Introduction to Scientific Computing Languages

Prof. **Paolo Bientinesi**

pauldj@aices.rwth-aachen.de

Languages for Scientific Computing

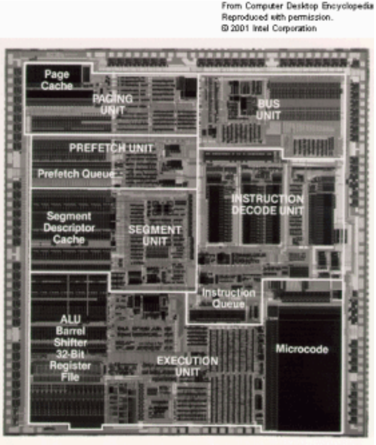
What is a programming language?

- A set of **instructions and constructs** for communicating with a computing device.
- Instructions and constructs are combined and organized into programs.
- Examples: Basic, Pascal, Cobol, Fortran, C, C++, Lisp, Prolog, SQL, Java, Perl, Python, Ruby, . . .

“Computing device”?

> sequential processors, embedded processors, . . . , parallel computers, supercomputers.

Processor's Components



Arithmetic Logic Unit (ALU) (control signals, inputs, outputs), Floating Point Unit (FPU), Prefetching Unit, Registers, . . .

- Languages let the users specify how to use these components.
- Only **Assembly** operates on components: Low-level language.
- High-level languages only specify the computations to be performed.
- A **compiler** and/or an **interpreter** translates high-level programs into a sequence of component actions.

Assembly

Low-level language

- Very fast!
- Not the lowest level. Not directly executable.
- **Assembler** translates assembly into machine code. Executable.
- Assembly consists of mnemonic codes.
  Machine code: only numbers.
- Translation Assembly $\leftrightarrow$ machine code is almost 1-1.
  This is not true for high-level languages.
- Assembler is hardware-specific. Control over chips' components.

Assembly

Example

```
        .text
        .globl poly
poly:
        li.s   $f0, 0.0          # y = 0, running & return result
        mtc1   $6  $f12          # x, move to float register
Loop:
        mul.s $f14, $f12, $f0    # compute (x * y)
        mul    $2, $5, 4         # $5 = i, compute address of a[i]
        addu   $3, $2, $4        # a + (i*4)
        l.s    $f16, 0($3)       # a[i], load coefficient
        add.s $f0, $f16, $f14    # y = a[i] + (x*y)
        addi   $5, $5, -1        # decrease i

        slt    $2, $5, $0        # $2 = 1 if i < 0
        beq    $2, $0, Loop      # goto Loop if i >= 0
Exit:
        j      $31

# Evaluate the value of a polynomial using Horner's algorithm.
# f = a[0] + a[1] * x + a[2] * x^2 + ... + a[n] * x^n
```

History

What is the oldest programming language still in use?

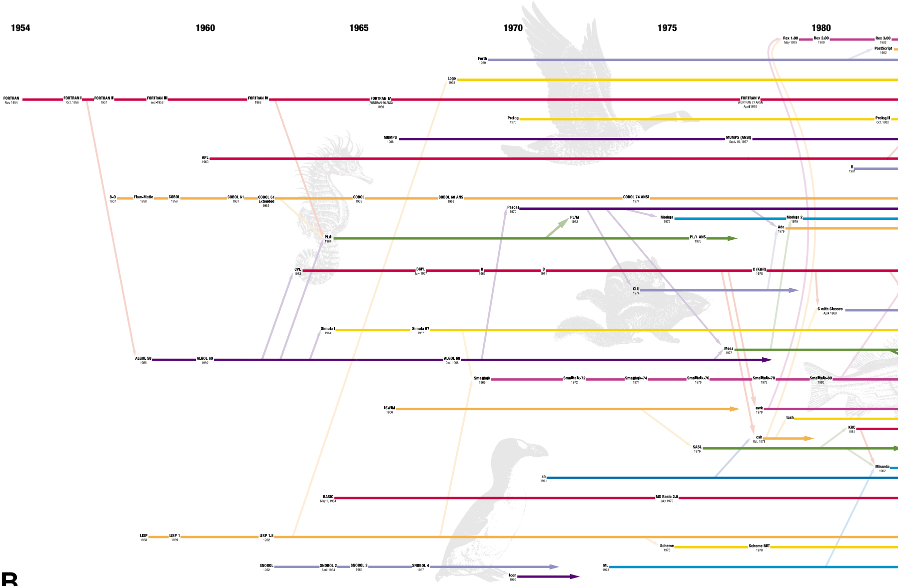
FORTRAN 1957, 1977, 1995, . . .

late ’50s	**Fortran**(’57)..., Algol(’58), Lisp(’59)
’60s	Cobol(’61), Basic(’64)
’70s	Pascal(’70), **C**(’72), Prolog(’72), SQL(’78), **Matlab**(’78)
’80s	C++(’83), Perl(’87), **Mathematica**(’87)
’90s	Python(’91), Ruby(’93), Java(’95)

Oldest programming language?

Plankalkül (1940s). For the Z1 computer, by Konrad Zuse.

