MF

THE UNIVERSITY OF BRITISH COLUMBIA
CPSC 110: MIDTERM 2 – November 4, 2014

Last Name: TAYLOR First Name: BRIAN

Signature: Brian Taylor UBC Student #: 5 2 3 1 1 8 5 9

Important notes about this examination

1. You have 120 minutes to complete this exam.
2. This exam will be graded largely on how well you follow the design recipes. You have been given a copy of the Recipe Exam Sheet. Use it!
3. Put away books, papers, laptops, cell phones... everything but pens, pencils, erasers and this exam.
4. Good luck!

Student Conduct during Examinations

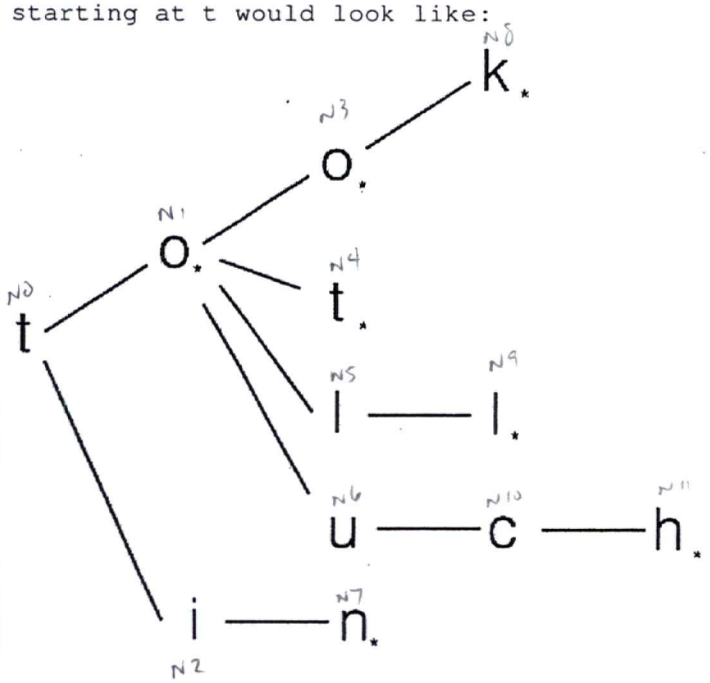
1. Each examination candidate must be prepared to produce, upon the request of the invigilator or examiner, his or her UBCcard for identification.
2. No questions will be answered in this exam. If you see text you feel is ambiguous, make a reasonable assumption, write it down, and proceed to answer the question.
3. No examination candidate shall be permitted to enter the examination room after the expiration of one-half hour from the scheduled starting time, or to leave during the first half hour of the examination. Should the examination run forty-five (45) minutes or less, no examination candidate shall be permitted to enter the examination room once the examination has begun.
4. Examination candidates must conduct themselves honestly and in accordance with established rules for a given examination, which will be articulated by the examiner or invigilator prior to the examination commencing. Should dishonest behaviour be observed by the examiner(s) or invigilator(s), pleas of accident or forgetfulness shall not be received.
5. Examination candidates suspected of any of the following, or any other similar practices, may be immediately dismissed from the examination by the examiner/invigilator, and may be subject to disciplinary action:
   i. speaking or communicating with other examination candidates, unless otherwise authorized;
   ii. purposely exposing written papers to the view of other examination candidates or imaging devices;
   iii. purposely viewing the written papers of other examination candidates;
   iv. using or having visible at the place of writing any books, papers or other memory aid devices other than those authorized by the examiner(s); and,
   v. using or operating electronic devices including but not limited to telephones, calculators, computers, or similar devices other than those authorized by the examiner(s)—(electronic devices other than those authorized by the examiner(s) must be completely powered down if present at the place of writing).
6. Examination candidates must not destroy or damage any examination material, must hand in all examination papers, and must not take any examination material from the examination room without permission of the examiner or invigilator.
7. Notwithstanding the above, for any mode of examination that does not fall into the traditional, paper-based method, examination candidates shall adhere to any special rules for conduct as established and articulated by the examiner.
8. Examination candidates must follow any additional examination rules or directions communicated by the examiner(s) or invigilator(s).

Please do not write in this space:

Question 1:	15	&
Question 2:	11	
Question 3:	19	JS
Question 4:	7	SM.
Question 5:	16	KJ
Question 6:	15	KJ

0 236366 453321

When spelling correction first appeared in typewriters it was very important to store the information about legal words as compactly as possible. One technique for doing so was to use prefix trees, in which all words that start with the same prefix share the same data representation for that common prefix, they only have different data for their differing endings. So for example part of the prefix tree starting at t would look like:



Of course the complete tree starting at t is much bigger.

