高等学校教学用书

高级程序设计语言概论

龚天富　李广星　编

电子科技大学出版社

高等学校教学用书

高级程序设计语言概论

龚天富　李广星　编

电子科技大学出版社

·1989·

内 容 简 介

本书讨论高级程序设计语言共同性的概念和规律，并介绍了新一代程序设计语言。前五章是本书的重点，基于对传统程序设计语言进行讨论。第一章绪论，第二章讨论概念的抽象，第三章介绍语言结构，第四章讨论数据类型，第五章讨论控制结构。第六章至第八章分别介绍函数式、逻辑式和面向对象程序设计语言。第九章简单介绍形式语义学。二至九章附有一定的问题。

本书可作为大学计算机专业高年级学生和研究生的教材或教学参考书，也可供计算机科学技术人员参考。

高等学校教学用书

高级程序设计语言概论

龚天富　李广里　编

电子科技大学出版社出版
（中国成都建设北路二段四号）
四川省金堂新华印刷厂印刷
四川省新华书店经销

开本 787×1092　1/16　印张 13.25　字数 320千字
版次 1989年11月第一版　印次 1989年11月第一次印刷
印数 1—3100 册
中国标准书号：ISBN 7-81016-206-3/TP·14
（15452·90）　定价 4.20 元

序

裘天富、李广星两位老师所编的《高级程序设计语言概论》出版了。它对我国计算机事业，特别是对计算机软件专业教育事业的发展将起到有益的作用。

随着电子计算机这一人化自然物的诞生，它作为人类认识和实践的对象的历史已有四十多年了。计算机作为思维对象，也正是在人这一思维主体与电子计算机——人化自然物之间以认识和实践为中介而联结和相互作用中生成和发展起来的。语言是思维的一种最重要、最有效的工具。对于计算机这一思维对象，人们在为实践和认识所进行的思维活动中，用程序设计语言这一人工语言作为一种最重要、最有效的工具，是客观的事实。程序设计语言已有不下几百种，它们之间不会是互不关联的。由于思维是有规律的，各种程序设计语言也必然有一些共同性的概念和规律。掌握这些规律性的东西，对软件管理人员、职业程序员和语言设计者大有裨益。《高级程序设计语言概论》一书在结构和取材方面正是符合这些客观要求的，在国内同类的教材中尚属鲜见。因此，它的问世值得欢迎。我向该书编者表示衷心的祝贺和感谢。感谢他们为我国计算机科学教育事业做了一件好事。

江 明 德

1989年6月9日于成都

前　言

程序设计语言是人与计算机进行通信的工具，它是软件开发最重要的工具。随着计算机科学技术的发展，计算机得到越来越广泛的应用，其中高级程序设计语言的发展起了重要的作用。也可以说，计算机科学技术的发展与高级程序设计语言的发展有着密切的联系。因此学习和研究高级程序设计语言是极其重要的。

自第一个高级程序设计语言FORTRAN问世后，三十多年来，已经诞生了上千种高级程序设计语言，它们体裁各异，千姿百态。一个计算机科技人员只能熟悉和使用这些语言中他所需要的少数几种，最多几十种。多年来，我们的同事们一直在讨论对这样众多的高级程序设计语言如何进行教学。我们对七七级和七八级同学开出多达四种不同体裁和结构的高级程序设计语言课程，但仍对提高学生水平收效不大。至今几乎缩减到只开出一种高级程序设计语言（Pascal）课程。我们认为，不仅是要教会学生使用某种高级程序设计语言，更重要的是使他们能掌握高级程序设计语言共同性的概念和规律，以便分析、鉴赏、评价、选择、学习和设计高级程序设计语言。为了培养学生具有上述能力，我们在教学计划中安排了《高级程序设计语言概论》课程。我们高兴地看到，目前不少学校的同行与我们具有相同的认识，有的学校将它称为《高级程序设计语言原理》。基于上述认识和有利于今后的教学工作，我们编写了此书。

