```
;;
;; (1)  10 POINTS
;; Consider the following program which consists of
;; two definitions followed by an expression.
;;
(define x 0)
(define y 1)

(+ x
   (local [(define x 2)
           (define y 3)
           (define (foo z)
             (* x y z))]
     (foo x)))

;; (1a) What is the value of the expression?

12     ;2 PTS

;; (1b) Show the first step of evaluating the local (which
;; is the 2nd step of evaluating the + expression). You should
;; show any lifted definitions, as well as showing the
;; renamed body of the local in the expression in which it
;; appears.
;; SUGGESTION: Check your answer by making sure it will
;; evaluate to the same answer you gave in part a.
;;


(define x_0 2)           ; 2 PTS EACH FOR :
(define y_0 3)           ; X, Y AND FOO BEING LIFTED
(define (foo_0 z)        ; AND *ALL* REFERENCES RENAMED
  (* x_0 y_0 z))
(+ 0 (foo_0 x_0))        ; 2 PTS BODY PROPERLY IN CONTEXT



;; ----------------

;;
;; (2)  40 POINTS
;; In this question you will be working with the following
;; structure and data definition:
;;

(define-struct node (value l r))
;;      /--------------------------------      2 PTS FOR
;; A BTree is one of:                |    |      EACH CORRECT
;;   - false                         |    |      LABELED LINE
;;   -(make-node v l r) where        |A   |B
;;       v is a Number               |    |
;;       l is a BTree----------------|    |
;;       r is a BTree---------------------|
;;

;;
;; (2a) Show two examples of BTrees. We are not asking you
;; for test cases here, just two expressions that produce
;; valid BTrees.
;;

false                                        ; 2 PTS FOR
(make-node 1 false false)                    ; EACH CORRECT
(make-node 1 (make-node 2 false false) false)  ; EXAMPLE (4 MAX)
```

```
;; (6 pts)
;; (2b) The BTree data definition is (circle all that apply):
;;
;;   an enumeration
;;  [an itemization]                               ; 2 PTS
;;   an interval                                   ; FOR EACH
;;  [a data definition involving multiple cases]  ; CORRECT
;;  [a self-referential data definition]          ; CIRCLE
;;   mutually-referential data definitions        ; -2 INCORRECT
;;

;; (4 pts see above)
;; (2c) IF you decided that the BTree data definition was self-
;; or mutually-referential, draw an arrow on the data definition
;; from each such reference back to the data definition. Label
;; your arrows (a), (b), etc.
;;

;; (10 pts)
;; (2d) Write the template for a function operating on BTrees.
;; If you drew arrows on your data definition then mark
;; the natural recursions in your template with the corresponding
;; letter (a), (b), etc.
;;
;;
;;
(define (fun bt)
  (cond [(false? bt) ...]                  ;<-- 2 PROPER BASE CASE QUESTION
        [else (... (node-value bt)         ;<-- 2 SELECTOR
                   (fun (node-l bt))    ;A ;<-- 1,1,1 NAT RECUR, SELECTOR, LABEL
                   (fun (node-r bt)))]));B ;<--     "


;; (18 points)
;; (2e) Design the function bt-sum which consumes a BTree and
;; returns the sum of all the node-values in the tree.
;;

;; bt-sum: BTree -> Number            ;3 PTS
;; sum all the node-values in bt      ;3 PTS


(check-expect (bt-sum false) 0)         ;3 PTS FOR BASE TEST 1ST
(check-expect (bt-sum (make-node 2      ;3 PTS FOR >=2 TESTS
                                 false
                                 false))
              2)
(check-expect (bt-sum (make-node 1      ;3 PTS FOR TEST THAT COVERS
                                        ;  BOTH BRANCHES
                                 (make-node 2 false false)
                                 (make-node 3 false false)))
              6)

;; DON'T DOUBLE-PENALIZE FOR TEMPLATE PROBLEMS
(define (bt-sum bt)
  (cond [(false? bt) 0]                 ; <<-- 3 PTS FOR 0 and +
        [else (+ (node-value bt)
                 (bt-sum (node-l bt))
                 (bt-sum (node-r bt)))]))



;; ----------------

;;
;; (3) 20 POINTS
;; In this problem you will be working with the following
;; structure and data definitions.
