`$Id: cse112-2020q1-midterm.mm,v 1.163 2020-02-10 15:03:22-08 - - $`

page 1	page 2	page 3	page 4	Total / 42

Last Name :
First Name :
CruzID : @ucsc.edu

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

OCaml. For all questions asking for code in OCaml, you need not explicitly specify the type of a function's argument or the type of its result. Obviously, OCaml uses type inference.

1. Define the function **sum** which will find the sum of all numbers in a list. Use as little function call stack space as possible. Do not use a higher order function.

   (a) ***Scheme.*** **[2✔]**

```
(define (sum list)          ... deduct 1 point if no tail recursion
   (define (summ list n)
      (if (null? list) n
         (summ (cdr list) (+ n (car list)))))
   (summ list 0))
```

   (b) ***OCaml.*** **[2✔]**

```
# sum;;
- : int list -> int = <fun>
```

```
let sum list =              ... deduct 1 point if no tail recursion
   let rec sum' list' n = match list' with
      | [] -> n
      | x::xs -> sum' xs (x + n)
   int sum' list 0
```

2. Define the fold left function. Use as little function call stack space as possible.

   (a) ***Scheme.*** The order of arguments are the same as in OCaml. **[2✔]**

```
> foldl
#<procedure:foldl>
```

```
(define (foldl f z list)
   (if (null? list) z
      (foldl f (f z (car list)) (cdr list))))
```

   (b) ***OCaml.*** **[2✔]**

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

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

3. Define the function **sum** using the fold left higher order function.

   (a) ***Scheme.*** Use **foldl**. **[1✔]**

```
(define (sum list) (foldl + 0 list))
```

   (b) ***OCaml.*** Use **List.fold_left**. **[1✔]**

```
let sum list = fold_left (+) 0 list
let sum = fold_left (+) 0              ... either answer is OK
```

4. ***Scheme.*** Define the function `evalexpr` which takes as an argument either a number or an expression. A number is always returned as real. An expression is a list of length 3, where the `car` is a binary function, and the `cdr` is a list of two expressions. See the Scheme interaction at the left. Note that there is no hash table for functions. The actual function's value is in the list in the function position. Note the quasiquote and unquote. **[2✔]**

```
> (evalexpr `(,+ (,* 2 3) (,* 4 5)))
26.0
> (evalexpr 3)
3.0
> (evalexpr `(,* (,+ 8 3) (,+ (,* 2 9) 6)))
264.0
```

```
(define (evalexpr expr)
    (if (number? expr) (+ expr 0.0)
       (apply (car expr)
          (map evalexpr (cdr expr)))))
```

5. ***Smalltalk.*** Extend class `Array` with a method `find`, which has a single argument. It returns the position (index) of the first occurrence of that argument in the array. It returns `nil` if not found. **[2✔]**

```
st> a := #( 3 1 4 1 5 9 2 6).
(3 1 4 1 5 9 2 6 )
st> a find: 9.
6
st> a find: 10.
nil
```

```
Array extend [
   find: value [
```

```
Array extend [
  find: value [
    1 to: self size do: [:i |
      (self at: i) = value ifTrue: [^i].
    ].
    ^nil
  ]
].
```

6. ***Smalltalk.*** Define a block closure `gcd` with two arguments that uses Euclid's algorithm to find the greatest common divisor of two integers.. It might be called with the expression `(gcd value: 20 value: 5)`. The equivalent function in C is shown here. Your algorithm may assume that both $x > 0$ and $y > 0$. **[2✔]**

```
int gcd (int x, int y) {
   while (x != y) {
      if (x > y) x = x - y;
            else y = y - x;
   }
   return x;
}
```

```
gcd := [:x :y |
```

```
gcd := [:x :y |
  |a b|
  a := x.  b := y.
  [a ~= b] whileTrue: [
    a > b ifTrue: [a := a - b] ifFalse: [b := b - a]
  ].
  ^a
].
```

