Midterm review

(including tutorial material for L08 to L10)

Scope of the midterm: Everything up to L10

- Numbers, Boolean values, and symbols
- Translating mathematical expressions and functions into Racket
- Recursion on natural numbers
  - Remember to test for base cases with `(zero? …)` and recur with `(sub1 …)`.
- Accessing elements of fixed length lists with `first`, `rest`, and `cons`
- Recursion on lists, functions that produce and consume lists, predicates on lists
  - Remember to test for base cases with `(empty? …)` and recur with `(rest …)`.
- Ordered lists, sorting lists
- Recursion on two lists, or on a list and a natural number
- Measuring efficiency using the stepper
- O(1) “constant” vs. $O(n)$ “linear” vs. $O(n^2)$ “quadratic” vs. $O(2^n)$ “exponential”.
- The final version of the “Rules of Recursion”

Structure of the midterm

- There are 10 questions, each worth 10%
- Later questions are harder than earlier questions, so leave time.
- You may use list abbreviations unless otherwise indicated.
- You may define helper functions unless otherwise indicated.
- Your answers can depend on any assumptions stated in the question.

Exclusions

- You need to understand any purpose or contract we provide on the exam, but we will not ask you to write a purpose or contract.
- You need to understand any data definition we provide on the exam, but we will not ask you to write a data definition.
- You should understand all of the terminology we routinely use in class (“produce”, “consume”, “predicate”, “recursion”, “accumulator”, etc.) but we will not ask you to write definitions or answer question in English. Questions will be multiple choice, true/false, or answerable in Racket.
- You will not need to use an accumulator unless we explicitly suggest it. Unless we explicitly suggest it, using an accumulator is probably a bad idea.
- The necessary material on efficiency is summarized in this review. We will not ask you to write or understand proofs.

We do not answer questions during exams

- If you believe there is an error in the exam, notify a proctor. An announcement will be made if a significant error is found.
- It is your responsibility to properly interpret a question.
- If there is a non-technical term you do not understand, you may ask for a definition.
- Otherwise, state your assumptions and proceed to the best of your ability.

Allowed constructs (to the end of lecture L10)

```
( ) [ ] + - * / = < > <= >= ;

abs acos add1 and append asin atan check-expect check-within
cond cons cons? cos define e else empty empty? exp expt
false first inexact? integer? length list list? log max min
not number? or pi quotient rational? remainder rest reverse
second sin sqr sqrt sub1 symbol? symbol=? tan third true
zero?

listof Any anyof Bool Int Nat Num Rat Sym
```

Example: Are two lists “the same”?

Write a predicate to compare two lists containing numbers, symbols and/or Boolean values and determine if they are “the same”.

```
(check-expect
  (same-list? (list 1 true 'hello) (list 1 true 'hello))
  true)
(check-expect
  (same-list? (list 1 2 3) (list 'hello 'world))
  false)
(check-expect (same-list? (list true false) empty) false)
(check-expect (same-list? empty empty) true)
```

To compare two lists, we have to compare elements

```
;; Predicate to compare any two values (Num, Bool or Sym)
;; and determine if they are the same value.
;; same?: Any -> Bool
(define (same? x y)...)

(check-expect (same? 1 3) false)
(check-expect (same? 3 3) true)
(check-expect (same? false 'hello) false)
(check-expect (same? 'hello 'hello) true)
(check-expect (same? 'hello 'world) false)
(check-expect (same? true true) true)
```

Break down the cases…

```
;; Predicate to compare any two values (Num, Bool, or Sym)
;; and determine if they are the same value.
;; same?: Any -> Bool
(define (same? x y)...)
```

1. `x` and `y` are both numbers and their values are the same.
2. `x` and `y` are both symbols and the symbols are the same.
3. `x` and `y` are both Boolean values and their values are the same.

We assume that `Num`, `Bool`, or `Sym` are the only possibilities.

Another problem is that `boolean?` is not on the list of allowed constructs 😞

Break down the cases…

```
;; Predicate to compare two values (Num, Bool, or Sym)
;; and determine if they are the same value.
;; same?: Any -> Bool
(define (same? x y)
  (cond [(number? x) ...]
        [(symbol? x) ...]
        [else ...])) ;; x is a Bool
```

Once we know the “type” of **`x`** (i.e., what kind of value it is) we can check the type of **`y`** and then compare values.

Complete the conditions…

```
;; Predicate to compare two values (Num, Bool, or Sym)
;; and determine if they are the same value.
;; same?: Any -> Bool
(define (same? x y)
  (cond [(number? x) (and (number? y) (= x y))]
        [(symbol? x) (and (symbol? y) (symbol=? x y))]
        [else (and (not (number? y))
                   (not (symbol? y))
                   (or (and x y) (not (or x y))))]))
```

How do we extend **`same?`** to **`same-list?`**

Reminder from class: Lockstep recursion on two lists.

