# A Compendium of Public Software Tools for Document Structuring, Application Scripting, and General-Purpose Programming

Hassan AÏT-KACI*

March 7, 2020

**Abstract**

In the course of building software, even for non-industrial pursuits, one is always confronted with having to decide which language and/or system to choose, for such tasks as editing, documenting, and realizing one's intended specification as running software. With the recent ever-growing proliferation of scripting and programming idioms and environments, this is the case today more than ever before. The criteria upon which one bases such a choice depend in a crucial way on the specific nature of the software's intended use (what we want to achieve) and a tool's properties (what it can do). The most popular languages and library systems have been designed and implemented by seasoned programmers precisely seeking to ease expression and use of most commonly appropriate computations in specific types of applications. While catering to specific needs, these languages have also become popular due to their ease of use and interoperability with other popular widely used platforms. This document gives entry pointers to a few such languages and systems that I have been led to use or considered using for software specification and documentation in the course of my research and development activities.

***Caveat Emptor***

This document is for my personal information on various publicly available software tools that I have encountered or needed in the course of some of my software projects. I share it "*as-is*" (on my ResearchGate site). Also, by its very nature, this is an *evolving* document; so make sure to refer to the latest version.

*HAK Language Technologies — hak@acm.org.

# Table of Contents

# 1 Terminological Preamble

We first define what is meant by the most popular existing styles of programming and what constitutes a language in such styles, and a program in such a language.

- **Style of programming** — here is a rough description of various existing styles for specifying a computation:
  - **imperative execution** — this is when a program specifies a *process* as sequences of instructions, the execution of which will produce desired side-effects in memory, or on available I/O and network devices;[1]
  - **functional evaluation** — this is when computation is specified as nested functional expressions and function composition, the evaluation of which will yield the desired result and produce the desired side-effects;
  - **logical proof** — this is when computation is specified as logical statements using predicates (*i.e.*, relations), the proof of which will yield a desired solution and produce the desired side effects;
  - **constraint solving** — this the logical style where predicates are constraints.[2]
- **Kind of language** — a program can then be formulated as one, or a combination, of several kinds of language computational units:
  - **macro language** — where computation consists in transforming a purely lexical expression by replacing strings of characters by strings of characters;
  - **interpreted language** — where a computational unit (of any kind of programming) is executed dynamically;
  - **compiled language** — where syntactic units (in any kind of programming language) are typically statically type-checked, then translated into computing units of another language of any style, then executed or saved as executable binaries;[3]
  - **typed language** — whose syntactic units (of any kind of programming) use expressions and data organized into various types;
  - **object-oriented language** — which is a typed language (of any kind of programming) usually with subtypes that inherit the attributes of their supertypes and may define new attributes;
  - **scripting language** — where a (possibly typed) program is interpreted and/or compiled and has the ability to construct, possibly compile, and evaluate syntactic units on the fly from lexical units built with a macro-language facility and/or metaprogramming tools.

[1] Executing such instructions need not be finite; *e.g.*, operating systems or client/server monitoring.

[2] A constraint is a kind of predicate that can be proven with a special-purpose efficient solving algorithm.

[3] Usually, but not always, the compiled code consists in imperative-style instruction-based units that can be saved for reuse (*e.g.*, to form libraries of pre-compiled linkable binary code).

It should be noted, however, that although usually categorized as languages of one or the other of the above programming styles, many existing programming idioms typically combine features of several the above styles, even if by integrating them as "*impure*" features.

Please refer to the Rosetta Code for a partial programming system ontology. It is partial as it does not include scripting languages, constraint languages, nor many others. But it is a wiki-platform that expects to be completed as it evolves. I also found a very informative site on the history of programming languages up to today.

# 2 Introduction

I have used several programming languages and systems in my career. I have also designed a few. Still, my most recent software specification has been mostly in Java. This language choice is deliberate, although certainly not because it is my favorite. The essential reason is portability. Indeed, once compiled into the JVM, Java classes are guaranteed to run *everywhere* with the latest compatible Java libraries and interoperate through well-defined interfaces organized as abstract object-oriented type specifications. Better designed languages (*viz.*, Scala) exist that also compile to the JVM.[4] It can therefore co-execute along with javac-produced JVM code. This also implies that Scala can use javac-produced code, as well as conversely.[5] It would therefore make sense for me to switch to Scala for further development.[6]

This being said, Java and its many useful libraries (as well as its contenders such as Scala and its Akka libraries), are still, for many applications, *too generic* when certain kinds of specific computations need to be made native for reasons of frequency of usage.

