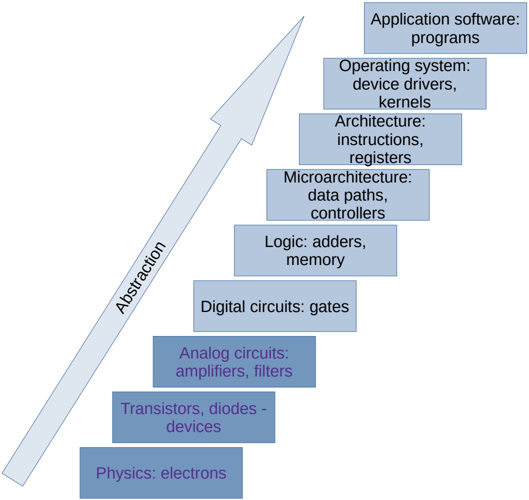
E2011: Theoretical fundamentals of computer science
Introduction to programming languages

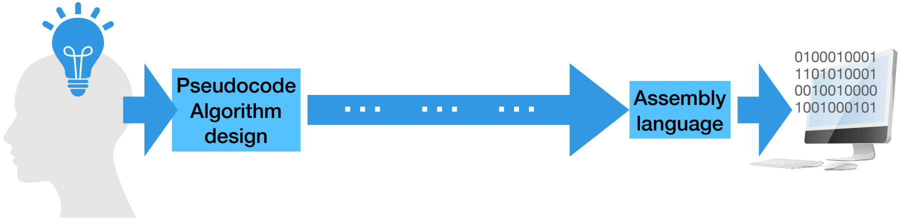
Vlad Popovici, Ph.D.

Fac. of Science - RECETOX

Outline

1. Programming languages
   - Imperative languages
   - Functional languages
   - Logic predicate languages
   - Object-oriented languages
2. Executing programs

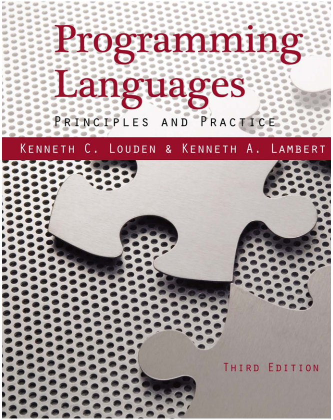

Application software: programs
Operating system: device drivers, kernels
Architecture: instructions, registers
Microarchitecture: data paths, controllers
Logic: adders, memory
Digital circuits: gates
Analog circuits: amplifiers, filters
Transistors, diodes - devices
Physics: electrons
Abstraction


Pseudocode
Algorithm
design
Assembly
language
0100010001
1101010001
0010010000
1001000101

Programming languages

"How we communicate influences how we think and vice versa."

"Similarly, how we program computers influences how we think about computation, and vice versa."



Programming languages

- it is a formal computation specification means
- it has a strict **syntax** specified by a **grammar**
- clear **semantics** for each syntactic construct
- various practical implementations:
  - real vs virtual machine
  - translation vs. compilation vs. interpretation

Levels of abstraction

- algebraic notation and floating point numebers: FORTRAN
- structured abstractions and machine independence Algol, Pascal
- architecture independence ($\lambda-$calculus, Lisp)

Levels of data abstraction

- basic: variables, data types, declarations
- structured abstractions: data structures, arrays
- unit abstractions: abstract data types (ADTs), classes, packages, namespaces
- information hiding, modularity, reusability, interoperability

Main programming paradigms

- *imperative*/procedural (e.g., C, C++): variables, assignment, other operators
- *functional* (e.g., Lisp, Scheme, ML, Haskell): abstract notion of a function, based on $\lambda-$calculus
- *logic* (e.g., Prolog): symbolic logic (e.g., predicate calculus)
- *object–oriented* (e.g., Java, Python, C++): encapsulation of data and control together
- *generic* (e.g., C++ and especially its standard library - STL): type abstraction and enforcement mechanisms
- several paradigms may be implemented in the same language
- (multi-paradigm languages)

