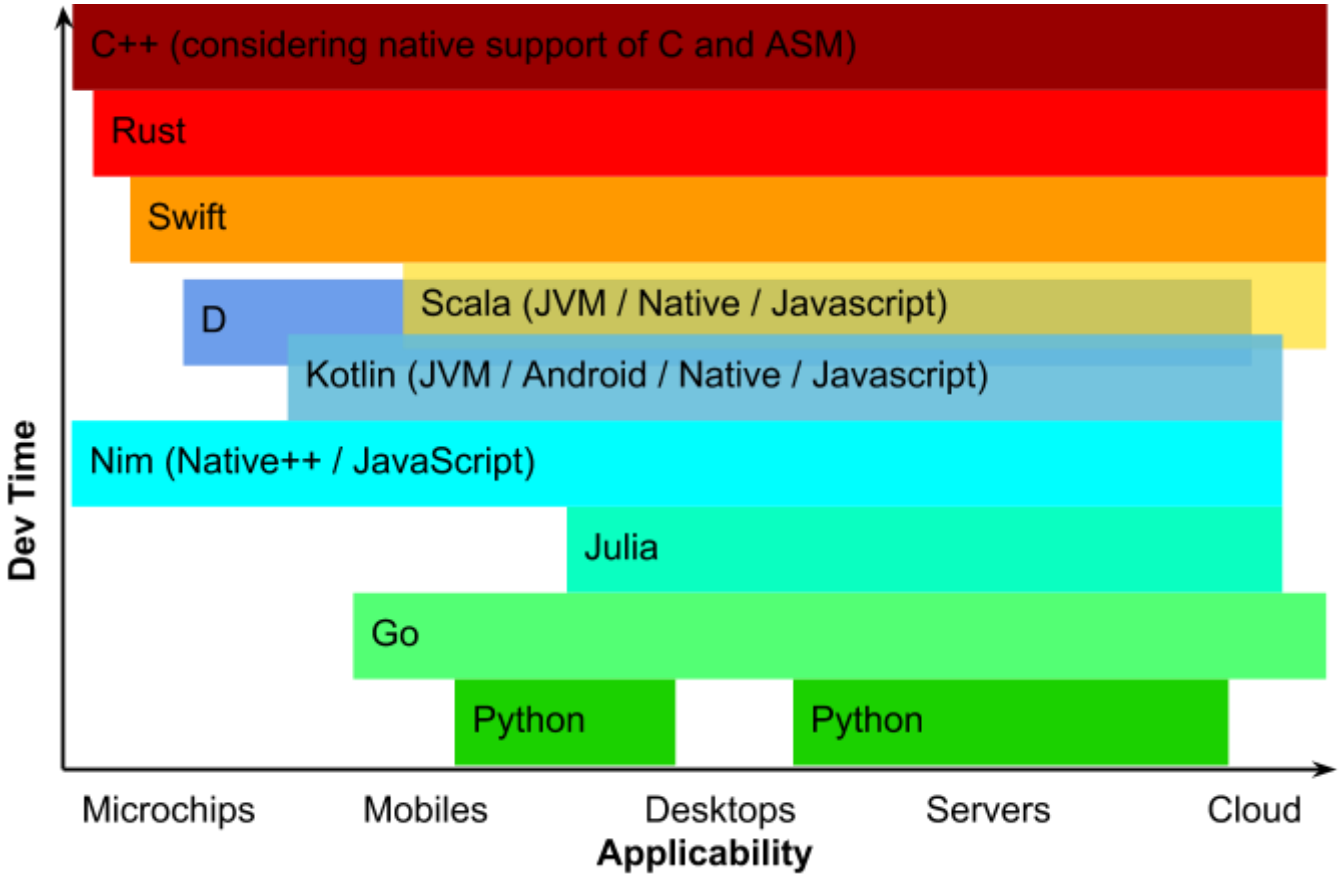
Overview of the Efficient Programming Languages

