CSCI 334:
Principles of Programming Languages

Lecture 7: Lisp, part II

Instructor: Dan Barowy

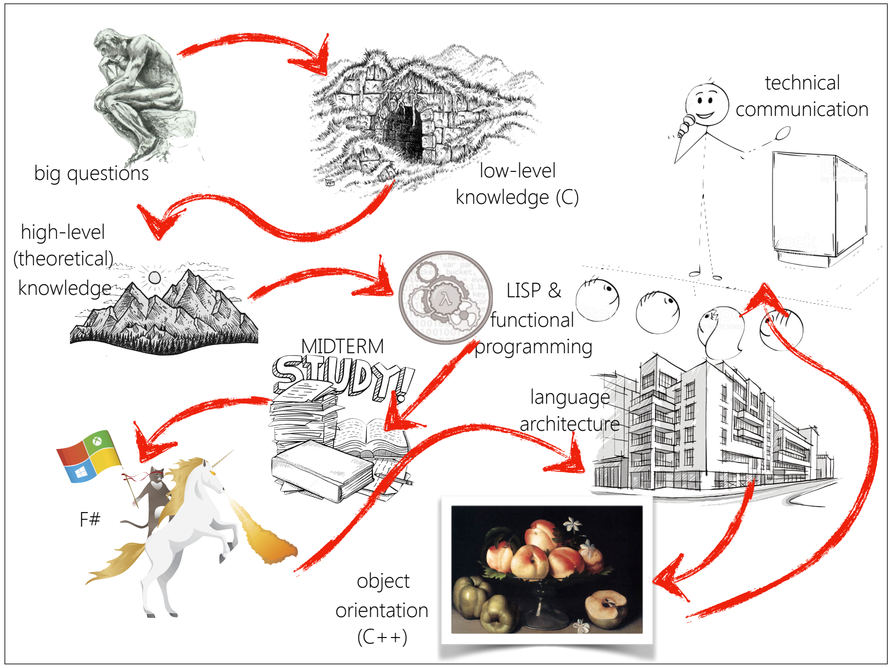
Williams

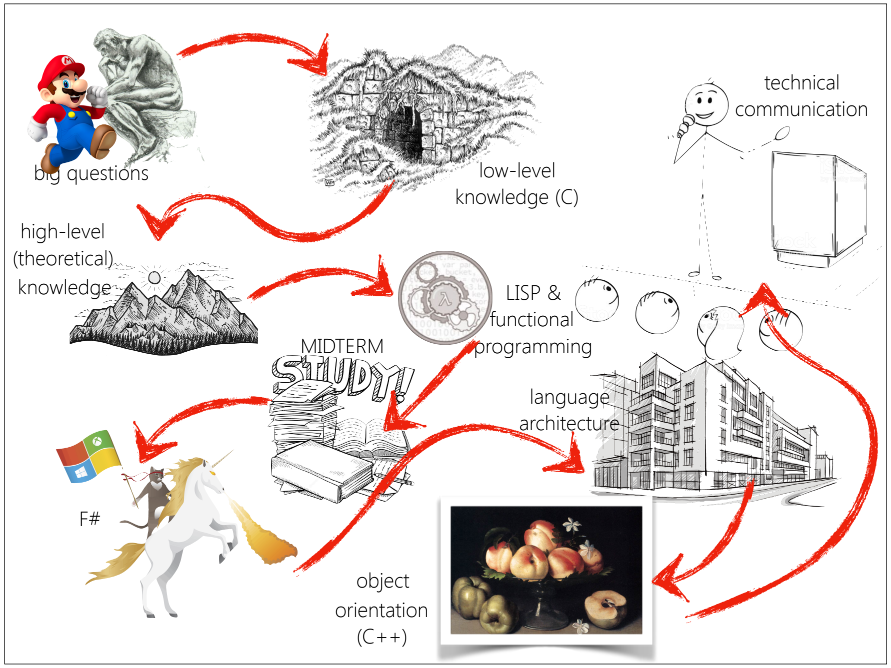
Announcements

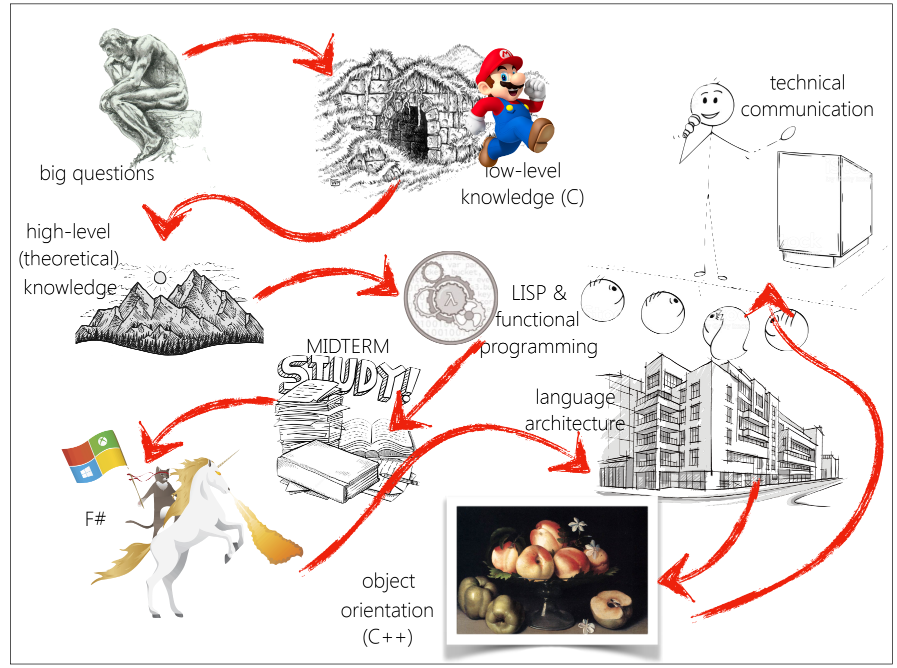
- Lab 4 due Sunday by 11:59pm
- Scheduled power outage: this Sunday at 10pm until Monday at 9am
