Which programming language will you learn today?

Finding your path in the jungle of programming languages

Cyril Soldani

Geeks Anonymes

Jan 28th 2016
Warning

Although the speaker is informed, this talk will be:

- overgeneralizing;
- oversimplifying;
- likely flawed in one way or another;
- and of course, **biased**.

Please, don't shoot the speaker!

Also note the slides are of little use without the talk.
There are surprisingly many programming languages

There have been programming languages since WW2.

Bill Kinnersley's (KU) language list collects information on about 2500 programming languages.

Wikipedia lists 678 notable programming languages, excluding dialects of BASIC and esoteric programming languages.

The TIOBE index of hot programming languages currently monitors 244 languages.

We won't discuss them all.
Why so many programming languages?

- ≠ languages for ≠ machines and devices
  Various assembly languages, CUDA/OpenCL, ...
- New ones (try to) fix (perceived) problems of old ones
- Evolution of software engineering, new paradigms
- Evolution of machines: ↑ CPU, ↑ RAM, ↑ BW
- Emergence of the PC, then the cloud
- NIH syndrome
- Timing, marketing, hype
- **Combinatorial** design space w.r.t. underlying paradigms and features
- ≠ fields of application
Many programming languages are (were) specialized

- Web: JS, PHP (, HTML, CSS)
- DB: SQL, LINQ
- Scripting: shells, Lua
- Graphics: MetaPOST, Postscript, Logo
- Text processing: TeX, PERL, Icon
- Business: COBOL
- Security-critical: Ada, ATS, Eiffel
- Math/science/HPC: Matlab, R, Mathematica, Fortress
- Optimization: Oz, AMPL
- Logic: Prolog
- Verification: Coq
- GPU: CUDA, OpenCL
- Embedded: various assembly languages
Alternative plan

Why care?

Look at some programming language paradigms.

Look at some programming language features.

A few practical considerations.

A tour of trendy programming languages.

Discussion.
Learning new programming languages is good

Gives **more tools on the shelf**.

The language shapes the **thought process**.

- Attested by linguistics research.
- We tend to follow main language paradigms.
- $\neq$ problems best solved by $\neq$ approaches.

Gives access to more tools and libraries.

Allows fixing/improving more tools and applications.

We generally don't miss features before knowing them.
“There is no silver bullet”

There exists no such thing as an ultimate programming language.

Even in multi-paradigm languages.

Available syntax options are limited.

Language knowledge is limited (≠ C++ coders may use ≠ subsets of C++).

Antagonist features (many examples to follow).
What is Turing-completeness?

Characterize calculability.

Equivalent to Church's lambda calculus and μ-recursive functions.

Does not matter much, all practical programming languages are Turing-complete.

And many impractical ones too!

But markup languages generally aren't.
Wikipedia lists even more paradigms.
Most important paradigms

Some languages historically tried to be one-style only.

Nowadays, most languages supports multiple paradigms.

Some languages are not categorized easily. Most often scripting, **glue** languages. Or **specialized** ones.

Functional programming especially tends to spread.
Procedural paradigm

Imperative: mutates global state.

Procedure-oriented, *verb*-oriented.

Top-down: going from `main()` to sub-functions to sub-sub-functions, etc.

Little encapsulation, data manipulated explicitly.

Generally simple, explicit, low-level languages.

Mainly for programming in the small.

Still hugely popular.

Examples: C, Fortran.
Object-Oriented (OO)

Imperative: mutates objects state.

Data-oriented.

Bottom-up.

Encapsulation: data and behaviour defined together, direct data access generally forbidden.

Often more complex, high-level languages.

For programming in the large.

Examples: C++, Java.
Functional programming

Declarative: functions in the mathematical sense. No mutable variables.

Uses ADT to represent data. Often allows pattern matching.

Uses recursion and function composition for control-flow.

Based on the concept of **closures**, and **high-order** functions.

Dual of OO: closures are poor-man objects, objects are poor-man closures.