我们查阅了国内外有关资料。早期的资料或教材大多是罗列一些常用的语言，从形式或结构上分析和对比它们的优缺点，这些资料不能满足我们的需要。近年来国外已出版了一些很好的教材，特别是C.Chezzi和M.Jazayeri编写的Programming Language Concepts一书，很符合我们的设想，这本书是我们编写本书的主要参考书。

本书的宗旨不是教会读者使用某种具体的高级程序设计语言，而是力图使读者掌握高级程序设计语言共同性的概念和规律，因而全书贯穿抽象的观点。全书共分三大部分，第一部分（第一章至第五章）基于传统程序设计语言讨论它们共同性的概念和规律（语言结构、数据类型和控制结构）；第二部分（第六章至第八章）介绍新一代程序设计语言（分别介绍逻辑式、函数式和面向对象程序设计语言）；第三部分简单介绍形式语义学。全书重点在前五章，其余内容可根据需要进行选取。二至九章均附有一定的习题。

本书由李广星编写第六章至第八章，以及§4.5节，其余由龚天富编写。本书的出版得到许多同事的帮助，他们为本书的编写思想提出了宝贵的建议、并为资料的收集和整理作了大量的具体工作，在此表示衷心的谢意。特别是要感谢江明德教授，他在百忙中审阅了全书，并为本书写了序。

由于编者的水平有限，编写时间特别仓促，书中肯定有不少缺点和错误，恳切希望广大读者批评指正。

编　者

1989年6月5日于成都

目　　录

第一章 绪 论

§1.1 引 言

语言是人们交流思想的工具。人类在长期发展过程中，为了交流思想，表达感情和交换信息，逐步形成了语言。这类语言，如英语和汉语，是人类在长期发展过程中逐步丰富形成的，通常称为**自然语言**。韦伯斯特（Webster）曾给语言下了一个直觉定义："语言是为相当大的团体的人所懂得并使用的字，以及组合这些字的统一方法。"这个定义揭示了语言的两个要素——词法和语法。然而，这个定义对于建立语言的数学理论是不够精确的。

1948年第一台数字计算机一问世就成为人们强有力的计算工具。使用计算机完成预定计算任务，必须告诉计算机"做什么"和"怎么做"，人要把有关信息告诉计算机；反之，又要求计算机把计算的结果告诉人。因此，人和机器之间需要进行通信（交流信息）。人将数据和怎样处理这些数据的一系列命令（指令）提交给计算机，而机器执行这些命令完成特定的计算任务。上述提交给计算机的数据和命令的集合通常称为**程序**。

既然人与计算机要进行通信，就需要信息的载体，通常选择语言作为这样的载体。换句话说，语言是人与计算机通信的工具。遗憾的是，由于现有的自然语言词汇量大，语法复杂和意义的理解依赖于上下文等因素，妨碍了通过自然语言来进行人机之间的通信。这样一来，迫使人们不得不设计出词汇较少，语法简单和意义明确的适合于计算机的语言，这样的语言通常称为**程序设计语言**。这类由人设计出来的语言又称为**人工语言**，以便区别于自然语言。本书讨论的对象是程序设计语言，为了叙述的简洁方便，在不引起混淆的情况下，把它简称为语言，读者在阅读下文时，应当严加注意。

每当设计出一台计算机，随之孪生一种该机能理解并可直接执行的语言，这样的语言称为**机器语言**。采用其它语言编写的程序，机器不能直接执行，必须将它翻译成相应（等价）的机器语言代码，机器才能直接执行。机器语言（包括汇编语言）是**与机器有关的语言**，又称**低级语言**；其它的语言是**与机器无关的语言**，又称**高级语言**。每种高级语言都有一个不大的字汇表，一种显式定义的文法，以及构造良好的语法规则和语义解释。这些基本属性的严格规定便于高级语言到低级语言的机器翻译。

