Module 01

Das

Module 01: CS-1319-1: Programming Language Design and Implementation (PLDI)

Course Information and Introduction

Partha Pratim Das

Department of Computer Science
Ashoka University

ppd@ashoka.edu.in, partha.das@ashoka.edu.in, 9830030880

August 28 and 29, 2023

Introductions

- Of the Instructor
- Of the TF and TAs
- Of the Students

Course Material

- Slides will be made available on course webpage
- Books:
  - **Programming Language Implementation**
    - ▷ **Compilers: Principles, Techniques, and Tools (2nd Edition) by A.V. Aho, Monica S Lam, R. Sethi, Jeffrey D. Ullman (Pearson / Addison-Wesley). (Dragon Book)**
    - ▷ Flex and Bison by John Levine (O'Reilly)
    - ▷ Compiler Design in C by Allen Holub
    - ▷ Advanced Compiler Design and Implementation by Steven Muchnick
  - **Programming Language Design**
    - ▷ **Concepts of Programming Languages, Robert W. Sebesta, 11th Edition. (Sebesta)**
    - ▷ Programming Languages: Principles and Practices by Kenneth C. Louden and Kenneth A. Lambert (Cengage Learning)
    - ▷ Programming Language: Principles and Paradigms by Allen Tucker and Robert Noonan (McGraw-Hill Education)
    - ▷ Programming Language Pragmatics by Michael L. Scott (Morgan Kaufmann)

Classes

- **Classes**
  - MON (16:40–18:10), TUE (16:40–18:10)
  - Classroom: AC-01-LT-207
  - Online on Google Classroom – offline when I am in campus
- **LMS**
  - Google Classroom: Announcements, submissions, lecture presentations, etc.
  - Post-delivery videos on YouTube, if recorded
- **Office Hours**
  - To decide based on mutual convenience
  - By appointment with Instructor / TF / TA, otherwise
  - Use `cs1319-staff@ashoka.edu.in`
- **Attendance**: Compulsory, but no marks for it. Still attend:
  - Gives you a chance to ask questions
  - Everything is not in the textbooks and Quizzes will use content covered in class
  - Class participation is the best way to learn
  - University is investing to get me here – make the most of it

Evaluation

- Assignments: 30%
- Quizzes: 20%
- Mid-term Test: 20%
- End-term Test: 30%

Assignments and Quizzes

- Assignments:
  - Assignments are individual or by groups of two students (to be specified)
  - Assignments will focus on problem solving – some will need paper work out while others will need working on the machine with OSS tools to build small utilities
  - All assignments have to be typed using latex (no handwritten) – for any drawings one can use image of hand drawn figures or generate by painting applications
  - The submissions for assignments will be accepted online up to the specified deadline
  - For each extra day of late submission, assignment loses 10% of its value. No submission beyond five days will fetch any credit
  - No submission through mail or directly to the TA will be entertained
  - Plagiarism in assignment will lead to 100% penalty for all parties involved
- Quizzes:
  - Continuous assessment through quizzes
  - Will be announced 2-3 days advance, or be a surprise
  - Quizzes to be answered as handwritten, scan of the same to be submitted

Prerequisites

[1] Programming in C / C++

[2] Data Structures

[3] Algorithms

[4] Software Engineering (desirable)

[5] Formal Languages and Automata Theory (desirable)

[6] Theory of Computation (desirable)

TAs and Teachers

Sl.#	Name	Email	Mobile
1	Adwaiya Srivastav	adwaiya.srivastav_tf@ashoka.edu.in	
2	Ahlah Husain	ahlah.husain_asp24@ashoka.edu.in	9935372173
3	Drumil Deliwala	drumil.deliwala_asp24@ashoka.edu.in	9082890833
4	Gautam Yajaman	gautam.yajaman_asp24@ashoka.edu.in	9082587028
5	Partha Pratim Das	partha.das@ashoka.edu.in	9830030880

Important Dates

To be announced

Why PLDI?

Why PLDI? What do you expect from this course?

