# CS 326
# Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Scheme

# The Scheme Programming Language

- 1950s-1960s: Lisp developed at MIT (John McCarthy), based on lambda calculus (Alonzo Church)
- Lisp - the first functional programming language
- No single standard evolved for Lisp
- Two major Lisp dialects - Common Lisp and Scheme

- Scheme - developed at MIT in 1970s
- Member of functional class of programming languages
- Actually - has a functional core plus some imperative features
- Main application - symbolic computing, artificial intelligence

# The Structure of a Scheme Program

- All programs and data are expressions
- Expressions can be atoms or lists
- Atom: number, string, identifier, character, boolean
- List: sequence of expressions separated by spaces, between parentheses

- Syntax:

expression $\rightarrow$ atom | list
atom $\rightarrow$ number | string | identifier | character | boolean
list $\rightarrow$ ( expr_seq )
expr_seq $\rightarrow$ expression expr_seq | expression

# Interacting with Scheme

- Interpretor: "read-eval-print" loop

```
> 1
1
```

Reads 1, evaluates it (1 evaluates to itself), then prints its value

```
> (+ 2 3)
5
```

+ => function +

2 => 2

3 => 3

Applies function + on operands 2 and 3 => 5

# Evaluation

- Constant atoms - evaluate to themselves

| | |
|---|---|
| 42 | - a number |
| 3.14 | - another number |
| "hello" | - a string |
| #/a | - character 'a' |
| #t | - boolean value "true" |

```
> "hello world"
"hello world"
```

# Evaluation

- Identifiers (symbols) - evaluate to the value bound to them

```
(define a 7)

> a
7

> +
#<procedure +>
```

# Evaluation

- Lists - evaluate as "function calls":

  (function arg1 arg2 arg3 ...)

- First element must evaluate to a function
- Recursively evaluate each argument
- Apply the function on the evaluated arguments

```
> (- 7 1)
6

> (* (+ 2 3) (/ 6 2))
15
```

# Operators - More Examples

- Prefix notation
- Any number of arguments

```
> (+)
0

> (+ 2)
2

> (+ 2 3)
5
```

```
> (+ 2 3 4)
9

> (- 10 7 2)
1

> (/ 20 5 2)
2
```

# Preventing Evaluation (quote)

- Evaluate the following:

```
> (1 2 3)
Error: attempt to apply non-procedure 1.
```

- Use the quote to prevent evaluation:

```
> (quote (1 2 3))
(1 2 3)
```

- Short-hand notation for quote:

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

# More Examples

```
(define a 7)

a                    => 7
'a                   => a

 (+ 2 3)    => 5
'(+ 2 3)    =>  (+ 2 3)
((+ 2 3))   =>  Error: attempt to apply non-procedure 5.

'(her 3 "sons")      => (her 3 "sons")
```

- Make a list:

```
(list 'her (+ 2 1) "sons")     => (her 3 "sons")
(list '(+ 2 1) (+ 2 1))        =>     ((+ 2 1) 3)
```

# Forcing Evaluation (eval)

```
 (+ 1 2 3)           => 6
'(+ 1 2 3)           => (+ 1 2 3)
(eval '(+ 1 2 3))    => 6

(list + 1 2) => (+ 1 2)
(eval (list + 1 2))    =>    3
```

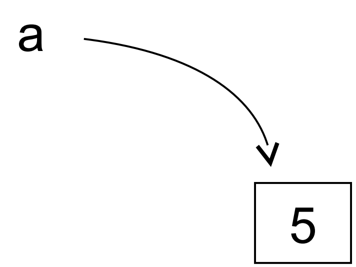
- Eval evaluates its single argument
- Eval is implicitly called by the interpretor to evaluate each expression entered:

"read-eval-print" loop

# Special Forms

- Have different rules regarding whether/how arguments are evaluated

(define a 5) ; binds a to 5, does not evaluate a




(quote (1 2 3)) ; does not evaluate (1 2 3)

- There are a few other special forms – discussed later

# Using Scheme

- Petite Chez Scheme
  - free downloads for Windows, Linux, Unix

- The essentials:

```
scheme    ; to start Scheme from the Unix prompt
^C        ; to interrupt a running Scheme program
(exit)    ; to exit Scheme
```

