

## Functional Programming -- Mathematical Foundations

---

- **Simple Functions**

→ **function definition -- usual form:**

$\text{cube}(x) \equiv x * x * x$ , where  $x$  is a real number  
then, e.g.,  $\text{cube}(2) = 8$

→ **However, defining a function and naming the function are conceptually distinct**

→ **Lambda notation (Alonzo Church, 1941) provides for nameless functions:**

$\lambda(x). x * x * x$

→ **Can apply just like a named function:**

$(\lambda(x). x * x * x) (2) = 8$

slide 210

## Functional Programming -- Mathematical Foundations

---

- **Functional Forms**

→ **function composition:** has two function parameters, yields a function whose value is the first function applied to the result of the second

$h = f \circ g$  -- means apply  $g$  first, then apply  $f$  to the result

example: if  $f(x) = x + 2$

$g(y) = 3 * y$

then  $h(z) = f(g(z)) = (3 * z) + 2$

→ **construction:** takes a list of functions and applies each in turn to the argument, creating a list of results

written by enclosing function names in brackets, e.g.  $[g, h, i]$

example: if  $g(x) = x * x$

$h(x) = 2 * x$

$i(x) = x / 2$

then  $[g, h, i] (4)$  yields  $(16, 8, 2)$

→ **apply-to-all:** takes a single function and applies it to a list of arguments, creating a list of values

denoted by  $\alpha$

example: if  $h(x) = x * x$

then  $\alpha(h, (2, 3, 4))$  yields  $(4, 9, 16)$


slide 211

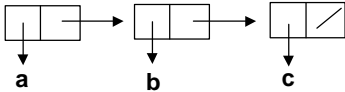
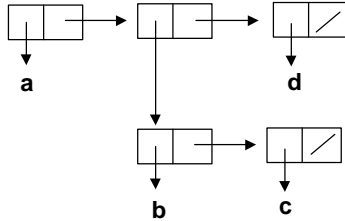
## Functional Programming

- **LISP: John McCarthy 1958 MIT**
  - List Processing => Symbolic Manipulation
- **Data Types**
  - Atoms : identifiers, symbols, numbers
  - Lists (Sexpressions)
    - (a b c d)
    - (a (b c) d e)
- **All data structures are single-linked nodes where each node has 2 pointers and represents a list element.**

slide 212

## Data Structures

- **Single atom:**

atom
- **List of atoms: (a b c)**
- **List containing list: (a (b c) d)**

slide 213

## LISP Primitives

---

- **quote => '**

(quote a) => 'a = a

(quote (a b c)) => '(a b c) = (a b c)

- **car : List => Sexp**

→ **One input arg => List**

→ **Returns first element of that list**

(car '(a b)) => a

(car '((a b) c)) => (a b)

(car '(a (b c))) => a

(car 'a) => undefined

(car '()) => undefined

slide 214

## LISP Primitives

---

- **cdr : List => List**

→ **One input arg => List**

→ **Returns list of all elements but the first element**

(cdr '(a b c)) => (b c)

(cdr '((a b) (c))) => ((c))

(cdr '(a)) => ()

(cdr 'a) => undefined

(cdr '()) => undefined

(cdr '55) => undefined

slide 215

## LISP Primitives

- **cons : Sexp X List => List**

→ 2 args as input: (cons a1 a2)

a1 : Sexp

a2 : List

→ Returns a2 with a1 inserted as its first element

(cons 'a '(b c)) => (a b c)

(cons 'a '()) => (a)

(cons '(a b) '((c d) e)) => ((a b) (c d) e)

→ Be careful, cons will take non list (atom) arguments and form "dotted" pairs, e.g.