  - **All CS lab machines**
  - All CS servers
- Colloquium: 2:30pm in Wege Auditorium (TCL 123)
  - "Adventures in Hybrid Architectures for Intelligent Systems," Nate Derbinsky, Northeastern

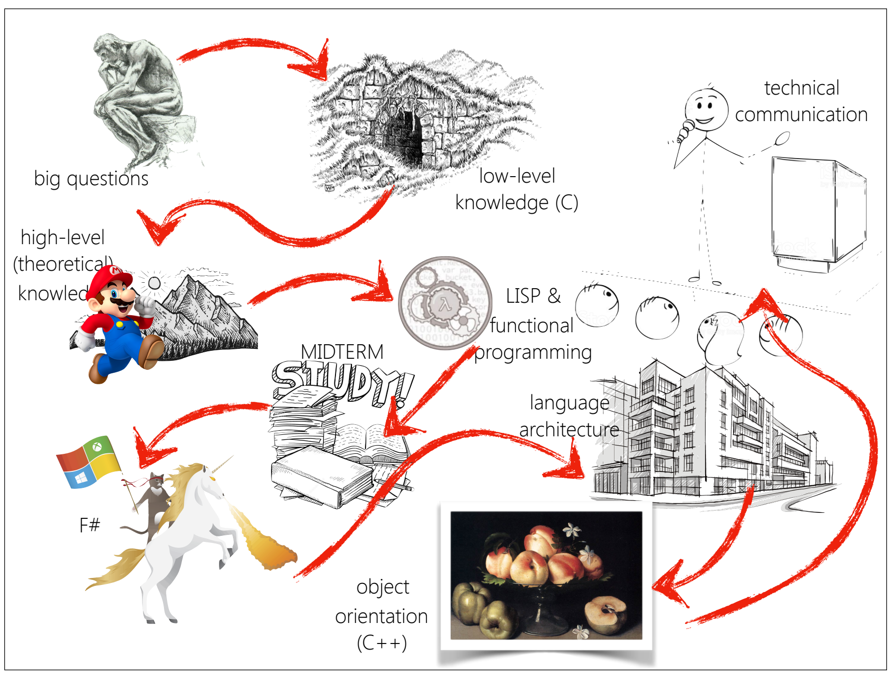
Outline

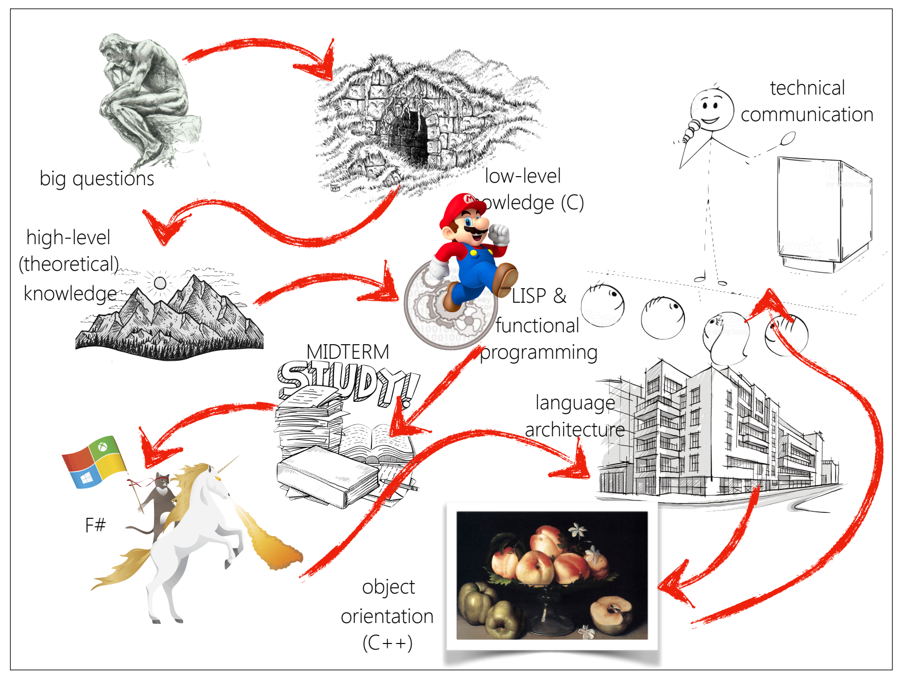
1. Happy/sad cards
2. More LISP
3. Garbage Collection