- Documentation available on Chez Scheme webpage

# Announcements

- Readings
  - May look at Scheme books or on-line references

# List Operations

- cons – returns a list built from head and tail

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

(cons 'a 'b)                => (a . b)        ; improper list
```

# List Operations

- car – returns first member of a list (head)

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

  (car '(a)) => a

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

  (car '(this (is no) more difficult)) => this

- cdr – returns the list without its first member (tail)

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

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

  (cdr '(a)) => ()

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

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

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

# List Operations

- null? – returns #t if the list is null ()

  #f otherwise

- list – returns a list built from its arguments

| | |
|---|---|
| (list 'a 'b 'c) | => (a b c) |
| (list 'a) | => (a) |
| (list '(a b c)) | => ((a b c)) |
| (list '(a b) 'c) | => ((a b) c) |
| (list '(a b) '(c d)) | => ((a b) (c d)) |

# List Operations

- length – returns the length of a list

  (length '(1 3 5 7)) => 4

  (length '((a b) c)) => 2

- reverse – returns the list reversed

  (reverse '(1 3 5 7)) => (7 5 3 1)

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

- append – returns the concatenation of the lists received as arguments

  (append '(1 3 5) '(7 9)) => (1 3 5 7 9)

  (append '(a) '()) => (a)

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

# Type Predicates

- Check the type of the argument and return #t or #f

```
(boolean? x)        ; is x a boolean?
(char? x)           ; is x a char?
(string? x)         ; is x a string?
(symbol? x)         ; is x a symbol?
(number? x)         ; is x a number?
(list? x)           ; is x a list?
(procedure? x)      ; is x a procedure (function)?
```

# Boolean Expressions

```
(< 1 2)                         => #t
(>= 3 4)                        => #f
(= 4 4)                         => #t
(eq? '(a b) '(a b))             => #f    ; same object?
(equal? '(a b) '(a b))                  => #t    ; recursively equivalent
   structure?

(not (> 5 6))                           => #t
(and (< 3 4) (= 2 3))                   => #f
(or  (< 3 4) (= 2 3))           => #t
```

- and, or are special forms - evaluate arguments only while needed

# Conditional Expressions

- if – has the form:

  (if <test_exp> <then_exp> <else_exp>)

  (if (< 5 6) 1 2) => 1

  (if (< 4 3) 1 2) => 2

- Anything other than #f is treated as true:

  (if 3 4 5) => 4

  (if '() 4 5) => 4 ; as opposed to Lisp!!

- if is a special form - evaluates its arguments only when needed:

  (if (= 3 4) 1 (2)) => Error: attempt to apply non-procedure 2.

  (if (= 3 3) 1 (2)) => 1

# Conditional Expressions

- cond – has the form:

```
(cond
  (<test_exp1> <exp1> ...)
  (<test_exp2> <exp2> ...)
  ...
  (else <exp> ...))

(define n -5)
(cond ((< n 0) "negative")
      ((> n 0) "positive")
      (else "zero"))          => "negative"
```

- cond is a special form - evaluates its arguments only when needed

# Syntax (C vs. Scheme)

| C | Scheme |
|---|---|
| 1 + 2 + 3 | (+ 1 2 3) |
| 3 + 4 * 5 | (+ 3 (* 4 5)) |
| factorial (9) | (factorial 9) |
| (a == b) && (c != 0) | (and (= a b) (not (= c 0))) |
| (low < x) && (x < high) | (< low x high) |
| f (g(2,-1), 7) | (f (g 2 -1) 7) |

# Functions

- Create a function by evaluating a lambda expression:

  (lambda (id1 id2 ...) exp1 exp2 ...)

  - id1 id2 ... - formal parameters
  - exp1 exp1 ... - body of the function
  - return value of function - last expression in body
  - return value of lambda expression - the (un-named) function

  (lambda (x) (* x x)) => #<procedure>

  - Returns an un-named function that takes a parameter and returns its square

# Functions

- Call a function by applying the evaluated lambda expression on its actual parameters:

```
((lambda (x) (* x x)) 3)    => 9
```

the function    the actual parameter

- How can you reuse the function?
  - You can't!
- Why is it then useful?
  - Return a function from another function
- What if you REALLY want to reuse it?

# Functions

- Bind a name to a function:

```
(define square (lambda (x) (* x x)))
```

```
square ──> #<procedure>
```

- Equivalent short-hand notation (typical way to use it):

```
(define (square x) (* x x))
```

- Now call the function:

```
(square 3)        => 9
```

# Functions

- Functions vs. variables:

```
(define f 3)
f                        => 3
(f)                      => Error: attempt to apply non-procedure 3.