(cons 'a 'b) => (a . b)

↑     ↑  
car   cdr

slide 216

## LISP Predicates

- Predicates are functions that return true (#t) or false (nil ())

→ the following return #t if the arguments are of the indicated type, and nil () otherwise

(symbol? 'a)    =>    #t

(symbol? '())   =>    ()

(number? '55)   =>    #t

(number? 55)    =>    #t

(atom? 'a)       =>    #t

\* (atom? '())    =>    #t

(atom? '(a))     =>    ()

(list? '(a))      =>    #t

(list? 'a)        =>    ()

\* (list? '())     =>    #t

(null? '())       =>    #t

(null? '(a c))    =>    ()

slide 217

## Additional Functions

---

- **eq? Sexp X Sexp => { #t , () }**
  - Returns true if objects are equal through pointer comparison. Guaranteed to work on symbols
- **equal? Sexp X Sexp => {#t , () }**
  - Recursively compares two objects to determine if they are equal (works on symbols, atoms, numbers, and lists.
- **= , < , > number X number => { #t , () }**
  - Performs numeric comparison on two numbers
- **+ , - , \* , / number X number => number**
  - Performs designated numeric operation

slide 218

## Control Flow

---

- **cond : (predicate Sexp)\* => eval ( Sexp )**
  - Evaluates ( predicate Sexp ) pairs.
  - Each predicate is evaluated in sequence until one is found to be true. The corresponding evaluated Sexp is returned.

```
(cond
  (Pred Sexp)
  (Pred Sexp)
  :
  (Pred Sexp)
  (t Sexp)    note: always true
)
```

```
(cond
  ((null? lis1) lis2)
  ((atom? (car lis1) (car lis1)))
  (t (cdr lis1))
)
```

slide 219

## Additional Functions

---

- **Additional control flow primitives that are available:**

```
(if Sexp1 Sexp2 [Sexp3])  
if (Sexp1) then Sexp2 [else Sexp3]
```

```
(while Sexp1 Sexp2 )  
while (Sexp1) do (Sexp2) od
```

- **Blocking primitive:**

```
(begin Sexp1 Sexp2 ... Sexpn )
```

- **Variable initialization primitive:**

```
(set! x Sexp)
```

- ***USE THESE FOR TESTING PURPOSES ONLY!***

slide 220

## Function Definition

---

```
(define (fctn_name arg1 arg2 ... argi )  
  Sexp  
)
```

```
(define (atom? atm)  
  (cond  
    ((list? atm) (null? atm))  
    ((symbol? atm) t)  
    (t nil))  
)
```

```
(define (equal? lis1 lis2)  
  (cond  
    ((atom? lis1) (eq? lis1 lis2))  
    ((atom? lis2) ())  
    ((equal? (car lis1) (car lis2))  
     (equal? (cdr lis1) (cdr lis2)))  
    (t nil))  
)
```

slide 221

## Function Definition

---

```
(define (member? atm lis)
  (cond
    ((null? lis) nil)
    ((eq? atm (car lis)) t)
    (t (member? atm (cdr lis)))
  )
)

(define (fac n)
  (cond
    ((eq? n 0) 1)
    (t (* n (fac (- n 1))))
  )
)

(define (append lis1 lis2)
  (cond
    ((null? lis1) lis2)
    (t (cons (car lis1)
              (append (cdr lis1) lis2)))
  )
)
```

slide 222

## Lambda Expressions

---

- Intuitively, *lambda expressions* allow one to define and use nameless functions and to pass them to be used in other functions

```
(lambda (lis) (car (cdr lis)))
```

- Given to a lisp interpreter, the above function definition returns the second element in a list, e.g.

```
((lambda (lis) (car (cdr lis))) '(a b c))
returns "b"
```

slide 223

## Lambda Expressions

- We CAN integrate the lambda expression into a function definition:

```
(define second
  (lambda (lis) (car (cdr lis))))
)
```

- Once "eval'd" by the interpreter, the function definition is "bound" to the name "second" such that

```
(second '(a b c)) => b
```

- But, our "standard" way of defining functions will work too ...

```
(define (second lis)
  (car (cdr lis)))
)
```

```
(second '(a b c)) => b
```

- **SO, WHAT DOES THE LAMBDA EXPRESSION BUY US?**

slide 224

## Lambda Expressions

- WE NOW HAVE THE CAPABILITY TO PASS FUNCTION DEFINITIONS AS PARAMETERS!

