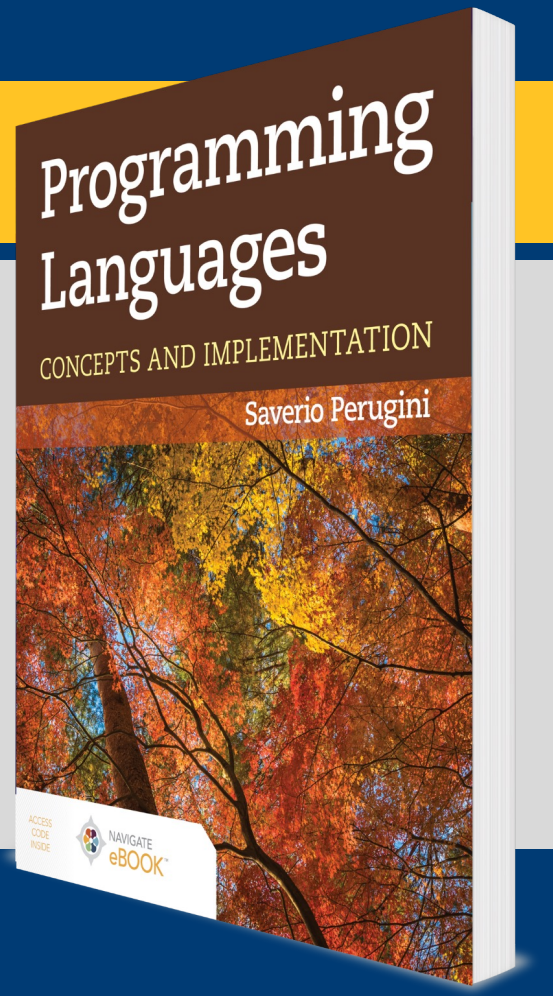
CHAPTER 1

Introduction

Additional Material: Brian Hare, UMKC



Chapter 1: Introduction

Language to the mind is more than light is to the eye.

— Anne Sullivan in *The Miracle Worker* (1956) by William Gibson

Welcome to the study of programming languages. This course of study is about *programming language concepts*—the building blocks of languages.

Outline

- **1.1 Text Objectives**
- 1.2 Chapter Objectives
- 1.3 The World of Programming Languages
- 1.4 Styles of Programming
- 1.5 Factors Influencing Language Development
- 1.6 Recurring Themes in the Study of Languages
- 1.7 What You Will Learn
- 1.8 Learning Outcomes
- 1.9 Thematic Takeaways

1.1 Text Objectives

- Establish an understanding of fundamental and universal language concepts and design/implementation options for them;
- Establish an improved ability to understand new programming languages and an improved background for selecting appropriate languages; and
- Expose readers to alternate styles of programming and exotic ways of affecting computation so to establish
  - an increased capacity for describing computation in a program,
  - a richer toolbox of techniques from which to solve problems, and
  - a more holistic picture of computing.

Outcomes

Given a (new) language, one can:

i. deconstruct it into its essential concepts and determine the implementation options for these concepts;

ii. focus on the big picture (i.e., core concepts/features and options) and not language nuisances or minutia (e.g., syntax);

iii. discern in which contexts (e.g., application domains) it is an appropriate or ideal language of choice; and, thus,

iv. learn to use, assimilate, and harness the strengths of the language quicker.

Outline

- 1.1 Text Objectives
- **1.2 Chapter Objectives**
- 1.3 The World of Programming Languages
- 1.4 Styles of Programming
- 1.5 Factors Influencing Language Development
- 1.6 Recurring Themes in the Study of Languages
- 1.7 What You Will Learn
- 1.8 Learning Outcomes
- 1.9 Thematic Takeaways

1.2 Chapter Objectives

- Establish a foundation for the study of concepts of programming languages.
- Introduce a variety of styles of programming.
- Establish the historical context in which programming languages evolved.
- Establish an understanding of the factors that (historically) influence language design and development and how those factors have changed over time.
- Establish objectives and learning outcomes for the study of programming languages.

