


Sebesta, Concepts of Programming Languages, 4th Ed

Chapter 14:
Functional Programming Languages




Oregon State University
Open mind. Open doors.

Functional versus von Neumann languages

- ¥ The design of the imperative languages 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 based on mathematical functions
- ¥ A solid theoretical basis that is also to the user, but relatively unconcerned the architecture of the machines on which programs will run

CS 381 Prog Lang Chapter 14 2




Oregon State University
Open mind. Open doors.

Mathematical Functions

- **Def: A mathematical function is a mapping of members of one set, called the *domain set*, to another set, called the *range set***

Def: $\text{Square}(x) = x * x$
Square(3) yields the value 9
Square(3.14159) yields the value 9.86958

CS 381 Prog Lang Chapter 14 3




Oregon State University
Open mind. Open doors.

Using Functions to Build New Functions

One way to use functions as building blocks is to define new functions that build on existing functions

Def Cube(x) = Square(x) * x
Cube(3) yields the value 27



OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.


CS 381 Prog Lang Chapter 14 4

Result Composition

Another way to create new values using functions is to apply the result of one function as argument to another:

Cube(Square(3)) yields 729

(which is 3 to the 6th power, more on that in a moment)




OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 5

Fundamentals of FPLS

- 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**




OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 6

Computation 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




OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 7

Example Problem, Computing Average of an array of value

```
Function average (v : array)
var n, sum, i : integer;
Begin
  n := v.length;
  sum := 0;
  for (i := 0; i < n; i++)
    sum := sum + v[i]
  return sum / n;
end
```



OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.


CS 381 Prog Lang Chapter 14 8

Computation in a Functional language

- In an FPL, as in mathematics, variables are not necessary, only arguments

**Def average (list) =
sum(list) / length(list)**

Only identifiers are formal parameter names, no assignment statements.




OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 9

Reuse and Recycle


- **Complex problems are addressed by breaking them into smaller steps, which are then addressed in a functional fashion.**
- **Emphasis is on building a library of useful forms that can be combined in a variety of different ways.**

 CS 381 Prog Lang Chapter 14 10

Referential Transparency

- **In an FPL, the evaluation of a function always produces the same result given the same parameters**
- **This is called *referential transparency***


Square(cube(3)) is always 27, regardless how it is interpreted, either doing square first or cube first.

 CS 381 Prog Lang Chapter 14 11

Tools for Dealing with Functions

In order to create a programming language based on functions we need tools for dealing with functions:

- **Defining named functions**
- **Defining unnamed functions**
- **Combining functions to yield new functions**
- **Applying functions to values**


 CS 381 Prog Lang Chapter 14 12

Lambda Expression

- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

`lambda(x) x * x * x`

for the function `cube (x) = x * x * x`



OREGON STATE UNIVERSITY
Open mind. Open doors.


CS 381 Prog Lang Chapter 14 13

Using a Nameless Function

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)(3)`
which evaluates to 27




OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 14

Functional Forms

- Def: A *higher-order function*, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both
- Examples: composition, construction, apply to all
- (in Lisp) mapping, filtering, curry, reduction.



OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 15


Function Forms:
 1. *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 is $f \circ g$

- which means $h(x)$ is $f(g(x))$

Ex: `sixthPower` is `cube` \circ `square`




CS 381 Prog Lang Chapter 14 16

Function Forms
 2. *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 $f(x)$ is $x * x * x$ and $g(x)$ is $x + 3$,
 $[f, g](4)$ yields $(64, 7)$




CS 381 Prog Lang Chapter 14 17

Functional Forms
 3. *Apply to All (or Map)*

- 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: a
 For $h(x)$ is $x * x * x$

$a(h, (3, 2, 4))$ yields $(27, 8, 64)$




CS 381 Prog Lang Chapter 14 18

Function Forms
4. *Reduction*

A functional form that takes a binary function as parameter, an identity, and a list, and evaluates the function between every element of the list

Reduce(+, 0, (1 2 3 4)) yields 10




OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 19

Function Forms
5. *Curry*

A functional Form that fixes one argument of a binary function, resulting in a one argument function.

