Chapter 15

Functional Programming Languages
Chapter 15 Topics

- Introduction
- Mathematical Functions
- Fundamentals of Functional Programming Languages
- The First Functional Programming Language: LISP
- Introduction to Scheme
- COMMON LISP
- ML
- Haskell
- 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 the following form:
  \[ \lambda(x) \ x \times x \times x \]
  for the function \( \text{cube}(x) = x \times x \times x \).
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 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 \( g(x) \equiv 3 \times x \),

\( h \equiv f \circ g \) 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
Referential Transparency

- 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: +, −, ∗, /, ABS, SQRT, REMAINDER, MIN, MAX
  e.g., (+ 5 2) yields 7
- QUOTE – takes one parameter; returns the parameter without evaluation
  - QUOTE is required because the Scheme interpreter, named EVAL, always evaluates parameters to function applications before applying the function. QUOTE is used to avoid parameter evaluation when it is not appropriate
  - QUOTE can be abbreviated with the apostrophe prefix operator
    ' (A B) is equivalent to (QUOTE (A B))
Function Definition: \textbf{LAMBDA}

\begin{itemize}
  \item Lambda Expressions
    \begin{itemize}
      \item Form is based on $\lambda$ notation
        \begin{itemize}
          \item \texttt{e.g.}, $\textsc{LAMBDA} \ (x) \ (* \ x \ x)$
        \end{itemize}
        \begin{itemize}
          \item \texttt{x} is called a bound variable
        \end{itemize}
    \end{itemize}
  \item Lambda expressions can be applied
    \begin{itemize}
      \item \texttt{e.g.}, $\textsc{(LAMBDA} \ (x) \ (* \ x \ x)) \ 7)$
    \end{itemize}
\end{itemize}
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)`
   
   Example use: `(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

- \#T is true and ( ) is false
- =, <>, >, <, >=, <=
- EVEN?, ODD?, ZERO?, NEGATIVE?
Control Flow: IF

- Selection– the special form, IF

  (IF predicate then_exp else_exp)

  e.g.,

  (IF (<> count 0)
       (/ sum count)
       0)
Control Flow: \texttt{COND}

- Multiple Selection – the special form, \texttt{COND}

General form:

\begin{verbatim}
(COND
  (predicate_1 expr {expr})
  (predicate_1 expr {expr})
  ...
  (predicate_1 expr {expr})
  (ELSE expr {expr}))
\end{verbatim}

- Returns the value of the last expr in the first pair whose predicate evaluates to true
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** takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result.
  - e.g., (CONS 'A ' (B C)) returns (A B C)

- **LIST** takes any number of parameters; returns a list with the parameters as elements.
  - (cons 1 (cons 2 (cons 3 nil)))
  - (list 1 2 3)
List Functions: CAR and CDR

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

- **CDR** takes a list parameter; returns the list after removing its first element

  e.g., \( \text{CDR } '(\text{A B C}) ) \) yields \( \text{(B C)} \)
  
  \( \text{CDR } '((\text{A B}) \text{ C D}) ) \) yields \( \text{(C D)} \)

IBM 704, car (Contents of Address Register) and cdr (Contents of Decrement Register)
Predicate Function: $\textit{EQ}$?

- $\textit{EQ}$? takes two expressions as parameters (usually two atoms); it returns $\#T$ if both parameters have the same pointer value; otherwise $\#F$

  e.g., $(\textit{EQ} \ ? \ 'A \ 'A)$ yields $\#T$
  $(\textit{EQ} \ ? \ 'A \ 'B)$ yields $\#F$
  $(\textit{EQ} \ ? \ 'A \ '(A \ B))$ yields $\#F$
  $(\textit{EQ} \ ? \ '(A \ B) \ '(A \ B))$ yields $\#T$ or $\#F$
  $(\textit{EQ} \ ? \ 3.4 \ (+ \ 3 \ 0.4)))$ yields $\#T$ or $\#F$
Predicate Functions: \texttt{LIST	exttt{?}} and \texttt{NULL	exttt{?}}

- \texttt{LIST	exttt{?}} takes one parameter; it returns \texttt{#T} if the parameter is a list; otherwise ()
- \texttt{NULL	exttt{?}} takes one parameter; it returns \texttt{#T} if the parameter is the empty list; otherwise ()
  - Note that \texttt{NULL	exttt{?}} returns \texttt{#T} if the parameter is ()
  - \((\texttt{NULL	exttt{?}} \ '(())\)) yields \texttt{#F}\)
Example Scheme Function: member

