语言的开发和标准化

语言开发的初始阶段：

- 一个人或一个设计组，根据对实际程序设计所需的一批基本要素的考虑，提出有关语言的基本构想，定义该语言的语法形式和语义
- 实现一个语言处理系统，使这个语言能实际用于程序设计
- 通过一些人的使用和反馈，修改完善语言的设计

少数比较成功的语言可能进入下面阶段：

- 将语言的处理系统提供给更大范围的用户，并得到更多反馈
- 除语言设计者（拥有者）外的其他人参与该语言实现的研究和开发，并引起软件产业界的注意，形成语言实现的产品和用户社团
- 受到标准化组织的注意，成立相应标准化小组（公司语言有版权问题）
- 经过认真的标准化工作，产生语言的标准化文本

参考：《C++语言的设计和演化》，Stroustrup，中译本：机械工业 2002

语言的开发和标准化

语言标准化的主要工作：

- 严格检查语言中的各种结构，严格定义它们的形式和意义
- 根据客观情况的需要，考虑加入新特征并给出严格定义
- 处理语言中各方面的一致性问题
- 考虑并确定语言中过时的旧特征，明确说明这些特征将逐步被淘汰

标准化通常是一个很困难的过程，其中一个困难是遗产问题：

- 与标准化前的版本或者前一个标准化版本的向后兼容问题
- 大的改变，可能使许多已有程序变得不再合法，招致老用户的反对
- 不改变，语言就可能由于逐渐过时而被淘汰

Fortran的几次标准化（尤其是Fortran 90），Ada 95的标准化，都特别明显地反应了这些问题。C++ 也做了两次标准化（1998,2010）

语言的实现：抽象机器

一部计算机就是连接起来的一组硬件器件

- 其作用是实现机器语言程序描述的计算过程
- 使用者可以不关心其内部实现，只关心其机器语言（指令形式和意义）
- 即使具体的硬件变了，只要它们提供的机器语言不变，使用方式没变，以前的程序都仍然可以用
- 因此：机器语言可以看作计算机硬件的“抽象”。一种机器语言对应一类计算机，或说对应于一种“抽象计算机”。如 X86 机器语言

一种高级语言也可以看作是一种抽象“计算机”的机器语言

例：C 语言，可以看作能直接执行 C 程序的高级“计算机”的“机器语言”

- 该“计算机”提供了 C 语言的各种基本的和高级的数据结构
- 能执行 C 语言的各种基本计算，基本操作和控制结构

常常没有这种计算机，因此需要考虑在已有的计算机上实现它

语言的实现

语言的实现牵涉到两种抽象机器（两种语言）：

- 需要实现的程序语言定义了一台抽象机
- 某种现存的准备用于运行程序的计算机也定义了一台抽象机
- 实现一种语言，就是在一台抽象机上做出另一台抽象机，用一台已有的抽象机去模拟另一需要实现的抽象机的行为（扮演另一不同的抽象机）

