PART I
FUNDAMENTALS

Chapter 1

Introduction

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

— Anne Sullivan in William Gibson's *The Miracle Worker* (1959)

That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted.

— George Boole (1854)

A language that doesn't affect the way you think about programming, is not worth knowing.

— Alan Perlis (1982)

"I don't see how he can ever finish, if he doesn't begin."

— Alice, in *Alice's Adventures in Wonderland* (1895) by Lewis Carroll

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

1.1 Text Objectives

The objectives of this text are to:

- Establish an understanding of fundamental and universal language concepts and design/implementation options for them.
- Improve readers' ability to understand new programming languages and enhance their background for selecting appropriate languages.
- Expose readers to alternative styles of programming and exotic ways of performing 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.

Since language concepts are the building blocks from which all languages are constructed and organized, an understanding of the concepts implies that, given a (new) language, one can:

- Deconstruct it into its essential concepts and determine the implementation options for these concepts.
- Focus on the big picture (i.e., core concepts/features and options) and not language nuisances or minutia (e.g., syntax).
- Discern in which contexts (e.g., application domains) it is an appropriate or ideal language of choice.
- In turn, learn to use, assimilate, and harness the strengths of the language more quickly.

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 influence language design and development and how those factors have changed over time.
- Establish objectives and learning outcomes for the study of programming languages.

1.3 The World of Programming Languages

1.3.1 Fundamental Questions

This text is about programming language concepts. In preparation for a study of language concepts, we must examine some fundamental questions:

- What is a language (not necessarily a programming language)? A *language* is simply a medium of communication (e.g., a whale's song).
- What is a program? A *program* is a set of instructions that 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. Perhaps the language concept that resonates most keenly with readers at this point in their formal study of computer science is that of parameter passing. Some languages implement parameter passing with pass-by-value, while others use pass-by-reference, and still other languages implement both mechanisms. In a general sense, a language concept is typically a universal principle of languages, for which individual languages differ in their implementation approach to that principle. The way a concept is implemented in a particular language helps define the semantics of the

language. In this text, we will demonstrate a variety of language concepts and implement some of them.

- What influences language design? How did programming languages evolve and why? Which factors form the basis for programming languages' evolution: industrial/commercial problems, hardware capabilities/limitations, or the abilities of programmers?

Since a programming language is a system for describing computation, a natural question arises: What exactly is the computation that a programming language describes? While this question is studied formally in a course on computability theory, some brief remarks will be helpful here. The notion of mechanical *computation* (or an algorithm) is formally defined by the abstract mathematical model of a computer called a *Turing machine*. A Turing machine is a universal computing model that establishes the notion of what is computable. A programming language is referred to as *Turing-complete* if it can describe any computational process that can be described by a Turing machine. The notion of *Turing-completeness* is a way to establish the power of a programming language in describing computation: If the language can describe all of the computations that a Turing machine can carry out, then the language is Turing-complete. Support for sequential execution of both *variable assignment* and *conditional-branching* statements (e.g., `if` and `while`, and `if` and `goto`) is sufficient to describe computation that a Turing machine can perform. Thus, a programming language with those facilities is considered Turing-complete.

Most, but not all, programming languages are Turing-complete. In consequence, the more interesting and relevant question as it relates to this course of study is not what is or is not formally computable through use of a particular language, but rather which types of programming abstractions are or are not available in the language for describing computation in a more practical sense. Larry Wall, who developed Perl, said:

> Computer languages differ not so much in what they make possible, but in what they make easy. (Christiansen, Foy, Wall, and Orwant, 2012, p. xxiii)

"Languages are *abstractions*: ways of seeing or organizing the world according to certain patterns, so that a task becomes easier to carry out. . . . [For instance, a] loop is an abstraction: a reusable pattern" (Krishnamurthi 2003, p. 315). Furthermore, programming languages affect (or should affect) the way we think about describing ideas about computation. Alan Perlis (1982) said: "A language that doesn't affect the way you think about programming, is not worth knowing" (Epigraph 19, p. 8). 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*. George Boole (1854) said: "Language is an instrument of human reason, and not merely a medium for the expression of thought[; it] is a truth generally admitted" (p. 24). 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

Bindings are central to the study of programming languages. *Bindings* refer to the association of one aspect of a program or programming language with another. For instance, in C the reserved word `int` is a mnemonic bound to mean "integer" by the language designer. A programmer who declares `x` to be of type `int` in a program (i.e., `int x;`) binds the identifier `x` to be of type integer. A program containing the statement `x = 1;` binds the value `1` to the variable represented by the identifier `x`, and `1` is referred to as the *denotation* of `x`. Bindings happen at particular times, called *binding times*. Six progressive binding times are identified in the study of programming languages:

1. *Language definition time* (e.g., the keyword `int` bound to the meaning of integer)
2. *Language implementation time* (e.g., `int` data type bound to a storage size such as four bytes)
3. *Compile time* (e.g., identifier `x` bound to an integer variable)
4. *Link time* (e.g., `printf` is bound to a definition from a library of routines)
5. *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)

↑ *static bindings* ↑

- -

↓ *dynamic bindings* ↓

6. *Run-time* (e.g., `x` bound to value 1)

Language definition time involves defining the *syntax* (i.e., form) and *semantics* (i.e., meaning) of a programming language. (Language definition and description methods are the primary topic of Chapter 2.) Language implementation time is the time at which a compiler or interpreter for the language is built. (Building language interpreters is the focus of Chapters 10–12.) At this time some of the semantics of the implemented language are bound/defined as well. The examples given in the preceding list are not always performed at the particular time in which they are classified. For instance, binding the variable `x` to the memory cell at address `0x7cd7` can also happen at run-time in cases where `x` is a variable local to a function or block.

The aforementioned bindings are often broadly categorized as either static or dynamic (Table 1.1). A *static* binding happens before run-time (usually at compile time) and often remains unchangeable during run-time. A *dynamic* binding happens at run-time and can be changed at run-time. Dynamic binding is also

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

Table 1.1 Static Vis-à-Vis Dynamic Bindings

referred to as *late binding*. It is helpful to think of an analogy to human beings. Our date of birth is bound statically at birth and cannot change throughout our life. Our height, in contrast, is (re-)bound dynamically—it changes throughout our life. Earlier times imply safety, reliability, predictability (i.e., no surprises at run-time), and efficiency. Later times imply flexibility. In interpreted languages, such as Scheme, most bindings are dynamic. Conversely, most bindings are static in compiled languages such as C, C++, and Fortran. Given the central role of bindings in the study of programming languages, we examine both the types of bindings (i.e., what is being bound to what) as well as the binding times involved in the language concepts we encounter in our progression through this text, particularly in Chapter 6.

1.3.3 Programming Language Concepts

Let us demonstrate some language concepts by example, and observe that they often involve options. You may recognize some of the following language concepts (though you may not have thought of them as language concepts) from your study of computing:

- 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)

We can draw an analogy between language concepts and automobile concepts. Automobile concepts include make (e.g., Honda or Toyota), model (e.g., Accord or Camry), engine type (e.g., gasoline, diesel, hybrid, or electric), transmission type (e.g., manual or automatic), drivetrain (e.g., front wheel, rear wheel, or all wheel), and options (e.g., rear camera, sensors, Bluetooth, satellite radio, and GPS navigation). With certain concepts of languages, their options are so ingrained into the fiber of computing that we rarely ever consider alternative options. For instance, most languages provide facilities for procedural and data abstraction. However, most languages do not provide (sophisticated) facilities for control abstraction (i.e., developing new control structures). The traditional `if`, `while`, and `for` are not the only control constructs for programming. Although some languages, including Go and C++, provide a `goto` statement for transfer of control, a `goto` statement is not sufficiently powerful to design new control structures. (Control abstraction is the topic of Chapter 13.)

