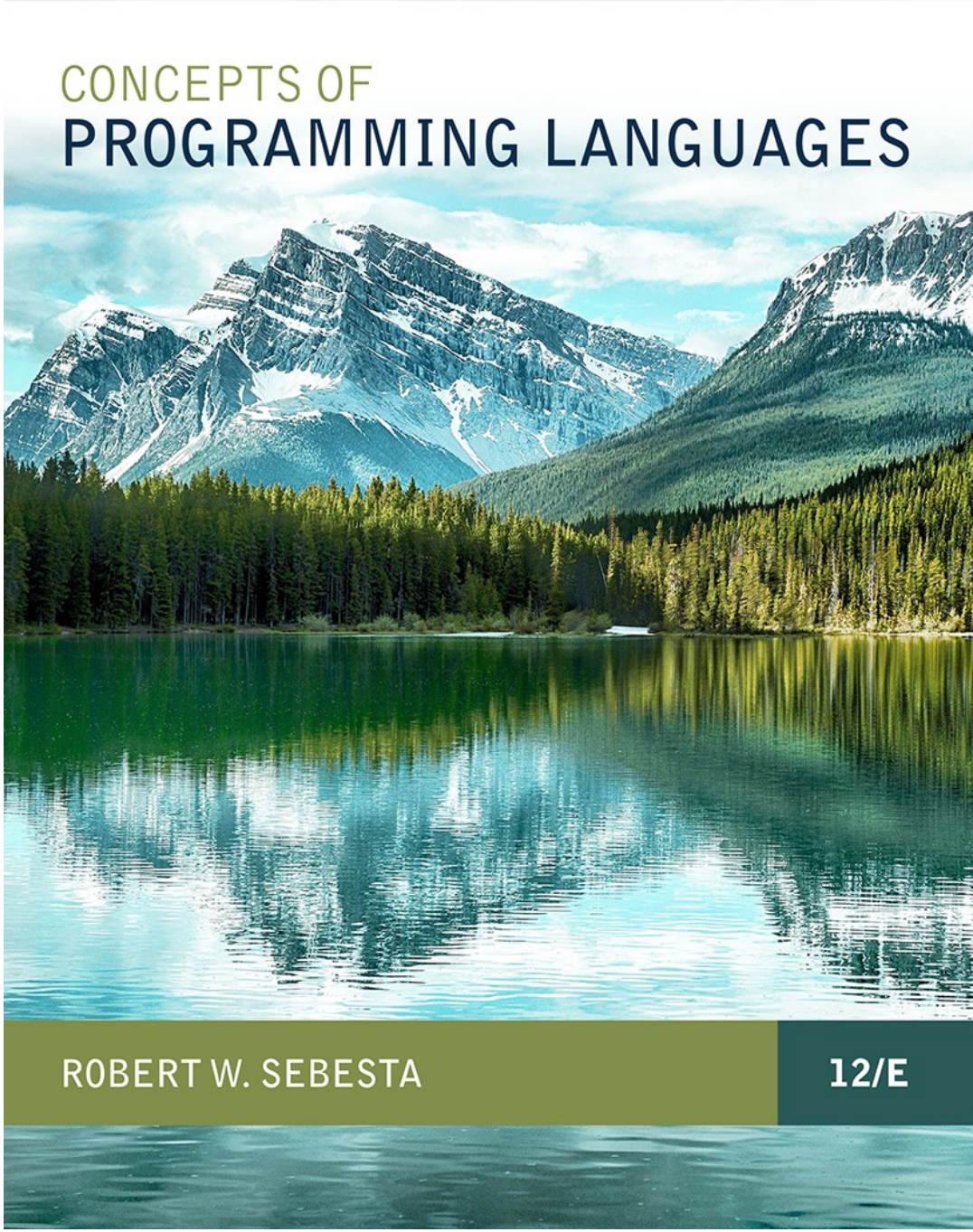
Chapter 15

Functional Programming Languages



ISBN 0-321-49362-1

Chapter 15 Topics

- Introduction
- Mathematical Functions
- Fundamentals of Functional Programming Languages
- Support for Functional Programming in Primarily Imperative Languages
- Comparison of Functional and Imperative Languages

