Exploring Languages with Interpreters and Functional Programming
Chapter 1

H. Conrad Cunningham

29 August 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of August 2018 is a recent version of Firefox from Mozilla.

1 Evolution of Programming Languages

1.1 Chapter Introduction

The goal of this chapter is motivate the study of programming language organization by:

- describing the evolution of computers since the 1940's and its impact upon contemporary programming language design and implementation
- identifying key higher-level programming languages that have emerged since the early 1950's

1.2 Evolving Computer Hardware Affects Programming Languages

To put our study in perspective, let's examine the effect of computing hardware evolution on programming languages by considering a series of questions.

1. *When were the first "modern" computers developed? That is, programmable electronic computers.*

   Although the mathematical roots of computing go back more than a thousand years, it is only with the invention of the programmable electronic digital computer during the World War II era of the 1930s and 1940s that modern computing began to take shape.

   One of the first computers was the ENIAC (Electronic Numerical Integrator and Computer), developed in the mid-1940s at the University of Pennsylvania. When construction was completed in 1946, it cost about $500,000. In today's terms, that is nearly $7,000,000.

   The ENIAC weighed 30 tons, occupied as much space as a small house, and consumed 160 kilowatts of electric power.

   Initially, the ENIAC had no main memory. Instead it had 20 accumulators, each 10 decimal digits wide. Later 100 *words* of core were added.

   Similarly, the ENIAC had no external memory as we know it today. It could read and write stacks of punch cards.

   The ENIAC was not a stored program computer. It was programmed mostly by connecting cables in plugboards. It took several days of careful work to enter one program. The program was only changed every few weeks.

   Aside: Many of the early programmers were women. This is quite a contrast to contemporary programming teams that are mostly male. What happened?

> The ENIAC and most other computers of that era were designed for military purposes, such as calculating firing tables for artillery or breaking codes. As a result, many observers viewed the market for such devices to be quite small. The observers were wrong!

Electronics technology has improved greatly in 70 years. Today, a computer with the capacity of the ENIAC would be smaller than a coin from our pockets, would consume little power, and cost just a few dollars on the mass market.

2. *How have computer systems and their use evolved over the past 70 years?*

    - Contemporary processors are much smaller and faster. They use much less power, cost much less money (when mass produced), and operate much more reliably.

    - Contemporary "main" memories are much larger in capacity, smaller in physical size, and faster in access speed. They also use much less power, cost much less money, and operate much more reliably.

    - The number of processors per machine has increased from one to many. First, channels and other co-processors were added, then multiple CPUs. Today, computer chips for common desktop and mobile applications have several processors—cores—on each chip, plus specialized processors such as graphics processing units (GPUs) for data manipulation and parallel computation. This trend toward multiprocessors will likely continue given that physics dictates limits on how small and fast we can make computer processors; to continue to increase in power means increasing parallelism.

    - Contemporary external storage devices are much larger in capacity, smaller in size, faster in access time, and cost less.

    - The number of computers available per user has increased from much less than one to many more than one.

    - Early systems were often locked into rooms, with few or no direct connections to the external world and just a few kinds of input/output devices. Contemporary systems may be on the user's desktop or in the user's backpack, be connected to the internet, and have many kinds of input/output devices.

    - The range of applications has increased from a few specialized applications (e.g., code-breaking, artillery firing tables) to almost all human activities.

    - The cost of the human staff to program, operate, and support computer systems has probably increased somewhat (in constant dollars).

3. *How have these changes affected programming practice?*

    - In the early days of computing, computers were very expensive and the cost of the human workers to use them relatively less. Today, the opposite holds. So we need to maximize human productivity.

- In the early days of computing, the slow processor speeds and small memory sizes meant that programmers had to control these precious resources to be able to carry out most routine computations. Although we still need to use efficient algorithms and data structures and use good coding practices, programmers can now bring large amounts of computing capacity to bear on most problems. We can use more computing resources to improve productivity to program development and maintenance. The size of the problems we can solve computationally has increased beyond what would be possible manually.

- In the early days of computing, multiple applications and users usually had to share one computer. Today, we can often apply many processors for each user and application if needed. Increasingly, applications must be able to use multiple processors effectively.

- Security on early systems meant keeping the computers in locked rooms and restricting physical access to those rooms. In contemporary networked systems with diverse applications, security has become a much more difficult issue with many aspects.

- Currently, industry can devote considerable hardware and software resources to the development of production software.

The first higher-level programming languages began to appear in the 1950s. IBM released the first compiler for a programming language in 1957–for the scientific programming language Fortran. Although Fortran has evolved considerably during the past 60 years, it is still in use today.

