Programming Languages: Concepts

(Lectures on High-performance Computing for Economists IV)

Jesús Fernández-Villaverde,\(^1\) Pablo Guerrón,\(^2\) and David Zarruk Valencia\(^3\)

September 7, 2020

\(^1\)University of Pennsylvania

\(^2\)Boston College

\(^3\)ITAM
Introduction
Motivation

- Since the invention of Fortran in 1954-1957 to substitute assembly language, hundreds of programming languages have appeared.

- Some more successful than others, some more useful than others.

- Moreover, languages evolve over time (different version of Fortran).

- Different languages are oriented toward certain goals and have different approaches.
Some references


- Essentials of Programming Languages (3rd Edition), by Daniel P. Friedman and Mitchell Wand.


- http://hyperpolyglot.org/
The basic questions

- Which programming language to learn?
- Which programming language to use in this project?
- Do I need to learn a new language?
Which programming language? 1

- Likely to be a large investment.

- Also, you will probably want to be familiar at least with a couple of them (good mental flexibility) plus $\LaTeX$.

**Alan Perlis**

A language that doesn’t affect the way you think about programming is not worth knowing.

- There is a good chance you will need to recycle yourself over your career.
Typical problems in economics can be:

1. CPU-intensive.
2. Memory-intensive.

Imply different emphasis.

Because of time constraints, we will not discuss memory-intensive tools such as Hadoop and Spark.
Classification
Classification

- There is no “best” solution.
- But there are some good tips.
- We can classify programming languages according to different criteria.
- We will pick several criteria that are relevant for economists:
  1. Level.
  2. Domain.
  3. Execution.
  4. Type.
  5. Paradigm.
• Levels:

1. machine code.


3. High level: like C/C++, Julia, ...

• You can actually mix different levels (C).

• Portability.

• You are unlikely to see low level programming unless you get into the absolute frontier of performance (for instance, with extremely aggressive parallelization).
Fibonacci number

Machine code:

```
8B542408 83FA0077 06B80000 0000C383 FA027706 B8010000 00C353BB
01000000 B9010000 008D0419 83FA0376 078BD98B C84AEF1 5BC3
```

Assembler:

```
ib:  mov edx, [esp+8]  cmp edx, 0  ja @f  mov eax, 0  ret
@@:  cmp edx, 2  ja @f  mov eax, 1  ret
@@:  push ebx  mov ebx, 1  mov ecx, 1
@@:  lea eax, 3  jbe @f  mov ebx, ecx  [ebx+ecx]  cmp edx, mov ecx, eax  dec edx  jmp @b
@@:  pop ebx  ret
```

C++:

```cpp
int fibonacci(const int x) {
    if (x==0) return(0);
    if (x==1) return(1);
    return (fibonacci(x-1))+fibonacci(x-2);}
```
• Domain:

  1. General-purpose programming languages (GPL), such as Fortran, C/C++, Python, ...

  2. Domain specific language (DSL) such as Julia, R, Matlab, Mathematica, ...

• Advantages/disadvantages:

  1. GPL are more powerful, usually faster to run.

  2. DSL are easier to learn, faster to code, built-in functions and procedures.
Execution 1

- Three basic modes to run code:
  1. Interpreted: Python, R, Matlab, Mathematica.
  2. Compiled: Fortran, C/C++.
  3. JIT (Just-in-Time) compilation: Julia.

- Interpreted languages can be used with:
  1. A command line in a REPL (Read–eval–print loop).

- Many DSL are interpreted, but this is neither necessary nor sufficient.

- Advantages/disadvantages: similar to GPL versus DSL.

- Interpreted and JIT programs are easier to move across platforms.
Execution II

- In reality, things are somewhat messier.

- Some languages are explicitly designed with an interpreter and a compiler (Haskell, Scala, F#).

- Compiled programs can be extended with third-party interpreters (CINT and Cling for C/C++).

- Often, interpreted programs can be compiled with an auxiliary tool (Matlab, Mathematica,...).

- Interpreted programs can also be compiled into byte code (R, languages that run on the JVM - by design or by a third party compiler).

- We can mix interpretation/compilation with libraries.
Types I

- Type strength:
  1. Strong: type enforced.
  2. Weak: type is tried to be adapted.

- Type expression:
  1. Manifest: explicit type.
  2. Inferred: implicit.

- Type checking:
  1. Static: type checking is performed during compile-time.
  2. Dynamic: type checking is performed during run-time.