The options for language concepts are rarely binary or discretely defined. For instance, multiple types of parameter passing are possible. The options available and the granularity of those options often vary from language to language and depend on factors such as the application domain targeted by the language and the particular problem to be solved. Some concepts, including control abstraction, are omitted in certain languages.

Beyond these fundamental/universal language concepts, an exploration of a variety of programming styles and language support for these styles leads to a host of other important principles of programming languages and language constructs/abstractions (e.g., closures, higher-order functions, currying, and first-class continuations).

1.4 Styles of Programming

We use the term "*styles* of programming" rather than perhaps the more common/conventional, but antiquated, term "*paradigm* of programming." See Section 1.4.6 for an explanation.

1.4.1 Imperative Programming

The primary method of describing/affecting computation in an *imperative style of programming* is through the execution of a sequence of commands or *imperatives* that use assignment to modify the values of variables—which are themselves abstractions of memory cells. In C and Fortran, for example, the primary mode of programming is imperative in nature. The imperative style of programming is a natural consequence of basing a computer on 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. (While the *Turing machine* is an abstract model that captures the notion of mechanical computation, the von Neumann architecture is a practical design model for actual computers. The concept of a Turing machine was developed in 1935–1937 by Alan Turing and published in 1937. The von Neumann architecture was articulated by John von Neumann in 1945.)

The main mechanism used to effect computation in the imperative style is the assignment operator. A discussion of the difference between *statements* and *expressions* in programs helps illustrate alternative ways to perform such computation. Expressions are evaluated for their value, which is returned to the next encompassing expression. For instance, the subexpression `(3*4)` in the expression `2+(3*4)` returns the integer `12`, which becomes the second operand to the addition operator. In contrast, the statement `i = i+1` has no return value.[1] After that statement is executed, evaluation proceeds with the following statement (i.e., sequential execution). *Expressions* are evaluated for *values* while *statements* are executed for *side effect* (Table 1.2). A *side effect* is a modification of a parameter to a function or operator, or an entity in the external environment (e.g., a change to a global variable or performing I/O, which changes the nature of the input

1. In C, such statements return the value of `i` after the assignment takes place.

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

Table 1.2 Expressions Vis-à-Vis Statements

stream/file). The primary way to perform computation in an imperative style of programming is through side effect. The assignment statement inherently involves a side effect. For instance, the execution of statement `x = 1` changes the first parameter (i.e., `x`) to the = assignment operator to 1. I/O also inherently involves a side effect. For instance, consider the following Python program:

```
x = int(input())
print (x + x)
```

If the input stream contains the integer `1` followed by the integer `2`, readers accustomed to imperative programming might predict the output of this program to be `2` because the `input` function executes only once, reads the value `1`,[2] and stores it in the variable `x`. However, one might interpret the line `print (x + x)` as `print (int(input()) + int(input()))`, since `x` stands for `int(input())`. With this interpretation, one might predict the output of the program to be `3`, where the first and second invocations to `input()` read `1` and `2`, respectively. While mathematics involves binding (e.g., *let* $x = 1$ *in* ...), mathematics does not involve assignment.[3]

The aforementioned interpretation of the statement `print (x + x)` as `print (int(input()) + int(input()))` might seem unnatural to most readers. For those readers who are largely familiar with the imperative style of programming, describing computation through side effect is so fundamental to and ingrained into their view of programming and so unconsciously integrated into their programming activities that the prior interpretation is viewed as entirely foreign. However, that interpretation might seem entirely natural to a mathematician or someone who has no experience with programming.

Side effects also make a program difficult to understand. For instance, consider the following Python program:

```
def f():
   global x
   x = 2
   return x
# main program
x = 1
print (x + f())
```

Function `f` has a side effect: After `f` is called, the `global` variable `x` has value `2`, which is different than the value it had prior to the call to `f`. As a result, the output of this program depends on the order in which the operands to the

2. The Python `int` function used here converts the string read with the `input` function to an integer.
3. The common programming idiom `x=x+1` can be confusing to nonprogrammers because it appears to convey that two entities are equal that are clearly not equal.

addition operator are evaluated. However, the result of a commutative operation, like addition, is not dependent on the order in which its operands are evaluated (i.e., 1 + 2 = 2 + 1 = 3). If the operands are evaluated from left to right (i.e., Python semantics), the output of this program is `3`. If the operands are evaluated from right to left, the output is `4`.

The concept of side-effect is closely related to, yet distinct from, the concept of *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. The function Python `f` given previously has a side effect and the expression `x + f()` is not referential transparent. The absence of side effects is not sufficient to guarantee referential transparency (Conceptual Exercise 1.8).

Since the von Neumann architecture gave rise to an imperative mode of programming, most early programming languages (e.g., Fortran and COBOL), save for Lisp, supported primarily that style of programming. Moreover, programming languages evolved based on the von Neumann model. However, the von Neumann architecture has certain inherent limitations. Since a processor can execute program instructions much faster than program instructions and program data can be moved from main memory to the processor, I/O between the processor and memory—referred to as the *von Neumann bottleneck*—affects the speed of program execution. Moreover, the reality that computation must be described as a sequence of instructions operating on a single piece of data that is central to the von Neumann architecture creates another limitation. The von Neumann architecture is not a natural model for other non-imperative styles of describing computation. For instance, recursion, nondeterministic computation, and parallel computation do not align with the von Neumann model.[4,5]

Imperative programming is programming by side effect; functional programming is programming without side effect. *Functional programming* involves describing and performing computation by calling functions that return values. Programmers from an imperative background may find it challenging to conceive of writing a program without variables and assignment statements. Not only is such a mode of programming possible, but it leads to a compelling higher-order style of program construction, where functions accept other functions as arguments and can return a function as a return value. As a result, a program is conceived as a collection of highly general, abstract, and reusable functions that build other functions, which collectively solve the problem at hand.

4. Ironically, John Backus, the recipient of the 1977 ACM A. M. Turing Award for contributions to the primarily imperative programming language Fortran, titled his Turing Award paper "Can Programming Be Liberated from the von Neumann Style?: A Functional Style and Its Algebra of Programs." This paper introduced the functional programming language FP through which Backus (1978) cast his argument. While FP was never fully embraced by the industrial programming community, it ignited both debate and interest in functional programming and subsequently influenced multiple languages supporting a functional style of programming (*Interview with Simon Peyton-Jones* 2017).

5. Computers have been designed for these inherently non-imperative styles as well (e.g., Lisp machine and Warren Abstract Machine).

1.4.2 Functional Programming

While the essential element in imperative programming is the assignment statement, the essential ingredient in functional programming is the function. Functions in languages supporting a functional style of programming are first-class entities. In programming languages, a *first-class entity* is a program object that has privileges that other comparable program entities do not have.[6] The designation of a language entity as first-class generally means that the entity can be expressed in the source code of the program and has a value at run-time that can be manipulated programmatically (i.e., within the source code of the program). 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. For instance, in many modern programming languages, functions are first-class entities because they can be created and manipulated at run-time through the source code. Conversely, labels in C passed to `goto` do not have run-time values and, therefore, are not first-class entities. Similarly, a class in Java does not have a manipulatable value at run-time and is not a first-class entity. In contrast, a class in Smalltalk does have a value that can be manipulated at run-time, so it is a first-class entity.