4. *How have the above changes affected programming language design and implementation over the past 60 years?*

    - Contemporary programming languages often use automatic memory allocation and deallocation (e.g., garbage collection) to manage a program's memory. Although programs in these languages may use more memory and processor cycles than hand-optimized programs, they can increase programmer productivity and the security and reliability of the programs. Think Java, C#, and Python versus C and C++.

    - Contemporary programming languages are often implemented using an interpreter instead of a compiler that translates the program to the processor's machine code–or be implemented using a compiler to a virtual machine instruction set (which is itself interpreted on the host processor). Again they use more processor and memory resources to increase programmer productivity and the security and reliability of the programs. Think Java, C#, and Python versus C and C++.

- Contemporary programming languages should make the capabilities of contemporary multicore systems conveniently and safely available to programs and applications. To fail to do so limits the performance and scalability of the application. Think Erlang, Scala, and Clojure versus C, C++, and Java.
- Contemporary programming languages increasingly incorporate declarative features (higher-order functions, recursion, immutable data structures, generators, etc.). These features offer the potential of increasing programming productivity, increasing the security and reliability of programs, and more conveniently and safely providing access to multicore processor capabilities. Think Scala, Clojure, and Java 8 and beyond versus C, C++, and older Java.

As we study programming and programming languages in this and other courses, we need to keep the nature of the contemporary programming scene in mind.

1.3 History of Programming Languages

From the instructor's perspective, key languages and milestones in the history of programming languages include the following.

Note: These descriptions use terminology such as imperative and function that is defined in the subsequent chapters on programming paradigms.

1950's

- Fortran, 1957; imperative; first compiler, math-like language for scientific programming, developed at IBM by John Backus, influenced most subsequent languages, enhanced versions still in use today (first programming language learned by the author in 1974)
- Lisp, 1958; mix of imperative and functional features; innovations include being *homoiconic* (i.e. code and data have same format), extensive use of recursion, syntactic macros, automatic storage management, higher-order functions; related to Church's lambda calculus theory, developed at MIT by John McCarthy, influenced most subsequent languages/research, enhanced versions still in use today
- Algol, 1958, 1960; imperative; innovations included nested block structure, lexical scoping, use of BNF to define syntax, call-by-name parameter passing; developed by an international team from Europe and the USA, influenced most subsequent languages
- COBOL, 1959; imperative; focus on business/accounting programming, decimal arithmetic, record data structures, key designer Grace Hopper, still in use today (third language learned by instructor in late 1975)

1960's

- Simula; 1962, 1967; imperative; original purpose for discrete-event simulation, developed in Norway by Ole-Johan Dahl and Kristen Nygaard, Simula 67 is first object-oriented language (in Scandinavian school of object-oriented languages), Simula 67 influenced subsequent object-oriented languages
- Snobol, 1962; imperative; string processing, patterns as first-class data, backtracking on failure, developed at AT&T Bell Laboratories by David J. Farber, Ralph E. Griswold and Ivan P. Polonsky
- PL/I, 1964; imperative; IBM-designed language to merge scientific (Fortran), business (COBOL), and systems programming (second language learned by the instructor in early 1975)
- BASIC, 1964; imperative; simple language developed for interactive computing in early timesharing and microcomputer environments, developed at Dartmouth College by John G. Kemeny and Thomas E. Kurtz
- Algol 68, 1968; imperative; ambitious and rigorously defined successor to Algol 60; designed by international team, greatly influenced computing science theory and subsequent language designs, but not widely or fully implemented because of its complexity

1970's

- Pascal, 1970; imperative; simplified Algol family language designed by Niklaus Wirth (Switzerland) because of frustration with complexity of Algol 68, structured programming, one-pass compiler, important for teaching in 1980s and 1990s, Pascal-P System virtual machine implemented on many early microcomputers (Pascal used by UM CIS in CS1 and CS2 until 1999)
- Prolog, 1972; logic (relational); first and most widely used logic programming language, originally developed by a team headed by Alain Colmerauer (France), rooted in first-order logic, most modern Prolog implementations based on the Edinburgh dialect (which ran on the Warren Abstract Machine), used extensively for artificial intelligence research in Europe, influenced subsequent logic languages and also Erlang
- C, 1972; imperative; systems programming language for Unix operating system, widely used today; developed by Dennis Ritchie at AT&T Bell Labs, influenced many subsequent languages (first used by the author in 1977)
- Smalltalk, 1972; imperative object-oriented; ground-up object-oriented programming language, message-passing between objects (in American school of object-oriented languages), extensive GUI development environment; developed by Alan Kay and others at Xerox PARC, influenced many subsequent object-oriented languages and user interface approches
- ML, 1973; mostly functional; polymorphic type system on top of Lisp-like language, pioneering statically typed functional programming, algebraic

data types, module system; developed by Robin Milner at the University of Edinburgh as the “meta language” for a theorem-proving system, influenced subsequent functional programming languages, modern dialects include Standard ML (SML), CAML, and OCAML

