Exploring Languages with Interpreters and Functional Programming Chapter 2

H. Conrad Cunningham

25 September 2018

Contents

Professor of Computer and Information Science

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

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

2 Programming Paradigms

2.1 Chapter Introduction

The goals of this chapter are to:

- introduce the concepts of procedural and data abstraction
- examine the characteristics and concepts the primary programming paradigms, imperative and declarative (including functional and relational)
- survey other paradigms such as procedural and modular programming

2.2 Abstraction

Programming concerns the construction of appropriate abstractions in a programming language. Before we examine programming paradigms, let's examine the concept of abstraction.

2.2.1 What is abstraction?

As computing scientists and computer programmers, we should remember the maxim:

> *Simplicity is good; complexity is bad.*

The most effective weapon that we have in the fight against complexity is *abstraction*. What is abstraction?

Abstraction is *concentrating on the essentials and ignoring the details.*

Sometimes abstraction is described as *remembering the "what" and ignoring the "how".*

Large complex problems can only be made understandable by decomposing them into subproblems. Ideally, we should be able to solve each subproblem independently and then compose their solutions into a solution to the larger problem.

In programming, the subproblem solution is often expressed with some kind of abstraction represented in a programming notation. From the outside, each abstraction should be simple and easy for programmers to use correctly. The programmers should only need to know the abstraction's *interface* (i.e. some small number of assumptions necessary to use the abstraction correctly).

2.2.2 Kinds of abstraction

Two kinds of abstraction are of interest to computing scientists: *procedural abstraction* and *data abstraction*.

Procedural abstraction: the separation of the logical properties of an *action* from the details of how the action is implemented.

Data abstraction: the separation of the logical properties of *data* from the details of how the data are represented.

In procedural abstraction, programmers focus primarily on the actions to be carried out and secondarily on the data to be processed.

For example, in the top-down design of a sequential algorithm, a programmer first identifies a sequence of actions to solve the problem without being overly concerned about how each action will be carried out.

If an action is simple, the programmer can code it directly using a sequence of programming language statements.

If an action is complex, the programmer can abstract the action into a subprogram (e.g. a procedure or function) in the programming language. The programmer must define the subprogram's name, parameters, return value, assumptions, etc.)—that is, define its interface. The programmer subsequently develops the subprogram using the same top-down design approach.

In data abstraction, programmers primarily focus on the problem's data and secondarily on its actions. Programmers first identify the key data representations and develop the programs around those and the operations needed to create and update them.

We address procedural and data abstraction further in Chapters 6 and 7.

2.2.3 Procedures and functions

Generally we make the following distinctions among subprograms:

- A *procedure* is (in its pure form) a subprogram that takes zero or more arguments but does not return a value. It is executed for its effects, such as changing values in a data structure within the program, modifying its reference or value-result arguments, or causing some effect outside the program (e.g. displaying text on the screen or reading from a file).
- A *function* is (in its pure form) a subprogram that takes zero or more arguments and returns a value but that does not have other effects.
- A *method* is a procedure or function often associated with an object or class in an object-oriented program. Some object-oriented languages use the metaphor of message-passing. A method is the feature of an object that receives a message. In an implementation, a method is typically a

procedure or function associated with the (receiver) object; the object may be an *implicit parameter* of the method.

Of course, the features of various programming languages and usual practices for their use may not follow the above pure distinctions. For example, a language may not distinguish between procedures and functions. One term or another may be used for all subprograms. Procedures may return values. Functions may have side effects. Functions may return multiple values. The same subprogram can sometimes be called either as a function or procedure.

Nevertheless, it is good practice to maintain the distinction between functions and procedures for most cases in software design and programming.

2.3 What is a Programming Paradigm?

According to Timothy Budd, a *programming paradigm* is "a way of conceptualizing what it means to perform computation, of structuring and organizing how tasks are to be carried out on a computer" [Budd 1995 (p. 3)].

Historically, computer scientists have classified programming *languages* into one of two primary paradigms: *imperative* and *declarative*.

This imperative-declarative taxonomy categorizes programming styles and language features on how they handle state and how they execute programs.

