`$Id: cmps112-2019q1-midterm.mm,v 1.114 2019-02-07 12:50:30-08 - - $`

page 1	page 2	page 3	page 4	Total / 42

PLEASE PRINT CLEARLY :

NAME :
CRUZID : `@ucsc.edu`

No books ; No calculator ; No computer ; No email ; No internet ; No notes ; No phone. Do your scratch work elsewhere and enter only your final answer into the spaces provided. Points will be deducted for messy answers. Unreadable answers will be presumed incorrect.

1. What is the output from each of the following when entered at the interactive prompt ? **[4✔]**

Ocaml	`# (+);;`	int -> int -> int = <fun>
Ocaml	`# List.map;;`	('a -> 'b) -> 'a list -> 'b list = <fun>
Ocaml	`# (<);;`	'a -> 'a -> bool = <fun>
Ocaml	`# (+.) 3.;;`	float -> float = <fun>
Scheme	`> (apply * '(1 2 3))`	6
Scheme	`> (map (lambda (x) (* 3 x)) '(1 2 3))`	(3 6 9)
Scheme	`> ((lambda (x y) (cons x y)) 3 '(4))`	(3 4)
Scheme	`> (let ((x 3) (y 4)) (* x y))`	12

2. Code the function **`maximum`** which takes two arguments : a comparison function, and a list. It returns the maximum element in the list if the list is non-empty.

(a) ***Ocaml.*** Return **`None`** if the list is empty. **[3✔]**

```
# maximum;;
- : ('a -> 'a -> bool) -> 'a list -> 'a option = <fun>
# maximum (>) [3;1;4;1;5];;
- : int option = Some 5
# maximum (<) [3.;1.;4.;1.;5.];;
- : float option = Some 1.
# maximum (=) [];;
- : 'a option = None
```

```
let maximum gt list = match list with
  | [] -> None
  | x::xs -> let rec max x xs = match xs with
              | [] -> x
              | y::ys -> max (if gt x y then x else y) ys
    in Some (max x xs)
```

(b) ***Scheme.*** Return **`#f`** if the list is empty. **[3✔]**

```
> maximum
#<procedure:maximum>
> (maximum > '(1 2 3 4 5))
5
> (maximum < '(1.0 2.0 3.0 4.0 5.0))
1.0
> (maximum = '())
#f
```

```
(define (maximum gt list)
  (define (max big listt)
    (if (null? listt) big
      (let ((a (car listt))
            (d (cdr listt)))
           (if (gt big a) (max big d)
                          (max a d)))))
  (if (null? list) #f
    (max (car list) (cdr list))))
```

3. Define the function `sum` without using any higher-order functions.

   (a) ***Ocaml.*** Use the following type : **[2✔]**

   ```
   val sum : float list -> float
   ```

   ```
   let sum list =
     let rec sum' lst acc = match lst with
       | [] -> acc
       | x::xs -> sum' xs (acc +. x)
     in sum' list 0.
   ```

   (b) ***Scheme.*** The result is whatever the `+` function naturally returns. **[2✔]**

   ```
   (define (sum list)
       (define (summ list acc)
           (if (null? list) acc
               (summ (cdr list) (+ (car list) acc))))
       (summ list 0))
   ```

4. Define the function `fold_left`, whose arguments are a folding function, a unit value, and a list. The unit value is used as the leftmost argument to the folding function.

   (a) ***Ocaml.*** Use the following type : **[2✔]**

   ```
   val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
   ```

   ```
   let rec fold_left fn unit list = match list with
     | [] -> unit
     | x::xs -> fold_left fn (fn unit x) xs
   ```

   (b) ***Scheme.*** Arguments are in the same order as the Ocaml question. **[2✔]**

   ```
   (define (fold_left fn unit list)
     (if (null? list) unit
       (fold_left fn (fn unit (car list)) (cdr list))))
   ```

5. Define the function `sum` making use of `fold_left`. The function should satisfy the same requirements as the previous `sum` question, and use the `fold_left` function defined above.

   (a) ***Ocaml.*** Fill in the space. Do not alter anything to the left of the equal (`=`) symbol. **[1✔]**

   ```
   # let sum = fold_left (fun a b -> a +. b) 0;;
   val sum : float list -> float = <fun>
   ```

   (b) ***Scheme.*** **[1✔]**

   ```
   (define (sum list) (fold_left (lambda (a b) (+ a b)) 0 list))
   ```

6. ***Smalltalk.*** Extend class **Array** with method **sum**, which returns the sum of all element of an array. Make no attempt to verify that elements are numbers. **[2✓]**

```
st> a := #(1 2 3 4 5).
(1 2 3 4 5 )
st> a sum.
15
st> a := #().
()
st> a sum.
0
```

```
Array extend [
  sum [
    |sum|
    sum := 0.
    self do: [:n| sum := sum + n].
    ^ sum.
  ]
].
```