- member takes an atom and a simple list; returns #T if the atom is in the list; () otherwise
  (DEFINE (member atm lis)
   (COND
    ((NULL? lis) ()
     ((EQ? atm (CAR lis)) #T)
     ((ELSE (member atm (CDR lis)))
    ))
  )
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

```scheme
(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: \texttt{append}

\begin{itemize}
  \item \texttt{append} takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end
\end{itemize}

(\texttt{DEFINE (append lis1 lis2)}

(\texttt{COND}

  ((\texttt{NULL? lis1)} \texttt{lis2})

  (\texttt{ELSE (CONS (CAR lis1)}

    (\texttt{(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
**LET Example**

\[
\begin{align*}
-b \pm \sqrt{b^2 - 4ac} \\
2a
\end{align*}
\]

(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))
  )  
)

LET as shorthand of LAMBDA

(LET ((alpha 7)) (* 5 alpha))

((LABMDA (alpha) (* 5 alpha)) 7)
Scheme Functional Forms

• Composition
  – The previous examples have used it
  – (CDR (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;
    (DEFINE (mapcar fun lis)
      (COND
        ((NULL? lis) '())
        (ELSE (CONS (fun (CAR lis))
                     (mapcar fun (CDR lis)))))
    )
Functions That Build Code

- It is possible in Scheme to define a function that builds Scheme code and requests its interpretation
- This is possible because the interpreter is a user-available function, EVAL
Adding a List of Numbers

```
((DEFINE (adder lis)
    (COND
      ((NULL? lis) 0)
      (ELSE (EVAL (CONS '+ lis)))
    )))
```

- The parameter is a list of numbers to be added; `adder` inserts a `+` operator and evaluates the resulting list
  - Use `CONS` to insert the atom `+` into the list of numbers.
  - Be sure that `+` is quoted to prevent evaluation
  - Submit the new list to `EVAL` for evaluation
COMMON LISP

• A combination of many of the features of the popular dialects of LISP around in the early 1980s
• A large and complex language--the opposite of Scheme
• Features include:
  - records
  - arrays
  - complex numbers
  - character strings
  - powerful I/O capabilities
  - packages with access control
  - iterative control statements
ML

- A static-scoped functional language with syntax that is closer to Pascal than to LISP
- Uses type declarations, but also does *type inferencing* to determine the types of undeclared variables
- It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations
ML Specifics

• The `val` statement binds a name to a value (similar to `DEFINE` in Scheme)

• Function declaration form:

  ```ml
  fun name (parameters) = body;
  e.g., fun cube (x : int) = x * x * x;
  ```
Haskell

• Similar to ML (syntax, static scoped, strongly typed, type inferencing)
• Different from ML (and most other functional languages) in that it is *purely* functional (e.g., no variables, no assignment statements, and no side effects of any kind)
• Most Important Features
  – Uses *lazy evaluation* (evaluate no subexpression until the value is needed)
  – Has *list comprehensions*, which allow it to deal with infinite lists
Function Definitions with Different Parameter Forms

- Fibonacci Numbers

\[
\begin{align*}
\text{fib } 0 &= 1 \\
\text{fib } 1 &= 1 \\
\text{fib } (n + 2) &= \text{fib } (n + 1) + \text{fib } n
\end{align*}
\]
Guards

- Factorial

\[
\text{fact } n \\
\begin{align*}
| \ n == 0 &= 1 \\
| \ n > 0 &= n \times \text{fact}(n - 1)
\end{align*}
\]

- The special word \textit{otherwise} can appear as a guard
Lists

- List notation: Put elements in brackets
e.g., directions = [“north”,
                 “south”, “east”, “west”]

- Length: #
e.g., #directions is 4

- Arithmetic series with the .. Operator
e.g., [2, 4..10] is [2, 4, 6, 8, 10]

- Catenation is with ++
e.g., [1, 3] ++ [5, 7] results in [1, 3, 5, 7]

- CONS, CAR, CDR via the colon operator (as in Prolog)
e.g., 1: [3, 5, 7] results in [1, 3, 5, 7]
Factorial Revisited

\[
\text{product } [ ] = 1 \\
\text{product } (a:x) = a \times \text{product } x
\]

\[
\text{fact } n = \text{product } [1..n]
\]
List Comprehension

• Set notation

• List of the squares of the first 20 positive integers: 
  \[ [n \times n \mid n \leftarrow [1..20]] \]

• All of the factors of its given parameter:

  \[
  \text{factors } n = [i \mid i \leftarrow [1..n \text{ div 2}],
  \quad n \mod i == 0]
  \]
Quicksort

sort [] = []

sort (a:x) =
    sort [b | b ← x, b <= a] ++
    [a] ++
    sort [b | b ← x, b > a]
Lazy Evaluation

• Only compute those that are necessary
• Positive numbers
  
  \[
  \text{positives} = [0..]
  \]

• **Determining if 16 is a square number**

  \[
  \text{member~}[~]~b = \text{False}
  \]

  \[
  \text{member~}(a:x)~b = (a == b) \lor \text{member~}x~b
  \]

  \[
  \text{squares} = [n \times n | n \leftarrow [0..]]
  \]

  \[
  \text{member~squares~16}
  \]
Member Revisited

- The member function could be written as:
  ```
  member [ ] b = False
  member(a:x) b=(a == b) | member x b
  ```
  - However, this would only work if the parameter to squares was a perfect square; if not, it will keep generating them forever. The following version will always work:
  ```
  member2 (m:x) n
  | m < n = member2 x n
  | m == n = True
  | otherwise = False
  ```
Applications of Functional Languages

- APL is used for throw-away programs
- 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
- COMMON LISP is a large LISP-based language
- ML is a static-scoped and strongly typed functional language which includes type inference, exception handling, and a variety of data structures and abstract data types
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- Purely functional languages have advantages over imperative alternatives, but their lower efficiency on existing machine architectures has prevented them from enjoying widespread use