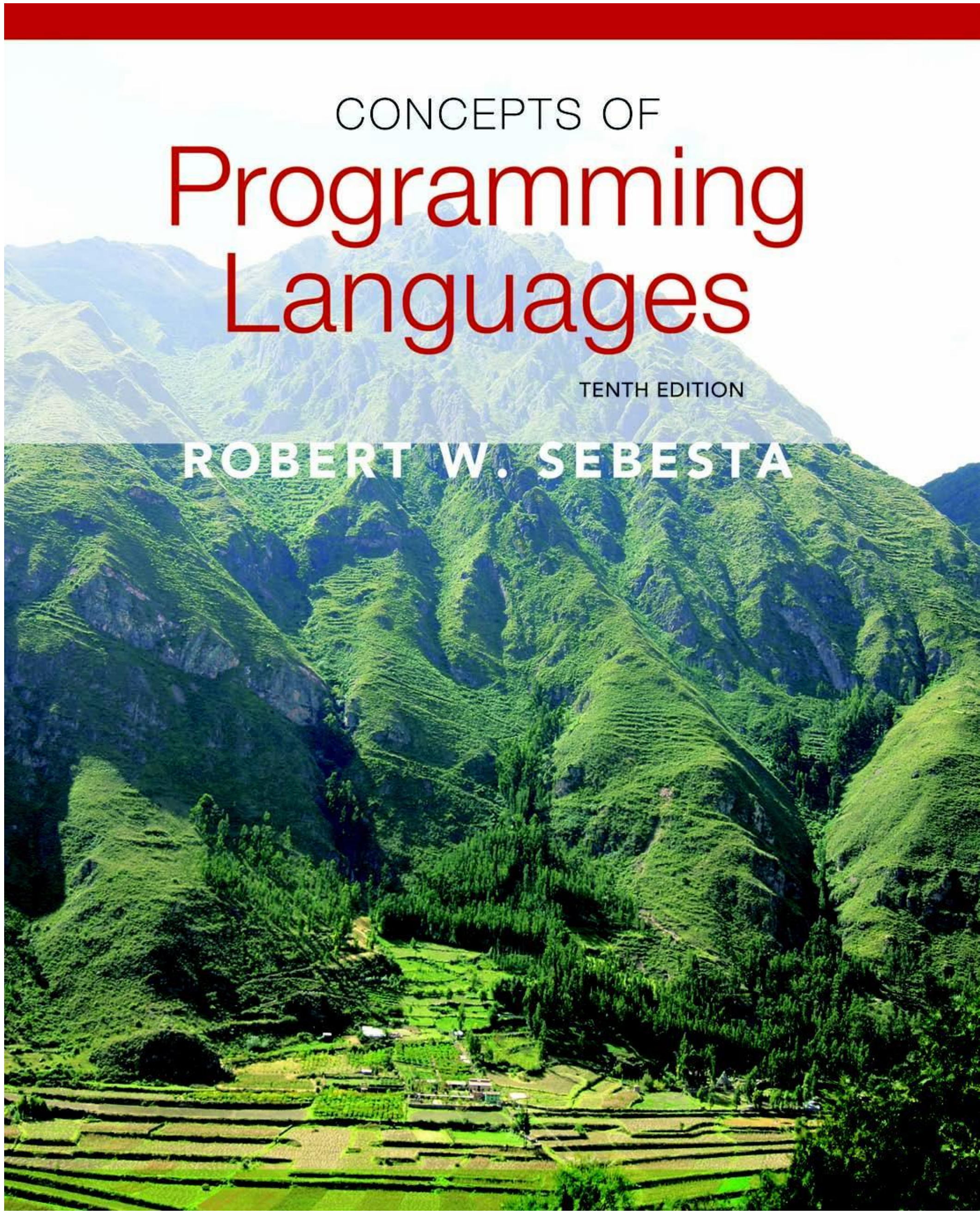
Chapter 15

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
- F#
- 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

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, passed as parameters, and returned from functions

The Scheme Interpreter

- In interactive mode, the Scheme interpreter is an infinite read-evaluate-print loop (REPL)
  - This form of interpreter is also used by Python and Ruby
- Expressions are interpreted by the function `EVAL`
- Literals evaluate to themselves