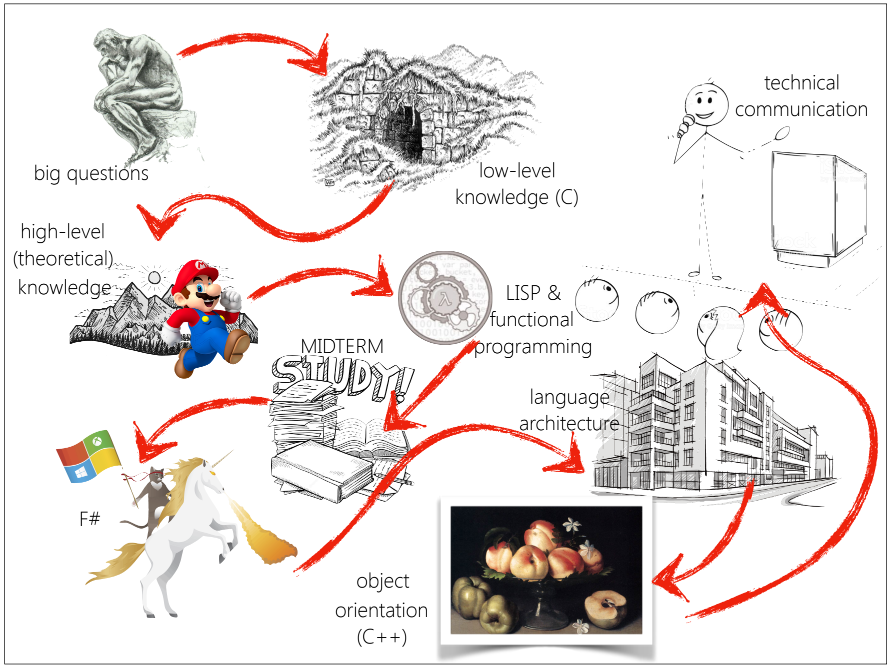


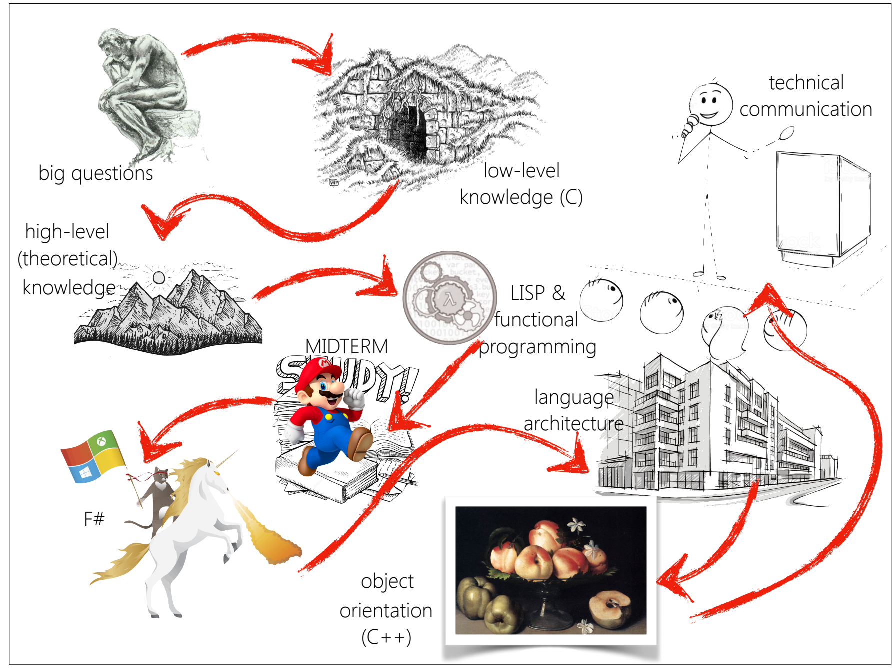

questions
low-level
knowledge (C)
technical
communication
high-level
(theoretical)
knowledge
LISP &
functional
programming
MIDTERM
STUDY!
language
architecture
F#
object
orientation
(C++)


