ADVANCED PROGRAMMING LANGUAGES

Fall 2024, CM 0632. MSc in Computer Science and Information Technology, Ca' Foscari University of Venice

Introduction

The course offers an in-depth analysis of the **advanced concepts and techniques of modern programming languages** together with an introduction to the foundational tools from **programming language theory** and **type theoretic frameworks** available to reason about and asses their design.

We will look at the principles underpinning the functional, object-oriented and concurrent paradigms, with specific **focus on functional languages** features and functional programming style, and analyze their practice and implementation in mainstream programming languages.

Most of the development will focus on **Scala**, but we will draw also from other statically typed languages such as ML, Haskell, Java as well as, occasionally, on dynamically typed languages like python.

We start with an overview of the history and evolution of programming languages.

Programming Language evolution

The **earliest programming languages** were developed with one simple goal in mind: to provide a vehicle through which one could control the behavior of computers. Not surprisingly, the early languages **reflected the structure of the underlying machines** fairly well.

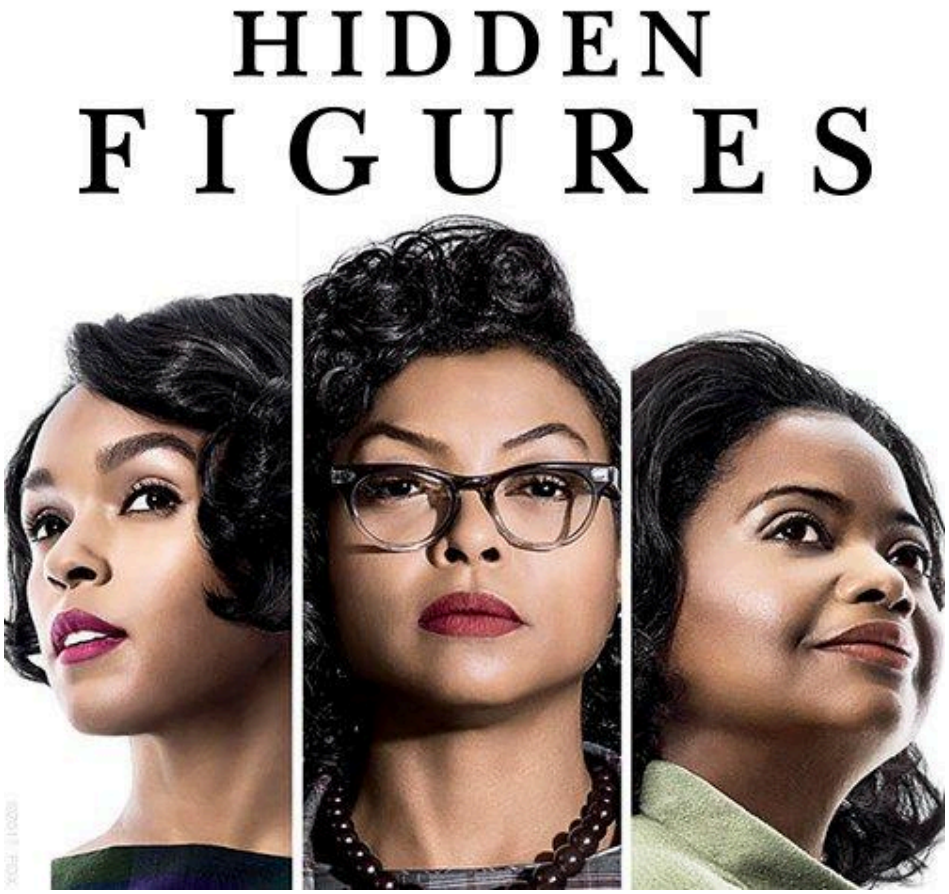
Although at first blush that goal seems eminently reasonable, the **viewpoint quickly changed** for two very good reasons.

- first it became obvious that what was **easy for a machine** to compute was **not necessarily easy for a human being** to reason about.
- second, as the number of different kinds of machines increased, the **need** arose **for a common language** with which to program all of them.