Imperative languages

- *variables and assignments*
- *sequential execution*
- *conditionals* (if-then-else) and *loops* (for-do, while-do, and do-while) are the building blocks
- *subroutines/functions/procedures* for procedural abstraction
- efficiency and low-level control

Imperative languages - sum of odd elements in C

```
#include <stdio.h>

int main() {
  int vector[] = {1, 2, 3, 4, 5, 6, 7, 8, 9}; // some data
  int size = sizeof(vector) / sizeof(vector[0]);
  int sum = 0;

  for (int i = 0; i < size; i++) {
    if (vector[i] & 1) { // use bit-wise AND
      sum += vector[i]; // Add odd element to the sum
    }
}

printf("Sum of odd elements in the vector: %d\n", sum);

return 0;
}
```

Imperative languages - example in Fortran 95

```
program sum_odd_elements_vectorized
implicit none
integer, parameter :: n = 9
integer :: vector(n) = [1, 2, 3, 4, 5, 6, 7, 8, 9]
integer :: sum

sum = sum(vector(mod(vector, 2) /= 0))

print *, 'Sum of odd elements in the vector:', sum
end program sum_odd_elements_vectorized
```

Functional programming

$\lambda-$calculus

- Alonzo Church, 1930s
- formal basis of all functional languages
- *infix notation*
- syntax:

$$\begin{aligned} expression \rightarrow\ & constant \\ & |variable \\ & |(expression expression) \\ & |(\lambda variable.expression) \end{aligned}$$

- example

$$(\lambda x. + 1\ x)\ 2 \Rightarrow (+1\ 2) \Rightarrow 3$$

Examples

Examples of FL: Haskell, JavaScript, Scala, Erlang, Lisp, ML, Clojure, OCaml, Lisp, etc. etc.

Lisp

```
(defun sum-odd-elements (lst)
  (if (null lst)
    0
    (if (oddp (car lst))
      (+ (car lst) (sum-odd-elements (cdr lst)))
      (sum-odd-elements (cdr lst)))))

(let ((vector '(1 2 3 4 5 6 7 8 9)))
  (format t "Result: ~a" (sum-odd-elements vector)))
```

OCaml

```
let rec sum_odd_elements lst =
  match lst with
    | [] -> 0
    | hd :: tl ->
      if hd mod 2 <> 0 then
        hd + sum_odd_elements tl
      else
        sum_odd_elements tl

let () =
  let vector = [1; 2; 3; 4; 5; 6; 7; 8; 9] in
  let result = sum_odd_elements vector in
  Printf.printf "Sum of odd elements in the list: %d\n" result
```

Logic programming

- based on *logical statements*
- uses *first order predicate calculus*; ingredients
  - variables: e.g. x, y, z
  - constants: e.g. $1, 2, a, b$
  - predicates: e.g. $P(x), Q(y)$, properties or relationships that can be true or false for objects or values
  - quantifiers: *universal quantifier* $\forall$, e.g.$\forall\ x\ P(x)$, and *existential quantifier* $\exists$, e.g. $\exists\ x\ P(x)$
  - connectives: AND ($\wedge$), OR ($\vee$), NOT ($\neg$), and IMPLIES ($\rightarrow$)
- example: $\forall\ x\ (P(x) \rightarrow Q(x))$ means "for all values of x, if P(x) is true, then Q(x) is also true"

Prolog

Example 1: family relations

```
male(harry).
female(liz).

parent(phil, chas).
parent(liz, chas).
parent(chas,harry).
parent(chas,wills).

grandmother(GM, C):-
        mother(GM, P),
        parent(P, C).

mother(M,C):-
        female(M),
        parent(M, C).

% Run: grandmother(liz, Who).
% Result: Who=harry and Who=wills
```