Checking whether the sequence of characters t-o-o is a legal word means starting with the t, then going to the o, then to the next o. When the last letter is gone, the tree is still at the second o above. The * on that node indicates that too is a legal word. Every full path is a legal word because it would waste space to store bad words that you don't need.

As another example, checking whether t-o-u is legal starts at the t, then goes to o, then u. Because there is no * there, tou is not a legal word.

So each node in the prefix tree has its letter and a flag indicating whether the path from the root of the tree to there is a legal word.

In the picture the * on some nodes indicates that the tree up to that point is a legal word. In the picture above t, tol, tou, touc and ti are not legal words.

To, too, took, tot, toll, touch and tin are legal words.

;PROBLEM 1:

Design a data definition to represent a prefix tree. Call the main type PrefixTree. Be sure to include any necessary define-structs, type comments, interpretations, examples and templates. If your types involve mutual recursion, group your data definitions appropriately. If your types involve a list type you may use the (listof Type) notation, but be sure to write a template and only a template for the listof type.

You MUST define an example prefix tree named PT1 that includes the following legal words:

to
too
tot

```
(define-struct (node letter flag subs))
;; Node is (make-node String Boolean (list of Node))
;; interp. letter is self-evident
;;         flag is true if a legal word terminates there
;;         subs is a list of nodes that continues the paths.

;; PrefixTree is one of:
;; - empty
;; - (cons Node (list of Node))
;; interp. a tree containing paths corresponding to legal words

(define N0 (make-node "t" false (list N1 N2)))
(define N1 (make-node "o" true (list N3 N4)))
(define N2 (make-node "i" false (list N7)))
(define N3 (make-node "o" true (list N8)))
(define N4 (make-node "t" true empty))
(define N5 (make-node "l" false (list N9)))
(define N6 (make-node "u" false (list N10)))
(define N7 (make-node "n" true empty))
(define N8 (make-node "k" true empty))
(define N9 (make-node "l" true empty))
(define N10 (make-node "c" false (list N11)))
(define N11 (make-node "h" true empty))
```

4
3
3
1
4
15

BLANK PAGE FOR YOUR ANSWER TO PROBLEM 1

(define PT1 (list N0)

```
;; template for (list of Node)
#;
(define (fn-for-lon)
  (cond [(empty? ...]
        [else
         (... (fn-for-node (first lon))
              (fn-for-lon (rest lon)))]))
```

PROBLEM 2:

Design a function that consumes a PrefixTree and produces a count of how many legal words are in the tree.

You must include signature, purpose, check-expects, and your function definition.

```
;; PrefixTree -> Natural
;; produces a count of how many legal words are in the PrefixTree

(check-expect (count-legal PT1) 7)
(check-expect (count-legal empty) 0)

(define (count-legal p)
  (local [(define (fn-for-node n)
            (if (node-flag n) 1 0))]
    (define (fn-for-lon lon)
      (cond [(empty? lon) 0]
            [else
             (+ (fn-for-node (first lon))
                (fn-for-lon (rest lon)))]))]
    (+ (fn-for-node (first p))
       (fn-for-lon (rest p)))))
```

2
1
2
2
1

1
0
0
1
1
0

11

BLANK PAGE FOR YOUR ANSWER TO PROBLEM 2

;PROBLEM 3:

Design a function that consumes a (listof String) and a (listof Natural) and produces a (listof String) that contains n copies of each String from the (listof String), where n is the corresponding Natural in the (listof Natural). You must assume that the two lists have the same length.

For example, (n-copies (list "a" "b" "c") (list 1 2 3)) produces (list "a" "b" "b" "c" "c" "c")

and (n-copies (list "a" "b" "c") (list 0 1 4)) produces (list "b" "c" "c" "c" "c")

and (n-copies (list "a" "b" "c") (list 2 0 3)) produces (list "a" "a" "c" "c" "c")

Treat this as a function operating on two complex data. You must show:
- signature, purpose, check-expects
- cross-product of types comment table with labeled axes and properly filled in cells
- complete function definition
- to show the correspondence between function and table you should number each case in the function's cond, and number the corresponding table cell(s) (if possible you must reduce the number of cases)

NOTE: No credit will be given to solutions that don't show a clear correspondence to the cross-product of types comment table.

```
(check-expect (string-copies empty empty) empty)
(check-expect (string-copies (list "a") (list 0)) empty)
(check-expect (string-copies (list "a") (list 1)) (list "a"))
(check-expect (string-copies (list "a" "b") (list 2 3)) (list "a" "a" "b" "b" "b"))

;; (listof String) (listof Natural) -> (listof String)
;; produces a list with each element of first list copied n times, where n
;;          is the corresponding element of the second list
```

	lon	empty	add1
~~List of~~ ~~String~~ los	empty	empty ①	not possible
	cons	Not possible	② (...) (first list of String) (first list of Natural)) -1

