Fall CS 145 Final Review Guide.

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

==

Topic. 8: Gen, RAM, CPU.

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.
==

Example. (sumto n) using RAM

```
<rkt>

  ;; Recall sumto with accumulator helper
  (define (sumto n)
    (define (sumacc i n a)
      (cond
        [(> i n) a]
        [else (sumacc (add1 i) n (+ a i))]))
    (sumacc 0 n 0))

  ;; Now we implement it using RAM
  ;; Rules:
  ;;   1. tail recursion only
  ;;   2. one parameter only -- must be RAM
  ;;   3. result must be RAM

  (define (sumto-ram n)
    (define (sumacc ram)
      ;; really (sumacc i n a)
      ;; this calculates a + (sum from i to n)
      ;; ram[0]: i
      ;; ram[1]: n
      ;; ram[2]: a
      (define i (ram-fetch ram 0))
      (define n (ram-fetch ram 1))
      (define a (ram-store ram 2))
      (cond
```

```
        [(> i n) ram]
        [else
          (define r1 (int-add ram 2 i))   ;; a += i
          (define r2 (int-add1 r1 0))     ;; i += 1
          (sumacc r2)]))
    (define r1 (ram-store ram 0 0))
    (define r2 (ram-store r1 1 n))
    (define r3 (ram-store r2 2 0))
    (ram-fetch (sumacc r3 2)))

</rkt>
```