7. ***Smalltalk.*** Extend class **Array** with method **reverse**, which reverses the elements of an array. **[3✓]**

```
st> a := #(1 2 3 4 5) copy.
(1 2 3 4 5 )
st> b := #(1 2 3 4) copy.
(1 2 3 4 )
st> a reverse.
(5 4 3 2 1 )
st> b reverse.
(4 3 2 1 )
```

```
Array extend [
  reverse [
    1 to: self size // 2 do: [:i|
      |j t|
      j := self size - i + 1.
      t := self at: i.
      self at: i put: (self at: j).
      self at: j put: t.
    ]
  ]
].
```

8. ***Ocaml.*** Define **eval** so that it takes an expression as defined here. Functions themselves are in the expression. There is no hash table in this question. An example interaction is shown. **[2✓]**

```
# type binfn = float -> float -> float;;
# type expr = Expr of binfn * expr * expr
             | Num of float;;
# eval;;
- : expr -> float = <fun>
# eval (Expr ((+.),
        Expr ((/.), Num 3., Num 4.),
        Expr ((/.), Num 7., Num 8.)));;
- : float = 1.625
```

```
let rec eval expr = match expr with
   | Num value -> value
   | Expr (fn, e1, e2) -> fn (eval e1) (eval e2)
```

9. ***Scheme.*** Define **evalexpr** which takes an expression as an argument. An expression is a number or a list consisting of a symbol representing an operator, followed by zero or more expressions. Assume **fnhash** is a hash table which maps symbols onto functions that may be used to evaluate operators. **Apply** the function to the result of **mapping** **eval** onto the list of arguments. Expressions are nested arbitrarily deeply. Assume all expresions are valid (no need for error checking). The result must always be a real or complex number. **[3✓]**

```
> (evalexpr '(+ (* 2 3) (* 4 5)))
26.0
> (evalexpr 3)
3.0
> (evalexpr '(* (+ 8 3) (+ (* 2 9) 6)))
264.0
> (evalexpr '(/ 4 0))
+inf.0
```

```
(define (evalexpr expr)
     (if (number? expr) (+ expr 0.0)
        (apply (hash-ref fnhash (car expr))
            (map evalexpr (cdr expr)))))
```

Multiple choice. To the ***left*** of each question, write the letter that indicates your answer. Write ***Z*** if you don't want to risk a wrong answer. Wrong answers are worth negative points. **[12✓]**

number of correct answers		× 1 =	= a
number of wrong answers		× ½ =	= b
number of missing answers		× 0 =	0
column total $c = \max(a - b, 0)$	12		= c

1. What kind of type checking is used in Ocaml ?
   (A) strong and dynamic
   [✓ (B) strong and static]
   (C) weak and dynamic
   (D) weak and static

2. What kind of type checking is used in Scheme ?
   [✓ (A) strong and dynamic]
   (B) strong and static
   (C) weak and dynamic
   (D) weak and static

3. ***Ocaml.*** `3 + 4` means :
   (A) `(+ 3 4)`
   (B) `(+) (3, 4)`
   [✓ (C) `(+) 3 4`]
   (D) `(apply + '(3 4))`

4. ***Ocaml.*** What is the type of `(+)` ?
   (A) `int * int * int`
   (B) `int * int -> int`
   (C) `int -> int * int`
   [✓ (D) `int -> int -> int`]

5. What takes a function and a list, applies that function to every element of the list, and returns a resulting list of the same length as the argument list ?
   (A) apply
   (B) fold_left
   (C) filter
   [✓ (D) map]

6. Assuming `x` is a proper list, what will return a list equivalent to `x` ?
   (A) `(car (cdr x) (cons x))`
   (B) `(car (cons x) (cdr x))`
   (C) `(cdr (car x) (cons x))`
   (D) `(cdr (cons x) (car x))`
   [✓ (E) `(cons (car x) (cdr x))`]
   (F) `(cons (cdr x) (car x))`

7. What is `(cadr '((1 2) (3 4)))`
   (A) `()`
   (B) `(2)`
   [✓ (C) `(3 4)`]
   (D) `1`

8. How much stack space (not time) is taken by the following function ?

```
(define (f n)
   (cond ((< n 0) #f)
         ((= n 0) 0)
         ((= n 1) 1)
         (else (+ (f (- n 1))
                  (f (- n 2))))))
```

   (A) $O(1)$
   [✓ (B) $O(n)$]
   (C) $O(n^2)$
   (D) $O(2^n)$

9. `(cddr '((1 2) (3 4)))`
   [✓ (A) `()`]
   (B) `(2)`
   (C) `(3 4)`
   (D) `1`

10. What will create the list `(1 2)` ?
   (A) `(cons (car 1) (cdr 2))`
   [✓ (B) `(cons 1 (cons 2 '()))`]
   (C) `(cons 1 (cons 2 ()))`
   (D) `(cons 1 2)`

11. The immediate ancestor of Scheme :
   (A) Fortran
   [✓ (B) Lisp]
   (C) ML
   (D) λ-calculus

12. ***Smalltalk.*** What is the order of evaluation of :
   `a*b+c*d`
   [✓ (A) `((a*b)+c)*d`]
   (B) `(a*(b+c))*d`
   (C) `(a*b)+(c*d)`
   (D) `a*((b+c)*d)`
   (E) `a*(b+(c*d))`

```
def duck_typing
  "quack!"
end
```