In a functional style of programming, the programmer describes computation primarily by calling a series of functions that cascade a set of return values to each other. Functional programming typically does not involve variables and assignment, so side effects are absent from programs developed using a functional style. Since side effect is fundamental to sequential execution, statement blocks, and iteration, a functional style of programming utilizes recursion as a primary means of repetition. The functional style of programming was pioneered in the Lisp programming language, designed by John McCarthy in 1958 at MIT (1960). Scheme and Common Lisp are dialects of Lisp. Scheme, in particular, is an ideal vehicle for exploring language semantics and implementing language concepts. For instance, we use Scheme in this text to implement recursion from first principles, as well as a variety of other language concepts. In contrast to the von Neumann architecture, the Lisp *machine* is a predecessor to modern single-user workstations. ML, Haskell, and F# also primarily support a functional style of programming.

Functional programming is based on *lambda-calculus* (hereafter referred to as λ-*calculus*)—a mathematical theory of functions developed in 1928–1929 by Alonzo Church and published in 1932.[7] Like the Turing machine, λ-*calculus* is an abstract mathematical model capturing the notion of mechanical computation (or an algorithm). Every function that is *computable*—referred to as *decidable*—by Turing machines is also computable in (untyped) λ-calculus. One goal of functional programming is to bring the activity of programming closer to mathematics, especially to formally guarantee certain safety properties and constraints. While the criterion of sequential execution of assignment and conditional statements is sufficient to determine whether a language is Turing-complete, languages without support for sequential execution and variable assignment can also be

6. Sometimes entities in programming languages are referred to as *second-class* or even *third-class* entities. However, these distinctions are generally not helpful.

7. Alonzo Church was Alan Turing's PhD advisor at Princeton University from 1936 to 1938.

Turing-complete. Support for (1) arithmetic operations on integer values, (2) a selection operator (e.g., `if` ··· `then` ··· `else` ···), and (3) the ability to define new recursive functions from existing functions/operators are alternative and sufficient criteria to describe the computation that a Turing machine can perform. Thus, a programming language with those facilities is also Turing-complete.

The concept of *purity* in programming languages also arises with respect to programming style. A language without support for side effect, including no side effect for I/O, can be considered to support a pure form of functional programming. Scheme is not pure in its support for functional programming because it has an assignment operator and I/O operators. By comparison, Haskell is nearly pure. Haskell has no support for variables or assignment, but it supports I/O in a carefully controlled way through the use of *monads*, which are functions that have side effects but cannot be called by functions without side effects.

Again, programming without variables or assignment may seem inconceivable to some programmers, or at least seem to be an ascetical discipline. However, modification of the value of a variable through assignment accounts for a large volume of bugs in programs. Thus, without facilities for assignment one might write less buggy code. "Ericsson's AXD301 project, a couple million lines of Erlang code,[8] has achieved 99.9999999% reliability. How? 'No shared state and a sophisticated error-recovery model,' Joe [Armstrong, who was a designer of Erlang] says" (Swaine 2009, p. 16). Moreover, parallelization and synchronization of single-threaded programs is easier in the absence of variables whose values change over time since there is no shared state to protect from corruption. Chapter 5 introduces the details of the functional style of programming. The imperative and functional modes of programming are not entirely mutually exclusive, as we see in Section 1.4.6.

1.4.3 Object-Oriented Programming

In object-oriented programming, a programmer develops a solution to a problem as a collection of *objects* communicating by passing *messages* to each other (Figure 1.1):

> I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning—it took a while to see how to do messaging in a programming language efficiently enough to be useful). (Kay 2003)

Objects are program entities that encapsulate data and functionality. An object-oriented style of programming typically unifies the concepts of data and procedural abstraction through the constructs of classes and objects. The object-oriented style of programming was pioneered in the Smalltalk programming language, designed by Alan Kay and colleagues in the early 1970s at Xerox PARC.

8. Erlang is a language supporting concurrent and functional programming that was developed by the telecommunications company Ericsson.

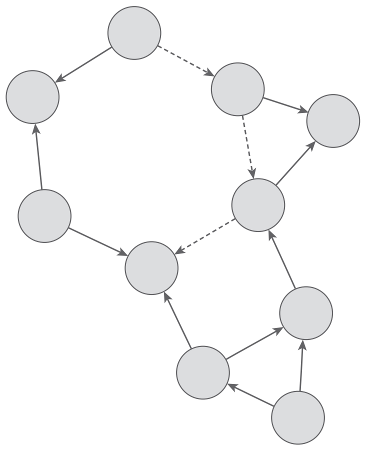

Figure 1.1 Conceptual depiction of a set of objects communicating by passing messages to each other to collaboratively solve a problem.

While there are imperative aspects involved in object-oriented programming (e.g., assignment), the concept of a *closure* from functional programming (i.e., a first-class function with associated bindings) is an early precursor to an *object* (i.e., a program entity encapsulating behavior and state). Alan Kay (2003) has expressed that Lisp influenced his thoughts in the development of object orientation and Smalltalk. Languages supporting an object-oriented style of programming include Java, C++, and C#. A language supporting a pure style of object-oriented programming is one where all program entities are objects—including primitives, classes, and methods—and where all computation is described by passing messages between these objects. Smalltalk and languages based on the Common Lisp Object System (CLOS), including Dylan, support a pure form of object-oriented programming.

Lisp (and the Lisp machine) and Smalltalk were the experimental platforms that gave birth to many of the commonly used and contemporary language features, including implicit pointer dereferencing, automatic garbage collection, run-type typing, and associated tools (e.g., interactive programming environments and pointing devices such as the mouse). Both languages significantly influenced the subsequent evolution of programming languages and, indeed, personal computing. Lisp, in particular, played an influential role in the development of other important programming languages, including Smalltalk (Kay 2003).

1.4.4 Logic/Declarative Programming

The defining characteristic of a *logic* or *declarative* style of programming is description of *what* is to be computed, not *how* to compute it. Thus, declarative programming is largely an activity of specification, and languages supporting declarative programming are sometimes called *very-high-level languages* or *fifth-generation languages*. Languages supporting a logic/declarative style of programming have support for reasoning about facts and rules; consequently, this style of programming is sometimes referred to as *rule-based*. The basis of the logic/declarative style of programming is *first-order predicate calculus*.

Prolog is a language supporting a logic/declarative style of programming. In contrast to the von Neumann architecture, the *Warren Abstract Machine* is a target platform for Prolog compilers. CLIPS is also a language supporting logic/declarative programming. Likewise, programming in SQL is predominantly done in a declarative manner. A SQL query describes what data is desired, not how to find that data (i.e., developing a plan to answer the query). Usually language support for declarative programming implies an inefficient language implementation since declarative specification occurs at a very high level. In turn, interpreters for languages that support declarative programming typically involve multiple layers of abstraction.

