```
(require 2htdp/image)
(require spd/tags)
```

```
Problem 1:

This is a 3 part problem, with all parts based on the following partial
data definitions.  We suggest you start by reading them carefully.  Do
not let yourself be confused by the non standard names.  Instead focus on
the structure of the type commnents.
```

```
(define-struct foo (a b))
;; Foo is (make-foo Number Bars)



(define-struct bar (x y z))
;; Bar is (make-bar String String Foos)


;; Foos is one of:

;;  - empty

;;  - (cons Foo Foos)


;; Bars is one of:

;;  - (cons Bar empty)

;;  - (cons Bar Bars)
```



```
Problem 1A:

Annotate the above type comments with reference arrows, and label each
arrow as to whether it is reference (R), self-reference (SR), or mutual
reference (MR).  This question will be marked very precisely.  To be
correct an arrow must clearly start from a reference and end at type
definition (type name before 'is'). An arrow that is a self-reference must
be labeled as such even if can also participate in a mutual reference cycle.
Extra incorrect arrows will lose marks equivalent to a correct arrow.
So please be sure to get your arrows exactly right.
```

Problem 1B:

Below are templates WITHOUT @dd-template-rules tags. Working FROM the type comments, write correct @dd-template-rules tags before each function definition. Don't rush, and don't work from memory, instead carefully work through the type comment to determine the tag. Note that there are two small bugs in these actual templates, which you will get to in part C, so be sure to work from the type comments not the provided templates.

Problem 1C:

There are two small but significant mistakes in the template functions below. Please neatly fix them. If you need to cross something out, then please cross out the minimum amount you can. Also be sure that if you add anything you write neatly.


```
(define (fn-for-foo f)
  (... (foo-a)
       (fn-for-bars (foo-b f))))


(define (fn-for-bar b)
  (... (bar-x b)
       (bar-y b)
       (fn-for-foos (bar-z b))))



(define (fn-for-foos fs)
  (cond [(empty? fs) (...)]
        [else
         (... (first fs)
              (fn-for-foos (rest fs)))]))



(define (fn-for-bars bs)
  (cond [(rest bs) (... (first bs))]
        [else
         (... (fn-for-bar (first bs))
              (fn-for-bars (rest bs)))]))
```

;
Problem 2

Consider the following function definition. Note that it is presented without the rest of the design elements because this is a question about how the Intermediate Student Language works, not about program design.

```
(define (baz lon)
  (cond [(empty? lon) 0]
        [else
         (local [(define (fn x) (* x (first lon)))]
           (+ (fn 3)
              (baz (rest lon))))]))
```

;
Now consider the expression (baz (list 1 2 3))

We want you to do two things:

(A) What value does the expression produce? This is probably harder than part B. Your answer MUST go in the following box:

;

(B) When the expression is evaluated there will be a number of lifted definitions. Please write the SECOND lifted function definition. We are not asking for a complete step by step evaluation, just the second lifted definition. Don't worry about the exact pattern ISL uses to give names to lifted definitions, just use some reasonable naming convention. Your answer MUST go in the following box:

;

;

Problem 3:

Consider this abstract function:

```
(define (moalt cmb con b lox)
  (cond [(empty? lox) b]
        [else
         (cmb (con (first lox))
              (moalt cmb con b (rest lox)))]))
```

Write the signature for the abstract function in the following box. Your signature must be as abstract as possible, meaning it must use type parameters wherever possible.

;

Now write a SINGLE check-expect that uses the abstract function to add up the lengths of the strings in a list. It is possible to do this in a simple compact way, and that is the answer we are looking for. Put your answer in the following box.

;

```
;
 Problem 4:

 Below is a revised version of part of the solutions to problem set 5.

 The main changes we have made were to begin to take advantage of what you
 have learned since the to make the code more concise. Specifically we
 use the (listof Type) notation to let us get rid of some ListOfType data
 definitions, and as part of that we updated a bunch of function signatures.

 We believe the next step is to change the body of sum-of-areas-of-squares to
 be a composition of 3 built in abstract functions, and that once you do that
 you can shorten the program by deleting code that is no longer needed.

 Working neatly, first rewrite the body of sum-of-areas-of-squares, to be a
 composition of three built-in abstract functions, then delete and/or revise
 the rest of the code accordingly.  You must delete functions that will no
 longer be used. Maximum credit will be given to solutions that clear,
 simple, follow the recipes, and are properly well cleaned up.
```



```
(define I10x20 (rectangle 10 20 "solid" "black"))
(define I20x10 (rectangle 20 10 "solid" "black"))
(define I10x10 (square    10    "solid" "black"))
(define I20x20 (circle    10    "solid" "black")) ;will be a square image

(@HtDF sum-of-areas-of-squares)
;; (listof Image) -> Natural
;; produce the sum of the areas of only those images that are square
(check-expect (sum-of-areas-of-squares empty) 0)
(check-expect (sum-of-areas-of-squares (cons I20x20 empty)) 400)
(check-expect (sum-of-areas-of-squares (cons I10x20
                                             (cons I20x20
                                                   (cons I10x10 empty))))
              (+ (sqr 20) (sqr 10)))

(@template use-abstract-fn fn-composition)

(define (sum-of-areas-of-squares loi)
  (sum (areas (squares-only loi))))


(@HtDF squares-only)
;; (listof Image) -> (listof Image)
;; produce list of only those images that are square
(check-expect (squares-only empty) empty)
(check-expect (squares-only (cons I20x20 empty))
              (cons I20x20 empty))
```

```
(check-expect (squares-only (cons I20x10 empty))
              empty)
(check-expect (squares-only (cons I20x10
                                  (cons I10x10
                                        (cons I20x20 empty))))
              (cons I10x10
                    (cons I20x20 empty)))

(@template (listof Image))

(define (squares-only loi)
  (cond [(empty? loi) empty]
        [else
         (if (square? (first loi))
             (cons (first loi) (squares-only (rest loi)))
             (squares-only (rest loi)))]))


(@HtDF square?)
;; Image -> Boolean
;; produce true if the provided image is square (width = height)
(check-expect (square? I20x10) false)
(check-expect (square? I20x20) true)

(@template Image)

(define (square? img)
  (= (image-width img) (image-height img)))


(@HtDF areas)
;; (listof Image) -> (listof Number)
;; produce a list of areas from each square in provided list
(check-expect (areas empty) empty)
(check-expect (areas (cons I20x20 empty))
              (cons 400 empty))
(check-expect (areas (cons I20x20
                             (cons I10x10 empty)))
              (cons 400 (cons 100 empty)))

(@template (listof Image))

(define (areas loi)
  (cond [(empty? loi) empty]
        [else
         (cons (image-area (first loi))
               (areas (rest loi)))]))
```

