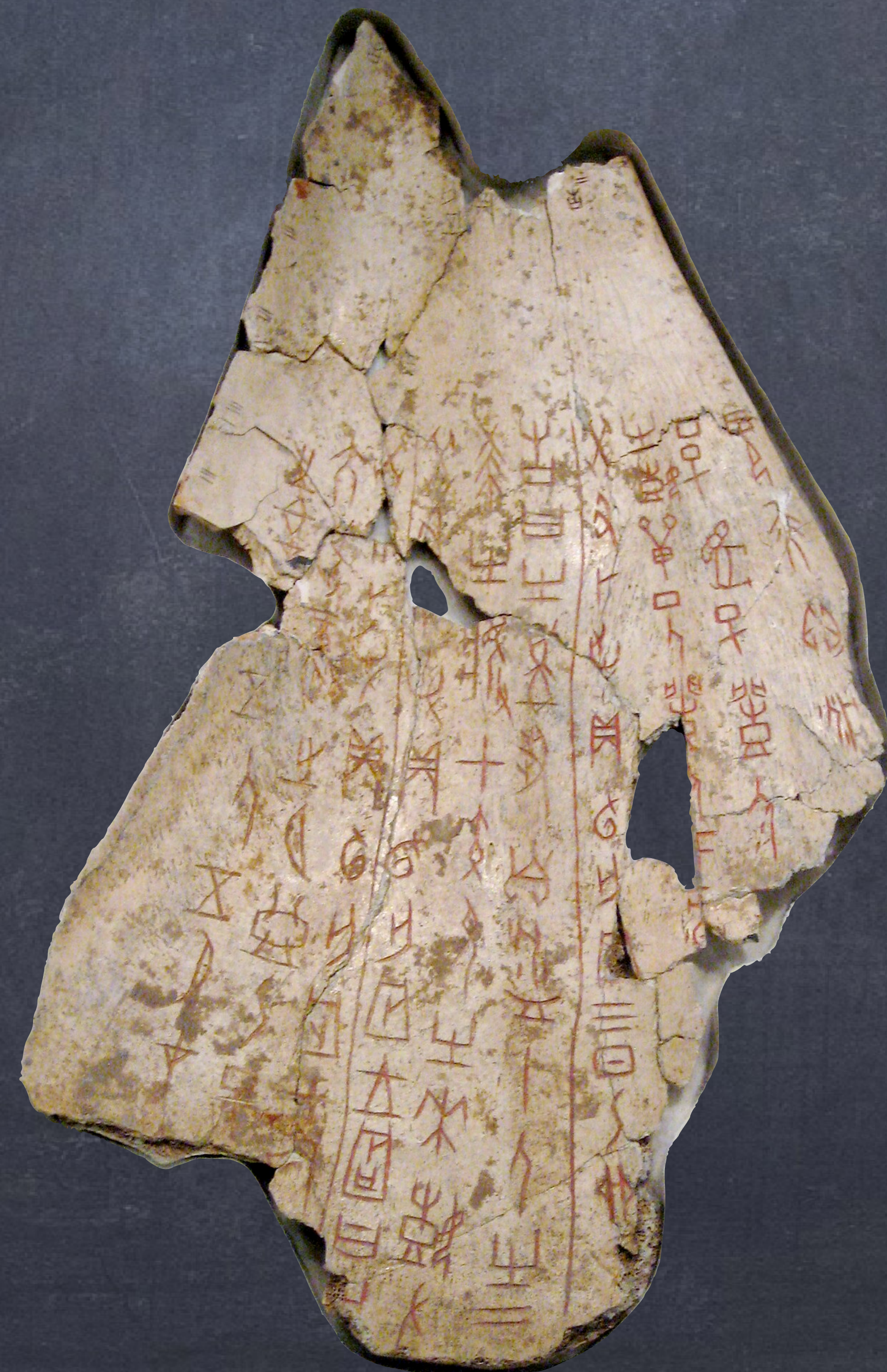
编程语言概述



提纲

- 编程语言的定义
- 编程语言学习的动机
- 编程语言的分类
- 编程语言的实现方法
- 编程语言的发展简史

什么是编程语言（programming language）？

编程语言

- 编程语言即为了书写计算机程序的标识符系统 (System of notation)。
  - 大部分是基于文本的形式语言
  - 但也有基于图形的

Refer to: Aaby, Anthony (2004). Introduction to Programming Languages.

编程语言

- 注意：不是所有的"计算机语言"（computer language）都是编程语言（programming language）
- 有些计算机语言并不是为了表达"程序"，比如：标记语言（Markup language） Markdown、Html、XML等
- 一般而言，可以用作一般性目的（general purpose）的计算机语言才是编程语言。

为什么我们要关心和学习编程语言知识？

学习的动机

1.可以提升表达能力

2.可以提高选择合适语言解决相关问题的能力

3.可以提高学习新语言的能力

4.可以更好的理解语言实现效果

5.可以全面提升解决软件工程相关问题的能力

学习的动机

1 提高表达能力！

Sapir-Whorf hypothesis, a.k.a, linguistic relativity:
"language is not simply a way of voicing ideas, but is the very thing that shapes those ideas. One cannot think outside the confines of their language."

学习的动机

- 考虑以下例子：
  - 没有"命名常量"或者枚举类型：
    - 那么只能用变量，然后初始化一个值，然后不再变化。
  - 如果没有类（比如C）
    - 那么只能尽力去用函数指针、结构体等形式去模拟。

学习的动机

2 提高选择合适语言解决相关问题的能力

- 比如对于软件分析这样的基于很多复杂规则的任务，使用基于逻辑的语言，如Datalog就比JAVA要顺畅很多
- 比如针对系统级开发，基于命令式的C语言要比脚本语言Python要合适

学习的动机

3 提升学习新语言的能力

- 大部分编程语言的核心概念往往相近
- 比如，你通过学习了JAVA中面向对象的设计思想，那么对于新语言比如Ruby，你就可以很容易就学会
- 很多C语言的表达式风格都被Python，JAVA继承了，所以学习JAVA不会很难

学习的动机

4 更好的理解语言实现效果

- 理解程序语言的实现方式将会让你了解一些结构的实现效率。比如，你知道了程序调用的实现方式，那么就会知道频繁地调用一个小的子程序是一种非常低效的设计

为什么不能只使用一个语言打天下？

学习的动机

- 一个好的语言需要考虑：
  - 可读性
  - 可写性
  - 可靠性
  - 成本

可读性

- 需要能够容易看懂和理解
  - **整体简洁性**：有时提供多种选项达成同样的目的会降低可读性，比如操作符重载，e.g. +，怎么理解"a" + "b"?

