# Programming Abstractions

**Lecture 2: Pairs, lists, and define**

**Stephen Checkoway**

# Procedures for pairs and lists

# Procedures for working with pairs
## Construct a pair

Lists are pairs whose second element is a list so these procedures work with lists

`cons` — (Construct) Create a pair
‣ `(cons 'x 'y)` creates the pair `('x . 'y)`
‣ `(cons 2 3)` creates the pair `'(2 . 3)`
‣ `(cons 5 null)` creates the list `'(5)`

If `lst` is a list, then `(cons x lst)` returns a new list starting with `x` and followed by the elements of `lst`
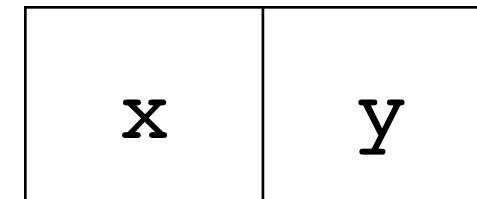‣ `(cons 8 (list 1 2 3))` produces the list `'(8 1 2 3)`

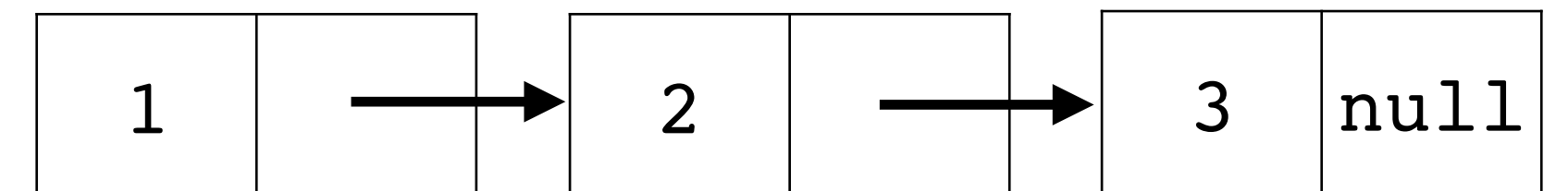What does `(cons 'a (cons 'b (cons 'c '())))` produce?

# Cons cells
## Construct a pair

`(cons x y)` creates a *cons-cell*

| x | y |
|---|---|

`(cons 1 (cons 2 (cons 3 null)))` produces

| 1 | |→| 2 | |→| 3 | null |

You'll notice that this is a linked list!

This is exactly the same list that's produced by `(list 1 2 3)`

# Adding to a list

If we have a list `lst` and an element `x`, prepend `x` to `lst`: `(cons x lst)`
‣ E.g., `(cons "c" (list "a" "b"))` => `'("c" "a" "b")`
‣ This works because the second argument to `cons` is a list so the result is a list

What if we want to append `x` to `lst`? Can we use `(cons lst x)`?
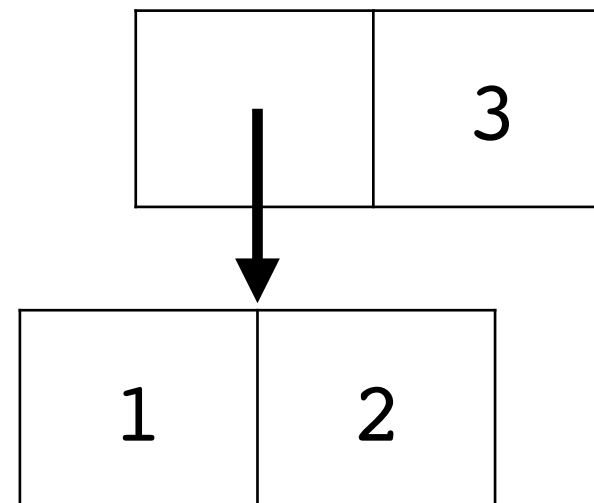‣ I.e., will `(cons '(1 2 3) 4)` produce `'(1 2 3 4)`?