Outline

- 1.1 Text Objectives
- 1.2 Chapter Objectives
- **1.3 The World of Programming Languages**
- 1.4 Styles of Programming
- 1.5 Factors Influencing Language Development
- 1.6 Recurring Themes in the Study of Languages
- 1.7 What You Will Learn
- 1.8 Learning Outcomes
- 1.9 Thematic Takeaways

1.3 The World of Programming Languages

- 1.3.1 Fundamental Questions
- 1.3.2 Bindings: Static and Dynamic
- 1.3.3 Programming Language Concepts

1.3.1 Fundamental Questions

- What is a *language* (not necessarily a programming language)?
  - A *language* is simply a medium of communication (e.g., whale song).
- What is a *program*?
  - A *program* is a set of instructions which a computer understands and follows.
- What is a *programming language*?
  - A *programming language* is a system of data-manipulation rules for describing computation.
- What is a *programming language concept*?
  - It is best defined by example. Typically have options.
- What influences language design?
- On which criteria can we evaluate programming languages?

What Influences Language Design?

- How did programming languages evolve and why?
  - Why are there so many languages?
  - If there are so many languages, why are so few commonly used?

- Which factors form the basis for programming languages' evolution:
  - industrial/commercial problems,
  - hardware capabilities/limitations, or
  - the abilities of programmers?

Sapir–Whorf Hypothesis

- In psychology, it is widely believed that one's capacity to think is limited by the language through which one communicates one's thoughts.
- This belief is known as the Sapir–Whorf hypothesis.
- As we will see, some programming idioms cannot be expressed as easily or at all in certain languages as they can in others.
- A universal lexicon has been established for discussing the concepts of languages and we must understand some of these fundamental/universal terms for engaging in this course of study. We encounter these terms throughout this chapter.

1.3.2 Bindings: Static and Dynamic (1 of 2)

Static bindings take place before run-time.

- Bindings refer to the association of one aspect of a program or programming language to another:
  - *language definition time* (e.g., the keyword `int` bound to the meaning of integer)
  - *language implementation time* (e.g., `int` data type bound to a storage size such as four bytes)
  - *compile time* (e.g., identifier `x` bound to an integer variable)
  - *link time* (e.g., `printf` is bound to a definition from a library of routines)
  - *load time* (e.g., variable `x` bound to memory cell at address `0x7cd7`—can happen at run-time as well; consider a variable local to a function)
  - *run-time* (e.g., `x` bound to value `1`)
- These can be broken down further—program load time vs. function call time vs. block entry time, etc

Dynamic bindings take place at run-time.

1.3.2 Bindings: Static and Dynamic (2 of 2)

- Earlier times imply
  - Safety
  - Reliability
  - Predictability, no surprises
  - Efficiency
- Later times imply flexibility.
- Interpreted languages (e.g., Scheme): most bindings are dynamic (i.e., happen at run-time)
- Compiled languages (e.g., C, C++, Fortran): most bindings are static (i.e., happen before run-time)

Static Vis-à-Vis *Dynamic* Bindings

Static	bindings occur	*before*	run-time and are	*fixed*	during run-time.
Dynamic	bindings occur	*at*	run-time and are	*changeable*	during run-time.

1.3.3 Programming Language Concepts

- Language implementation (e.g., interpreted or compiled)
- Parameter passing (e.g., by-value or by-reference)
- Abstraction (e.g., procedural or data)
- Typing (e.g., static or dynamic)
- Scope (e.g., static or dynamic)

Each concept often has multiple options.

