Defining Racket: Pairs, Lists, and GC

+lists.rkt

Topics

- Compound values:
  - *Cons cell*: pair of values
  - *List*: ordered sequence of parts
- Programming with pairs and lists
- Implementation consideration: garbage collection (GC)

Pairs: cons cells

***Cons**truct a cons cell holding 2 values*:
`cons` built-in function, takes 2 arguments

***Access** parts:*
`car` built-in function, takes 1 argument
    returns first (left) part if argument is a cons cell
`cdr` built-in function, takes 1 argument
    returns second (right) part if argument is a cons cell

mnemonic: `car` precedes `cdr` in alphabetic order

Names due to the IBM 704 computer assembler language
    (used for first Lisp implementation, 1950s)
    ***c**ontents of the **a**ddress/**d**ecrement part of **r**egister number*

cons *expressions* build cons *cells*

Syntax: `(cons e1 e2)`

`cons` is a function, so why define evaluation rules?

Evaluation:

1. Evaluate `e1` to a value `v1`.
2. Evaluate `e2` to a value `v2`.
3. The result is a cons ***cell*** containing ***`v1`*** as the left value and ***`v2`*** as the right value: `(cons v1 v2)`

$$\frac{E \vdash e1 \downarrow v1 \qquad E \vdash e2 \downarrow v2}{E \vdash (\texttt{cons}\ e1\ \ e2) \downarrow (\texttt{cons}\ v1\ \ v2)}\ [\text{cons}]$$

cons *cells* are values

Syntax: `(cons v1 v2)`

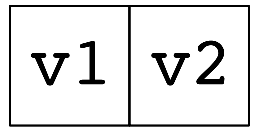
- `(cons 17 42)`
- `(cons 3.14159 #t)`
- `(cons (cons 3 4.5) (cons #f 5))`

So is `(cons 17 42)` a function application expression or a value?