# Aside: Trees from pairs

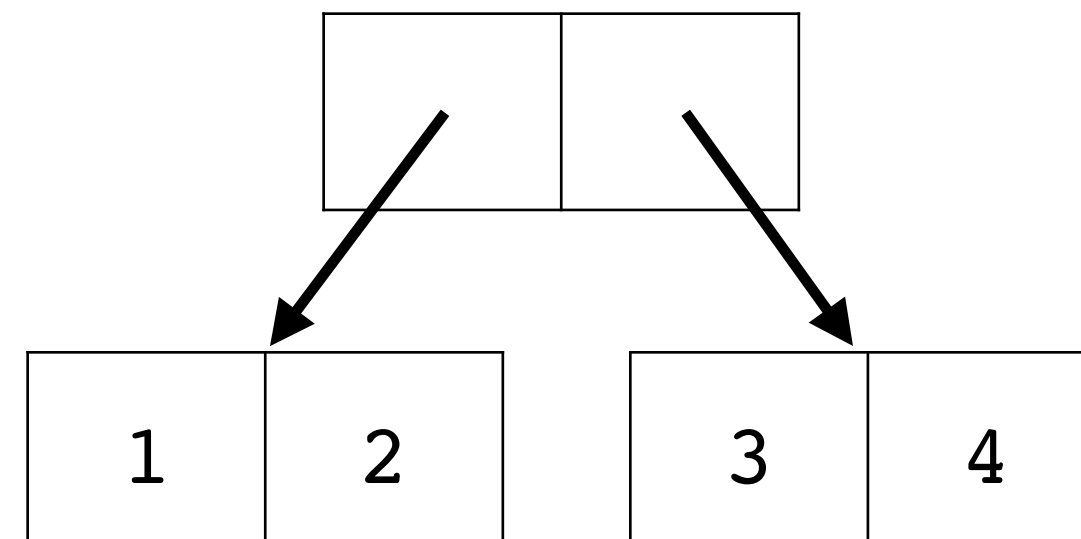Nothing says our cons-cells need to be used for lists

`(cons #t 5)`

| #t | 5 |
|----|---|

`(cons (cons 1 2) 3)`

|   | 3 |
|---|---|

| 1 | 2 |
|---|---|

`(cons (cons 1 2)`
`      (cons 3 4))`

|   |   |
|---|---|

| 1 | 2 |
|---|---|

| 3 | 4 |
|---|---|

# Procedures for working with pairs

## Extract the first element of a pair

`car` — (Contents of the Address part of a Register*) Returns the first element of a pair (or the head of a list)

- ‣ `(car (cons 5 8))` (equivalently `(car '(5 . 8))`) returns 5
- ‣ `(car '(1 2 3 4))` returns 1
- ‣ `(car (1 2 3 4))` is an error because `(1 2 3 4)` is invalid

\* This terminology comes from the IBM 704, an ancient computer

# Procedures for working with pairs
## Extract the second element of a pair

`cdr` — (Contents of the Decrement part of a Register\*) Returns the second element of a pair (or the tail of a list); pronounced "could-er"

- `(cdr (cons 5 8))` (equivalently `(cdr '(5 . 8))`) returns 8
- `(cdr '(1 2 3 4))` returns the list `'(2 3 4)`
- `(cdr '(5))` returns the empty list, DrRacket will display `'()`

\* This terminology comes from the IBM 704, an ancient computer

`car` returns the first element of a pair
`cdr` returns the second element of a pair

If `lst` is a list how do we get the second element of `lst`? E.g., if `lst` is `'(2 3 5 7)`, the code should return 3

A. `(car lst)`

B. `(cdr lst)`

C. `(car (cdr lst))`

D. `(cdr (car lst))`

E. `(cdr (cdr lst))`

# Procedures for working with lists
## (Traditional)

Scheme has a bunch of shorthands for combining car and cdr to extract elements from lists (or any data structure built from cons-cells)

‣ `(cadr lst)` is `(car (cdr lst))`
   ```
   (cadr '(1 2 3 4)) => (car (cdr '(1 2 3 4)))
                       => (car '(2 3 4)) => 2
   ```
   I.e., it extracts the second element of a list

‣ `(caddr lst)` is `(car (cdr (cdr lst)))`

‣ `(cdar lst)` is `(cdr (car lst))`
   ```
   (cdar '((1 2 3) (4 5 6))) => (cdr '(1 2 3)) => '(2 3)
   ```

‣ Many others, e.g., `caddr`, `cadddr`, all with their own pronunciations

# Procedures for working with lists
## (Modern)

The traditional functions work on arbitrary data structures (like trees) built from pairs

Unless we're working with pairs explicitly, we don't need to use `car`, `cdr`, `cadr`, or any other the others as we have better named functions that only work on lists

