$Id: cmps112-2018q2-midterm.mm,v 1.132 2018-05-03 13:23:42-07 - - $

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. Fill in the table with one of the following names : John Backus, Alonzo Church, Edsger Dijkstra, James Gosling, Grace Hopper, John McCarthy, Dennis Ritchie, Bjarne Stroustrup, Alan Turing. **[1✔]**

C++	Cobol	Fortran	Lisp
Bjarne Stroustrup	Grace Hopper	John Backus	John McCarthy

2. Define the function **reverse**. Do not use higher-order functions.

   (a) ***Scheme.*** **[2✔]**

```
> (reverse '(1 2 3 4))
(4 3 2 1)
> (reverse '("foo" "bar" "baz"))
("baz" "bar" "foo")
```

```
(define (reverse list)
    (define (rev in out)
        (if (null? in) out
            (rev (cdr in) (cons (car in) out))))
    (rev list '()))
```

   (b) ***Ocaml.*** **[2✔]**

```
# reverse [1; 2; 3; 4];;
- : int list = [4; 3; 2; 1]
# reverse ["foo"; "bar"; "baz"];;
- : string list = ["baz"; "bar"; "foo"]
```

```
let reverse list =
    let rec rev inl outl = match inl with
        | [] -> outl
        | x::xs -> rev xs (x::outl)
    in rev list []
```

3. ***Scheme.*** Using a canonical representation for a multiprecise number as was specified in the Ocaml project, code the function **add** which returns the sum of two lists. The function **add** takes two lists as arguments and defines in inner function **addc** as the worker function. Be sure to use proper indentation in your answer. **[5✔]**

```
> (add '(1 2 3) '(4 5 ))
(5 7 3)
> (add '(9 9 9) '(9 9 9))
(8 9 9 1)
> (add '(9 9 9 9) '(3))
(2 0 0 0 1)
> (add '(1 2) '(1 2 3 4))
(2 4 3 4)
> (add '(0) '(1 2 3))
(1 2 3)
```

```
(define (add num1 num2)
    (define (addc num1 num2 carry)
```

```
(define (add num1 num2)
    (define (addc num1 num2 carry)
        (cond ((and (null? num1) (= carry 0)) num2)
              ((and (null? num2) (= carry 0)) num1)
              ((null? num1) (addc (list carry) num2 0))
              ((null? num2) (addc num1 (list carry) 0))
              (else (let ((sum (+ (car num1) (car num2) carry)))
                    (cons (remainder sum 10)
                    (addc (cdr num1) (cdr num2)
                        (floor (/ sum 10)))))))))
    (addc num1 num2 0))
```

```
    )
    (addc num1 num2 0))
```

4. What is the output from each of the following expressions? **[2✔]**

`(apply + '(1 2 3))`	6
`(map (lambda (x) (+ x 5)) '(1 2 3))`	(6 7 8)
`List.fold_left (-) 0 [1;2;3;4];;`	int = -10
`List.map ((-)1) [1;2;3;4];;`	int list = [0; -1; -2; -3]

5. λ-*calculus.* Given the expression in the λ-calculus shown at the top of each box, show the derivation order to the number 25 for each of normal order and applicative order evaluation. **[2✔]**

normal order evaluation	applicative order evaluation
$(\lambda x\,.\ *\ x\ x)\ \ (+\,2\ 3)$	$(\lambda x\,.\ *\ x\ x)\ \ (+\,2\ 3)$
= (* (+ 2 3) (+ 2 3))	= (\x. * x x) 5
= (* 5 5)	= (* 5 5)
= 25	= 25
=	=
=	=
=	=
=	=
=	=
= 25	= 25

6. Name two kinds of ***universal polymorphism*** and give an example of each. **[2✔]**

Parametric - example something involving C++ templates, Java generics, or Ocaml type parameters.
Inclusion - example involving inheritance, virtual functions, or OOP, etc.

7. Name two kinds of ***ad hoc polymorphism*** and give an example of each. **[2✔]**

conversion - example involving declaration of a function and passing a parameter of a different type
overloading - example showing multiple function declarations given different types

8. `Ocaml`. Define `max` consistent with the examples shown here. **[2✔]**

```
# max;;
- : ('a -> 'a -> bool) -> 'a list -> 'a option = <fun>
# max (>);;
- : '_a list -> '_a option = <fun>
# max (>) [];;
- : 'a option = None
# max (>) [3];;
- : int option = Some 3
# max (>) [3;1;4;1;5;9;2;6];;
- : int option = Some 9
# max (<) [3;1;4;1;5;9;2;6];;
- : int option = Some 1
# max (>) ["foo";"bar";"baz"];;
- : string option = Some "foo"
# max (<) [sqrt 2.;exp 1.];;
- : float option = Some 1.41421356237309515
```

```
let max 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);;
```

9. Define the function **zipwith** that takes a function and two lists and uses that function to join the lists into a single result list. If the lists are of different length, use **failwith** to raise an exception. Do not use the length function. **[2✔]**

```
# zipwith;;
- : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>
# zipwith (+) [1;2;3] [4;5;6];;
- : int list = [5; 7; 9]
# zipwith (/.) [1.;2.;3.] [4.;5.;6.];;
- : float list = [0.25; 0.4; 0.5]
# zipwith (+) [1;3;5] [4];;
Exception: Failure "zipwith".
```

```
let rec zipwith fn list1 list2 = match list1, list2 with
  | [], [] -> []
  | _, [] -> failwith "zipwith"
  | [], _ -> failwith "zipwith"
  | x::xs, y::ys -> fn x y :: zipwith fn xs ys;;
```

10. Without using any higher-order functions, code the function **find**, which will return a value associated with a given key. Use the sample interactions to figure out the structure and arguments to this function.