Example 2 (recursion): sum of elements from a list

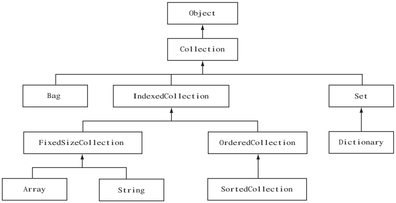
```
sumlist([],0).
ssumlist([H|T],N) :-
        sumlist(T,N1),
        N is N1+H.
```

Example 3: sum of odd elements

```
sum_odd_elements([], 0).

sum_odd_elements([Head|Tail], Sum) :-
  0 is Head mod 2, % If Head is even,
  sum_odd_elements(Tail, Sum). % skip Head

sum_odd_elements([Head|Tail], Sum) :-
  1 is Head mod 2, % If Head is odd,
  sum_odd_elements(Tail, TailSum), % sum of odd elem. in Tail,
  Sum is Head + TailSum. % and add Head to the sum.

% Example usage:
% sum_odd_elements([1, 2, 3, 4, 5, 6, 7, 8, 9], Result).
% Result will contain the sum of odd elements: 25
```

Object-Oriented Programming

- satistifes three important needs:
  - reusability
  - minimal changes for modifying behaviour
  - independence of components
- extension of data and/or operations
- redefinition of operations
- abstraction
- polymorphism



Java

```
public class SumOddElements {
  public static void main(String[] args) {
    int[] vector = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int sum = 0;

    for (int element : vector) {
      if (element % 2 != 0) { // Check if the element is odd
        sum += element; // Add odd element to the sum
      }
    }

    System.out.println("Result: " + sum);
  }
}
```

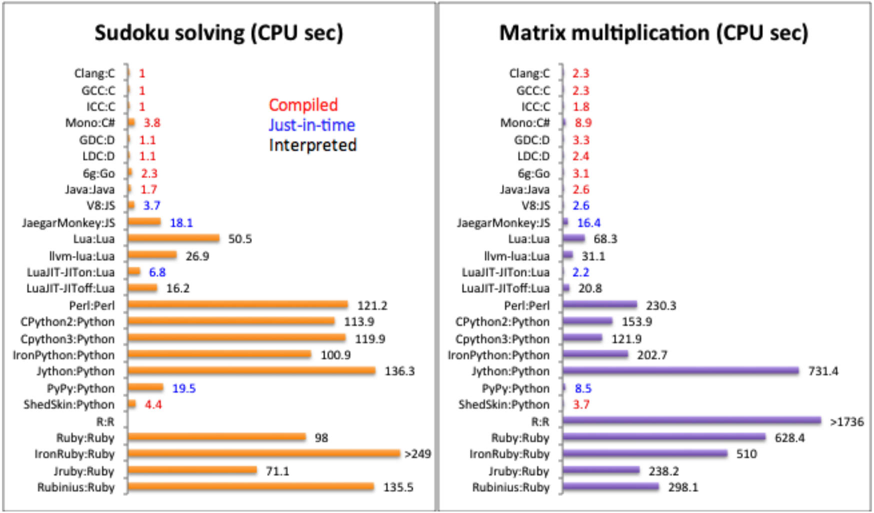
Python - while OOP, does not impose it

```
vector = [1, 2, 3, 4, 5, 6, 7, 8, 9]
sum = 0

for element in vector:
  if element % 2 != 0:
    sum += element

print("Sum of odd elements in the list:", sum)
```