big questions
ow-level
knowledge (C)
technical
communication
high-level
(theoretical)
knowledge
LISP &
functional
programming
MIDTERM
STUDY!
language
architecture
F#
object
orientation
(C++)


big questions
low-level
knowledge (C)
technical
communication
high-level
(theoretica
knowled
LISP &
functional
programming
MIDTERM
STUDY!
language
architecture
F#
object
orientation
(C++)


big questions
low-level
owledge (C)
technical
communication
high-level
(theoretical)
knowledge
LISP &
functional
programming
MIDTERM
STUDY!
language
architecture
F#
object
orientation
(C++)





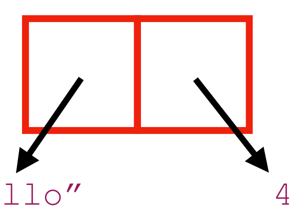
Midterm Exam

- Friday, March 20, in class

Lisp syntax: data structure

- Historically, Lisp has exactly one data structure: the **cons cell**.
- The “cons cell” allows “composing” values

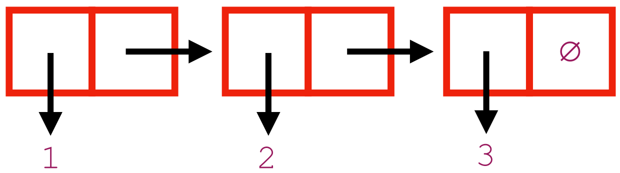
```
(cons “hello” 4)
```



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

```
(cons 1 (cons 2 (cons 3 nil)))
```



- Lisp has a shorthand for this:

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

"Recursive Functions […]" (McCarthy)

Lisp	C
`car`	`head`
`cdr`	`tail`
`cons`	`prepend`

Lisp syntax: **`car`** and **`cdr`**

- Access the first element of a cons cell with `car`

```
(car (cons 1 2)) = 1
```

- Access the second element with `cdr`

```
(cdr (cons 1 2)) = 2
```

- What's the value of the following expression?

```
(car '(1 2 3))
```

- What about this?

```
(cdr '(1 2 3))
```

Lisp syntax: functions

- Everything else is a function (or "special form")
- There are a bunch of built-in functions

`(car …)`

`(cdr …)`

`(append …)`, etc.

- And you can define your own

```
(defun my-func (arg) (value))
```

Lisp syntax: conditionals

- In Lisp, `if`/`else` is called `cond`

  ```
  (cond (test₁ value₁)
   …)
  ```

- E.g., `(cond ((eq 1 x) (cons x xs)) …)`
- Does the same as the Java

  ```
  if (x == 1) {
    xs.add(x);
  } …
  ```

Lisp syntax: conditionals

- `cond` is more general than `if`/`else`.

  ```
  (cond (test₁ value₁)
        (test₂ value₂)
   …)
  ```

demo

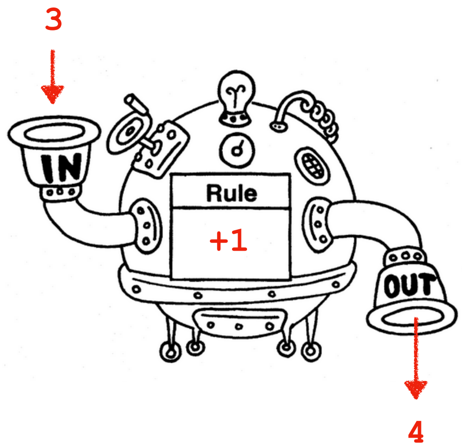
Lisp syntax: conditionals

```
(defun only-positives (xs)
  (cond
    ; empty list
    ((eq xs nil) nil)
    ; element is positive
    ( (> (car xs) 0)
      (cons (car xs) (only-positives (cdr xs)))
    )
    ; element is not positive
    ( t
      (only-positives (cdr xs))
    )
  )
)
```