随着计算机科学的不断发展，以及计算机越来越广泛的应用，人们出于种种不同的需要，已经设计并加以应用的语言数以千计，它们千姿百态，体裁[1]各异。本书将讨论语言最重要的概念和它们的特点，不是对语言各种性能的研究，而是给出并评价一些能用来理解现有上千种语言的主要概念，以及这些概念在程序设计中的地位和作用。我们采用一种比较的方法，概念的给出是通过对比它们在各种语言中的表现形式来完成的。全书强调抽象的原则，

---

[1] 英文Style用于修饰程序设计语言或程序设计，在中文中可翻译成风格、技巧或体裁。编者认为，翻译成体裁更为合适。

将其视作理解和驾驭复杂事物的最好方法。同时还强调程序设计体裁的多样化，例如，命令式、函数式、对象式和陈述式的语言体裁，并评述它们的特点，指出它们的不足。

每当我们讨论一个概念时，强调首创该概念的语言。例如，SIMULA 67 的类，ALGOL68 的强类型，Pascal 的类型定义，SNOBOL 4 的串和模式匹配，LISP（作为数学对象）的函数及PROLOG的陈述说明等等。为了广泛和深刻地揭示概念和说明概念，我们从大量的语言中引入许多例子，这些语言是 Ada、ALGOL60、ALGOL68、APL、C、CLU、COBOL、Euclid、FORTRAN、FP、Mesa、Modula-2、SETL、SQL 和 Smalltalk等。

本书与其它基于某种特定程序设计语言的教科书不同，其目的不是想让读者成为某种语言的熟练的程序员，而是想提高读者鉴赏和评价程序设计语言（或语言设计方案）的能力；读者能了解语言的重要概念、语言的功能及限制，以便选择一种恰当的语言来为某种目的而使用；同时，也为读者提供了设计一种新语言或扩充现有语言的能力。总之，这不是一本讨论如何进行程序设计，而是讨论如何鉴赏、评价、选择和设计程序设计语言的书。它对软件管理人员、职业程序人员和语言设计人员，特别是计算机专业的大学生和研究生将是一本有益的教材。

本章将从软件开发过程和语言在该过程中的作用着手讨论，后面的各章将讨论语言的各种概念和特点。

§1.2 软件开发过程

软件开发过程可分为五个阶段，这些阶段是相继进行的五个步骤，每一阶段都有明确的任务和相应的“产品”。

1．需求分析　一个用户团体想用一个软件系统来实现其某种目的，就去找软件开发者来进行开发。因此，任何软件系统都是用来满足某个用户团体的要求的。但是，用户仅仅是感觉到需要，未必能充分提出详尽的要求，必须由用户和软件开发人员共同对任务进行分析，开发出需求，并以适当的形式进行描述，以便双方都能理解。系统的成功与否取决于开发出来的软件满足需求的程度，以及需求满足用户实际需要的程度。

这一阶段的“产品”是需求文挡（用以描述系统应当做什么），另外还有用户手册，成本和可行性研究报告，性能要求和开发计划等。需求文挡只描述系统做什么，而不管系统应当如何做。

2．软件设计　软件开发者按需求文挡来着手进行软件系统的设计。这一阶段的“产品”是系统设计规范说明文挡及组成系统的所有模块和它们之间的接口规范说明。现有许多不同的软件设计和说明方法，但还没有一个是被普遍接受的方法。这一阶段所采用的软件设计方法对下一阶段的实现工作有很大的影响，将在 §1.4节进一步讨论。

3．实现（编码）　实现工作主要是程序设计，选择一种适当的语言进行编码，用以满足设计要求。在软件开发过程中，仅在这一阶段直接使用语言。本阶段的“产品”是带文挡的完全实现的系统。