▸ `(first '(1 2 3)) => 1`
▸ `(rest '(1 2 3)) => '(2 3)`
▸ `(second '(1 2 3)) => 2`
▸ `(third '(1 2 3)) => 3`
▸ `fourth, fifth, sixth, seventh, eighth, ninth, tenth`
▸ `(last '(1 2 3)) => 3`

Recall, we can use `empty` for the empty list in place of `null`

# Defining data and procedures

# Special forms

We'll see how DrRacket evaluates expression in more detail shortly, e.g., how `(+ 2 3)` evaluates to 5

Essentially, when presented with a list `(foo arg1 arg2 …)` it looks at the first element of the list (here, `foo`)
‣ If `foo` is a *special form* (e.g., `and`, `or`, `define`, `if`, `cond`), it takes steps specific to that particular special form
  - E.g., `(and exp1 exp2)` will evaluate `exp1`. If it's `#f`, then the whole expression is `#f`. Otherwise, it'll evaluate `exp2` and return the result
‣ If `foo` is a procedure (e.g., `+`, `*`, `first`, `list`, `string-append`) it applies the procedure to the arguments and returns the result
‣ Otherwise, it's an error.
  - E.g., `(1 2 3)` is an error; `1` is neither a special form nor a procedure

# Define a new variable

`(define id s-exp)`

The define special form binds an identifier (a variable) to a value
- This modifies the *environment*, the mapping of identifiers to values
- `(define WIDTH 200)`
- `(define AREA (* WIDTH WIDTH))`
- `(define CS-PROFESSORS '("Adam" "Bob" "Cynthia"))`
  `(third CS-PROFESSORS) => "Cynthia"`

The expression is evaluated so `AREA` will be bound to the value `40000` rather than the expression `(* WIDTH WIDTH)`

One of the most common things we'll want to do is bind a procedure to an identifier

# Creating procedures

Procedures are creating using the `lambda` (or λ) special form
‣ `(lambda parameters body…)`
   - `parameters` is an unevaluated list of identifiers which will be bound to the values of the procedure's arguments when procedure is called
   - body is a sequence of s-expressions that form the body of the procedure, they're evaluated in turn

Examples
‣ `(lambda (x y)`
    `(/ (+ x y) 2))`
‣ `(λ (name)`
   `(display "Hello ")`
   `(display name))`

# Binding identifiers to procedures

Unlike functions in C, procedures in Scheme are **values**, we can bind identifiers to procedures
```
(define mean
  (λ (x y)
    (/ (+ x y) 2)))
(mean 37 42) => 39 1/2
```

# Swapping the first two elements of a list

Let's define a procedure `swap` that takes a list as input and returns a new list
with the first two elements swapped so
```
(swap '(a b c d))
```
returns
```
'(b a c d)
```

# Binding identifiers to procedures

Binding identifiers to procedures is so common, there's a special syntax for it
‣ `(define (name parameters) body…)`

```
(define (mean x y)
  (/ (+ x y) 2))
```

# Multiple ways to define procedures

add1 takes a single integer argument and returns the result of adding 1 to it.

```
(define add1
  (lambda (x)
    (+ x 1)))

(define add1
  (λ (x)
    (+ x 1)))

(define (add1 x)
  (+ x 1))
```

# Closures: procedure values

The expression of `(lambda parameters body…)` evaluates to a *closure* consisting of
‣ The parameter list (a list of identifiers)
‣ The body as un-evaluated expressions (often just one expression)
‣ The environment (the mapping of identifiers to values) **at the time the lambda expression is evaluated**

# Applying a closure to arguments

```
(define A 10)
(define add-a
  (λ (x)
    (+ x A)))
```

Calling the closure extends the closure's environment with its parameters bound to the arguments

```
(add-a 20)
```

The closure's body is evaluated with this new environment

Environment of the closure

| A | 10 |
|---|----|

Environment of the call

| A | 10 |
|---|----|
| x | 20 |

# Closures are values: we can return them!

The result of `(λ (x y z) …)` is a closure and closures are values
‣ Hence `(define fun (λ (x y z) …))` defines `fun` to be the closure and we can call `(fun 1 2 3)`

But we can also return closures from procedures

```
(define f
  (λ (x)
    (λ (y)
      (+ x y))))

(define (f x)
  (λ (y)
    (+ x y)))
```

22

```
(define g
  (λ (x)
    (λ (y)
      (- x y))))
```

What is `(g 3 4)`?

A. 3

B. 4

C. -1

D. 1

E. An error