的定义中声明这个方法。然后，这个方法在另一个地方被定义，即在另一个地方给出了执行这个方法的代码。类似地，我们会经常看到在一个文件中定义了一个 C 语言的函数，然后在其他使用这个函数的文件中声明这个函数。

1.6.5　动态作用域

从技术上讲，如果一个作用域策略依赖于一个或多个只有在程序执行时刻才能知道的因素，它就是动态的。然而，术语*动态作用域*通常指的是下面的策略：对一个名字 x 的使用指向的是最近被调用但还没有终止且声明了 x 的过程中的这个声明。这种类型的动态作用域仅仅在一些特殊情况下才会出现。我们将考虑两个动态作用域的例子：C 预处理器中的宏扩展，以及面向对象编程中的方法解析。

例 1.7　在图 1-12 给出的 C 程序中，标识符 a 是一个代表了表达式 $(x+1)$ 的宏。但 x 到底是什么呢？我们不能够静态地(也就是说通过程序文本)解析 x。

实际上，为了解析 x，我们必须使用前面提到的普通的动态作用域规则。我们检查所有当前活跃的函数调用，然后选择最近调用的且具有一个对 x 的声明的函数[⊖]。对 x 的使用就是指向这个声明。

```
#define a (x+1)
int x = 2;
void b() { int x = 1; printf("%d\n", a); }
void c() { printf("%d\n", a); }
void main() { b(); c(); }
```

图 1-12　一个其名字的作用域必须动态确定的宏

在图 1-12 的例子中，函数 *main* 首先调用函数 b。当 b 执行时打印宏 a 的值。因为首先必须

⊖ 这个规则可能只对当前的例子成立。如果将图 1-12 的例子中的函数 b 改成 `void b(){int x =1; printf("% d\n", a); c(); }`，那么当 `main` 函数调用函数 b，函数 b 又调用 c 的时候，c 中的 `printf("% d\n", a)` 语句依然打印值 2。即此时对 x 的使用对应的仍然是全局的 x，而不是按照规则确定的函数 b，即“最近调用的且有一个对 x 的声明的函数”。——译者注

用($x+1$)替换掉a，所以我们把本次对x的使用解析为对函数b中的声明 int x = 1。原因是b有一个x的声明，因此b中的 printf 中的($x+1$)指向这个x。因此，打印出的值是2。

在b运行结束之后，函数c被调用，我们依旧需要打印宏a的值。然而，唯一可以被c访问的x是全局变量x。函数c中的 printf 语句指向x的这个声明，且被打印的值是3。 □

动态作用域解析对多态过程是必不可少的。所谓多态过程是指对于同一个名字根据参数类型具有两个或多个定义的过程。在有些语言中，比如 ML(见 7.3.3 节)，人们可以静态地确定名字所有使用的类型。在这种情况下，编译器可以把每个名字为p的过程替换为对相应的过程代码的引用。但是，在其他语言中，比如在 Java 和 C++ 中，编译器有时不能够做出这样的决定。

静态作用域和动态作用域的类比

虽然可以有各种各样的静态或者动态作用域策略，在通常的(块结构的)静态作用域规则和通常的动态策略之间有一个有趣的关系。从某种意义上说，动态规则处理时间的方式类似于静态作用域处理空间的方式。静态规则让我们寻找的声明位于最内层的、包含变量使用位置的单元(块)中；而动态规则让我们寻找的声明位于最内层的、包含了变量使用时间的单元(过程调用)中。

例 1.8 面向对象语言的一个突出特征就是每个对象能够对一个消息做出适当反应，调用相应的方法。换句话说，执行$x.m()$时调用哪个过程要由当时x所指向的对象的类来决定。一个典型的例子如下：

1) 有一个类C，它有一个名字为$m()$的方法。

2) D是C的一个子类，而D有一个它自己的名字为$m()$的方法。

3) 有一个形如$x.m()$的对x的使用，其中x是类C的一个对象。

正常情况下，在编译时刻不可能指出x指向的是类C的对象还是其子类D的对象。如果这个方法被多次应用，那么很可能某些调用作用在由x指向的类C的对象，而不是类D的对象，而其他调用作用于类D的对象之上。只有到了运行时刻才可能决定应当调用m的哪个定义。因此，编译器生成的代码必须决定对象x的类，并调用其中的某一个名字为m的方法。 □

1.6.6 参数传递机制

所有的程序设计语言都有关于过程的概念，但是在这些过程如何获取它们的参数方面，不同的语言之间有所不同。在本节，我们将考虑*实在参数*(在调用过程时使用的参数)是如何与*形式参数*(在过程定义中使用的参数)关联起来的。使用哪一种传递机制决定了调用代码序列如何处理参数。大多数语言要么使用“值调用”，要么使用“引用调用”，或者两者都用。我们将解释这些术语以及另一个被称为“名调用”的方法，解释后者主要是基于对历史的兴趣。