4．测试　测试阶段对将要提供给用户的已实现的软件系统进行评估。事实上，测试工作在软件开发过程的各个阶段都要进行，用来检验每一阶段的“中间产品”是否达到了该阶

段的目标。

在需求分析阶段，对需求文挡进行检验，用以证明需求是否满足用户的需要；设计阶段验证设计说明是否与需求一致。实现阶段进行两类测试，即模块测试和综合测试。模块测试应由正在对该模块进行工作的程序员来进行，以保证其能满足接口说明。综合测试是对一些模块进行的，以保证各模块之间的接口的一致性。此外，尚可进行形式化的程序正确性证明等方面的工作。总之，在每一阶段，随时都要回答两个问题："是否在正确地建立产品？"以及"是否在建立正确的产品？"

上述实现阶段的详细测试并不排斥需要一个单独的测试阶段，以确保整个系统的质量。通常把测试阶段称为质量保证。这一阶段的测试工作不是由开发小组而是由一个专门的测试小组进行的。小组的主要工作是运行验收测试，在强负载下对软件进行测试，并检查程序和文挡是否满足预先说明的标准。

5．维护　当系统交付使用之后，由于发现故障（错误），或希望加入新的功能，或想对原有的功能进行改造，都需要对系统进行修改，这些修改统称为维护。通常，一个软件系统的维护代价至少与上述其它阶段的总代价相等，可见维护阶段是极为重要的。

我们简单介绍了软件开发过程的五个阶段，其目的是要进一步讨论程序设计语言与软件开发的关系，下面几节将从几个方面来讨论有关问题。

§1.3　软件开发环境与程序设计语言

所谓软件开发环境就是辅助软件开发的一整套开发工具和技术。环境用于软件开发的所有阶段：需求、设计、实现、测试和维护等阶段。在编码（实现）阶段，主要使用语言作为工具。语言已成为软件开发环境的重要组成部分，并有丰富的其它辅助工具的支持，例如，文本编辑程序、编译程序、连接程序、装入程序和库等工具。在早期的计算过程中，程序员必须把程序穿在卡片或纸带上，然后输入机器提交给编译程序进行编译。在穿孔或誊抄过程中，都可能出现错误，增加了测试的难度。随着计算机科学各个领域的发展，完成这些任务的工作已逐步自动化了。交互式的编辑程序可用来建立程序，并用文件管理程序将建立的程序存储在一个程序库中以备使用。当需要的时候，可将预先建立并且（可能）已编译的几个程序连接成一个可执行的程序，装入机器执行，用以完成预定的计算任务。这些辅助程序作为软件开发工具，减少了出错机会，提高了软件开发效率。

软件开发比通常所说的"程序设计"所涉及的范围要广泛得多，若要提高软件的生产率，就必须设计出支持软件开发各个阶段的环境。使用这样一个环境的理想方案是由应用和计算机专业人员组成的小组，开发出系统需求，然后利用环境随时跟踪需求的开发和更新，检查任何可能出现的不完整性和不一致性，并保证文挡能随需求的更新而更新。

需求分析阶段结束之后，受环境影响的系统设计者将逐步细化系统的设计，规范说明必要的模块和模块之间的接口，测试数据也应在这一阶段随之产生，实现者则基于设计着手实现系统。这一阶段软件开发环境所提供的工具是大家所熟知的，例如程序设计语言、编辑程序、编译程序、仿真程序、解释程序、连接程序和调试程序等。

就环境的有用性而论，不仅这些工具应在一起友好合作，而且应与其它阶段所使用的工具兼容。例如，语言应与设计阶段所支持的设计方法兼容。若使用的是自顶向下的设计方

法，那么语言就应支持这种方法，即应允许分层次进行程序设计。下一节将特别讨论设计方法对语言的影响。应着重强调的是，环境应看成由工具和方法两方面所组成。为了获得最大的效益，它们应当是兼容的。

在测试阶段，环境提供各式各样的帮助：可用来收集需求，以及设计和实现各阶段的测试数据；随着设计说明的更改，也可用来保证这些测试数据保持现实性；支持使用模块的形式化说明及证明；对测试结果进行收集和分类；以及采取适当的量级来确定测试策略的合理性和正确性等等。