An objective of logic/declarative programming is to support the specification of both what you want and the knowledge base (i.e., the facts and rules) from which what you want is to be inferred without regard to how the system will deduce the result. In other words, the programmer should not be required or permitted to codify the facts and rules in the program in a form that imparts control over or manipulates the built-in deduction algorithm for producing the desired result. No control information or procedural directives should be woven into the knowledge base so to direct the interpreter's deduction process. Specification (or declaration) should be order-independent. Consider the following two logical propositions:

If it is	raining	and	windy,	I carry an umbrella.	$(R \land W) \supset U$
If it is	windy	and	raining,	I carry an umbrella.	$(W \land R) \supset U$

Since the conjunction logical operator ($\land$) is commutative, these two propositions are semantically equivalent and, thus, it should not matter which of the two forms we use in a program. However, since computers are deterministic systems, the interpreter for a language supporting declarative programming typically evaluates the terms on the left-hand side of these propositions (i.e., R and W) in a left-to-right or right-to-left order. Thus, the desired result of the program can—due to side effect and other factors—depend on that evaluation order, akin to the evaluation order of the terms in the Python expression `x + f()` described earlier. Languages supporting logic/declarative programming as the primary mode of performing computation often equip the programmer with facilities to impart control over the search strategy used by the system (e.g., the cut operator in Prolog). These control facilities violate a defining principle of a declarative style—that is, the programmer need only be concerned with the logic and can leave the control (i.e., the inference methods used to produce program output) up to the system. Unlike Prolog, the Mercury programming language is nearly pure in its support for declarative programming because it does not support control facilities intended to circumvent or direct the search strategy built into the system (Somogyi, Henderson, and Conway 1996). Moreover, the form of the specification of the facts and rules in a logic/declarative program should have no bearing on the output of the program. Unfortunately, it often does. Mercury is the closest to a language

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

Table 1.3 Purity in Programming Languages

supporting a purely logic/declarative style of programming. Table 1.3 summarizes purity in programming styles. Chapter 14 discusses the logic/declarative style of programming.

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. This process is called *bottom-up programming* and the resulting language is typically either an *embedded* or a *domain-specific language*. Bottom-up programming is not on the same conceptual level as the other styles of programming discussed in this chapter—it is on more of a *meta*-level. Similarly, Lisp is not just a programming language or a language supporting multiple styles of programming. From its origin, Lisp was designed as a language to be extended (Graham 1993, p. vi), or "a programmable programming language" (Foderaro 1991, p. 27), on which the programmer can build layers of languages supporting multiple styles of programming. For instance, the abstractions in Lisp can be used to extend the language with support for object-oriented programming (Graham 1993, p. ix). This style of programming or metaprogramming, called bottom-up programming, involves using a programming language not as a tool to write a target program, but to define a new targeted (or domain-specific) language and then develop the target program in that language (Graham 1993, p. vi). In other words, bottom-up programming involves "changing the language to suit the problem" (Graham 1993, p. 3). "Not only can you program *in* Lisp (that makes it a programming language) but you can program the language itself" (Foderaro 1991, p. 27). It has been said that "[i]f you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases" (Friedman and Felleisen 1996b, p. 207).

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

Table 1.4 Practical/Conceptual/Theoretical Basis for Common Styles of Programming

syntax: *form* of language
semantics: *meaning* of language
first-class entity
side effect
referential transparency

Table 1.5 Key Terms Discussed in Section 1.4

Other programming languages are also intended to be used for bottom-up programming (e.g., Arc[9]). While we do return to the idea of bottom-up programming in Section 5.12 in Chapter 5, and in Chapter 15, the details of bottom-up programming are beyond the scope of this text. For now it suffices to say that bottom-up design can be thought of as building a library of functions followed by writing a concise program that calls those functions. "However, Lisp gives you much broader powers in this department, and augmenting the language plays a proportionately larger role in Lisp style—so much so that [as mentioned previously] Lisp is not just a different language, but a whole different way of programming" (Graham 1993, p. 4).

A host of other styles of programming are supported by a variety of other languages: concatenative programming (e.g., Factor, Joy) and dataflow programming (e.g., LabView). Table 1.4 summarizes the origins of the styles of programming introduced here. Table 1.5 presents the terms introduced in this section that are fundamental/universal to the study of programming languages.

1.4.6 Synthesis: Beyond Paradigms

Most languages have support for imperative (e.g., assignment, statement blocks), object-oriented (e.g., objects, classes), and functional (e.g., λ/anonymous [and

9. http://arclanguage.org

first-class] functions) programming. Some languages even have, to a lesser extent, support for declarative programming (e.g., pattern-directed invocation).

What we refer to here as *styles of programming* was once—and in many cases still is—referred to as *paradigms of languages*.[10] Imperative, functional, logic/declarative, and object-oriented have been, traditionally, the four classical paradigms of languages. However, historically, other paradigms have emerged for niche application domains,[11] including languages for business applications (e.g., COBOL), *hardware description languages* (e.g., Verilog, VHDL), and *scripting languages* (e.g., `awk`, Rexx, Tcl, Perl). Traditional scripting languages are typically interpreted languages supporting an imperative style of programming with an easy-to-use command-and-control–oriented syntax and ideal for processing strings and generating reports. The advent of the web ignited the evolution of languages used for traditional scripting-type tasks into languages supporting multiple styles of programming (e.g., JavaScript, Python, Ruby, PHP, and Tcl/Tk). As the web and its use continued to evolve, the programming tasks common to web programming drove these languages to continue to grow and incorporate additional features and constructs supporting more expressive and advanced forms of functional, object-oriented, and concurrent programming. (Use of these languages with associated development patterns [e.g., Model-View-Controller] eventually evolved into *web frameworks* [e.g., Express, Django Rails, Lavavel].)

The styles of programming just discussed are not mutually exclusive, and language support for multiple styles is not limited to those languages used solely for web applications. Indeed, one can write a program with a functional motif while sparingly using imperative constructs (e.g., assignment) for purposes of pragmatics. Scheme and ML primarily support a functional style of programming, but have some imperative features (e.g., assignment statements and statement blocks). Alternatively, one can write a primarily imperative program using some functional constructs (e.g., λ/anonymous functions). Dylan, which was influenced by Scheme and Common Lisp, is a language that adds support for object-oriented programming to its functional programming roots. Similarly, the *pattern-directed invocation* built into languages such as ML and Haskell is declarative in nature and resembles the rule-based programming, at least syntactically, in Prolog. Curry is a programming language derived from Haskell and, therefore, supports functional programming; however, it also includes support for logic programming. In contrast, POP-11 primarily facilitates a declarative style of programming, but

10. A *paradigm* is a worldview—a model. A *model* is a simplified view of some entity in the real world (e.g., a model airplane) that is simpler to interact with. A *programming language paradigm* refers to a style of performing computation from which programming in a language adhering to the tenets of that style proceeds. A language paradigm can be thought of as a family of natural languages, such as the Romance languages or the Germanic languages.

11. In the past, even the classical functional and logic/declarative paradigms, and specifically the languages Lisp and Prolog, respectively, were considered paradigms primarily for artificial intelligence applications even though the `emacs` text editor for UNIX and *Autocad* are two non-AI applications that are more than 30 years old and were developed in Lisp. Now there are Lisp and Prolog applications in a variety of other domains (e.g., *Orbitz*). We refer the reader to Graham (1993, p. 1) for the details of the origin of the (accidental) association between Lisp and AI. Nevertheless, certain languages are still ideally suited to solve problems in a particular niche application domain. For instance, C is a language for systems programming and continues to be the language of choice for building operating systems.

supports first-class functions. Scala is a language with support for functional programming that runs on the Java virtual machine.

