Functional programming Languages

And a brief introduction to Lisp and Scheme

Pure Functional Languages

- The concept of assignment is not part of functional programming
  1. no explicit assignment statements
  2. variables bound to values only through parameter binding at functional calls
  3. function calls have no side-effects
  4. no global state

- Control flow: functional calls and conditional expressions
  - ⇒ no iteration!
  - ⇒ repetition through **recursion**

Referential transparency

Referential transparency: the value of a function application is independent of the context in which it occurs

- i.e., value of f(a, b, c) depends only on the values of f, a, b, and c
- value does not depend on global state of computation

⇒ all variables in function must be local (or parameters)

Pure Functional Languages

All storage management is implicit

- copy semantics
- needs garbage collection

Functions are *first-class values*

- can be passed as arguments
- can be returned as values of expressions
- can be put in data structures
- unnamed functions exist as values

Functional languages are simple, elegant, not error-prone, and testable

FPLs vs imperative languages

- Imperative programming languages
  - Design is based directly on the von Neumann architecture
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- Functional programming languages
  - 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

Lambda expressions

- 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 describe *nameless functions*

Lambda expressions

- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression, as in

```
(λ(x)  x * x * x)(3)
```

  which evaluates to 27

- What does the following expression evaluate to?

```
(λ(x)  2 * x + 3)(2)
```

Functional forms

- A functional form, or higher-order function, is one that either
  - takes functions as parameters,
  - yields a function as its result, or
  - both
- We consider 3 functional forms:
  - Function composition
  - Construction
  - Apply-to-all

Function 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 ≡ f ▯ g`
  which means `h(x) ≡ f(g(x))`
- If `f(x) = 2*x` and `g(x) = x – 1`
  then `f▯g(3)= f(g(3)) = 4`

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) = x * x * x`
  and `g(x) = x + 3`,
  `[f, g](4)` yields `(64, 7)`

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 * x,`

  `α(h, (3,2,4))`

  yields `(27, 8, 64)`

Fundamentals of FPLs

- The objective of the design of a FPL is to mimic mathematical functions as much as 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 languages
  - In an FPL, variables are not necessary, as is the case in mathematics
- The evaluation of a function always produces the same result given the same parameters. This is called ***referential transparency***

LISP

- Functional language developed by John McCarthy in the mid 50’s
- Semantics based on the *lambda-calculus*
- All functions operate on lists or symbols (called **S-expressions**)
- Only 6 basic functions
  - list functions: cons, car, cdr, equal, atom
  - conditional construct: cond
- Useful for list processing
- Useful for Artificial Intelligence applications: programs can read and generate other programs

Common LISP

- Implementations of LISP did not completely adhere to semantics
- Semantics redefined to match implementations
- Common LISP has become the standard
  - committee designed language (c. 1980s) to unify LISP variants
  - many defined functions
  - simple syntax, large language

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

Basic workings of LISP and Scheme

- Expressions are written in prefix, parenthesised form:

```
1 + 2  => (+ 1 2)
2 * 2 + 3 => (+ (* 2 2) 3)
(func arg1 arg2… arg_n)
(length ‘(1 2 3))
```

Operational semantics: to evaluate an expression:

- evaluate *func* to a function value
- evaluate each *arg_i* to a value
- apply the function to these values

S-expression evaluation

Scheme treats a parenthetic S-expression as a function application

```
(+ 1 2)
value: 3
(1 2 3)
;error: the object 1 is not applicable
```

Scheme treats an alphanumeric atom as a variable (or function) name

```
a
;error: unbound variable: a
```

Constants

To get Scheme to treat S-expressions as constants rather than function applications or name references, precede them with a ’

```
‘(1 2 3)
value: (1 2 3)
‘a
value: a
```

’ is shorthand for the pre-defined function *quote:*

```
(quote a)
value: a
(quote (1 2 3))
value: (1 2 3)
```

Conditional evaluation

If statement:

```
(if <conditional-S-expression>
    <then-S-expression>
    <else-S-expression>  )

(if  (> x 0)     #t#f )
    (if (> x 0)
          (/ 100 x)
          0
    )
```

