Programming Abstractions

Week 1: Introduction

Stephen Checkoway

About the course

This is a course about Programming Languages

We will use the language Scheme to discuss, analyze and implement various aspects of programming languages.

Course website: https://checkoway.net/teaching/cs275/2021-fall/
- Contains the syllabus, readings, homeworks, and slides

Office hours in King 231:
- Tuesday 13:30–14:30
- Friday 13:30–14:30

Parts of the course

Scheme and things you can do with it (5 weeks)

Implementing Scheme and other languages (4 weeks)

Advanced issues (delayed evaluation, continuations, etc.) (2 weeks)

~~Logic Programming, and Prolog (maybe) (2 weeks)~~

Who am I?
Professor Stephen Checkoway

- Research:
  - Computer/Embedded systems security
  - Hacking computers in things like cars and planes

- Fun Facts:
  - I enjoy picking locks
  - I have two Oberlin cats, Kirk and Bones
  - I have a *very* hard time recognizing faces

A quick history of Scheme

John McCarthy invented LISP at MIT around 1960 as a language for AI.

LISP grew quickly in both popularity and power. As the language grew more powerful it required more and more of a system's resources. By 1980 5 simultaneous LISP users would bring a moderately powerful PDP-11 to its knees.

Guy Steele developed Scheme at MIT 1975-1980 as a minimalist alternative to LISP.

Scheme is an elegant, efficient subset of LISP. It has some nice properties that we will look at that allow it to be implemented efficiently. For example, most recursions in Scheme turn into loops.

Why Scheme for CS 275?

All LISP-type languages have lists as the main data structure

- Programs are lists
- Data are lists
- Scheme programs can reason about other programs. This makes Scheme useful for thinking about programming languages in general.

Scheme is a different programming paradigm

- Python, Java, C and other languages are imperative languages. Programs in these languages do their work by changing data stored in variables
- Scheme programs can be written as functional programs—they compute by evaluating functions and avoid variable assignments.

Why Scheme for CS 275?

Scheme is very elegant. It is much less verbose than Java, which means it is easier to see what is happening in a Scheme program.

It is fun!

Assessment

Eight homeworks

- Between about 7 and 10 days per homework
- You can work by yourself or in groups of 2
- Each has lots of small, independent parts
- Three free late days to use throughout the semester

Two midterms exams

One final exam (optional)

Class participation

Clickers!

- Lets you vote on multiple choice questions in real time.
- You need one by next Monday

Clicker poll questions

Peer instruction

I'll read the question

Answer the poll individually

Group discussion, come to consensus

Everybody in group votes the consensus

Report your group's vote/thinking

Some question about a concept we just talked about?

A. Distractor answer 1
B. The right answer
C. Distractor answer 2
D. Distractor answer 3
E. None of the above

9

Group Discussion Norms

Make sure everyone gets to talk.

Have everyone state their answer before discussing which answer is correct.

Take turns reporting out.

If you think someone is wrong, ask them to explain their thinking rather than just dismissing it.

Class Norms

Contribute as you feel comfortable

- If you're not comfortable answering, you can pass.
- If you're not usually inclined to speak much in class, push yourself to ask questions more often.

Be aware of the space you take up in class

- Make space for others, use some space for yourself

The main goal of every person in the class should be to engage proactively with the ideas we understand the least. If someone asks a question/makes a comment that seems obvious to you, show them respect.

Scheme interpreter: DrRacket

Racket is Scheme plus extra nice stuff

- One consequence is Scheme has a bunch of traditional names for list functions that are bad, Racket has better names! We'll learn and use both as appropriate

We're actually going to be using Racket in this course

- I'm probably going to use Racket and Scheme interchangeably (sorry)

DrRacket is free https://www.racket-lang.org

Todo this week

Readings from *How to Design Programs*

- Prologue
- Chapter 1 (section 1.4 is optional)
- Sections 2.1–2.4
- Sections 4.1–4.2
- Section 8.1

That's a lot, but this is an introductory text and y'all already know how to program!