值调用

在*值调用*(call-by-value)中，会对实在参数求值(如果它是表达式)或拷贝(如果它是变量)。这些值被放在属于被调用过程的相应形式参数的内存位置上。这种方法在 C 和 Java 中使用，也是 C++ 语言及大部分其他语言的一个常用选项。值调用的效果是，被调用过程所做的所有有关形式参数的计算都局限于这个过程，相应的实在参数本身不会被改变。

然而请注意，在 C 语言中我们可以传递变量的一个指针，使得该变量的值能够被被调用者修改。同样，C、C++ 和 Java 中作为参数传递的数组名字实际上向被调用过程传递了一个指向该数组本身的指针或引用。因此，如果a是调用过程的一个数组的名字，且它被以值调用

的方式传递给相应的形式参数 x，那么像 x[2] = i 这样的赋值语句实际上改变了数组元素 $a[i]$。原因是虽然 x 是 a 的值的一个拷贝，但这个值实际上是一个指针，指向被分配给数组 a 的存储区域的开始处。

类似地，Java 中的很多变量实际上是对它们所代表的事物的引用，或者说指针。这个结论对数组、字符串和所有类的对象都有效。虽然 Java 只使用值调用，但只要我们把一个对象的名字传递给一个被调用过程，那个过程收到的值实际上是这个对象的指针。因此，被调用过程是可以改变这个对象本身的值的。

引用调用

在引用调用(call-by-reference)中，实在参数的地址作为相应的形式参数的值被传递给被调用者。在被调用者的代码中使用形式参数时，实现方法是沿着这个指针找到调用者指明的内存位置。因此，改变形式参数看起来就像是改变了实在参数一样。

但是，如果实在参数是一个表达式，那么在调用之前首先会对表达式求值，然后它的值被存放在一个该值自己的位置上。改变形式参数会改变这个位置上的值，但对调用者的数据没有影响。

C++中的"ref"参数使用的是引用调用。而在很多其他语言中，引用调用也是一种选项。当形式参数是一个大型的对象、数组或结构时，引用调用几乎是必不可少的。原因是严格的值调用要求调用者把整个实在参数拷贝到属于相应形式参数的空间上。当参数很大时，这种拷贝可能代价高昂。正如我们在讨论值调用时所指出的，像 Java 这样的语言解决数组、字符串和其他对象的参数传递问题的方法是仅仅复制这些对象的引用。结果是，Java 运行时就好像它对所有不是基本类型(比如整数、实数等)的参数都使用了引用调用。

名调用

第三种机制——名调用——被早期的程序设计语言 Algol 60 使用。它要求被调用者的运行方式好像是用实在参数以字面方式替换了被调用者的代码中的形式参数一样。这么做就好像形式参数是一个代表了实在参数的宏。当然被调用过程的局部名字需要进行重命名，以便把它们和调用者中的名字区别开来。当实在参数是一个表达式而不是一个变量时，会发生一些和直觉不符的问题。这也是今天不再采用这种机制的原因之一。

1.6.7　别名

引用调用或者其他类似的方法，比如像 Java 中那样把对象的引用当作值传递，会引起一个有趣的结果。有可能两个形式参数指向同一个位置，这样的变量称为另一个变量的*别名*(alias)。结果是，任意两个看起来从两个不同的形式参数中获得值的变量也可能变成对方的别名。

例 1.9　假设 a 是一个属于某个过程 p 的数组，且 p 通过调用语句 $q(a, a)$ 调用了另一个过程 $q(x, y)$。再假设像 C 语言或类似的语言那样，参数是通过值传递的，但数组名实际上是指向数组存放位置的引用。现在，x 和 y 变成了对方的别名。要点在于，如果 q 中有一个赋值语句 x[10] = 2，那么 y[10]的值也是 2。 □

事实上，如果编译器要优化一个程序，就要理解别名现象以及产生这一现象的机制。正如我们从第 9 章看到的，在很多情况下我们必须在确认某些变量相互之间不是别名之后才可以优化程序。比如，我们可能确定 x = 2 是变量 x 唯一被赋值的地方。如果是这样，那么我们可以把对 x 的使用替换为对 2 的使用。比如，把 a = x + 3 替换为较简单的 a = 5。但是，假设有另一个变量 y 是 x 的别名。那么，一个赋值语句 y = 4 可能具有意想不到的改变 x 的值的效果。这可能也意味着把 a = x + 3 替换为 a = 5 是一个错误，此时，a 的正确值可能是 7。