7. ***Smalltalk.*** Extend class `Array` with a method `fold:unit:`, which takes as arguments a block which combines two elements into a single element, and a unit which is the identity operation for the the block. The result is the array folded into a single element using the unit as a starter element. Hint: Think about fold left in a functional language. **[2✔]**

```
st> a := #(1 2 3 4 5).
(1 2 3 4 5 )
st> sum := [:x :y | x + y].
a BlockClosure
st> s := a fold: sum unit: 0.
15
st> prod := [:x :y | x * y].
a BlockClosure
st> p := a fold: prod unit: 1.
120
```

```
Array extend [
   fold: block unit: id [
      |result|
      result := id.
```

```
Array extend [
  fold: block unit: id [
    |result|
    result := id.
    self do: [:item | result := block value: result value: item.].
    ^ result.
  ]
].
```

8. ***Smalltalk.*** Extend class `Array` with the keyword method `innerprod` which computes the inner product of itself with its argument. Note that an error is thrown if the sizes of the arrays are different. The formula for an inner product is shown at the left. Finish the method. **[2✔]**

$$p = \sum_i a_i b_i$$

```
Array extend [
   innerprod: other [
      self size = other size
      ifFalse: [^ self error: 'innerprod different sizes']
      ifTrue: [
```

```
Array extend [
  innerprod: other [
    self size = other size
    ifFalse: [ ^ self error: 'innerprod different sizes' ]
    ifTrue: [
      |sum|  sum := 0.
      1 to: self size do: [:i|
        sum := sum + (self at: i) * (other at: i)
      ].
      ^ sum
    ]
  ]
]
```

9. ***OCaml.*** Assume the declarations in the top of the table at the left. Define the function **`eval`** which takes an **`expr`** as an argument and returns the float value. Output should match the bottom of the table. Note that there are no hash tables in this question. The functions are actually present as arguments to the **`Expr`** constructor. **[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 (binfn, e1, e2) -> binfn (eval e1) (eval e2);;
```

10. Write a function to reverse a list.

(a) ***Scheme.*** **[2✔]**

```
> reverse
#<procedure:reverse>
> (reverse '(1 2 3 4 5))
(5 4 3 2 1)
> (reverse '())
()
```

```
(define (reverse list)        ... deduct 1 point if no tail recursion
    (define (rev in out)
        (if (null? in) out
            (rev (cdr in) (cons (car in) out))))
    (rev list '()))
```

(b) ***OCaml.*** **[2✔]**

```
# reverse;;
- : 'a list -> 'a list = <fun>
# reverse [1;2;3;4;5];;
- : int list = [5; 4; 3; 2; 1]
# reverse [];;
- : 'a list = []
```

```
let reverse list =           ... deduct 1 point if no tail recursion
   let rec rev inl outl = match inl with
      | [] -> outl
      | x::xs -> rev xs (x::outl)
   in rev list []
```

(c) ***C/C++.*** Do not use **`malloc`**(3) or **`free`**(3). Do not change any value field. Manipulate the pointer fields. After modifying the list, return a pointer to the new head of the list. Use a loop, not recursion. **[2✔]**

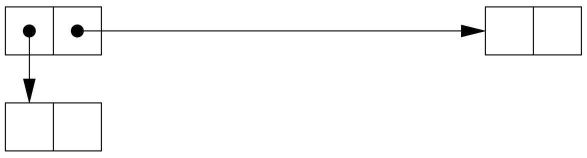
```
struct node {
   int value;
   struct node* link;
};
```

```
struct node* reverse (struct node* head) {
```

```
struct node* reverse (struct node* head) {
   node* list = NULL;
   while (head != NULL) {
      struct node* t = head;
      head = head->link;
      t->link = list;
      list = t;
   }
   return list;
```

11. ***Scheme.*** Draw a picture of the following expression. For each cell, draw a small box with two parts. The first few cells are drawn for you. Fill it in as appropriate and draw more boxes. Write the Greek letter phi (φ) in each part of a cons cell that has a null pointer. Hint: Since this is a proper list, only some **`cdr`**s have null pointers. For each part of a cell containing a symbol, write the appropriate letter inside that part of the cell. If a part of a cell contains a pointer, draw an arrow from that cell to the one it points at. If the arrow is in the **`cdr`** part of the cell draw it horizontally to the right. If it is in the **`car`** part of the cell, draw it pointing downward. **[2✔]**