===

Example. PairRAM

A pair-ram can store ints or cons pairs.
pairram[0]: where the list starts
pairram[1..99]: open to interpretation.

```
<rkt>
  (define listram (ram-store ram 0 100))  ;; list-start = 100.
  (define (list-cons r head-value tail-addr)
    (let* {[tail-value (ram-fetch r tail-addr)]
           [list-start (ram-fetch r 0)]
           [r1 (ram-store r list-start head-value)]   ;; we store head-value at list-start
           [r2 (ram-store r1 (add1 list-start) tail-value)]  ;; we store tail-value at (add1 list-start)
           [r3 (ram-store r2 tail-addr list-start)]   ;; now tail-addr points to list-start
           [r4 (ram-store r3 0 (+ 2 list-start))]}    ;; we modify list-start (by adding 2 to it)
         r4))

</rkt>

<stepper>
  ~> (define listram (ram-store ram 0 100))
  ~> (define p1 listram)
  ~> (define p2 (ram-store p1 77 0))   ;; [77] = 0
  ~> (define p3 (list-cons p2 10 77))  ;; [77] = '(10)

  => head-value = 10; tail-addr = 77
  => tail-value = (ram-fetch p2 77) = 0 (which is empty)
     list-start = (ram-fetch p2 0 ) = 100
     r1 = (ram-store p2 100 10) = [100]: 10
     r2 = (ram-store r1 101 0)  = [101]: 0
     r3 = (ram-store r2 77 100) = [77] : 100
     r4 = (ram-store r3 0  102) = [0] : 102

  ;; Right now: [0]   : 102     ;; address 0 points to 102, which is our next empty spot to add values.
               [77]  : 100     ;; address 77 points to a list, which starts at 100.
               [100] : 10      ;; address 100 stores 10, which is the first value of our list.
               [101] : 0       ;; address 101 stores 0, indicates there is no more value in the list.



  ~> (define p4 (list-cons p3 20 77))  ;; [77] = '(20 10)

  => head-value = 20; tail-addr = 77
```

```
  => tail-value = (ram-fetch p3 77) = 100
     list-start = (ram-fetch p3 0 ) = 102
     r1 = (ram-store p3 102 20)  = [102]: 20
     r2 = (ram-store r1 103 100) = [103]: 100
     r3 = (ram-store r2 77  102) = [77] : 102
     r4 = (ram-store r3 0   104) = [0]  : 104

  ;; Right now: [0]   : 104     ;; address 0 points to 104, which is our next empty spot to add values.
                [77]  : 102     ;; address 77 points to a list, which starts at 102.
                [100] : 10      ;; address 100 stores 10
                [101] : 0       ;; address 101 stores 0, indicates there is no more value in the list.
                [102] : 20      ;; address 102 stores 20
                [103] : 100     ;; address 103 points to 100, which is the address storing the element
                                ;; after 20.

  ~> (define p5 (list-cons p4 30 77))  ;; [77] = '(30 20 10)

  => head-value = 30; tail-addr = 77
  => tail-value = (ram-fetch p4 77) = 102
     list-start = (ram-fetch p4 0 ) = 104
     r1 = (ram-store p4 104 30)  = [104]: 30
     r2 = (ram-store r1 105 102) = [105]: 102
     r3 = (ram-store r2 77  104) = [77] : 104
     r4 = (ram-store r3 0   106) = [0]  : 106

  ;; Right now: [0]   : 106     ;; address 0 points to 104, which is our next empty spot to add values.
                [77]  : 104     ;; address 77 points to a list, which starts at 104.
                [100] : 10      ;; address 100 stores 10
                [101] : 0       ;; address 101 stores 0, indicates there is no more value in the list.
                [102] : 20      ;; address 102 stores 20
                [103] : 100     ;; address 103 points to 100, which is the address storing the element
                                ;; after 20.
                [104] : 30      ;; address 104 stores 30
                [105] : 102     ;; address 105 points to 102, which is the address stroing the element
                                ;; after 30.

  ;; At this point, you should realize that tail-addr (in this case 77) always points to the address
  ;; of first element in the list.

  ;; Thus, following the pointers we can retrieve the entire list.
</stepper>

<rkt>
  (define (list-empty? r list-addr)
    (zero? (ram-fetch r list-addr)))

  ;; get the first element of the list
  ;; r: ram
  ;; list-addr: where does our list start?
  (define (list-first r list-addr)
    (let* {[addr-of-car (ram-fetch r list-addr)]
           [val-of-car (ram-fetch r addr-of-car)]}
           val-of-car))
```

```
;; drop the first element of the list, aka replace the first element with the second element
;; and since this is a linked list, we don't need to worry about anything else.
;; r: ram
;; list-addr: where does our list start?
(define (list-dropfirst r list-addr)
  (let* {[addr-of-car (ram-fetch r list-addr)]
         [addr-of-cdr (add1 addr-of-car)] ;; make sure you understand this step
         [val-of-second (ram-fetch addr-of-cdr)]}
       ;; now we want to store value of second at the address which originally stores first.
       (ram-store r addr-of-car val-of-second)))

;; store val as the first element.
;; r: ram
;; list-addr: where does our list start?
;; mew-val: our new value for first element.
(define (list-setfirst r list-addr new-val)
  (let* {[addr-of-car (ram-fetch r list-addr)]
         [new-ram (ram-store r addr-of-car new-val)]}
       new-ram))

;; This is explained in CPU.pdf, I'm too lazy to do it again.
(define (ram-copy ram from-addr to-addr)
  (ram-store ram to-addr (ram-fetch ram from-addr)))

;; listasRAM->list
(define (outstream r list-addr)
  (define r1 (ram-copy r list-addr 55));; 55 is arbitrary. We copy [list-addr] to [55]
  (define (iterate r)
    (cond
      [(list-empty? r 55) empty]
      [else (cons (list-first r 55) (iterate (list-dropfirst r 55)))]))
  (iterate r1))

;; list->listasRAM
(define (instream ram list-addr lst)
  (define r1 (ram-store ram list-addr 0))
  (define (helper r lst)
    (cond
      [(empty? lst) r]
      [else (helper (list-cons r (first lst) list-addr) (cdr-lst))]))
  (helper r1 (reverse lst))) ;; make sure you understand why we need to reverse the list.
</rkt>
```

===

END

[redacted]