In recent years, many imperative languages have added more declarative features, so the distinction between languages has become blurred. However, the concept of *programming* paradigm is still meaningful.

2.4 Imperative Paradigm

A program in the imperative paradigm has an *implicit state* (i.e. values of variables, program counters, etc.) that is modified (i.e. side-effected or mutated) by *constructs* (i.e. commands) in the source language [Hudak 1989].

As a result, such languages generally have an explicit notion of *sequencing* (of the commands) to permit precise and deterministic control of the state changes.

Imperative programs thus express *how* something is to be computed. They emphasize procedural abstractions.

Consider the following Java program fragment (from file Counting.java).

```
int count =  0 ;
int maxc  = 10 ;
while (count <= maxc) {
    System.out.println(count) ;
```

```
        count = count + 1 ;
    }
```

In this fragment, the program's *state* includes at least the values of the variables `count` and `maxc`, the sequence of output lines that have been printed, and an indicator of which statement to execute next (i.e. location or program counter).

The assignment statement changes the value of `count` and the `println` statement adds a new line to the output sequence. These are *side effects* of the execution.

Similarly, Java executes these commands in sequence, causing a change in which statement will be executed next. The purpose of the `while` statement is to cause the statements between the braces to be executed zero or more times. The number of times depends upon the values of `count` and `maxc` and how the values change within the `while` loop.

We call this state *implicit* because the aspects of the state used by a particular statement are not explicitly specified; the state is assumed from the context of the statement. Sometimes a statement can modify aspects of the state that are not evident from examining the code fragment itself.

The Java variable `count` is *mutable* because its value can change. After the declaration, `count` has the value 0. At the end of the first iteration of the `while` loop, it has value 1. After the `while` loop exits, it has a value 10. So a reference to `count` yields different values depending upon the state of the program at that point.

The Java variable `maxc` is also *mutable*, but this code fragment does not change its value.

Of course, the Java fragment above must be included within a `main` method to be executed. A `main` method is the entry point of a Java program.

```
public class Counting {
    public static void main(String[] args) {
        /* Java code fragment above */
    }
 }
```

Imperative languages are the "conventional" or "von Neumann languages" discussed by John Backus in his 1977 Turing Award address [Backus 1978]. (See a later section in this chapter.) They are well suited to traditional computer architectures.

Most of the languages in existence today are primarily imperative in nature. These include Fortran, C, C++, Java, C#, Python, Lua, and JavaScript.

2.5 Declarative Paradigm

A program in the declarative paradigm has *no implicit* state. Any needed state information must be handled explicitly [Hudak 1989].

A program is made up of *expressions* (or terms) that are *evaluated* rather than commands that are executed.

Repetitive execution is accomplished by *recursion* rather than by sequencing.

Declarative programs express *what* is to be computed (rather than how it is to be computed).

The declarative paradigm is often divided into two types: *functional* (or applicative) and *relational* (or logic).

2.5.1 Functional paradigm

In the functional paradigm the underlying model of computation is the mathematical concept of a *function*.

In a computation, a function is applied to zero or more arguments to compute a single result; that is, the result is deterministic (or predictable).

Consider the following Haskell code (from file Counting.hs). (Don't worry about the details of the language for now. We study the syntax and semantics of Haskell in an upcoming chapter.)

```
counter :: Int -> Int -> String
counter count maxc
    | count <= maxc = show count ++ "\n"
                      ++ counter (count+1) maxc
    | otherwise     = ""
```

This fragment is similar to the Java fragment above. This Haskell code defines a function `counter` (i.e. a procedural abstraction) that takes two integer arguments, `count` and `maxc`, and returns a string consisting of a sequence of lines with the integers from `count` to `maxc` such that each would be printed on a separate line. (It does not print the string, but it inserts a newline character at the end of each line.)

In the evaluation (i.e. "execution") of a function call, `counter` references the *values* of `count` and `maxc` corresponding to the explicit arguments of the function call. These values are not changed during the evaluation of that function call. However, the values of the arguments can be changed as needed for a subsequent *recursive* call of `counter`.