;;
;; !!!! NOTE THAT IN THESE SOLUTIONS NODE IS CALLED NODE2 !!!!
;; !!!! TO AVOID NAME CONFLICTS W/ EARLIER PROBLEM        !!!!
(define-struct node2 (left right))
;; Tree is one of:
;;  - Number
;;  - (make-node Tree Tree)
;;
;; Path is one of:
;;  - empty
;;  - (cons "L" Path)
;;  - (cons "R" Path)
;;

;;
;; (3a) Give two examples of a Tree. We are not asking you
;; for test cases here, just two expressions that produce
;; valid Trees.
;;
1                                   ;2 PTS EACH, MAX 4
(make-node2 (make-node2 1 2) 3)



;;
;; (3b) Give two examples of a Path.
;;
empty                               ;2 PTS EACH, MAX 4
'("L" "R")


;;
;; (3c) Show the template for a function that consumes a Tree and
;; a Path.
;;

;;
;; SHOULD BE 6 CASES
;; 2 PTS FOR EACH:
;;   1 FOR CORRECT QUESTION (AND .. ..)
;;   1 FOR CORRECT ANSWER
;;
;; OK TO COMBINE (CONS "L" ..) AND (CONS "R" ..) INTO SINGLE
;; CASE IF ANSWER PART OF TEMPLATE USES FIRST
;;
;; HOLD BACK PTS AS FOLLOWS:
;;   -2 NESTED CONDS INSTEAD OF ONE COND W/ ANDS
;;   -1 SINGLE COND BUT USES ELSE
;;

(define (fn tree path)
  ;; there are 6 cases (2 tree) * (3 path)
  (cond [(and (number? tree) (empty? path))               (... tree)]
        [(and (number? tree) (string=? (first path) "L")) (... tree)]
        [(and (number? tree) (string=? (first path) "R")) (... tree)]
        [(and (node2?   tree) (empty? path))
         (... (node2-left tree)
              (node2-right tree))]
        [(and (node2?   tree) (string=? (first path) "L"))
         (... (fn (node2-left tree) (rest path))
              (fn (node2-right tree) (rest path)))]
        [(and (node2?   tree) (string=? (first path) "R"))
         (... (fn (node2-left tree) (rest path))
              (fn (node2-right tree) (rest path)))]))
```

```
;; ----------------

;;
;; (4) 15 POINTS
;; Consider the following only-as and above functions:
;;

;; only-as: (listof String) -> (listof String)
;; keep only the "a"s from los
(check-expect (only-as empty) empty)
(check-expect (only-as '("a" "b" "a")) '("a" "a"))
#;
(define (only-as los)
  (cond [(empty? los) empty]
        [else (cond [(string=? (first los) "a")
                     (cons (first los)
                           (only-as (rest los)))]
                    [else (only-as (rest los))])]))

;; above: Number (listof Number) -> (listof Number)
;; keep only those numbers in lon above n
(check-expect (above 2 empty) empty)
(check-expect (above 2 '(1 2 3 4)) '(3 4))
#;
(define (above n lon)
  (cond [(empty? lon) empty]
        [else (cond [(> (first lon) n)
                     (cons (first lon)
                           (above n (rest lon)))]
                    [else (above n (rest lon))])]))


;;
;; (4a) Use functional abstraction to write the definition
;; of an abstraction of these two functions. Also include
;; new definitions of only-as and above that use the
;; new function.
;;
;; Extra credit if your abstract function accepts 2 arguments
;; instead of 3. But, a SUGGESTION, write the 3 argument version
;; first and then the 2 argument version.
