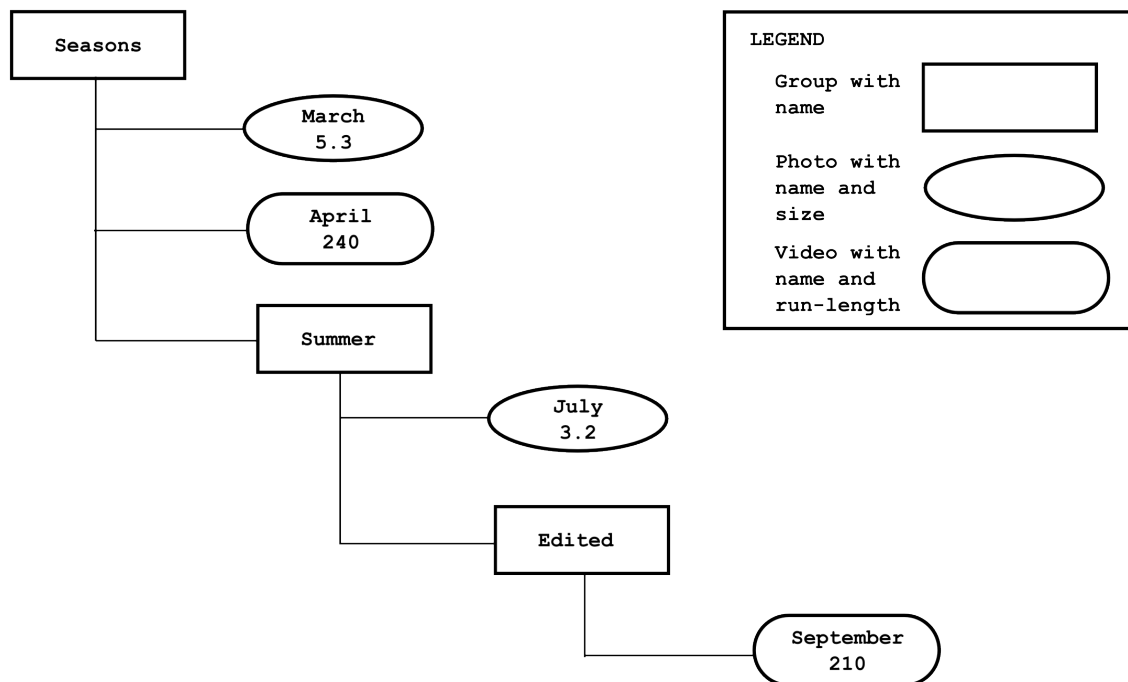


Problem 1: Designing Data Definitions

Suppose you have a collection of digital media. In the context of this question, digital media can be photos or videos. Photos have a name and a size in mega-pixels. Videos have a name and a run-length in seconds. Digital media can be organized into groups that, in addition to containing photos and videos, can also contain other groups of digital media. In addition to associated content, each group has a name. An arbitrary number of photos and videos can be placed in a group. Further, groups can be nested inside other groups to arbitrary depth.

Design data definitions for the information described in the paragraph above. It is not necessary to include the template rules you use to design your templates. Feel free to use the built-in `(listof ...)` type.

Your data definition must include an example that represents the collection of digital media shown in the following picture:



Note that we have provided two pages for your answer to this question, so don't try to squeeze it all onto the next page!

Problem 2: Hand-Stepping Function Calls

Given the following function definition:

```
(define (foo b n)
  (local [(define (bar a) (+ a n))]
    (bar b)))
```

hand-step the evaluation of the following expression:

```
(foo (foo 1 10) 20)
```

Problem 3: Two One-Ofs

Design a function that consumes a `Natural` named `pos` and a `(listof String)`. The function must produce the element at the position specified by `pos`, if such an element exists. Otherwise, the function must produce `false`. Assume that the first position in the list is 0. You must design your function using the two one-ofs recipe. You must therefore include a correctly labelled and simplified cross-product of types comments table. Study the following examples carefully before continuing.

<code>(element-at 0 empty)</code>	<code>produces</code>	<code>false</code>
<code>(element-at 0 (list "a" "b" "c"))</code>	<code>produces</code>	<code>"a"</code>
<code>(element-at 2 (list "a" "b" "c"))</code>	<code>produces</code>	<code>"c"</code>
<code>(element-at 3 (list "a" "b" "c"))</code>	<code>produces</code>	<code>false</code>

Problem 4: Built-In Abstract Functions

For each function described below, identify the built-in abstract function that you would use to design the function, the signature of the function that will be passed to that abstract function and whether or not the function is a closure.