We call the state of `counter` *explicit* because it is passed in arguments of the function call. These parameters are *immutable* (i.e. their values cannot change)

within the body of the function. That is, any reference to `count` or `maxc` within a call gets the same value.

In a pure functional language like Haskell, the names like `count` and `maxc` are said to be *referentially transparent*. In the same context (such as the body of the function), they always have the same value. A name must be defined before it is used, but otherwise the order of evaluation of the expressions within a function body does not matter; they can even be evaluated in parallel.

There are no "loops". The functional paradigm uses recursive calls to carry out a task repeatedly.

As we see in later chapters, referential transparency is probably the most important property of functional programming languages. It underlies Haskell's evaluation model (Chapter 8). It also underlies the ability to state and prove "laws" about Haskell programs (e.g. Chapters 25 and 26). Haskell programmers and Haskell compilers can use the "mathematical" properties of the programs to transform programs that are more efficient.

The above Haskell fragment does not really carry out any actions; it just defines a mapping between the arguments and the return value. We can "execute" the `counter` function above with the arguments 0 and 10 with the following `IO` program.

```
main = do
    putStrLn (counter 0 10)
```

By calling the `main` function from the `ghci` interpreter, we get the same displayed output as the Java program.

Haskell separates pure computation (as illustrated by function `counter`) from computation that has effects on the environment such as input/output (as illustrated by `IO` function `main`).

In most programming languages that support functional programming, functions are treated as *first-class* values. That is, like other data types, functions can be stored in data structures, passed as arguments to functions, and returned as the results of functions. (The implementation technique for first-order functions usually involves creation of a *lexical closure* holding the function and its environment.)

In some sense, functional languages such as Haskell merge the concepts of procedural and functional abstraction. Functions are procedural abstractions, but they are also data.

A function that can take functions as arguments or return functions in the result is called a *higher-order function*. A function that does not take or return functions is thus a *first-order function*. Most imperative languages do not fully support higher-order functions.

The higher-order functions in functional programming languages enable regular

and powerful abstractions and operations to be constructed. By taking advantage of a library of higher-order functions that capture common patterns of computation, we can quickly construct concise, yet powerful, programs.

Purely functional languages include Haskell, Idris, Miranda, Hope, Elm, and Backus' FP.

Hybrid functional languages with significant functional subsets include Scala, F#, OCaml, SML, Erlang, Elixir, Lisp, Clojure, and Scheme.

Mainstream imperative languages such as Java (beginning with version 8), C#, Python, Ruby, Groovy, Rust, and Swift have recent feature extensions that make them hybrid languages as well.

2.5.2 Relational (or logic) paradigm

In the relational (logic) paradigm, the underlying model of computation is the mathematical concept of a *relation* (or a *predicate*) [Hudak 1989].

A computation is the (nondeterministic) association of a group of values—with backtracking to resolve additional values.

Consider the following Prolog code (from file Counting.pl). In particular, this code runs on the SWI-Prolog interpreter. (Don't worry about the details of the language.)

```
counter(X,Y,S) :- count(X,Y,R), atomics_to_string(R,'\n',S).

count(X,X,[X]).
count(X,Y,[])     :- X  > Y.
count(X,Y,[X|Rs]) :- X < Y, NX is X+1, count(NX,Y,Rs).
```

This fragment is somewhat similar to the Java and Haskell fragments above. It can be used to generate a string with the integers from `X` to `Y` where each integer would be printed on a separate line. (As with the Haskell fragment, it does not print the string.)

This program fragment defines a *database* consisting of four *clauses*.

The clause

```
count(X,X,[X]).
```

defines a *fact*. For any *variable* value `X` and list `[X]` consisting of the single value `X`, `count(X,X,[X])` is asserted to be true.

The other three clauses are *rules*. The left-hand-side of `:-` is true if the right-hand-side is also true. For example,

```
count(X,Y,[]) :- X > Y.
```

asserts that

```
count(X,Y,[])
```

is true when `X > Y`. The empty brackets denote an empty list of values.