Programs: from code to execution

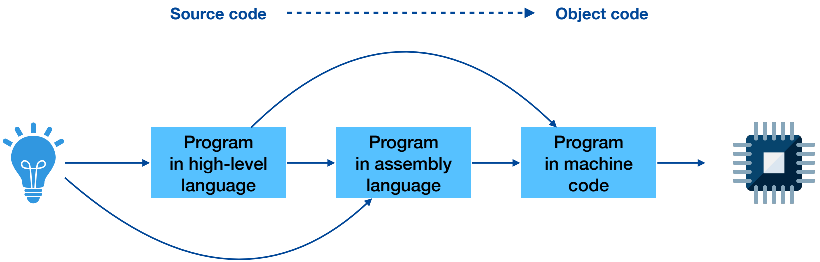
- the *source code* is not directly runnable on the CPU
- means to run a program
  - *compilation* - ahead-of-time compilation for *compiled languages* (e.g. C/C++/Fortran...)
  - *interpretation* - a special software interprets and executes the instructions in a program - for *interpreted languages* (e.g. Lisp, Prolog, Python)
  - *just-in-time compilation* for programs running on virtual machines (e.g. Java, Python (CPython))
- compilation allows code optimization
- sometimes several techniques are used for a single language: e.g. Java is compiled to bytecode, runs on VM and has a JIT

AOT vs JIT vs interpretation



from https://attractivechaos.github.io/plb/

AOT compilation

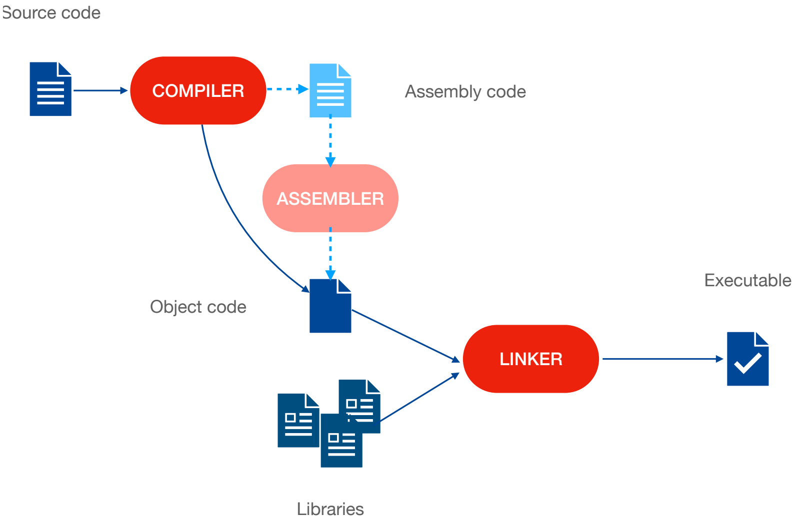



```
# let rec sort = function
   | [] -> []
   | x :: l -> insert x (sort l)
 and insert elem = function
   | [] -> [elem]
   | x :: l -> if elem < x then elem :: x :: l
           else x :: insert elem l;;
```

```
mov ax,'00'
mov di,counter
mov cx,digits+cntDigits/2
cld
rep  stosw
inc  ax
mov [num1 + digits - 1],al
mov [num2 + digits - 1],al
mov [counter + cntDigits - 1],al
```

```
0100010001
1101010001
0010010000
1001000101
```

- Advantages:
  - efficiency
  - privacy
  - offline execution: no need for the original source code
- Disadvantages:
  - low portability
  - build complexity and time


Source code
COMPILER
Assembly code
ASSEMBLER
Object code
Executable
LINKER
Libraries

Interpretation

- there is an *interpreter* software that reads and executes the program instruction-by-instruction
- supports "write-once-execute-everywhere" idea
- Advantages:
  - portability
  - dynamic features
  - ease of debugging
- Disadvantages:
  - low performance
  - dependency on interpreter
- examples: Python, R

Virtual machines and JIT

- VM: an abstract machine (program) that runs the *bytecode* represenation of the program
- the program is compiled to bytecode either by AOT compiler or by a *just-in-time (JIT)* compiler
- JIT may operate at various time points and different granularity to optimize the (byte)code
- Advantages: portability and good performance
- Disadvantages: overhead and dependency on VM
- example: Java Virtual Machine (JVM) contains bothseveral an interpreter for bytcode, a compiler from code to bytecode and from an optimizing JIT
- there are also JITs for other interpreted languages (e.g. Python)

If you're curious about other programming languages, check out
`http://helloworldcollection.de`

Questions?