$e ::= v \mid \ldots$

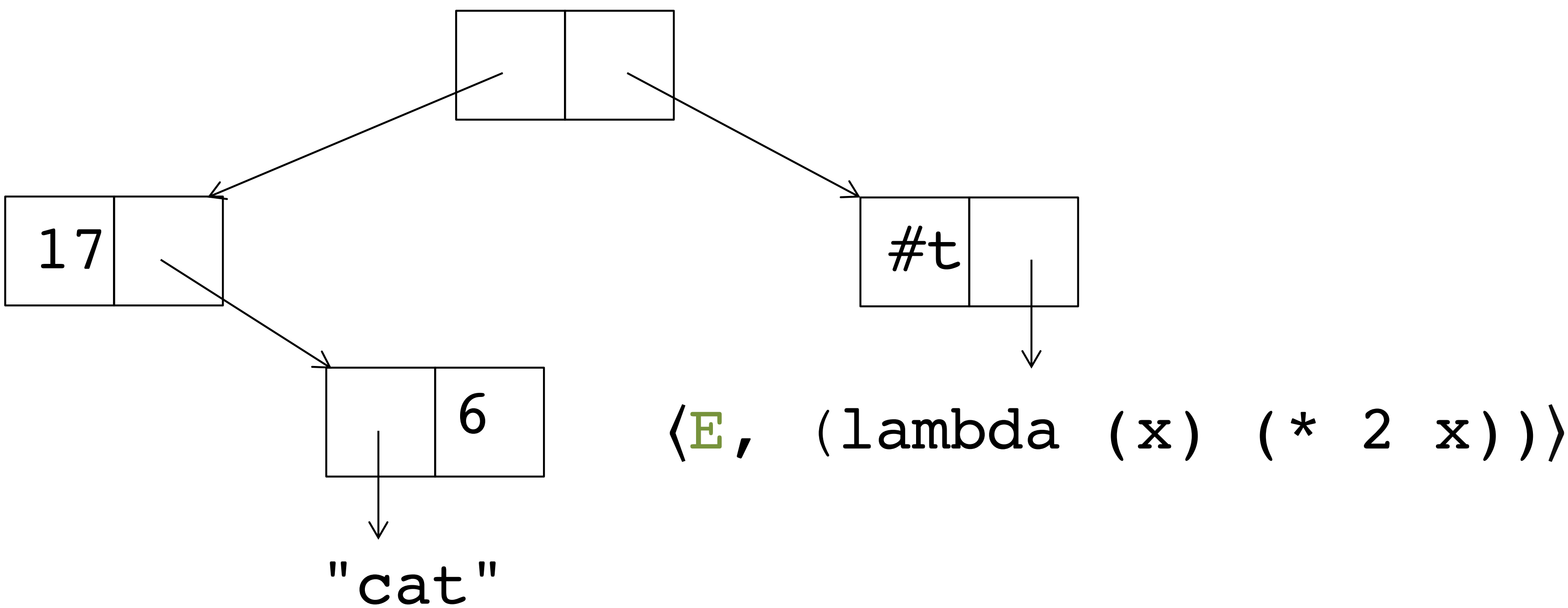
cons cell diagrams

`(cons v1 v2)`



Convention: put “small” values (numbers, booleans, characters) inside a box, and draw a pointers to “large” values (functions, strings, pairs) outside a box.

```
(cons (cons 17 (cons "cat" 6))
      (cons #t (lambda (x) (* 2 x))))
```



`car` and `cdr` expressions

Syntax: `(car e)`

Evaluation:

1. Evaluate **e** to a cons cell.
2. The result is the **left** value in the cons cell.

$$\frac{E \vdash e \downarrow (\texttt{cons v1 v2})}{E \vdash (\texttt{car e}) \downarrow \texttt{v1}} \ [\text{car}]$$

Syntax: `(cdr e)`

Evaluation:

1. Evaluate **e** to a cons cell.
2. The result is the **right** value in the cons cell.

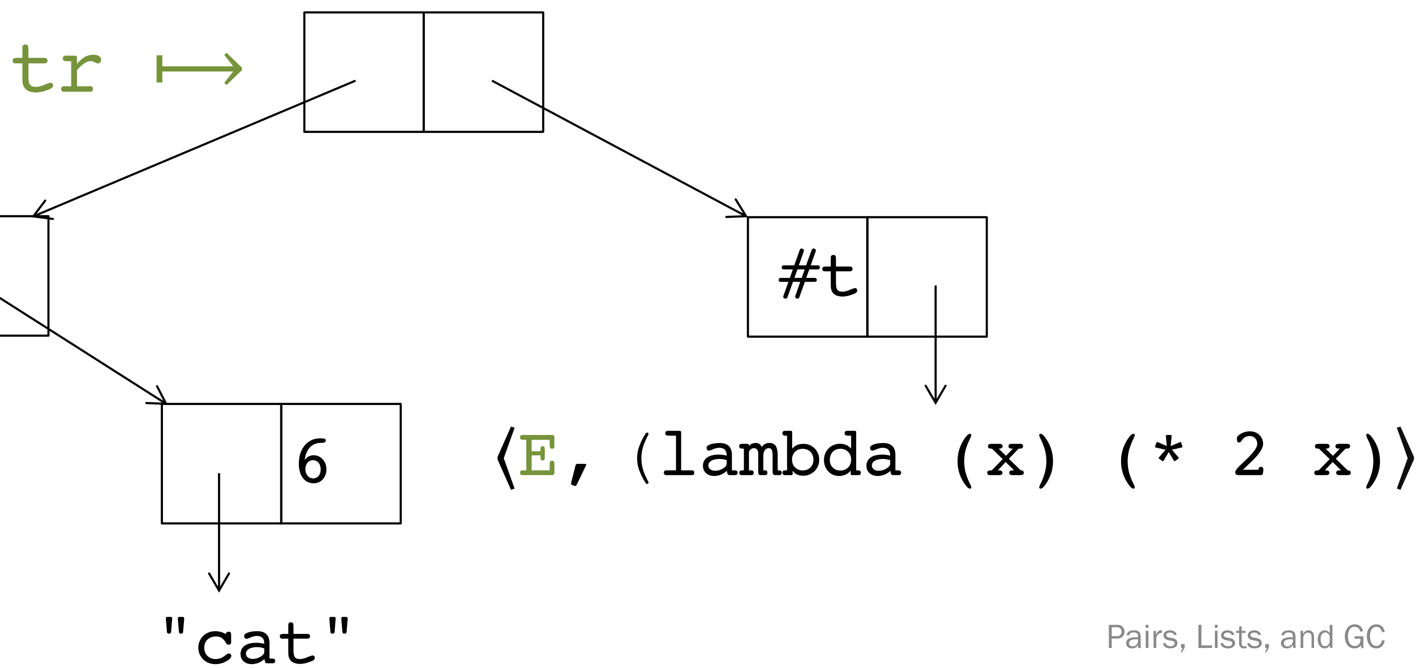
$$\frac{E \vdash e \downarrow (\texttt{cons v1 v2})}{E \vdash (\texttt{cdr e}) \downarrow \texttt{v2}} \ [\text{cdr}]$$