(a) ***Ocaml.*** **[2✔]**

```
# find;;
- : ('a -> 'b -> bool) -> 'a -> ('b * 'c) list -> 'c option = <fun>
# find (=) 3 [(1,2);(3,4);(5,6)];;
- : int option = Some 4
# find (=) 3 [(5,6);(7,8)];;
- : int option = None
```

```
let rec find cmp key list = match list with
  | [] -> None
  | (k,v)::xs -> if cmp key k then Some v
                 else find cmp key xs;;
```

(b) ***Scheme.*** Use **cond**. Do not use **if**. Return **#f** if not found. **[2✔]**

```
> (find = 3 '((1 2) (3 4) (5 6)))
4
> (find = 3 '((5 6) (7 8)))
#f
```

```
(define (find cmp key list)
  (cond ((null? list) #f)
        ((cmp key (caar list)) (cadar list))
        (else (find cmp key (cdr list)))))
```

11. Define the function **sum** which returns the sum of a list of integers. Use a higher-order function.

(a) ***Scheme.*** **[1✔]**

```
> foldl
#<procedure:foldl>
(define (sum list) (fold_left + 0 list) )
> (sum '(1 2 3))
6
```

(b) ***Ocaml.*** **[1✔]**

```
# let sum = List.fold_left let sum = fold_left (+) 0 ;;
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
val sum : int list -> int = <fun>
# sum [1;2;3];;
- : int = 6
```

12. Define the function **length**.

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

```
> (define (length list) (foldl (lambda (_ x) (+ x 1)) 0 list ))
> (length '(1 3 5 7))
4
```

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

```
# let length = List.fold_left (fun x _ -> x + 1) 0 ;;
val length : '_a list -> int = <fun>
# length [1;2;3;4];;
- : int = 4
```

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. With respect to Java, the term "overloading" refers to:
   (A) Automatic type conversion when the argument does not match the declared type of the parameter.
   (B) Generic classes with type parameterization.
   (C) Multiple functions with the same name and different signatures, defined in the same class.
   (D) Multiple functions with the same name and signature, defined in a base class and also in its derived classes.

2. Which language uses normal order evaluation of expressions?
   (A) Fortran
   (B) Haskell
   (C) Ocaml
   (D) Scheme

3. Which Ocaml expression will generate an error message?
   (A) `(sqrt 2.)`
   (B) `sqrt (2)`
   (C) `sqrt (2.)`
   (D) `sqrt 2.`

4. What is the type of `List.map((+)3)` ?
   (A) `(int -> int) -> int list -> int list`
   (B) `int list -> (int -> int) list`
   (C) `int list -> int list`
   (D) `int list`

5. The PL/1 language allows a non-local `goto` directly from a function to a label in a function deeper down in the function call stack, thus returning past several levels of function calls. In Java and C++, something similar can be accomplished by what statement?
   (A) `break`
   (B) `continue`
   (C) `return`
   (D) `throw`

6. What is the type of `(>)` ?
   (A) `'a * 'a -> bool`
   (B) `'a -> 'a -> bool`
   (C) `bool -> 'a -> 'a`
   (D) `int -> int -> bool`

7. What is `(cddr '((1 2 3) (4 5 6) (7 8 9)))` ?
   (A) `((7 8 9))`
   (B) `(2 3)`
   (C) `(4 5 6)`
   (D) `1`

8. What is `(cdar '((1 2 3) (4 5 6) (7 8 9)))` ?
   (A) `((7 8 9))`
   (B) `(2 3)`
   (C) `(4 5 6)`
   (D) `1`

9. In the λ-calculus expression $(\lambda x.+x\ y)$:
   (A) x is bound and y is bound.
   (B) x is bound and y is free.
   (C) x is free and y is bound.
   (D) x is free and y is free.

10. What is tye type of `(-)` ?
    (A) `int * int * int`
    (B) `int * int -> int`
    (C) `int -> int * int`
    (D) `int -> int -> int`

11. What is the type of `List.map` ?
    (A) `('a -> 'b) -> 'a list -> 'b list`
    (B) `('a * 'b) * 'a list * 'b list`
    (C) `'a list -> 'b list -> ('a -> 'b)`
    (D) `('a list -> 'b list) -> 'a -> 'b`

12. What is the type of `reverse` from the first page?
    (A) `'a list -> 'a list`
    (B) `'a list -> 'b list`
    (C) `int list -> int list`
    (D) `string list -> string list`

The Antikythera mechanism, built circa 150–100 BCE, is the oldest known complex scientific calculator, and is sometimes called the first known analog computer, with operational instructions written in Greek. `http://en.wikipedia.org/wiki/Antikythera_mechanism`