(define (f) 3)
f                        => #<procedure>
(f)                      => 3
```

- Last definition is equivalent to:

```
(define f (lambda () 3))      ; a function that takes no parameters and
                              ; returns 3
```

# Functions

**C**

```
if (a == 0)
    return f(x,y) ;
else
    return g(x,y) ;
```

**Scheme**

```
(if (= a 0)
    (f x y)
    (g x y))
```

# Functions

C

```
if (a == 0)
    return f(x,y) ;
else
    return g(x,y) ;
```

repeated arguments (x,y)

- Can we write it better in Scheme?

Scheme

```
((if (= a 0)
    f
    g)
x y)
```

evaluates to either #<procedure f> or #<procedure g>

it is then applied on arguments x and y

# Interacting with Scheme

- Instead of writing everything at the Scheme prompt, you can:
  - write your function definitions and your global variable definitions (define...) in a file ("file_name")
  - at the Scheme prompt, load the file with: (load "file_name")
  - at the Scheme prompt call the desired functions
  - there is no "formal" main function

# Announcements

- Homework
  - HW 2 out – due February 20
  - Submission
    - Submit your code in Canvas as one “hw2.txt” file containing all your functions.
    - The file must be able to load and be tested in the interpreter.
      - Consequently, the file must be in a plain text format; do <u>not</u> submit Word, PDF, RTF, JPG or any such types of files.
      - Also make sure that any auxiliary information (such as your name or question numbers) is commented out.

# Recursion

- Recursion plays a greater role in Scheme than in other languages
- Why?

- Functional programming – avoid side effects (assignments) and iterations

- Recursive data structures – a list is either empty, or has a car and a cdr; the cdr is (again) a list

- Elegance – recursive algorithms are considered more elegant than iterative ones (just ask a Scheme or Lisp programmer!)

# Recursion

- How do you solve a problem recursively?

- Do not rush to implement it
- Think of a recursive way to describe the problem:
  - Show how to solve the problem in the general case, by decomposing it into similar, but smaller problems
  - Show how to solve the smallest version of the problem (the base case)
- Now the implementation should be straightforward (in ANY language)
  - But don't forget to handle base case first when implementing

# Recursion

- How do you THINK recursively?

- Example: define factorial

$$\text{factorial}(n) = \underbrace{1 * 2 * 3 * \ldots(n-1)}_{\text{factorial }(n-1)} * n$$

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=1 \quad \text{(the base case)} \\ n * \text{factorial}(n-1) & \text{otherwise} \quad \text{(inductive step)} \end{cases}$$

# Recursion

- Implement factorial in Scheme:

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

(factorial 4)       => 24
```

# Recursion

- Fibonacci:

$$
\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}
$$

- Implement in Scheme:

```
(define (fib n)
    (cond
      ((= n 0) 0)
      ((= n 1) 1)
      (else   (+ (fib (- n 1)) (fib (- n 2))))))
```

# Recursion

- Length of a list:

$$\text{len(lst)} = \begin{cases} 0 & \text{if list is empty} \\ 1 + \text{len ( lst-without-first-element )} & \text{otherwise} \end{cases}$$

- Implement in Scheme:

```
(define (len lst)
   (if (null? lst)
      0
      (+ 1 (len (cdr lst)))))
```

# Recursion

- Sum of elements in a list of numbers:

$$\text{sum(lst)} = \begin{cases} 0 & \text{if list is empty} \\ \text{first-element + sum (lst-without-first-element)} & \text{otherwise} \end{cases}$$

- Implement in Scheme:

```
(define (sum lst)
  (if (null? lst)
    0
    (+ (car lst) (sum (cdr lst)))))
```

# Recursion

- Check membership in a list:

```
(define (member? x lst)
   (cond
        ((null? lst) #f)
        ((equal? x (car lst)) #t)
        (else (member? x (cdr lst)))))
```

# Recursion

- Return the nth element in a list:

```
(cadddddddd...dddddddr lst)          ; don't try this at home!

(define (nth lst n)  ; assumes lst is non-empty and n >= 0
   (if (= n 0)
       (car lst)
       (nth (cdr lst) (- n 1))))
```

- Return a reversed list:

```
(define (reverse lst)
   (if (null? lst)
       lst
       (append (reverse (cdr lst)) (list (car lst)))))
```

# Recursion

- In general:
  - When recurring on a list lst, ask two questions about it: (null? lst) and else
  - When recurring on a number n, ask two questions about it: (= n 0) and else