Practice with car and cdr

Write expressions using `car`, `cdr`, and `tr` that extract the five leaves of this tree:

```
(define tr (cons (cons 17 (cons "cat" 6))
                 (cons #t (lambda (x) (* 2 x)))))
```

```
tr ⟼ (cons (cons 17 (cons "cat" 6))
             (cons #t (lambda (x) (* 2 x)))), …
```



Rule check

What is the result of evaluating this expression?

```
(car (cons (+ 2 3) (cdr 4)))
```

Examples

```
(define (swap-pair pair)
    (cons (cdr pair) (car pair)))

(define (sort-pair pair)
    (if (< (car pair) (cdr pair))
        pair
        (swap pair)))
```

What are the values of these expressions?

```
(swap-pair (cons 1 2))

(sort-pair (cons 4 7))

(sort-pair (cons 8 5))
```

Lists

A list is one of:

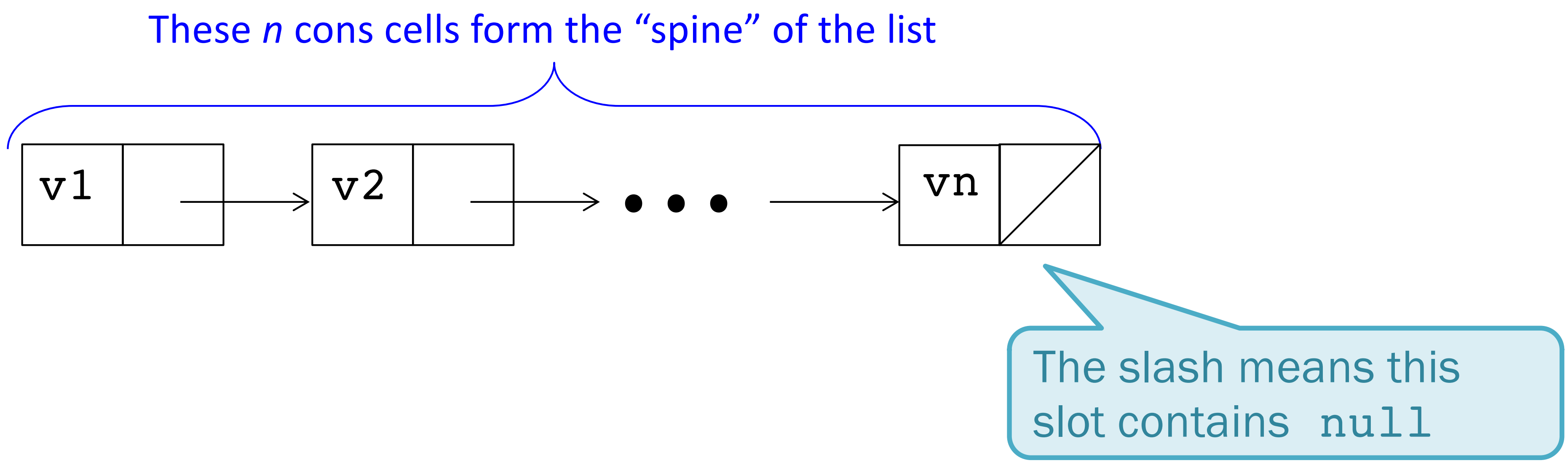
- The empty list: `null`
- A pair of the first element, $\texttt{v}_{\texttt{first}}$, and a smaller list, $\texttt{v}_{\texttt{rest}}$, containing the rest of the elements:

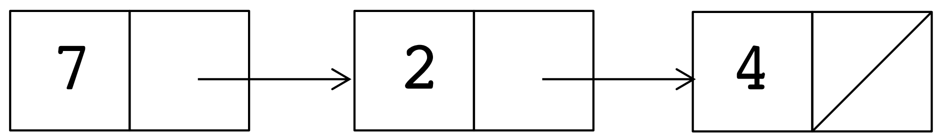
  $\texttt{(cons v}_{\texttt{first}}\texttt{ v}_{\texttt{rest}}\texttt{)}$

A list of the numbers 7, 2, and 4:

```
(cons 7 (cons 2 (cons 4 null)))
```

List diagrams





`list` as sugar*

- `(list)` desugars to `null`
- `(list e1 …)` desugars to `(cons e1 (list …))`

Example: `(list (+ 1 2) (* 3 4) (< 5 6))`

desugars to `(cons (+ 1 2) (list (* 3 4) (< 5 6)))`

desugars to `(cons (+ 1 2) (cons (* 3 4) (list (< 5 6))))`

desugars to `(cons (+ 1 2) (cons (* 3 4) (cons (< 5 6) (list))))`

desugars to `(cons (+ 1 2) (cons (* 3 4) (cons (< 5 6) null)))`

* Close enough for now, but actually a variable-argument function.