Why study *Programming Language Design and Implementation?*

Programming Languages are living entities of CS

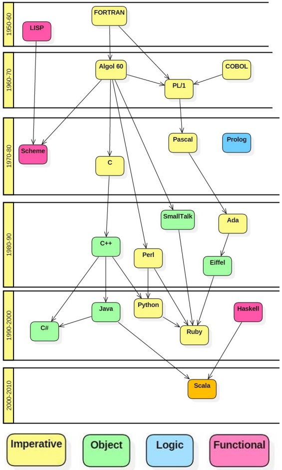
- They are born (designed and implemented)
- They grow in their use, and
- They fade out to others
- Long serving languages need regular re-births and / or repurposing (C++98 $\rightarrow$ C++11 $\rightarrow$ C++14 $\rightarrow \cdots$)
- Lot of languages (actually most) die when they stop serving their computing role
- A few are resurrected (like LISP), when we rediscover their worth after decades

Evolution of Programming Languages: Factors of Influence

- **Efficiency**: Time, Space, Power & Footprint (hand-held devices)
- **Ease of Learning, Reading, Writing**: Python, Basic
- **Library Support**: Standard (C++11) & 3^{rd}-party (Python)
- **Ease of Documentation**: Python
- **Availability of Tools**: C for hand-held devices for frugal tools
- **Portability**: Java, Python
- **Safety**: Robustness on resource leaks & errors (Java, Python), run without failure (Ada), difficult to hack
- **Provability**: 70%+ of Boeing & Airbus is s/w, believed to be 99% proven
- **Mathematical Foundation**: Haskell
- **Systems' Level Access**: C, C++
- **Politics**: Microsoft created C# as Java was proprietary of Sun Microsystems
- ...

Evolution of Programming Languages: Programming Language Genealogy

History of Programming Languages

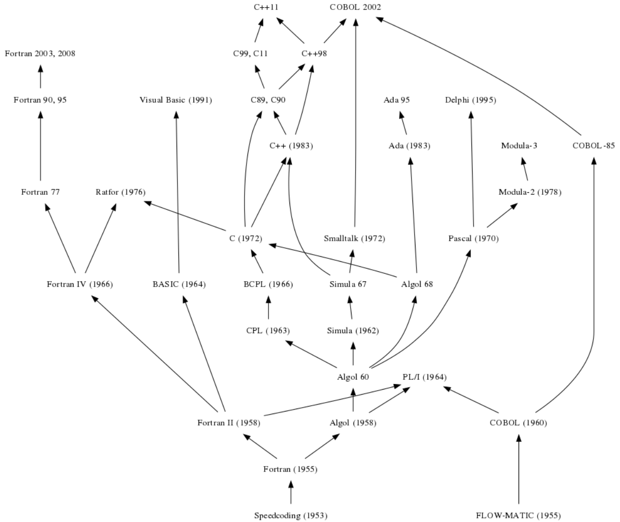


Paradigms: *Imperative*: Algorithms + Data, *Object*: Data, *Logic*: Facts + Rules + Queries, and *Functional*: Functions

- **FORTRAN**: John Backus, IBM
- **LISP**: John McCarthy
- **Algol 60**: John Backus & Peter Naur
- **COBOL**: Grace Murray Hopper
- **PASCAL**: Niklaus Emil Wirth
- **Prolog**: Alain Colmerauer & Philippe Roussel
- **Scheme**: Guy L. Steele & Gerald Jay Sussman
- **C**: Brian W. Kernighan & Dennis M. Ritchie
- **SmallTalk**: Alan Kay, Dan Ingalls, & Adele Goldberg
- **Ada**: Jean Ichbiah & Tucker Taft
- **C++**: Bjarne Stroustrup
- **Objective-C**: Brad Cox
- **Perl**: Larry Wall
- **Java**: James Gosling
- **Python**: Guido van Rossum
- **Haskell**: Paul Hudak
- **C#**: Microsoft Corporation
- **Ruby**: Yukihiro Matsumoto
- **Scala**: Martin Odersky

Source: Programming Language Evolution, Computer History: A Timeline of Computer Programming Languages, HP Tech Takes

Evolution of Programming Languages: Programming Language Genealogy



Evolution of Programming Languages: 1950's

First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
1957	2018	**FORTRAN**	John Backus IBM	imperative	Scientific computations, formula translation	Paradigms added: structured (FORTRAN 77), generic & array (FORTRAN 90), OO (Fortran 2003), concurrent (Fortran 2008)
1958		**Algol** = **ALGO**-rithmic **L**anguage	John Backus, Peter Naur	imperative, structured	Algorithmic programming	Worked as a basis for Pascal, C etc. Introduced BNF
1959	2014	**COBOL**	Grace Murray Hopper CODASYL	imperative	Business, file handling, portable with English-like (Verb) syntax	Paradigms added: generic & OO (COBOL 2002)
COmmon Business-Oriented Language						
1959		**LISP** = **LIS**t **P**rocessing	John McCarthy MIT	functional, procedural, reflective, meta	Symbolic computation, functional	Resurrected in 2000. Dialects: Common Lisp, Scheme, Arc, Hy, Nu, Liskell, LFE