在理想环境下，所有这些互相影响的结果应当是一个文挡齐全的，被验证过的，且与最初所提出的需求所一致的系统。若在以后任何时候需要修改系统（需求），环境必须能够跟踪设计规范说明、代码、测试数据和文挡的改变，并标明受影响的地方。

上面描述了一个简单的理想的软件开发环境，尽管现在尚无这样的环境，但从技术上将这一设想变成现实已为时不远了。事实上，上述环境的各个组件（成分）都已构成并实现，只是尚缺乏将这些组件组合成一体的系统工程方法。本节讨论的重点是软件开发者所涉及的整体活动，以及语言在软件开发中所起的作用，语言仅是软件开发者所使用的一种工具，它的实用性只能通过在软件开发过程中所作出的贡献来衡量，因而语言与软件开发环境中的其它组件间的关系是极为重要的。

或许读者会认为，第一个程度设计语言是为“程序设计”而不是为“软件开发”而设计的。诚然，这是历史的事实，第一个语言的设计者当初尚未预料到“软件生产”，但是，即使如此，今天我们总是要以这个准则来对一个语言进行评价的。本书将遵循这个准则。

§1.4 软件设计方法与程序设计语言

我们在上节已经阐明，软件设计方法和语言都是软件开发环境的一个组成部分，它们之间有着密切的联系。若选定某种方法进行设计时，就会发现某些语言比另一些语言更好用。例如，早期的 FORTRAN 并不是为支持某种特定的设计方法而设计的，由于它缺乏适当的高级控制结构，所以难于系统地实现自顶向下的设计方法。相比之下，Pascal 就明显地支持自顶向下的设计，以及结构程序设计。语言的发展趋势表明，一个语言应当至少支持一种设计方法。当然，语言学家正在研究支持多种程序设计方法的新语言，至今已在理论上取得了可喜的进展。

软件设计方法和语言这两个领域可以凝聚在一起。当前已有许多这方面的结果，信息隐蔽和数据抽象就是这方面最显著的例子。

信息隐蔽是将系统分解成模块的设计技术。如何分解一个软件，以便得到最佳的模块集合，信息隐蔽技术要求：模块应由这样一些设计决择来表征，对（每一个）决择而言，所有的其它选择对它都是不知道的。换句话说，模块应这样进行指定和设计，使得包含在模块内的信息（过程和数据）对无需这些信息的其它模块是不可访问的。每个模块都隐藏着一个“秘密”，通常指特定的数据结构（记录和文件等）的格式，模块提供可以用于查询和更新数据结构中所含信息的访问函数。模块的使用者（即其它模块）可以通过对这些函数的调用访问模块所提供的信息，而不是通过直接处理模块的数据结构或使用全局变量来访问这些信息。

信息隐蔽的目的就是通过将系统分解成具有清晰接口的、高度独立的模块来管理复杂对象，以便实现“分而治之”和“关系分离”。这样做的结果使软件变得更加可读，更能分块独立开发，特别是更易修改和维护。

许多具有数据抽象功能的语言，例如，CLU、SIMULA67 、Modula-2、Smalltalk和Ada 都支持这种技术。SIMULA67 的类、CLU的簇、Modula-2 的模块、Smalltalk的类或Ada 的程序包都能直接表达信息隐蔽模块。

信息隐蔽以及上述语言表达信息隐蔽的数据抽象能力可能是独立发展的，也可能是受到另一方的影响而发展的。我们对它们的历史发展过程不太感兴趣，而更关心设计方法的选择可能影响语言的设计，还会影响到所选用的语言。