- Suppose that we want to write an "apply-to-all" function that takes a function definition and list as its arguments and applies its function argument to all elements in the list.

```
(define (mapcar fctn lis)
  (cond
    ((null? lis) nil)
    (t (cons (fctn (car lis))
              (mapcar fctn (cdr lis))))
  )
)
```

- then

```
(mapcar(lambda(num)(* num num)) '(2 4 6))
```

returns a list containing the square of all elements in the original list, i.e.,

```
(4 16 36)
```

slide 225



## Lambda Expressions

- Why not simply define a function “f” that performs a specified operation on one element and then pass IT to mapcar, e.g.

```
(define (square x)
  (* x x)
)
(define (mapcar fctn lis)
  (cond
    ((null? lis) nil)
    (t (cons (fctn (car lis))
              (mapcar fctn (cdr lis))))
  )
)
```

→ and then....

```
(mapcar square '(2 4 6)) ????
```

slide 226

## Scoping in LISP

- In reality, Lisp does allow one to define global and local variables, e.g.

```
(define x 5)           ; Global x
(set! x (car '(a b c))) ; Gbl/Lcl x
```

If (set! ...) *inside* a function defn

If (set! ...) *outside* a function defn

**In reality, Lisp does allow programs to reference "unbound" variables, e.g.**

```
(define (f atm)
  (cons x y)           ; y is an unbound variable
)                       ; x assumes the prev (set! x.... )
```

- What are the implications of these capabilities with respect to scoping?

slide 227

## Scoping in LISP

---

- Consider the following example:

```
(define (A ...)  
  ... (car X) ... ; unbound ref  
)
```

```
(define (B Fctn X)  
  ... Fctn ... ; invoke Fctn *  
)
```

```
(define (C X Z)  
  ... A ... ; invoke fctn A **  
  ... (B A Z) ... ; invoke fctn B ***  
)
```

slide 228

## Scoping in LISP

---

- Assuming that "our" Lisp is *statically* scoped (and most current Lisps are statically scoped), let's consider the impact of the following invocation of C:

```
(C '(i j) '(k l m))
```

- What is the binding of X in A after being invoked at \*\* ?
- What is the binding of X in A after being sent to B through the call \*\*\* and being invoked at \* ?
- Is it what you expected?

- What is needed is the ability to bind an execution environment at same time a function is passed as a parameter! (Funarg Problem)

- Solution: ... (B (function A) Z) ...

slide 229

## SCOPING IN LISP

---

- Consider the same scenario again:

```
(define (A ...)  
  ... (car X) ...  
)  
  
(define (B Fctn X)  
  ... Fctn ...  
)  
  
(define (C X Z)  
  ... A ...  
  ... (B (function A) Z) ...  
)
```

- In Pascal, static scoping and lexical scoping are effectively synonymous.
  - Although we are able to achieve "static" scoping through the use of the *function* primitive, is this also a "lexical" scoping?

slide 230

## XSCHEME

---

- XSCHEME is a dialect of LISP
- XSCHEME extends LISP
  - Minor syntax changes
    - will not affect us
  - Has extensions
    - additional functions  
we will not use
- We will use only the "pure" LISP parts of XSCHEME

slide 231

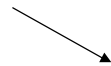
## XSCHEME: Helpful Hints

---

- Place “(exit)” at end of file
- Run “xscheme < infile > outfile”

- For debugging purposes only:

→ (if (< x y) x  
(if .....)



(begin

(write “x is”)

(write x)

(if (< x y) x

(if .....

) ; for begin

Allows you to  
display intermediate  
computations