Introduction and Definitions

Most programming languages describe computation in an imperative style.

Some definitions from Wikipedia:

- Imperative Programming
  - Imperative programs define sequences of commands for the computer to perform
  - A programming paradigm that describes computation in terms of statements that change a program state
- Declarative programming
  - Declarative programs express what the program should accomplish without prescribing how to do it in terms of sequences of actions to be taken.
  - A programming paradigm that expresses the logic of a computation without describing its control flow

Introduction and Definitions (continued)

- From Gabbrielli and Martini, Chapter 13:

  > *From the intuitive viewpoint, the slogan of declarative programming, so to speak, is that the activity of programming should concentrate on what is to be done, leaving the language interpreter to concentrate on how to reach the desired result. In imperative programming, on the other hand, the programmer must specify both the what and the how.*

- This is idealized and simplistic, but is a good summary.

More about Paradigms

- From Gabbrielli and Martini, Chapter 13:

  > *. . . there are many useful paradigms. Each paradigm has its place: each has problems for which it gives the best solution (simplest, easiest to reason about, or most efficient).*

- Another word for "problems" is *applications*, for example:
  - banking
  - communications
  - games
  - scientific computing
  - ⋮

Paradigms (continued)

- The best solution:
  - the simplest
  - mathematical reasoning is possible and easy
  - the most efficient

- "Easiest to reason about": This is desired when security is important, for example.
- Trade-off with "efficiency": For example, more efficient often means harder to reason about, and therefore less secure.

Languages, Paradigms, et Concepts

Peter Van Roy, Programming Paradigms for Dummies: What Every Programmer Should Know, in "New Computational Paradigms for Computer Music"

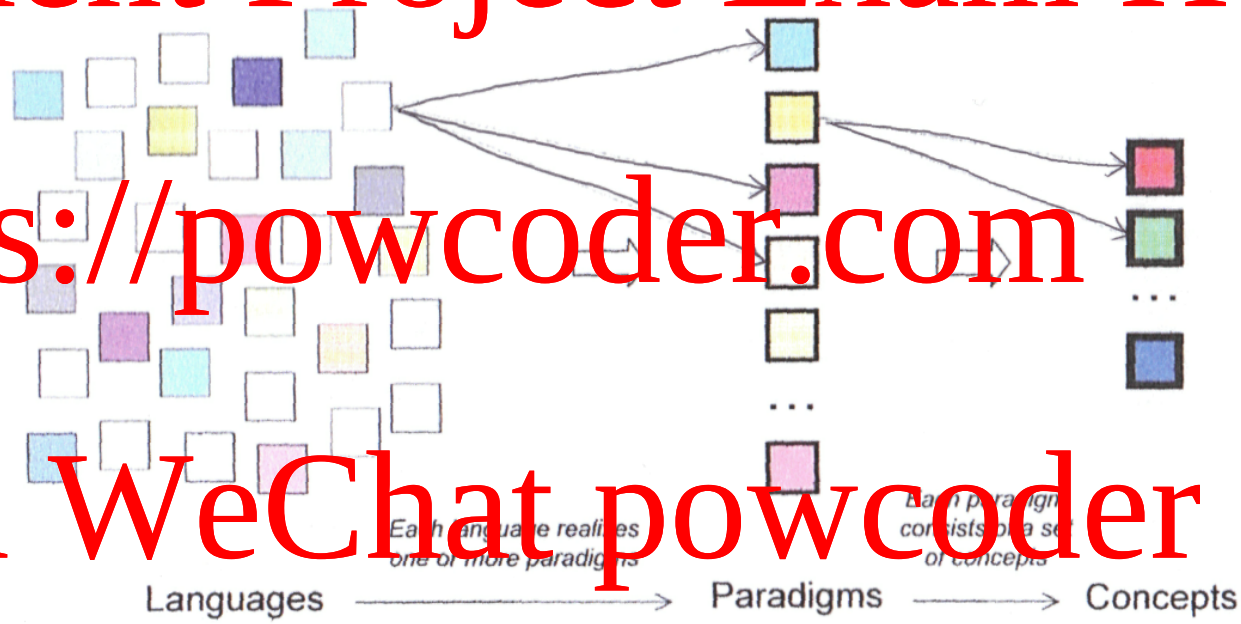

Figure 1. Languages, paradigms, and concepts

A Short Historical Perspective

- Chapter 13 of Gabbrielli and Martini (required reading)
- This summary will set the context for when various paradigms emerged and why.
- Available hardware influenced the kinds of programming languages; we start with the first computers.

