Lecture 4

Review **let**

- (**let** ((*name1 expr1) (name2 expr2))*
  expr)
- This *binds* the value of *expr1* to *name1* when evaluating *expr*
  - But nowhere else!

let practice

```
1. (define (foo x)
              (+ x 3))
3. foo
4. (foo 5)
5. (define bar 5)
6. (define (baz) 5)
7. bar
8. baz
```

let practice

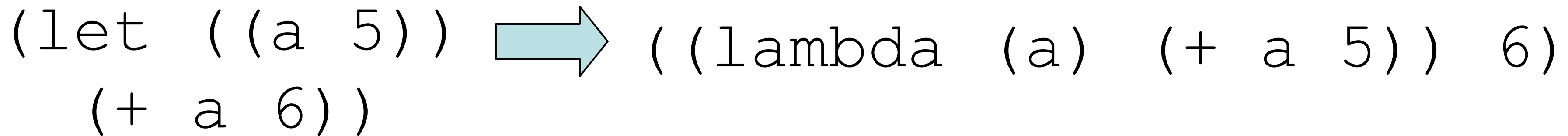
1. `(bar)`
2. `(baz)`
3.
```
(let ((a 3)
      (b 5))
  (+ a b))
```

Syntactic Sugar

- Syntax – Describes how to correctly arrange the language to describe computation
- Syntactic Sugar “sweetens” the syntax by making it more convenient
  - Doesn’t add any new capabilities that you couldn’t already do.

let is syntactic sugar

- How can we express **let** using things that we’ve already learned?
- Use **lambda**



let practice

```
2. (let ((+ *)
         (* +))
       (+ 3 (* 4 5)))
```

let practice

- ```
  (define m 3)
  ```
- ```
  (let ((m (+ m 1)))
        (+ m 1))
  ```

5. ```
   (define n 4)
   ```
6. ```
   (let ((n 12)
             (o (+ n 2)))
      (* n o))
   ```

More Syntactic Sugar

- Are you tired of typing (define my-func (lambda (….. Yet?
- Scheme provides a shortcut that’s more convenient
- (define (my-fun x y z) *exprs*) is the same as (define my-fun (lambda (x y z) *exprs*)

Danger in Recursion

- Load your version of sumton
- Evaluate sumton for n=100
- Grow increase n by a factor of 10 a few times.
- It should get slow fast

Why?

- Let’s write how sumton gets an answer
- (sumton 5) =

  (+ 5 (sumton 4) =

    (+ 5 (+ 4 (+ sumton 3))))

  …
- Each addition is a pending operation
- We have to wait to evaluate it
- The interpreter has to store it

More efficient solution

- Let’s get rid of the pending operation
- Introduce a new variable

```
(define (sumton n)
  (if (= n 1)
      1
      (+ n (sumton (- n 1)))))
```

OLD

```
(define (sumton2 n ans)
  (if (= n 1)
      (+ ans 1)
      (sumton2 (- n 1) (+ ans n))))
```

NEW

Are there pending operations now?

```
(define (sumton2 n ans)
  (if (= n 1)
      (+ ans 1)
      (sumton2 (- n 1) (+ ans n))))
```

A computational process that involves pending operations is called a *recursive* process

A computational process without pending operations is an *iterative* process

This boils down to two different ways of keeping track of the *state* of the process

```
(define (sumton n)
  (if (= n 1)
      1
      (+ n (sumton (- n 1)))))
```

OLD

```
(define (sumton2 n ans)
  (if (= n 1)
      (+ ans 1)
      (sumton2 (- n 1) (+ ans n))))
```

NEW

Making recursive procedures friendlier

- Having to remember to add the 0 to (sumton2) is annoying
- We'll use something called a helper function

```
(define (sumton3-helper n ans)
  (if (= n 1)
      (+ ans 1)
      (sumton2 (- n 1) (+ ans n))))

(define (sumton3 n)
  (sumton3-helper n 0))
```

Don’t always need a helper function

- Remember `pad`
- Implement it with and without a helper function

Practice Problem

- Using string-append, write a procedure pad, which takes a string and a number, that returns the string with that number of spaces added to the end.

```
(pad "yay" 0)
;Value: "yay"
(pad "yay" 1)
;Value: "yay "
(pad "yay" 3)
;Value: "yay   "
(define pad
```

Which version is recursive?

```
(define pad
  (lambda (s n)
    (if (= n 0)
    s
    (string-append (pad s (- n 1)) " "))))
```

```
(define pad
  (lambda (s n)
    (if (= n 0)
    s
    (pad (string-append s " ") (- n 1)))))
```