Moreover, some languages support database connectivity to make (declaratively written) queries to a database system. For instance, C# supports "Language-INtegrated Queries" (LINQ), where a programmer can embed SQL-inspired declarative code into programs that otherwise use a combination of imperative, functional, object-oriented, and concurrent programming constructs. Despite this phenomenon in language evolution, both the concept and use of the term *paradigm* as well as the classical boundaries were still rigorously retained. These languages are referred to as either *web programming languages* (i.e., a new paradigm was invented) or *multi-paradigm languages*—an explicit indication of the support for multiple paradigms needed to maintain the classical paradigms.

Almost no languages support only one style of programming. Even Fortran and BASIC, which were conceived as imperative programming languages, now incorporate object-oriented features. Moreover, Smalltalk, which supports a pure form of object-oriented programming, has support for closures from functional programming—though, of course, they are accessed and manipulated through object orientation and message passing. Similarly, Mercury, which is considered nearly a pure logic/declarative language, also supports functional programming. For example, while based on Prolog, Mercury marries Prolog with the Haskell type system (Somogyi, Henderson, and Conway 1996). Conversely, almost all languages support some form of concurrent programming—an indication of the influence of multicore processors on language evolution (Section 1.5). Moreover, many languages now support some form of λ/anonymous functions. Languages supporting more than one style of programming are now the norm; languages supporting only one style of programming are now the exception.[12]

Perhaps this is partial acknowledgment from the industry that concepts from functional (e.g., first-class functions) and object-oriented programming (e.g., reflection) are finding their way from research languages into mainstream languages (see Figure 1.4 and Section 1.5 later in this chapter). It also calls the necessity of the concept of *language paradigm* into question. If all languages are multi-paradigm languages, then the concept of language paradigm is antiquated. Thus, the boundaries of the classical (and contemporary) paradigms are by now thoroughly blurred, rendering both the boundaries and the paradigms themselves irrelevant: "Programming language 'paradigms' are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them?" (Krishnamurthi 2008). The terms originally identifying language paradigms (e.g., imperative, object-oriented, functional, and declarative) are more *styles* of programming[13, 14] than descriptors for languages or patterns for languages to follow. Thus, instead of talking about

12. The miniKanren family of languages primarily supports logic programming.
13. John Backus (1978) used the phrase "functional style" in the title of his 1977 Turing Award paper.
14. When we use the phrase "styles of programming" we are not referring to the program formatting guidelines that are often referred to as "program style" (e.g., consistent use of three spaces for indentation or placing the function return type on a separate line) (Kernighan and Plauger 1978), but rather the style of effecting and describing computation.

a "functional language" or an "object-oriented language," we discuss "functional programming" and "object-oriented programming."

A style of programming captures the concepts and constructs through which a language provides support for effecting and describing computation (e.g., by assignment and side effect vis-á-vis by functions and return values) and is not a property of a language. The essence of the differences between styles of programming is captured by how *computation* is fundamentally effected and described in each style.[15]

1.4.7 Language Evaluation Criteria

As a result of the support for multiple styles of programming in a single language, now, as opposed to 30 years ago, a comparative analysis of languages cannot be fostered using the styles (i.e., "paradigms") themselves. For instance, since Python and Go support multiple overlapping styles of programming, a comparison of them is not as simple as stating, "Python is an object-oriented language and Go is an imperative language." Despite their support for a variety of programming styles, all computer languages involve a core set of universal concepts (Figure 1.2), so concepts of languages provide the basis for undertaking comparative analysis. Programming languages differ in terms of the implementation options each employs for these concepts. For instance, Python is a dynamically typed language and Go is a statically typed language. 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.

There are other criteria—sometimes called *nonfunctional requirements*—by which to evaluate languages. Traditionally, these criteria include readability, writability, reliability (i.e., safety), and cost. For instance, all of the parentheses in Lisp affect the readability and writability of Lisp programs.[16] Others might argue that the verbose nature of COBOL makes it a readable language (e.g., `ADD 1 TO X GIVING Y`), but not a writable language. How are readability and writability related? In the case of COBOL, they are inversely proportional to each other. Some criteria are subject to interpretation. For instance, cost (i.e., efficiency) can refer to the cost of execution or the cost of development. Other language evaluation criteria include portability, usability, security, maintainability, modifiability, and manageability.

Languages can be also be compared on the basis of their implementations. Historically, languages that primarily supported imperative programming

15. For instance, the *object-relational impedance mismatch* between relational database systems (e.g., PostgreSQL or MySQL) and languages supporting object-oriented programming—which refers to the challenge in mapping relational schemas and database tables (which are set-, bag-, or list-oriented) in a relational database system to class definitions and objects—is more a reflection of differing levels of granularity in the various data modeling support structures than one fundamental to describing computation.

16. Some have stated that Lisp stands for **L**isp **I**s **S**uperfluous **P**arentheses.

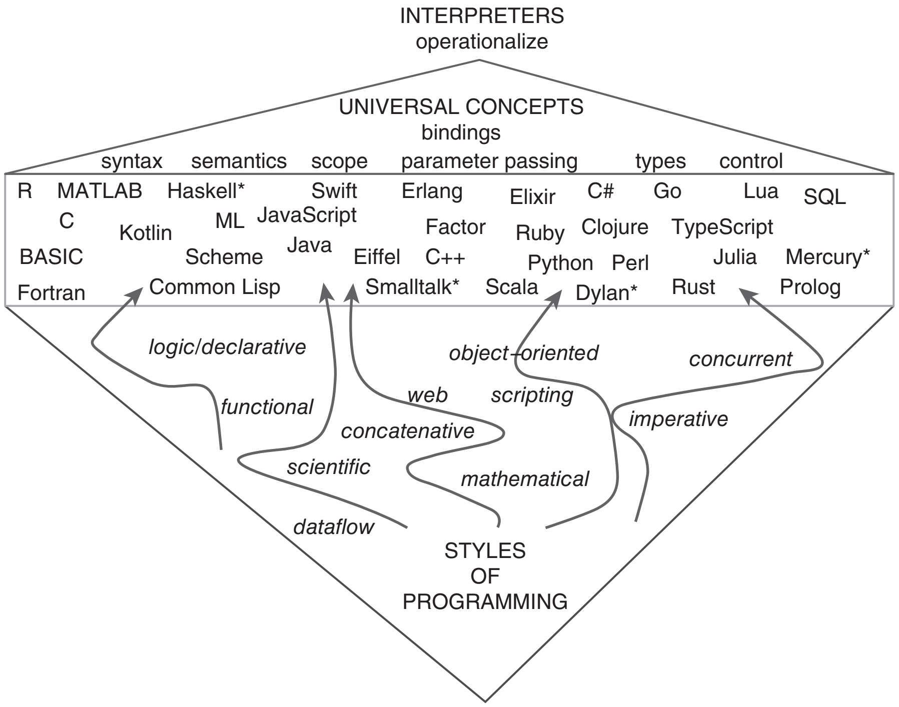

Figure 1.2 Within the context of their support for a variety of programming styles, all languages involve a core set of universal concepts that are operationalized through an interpreter and provide a basis for (comparative) evaluation. Asterisks indicate (near-)purity with respect to programming style.

involved mostly static bindings and, therefore, tended to be compiled. In contrast, languages that support a functional or logic/declarative style of programming involve mostly dynamic bindings and tend to be interpreted. (Chapter 4 discusses strategies for language implementation.)