The First Computers

- ASCC/MARK I and ENIAC (1946)
  - Able to execute sequences of arithmetic operations in a controlled fashion using a real program
  - Program not stored
  - Program expressed using very rough formalisms (different inputs represented by connecting different physical parts of the computer)
  - Often not even considered computers

Computers and First-Generation Languages

- One definition states that a computer must have the following properties:
  - It is electronic and digital.
  - It is able to perform the four elementary arithmetic operations.
  - It is programmable.
  - It allows the storage of programs and data. (This is not a property of the above "computers".)
- The first computers to meet this definition were EDSAC (1949) and EDVAC (1951)
  - To program, one used a low-level machine language which described, using binary code, the operations and calculation mechanisms of the machine itself.
  - This is machine language, composed of elementary instructions (for example, instructions for adding, loading a value into a register, and so on) that could be immediately executed by the processor, also called first-generation languages (1GL)

Second-Generation Languages

- It was soon realized that to exploit the full use of the power of the computer, it was necessary to develop adequate formalisms that were far from the machine "languages" and closer to the user's natural language.
- A first step in this direction was the introduction of assembly languages, which are symbolic representations of the machine language that can be translated to machine language easily, by assemblers.
- Every computer model had its own assembly language.
- Assembly languages are also called second-generation languages (2GL)

Third-Generation Languages

- A true jump in quality was achieved in the 1950s with the introduction of high-level languages, also called third-generation languages (3GL).
- The first language was FORTRAN (FORmula TRANslation) (1957)
  - introduced symbolic notation to indicate arithmetic expressions, for example: a * 2 + b
  - symbolic notation translated automatically into executable instructions
- Now there are many hundreds of languages, each encompassing a set of concepts. A language encapsulates many concepts; a concept is implemented in many languages.
- These concepts are the subject of this course, studied through a variety of languages, with a focus on OCaml, which allows us to study many of them in depth.

Factors in the Development of Languages

Situation in the 1950s:

- The hardware was certainly the most expensive and important resource.
- The first high-level languages were therefore designed with the aim of obtaining efficient programs, which would use the potential of the hardware to the maximum.
- As a result, many constructs inspired directly by the structure of the physical machine.
- The fact then that programming was very difficult and required very long times was considered a problem of secondary importance, which could be solved by means of large amounts of human resources, which were certainly less expensive than the hardware.

Today the situation is the direct opposite.

- Hardware cheap and efficient.
- Most costs are in SW development.
- Considerations of correctness and security that were not there 50 years ago, are very important.
- Modern languages are therefore designed taking into account first the improvement of various software project activities.
- Efficient use of the physical machine is secondary, except in some particular cases.

Other Factors Besides Hardware and Software

- Applications
  - Initially applications were solely numeric, but now require processing of non-numeric information. For example:
    - artificial intelligence and knowledge processing languages
    - languages for writing computer games
    - ⋮
- New Methodologies
  - For example, programming-in-the-large using object-oriented design practices

Other Factors Besides Hardware and Software

- Implementation of language constructs and experience using them influences new languages.
- Theory plays a role in identifying new technical tools to improve the programming activity. For example:
  - modeling
  - types
  - ...

1950s and 60s

Hardware: mainframes

- used batch processing: take a "batch" of data as input and produce a batch of data as output
- very little interaction with the user

Languages:

- FORTRAN was the first high-level imperative language.
  - A program consists of a main routine and a series of subprograms that can be separately compiled.
  - Not possible to define nested environments
  - No dynamic memory management
  - First version close to assembly, `goto` was central
  - Later versions introduced `if then else`
  - Parameter passing is call by reference or call by value-result.
  - Limited types: only numeric (integer, etc.), boolean, array, string, and file

1950s and 60s (continued)

Languages (continued):