可读性

- 需要能够容易看懂和理解
  - **提供数据类型**：如果提供数据类型，往往会提高可读性。比如JAVA、C，但对于类似Javascript、python等支持动态类型语言，就会降低可读性

可读性

- 需要能够容易看懂和理解
  - 语法的设计：结构化的表达会影响可读性，比如你觉得C、JAVA用{}表达块好，还是Python的缩进方式更好？

可写性

- 能够容易的写出解决问题的代码
  - 正交性：只给出少量的原始构造规则，但是他们可以任意组合形成更复杂的应用。比如考虑数据类型(integer, float, double, and character) 和 数据操作符数组和指针，进行组合的话就会有intger数组、integer指针、float数组、float指针...

可写性

- 能够容易的写出解决问题的代码
  - 可表达性：提供简洁的对于运算的操作方式。比如对于C而言写一句 count++ 要比 count = count + 1 更加方便，JAVA中用for来写counting loops要比while更加简单

可靠性

- 一个程序能够在尽可能多的情况下执行不出意外
  - 类型检查：检查类型错误，比如提供机制在编译下或者运行时来发现类型不一致错误（比如赋给一个String A = true;）
  - 越早发现这种错误越好，减少差错代价。比如JAVA要求几乎所有的变量和表达式都要做完类型检查。而c语言在这方面几乎不做检查。

可靠性

- 一个程序能够在尽可能多的情况下执行不出意外
  - 异常处理：提供运行时错误的拦截，并采取处理措施。如C++、Java都提供这种可靠性方法，但c没有。

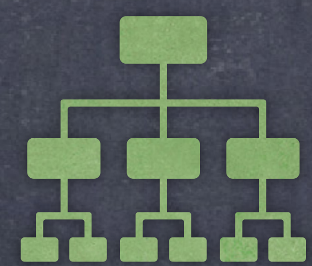
成本

- 学习曲线以及程序可维护性，是否整体简洁可读？
- 写一个程序的代价，可写性强不强？是否提供正交性？
- 执行成本如何？如果有大量的运行时检查，虽然提高了可靠性，但也加大了运行成本。是否跨平台（是否平台独立）？

学习的动机

- 然而很多都是对立的
  - 比如提供可靠性的类型检查（JAVA），却可能提升运行成本！
  - 可表达性过于强有时也会导致学习成本大！
- 因此，没有免费的午餐！

如此多的语言，难道都要学？


我们可以从更高的层面学习，提取共性

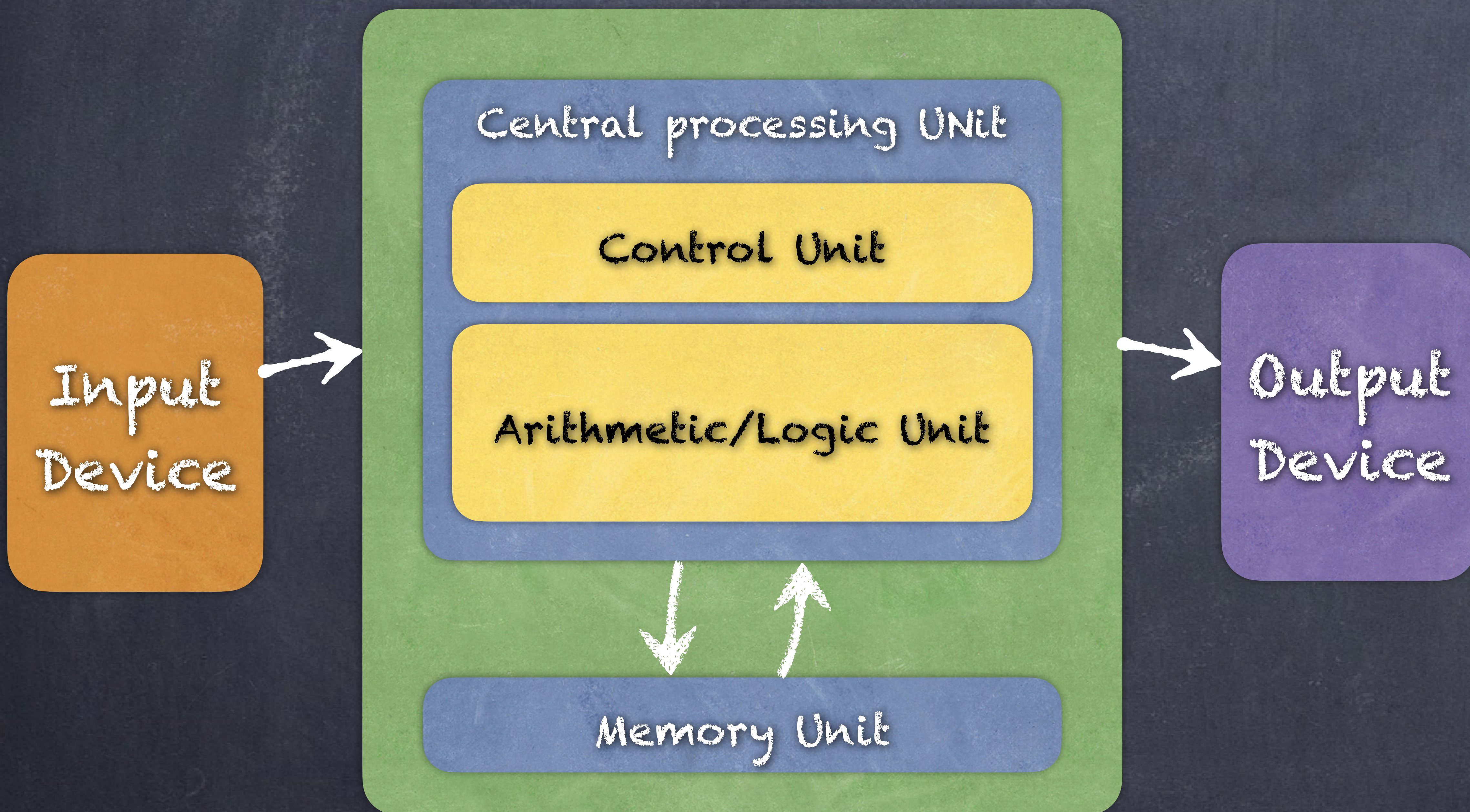
学习其中的典型

根据编程风格的分类

- 大体上，我们可以分为如下几类：
  - 命令式（Imperative）
    - 过程式（Procedure）
    - 面向对象（object-Oriented）
  - 声明式（Declarative）
    - 函数式（Functional）
    - 基于逻辑（Logic）

命令式

- 冯诺依曼结构



- fetch-execute cycle

```
initialize the program counter
repeat forever
    fetch the instruction pointed to
    by the program counter
    increment the program counter to
    point at the next instruction
    decode the instruction
    execute the instruction
end repeat
```