  - When recurring on a list lst, make your recursive call on (cdr lst)
  - When recurring on a number n, make your recursive call on (- n 1)

# Local Definitions

- let - has a list of bindings and a body
  - each binding associates a name to a value
  - bindings are local (visible only in the body of let)
  - let returns the last expression in the body

```
> (let ((a 2) (b 3))  ; list of bindings
   (+ a b))                    ; body - expression  to evaluate and
  return
5

a                              => Error: variable a is not bound.
b                              => Error: variable b is not bound.
```

# Local Definitions

- Factor out common sub-expressions:

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

$$a = 1+xy$$
$$b = 1-y$$

$$f(x,y) = xa^2 + yb + ab$$

# Local Definitions

$a = 1+xy$

$b = 1-y$

$f(x,y) = xa^2 + yb + ab$

- Locally define the common sub-expressions:

```
(define (f x y)
   (let ((a (+ 1 (* x y)))
            (b (- 1 y)))
         (+ (* x a a) (* y b) (* a b))))

(f 1 2)                         => 4
```

# Local Definitions

- Functions can also be declared locally:

```
(let ((a 3)
      (b 4)
      (square (lambda (x) (* x x)))
      (plus +))
  (sqrt (plus (square a) (square b))))                    => 5
```

# Local Definitions

- let makes its bindings in parallel:

```
(define x 'a)
(define y 'b)
(list x y)                    => (a b)

(let ((x y) (y x)) (list x y))       => (b a)
        b     a
```

1. First evaluate all expressions in the binding list
2. Then bind all names in binding list to those values

# Local Definitions

- Bad let bindings:

```
(let ((x 1)
      (y (+ x 1)))
  (list x y))
```

=> Error: variable x is not bound.

- What if we want y to be computed in terms of x?
  - Need to use let*

# Local Definitions

- let* - similar to let, except that bindings are made sequentially

```
(let* ((x 1) (y (+ x 1)))
   (list x y))                              => (1 2)
```

- How can you do this using let?

- let* is equivalent to ("syntactic sugar" for):

```
(let ((x 1))
   (let ((y (+ x 1)))
      (list x y)))
```

# Local Definitions

- What if we want to locally define a recursive function?
- letrec - similar to let*, but allows recursive definitions
- Define factorial locally:

```
(letrec ((factorial (lambda (n)
                        (if (= n 1)
                            1
                            (* n (factorial (- n 1)))))))
        (factorial 5))

=> 120
```

# Indentation

- Indentation is essential in Scheme. With all these parentheses, we need something to depend on
- There are two main schools of thought on this:

```
(define (f x)
        (let ((a 1))
                  (if (<
  x 5)
                    (+
  a x)
                    (* a
  x))))
```

OR

```
(define (f x)
     (let ((a 1))
        (if (< x 5)
     (+ a x)
     (* a x)
              )
     )
)
```

- You may choose the style you like most, but be consistent and ALWAYS indent. Otherwise you will not be able to read or debug your own program

# Input and Output

- read - returns the input from the keyboard

```
> (read)
234                ; user types this
234                ; the read function returns this
> (read)
"hello world"
"hello world"
```

- display - prints its single parameter to the screen

```
> (display "hello world")
hello world
> (display (+ 2 3))
5
```

# Input and Output

- newline - displays a new line

- Define a function that asks for input:

```
(define (ask-them str)
   (display str)
   (read))

> (ask-them "How old are you? ")
How old are you? 32
32
```

# Input and Output

- Define a function that asks for a number (if it's not a number it keeps asking):

```
(define (ask-number)
  (display "Enter a number: ")
  (let ((n (read)))
  (if (number? n)
                n
  (ask-number))))

> (ask-number)
Enter a number: a
Enter a number: (5 6)
Enter a number: "Why don't you like these?"
Enter a number: 7
7
```

# Input and Output

- An outer-level function to go with factorial, that reads the input, computes the factorial and displays the result:

```
(define (factorial-interactive)
  (display "Enter an integer: ")
  (let ((n (read)))
    (display "The factorial of ")
    (display n)
    (display " is ")
    (display (factorial n))
    (newline)))

> (factorial-interactive)
Enter an integer: 4
The factorial of 4 is 24
```

# Announcements

- Readings
  - May look at Scheme books or on-line references

# Higher-Order Functions

- In Scheme, a function is a first-class object – it can be passed as an argument to another function, it can be returned as a result from another function, and it can be created dynamically

- A function is called a higher-order function if it takes a function as a parameter, or returns a function as a result