Three amazing concepts from LISP

- First-class functions
- Higher-order functions
  - map
  - fold

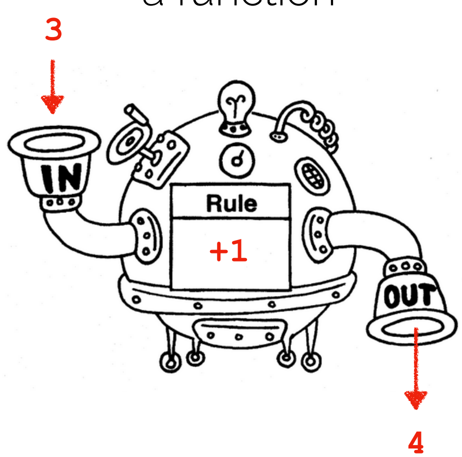
a function



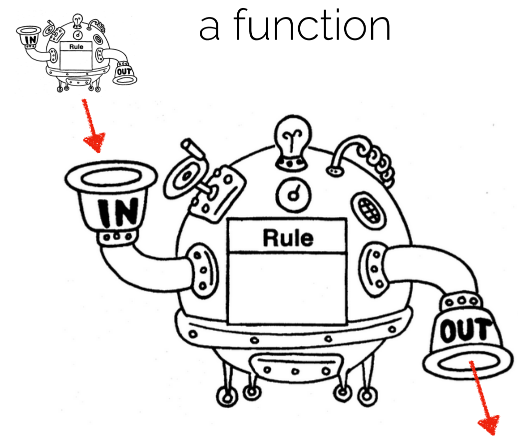
“first class” function

Functions are **values** in a functional programming language

a function

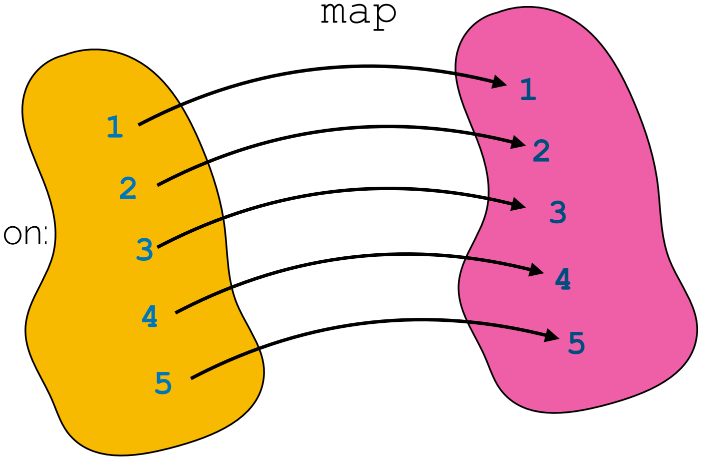


a function



map

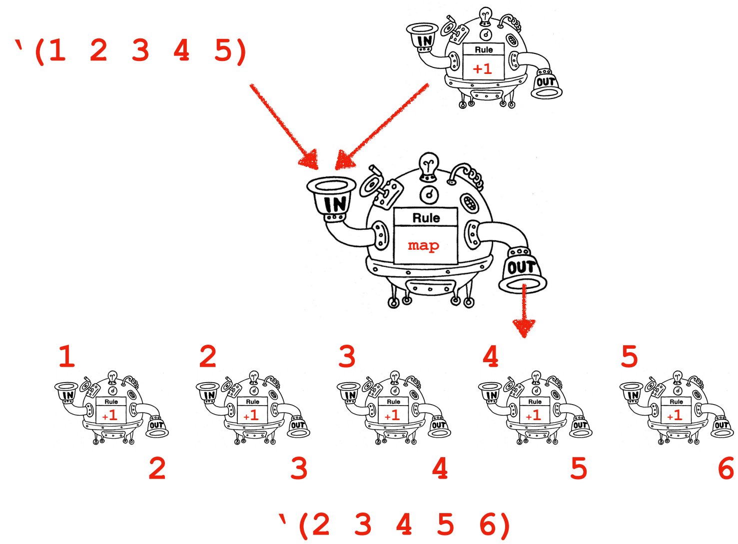
Intuition:



Like a `for` loop, but without mutable variables

```
(mapcar (lambda (x) (+ x 1)) '(1 2 3 4 5))
```

map



fold

Intuition:

fold left

```
(reduce #'+ '(1 2 3) :initial-value 0)
```

```
acc = 0, '(1 2 3)
acc = 0+1, '(2 3)
acc = 1+2, '(3)
acc = 3+3, nil
returns acc = 6
```

fold right

```
(reduce #'+ '(1 2 3):initial-value 0
                    :from-end t)
```

```
'(1 2 3), acc = 0
'(1 2), acc = 0+3
'(1), acc = 2+3
nil acc = 5+1
returns acc = 6
```

what does this print?

```
(reduce #'append '((2) (0))
    :initial-value '(w i l l i a m s))
```

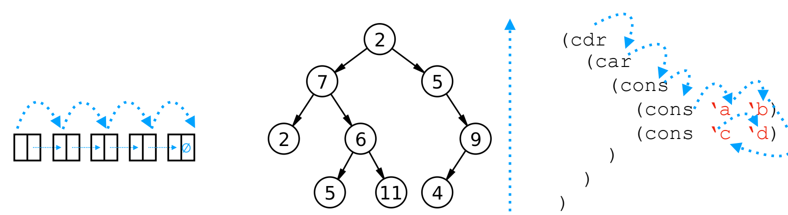
how about?

```
(reduce #'append '((2) (0))
    :initial-value '(w i l l i a m s)
    :from-end t)
```

fold

structural recursion → fold it!