通用图灵机的存在性以及图灵论题说明：

只要一台抽象机的功能足够强（其功能等价于通用图灵机，具有图灵完全性），就可以用它实现其他任何抽象机的行为

现代计算机都具有图灵完全性，因此可以用于实现任何程序设计语言

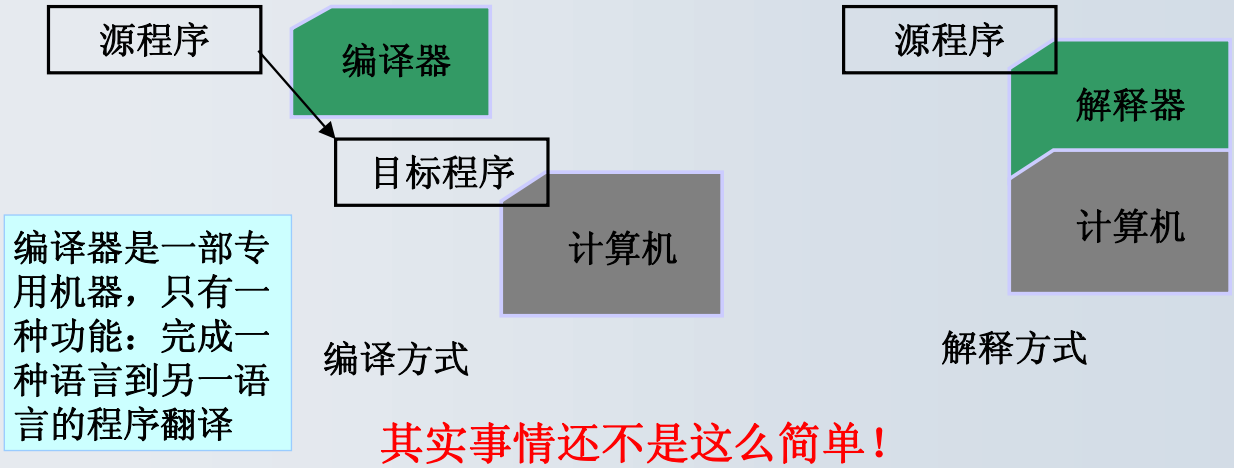
在抽象机 A 上实现抽象机 B，就是希望基于 A 执行用 B 的语言写的程序

- 如何在抽象机 A 上实现抽象机 B？
- 存在哪些可能的技术路线？

实现：方式

常见的说法是（高级）语言的实现有两种方式，编译和解释。

- 编译：把源程序编译为机器语言目标程序后执行
- 解释：在目标机器上实现一个源语言的解释器，由这个解释器直接解释执行源语言程序（它实际上实现了另一部抽象机）

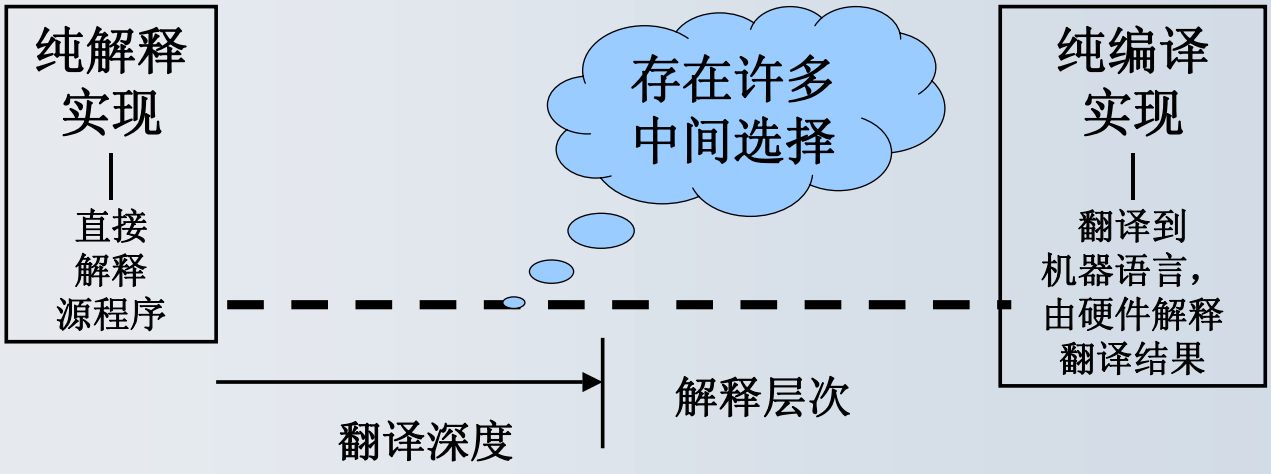


其实事情还不是这么简单！

实现：方式

语言实现方式多种多样，纯粹的编译或纯粹的解释只是两个极端

- C、Fortran 语言的常见实现方式可以认为是比较纯粹的编译方式
- 早期的 BASIC，DOS 的 bat 文件，现在有些脚本语言实现，采用的基本上是纯粹的解释方式