Early AI programming languages provided lexical and syntactic tools to let one reshape the language to cater to one's most common specific needs. For example, quote, backquote, macros, memo-functions, in Lisp;[7] or cut, assert/retract, dynamic-arity symbols and DCGs (Definite Clause Grammars), in Prolog. This, however, led to anarchic non-standard tools. For this reason, the Java generation was given less freedom with syntax. This curbed the syntactic anarchy and allowed interoperability; however, this also frustrated the need for specific operators. Many new languages that followed can be seen as addressing such specific needs with the worry of garanteeing interoperability by dissociating the syntax level, which can thereby be adapted as needed, from the execution level, which is in fact the only part that must be kept compatible with the rest of the universe by sharing the most successful virtual machines such as the JVM, and more recently the LLVM.

[4] See this detailed explanation of how Scala features (*e.g.*, lazy access) are rendered using JVM instructions.

[5] But several Scala features do not exist in Java and require non-trivial Java constructs to be decompiled into (besides automatic setters and getters).

[6] To "*eScalate*" my software, so to speak. This has been on my stack for a while now, but I have been too lazy, and there have been more immediately pressing pragmatic priorities.

[7] Instructive links: S-Expression parsers in 35 languages; Lisp in C; S-Expressions in Scala.

# 3 My Program Development Tools

All the software I use for research and development is *free*. So you can use the same. All my research and development is done through Gnu Emacs and Cygwin.

## 3.1 Gnu Emacs

I discovered Emacs in 1980. There was no Gnu Emacs then. Emacs is the ancestor of Gnu Emacs in more than one respect. It was written in C in the 70s by James Gosling while he was in grad school at CMU.[8] Its name stood for "*E*ditor *MAC*ro*S*." It introduced many of the features that became part of Gnu Emacs, especially it programmability in Mocklisp, a macro-dialect of Lisp that Gosling conceived for defining text-editing macros.[9] Gosling's Emacs was discontinued in 1982 and part of its code reused and extended by Richard Stallman.[10]

All words decribing my appreciation for Emacs, and later Gnu Emacs, as a software-development tool will be understatements. It is my universal portal to ***everything*** concerning software development.[11] Why so? Well, the folklore says that "*EMACS*" stands for the standard keyboard "*Escape*, *Meta*, *Alt*, *Control*, *Shift*" keys, and this may be good intuitive way to think about it. Indeed, anything that would require a lot of mouse movements and clicks on a visual GUI editor, you can do extremely fast in Emacs with a combination of so-altered keyboard keys. Each of these five key-altering ways of keyboard input has a standard metamnemonic interpretation, and so there is a commonsense logic to it. Add to that automatic word completion with, or without, dictionary (it just looks around in its current buffers for a word that fits what you have typed so far). And you can use keyboard macros by telling Emacs to remember any sequence of keyboard keys that you type and repeat it *ad libitum*.

I switched to Gnu Emacs in 1982. Its most appreciated feature is that it is fully programmable and customizable as you wish using Gnu Emacs Lisp, a full Lisp interpreter, to change key bindings, define new functionality, "*the works!*" There are many predefined modes for most common programming languages, and of course you can define your own. The number of amazing features is too long to list all here.

Accordingly, using a proprietary system (*e.g.*, Unix, Linux, MacOS, Windows) makes no sense to me either. However, for pragmatic reasons, I use Windows as a bootstrap system.[12] However, I lobotomize it to its strict desktop functionality, and install all the free software running

[8] He is the same person that later designed and implemented Java.

[9] Gnu Emacs Lisp was a redesign of the original Gosling Emacs Mocklisp as a full Lisp interpreter and compiler.

[10] There has been a dispute between Gosling and Stallman regarding the paternity the many ideas in this tool. Stallman is also an outspoken political activist and a defender of free software through the Gnu organization, by describing his operation system as the left-recursive "*Gnu's Not Unix*." He is the founder of the Free Software Foundation (FSF).

[11] If you don't know Gnu Emacs and you do not want to rely on software-vendor products, check it out.

[12] Why so? Because it is the most popular system and therefore it is familiar to the great majority of users. It is for the same reasons that I use English to write this up. I avoid *all* Apple products like the plague as they are deliberately conceived with no interface to other products, software or hardware — even their own previous versions. In addition, Apple software contains system backdoors that make your private data accessible to them. This is a business policy aimed at trapping all customers into using only Apple products.

on Windows that I need for applications; *viz.*, Cygwin, Gnu Emacs, LaTeX, FileZilla, GNUzilla, Thunderbird, *etc.*, ... However, one must beware that freely available software (such as, *e.g.*, this) is no necessarily free in Gnu's sense.