```
'((a (b)) (c d e) ((e) f))
```



```
[.|.]---------->[.|.]------------------>[.|*]
 |              |                       |
 V              V                       V
[a|.]-->[.|*]  [c|.]-->[d|.]-->[e|*]  [.|.]-->[f|*]
         |                             |
         V                             V
        [b|*]                         [e|*]
```

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 function is equivalent to a loop in an imperative language ?
   (A) head recursive
   (B) mutually recursive
   (C) self recursive
   [✓ (D) tail recursive]

2. ***OCaml.*** Fill in the blank.
```
# ______ ((+)1) [1;2;3];;
- : int list = [2; 3; 4]
```
   (A) `List.fold_left`
   (B) `List.fold_right`
   [✓ (C) `List.map`]
   (D) `List.tl`

3. ***OCaml.*** Fill in the blank. Use as little function call stack space as possible.
```
# ______ (+) 0 [1;2;3];;
- : int = 6
```
   [✓ (A) `List.fold_left`]
   (B) `List.fold_right`
   (C) `List.map`
   (D) `List.tl`

4. Which of the following languages has a strong and dynamic type checking system ?
   (A) C and OCaml.
   (B) OCaml and Scheme.
   (C) Scheme and C/C++.
   [✓ (D) Scheme and Smalltalk.]

5. ***Bash.*** Which environment variable must be adjusted in your `.bash_profile` so that programs in a non-standard location may be used ?
   (A) `$HOME`
   (B) `$LANG`
   [✓ (C) `$PATH`]
   (D) `$USER`

6. Parsing refers to :
   (A) Executing a sequence of statements in the proper order, poasibly altered by interpreting gotos.
   (B) Inserting labels into the label table.
   (C) Reading characters from the input and assembling them into tokens.
   [✓ (D) Verifying tokens are in correct syntactic order and assembling the abstract syntax.]

7. ***Scheme.*** What is `2` ?
   (A) `(caar '(1 2 3))`
   [✓ (B) `(cadr '(1 2 3))`]
   (C) `(cdar '(1 2 3))`
   (D) `(cddr '(1 2 3))`

8. ***OCaml.*** What is the type of `( * )` ?
   (A) `int * int -> int`
   (B) `int * int <- int`
   [✓ (C) `int -> int -> int`]
   (D) `int <- int <- int`

9. ***Smalltalk.*** What does `3 + 4` mean ?
   (A) `3` and `+` are messages that are sent to the object `4`.
   (B) The function `+` takes `3` and `4` as arguments and returns a value.
   [✓ (C) The message `+ 4` is sent to the object `3`.]
   (D) The message `+` is sent to the object `3`, which results in a curried function that accepts `4` as an argument.

10. ***Scheme.*** Given `(define (f x y) (x y y))`, what will return `6` ?
   (A) `(f * 2)`
   [✓ (B) `(f + 3)`]
   (C) `(f 3 +)`
   (D) `(f 3 3)`

11. ***OCaml.*** How much function call stack space is occupied by the following function ?
```
let f n = if n <= 1 then n
          else f (n - 1) + f (n - 2);;
```
   (A) $O(1)$
   [✓ (B) $O(n)$]
   (C) $O(n^2)$
   (D) $O(2^n)$

12. ***C/C++/Java.*** Which of the following binary operators is evaluated in a normal (meaning lazy) order ? That means that sometimes the right operand is not evaluated.
   (A) `++`
   (B) `==`
   (C) `[]`
   [✓ (D) `||`]