As a logic or relational language, we can *query* the database for any missing components. For example,

```
count(1,1,Z).
```

yields the value `Z = [1]`. However,

```
count(X,1,[1]).
```

yields the value `X = 1`. If more than one answer is possible, the program can generate all of them in some nondeterministic order.

So, in some sense, where imperative and functional languages only run a computation in one direction and give a single answer, Prolog can potentially run a computation in multiple directions and give multiple answers.

As with Haskell, the above Prolog fragment does not really carry out any computational actions; it just adds facts to the database and defines general relationships among facts. We can "execute" the query `counter(0,10,S)` above and print the value of `S` using the following rule.

```
main :- counter(0,10,S), write(S).
```

Example relational languages include Prolog, Parlog, and miniKanren.

Most Prolog implementations have imperative features such as the "cut" and the ability to assert and retract clauses.

2.6 Other Programming Paradigms

As we noted, the imperative-declarative taxonomy described above divides programming styles and language features on how they handle state and how they are executed.

The computing community often speaks of other paradigms—procedural, modular, object-oriented, concurrent, parallel, language-oriented, scripting, reactive, and so forth. The definitions of these "paradigms" may be quite fuzzy and vary significantly from one writer to another.

Sometimes a term is chosen for "marketing" reasons—to associate a language with some trend even though the language may be quite different from others in that paradigm—or to make a language seem different and new even though it may not be significantly different.

These paradigms tend to divide up programming styles and language features along different dimensions than the primary taxonomy described in the previous sections. Often the languages we are speaking of are subsets of the imperative paradigm.

This section briefly discusses some of these paradigms. We discuss the prominent object-based paradigms in the next chapter.

2.6.1 Procedural paradigm

The *procedural paradigm* is a subcategory of the imperative paradigm. It organizes programs primarily using procedural abstractions. A procedural program consists of a sequence of steps that access and modify the program's state. Some of the steps are abstracted as subprograms—procedures or functions—that can be reused. In some cases, subprograms may be nested inside other subprograms, thus limiting the part of the program in which the nested subprogram can be called.

Consider the following Python 3 code (from file CountingProc.py). (Don't worry about the details of the language.)

```
def counter(count,maxc):
    def has_more(count,maxc): # new variables
        return count <= maxc
    def incr():
        nonlocal count     # from counter
        count = count + 1
    while has_more(count,maxc):
        print(f'{count}')  # Python 3.6+ string interpolation
        incr()
```

When called as

```
counter(0,10)
```

this imperative Python 3 "procedure" executes similarly to the Java program fragment we examined in a previous section.

Python does not distinguish between procedures and functions as we have defined them. It uses the term "function" for both. Both return values and can have side-effects. The value returned may be the special default value `None`.

This Python code uses procedural abstraction more extensively than the earlier Java fragment. The Python procedure encloses the `while` loop in procedure `counter` and abstracts the loop test and incrementing operation into function `has_more` and procedure `incr`, respectively.

Like many procedural languages, Python uses *lexical scope* for variable, procedure, and function names. That is, the *scope* of a name (i.e. range of code in which it can be accessed) begins at the point it is defined and ends at the end of that block of code (e.g. function, class, or module).

Function `has_more` and procedure `incr` are encapsulated within `counter`. They can only be accessed inside the body of `counter` after their definitions.

Parameters `count` and `maxc` of procedure `counter` can be accessed throughout the body of `counter` unless hidden by another variable or parameter with the same name. They are hidden within the function `has_more`, which reuses the names for its parameters, but are accessible within procedure `incr`.

But to allow assignment to `count` within the nested procedure `incr`, the variable must declared as `nonlocal` in the inner procedure. Otherwise, the assignment would have created a new variable with the name `count` within the body of procedure `incr`.

Languages like Python, C, Fortran, Pascal, and Lua are primarily procedural languages, although most have evolved to support other styles.

2.6.2 Modular paradigm

Modular programming refers more to a design method for programs and program libraries than to languages.

Modular programming means to decompose a program into units of functionality (i.e. modules) that can be developed separately. These *modules* can hide (i.e. encapsulate) key design and implementation details within the module.

