CMSC 15100, Fall 2005

Midterm Exam 1

Name ____

1a	(from 3)	
1b	(from 3)	
1c	(from 4)	
2a	(from 5)	
2b	(from 5)	
2c	(from 5)	
2d	(from 10)	
3a	(from 5)	
3b	(from 5)	
3c	(from 4)	
3d	(from 1)	
3e	(from 15)	
4a	(from 5)	
4b	(from 5)	
4c	(from 10)	
4d	(from 5)	
4e	(from 10)	
Total	(from 100)	

1 [10 total points]. The formula for converting degrees Fahrenheit to degrees Celsius is

$$C = \frac{5}{9}(F - 32)$$

The formula for converting degrees Celsius to degrees Kelvin is

$$K = C + 273$$

1a [3 points]. Fill in the body of f→c below:

```
;; f→c : num → num
;; to convert degrees Fahrenheit to degrees Celsius
;; e.g. (f→c 32) = 0, (f→c −40) = −40
(define (f→c F)
```

Solution

```
(define (f→c F)
  (∗ 5/9 (− F 32)))
```

1b [3 points]. Fill in the body of c→k below:

```
;; c→k : num → num
;; to convert degrees Celsius to degrees Kelvin
;; e.g. (c→k 0) = 273, (c→k −273) = 0
(define (c→k C)
```

Solution

```
(define (c→k C)
  (+ C 273))
```

1c [4 points]. Fill in the body of f→k below:

```
;; f→k : num → num
;; to convert degrees Fahrenheit to degrees Kelvin
;; e.g. (f→k 32) = 273, (f→k −40) = 233
(define (f→k F)
```

Solution

```
(define (f→k F)
  (c→k (f→c F)))
```

2 [25 total points].

```
;; A nonempty-list-of-booleans is:
;; - (cons boolean empty)
;; - (cons boolean nonempty-list-of-booleans)
```

2a [5 points]. Write a template for functions that process nonempty-lists-of-booleans.
Solution

```
(define (fun-for-nelob a-nelob)
  (cond
    [(empty? (rest a-nelob)) ... (first a-nelob) ...]
    [else
     ... (first a-nelob) ...
     ... (fun-for-nelob (rest a-nelob)) ...]))
```

2b [5 points]. Write down the contract, purpose, and header for the function *all-true?* which determines if a given nonempty-list-of-booleans contains only true values.
Solution

```
;; all-true? : nonempty-list-of-booleans → boolean
;; to determine if alob contains only trues
(define (all-true? alob) ...)
```

2c [5 points]. Write down three examples for *all-true?*.
Solution

(*all-true?* (*list true true true*)) = *true*
(*all-true?* (*list true true false true*)) = *false*
(*all-true?* (*list true*)) = *true*

2d [10 points]. Develop the function *all-true?*.
Solution

```
(define (all-true? nelob)
  (cond
    [(empty? (rest nelob)) (first nelob)]
    [else
     (and (first nelob) (all-true? (rest nelob)))]))
```

3 [30 total points]. The function *partial-products : list-of-numbers* $\rightarrow$ *list-of-numbers* takes a list of numbers and returns a list of the same length. The ith number in the result is the product of the first i numbers in the input list — that is, the first element in the result is the same as the first element in the input, the second element is the product of the first two, the third element is the product of the first three, and so on. For instance, it's easy to see how many seconds there are in a minute, hour, day, or year by computing

(*partial-products* (*list* 60 60 24 365)) = (*list* 60 3600 86400 31536000)

3a [5 points]. Write a data definition for lists of numbers.
Solution
a list-of-numbers is:

- *empty*
- (*cons num list-of-numbers*)

3b [5 points]. Write a template for functions that process a list of numbers.
Solution

```
(define (fun-for-lon alon)
  (cond
    [(empty? alon) ...]
    [else
     ... (first alon) ...
     ... (fun-for-lon (rest alon)) ...]))
```

3c [4 points]. Assume you've implemented *partial-products* using the template you wrote down. Your template has some number of recursive calls in it; for each one of those calls, what value will the result of the call have when the function is called with (*list* 4 6 2)[1]?

Solution

There is one recursive call. It will be (*list* 6 12).

