CS 61A SU26

Lecture 19: Scheme Lists

July 23, 2026

Rebecca Dang

Join pollev.com/rebeccadang831 (or scan QR code) on your phone or laptop

We'll begin at Berkeley time (10 minutes after), as per tradition!

Potpourri

5 min

- Announcements
- Computing in the News

Announcements

- Quiz 3 grades later today
- Quiz 4 logistics released

Computing in the News

- Harry Potter publisher to receive millions in Anthropic copyright settlement (The Guardian)
  - "**The publisher of Harry Potter has received a multimillion-pound payout** as a beneficiary of a $1.5bn (£1.12bn) **copyright settlement between the AI startup Anthropic** and **thousands of authors** over the use of their protected work to power chatbots."
  - "**Bloomsbury**, which is home to the bestselling novelists **Sarah J Maas** and **Susanna Clarke** as well as **JK Rowling**, said it had **14,087 titles** listed within the settlement, with a proposed compensation of about **$3,000 a title**."

Review: Scheme

5 min

- `/` vs. `quotient`
- `let` and `if`: WWSD

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Select all of the following Scheme expressions which evaluate to 2.

/ vs. quotient

`(/ <dividend> [divisor] ...)`

- **Takes 1 or more arguments**
  - If only the dividend is provided, evaluates to 1 / dividend
  - Otherwise, repeatedly divide
- Performs **true division**
- **Result evaluates to a float only if absolutely necessary** (see previous slide)
  - Ex: `(/ 7 2)` evaluates to 3.5 but `(/ 6.0 2.0)` evaluates to 3

`(quotient <dividend> <divisor>)`

- **Takes exactly 2 arguments**
- Performs **floor division**
- **Evaluates to integer only if both the dividend and divisor are integers**
  - Otherwise, evaluates to float

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Let and If: What would Scheme display?

Scheme Lists

20 min

- Pairs and Lists
- History of `car` and `cdr`
- List Manipulation Basics
- Append
- Map
- Filter
- Reduce

Pairs and Lists

- **Pairs** are a built-in data structure consisting of two fields, a `car` and a `cdr` (like `first` and `rest` in the `Link` class).
  - The first value can contain any data type
  - However, the second value must contain `nil` or a pair
- A **list** is defined as either `nil` or a pair whose `cdr` is another list.

[out of scope] History of `car` and `cdr`

"The **mapping between a higher-level abstraction and the hardware** implementing it gave rise to the use of the names CAR and CDR in LISP... **Clearly, these names are not at all indicative of their functions.** The first LISP implementation was on an IBM 704, a 36 bit computer whose instruction word was divided into four subfields, with a special instruction to access the contents of each field. The "address" and "decrement" fields, each 15 bits, were used to hold the two pointers that made up a list cell, and the LISP operations CAR and CDR were named for the 704 operations that implemented them: **"Contents of Address field to Register"** and **"Contents of Decrement field to Register"**.

Source: The Evolution of Abstraction in Programming Languages (Guarino 1978)

Want to learn what a "register" is? Take CS 61C!

List Manipulation Basics (1 of 4)

```
scm> (cons 1)
Error
scm> (cons 1 nil)
(1)
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
scm> (cons (cons 1 nil) (cons 2 nil))
((1) 2)
scm> (list 1 2 3)
(1 2 3)
```

List Manipulation Basics (2 of 4)

```
(define a (cons 1 (cons 2 (cons 3 nil))))
(define b (cons (cons 1 nil) (cons 2 nil)))
scm> a
(1 2 3)
scm> (car a)
1
scm> (cdr a)
(2 3)
scm> (car (cdr a))
2
scm> (car (cdr (cdr a)))
3
scm> (cdr (car b))
()
```

List Manipulation Basics (3 of 4)

```
scm> (length (list 1 2 3))
3
scm> (length nil)
0
scm> (length ())
0
scm> (length (cons (cons 1 nil) (cons 2 nil)))
2
```

List Manipulation Basics (4 of 4)

```
scm> (list? nil)
#t
scm> (list? (cons 1 nil))
#t
scm> (pair? nil)
#f
scm> (pair? (cons 1 nil))
#t
scm> (null? nil)
#t
scm> (null? (cons 1 nil))
#f
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Which of the following diagrams represents the following Scheme list?
 (cons 1 (cons (cons 2 (cons (cons 3 nil) nil)) (cons 4 (cons 5 nil))))

Append

```
scm> (append)
()
scm> (append (list 1 2 3) (list 4 5))
(1 2 3 4 5)
scm> (append (list 1 2) (list 3 4) (list 5 6))
(1 2 3 4 5 6)
```

Caution: The behavior of `append` in Scheme is more analogous to `extend` in Python!

Map

```
(map <proc> <lst>)
```

Returns a list constructed by calling `proc` (a one-argument procedure) on each item in `lst`.

