/* Cover Page */
Use this page if you need more space to answer a question.
Question 1: Multiple Choice

For each of the following questions, circle the letter corresponding to the single best answer.

I) Suppose you are designing a function that consumes two complex data. Further suppose that one type is an itemization with three cases, while the other is an itemization with four cases. How many different cases will there be when you form the cross-product of the types comments?

   a) 3  
   b) 4  
   c) 7  
   d) 12  
   e) none of the above

II) Given the variable definition:

   `(define LOS (list "b" "c"))`

what is the value of the following expression?

   `(list "a" LOS)`

   a) (list "a" "b" "c")  
   b) (list (list "a") (list "b" "c"))  
   c) (list "a" (list "b" "c"))  
   d) (list (list "a") "b" "c")  
   e) none of the above

III) Consider the following function definition:

   `(define (do-lon t lon)
      (map (λ(n) (* n t)) lon))`

What is the value of the following expression?

   `(do-lon 3 (list 1 2 3))`

   a) (list 3 6 9)  
   b) (list 9 6 3)  
   c) undefined because the anonymous function doesn't consume t as a parameter  
   d) undefined for some reason other than that stated in c)
**Question 2: Function design**

In this question we will design a function that draws a bus route onto a map. A bus route is made up of an arbitrary number of points on an image of the map. To draw a route we draw a line between each successive pair of points in the route. So, if a route consists of the points P1, P2, P3 and P4, we draw that route by drawing a line from P1 to P2, P2 to P3 and P3 to P4. We assume that a route contains a minimum of two points.

Complete data definitions have been provided for you. Pay particular attention to the template for a function that consumes a Route. Given that a route is a list of at least two points, we can do something with the first AND the second item in the list, in both the base case and the general case.

You must design a function that consumes a route and an image of a map. The function must produce an image of the route plotted in red onto the map. Do not use any abstract list processing functions like `map`, `filter` or `foldr`. Do not use local expressions to encapsulate helper functions (in other words, you must provide a signature, purpose and tests for any helper function that you design).

**Hint:** the following built-in function will be useful

```lisp
;; Image Natural Natural Natural Natural String -> Image
(add-line img x1 y1 x2 y2 c)
;; produces an image obtained by drawing a line of color c
;; from (x1, y1) to (x2, y2) onto img
```

// continued on the next page
(require 2htdp/image)
(define MAP (square 100 "solid" "gray")); simple map

(define-struct point (x y))
;;;; Point is (make-point Natural Natural)
;;;; interp. a point at position (x, y) in an image
(define P1 (make-point 0 0))
(define P2 (make-point 10 30))
(define P3 (make-point 10 60))
(define P4 (make-point 25 65))
(define P5 (make-point 25 75))

;;;; (define (fn-for-point p)
;;;;   (... (point-x p)
;;;;       (point-y p)))

;;;; Route is (listof Point)
;;;; interp. a route is a list of at least two points
(define R1 (list P1 P2))
(define R2 (list P2 P3 P4 P5))

;;;; (define (fn-for-route r)
;;;;   (cond [(empty? (rest (rest r)))
;;;;          (... (fn-for-point (first r))
;;;;                (fn-for-point (second r)))]
;;;;          [else (... (fn-for-point (first r))
;;;;                   (fn-for-point (second r))
;;;;                   (fn-for-route (rest r)))])

// continued on next page
Design your function and any necessary helper functions below. There is additional space on the next page, if needed.

Reminder:
- do not use built-in abstract list processing functions
- do not use local expressions or anonymous helper functions
Question 3: Using built-in abstract functions

a) Design a function that consumes an album name and a list of photos and that produces a list of only those photos that are favourites and that belong to the given album. You must use built-in abstract functions wherever possible. You must encapsulate helper functions, if any, in a local expression or use anonymous helper functions. When designing your tests, use the constants that have been provided.

(define-struct photo (location album favourite))
;; Photo is (make-photo String String Boolean)
;; interp. a photo having a location, belonging to an album and having a
;; favourite status (true if photo is a favourite, false otherwise)

(define PHT1 (make-photo "photos/2012/june" "Victoria" true))
(define PHT2 (make-photo "photos/2012/june" "Victoria" false))
(define PHT3 (make-photo "photos/2012/august" "Seattle" true))
b) Complete the design of a function that consumes a list of images and that overlays them one on top of the other so that the first image in the list ends up on the bottom and the last one ends up on top. You must use built-in abstract functions \textit{wherever possible}. You must encapsulate helper functions, if any, in a local expression or use anonymous helper functions. Study the provided test carefully.

\begin{verbatim}
(define MTS (square 0 "solid" "white"))