### 3.2 Cygwin

Cygwin is an amazing and wonderful *free* Unix emulator running on Windows. So I use Gnu Emacs on Windows as a universal editor capable spawning any number of Cygwin shells I need, each running Cygwin's tcsh that I customize with my own .profile. In this manner, I run all software through a Cygwin tcsh shell under Gnu Emacs. I start with a mother Cygwin bash terminal window. This initializes all my system-wide variables, starts an X Windows server running in the background, then switch to tcsh. From this terminal, I can then create as many Gnu Emacs windows as I want, each customized by loading my own ~/.emacs file containing all the Emacs variables and functions in Gnu Emacs Lisp. So I end up with a portable *free* virtual UNIX emulator through fully editable Gnu Emacs windows whose shell command is defined to run tcsh, customized as I wish in all minute details. All the while, I also have access to any useful freeware running on Windows that I can download from the Internet, while never concerning myself with any junkware or adware from the latter.

## 4 Document Preparation Systems

Whenever possible, I avoid using any proprietary document-preparation system. In addition, the quality of scientific type-setting they offer does not match that of free systems such as TeX and LaTeX.

### 4.1 TeX and LaTeX

The type-setting system TeX was designed and implemented by Donald Knuth in 1980, along with its font system METAFONT, and much more, specifically for his own use as he was frustrated with the ugly rendering of mathematical type-setting done by editors. However, TeX is a low-level type-setting system that is not intended for *naïve* users: while it is fully customizable, it is many a time at one's own risk.

At about the same time (1980), a type-setting system called Scribe was designed and implemented by Brian Reid for his PhD at CMU. Contrary to TeX, ease of use was its major feature. In particular, it made use of begin/end parenthetic markups for annotating a text document. In 1982, I was working on my PhD at Penn. To type-set my thesis, I first attempted using TeX but quickly gave up as it required too high an investment just for learning a tool. So I used one of the early public releases of Scribe and really enjoyed the ease of use and freedom it offered me, letting me concentrate on the contents of my thesis rather than its type-setting.

There was a need to bridge the gap between TeX and Scribe in order to offer the best of two worlds. So Leslie Lamport implemented LaTeX at SRI as a set of TeX macros emulating the

kind of higher-level document type setting annotation used by Scribe.[13] LaTeX was publicly made available in 1984. I welcomed it as soon as it was released and have been a faithful user since then. My investment in learning TeX was not wasted either as it has helped me enhance LaTeX with my own style configurations.[14]

Most well-known research scientists today use LaTeX for composing their research reports and publications, and most scientific editors offer LaTeX-based styling macro-packages for conference papers, journal articles, and books.

## 4.2 Internet Document Markup Languages

### 4.2.1 HTML and CSS

HTML (Hyper-Text Markup Language) is an instance of XML (eXtensible Markup Language), itself an adaptation of Library's Science's SGML (Standard Generalized Markup Language). This family of languages are used on text files as angle-bracketed markup tags to determine how to interpret the text so-bracketed. A marked-up text file is therefore a finite tree of text units.

HTML is a markup language specifically meant for type-setting documents containing text, and hyper-text references to external local or Internet resources (other HTML files, images, audio, *etc.*). The current official version of HTML is HTML5.

CSS (Cascading Style Sheets) is the parameter-configuration language used for customizing HTML tags. Many fonctionalities that should be parameterizable in previous versions of HTML were eliminated in favor of CSS-specifiable parameter configuration.

### 4.2.2 Markdown

Markdown is a lightweight markup language (compiles to HTML).

See Pandoc the universal document converter.

### 4.2.3 Whatwg

Whatwg is an evolving version of HTML.

# 5 Graphical User Interfaces

I personally avoid using any such tools.[15] However, most of my students, many colleagues, and the vast majority of industrial software developers use them. Also, many popular languages are supported.

Eclipse and IntelliJ IDEA are the most popular software-development IDEs. They both support advanced software-development tools; the essential difference the former is open-source

[13] The same sort of type-setting text annotation used later is HTML.

[14] I even wrote the official technical-report LaTeX styles of several research labs I worked for.

[15] See why in Section 3.

from a non-profit organization, while the latter is a private company's product.