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

Primitive Functions & `LAMBDA` Expressions

- Primitive Arithmetic Functions: `+`, `-`, `*`, `/`, `ABS`, `SQRT`, `REMAINDER`, `MIN`, `MAX`
  e.g., `(+ 5 2)` yields `7`

- Lambda Expressions
  - Form is based on λ notation

  e.g., `(LAMBDA (x) (* x x)`

  `x` is called a bound variable

- Lambda expressions can be applied to parameters

  e.g., `((LAMBDA (x) (* x x)) 7)`

- `LAMBDA` expressions can have any number of parameters

  `(LAMBDA (a b x) (+ (* a x x) (* b x)))`

Special Form Function: `DEFINE`

- `DEFINE` – Two forms:
  1. To bind a symbol to an expression

     e.g., `(DEFINE pi 3.141593)`

     Example use: `(DEFINE two_pi (* 2 pi))`

     These symbols are not variables – they are like the names bound by Java's **`final`** declarations
  2. To bind names to lambda expressions (`LAMBDA` is implicit)

     e.g., `(DEFINE (square x) (* x x))`

     Example use: `(square 5)`

  - The evaluation process for `DEFINE` is different! The first parameter is never evaluated. The second parameter is evaluated and bound to the first parameter.

Output Functions

- Usually not needed, because the interpreter always displays the result of a function evaluated at the top level (not nested)
- Scheme has `PRINTF`, which is similar to the `printf` function of C
- Note: explicit input and output are not part of the pure functional programming model, because input operations change the state of the program and output operations are side effects

Numeric Predicate Functions

- `#T` (or `#t`) is true and `#F` (or `#f`) is false (sometimes `()` is used for false)
- `=`, `<>`, `>`, `<`, `>=`, `<=`
- `EVEN?`, `ODD?`, `ZERO?`, `NEGATIVE?`
- The `NOT` function inverts the logic of a Boolean expression

Control Flow

- Selection- the special form, `IF`

  `(IF` predicate then_exp else_exp`)`

```
    (IF (<> count 0)
        (/ sum count)
    )
```

- Recall from Chapter 8 the `COND` function:

```
(DEFINE (leap? year)
   (COND
      ((ZERO? (MODULO year 400)) #T)
      ((ZERO? (MODULO year 100)) #F)
      (ELSE (ZERO? (MODULO year 4)))
))
```

List Functions

- `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))`

- Recall that `CAR`, `CDR`, and `CONS` were covered in Chapter 6

List Functions (continued)

- Examples:

`(CAR '((A B) C D))` returns `(A B)`

`(CAR 'A)` is an error

`(CDR '((A B) C D))` returns `(C D)`

`(CDR 'A)` is an error

`(CDR '(A))` returns `()`

`(CONS '() '(A B))` returns `(() A B)`

`(CONS '(A B) '(C D))` returns `((A B) C D)`

`(CONS 'A 'B)` returns `(A . B)` (a *dotted pair*)

List Functions (continued)

- `LIST` is a function for building a list from any number of parameters

  `(LIST 'apple 'orange 'grape)` returns

  `(apple orange grape)`

Predicate Function: `EQ?`

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

  `(EQ? 'A 'A)` yields `#T`

  `(EQ? 'A 'B)` yields `#F`

  `(EQ? 'A '(A B))` yields `#F`

  `(EQ? '(A B) '(A B))` yields `#T` or `#F`

  `(EQ? 3.4 (+ 3 0.4)))` yields `#T` or `#F`

Predicate Function: `EQV?`

- `EQV?` is like `EQ?`, except that it works for both symbolic and numeric atoms; it is a value comparison, not a pointer comparison

  `(EQV? 3 3)` yields `#T`

  `(EQV? 'A 3)` yields `#F`

  `(EQV 3.4 (+ 3 0.4))` yields `#T`

  `(EQV? 3.0 3)` yields `#F` (floats and integers are different)

Predicate Functions: `LIST?` and `NULL?`

- `LIST?` takes one parameter; it returns `#T` if the parameter is a list; otherwise `#F`

  `(LIST? '())` yields `#T`

- `NULL?` takes one parameter; it returns `#T` if the parameter is the empty list; otherwise `#F`