```
scm> (map (lambda (x) (* x x)) (list 1 2 3))
(1 4 9)
```

Filter

```
(filter <pred> <lst>)
```

Returns a list consisting of only the elements of `lst` that return true when called on `pred` (a one-argument procedure, aka the predicate).

```
scm> (filter even? (list 1 2 3 4 5))
(2 4)
scm> (filter (lambda (x) x) (list 1 "hi" nil 0 #t #f))
(1 "hi" #t)
```

Reduce

```
(reduce <combiner> <lst>)
```

- Returns the result of sequentially combining each element in `lst` using `combiner` (a two-argument procedure).
- `reduce` works from left-to-right, with the existing combined value passed as the first argument and the new value as the second argument.
- `lst` must contain at least one item.

```
scm> (reduce + (list 1 2 3 4))
10
scm> (reduce + (list 1))
1
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Map/Filter/Reduce: What would Scheme display?

Quotes

20 min

- Quote
- Quasiquote and Unquote
- Apply
- Eval
- Equality

Quote (Special Form)

`(quote <expression>)` or `'<expression>`

Returns the literal `expression` without evaluating it.

```
scm> (quote (1 2 3))
(1 2 3)
scm> '(1 2 3)
(1 2 3)
scm> (list? '(1 2 3))
#t
scm> 'a
a
scm> (symbol? 'a)
#t
```

Quasiquote and Unquote (Special Forms)

`(quasiquote <expression>)` or `` `<expression> ``

Returns the literal `expression` without evaluating it, unless a subexpression of expression is of the form:

`(unquote <expr2>)` or `,<expr2>`

```
scm> (define a (list 4 5 6))
a
scm> (quasiquote (1 2 3 ,a))
(1 2 3 (4 5 6))
scm> `(1 2 3 ,a)
(1 2 3 (4 5 6))
```

Apply

```
(apply <procedure> <args>)
```

Calls `procedure` with the given **list** of `args`.

More on this when we talk about macros!

```
scm> (apply + (list 1 2 3))
6
scm> (apply + '(1 2 3))
6
```

Eval

```
(eval <expression>)
```

Evaluates `expression` in the current environment.

```
scm> (eval '(cons 1 (cons 2 nil)))
(1 2)
```

Equality

	`=`	`eq?`	`equal?`
Arguments	Exactly 2 numbers	Exactly 2 values	Exactly 2 values
Semantics	Returns #t if they are equivalent.	For numbers, booleans, or symbols, returns #t if they are equivalent. Otherwise, returns #t if they refer to the same object in memory (e.g. checks for identity).	Returns #t if they are equivalent (e.g. 2 lists are equivalent if their cars and cdrs are equivalent) (e.g. checks for value equality).

Examples	`scm> (= "hi" "bye")`
	`Error`
	`scm> (= 5 5)`
	`#t`
		`(define a (list 1 2 3))`
		`(define b (list 1 2 3))`
		`scm> (eq? a b)`
		`#f`
		`scm> (eq? "hi" "bye")`
		`#f`
		`scm> (eq? 5 5)`
		`#t`
			`(define a (list 1 2 3))`
			`(define b (list 1 2 3))`
			`scm> (equal? a b)`
			`#t`
			`scm> (equal? "hi" "bye")`
			`#f`
			`scm> (equal? 5 5)`
			`#t`

5 min break

Practice

(reminder to self to use high contrast theme)

30 min

- Reverse
- Mountain

Reverse

- Implement `reverse`
- Download starter code `19.scm` from the course website under Lecture 19
- Run tests by copying and pasting your code into a new file in code.cs61a.org and clicking on the red test tube button on the top right
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

Mountain

- Implement `mountain`
- Download starter code `19.scm` from the course website under Lecture 19
- Run tests by copying and pasting your code into a new file in code.cs61a.org and clicking on the red test tube button on the top right
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?