# Higher-Order Functions

- map
  - takes as arguments a function and a sequence of lists
  - there must be as many lists as arguments of the function, and lists must have same length
  - applies the function on corresponding sets of elements from the lists
  - returns all the results in a list

```
(define (square x) (* x x))
(map square '(1 2 3 4 5))  => (1 4 9 16 25)
(map + '(1 2 3) '(4 5 6))    =>                  (5 7 9)
```

- You can also define the function in-place:

```
(map (lambda (x) (* 2 x)) '(1 2 3))  =>          (2 4 6)
```

# Higher-Order Functions

- apply
  - takes a function and a list
  - there must be as many elements in the list as arguments of the function
  - applies the function with the elements in the list as arguments

(apply * '(5 6)) => 30

(apply append '((a b) (c d))) => (a b c d)

- Comparison to eval:

(eval <expression>) or (eval (<func> <arg1> <arg2>...))

(apply <func> (<arg1> <arg2>...))

# Higher-Order Functions

- Define a function compose, that composes two functions given as parameters:

```
(define (compose f g)
   (lambda (x)
                           (f (g x))))

((compose car cdr) '(1 2 3))          => 2
```

- compose not only takes functions as arguments, but also returns a function

# Higher-Order Functions

- Define a function filter, that takes a predicate function (one that returns #t or #f) and a list, and returns a list containing only elements that satisfy the predicate:

```
(define (filter f lst)
  (cond
        ((null? lst)        '())
        ((f (car lst))      (cons (car lst) (filter f (cdr lst))))
        (else               (filter f (cdr lst)))))
```

```
(filter number? '(a 4 (1 2) "testing" +))          => (4)
(filter (lambda (x) (>= x 0)) '(-2 3 -1 1 4))       => (3 1 4)
```

# Higher-Order Functions

- Building procedures:

```
(define (greater-than-n? n)
   (lambda (x) (>= x n)))

greater-than-n?                    => #<procedure>
(greater-than-n? 2)                => #<procedure>
   ; this is a predicate with one parameter x, that checks if x >= 2

((greater-than-n? 2) 3)   =>       #t

(define greater-than-2? (greater-than-n? 2))
(greater-than-2? 3)        => #t
```

# Sequencing

- begin
  - defines a block of expressions that are sequentially evaluated
  - returns the last expression in the block

    (begin <exp1> <exp2> ... <expn>)
- Typical use - in if expressions, where normally only one expression is allowed for each branch:

```
(if (< x 0)
  (begin
              (display "negative")
              (f x))
  (begin
              (display "positive")
              (g x)))
```

# Programming without Side-Effects

```
(define x 1)
(+ x 1)                    => 2
x                          => 1

(define y '(1 2 3))
(reverse y)                => (3 2 1)
y                          => (1 2 3)
```

- Functions compute a result based on the input, and return it
- They do not alter the input

# Coming Next

- What is missing so far?

- assignment operations
- iterations ("for", "while"...)
- pointers
- allocation

# Imperative Features

- So far, almost everything we discussed has conformed to a "pure" functional programming style (without side-effects)
- What has not?
  - define
  - display
- Significant departure from functional programming - discuss a number of imperative extensions to the Scheme language – assignment, iteration
- In general, functional programming is the preferred style (also easier to use) in Scheme

# Assignment

- set!
  - assigns a value to a variable
  - return value is unspecified

```
(set! x 1)
(set! y "apple")
```

- Difference between set! and define:
  - define introduces a new symbol in the current scope, and binds it to a value
  - set! modifies an existing symbol

# Assignment

- Difference between set! and define:

```
(begin
    (define x 3)
    (display x)
    (let ((y 4))
        (define x 5)
        (display x))
    (display x))
```

What is displayed? 353

```
(begin
    (define x 3)
    (display x)
    (let ((y 4))
        (set! x 5)
        (display x))
    (display x))
```

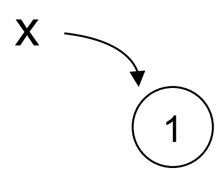
What is displayed? 355

- define introduced a new variable *x*, locally within *let*
- set! just changed the global variable *x*