Evolution of Programming Languages: 1960's

First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
1962	2022	**Simula**	Ole-Johan Dahl, Kristen Nygaard Norwegian Computing Center	imperative, structured, OO	Simulating VLSI designs, process modeling, comm. protocols, algorithms, typesetting, graphics, & education	Considered the first OOPL. Versioned as **Simula 67**. Influenced design of Smalltalk, C++, Java, & C#
1964	2019	**PL/I**	IBM	imperative, structured	One language, all features	Surprisingly still in use
1964		**BASIC**	John G. Kemeny, Thomas E. Kurtz	non-structured	For students without strong technical / math background	Resurrected in 1990 by MS VB. Paradigms added: imperative & OO
Beginners' All-purpose Symbolic Instruction Code						

Evolution of Programming Languages: 1970's: First Half

First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
1970	2022	**Pascal** / **Delphi**	Niklaus Wirth	imperative, structured	A small, efficient language to encourage structured programming & data structure	An algorithmic language simpler than Algol
1972	2000	**Prolog** = **PRO**gramming in **LOG**ic	Alain Colmerauer, Philippe Roussel, Robert Kowalski U of Edinburgh	logic, declarative	Logic programming for AI and computational linguistics	Based on first-order logic and is declarative
1972	1980	**Smalltalk**	Alan Kay, Dan Ingalls, Adele Goldberg Xerox PARC	object-oriented	Object-oriented, dynamically typed reflective programming language	Created for educational use - constructionist learning
1972	2018	**C**	Dennis Ritchie Bell Labs	imperative, structured	General-purpose programming - good to build Systems	Used for OS, DD, protocol stacks, and embedded systems
Byproduct of UNIX kernel design. Brian Kernighan helped write the K&R C book.						
1972	2016	**SQL** = *sequel*	Donald D. Chamberlin, Raymond F. Boyce	declarative	Database language on relational model	Structured Query Language

Evolution of Programming Languages: 1970's: Second Half

First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
1975	2013	**Scheme**	Guy L. Steele, Gerald Jay Sussman MIT AI Lab	functional, imperative, meta	General purpose functional programming	Dialect of the Lisp family
1978	2021	**Modula-2**	Niklaus Wirth ETH Zurich	imperative, structured, modular, data and procedure hiding, concurrent	Programming OS and application software	Preceded by Pascal, followed by Modula-3 & Oberon
1978	2022	**MATLAB** = **MAT**rix+ **LAB**oratory	Cleve Moler MathWorks	functional, imperative, OO, array	Proprietary multi-paradigm numeric computing environment supporting matrix manipulations, plotting of functions and data, implementation of algorithms, creation of UIs, and interfacing with other languages	MATLAB has more than 4 million users worldwide from engineering, science, and economics (2020)

Evolution of Programming Languages: 1980's: First Half

First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
1980	2019	**VHDL = VHSIC HDL**	US DoD	concurrent, reactive, dataflow	Model the behavior & structure of digital systems at multiple levels of abstraction - system level to logic gates, for design entry, documentation, and verification	Language for physical and behavioural circuit design **HDL = Hardware Description Language**
1980	2012	**Ada**	Jean Ichbiah, Tucker Taft US DoD	structured, meta, imperative, OO, AO, concurrent, array, distributed, generic	Structured, statically typed, imperative, and OO programming	Focused on code safety and maintainability
1983	2020	**C++**	Bjarne Stroustrup	imperative, OO, generic, modular	General-purpose programming for Systems & Application	Redesign of C with OOP
1984	2005	**Verilog**	Gateway Design Automation	Structured	HDL to model design and verification of digital circuits at register-transfer level (RTL)	Circuit design in C style
1984	2017	**Objective-C**	Brad Cox, Tom Love	imperative, OO	General-purpose, OO programming	Smalltalk-style messaging to C

Evolution of Programming Languages: 1980's: Second Half

First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
1985	2020	**Miranda**	David Turner Research Software Ltd	lazy, functional, declarative	Lazy, purely functional programming language	First commercial purely functional language
1986	2022	**Eiffel**	Bertrand Meyer Eiffel Software	OO (Class-based), generic, concurrent	Reliable development of commercial software	Principles: design by contract, command–query separation, uniform-access, single-choice, open–closed, option–operand separation
1986	2021	**Erlang**	Joe Armstrong, Robert Virding, Mike Williams Ericsson	concurrent, functional	General-purpose, concurrent, functional programming with GC runtime system	WhatsApp is written in Erlang
Named for the Danish mathematician Agner Krarup Erlang						
1987	2022	**Perl**	Larry Wall	functional, imperative, OO (class-based), reflective	General-purpose, interpreted, dynamic programming language for scripting	Text editing for report processing **Perl 5**: 2000-19 **Perl 6**: 2019-, known as **/Raku**

