Fall CS 145 Final Review Guide.

֍ Prof: Gordon Cormack
֍ ISA: Ashish Mahto
֍ David Duan - j32duan@edu.uwaterloo.ca

==

Release Note

V0.X

Notes from lecture, tutorial, etc.

V1.0

Dec.1 21:15

Topic. 1: Introduction.

Subtopic. ★☆ 1.1 Creating your own data structure.

Created the entire subtopic.

V1.1

Dec.2 15:45

Topic. 1: Introduction.

Subtopic. ★ 1.X Other useful stuff

Created section of let, let* statement.

Topic. 8: Gen, RAM, CPU.

Subtopic. ★★★☆ 8.1: Trie

Created the entire subtopic.

Subtopic. ★★★☆ 8.2: RAM

Created the entire subtopic.

==

Remark.

1. For each topic there is a "level of difficulty" (which is very subjective) using symbols ★ and ☆; each full star represents numerical value 1.0 and each empty star represents numerical value 0.5. Max difficulty is set to 5.0 (which is ★★★★★) and min difficulty is set to 0.5 (which is ☆).

2. Right now this is still somewhat a draft, so I can't really see the big picture and thus won't be able to provide an "outline" sort of thing. The release version (the last version, which hopefully would be compiled by the end of Tuesday) will include an outline and an overview (gonna be sketchy for sure lol) of the entire course.

3. I will try to include as much valuable information as possible, but since each of us has different strengths and weaknesses, there probably will be parts where I find extremely difficult but you find it pretty easy, and parts where I think is relatively reasonable but you feel the opposite way. As a result, when you think a part is too easy, simply skip it and move onto the next section; when you find a part challenging and my explanation is not detailed enough, please contact me and I will make sure to write my explanation in a better way (email at line 4).

4. Format/Hierarchy of the study guide:

List of keywords	Meaning
Topic.	
Subtopic.	
#A0.z	Assignment question
Note.	
Definition.	
Theorem.	
Remark.	
Example.	
Implementation.	`<rkt> ;; The actual code </rkt>`

===

Topic. 1: Introduction

Subtopic. ★☆ 1.1 Creating your own data structure.

Note. #A3a

You are to write a program that begins with the following definitions.

```
<rkt>

   (define-struct thing (a b))
   (define u (make-thing 1 2))
   (define v (make-thing
               (make-thing
                 (make-thing 1 2)
                 (make-thing 3 4))
               (make-thing
                 (make-thing 5 6)
                 (make-thing 7 8))))
   (define w (make-thing
               (make-thing
                 (make-thing
                   (make-thing
                     (make-thing
                       (make-thing
                         (make-thing 1 2) 3) 4) 5) 6) 7) 8))

</rkt>
```

Example. Part u.
Define a function sum-thing that consumes a thing that contain two numbers, and produces the sum of those two numbers.

Implementation.
```
<rkt>  (define (sum-thing x) (+ (thing-a x) (thing-b x)))   </rkt>
```

Example. Part v.
Extend your function sum-thing to consume either a number or a thing that contains two numbers, and to produce either the number

or the sum of the two numbers.

Implementation.

```
<rkt>  (define (sum-thing x)
          (if (number? x) x (+ (thing-a x) (thing-b x))))   </rkt>
```

Example. Part w.

Extend your function sum-thing to consume either a number or a thing that contains either numbers of things, where these things also contain either numbers of things, and so on. Your function should produce the sum of all of the numbers contained within its argument.

Implementation.

```
<rkt>  (define (sum-thing x)
          (if (number? x) x
             (+ (sum-thing (thing-a x))
                (sum-thing (thing-b x)))))    </rkt>
```

Remark.

Part u~w should be trivial.

Example. Part x.

Define a function build-thing that consumes two integers m and n, where n is greater than m, and produces a thing with the same form as w above, containing the numbers m through n.

Implementation.

```
<rkt>  (define (build-thing m n)   ;; m > n
          (if (= m (sub1 n))
             (make-thing m n)
             (make-thing (build-thing m (sub1 n)) n)))

       (define (build-thing m n)
          (if (= m n)
             m
             (make-thing (build-thing m (sub1 n) n)))) </rkt>
```

Remark.

Basically two different base case conditions.
The first one says if n - m = 1 then I'll (make-thing m n),
where the second says if m = n then I'll return the number m.
Not that much of a difference.

Example. Part y.

Define a function build-thing-or-number that consumes a positive integer n, and produces either the number 1 or a thing with the same form as v above, containing the numbers 1 through n. Assume that n is a power of 2.