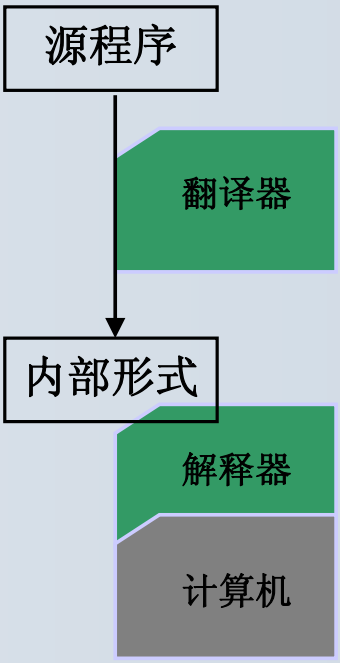


实现：Lisp

人们常说 Lisp 是解释的。其实不准确！

Lisp 的基本实现方式：

- 先把源程序翻译成一种内部形式（类似于语法树的数据结构）
- 由解释器解释这种内部形式（遍历数据结构，实现程序所描述的行为）



许多 Lisp 系统的翻译器比较简单（翻译较浅）

这只是一种常见的基本实现方式

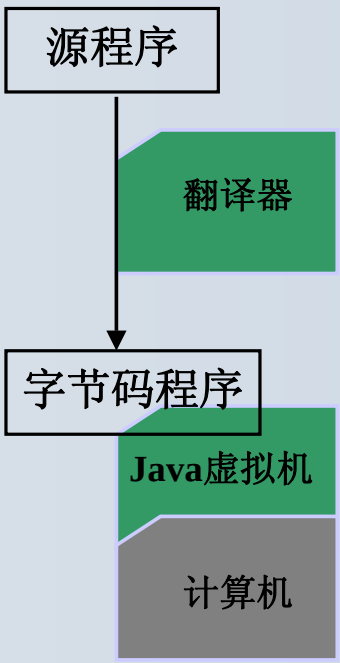
目前也有许多Lisp系统采用更复杂的实现方式，例如采用即时编译（Just-In-Time Compilation）和动态编译（Dynamic Compilation）技术

实现：Java

Java 的基本实现方式与 Lisp 类似，但所用的中间表示（字节码，bytecode）更接近计算机（翻译较深），而且有外部存储形式（独立存在）

由于 Java 已成为软件开发的一种主流语言，Java 程序的执行效率受到特别重视，人们用了许多技术来提高执行效率（这些技术都是早在函数式语言领域开发的）

- 即时编译（JIT）：在装载字节码程序时，即时地将程序编译为运行所在的机器的本地代码程序，而后直接执行（由硬件直接解释）
- Hotspot 和动态编译：为避免即时编译带来大的开销，开始时直接解释执行并监视执行情况，如果发现热点（如频繁执行的循环、频繁调用的函数），就动态编译这部分代码，生成本地代码



实现：C++

- 第一个 C++ 实现（B. Stroustrup）用一个编译器把 C++ 源程序编译为标准 C 语言程序，而后就可以用任何 C 编译器实现

  这个 C++ 编译器做完全的语法和静态语义检查，如果完成第一个编译，得到的 C 程序是没有错误的

- 现在许多试验性语言和实现采用把 C 语言作为编译器目标的技术

  B. Stroustrup 在《C++ 语言的设计和演化》里仔细讨论了这种实现

许多计算机（特别是一些 CISC 机器）采用微程序结构

硬件提供一套微指令，对外提供的机器指令集通过一组微程序实现，由一个硬件的微程序解释器解释执行

在这种机器上，编译器生成的结果并不是硬件可直接执行的微代码，而是 CPU 的硬件解释器实现的外部代码，由硬件解释器解释执行

也可能用编译器直接实现微代码

实现

- 程序功能的实现通常可以分为两个阶段：
- 静态处理阶段（static），在程序本身开始执行前的处理。通常包括：
  - 翻译阶段（编译）：对源程序做各种检查（语法检查，类型检查等）和变换，将其转变为某种适宜动态执行的形式
  - 连接阶段（可能）：构造出可动态运行的程序形式
  - 装载阶段（可能）：把可运行程序装入运行环境，必要的处理
- 动态执行阶段（run-time），指程序的实际运行期间
- 课程讨论中经常提到静态处理阶段和动态运行阶段