## 5.1 Eclipse

Eclipse is a freely available multi-language IDE.

It is developed by the Eclipse Foundation, which provides interfaces for most of the popular languages, and many more through plugins; *e.g.*, Scala, Clojure, Kotlin, Prolog, DCGs, *etc.*, ...

## 5.2 IntelliJ IDEA

IntelliJ IDEA is also a freely available multi-language IDE.

It is developed by IntelliJ, which provides interfaces for many popular languages and in particular a Kotlin IDE. See IntelliJ IDEA's minimal survival guide.

## 5.3 Atom

Atom is a multiplatform GUI interface all in HTML. Its essential ideas is that, just like everything is Lisp is an S-Expression, or everything is Prolog is a First-Order Term, in Atom, everything is a web site:

> "*Atom is a specialized variant of Chromium designed to be a text editor rather than a web browser. Every Atom window is essentially a locally-rendered web page.*[16] *Atom Flight Manual*"

Atom also provides an interactive visual editor for Julia.

## 5.4 Juno

Juno is an IDE for Julia built on top of Atom.

## 5.5 Octave

Octave is a scientific programming language developed as a Gnu software tool for graph plotting and general statistical data analysis. It has the following features:[17]

- Powerful mathematics-oriented syntax with built-in plotting and visualization tools.
- Free software, runs on GNU/Linux, macOS, BSD, and Windows Drop-in compatible with many Matlab scripts

[16] https://flight-manual.atom.io/getting-started/sections/why-atom/#the-native-web
[17] Quoting from their site.

An Octave mode for Gnu Emacs is available here.

There is a project called Octclipse making Octave run with Eclipse.

There is an interface for Octave to work with the JVM (and therefore use Java and Scala code). Regarding JIT compiling, there is a Gnu project to make Octave target the LLVM. But there are issues (mainly due to the fact that the LLVM is being a constantly moving target).

## 5.6 R and RStudio

R is also a popular open-source freely-available scripting, graphing, and general analytical computation system for statistical data analysis with its own IDE (RStudio) which is not based on Eclipse.

Using R through Eclipse is done with the Statet plugin.

R comes with a (still maturing but interesting) package called Tidy Eval that provides tools to work with core language features of R using an approach to code manipulation known as the tidyverse.[18] The main construct it provides is a quote/unquote mechanism for building quoted closures (what they call "*quosure*.")

## 5.7 Other open-source systems for statistical data analysis.

See here for other (not necessarily free or Eclipse-compatible) systems for statistical data analysis.

# 6 Software Orchestration

## 6.1 Configuration Files

### 6.1.1 Make

Make is a Unix command that takes instructions from a Makefile for coordinating the compiling of sources, and general directory and file creation, manipulation, and processing, by naming tasks defined as sequences of parameterizable shell commands and specified as rule-based inter-task dependencies. It uses the same idea of environment variables used by Unix which can be accessed, redefined, as well as new ones defined.

### 6.1.2 Ant

Ant is an XML-based improvement from Apache on ideas from the classical Unix tool Make. It takes its specification in text form like Make, but what corresponds to a Makefile is now encoded in XML in an Ant Builfile. Although Ant/AntBuilder is admittedly an improvement in portability and functionality on Make/Makefile, this not so however concerning syntax. But this is a moot restriction since Ant is not really meant for direct human pro-

[18] This system has nothing to do with Tide nor *vice versa*.

duction nor consumption, but to be used as XML-encoded configuration integrated in software-development platforms (*e.g.*, Eclipse) for Java, Python, C++, or whatever, on any platform (*e.g.*, Gradle's Ant for Gradle).

## 6.2 Scripting Languages

A scripting language is a programming language to specify how to orchestrate computation consisting of running several programs and organize their consistent composition and input/output manipulation. It can be (and quite often is) used with statistical data analysis tools (such as Octave, R, *etc.*).

I first started looking at Python, then Rust, then Julia, *etc.*, ... The latter is an interactive typed scripting language, with type inference and default parameters, with all the capabilities of a static and dynamic programming language. In the process of downloading a working version, I ran into a few other systems that Julia has a connection with: Kotlin, Atom, *etc.*, ...

There is no order in this list. So the rest is organized as a set of personal notes in learning about each new language and/or software system as I proceeded.

There are many existing scripting languages. The best known is JavaScript.[19] Among the (many, many) other script languages, there is also TypeScript and Kotlin.