# Internal Structure of Expressions

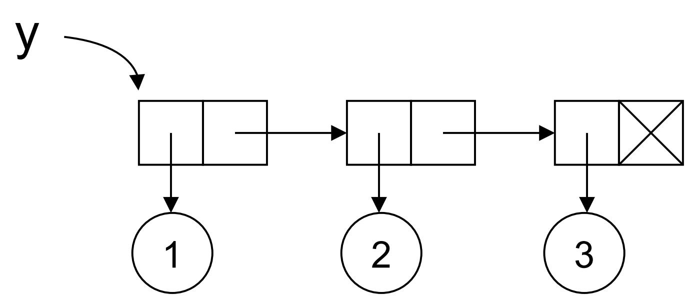
- Implicitly, all variables are pointers that are bound to values
  - Atom values:

    (define x 1)




  - List values:

    (define y '(1 2 3))



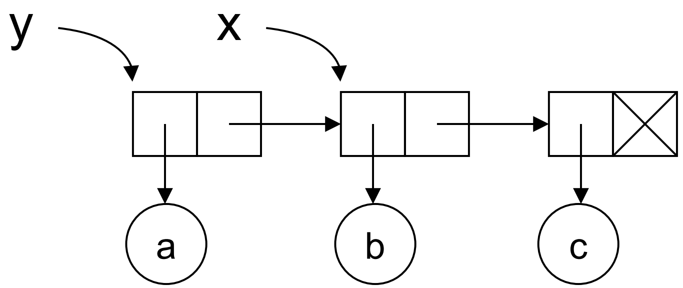

- Each element in the list is a cons cell, which contains:
  - a pointer to a value
  - a pointer to the next cons cell

# Internal Structure of Expressions

```
(define x '(b c))
(define y (cons 'a x))
```

x => (b c)

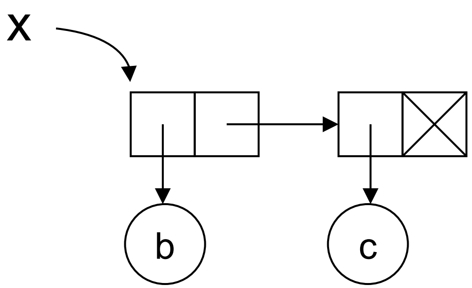
y => (a b c)



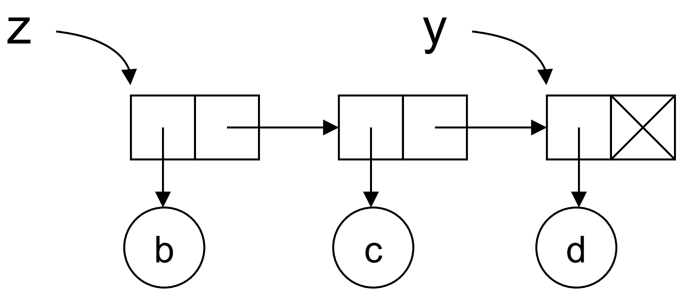

- cons did not alter list *x*
- the list elements *b* and *c* are shared
- no side-effects

# Internal Structure of Expressions

```
(define x '(b c))
(define y '(d))
(define z (append x y))
```



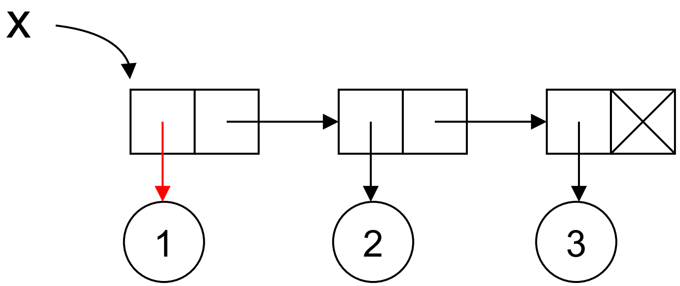

| | |
|---|---|
| x | => (b c) |
| y | => (d) |
| z | => (b c d) |



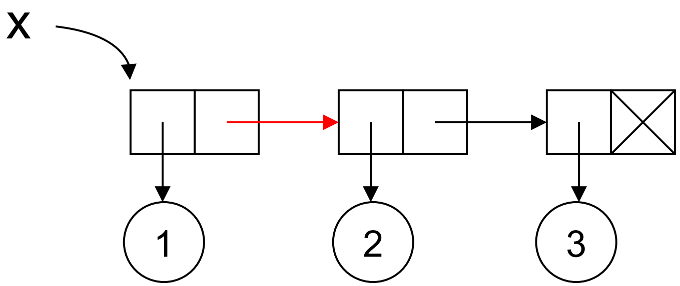

- in order not to alter list *x*, append created new instances of *b*, *c*
- no side-effects

# List Surgery

- To alter the structure of a list:
- set-car! - changes the *pointer-to-value* in the first cons cell