例：若变量 x 的（绝对或相对）位置可静态确定，运行中就可以直接访问。若不能静态确定，运行中每次访问都要查表，效率就比较低

语言的“编译实现”：在静态阶段完成尽可能多的处理工作，对源程序做深入的分析和变换，生成的目标形式通常与源程序差异巨大

实现

编译实现：

- 源程序只处理一次，可任意次执行；运行效率高，可通过深度优化而尽量利用运行平台的特点；目标程序运行时不需要复杂的运行系统支持
- 编译耗时；源程序的许多信息在编译时丢掉了；灵活性差（编译时把许多东西固定下来了）；对程序开发的支持比较复杂

解释实现：

- 可以提供更大的灵活性（例如程序可以动态变化）；源程序的信息可用；比较容易实现功能强大的程序开发系统
- 可能要反复检查分析程序的信息，导致较低执行效率；难做程序优化
- 有些语言或语言特征难以编译实现
- 常规语言（如C/Fortran）的主要设计目标之一就是有效的编译实现
- 一些程序开发环境结合了这两方面，在开发时解释执行，发布前编译

语法、语义和语用：简述

语法：规定合法程序的形式，常用某种形式化描述，如BNF等语法描述方式清晰定义语言中各种结构的形式。（编译原理课程的内容）

上下文关系通常用自然语言说明，例如：变量要先定义后使用，表达式里的变量应该与运算符的类型匹配。这类规定也被称为静态语义，编译器可以检查有关规则。这方面最重要的是“类型检查”

语义：规定合法的程序（各种程序结构）的意义，即程序执行时所产生的效果。语言手册中用尽可能严格的自然语言叙述。

形式语义学：研究如何用严格的形式化的方式，定义程序设计语言的语义，进而也就定义了所有程序的语义。（计算机系的研究生课）

语用：程序设计技术，具体语言的使用技术和惯用法等。这些不是语言规范规定的，但语言的设计中隐含了对这些方面的许多考虑。（程序设计技术，程序设计方法学，设计模式等等）

语言的设计目标及其演化

Fortran 设计中最主要的考虑是易用性（与机器和汇编语言比较）和高效实现，特别关注程序能翻译成高效执行的代码，因为这样才可能被接受（今天 Fortran 仍高效）。随着计算机变得越来越快，越来越便宜，效率问题虽然还是很重要，但重要性已大大下降。易用性方面的考虑仍非常明显：

- 提高程序设计工作的效率
- 帮助人们提高程序（软件）的质量，可靠性
- 设法支持某些高级的软件设计技术

语言的最主要作用是用于描述所需要的计算过程。为此需要：

- 清晰，简洁的形式（例子：C，Pascal，APL ）
- 清晰简单的语义（易理解，易验证）
- 正交性（避免重复的可相互替代的特征，人们对此有些不同意见）
- 可读性（人容易阅读理解的东西，计算机也容易处理）

语言设计：目标演化

随着程序设计语言的发展，语言的设计目标也发生了很大变化

- 语言的初始设计目标就是更方便地为计算机写程序
- 后来人们认识到，程序设计语言也是人的工具，用于描述算法、交流算法，用于服务于交流、教学和科研的需要

随着计算机应用发展，程序变得越来越复杂，开发程序变成代价高昂的工作。为支持复杂程序开发，提高开发工作的效率，语言设计有了许多新目标：

1，支持对基本语言的扩充，提供各种扩充定义和抽象机制

- 过程、函数定义机制，扩充语言的基本操作
- 数据类型定义机制（及OO机制），扩充数据描述方式和功能

例：C++ 在语言机制扩充方面有许多考虑（如运算符重载）

可扩充语言（Extendable Languages），允许程序形式的改变（Lisp）

语言设计：目标演化

2，提供支持复杂程序所需的高级组织的机制，支持大型程序开发

- 模块机制（信息隔离和屏蔽）
- 支持分别编译的机制，支持程序的物理组织

3，支持软件重用，包括软件中的部分的重用

- 支持通用的基本程序库。 Pascal 失败之处之一就是忽略了库的开发。C/Fortran 都做得很好。Ada、C++ 和 Java 的设计都特别考虑了对库的支持。许多新语言定义了功能非常丰富的标准程序库
- 支持库开发：库是最基本的重用方式。是否支持库开发，决定了语言能否大范围使用。支持用户和第三方供应商开发各种扩充的和专用的库
- 支持某些层次或者方式的软件部件概念