命令式

- (How)： 具体指明程序要做哪些步骤，一般支持如下几种语句

1，运算语句

2，循环

3，条件分支

4，无条件分支

命令式

```
Dictionary<string, Grouping> groups = new Dictionary<string, Grouping>();
foreach (Product p in products)
{
    if (p.UnitPrice >= 20)
    {
        if (!groups.ContainsKey(p.CategoryName))
        {
            Grouping r = new Grouping();
            r.CategoryName = p.CategoryName;
            r.ProductCount = 0;
            groups[p.CategoryName] = r;
        }
        groups[p.CategoryName].ProductCount++;
    }
}

List<Grouping> result = new List<Grouping>(groups.Values);
result.Sort(delegate(Grouping x, Grouping y)
{
    return
        x.ProductCount > y.ProductCount ? -1 :
        x.ProductCount < y.ProductCount ? 1 :
        0;
});
```

声明式

- (What)： 表达想要计算的逻辑，而不需要给出具体步骤
  - 我们其实真正关心的是解决问题，只要定义好问题，以及该问题的解需要满足的要求，答案应该不言自明！
  - 旨在最小化甚至消除与问题无关的琐碎步骤（比如定义中间变量、寄存器操作…）

声明式

```
var result = products
     .Where(p => p.UnitPrice >= 20)
     .GroupBy(p => p.CategoryName)
     .OrderByDescending(g => g.Count())
     .Select(g => new { CategoryName = g.Key, ProductCount = g.Count() });
```

命令式vs声明式

- An imperative approach (HOW): 命令式
  - "I see that table located under the Gone Fishin' sign is empty. My husband and I are going to walk over there and sit down."
- A declarative approach (WHAT): 声明式
  - "Table for two, please."

面向对象

- 目标：程序更加模块化和可维护
  - 用对象（Object）这个抽象来融合数据和函数！
  - 通讯机制：消息机制
    - (15 * 19) + (37 squared)
    - 向15发送消息*，参数为19；向37发送消息squared；最后向15*19的结果发送消息+，参数为37 squared的结果。

过程式 vs 面向对象

过程式

//起床开始

openEye(Bob);//睁眼，开始到结

Dressed(Bob);//穿衣服，开始到结束

getOutOfBed(Bob);//下床，开始到结束

//起床结束

面向对象

```
class People {
    public void getUp(){
        eye.open();//睁眼
        body.dressed();//穿衣服
        body.getOutOfBed();//下床
    }
}
```

函数式

- 函数就是一阶公民
  - 本质上，函数就是比特流，数据也是比特流，他们有什么不同吗？
    - 没有，因此，函数也可以像数据一样作为参数、返回值传递

函数式

C（命令式）：

```
int factorial(int n){
    int f = 1;
    for (; n > 0; --n) f *= n;
    return f;
}
```

Lisp（函数式）：

```
(defun factorial(n)
  (if (= n 0)
    1                                  //  若n等于0，则n!等于1
    (* n (factorial(- n 1)))))         //  否则n!等于n* (n-1)
```

逻辑式

- 证明即是程序！（Curry-Howard isomorphism）
  - 证明一个定理的过程 和 写一个程序是同构的，所证明的定理就是程序的返回类型（什么是类型，值的集合！）
  - 所以证明是可以被"执行的"！

逻辑式

- man(socrates).
- mortal(X) :- man(X).
- ?- mortal(socrates).

- 事实："苏格拉底是人"
- 规则："如果X是人，则X是凡人"，
- 查询："苏格拉底是否是凡人？"

逻辑式

Prolog（逻辑式）

```
// 0! 等于1
factorial(0,1).

// 若M等于N-1且 M!等于Fm且F等于N*Fm，则N! 等于F
factorial(N,F) :-   M is N-1, factorial(M,Fm), F is N * Fm.
```

实现方式

- 编程语言与其实现是不同的
  - 编程语言是创造出来为了让人类不必忍受枯燥乏味"低级"机器硬件指令，而能够"轻松"地进行编程的语言。
  - 而机器运行商则更加考虑的是如何让机器更加高效和经济，他们不会考虑人类的接受度。

实现方式

- 因此，实现编程语言需要：
  - 编程语言需要"某种机制"让其表达的程序在机器上执行起来
  - 这种机制就是翻译系统（translation system）