Outline

- 1.1 Text Objectives
- 1.2 Chapter Objectives
- 1.3 The World of Programming Languages
- **1.4 Styles of Programming**
- 1.5 Factors Influencing Language Development
- 1.6 Recurring Themes in the Study of Languages
- 1.7 What You Will Learn
- 1.8 Learning Outcomes
- 1.9 Thematic Takeaways

1.4 Styles of Programming

- 1.4.1 Imperative Programming
- 1.4.2 Functional Programming
- 1.4.3 Object-Oriented Programming
- 1.4.4 Logic/Declarative Programming
- 1.4.5 Bottom-up Programming
- 1.4.6 Synthesis: Beyond Paradigms
- 1.4.7 Language Evaluation Criteria
- 1.4.8 Thought Process for Problem Solving

1.4.1 Imperative Programming

- Natural consequence of the von Neumann architecture, which is defined by its uniform representation of both instructions and data in main memory and its use of a fetch-decode-execute cycle
- The primary method of describing/affecting computation is through the execution of a sequence of commands or imperatives which use assignment to modify the value of variables—which are abstractions of memory cells.
  - These *side effects* are expected to influence future computation; e.g., the loop will stop when n reaches 0.
- Instructions are imperative statements that affect, through an assignment operator, the value of variables which are abstractions of memory locations.
- C and Fortran are languages where the primary mode of programming is imperative in nature.

Expressions Vis-à-Vis *Statements*

Expressions	are	evaluated	for	*value*.
Statements	are	executed	for	*side effect*.

Key Terms Discussed in Section 1.4 (1 of 2)

(see Table 1.5)

- *Syntax*: form of language
- *Semantics*: meaning of language
- First-class entity
- Side effect
- Referential transparency

First-Class Entity

- A *first-class entity* is a program object that has privileges that other comparable program entities do not have.
- Traditionally, this has meant that a first-class entity can be
  - stored (e.g., in a variable or data structure),
  - passed as an argument, and
  - returned as a value.

Key Terms Discussed in Section 1.4 (2 of 2)

- *Side effect*: a modification of a parameter to a function or operator, or an entity in the external environment (e.g., change to a global variable or performing I/O—which changes the nature of the input stream/file).

- *Referential transparency*: expressions and languages are said to be *referentially transparent* (i.e., independent of evaluation order) if the same arguments/operands to a function/operator yield the same output irrespective of the context/environment in which the expression applying the function/operator is evaluated.
  - A referentially transparent function that has no side effects is said to be *pure.*

1.4.2 Functional Programming

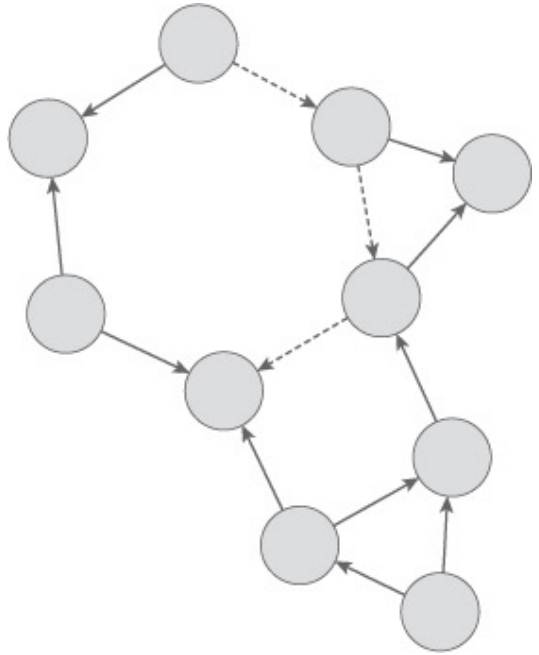
- Programmer describes computation primarily by calling a series of functions, which cascade a set of return values to each other.
- Largely without variables, assignment, and iteration
- It is possible to program without variables and assignment statements
- Grounded in λ-*calculus*—a mathematical theory of functions
- Functions treated as *first-class entities*
- Pioneered in the Lisp programming language, designed by John McCarthy in 1958 at MIT (McCarthy 1960)
- Scheme and Common Lisp are dialects of Lisp.
- Scheme is an ideal vehicle for exploring language semantics and implementing language concepts.
- ML, Haskell, and F# also primarily support a functional style of programming.

1.4.3 Object-Oriented Programming (OOP) (1 of 2)

- A solution to a problem is expressed as a collection of objects communicating by passing messages messages to each other
- Objects are program entities that encapsulate data and functionality.
- OOP unifies the concepts of data and procedural abstraction through the constructs of classes and objects.
- Pioneered in the Smalltalk programming language, designed by Alan Kay and colleagues in the early 1970s at Xerox PARC
- Python, Java, C#, C++, and Dylan support OOP.