问题：库开发或者部件是否需要本语言之外的功能？

OO 概念可能在上面许多方面起作用，因此成为复杂软件开发的重要方法

语言设计：目标演化

语言设计中需要考虑的另外一些重要问题：

- 正常处理的异常/错误处理的良好集成（在产品软件的程序里，处理错误和各种特殊情况的代码占很大的比例，可能达 70%）
- 对于程序的易修改可维护性的支持
- 对于并发程序设计的支持，用什么样的机制支持并发程序设计。这方面的问题将长期成为语言研究和设计的热点问题
- 安全性设计：是否有助于程序员写出安全可靠的程序？这一问题在未来许多年都会是语言设计的一个重要关注点
- 对于业务流程和事务处理（transaction）的支持，如对于 all or nothing 语义（或者完成整个工作，或者什么效果也没有）的支持

由于语言承载的功能越来越多，设计时需考虑的问题越来越多，新语言正在变得越来越复杂，语言的实现需要做的工作也越来越多（基本处理、对开发过程的支持、库等等），设计一种语言，支持所有需要变得越来越困难

语言设计：通用和专用

常规程序语言是通用的（**general purpose languages**），设计目标是支持所有可能的程序的开发工作

随着应用的发展，程序变得越来越复杂，程序的规模越来越大。直接在基本语言机制的基础上描述特殊的功能也变得越来越困难

在常规语言的基础上支持专门用途的可能方式：

- 扩充常规语言，增加特殊的服务于专门用途的机制。这方面的例子：

  数据库查询，图形用户界面，异常处理等等

- 利用常规语言的扩充机制，开发支持特殊使用领域的程序库，实例：

  扩充的标准库（如**Java**和许多脚本语言），支持平台开发的扩充库，各种第三方软件商提供的库

在常规语言里支持专门应用，受到基本语言的描述方式的限制。处理复杂问题的一种途径是使用专门的描述方式，设计提供特殊的专门用途语言

语言设计：通用和专用

目前有一个研究领域称为 **Domain Specific Language，SDL**

- 研究具体的专用语言的设计和实现技术
- **SDL** 常常是比较小的轻量级语言，人们也在研究支持这类语言的通用框架，为它们的设计和实现提供支持，实现有价值的支持工具

语言实现的最基本支持工具已经比较成熟，有许多工具可以从语言的词法和语法描述出发，自动生成相应的词法和语法分析器

进一步的问题是语言中的上下文关系（如类型关系），人们正在开发包含类型检查功能自动生成的语言支持工具

专用语言的问题：除了专用部分外，常常还需要通用功能和部分，如使用其他语言的库，实现 **I/O** 和图形用户界面等。常用技术：

- 提供与常规语言的接口，允许用常规语言扩充功能
- 把专用语言嵌入常规语言中

语言设计：实例

不同设计目标导致了不同的语言设计：

- **Fortran**：高效的大规模数值计算
- **Lisp**：灵活的符号表达式处理，数据与程序的统一（后来认识到重要性）
- **Pascal**：结构化程序设计，教学，清晰的基本程序设计概念
- **C**：系统程序设计，低级操作与高级语言结构的结合
- **Smalltalk**：概念的统一性和最小化，面向对象
- **C++**：复杂的系统程序设计，低级机制与高级抽象机制（数据抽象的面向对象）的有机融合，支持多种范型的程序设计技术
- **Ada**：复杂软件，复杂的嵌入式系统、实时系统
- **Java**：支持面向对象的程序设计，平台无关性，一次编程到处使用，网络环境下的应用，安全性

