Chapter 15

Functional Programming Languages

SEVENTH EDITION

Robert W. Sebesta
Chapter 15 Topics

• Introduction
• Mathematical Functions
• Fundamentals of Functional Programming Languages
• The First Functional Programming Language: LISP
• Introduction to Scheme
• COMMON LISP
• Applications of Functional 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 following form
  \[ \lambda(x) \ x \ast x \ast x \]
for the function *cube* \( (x) = x \ast x \ast x \ast x \)
Lambda Expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression
e.g., \((\lambda (x) \ x \ * \ x \ * \ x) (2)\)
which evaluates to \(8\)
Functional form and Composition

- **Functional Form** (or say *higher-order function*): it is the 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 \( g(x) \equiv 3 \times x \),
  \( h \equiv f \circ g \) yields \((3 \times x) + 2\)
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: $\alpha$

For $h(x) \equiv x \times x$

$\alpha(h, (2, 3, 4))$ 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.
- In an FPL, the evaluation of a function always produces the same result given the same parameters.
LISP Data Types and Structures

• *Data object types*: originally only atoms and lists
• *List form*: parenthesized collections of sublists and/or atoms
  e.g., \((A \ B \ (C \ D) \ E)\)
• Originally, LISP was a typeless language
• LISP lists are stored internally as single-linked lists
LISP Interpretation

• Lambda notation is used to specify functions and function definitions. Function applications and data have the same form. e.g., If the list \((A \ B \ C)\) is interpreted as data it is a simple list of three atoms, \(A\), \(B\), and \(C\). If it is interpreted as a function application, it means that the function named \(A\) is applied to the two parameters, \(B\) and \(C\).

• The first LISP interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation.
Origins of Scheme

• A mid-1970s dialect of LISP, designed to be a cleaner, more modern, and simpler version than the contemporary dialects of LISP

• Uses only static scoping

• Functions are first-class entities
  – They can be the values of expressions and elements of lists
  – They can be assigned to variables and passed as parameters
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
Primitive Functions

• **Arithmetic:**
  - \(+, -, *, /, \text{ABS, SQRT, REMAINDER, MIN, MAX}\)
    
    e.g., \((+ 5 2)\) yields 7
    \((- 24 (* 4 3))\) yields 12

• **QUOTE** - takes one parameter; returns the parameter without evaluation
  
  - it is required because the Scheme interpreter, named \textsc{EVAL}, always evaluates parameters to function applications before applying the function. **QUOTE** is used to avoid parameter evaluation when it is not appropriate
  
  - it an be abbreviated with the apostrophe prefix operator
    \(' (A B)\) is equivalent to \((\text{QUOTE} (A B))\)
Function Definition: LAMBDA

• Lambda Expressions
  – Form is based on \( \lambda \) notation
  e.g., \((\text{LAMBDA } (x) (* \ x \ x))\)
  \(x\) is called a bound variable

• Lambda expressions can be applied
  e.g., \(( (\text{LAMBDA } (x) (* \ x \ x)) \ 7)\)
Special Form Function: **DEFINE**

- A Function for Constructing Functions

**DEFINE** - Two forms:

1. To bind a symbol to an expression
   
   e.g., `(DEFINE pi 3.141593)`
   
   `(DEFINE two_pi (* 2 pi))`

2. To bind names to lambda expressions
   
   e.g., `(DEFINE (square x) (* x x))`

Example use: `(square 5)`
Output Functions

- `(DISPLAY expression)`
- `(NEWLINE)`
Numeric Predicate Functions

- A predicate function is one that returns a Boolean value (#T or #F).

- Numeric predefined predicate functions in Scheme
  - =, <>, >, <, >=, <=
  - EVEN?, ODD?, ZERO?, NEGATIVE?
  - Return FALSE using empty list, (), instead of #F
  - Return Non-null list is interpreted as #T
Control Flow: IF

• Selection- the special form, **IF**
  
  \[(IF \text{ predicate then\_exp else\_exp})\]
  
  e.g.,
  
  \[(IF (<> \text{ count } 0)\]
  
  \[(/ \text{ sum count})\]
  
  0)
Control Flow: \textbf{COND}

• Multiple Selection - \textbf{COND}

General form:

\[
\text{(COND } \\
(\text{predicate}_1 \ \text{expr} \ \{\text{expr}\}) \\
(\text{predicate}_2 \ \text{expr} \ \{\text{expr}\}) \\
\ldots \\
(\text{predicate}_n \ \text{expr} \ \{\text{expr}\}) \\
(\text{ELSE } \text{expr} \ \{\text{expr}\}))
\]

• The predicates are evaluated one at a time, in order from the first, till one evaluates to \# T.

• The expressions that follow the first predicate that is found to be \#T are then evaluated, and its value is returned as the value of COND.
Example of COND

(DEFINE (compare x y)
    (COND
        ((> x y) (DISPLAY "x is greater than y"))
        ((< x y) (DISPLAY "y is greater than x"))
        (ELSE (DISPLAY "x and y are equal"))
    )
)
List Functions: **CONS** and **LIST**

- **CONS** - List constructor: it builds a list from its two arguments.
  - the first parameter: an atom or a list.
  - the second parameter: a list;
  - return: a new list that includes the first parameter as its first element and the second one as the remainder of its result
  - e.g., `(CONS 'A '(B C))` returns `(A B C)`
    `(CONS '(A B) '(C D))` returns `((A B) C D)`

- **LIST** - takes any number of parameters; returns a list with the parameters as elements
  - e.g, `(LIST 'apple 'orange 'egg)` returns `(apple orange egg)`
List Functions: **CAR** and **CDR**