The module's *public* features can be accessed through its interface; its *private* features cannot be accessed from outside the module. Thus a module supports the principle of *information hiding*. This method also keeps the interactions among modules at a minimum, maintaining a low degree of coupling.

We discuss modular programming in more depth in Chapters 6 and 7

A language that provides constructs for defining modules, packages, namespaces, or separate compilation units can assist in writing modular programs.

We examine modular programming in the functional programming language Haskell in later chapters.

Here let's consider the following Python 3 code (from file CountingMod.py) to illustrate use of modules in programs. This module is similar to the procedural program in the previous section.

```
# File CountingMod.py
count = 0
maxc  = 10

def has_more(count,maxc):
    return count <= maxc

def incr():
    global count
    count = count + 1
```

```
def counter():
    while has_more(count,maxc):
        print(f'{count}') # Python 3.6+ string interpolation
        incr()
```

This module creates two module-level *global* variables `count` and `maxc` and defines three module-level functions `has_more`, `incr`, and `counter`.

The module assigns initial values to the variables. Their values can be accessed anywhere later in the module unless hidden by local features with the same name (e.g. by the parameters of `has_more`).

Procedure `incr` assigns a new value to `count`. It must declare `count` as `global` so that a new local variable is not created.

Variable `maxc` is also mutable, but this module does not modify its value.

Each module is a separate file that can be imported by other Python code. It introduces a separate name space for variables, functions, and other features.

For example, we can import the module above and execute counter with the following Python code (from file `CountingModTest1.py`):

```
from CountingMod import counter
counter()
```

The `from-import` statement imports feature `counter` (a Python function) from the module in file `CountingMod.py`. The imported name `counter` can be used without qualifying it. The other features of `CountingMod` (e.g. `count` and `incr`) cannot be accessed.

As an alternative, we can import the module (from file `CountingModTest2.py`) as follows:

```
import CountingMod

CountingMod.count = 10
CountingMod.maxc  = 20
CountingMod.counter()
```

This code imports all the features of the module. It requires the variables and functions to be accessed with the name prefix `CountingMod.` (i.e. the module name followed by a period). This approach enables the importing code to modify the values of global variables in the imported module.

In this second example, the importing code can both access and modify the global variables of the imported module.

Most modern languages support "modules" in some way. Other languages (e.g., Standard ML) provide advanced support for modules with the ability to encapsulate features and provide multiple implementations of common interfaces.

2.6.3 Object-based paradigms

The dominant paradigm since the early 1990s has been the *object-oriented paradigm*. Because this paradigm is likely familiar with most readers, we examine it and related object-based paradigms in the next chapter.

2.6.4 Concurrent paradigms

TODO: Perhaps describe a paradigm like actors and give an example in Elixir.

2.7 Motivating Functional Programming: John Backus

In this book we focus primarily on the functional paradigm—on the programming language Haskell in particular. Although languages that enable or emphasize the functional paradigm have been around since the early days of computing, much of the later interest in functional programming grew from the 1977 Turing Award lecture.

John W. Backus (December 3, 1924 – March 17, 2007) was a pioneer in research and development of programming languages. He was the primary developer of Fortran while a programmer at IBM in the mid-1950s. Fortran is the first widely used high-level language. Backus was also a participant in the international team that designed the influential languages Algol 58 and Algol 60 a few years later. The notation used to describe the Algol 58 language syntax—Backus-Naur Form (BNF)—bears his name. This notation continues to be used to this day.

In 1977, ACM bestowed its Turing Award on Backus in recognition of his career of accomplishments. (This award is sometimes described as the "Nobel Prize for computer science".) The annual recipient of the award gives an address to a major computer science conference. Backus's address was titled "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs".

Although functional languages like Lisp go back to the late 1950's, Backus's address did much to stimulate research community's interest in functional programming languages and functional programming over the past four decades.

The next subsection gives excerpts from Backus' Turing Award address published as the article "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" [Backus 1978].

2.7.1 Excerpts from Backus's Turing Award Address

Programming languages appear to be in trouble. Each successive language incorporates, with little cleaning up, all the features of its predecessors plus a

few more. Some languages have manuals exceeding 500 pages; others cram a complex description into shorter manuals by using dense formalisms. ... Each new language claims new and fashionable features, such as strong typing or structured control statements, but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them.

