Functional programming Languages

Hecker, Tilevich

1

Pure Functional Languages

- The concept of assignment is not part of functional programming
  1. no explicit assignment statements
  2. variables bound to values only through parameter binding at functional calls
  3. function calls have no side-effects
  4. no global state
- Control flow: functional calls and conditional expressions
  - no iteration!
  - repetition through **recursion**

2

Referential transparency

Referential transparency: the value of a function application is independent of the context in which it occurs

- i.e., value of f(a, b, c) depends only on the values of f, a, b, and c
- value does not depend on global state of computation

- all variables in function must be local (or parameters)

3

Pure Functions

- Do not:
  - Reassign variables
  - Modify a data structure in place
  - Set a field on an object
  - Throw an exception or halt with an error*
  - Print to the console or read user input
  - Read from or write to a file
  - Draw on the screen
- No side-affects

4

Pure Functional Languages

All storage management is implicit
- copy semantics
- needs garbage collection

Functions are *first-class values*
- can be passed as arguments
- can be returned as values of expressions
- can be put in data structures
- unnamed functions exist as values

Functional languages are simple, elegant, not error-prone, and testable

5

FPLs vs imperative languages

- Imperative programming languages
  - Design is based directly on the von Neumann architecture
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- Functional programming languages
  - 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

6

Lambda expressions

- 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 describe *nameless functions*

7

Lambda expressions

- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression, as in

  `(λ(x) x * x * x)(3)`

  which evaluates to 27
- What does the following expression evaluate to?

  `(λ(x) 2 * x + 3)(2)`

8

Functional forms

- A functional form, or higher-order function, is one that either
  - takes functions as parameters,
  - yields a function as its result, or
  - both
- We consider 3 functional forms:
  - Function composition
  - Construction
  - Apply-to-all

9

Function composition

- A functional form that takes two functions as parameters and yields a function whose result is a function whose value is the first actual parameter function applied to the result of the application of the second.
- Form: `h ≡ f ⍰ g`
  which means `h(x) ≡ f(g(x))`
- If `f(x) = 2*x` and `g(x) = x – 1`
  then `f⍰g(3)= f(g(3)) = 4`

10

Construction

- A functional form that takes a list of functions as parameters and yields a list of the results of applying each of its parameter functions to a given parameter
- Form: `[f, g]`
- For `f(x) = x * x * x`
  and `g(x) = x + 3`,
  `[f, g](4)` yields `(64, 7)`

11

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 * x`,
  `α(h, (3,2,4))`
  yields `(27, 8, 64)`

12

Fundamentals of FPLs

- The objective of the design of a FPL is to mimic mathematical functions as much as 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 languages
  - In an FPL, variables are not necessary, as is the case in mathematics
- The evaluation of a function always produces the same result given the same parameters. This is called ***referential transparency***

13