- Algol (ALGOrithmic Language), a family of imperative languages
  - universal (not only numeric) computations
  - great increase in machine-independence of the language
  - notation closer to mathematical notation
  - introduced many new features found in many of today's languages, for example:
    - ★ call by name parameter passing (important for passing functions as arguments)
    - ★ blocks (nested environments)
    - ★ syntax based on a context-free grammar or Backus Naur Form (BNF)
    - ★ recursion
    - ★ dynamic memory management
    - ★ type systems with the ability to permit new user-defined types
    - ★ many structured commands in the form we use now (`if then else`, `for`, `while`)

1950s and 60s (continued)

- **Languages** (continued):
  - LISP (LISt Processor), declarative, functional, Scheme is a dialect
    - for non-numeric, specifically symbolic expressions, which are basically lists, used in artificial intelligence for example
    - new features:
      - ★ higher-order features, e.g., functions as arguments and results of computation
      - ★ dynamic memory management using a heap and a garbage collector
      - ★ dynamic scope (Scheme is a statically-scoped variant)

1950s and 60s (continued)

Languages (continued):

- COBOL (COmmon Business Oriented Language), imperative
  - specific to commercial applications with syntax as close as possible to the English language
  - introduced rudimentary mechanisms for features such as abstract data types and modules (much improved in modern languages)
- Simula, precursor to object-oriented (ahead of its time)
  - extension of Algol
  - adds pointers, coroutines (early version of threads), classes, objects
  - designed for discrete-event simulation applications

The 1970s

Hardware: minicomputer

- more interaction with user, which requires constraints on response time

Languages:

- C (imperative)
  - designed as a systems programming language for Unix, became general purpose
  - more features which allow access to low-level functionality of the machine
  - more features for interaction
  - programs can be translated to efficient machine code
  - explicit pointers equivalent to arrays, very powerful, but prone to errors
  - efficiency more important than reliability

The 1970s (continued)

Languages:

- Pascal, imperative
  - descendent of Algol
  - introduced intermediate code (like Java bytecode used today), thus easy to port to a different machine
  - more limited pointers than C, which avoid some of the pitfalls
- Smalltalk, object-oriented
  - new way to provide encapsulation and information hiding with new concepts of class and object

The 1970s (continued)

Languages:

- ML (Meta Language), declarative, functional
  - introduced imperative constructs in a functional setting by assignment to "reference cells" (modifiable variables)
  - type system is the most important contribution, no runtime type errors, supports type inference, supports parametric polymorphism
- Prolog, declarative, logic programming
  - studied in previous course

The 1980s

Hardware:

- PCs
  - need for graphical interfaces, re-use important, ideal for object-oriented
- Embedded Systems (e.g., microwave ovens, aircraft)
  - need for reliability, correctness, response time

Languages:

- C++, object-oriented
  - added classes and (multiple) inheritance to C without compromising efficiency and compatibility with C
  - no garbage collection, improved type system, parametric polymorphism

The 1980s (continued)

Languages (continued):

- Ada, imperative
  - improved Pascal with new constructs for real-time and embedded systems, concurrent execution of tasks
- CLP (Constraint Logic Programming), declarative, logic programming
  - Prolog with constraint solvers

The 1990s

Hardware:

- internet and WWW, and eventually browsers
  - browsers require portability and security

Languages:

- Java, object-oriented
  - based on C++
  - to be used in small computing devices with limited power, connected to a network
  - JVM designed to meet portability requirements
  - security partially addressed by types
  - avoidance of pointers and inclusion of garbage collection to improve reliability and simplicity
  - synchronization and communication primitives for threads contribute to portability

The 1990s (continued)

Languages:

- Java (continued)
  - security and reliability features have a cost in terms of efficiency, but not particularly important for typical application domains
  - For example, "it is certain that the time spent waiting by a browser or use of the network makes the execution times of Java applets negligible."

Recent Languages (Multi-paradigm)

Python:

- imperative
- object-oriented
- functional
- aspect-oriented
- Often used as a scripting language for web applications.
- A scripting language is a programming language that allows control of one or more software applications. Scripts are distinct from the core code of the application, as they are usually written in a different language and are often created or at least modified by the end-user.
- Scripts are often interpreted from source code or bytecode, whereas application software is typically first compiled to a native machine code or to an intermediate code.
- Source: Wikipedia

Recent Languages (Multi-paradigm)

OCaml:

- functional
- imperative
- object-oriented
- scripting language: `https://blog.janestreet.com/ocaml-as-a-scripting-language/`
- and more ...