Evolution of Programming Languages: 1990's: First Half

First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
1990	2010	**Haskell**	Paul Hudak & others	Purely functional	General-purpose, statically-typed, purely functional programming w/ type inference & lazy eval.	Designed for teaching, research & industrial apps, Semantics based on **Miranda**
1991	2021	**Visual Basic**, **Visual Basic. NET**	Microsoft	structured, imperative, OO, declarative, generic, reflective, event-driven	Multi-paradigm, OOP language application development on .NET Framework	**VB**: 1991-98 **VB.NET**: 2001-21 Easy drag-and-drop & resource management, Windows specific
1991	2022	**Python**	Guido van Rossum	OO, imperative, functional, structured, reflective	General-purpose programming language with dynamic types and garbage-collection	Easy to learn, read, & write with indentation, Huge 3^{rd} party libraries
1993	2022	**R** R = Initials of creators	Ross Ihaka, Robert Gentleman R Core Team & R Foundation for Statistical Computing	imperative, OO, functional, reflective, array	Programming language for statistical computing, data mining, and graphics	Written in C, FORTRAN, and R itself

Evolution of Programming Languages: 1990's: Second Half

First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
1995	2022	**Java**	James Gosling Sun Microsystems	generic, OO functional, imperative, reflective, concurrent	General-purpose class-based, OOP language to let programmers *write once, run anywhere (WORA)*	Min. implementation dependencies, Good for hand-held devices, Portable
1995	2021	**JavaScript / JS**	Brendan Eich Netscape	event-driven, functional, imperative, OO	Used in 98% of web on the client side for webpage behavior, with 3^{rd}-party libraries	JIT compiled with dynamic typing, prototype-based OO, 1^{st}-class fns
1995	2022	**PHP = PHP: Hypertext Preprocessor**	Rasmus Lerdorf	imperative, OO, functional, reflective	General-purpose scripting language geared towards web development	Processed on a webserver by PHP interpreter as daemon or as CGI, Earlier, **Personal Home Page**
1995	2022	**Ruby**	Yukihiro Matsumoto	functional, imperative, OO, reflective	General purpose + interpreted web applications	Emphasis on productivity & simplicity
1997	2022	**E**	Mark S. Miller Electric Communities	OO, message passing	OOPL for secure distributed computing	Good for h/w, s/w co-design

Evolution of Programming Languages: 2000's

First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
2000	2021	**C# / C Sharp**	Anders Hejlsberg, Mads Torgersen Microsoft	structured, OO, imperative, event/ task-driven, reflective, functional, meta, concurrent	General-purpose, multi-paradigm programming	C# = C++ + VisualBasic, retaliatory tonewline Java, Windows specific
2001	2022	**D / dlang**	Walter Bright, Andrei Alexandrescu	functional, imperative, object-oriented	Multi-paradigm system programming language	D = C++ - C + TDD, safety, and expressive power
2002	2018	**System Verilog**	Synopsys	Structured (design), OO (verification)	HDL & HVL to model, design, simulate, test & implement systems	Preceded by Verilog, used in semiconductor
2003	2022	**Scala** = *SCA*lable+ *LA*nguage	Martin Odersky	concurrent, functional, imperative, OO	Strong statically typed general-purpose programming	Designed to address criticisms of Java
2003	2022	**Apache Groovy**	James Strachan, Bob McWhirter	OO, imperative, functional, AO, scripting	OOP language for Java platform for programming & scripting	Offshoot of Java
2009	2022	**Go / Golang**	Robert Griesemer, Rob Pike, Ken Thompson Google	concurrent, imperative, OO	Programming large codebases on n/w & multicore machines	Improved programming productivity