Thus from primitive assembly languages (which were at least a step up from raw machine code) there grew a plethora of high-level programming languages, beginning with **FORTRAN in the 1950s.**

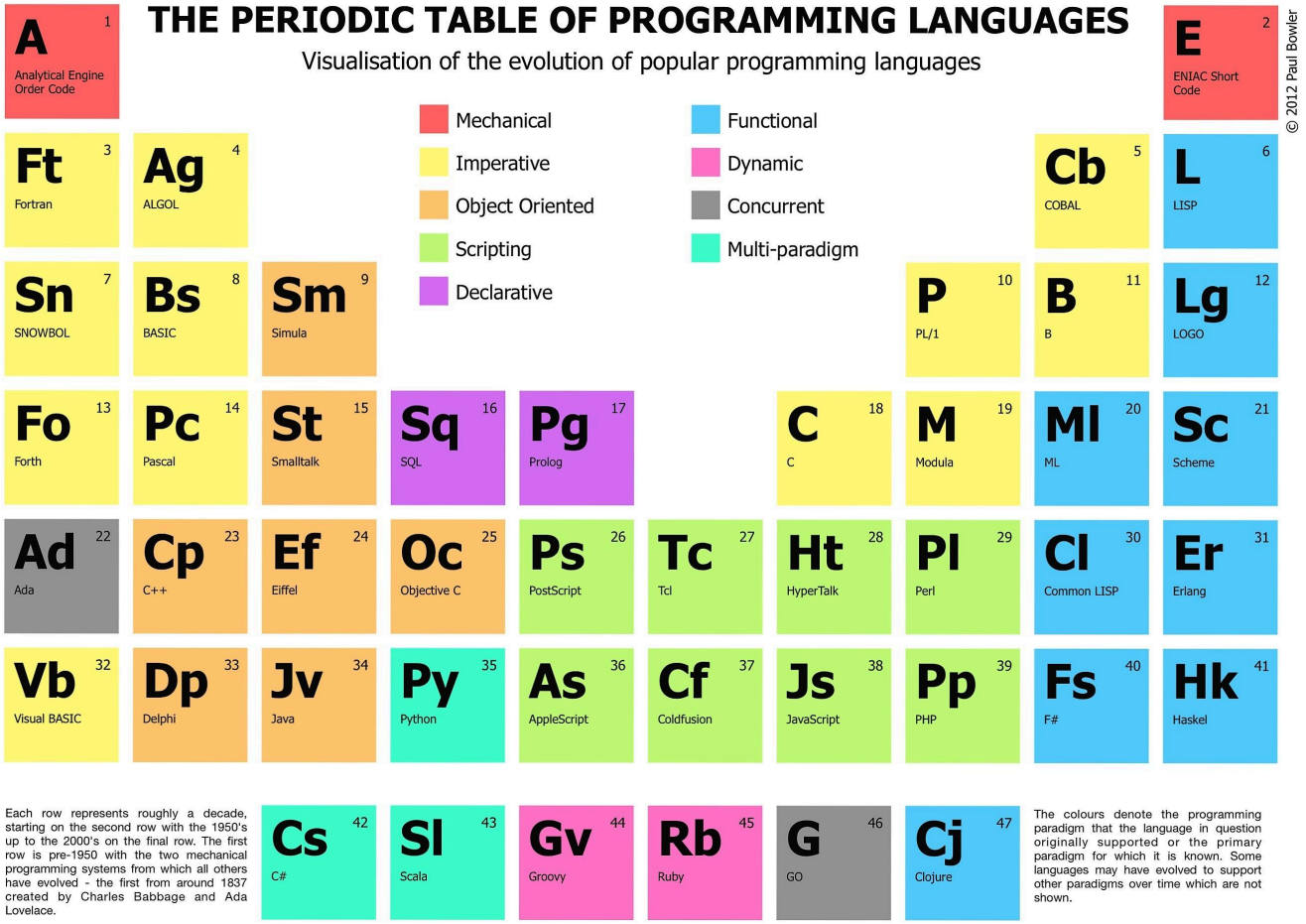
Question: Who knows from which movie is this picture taken?

Answer:



The new programming languages that followed Fortran and its success are best classified according to ***paradigms***. These emerged around **1970s-1980s** with the advent of languages like C, Simula, Smalltalk, ML and Prolog which marked the rise of the various approaches (the paradigms) to programming: imperative, declarative (functional, logical), object oriented. The development of such paradigms was driven by the desire to **find appropriate abstractions** to express **algorithms**, encode and manipulate **data structures** in a principled way, and provide effective (formal) **tools to reason about computations** and program behavior.

The following picture (from https://www.reddit.com/) gives a sense of the extent of the diversity of paradigms and different languages created over time.



We will return to paradigms shortly. The point we want to make here is that the quest for adequate abstractions has almost always been driven by the need to respond to the challenges emerging from the technological evolution.

Technological drivers powering PL evolution

The case of object-oriented programming (OOP) is particularly instructive. In his renown paper , [“The free lunch is over”](#) Herb Sutter recalls that even though OOP was introduced back in the 1960s, object-orientation didn’t become popular and widely used the 1990s.