1.4.3 Object-Oriented Programming (OOP) (2 of 2)

Conceptual depiction of a set of objects communicating by passing messages to each other to collaboratively solve a problem



1.4.4 Logic/Declarative Programming

- Describe *what* is to be computed, not *how* to compute it.
  - UNIX regular expressions
  - SQL queries

- Grounded in *first-order predicate calculus*—a formal system of symbolic logic
- Languages supporting declarative programming are sometimes called *very-high-level languages.*

- Prolog, Mercury, and CLIPS support a logic/declarative style of programming.

Purity in Programming Languages

Style of Programming	Purity Indicates	(Near-)Pure Language(s)
Functional programming	No provision for side effect	Haskell
Logic/declarative programming	No provision for control	Mercury
Object-oriented programming	No provision for performing computation without message passing; all program entities are objects	Smalltalk, Ruby, and CLOS-based languages

Practical/Conceptual/Theoretical Basis for Dominant Styles of Programming

Style of Programming	Practical/Conceptual/Theoretical Foundation	Defining/Pioneering Language
Imperative programming	von Neumann architecture	Fortran
Functional programming	λ-calculus; Lisp machine	Lisp
Logic/declarative programming	First-order Predicate Calculus; Warren Abstract Machine	Prolog
Object-oriented programming	Lisp, biological cells, individual computers on a network	Smalltalk

1.4.5 Bottom-up Programming

- A compelling style of programming is to use a programming language not to develop a solution to a problem, but rather to build a language specifically tailored to solving a family of problems for which the problem at hand is an instance.
- The programmer subsequently uses this language to write a program to solve the problem of interest.
  - FORTH, Lua are prime examples: In both cases, the core language is small (< 25 commands), but can be used to extend the language as desired
  - Racket is the successor to Scheme, and has added features specifically for defining DSLs
- This process is called *bottom-up programming* and the resulting language is typically either an *embedded* or a *domain-specific language*.
  - Domain-specific languages (DSLs) typically don't worry about being useful general-purpose languages; rather, they focus on doing one narrow thing particularly well.
  - Web pages: HTML/CSS. Document markup: LaTeX, PostScript. Docker or Jenkins scripting languages.

1.4.7 Language Evaluation Criteria

- Concepts of languages provide the basis from which to foster comparative analysis.
- Languages differ on the implementation options each employs for these concepts.
  - e.g., Python is a dynamically typed language and Go is a statically typed language.
  - See Figure 1.2
- The construction of an interpreter for a computer language operationalizes (or instantiates) the design options or semantics for the pertinent concepts. (*Operational semantics* supplies the meaning of a computer program through its implementation.)
- One objective of this text is to provide the framework in which to study, compare, and select from the available programming languages.
- *Non-functional requirements*—on which to evaluate languages; traditionally, these criteria include:
  - readability,
  - writability,
  - reliability (i.e., safety), and
  - cost (of execution or development or both?)
- How are readability and writability related?
- Others?