(in a nutshell: any problem that recurses on a subset of input)



list length   tree height   evaluation

That's pretty much it!

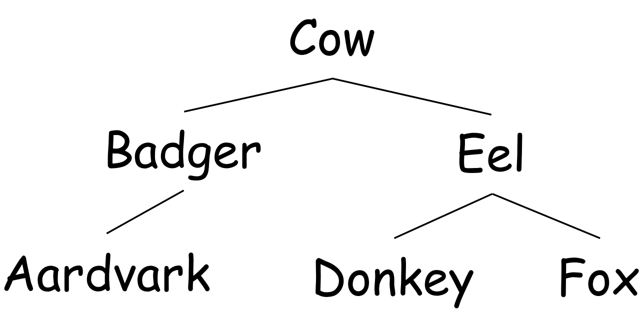
- See "LISP Notes" for all the syntax you need to know on course webpage

Activity

list length

```
(length-list '(1 2 3 4 5 6)) ↠ 6
```

Activity



Activity

Write a function (using `mapcar`) that replaces the number 3 in a list with the
number 6

```
(mapcar #'my-replace '(1 2 3 4 5 6))
          '(1 2 6 4 5 6)
```

Activity

Write a function (using `mapcar`) that replaces the number 3 in a list with the

number 6

```
       (defun my-replace (x)
         (cond
           ((equal x 3) 6)
           (t x)
         )
       )
(mapcar #'my-replace '(1 2 3 4 5 6))
            '(1 2 6 4 5 6)
```

Automatic Memory Management

Memory management

- C:

  When you want to use a variable, you have to

  allocate it first, then *decallocate* it when done.

```
MyObject *m = malloc(sizeof(MyObject));
m->foo = 2;
m->bar = 3;
… do stuff with m …
free(m);
```

Memory management

- Java:

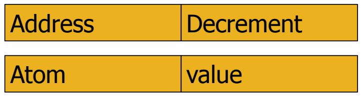
  You barely need to think about this at all.

```
MyObject m = new MyObject(2,3);
… do stuff with m …
```

- Same with LISP!

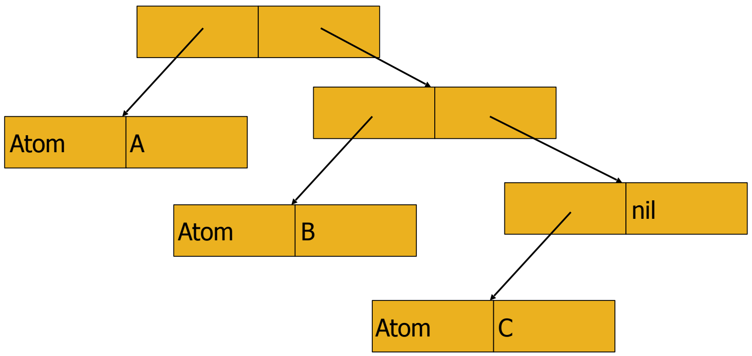
```
(cons 2 3)
```

Lisp memory model

Cons cell:


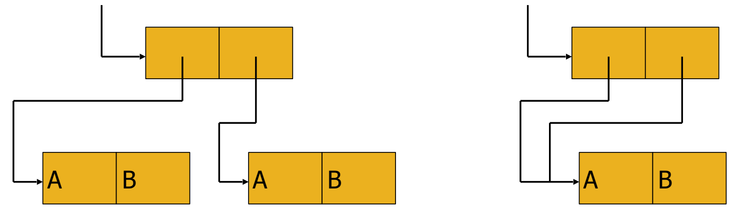
Atom:

```
(cons 'A (cons 'B (cons 'C nil)))
```



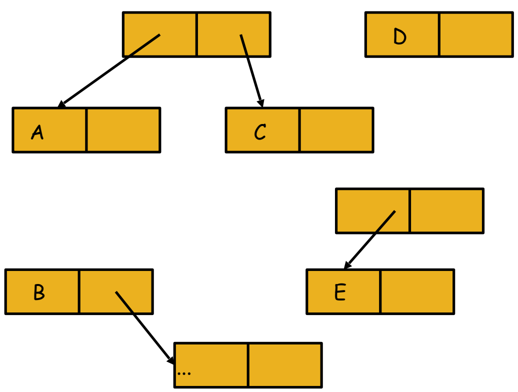
Sharing data

(a) (b)



- Which is the result of evaluating
  `(cons (cons 'A 'B) (cons 'A 'B))` ?