Is this iterative or recursive?

```
(define (remainder x y)
  (if (< x y)
      x
      (remainder (- x y) y)))
```

Exponentiation

- Calculate x^n where $n>=0$ and n is an integer
- Write iterative version

Factorial

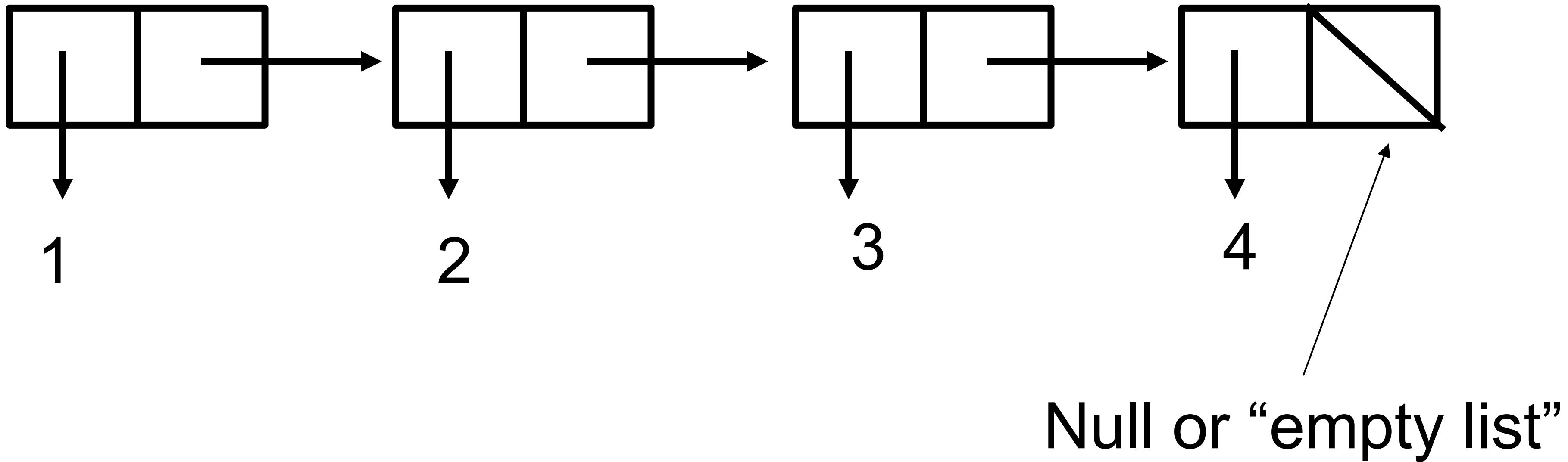
- Write fact as an iterative process

Lists

- Basic data structure in Scheme
- Very useful!
- You can make a list using **list**
  - (**list** 1 2 3 4 5 6)
- Type it in
- (define lst (list 1 2 3 4 5))

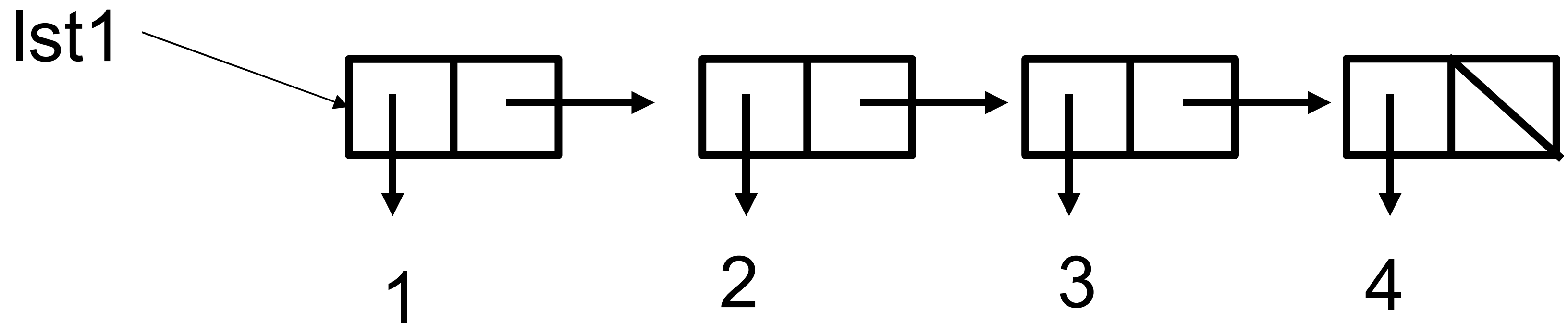
How are lists represented?