- **CAR** takes a list parameter; returns the first element of that list
  
  e.g., \( \text{(CAR ' (A B C)) yields A} \)
  \( \text{(CAR ' ((A B) C D)) yields (A B)} \)

- **CDR** takes a list parameter; returns the list after removing its first element
  
  e.g., \( \text{(CDR ' (A B C)) yields (B C)} \)
  \( \text{(CDR ' ((A B) C D)) yields (C D)} \)
Predicate Function: \( \text{EQ} ? \)

- \( \text{EQ} ? \) takes two symbolic parameters; it returns \#T if both parameters are atoms and are the same

  e.g., \((\text{EQ}\ ?\ 'A \ 'A)\) yields \#T

  \((\text{EQ}\ ?\ 'A \ 'B)\) yields ()

  – Note that if \( \text{EQ} ? \) is called with list parameters, the result is not reliable

  • \((\text{EQ}\ ?\ '(A \ B) \ '(A \ B))\) returns () or \#T

  – Also \( \text{EQ} ? \) does not work for numeric atoms (which use predicate =)
Predicate Functions: \texttt{LIST?} and \texttt{NULL?}

- \textbf{LIST?} takes one parameter; it returns \texttt{#T} if the parameter is a list; otherwise ( )
  - (\texttt{LIST? ‘(X Y)}) return \texttt{#T}
  - (\texttt{LIST? ‘X}) return ( )
  - (\texttt{LIST? ‘()}) return \texttt{#T}

- \textbf{NULL?} takes one parameter; it returns \texttt{#T} if the parameter is the empty list; otherwise ( )
  - Note that \texttt{NULL?} returns \texttt{#T} if parameter is ( )
  - (\texttt{NULL? ‘(A B)}) returns ( )
  - (\texttt{NULL? ‘()}) returns \texttt{#T}
Example Scheme Function: \texttt{member}

- \texttt{member} takes an atom and a simple list; returns \texttt{#T} if the atom is in the list; \texttt{()} otherwise

\begin{verbatim}
DEFINE (member atm lis)
  (COND
    ((NULL? lis) '())
    ((EQ? atm (CAR lis)) #T)
    ((ELSE (member atm (CDR lis))))
  )
\end{verbatim}
Example Scheme Function: `equalsimp`

- `equalsimp` takes two simple lists as parameters; returns `#T` if the two simple lists are equal; `()` otherwise

```
(define (equalsimp lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) '())
    ((eq? (car lis1) (car lis2))
      (equalsimp (cdr lis1) (cdr lis2)))
    (else '()))
```
Example Scheme Function: `equal`

- `equal` takes two general lists as parameters; returns `#T` if the two lists are equal; `()` otherwise.

```
(DEFINE (equal lis1 lis2)
  (COND
    ((NOT (LIST? lis1)) (EQ? lis1 lis2))
    ((NOT (LIST? lis2)) '())
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) '())
    ((equal (CAR lis1) (CAR lis2))
      (equal (CDR lis1) (CDR lis2)))
    (ELSE '())
  ))
```
Example Scheme Function: append

- append takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

```
(define (append lis1 lis2)
  (cond
    ((null? lis1) lis2)
    (else (cons (car lis1)
                 (append (cdr lis1) lis2)))))
```
Example Scheme Function: \texttt{LET}

- General form:
  \begin{verbatim}
  (LET (  
        (name_1 expression_1)  
        (name_2 expression_2)  
        ...  
        (name_n expression_n))
  
  body
  )
  \end{verbatim}

- Evaluate all expressions, then bind the values to the names; evaluate the body
Example

(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
    (DISPLAY (+ minus_b_over_2a root_part_over_2a))
    (NEWLINE)
    (DISPLAY (- minus_b_over_2a root_part_over_2a)))
))
Scheme Functional Forms

• **Composition**
  - The previous examples have used it
  - \((\text{CDR } (\text{CDR } '(A \ B \ C)))\) returns \((C)\)

• **Apply to All** - one form in Scheme is **mapcar**
  - Applies the given function to all elements of the given list;
    - \((\text{DEFINE } (\text{mapcar } \text{fun} \ \text{lis})\))
      - \((\text{COND}\)
        - ((NULL? \ text{lis}) '())
        - (ELSE \ (\text{CONS} \ (\text{fun} \ (\text{CAR} \ \text{lis}))
          \ (\text{mapcar} \ \text{fun} \ (\text{CDR} \ \text{lis}))))\)
      - \))

• **Example of the use of** **mapcar**
  - \((\text{mapcar} (\text{LAMBDA} \ (\text{num}) (* \ \text{num} \ \text{num} \ \text{num} \ \text{num})) \ '(3 4 2 6))\)
  - This call returns \((27 \ 64 \ 8 \ 216)\)
Applications of Functional Languages

• LISP is used for artificial intelligence
  – Knowledge representation
  – Machine learning
  – Natural language processing
  – Modeling of speech and vision

• Scheme is used to teach introductory programming at a significant number of universities
Comparing Functional and Imperative Languages

• Imperative Languages:
  – Efficient execution
  – Complex semantics
  – Complex syntax
  – Concurrency is programmer designed

• Functional Languages:
  – Simple semantics
  – Simple syntax
  – Inefficient execution
  – Programs can automatically be made concurrent
Summary

• Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution instead of imperative features such as variables and assignments
• LISP began as a purely functional language and later included imperative features
• Scheme is a relatively simple dialect of LISP that uses static scoping exclusively
• CMMON LISP is a large LISP-based language
• Purely functional languages have advantages over imperative alternatives, but their lower efficiency on existing machine architectures has prevented them from enjoying widespread use