Conditional evaluation

Cond statement:

```
(cond (<conditional-S-expression1>  <then-S-expression1>)
        ...
      (<conditional-S-expression_n>  <then-S-expression_n>)
   [ (else <default-S-expression>)   ] )
```

```
(cond  ( (> x 0)  (/ 100 x) )
       ( (= x 0)  0 )
       ( else     (* 100 x) ) )
```

Defining functions

```
(define (<function-name>  <param-list> )
   <function-body-S-expression>
)
```

E.g.,

```
      (define (factorial x)
         (if (= x 0)
      1
      (*  x   (factorial (- x 1)) )
         )
      )
```

Some primitive functions

- **`CAR`** returns the first element of its list argument:

  `(car '(a b c))` returns `a`

- **`CDR`** returns the list that results from removing the first element from its list argument:

  `(cdr '(a b c))` returns `(b c)`

  `(cdr '(a))` returns `()`

- **`CONS`** constructs a list by inserting its first argument at the front of its second argument, which should be a list:

  `(cons 'x '(a b))` returns `(x a b)`

Scheme lambda expressions

- Form is based on λ notation:

```
        (LAMBDA (L) (CAR (CAR L)))
```

  The L in the expression above is called a ***bound variable***

- Lambda expressions can be applied:

```
((LAMBDA (L) (CAR (CAR L))) '((A B) C D))
```

  The expression returns **A** as its value.

Defining functions in Scheme

- The Scheme function DEFINE can be used to define functions. It has 2 forms:
  - To bind a symbol to an expression:

```
        (define pi 3.14159)
        (define two-pi (* 2 pi))
```

  - To bind names to lambda expressions:

```
        (define (cube x) (* x x x))
       ; Example use: (cube 3)
```

  - Alternative way to define the cube function:

```
(define cube (lambda (x) (* x x x)))
```

Expression 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 that is evaluated is the value of the function
- Note: special forms use a different evaluation process

Map

Map is pre-defined in Scheme and can operate on multiple list arguments

```
> (map + '(1 2 3) '(4 5 6))
(5 7 9)

> (map + '(1 2 3) '(4 5 6) '(7 8 9))
(12 15 18)

> (map   (lambda (a b) (list a b))    '(1 2 3)    '(4 5 6))
((1 4) (2 5) (3 6))
```

Scheme functional forms

- Composition—the previous examples have used it:

```
(cube (* 3 (+ 4 2)))
```

- Apply-to-all—Scheme has a function named `mapcar` that applies a function to all the elements of a list. The value returned by `mapcar` is a list of the results.

  Example: `(mapcar cube '(3 4 5))` produces the list `(27 64 125)` as its result.

Scheme functional forms

- 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**
- For example, suppose we have a list of numbers that must be added together

```
(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. For example,
`(adder '(1 2 3 4))` returns the value 10.

The Scheme function APPLY

- **`APPLY`** invokes a procedure on a list of arguments:

```
(APPLY + '(1 2 3 4))
```

returns the value 10.

Imperative features of Scheme

- **`SET!`** binds a value to a name
- **`SETCAR!`** replaces the car of a list
- **`SETCDR!`** replaces the cdr of a list

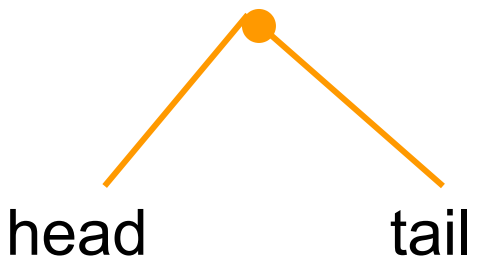
A sample Scheme session