  `(NULL? '(()))` yields `#F`

Example Scheme Function: member

- member takes an atom and a simple list; returns #T if the atom is in the list; #F otherwise

```
DEFINE (member atm a_list)
(COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR lis)) #T)
    ((ELSE (member atm (CDR a_list)))
))
```

Example Scheme Function: `equalsimp`

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

```
(DEFINE (equalsimp list1 list2)
(COND
   ((NULL? list1) (NULL? list2))
   ((NULL? list2) #F)
   ((EQ? (CAR list1) (CAR list2))
         (equalsimp(CDR list1)(CDR list2)))
   (ELSE #F)
))
```

Example Scheme Function: `equal`

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

```
(DEFINE (equal list1 list2)
   (COND
     ((NOT (LIST? list1))(EQ? list1 list2))
     ((NOT (LIST? lis2)) #F)
     ((NULL? list1) (NULL? list2))
     ((NULL? list2) #F)
     ((equal (CAR list1) (CAR list2))
            (equal (CDR list1) (CDR list2)))
     (ELSE #F)
))
```

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 list1 list2)
   (COND
     ((NULL? list1) list2)
     (ELSE (CONS (CAR list1)
                 (append (CDR list1) list2)))
))
```

Example Scheme Function: LET

- Recall that LET was discussed in Chapter 5
- LET is actually shorthand for a LAMBDA expression applied to a parameter

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

is the same as:

```
((LAMBDA (alpha) (* 5 alpha)) 7)
```

LET 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)))
  (LIST (+ minus_b_over_2a root_part_over_2a))
        (- minus_b_over_2a root_part_over_2a))
))
```

Tail Recursion in Scheme

- 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
- Scheme language definition requires that Scheme language systems convert all tail recursive functions to use iteration

Tail Recursion in Scheme – continued

- Example of rewriting a function to make it tail recursive, using helper a function

Original:

```
(DEFINE (factorial n)
   (IF (<= n 0)
       1
       (* n (factorial (- n 1)))
 ))
```

Tail recursive:

```
(DEFINE (facthelper n factpartial)
    (IF (<= n 0)
        factpartial
        facthelper((- n 1) (* n factpartial)))
  ))
(DEFINE (factorial n)
    (facthelper n 1))
```

Functional Form – Composition

- Composition
  - If `h` is the composition of `f` and `g`, `h(x) = f(g(x))`

    ```
    (DEFINE (g x) (* 3 x))
    (DEFINE (f x) (+ 2 x))
    (DEFINE h x) (+ 2 (* 3 x))) (The composition)
    ```

  - In Scheme, the functional composition function `compose` can be written:

    ```
    (DEFINE (compose f g) (LAMBDA (x) (f (g x))))
    ((compose CAR CDR) '((a b) c d)) yields c
     (DEFINE (third a_list)
         ((compose CAR (compose CDR CDR)) a_list))
    ```

    is equivalent to `CADDR`

Functional Form – Apply-to-All

- Apply to All – one form in Scheme is `map`
  - Applies the given function to all elements of the given list;

```
(DEFINE (map fun a_list)
  (COND
    ((NULL? a_list) '())
    (ELSE (CONS (fun (CAR a_list))
                (map fun (CDR a_list))))
))
```