Quoted notation (only the basics)

Read Racket docs for more.

Symbols are values: `'a`

where ***a*** is any valid identifier or other primitive value

number and boolean symbols identical to values: `'#f` is `#f`

Atoms: symbols, numbers, booleans, null

Quoted notation of cons/list values:

- A cons cell `(cons 1 2)` is displayed `'(1 . 2)`
- `null` is displayed `'()`
- A cons cell `(cons 1 null)` is displayed `'(1)`
- A cons cell `(cons 1 (cons 2 null))` is displayed `'(1 2)`
- `(list 1 2 3)` is displayed `'(1 2 3)`
- `'(cons 1 2)` is the same as `(list 'cons '1 '2)`
- `(cons (cons 1 2) (cons 3 4))` is displayed `'((1 . 2) 3 . 4)`

List practice

```
(define LOL
  (list (list 17 19)
        (list 23 42 57)
        (list 115 (list 111 230 235 251 301) 240 342)))
```

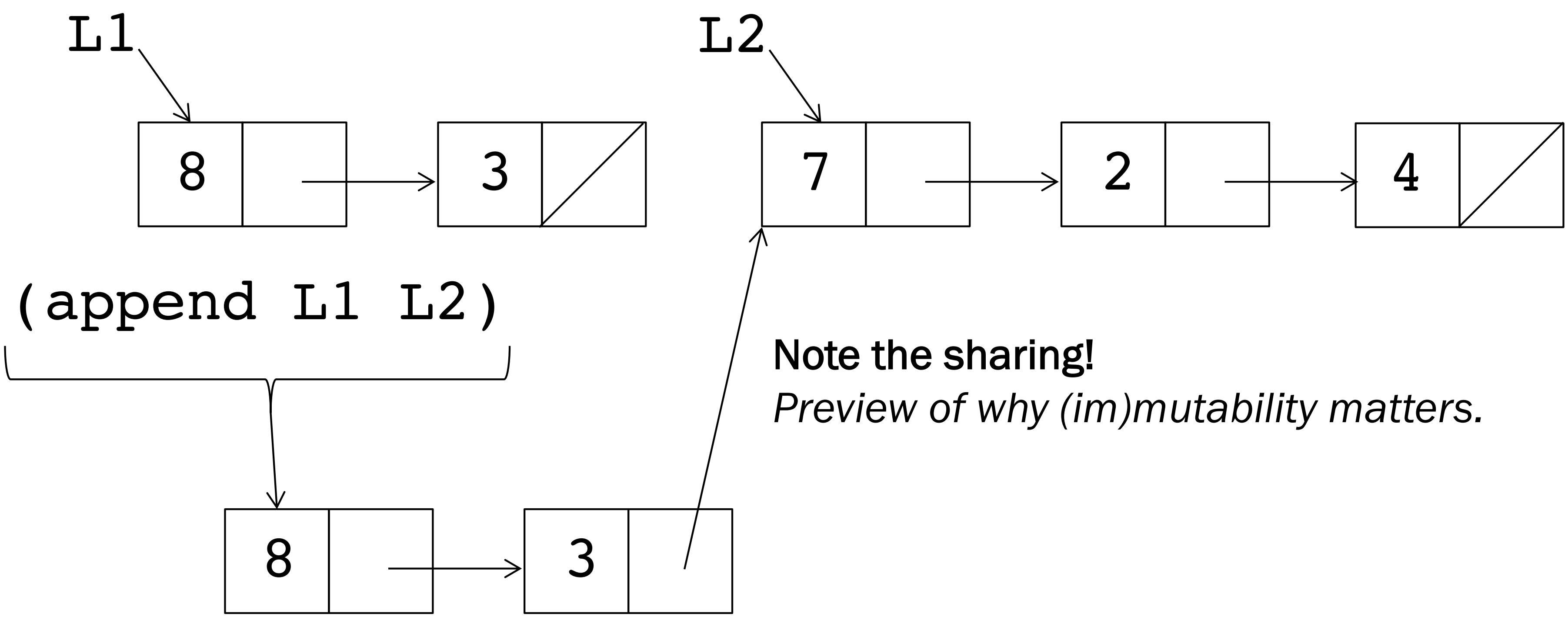
1. Draw the diagram for the value bound to LOL.
2. Write the printed representation of the value bound to LOL.
3. Give expressions using `LOL` (and no number values) that evaluate to the following values: `19`, `23`, `57`, `251`, `'(235 251 301)`
4. Write the the result of evaluating:

```
(+ (length LOL)
   (length (third LOL))
   (length (second (third LOL))))
```