2 2 0 3 0 3 3 4 2 = 19

BLANK PAGE FOR YOUR ANSWER TO PROBLEM 3

```
(define (string-copies los lon)
  (cond [(empty? los) empty]                                          ①
        [else
         (append (repeat-string (first los) (first lon))              ②
                 (string-copies (rest los) (rest lon)))]))
```

should really be cond, not if, -didn't take off marks for this question

need signature, purpose ... etc. or put in a local (-1)

```
(check-expect (repeat-string "a" 0) empty)
(check-expect (repeat-string "a" 2) (list "a" "a"))

(define (repeat-string s n)
  (cond [(= 0 n) empty]
        [else
         (append (list s) (repeat-string s (sub1 n)))]))
```

```
;
 PROBLEM 4:

 The spinning bears world problem from Coursera is on this page and the next two
 pages.

 Where possible, refactor the functions so that they use built-in abstract functions.
 You should cross out the function bodies of the functions that you are refactoring
 and neatly write the new function bodies beside the original (crossed out) ones.

 You do not need to make any other changes to this program.
```

```
(require 2htdp/image)
(require 2htdp/universe)

;; Spinning Bears

;; =================
;; Constants:

(define WIDTH 600) ; width of the scene
(define HEIGHT 700) ; height of the scene

(define SPEED 3)  ; speed of rotation

(define MTS (empty-scene WIDTH HEIGHT)) ; the empty scene

(define BEAR-IMG  )

;; =================
;; Data definitions:

(define-struct bear (x y r))
;; Bear is (make-bear Number[0,WIDTH] Number[0,HEIGHT] Number)
;; interp.  (make-bear x y r) is the state of a bear, where
;;  x is the x coordinate in pixels,
;;  y is the y coordinate in pixels, and
;;  r is the angle of rotation in degrees

(define B1 (make-bear 0 0 0)) ; bear in the upper right corner
(define B2 (make-bear (/ WIDTH 2) (/ HEIGHT 2) 90)) ; sideways bear in the middle

#;
(define (fn-for-bear b)
  (... (bear-x b)      ; Number[0,WIDTH]
       (bear-y b)      ; Number[0,HEIGHT]
       (bear-r b)))    ; Number

;; Template Rules Used:
;; - compound: 3 fields

;; ListOfBear is one of:
;; - empty
;; - (cons Bear ListOfBear)
(define LB0 empty)
(define LB1 (cons B1 empty))
(define LB2 (cons B1 (cons B2 empty)))
```

```
;
(define (fn-for-lob lob)
  (cond [(empty? lob) (...)]
        [else
         (... (fn-for-bear (first lob))
              (fn-for-lob (rest lob)))]))

;; Template Rules Used:
;; - one of: 2 cases
;; - atomic distinct: empty
;; - compound: 2 fields
;; - reference: (first lob) is Bear
;; - self-reference: (rest lob) is ListOfBear

;; =================
;; Functions:

;; ListOfBear -> ListOfBear
;; start the world with (main empty)
;;
(define (main lob)
  (big-bang lob                          ; ListOfBear
            (on-tick   spin-bears)       ; ListOfBear -> ListOfBear
            (to-draw   render-bears)     ; ListOfBear -> Image
            (on-mouse  handle-mouse)))   ; ListOfBear Integer Integer MouseEvent -> ListOfBear

;; ListOfBear -> ListOfBear
;; spin all of the bears forward by SPEED degrees

(check-expect (spin-bears empty) empty)
(check-expect (spin-bears
               (cons (make-bear 0 0 0) empty))
              (cons (make-bear 0 0 (+ 0 SPEED)) empty))
(check-expect (spin-bears
               (cons (make-bear 0 0 0)
                     (cons (make-bear (/ WIDTH 2) (/ HEIGHT 2) 90)
                           empty)))
              (cons (make-bear 0 0 (+ 0 SPEED))
                    (cons (make-bear (/ WIDTH 2) (/ HEIGHT 2) (+ 90 SPEED))
                          empty)))

;; Took template from ListOfBears

(define (spin-bears lob)
  (cond [(empty? lob) empty]                    (map spin-bear lob)
        [else
         (cons (spin-bear (first lob))
               (spin-bears (rest lob)))]))

;; Bear -> Bear
;; spin a bear forward by SPEED degrees

(check-expect (spin-bear (make-bear 0 0 0)) (make-bear 0 0 (+ 0 SPEED)))
(check-expect (spin-bear (make-bear (/ WIDTH 2) (/ HEIGHT 2) 90))
              (make-bear (/ WIDTH 2) (/ HEIGHT 2) (+ 90 SPEED)))

;; Took template from Bear
(define (spin-bear b)
  (make-bear (bear-x b)
             (bear-y b)
             (+ (bear-r b) SPEED)))
```

```
;; ListOfBear -> Image
;; render the bears onto the empty scene

(check-expect (render-bears empty) MTS)
(check-expect (render-bears (cons (make-bear 0 0 0) empty))
              (place-image (rotate 0 BEAR-IMG) 0 0 MTS))
(check-expect (render-bears
               (cons (make-bear 0 0 0)
                     (cons (make-bear (/ WIDTH 2) (/ HEIGHT 2) 90)
                           empty)))
              (place-image (rotate 0 BEAR-IMG) 0 0
                           (place-image (rotate 90 BEAR-IMG) (/ WIDTH 2) (/ HEIGHT 2)
                                        MTS)))


;; Took Template from ListOfBear

(define (render-bears lob)
  (cond [(empty? lob) MTS]
        [else
         (render-bear-on (first lob) (render-bears (rest lob)))]))
```