- Type safety:
  2. Unsafe: no error.
Types II

- Advantages of strong/manifest/static/safe type:
  1. Easier to find programming mistakes ⇒ ADA, for critical real-time applications, is strongly typed.
  2. Easier to read.
  3. Easier to optimize for compilers.
  4. Faster runtime not all values need to carry a dynamic type.

- Disadvantages:
  1. Harder to code.
  2. Harder to learn.
  3. Harder to prototype.
• You implement strong/manifest/static/safe typing in dynamically typed languages.

• You can define variables explicitly. For example, in Julia:

```julia
a::Int = 10
```

• It often improve performance speed and safety.

• You can introduce checks:

```julia
a = "This is a string"
if typeof(a) == String
    println(a)
else
    println("Error")
end
```
<table>
<thead>
<tr>
<th>Sep 2019</th>
<th>Sep 2018</th>
<th>Change</th>
<th>Programming Language</th>
<th>Ratings</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>Java</td>
<td>16.661%</td>
<td>-0.78%</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>C</td>
<td>15.205%</td>
<td>-0.24%</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>Python</td>
<td>9.874%</td>
<td>+2.22%</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td>C++</td>
<td>5.635%</td>
<td>-1.76%</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>↑</td>
<td>C#</td>
<td>3.399%</td>
<td>+0.10%</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>↓</td>
<td>Visual Basic .NET</td>
<td>3.291%</td>
<td>-2.02%</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>↑</td>
<td>JavaScript</td>
<td>2.128%</td>
<td>-0.00%</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>↑</td>
<td>SQL</td>
<td>1.944%</td>
<td>-0.12%</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>↓</td>
<td>PHP</td>
<td>1.863%</td>
<td>-0.91%</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
<td>Objective-C</td>
<td>1.840%</td>
<td>+0.33%</td>
</tr>
<tr>
<td>11</td>
<td>34</td>
<td>↑</td>
<td>Groovy</td>
<td>1.502%</td>
<td>+1.20%</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>↑</td>
<td>Assembly language</td>
<td>1.378%</td>
<td>+0.15%</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>↓</td>
<td>Delphi/Object Pascal</td>
<td>1.335%</td>
<td>+0.04%</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>↑</td>
<td>Go</td>
<td>1.220%</td>
<td>+0.14%</td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>↓</td>
<td>Ruby</td>
<td>1.211%</td>
<td>-0.08%</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>↓</td>
<td>Swift</td>
<td>1.100%</td>
<td>-0.12%</td>
</tr>
<tr>
<td>17</td>
<td>20</td>
<td>↑</td>
<td>Visual Basic</td>
<td>1.084%</td>
<td>+0.40%</td>
</tr>
<tr>
<td>18</td>
<td>13</td>
<td>↓</td>
<td>MATLAB</td>
<td>1.062%</td>
<td>-0.21%</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>↓</td>
<td>R</td>
<td>1.049%</td>
<td>+0.03%</td>
</tr>
<tr>
<td>20</td>
<td>17</td>
<td>↓</td>
<td>Perl</td>
<td>1.049%</td>
<td>-0.02%</td>
</tr>
<tr>
<td>----------------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>Java</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Python</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>C++</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Visual Basic .NET</td>
<td>5</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C#</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>JavaScript</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>PHP</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>SQL</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>89</td>
<td>-</td>
</tr>
<tr>
<td>Objective-C</td>
<td>10</td>
<td>3</td>
<td>29</td>
<td>37</td>
<td>-</td>
</tr>
<tr>
<td>Perl</td>
<td>17</td>
<td>11</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Lisp</td>
<td>32</td>
<td>15</td>
<td>17</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Pascal</td>
<td>222</td>
<td>16</td>
<td>14</td>
<td>88</td>
<td>6</td>
</tr>
</tbody>
</table>
Language popularity

- C family (a subset of the ALGOL family), also known as “curly-brackets languages”:

- Python: position 1, 16.66%.

- Matlab: position 18, 1.06%.

- R: position 19, 1.05%.

- Fortran: position 34, 0.36%.

- Julia: position 36, 0.34%.
High-performance and scientific computing is a small area within the programming community.

Thus, you need to read the previous numbers carefully.

For example:

1. You will most likely never use JavaScript or PHP (at least while wearing with your “economist” hat) or deal with an embedded system.

2. C# and Objective-C are cousins of C focused on industry applications not very relevant for you.

3. Java (usually) pays a speed penalty.

4. Fortran is still used in some circles in high-performance programming, but most programmers will never bump into anyone who uses Fortran.
Multiprogramming

- Attractive approach in many situations.

- Best IDEs can easily link files from different languages.

- Easier examples:
  2. Rcpp.
  3. Mex files in Matlab.