;; (listof Image) -> Image
;; Overlays the images in loi to produce a single image
(check-expect (overlay-all (list (circle 30 "solid" "green")
                                 (circle 20 "solid" "yellow")
                                 (circle 10 "solid" "red"))
                 (overlay (circle 10 "solid" "red")
                          (circle 20 "solid" "yellow")
                          (circle 30 "solid" "green")
                          MTS))

(define (overlay-all loi) MTS) ;stub
\end{verbatim}
c) Complete the design of a function that consumes a threshold value (a natural number) and a list of natural numbers. The function must multiply the given numbers by 2 and produce a list of only those numbers which, having been multiplied by 2, are less than the given threshold value. You must use built-in abstract functions wherever possible. You must encapsulate helper functions, if any, in a local expression or use anonymous helper functions.

```scheme
;; Natural (listof Natural) -> (listof Natural)
;; multiplies numbers in lon by 2 and produces list of values
;; less than tv
(check-expect (times2/below 100 (list 10 20 30 40 50 60))
  (list 20 40 60 80))

(define (times2/below tv lon) empty) ;stub
```
Question 4: Mutable variables
Design a function that consumes a list of strings and produces the difference between the number of strings that have odd length and the number of strings that have even length. You must use for-each.

Hint:
- for full marks you must use a single for-each
- be sure to provide the type and an invariant for any accumulators that you use

(define LOS1 (list "a" "abc" "abcde"))
(define LOS2 (list "ab" "abdc"))
(define LOS3 (append LOS1 LOS2))

;; (listof String) -> Integer
;; produces difference between number of strings that have odd
;; length and number that have even length
(check-expect (diff-odd-even LOS1) (- 3 0)) ; 3 odd, no even
(check-expect (diff-odd-even LOS2) (- 0 2)) ; 0 odd, 2 even
(check-expect (diff-odd-even LOS3) (- 3 2)) ; 3 odd, 2 even
Question 5: Building templates from data definitions

Four data definitions are provided below. For each of these four data definitions, you must write a template for a function that consumes that type of data. Although no marks will be awarded for doing so, it is strongly recommended that you draw arrows on the types comments to identify references, self-references and mutual references in the data definitions.

```
(define-struct tarble (foo bar))
;;;; Tarble is (make-tarble Nurdle Wurdle)
;;;; a tarble with a foo and a bar

(define-struct nurdle (name lot))
;;;; Nurdle is (make-nurdle String ListOfTarble)
;;;; interp. a nurdle with a name and list of tarbles

;;;; ListOfTarble is one of:
;;;; - empty
;;;; - (cons Tarble ListOfTarble)
;;;; interp. a list of tarbles

;;;; Wurdle is String
;;;; interp. a wurdle's description
```

Note: there is additional space on the next page for your answer to this question, if needed.
**Question 6: Functions that consume two complex data**

In this question, we explore Huffman coding – a technique used to compress and decompress data. The idea is that each character is given a corresponding code that can be stored using less memory than the character itself. We won’t go into all the details here but will focus on just one part of the process. During the decompression phase, codes are used to traverse a binary tree and retrieve the corresponding character.

Here’s a sample Huffman tree (represented by the constant HT3 on the next page):

![Huffman Tree Diagram]

Note that every node stores a single character but we care only about the characters that are stored in leaf nodes – these are nodes whose left and right sub-trees are false. For this reason, we don’t show characters in the non-leaf nodes.

A Huffman code is a list of 0s and 1s that describes a path through the tree. Starting at the root of the tree, a 0 tells us to go to the left sub-tree and a 1 tells us to go to the right sub-tree. So, if we follow the code `(list 0)`, we end up at the node that contains the string “A”. Similarly, if we follow the code `(list 1 0 1)`, we end up at the node that contains the string “T”.

A code that does not lead to a leaf node is not valid. So, for example, given the tree above, the codes `empty`, `(list 1)`, `(list 1 0)`, `(list 0 1)`, `(list 1 0 1 0)` are not valid as they do not lead to leaf nodes.

Data definitions for a Huffman tree and a Huffman code are provided on the next page. When reading these data definitions, keep in mind that `1-String` is a built-in data definition that represents a string consisting of a single character.
(define-struct node (char left right))

;; HT (HuffmanTree) is one of:
;; - false
;; - (make-node 1-String HT HT)
;; interp. a Huffman coding tree where each node stores
;; a single character and has a left and right sub-tree

(define HT1 false)
(define HT2 (make-node "T" false false))
(define HT3 (make-node " "
   (make-node "A" false false)
   (make-node " "
      (make-node " "
         (make-node "R" false false)
         (make-node "T" false false))
      (make-node "P" false false))))