```
[1] (define a '(1 2 3))
A
[2] a
(1 2 3)
[3] (cons 10 a)
(10 1 2 3)
[4] a
(1 2 3)
[5] (set-car! a 5)
(5 2 3)
[6] a
(5 2 3)
```

Lists in Scheme

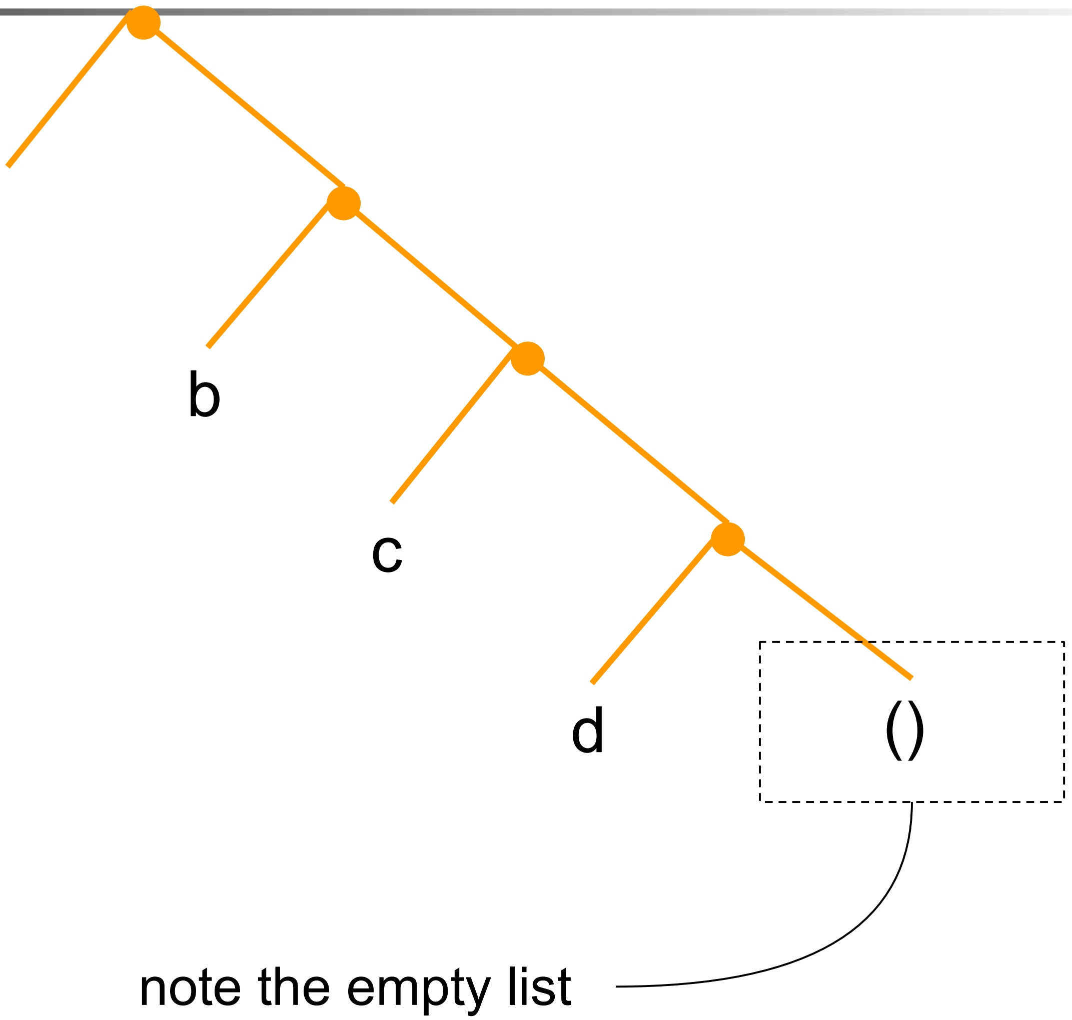
A **list** is an S-expression that isn't an atom

Lists have a tree structure:



List examples

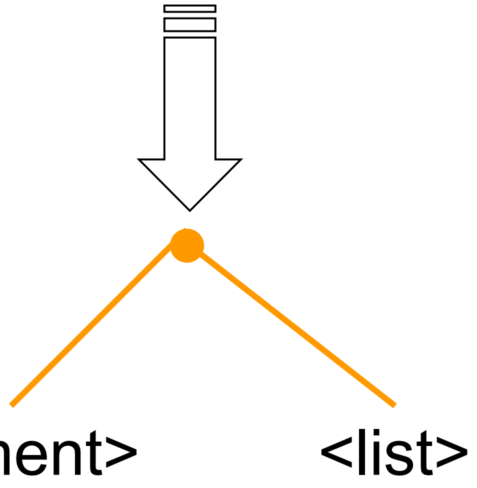
(a b c d)



Building Lists