Evolution of Programming Languages: 2010's

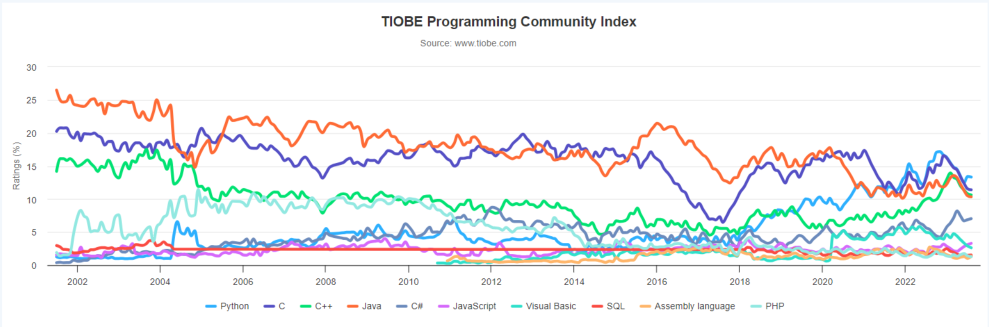
First Rel.	Stable Rel.	Language	Creator / Organization	Paradigm	Domain	Remarks
2011	2022	**C++11**, **C++14**, **C++17**, **C++20**	Bjarne Stroustrup, C++ Standards Committee	procedural, imperative, functional, object-oriented, generic, modular	General purpose programming, Systems programming, Application programming	C++ Core lang.: multithreading, metaprogramming, uniform init., performance. C++ Std. Library
2011	2022	**Dart**	Lars Bak, Kasper Lund Google	functional, imperative, OO, reflective	Web & mobile client dev., also to build server and desktop	Usable with Open UI SDK **Flutter**
2012	2022	**Julia**	Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah	multiple dispatch, procedural, functional, meta, multi-staged	General-purpose high-performance, DP language well suited for numerical analysis and computational science	GC, eager evaluation, & libs for FP calculations, linear algebra, RNG, and RE matching
2014	2022	**Swift**	Chris Lattner, Doug Gregor, John McCall, Ted Kremenek, Joe Groff Apple	protocol-oriented, object-oriented, functional, imperative, block structured, declarative, concurrent	App development in Apple's Cocoa and Cocoa Touch frameworks	Replaces Objective-C while reusing its codebase

Evolution of Programming Languages: All Decades

Some of important the programming languages not covered in the earlier lists include:

AWK (1972/1985), ArnoldC (created with Scala by Lauri Hartikka in 2013), AspectC++, AspectJ, Clojure, eC, Elixir, Esolang: Esoteric programming languages F#, Kotlin, ML, OCaml, Object Pascal, Objective-C, Racket, SIMSCRIPT, SystemC, Wolfram, Xojo

Evolution of Programming Languages: TIOBE Programming Community Index: 2002-2023



Source: `https://www.tiobe.com/tiobe-index/`

Evolution of Programming Languages: TIOBE Programming Community Index: 1987-2023

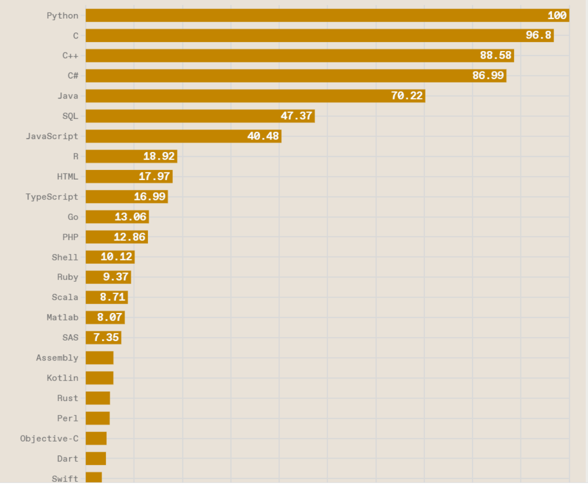
Programming Language	2023	2018	2013	2008	2003	1998	1993	1988
Python	1	4	8	7	12	24	18	-
C	2	2	1	2	2	1	1	1
Java	3	1	2	1	1	16	-	-
C++	4	3	4	4	3	2	2	5
C#	5	5	5	8	9	-	-	-
Visual Basic	6	17	-	-	-	-	-	-
JavaScript	7	7	11	9	8	21	-	-
SQL	8	251	-	-	7	-	-	-
PHP	9	8	6	5	6	-	-	-
Assembly language	10	12	-	-	-	-	-	-
Fortran	19	30	27	21	13	8	3	16
Objective-C	22	16	3	41	55	-	-	-
Ada	26	28	21	19	16	14	5	3
Lisp	29	31	12	17	14	9	6	2
(Visual) Basic	-	-	7	3	5	3	8	6