(**list** 1 2 3 4**)**



How are lists represented?

(define lst1 **(list** 1 2 3 4**))**

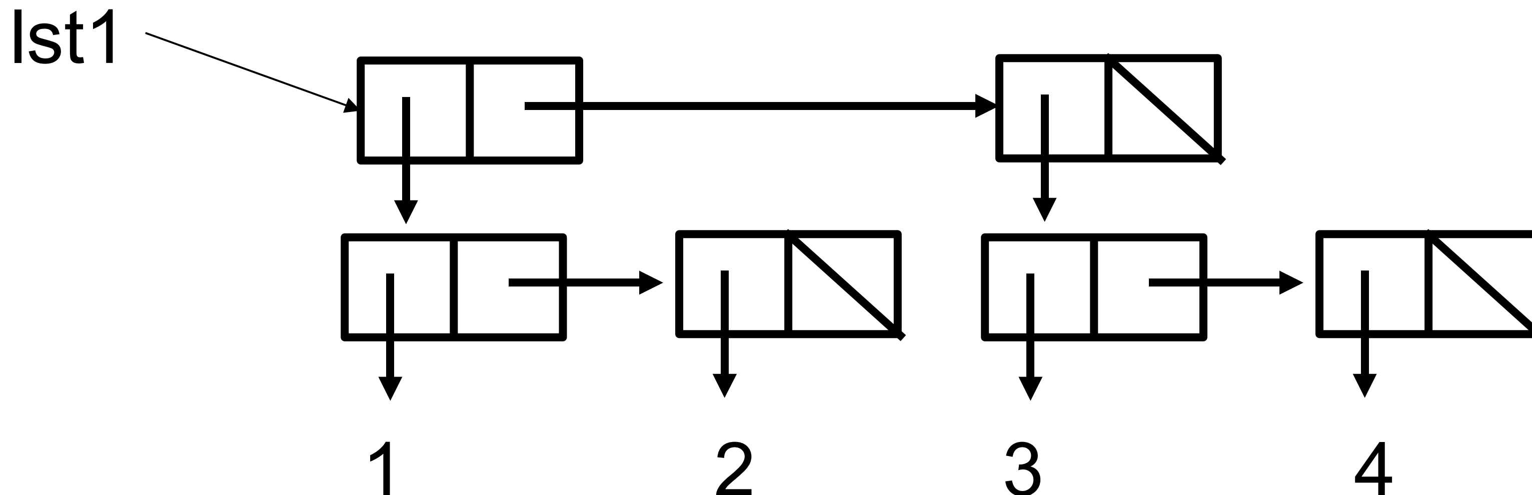


The value returned by list is a pointer that points to
The first pair in the list

Lists

- Anything can be in a list

(define lst1 **(list (list** 1 2) (**list** 3 4**)))**

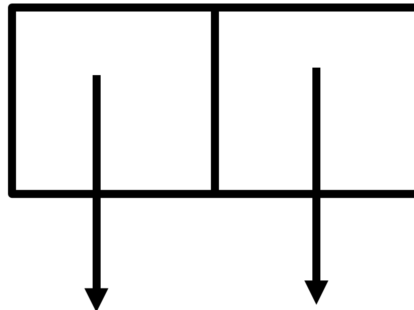


List Procedures

- (**list-ref** *lst n*)
  - (list-ref lst 0) returns the first element
- (**append** *lst1 lst2*) returns a new list with *lst2* merged t the back of *lst1*.

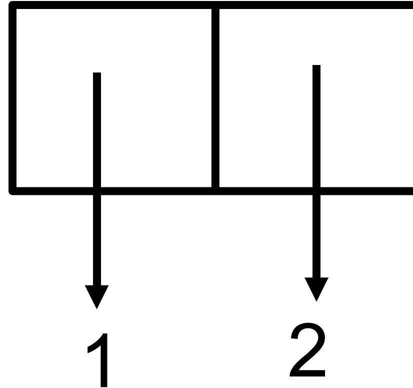
Pairs

- A list is built out of pairs
- A pair is box with two pointers
- To create a pair use **cons**
- A pair is also called a cons object

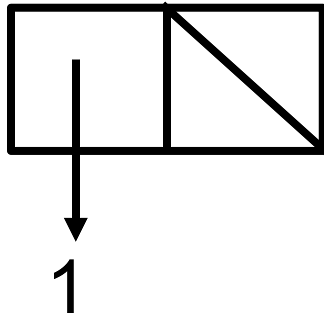


Pairs

- (**cons** 1 2)