§1.5 计算机体系结构与程序设计语言

在建立需求，并用语言来实现这些需求的意义下，正如上节所述，设计方法对语言是有影响的。但从限制语言设计，以便更有效地在现有机器上实现该语言这一角度出发，机器的体系结构也以相反的作用产生对语言的影响。由于现在几乎所有的计算机都类似于原始的冯·诺意曼体系结构，所以语言也受到冯·诺意曼概念的限制。

所谓冯·诺意曼（机）概念是基于这样的概念：一个存储器（用来存储指令和数据），一个控制器和一个处理器。控制器负责从存储器中逐条取机器指令（机器指令是非常低级的，通常它还要从存储器中取得数据）；处理器通过算术或逻辑操作来处理数据，最后的结果还必须送回存储器的单元中。

常规的语言至少和冯·诺意曼机共享两个最基本的概念：指令的按步顺序执行和存储的值可被修改。读者回顾所使用过的语言，可知语句（命令）是逐条执行的，相应于机器逐条取出指令并执行；赋值语句可用来修改变量的值，相应于冯·诺意曼机存储器的行为。目前所谓的面向高级语言的体系结构并非是由实际语言的需要所推动的，因为语言自身就是被冯·诺意曼概念所驱使的，所以面向高级语言的体系结构未能冲破冯·诺意曼机概念的束缚。

巴科斯（Backus）在1977年接受图灵奖的演说中，强烈号召人们使用函数式语言，因为它具有更简单更良好的数学性质和更高的表达能力。LISP 是最显著的函数式语言，为了把它在传统的体系结构（冯·诺意曼机）上有效地实现，不得不牺牲它的许多函数特性，所以它只能在某些领域取代常规语言。巴科斯认为：只有计算机能有效地执行的语言，才有可能放弃传统的语言而采用这样的函数式语言。他还认为：赋值语句和变量概念是许多祸害的根源，而冯·诺意曼体系结构是带来这些祸害的根源。为此，人们正在致力于研究冲破传统体系结构的非冯·诺意曼机器。我们深信，这种硬件技术将很快成为现实，到那时计算机科学技术将发生一场新的革命。

函数式语言基于函数的数学概念、函数的组合及函数的作用；逻辑式语言基于逻辑，特别是谓词逻辑。自1982年日本确定开发新一代计算机——第五代计算机，并宣称将选择逻辑程序设计作为新一代计算机的基础以来，关于不同程序设计体裁的问题在计算机界，特别是在语言专家中产生了激烈的争论。这些争论可归结为：是否应当保持各种体裁的分离，根据实际应用来选择它们？是否在所有应用中只使用一种体裁？是否应当将所有体裁集中在一起并一

体化？语言学家面对大量的类似问题，正在致力于精心的研究。

本书将讨论流行近三十年来的语言（所谓传统语言）以及影响它们的设计问题。第三章至第五章重点讨论所谓面向语句的语言，第六章讨论函数式语言的主要问题，第七章和第八章分别讨论逻辑式和对象式语言的主要问题。第九章简单介绍形式语义学。

§1.6 软件开发过程与程序设计语言

程序设计语言是软件开发的工具之一，本节将讨论软件开发对语言的要求。这里仅就软件的**可靠性**、**可维护性**和**有效性**来讨论对语言的要求。

（1）软件必须可靠

一个良好的软件应当得到用户的信任，哪怕是出现了异常或不期望的事件（例如硬件或软件失败），用户也应在使用软件时觉得安然。若软件是按需求说明工作的，那么它就是正确的，越是精确的说明，就越能作出令人信服的程序正确性证明。一个含有不确定特性的软件，就难于取得用户的信任，更难于作出程序正确性证明。随着软件任务变得越来越复杂，可靠性要求就变得越来越重要了。

（2）软件必须可维护

随着软件系统的开发日益复杂，成本日益增高，出于经济上的考虑，已不再轻易放弃一个已有的软件而重新再开发一个类似的软件，通常都是对现有的软件进行修改，以满足新的要求。另外，对一个复杂系统的开发，一开始就给出正确的实际需求几乎是不可能的，人们只能通过逐步改进来达到所期望的目标。综上所述，软件的可维护性是极为重要的。