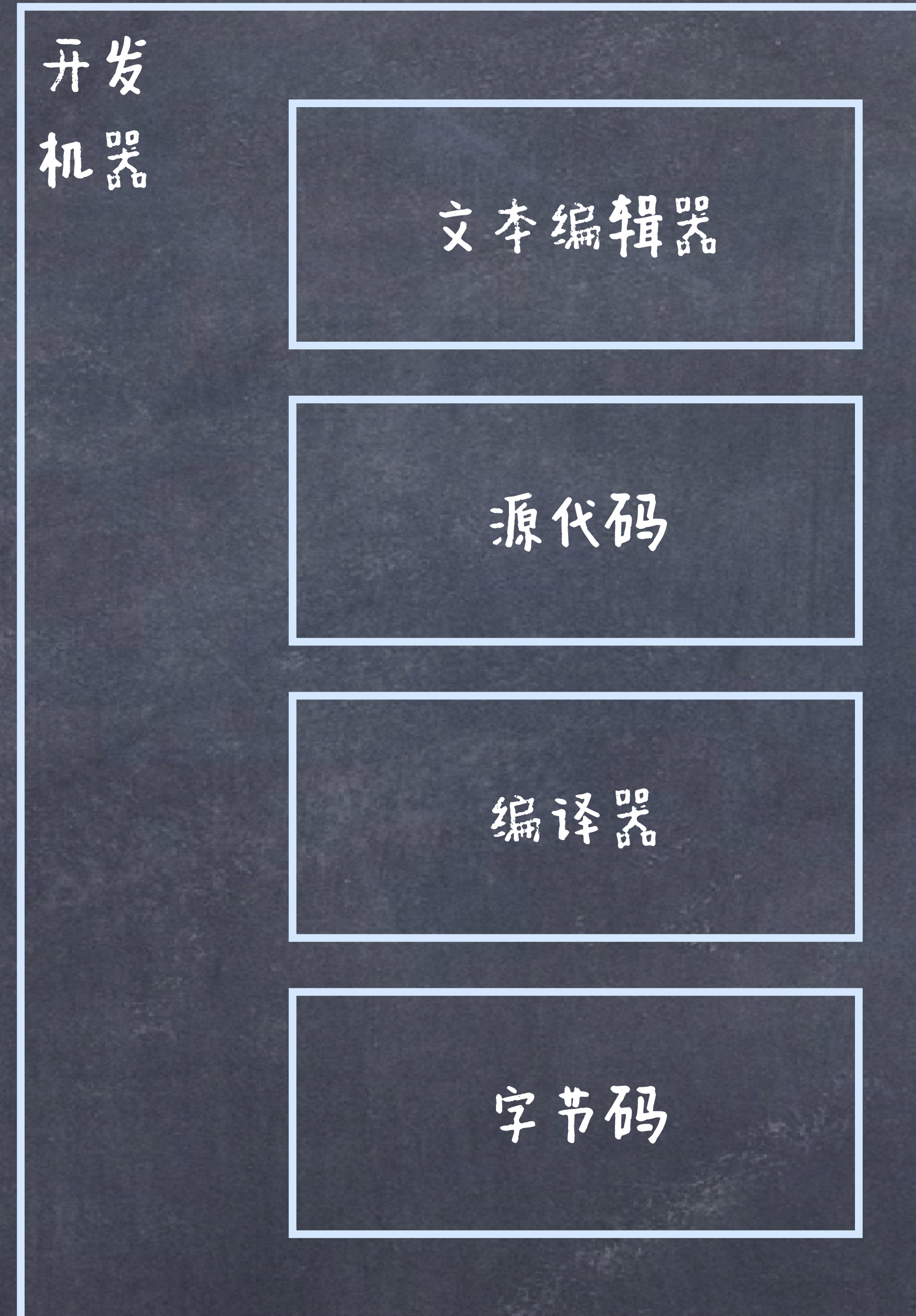
实现方式

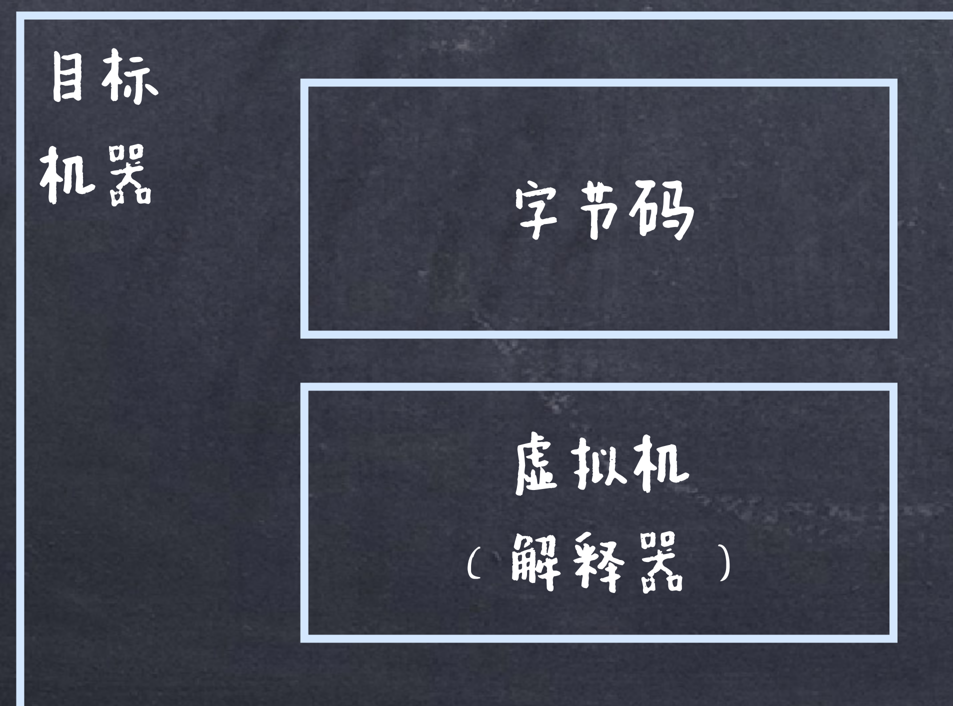
- 编译（compiling）
  - 把整个程序源代码翻译成另外一种代码，然后等待被执行，发生在运行之前，产物是「另一份代码」。
- 解释（Interpretation）
  - 把程序源代码一行一行的读懂然后执行，发生在运行时，产物是「运行结果」。

实现方式

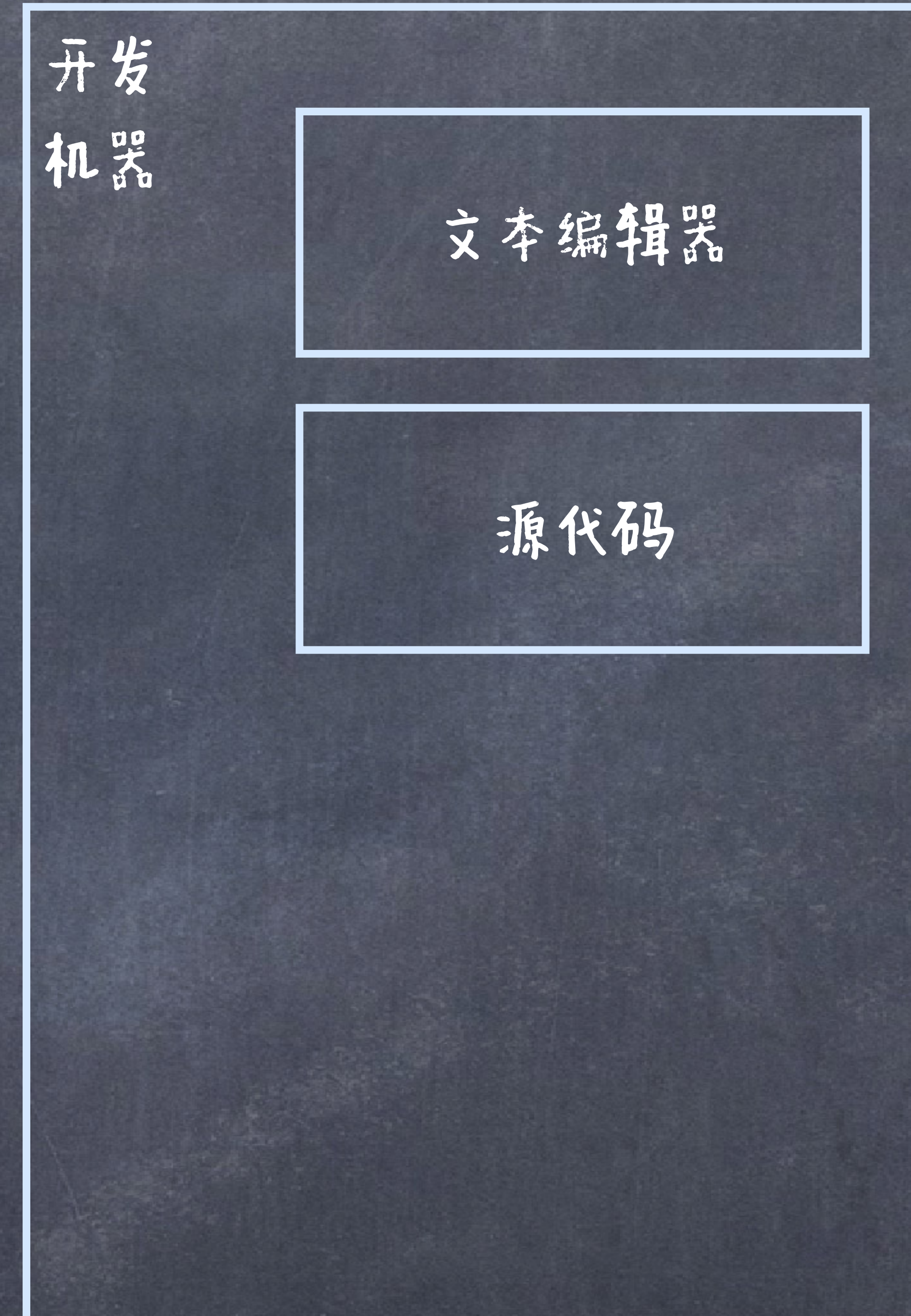
Example:

