```
(require spd/tags)



(@problem 1)
;;
;; PUT YOUR CS ID HERE ON THIS LINE: _____________________________
;;



(@problem 2)
#|
Below are partial designs for 4 functions.  You must complete each design
with a function definition using one or more built-in abstract functions. You
must NOT use the (listof X) or the recursive Natural template. You must use
the most appropriate abstract function or functions in each case.
|#

(@htdf xplode)
(@signature String -> (listof 1String))
;; produce a list consisting of the letters of str
(check-expect (xplode "it") (list (string-ith "it" 0) (string-ith "it" 1)))

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

```
(@htdf multiples-of)
(@signature Natural (listof Integer) -> (listof Integer))
;; produce only those numbers n in lon for which (remainder n x) produces 0
(check-expect (multiples-of 3 (list 2 3 5 6)) (list 3 6))

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

```
(@htdf sop)
(@signature Natural (listof Number) -> Number)
;; produce the sum of each element raised to expt p
(check-expect (sop 3 (list 5 9)) (+ (expt 5 3) (expt 9 3)))

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

```
(@htdf plot)
(@signature (listof Posn) Image -> Image)
;; place one DOT at each x,y position on img
(check-expect (plot (list (make-posn 2 3) (make-posn 10 5))
                    (empty-scene 50 50))
              (place-image DOT 2 3 (place-image DOT 10 5 (empty-scene 50 50))))

(@template use-abstract-fn)
;; NOTE THAT THE SELECTORS FOR POSN ARE: posn-x and posn-y
```

```
(@problem 3)
#|
In this problem you are asked to fixed a bug in a program.

When this program is run, the 2nd and 4th test fail, and the last test gets the
following error:

Check failures:
 check-expect encountered the following error instead of the expected value,
       (list (make-thing "B" '())
             (make-thing "B" '())).

   :: append: expects a list, given (make-thing "B" '())


Please fix the program so that it no longer gets this error and all of the
tests run correctly.  Do not rewrite the program substantially, make just the
minimal change required so that it no longer gets this error and all of the
tests run correctly.

|#
(@htdd Thing)

(define-struct thing (nm subs))

;; Thing is (make-thing String (listof Thing))

;; interp. arbitrary-arity tree. Each thing has a name and a list of children.

(define T0 (make-thing "X" empty))

(define T1
  (make-thing "A"
              (list (make-thing "B" empty)
                    (make-thing "C"
                                (list (make-thing "D"
                                                  (list
                                                   (make-thing "A" empty)
                                                   (make-thing "B" empty))))))))
```

```
(@htdf all-with-name)
(@signature Thing String -> (listof Thing))
;; produce all things in tree with given name
(check-expect (all-with-name T0 "Y") (list))
(check-expect (all-with-name T0 "X") (list T0))
(check-expect (all-with-name T1 "Y") (list))
(check-expect (all-with-name T1 "A") (list T1 (make-thing "A" empty)))
(check-expect (all-with-name T1 "B")
              (list (make-thing "B" empty) (make-thing "B" empty)))
(@template Thing (listof Thing) encapsulated)
(define (all-with-name t n)
  (local [(define (fn-for-t t)
            (if (string=? (thing-nm t) n)
                t
                (fn-for-lot (thing-subs t))))
          (define (fn-for-lot lot)
            (cond [(empty? lot) empty]
                  [else
                   (append (fn-for-t (first lot))
                           (fn-for-lot (rest lot)))]))]
    (fn-for-t t)))
```

```
(@problem 4)
#|
The following function uses two accumulators.  The function works correctly,
but the function purpose, tests, as well as the accumulator types and
invariants have deliberately been left out.  You must fill in 5 things:
function purpose, type and invariant for acc1, and type and invariant for
acc2.  Your invariants must be specific to the behaviour of this function.
Saying something generic like "accumulate the data" will receive zero marks.
You do not need to fill in any tests.
|#

(@htdf foo)
(@signature (listof Integer) -> Integer)
;; purpose:




