Important notes about this examination

1. You have 90 minutes to write this examination.
2. This exam will be graded significantly on how well you follow the design recipes. You have been given a copy of the Recipe Summary and Template Rules. Use them!
3. Put away your books, notebooks, cell phones... everything but pens, pencils, erasers and this exam.
4. There are blank pages at the end of the exam you can use for extra space. When you need extra space, mark that clearly both at the original problem and on the extra sheet.

Rules Governing Formal Examinations

1. Each candidate must be prepared to produce, upon request, a UBCcard for identification.
2. Candidates are not permitted to ask questions of the invigilators, except in cases of supposed errors or ambiguities in examination questions.
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.

<table>
<thead>
<tr>
<th>Question</th>
<th>Mark</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1 - (16 points)

The following program has a function definition followed by a marked expression.

```
(define (f a b)
  (local [(define (g b c) (* a b c))]
    (g 3 4)))

(f 1 2) ;<<< MARKED EXPRESSION
```

In this problem you need to show how the evaluation of the marked expression proceeds over time. You must show the state of the evaluation at three points in time. You may use the scratch sheets if you need to, but be sure to show the entire current state of the execution at each point in time, and be sure not to include extraneous expressions in that state. Be sure to show any lifted definitions at the appropriate points.

(A) Right after the call to \texttt{f1} has been replaced by the body of \texttt{f1}. In other words, right when the evaluation of the \texttt{local} begins.

(B) Right after renaming and lifting, in other words right when the body of the \texttt{local} is about to be evaluated.

(C) Right when the evaluation of the * expression is about to begin.
Problem 2 - (22 points)

For this problem do not use the (listof Tree) type, use ListOfTree instead. Also do not use any built-in abstract list functions.

Consider the following partial data definitions:

```
(define-struct node (v subs))
;; Tree is (make-node Integer ListOfTree)
;; Interp. an arbitrary arity tree
;; nodes contain a single integer and a list of children

;; ListOfTree is one of:
;; - empty
;; - (cons Tree ListOfTree)
;; interp. a list of trees
```

(A) Neatly annotate the type comments above by drawing a line from each type reference to the corresponding type definition. All your arrows should have the 'pointy-end' at one of the names that appear right before 'is'. Neatly label each line with one of MR, SR or R, depending on whether the reference is a mutual reference, self reference or an ordinary reference.

(B) Write 2 examples for Tree.
(C) Write the template for a function operating on Tree. Include any additional template functions the rules require.

(D) In such a tree a node with no children is called a leaf node.

Design a function to count the number of leaf nodes in such a tree. You may need more examples than the two above, feel free to define them as constants if you want to.
Problem 3 - (22 points)

Start with the following two function designs and design an abstract function from them. Also redefine the two functions as 'one-liners' using the abstract function. For the redefinitions you just need to write the function definition, you do not need to re-write the signature, purpose or tests. Please write everything you would like to have graded on the next page, which is blank.

;;; (listof Number) -> Number
;;; produce first positive number in lon; 0 if none is found
(check-expect (f1 empty) 0)
(check-expect (f1 (list -1 2 3)) 2)

(define (f1 lon)
  (cond [(empty? lon) 0]
        [else
         (if (positive? (first lon))
             (first lon)
             (f1 (rest lon)))]))

;;; (listof Image) -> Image
;;; produce first square image in loi; false if none is found
(check-expect (f2 empty) false)
(check-expect (f2 (list (rectangle 10 20 "solid" "red")
                      (square 10 "solid" "red")))
              (square 10 "solid" "red"))

(define (f2 is)
  (cond [(empty? is) false]
        [else
         (if (square? (first is))
             (first is)
             (f2 (rest is)))]))
Problem 4 - (16 points)

Dr Racket's 2htdp/image library includes a function called rotate, described as follows:

;; Integer Image -> Image
;; rotate image by specified number of degrees

Sally has a list of images to rotate, all by the same number of degrees. Design a function that consumes Integer and (listof Image) and rotates all the images. Your function should use map. In your examples you may assume that the constants I1, I2 and I3 are defined as images. The signature and purpose of map is:

;; (X -> Y) (listof X) -> (listof Y)
;; (map f (list x1 x2 ...)) produces (list (f x1) (f x1) ...)

Problem 5 - (24 points)

Consider the following two data definitions:

```
(define-struct node (k v l r))
;; BinaryTree is one of:
;;  - false
;;  - (make-node Natural String BinaryTree BinaryTree)
;; interp.
;;  a binary tree, each node has a key, value and l/r children
(define BT1 false)
(define BT2 (make-node 1 "a" false false))
(define BT3 (make-node 4 "d"
    (make-node 2 "b"
        (make-node 1 "a" false false)
        (make-node 3 "c" false false)
    )
    (make-node 7 "g" false false)))
```

```
;; Path is one of:
;;  - empty
;;  - (cons "L" Path)
;;  - (cons "R" Path)
;; interp.
;;  A sequence of left and right 'turns' down through a BinaryTree
;;  (list "L" "R" "R") means take the left child of the root, then
;;  the right child of that node, and the right child again.
;;  empty means you have arrived at the destination.
(define P1 empty)
(define P2 (list "L" "R"))
```

Design the function **has-path?** that consumes **BinaryTree** and **Path.** The function should produce **true** if following the path through the tree leads to a node, if the path leads to **false,** or runs into **false** before reaching the end of the path the function should produce **false.**

Clearly this is a function operating on two complex data. Your answer must include a cross product of type comments table showing what to do for each cell in the table.

You should simplify your function as much as you can using the table and the order of the **cond** clauses. Make it clear by annotating the table how you arrived at table-based simplifications.

The ideal solution has **four** **cond** clauses, but more partial credit will be awarded to correct solutions with more clauses than incorrect solutions with fewer clauses.