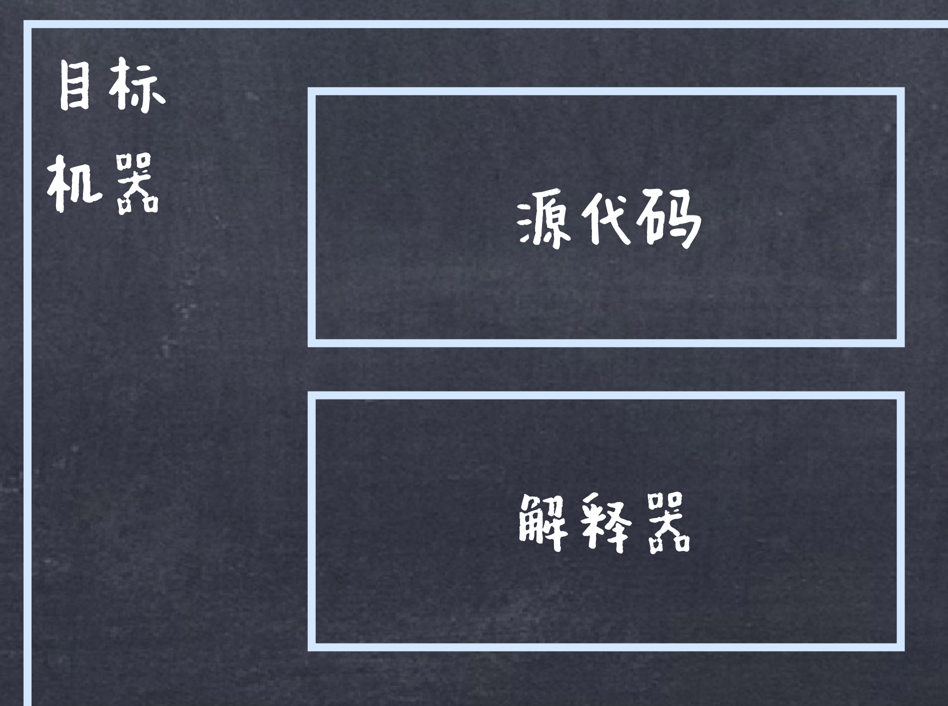
Java



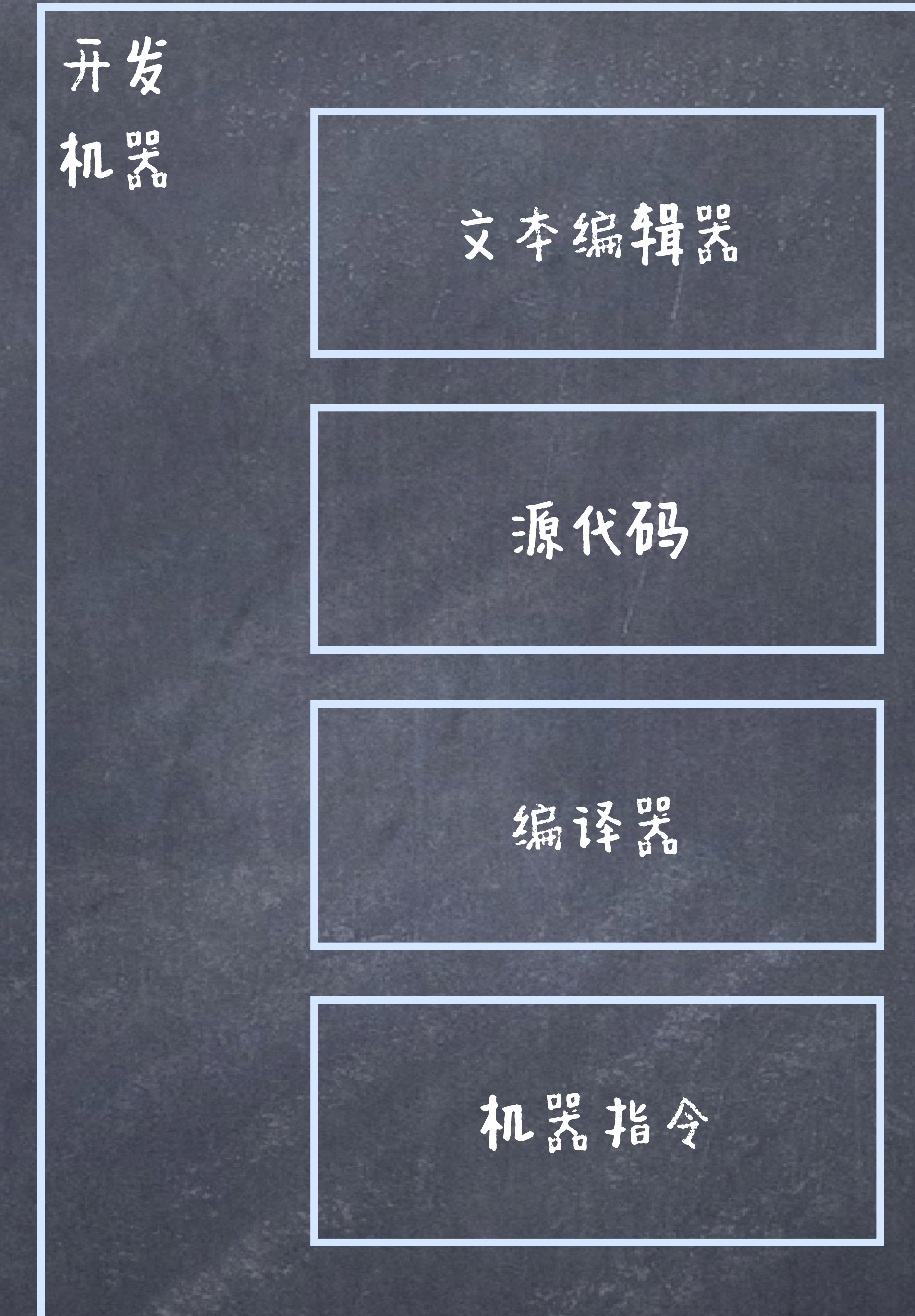


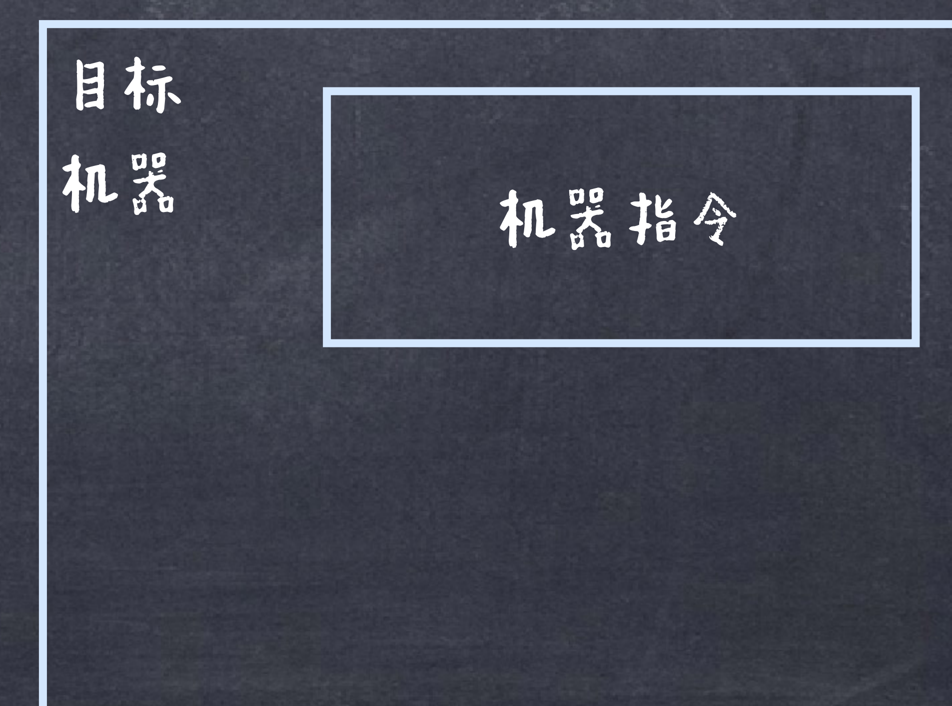
Basic





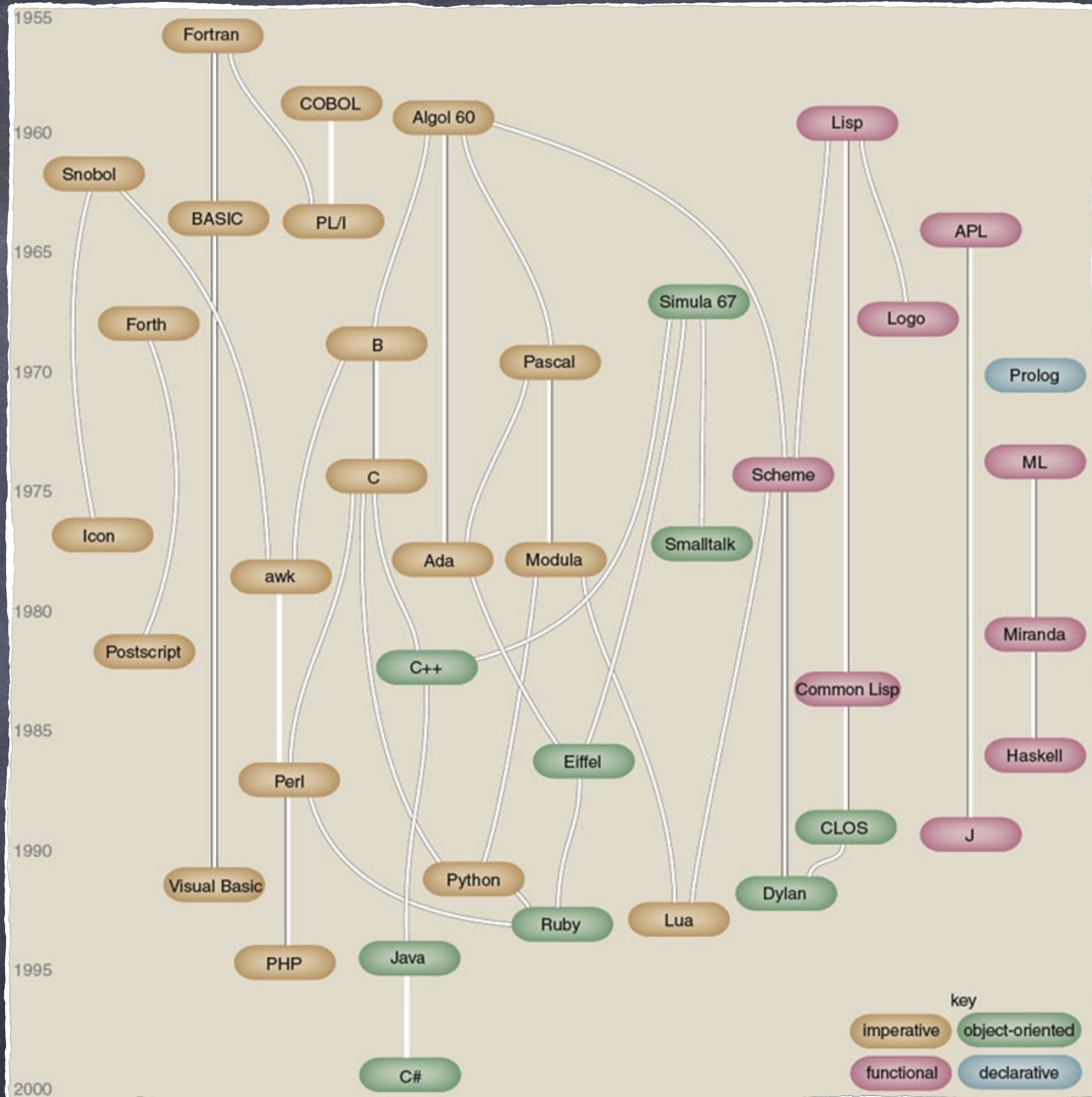
C





