FP Foundations, Scheme

In Text: Chapter 11

Outline

• Mathematical foundations
• Functional programming
• λ-calculus
• LISP
• Scheme
Imperative Languages

• We have been discussing imperative languages
  – C/C++, Java, and Pascal are imperative languages
  – They follow the von Neumann architecture [1]

Functional Programming

• A different way of looking at things
  – It is based on mathematical functions
  – It is supported by functional, and applicative, programming languages
    • LISP, ML, Haskell
Mathematical Foundations

- A **mathematical function** is a mapping of members from one set to another set
  - The “input” set is called the **domain**
  - The “output” set is called the **range**
- A mathematical function defines a value, rather than specifying a sequence of operations on values in memory to produce a value

Mathematical Foundations

- The evaluation order of mapping expressions is controlled by **recursion and conditional expressions**, rather than by the sequencing and iterative repetition
- Functions do not have states
  - They have no side effects
  - They always produce the same output given the same input parameters
Simple Functions

• Usual form:
  function name + a list of parameters in parentheses + mapping expression
• E.g., cube(x) = x * x * x, where
  – both the domain and range sets are real numbers, and
  – x can represent any member of the domain set, but it is fixed to represent one specific element during the expression evaluation

Function Application

• It is specified by paring the function name with a particular element of the domain set
• The range element is obtained by evaluating the function-mapping expression with the domain element substituted for the particular element
  – Cube(2.0) = 2.0 * 2.0 * 2.0 = 8.0
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

• Two common functional forms
  – Function composition
  – Apply-to-all

Function Composition

• Function composition has two functional parameters and yields a function whose value is the first function applied to the result of the second

• It is written as an expression, using a \( \circ \) operator (called “circle” or “round“)
  – E.g., \( h = f \circ g \)
  
  \[
  \text{if } f(x) = x + 2, \text{ and } \\
  g(x) = 3 \times x \\
  \text{then } h(x) = f(g(x)) = (3 \times x) + 2
  \]
Apply-to-all

- **Apply-to-all** takes a single function as a parameter
- If applied to a list of arguments, apply-to-all applies its functional parameter to each element of the list, and then collects results in a list or sequence
- It is denoted by $\alpha$
  - E.g., $h(x) = x \times x$, then $\alpha(h, (2, 3, 4)) = (4, 9, 16)$

Lambda expression

- Early theoretical work on functions separated the task of defining a function from that of naming the function
- Lambda notation, $\lambda$, provides a method for defining nameless functions
- A **lambda expression** is a function, which specifies the parameters, and the mapping expression
  - E.g., $\lambda(x)x \times x \times x$
Lambda-Calculus

• In the mid 1960s, Peter Landin observed that a complex programming language can be understood by formulating it as a tiny core calculus capturing the language’s essential mechanisms, together with a collection of convenient derived forms whose behavior is understood by translating them into the core.

Lambda-Calculus

• The core language used by Landin was the lambda-calculus, a formal system invented in the 1920s by Alonzo Church in which all computation is reduced to the basic operations of function definition and application.
factorial Example

• \text{factorial}(n) =
  \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{factorial}(n - 1)

• The corresponding \(\lambda\)-calculs term is:
  \text{factorial}(n) =
  \lambda n. \text{if } n=0 \text{ then } 1 \text{ else } n \times \text{factorial}(n - 1)

• Meaning
  – For each nonnegative number \(n\), instantiating the function with the argument \(n\) yields the factorial of \(n\) as result

\(\lambda\)-calculus

• Lambda-calculus embodies function definition and application in the purest possible form

• In the lambda-calculus, everything is a function
  – the arguments accepted by functions are themselves functions, and
  – the result returned by a function is another function
Syntax of $\lambda$-calculus

$t ::= x \quad \text{(a variable)}$
$\mid \lambda x.t \quad \text{(a function)}$
$\mid t \; t \quad \text{(function application)}$

- The syntax of lambda-calculus comprises three sorts of terms
  - Variable itself is a term
  - The abstraction of a variable $x$ from a term $t$ is a term
  - The application of term $t_1$ to another term $t_2$, is a term

Two conventions of writing lambda-terms

- Application is left associative
  - Given $s \; t \; u$, the calculation is $(s \; t) \; u$

\[
\begin{array}{c}
\text{apply} \\
\text{apply} \\
\text{apply} \\
s \\
\downarrow \\
t \\
\downarrow \\
u \\
\end{array}
\]
Two Conventions

• The body of abstraction is extended to right as much as possible
  – Given $\lambda x. \lambda y. x \ y \ x$, the calculation is $\lambda x. (\lambda y. ((x \ y) \ x))$

Scope

• An occurrence of the variable $x$ is said to be bound when it occurs in the body $t$ of an abstraction $\lambda x. \ t$
• An occurrence of $x$ is free if it appears in a position where it is not bound by an enclosing abstraction on $x$
  – In $x \ y$, and $\lambda y. x \ y$, $x$ is free
  – In $\lambda x. x$, and $\lambda z. \lambda x. \lambda y. x (y\ z)$, $x$ is bound