(@template (listof Number) accumulator)
(define (foo lon0)
  ;; acc1 is:
  ;;
  ;;
  ;;
  ;; invariant:
  ;;
  ;;
  ;;
  ;;
  ;; acc2 is:
  ;;
  ;;
  ;;
  ;; invariant:
  ;;
  ;;
  ;;
  ;;
  (local [(define (fn-for-lon lon acc1 acc2)
            (cond [(empty? lon) (- acc1 acc2)]
                  [else
                   (if (odd? (first lon))
                       (fn-for-lon (rest lon) (+ acc1 (first lon)) acc2)
                       (fn-for-lon (rest lon) acc1 (+ acc2 (first lon))))]))]
    (fn-for-lon lon0 0 0)))
```

```
(@problem 5)
#|

Here, with revisions, is part of the Wikipedia for a ternary search tree:

Each node of a ternary search tree stores a single character, references to its
three children conventionally named equal kid, lo kid and hi kid. The lo kid
must be a node whose character value is less than the current node. The hi kid
must be a node whose character is greater than the current node. The equal kid
has the next character in the word. The figure below shows a ternary search
tree with the strings "cute", "cup", "at", "as", "he", "us" and "i":

          c
        / | \
       a  u  h
       |  |  | \
       t  t  e  u
     /  / |   / |
    s  p  e  i  s

As with other trie data structures, each node in a ternary search tree
represents a prefix of the stored strings. All strings in the middle
subtree of a node start with that prefix.  Note that this means that
words only continue along middle branches.  In particular:
  - cute is in the above tree, because starting at c the word continues
    to u, then continues to t and so on.
  - cat is not in the above tree because starting at c the word does
    not continue to a. instead a new word can start at a.


Note that when the above says a character is greater or less than we mean
using the string<? predicate.  (string<? "a" "c") is true for example.

Here is a data definition for such ternary search trees.

|#
(@htdd TSTree)
(define-struct node (s lo eq hi))
;; TSTree is one of:
;;   - false
;;   - (make-node 1String TSTree TSTree TSTree)
;; interp.  A ternary tree.  Note that 1String means a string 1 character long.
;; CONSTRAINT: At any given node, words in which
;;             the next character is < s are in the lo branch
;;             the next character is = s are in the eq branch
;;             the next character is > s are in the hi branch
;;
```

```
(@dd-template-rules one-of
                    atomic-distinct
                    compound
                    self-ref
                    self-ref
                    self-ref)
#;
(define (fn-for-tstree t)
  (cond [(false? t) (...)]
        [else
         (... (node-s t)
              (fn-for-tstree (node-lo t))
              (fn-for-tstree (node-eq t))
              (fn-for-tstree (node-hi t)))]))

;; The following constant represents the tree shown above.
;;
;; WE STRONGLY ADVISE YOU TO TAKE THE TIME RIGHT NOW TO BE SURE YOU UNDERSTAND
;; HOW THIS CONSTANT IS THAT TREE, AND HOW IT REPRESENTS THE WORDS "cute",
;; "cup", "at", "as", "he", "us" and "i". ALSO BE SURE TO UNDERSTAND WHAT
;; WORDS IT DOES NOT REPRESENT.
;;
(define WORDS
  (make-node "c"
             (make-node "a"
                        false
                        (make-node "t"
                                   (make-node "s" false false false)
                                   false
                                   false)
                        false)
             (make-node "u"
                        false
                        (make-node "t"
                                   (make-node "p" false false false)
                                   (make-node "e"  false false false)
                                   false)
                        false)
             (make-node "h"
                        false
                        (make-node "e" false false false)
                        (make-node "u"
                                   (make-node "i" false false false)
                                   (make-node "s" false false false)
                                   false))))


#|
```

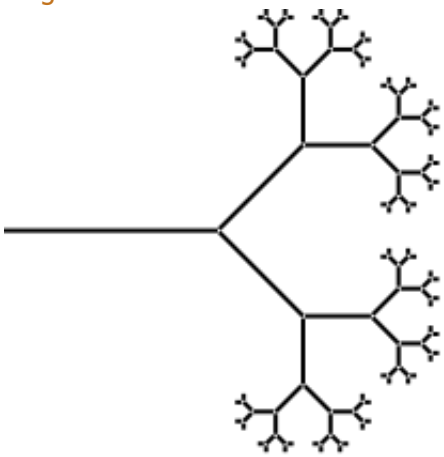
```
Complete the function definition below to produce a working version of this
function.

|#
(@htdf contains?)
(@signature TSTree (listof 1String) -> Boolean)
;; produce true if the tree contains the word formed from the characters