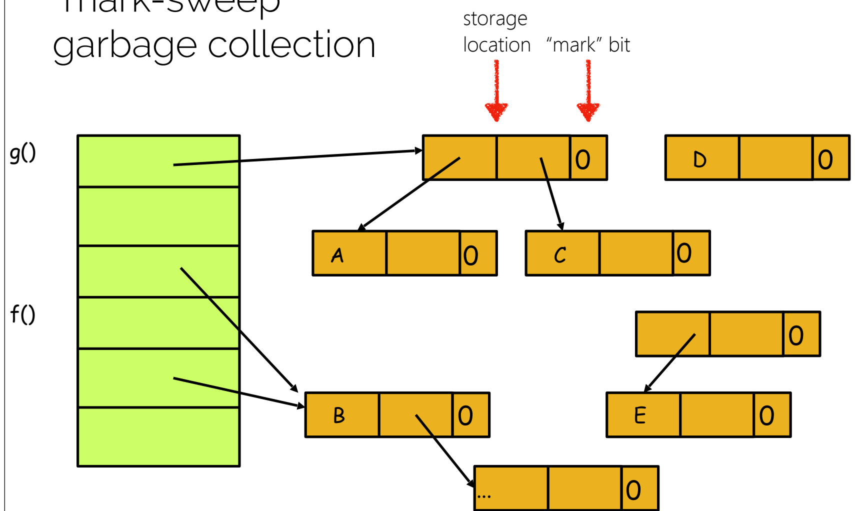
Garbage collection



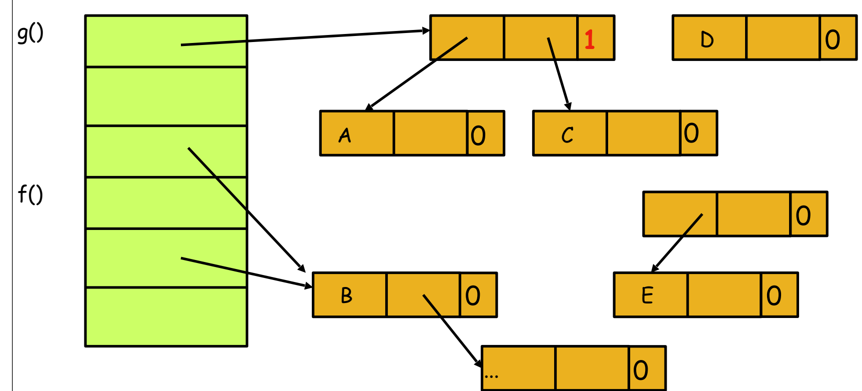
Garbage collection

A **garbage collection algorithm** is an algorithm that determines whether the storage, occupied by a value used in a program, can be reclaimed for future use. Garbage collection algorithms are often tightly integrated into a programming language **runtime**.