append

```
(define L1 (list 8 3))
(define L2 (list 7 2 4))
```

The `append` function takes two lists as arguments and returns a list of all the elements of the first list followed by all the elements of the second list.



Note the sharing!
Preview of why (im)mutability matters.

List practice

```
(define L1 '(7 2 4))
(define L2 '(8 3 5))
```

For each of the following three lists:

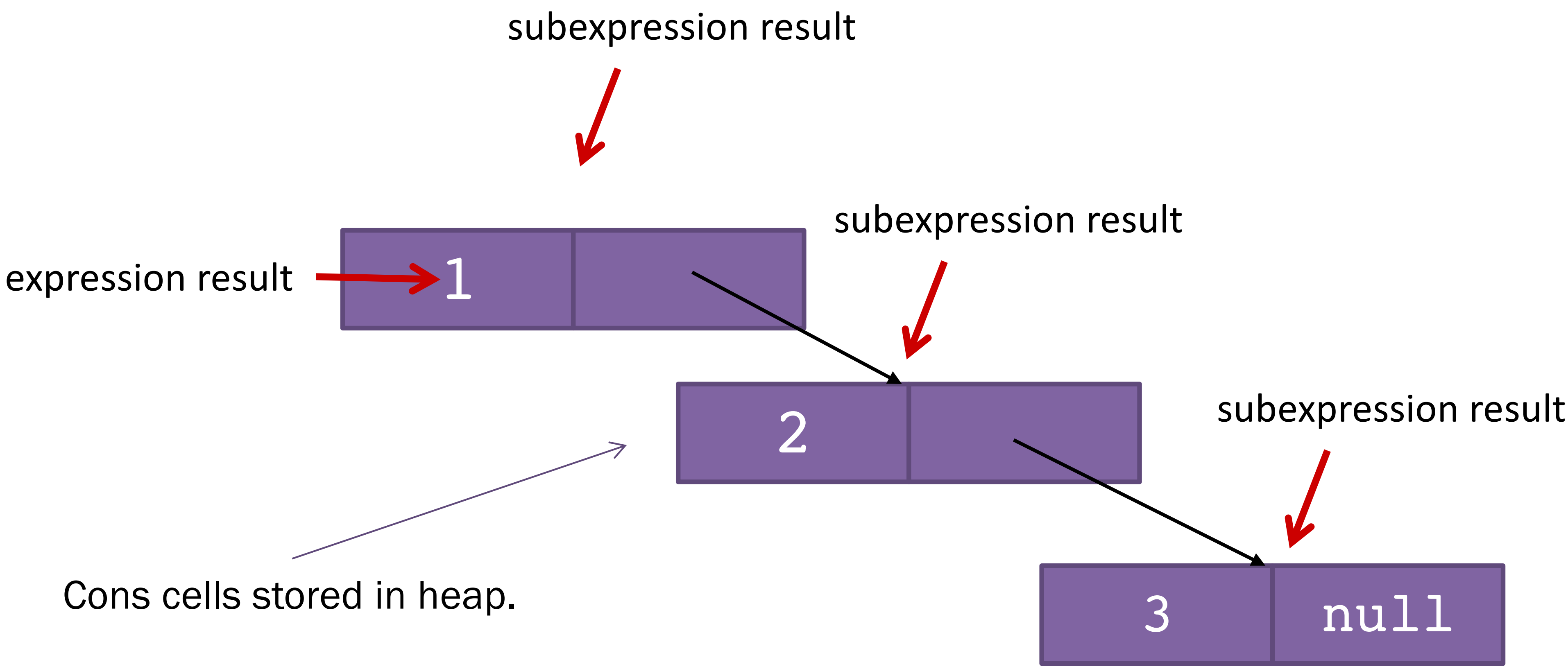
1. Draw the diagram for its value.
2. Indicate the number of cons cells ***created*** for its value.
   (Don't count pre-existing cons cells.)
3. Write the quoted notation for its value.
4. Determine the list length of its value .

```
(define L3 (cons L1 L2))
(define L4 (list L1 L2))
(define L5 (append L1 L2))
```