;; Code is one of:
;; - empty
;; - (cons 0 Code)
;; - (cons 1 Code)
;; interp. a Huffman code

(define C1 empty)
(define C2 (list 0))
(define C3 (list 1))
(define C4 (list 0 1))
(define C5 (list 1 0 0))
(define C6 (list 1 1 0))

// continued on next page
Design a function `decode` that consumes a Huffman code and a Huffman tree and that produces the string corresponding to the given code or false if the code is not valid. You may assume that the following function has already been designed and is available for your use in this question:

```scheme
;; HT -> Boolean
;; determines if Huffman tree is a leaf node
(check-expect (leaf? HT1) false)
(check-expect (leaf? HT2) true)
(check-expect (leaf? HT3) false)
(check-expect (leaf? (make-node " " false HT3)) false)
(check-expect (leaf? (make-node " " HT3 false)) false)
```

a) Note that the `decode` function consumes two complex data. Start by drawing the cross-product of types comments table and simplify.
b) Now use your cross-product of types-comments table to complete the design of the function `decode`. Be sure to use the constants C1, C2, C3, HT1, HT2 and HT3 when designing your tests.
Question 7: Tree traversals and tail recursion

(define-struct person (name mother father))

;; AFT (Ascendant Family Tree) is one of:
;; - empty
;; - (make-person String AFT AFT)
;; interp. an ascendant family tree where each person has a name,
;; mother and father
(define AFT1 empty)
(define AFT2 (make-person "Wil"
  (make-person "Chris"
    (make-person "Hua" empty empty)
    (make-person "Jian" empty empty))
  (make-person "Bob" empty empty))

;; AFT1 -> (listof String)
;; produces a list of the names of all the people in aft
;; <tests deliberately omitted: we don't want to give you the answer>
(define (all-names aft)
  (cond[(empty? aft) empty]
       [else
        (append (list (person-name aft))
                (all-names (person-mother aft))
                (all-names (person-father aft)))]))

a) Draw a picture of AFT2.

b) The following function consumes an AFT and produces a list of the names of all the people in the tree:

(define (fn-for-aft aft)
  (cond[(empty? aft) (...)]
       [else
        (... (person-name aft)
             (fn-for-aft (person-mother aft))
             (fn-for-aft (person-father aft)))]))

What list of names does the following function call produce? The order matters!

(all-names AFT2)
c) There are two hard questions on the exam – this is one of them. Redesign the `all-names` function so that the all recursive calls are in tail position. It is not necessary to provide tests for this function. Note that the order in which the names appear in the list does not have to be the same as for the function provided in part b).

**Hint:** you will need two accumulators, one of which is a work-list accumulator (also called a to-do accumulator). Be sure to include the type and an invariant for each accumulator.

```scheme
;; AFT -> (listof String)
;; produces a list of the names of all the people in aft
;; <tests deliberately omitted: we don't want you to concern yourself
;; with the order in which names appear in the list>

(define (all-names aft) empty) ; stub
```
**Question 8: Graphs**
In this question we model a rail network using a directed graph. Nodes on the graph will represent the train stations. If there is a train service from one station to another, we add an edge from one node to the other. Each station has a name, number of platforms and is designated wheelchair accessible or not. Here is a sample network:

There are 7 stations on this particular network. One of them, CRW, has 5 platforms and is not wheelchair accessible, whereas LIV has 8 platforms and is wheelchair accessible.

Data definitions have been provided below. At the very end of this question, you'll find a constant that represents the network pictured above, along with associated helper functions.

```
;; StnName is String
;; interp. the name of a station on a rail network
;; (get-stn StnName) produces the Station with the given name.
(define SNa "EUS")
(define SNb "LIV")

(define-struct stn (name platforms accessible links))
;; Station is (make-stn StnName Natural Boolean (listof StnName))
;; A station, with a name, a number of platforms, wheelchair accessibility (true if accessible, false otherwise) and the names of stations it links to.
(define Sa (make-stn "CRW" 5 false (list "HOL" "LIV")))
(define Sb (make-stn "LIV" 8 true (list "MAN")))

;; (listof StnName)
(define LOSNa (list "EUS" "LIV"))
```
This page left blank deliberately.
a) Edit the template provided below so that the function produces the largest number of platforms at any one station among those reachable from the given origin. This template is designed to visit all the stations reachable from origin at most once. Note that the template continues onto the next page. Be sure to provide the type and an invariant for the accumulator.

```scheme
;; StnName -> Natural
;; Produces the largest number of platforms at any one station among
;; those reachable from origin
(check-expect (max-platforms "BRM") 10)
(check-expect (max-platforms "CRW") 8)

(define (max-platforms origin)
  ;; Accumulator: visited is (listof StnName)
  ;; Invariant: the names of the stations already visited by fn-for-stn
  ;; Accumulator: todo is (listof StnName)
  ;; Invariant: links to be visited from stations already visited by -stn
  ;; Accumulator: acc is
  ;; Invariant:

  ;; Termination argument:
  ; base case: todo list is empty
  ; reduction step: add current station's links to todo list

  This traversal visits any station at most once. Since the network is finite it will terminate.

  (local [(define (fn-for-stn-name sn visited todo acc)
    (if (member? sn visited)
      (fn-for-losn visited todo
       (... sn acc))
      (fn-for-stn (get-stn sn) visited todo
       (... sn acc)))]))
```
(define (fn-for-stn s visited todo acc)
  (fn-for-losn (cons (stn-name s) visited)
    (append (stn-links s) todo)
    (... (stn-name s)
      (stn-platforms s)
      (stn-accessible s)
      (stn-links s)
      acc)))