Somewhat niche since the 60's, becomes really popular lately (for concurrency).
Functional programming example

-- Simplifying an expression

-- Abstract Data Type
data Expr
    = Num Int
    | Add Expr Expr
    | Mul Expr Expr
    | ...

simplify (Mul x y) = -- Returns a new expression
    case (x', y') of -- Pattern matching
        (Num 0, _ )  -> Num 0 -- _ means "whatever"
        (_ , Num 0)  -> Num 0
        (Num 1, y'' ) -> y''
        (x'', Num 1) -> x''
        _           -> Mul x' y'
    where x' = simplify x -- Recursion
            y' = simplify y
...

-- map :: (a -> b) -> [a] -> [b] higher-order function
simplifyListOfExprs = map simplify
Logical programming

Can cover several different paradigms:

- Logic programming: mostly boolean logic, Prolog.
- Constraint programming: numerical constraints, Oz, AMPL.
- Theorem provers: Coq, Agda, Isabelle.

Very declarative. Little explicit control flow.

More specialized than other paradigms (when used alone).
/* 1) There are 5 colored houses in a row, each having an owner, which has an animal, a favorite cigarette, a favorite drink.  
    2) The English lives in the red house. 
    3) The Spanish has a dog. 
    4) They drink coffee in the green house.  
    ...  
    15) The Norwegian lives near the blue house. 
Who has a zebra and who drinks water? */

start(Sol):- length(Sol,5),  
    member([english,___,red],Sol),  % 1
    member([spanish,dog,___],Sol),  % 2
    member([___,coffee,green],Sol),  % 3
    member([ukrainian,___,tea,___],Sol),  % 4
    right([___,___,green],[___,___,white],Sol),  % 5
    member([___,snake,winston,___],Sol),  % 6
    member([___,kool,___,yellow],Sol),  % 7
    Sol= [___,___,milk,___,___],  % 8
    Sol= [[norwegian,___,___],___,___],  % 9
    next([___,chesterfield,___],[___,fox,___,___],Sol),  % 10
    next([___,___,___],[___,___,blue],Sol),  % 11
    member([___,___,water,___],Sol),  % 12
    member([___,zebra,___,___],Sol).  % 13
Programming language features

- High-level, low-level or both.
- One paradigm (Prolog) vs multi-paradigm (Scala, Oz).
- General-purpose (C++) vs specialized (TeX, R).
- Safety. Boils down to strong typing vs weak/no typing.
- Sequential vs concurrent (Go, Erlang).
- Cross-platform vs target-specific (.NET, Swift).
- Good Foreign Function Interface (FFI)?
- Experimental vs stable.
- Small-scale rapid development vs large-scale, modular.
- Good for prototyping vs industrial strength.
- Garbage-collected vs manual memory management.
- Meta vs non-meta.
- Error-handling mechanisms (exceptions?).
- Available tools (compilers, IDEs, REPL, **libs**), support.
Compiled, interpreted or bytecode?

Compiled languages (C++, Fortran) transform source code into native machine code, which then runs very efficiently.

Dynamic languages (Python, PERL) use an interpreter, which reads and executes source code at runtime, much slower.

There are hybrids, where part of the program can be compiled at runtime: JIT, HotSpot.

Some languages emits bytecode (Java, Python, .NET), which is then executed by a fast interpreter.

Some languages compiles to native code, but for an abstract machine (Haskell). The program is like data for the abstract machine.
Eager/strict vs lazy

```c
int incFirst(int x, int y __attribute__((unused))) {
    // Both x and y are simple, already computed values
    return x + 1;
}

int main(void) {
    return incFirst(-1, printf("Printed before call to f()\n"));
}
```

```
incFirst x _ = x + 1
main = do
    let zero = incFirst (-1) (error "Instant Crash")
    putStrLn "Will be printed"
    head 1 (mapM heavyComputation [1..]) -- Compute only once
```

Lazyness allows defining your own control flow.

Lazyness allows using infinite data structures.