;;

;; REGULAR VERSION                  ;; 1 PT FOR EACH OF THE 8
(define (filter3 pred v los)        ;; PLACES THAT SHOULD NOW BE
  (cond [(empty? los) empty]        ;; EITHER PRED OR V
        [else (cond [(pred (first los) v)
                     (cons (first los)
                           (filter3 pred v (rest los)))]
                    [else (filter3 pred v (rest los))])]))

(define (only-as los) (filter3 string=? "a" los)) ;1 PT FOR EACH
(define (above n lon) (filter3 >        n   lon)) ; CORRECT REIMPL

;!!! extra credit version
;     IF CORRECT EXCEPT FOR PARENS AND OTHER TRUE DETAILS 10 PTS
;     AND DON'T COUNT THE 8 FROM ABOVE
(define (filter2 pred los)
  (cond [(empty? los) empty]
        [else (cond [(pred (first los))
                     (cons (first los)
                           (filter2 pred (rest los)))]
                    [else (filter2 pred (rest los))])]))

;    IF REIMPLS ARE CORRECT EXCEPT FOR PARENS AND OTHER TRUE
;    DETAILS 2 PTS EACH AND DON'T COUNT THE 2 FROM ABOVE

(define (only-as2 los)
  (local [(define (=a? s) (string=? s "a"))]
    (filter2 =a? los)))

(define (above2 n lon)
  (local [(define (>n m) (> m n))]
    (filter2 >n lon)))

;;
;; (4b) What is the contract of your abstract function?
;; If you can make it more abstract do so.
;;

;;       ANY OF THESE ARE FINE FOR 4 POINTS
;;
;; (X X -> Boolean) X (listof X) -> (listof X)
;; (X Y -> Boolean) Y (listof X) -> (listof X)
;; (X -> Boolean) (listof X) -> (listof X)

;;
;; (4c) What is the usual name of the 2 argument version of
;; this abstract function in ISL and Scheme?
;;

; FILTER      1S POINTS

;-----------------

;;
;; (5) 15 POINTS
;; Consider the following structures and data definitions.
;;

(define-struct foo (a))
(define-struct bar (b))
;; A Snarfle is one of:
;;   - a Number
;;   - a (make-foo w) where w is a Whatzzle
;;
;; A Whatzzle is one of:
;;   - a String
;;   - a (make-bar s) where s is a Snarfle


;;
;; (5a) Show THREE examples of a Snarfle:
;;
;                        1 PT EACH UP TO 3
1
(make-foo "s")
(make-foo (make-bar 2))


;;
;; (5b) Show the template for function(s) operating on a
;; Snarfle:
;;
;;        FOR EACH OF 2 FUNCTIONS, GIVE POINTS AS FOLLOWS:
;;           1 - EXISTENCE OF FUNCTION
;;           1 - 2 CASES
;;           1 - PROPER BASE QUESTION
;;           1 - PROPER ANSWER TEMPLATE
;;           1 - ELSE CLAUSE
;;           1 - W/ PROPER NATURAL RECURSION W/SELECTOR
;;
```

```
(define (snarfle-fun s)
  (cond [(number? s) (... s)]
        [(foo? s)    (whatzzle-fun (foo-a s))]))

(define (whatzzle-fun w)
  (cond [(string? w) (... w)]
        [(foo? w)    (snarfle-fun (bar-b w))]))


;;
;; (EXTRA CREDIT)
;; Define the function(s) snarfle-contents that consumes a
;; Snarfle and returns (listof Number/String) that contains
;; the numbers/strings reachable from the Snarfle.
;;

;;!!!
;; snarfle-contents: Snarfle -> (listof (listof Number) (listof String))
;; whatzzle-contents
;;
;;     1 POINT EACH FOR THREE TESTS (BASE AND +1, +2 CHAINS)
;;
(check-expect (snarfle-contents  1)   (list 1))
(check-expect (whatzzle-contents "a") (list "a"))
(check-expect (snarfle-contents
               (make-foo "a"))
              (list "a"))
(check-expect (snarfle-contents
               (make-foo (make-bar (make-foo "b"))))
              (list "b"))

;;
;;       5 POINTS IF CORRECT
;;
(define (snarfle-contents s)
  (cond [(number? s) (list s)]
        [(foo? s)    (whatzzle-contents (foo-a s))]))


(define (whatzzle-contents w)
  (cond [(string? w) (list w)]
        [(bar? w)    (snarfle-contents (bar-b w))]))

;;
;; What strikes you as strange about these data definitions?
;; (Aside from their

;;  2 POINTS FOR
;;
;; THEY ARE MUTUALLY RECURSIVE, SO THEY CAN BUILD ARBITRARY
;; SIZE STRUCTRES, BUT NO MATTER HOW BIG THEY GET THEY ONLY
;; STORE ONE STRING OR NUMBER
;;
```