1.4.8 Thought Process for Problem Solving

While most languages now support multiple styles of programming, use of the styles themselves involves a shift in one's problem-solving thought process. Thinking in one style (e.g., iteration—imperative) and programming in another style (e.g., functional, where recursive thought is fundamental) is analogous to translating into your native language every sentence you either hear from or speak to your conversational partner when participating in a synchronous dialog in a foreign language—an unsustainable strategy. Just as a one-to-one mapping between phrases in two natural languages—even those in the same family of languages (e.g., the Romance languages)—does not exist, it is generally not possible to translate the solution to a problem conceived with thought endemic to one style (e.g., imperative thought) into another (e.g., functional constructs), and vice versa.

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. Learning to think and program in alternative styles typically entails unlearning bad habits acquired unconsciously through the use of other languages to accommodate the lack of support for that style in those languages. Consider how a programmer might implement an inherently recursive algorithm such as mergesort using a language without support for recursion:

> Programming languages teach you not to want what they cannot provide. You have to think in a language to write programs in it, and it's hard to want something you can't describe. When I first started writing programs—in Basic—I didn't miss recursion, because I didn't know there was such a thing. I thought in Basic. I could only conceive of iterative algorithms, so why should I miss recursion? (Graham 1996, p. 2)

Paul Graham (2004b, p. 242) describes the effect languages have on thought as the *Blub Paradox*—"[t]he inability to understand the power of programming languages more powerful than the ones you're used to thinking in."[17] Programming languages and the use thereof are—perhaps, so far—the only conduit into the science of computing experienced by students. Because language influences thought and capacity for thought, an improved understanding of programming languages and the different styles of programming supported by that understanding result in a more holistic view of computation.[18] 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). An understanding of Latin—even an elementary understanding—not only helps one learn new languages but also improves one's use and command over their native language. 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. "[L]earning Lisp will teach you more than just a new language—it will teach you new and more powerful ways of thinking about programs" (Graham 1996, p. 2).

1.5 Factors Influencing Language Development

Surprisingly enough, programming languages did not historically evolve based on the abilities of programmers (Weinberg 1988). (One could argue that programmers'

17. Notice use of the phrase "thinking in" instead of "programming in."
18. The study of formal languages leads to the concept of a Turing machine; thus, language is integral to the theory of computation.

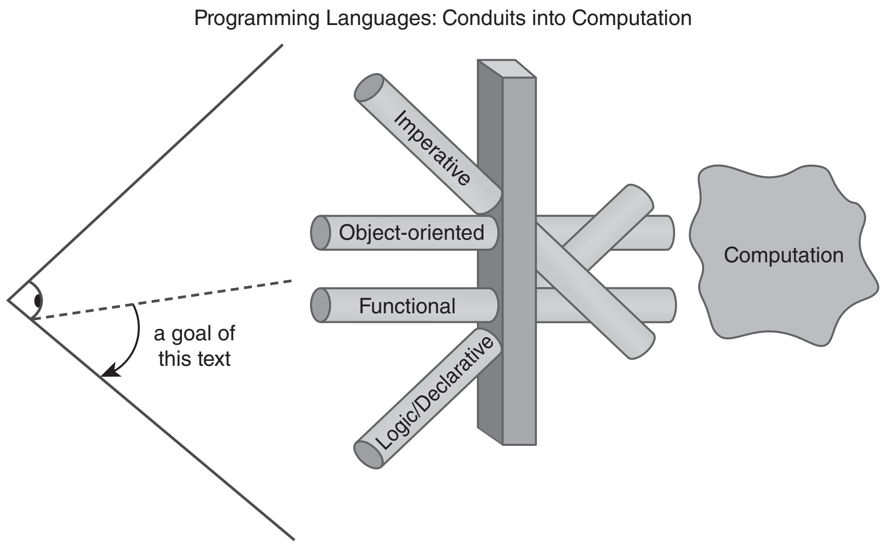

Figure 1.3 Programming languages and the styles of programming therein are conduits into computation.

abilities evolved based on the capabilities and limitations of programming languages.) 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. In the von Neumann architecture, a sequence of program instructions and program data are both stored in main memory. Similarly, the languages inspired by this model view variables (in which to store program data) as abstractions of memory cells. Further, in these languages variables are manipulated through a sequence of commands, including an assignment statement that changes the value of a variable.

Fortran is one of oldest programming languages still in use whose design was based on the von Neumann architecture. The primary design goal of Fortran was speed of execution since Fortran programs were intended for scientific and engineering applications and had to execute fast. Moreover, the emphasis on planning programs in advance advocated by software design methodologies (e.g., *structured programming* or *top-down design*) resulting from the *software crisis*[19] in the 1960s and 1970s promoted the use of static bindings, which then reinforces the use of compiled languages. The need to produce programs that executed fast helped fuel the development of compiled languages such as Fortran, COBOL, and C. Compiled languages with static bindings and top-down design reinforce each other.

Often while developing software we build *throwaway prototypes* solely for purposes of helping us collect, crystallize, and analyze software requirements, candidate designs, and implementation approaches. It is widely believed that

19. The software crisis in the 1960s and 1970s refers to the software industry's inability to scale the software development process of large systems in the same way as other engineering disciplines.

writing generates and clarifies thoughts (Graham 1993, p. 2). For instance, the process of enumerating a list of groceries typically leads to thoughts of additional items that need to be purchased, which are then listed, and so on. An alternative to structured programming is *literate programming*, a notion introduced by Donald Knuth. Literate programming involves crafting a program as a representation of one's thoughts in natural language rather than based on constraints imposed by computer architecture and, therefore, programming languages.[20] Moreover, in the 1980s the discussion around the ideas of object-oriented design emerged through the development of Smalltalk—an interpreted language. Advances in computer hardware, and particularly *Moore's Law*,[21] also helped reduce the emphasis on speed of program execution as the overriding criterion in the design of programming languages.

While fewer interpreted languages emerged in the 1980s compared to compiled ones, the confluence of literate programming, object-oriented design, and Moore's Law sparked discussion of speed of development as a criterion for designing programming languages.

The advent of the World Wide Web in the late 1990s and early 2000s and the new interactive and networked computing platform on which it runs certainly influenced language design. Language designers had to address the challenges of developing software that was intended to run on a variety of hardware platforms and was to be delivered or interacted with over a network. Moreover, they had to deal with issues of maintaining state—so fundamental to imperative programming—over a stateless (`http`) network protocol. For all these reasons, programming for the web presented a fertile landscape for the practical exploration of issues of language design. Programming languages tended toward the inclusion of more dynamic bindings, so more interpreted languages emerged at this time (e.g., JavaScript).

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 ability, or lack thereof, to delay bindings until run-time affects flexibility of program development. The more dynamic bindings a language supports, the fewer the number of commitments the programmer must make during program development. Thus, dynamic bindings provide for convenient debugging, maintenance, and redesign when dealing with errors or evolving program requirements. For instance, run-time binding of messages to methods in Python allows programs to be more easily designed during their initial development and then subsequently extended during their maintenance.

20. While a novel concept, embraced by tools (e.g., *Noweb*) and languages (e.g., the proprietary language Miranda, which is a predecessor of Haskell and similarly supports a pure form of functional programming), the idea of literate programming never fully caught on.

21. *Moore's Law* states that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years and describes the evolution of computer hardware.

Graham (2004b) describes this process with a metaphor—namely, an oil painting where the painter can smudge the oil to correct any initial flaws. Thus, programming languages that support dynamic bindings are the oil that can reduce the cost of mistakes. There has been an incremental and ongoing shift toward support for more dynamic bindings in programming languages to enable the creation of malleable programs.