Subjective evaluation by Artem V L, 2015-6 (updated: 2016-11, 2018-04)

Foreword

Our productivity tremendously depends on the tools we use. One of the fundamental tools for the software developers, researchers and analysts is the programming language. There are no silver bullets, each language fits the best for the specific purpose. Sometime it's critical to prototype fast and we often use Python, but when it comes to the performance then C++ or C are the standard choice. The Open Source movement significantly accelerated evolution and capabilities of the software ecosystems. It was literally impossible to build a full-fledged operation system within one-two years no so long ago. You can argue that Linux did it, but Linux was only a kernel that uses GNU ecosystem being developed many years before. In contrary, Redox OS appeared in the end of 2015 being developed using Rust language ecosystem and become probably the most secure existing OS. This is an excellent example of how the language selection impacts the project destiny (execution speed, reliability, security, development community, etc.).

To take a careful decision about the language selection, we need to evaluate the language features from the project priorities viewpoint. I have listed such most influencing features of the languages in the table for the convenience. The initial post has been updated in the end of 2016 since Rust and Swift evolved significantly. Nim and Kotlin are included in the begin of 2018.



Benchmarks: https://github.com/kostya/benchmarks, https://benchmarksgame.alioth.debian.org/

Comparative Aspects of the Programming Languages

Grades notation: 1 (low) - 5 (high); - (not supported), + (supported)

Aspects \ Langs	**C/C++**	**Nim**	**Rust**	**D**	**Python**	**Julia**	**Go**	**Swift**	**Kotlin (Multi)**	**Scala (JVM/Bin)**
Bin Deployment	+	+	+	+	Indirect (Cython)	+	+	+	+jar/native/Zing	+(jar/native/Azul Zing)
Packaging, Deps Management	Not native*	+	+	+ (dub)	+	+	+	+	Gradle, Ant, ...	Maven, Ant, Gradle, Ivy
Compilation Speed	2 (slow)	2	**1**	3	5	4	5	**1** (type infer.)	4	2 (slow)
Execution Speed	5 (fast)	5	5	4	1-2(pypy/Cython)	4	4	3	3-4	3-4
Startup time	5 (fast)	5	5	5	4	**1**	5	5	3-4	3-4
Memory Consumption	**1** (low)	**1**	2 (> 5 MB)	1(nRT) -2	3	4 (> 50 MB)	2	2	4-5	**5** (huge, > 100 MB)
Garbage **C**ollector	-	-/various	- /ref cnt	opt, STW	transp, STW	transp, STW	transp,~ no STW	ref cnt	transp, STW	transp. Stops World
MemAlloc > RAM	+	+	+	+	-	?	-	+	- (partial)	- (partially on x64)
Multithreading	STL, libs	+	+	+	-GIL;libs/ backends	+	+	libs	+	+
Async IO	libs	+	+	+	+	native	native	libs	+	+
Foreign Linking	3	**5** (C++& JS)	3	4 (links C++)	3	3 (types conv)	4 (native for C)	2 (variadic args)	4 (JNI/native)	4 (JNI/native)
Linking by Foreigners	**5** native + SWIG	**5** native	5 (C lib)	2 (C lib)	3 (RTL)	3 (types conv)	3 (C lib + GC)	4 (C lib)	4(jar/native)	4 (jar/native)
Cross-platform GUI	3 (libs)	**5** native	3	3	5 (QT, WxWidg)	3 (GTK, QT)	4 (ui, QT)	3 (libs)	3 (Java based)	3 (Java based)
Private values (encapsulation)	Class, file, nsp	Module	Module	Class, file nsp	- (convention)	Module, method	Module, method	Module file nsp	Class, pkg	Class, package
Mobile Dev	+ emb	+emb	+emb	?	via ports	?	+	+ emb	+	via jvm/lib
Metaprogramming	4 **	5 native	3	5 (mixin)	4-5 ***	5	1 **** (pure)	2 (Sourcery)	2(GroovyShell, ...)	4(Scalameta;Rhino,GroovyShell)
Concise,expressive, uniform & intuitive	**1**(-concise)	2(-unifm.)	3	3	**5**	4	4	4	4	3

* CMake (see biicode for dependency tracking), Gradle, **0instal** (cross-platform package management with binary distribution support), **CPM** (cross-platform package manager) and others.

