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/cs275/2020-fall/
‣ Contains the syllabus, readings, homeworks, and slides

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)

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 or nine 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

Class participation

~~Clicker~~ Zoom poll questions

Peer instruction

I'll read the question

Answer the poll individually

Group discussion in a breakout room, 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

Can't see slides in breakout rooms 😠

Slides are posted before class (sometimes changes are made *right before* class)

https://checkoway.net/teaching/cs275/2020-fall/schedule.html

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

Read chapters 1–3 of The Little Schemer
‣ This isn't a great book unfortunately
  – It uses strange names for functions which we're not going to use
  – It's also pretty old

Install DrRacket before next class

Do Homework 1
‣ Due next Tuesday, September 8 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

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!

There's a 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

Expression evaluation

Scheme evaluates s-expressions to produce values

- ‣ The value of `'()` is null
- ‣ The value of a variable is the value bound to it
  - – E.g., the variable `null` is bound to `'()`
- ‣ The value of an atom is the atom itself
- ‣ The value of a non-null list depends on the head of the list
  - – If the head is one of a specific set of symbols (e.g., define, lambda, λ, and let), it's a *special form*. Each special form has its own way of being evaluated
  - – Otherwise, it's procedure application

Procedure evaluation

`(foo 1 2 #t)` applies the procedure bound to the variable `foo` to the arguments 1, 2, and #t

‣ `(+ 1 2 3)` — applies + to 1, 2, and 3, performing addition
‣ `(* 5 (- x y) (/ z 8))` — computes $5(x - y)(z / 8)$
‣ `(list 32 5 8)` — creates the list `(32 5 8)`
‣ `(list-ref (list 32 5 8) 2)` — returns the element of `'(32 5 8)` at index 2 namely 8

Note that `(1 2 3)` is invalid because 1 isn't a special form nor is it a procedure

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)`

Procedure evaluation order

`(s-exp0 s-exp2 ... s-expn)`

Scheme evaluates each of the s-expressions in turn
- `s-exp0` must evaluate to a procedure value (more about this shortly)
- `s-exp1` through `s-expn` are evaluated to produce values
- Then, the procedure is applied to the *n* arguments

`(+ (* 2 3) 8)`
- `+` evaluates to the addition procedure
- `(* 2 3)` is evaluated
  - `*` evaluates to the multiplication procedure
  - `2` and `3` evaluate to themselves
  - multiplication procedure is applied to `2` and `3`, producing `6`
- `8` evaluates to itself
- addition procedure is applied to 6 and 8, producing 14

Using quote ' to prevent evaluation

`(quote (1 2 3))` evaluates to the list `(1 2 3)`

`'(1 2 3)` is shorthand for this and is how DrRacket will display lists

`'()` is null and how DrRacket displays the empty list

We can quote identifiers (e.g., variable names) and use them as symbols rather than the value the identifier is bound to (if any)

▸ `'red`, `'green`, `'blue`, `'+`, `'list`, etc

E.g., `(+ 1 2)` evaluates to `3`, `'(+ 1 2)` evaluates to the list `'(+ 1 2)`

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
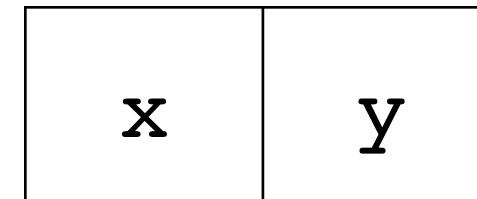
`cons` — (Construct) Create a pair

‣ `(cons x y)` creates the pair `(x . y)`
‣ `(cons 5 null)` creates the list `'(5)`
‣ If `lst` is a list, then `(cons x lst)` returns a new list with head `x` and tail `lst`
‣ `(cons 8 (list 1 2 3))` produces the list `(8 1 2 3)`
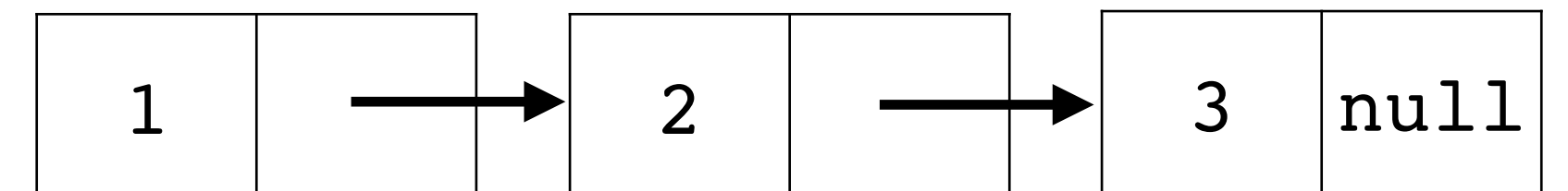
Cons cells
Construct a pair

`(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!