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

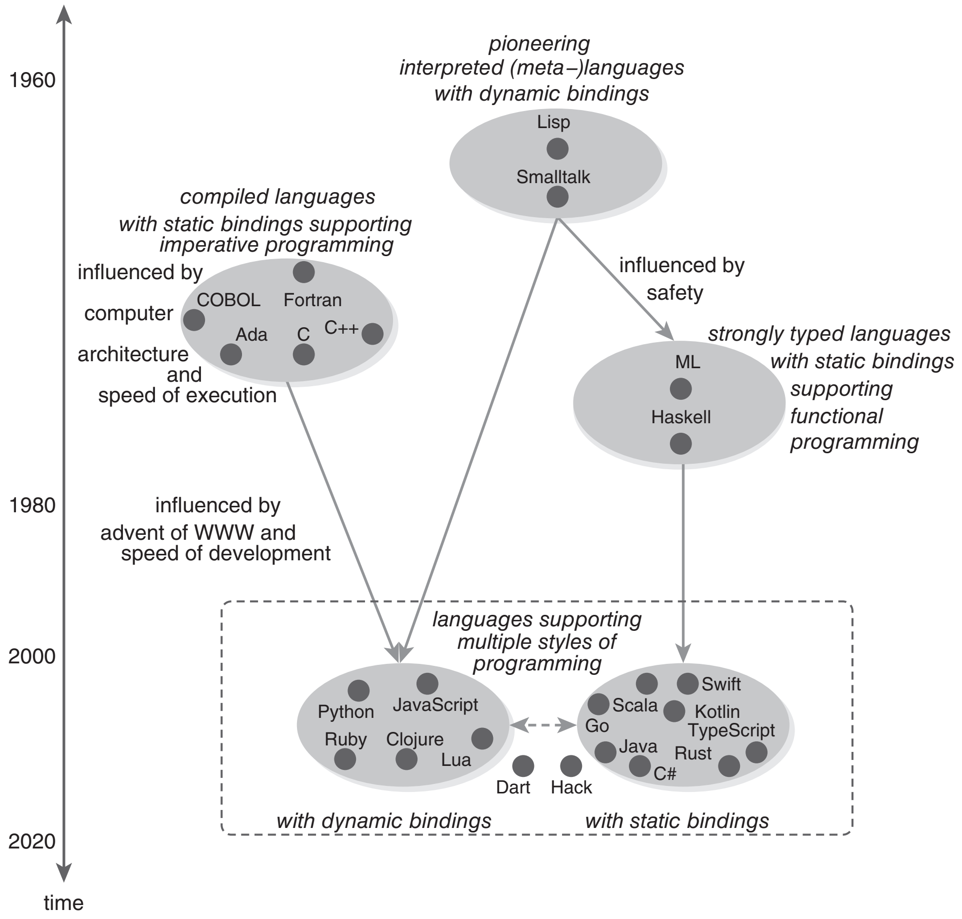
1960
1980
2000
2020
time
pioneering interpreted (meta–)languages with dynamic bindings
Lisp
Smalltalk
compiled languages with static bindings supporting imperative programming
influenced by computer architecture and speed of execution
COBOL
Fortran
Ada
C
C++
influenced by safety
strongly typed languages with static bindings supporting functional programming
ML
Haskell
influenced by advent of WWW and speed of development
languages supporting multiple styles of programming
Python
JavaScript
Ruby
Clojure
Lua
Dart
Hack
Swift
Scala
Kotlin
Go
TypeScript
Java
Rust
C#
with dynamic bindings
with static bindings

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

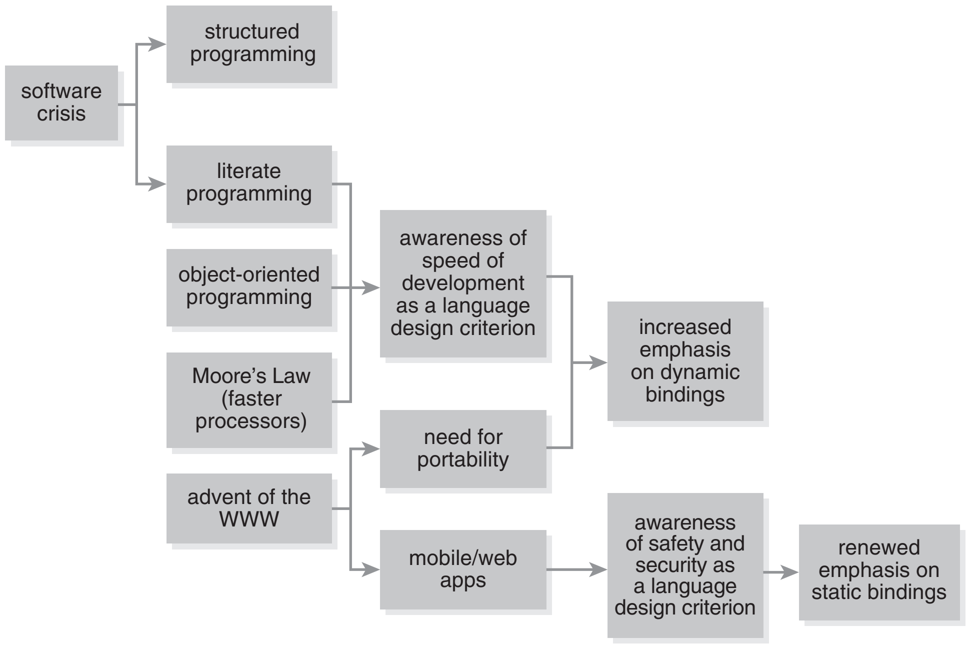
software crisis
structured programming
literate programming
object-oriented programming
Moore's Law (faster processors)
advent of the WWW
awareness of speed of development as a language design criterion
need for portability
mobile/web apps
increased emphasis on dynamic bindings
awareness of safety and security as a language design criterion
renewed emphasis on static bindings

Figure 1.5 Factors influencing language design.

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.

With the computing power available today and the time-to-market demands placed on software development, speed of execution is now less emphasized as a design criterion than it once was.[22] Software development process methodologies have commensurately evolved in this direction as well and embrace this trend. *Agile methods* such as *extreme programming* involve repeated and rapid tours through the software development cycle, implying that speed of development is highly valued.

1.6 Recurring Themes in the Study of Languages

The following is a set of themes that recur throughout this text:

- 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.

22. In some engineering applications, speed of execution is still the overriding design criterion.

- 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. For instance, there is an inverse relationship between static (rigid and fast) and dynamic (flexible and slow) bindings. Reliability, predictability, and safety are the primary motivations for using a statically typed programming language, while flexibility and efficiency are motivations for using a dynamically typed language.
- Side effects are often the underlying culprit of many programming perils.
- 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.
    - Some idioms cannot be expressed as easily or at all in certain languages as they can in others.
    - Languages, through their support for a variety of programming styles (e.g., functional, declarative), require programmers to undertake a shift in thought process toward problem solving that develops additional avenues through which programmers can describe ideas about computation and, therefore, provides a more holistic view of computer science.
- Languages are built on top of languages.
- 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.[23] "Great software, likewise, requires a fanatical devotion to beauty" (Graham 2004b, p. 29).

23. Architect Christopher Alexander and colleagues (1977) explored the relationship between (architectural) patterns and languages and, as a result, inspired *design patterns* in software (Gamma et al. 1995).

- Problem solving and subsequent programming implementation require pattern recognition and application, respectively.