Install DrRacket before next class

Do Homework 1

- Due Friday, October 15 at 23:59

Introducing Scheme

Expression in Scheme (s-expression)

(Traditional)

A symbolic expression (s-expression) is one of the following

- An atom
  - A number, e.g., `5`, `-10`, `8.3`
  - Boolean values `#t` and `#f`
  - A string, e.g., `"foo"`
  - A symbol, e.g., `'foo`, `'list-ref`, `'pair?`, `'set!`
- Null
  - Written `null` or `'()`
- A pair
  - Written `(x . y)` where `x` and `y` are s-expressions
- A variable, e.g., `foo`, `list-ref`, `pair?`

Expressions in Racket

(Modern)

The concept of an atom isn't as meaningful now

Racket adds additional data types that aren't pairs, aren't null, and aren't really atoms (like vectors)

For the most part, we're going to ignore these in this course

Arithmetic/logical/string operations

3 + 5: `(+ 3 5)`

x * (4 + y + z): `(* x (+ 4 y z))`

x and y: `(and x y)`

x or y or z: `(or x y z)`

"hello" + " " + "world": `(string-append "hello" " " "world")`

In C, Python, or Java, we would compute the arithmetic mean (average) of two numbers (or variables holding numbers) as `(x + y) / 2`. How do we do this in Scheme or Racket?

A. `(x + y) / 2`

B. `((x + y) / 2)`

C. `(+ x y / 2)`

D. `(+ (/ x y) 2)`

E. `(/ (+ x y) 2)`

Lists

Lists are the most important data type in Scheme

A list is one of two things

- `null`, the empty list
- A pair `(x . y)` where `x` is an s-expression and `y` is a list
  - `x` is called the head of the list and `y` is the tail

This is a recursive type definition: a type defined in terms of itself!

Special syntax for lists

`'(42 -8 #t + "foo")` is a list of 5 atoms ('+ is a symbol)

It's equivalent to
`'(42 . (-8 . (#t . (+ . ("foo" . ())))))`

Lists are heterogeneous (they can contain elements of different types)

The empty list

There are three ways to write the empty list, they're equivalent

- `null`
- `empty`
- `'()` — We'll see shortly why this has a leading ' like a symbol does

All of these are simply a null pointer

We can use them mostly interchangeably, but when working with lists (as opposed to some other data type we might build out of pairs), using `empty` or `'()` can make it clear you mean the empty list specifically

Creating a list

`(list)` produces the empty list `'()`

- `null`, `empty`, and `'()` also do this

`(list 1 3 5 2)` produces the list `'(1 3 5 2)`

`(list #t 5 "foo")` produces the list `'(#t 5 "foo")`

`(list (* 2 3) (and #t #f) 8)` produces `'(6 #f 8)`

Quoting

Placing a `'` before an s-expression "quotes" it

- The quoted expression is treated as data, not code
- DrRacket displays lists with the quote

`'(1 4 5)` is a 3-element list

We saw `(list (* 2 3) (and #t #f) 8)` produces `'(6 #f 8)`

`'((* 2 3) (and #t #f) 8)` produces `'((* 2 3) (and #t #f) 8)`

- This is a 3 element list:

```
'((* 2 3)            ; 1st element, itself a 3-element list
  (and #t #f)        ; 2nd element, another 3-element list
  8)                 ; 3rd element, the number 8
```

Quoting

Quoting a number, boolean, or string returns that number, boolean, or string

- `'35` gives `35`
- `'#t` gives `#t`
- `'"Hello!"` gives `"Hello!"`

Quoting a variable gives a symbol

- `+` and `string-append` are variables whose values are procedures
- `'+` and `'string-append` are symbols

Quoting a list gives a list of quoted elements

- `'(1 2 x y)` is the same as `(list 1 2 'x 'y)`
- `'(() (1) (1 2 3))` is the same as `(list '() '(1) '(1 2 3))`

Given variables x and y, how do we create a list containing the values of x, y, and x+y? I.e., if x is 10 and y is 15, the list we create is `'(10 15 25)`.

A. `(list x y (+ x y))`

B. `(list 'x 'y (+ 'x 'y))`

C. `(list 'x 'y '(+ x y))`

D. `'(x y (+ x y))`

E. `All of the above`

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`

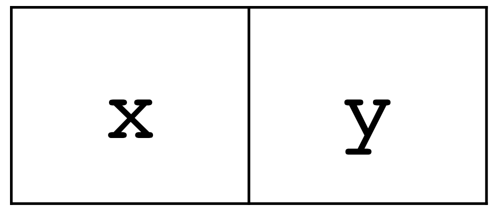
- `(cons 8 (list 1 2 3))` produces the list `'(8 1 2 3)`

