Rules Governing Formal Examinations

1. Each candidate must be prepared to produce, upon request, a UBC card for identification.

2. Candidates are not permitted to ask questions of the invigilators, except in cases of supposed errors or ambiguities in examination questions. 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 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.

4. Candidates suspected of any of the following, or similar, dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action:
   - having at the place of writing any books, papers or memoranda, calculators, computers, sound or image players/recorders/transmitters (including telephones), or other memory aid devices, other than those authorized by the examiners;
   - speaking or communicating with other candidates; and
   - purposely exposing written papers to the view of other candidates or imaging devices. The plea of accident or forgetfulness shall not be received.

5. Candidates must not destroy or mutilate any examination material; must hand in all examination papers; and must not take any examination material from the examination room without permission of the invigilator.

6. Candidates must follow any additional examination rules or directions communicated by the instructor or invigilator.
Problem 1

Given the following function definition:

(define (bar x a b)
  (local [(define (times-x i)
           (* i x))]
    (list (times-x a) (times-x b))))

(A) What is the value produced by

(bar 3 4 5)

(B) Write out the state of the hand-evaluation immediately after the second multiplication. Include all lifted definitions as well as the current state of the evaluation. You may wish to do more hand-stepping to get to the point in the evaluation we are asking for, but if you do so, be sure to clearly identify what part of what you are written should be graded.
Problem 2

Given the following data definitions

```
(define-struct box (h w d))
;; Box is (make-box Natural Natural Natural)
;; interp. a shipping box, with height, width and depth
(define B1 (make-box 3 4 5))
(define B2 (make-box 5 6 7))

;; ListOfBox is one of:
;; - empty
;; - (cons Box ListOfBox)
;; interp. all the boxes in a load
(define LOB1 empty)
(define LOB2 (list B1 B2))
```

Design a function that consumes a list of boxes and produces the sum of the volumes of the individual boxes. (Remember that the volume of a box is width * height * depth. Be sure to follow all appropriate helper function rules. Do not use built in abstract functions. If you would like to, you are free to use encapsulation to hide helpers. You might want to start by annotating the type comments with reference arrows and writing templates, but you are not required to do so.
Problem 3

Complete the design of each of the following functions. You should write BOTH the template you use, and a completed function definition. We would like you to use built-in abstract functions, but partial credit will be awarded to solutions that use traditional templates.

(A)
;; (listof String) -> (listof Natural)
;; produce list of the string-lengths of the strings in los
(check-expect (lengths (list "foo" "apple")) (list 3 5))

(define (lengths los) empty) ;stub

;<put the template here>

;<put the completed function definition here>
(B)
Assume the data definition of Box from problem 2.

;; Natural (listof Box) -> (listof Box)
;; produce a list of only those boxes with height greater than h
(check-expect (only-taller 4 (list B1 B2)) (list B2))

(define (only-taller h lob) empty) ;stub

;<put the template here>

;<put the completed function definition here>
(C)
;; Natural -> Natural
;; Produce the sum of the squares of the first n natural numbers.
(check-expect (sos 4) (+ (sqr 0) (sqr 1) (sqr 2) (sqr 3)))

(define (sos n) 0) ;stub

This problem has a very nice solution that uses 3 built-in abstract function; it also has a
good solution using just 2 built in abstract functions. Hint: Work carefully to count the
number of differences (or dependencies) between the natural consumed and the natural
produced.

;; <put the template here>

;; <put the completed function definition here>
Problem 4

In this problem we are going to work with a very simple XML-like data definition. Don’t worry, you don’t need to remember anything about XML, our data definition has everything you need to know. This definition lets us represent documents that look a little like Doctor Racket boxes:

