Programming Paradigms CSI2120 – Winter 2018

Jochen Lang

EECS, University of Ottawa

Canada

Université d'Ottawa | University of Ottawa

uOttawa

L'Université canadienne
Canada's university

uOttawa.ca

Functional Programming in Scheme

- Equivalency predicates
- Lists
- List operations
- Tail Recursions

Simple Predicate Functions

- **The predicate symbol ? indicates a boolean function returning #t or #f.**
  - `(symbol? x)` true if x is a symbol
  - `(number? x)` true if x is a number
  - `(eq? x y)` true if x and y have internally the same representation (think of it as same pointer value)
  - `(equal? x y)` true if x and y are identical objects (not necessarily atomic but same structure and content)
  - `(null? x)` true if x is the empty lists ()
  - `(pair? x)` true if x is a list or pair
  - `(procedure? x)` true if x is a procedure
  - `(list? x)` true if x is a list

Equality Test `eq?`

- **eq? compares internal representations**
  - addresses (pointer values)
  - Cannot be used to reliably compare
    - numbers
    - characters

```
(define hello "bonjour")
(eq? hello hello)
=> #t
(eq? "bonjour" "bonjour")
=> #f or #t
```

Equality Test `eqv?`

- **eqv? is similar to eq?**
  - But can be used for characters and numbers
    - Characters and numbers are compared by their ***value***
  - Can not be used to compare lists, strings or functions

```
(eqv? 1 1)
#t
(eqv? 2 (+ 1 1))
#t
(eqv? 1 1.0)
#f
```

Equality Test `equal?`

- **equal? compares the structure and contents**
  - works for lists, strings and functions

```
(equal? `(a 1 2) `(a 1 2))
=> #t
(equal? "bonjour" "bonjour")
=> #t
(equal? (list 1 2) `(1 2))
=> #t
(equal? `a `a)
=> #t
(equal? 2 2)
=> #t
```

Control Structures

- Control structures in Scheme are simple. There are “no” loops. There are only functions, conditional expressions, and the sequence (a concession to programmers used to imperative languages).
- Sequences start with begin

```
(begin (display 'okay) (display '(great)))
=> okay(great)
```

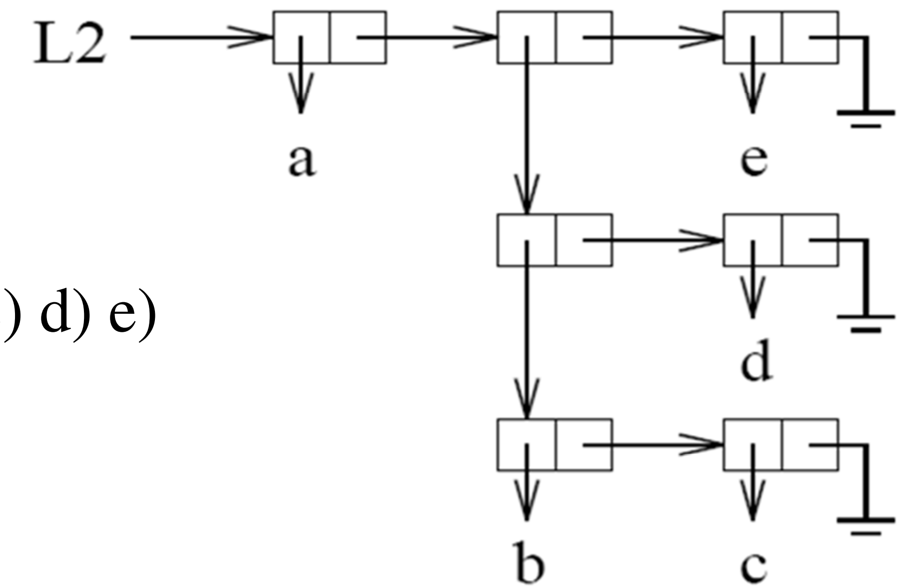
- The value returned by (begin ...) is the value of the last expression.

```
(begin (+ 1 2) (- 1 2))
=> -1
```

List Representation

- **Internally a list consists of two pointers**
  - The first of these pointers gives the address of the atom or the corresponding list.
  - The second pointer gives the address of the next cell.

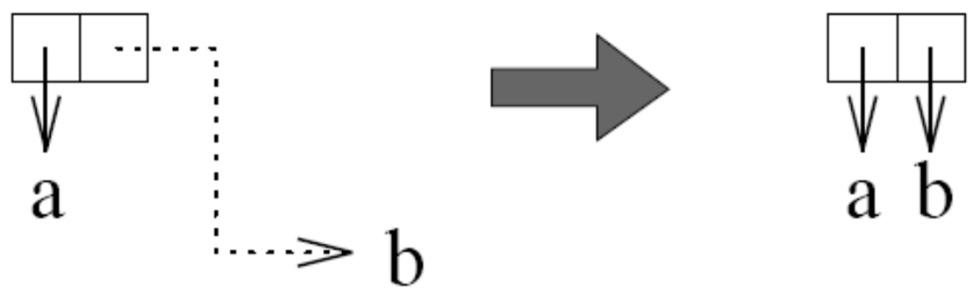
(a ((b c) d) e)