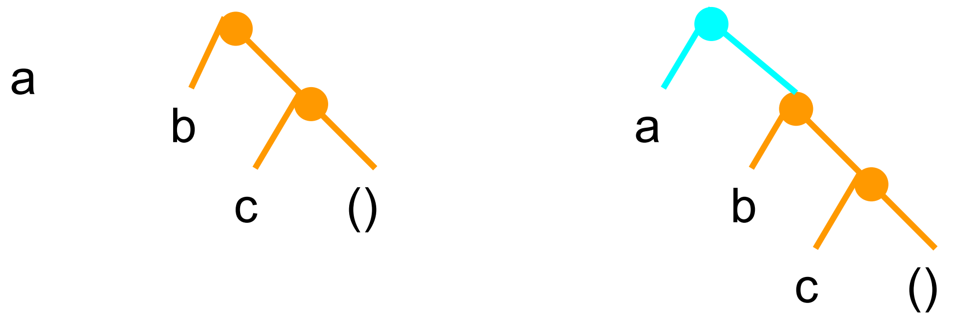
Primitive function: *cons*

(cons <element> <list>)

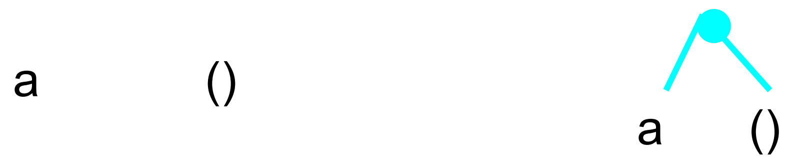


Cons examples

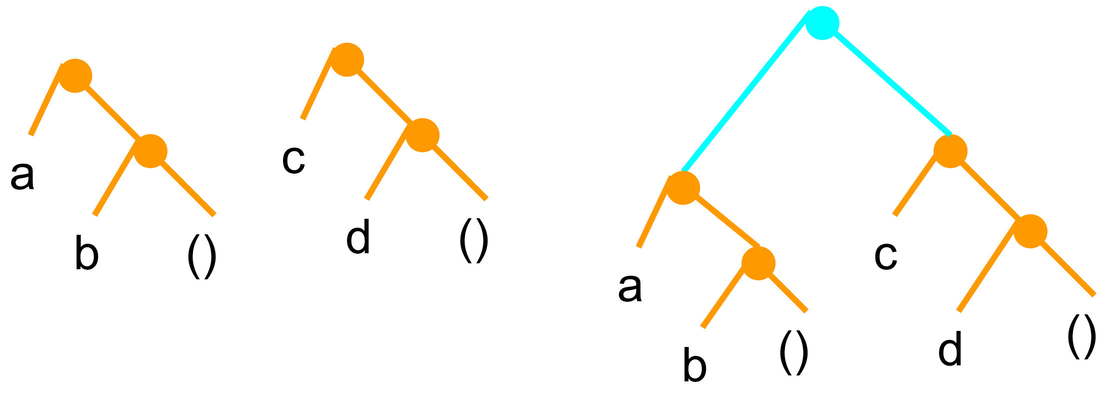
(cons 'a '(b c)) = (a b c)



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



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

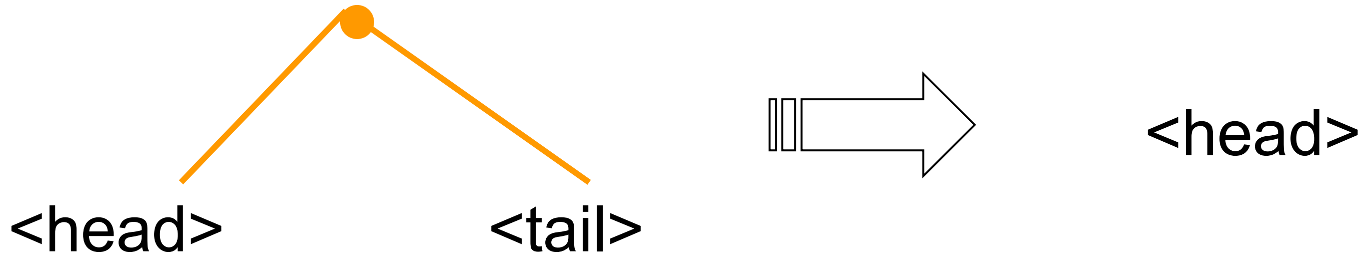