Source: `https://www.tiobe.com/tiobe-index/`

Ranking: TIOBE Programming Community Index: Aug 2023

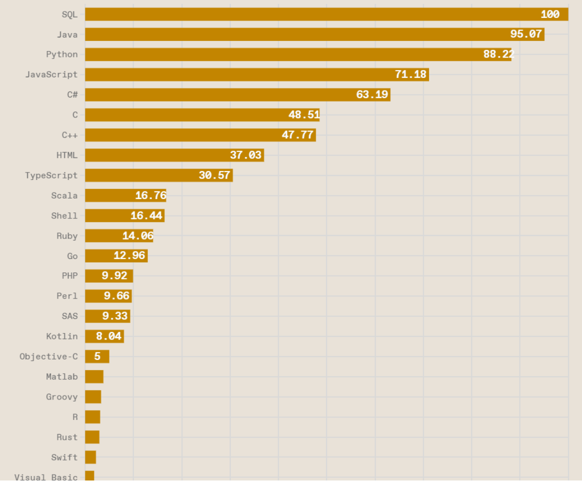
Aug 2023	Aug 2022	Change	Programming Language	Ratings	Change
1	1		Python	13.33%	-2.30%
2	2		C	11.41%	-3.35%
3	4	^	C++	10.63%	+0.49%
4	3	v	Java	10.33%	-2.14%
5	5		C#	7.04%	+1.64%
6	8	^	JavaScript	3.29%	+0.89%
7	6	v	Visual Basic	2.63%	-2.26%
8	9	^	SQL	1.53%	-0.14%
9	7	v	Assembly language	1.34%	-1.41%
10	10		PHP	1.27%	-0.09%

Ranking: IEEE Spectrum - Ranking: 2022



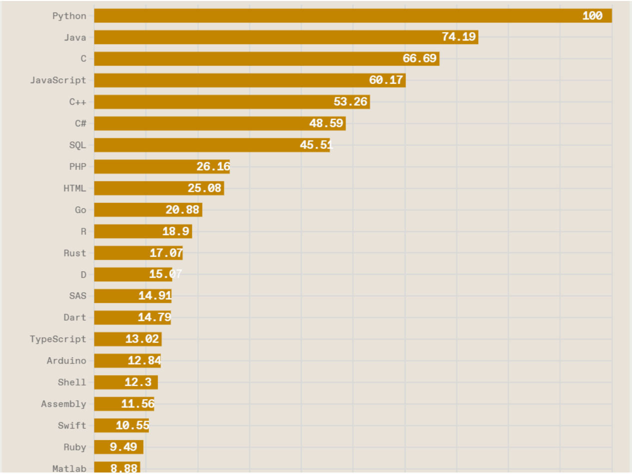
Source: https://spectrum.ieee.org/top-programming-languages-2022

Ranking: IEEE Spectrum - Jobs: 2022



Source: https://spectrum.ieee.org/top-programming-languages-2022

Ranking: IEEE Spectrum - Trending: 2022



Source: https://spectrum.ieee.org/top-programming-languages-2022

Attributes of Programming Languages

- **Widely-Applicable Design and Implementation Techniques**
  - Domain Abstractions ⇒ Programming Language Models / Features
  - Model of Programming Language ⇒ Design and Implementation of Abstraction
- **Domain Specific Languages or Virtual Machines**
  - Mathematica and MATLAB – manipulating mathematical formulas
  - Verilog and VHDL – describing computer hardware circuit designs
  - Cg (C for Graphics) – rendering algorithms that run directly on graphics hardware
  - LaTeX – typesetting, Flex and Bison – translators, e – h/w-s/w co-design etc.
- **Software Models in Languages**
  - Knowledge of OOP (Java) expedites learning of C++ / C# / Python
  - Knowledge of Managed Resources (Java) expedites learning of C# / Python
  - Knowledge of Functional Programming (LISP) expedites learning MapReduce mechanism

Why Undergraduates Should Learn the Principles of Programming Languages?, ACM SIGPLAN Education Board, 2011

Attributes of Programming Languages

- **Choice of the Right Language**
  - Most systems need several languages for different parts of the system
    - HTML for front-end rendering and Javascript for active front-end logic
    - Java for servlet (business layer) and JSP for server-end embedding
    - SQL for data manipulation
  - Nature of Application decides the suitable language
    - Systems Programming ⇒ C++ (very high performance with complex behavior)
    - Embedded Programming ⇒ C (very high performance with frugal dev tools)
    - Application Programming ⇒ Java (medium performance with quick & robust app)
    - Web Programming ⇒ Python (low performance with portability)
    - Machine Learning ⇒ Python (rich collection of 3^{rd} party libraries) with C/C++ backend engine (for efficiency)