Pairs: `(cons obj1 obj2)`

- **Cons is the pair constructor**
- **The use of pointing in Scheme pairs is not recommended (dotted pairs are not lists!)**

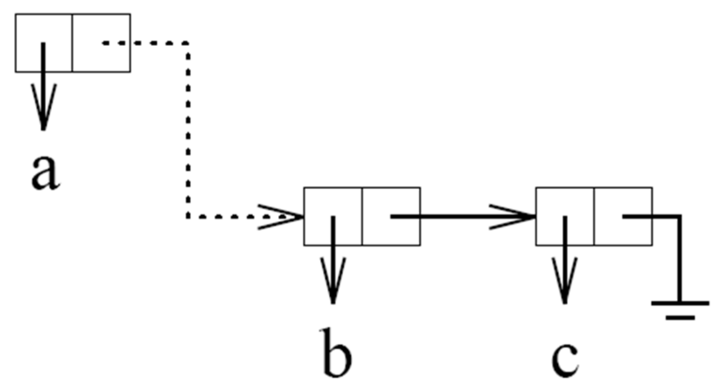
```
(cons `a `b)
=> (a . b)
```



List Construction with `cons`

- **The first parameter of cons is an object which is the beginning of the list. The second parameter is an object which is the tail of the list.**
  - Essentially two pointers in so-called cons cells
- **Internally a new memory cell is created**
  - The first points to the first object passed as parameter
  - The second pointer points to the second object

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



CAR and CDR

- **CAR stand for Content of the Address Register**

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

- **CDR stand for Content of the Decrement Register**

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

Nesting List Expressions

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

can be written as cdr car cdr = cd a dr = cdadr

– *works up to four combinations*

```
(cdadr '(a (b c d) e))
=> (c d)

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

Recursive Concatenation of Two Lists

```
(define (append-list L1 L2)
  (if (null? L1)
      L2
      (cons (car L1) (append-list (cdr L1) L2))))
=> append-list

(append-list '(a b) '(c d))
=> (a b c d)
```

- ***Note: There is a pre-defined function `append`.***

Recursive Inverting of a List

```
(define (invert-list L)
  (if (null? L)
      '()
      (append-list (invert-list (cdr L))
                   (list (car L)))))
=> invert-list

(invert-list '(a b c d))
=> (d c b a)
```

- ***Note: There is a pre-defined function `reverse`.***

Recursive List Membership

- **Find the member in the list and return a list with this member as the `car` of a list.**

```
(define (member-list a L)
  (cond ((null? L) '())
        ((equal? a (car L)) L)
        (#t (member-list a (cdr L)))))

(member-list 'a '(a b c))
=> (a b c)
(member-list 'b '(a b c))
=> (b c)
(member-list 'd '(a b c))
=> ()
```

Note: There is a pre-defined function `member`.

Recursive Size (Length) of a List

```
(define (length-list L)
  (if (null? L)
      0
      (+ 1 (length-list (cdr L)))))
=> length-list

(length-list '(a b c))
=> 3
```

- ***Note: There is a pre-defined function `length`.***

Another Recursive List Example

- **Function that finds if an element in the list has a neighbour which is the same as itself**

```
(define (same-neighbours? L)
  (cond
   ((null? L) #f)
   ((null? (cdr L)) #f)
   ((equal? (car L)(cadr L)) #t)
   (else
    (same-neighbours? (cdr L)))))
=> same-neighbours?
(same-neighbours? '(1 2 3 3 5))
=> #t
```

Predicate Function for Number-only Lists

```
(define (number-list? x )
  (cond
   ((not ( list? x )) #f)
   ((null? x ) #t)
   ((not (number? (car x))) #f)
   (else (number-list? (cdr x )))))
=> number-list
(number-list? '(1 2 3 4))
=> #t
(number-list? '(1 2 3 bad 4))
=> #f
```

Equivalence of Two Lists?

```
(define (eqExpr? x y)
  (cond
   ((symbol? x)  (eq? x y))
   ((number? x)  (eqv? x y))
   ; x is a list:
   ((null? x)  (null? y))
   ; x is a non-empty list
   ((null? y) #f)
   ((eqExpr? (car x) (car y))
    (eqExpr? (cdr x) (cdr y))) ; recurse on car and cdr
   (else  #f)))

(eqExpr? '(1 2 3 4) '(1 2 3 4))
=> #t
(eqExpr? '(1 2 3 4) '(1 2 '(3 4)))
=> #f
```

Removing Duplicates from a List

```
(define (repeated-elements L)
  (if (list? L)
      (do-repeated-elements L)
      'list-error))

(define (do-repeated-elements L)
  (cond
   ((null? L) '())
   ((member (car L) (cdr L))
    (do-repeated-elements (cdr L)))
   (else (cons (car L)
              (do-repeated-elements (cdr L))))
   ))