** GCC 5+ has libgccjit shared library for embedding in other processes (such as interpreters), suitable for Just-In-Time compilation to machine code, also as Clang LLVM.
*** Python metaclasses and introspection are sufficient to perform any operation with the scripting (a typical use case, CPython) but give less capabilities than similarly ranked compiled languages for the compiled Python (Cython, etc.).
**** Special tools like *gen* and *genny* are used, embedding generics declarations into comments (it is definitely not an elegant approach).

Known Pitfalls, Caveats and Peculiarities

Note: Many issues are related to the immature nature of the languages (compilers) and might be solved in near future.

C++: is a mature language and all possible issues somehow have been solved, but it requires significant experience and is not a fast-study language by newcomers; good for the high performance and highly portable code especially in the resource-constraint environments. Among the C++ drawbacks and strengths are the backward compatibility with C and the highest level of flexibility. The backward compatibility, particularly the C macro definitions and complex headers/classes module system tremendously drops down the compilation speed, but allows to reuse the code written dozens years ago, which provides the largest existing software ecosystem and makes C libs ABI (and to some extend C++) a defacto standard. To retain the backward compatibility, the standard is updated relatively rare (each 3 years since 2011), however all the new features are reviewed extra thoroughly by the top experts in the field and verified in practice by another languages. Amazing flexibility of C/C++ causes hidden bugs, but provides vast optimization capabilities for the experts, which are probably the most qualified experts in IT. Recent standards (C++14 and 17) added lots of lacked features including generic lambda functions, standard cross-platform I/O, standard and extremely flexible cross-platform parallelism, functional programming, much more powerful metaprogramming and much more speeding up the development process! SWIG (v.4 from the master) supports most of the C++14 features and can be used to generate NATIVE interfaces for Pyhon, Ruby, C#, Java and many other languages.

Nim: was introduced in 2008 as a unique compiled language, which has a custom backend (compilable to C/C++/ObjectiveC/JavaScript), truly optional and customizable Garbage Collector (gc:refc|v2|markAndSweep|boehm|go|none|regions), extremely powerful metaprogramming and native toolkit (native package manager, conversion of Pascal and C/C++ sources to Nim, native doc comments, ...). Often, it allows to write a single program, which can be compiled to both native binary and JS script executable on either inside the HTML page or on the NodeJS. Nim provides excellent bidirectional binding with any of it's backends (via the interfacing and "emit"ing of the target code). Among the disadvantages are relatively small community, absence of the major version that causes minor incompatibilities between the ~yearly language releases (however these incompatibilities are really minor unlike in the Julia language releases). Nim by default provides thread local soft real time garbage collector, which is really efficient, cool and optional. The resulting binary and execution speed are comparable to the pure C implementation. Nim is one of few languages providing a cross-platform native UI library. Among the disadvantages should be mentioned the overwhelming number of pragmas and keywords / operators / functions ("notin" and "isnot" besides "is", "not", "in"; multiple "newXXX(...)" instead of the uniform "new Type(...)", etc.), inconsistent interaction with containers ("array" and "seq"[uence] passed differently to functions and have inconsistent initialization syntax). The excessive number of the keywords hopefully will be reduced / fixed on the first major release... Nim is extremely productive but the outlined inconsistency could be annoying to some extent.