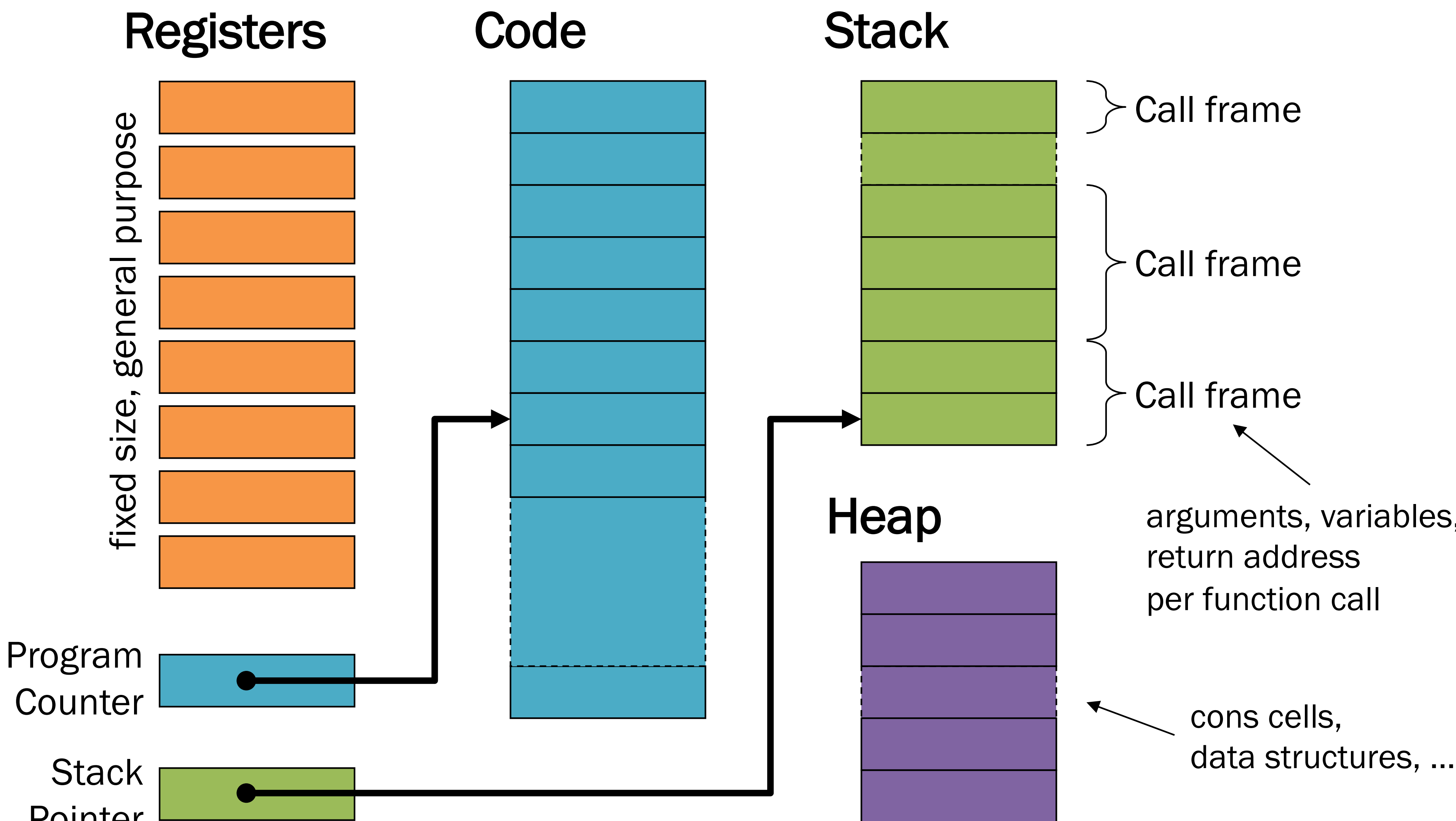
Implementation: memory management

Who cleans up all those cons cells when we're done with them?