(repeated-elements '(1 2 3 2 2))
=> (1 3 2)
```

Stack – Basic Definition

```
(define (empty? stack)
   (null? stack))

(define (push e stack)
   (cons e stack))

(define (pop stack)
  (if (empty? stack)
      '()
      (cdr stack)))
```

```
(define (top stack)
  (if (empty? stack)
      ()
      (car stack)))
(empty? '())
=> #t
(push 5 '(2 3 4))
=> (5 2 3 4)
(top '(2 3 4))
=> 2
(pop '(2 3 4))
=> (3 4)
```

Minimal Element in a List

```
(define (min-list x)
  (if (null? x)
      x
      (min-list-aux (car x)(cdr x))))

(define (min-list-aux e l)
  (cond
   ((null? l) e)
   ((> e (car l))
    (min-list-aux (car l)(cdr l)))
   (else (min-list-aux e (cdr l)))))

(min-list '(4 8 9 2 8))
=> 2
```

List Minimum Using Local Variables

```
(define (min-list-aux e l)
  (if (null? l) e
      ; else
      (let ((v1 (car l))
        (v2 (cdr l)))
    (if
     (> e v1)
     (min-list-aux v1 v2)
     (min-list-aux e v2)
     ))
    ))
```

Other Example of Using Local Scope

- **Function quadruple using double with local scope**

```
(define (quadruple x)
  (let ((double (lambda (x) (+ x x))))
    (double (double x))
))

(quadruple 8)
=> 32
(double 8)
=> ;Unbound variable: double
```

Traversal Applying a Function

- **Accept function as an argument**
  - cdr to move to the end of the list
  - cons to add the changed element at the beginning

```
(define (apply-list fct L)
   (if (null? L)
   '()
   (cons (fct (car L))
         (apply-list fct (cdr L)))))

(apply-list (lambda(x) (+ x 4)) '(1 2 3 4))
=> (5 6 7 8)
```

Adding a Prefix to the Elements of a List

- **Turn each element of a list into a pair (using cons) attaching the prefix parameter**

```
(define (prefix-list p L)
    (apply-list
        (lambda(e) (cons p e)) L))

(prefix-list 2 '(1 2 3))
=> ((2 . 1)(2 . 2)(2 . 3))
```

Generating Combinations

- **In combinations order does not matter**
  - Aside: append concatenates the input lists

```
(define (combine dim set)
  (cond
   ((= dim 0) '(()))
   ((null? set) '())
   (else
    (append (prefix-list (car set)
                (combine (- dim 1) (cdr set)))
         (combine dim (cdr set))))))

(combine 2 '(1 2 3))
=> ((1 2) (1 3) (2 3))
```

Reduction of a List to a Value

- **Apply a function to all elements and return the result**
  - `F0` is the value of the reduction for the empty list

```
(define (reduce F F0 L)
  (if (null? L)
      F0
      (F (car L)
         (reduce F F0 (cdr L)))
))

(reduce * 1 '(1 2 3 4))
=> 24
```

Loops as Recursions

- **Looping N times**

```
(define (loop P N)
  (cond ((zero? N) '())
  (#T (display P) (loop P (- N 1)))))
```

- **Loop over range**

```
(define (loop2 P inf sup)
  (cond ((> inf sup) '())
  (#T (display P) (loop2 P (+ inf 1) sup))))
```

  – NOTE: These functions have a tail recursion (tail recursion) which is easier to optimize by a compiler

Traversal using a Tail Recursion

- **Any recursive function can be in the form of tail recursion using an accumulator (variables) for intermediate results**

```
(define (apply-list2 fct L Lacc)
  (if (null? L)
      Lacc
      (apply-list2 fct (cdr L)
               (append Lacc (list (fct (car L))))
)))
(define (apply-list fct L)
  (apply-list2 fct L '()))

(apply-list abs '(-3 -2.1 2 3.4))
=> (3 2.1 2 3.4)
```

Factorial Example

```
(define (factorial n)
     (if (<= n 0)
         1
         (* n (factorial (- n 1))) ) )
```

- **To turn this into a tail recursion, the function needs to return the result of the recursive call without changes**

```
(define (factorial n) (factorialb n 1))
(define (factorialb n answer)
  (if (<= n 0)
      answer
      (factorialb (- n 1) (* n answer))))
```

Map Procedure

- **Map applies a function to every element of a list. It can be more convenient than an explicit loop**

```
(map abs `(1 -2 3 -4 5 -6))
(1 2 3 4 5 6 )
```

- **Define a lambda in the same line**
  - function taking two arguments
  - supply two lists

```
(map (lambda (x y) (* x y))
   `(1 2 3 4) `(8 7 6 5))
(8 14 18 20)
```

Summary

- **Equivalency predicates**
- **Lists**
- **List operations**
  - concatenate, inverse, membership, length, list neighbours, number-only predicate, list equivalence, duplicate removal, list as a stack, minimum, functions using local scope, applying a function to list elements, adding a prefix, combination, reduction of a list
- **Tail Recursions**
  - Loops
  - Factorials
  - Map Procedure