There are two base cases and recursion is on both lists.

```
;; compute the dot product of two vectors
;; dot-product: (listof Num) (listof Num) -> Num
(define (dot-product lst1 lst2)
  (cond [(empty? lst1) 0]
        [(empty? lst2) 0]
        [else (+ (* (first lst1) (first lst2))
                 (dot-product (rest lst1) (rest lst2)))]))
```

We can follow the same approach for **`same-list?`**

We have two base cases.

```
;; Predicate to compare two lists of Num, Bool, or Sym
;; and determine if they are the same.
;; same-list?: (listof Any) (listof Any) -> Num
(define (same-list? lst1 lst2)
  (cond [(empty? lst1) ...]
        [(empty? lst2) ...]
        [...]
        [else ...]))
```

If both lists are not empty, then what?

```
;; Predicate to compare two lists of Num, Bool, or Sym
;; and determine if they are the same.
;; dot-product: (listof Any) (listof Any) -> Num
(define (same-list? lst1 lst2)
  (cond [(empty? lst1) (empty? lst2)]
        [(empty? lst2) false]
        [...]
        [else ...]))
```

We have two cases: 1) the first elements are the same, and 2) they are different.

If both lists are not empty, then what?

```
;; Predicate to compare two lists of Num, Bool, or Sym
;; and determine if they are the same.
;; dot-product: (listof Any) (listof Any) -> Num
(define (same-list? lst1 lst2)
  (cond [(empty? lst1) (empty? lst2)]
        [(empty? lst2) false]
        [(not (same? (first lst1) (first lst2))) false]
        [else (same-list? (rest lst1) (rest lst2))]))
```

If they are not the same, we can “short circuit”.

Efficiency

Let's consider efficiency when lists are the same, with length *n*:

`(same-list? empty empty)` ⇒ 5 steps

`(same-list? (list 1) (list 1))` ⇒ 23 steps
`(same-list? (list 'hello) (list 'hello))` ⇒ 25 steps
`(same-list? (list false) (list false))` ⇒ 32 steps

If we examine `same?` we see that the "worst case" is when all values are `false`.

`(same-list? (list false false) (list false false))` ⇒ 59 steps
`(same-list? (list false false false)`
`(list false false false))` ⇒ 86 steps

steps = $f(n) = 5 + 27n = O(n)$, i.e., "linear time"

Efficiency

What about if the lists are different at some point?

Since the function “short circuits” when a difference is encountered, the worst case for a list of length *n* is when the elements are the same.

```
(same-list? (list false false false true 'hello 'world)
            (list false false 4 5 6 7 8 9 10))
```
⇒ 73 steps

Even if both lists have hundreds or millions of elements, as soon as they are different the function produces **`false`**.

For problems like this, in the mid-term we will give you the number of steps for different values *n*, from which you can derive a function.

All you need to know about efficiency for the mid-term

1. Most built-in functions and functions that don't use recursion are $O(1)$, i.e., "constant time".
2. Efficiency is measured by counting substitution steps. The number of steps will be expressed in terms of a natural number, such as the length of a list, n.
3. If the number of steps is of the form $a + bn$, efficiency is $O(n)$ or "linear".
4. While they only take one step in the stepper, we want you to think of some built-in functions as linear: **`length`**, **`append`**, and **`reverse`**
5. If the number of steps is of the form $a + bn + cn^2$, efficiency is $O(n^2)$ or "quadratic".
6. The number of steps can be "exponential" in n, e.g., $O(2^n)$. Large n will cause "exponential blowup". The function will be unusably slow for larger n.

Example: Largest value in a list

In class we saw a linear time function to determine the largest value in a list.

```
;; produce the largest of a non-empty list of numbers
;; largest: (listof Num) -> Num
(define (largest lst)
  (cond [(empty? (rest lst)) (first lst)]
        [else (larger (first lst) (largest (rest lst)))]))
```

This version has been slightly simplified from the one in class by using the built-in function **`max`**.

To provide a simple example, let's think about writing **`largest`** with an accumulator.

How would a person find the largest value in a list?

Think about the list: 39, 78, 54, 98, 504, 357, 114. What's the largest value?

Perhaps you scanned the list remembering the "largest value see so far". When you saw a value that's larger than the "largest value seen so far", you remembered the new value – until you saw one that is still larger. When you reached the end of the list, the "largest value seen so far" is the largest value in the list.

Computationally, we can pass down the "largest value seen so far" as accumulator. This parameter accumulates the result of prior computation, and is used to compute the final answer that is produced in the base case.

Example: Largest value in a list with an accumulator

```
;; produce the largest of a list of numbers
;; that is larger than the largest seen so far.
;; largest/acc: Num (listof Num) -> Num
(define (largest/acc largest-seen lst)
  (cond [(empty? lst) largest-seen]
        [(> (first lst) largest-seen)
         (largest/acc (first lst) (rest lst))]
        [else (largest/acc largest-seen (rest lst))]))
(check-expect (largest/acc 100 (list 1 3 2)) 100)
(check-expect (largest/acc 0 (list 1 3 2)) 3)
(check-expect (largest/acc 999 empty) 999)
```

Example: Largest value in a list with an accumulator

```
;; produce the largest of a non-empty list of numbers
;; largest: (listof Num) -> Num
(define (largest lst)
  (largest/acc (first lst) (rest lst)))
(check-expect (largest (list 39 78 54 98 504 357 114)) 504)
```

The “largest seen so far” can be `(first lst)` because the list is assumed to be non-empty.

This function (including the helper function `largest/acc`) is $O(n)$.