Accessing list components

Get the head of the list:

Primitive function: *car*

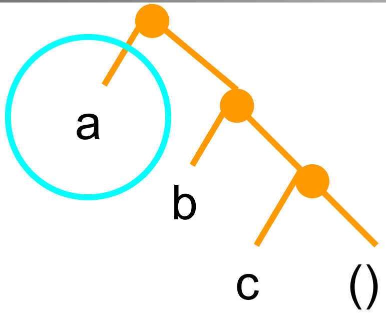
(car <list>)



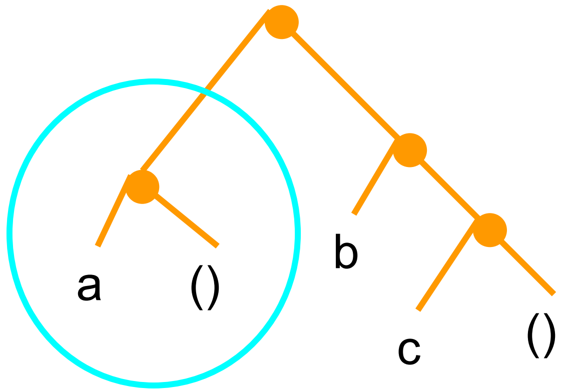
(i.e., *car* selects left sub-tree)

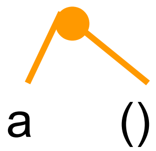
Car examples

(car ‘(a b c)) = a



(car ‘((a) b c)) = (a)

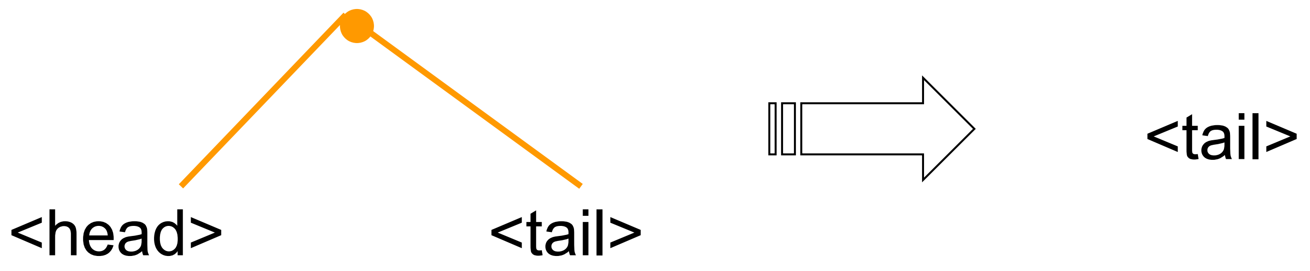