1.4.7 Language Evaluation Criteria – Full List

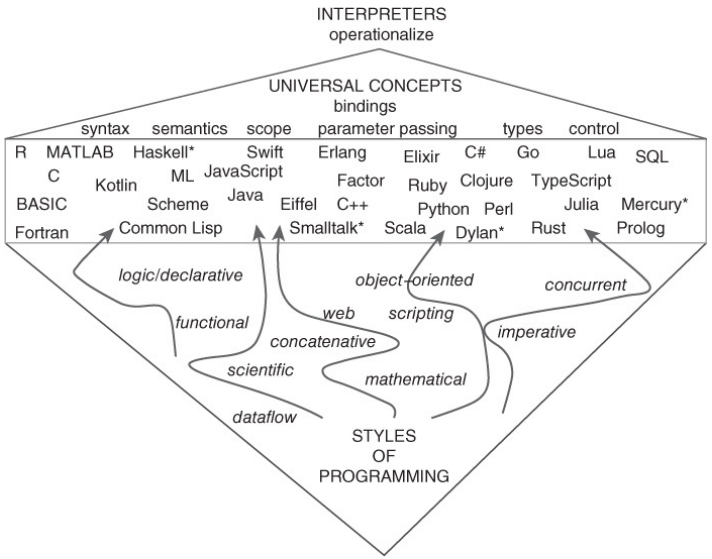
- Readability – Can a non-expert in the language read the source code & understand what it's doing?
- Writeability – How easy is it to produce a unit (line, block, subroutine) of code? How 'finicky' is the language? How elaborate are the syntactical requirements?
- Reliability – If something goes wrong in the program, do we have a way of detecting it? Of recovering from it? Of avoiding having things go wrong in the first place?
- Maintainability – Can the code be modified, bug-fixes applied, refactoring carried out, relatively easily? Or is rewriting from the beginning the only real option?
- Expressivity – How easy is it to carry out a task, to "say what we mean" in code?
- Efficiency – How quickly does the interpreted or compiled code do what it's supposed to do? Is memory used and managed effectively? How much overhead is added by the interpretation / compilation process?

1.4.7 Language Evaluation Criteria – Full List

- Support for abstraction – Can we work at higher levels of abstraction easily? Can we go from talking about properties of one entity, to talking about all entities of that type?
- Orthogonality – How do the components of the language go together? Can we arrange them in any order and still have a meaningful program, and do all components mean the same thing? The more “this can only appear after that” rules there are, the less orthogonal a language is
- Ease of learning – Can a language be learned reasonably quickly?
- Cost – Are compilers and tools expensive, or are there reasonably priced or open-source options?
- Standardization – Are there dozens of slightly-different versions for different platforms, or one standard version that everyone uses?
- Development tools – Are there good debuggers, testing frameworks, profilers, etc., available at reasonable cost and ease of use?

Languages Involve a Core Set of Universal Concepts

Despite of their support for a variety programming styles, all computer languages involve a core set of universal concepts.



1.4.8 Thought Process for Problem Solving (1 of 2)

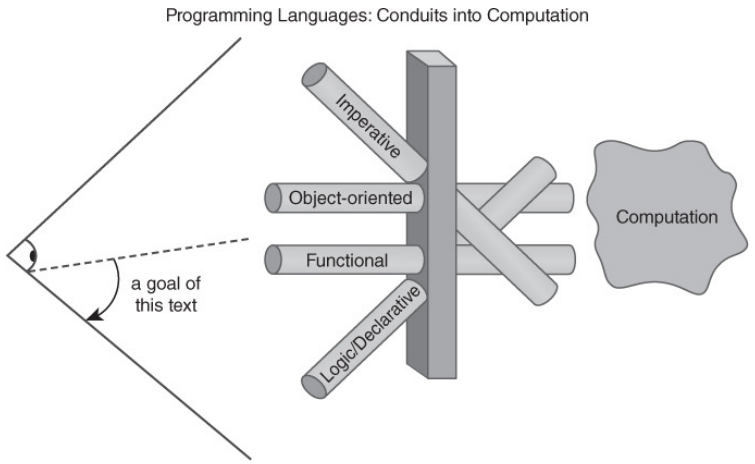
- While most languages now support multiple styles of programming, use of the styles themselves involves a shift in one's problem-solving thought process.
- An advantageous outcome of learning to solve problems using an unfamiliar style of programming (e.g., functional, declarative) is that it involves a fundamental shift in one's thought process toward problem decomposition and solving.
- "A programming language that doesn't affect the way you think about programming isn't worth learning." -- Alan Perlis, founding faculty member of Carnegie Institute of Technology (now CMU), first Turing Award winner.

1.4.8 Thought Process for Problem Solving (2 of 2)

- Indeed, a covert goal of this text or side effect of this course of study is to broaden the reader's understanding of computation by developing additional avenues through which to both experience and describe/effect computation in a computer program (Figure 1.3).
- Similarly, an understanding of both Lisp and the linguistic ideas central to it—and, more generally, the concepts of languages—will help you more easily learn new programming languages and make you a better programmer in your language of choice.