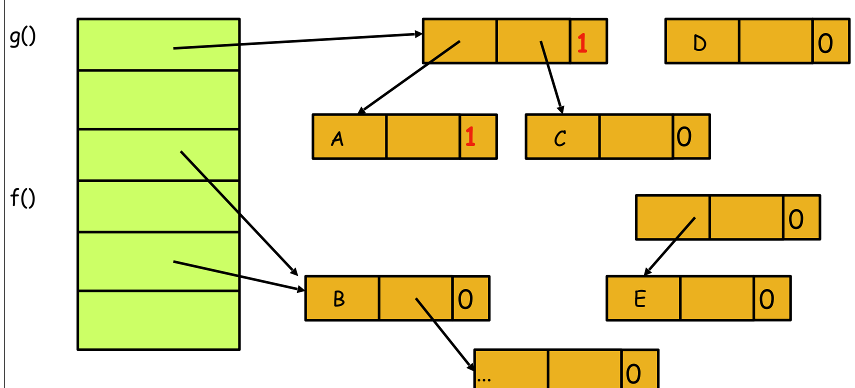
“mark-sweep” garbage collection



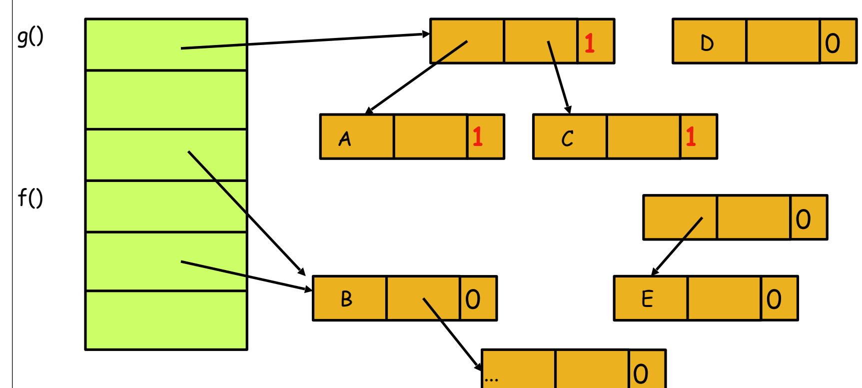
1. Mark reachable cells



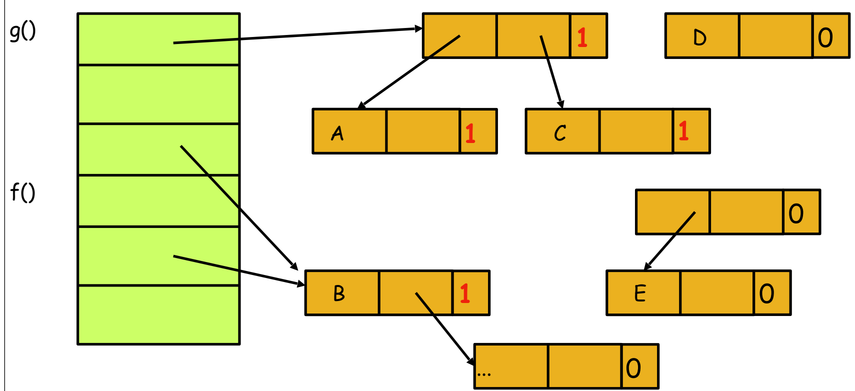
1. Mark reachable cells



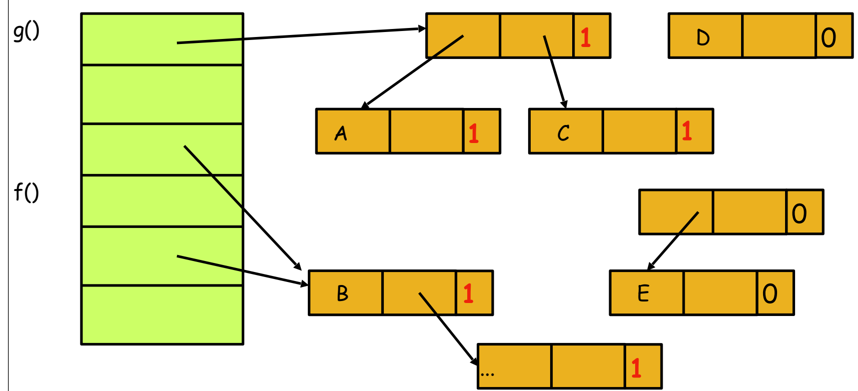
1. Mark reachable cells



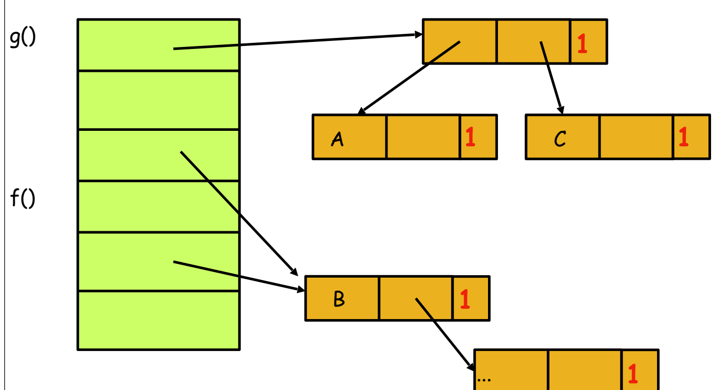
1. Mark reachable cells



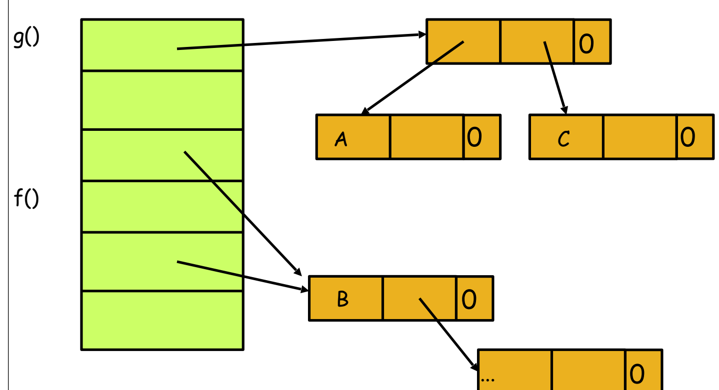
1. Mark reachable cells



2. Free ("sweep") unreachable cells



3. Clear tags



More lambda calculus practice?

1. `(λx.x)(λx.xx)(λx.xa)`
   reduces to: **`aa`**
2. `(λx.x)(λy.yy)(λz.za)`
   reduces to: **`aa`**
3. `(λx.λy.xyy)(λa.a)b`
   reduces to: **`bb`**
4. `(λx.xx)(λy.yx)z`
   reduces to: **`xxz`**
5. `(λx.(λy.(xy))y)z`
   reduces to: **`zy`**

Recap & Next Class

Today we covered:

More LISP

Garbage collection

Next class:

Halting Problem