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



CS 240-style machine model

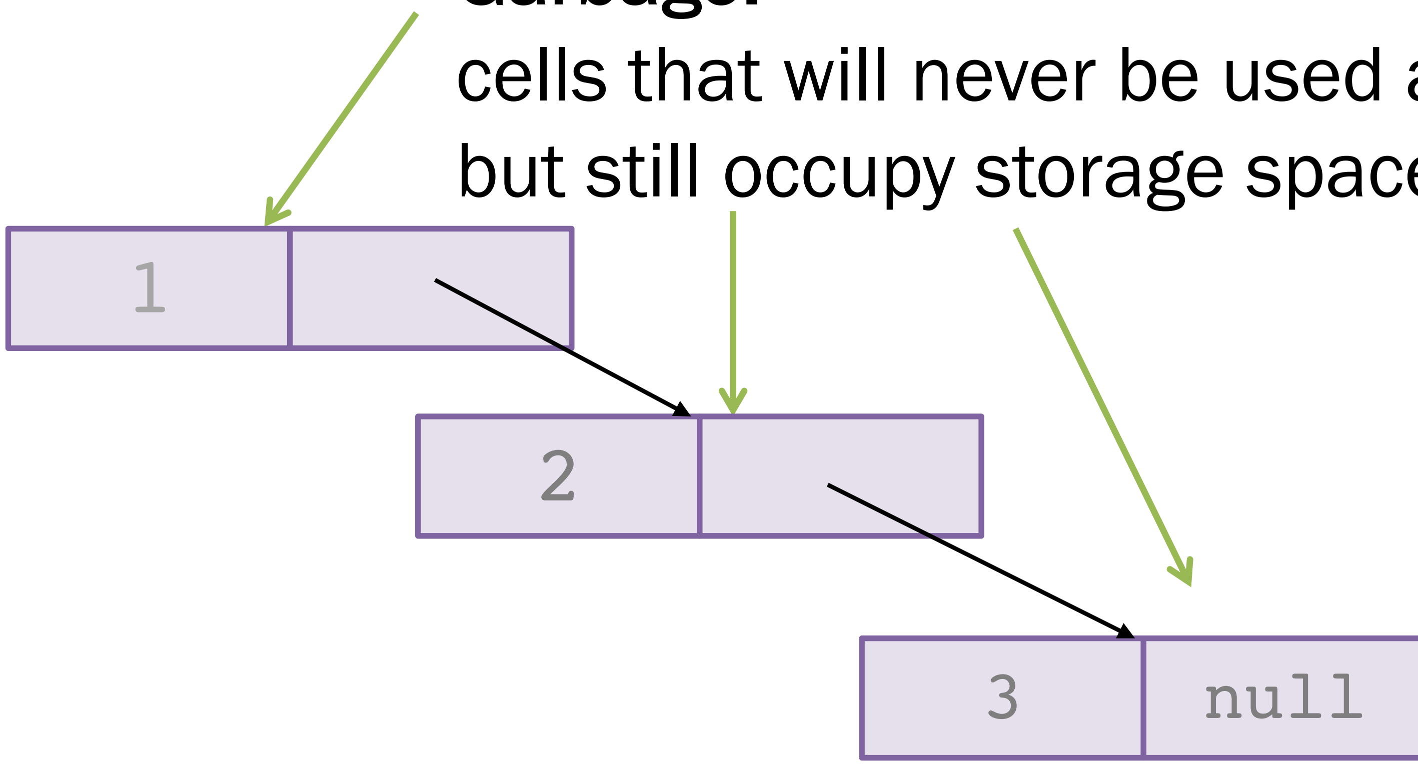


Implementation: memory management

Who cleans up all those cons cells when we're done with them?

`(car (cons 1 (cons 2 (cons 3 null))))` ↓ 1

Garbage:
cells that will never be used again,
but still occupy storage space.



Garbage Collection (GC)

- A cell or object is ***garbage*** once the remainder of evaluation will never access it.

- **Garbage collection:**
  Reclaim space used by garbage.

- Required/invented to implement Lisp.
  - Immutability ⇒ fresh copies
  - Rapid allocation, rapid garbage creation

GC: Reachability

Goal: Reclaim storage used for *all* garbage cells.

Reality?
```
(let ([garbage (list 1 2 3)])
  (if e (length garbage) 0))
```

Achievable goal: Reclaim storage used for all *unreachable* cells.