Since large increases in size bring only small increases in power, smaller, more elegant languages such as Pascal continue to be popular. But there is a desperate need for a powerful methodology to help us think about programs, and no conventional language even begins to meet that need. In fact, conventional languages create unnecessary confusion in the way we think about programs. ... In order to understand the problems of conventional programming languages, we must first examine their intellectual parent, the von Neumann computer. What is a von Neumann computer? When von Neumann and others conceived of it ... [in the 1940's], it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed radically, we nevertheless still identify the notion of "computer" with this ... concept.

In its simplest form a von Neumann computer has three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the *von Neumann bottleneck*. The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear.

Ironically, a large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. Before a word can be sent through the tube its address must be in the CPU; hence it must either be sent through the tube from the store or be generated by some CPU operation. If the address is sent form the store, then its address must either have been sent from the store or generated in the CPU, and so on. If, on the other hand, the address is generated in the CPU, it must either be generated by a fixed rule (e.g. "add 1 to the program counter") or by an instruction that was sent through the tube, in which case its address must have been sent, and so on.

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. ...

Conventional programming languages are basically high level, complex versions of the von Neumann computer. Our ... old belief that there is only one kind of

computer is the basis our our belief that there is only one kind of programming language, the conventional—von Neumann—language. The differences between Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the von Neumann computer. Although I refer to conventional languages as "von Neumann languages" to take note of their origin and style, I do not, of course, blame the great mathematician for their complexity. In fact, some might say that I bear some responsibility for that problem.

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-at-time terms in much the same way the computer's bottleneck does.

Consider a typical program; at its center are a number of assignment statements containing some subscripted variables. Each assignment statement produces a one-word result. The program must cause these statements to be executed many times, while altering subscript values, in order to make the desired overall change in the store, since it must be done one word at a time. The programmer is thus concerned with the flow of words through the assignment bottleneck as he designs the nest of control statements to cause the necessary repetitions.

Moreover, the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.

The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements in the language exist in order to make it possible to perform a computation that must be based on this primitive construct: the assignment statement.

This world of statements is a disorderly one, with few useful mathematical properties. Structured programming can be seen as a modest effort to introduce some order into this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control.

Our fixation on von Neumann languages has continued the primacy of the von Neumann computer, and our dependency on *it* has made non-von Neumann languages uneconomical and has limited their development. The absence of full scale, effective programming styles founded on non-von Neumann principles has deprived designers of an intellectual foundation for new computer architectures.
...

2.7.2 Aside on the disorderly world of statements

Backus states that "the world of statements is a disorderly one, with few mathematical properties". Even in 1977 this was a bit overstated since work by Hoare on *axiomatic semantics* [Hoare 1969], by Dijkstra on the *weakest precondition (wp) calculus* [Dijkstra 1975], and by others had already appeared.

However, because of the referential transparency property of purely functional languages, reasoning can often be done in an equational manner within the context of the language itself. We examine this convenient approach later in this book.

In contrast, the *wp*-calculus and other axiomatic semantic approaches must project the problem from the world of programming language statements into the world of predicate calculus, which is much more orderly. We leave this study to courses on program derivation and programming language semantics.

2.7.3 Perspective from four decades later

In his Turing Award Address, Backus went on to describe FP, his proposal for a functional programming language. He argued that languages like FP would allow programmers to break out of the von Neumann bottleneck and find new ways of thinking about programming.

FP itself did not catch on, but the widespread attention given to Backus' address and paper stimulated new interest in functional programming to develop by researchers around the world. Modern languages like Haskell developed partly from the interest generated.

In the 21st Century, the software industry has become more interested in functional programming. Some functional programming features now appear in most mainstream programming languages (e.g. in Java 8+). This interest seems to driven primarily by two concerns:

- managing the complexity of large software systems effectively
- exploiting multicore processors conveniently and safely

The functional programming paradigm is able to address these concerns because of such properties such as referential transparency, immutable data structures, and composability of components. We look at these aspects in later chapters.