Programming Languages Are Conduits into Computation

Programming languages and the styles of programming therein are conduits into computation.



Outline

- 1.1 Text Objectives
- 1.2 Chapter Objectives
- 1.3 The World of Programming Languages
- 1.4 Styles of Programming
- **1.5 Factors Influencing Language Development**
- 1.6 Recurring Themes in the Study of Languages
- 1.7 What You Will Learn
- 1.8 Learning Outcomes
- 1.9 Thematic Takeaways

1.5 Factors Influencing Language Development

- Why did programming languages evolve as they did?
- Surprisingly enough, programming languages did not historically evolve based on the abilities of programmers (Weinberg 1988).
- Historically, computer architecture influenced programming language design and implementation.
- Use of the *von Neumann architecture* inspired the design of many early programming languages that dovetailed with that model.

Static Vis-à-Vis Dynamic Languages

Dynamic Languages

- On the one hand, the need to develop applications with ever-evolving requirements rapidly has attracted attention to the speed of development as a more prominent criterion in the design of programming languages and has continued to nourish the development of languages adopting more dynamic bindings (e.g., Python).
- The more dynamic bindings a language supports, the fewer the number of commitments the programmer must make during program development.
- There has been an incremental and ongoing shift toward support for more dynamic bindings in programming languages to enable the creation of malleable programs.

Static Vis-à-Vis Dynamic Languages

Static Languages

- On the other hand, static type systems support program evolution by automatically identifying the parts of a program affected by a change in a data structure, for example (Wright 2010).
- Moreover, program safety and security are new applications of static bindings in languages (e.g., development of TypeScript as JavaScript with a safe type system).
- Figure 1.4 depicts the (historical) development of contemporary languages with dynamic bindings and languages with static bindings—both supporting multiple styles of programming.
- Languages reconciling the need for both safety and flexibility are also starting to emerge (e.g., Hack and Dart).
- Figure 1.5 summarizes the factors influencing language design discussed here.

Evolution of Programming Languages

Figure 1.4 Evolution of programming languages emphasizing multiple shifts in language development across a time axis.
(Time axis not drawn to scale.)