It may improve or decrease performance.
Static vs dynamic typing

```c
int add(int x, int y) { return x + y; } // Safe, efficient
```

```python
def add(x, y): return x + y
```

```c
Object *add(Object *x, Object *y) { // Will never be inlined
    switch (x->type) {
        case INTEGER:
            if (y->type = INTEGER)
                return newInteger(x->integerVal + y->integerVal);
            else {
                Object *yInt = convert(y, INTEGER); // May fail
                return newInteger(x->integerVal + yInt->integerVal);
            }
        case STRING:
            if (y->type = STRING)
                return strintCat(x, y);
            else
                ...
        ...
        default:
            // Raise runtime error
    }
}
```
Explicit typing vs type inference

It is not because typing is not explicit that it is dynamic. There are languages whose compilers *guess* the types at compile time.

It can make the code more (but also less) readable.

```cpp
for (std::vector<std::vector<double>>::size_type i = 0;
     i < v.size(); ++i) {
    update(i, v[i]);
    for (std::vector::const_iterator it = v[i].begin();
         it != v[i].end(); ++it)
        someFun(it);
}

for (auto i = 0u; i < v.size(); ++i) { // Cheating here
    update(i, v[i]);
    for (const auto &v : v[i])
        someFun(v);
}
```
Polymorphism

When one piece of code can act on several ≠ but related types.

Templates in C++, generics in Java.

Built-in in most functional languages (Haskell, OCaml).

Duck typing in dynamic languages (Python).

\[
\text{map :: (a -> b) -> [a] -> [b]}
\]

\[
\text{template<class RandIt> // May give terrible error messages}
\text{void sort(RandIt first, RandIt last);}\]

\[
\text{def f(x, y): return x + y # Could do anything, really}
\]
Multiple inheritance and similar mechanisms

C++-like multiple inheritance: allows implementation reuse, but beware of diamonds!

Java interfaces: safe, but no implementation reuse.

Aggregation (Go): rather than inherit, have members for parents. Delegate parent methods explicitly.

Haskell type classes: allows reuse of behaviour, but data is not inherited, so no diamonds. May add boilerplate.
What matters the most?

Though shaping.

Features that could not be implemented easily in a library (GC, laziness, concurrency).

Fitness to task.

Safety, law enforcement, ease of reasoning about code.

Boilerplate reduction.

Readability, maintainability.

Tools, libraries, support.
Performance does not matter ... except when it does!

A 2-5 slowdown compared to C is very often OK.

Slowdown can be as high as $1000\times$ however, with some high-level interpreted languages.

Find better algorithms!

Use libraries! Python with BLAS/LAPACK faster than your own-cooked matMul in assembly.

Bare numerical performance still matters for custom code.

With a good FFI, you can use a slow high-level language and optimize only performance-critical code in another language.
When in Rome, do like the romans

Learn/use language(s) of your domain:

- Easier to understand what others do
- Easier to find doc
- Easier to reuse material
- Easier to make yourself understood and cited

Except if you obtain an edge using another (let the dinos die).

Learning new languages still useful even if you keep your main one.
# Hot languages

<table>
<thead>
<tr>
<th>Java</th>
<th>Python ↑</th>
<th>PHP ↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>C/C++</td>
<td>JavaScript ↑</td>
</tr>
<tr>
<td>Objective-C</td>
<td>Swift ↑↑</td>
<td>R</td>
</tr>
<tr>
<td>Matlab ↓</td>
<td>Ruby ↓</td>
<td>PERL</td>
</tr>
<tr>
<td>Scala</td>
<td>Lua</td>
<td>Haskell</td>
</tr>
</tbody>
</table>
Most loved languages

According to StackOverflow 2015 survey.

1. Swift
2. C++11
3. Rust
4. Go
5. Closure
6. Scala
7. F#
8. Haskell
9. C#
10. Python
Which language for what?

If you only ever want a single one: Python or C++ or Java (but have you listened this talk?)

Data science: R, Haskell, Scala

Web: JavaScript (+ Darts?), PHP, Go + markup languages of course.

Apple fan: Swift

Going for a change: Haskell (or OCaml), Rust
Resources

- The Computer Language Benchmarks Game
- Rosetta Code
- Programming Paradigms for Dummies, Peter Van Roy (UCL)
- StackOverflow Developer Survey
- Lifehacker.com: Which programming language to learn
- TIOBE Index
- RedMonk Programming Language Rankings
- PYPL Popularity of Programming Language
- GitHub Archive