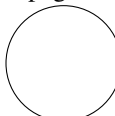
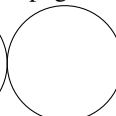
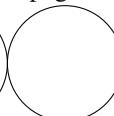
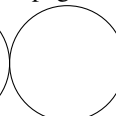



\$Id: cmpls112-2018q2-midterm.mm,v 1.132 2018-05-03 13:23:42-07 - - \$

page 1	page 2	page 3	page 4	Total / 42	<i>Please print clearly :</i>
					<b>Name :</b>
					<b>CruzID :</b> @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
-----	-------	---------	------

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")
```

(b) **Ocaml.** **[2✓]**

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

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))      (define (add num1 num2)
(5 7 3)                      (define (addc num1 num2 carry)
> (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)
```

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

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

<code>(apply + '(1 2 3))</code>	
<code>(map (lambda (x) (+ x 5)) '(1 2 3))</code>	
<code>List.fold_left (-) 0 [1;2;3;4];;</code>	
<code>List.map ((-)1) [1;2;3;4];;</code>	

5.  $\lambda$ -calculus. Given the expression in the  $\lambda$ -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
$(\lambda x . * x x) (+ 2 3)$ = = = = = = = = = 25	$(\lambda x . * x x) (+ 2 3)$ = = = = = = = = = 25

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

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

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
```

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".
```

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
```

(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
```

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) _____)
> (sum '(1 2 3))
6
```

(b) *Ocaml*. [1✓]

```
# let sum = List.fold_left _____ ;;
# 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) (_____))
> (length '(1 3 5 7))
4
```

(b) *Ocaml*. Use **List.fold\_left**. [1✓]

```
# let length = List.fold_left _____ ;;
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		$\times 1 =$	$= a$
number of wrong answers		$\times \frac{1}{2} =$	$= b$
number of missing answers		$\times 0 =$	0
column total $c = \max(a - b, 0)$	12		$= c$

- With respect to Java, the term “overloading” refers to:
  - Automatic type conversion when the argument does not match the declared type of the parameter.
  - Generic classes with type parameterization.
  - Multiple functions with the same name and different signatures, defined in the same class.
  - Multiple functions with the same name and signature, defined in a base class and also in its derived classes.
- Which language uses normal order evaluation of expressions?
  - Fortran
  - Haskell
  - Ocaml
  - Scheme
- Which Ocaml expression will generate an error message?
  - `(sqrt 2.)`
  - `sqrt (2)`
  - `sqrt (2.)`
  - `sqrt 2.`
- What is the type of `List.map((+)3)`?
  - `(int -> int) -> int list -> int list`
  - `int list -> (int -> int) list`
  - `int list -> int list`
  - `int list`
- 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?
  - `break`
  - `continue`
  - `return`
  - `throw`
- What is the type of `(>)`?
  - `'a * 'a -> bool`
  - `'a -> 'a -> bool`
  - `bool -> 'a -> 'a`
  - `int -> int -> bool`
- What is `(caddr '((1 2 3) (4 5 6) (7 8 9)))`?
  - `((7 8 9))`
  - `(2 3)`
  - `(4 5 6)`
  - 1
- What is `(cdar '((1 2 3) (4 5 6) (7 8 9)))`?
  - `((7 8 9))`
  - `(2 3)`
  - `(4 5 6)`
  - 1
- In the  $\lambda$ -calculus expression  $(\lambda x. + x y)$ :
  - $x$  is bound and  $y$  is bound.
  - $x$  is bound and  $y$  is free.
  - $x$  is free and  $y$  is bound.
  - $x$  is free and  $y$  is free.
- What is the type of `(-)`?
  - `int * int * int`
  - `int * int -> int`
  - `int -> int * int`
  - `int -> int -> int`
- What is the type of `List.map`?
  - `('a -> 'b) -> 'a list -> 'b list`
  - `('a * 'b) * 'a list * 'b list`
  - `'a list -> 'b list -> ('a -> 'b)`
  - `('a list -> 'b list) -> 'a -> 'b`
- What is the type of `reverse` from the first page?
  - `'a list -> 'a list`
  - `'a list -> 'b list`
  - `int list -> int list`
  - `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](http://en.wikipedia.org/wiki/Antikythera_mechanism)