Implementation.

```
<rkt>  (define (build-thing-or-number n)  ;; n = 2^k, integer k >= 0
          (define (helper a b)   ;; b > a
            ;; (printf "a: ~a, b: ~a\n" a b)
            (if (= (add1 a) b)
              (make-thing a b)
              (make-thing  (helper a (quotient (+ a b) 2))
                           (helper (add1 (quotient (+ a b) 2)) b))))
          (if (zero? n) 1 (helper 1 n))                  </rkt>
```

Remark.
Base case: when a and b differ by 1, we do (make-thing a b).
Recursion: when a and b differ by k>1, we compute (a+b)/2,
send the first part from a to (a+b)/2 to an application of helper,
and send the second part from ((a+b)/2 + 1) to b to another
application of helper.

```
<stepper>

;; Example: let n = 8.
~> (build-thing-or-number 8)

;; Evaluate if statement.
=> (if (zero? 8) 1 (helper 1 8))

;; Else clause
=> (helper 1 8)

;; Call helper
=> (if (= (add1 1) 8)
       (make-thing 1 8)
       (make-thing  (helper 1 (quotient (+ 1 8) 2))
                    (helper (add1 (quotient (+ 1 8))) 8)))

;; Else clause
=> (make-thing
     (helper 1 4)
     (helper 5 8))

;; Have some faith in recursion...
=> (make-thing
     (make-thing
       (helper 1 2)
       (helper 3 4)
     (make-thing
       (helper 5 6)
       (helper 7 8))))

;; Finally...
=> (make-thing
     (make-thing
       (make-thing 1 2)
       (make-thing 3 4)
     (make-thing
       (make-thing 5 6)
       (make-thing 7 8))))
```

```
    ;; Done
    </stepper>
```

Subtopic. ★ 1.X Other useful stuff

Note. Local binding: let and let* statement

Sometimes it's better to give names and create "variables" inside a massive
function, making it both easier to read and debug. I have to say this thing
really saved my ass in A12c. Ugh disgusting... Since let* is an advanced
version of let I'll only talk about let*. If you want to read more about
local binding, check out the link below.

Syntax:
```
<rkt>
  (let*{[<var-name1> <val>]
        [<var-name2> <val>]
        ...}
       <expr>)
</rkt>
```

Note.
var-name: name of variable.
val: the value you want to store in var-name.
<expr> on the last line:
This is the value for the entire let* statement.

Example.
Say, I want to create a ram which has value 100 at address 1, 200 at
address 2, and 300 at address 3, this is how I would do it:

```
<rkt>
  ;; (ram-store r addr val)
  ;; store value <val> at address <addr> inside ram <r>.

  ;; Suppose ram is an unbounded empty ram.
  (define my-ram
    (let* {[r0 (ram-store ram 1 100)]
           [r1 (ram-store r0  2 200)]
           [r2 (ram-store r1  3 300)]}
          r2))

  ;; Note that I can use r0 inside my definition for r1.
  ;; This is the difference between let and let*.
  ;; This function will return r2, which is a ram that has [1] = 100, [2] = 200,
  ;; and [3] = 300.
</rkt>
```

Read more: https://docs.racket-lang.org/reference/let.html

==
==
Topic. 8: Gen, RAM, CPU.

Remark.
Honestly, I have no clue what we are supposed to know about this part...
I did section 8.1 and 8.2 when I was solving A12, so the format of notes
might look a bit different from the rest of the file.

Subtopic. ★★★☆ 8.1 Trie

Source: https://www.student.cs.uwaterloo.ca/~cs145/handouts/RAM.rkt

Remark.
We use tries to hold data for us.
Similar to tree ADT, a trie has trie-left, trie-right, and trie-val.

Implementation.
```
<rkt>

(define-struct trie (left right val) #:transparent)
(define (t-left t) (if (empty? t) empty (trie-left t)))
(define (t-right t) (if (empty? t) empty (trie-right t)))
(define (t-val t) (if (empty? t) 'undefined (trie-val t)))

</rkt>
```

Remark.
We also need to implement trie-fetch and trie-store.