不同考虑，都可能造就出成功的语言

重要性

程序设计语言的研究和开发处于计算机科学技术发展的中心：

- 计算机理论和方法的研究，许多是由于语言发展的需要
- 许多理论研究成果体现到程序语言的设计中
- 实际应用中最本质的需要常反映到程序语言里，推动语言的演化和发展
- 语言实现的需要是推动计算机体系结构演化的一个重要因素（如**RISC**）
- 计算机硬件的能力和特征也对程序语言的发展变化有着重要影响（今天和明天，并行性问题）
- 理解程序设计语言，有助于提高对整个计算机科学技术领域的认识

推动语言演化发展的要素：

实际应用的需要，硬件的发展和变化，人们对于程序设计工作的认识发展，实现技术的开发，理论研究的成果

重要性：图灵奖

1966-2010，45届图灵奖，15届由于与程序设计语言有关的工作获奖：

1966, Alan J. Perlis, 早期语言和Algol 60的贡献，图灵奖第一位获奖者

1971, John McCarthy, LISP语言, 程序语义, 程序理论

1972, E. W. Dijkstra, Algol编译, 结构化程序设计, 并发概念和原语, 形式化推导, 卫式命令等

1977, John Backus, Fortran语言, FP语言, BNF等

1978, Robert Floyd, Algol编译, 编译技术, 程序优化, 归纳断言法和前后断言, 程序正确性, 编译生成

1979, K. E. Iverson, APL语言

1980, C.A.R. Hoare, 结构化程序设计, case语句, 公理语义学, 并发程序理论, CSP 等

1983, Dennis Ritchie 和 Thompson, C语言和 UNIX

1984, Niklaus Wirth, Algol W, PL 360, Pascal, Modula-1/2, Oberon, 逐步求精, 结构化程序设计, 语法图

1991, Robin Milner, ML语言, 并发理论, CCS

2001, Ole-Johan Dahl 和 Kristen Nygaard, Simula语言, OO概念

2003, Alan Kay, Smalltalk语言, OO概念、语言和程序设计

2005, Peter Naur, Algol 60 语言的设计和定义, 编译, 程序设计的原理和实践

2006, Frances Allen, 优化编译和并行化

2008, Barbara Liskov, 数据抽象/OO/容错/分布式计算程序的基础和语言

2005/2006/2008 图灵奖

2005: Peter Naur, "fundamental contributions to programming language design and the definition of Algol 60, to compiler design, and to the art and practice of computer programming."

2006: Frances E. Allen, "pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution."

2008: Barbara Liskov, "has led important developments in computing by creating and implementing programming languages, operating systems, and innovative systems designs that have advanced the state of the art of data abstraction, modularity, fault tolerance, persistence, and distributed computing systems. ... The CLU programming language was one of the earliest and most complete programming languages based on modules formed from abstract data types and incorporating unique intertwining of both early and late binding mechanisms. ..."