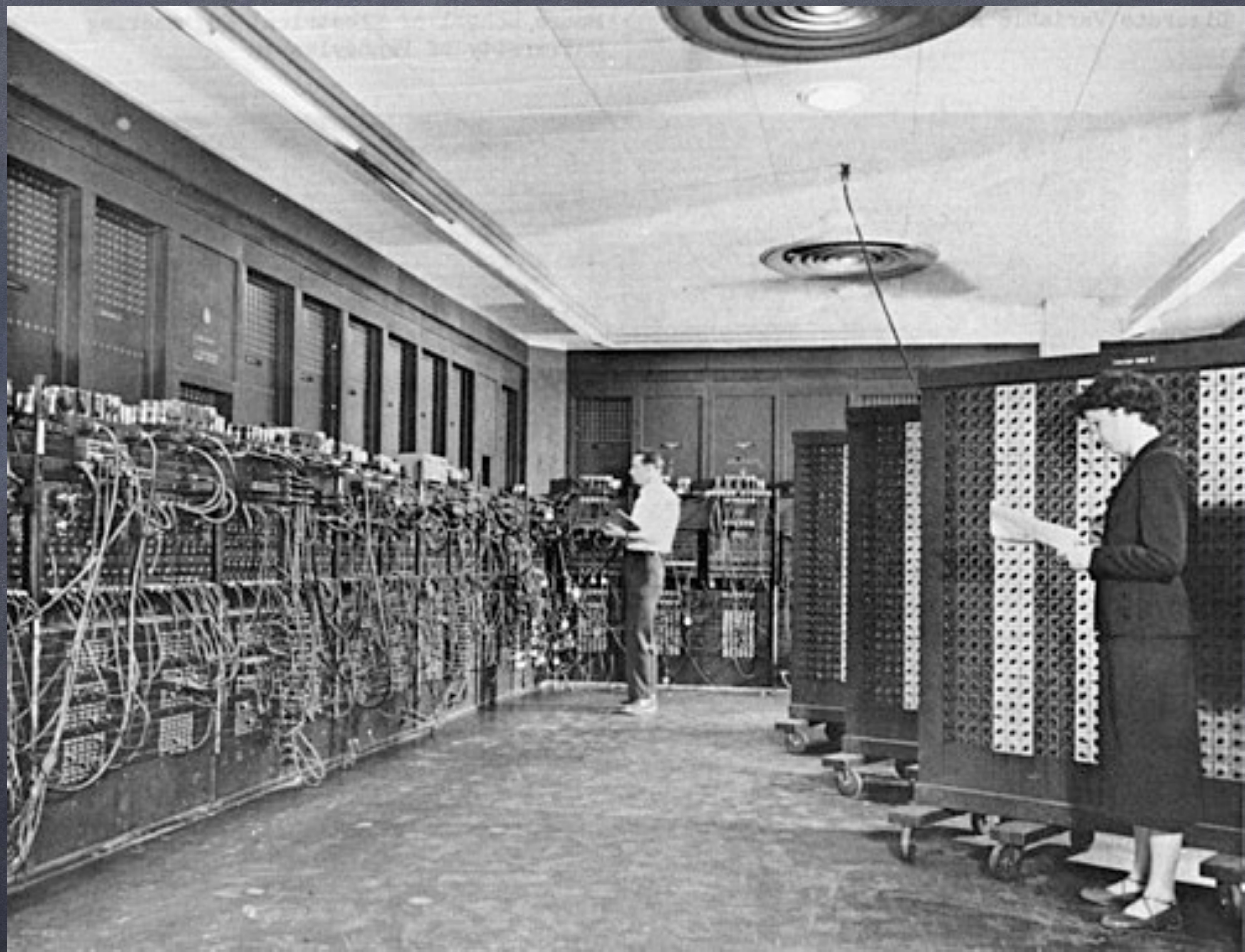
编程语言不是凭空而来，它的发展有迹可循

Figure by Brian Hayes
Brian Hayes, “The Semicolon Wars.” American Scientist, July-August 2006, pp.299-303



1940s

一开始，什么都没有，
只有硬件、线路、继电器