Rust: the incremental compilation has been implemented since v1.24 (Feb, 2018, thought the language is still immature and there is the ongoing work; the alpha version of the incremental compilation appeared in Sept, 2016), before this the compilation was too slow. The memory usage for small applications often relatively high (5Mb with the default jemalloc allocator vs 1Mb for the similar app in C/C++). Rust is a memory safe and secure language. It has relatively compact runtime (similar to C runtime, don't misinterpret it as the virtual execution environment) and does not have a garbage collector that makes it perfect for IoT projects (Redox OS is implemented on Rust). Supports design by the contract via external libs. Rust does not allow implicit types conversion and upcasting suggesting explicit coercion that requires some additional efforts for the polymorphic behaviour. Rust still lacks flexible memory management features for the heap allocated objects present in C and C++ languages. Having arguably more powerful macro system comparing to C++, Rust still lacks some C++ metaprogramming capabilities including constexpr, variadic functions and templates overloading.

D: the standard library relies on the GC a lot (though being gradually updated to reduce this dependency), wich means in practice there hard to develop applications fast not using GC. Embedded devices have to use external implementation of the D runtime. D can't be used on less than 32 bit CPUs. GC requires linkage of the D runtime library and causes issues (prevents in practice) D libs linkage from other languages, even from Python. D provides amazing compilation speed, ability to natively link both C and C++ libs (also Python can be used from D via the external libs), and the high memory safety even operating with pointers, but has very moderate GC (sequential, i.e. "stops the world" and not very efficient). D features overview. And in brief some extremely convenient capabilities including Uniform Function Call Syntax, Design by the contract, template mixins, automatic dependencies management and packaging. There are a few arguable features / drawbacks in D like auto-decoding of strings (utf).
Since the late 2015 D development is speeded up, Vision documents appeared including intention to finally *make GC really optional* (eliminating standard library dependence on GC) that should open doors for the wide adoption of D in the embedded systems and anywhere else where C++ is used now. Also, there are plans to add out of the box code execution on GPUs. D also has been used in the cloud environments by some companies including Netflix, Sociomantic and Weka.io.

Python: the standard CPython interpreter is rather slow (10x slower than Pypy JIT); PyPy is much faster but has different ABIs and does not support BoostPython to link C++ libs also as lacks support of many Python modules. The main bottleneck of Python is GIL lock, which prevents multithreading apps development on native platform (is not an issue in VM based implementations like Jython). Pebble and PyExPool modules can be used for the truly parallel execution of the Python functions, modules and external binaries. NumPy, ScyPy, Pandas, Blaze (!does not support custom aggregate functions) and other efficient computation libraries exist with C backend. Cython can be used for very efficient computations and even wrapping and linking C++ libs. Like C/C++, Python3 provides stable API with awesome long-term backward compatibility, which forms a solid and reliable code infrastructure.

Julia: precompilation of including packages has been implemented, but not the packaging yet (the language is immature). 3rd party binary deployment of .jl scripts exists, but has limited support on Linux. Also a custom Julia image can be built with user-defined packages in base/userimg.jl. Julia has far not the most efficient GC, consumes relatively lots of RAM and has slow startup (2-5 sec), i.e. does not fit to execute small fast tasks. Julia is a very good choice when high-performance is necessary with simplicity of Python and support of extensive scientific calculations. There is an Escher lib for a web server embedding and interactive UIs. C++ code can be called from Julia using

Cxx.jl or CxxWrap.jl. Interactive iterative development is very convenient with Juno IDE based on Atom, also Jupyter / IJulia exists. A stable API of v.1.0 has not been released yet, so each ongoing Julia release is partially incompatible with the existing (former) packages, which reduces the code infrastructure and limits / prevents applicability in the long-term commercial projects. However, Julia is extensively used by the scientific and research communities and has been deployed in cloud environment and mainframes. Also, Julia naively links both C and Fortran libraries, which is extremely beneficial in the high performance computing environment.