F = curry(+, 3)
F(7) yields 10



OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 20


LISP - the first functional programming language

LISP - LIST Processing language
Developed in the 1950's

Data Object Types: Atoms (Numbers and Symbols) and Lists

List Form: Parenthesized Collections of Atoms or Lists

(A 2 3 (B 3))




OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 21

Data and Functional Interpretation

A List, such as (A B C) can be interpreted in two ways

- As data it is a simple list of three elements
 - E.g. (2 3 4)
- As a function it means execute function A with arguments B and C
 - E.g. (+ 2 4)




CS 381 Prog Lang Chapter 14 22

Lisp dialects

Lots of versions of Lisp created in the last half century, at present two most common are:

- Scheme : minimal Lisp
- Common Lisp : Most portable, powerful Lisp




CS 381 Prog Lang Chapter 14 23

Features of Lisp

Atoms, Symbols

Atoms consist of numbers (integer and real), strings, and symbols.

2 3.14159 A "ABC"




CS 381 Prog Lang Chapter 14 24

Features of Lisp
Lists

A List is a collection of values, which can be atoms or other lists.

(2 3 4) (A B C)
(2 3 (A B C) (3 (3.14))) ()




OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 25

Two Special Values
t and nil

Two Values have special meaning

- t is boolean value true
- nil is boolean false, also empty list (!)



OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 26


Features of Lisp
Quote

Since a functional interpretation is default, quote is needed to avoid execution

'(A B C) 'A

'(+ 2 3) versus (+ 2 3)

Shorthand for (QUOTE (A B C))




OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 27

Features of LISP
Primitive Functions

1. Arithmetic: +, -, *, /, ABC, SQRT
2. Quote
3. Predicates
4. List Constructors and Destructors
5. Conditionals
6. Function Definition




CS 381 Prog Lang Chapter 14 28

Features of Lisp
Predicates

Predicates test an argument and return a boolean (t or nil)

- Atom - true if argument is atom
- Symbol
- Numberp
- Null
- > >= = <> < <= Even Odd zero

> (atom 2)
t




CS 381 Prog Lang Chapter 14 29

Features of Lisp
car and cdr

car takes a list as parameter and return first element
(car '(A B C)) yields 'A

cdr takes a list as parameter and returns new list with first element removed
(cdr '(A B C)) yields '(B C)

Names come from IBM 704 Machine Operations




CS 381 Prog Lang Chapter 14 30

Features of LISP
Cons

cons takes two parameters, the first can be either a list or an atom, the second must be a list; returns a new list with first argument as head and second argument as remainder

(cons 2 '(3 4)) yields (2 3 4)

(cons '(2 3) '(4 5)) yields ((2 3) 4 5)



CS 381 Prog Lang Chapter 14 31


Features of Lisp
conditional

Unlike imperative languages, conditional in Lisp returns a value. First argument is test, second is returned if expression is true, third is returned if expression is false

(if (< 2 7) 4 32) yields 4

(if (<> 3 7) '(a b c) '(2 7)) yields (2 7)

(Later we will see a more general conditional)




CS 381 Prog Lang Chapter 14 32

Features of Lisp
defun

defun is used to define a new function. Arguments are name, argument list, and result expression.

(defun addOne (x) (+ x 1))

(addOne 42) yields 43




CS 381 Prog Lang Chapter 14 33

Evaluation Process

Evaluation process for normal functions

1. Parameters are evaluated, in no particular order
2. The values of the parameters are substituted into the function body
3. The function body is evaluated
4. The value of the last expression in the body is the value of the function




CS 381 Prog Lang Chapter 14 34

Recursion is Fundamental

Since Lists are recursively defined, most Lisp functions are themselves recursive. Typical function is an if statement around the base case and the inductive case:

```
(defun length (lst)
  (if (null lst) 0 (+ 1 (length (cdr lst)))))
```




CS 381 Prog Lang Chapter 14 35

Thinking Recursively

In order to create recursive functions you need to always ask yourself:

- How do I identify the base case?
- What is the result in the base case?
- How do I reduce the general case to a simpler form?