(check-expect (contains? WORDS (list "a" "s"))         true)
(check-expect (contains? WORDS (list "a" "t"))         true)
(check-expect (contains? WORDS (list "c" "u" "p"))     true)
(check-expect (contains? WORDS (list "c" "u" "t" "e")) true)
(check-expect (contains? WORDS (list "h" "e"))         true)
(check-expect (contains? WORDS (list "i"))             true)
(check-expect (contains? WORDS (list "u" "s"))         true)
(check-expect (contains? WORDS (list "c" "u"))         false)
```

```
#|intentionally blank for extra space for problems 5 and 6 if needed|#
```

```
(@problem 6)
#|
Design a function to produce the following fractal.  The function should consume
the length of the longest line as an argument.  Do not worry about the exact
geometry. Just choose reasonable values for cutoff, reduction and rotation.
Remember that (rectangle 30 2 "solid" "black") is a good way to produce a line
2 pixels thick by 30 pixels long.  Be sure to include a 3 part termination
argument.
```



```
|#
```

```
#|intentionally blank for extra space for problem 7 if needed|#
```

```
(@problem 7)
#|
The CS department is so happy to hear about TA scheduling that they have decided
to ask 110 to solve another scheduling problem! They think it will be difficult,
but you have a systematic way to design search functions that will make it
easier for you.

The department wants you to design a function that will consume a list of
classes and a list of classroom slots, and assign each class to a slot if
possible.

The simplest solution to this problem is a structural recursion. NOT a tail
recursion, and we recommend you take that approach.

NOTE that:

 - You MUST use the data definitions we provide below.

 - We are giving you a helper function that computes the wasted space in a
   list of pairs. Use it.

 - We are representing times as integers. So we have 1000, 1100 etc. We
   are ignoring days of the week.

 - Be sure to include a template tag and termination argument.  As well be sure
   to include type and invariant if you use any accumulators.
|#

(@htdd Class)
(define-struct class (num size))
;; Class is (make-class Natural Natural)
;; A class with course number and number of students.

(@htdd Slot)
(define-struct slot (room time size))
;; Slot is (make-slot String Integer Natural)
;; interp. A time slot with the room name, time, and room size

(@htdd Assign)
(define-struct assign (class slot))
;; Assign is (make-assign Class Slot)
;; interp. an Assign represents that the class is scheduled for the room slot
;;         so a list of assigns is a schedule of classes into rooms
(define C110 (make-class 110 280))
(define C121 (make-class 121 220))
(define C210 (make-class 210 270))
(define C221 (make-class 221 180))
(define C213 (make-class 213 180))
```

```
(define SA10 (make-slot "A" 1000 300))
(define SA11 (make-slot "A" 1100 300))
(define SB10 (make-slot "B" 1000 350))
(define SB11 (make-slot "B" 1100 350))
(define SC10 (make-slot "C" 1000 200))
(define SC11 (make-slot "C" 1100 200))



(@htdf wasted-space)
(@signature (listof Assign) -> Natural)
;; produce total wasted space in a schedule
;; CONSTRAINT: in every pair the room is big enough to hold the class
(check-expect (wasted-space empty) 0)
(check-expect (wasted-space (list (make-assign C110 SB11)
                                  (make-assign C121 SA10)))
              (+ (- (slot-size SB11) (class-size C110))
                 (- (slot-size SA10) (class-size C121))))

(@template fn-composition)

(define (wasted-space lop)
  (- (foldl + 0 (map  slot-size (map assign-slot  lop)))
     (foldl + 0 (map class-size (map assign-class lop)))))

#|
Here is the beginning of the function design you must complete.
|#

(@htdf solve-schedule)
(@signature (listof Class) (listof Slot) -> (listof Assign))
;; produce schedule assigning classes to classroom slots

(check-expect (solve-schedule (list C110) empty) false)
(check-expect (solve-schedule (list C121 C210 C221 C213 C110)
                              (list SA10 SB10 SB11 SC10 SC11))
              (list (make-assign C110 SB11)
                    (make-assign C213 SC11)
                    (make-assign C221 SC10)
                    (make-assign C210 SB10)
                    (make-assign C121 SA10)))
```

```
#|intentionally blank for space for problem 8|#
```

```
#|intentionally blank for extra space for problem 8 if needed|#
```

#|intentionally blank for extra space if needed|#

#|intentionally blank for extra space if needed|#