Accessing list components

Get the tail of the list:
Primitive function: *cdr*

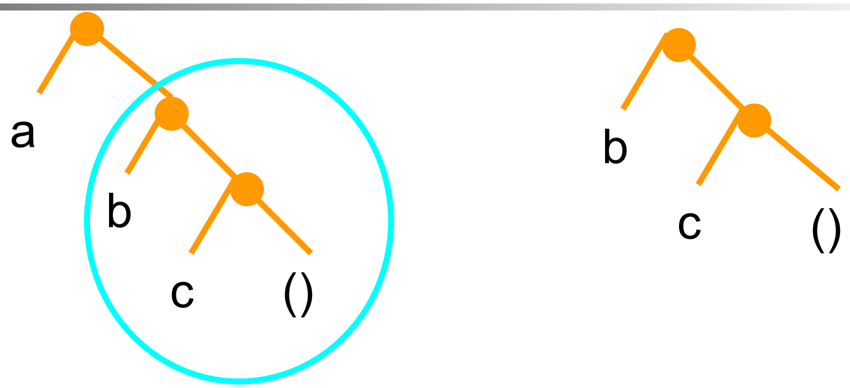
(cdr <list>)



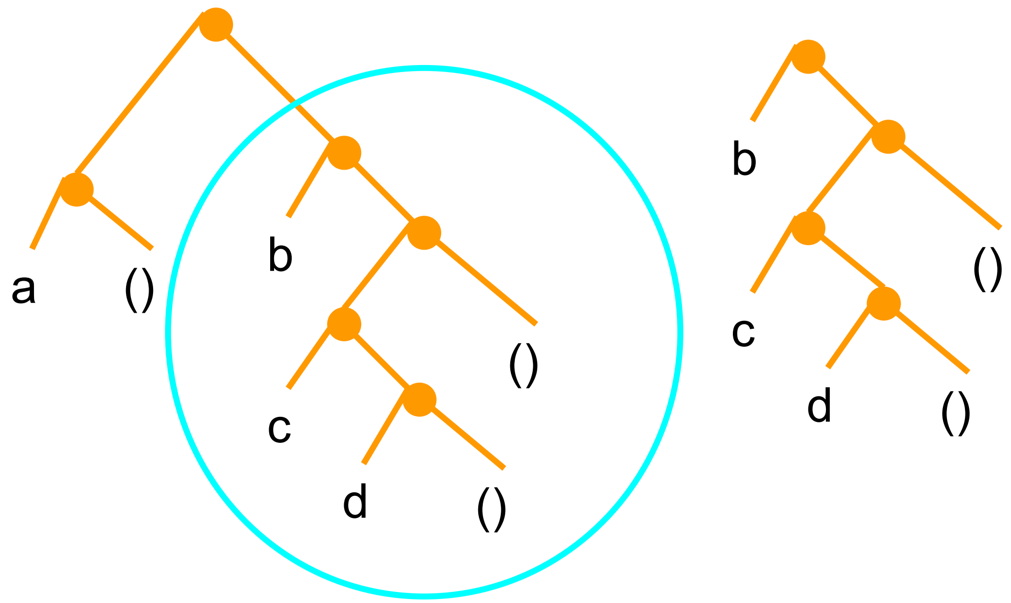
(i.e., *cdr* selects right sub-tree)

Cdr examples

(cdr ‘(a b c)) = (b c)



(cdr ‘((a) b (c d))) = (b (c d))



Car and Cdr

car and *cdr* can deconstruct any list