（3）软件必须有效地执行

软件系统的执行效率是评价软件的一个重要指标，它与程序设计语言是否可在现有的体系结构上有效地执行，以及所选用的算法有着密切的关系。在软件环境中采用适当的工具，如某种合适的语言，可以提高软件的可靠性、可维护性和有效性。现就涉及语言的问题进一步加以讨论。

1.6.1 程序设计语言与可靠性

软件的可靠性是由语言的下列性质所决定的：

（1）可写性

可写性是一个难以捉摸的性质，基本上是指，用一种对问题以自然的方式来表述一个程序的可能性，以便程序员不必注意语言的细微末节和技巧，把精力集中于问题求解上。虽然这是一个很主观的准则，但谁都认为高级语言比低级语言（机器语言和汇编语言）的可写性好，一个汇编语言程序员的注意力常常被分散到某些数据的寻址机构和变址寄存器定位上。可写性越好的语言，程序员的书写错误就越少，从而使程序容易调试，软件的生产率越高。

（2）可读性

语言的可读性表现在它的程序逻辑可以进行跟踪。通过跟踪程序的检验，应能发现程序中所出现的错误。可读性也是一个主观准则，在很大程度上依赖于感觉和风格。然而，语言越简单，表达算法的方式越自然，通过检验代码来了解一个程序要做什么就越容易。例如，goto语句使得程序难于阅读，因为程序员不能以自顶向下的方式来阅读和理解程序，人们必

须在程序中跳来跳去查找 goto 语句的转移目标。当初有人提出使用 goto 语句的有害性，许多程序员持不同意见，然而，今天普遍认为绝大多数情况下应尽量避免使用 goto 语句。

我们将看到，许多能使人读起来容易的程序性质，也会给计算机自动检验程序带来方便。由计算机执行的自动检验对程序正确性具有很大的贡献。

（3）处理例外情况的能力

语言应具有处理非期望事件（如算术溢出和无效输入等）的能力，并应对这些事件给出适当的响应，以便在异常情况下，系统的行为也是可以预料的。

1.6.2 程序设计语言与可维护性

软件的可维护性，要求所用语言编写的程序具有良好的可读性和可修改性。可修改性在某种意义下也是主观的准则，但我们能指出一些性质，可使得程序更具可修改性。

例如因子化，所谓因子化就是关于某个事实的描述只出现在程序中的某个位置，当在几处重复的某个代码段出现时，就可以提公因子于一个子程序中，并用子程序调用来代替该代码段的出现。如果给子程序以某种含义的名字，程序就变得更加可读了。当要修改这个程序时，只需修改子程序，从而增强了程序的可修改性。

再如，许多语言允许给常数定义一个符号名。当对一个常数选择一个适当的名字时，增强了程序的可读性，用 PI（即 π）就比用3.14读起来方便得多。当需要对这个常数进行修改时，不用修改它的每次出现，只需修改一次常数定义，从而增强了程序的可修改性。

1.6.3 程序设计语言与效率

执行效率的要求一直是指导语言设计的重要因素，许多语言把效率作为主要设计目标，有的是隐含的，有的是显式的。例如最初设计 FORTRAN 时，是为了在特定的机器 IBM 704上实现的，因此 FORTRAN 对数组的维数及下标表达式的形式所采取的种种限制，都是为了能在 IBM704机器上能有效地实现。

然而，人们对效率的观点已经有了较大的改变，现在已不只是以执行速度和空间来衡量效率了。开始生产一个程序或系统所做的努力，以及将来维护方面所做的工作，都被看成衡量效率的内容。换句话说，我们对软件开发的生产率比最终产品的性能做了更多的考虑，这是对语言的一个很大的冲击。