- set-cdr! - changes the *pointer-to-next* in the first cons cell



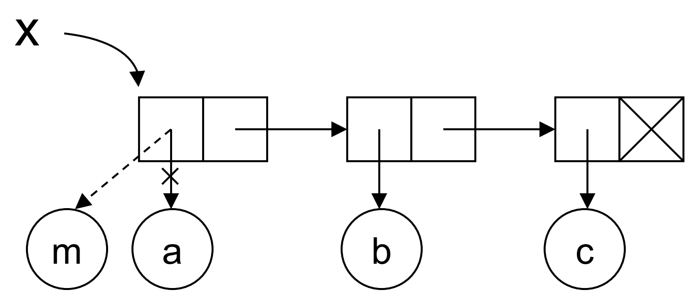

- They alter the arguments (lists) passed to them
- Exclamation mark (!) traditionally used in names of functions that alter their arguments
- Exhibit side-effects, not a functional feature

# List Surgery

- Change the first element in a list:

```
(define x '(a b c))
x                          => (a b c)

(set-car! x 'm)
x                          => (m b c)
```

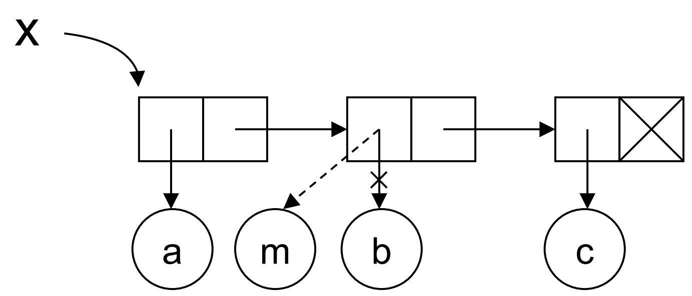


# List Surgery

- Change the second element in a list:

(define x '(a b c))

x => (a b c)




(set-car! (cdr x) 'm)

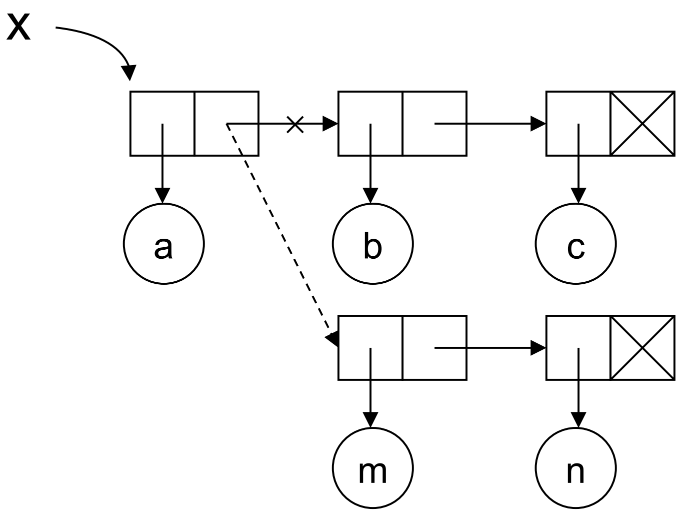
x => (a m c)

# List Surgery

- Change the list tail:

(define x '(a b c))

x => (a b c)



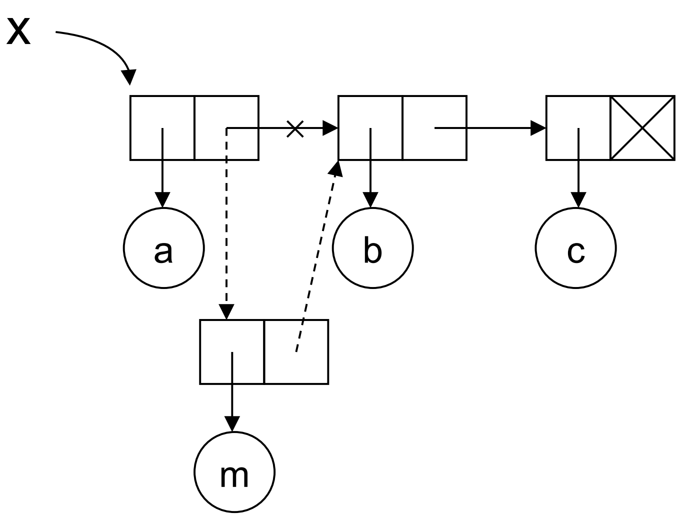

(set-cdr! x '(m n))

x => (a m n)