`(map (LAMBDA (num) (* num num num)) '(3 4 2 6))` yields `(27 64 8 216)`

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 a_list)
   (COND
     ((NULL? a_list) 0)
     (ELSE (EVAL (CONS '+ a_list)))
))
```

- 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

Common LISP (continued)

- Macros
  - Create their effect in two steps:
    - Expand the macro
    - Evaluate the expanded macro
- Some of the predefined functions of Common LISP are actually macros
- Users can define their own macros with `DEFMACRO`

Common LISP (continued)

- Backquote operator (`` ` ``)
  - Similar to the Scheme's `QUOTE`, except that some parts of the parameter can be unquoted by preceding them with commas

    `` `(a (* 3 4) c) `` evaluates to `(a (* 3 4) c)`

    `` `(a ,(* 3 4) c) `` evaluates to `(a 12 c)`

Common LISP (continued)

- Reader Macros
  - LISP implementations have a front end called the *reader* that transforms LISP into a code representation. Then macro calls are expanded into the code representation.
  - A reader macro is a special kind of macro that is expanded during the reader phase
  - A reader macro is a definition of a single character, which is expanded into its LISP definition
  - An example of a reader macro is an apostrophe character, which is expanded into a call to `QUOTE`
  - Users can define their own reader macros as a kind of shorthand

Common LISP (continued)

- Common LISP has a symbol data type (similar to that of Ruby)
  - The reserved words are symbols that evaluate to themselves
  - Symbols are either bound or unbound
    - Parameter symbols are bound while the function is being evaluated
    - Symbols that are the names of imperative style variables that have been assigned values are bound
    - All other symbols are unbound

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
- Does not have imperative-style variables
- Its identifiers are untyped names for values
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations

ML Specifics

- A table called the *evaluation environment* stores the names of all identifiers in a program, along with their types (like a run-time symbol table)
- Function declaration form:

  fun *name* (*formal parameters*) = *expression;*

  e.g., `fun cube(x : int) = x * x * x;`

  - The type could be attached to return value, as in
    ```
    fun cube(x) : int = x * x * x;
    ```
  - With no type specified, it would default to `int` (the default for numeric values)
  - User-defined overloaded functions are not allowed, so if we wanted a `cube` function for real parameters, it would need to have a different name

ML Specifics (continued)

- ML selection

  if *expression* **then** *then_expression*
    else *else_expression*

  where the first expression must evaluate to a Boolean value
- Pattern matching is used to allow a function to operate on different parameter forms

```
fun fact(0) = 1
|   fact(1) = 1
|   fact(n : int) : int = n * fact(n – 1)
```

ML Specifics (continued)

- Lists

  Literal lists are specified in brackets

  `[3, 5, 7]`

  `[]` is the empty list

  `CONS` is the binary infix operator, `::`

  `4 :: [3, 5, 7]`, which evaluates to `[4, 3, 5, 7]`

  `CAR` is the unary operator `hd`

  `CDR` is the unary operator `tl`

```
fun length([]) = 0
|   length(h :: t) = 1 + length(t);

fun append([], lis2) = lis2
|   append(h :: t, lis2) = h :: append(t, lis2);
```

ML Specifics (continued)

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

```
val distance = time * speed;
```

  - As is the case with `DEFINE`, `val` is nothing like an assignment statement in an imperative language
  - If there are two `val` statements for the same identifier, the first is hidden by the second
  - `val` statements are often used in `let` constructs

```
let
   val radius = 2.7
   val pi = 3.14159
 in
   pi * radius * radius
 end;
```

ML Specifics (continued)

- `filter`
  - A higher-order filtering function for lists
  - Takes a predicate function as its parameter, often in the form of a lambda expression
  - Lambda expressions are defined like functions, except with the reserved word **`fn`**

```
filter(fn(x) => x < 100, [25, 1, 711, 50, 100]);
```

This returns `[25, 1, 50]`

ML Specifics (continued)

- `map`
  - A higher-order function that takes a single parameter, a function
  - Applies the parameter function to each element of a list and returns a list of results

    ```
    fun cube x = x * x * x;
     val cubeList = map cube;
     val newList = cubeList [1, 3, 5];
    ```

    This sets `newList` to `[1, 27, 125]`
  - Alternative: use a lambda expression

```
val newList = map (fn x => x * x * x, [1, 3, 5]);
```

ML Specifics (continued)

- Function Composition
  - Use the unary operator, o

```
val h = g o f;
```

ML Specifics (continued)

- Currying
  - ML 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
  - An ML 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;`

ML Specifics (continued)

- 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 *)
```

Haskell

- Similar to ML (syntax, static scoped, strongly typed, type inferencing, pattern matching)
- 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)

Syntax differences from ML

```
fact 0 = 1
fact 1 = 1
fact n = n * fact (n - 1)

fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

Function Definitions with Different Parameter Ranges

```
fact n
   |  n == 0 = 1
   |  n == 1 = 1
   |  n > 0 = n * fact(n - 1)


  sub n
    | n < 10       = 0
    | n > 100      = 2
    | otherwise    = 1


  square x = x * x
```

- Because Haskell support polymorphism, this works for any numeric type of `x`

Haskell 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
  e.g., `1:[3, 5, 7]` results in `[1, 3, 5, 7]`

Haskell (continued)

- Pattern Parameters

