• Simple Functions

→ function definition -- usual form:
  \[ \text{cube}(x) = x \times x \times x, \text{ where } x \text{ is a real number} \]
  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:
  \[ \lambda(x) \times x \times x \]

→ Can apply just like a named function:
  \( (\lambda(x) \times x \times x)(2) = 8 \)

• 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 \( f(x) = x + 2 \)
  \[ g(y) = 3 \times y \]
  then \( h(2) = f(g(2)) = (3 \times 2) + 2 \)

→ 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 \( g(x) = x \times x \)
  \[ h(x) = 2 \times x \]
  \[ i(x) = x / 2 \]
  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 \( h(x) = x \times x \)
  then \( \alpha(h, (2, 3, 4)) \) yields \((4, 9, 16)\)
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.

Data Structures

- Single atom: a (in atom space)

- List with one atom:

- List of atoms: (a b c)

- List containing list: (a (b c) d)
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

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
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)`

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? '(')) => ()`
    - `(atom? '(a)) => ()`
    - `(list? '(a)) => #t`
    - `(list? 'a) => ()`
    - *(list? '(')) => #t
    - `(null? '(')) => #t`
    - `(null? '(a c)) => ()`
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
  → Scheme provides exact and inexact numbers

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)
     (else Sexp)
     )
    )
    (cond
     ((null? lis1) lis2)
     ((atom? (car lis1)) (car lis1))
     (else (cdr lis1))
     )
Additional Functions

- **Additional control flow primitives that are available:**
  
  ```scheme
  (if Sexp1 Sexp2 [Sexp3])
  if (Sexp1) then Sexp2 [else Sexp3]
  
  (while Sexp1 Sexp2)
  while (Sexp1) do (Sexp2) od
  
  **Blocking primitive:**
  
  ```scheme
  (begin Sexp1 Sexp2 ... Sexpn)
  ```

- **Variable initialization primitive:**
  
  ```scheme
  (set! x Sexp)
  ```

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

Function Definition

```scheme
(define (fctn_name arg1 arg2 ... argi)
  SEXP)
```

```scheme
(define (atom? atm)
  (cond
   ((list? atm) (null? atm))
   ((symbol? atm) #t)
   (else ()))
)
```

```scheme
(define (equal? lis1 lis2)
  (cond
   ((atom? lis1) (eq? lis1 lis2))
   ((atom? lis2) ())
   ((equal? (car lis1) (car lis2))
    (equal? (cdr lis1) (cdr lis2)))
   (else ()))
)
```
Function Definition

(define (member? atm lis)
  (cond
    ((null? lis) ())
    ((eq? atm (car lis)) #t)
    (else (member? atm (cdr lis)))))

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

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

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"
Chapter 13, Slide 15

Lambda Expressions

- We CAN integrate the lambda expression into a function definition:
  
  ```scheme
  (define second
    (lambda (lis) (car (cdr lis))))
  )
  ```

  → Once "evaled" by the interpreter, the function definition is "bound" to the name "second" such that

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

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

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

  → SO, WHAT DOES THE LAMBDA EXPRESSION BUY US?

Chapter 13, Slide 16

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.

  ```scheme
  (define (mapcar fctn lis)
    (cond
      ((null? lis) ())
      (else (cons (fctn (car lis))
                  (mapcar fctn (cdr lis))))
    )
  )
  ```

  → then

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

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

  ```scheme
  (4 16 36)
  ```
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.

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

→ and then....

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

Scoping in LISP

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

  ```lisp
  (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.

  ```lisp
  (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?
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 *** 
  )

When McCarthy’s students executed

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

- Results at **:

  A  i (car '(l j))

- Results at *** (which takes us to *):

  A  k (car '(k l m))

- Not what was expected

  - Expected A  i both times
  - Need to bind referencing environment
  - “FunArg” problem
Solution to FunArg Problem

- \( (B \text{ (function } A) \text{ } Z) \)
  - \textit{function} is a command that passes a reference to the current environment for \( A \) to execute in

- Now execute:

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

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

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

Scoping in LISP

- Consider the following

  (define (A ...)
    ... (cons (car X) Y) ... ; unbound ref
  )

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

  (define (C X Z)
    ... A ... ; invoke fctn A **
    ... (B (function A) Z '(m n)) ... ; invoke fctn B ***
  )

  (C '(i j) '(k l m)) \text{ A } (i \text{ m n})

- Note: 2 different binding approaches
  - \( i \) via function
  - \( mn \) dynamic
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?

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?

Global table of variables

Implicit declaration

Types can change dynamically
  → Requires type checking at runtime
  → but...

We can associate variable references with declarations at compile time!

So.... Static scoping, but **must** have dynamic type checking!

Why? (#2)