Introduction

- The design of the imperative languages is based directly on the *von Neumann architecture*
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on *mathematical functions*
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Mathematical Functions

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*
- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

  `λ(x) x * x * x`

  for the function `cube(x) = x * x * x`

Lambda Expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

  e.g., `(λ(x) x * x * x)(2)`

  which evaluates to `8`

Functional Forms

- A higher-order function, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both

Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

  Form: $h \equiv f \circ g$

  which means $h(x) \equiv f(g(x))$

  For $f(x) \equiv x + 2$ and $g(x) \equiv 3 * x$,

  $h \equiv f \circ g$ yields $(3 * x) + 2$

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

  Form: α

  For `h(x) ≡ x * x`

  α`(h, (2, 3, 4))` **yields** `(4, 9, 16)`

Fundamentals of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
  - In an imperative language, operations are done and the results are stored in variables for later use
  - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics
- *Referential Transparency* – In an FPL, the evaluation of a function always produces the same result given the same parameters

Primitive Function Evaluation

- Parameters are evaluated, in no particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- The value of the last expression in the body is the value of the function

Currying

- Currying
  - Functions actually take just one parameter—if more are given, it considers the parameters a tuple (commas required)
  - Process of *currying* replaces a function with more than one parameter with a function with one parameter that returns a function that takes the other parameters of the original function
  - A function that takes more than one parameter can be defined in curried form by leaving out the commas in the parameters

```
fun add a b = a + b;
```

A function with one parameter, `a`. Returns a function that takes `b` as a parameter. Call: `add 3 5;`

Partial Evaluation

- Partial Evaluation
  - Curried functions can be used to create new functions by partial evaluation
  - Partial evaluation means that the function is evaluated with actual parameters for one or more of the leftmost actual parameters

```
fun add5 x add 5 x;
```

Takes the actual parameter `5` and evaluates the `add` function with `5` as the value of its first formal parameter. Returns a function that adds `5` to its single parameter

```
val num = add5 10;   (* sets num to 15 *)
```

Lazy Evaluation

- A language is *strict* if it requires all actual parameters to be fully evaluated
- A language is *nonstrict* if it does not have the strict requirement
- Nonstrict languages are more efficient and allow some interesting capabilities – *infinite lists*
- Lazy evaluation – Only compute those values that are necessary
- Positive numbers

```
positives = [0..]
```

- Determining if `16` is a square number

```
member [] b = False
member(a:x) b=(a == b)||member x b
squares = [n * n | n ← [0..]]
member squares 16
```

Tail Recursion

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster

Functional Programming Languages

- Lisp (LISt Processor): *first* functional programming language
  - Scheme
  - Common Lisp
- ML (Meta Language)
- Haskell: *purely* functional (e.g., no variables, no assignment statements, and no side effects of any kind)
- F#

Support for Functional Programming in Primarily Imperative Languages

- Support for functional programming is increasingly creeping into imperative languages
  - Anonymous functions (lambda expressions)
    - JavaScript: leave the name out of a function definition
    - C#: `i => (i % 2) == 0` (returns true or false depending on whether the parameter is even or odd)
    - Python: `lambda a, b : 2 * a  - b`

Support for Functional Programming in Primarily Imperative Languages (continued)

- Python supports the higher-order functions filter and map (often use lambda expressions as their first parameters)

```
map(lambda x : x ** 3, [2, 4, 6, 8])
```

Returns `[8, 64, 216, 512]`

- Python supports partial function applications

```
from operator import add
add5 = partial (add, 5)
```

(the first line imports add as a function)

Use: `add5(15)`

Comparing Functional and Imperative Languages

- Imperative Languages:
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed
- Functional Languages:
  - Simple semantics
  - Simple syntax
  - Less efficient execution
  - Programs can automatically be made concurrent

Summary

- Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution
- Haskell is a lazy functional language supporting infinite lists and list comprehension.
- Some primarily imperative languages now incorporate some support for functional programming
- Purely functional languages have advantages over imperative alternatives, but still are not very widely used