1.6.8 1.6 节的练习

练习 1.6.1：对图 1-13a 中的块结构的 C 代码，指出赋给 w、x、y 和 z 的值。

```
int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;
```

a) 练习 1.6.1 的代码

```
int w, x, y, z;
int i = 3; int j = 4;
{   int i = 5;
    w = i + j;
}
x = i + j;
{   int j = 6;
    i = 7;
    y = i + j;
}
z = i + j;
```

b) 练习 1.6.2 的代码

图 1-13 块结构代码

练习 1.6.2：对图 1-13b 中的代码重复练习 1.6.1。

练习 1.6.3：对于图 1-14 中的块结构代码，假设使用常见的声明的静态作用域规则，给出其中 12 个声明中的每一个的作用域。

```
{   int w, x, y, z;      /* 块 B1 */
    {   int x, z;        /* 块 B2 */
        {   int w, x;    /* 块 B3 */ }
    }
    {   int w, x;        /* 块 B4 */
        {   int y, z;    /* 块 B5 */ }
    }
}
```

图 1-14 练习 1.6.3 的块结构代码

练习 1.6.4：下面的 C 代码的打印结果是什么？

```
#define a (x+1)
int x = 2;
void b() { x = a; printf("%d\n", x); }
void c() { int x = 1; printf("%d\n", a); }
void main() { b(); c(); }
```

1.7 第 1 章总结

- *语言处理器*：一个集成的软件开发环境，其中包括很多种类的语言处理器，比如编译器、解释器、汇编器、连接器、加载器、调试器以及程序概要提取工具。
- *编译器的步骤*：一个编译器的运作需要一系列的步骤，每个步骤把源程序从一个中间表示转换成为另一个中间表示。
- *机器语言和汇编语言*：机器语言是第一代程序设计语言，然后是汇编语言。使用这些语言进行编程既费时，又容易出错。
- *编译器设计中的建模*：编译器设计是理论对实践有很大影响的领域之一。已知在编译器设计中有用的模型包括自动机、文法、正则表达式、树型结构和很多其他理论概念。
- *代码优化*：虽然代码不能真正达到最优化，但提高代码效率的科学既复杂又非常重要。它是编译技术研究的一个主要部分。
- *高级语言*：随着时间的流逝，程序设计语言担负了越来越多的原先由程序员负责的任务，比如内存管理、类型一致性检查或代码的并发执行。
- *编译器和计算机体系结构*：编译器技术影响了计算机的体系结构，同时也受到体系结构发展的影响。体系结构中的很多现代创新都依赖于编译器能够从源程序中抽取出有效利用硬件能力的机会。
- *软件生产率和软件安全性*：使得编译器能够优化代码的技术同样能够用于多种不同的程序分析任务。这些任务既包括探测常见的程序错误，也包括发现程序可能会受到已被黑

客们发现的多种入侵方式之一的伤害。

- *作用域规则*：一个 x 的声明的作用域是一段上下文，在此上下文中对 x 的使用指向这个声明。如果仅仅通过阅读某个语言的程序就可以确定其作用域，那么这个语言就使用了*静态作用域*，或者说*词法作用域*。否则这个语言就使用了*动态作用域*。
- *环境*：名字和内存位置关联，然后再和值相关联。这个情况可以使用*环境*和*状态*来描述。其中环境把名字映射成为存储位置，而状态则把位置映射到它的值。
- *块结构*：允许语句块相互嵌套的语言称为*块结构*的语言。假设一个块中有一个 x 的声明 D，而嵌套于这个块中的块 B 中有一个对名字 x 的使用。如果在这两个块之间没有其他声明了 x 的块，那么这个 x 的使用位于 D 的作用域内。
- *参数传递*：参数可以通过值或引用的方式从调用过程传递给被调用过程。当通过值传递方式传递大型对象时，实际被传递的值是指向这些对象本身的引用。这样就变成了一个高效的引用调用。
- *别名*：当参数被以引用传递方式(高效地)传递时，两个形式参数可能会指向同一个对象。这会造成一个变量的修改改变了另一个变量的值。

1.8 第 1 章参考文献

对于在 1967 年之前被开发并使用的程序设计语言(包括 Fortran、Algol、Lisp 和 Simula)的发展历程见[7]。对于 1982 年前被创建的语言(包括 C、C++、Pascal 和 Smalltalk)见[1]。

GNU 编译器集合 gcc(GNU Compiler Collection)是 C、C++、Fortran、Java 和其他语言的开源编译器的流行源头[2]。Phoenix 是一个编译器构造工具包，它提供了一个集成的框架，用于建立本书中提到的编译器的程序分析、代码生成和代码优化步骤[3]。

要获取更多的关于程序设计语言概念的信息，我们推荐[5, 6]。要知道更多的关于计算机体系结构信息，以及体系结构是如何影响编译的，我们建议阅读[4]。

1. Bergin, T. J. and R. G. Gibson, *History of Programming Languages*, ACM Press, New York, 1996.
2. `http://gcc.gnu.org/` .
3. `http://research.microsoft.com/phoenix/default.aspx` .
4. Hennessy, J. L. and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan-Kaufmann, San Francisco, CA, 2004.
5. Scott, M. L., *Programming Language Pragmatics, second edition*, Morgan-Kaufmann, San Francisco, CA, 2006.
6. Sethi, R., *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1996.
7. Wexelblat, R. L., *History of Programming Languages*, Academic Press, New York, 1981.