```
product [] = 1
product (a:x) = a * product x
```

  - Factorial:

```
fact n = product [1..n]
```

- List Comprehensions (Chapter 6)

```
[n * n * n | n <- [1..50]]
```

The qualifier in this example has the form of a *generator*. It could be in the form of a test

```
factors n = [i | i <- [1..n `div` 2], n `mod` i == 0]
```

The backticks specify the function is used as a binary operator

Quicksort

```
sort [] = []
sort (h:t) =
      sort [b | b ← t; b <= h]
    ++ [h] ++
      sort [b | b ← t; b > h]
```

Illustrates the concision of Haskell

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
```

Member Revisited

- The member function could be written as:

```
member b [] = False
member b (a:x)=(a == b) || member b x
```

- 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 n (m:x)
    | m < n = member2 n x
    | m == n = True
    | otherwise = False
```

F#

- Based on Ocaml, which is a descendant of ML and Haskell
- Fundamentally a functional language, but with imperative features and supports OOP
- Has a full-featured IDE, an extensive library of utilities, and interoperates with other .NET languages
- Includes tuples, lists, discriminated unions, records, and both mutable and immutable arrays
- Supports generic sequences, whose values can be created with generators and through iteration

F# (continued)

- Sequences

  ```
  let x = seq {1..4};;
  ```

  - Generation of sequence values is lazy

    ```
    let y = seq {0..10000000};;
    ```

    Sets `y` to `[0; 1; 2; 3;…]`

  - Default stepsize is `1`, but it can be any number

    ```
    let seq1 = seq {1..2..7}
    ```

    Sets `seq1` to `[1; 3; 5; 7]`

  - Iterators – not lazy for lists and arrays

  ```
  let cubes = seq {for i in 1..4 -> (i, i * i * i)};;
  ```

  Sets cubes to `[(1, 1); (2, 8); (3, 27); (4, 64)]`

F# (continued)

- Functions
  - If named, defined with `let`; if lambda expressions, defined with `fun`

```
(fun a b -> a / b)
```

  - No difference between a name defined with `let` and a function without parameters
  - The extent of a function is defined by indentation

```
let f =
      let pi = 3.14159
      let twoPi = 2.0 * pi;;
```

F# (continued)

- Functions (continued)
  - If a function is recursive, its definition must include the **`rec`** reserved word
  - Names in functions can be outscoped, which ends their scope

```
let x4 =
      let x = x * x
      let x = x * x
```

The first **`let`** in the body of the function creates a new version of `x`; this terminates the scope of the parameter; The second **`let`** in the body creates another `x`, terminating the scope of the second `x`

F# (continued)

- Functional Operators
  - Pipeline (|>)
  - A binary operator that sends the value of its left operand to the last parameter of the call (the right operand)

```
let myNums = [1; 2; 3; 4; 5]
let evensTimesFive = myNums
     |> List.filter (fun n -> n % 2 = 0)
     |> List.map (fun n -> 5 * n)
```

The return value is `[10; 20]`

ll

- ;lkj;

F# (continued)

- Functional Operators (continued)
  - Composition (>>)
    - Builds a function that applies its left operand to a given parameter (a function) and then passes the result returned from the function to its right operand (another function)

      The F# expression `(f >> g) x` is equivalent to the mathematical expression g(f(x))
- Curried Functions

```
let add a b = a + b;;
 let add5 = add 5;;
```

F# (continued)

- Why F# is Interesting:
  - It builds on previous functional languages
  - It supports virtually all programming methodologies in widespread use today
  - It is the first functional language that is designed for interoperability with other widely used languages
  - At its release, it had an elaborate and well-developed IDE and library of utility software

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)`

Support for Functional Programming in Primarily Imperative Languages (continued)

- Ruby Blocks
  - Are effectively subprograms that are sent to methods, which makes the method a higher-order subprogram
  - A block can be converted to a subprogram object with **`lambda`**

    ```
    times = lambda {|a, b| a * b}
    ```

    Use: `x = times.(3, 4)` (sets `x` to `12`)
  - Times can be curried with

    ```
    times5 = times.curry.(5)
    ```

    Use: `x5 = times5.(3)` (sets `x5` to `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
- 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 that uses type inference
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- F# is a .NET functional language that also supports imperative and object-oriented programming
- 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