(define (fn-for-losn visited todo acc)
  (cond [(empty? todo) (... acc)]
    [else
      (fn-for-stn-name (first todo)
        visited (rest todo) todo)
      (... (first todo) acc))]))

(fn-for-stn-name origin empty empty ...)
b) There are two hard questions on the exam - this is one of them. Design a function that consumes the name of an origin station and the name of a destination station. The function must produce true if there is a route from the origin to the destination for which all stations on the route are wheelchair accessible (including the origin and the destination). Otherwise, it must produce false. Note that the functions \texttt{fn-for-stn-name} and \texttt{fn-for-losn} have been written for you. You must complete the design of \texttt{fn-for-stn} on the next page.

\textbf{Note:}

- we re-ordered the functions in the local expression for convenience only, so \texttt{fn-for-losn} now appears before \texttt{fn-for-stn}
- no accumulators, other than those provided, are needed in this question
- use the \texttt{todo} accumulator to ensure that you follow only those routes that can feasibly lead to a solution (hint!)
- all recursive and mutually recursive calls must be in tail position
- you are not required to provide a termination argument

\begin{verbatim}
(defun (accessible-route? origin dest)
  ;; StationName StationName -> boolean
  ;; determines if there is a route from origin to destination for which all
  ;; stations are wheelchair accessible
  (check-expect (accessible-route? "CRW" "LIV") false) ;orig not accessible
  (check-expect (accessible-route? "EUS" "CRW") false) ;dest not accessible
  (check-expect (accessible-route? "EUS" "HOL") false) ;CRW not accessible
  (check-expect (accessible-route? "CRW" "BRM") false) ;no route exists
  (check-expect (accessible-route? "EUS" "LIV") true) ;accessible via MAN

  (define (accessible-route? origin dest)
    ;; Accumulator: visited is (listof StnName)
    ;; Invariant: the names of the stations already visited by fn-for-stn
    ;; Accumulator: todo is (listof StnName)
    ;; Invariant: links to be visited from stations already visited by -stn
    (local [(define (fn-for-stn-name sn visited todo)
              (if (member? sn visited)
                  (fn-for-losn visited todo)
                  (fn-for-stn (get-stn sn) visited todo)))

              (define (fn-for-losn visited todo)
                (cond [[(empty? todo) false]
                           [else
                            (fn-for-stn-name (first todo)
                                             visited
                                             (rest todo))]]))))
\end{verbatim}
(define (fn-for-stn s visited todo)
  (fn-for-stn-name origin empty empty)))
; Defines a sample rail network

(define NETWORK
  (list (make-stn "EUS" 10 true (list "WAT" "CRW" "MAN"))
        (make-stn "CRW" 5 false (list "HOL" "LIV"))
        (make-stn "HOL" 2 true (list "CRW"))
        (make-stn "LIV" 8 true (list "MAN"))
        (make-stn "MAN" 7 true (list "LIV"))
        (make-stn "BRM" 5 false (list "EUS"))
        (make-stn "WAT" 2 true (list "EUS")))))

; Helper functions

; StnName -> Station
; get a station given the station name
(check-expect (get-stn "EUS") (first NETWORK))
(check-expect (get-stn "HOL") (first (rest (rest NETWORK))))

(define (get-stn sn)
  (find (lambda (s) (string=? (stn-name s) sn)) NETWORK))

; (X -> Boolean) (listof X) -> X or false
; find first element of lox for which pred produces true;
; return false if none
(check-expect (find odd? empty) false)
(check-expect (find odd? (list 1 2 3)) 1)
(check-expect (find even? (list 5 6 7)) 6)

(define (find pred lst)
  (cond [(empty? lst) false]
        [else
         (if (pred (first lst))
             (first lst)
             (find pred (rest lst)))]))