2.8 What Next?

This chapter introduced the concepts of abstraction and programming paradigm and surveyed the imperative, declarative, functional, and other paradigms.

The next chapter continues the discussion of programming paradigms by examining the object-oriented and related object-based paradigms.

The subsequent chapters use the functional programming language Haskell to illustrate general programming concepts and explore programming language design and implementation using interpreters.

2.9 Exercises

1. This chapter used Python 3 to illustrate the procedural paradigm. Choose a language such as Java, C, C++, or C#. Describe how it can be used to write programs in the procedural paradigm.
2. This chapter used Python 3 to illustrate the modular paradigm. For the same language chosen for previous exercise, describe how it can be used to write programs in the modular paradigm.
3. Repeat the previous two exercises with a different language.
4. This chapter used Haskell to illustrate the functional paradigm. Choose a language such as Java, Scala, Python 3, or C#. Describe how it can be used to write programs in the functional paradigm. Consider how well the language supports tail recursion.

2.10 Acknowledgements

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

- Abstraction (Section 2.2) from the "What is Abstraction?" section of my Data Abstraction notes [Cunningham 2018a], which I wrote originally for the first C++ (CSci 490) and Java-based (CSci 211) classes at UM in 1996 but expanded and adapted for other courses
- Discussion of the primary programming paradigms (Sections 2.3-2.6) from Chapter 1 of my *Notes on Functional Programming with Haskell* [Cunningham 2014]. I updated this content to expand the discussion of the paradigms to include examples.
- Motivating Functional Programming (Section 2.7) from Chapter 1 of my *Notes on Functional Programming with Haskell* [Cunningham 2014].

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

In Spring and Summer 2018, I reorganized and expanded the previous Fundamentals chapter into four chapters for the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* These are

Chapter 1, Evolution of Programming Languages; Chapter 2, Programming Paradigms (this chapter); Chapter 3, Object-based Paradigms; and Chapter 80, Review of Relevant Mathematics.

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

2.11 References

[**Backus 1978**] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Communications of the ACM*, Vol. 21, No. 8, pages 613–41, August 1978 (ACM Turing Award Lecture, 1977). [local]

[**Bird 1988**]**:** Richard Bird and Philip Wadler. *Introduction to Functional Programming*, First Edition, Prentice Hall, 1988.

[**Bird 1998**]**:** Richard Bird. *Introduction to Functional Programming using Haskell*, Second Edition, Prentice Hall, 1998.

[**Bird 2015**]**:** Richard Bird. *Thinking Functionally with Haskell*, Second Edition, Cambridge University Press, 2015.

[**Budd 1995**] Timothy A. Budd. *Multiparadigm Programming in Leda*, Addison-Wesley, 1995.

[**Cunningham 2014**]**:** H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1993-2014.

[**Cunningham 2018a**]**:** H. Conrad Cunningham. *Notes on Data Abstraction*, 1996-2018.

[**Dijkstra 1975**]**:** Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Communications of the ACM*, Vol. 18, No. 8, pp. 453-457, 1975. [local]

[**Hoare 1969**]**:** Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming, *Communications of the ACM*. Vol. 12, No. 10, pp. 576-580, 1969. [local]

[**Hudak 1989**] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages, *ACM Computing Surveys*, Vol. 21, No. 3, pp. 359-411, 1989. (Especially the "Introduction" section.) [local]

2.12 Terms and Concepts

Abstraction, procedural abstraction, data abstraction, interface, procedures, functions, methods; programming language paradigm, primary paradigms (imperative, declarative, functional, relational or logic language); other paradigms (procedural, modular, object-oriented, concurrent); program state, implicit versus explicit state, execution of commands versus evaluation of expressions, mutable versus immutable data structures, side effects, sequencing, recursion, referential

transparency, first-class values, first-order and higher-order functions, lexical scope, global versus local variables, public versus private features, information hiding, encapsulation, lexical closure; von Neumann computer, von Neumann language, worlds of expressions and statements, axiomatic semantics, weakest precondition calculus.