- a)** A function that consumes a natural number n and that produces a list of images of circles of size 1, 2, 3, ..., n . So, for example, given the natural number 3, the function produces a list of images of circles of size 1, 2 and 3.

i) Abstract function:

ii) Signature of function passed as an argument to abstract function:

iii) Is the function specified in ii) a closure?

- b)** A function that consumes a list of strings and a natural number n . The function produces true if all the strings in the list have length at least n .

i) Abstract function:

ii) Signature of function passed as an argument to abstract function:

iii) Is the function specified in ii) a closure?

- c)** A function that consumes a list of images and produces the total area of all the images in the list. The area is computed as simply the image width times the image height.

i) Abstract function:

ii) Signature of function passed as an argument to abstract function:

iii) Is the function specified in ii) a closure?

Problem 5: Designing Abstract Functions

```

(define-struct node (key val l r))
;; A BT (Binary Tree) is one of:
;; - false
;; - (make-node Integer String BT BT)
;; interp. false means empty BT
;;         key is the node key
;;         val is the node val
;;         l and r are left and right subtrees
;; INVARIANT:
;;         the same key never appears twice in the tree
(define BT0 false)
(define BT1 (make-node 10 "a" false false))
(define BT2 (make-node -14 "d" (make-node -8 "r" false false)
                        false))
(define BT3 (make-node 31 "i" BT1 BT2))
(define BT4 (make-node 42 "i"
                      (make-node 7 "w"
                                (make-node 14 "o" false false)
                                false)
                      (make-node 15 "d" false false)))
(define BT5 (make-node 18 "h" BT3 BT4))

#;
(define (fn-for-bt t)
  (cond [(false? t) (...)]
        [else
         (... (node-key t)
              (node-val t)
              (fn-for-bt (node-l t))
              (fn-for-bt (node-r t)))])])

```


- a)** Use the data definition for BT (Binary Tree) above to design an abstract fold function named `fold-bt` for BT. You must use the examples (not necessarily all of them) provided with the data definition for BT to design your tests.

- b)** Use your `fold-bt` function to design a function that consumes a BT and produces the largest key in the tree. You must use the examples (not necessarily all of them) provided with the data definition for BT to design your tests.

Problem 6: Generative Recursion


a) Write the template for generative recursion for a function that consumes a single parameter s .


b) Design a function to produce a circles fractal. In general, a circles fractal of size s is constructed by placing:

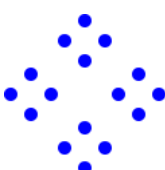
- a circles fractal of size $s/2$
- above
- two circles fractals of size $s/2$ aligned either side of a white circle of size s
- above
- a circles fractal of size $s/2$.

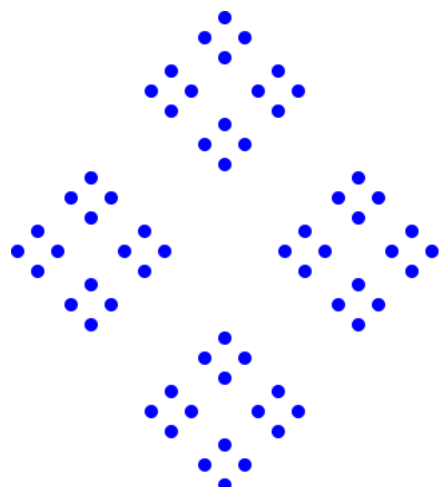
If the size is smaller than 10, we simply draw a blue circle of size 5. Your function design must include a termination argument.

Here are some examples

(circles 5) 

(circles 10) 

(circles 20) 

(circles 40) 

Note that we have provided the check-expects for you - use them! Be sure to use a local expression to avoid re-computation.

```
;; Constants
(define CIRCLE_CUTOFF 10)
(define CIRCLE (circle 5 "solid" "blue"))
(define CIRCLE10 (above CIRCLE
                        (beside CIRCLE
                               (circle 10 "solid" "white")
                               CIRCLE)
                        CIRCLE))

;; Natural -> Image
;; produces an image of the circles fractal of size s
(check-expect (circles 5) CIRCLE)
(check-expect (circles 10) CIRCLE10)
(check-expect (circles 20) (above CIRCLE10
                                   (beside CIRCLE10
                                           (circle 20 "solid" "white")
                                           CIRCLE10)
                                   CIRCLE10))

(define (circles s) CIRCLE) ;stub
```

Problem 7: Designing with Accumulators

Design a function that consumes a list of integer and produces the sum of the unique integers in the list. So, if a list contains an integer more than once, that integer is added to the sum only once. Study the provided check-expects carefully before continuing. Do not use built-in abstract functions.