Note that each doc starts with a single line of attributes, and then contains any number of words and docs. Here are the types, interpretations and examples. The examples are for the above 3 docs.

```
(define-struct doc (attrs contents))
;; Doc is (make-doc ListOfString ListOfElement)
;; interp. an XML like document, with attributes and contents

;; ListOfString is one of:
;; - empty
;; - (cons String ListOfString)
;; interp. the attributes of a Doc

;; ListOfElement is one of:
;; - empty
;; - (cons Element ListOfElement)
;; interp. the contents of a Doc

;; Element is one of:
;;   Doc
;;   String
;; interp. the elements of a Doc

(define D1 (make-doc (list "a") empty))
(define D2 (make-doc (list "b" "c") (list "hello" "tom")))
(define D3
  (make-doc (list "a" "b")
    (list "doc" "has" "two" "subs"
      (make-doc empty (list "hi")))
    (make-doc (list "d"))
```
(list "there"
    (make-doc empty
        (list "dog")))))))

;;
(define (fn-for-doc b)
  (... A:(fn-for-los (doc-attrs b))
      B:(fn-for-loe (doc-contents b))))

;;
(define (fn-for-los los)
  (cond [(empty? los) (...)]
        [else (... (first los)
                      C:(fn-for-los (rest los)))]))

;;
(define (fn-for-loe loe)
  (cond [(empty? loe) (...)]
        [else (... D:(fn-for-element (first loe))
                    E:(fn-for-loe (rest loe)))]))

;;
(define (fn-for-element e)
  (cond [(doc? e) (... F:(fn-for-doc e))]
        [(string? e) (... e)]))

(A) Neatly annotate the type comments by doing 3 things:
   • Draw an arrow from each type reference to the the corresponding type definition. All your arrows should have the pointy-end ending at a type name that appears before ‘is’.
   • Neatly label each arrow with one of MR, SR or R, depending on whether the reference is a mutual reference, self reference or an ordinary reference.
   • We have labeled each helper function call in the templates with a letter A, B ... F. Neatly label each of your arrows with the letter of the corresponding helper function call.
(B) Design a function that consumes a doc and produces the total size of its contents, which is defined as the sum of the string-lengths of all the strings in the box’s contents, this includes all its sub-docs and their sub-docs etc. It does not include attributes. To save you writing we are putting a copy of the templates below. We suggest you work by writing code around these templates, crossing part of them out, filling them in etc. Don’t forget your signature, purpose and check-expects above the function definition. You can abbreviate check-expect to c-e, and you should use the example data defined above.

```
(define (fn-for-doc b)
  (... A:(fn-for-los (doc-attrs b))
    B:(fn-for-loe (doc-contents b))))

(define (fn-for-los los)
  (cond [(empty? los) (...)]
    [else (... (first los)
        C:(fn-for-los (rest los)))]))

(define (fn-for-loe loe)
  (cond [(empty? loe) (...)]
    [else (... D:(fn-for-element (first loe))
        E:(fn-for-loe (rest loe)))]))

(define (fn-for-element e)
  (cond [(doc? e) (... F:(fn-for-doc e))]
    [(string? e) (... e)]))
```
This is the hardest part of the midterm. If you are unsure how to do it we encourage you to go back and double-check your other work before working on this part.

Design an abstract function that can operate on a doc and its attributes, and all the contents recursively. In other words, turn the templates into an abstract function. Also include a signature. You do not need to include purpose or check-expects.

To save you writing we are putting a copy of the templates here. We suggest you work by writing code around these templates, crossing part of them out, filling them in etc.

```
(define (fn-for-doc b)
  (... A:(fn-for-los (doc-attrs b))
       B:(fn-for-loe (doc-contents b))))

(define (fn-for-los los)
  (cond [(empty? los) (...)]
        [else
         (... (first los)
              C:(fn-for-los (rest los)))]))

(define (fn-for-loe loe)
  (cond [(empty? loe) (...)]
        [else
         (... D:(fn-for-element (first loe))
              E:(fn-for-loe (rest loe)))]))

(define (fn-for-element e)
  (cond [(doc? e) (... F:(fn-for-doc e))]
        [(string? e) (... e)]))
```
(D) Design a function that consumes a box and produces a flat list of all the attributes in 
the box and its sub-boxes. Duplicates are OK in the result. Your implementation 
should use the abstract function you designed in part C. No partial credit will be 
awarded for this part of the problem if the function definition does not use the abstract 
function from part C.

Hint: recall that (append (list "a" "b") empty (list "x" "y")) produces 
(list "a" "b" "x" "y").
Use this page for extra space, clearly label any problem solution you write here.