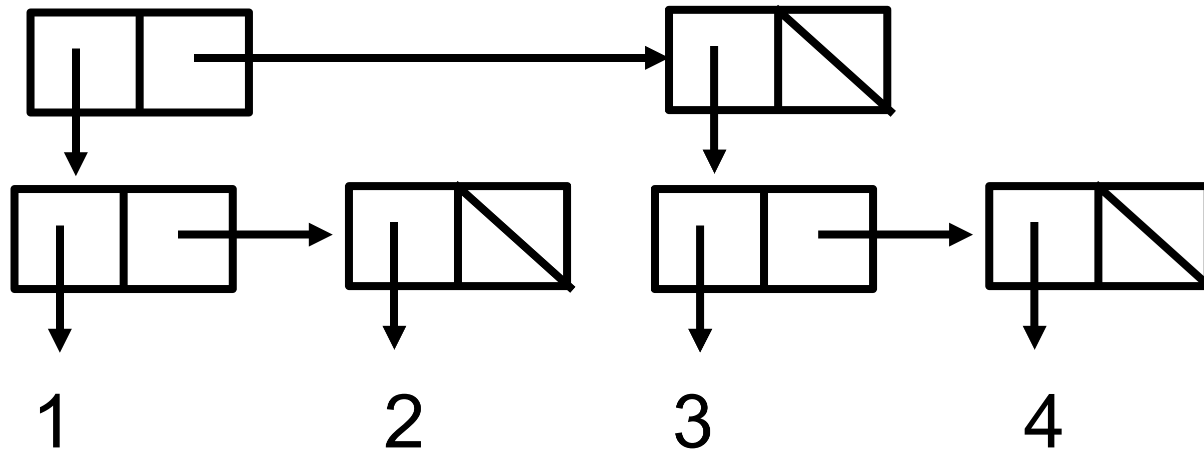
- (**cons** 1 null)



Building lists

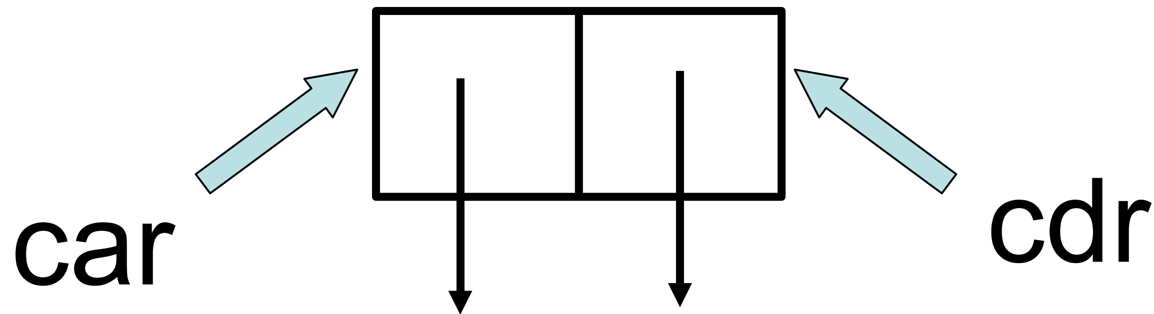
- We can use **cons** to build a list
- Draw the box and pointer diagram for
- (**cons** 1 (**cons** 2 (**cons** 3 (**cons 4 null**))))

Build this lists out of cons



Getting stuff out of pairs

- We can put stuff in a cons object, but how do we get stuff out?



Examples

```
(car (cons 1 2))
(cdr (cons 1 2))

(define lst (list 1 2))

(car lst)
(cdr lst)
(cdr (cdr lst))
```

Examples

```
(define lst (list (list 1 2 3) (list 1) (list 2 3)))
(car lst)
(car (car lst))
(cdr (car lst))
(car (cdr lst))
```

Problem

- Write the string of **car** and **cdr** operations that will return 4

- ((7) (6 5 4) (3 2) 1)
- (7 (6 (5 (4 (3 (2 (1)))))))
- (7 ((6 5 ((4)) 3) 2) 1)

Writing a simple list procedure

- Define **list-length** that computes the length of a list.
- What’s the recursive case?

- What’s the base case?

Checking for the empty list

- The base case is an empty list.
- How do we check for an empty list?
  - **null?** Predicate
- (null? lst) evaluates to #t if lst is the empty list

length

- Plan:
  - Base Case: Empty list -> 0
  - Recursive Case: 1 + length of “rest of the list”

- (define (length list)

  to-be-completed)