- *Why Undergraduates Should Learn the Principles of Programming Languages?*, ACM SIGPLAN Education Board, 2011
- *How to choose a programming language?*, 2019

Understanding Computation

- **Languages**:
  - Fortran, LISP, Algol, Cobol, APL, Simula, SNOBOL, BASIC, PL/1, B, Pascal, Forth, C, Smalltalk, Prolog, ML, Scheme, C++, Ada, Eiffel, Objective-C, Erlang, Perl, Tcl, Haskell, Python, Visual Basic, Ruby, R, Java, Javascript, PHP, D, C#, AspectJ, Visual Basic.NET, AspectC++, Scala, F#, Go
  - SQL
  - MATLAB
  - VHDL, Verilog, SystemC, e

  Unheard of, Aware, Can read programs, Can write programs, Have developed meaningful applications
- **Paradigms**:
  - Imperative, Procedural, Declarative, Object-Oriented, Functional, Generic, Meta, Modular, Concurrent, Logic

  Unknown, Heard of, Vaguely understand, Wholly understand, Is master of

Understanding Computation

- **Computation Model**:
  - Turing Machine, Lambda Calculus, Predicate Calculus, Relational Calculus, Communicating Sequential Processes (CSP)

  Unknown, Heard of, Vaguely understand, Wholly understand, Is master of
- **Application Domains**:
  - System Applications, Business Applications, Web Applications, Embedded Applications, Engineering Applications, Graphics Applications

  Unfamiliar, Remotely familiar, Deeply familiar, Have developed meaningful applications
- **Language – Library Trade-off**:
  - (C / C++, pthread) & Java / C++11; (C++, list) & Python; (C, setjmp) & C++; (C++, SystemC) & e; (C, string) & Python; (Python, ML Libraries) & ?;
- **Compilation Style**:
  - Compiled Language, Interpreted Language, Just-In-Time (JIT) Compilation, Cross Compilation
- ...

Programming Paradigms

- **Imperative** (Fortran, COBOL, Algol, Basic, C, C++): Uses *statements that change a program's state* - consists of *commands for the computer to perform*. Imperative programming focuses on describing how a program operates step by step, rather than on high-level descriptions of its expected results (*declarative paradigm*).
- **Procedural** (Fortran, COBOL, Basic, C): Derived from *imperative*, *based on the procedure call*. Processors provide h/w support for procedural programming through a *stack register and instructions for calling procedures and returning from them*.
- **Declarative** (SQL, HTML, LISP, Prolog, AWK): A style of building the structure and elements of computer programs - that expresses the *logic of a computation* **without** *describing its control flow*. Tries to answer *what* as opposed to *how*.

Imperative (HOW?)	**Declarative (WHAT?)**

<pre>sqrt(n)
    y = n / 2
    repeat
        x = y
        y = (x + n / x) / 2
    until |x - y| < 0.000001
    return y</pre>
	<pre>sqrt(n)
	    return m, where |m * m - n| < 0.000001</pre>

Programming Paradigms

- **Object-Oriented** (Smalltalk, Java, C++, C#, Python, R, PHP, VB.NET, JavaScript, MATLAB): Based on the concept of *objects* containing *data* and *code*: *data* as *fields* (aka *attributes* / *properties*), and *code* as *procedures* (aka *methods*). A *method call* is also known as *message passing* – a message (*method* + *parameters*) passed to the object for dispatch. Often, `this` or `self` is used to refer to the current object.
- **Functional** (Common Lisp, Scheme, Erlang, Haskell. Partial support in C++11, C#, Perl, PHP, Python, Go, Rust, Raku, Scala, Java): Constructed by *applying* and *composing functions*. It is a *declarative paradigm* in which function definitions are *trees of expressions that map values to other values*, rather than a *sequence of imperative statements which update the running state of the program*.
  In this, functions are treated as *first-class citizens*, meaning that they can be *bound to names* (including local identifiers), *passed as arguments*, and *returned from other functions*, just as any other data type can.
  Functional programming has its roots in the *lambda calculus*, a formal system of computation based only on functions.