Hints:

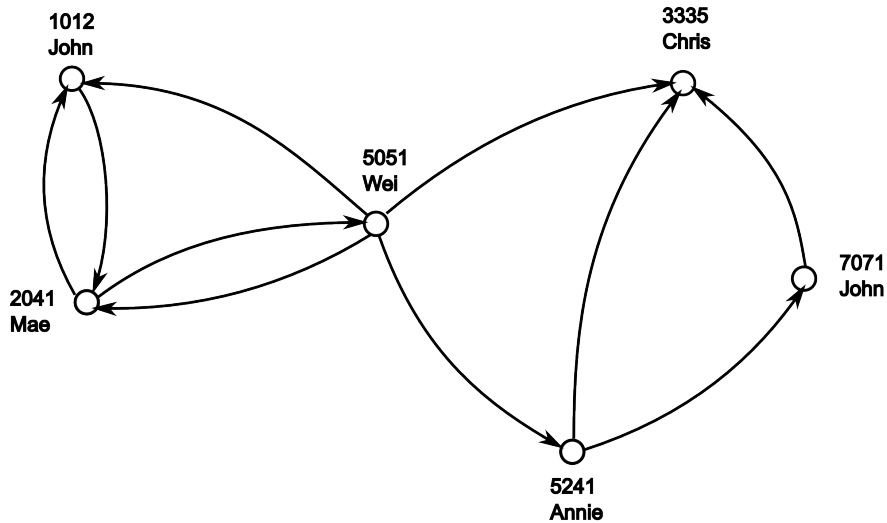
- you will need to introduce an accumulator – the type of the accumulator and an invariant must be provided
- it is not necessary to put the recursive calls into tail position

```
;; (listof Integer) -> Integer
;; produces sum of unique integers in loi
(check-expect (unique-sum (list 3 3 3 3 3)) 3)
(check-expect (unique-sum (list 1 5 2 4 5 2 5)) 12)

(define (unique-sum loi) 0) ;stub
```

Problem 8: Graphs

The data definition for a social network that appears below is the same as the one that was used on Problem Set 11. Each user has a unique ID, a name (that is not necessarily unique in the network) and a list of other users that they are following on that network.



```
;; Data Definitions
```

```

(define-struct user (id name following))
;; User is (make-user Natural String (listof User))
;; a user in a social network with an id, name and list of users
;; they are following

; Note that the following constant represents the network shown
; in the picture above
(define SN (shared
  ((-1012- (make-user 1012 "John" (list -2041-)))
    (-2041- (make-user 2041 "Mae" (list -1012- -5051-)))
    (-5051- (make-user 5051 "Wei" (list -2041- -5241-
                                         -1012- -3335-)))
    (-3335- (make-user 3335 "Chris" empty))
    (-5241- (make-user 5241 "Annie" (list -7071- -3335-)))
    (-7071- (make-user 7071 "John" (list -3335-))))
  (list -3335- -5241- -2041- -5051- -7071- -1012-)))

(define U3335 (first SN))
(define U5241 (first (rest SN)))
(define U2041 (first (rest (rest SN))))
(define U5051 (first (rest (rest (rest SN)))))

```

```
#;
(define (fn-for-user u0)
  (local [;; todo: (listof User), list of users yet to be visited
          ;; by fn-for-user
          ;; visited: (listof Natural), list of ids of users
          ;; already visited
          (define (fn-for-user u todo visited)
            (if (member? (user-id u) visited)
                (fn-for-lou todo visited)
                (fn-for-lou (append (user-following u) todo)
                           (cons (user-id u) visited))))

          (define (fn-for-lou todo visited)
            (cond [(empty? todo) (...)]
                  [else
                   (fn-for-user (first todo) (rest todo) visited)]))]
    (fn-for-user u0 empty empty)))
```