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

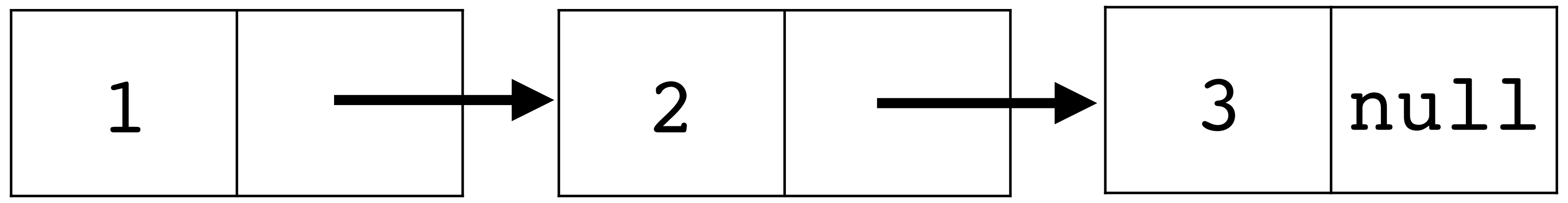
Cons cells

Construct a pair

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



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



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

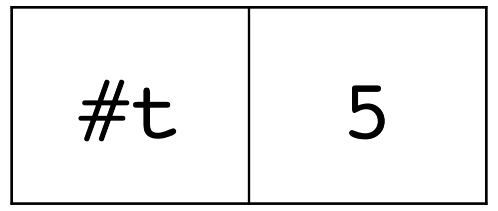
What if we want to append `x` to `lst`? Can we use `(cons lst x)`?

- I.e., will `(cons '(1 2 3) 4)` produce `'(1 2 3 4)`?

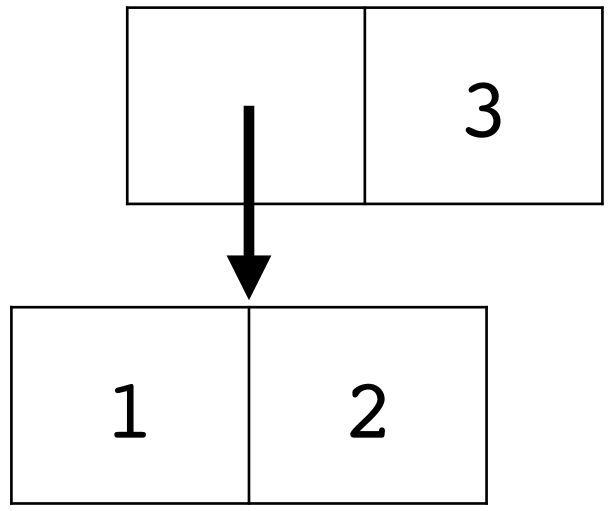
Aside: Trees from pairs

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

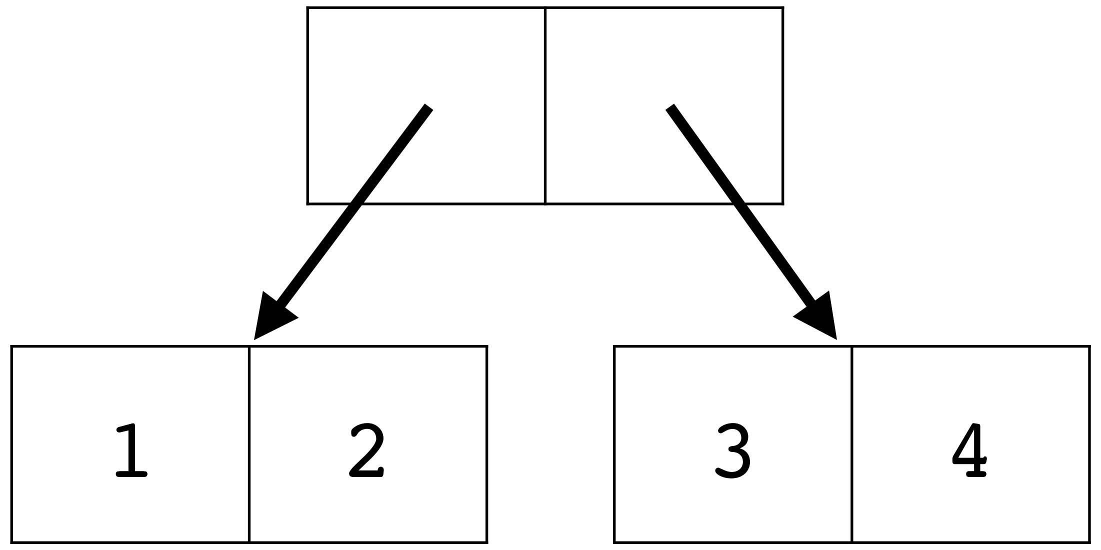
```
(cons #t 5)
```




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




```
(cons (cons 1 2)
      (cons 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)))
```