Implementation. trie-store
```
<rkt>
;; tr: the trie we are working with.
;; addr: the address where we want to store the "bit".
;; bit: the number we want to store.

;; Picture 1.1: addr = 0
;;                  The original trie, call it "tr"
;;                /                                  \
;;           (t-left tr)                          (t-right tr)
;;         /              \
;;    (t-left              (t-left
;;     (t-left tr))         (t-right tr))

;; We kept the t-val and t-right for tr, and built a new left subtrie using
;; the original (t-left (t-left tr)) and (t-right (t-left tr)), but with
;; a different value (argument "bit").

;; Picture 1.2: addr = 1
;; The graph is symmetrical to the one above.
;; We kept the t-val and t-left for tr, and built a new right subtrie using
;; the original (t-left (t-right tr)) and (t-left (t-right tr)), but with
;; a different value (argument "bit").

(define (trie-store tr addr bit)
  (cond
    [(zero? addr)                   ;; If we want to store something at address 0,
      (make-trie
        (make-trie                        ;; We create a new left branch for the trie;
          (t-left (t-left tr))            ;; its children remain the same but its value is modified.
          (t-right (t-left tr))    ;; Essentially we destructively create a left child.
          bit)
        (t-right tr)                 ;; We preserve the current trie-right and trie-val.
        (t-val tr))]
     [(= 1 addr)                          ;; If we want to store something at address 1,
      (make-trie
        (t-left tr)                       ;; We preserve the current trie-left and trie-val.
        (make-trie                        ;; We create a new right branch for the trie;
          (t-left (t-right tr))           ;; its children remain the same but its value is modifed.
          (t-right (t-right tr))   ;; Essentially we destructively create a right child.
          bit)
        (t-val tr))]
     [(even? addr)                        ;; Now if we want to store something at an even address,
      (make-trie                          ;; we recursively go down to the left side until the we reach addr = 1.
        (trie-store (t-left tr) (quotient addr 2) bit)
        (t-right tr)
        (t-val tr))]
     [true
        (make-trie                        ;; If we want to store something at an odd address,
          (t-left tr)                     ;; we recursively go down to the right side until we reach addr = 1.
          (trie-store (t-right tr) (quotient addr 2) bit)
          (t-val tr))]))

</rkt>
```

Example.

1. Store 1 at address 0

Remark.

Call this trie "trie1".

```
  <stepper>

    ~> (trie-store empty 0 1)

    => (make-trie
          (make-trie
            (t-left (t-left empty))     ;; empty
            (t-right (t-left empty))    ;; also empty
            1)                          ;; value we want to store
          (t-right empty)               ;; empty
          (t-val empty))                ;; 'undefined



                    empty [val='undefined]
               /                              \
              1                              empty
         /       \
      empty     empty

  </stepper>
```

Example.
2. Store 3 at address 1

Remark.
Call the trie "trie2".

```
<stepper>

   ~> (trie-store trie1 1 3)

   => (make-trie
        (t-left trie1
        (make-trie
          (t-left (t-right trie1))
          (t-right (t-right trie1))
          3)
        (t-val trie1)))

                  empty [val='undefined]
             /                              \
            1                                 3
                                         /        \
                                     empty        empty

</stepper>
```

Example.
3. Store 2048 at address 4

Remark.
Call this trie "trie3".

```
<stepper>

   ~> (trie-store trie2 4 2048)

   => [(even? 4)
       (make-trie
        (trie-store (t-left trie2) (quotient 4 2) 2048)
        (t-right trie2)
        (t-val trie2))]

   => (make-trie
        (trie-store (t-left trie2) 2 2048) ;; recursive call [▲]
        (t-right trie2)
        (t-val trie2))

   ;; note that we are dealing with (t-left trie2) now, instead of trie2 itself.
   => [(even? 2)
       (make-trie
        (trie-store (t-left (t-left trie2)) (quotient 2 2) 2048)
        (t-right (t-left trie2))
        (t-val (t-left trie2)))]
```

```
  ;; This step is what we get from the recursive call at line [▲]. After this step we want to
  ;; go back to the original call, so we substitute this result for line [▲].
  => (make-trie
       (trie-store (t-left (t-left trie2)) 1 2048)
       (t-right (t-left trie2))
       (t-val (t-left trie2)))

  => (make-trie
       (make-trie
         (trie-store (t-left (t-left trie2)) 1 2048)    ;; Now we expand this step as in Example. 2
         (t-right (t-left trie2))
         (t-val (t-left trie2)))
       (t-right trie2)
       (t-val trie2))

  => (make-trie
       (make-trie
         (make-trie
           (t-left (t-left (t-left trie2)))
           (make-trie
             (t-left (t-right (t-left trie2))
             (t-right (t-right (t-left trie2))
             2048)
         (t-right (t-left trie2))
         (t-val (t-left trie2)))
       (t-right trie2)
       (t-val trie2))


                     trie3
                 /              \
                1                3
              /
           UNDEF
          /    \
         E     2048
              /    \
             E      E

</stepper>
```