- All unreachable cells are garbage.
- Some garbage cells are reachable.

A cell is reachable if it is:

roots
- a subexpression of the expression currently being evaluated; or
- bound in the current environment*; or

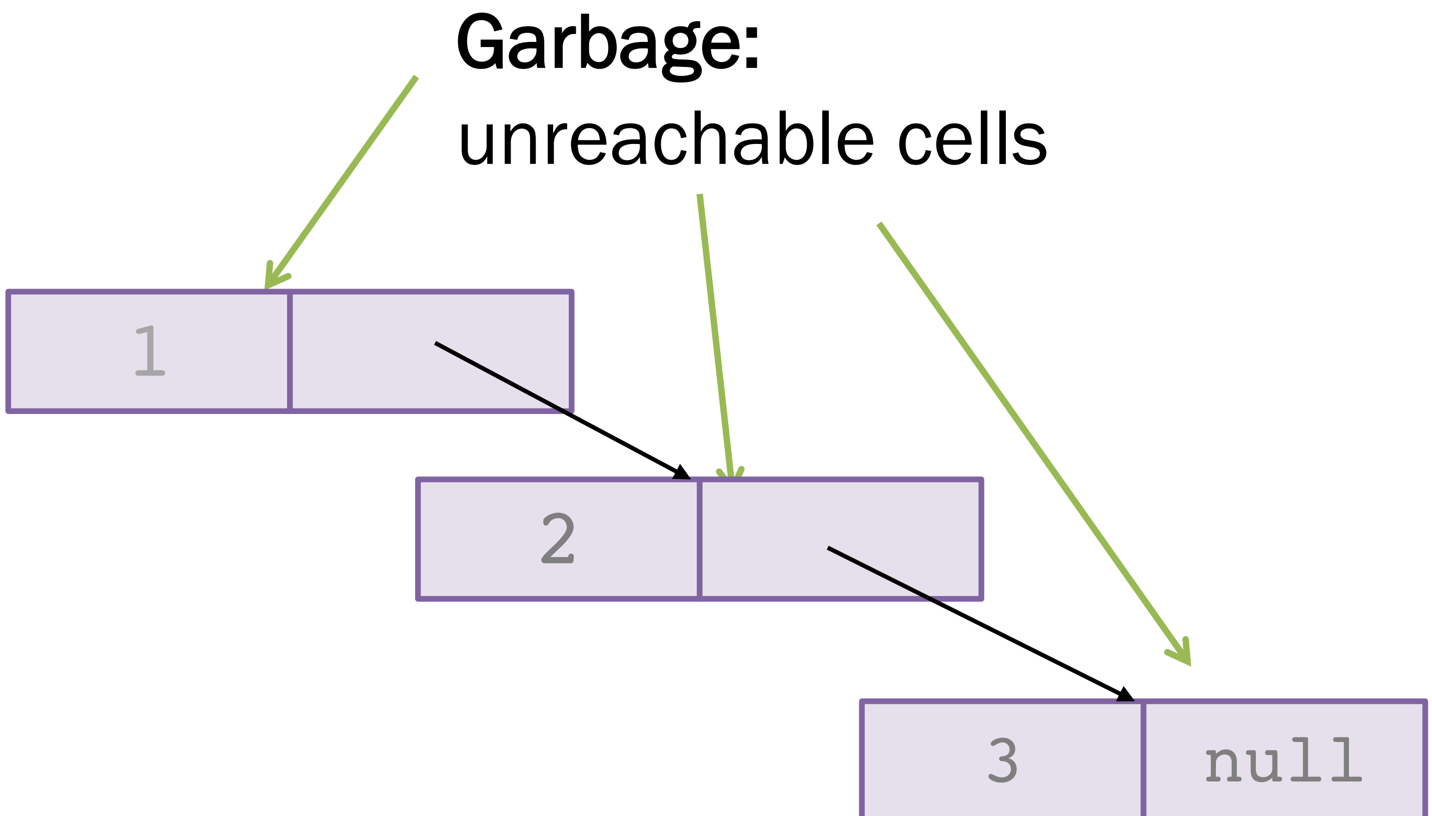
recursive heap cases
- bound in the environment of any reachable closure; or
- the referent of the *car* or *cdr* of any reachable cons cell.

*roughly

GC: Reachability

Who cleans up all those cons cells when we're done with them?

`(car (cons 1 (cons 2 (cons 3 null))))` ↓ 1



You will read more about GC on the next assignment.