> *“when the industry was driven by requirements to write larger and larger systems and OOP’s strengths in abstraction and dependency management made it a necessity for achieving large-scale software development that is economical, reliable, and repeatable”.*

In fact, it was with the development of Graphical User Interfaces (GUIs) that the object-oriented model, long studied in academia, started to spread and becore pervasive in applications and mainstream programming.

Similar evolutionary leaps can be observed in the more recent history of programming languages.

Catalysts for change

We can identify a number of *catalysts* that powered significant changes in mainstream programming.

Internet and the web

First of all, the advent of the Internet (and its appeal to the market) shifted the programming language goals **from efficiency to portability and security**. This is the scenario where **Java** came into the arena, with the **JVM** that brought to the forefront the concept of virtual machine bytecode, already used, e.g., in ML and Smalltalk.

But the growth of the web had an impact also on the popularity of other languages: the so-called **scripting languages**, such as PHP, JavaScript, Python, Ruby, which proved well suited to the development of short programs to be embedded into web pages and web servers. A further reason for the widespread adoption of such scripting languages is their high-level, declarative programming model which enhances productivity and **fast prototyping** and provides a much concise and agile tool with respect to the heavy verbosity and to the picky, sometimes unfriendly discipline imposed by strongly typed languages such as Java or C#.

Multi-core hardware

Another landmark in the history of programming languages is **concurrent programming**, whose catalyst was the emergence of multi-core hardware calling for more effective use of the available computing power. Moore's law, according to which CPU performance doubles approximately every two years, is still valid only because performance gains can nowadays be achieved in fundamentally different ways: by means of **CPU hyperthreading** (i.e. many threads in a single CPU) and **multicore** (many CPUs on one chip). However, in order to benefit from such a new hardware, applications must be parallel (or concurrent). This has given renewed popularity to functional languages such as Haskell, and more recently F# and Scala, which are amenable to parallel execution.

> **Side remark:** parallel and concurrent computing are different concepts, even though they are often used as synonyms. They both refer to computations where more than one computation thread can make progress at the same time. However:

> - parallel computing stresses on the fact that many computations are actually carried out simultaneously by means of parallel hardware, such as multi-core computers or clusters:
> - concurrent computation generally refers to tasks that may be executed in parallel either effectively, on a parallel hardware, or virtually, by interleaving the execution steps of each thread on a sequential hardware.

Distributed and cloud computing

Besides multicores and GPUs, which boosted the popularity of concurrent programming, another important achievement of modern technology is cloud computing, which is acting as a catalyst for distributed programming.

Distributed systems have long been an established and well studied computation model, together with a number of successful **programming abstractions** like **sockets, RPCs (Remote Procedure Call), RMI (Remote Method Invocation)**.

However, it is **with cloud applications** that we have seen the real **shift from client-server architectures to applications deployed on heterogeneous platforms**, from mobile devices to cloud-based clusters, running thousands of multicore processors.

Such technological advance requires ad hoc solutions to tackle scalability, hardware heterogeneity, fault tolerance, security and privacy effectively, but at the same time demands new programming models with adequate abstractions. The **Golang** language introduced by Google in 2007 is an example of a new, technology driven programming language design to respond to such needs.

Big data

Current technological trends are driving new challenges associated with AI and big data. In particular, big data applications require high-performance and data-parallel processing at a very high order of magnitude, which in turn need programming abstractions specifically designed for scale-out computations on supercomputers with massively parallel hardware.

A response to this new challenge is the **X10** programming language, an open-source language developed at IBM Research, whose design recombines earlier programming abstraction into a new mix in order to fit the requirements of High Performance Computing. X10 is designed around the **place abstraction**, which represents a virtual computational node that can be mapped onto a computer node in the cluster, or onto a processor or a core in a supercomputer: a program may then run on a collection of places, it can create global data structures spanning multiple places, and it can spawn tasks at remote places and detect their termination.

On a different level, the map-reduce model implemented by **Google's MapReduce, Apache Hadoop** and **Apache Spark** provides **specific abstractions** to support the logic of big data applications **without having to explicitly refer to distribution and parallelism**, thus filling the gap between the logic of the application and the execution infrastructure is particularly large. Interestingly, this model is inspired by the map and reduce combinators distinctive of functional programming.

CREDITS

The material on technological drivers of PL evolution is based on [this paper (https://arxiv.org/pdf/1507.07719.pdf) by Silvia Crafa on an *Evolutionary view of programming languages abstractions*.