Aside: Trees from pairs
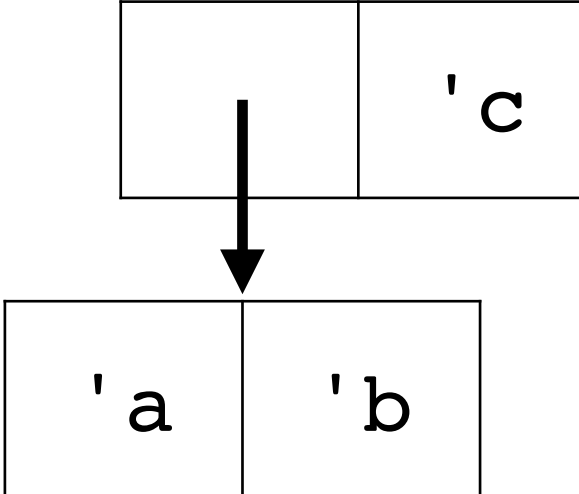
Nothing says our cons-cells need to be used for lists
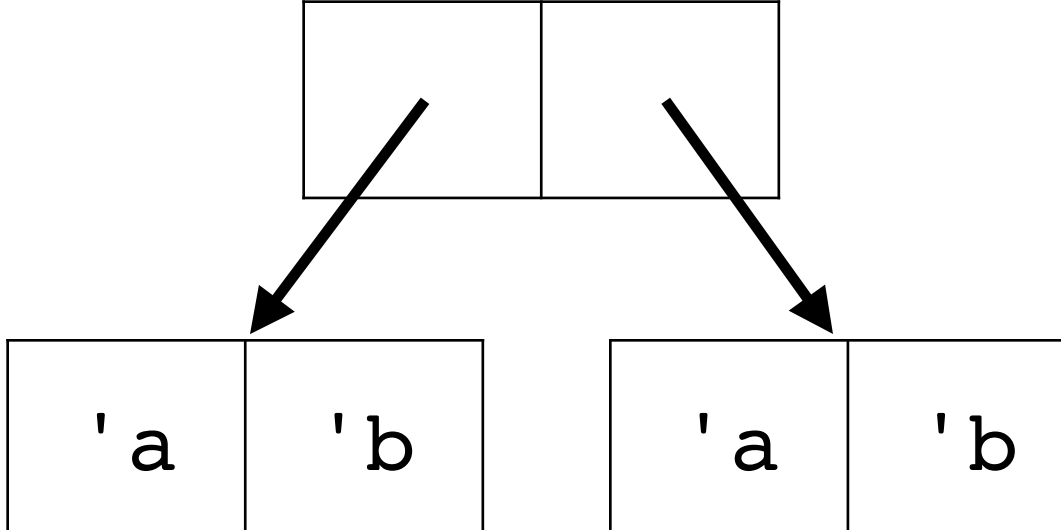
```
(cons 'a 5)
```
'a	5

```
(cons (cons 'a 'b) 'c)
```



```
(cons (cons 'a 'b)
      (cons 'c 'd))
```

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 IMB 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 IMB 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

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 x 5)`
- ‣ `(define y (+ x x x))`
- ‣ `(define cs-professors '("Adam" "Bob" "Cynthia"))`
  `(third cs-professors) => "Cynthia"`

The expression is evaluated so `y` will be bound to the value `15` rather than the expression `(+ x x x)`

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 so common, there's a special syntax for it

‣ `(define (name parameters) body…)`

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

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

We'll see later how closures containing the current environment can be extremely useful

Applying a closure to arguments

```
(define a 10)
(define (foo x)
   (+ x a))
```

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

```
(foo 20)
```

Environment of the closure

a	10

Environment of the call

a	10
x	20

Conditionals

If expression

```
(if test-exp then-exp else-exp)
```

If test-exp evaluates to anything other than #f, then the whole expression evaluates to the evaluation of then-exp

If test-exp evaluates to #f, then the whole if expression evaluates to the evaluation of else-exp

Examples

‣ ```
(if (= x y)
    (+ x 2)
    y)
```
‣ ```
(if (empty? lst)
    "The list is empty"
    "The list is not empty")
```

Conditional expressions

```
(cond [test-exp1 exp1] … [test-expn expn])
```

Evaluates the test-exp expressions in turn. The first one that evaluates to true has its corresponding exp evaluated which becomes the value of the whole expression

We can use else as the last test expression

```
(cond [(zero? x) 0]
      [(> x 0) 1]
      [else -1])
```

Some examples

```
(define (signum x)
  (cond [(zero? x) 0]
        [(> x 0) 1]
        [else -1]))


(define (length lst)
  (cond [(empty? lst) 0]
        [else (+ 1 (length (rest lst)))]))
```

Equality

Scheme's equality operators

`=`, `eq?`, `eqv?`, and `equal?`

`(= a b)` — compares only numbers, cannot be used for anything else

`(equal? a b)` — compares structures recursively
‣ you almost always want this one and not any of the two below

`(eq? a b)` — compares if a and b refers to the same object in memory
‣ This can be used on symbols `(eq? 'foo 'foo)` returns #t, but is otherwise not good
‣ `(eq? 2.0 (+ 1.0 1.0))` can (and in DrRacket does) return #f!

`(eqv? a b)` — like `eq?` but also works with characters and numbers

Testing for equality

Moral

Are you dealing with numbers? Use =

Are you dealing with anything else? Use equal?

(You can use `eq?` or `eqv?` with symbols like `(eq? sym 'foo)` to determine if `sym` is the symbol `'foo`)

Let's write some Racket!

`(reverse lst)` — reverse the list lst
‣ `(reverse '(1 2 (3 4) 5)) => '(5 (3 4) 2 1)`

`(remove-numbers lst)` — Remove all of the numbers from lst
‣ `(remove-numbers '(foo 3 5 (6 8))) => '(foo (6 8))`

`(filter pred lst)` — Takes a predicate and a list and returns a list consisting only of the elements satisfying the predicate
‣ `(filter (λ (x) (> x 0)) '(1 2 -8 3 0 -4)) => '(1 2 3)`

Let's write some Racket!

`(filter-not pred lst)` — Like `filter`, but only keeps the elements not matching the predicate

`(filterer pred)` — returns a procedure that takes a list and filters the list by pred
- ‣ (define f (filterer even?))
  (f '(1 2 3 4 5 6)) => '(2 4 6))