# List Surgery

- Insert an element in the second position:

```
(define x '(a b c))
x                          => (a b c)
```



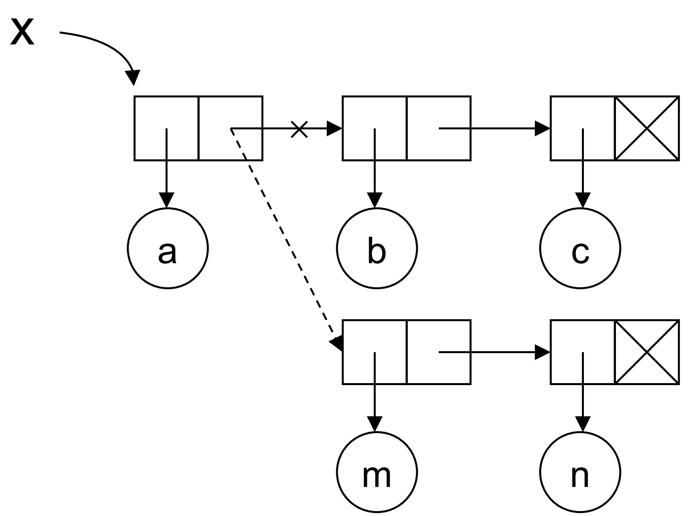

```
(set-cdr! x (cons 'm (cdr x)))
x                          => (a m b c)
```

# Memory Management

- The programmer never needs to explicitly allocate memory
- Memory is allocated as needed by the language implementation (interpretor)

(define x '(a b c))

(set-cdr! x '(m n))




- How can the memory be deallocated? What happens to the list (b c)?
  - Garbage collection (periodically checks and deallocates any memory cells that are not referenced by anyone)

# Iteration

- do
  - very general iteration mechanism
  - has three parts
    - a list of triples (<variable> <init-value> <update-value>)
    - a pair (<termination-test> <return-value>)
    - body (sequence of expressions to be evaluated in each iteration)

# Iteration

- Display all integer numbers from 0 to n:

```
(define (display-num n)
  (do ( (i 0 (+ i 1)) )          ; initially 0, incremented each iteration
      ( (> i n) "Done!" )        ; termination test and return value
      (display i)                ; body
      (newline)))                ; body

> (display-num 3)
0
1
2
3
"Done!"
```

# Iteration

- Iterative version of factorial:

```
(define (factorial-iter n)
   (do ( (i 1 (+ i 1)) ; initially 1, incremented each iteration
         (a 1 (* a i)) ) ; initially 1, set to a*i each iteration
       ( (> i n) a )))   ; termination test and return value
                                        ; empty body

> (factorial-iter 5)
120
```

# Iteration

- Display the first n Fibonacci numbers:

```
(define (display-fib n)
   (do ( (i 0 (+ i 1)) ; initially 0, incremented each iteration
        (a1 0 a2)      ; initially 0, set to a2 each iteration
        (a2 1 (+ a1 a2)) )      ; initially 1, set to a1+a2 each iteration
      ( (= i n) "Done!" )       ; termination test and return value
      (display a1)    ; body
      (display " "))) ; body

> (display-fib 7)
0 1 1 2 3 5 8 "Done!"
```

# A Checkbook Program

```
(define (checkbook)
    (letrec ((IB "Enter initial balance: ")
             (AT "Enter transaction (- for withdrawal): ")
             (FB "Your final balance is: ")

             (prompt-read (lambda (Prompt)
                     (display Prompt)
            (read)))

   ; Compute the new balance given an initial balance init and a new transaction t
   ; Termination occurs when t is 0
              (newbal (lambda (Init t)
            (if (= t 0)
               (list FB Init)
               (transaction (+ Init t)))))

   ; Read the next transaction and pass the information to newbal
               (transaction (lambda (Init)
            (newbal Init (prompt-read AT)))))

   ; Body of checkbook - prompts for the starting balance
      (transaction (prompt-read IB))))
```

# A Checkbook Program

- Here is a sample run:

```
> (checkbook)
Enter initial balance: 100
Enter transaction (- for withdrawal): -20
Enter transaction (- for withdrawal): -40
Enter transaction (- for withdrawal): 150
Enter transaction (- for withdrawal): 0
("Your final balance is: " 190)
```

- Note the use of recursion instead of iteration

# Announcements

- Readings
  - Chapter 3, up to 3.2.3