```
(car (cdr (cdr '((a) b (c d)) ) ) )   =>   (c d)
```

Special abbreviation for sequences of *car*s and *cdr*s:

- keyword: 'c' and 'r' surrounding sequence of 'a's and 'd's for *car*s and *cdr*s, respectively

```
(caddr '((a) b (c d)))   =>  (c d)
```

Using *car* and *cdr*

Most Scheme functions operate over lists recursively using *car* and *cdr*

```
    (define (len   l)
        (if (null? l)
              0
              (+  1   (len  (cdr
        l) ) )
        )
     )
    (len ‘(1 2 3))
value: 3
```

```
    (define (sum   l)
        (if (null? l)
              0
              (+  (car  l)   (sum  (cdr
        l) ) )
        )
     )
    (sum ‘(1 2 3))
value: 6
```

Some useful Scheme functions

- Numeric: +, -, *, /, = (equality!), <, >
- eq?: equality for names

  E.g., (eq? 'a 'a) => #t
- null?: is list empty?

  E.g., (null? '()) => #t

  (null? '(1 2 3)) => #f
- Type-checking:
  - list?: is S-expression a list?
  - number?: is atom a number?
  - symbol?: is atom a name?
  - zero?: is number 0?
- list: make arguments into a list

  E.g., (list 'a 'b 'c) => (a b c)

Note Scheme convention:

Boolean function names end with ?

How Scheme works: The READ-EVAL-PRINT loop

READ-EVAL-PRINT loop:

READ: input from user

- a function application

EVAL: evaluate input

- (f arg_1 arg_2 … arg_n)
  1. evaluate f to obtain a function
  2. evaluate each arg_i to obtain a value
  3. apply function to argument values

may involve repeating this process recursively!

PRINT: print resulting value, either the result of the function application

How Scheme works: The READ-EVAL-PRINT loop

Alternatively,

READ-EVAL-PRINT loop:

- READ: input from user
  - a symbol definition
- EVAL: evaluate input
  - store function definition
- PRINT: print resulting value
  - the symbol defined

Polymorphism

Polymorphic functions can be applied to arguments of different types

- function *length* is polymorphic:
  - (length ‘(1 2 3))

    value: 3
  - (length ‘(a b c))

    value: 3
  - (length ‘((a) b (c d)))

    value: 3
- function *zero?* is not polymorphic (monomorphic):
  - (zero? 10)

    value: #t
  - (zero? ‘a)

    error: object a is not the correct type

Defining global variables

The predefined function *define* merely associates names with values:

- (define moose ‘(a b c))

value: moose

- (define yak ‘(d e f))

value: yak

- (append moose yak)

value: (a b c d e f)

- (cons moose yak)

value: ((a b c) d e f)

- (cons ’moose yak)

value: (moose d e f)

Unnamed functions

Functions are **values**

=> functions can exist without names

Defining function values:

- notation based on the *lambda-calculus*
- lambda-calculus: a formal system for defining recursive functions and their properties

```
(lambda (<param-list>) <body-S-expression>)
```

Using function values

Examples:

- (* 10 10)

value: 100

- (lambda (x) (* x x))

value: compound procedure

- ((lambda (x) (* x x))
  10)

value: 100

```
    (define (square x)  (*  x  x))
    (square 10)
value: 100

    (define sq  (lambda (x)  (*  x  x)) )
    (sq 10)
value: 100
```

alternative form of function definition

Higher-order Functions

Functions can be return values:

- (define (double n)   (* n 2))
- (define (treble n)   (* n 3))
- (define (quadruple n) (* n 4))

Or:

- (define (by_x x) (lambda (n) (*   n x)))
- ((by_x 2) 2)

value: 4

- ((by_x 3) 2)

value: 6

Higher-order Functions

Functions can be used as parameters:

- (define (f g x) (g x))
- (f number? 0)

value: #t

- (f length ‘(1 2 3))

value: 3

- (f (lambda (n) (* 2 n)) 3)

value: 6

Functions as parameters

Consider these functions:

```
; double each list element
(define  (double  l)  (if  (null?  l)  '()
                      (cons (* 2 (car  l))  (double (cdr  l)))  ))

; invert each list element
(define  (invert  l)  (if  (null?  l)  '()
                      (cons (/ 1  (car  l))  (invert  (cdr  l)))  ))

; negate each list element
(define  (negate  l)  (if  (null?  l)  '()
                      (cons (not  (car  l))  (negate  (cdr  l)))     ))
```

Functions as parameters

Where are they different?

```
; double each list element
(define  (double  l)  (if  (null?  l)  '()
                   (cons (* 2 (car  l))   double (cdr  l))) ))

; invert each list element
(define  (invert  l)  (if  (null?  l)  '()
                   (cons (/ 1  (car  l))   invert  (cdr  l))) ))

; negate each list element
(define  (negate  l)  (if  (null?  l)  '()
                   (cons  not  (car  l))   negate  (cdr  l)))    ))
```

Environments

The special forms *let* and *let** are used to define local variables:

(let ((v_1 e_1) (v_2 e_2) … (v_n e_n)) <S-expr>)
(let* ((v_1 e_1) (v_2 e_2) … (v_n e_n)) <S-expr>)

Both establish bindings between variable v_i and expression e_i

- let does bindings in parallel
- let* does bindings in order

End of Lecture