Programming Paradigms

- **Generic** (*generics* in C#, Delphi, Java, Python, and VB.NET; *parametric polymorphism* in Scala, Julia, and Haskell; *templates* in C++ and D): Algorithms are written in terms of *types to-be-specified-later* that are then *instantiated when needed for specific types provided as parameters*. For example, in C++, `template<typename T>class Stack;` written generically, may be instantiated as `Stack<int>` or `Stack<string>`.
- **Meta-Programming**: Treats other *programs as their data – reads, generates, analyzes or transforms other programs, and even modifies itself while running*. It can *move computations from run-time to compile-time*, to *generate code using compile time computations*, and to *enable self-modifying code*. For example, with templates in C++:

```
template <int N>struct Factorial{ enum { value = N * Factorial<N - 1>::value }; };
template <>struct Factorial<0>{ enum { value = 1 };
};
void foo() {
    int x = Factorial<4>::value; // == 24 // COMPUTED DURING COMPILATION
    int y = Factorial<0>::value; // == 1  // COMPUTED DURING COMPILATION
}
```

A language (like C++) supports *reflection if it its own metalanguage*.

Note: *Generic* and *Meta* programming are closely related and often supported together. Related terms include *Generative* or *Automatic* programming.

Programming Paradigms

- **Modular** (C++20[1], C#, COBOL, Common Lisp, Fortran, Java, Modula, Python, VB.NET): A software design technique that *emphasizes separating the functionality of a program into independent, interchangeable modules*, such that each contains everything necessary to execute only one aspect of the desired functionality.
- **Concurrent** (C++ [std::thread], C#, D, Fortran [coarrays, do concurrent], Haskell, Java [Java—thread class], Python): Several computations are *executed concurrently* – during *overlapping time periods* – instead of *sequentially*. Typically, languages use multi-threading as language feature or with standard library.
- **Logic** (Prolog): Largely based on *formal logic* (typically *Predicate Calculas* or similar). Any program written in a logic programming language is a *set of sentences in logical form, expressing facts and rules about some problem domain*.
  The *declarative* reading of logic programs can be used by a programmer to *verify their correctness*.

[1] C and earlier versions of C++ do not support well-defined modularity

Compilation Styles

- A program in a compiled language translated by a compiler to generate machine code from source code, and then executed.
  - Compile once (static time)
  - Execute many times (run time)

  Pros and Cons of compiled languages include:
  - *Faster than interpreted code*
  - *Good for static binding and static typing*
  - *Makes build-and-test cycles slow*
  - *Platform dependence of the generated binary code*

  Fortran, C, C++, COBOL etc. are complied languages.

Compilation Styles

- A program in an interpreted language is translated step-by-step and executed.
  - Compile and Execute during run time in alternating cycles

  An interpreter generally uses one of the following strategies for program execution:
  - Parse the source code and perform its behavior directly (HTML)
  - Translate source code into some efficient intermediate representation or object code and immediately execute that (Perl, Python)
  - Explicitly execute stored pre-compiled bytecode (aka portable code / p-code) made by a compiler and matched with the interpreter Virtual Machine (Java by JVM, Python by PVM)

  Pros and Cons of interpreted languages include:
  - *Usually are more flexible*
  - *Good for dynamic binding and dynamic typing*
  - *Typically platform independent*
  - *Slower than compiled code (program to do same task in Python takes 10 to 100 time more time compared to C++)*

What in PLDI?

What in PLDI? What do you expect to study?

What in PLDI? What do you expect to study?

First, how languages are implemented

- How to deal with their *syntax*, *semantics*, and *pragmatics* – traditionally known as *Compilation Techniques*
- Covers
    - lexical and syntax analysis
    - memory management
    - runtime behaviour of programs
    - translation to machine code, and
    - optimization
- Background material like
    - fundamentals of automata and formal languages

What in PLDI? What do you expect to study?

Second, how languages are designed

- Design is a complex process involving the *understanding of requirements of the target users* on one hand and the *constraints of the syntax, semantics, and pragmatics* that is necessary for its implementation on the other
- A good designer like Niklaus Wirth of Pascal, Dennis Ritchie of C, Bjarne Stroustrup of C++, or Guido van Rossum of Python works out a best match between the two
- You will be exposed to the parameters a designer has to juggle with including
  - Name scoping and binding;
  - Programming elements – expressions, control flow, procedures, exception, and concurrency;
  - Data types and abstraction - polymorphism;
  - Type Systems;
  - Computing elements – iteration, recursion, and calculus;
  - Paradigm choice; and
  - Domain specific languages

Expected Learning Outcome

Expected Learning Outcome

Expected Learning Outcome

- You will learn to be a designer of your own language and be able to implement it
- You will develop deep understanding for why the programming languages are designed the way they are
- You will learn the basic language translation techniques and a few interesting languages from different paradigms