CS 381 Prog Lang Chapter 14 36

Another Example, Sum

- Another example, sum of a list:
- What is base case? Empty list
- What is sum of an empty list? Zero:
- How do you reduce general case to something smaller? Add first element to sum of remainder of list:

```
(defun sum (lst)
  (if (null lst) 0
      (+ (car lst) (sum (cdr lst))))))
```

CS 381 Prog Lang Chapter 14 37

Another Example, Append

Append one list to another

Base case? Second list

Induction? Cons first element of first argument with append of remainder of first argument

```
(defun append (list1 list2)
  (if (null list1) list2
      (cons (car list1)
            (append (cdr list1) list2))))
```

CS 381 Prog Lang Chapter 14 38

Double RecursionSumAll

Our Sum function works for simple lists, but not for lists that contains lists, such as (2 (3 4))

To fix this, we add another base case, and two types of recursion:


```
(defun sumAll (lst)
  (if (null lst) 0
      (+ (if (atom (car lst)) (car lst)
            (sum (car lst))) (sum (cdr lst)))))
```

CS 381 Prog Lang Chapter 14 39

The Generalized Conditional

The Generalized Condition takes a list of test/value pairs, evaluates each test in turn, returns value of first one which is true

```
(cond
  (test value)
  (test value)
  (test value))
```




CS 381 Prog Lang Chapter 14 40

RewriteSumAll using cond

```
(defun sumAll (lst)
  (cond
    ((null lst) 0)
    ((atom (car lst))
     (+ (car lst) (sum (cdr lst))))
    (t (+ (sum (car lst))
          (sum (cdr lst))))))
```

(t is used as an else branch)




CS 381 Prog Lang Chapter 14 41

Another Use of cond

equalSimp

equalSimp - test two simple lists for =

```
(defun equalSimp (list1 list2)
  (cond
    ((null list1) (null list2))
    ((null list1) nil)
    ((eq (car list1) (car list2))
     (equalSimp (cdr list1)
                 (cdr list2)))
    (t nil)))
```




CS 381 Prog Lang Chapter 14 42

A Higher Order Function - Map

```
(defun map (fun lst)
  (if (null lst) nil
      (cons (funcall fun (car lst))
              (map fun (cdr lst))))))
```

Example: (map #'numberp '(2 A 3)) yields (t nil t)
(map #'square '(2 4 3 9)) yields (4 16 9 81)




CS 381 Prog Lang Chapter 14 43

Another higher order function - curry

A curry of a binary function binds one argument yielding a unary function

```
(defun currySecond (fun y)
  (function (lambda (x)
              (funcall fun x y))))
```


Example: (curry #'+ 2) yields a function that adds two to the argument



CS 381 Prog Lang Chapter 14 44

The filter functional

```
(defun filter (pred lst)
  (cond
    ((null lst) nil)
    ((funcall pred (car lst))
     (cons (car lst)
            (filter pred (cdr lst))))
    (t (filter pred (cdr lst)))))
```




CS 381 Prog Lang Chapter 14 45

Using Filter and Curry

Trace the execution of the following:

```
(defun smallThanfirst (lst)
  (filter (currySecond #'<
    (car lst)) (cdr lst)))


(smallerThanFirst '(4 2 5 3 7 6 2 4))
```

 CS 381 Prog Lang Chapter 14 46

Imperative Features of Lisp
(We won't use!)


Lisp does have some imperative features, if you really want to

- (set x y) assignment statement
- (set-car x y) or (replaca x y)
- (set-cdr! X y) or (replacd x y)
- (prog stmt1 stmt2 ... stmtn)


 CS 381 Prog Lang Chapter 14 47

Another Functional Language
- ML

- A static-scoped functional language with syntax 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 Lisp is essentially typeless) and has no type coercions
- Includes exception handling and a module facility for implementing abstract data types

 CS 381 Prog Lang Chapter 14 48

An Example ML Program




OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 49

Another Function language - 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)



OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.


CS 381 Prog Lang Chapter 14 50

An Example Haskell Function

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


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

- Note two styles of function definition



OSU
OREGON STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 51




Applications of Function Programming Languages

- **APL is used for matrix manipulation 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 number of universities**

OSHEA STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 52



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

OSHEA STATE UNIVERSITY
Open mind. Open doors.

CS 381 Prog Lang Chapter 14 53