3d [1 points]. What will the result of the entire function be when called with (*list* 4 6 2)?

Solution

It will be (*list* 4 24 48).

[1]From tennis: four points to win a game, six games to win a set, two sets to win a match.

3e [15 points]. Develop the function *partial-products*. (Remember that if you develop a helper function you need to state its contract, purpose, and header, and to write examples and their expected answers.)

Solution

```
;; partial-products : list-of-numbers → list-of-numbers
(define (partial-products units)
  (cond
    [(empty? units) empty]
    [else
     (cons (first units) (multiply-all (first units) (partial-products (rest units))))]))

;;multiply-all : num list-of-numbers → list-of-numbers
;; to produce the list consisting of each element of lon multiplied by n
(define (multiply-all n lon)
  (cond
    [(empty? lon) empty]
    [else (cons (* n (first lon)) (multiply-all n (rest lon)))]))
```

4 [35 total points]. A *tri-tree* is a tree where each node holds a number and three sub-trees: a left, a right, and a middle branch. We can represent them as follows:

```
;; a tri-tree is either:
;; - false
;; - (make-node num tri-tree tri-tree tri-tree)
(define-struct node (value l m r))
```

For instance, the following are tri-trees:

```
(define t1 (make-node 5
                (make-node 3 false false false)
                (make-node 4 false false false)
                (make-node 1 false false false)))
(define t2 (make-node 12
              false
              false
              (make-node 1
                (make-node 2 false false false)
                false
                (make-node 3 false false false))))
```

4a [5 points]. Write an example of a tri-tree that contains at least 3 uses of *make-node*.
Solution

```
(define t3
  (make-node 1
    (make-node 2 false (make-node 3 false false false) false)
    false
    false))
```

4b [5 points]. Write a template for functions that process tri-trees.
Solution

```
(define (fun-for-tt a-tt)
  (cond
    [(boolean? a-tt) ...]
    [else
     ... (node-value a-tt) ...
     ... (fun-for-tt (node-l a-tt)) ...
     ... (fun-for-tt (node-m a-tt)) ...
     ... (fun-for-tt (node-r a-tt)) ...]))
```

4c [10 points]. Develop the function *total-value : tri-tree* → *num*, which determines the sum of all values in a given tri-tree. For instance, (*total-value t1*) = 13, (*total-value t2*) = 18.

Solution

```
;; total-value : tri-tree → num
;; to determine the total of all values in tt
;; e.g. (total-value t1) = 13
;; (total-value t2) = 18
;; (total-value t3) = 6
;; (total-value false) = 0
(define (total-value tt)
  (cond
    [(boolean? tt) 0]
    [else
     (+ (node-value tt)
        (total-value (node-l tt))
        (total-value (node-m tt))
        (total-value (node-r tt)))]))
```

4d [5 points]. A *perfect* tri-tree is a tri-tree that additionally has the Perfect Tri-Tree Property: for every node in the tree, the sum of the values in the middle tree is equal to the sum of the values in the left and right trees combined.
A perfect-tri-tree is either

```
– false
– (make-node num
             perfect-tri-tree[left]
             perfect-tri-tree[center]
             perfect-tri-tree[right])
```

INVARIANT: the sum of the values in left plus the sum of the values in right is equal to the sum of the values in center.

For instance, *t1* from the previous problem is a perfect tri-tree but *t2* is not.Write another example of a perfect tri-tree that contains at least 4 *make-node* uses.

Solution

```
(define pt2 (make-node 10
                       (make-node 3 false false false)
                       (make-node 46 false false false)
                       (make-node 43 false false false)))
```

4e [10 points]. Develop the function *p-total-value : perfect-tri-tree* → *num*, which determines the sum of all values in a given perfect tri-tree. For instance, (*b-total-value t1*) = 13. You must make use of perfect tri-tree property in your solution.

Solution

```
;; p-total-value : tri-tree[balanced] → num
;; to determine the sum of all values in the given perfect tri-tree.
;; e.g., (p-total-value false) = 0, (p-total-value t1) = 13, (p-total-value bt2) = 102
(define (p-total-value ptt)
  (cond
    [(boolean? ptt) 0]
    [else
     (+ (node-value ptt)
        (* 2 (p-total-value (node-m ptt))))]))
```