Declarative scripting uses constraint-solving to ease scripting specification in the same manner as Constraint Logic Programming use constraint-solving for general-purpose programming.

### 6.2.1 JavaScript

The name invites the confusion with the general-purpose compiled language Java, but the two systems have little in common and were designed by different people at different times in different companies for different purposes. Java was designed by James Gosling at Sun Microsystems in 1984, and JavaScript was designed by Brendan Eich at Netscape in 1995. Calling it JavaScript was a marketing ploy.

JavaScript is the oldest and most popular scripting language. Being the oldest, it has been optimized over the years to become also the most efficient, so that most of the more recent languages used for scripting are actually syntactic front-ends that are compiled into JavaScript. (See also this list.)

Here is a Gnu Emacs mode for JavaScript.

### 6.2.2 TypeScript

This the scripting language designed by Microsoft that has a Visual Studio interface. The command-line TypeScript compiler can be installed as a Node.js package.[20]

[19] Check out these sites for JavaScript resources: client-server orchestration (Node.js), various public resources, SpeckyBoy libraries.

[20] See samples and TypeScriptDoc for more documentation.

Node.js© is a JavaScript runtime built on Chrome's V8 JavaScript engine. Many tools (*e.g.*, AngularJS, *etc.*) are just defined like this.

### 6.2.3 Python

This has become one of most preferred and used language because it is convenient both for scripting and general-purpose programming and can run on varied plaforms (including Android for easy development of cell apps), with a good set of publicly available tools, such as:

- the Python Standard Library;
- PyPi, the Python Package Index;
- NumPy, a widely used Python package for numerical calculations, in particular the kind required for data analysis;
- PyTorch, one of the hottest Python libraries for Machine Learning applications.

### 6.2.4 Julia

Julia is a multi-purpose interactive script-like functional object-oriented language. It compiles to the LLVM. According to their website, Julia offers:

- Multiple dispatch: providing ability to define function behavior across many combinations of argument types;
- Dynamic type system: types for documentation, optimization, and dispatch;
- Good performance, approaching that of statically-compiled languages like C;
- Built-in package manager;
- Lisp-like macros and other metaprogramming facilities;
- Call Python functions: use the PyCall package;
- Call C functions directly: no wrappers or special APIs;
- Powerful shell-like capabilities for managing other processes;
- Designed for parallelism and distributed computation;
- Coroutines: lightweight "green" threading;
- User-defined types are as fast and compact as built-ins;
- Automatic generation of efficient, specialized code for different argument types;
- Elegant and extensible conversions and promotions for numeric and other types;
- Efficient support for Unicode, including but not limited to UTF-8;
- MIT licensed: free and open source.

Atom provides an IDE for Julia.

There is a Gnu Emacs mode for Julia called 'julia-mode.' It is available on GitHub.

### 6.2.5 Kotlin

Kotlin is a JVM-compiled multi-platform for developing all-purpose apps. It compiles to Android, JavaScript, or native assembly language for:

- iOS (arm32, arm64, emulator x86_64)
- MacOS (x86_64)
- Android (arm32, arm64)
- Windows (mingw x86_64)
- Linux (x86_64, arm32, MIPS, MIPS little endian)
- WebAssembly (wasm32)

It comes with a Java-to-Kotlin code converter.

Kotlin interfaces well with existing Java IDE tools. This is the case for Eclipse and IntelliJ IDEA (using Maven for software-project management).[21] There are others; *e.g.*, Gradle, AndroidStudio, *etc.*, ...

### 6.2.6 Rust

Rust is a relatively recent all-purpose programming language described as,

> "*Rust is a modern systems programming language focusing on safety, speed, and concurrency. It accomplishes these goals by being memory safe without using garbage collection.*"

This last feature (no garbage-collection) is a bit of a marketing ploy. Indeed, if Rust has no garbage collector, how does it clean up memory? In fact, what it does is, contrary to languages that perform garbage-collection either explicitly (user-triggered) or periodically (when memory runs low), it *generates memory-reclaiming code at compile-time*. So, technically, it is true that there is no garbage-collector in Rust because garbage-collection code is distributed all over the code and executed each time a memory-consuming unit is run and exited. This does save the user the need to produce such code (as in C, or C++), as well as the runtime's need to check on the need to reclaim memory periodically when running low. However, Rust does spend memory-reclaiming processing time while executing memory-hungry code — or it does not, but then memory explodes. There is no magic.