- Scheme, 1975; mixed functional and imperative; minimalist dialect of Lisp with lexical scoping, tail call optimization, first-class continuations; developed by Guy Steele and Gerald Jay Sussman at MIT, influenced subsequent languages/research
- Icon, 1977; imperative; structured programming successor to Snobol, uses goal-directed execution based on success or failure of expressions; developed by a team led by Ralph Griswold at the University of Arizona

1980’s

- C++, 1980; imperative and object-oriented; C with Simula-like classes; developed by Bjarne Stroustrup (Denmark)
- Ada, 1983; imperative and modular; designed by US DoD-funded committee as standard language for military applications, design led by Jean Ichbiah and a team in France, statically typed, block structured, modular, synchronous message passing, object-oriented extensions in 1995 (instructor studied this language while working in the military aerospace industry 1980-83)
- Eiffel, 1985; imperative object-oriented language; designed with strong emphasis on software engineering concepts such as design by contract and command-query separation; developed by Bertrand Meyer (France)
- Objective C, 1986; imperative object-oriented; C with Smalltalk-like messaging; developed by Brad Cox and Tom Love at Stepstone, selected by Steve Jobs’ NeXT systems, picked up by Apple when NeXT absorbed, key language for MacOS and iOS
- Erlang, 1986; functional and concurrent; message-passing concurrency on functional programming base (actors), fault-tolerant/real-time systems, dynamic typing, virtual machine, originally used in real-time telephone switches; developed by Joe Armstrong, Robert Virding, and Mike Williams at Ericsson (Sweden)
- Self, 1986; imperative prototype-based; dialect of Smalltalk, first prototype-based language, used virtual machine with just-in-time compilation (JIT); developed by David Ungar and Randall Smith while at Xerox PARC, Stanford University, and Sun Microsystems, language influenced JavaScript and Lua, JIT influenced Java HotSpot JIT development
- Perl, 1987; imperative; dynamic programming language originally focused on providing powerful text-processing facilities based around regular expressions; developed by Larry Wall

1990's

- Haskell, 1990; purely functional language; non-strict semantics (i.e. lazy evaluation) and strong static typing; developed by an international committee of functional programming researchers, widely used in research community
- Python, 1991; imperative, originally object-based; dynamically typed, multiparadigm language; developed by Guido van Rossum (Netherlands)
- Ruby, 1993; imperative, object-oriented; dynamically typed, supports reflective/metaprogramming and internal domain-specific languages; developed by Yukihiro "Matz" Matsumoto (Japan), popularized by Ruby on Rails web framework, influenced subsequent languages
- Lua, 1993; imperative; minimalistic language designed for embedding in any environment supporting standard C, dynamic typing, lexical scoping, first-class functions, garbage collection, tail recursion optimization, pervasive table/metatable data structure, facilities for prototype object-oriented programming, coroutines, used as scripting language in games; developed by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes (Brazil)
- R, 1993; imperative; designed for statistical computing and graphics, open-source implementation of the language S; developed by Ross Ihaka and Robert Gentleman (New Zealand), influenced programming in the data science community
- Java, 1995; imperative object-oriented; statically typed, virtual machine, version 8+ has functional programming features (higher-order functions, streams); developed by Sun Microsystems, now Oracle
- JavaScript, 1995 (standardized as ECMAScript); imperative and prototype-based; designed for embedding in web pages, dynamic typing, first-class functions, prototype-based object-oriented programming, internals influenced by Scheme and Self but using a Java-like syntax; developed by Brendan Eich at Netscape in 12 days to meet a deadline, became popular quickly before language design made clean, evolving slowly because of requirement to maintain backward compatibility
- PHP, 1995; imperative; server-side scripting language fordynamic web applications; originally developed by Rasmus Lerdorf (Canada), evolved organically
- OCaml (originally Objective Caml), 1996; mostly functional with imperative and object-oriented features; a dialect of ML that adds object-oriented constructs, focusing on performance and practical use; developed by a team lead by Xavier Leroy (France)