To close the loop, we return to these themes in Chapter 15 (Conceptual Exercise 15.3).

1.7 What You Will Learn

The following is a succinct summary of some of the topics about which readers can expect to 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

One approach to learning language concepts is to implement the studied concepts through the construction of a progressive series of interpreters, and to assess the differences in the resulting languages. One module of this text uses this approach. Specifically, in Chapters 10–12, we implement a programming language, named Camille, supporting functional and imperative programming through the construction of interpreters in Python.

We study and use type systems and other concepts of programming languages (e.g., type inference or currying) through the type-safe languages ML and Haskell in Chapter 7. We discuss a logic/declarative style of programming through use of Prolog in Chapter 14.

1.8 Learning Outcomes

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
- 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

The study of language concepts involves the development of a methodology and vocabulary for the subsequent comparative study of particular languages and results in both an improved aptitude for choosing the most appropriate language for the task at hand and a larger toolkit of programming techniques for building powerful and programming abstractions.

Conceptual Exercises for Chapter 1

Exercise 1.1 Given the definition of *programming language* presented in this chapter, is HTML a programming language? How about LaTeX? Explain.

Exercise 1.2 Given the definition of a *programming language* presented in this chapter, is Prolog, which primarily supports a declarative style of programming, a programming language? How about Mercury, which supports a pure form of logic/declarative programming? Explain.

Exercise 1.3 There are many *times* in the study of programming languages. For example, variables are bound to types in C at *compile time*, which means that they remain fixed to their type for the lifetime of the program. In contrast, variables are bound to values at *run-time* (which means that a variable's value is not bound until run-time and can change at any time during run-time). In total, there are six (classic) times in the study of programming languages, of which compile time and run-time are two. Give an alternative time in the study of programming languages, and an example of something in C which is bound at that time.

Exercise 1.4 Are objects *first-class* in Java? C++?

Exercise 1.5 Explain how *first-class functions* can be simulated in C or C++. Write a C or C++ program to demonstrate.

Exercise 1.6 For each of the following entities, give all languages from the set {C++, ML, Prolog, Scheme, Smalltalk} in which the entity is considered *first-class*:

(a) Function
(b) Continuation
(c) Object
(d) Class

Exercise 1.7 Give a code example of a *side effect* in C.

Exercise 1.8 Are all functions without *side effect referentially transparent*? If not, give a function without a side effect that is not referentially transparent.

Exercise 1.9 Are all *referentially transparent* functions without *side effect*? If not, give a function that is referentially transparent, but has a side effect.

Exercise 1.10 Consider the following Java method:

```
1  int f() {
2     int a = 0;
3     a = a + 1;
4     return 10;
5  }
```

This function cannot modify its parameters because it has none. Moreover, it does not modify its external environment because it does not access any global data or perform any I/O. Therefore, the function does not have a side effect. However, the assignment statement on line 3 does have a side effect. How can this be? The function does not have a side effect, yet it contains a statement with a side effect—which seems like a contradiction. Does `f` have a side effect or not, and why?

Exercise 1.11 Identify two language evaluation criteria other than those discussed in this chapter.

Exercise 1.12 List two language evaluation criteria that conflict with each other. Provide two conflicts not discussed in this chapter. Give a specific example of each to illustrate the conflict.

Exercise 1.13 Fill in the blanks in the expressions in the following table with terms from the set:

{Dylan, garbage collection, Haskell,
lazy evaluation, Prolog, Smalltalk, static typing}

Go	=	C	+	____________
Curry	=	____________	+	Prolog
____________	=	Lisp	+	Smalltalk
Objective-C	=	C	+	____________
TypeScript	=	JavaScript	+	____________
Mercury	=	____________	−	impurities
Haskell	=	ML	+	____________

Exercise 1.14 What is *aspect-oriented programming*?

Exercise 1.15 Explore the *Linda* programming language. What styles of programming does it support? For which applications is it intended? What is *Linda-calculus* and how does it differ conceptually from λ-calculus?

Exercise 1.16 Identify a programming language with which you are unfamiliar—perhaps even a language mentioned in this chapter. Try to describe the language through its most defining characteristics.

Exercise 1.17 Read M. Swaine's 2009 article "It's Time to Get Good at Functional Programming" in *Dr. Dobb's Journal* and write a 250-word commentary on it.

Exercise 1.18 Read N. Savage's 2018 article "Using Functions for Easier Programming" in *Communications of the ACM*, available at https://doi.acm.org/10.1145/3193776, and write a 100-word commentary on it.

Exercise 1.19 Write a 2000-word essay addressing the following questions:

- What interests you in programming languages?
- Which concepts or ideas presented in this chapter do you find compelling? With what do you agree or disagree? Why?
- What are your goals for this course of study?
- What questions do you have?

1.9 Thematic Takeaways

- 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).
- 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).

See also the recurrent themes in Section 1.6.

1.10 Chapter Summary

This text and course of study are 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 *imperative style of programming* is a natural consequence of the *von Neumann architecture*: Instructions are imperative statements that affect, through an assignment operator, the values of variables, which are themselves abstractions of memory locations. Historically, programming languages were designed based on the computer architecture on which the programs written using them were intended to execute. The *functional style of programming* is rooted in *λ-calculus*—a mathematical theory of functions. The *logic/declarative style of programming* is grounded in *first-order predicate calculus*—a formal system of symbolic logic.

Thirty years ago, programming languages were clearly classified in these discrete categories or language *paradigms*, but that is no longer the case. Now most programming languages support a variety of styles of programming, including imperative, functional, object-oriented, and declarative programming (e.g., Python and Go). This diversity in programming styles supported in individual languages provides programmers with a richer palette in a single language for expressing ideas about computation—programming languages and the styles of programming used in these languages are conduits into computation. A goal of this text is to expose reader to these alternative styles of programming (Figure 1.3).

Within the context of their support for a variety of programming styles, all languages involve a core set of universal concepts (Figure 1.2). Programming languages differ in their design and implementation options for these core concepts as well as in the variety of concepts from the host of programming styles they support. This diversity of options in supporting concepts provides fertile ground for fostering a more meaningful comparative analysis of languages, while rendering the prevalent (and superficial) mode of language comparison of the past—putting languages in paradigms and comparing the paradigms—both irrelevant and nearly impossible. The evolution of programming languages bifurcated into languages involving primarily static binding and those involving primarily dynamic bindings (Figure 1.4).

Since language concepts are the building blocks from which all languages are constructed/organized, an understanding of the concepts implies that one can focus on the core language principles (e.g., parameter passing) and the particular options (e.g., pass-by-reference) used for those principles in (new or unfamiliar) languages rather than fixating on the details (e.g., syntax), which results in an improved dexterity in learning, assimilating, and using programming languages. Moreover, an understanding and experience with a variety of programming styles and exotic ways of performing computation establishes an increased capacity for describing computation in a program, a richer toolbox of techniques from which to solve problems, and a more well-rounded picture of computing.

1.11 Notes and Further Reading

The term *paradigm* was coined by historian of science Thomas Kuhn. Since most programming languages no longer fit cleanly into the classical language paradigms, the concept of language *purity* (with respect to a particular paradigm) is pragmatically obsolete. The notion of a *first-class entity* is attributed to British computer scientist Christopher Strachey (Abelson and Sussman 1996, p. 76, footnote 64). John McCarthy, the original designer of Lisp, received the ACM A. M. Turing Award in 1971 for contributions to artificial intelligence, including the creation of Lisp.