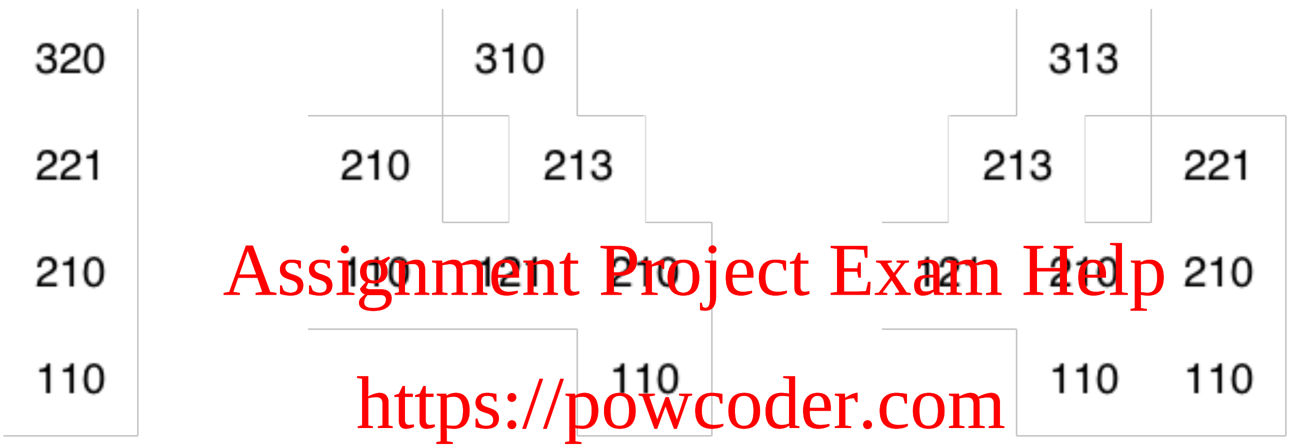
```
(@HtDF image-area)
;; Image -> Number
;; produce area of provided image
(check-expect (image-area I10x20) 200)
(check-expect (image-area I10x10) 100)

;(define (image-area img) 0)

(@template Image)

(define (image-area img)
  (* (image-width img) (image-height img)))

(@HtDF sum)
;; (listof Number) -> (listof Number)
;; produce the sum of all numbers in provided list
(check-expect (sum empty) 0)
(check-expect (sum (cons 100 empty)) 100)
(check-expect (sum (cons 400 (cons 900 (cons 0 empty))))
              (+ 400 900 0))

(@template (listof Number))

(define (sum lon)
  (cond [(empty? lon) 0]
        [else (+ (first lon)
                 (sum (rest lon)))]))
```

;

Problem 5:

In this problem we will revisit the problem of visually representing information about CPSC courses. This time we want to focus on the pre-requisite chains that different courses have. For example, if we look at the pre-requisites of 320, it requires 221, which requires 210, which requires 110. We can render this as shown below on the right.

310 on the other hand requires both 210 and 213 210 requires 110. But 213 requires both 121 and 210. If we look at the figure in the middle we see this structure. Note that 210 and 110 show up more than once. This is on purpose, since we are representing direct prerequisites.



In this two part problem you will first design a data definition to represent this information and then design a function to render it.

Problem 5A:

Design a data definition to represent pre-requisite trees as shown above. One of your data examples MUST BE the tree for CPSC 310. Be sure to follow all steps of the HtDD recipe. Note that:

- if you need a list of type you must write out the type comment as ListOfXxx, you cannot just use (listof Xxx)
- you do not need to include @dd-template rules
- if you define multiple types you may, but are not required to encapsulate the template

;

```
Problem 5B:

Design a function to render pre-requisite trees.  Your rendering can be very
simple.  It should have tree-like structure, but does not need lines, and
does not need a sophisticated layout.  If you solution includes multiple
functions you may but are not required to encapsulate them.  Be sure to use
@HtDF and @template tags.

To make your work easier, you can assume the existence of a helper called
pad, with the following signature and purpose.

;; Number -> Image
;; produce image of number with 15p of padding around the outside of it
```