(foldr render-bear-on MTS lob) ✓

2 1 2 1 1 / 7

```
;; Bear Image -> Image
;; render an image of the bear on the given image

(check-expect (render-bear-on (make-bear 0 0 0) MTS)
              (place-image (rotate 0 BEAR-IMG) 0 0 MTS))
(check-expect (render-bear-on (make-bear (/ WIDTH 2) (/ HEIGHT 2) 90) MTS)
              (place-image (rotate 90 BEAR-IMG) (/ WIDTH 2) (/ HEIGHT 2) MTS))

;; Took Template from Bear
(define (render-bear-on b img)
  (place-image (rotate (modulo (bear-r b) 360) BEAR-IMG) (bear-x b) (bear-y b) img))
```

```
;; ListOfBear Integer Integer MouseEvent -> ListOfBear
;; On mouse-click, adds a bear with 0 rotation to the list at the x, y location
(check-expect (handle-mouse empty 5 4 "button-down") (cons (make-bear 5 4 0) empty))
(check-expect (handle-mouse empty 5 4 "move") empty)

;; Templated according to MouseEvent large enumeration.
(define (handle-mouse lob x y mev)
  (cond [(mouse=? mev "button-down") (cons (make-bear x y 0) lob)]
        [else lob]))
```

PROBLEM 5:

Consider the data definitions below.

Design an abstract fold function for Node. You must neatly edit the encapsulated templates below and write the signature and purpose. You do not need to write any check-expects.

```
(define-struct node (name edges))
;; Node is (make-node String (listof Edge))
;; interp. a node with a name and associated edges

(define-struct edge (weight dest))
;; Edge is (make-edge Natural Node)
;; interp. an edge with the given weight, ending at the dest(ination) node

(define N0 (make-node "A" empty))
(define N1 (make-node "B" empty))
(define E1 (make-edge 1 N0))
(define E2 (make-edge 8 N1))
(define N2 (make-node "C" (list E1 E2)))
```

;; (String x -> y) (y x -> x) (Natural y -> y) x Node -> x
;; abstract fold function for Node.

```
#;(define (fold-node c1 c2 c3 b1 n0)
(define (fn-for-node n0)

  (local [(define (fn-for-node n)
            (c1 (node-name n)
                (fn-for-loe (node-edges n))))
          (define (fn-for-loe loe)
            (cond [(empty? loe) (b1 ...)]
                  [else
                   (c2 (fn-for-edge (first loe))
                       (fn-for-loe (rest loe)))]))
          (define (fn-for-edge e)
            (c3 (edge-weight e)
                (fn-for-node (edge-dest e))))]

    (fn-for-node n0)))
```

3
2
2
1
1
1/10

1
1
2
2/6
16/

;

PROBLEM 6:

Design a function that consumes a Node n and produces a list of the names of all nodes that can be reached from n.

Your function MUST CALL the abstract fold function for Node that you designed in Problem 5.

You must include signature, purpose, check-expects, and your function definition.

Carefully consider your check-expects - to get full marks the lists that you are expecting to be produced by your function must be in the right order.

```
(define N3 (make-node "D" (list (make-edge 1 N2))))
(check-expect (all-dest N0) (list "A"))
(check-expect (all-dest N1) (list "B"))
(check-expect (all-dest N2) (list "C" "A" "B"))
(check-expect (all-dest N3) (list "D" "C" "A" "B"))

;; Node -> (list of String)
;; produce a list of names of all nodes that can be reached from a node.

(define (all-dest n)
  (fold-node cons
             cons append
             (λ (x y) y)
             empty
             n))
```

c1 : cons
c2 : cons
c3 : (λ (x y) y)
b1 : empty

2
2
2
2/8

2
0
2
2
1/7

15/16

BLANK PAGE FOR YOUR ANSWER TO PROBLEM 6