1940s

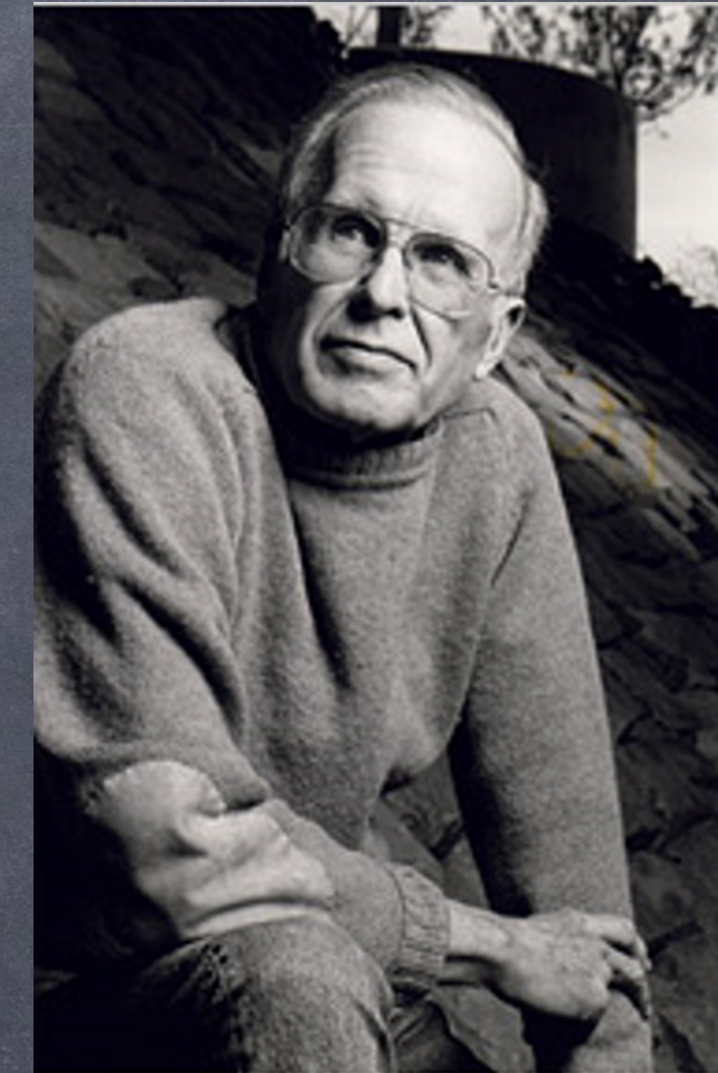
Konrad Zuse

Plankalkül: proposal for first high-level programming language; but never implemented

1950s

FORTRAN (FORmula TRANslation):