Go: has relatively large binaries (~2Mb hello world, godoc is 16 MB), which is not appropriate for the embedded devices and SoCS (but this might be caused by the static linking, i.e. compile with gccgo and use dynamic linking to reduce the size, but "performance and security are seriously harmed by dynamic linking"; see also the row hammer attack that exposes the read-only shared memory: data can be changed without the actual write operation). Also it has slow (but very convenient) data interoperation with the linked C libs. Good for the fast development of powerful distributed systems, easy to learn and provides flexible operations with both pointers and references in spite of having GC! The GC is parallel and very efficient not stopping the world for more than 10 ms since Go 1.5. Among the main drawbacks is very limited and indirect support of the generic programming. Features overview: "Simple, Clutter-free Programming with Go". The payoff for the non-stopping the word GC is increased CPU usage / power consumption.

Swift: relies heavily on the Objective-C libraries that limits portability, does not have integrated async IO and native concurrency features (under active development). Has huge issues with compilation time in some cases, executes a few times slower than C++ and slower than JVM (Java / Scala), but does not have stop of the world problem inherent to the most of GC-based languages. Linking of C libraries with variadic args requires additional efforts. Swift uses smart pointers to manage memory allocated on heap, but does NOT provide custom references and pointers that affects the performance: "structure instances are always passed by value, and class instances are always passed by reference". So, in both theory and practice is far less flexible than Golang, Rust, D and C/C++. Swift was adopted by IBM as one of the languages for the Bluemix cloud platform.

Kotlin: is officially supported by Google for the Android development, also it can be compiled to JavaScript and even to the native executable (with some limitations). Kotlin provides concise and expressive syntax being not so verbose as Java and much faster to master than Scala. However, the cost of being concise is the underlying generation of the larger amount of the low-level methods comparing to the pure Java, which results in the larger jar/binary and slower execution speed. Development on Kotlin is faster than on Java but the code optimization by the compiler is weaker, which often is not a huge issue. Kotlin is an excellent choice for the Android development, prototyping and fast development of many projects where Java formerly used.

Scala: has rather slow compiler that causes long dev cycle. It typically runs on JVM => consumes a lots of RAM and limited with the available physical memory (garbage collection is directly uncontrollable and become too slow setting the heap larger than the size of the physical memory) and has other drawbacks (see JVM notes below). The native ahead-of-time compiler and lightweight managed runtime for Scala is available also as compilation to JavaScript. Scala provides extensive functionality including metaprogramming at the expense of the language complexity and slow compilation speed. Some Big Data processing & analysis platforms either built on Scala or provide API for it (Spark, Flink and others). Since the mid. 2017 Scala compilation speed is significantly improved but still much slower than Java/Kotlin. Formally, Scala has roughly twice less keywords and operators than Kotlin but providing a great power to developers with multitude ways to solve tasks, Scala is more complex and harder to master than Kotlin or Java.

JVM Notes: JVM is not the best choice for the memory bound / intensive or CPU intensive tasks. JVM might stop the word for up to a few seconds (especially for the large heap size) and it's G1 GC caused memory corruption on production systems, JVM allows to reuse huge amount of existing .jar libraries and run the same app on various platforms, but also brings huge overhead mainly in terms of the memory consumption. Since the Docker appeared and containers technology (LXC, Docker, LXD, Kubernetes) became mature, high performance systems executed in isolation nowadays often employ containers rather than JVM. Unlike JVM, containers allow much more flexible and efficient resource management avoiding intermediate overheads. Also, JVM JITs has been released for the computationally-demanding services: Azul Zing JVM JIT based on LLVM become available since 2017 (for both Java/Kotlin/Scala) instead of the ordinary JVM of OpenJDK / Oracle JDK and GraalVM JIT released in Apr, 2018.

See also:

- Reddit discussion of this post with some details about capabilities of some languages
- Energy Efficiency Across Programming Languages
- D vs Go vs Rust by Andrei Alexandrescu
- Google Trends
- Comparison of programming languages on Wikipedia
- Reference Counting VS Garbage Collection
- **Profilers Comparison**