2000's

- C#, 2001; imperative object-oriented programming; statically typed, language runs on Microsoft's Common Language Infrastructure; developed by Microsoft (in response to Sun's Java)
- F#, 2002; OCaml re-envisioned for Microsoft's Common Language Infrastructure (.Net), replaces OCaml's object and module systems with .Net concepts; developed by a team led by Don Syme at Microsoft Research in the UK
- Scala, 2003; hybrid functional and object-oriented language; runs on the Java Virtual Machine and interoperates with Java; developed by Martin Odersky's team at EPFL in Switzerland
- Groovy, 2003; imperative object-oriented; dynamically typed "scripting" language, runs on the Java Virtual Machine; originally proposed by James Strachan
- miniKanren, 2005; relational; a family of relational programming languages, developed by Dan Friedman's team at Indiana University, implemented as an extension to other languages (originally Scheme), most popular current usage probably in Clojure
- Clojure, 2007; mixed functional and imperative; Lisp dialect, runs on Java Virtual Machine, Microsoft Common Language Runtime, and JavaScript platform, emphasis on functional programming, concurrency (e.g., software transactional memory), and immutable data structures; developed by Rich Hickey

2010's

- Idris, 2011 (1.0 release 2017); functional; eagerly evaluated, Haskell-like language with dependent types, incorporating ideas from proof assistants (e.g. Coq), intended for practical programming; developed by Edwin Brady (UK)
- Julia, 2012 (1.0 release 2018); dynamic programming language designed to address high-performance numerical and scientific programming, intended as a modern replacement for MATLAB, Python, and R
- Elixir, 2012 (1.0 release 2014); functional concurrent programming language; dynamic strong typing, metaprogramming, protocols, Erlang actors, runs on Erlang Virtual Machine, influenced by Erlang, Ruby, and Clojure; developed by a team led by Jose Valim (Brazil)
- Elm, 2012 (0.18 release November 2016); simplified, eagerly evaluated Haskell-like functional programming language that compiles to JavaScript, intended primarily for user-interface programming in a browser, supports reactive-style programming; developed by Evan Czaplicki (original version for his senior thesis at Harvard)

- Rust, 2012 (1.0 release 2015); imperative; systems programming language that incorporates contemporary language concepts and focuses on safety and performance, meant to replace C and C++; developed originally at Mozilla Research by Graydon Hoare
- PureScript, 2013 (0.12 release May 2018); mostly functional; an eagerly evaluated language otherwise similar to Haskell, primarily compiles to human-readable JavaScript; originally developed by Phil Freeman
- Swift, 2014; Apple's replacement for Objective C that incorporates contemporary language concepts and focuses on program safety; "Objective C without the C"

The evolution continues!

1.4 What Next?

Computer systems, software development practices, and programming languages have evolved considerably since their beginnings in the 1940s and 1950s. Contemporary languages build on many ideas that first emerged in the early decades of programming languages. But they mix the enduring ideas with a few modern innovations and adapt them for the changing circumstances.

This textbook explores both programming and programming language organization with the following approach:

- emphasize important concepts and techniques that have emerged during the decades since the 1940s
- teach functional and modular programming primarily using the language Haskell, a language that embodies many of the important concepts
- explore the design and implementation of programming languages by building interpreters for simple languages

The next two chapters explore the concept of programming paradigms.

1.5 Exercises

1. Choose some programming language not discussed above and investigate the following issues.

   a. When was the language created?
   b. Who created it?
   c. What programming paradigm(s) does it support? (See Chapters 2 and 3 for more information about programming paradigms.)
   d. What are its distinguishing characterists?
   e. What is its primary target domain or group of users?

   f. What are other interesting aspects of the language, its history, use, etc?

2. Repeat the previous exercise for some other language.

1.6 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work in from my previous materials:

- Evolving Computer Hardware Affects Programming Languages from my notes *Effect of Computing Hardware Evolution on Programming Languages*, which were based on a set of unscripted remarks I made in the Fall 2014 offering of CSci 450, Organization of Programming Languages
- History of Programming Languages from my notes *History of Programming Languages*, which were based on a set of unscripted remarks I made in the Fall 2014 offering of CSci 450, Organization of Programming Languages. Those remarks drew on the following:
  - O'Reilly History of Programming Languages poster [O'Reilly 2004]
  - Wikipedia article on History of Programming Languages [Wikipedia 2018]

In 2017, I continued to develop this material as a part of Chapter 1, Fundamentals, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I reorganized and expanded the previous Fundamentals chapter into four chapters for the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* These are Chapter 1, Evolution of Programming Languages (this chapter); Chapter 2, Programming Paradigms); chapter 3, Object-Based Paradigms; and Chapter 80 (an appendix), Review of Relevant Mathematics.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

1.7 References

[O'Reilly 2004]: O'Reilly Media. History of Programming Languages Poster](http://www.oreillynet.com/pub/a/oreilly/news/languageposter_0504.html), 2004. (30 May 2018: This link seems broken. A temporary link is here.)

[Wikipedia 2018]: Wikipedia. History of Programming Languages, 2018.

1.8 Terms and Concepts

The evolution of computer hardware since the 1940s; impacts upon programming languages and their subsequent evolution.