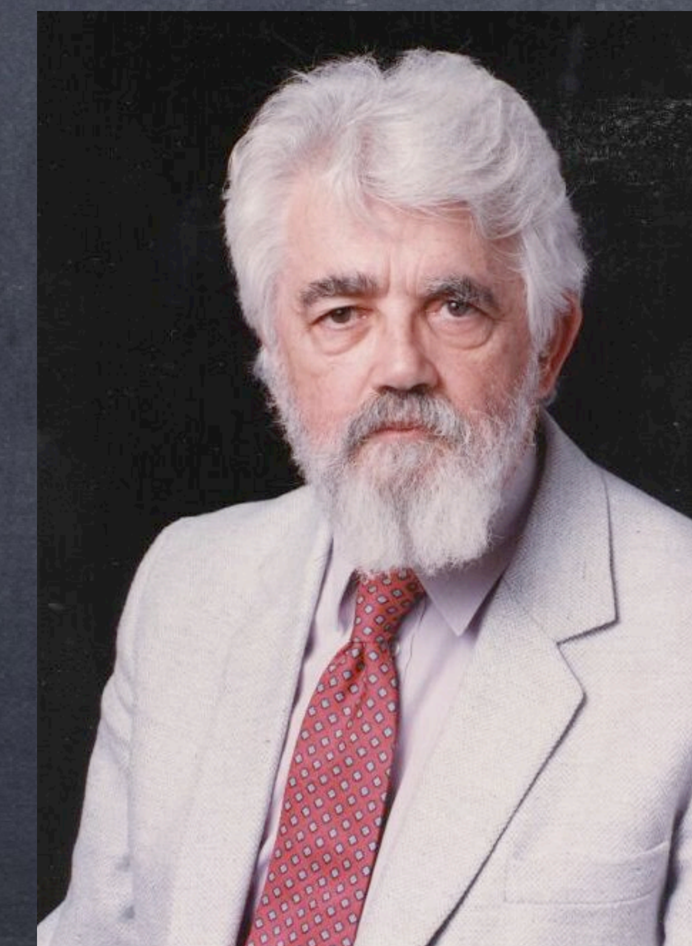
first high-level language with an optimizing compiler; introduces assignments, nested expressions, conditionals,



John Backus

LISP (LISt Processor):

list-processing language, oriented around symbols instead of numbers; garbage collection; conditional expressions; self-interpret



John McCarthy

1960s

ALGOL (ALGOrithmic Language) 60:

early high-level language standard; groups statements into nestable blocks; has lexical scope for variables

COBOL (COmmon Business-Oriented Language):

high-level language designed for portability and human readability

Team work

1960s

APL:

array processing language with special graphical notation; concise chains of operators act on arrays

Kenneth E. Iverson

Simula 67:

first object-oriented programming language, intended for simulations; extension of ALGOL 60; includes classes, subclasses, inheritance, coroutines

Ole-Johan Dahl

1970s

Pascal:

Emphasizes structured programming rather than arbitrary GOTO control flow

Niklaus Emil Wirth

C:

Systems programming language with static types and good facilities for raw memory manipulation; implementation language of Unix, as well as much systems software up to the present day

Dennis Ritchie

1970s

Prolog:

one of the first logic programming languages; programs made up of relations rather than statements or expressions

Smalltalk:

Early completed object-oriented programming language; built around message passing and late binding; all entities in the language are objects, no passive data

Team work

1980s

C++:

adds low-cost abstractions to C for generic programming and object-oriented programming

Bjarne Stroustrup

Perl:

general-purpose scripting language with powerful string manipulation tools

Larry Wall

1990s

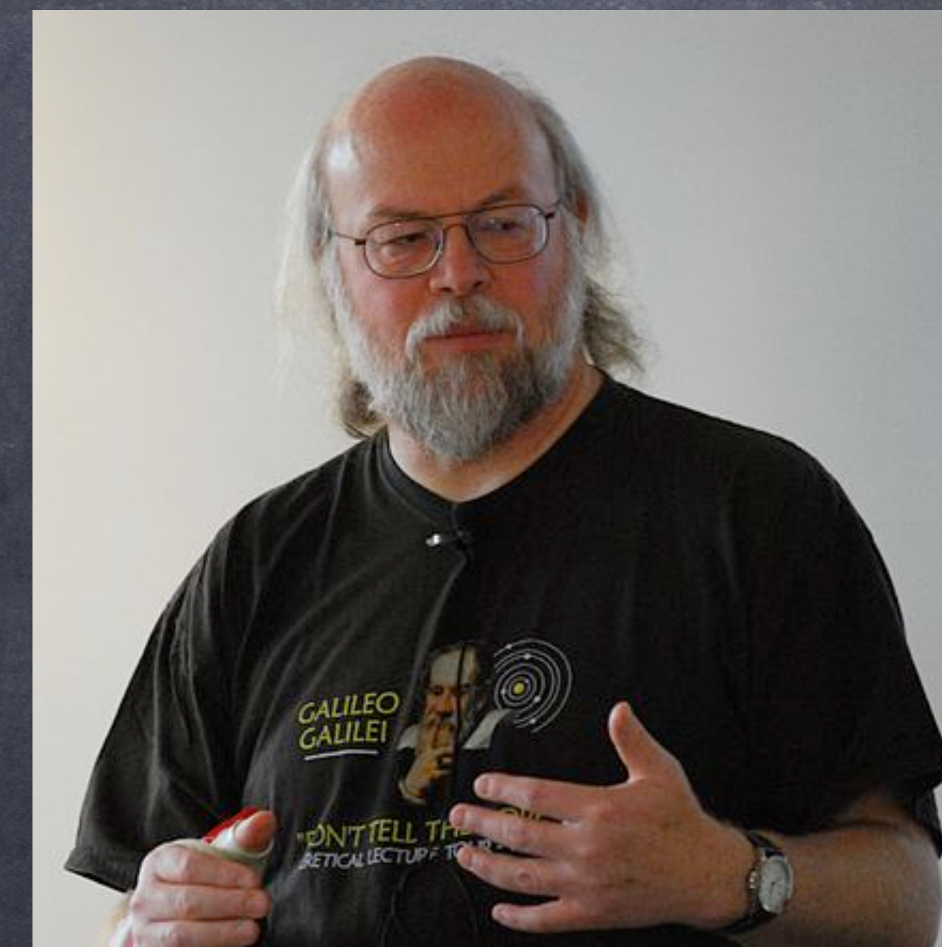
Python:

Scripting language in similar niche to Perl; clean design, whitespace syntax, and list comprehensions

Guido van Rossum

Java:

object-oriented programming language with automatic memory management; combination of ideas from C++ and Smalltalk; originally meant for the web, but grows to dominate enterprise software



James Gosling

编程语言时间线

- 1990s : Ruby, Javascript, VB, PHP, Haskel....
- 2000s - : C#, Scala, Go, Rust, Swift, ...

语言使用趋势

Any questions ?

我们的编程语言能做的事情的上限是什么？
什么都能做到吗？