On the other hand, Rust uses new interesting programming concepts. For example, it has no classes but uses Haskell-inspired interfaces called *traits* (which specify the *behavior* of data sharing a trait), as separate from *types* (which specify the *structure* of data), for which an *implementation* of the traits a type shares can be specified. Reflection in Rust is much better conceived than in Java or Scala and offers the convenience of easily adapting the syntax as one wishes. This feature of Rust makes it a powerful scripting as well as general-purpose language. Finally, Rust compiles to the LLVM.

[21] Using Maven for Kotlin.

Rust has a complete and well conceived documentation, and it comes with many popular program-development tools and useful interfaces: *e.g.*, command-line interpreter, Gnu Emacs, Eclipse, Atom, IntelliJ IDEA, *etc.*, ...

### 6.2.7 Tide

Quoting from the Tide site:

> "*Tide is a concurrent stream-based programming language, where the stream is both the base datatype and the main method of function execution and coordination. Tide takes inspiration from languages such as Haskell, Python, JavaScript, and bash.*"

There are a few Gnu Emacs Tide modes available. I like this one.

### 6.2.8 Clojure

Clojure is a script dialect of Lisp with macros.

### 6.2.9 Which to choose?

Which to choose: Python, Rust, Julia, Kotlin, ...? For the two latter ones, see how they compare. However, it appears that some disagree regarding Julia's attractiveness. Some also complain about Rust.

# 7 Virtual Machine Languages

Compilers generating portable lower-level virtual-machine code is the key to programming language interoperability. The best known is the JVM; the LLVM is more recent with new features and using it is gaining momentum.

## 7.1 JVM

The *Java Virtual Machine*, or JVM, is what made Java really popular as it made it portable on virtually any system that would provide a backend for it.

Java lacks facilities for easy metaprogramming and "on-the-fly" program synthesis and execution (its syntax forbids it). But is supports a limited form of code manipulation using *reflection*. This "compile-&-go" is made possible by the JVM's "*Just-In-Time*" compiler.

Since Scala also compiles to the JVM, it uses JIT for reflection too. Being a functional language, Scala has more flexibility; However, being also a typed language, it is limited in this regard as well.[22]

[22] See also this.

### 7.2 LLVM

The "*Low-Level Virtual Machine*" (LLVM) is a language-independent abstract machine ensuring portability and supporting client/server process distribution. It is meant, like the JVM to be machine code, produced by a higher-level-language compiler. Quoting from the LLVM site:

> "*... you can use LLVM to compile Ruby, Python, Haskell, Java, D, PHP, Pure, Lua, and a number of other languages.*"[23]

Its originality and particularity, compared to the JVM which has been also the target of many languages, is that it constantly optimizes its code through lifelong code analysis and transformation.[24] These features have made it become the target language of choice of several languages (*e.g.*, Julia).

## 8 Recapitulation

The various software-building idioms and tools that have been enumerated in this document do not, by far, constitute an exhaustive set. They have been deliberately limited to some of the most popular in the present time for producing Internet-aware software applications.

An essential idea, in our opinion, that seems to have emerged quite naturally is that the context of local script or program working on:

1. *files*, organized in some kind of:
2. *directory-tree file system*,

has now moved to Internet-aware type-safe self-optimizing multi-platform software working on:

1. *URI*s, organized in some kind of:
2. *distributed network*, whether local or over the Internet.

Tools for editing and building software such as IDEs exist for several of the most popular programming languages. This is made possible by using systems that provide multi-platform tools for front-end tasks such as editing and compiling, and running them on a language-independent target substrate.

Although still evolving, the LLVM is slowly emerging as a popular target language over the widely used JVM as a language-independent lower level architecture since it not only enables compilation of source code (of the most popular languages) into virtual machine code, it also constantly performs execution optimization through dynamic code transformation.

[23] Although the LLVM official site states that "LLVM" is not an acronym, it was one originally since it started as the "Low-Level Virtual Machine."

[24] See detailed description.

# Named-site index in alphabetical order

**J** Java
javac
Javadoc
JavaScript
JIT
Julia
Juno
JVM

**K** Kotlin

**L** LaTeX
Leslie Lamport
LLVM
LearningJS
Linux
Lisp
Lua

**M** MacOS
Make
Makefile
Markdown
Maven
METAFONT
Mocklisp

**N** NumPy
NumPy Tutorial

**O** Octave
Octclipse

**P** Pandoc
PHP
Prolog
Pure
PyCall
PyPi
Python
PyTorch

**R** R
Rosetta Code
RStudio
Ruby
Rust