如果一个语言具有可写性、可维护性和可优化性，那么称该语言是支持效率的语言。可写性和可维护性已经进行过讨论，可优化性是指允许程序自动优化的特性。

由于通常程序设计把大部分时间都花费在解决问题的有效方法上，所以可优化性是很重要的。对一个问题进行程序设计的早期阶段，不必将注意力放在优化上。最理想的方法是，先生产出一个正确的程序，然后通过一系列提高有效性的变换，从而得到一个既正确而又有效的程序。如果一个语言能够自动进行这种变换，那么就称它为可优化语言，它应具有良好的数学性质。goto 语句的使用使得自动优化变得复杂了，一般说来，许多降低可优化性的性质同样会影响可读性。

§1.7 程序设计语言发展简介

在这一节中我们不打算对语言的发展历史作详尽的综合评述，因为这不是本书的目的。同时，这方面的著作和论文很多，读者若有兴趣，可以查阅有关的资料。

在这一节仍坚持我们的宗旨，主要通过语言的发展过程，对语言设计所提出的种种概念加以评价，并展望这些概念的前景。

从今天软件开发过程的观点来看，最早的软件开发仅有编码阶段。早期的计算机仅用于科学计算，一个任务可由一个人编写程序加以实现。他所要解决的问题（如微分方程求解）都是被他完全理解的，无需进行需求分析或设计规范说明，甚至维护，只需从数学家那里找到相应的数值解法就可以编程实现。这时的语言着眼于支持程序员个人。按今天的观点来说，程序员所进行的工作仅是简单的应用。

随着计算机的发展，人们期望计算机应用日益广泛，任务也越来越复杂，计算机处于一个越来越难于理解和更加复杂的环境之中。只靠程序员个人在脑子里进行需求分析和设计已不适应复杂任务的要求，这时需要一个程序员“队伍”来共同进行需求分析和设计，协同工作完成预定的任务。同时，开发一个系统需要花费大量的人力和物力，当要求一个新系统时，即使出于经济上的考虑，也不能完全把旧系统抛开，而是将它修改和完善来满足新的要求，因而，程序维护就成为一个重要的现实问题。

系统变得越来越复杂，系统的可靠性就成为另一个重要的现实问题。一个不甚了解计算机的人使用一个不可靠的系统，可能会出现一个无所适从的局面。若把一个不可靠的系统用来控制核发电，当出现系统失败时，可能会出现灾难性的后果。

人们面临某个计算机应用，设计一个程序设计语言来实现这种应用。在使用过程中发现它的不足（局限性），或者有新的应用要求，或者要扩充应用的能力，就又设计新的语言来适应情况的变化，这就促进了语言的发展。下面我们就沿着这个发展过程，展开一些讨论。

1.7.1 早期的高级语言

第一个高级语言的定义可能要追溯到五十年代。当时计算机非常昂贵而又极为稀少，有效地使用计算机是一个重要的问题。另一方面，机器的执行效率也是人们追求的。为了有效地使用计算机，人们设计了高级语言，用以表达用户的需求。然而，高级语言程序需要翻译，机器才能执行，它占去了一些机器时间，影响机器的执行效率。但是，实践证明，高级语言是有效地使用机器与机器执行效率之间的一个很好的折衷。

这一历史时期最重要而又最有代表性的语言是 FORTRAN、ALGOL60 和 COBOL。FORTRAN和 ALGOL60 是用于解决科学数值计算的工具，即解决相对说来数据少而又简单的复杂计算问题。COBOL 是用于解决数据处理问题的工具，即解决数据繁多的简单计算问题。

这些语言在整个计算机科学历史发展中起了重要的作用。它们证明了高级语言的设想在技术上是完善的，在经济上是可行的。此外，这些语言还引入了大量的重要概念。例如，FORTRAN 引入了独立开发和编译子程序的模块概念，以及在模块中通过全局(COMMON) 环境共享数据的可能性。ALGOL60引入了分程序和递归过程概念，它首次对语言