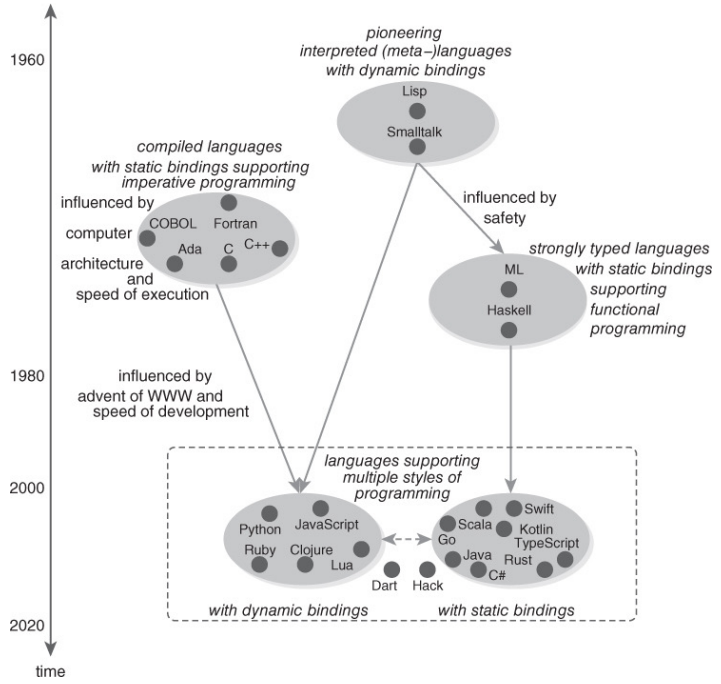
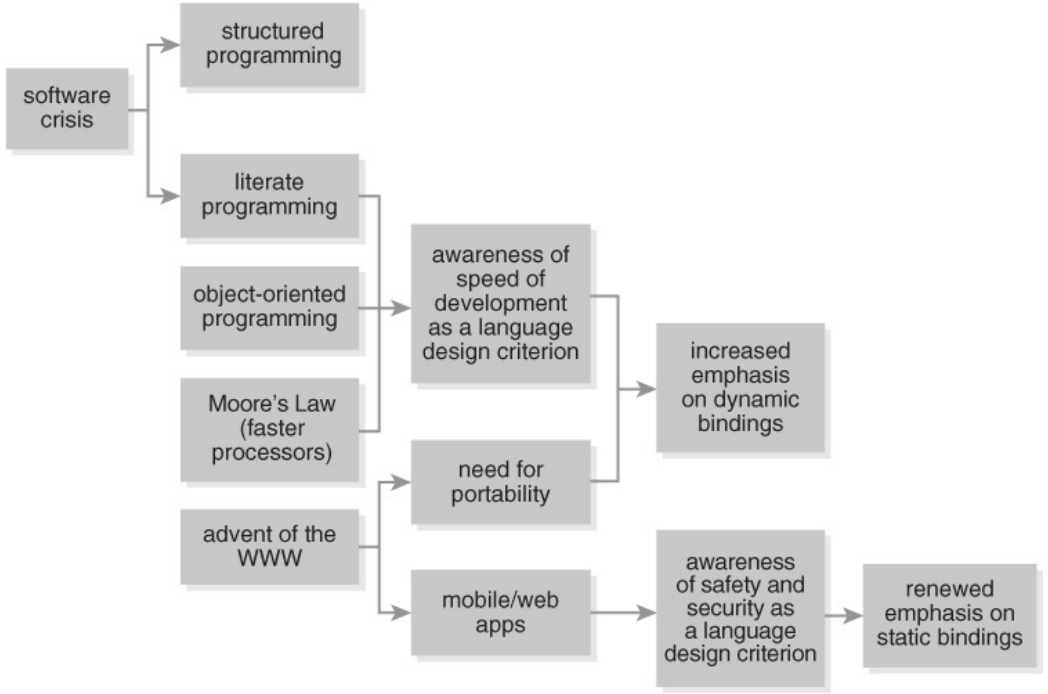


Figure 1.5 Factors Influencing Language Design



Key: arrows indicate "led to."

Outline

- 1.1 Text Objectives
- 1.2 Chapter Objectives
- 1.3 The World of Programming Languages
- 1.4 Styles of Programming
- 1.5 Factors Influencing Language Development
- **1.6 Recurring Themes in the Study of Languages**
- 1.7 What You Will Learn
- 1.8 Learning Outcomes
- 1.9 Thematic Takeaways

1.6 Recurring Themes in the Study of Languages (1 of 3)

- A core set of language concepts are universal to all programming languages.
- There are a variety of options for language concepts, and individual languages differ on the design and implementation options for (some of) these concepts.
- The concept of binding is fundamental to many other concepts in programming languages.
- Most issues in the design, implementation, and use of programming languages involve important practical trade-offs.
- Side effects are often the underlying culprit of many programming perils.

1.6 Recurring Themes in the Study of Languages (2 of 3)

- Like natural languages, programming languages have exceptions in how a language principle applies to entities in the language. Some languages are consistent (e.g., in Smalltalk everything is an object; Scheme uses prefix notation for built-in and user-defined functions and operators), while others are inconsistent (e.g., Java uses pass-by-value for primitives, but seemingly uses pass-by-reference for objects). There are fewer nuances to learn in consistent languages.
- There is a relationship between languages and the capacity to express ideas about computation.
- Languages are built on top of languages.

1.6 Recurring Themes in the Study of Languages (3 of 3)

- Languages evolve: The specific needs of application domains and development models influence language design and implementation options, and vice versa (e.g., speed of execution is less important as a design goal than it once was).
- Programming is an art (Knuth 1974a), and programs are works of art. The goal is not just to produce a functional solution to a problem, but to create a beautiful and reconfigurable program. Consider that architects seek to design not only structurally sound buildings, but buildings and environments that are aesthetically pleasing and foster social interactions.
- "Great software, likewise, requires a fanatical devotion to beauty" (Graham 2004b, p. 29).
- Problem solving and subsequent programming implementation require pattern recognition and application, respectively.