History of Programming Languages



WEB

Compiled vs. Interpreted Languages

Compiled Languages

- The program is first **compiled**, i.e., reduced to architecture-dependent instructions and stored in an executable file.
- The program can then be **executed** separately, at a later time.
- The executable is portable only to compatible platforms. The program?
- **Speed!**
- Examples: C, Fortran.

Interpreted Languages

- The instructions are parsed and executed in real time by an **interpreter**.
- No generated code. The interpreter is always needed.
- **Ease!**
- Examples: Matlab, Mathematica, Python.

Computer Programs

Program:
sequence of instructions expressing the operations to be performed on a target computing platform.

- Each program $\mathcal{P}$ has a meaning. It implements a function.
  $\{\text{Initial State}\} \quad \mathcal{P} \quad \{\text{Final State}\}$.
- $[[\mathcal{P}]]$ is the *semantics* of the program $\mathcal{P}$.
  $[[\ ]]$ = Semantics operator. Operational, Denotational, Axiomatic.
  Out of the scope of this class.
- Generally, we want $\mathcal{P}$ to compute $f(i)$, with $i \in I$.
  f is a mathematical function, a procedure, a simulation, ...
- The question is: "does $\mathcal{P}$ implement the function that we have in mind?"
- A program $\mathcal{P}$ is **correct** if $\forall i \in I, \mathcal{P}(i) \equiv f(i)$.
- Surprisingly... when working with floating point numbers, correctness is not enough!

Subroutines

subroutine = function = procedure = subprogram (= module = method)

- Portions of the code that perform one specific task and that are reusable.
- They are very much like mathematical functions:

```
result := routine_name( arguments )
```

- BUT! One difference: side-effects.
  Many languages allow subroutines to have side-effects.
  The routine alters the state of the system even after its completion.

Side Effects

$\{\ (\texttt{res} = \ldots) \wedge \text{State}\ \}$

```
res := routine_name( args );
```

$\{\ (\texttt{res} = \ldots) \wedge \text{State}'\ \}$

- If (State = State′) $\rightarrow$ no side-effects.
- Most languages allow constructs with side-effects.
- Print statements; iterative constructs; . . .

More on Subroutines

Subroutines are good!

- Improve **readability**: code is shorter.
- Enable **modularity**: programs are built as a composition of functionalities. Avoid reinventing the wheel.
- **Optimization**: They solve a smaller and well-defined task. Better suited to be optimized.
- Structure:

```
routine_name( args )
    //
    body
    //
return( value )
```

`args`, `body` and `value` are optional, depending on the language.

Questions about Subroutines

```
routine_name_1(args_1)      routine_name_2(args_2)
    //                          //
    body_1                      body_2
    //                          //
return( value_1 )           return( value_2 )
```

- Can `body_1` include a call to `routine_name_1`?
  - Yes! → Recursion. Recursion Limit? Termination?
  - No → Iteration. Fortran '77.
- What if `body_1` includes a call to `routine_name_2` and `body_2` includes a call to `routine_name_1`?
  - Mutual recursion. Fortran '77: No.
- Are recursive languages more expressive than iterative ones? Can they compute more or fewer functions?

  Recursion ≡ iteration!

From Subroutines to Libraries

- A subroutine solves a specific problem / it computes a specific operation.
- It is reusable, i.e., it provides a certain functionality.
- A collection of germane subroutines yields a **library**.
- A library is not a program per se.
  It provides building blocks to be used when writing a program.
- BLAS LAPACK PETSc MPI Pthreads ...
  LINPACK
  EISPACK
- Libraries can be written in one or more languages.
  Can they be accessed from a program written in a different language?

Imperative vs. Functional Languages

Imperative Languages

- Concept of **Variables** and **State**.
- Program is an ordered sequence of commands and **assignments**.
- Commands modify state. Side-effects.
- C, C++, Fortran, Java, Python, Matlab, . . .

Functional Languages

- No variables or assignments.
- Program consists of **Functions** and **Recursion**.
- No side-effects!
- Subset of Declarative Languages.
- Lisp, APL, ADA, Haskell, Mathematica, Clojure, F# . . .

Scope

```
Program A;
Var I:Integer;
    K:Char;
    R:Real;

    Procedure B;
    Var K:Real;
        L:Integer;

        Procedure C;
        Var M:Real;
        Begin
        // Body #1
        End;
    Begin
    // Body #2
    End;
Begin
// Body #3
End;
```

- Which variables (of which type) are defined in `Body #1`?

  `I:Integer`, `R:Real`, `K:Real`, `L:Integer`, `M:Real`

- Where is `K` used as `Real`?

  `Body #1` and `Body #2`

- Can `L` be referenced in `Body #2`? `Body #3`?

  `Body #2`: yes; `Body #3`: no

Scope (2)

```
program main
var y: Real;
    procedure compute()
    var x : Integer;
        procedure initialize()
        var y: Integer;
        var z: Real;
        begin {initialize}
        // Body #1
        end {initialize}

        procedure transform()
        var x: Real;
        begin {transform}
        // Body #2
        end {transform}
    begin {compute}
    // Body #3
    end {compute}
begin {main}
// Main body
end {main}
```

- What is the scope of the variable `x` declared in the procedure `compute`?

  `Body #1` and `Body #3`

- What is the environment for the procedure `transform`?

  `y:Real` and `x:Real`