Example.
  4. Store 4096 at address 5

Remark.
  Call this trie "trie4".

```
<stepper>

~> (trie-store trie3 5 4096)

=> [true
     (make-trie
        (t-left trie3)
        (trie-store (t-right trie3) (quotient 5 2) 4096)
        (t-val trie3))]))
=> (make-trie
     (t-left trie3)
     (trie-store (t-right trie3) 2 4096)  ;; recursion
     (t-val trie3))

=> (make-trie
     (t-left trie3)
     (make-trie
        (trie-store (t-left (t-right trie3)) (quotient 2 2) 4096)
        (t-right (t-right trie3))
        (t-val (t-right trie3)))
     (t-val trie3))
=> (make-trie
     (t-left trie3)
     (make-trie
        (trie-store (t-left (t-right trie3)) 1 4096)
        (t-right (t-right trie3))
        (t-val (t-right trie3)))
     (t-val trie3))

=> (make-trie
     (t-left trie3)
     (make-trie
        (make-trie
          (t-left (t-left (t-right trie3)))
          (make-trie
               (t-left (t-right (t-right trie3)))
               (t-right (t-right (t-right trie3)))
               4096)
          (trie-val (t-left (t-right trie3)))
        (trie-right (t-right trie3))
        (trie-val (t-right trie3)))
     (t-val trie3)))

                        trie4
                  /                \
                1                   3
              /                   /
           UNDEF               UNDEF
          /    \              /    \
        E     2048          E     4096
              /    \
            E      E
</stepper>
```

Remark. A general picture of how a trie looks like. Numbers indicates address.

```
                              trie
                            /              \
                          0                 1
                         /   \              /   \
                              E    2           E     3
                             / \    / \         / \    / \
                           E   4 E   6       E   5 E   7
```

Implementation. trie-fetch

```
<rkt>

  (define (trie-fetch tr addr)
    (cond
      [(zero? addr) (t-val (t-left br))] ;;base case 1: addr=0, get left subtrie's value
      [(= 1 addr) (t-val (t-right br))]  ;;base case 2: addr=1, get right subtrie's value
      [(even? addr) (trie-fetch (t-left br) (quotient addr 2))] ;; recursive case 1: addr=even, go down left
      [else (trie-fetch (t-right br) (quotient addr 2))] ;; recursive case 2: go down right

</rkt>
```

Remark.
This part should be quite intuitive, since it's pretty similar to our tree-val function.
Basically we keep deviding the address by 2, if it's even we go to left side and if it's odd we go to the right side, until the address is reduced down to 1, which we will return the value stored in the right subtrie. Note that we will never use the first line unless we are given address = 0.

Remark. What trie really is...
"Reverse Postfix Search"
For base-10 trie, there would be 11 children, 10 for numbers 0~9 and 1 for an extra boolean value.
In the examples above, the (quotient addr 2) operations essentially convert our base-10 numbers to base-2, so we can use a binary trie.

For example, if we search for addr=13, we convert 13 to binary which is 8+4+0+1 so 1101.
We start from the end (essentially reverse the binary number to 1011, which is what prof Cormack does in his implementation), 1 means right, then left (0), then right (1), and right (1).

Credit: Felix

Subtopic. ★★★☆ 8.2: RAM.

Remark.
Now we implement RAM using trie.
Well, basically just a wrapper...

Implementation.

```
<rkt>

  (define-struct wrapper (w))   ;; a wrapper struct
  (define ram (wrapper empty))  ;; the empty ram

</rkt>
```

Implementation. Store and fetch

```
<rkt>

  ;; r: ram we are working with
  ;; addr: address we want to store the number at
```

```
    ;; num: the number we want to store in the ram
    (define (ram-store r addr num)
      (make-wrapper
        (trie-store (wrapper-w r) addr
          (if (and (integer? num) (<= 0 num))
            num
            (error "attempt to store non-Nat value")))))
    ;; A wrapper contains a single object: w.
    ;; w is a trie.
    ;; We want to store num at address=addr inside w.
    ;; Also, before we storing the value inside w, we check if num is a natural number.

    ;; r: the ram we are working with
    ;; addr: the address we want to get value from
    (define (ram-fetch r addr)
      (define val (trie-fetch (wrapper-w r) addr))
      (if (number? val)
        val
        (error "attempt to fetch undefined value")))
    ;; (wrapper-w r) unwraps the ram, gives us the trie.
    ;; Then we use trie-fetch to get the value at address=addr from the trie.
    ;; Next we test if val is a legit value (aka if it's a natural number).
    ;; If yes, we return val.
    ;; Otherwise we raise an error.

  </rkt>
```

Implementation. core-dump

```
  <rkt>

    (define (core-dump ram)
      (define (core-dump ram start finish)
        (define (ram-fetch r addr)
          (define val (trie-fetch (wrapper-w r) addr))
          val)
        (cond
          [(> start finish) (void)] ;; base case: if start > finish, the job is finished.
          [else
            (define v (ram-fetch ram start));; the value we get from ram-fetch from ram at address=start
            (if (number? v)   ;; tests if it is a legit value or not
              (printf "~a: ~a\n" start r)
              (void))
            (core-dump ram (add1 start) finish)]))
        (core-dump ram 0 1000))

  </rkt>
```

Remark.
This one should be pretty easy to understand.
(core-dump ram start finish): local helper.
(ram-fetch r addr): the ram-fetch function without the if-statement which checks if the value is legit.
v: the value we get from ram-fetch from ram at address=start
Think of this as a printing out elements of the list one by one.

==

[redacted]