Outline

- 1.1 Text Objectives
- 1.2 Chapter Objectives
- 1.3 The World of Programming Languages
- 1.4 Styles of Programming
- 1.5 Factors Influencing Language Development
- 1.6 Recurring Themes in the Study of Languages
- **1.7 What You Will Learn**
- 1.8 Learning Outcomes
- 1.9 Thematic Takeaways

1.7 What You Will Learn

- Fundamental and universal concepts of programming languages (e.g., scope and parameter passing) and the options available for them (e.g., lexical scoping, pass-by-name/lazy evaluation), especially from an implementation-oriented perspective
- Language definition and description methods (e.g., grammars)
- How to design and implement language interpreters, and implementation strategies (e.g., inductive data types, data abstraction and representation)
- Different styles of programming (e.g., functional, declarative, concurrent programming) and how to program using languages supporting those styles (e.g., Python, Scheme, ML, Haskell, and Prolog)
- Types and type systems (through Python, ML, and Haskell)
- Other concepts of programming languages (e.g., type inference, higher-order functions, currying)
- Control abstraction, including first-class continuations

Outline

- 1.1 Text Objectives
- 1.2 Chapter Objectives
- 1.3 The World of Programming Languages
- 1.4 Styles of Programming
- 1.5 Factors Influencing Language Development
- 1.6 Recurring Themes in the Study of Languages
- 1.7 What You Will Learn
- **1.8 Learning Outcomes**
- 1.9 Thematic Takeaways

1.8 Learning Outcomes (1 of 2)

Satisfying the text objectives outlined in Section 1.1 will lead to the following learning outcomes:

- An understanding of fundamental and universal language concepts, and design/implementation options for them
- An ability to deconstruct a language into its essential concepts and determine the implementation options for these concepts
- An ability to focus on the big picture (i.e., core concepts/features and options) and not the minutia (e.g., syntax)
- An ability to (more rapidly) understand (new or unfamiliar) programming languages

1.8 Learning Outcomes (2 of 2)

- An improved background and richer context for discerning appropriate languages for particular programming problems or application domains
- An understanding of and experience with a variety of programming styles or, in other words, an increased capacity to describe computational ideas
- A larger and richer arsenal of programming techniques to bring to bear upon problem-solving and programming tasks, which will make you a better programmer, in any language
- An increased ability to design and implement new languages
- An improved understanding of the (historical) context in which languages exist and evolve
- A more holistic view of computer science

Outline

- 1.1 Text Objectives
- 1.2 Chapter Objectives
- 1.3 The World of Programming Languages
- 1.4 Styles of Programming
- 1.5 Factors Influencing Language Development
- 1.6 Recurring Themes in the Study of Languages
- 1.7 What You Will Learn
- 1.8 Learning Outcomes
- **1.9 Thematic Takeaways**

1.9 Thematic Takeaways (1 of 2)

- This course of study is about concepts of programming languages.
- There is a universal lexicon for discussing the concepts of languages and for, more generally, engaging in this course of study, including the terms binding, side effect, and first-class entity.
- Programming languages differ in their design and implementation options for supporting a variety of concepts from a host of programming styles, including imperative, functional, object-oriented, and logic/declarative programming.
- The support for multiple styles of programming in a single language provides programmers with a richer palette in that language for expressing ideas about computation.
- Programming languages and the various styles of programming used therein are conduits into computation (Figure 1.3).

1.9 Thematic Takeaways (2 of 2)

- Within the context of their support for a variety of programming styles, all languages involve a core set of universal concepts that are operationalized through an interpreter and provide a basis for (comparative) evaluation (Figure 1.2).
- The diversity of design and implementation options across programming languages provides fertile ground for comparative language analysis.
- A variety of factors influence the design and development of programming languages, including (historically) computer architecture, abilities of programmers, and development methodologies.
- The evolution of programming languages bifurcated into languages involving primarily static binding and those involving primarily dynamic bindings (Figure 1.4).