新趋势：并行

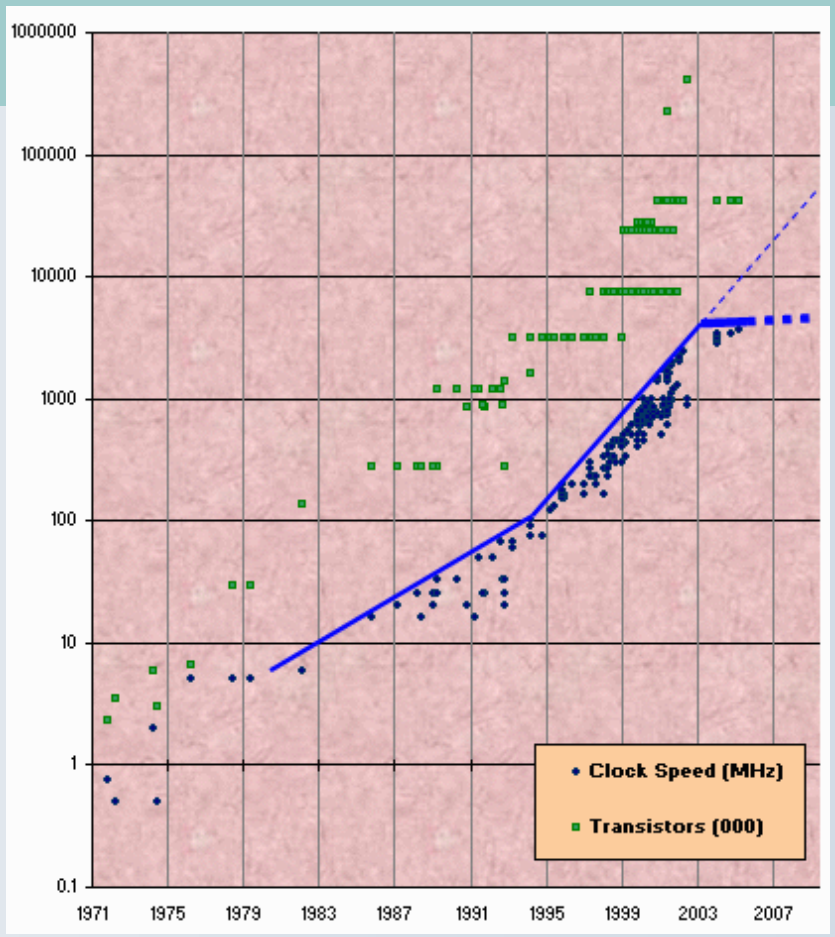
狭义的摩尔定律已失效，提高主频的趋势已停止

并行环境已逐渐成为我们周围最常见计算机的基本结构的一部分

如何做并行程序设计的问题变成对每个计算机工作者的挑战

程序设计语言也需要反应这方面的需求

有关并行语言、程序和程序设计的问题，将在今后很多年里成为程序设计语言研究领域里最重要的问题



新趋势：并行

有关并行系统和并行程序设计的研究已经进行了近40年，但对并行系统和如何设计实现并行系统的认识仍很不成熟：

- 已开发的并行系统（及分布式系统）经常出现意料之外的错误
- 并行系统的开发方法很难使用，开发低效，对开发人员缺乏良好支持
- 描述并发系统的记法形式过于低级和细节，缺乏有效抽象手段
- 并发系统的验证技术不成熟，系统缺乏可靠性的保证

对于上述问题的研究和并发程序开发实践将未来语言的发展影响有重大影响

- 许多新语言里加入了并行特征，包括 Java、C# 等
- 一些并行理论的研究成果被用于实践，如 JCSP
- 人们重新开始重视无状态的程序设计，函数式程序设计（如 Erlang 语言受到许多人推崇），提出了一些新想法

这方面的理论和实际技术研究将成为很长时间的研究热点

新趋势：脚本语言

近年脚本语言在计算机应用盛行起来，重要实例：

- 用于开发 **Web** 服务端的 **PHP**、**ASP**、**JSP** 等
- 用于 **Web** 客户端网页嵌入应用的 **JavaScript** 等
- 用于更广泛的应用开发的 **Perl**、**Python**、**Ruby** 等
- 其他各种专门用途的脚本语言，如描述图形界面的 **Tcl/tk**

与通用程序设计语言相比，通用脚本语言有如下特点：

- 丰富的基础数据结构，灵活的使用方式，支持快速的应用开发
- 基于解释器的执行，或者解释和编译的结合，可以立即看到开发的效果
- 通常都没有标准化，随着应用的发展变化和很快地扩充
- 一些语言形成了很好的社团，开发了大量有用的库

脚本语言将如何发展？其发展趋势怎样？

新趋势：其他

软件设计技术的一些新趋势

- 基于组件和服务的软件开发
- 业务流程和事务处理（**all or nothing** 语义）
  - **Web** 服务和服务组合，如 **WS-BPEL** 等
  - 分布式系统的全局描述语言，如 **WS-CLD** 等
  - 复杂流程的描述，语言基础
- **Aspect-Oriented Programming**
  - 对于软件的基于功能和特征的切分描述
  - 通过编织的实现

  会不会发展出直接面向 **AOP** 的语言？
- 等等