$Id: cmps112-2016q4-examl.mm,v 1.72 2016-10-21 16:04:30-07 - - $

page 1 page 2 page 3 page 4 page 5 Total / 43

Please print clearly :

Name : SOLUTION

Login : @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. *Ocaml.* Define a tail-recursive function `evenlen` which returns true if its argument is a list of even length and false if there are an odd number of elements in the list. Do not use a higher-order function. **[2✓]**

```
# evenlen [];;
- : bool = true
# evenlen [1];;
- : bool = false
# evenlen [1;2];;
- : bool = true
```

```
let rec evenlen l = match l with
    | [] -> true
    | [_] -> false
    | _::_::xs -> evenlen xs
```

2. *Scheme.* Define a tail-recursive function `oddlen` which returns `#t` if the list has an odd number of elements, and `#f` if not. Do not use a higher-order function. **[2✓]**

```
> (oddlen '())
#f
> (oddlen '(1))
#t
> (oddlen '(1 2))
#f
```

OK to use (if ...

```
(define (oddlen l)
        (cond ((null? l) #f)
              ((null? (cdr l)) #t)
              (else (oddlen (cddr l)))))
```

3. *Scheme.* Define the tail-recursive function `foldl` so that it may be used in the following example. **[2✓]**

```
> (define (sum list) (foldl + 0 list))
> (sum '(1 2 3))
6
```

```
(define (foldl f u l)
        (if (null? l) u
            (foldl f (f u (car l)) (cdr l))))
```

4. *Ocaml.* Define the function `reverse` to reverse an arbitrary list. Use $O(n)$ time and $O(1)$ stack. You may code tail recursion explicitly, or express the function as a one-liner using `List.fold_left`. **[2✓]**

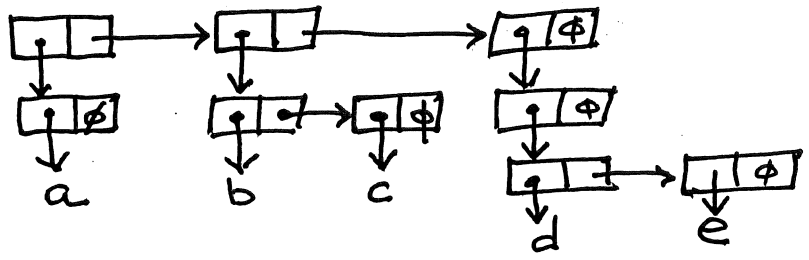
```
val reverse : 'a list -> 'a list
# reverse [1;2;3;4];;
- : int list = [4; 3; 2; 1]
```

```
let reverse l =
    let rec rev l m = match l with
        | [] -> m
        | x::xs -> rev xs (x::m)
    in rev l []
```

```
let reverse =
    List.fold_left
    (fun t h -> h::t) []
```

5. *Scheme.* Draw a picture of the following Scheme expression. For each cons cell, draw a small box with two arrows coming out of it. Each arrow should point at either another cell or an atom. Write the Greek letter phi (ϕ) to indicate a null pointer. **[2✓]**

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



6. *Ocaml.* Define `length` and `sum` for a list using a β-reduced version of the definitions. Fill in the space with an appropriate `fun` and another argument. **[2✓]**

```
# let foldl = List.fold_left;;
val foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

# let length = foldl (fun x _ -> x+1) ∅ ;;
val length : '_a list -> int

# let sum = foldl (+) ∅ ;;
val sum : int list -> int
```

7. *Ocaml.* Define the function `zipwith` that takes a function and two lists and returns a new list with the elements combined. Use `failwith` if the lists are not of the same length. **[2✓]**

```
val zipwith : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
# zipwith (+) [1;2;3;4] [5;6;7;8];;
- : int list = [6; 8; 10; 12]
# zipwith (+) [1;2;3;4] [5;6;7;8;9];;
Exception: Failure "zipwith".
# zipwith (fun a b -> a,b) [1;2;3] ['a';'b';'c'];;
- : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
# zipwith max [1;2;3;4] [4;3;2;1];;
- : int list = [4; 3; 3; 4]
```

```
let rec zipwith f l1 l2 = match l1,l2 with
  | [],[] -> []
  | [],_ -> failwith "zipwith"
  | _,[] -> failwith "zipwith"
  | h1::t1,h2::t2 -> f h1 h2 :: zipwith f t1 t2
```

8. *Ocaml.* In a strongly typed language :
   (a) Every expression has exactly one type.
   (b) When an expression is evaluated, exactly one of the following general things might happen : **[2✓]**
   (i) evaluate to a value of the given type
   (ii) throw or raise an exception
   (iii) not terminate (∞ loop or recursion)
   (iv) exit the program

9. *Scheme.* The Collatz conjectures states that for any positive integer n, if it is repeatedly replaced by $n/2$ when even and $3n+1$ when odd, it eventually converges on the integer 1. Write a function that uses a tail-recursive inner function to return a list of all integers starting from the argument and ending with 1. The inner function produces the list in the reverse order, but the outer function reverses the list. Some Scheme functions to use : `remainder`, `quotient`, `reverse`, etc. **[4✓]**

```
> (collatz 4)
(4 2 1)
> (collatz 10)
(10 5 16 8 4 2 1)
> (collatz 20)
(20 10 5 16 8 4 2 1)
> (collatz 16)
(16 8 4 2 1)
> (collatz 17)
(17 52 26 13 40 20 10 5 16 8 4 2 1)
```

```
(define (collatz n)
  (define (even i)(=(remainder i 2)0))
  (define (cfn n m)
    (if (<= n 1) (cons 1 m)
      (if (even n)
        (cfn (quotient n 2)(cons n m))
        (cfn (+ (* n 3) 1) (cons n m)))))
  (reverse (cfn n '())))
```

10. *C* or *C++*. Code a function to reverse a list. Do not allocate or free any memory. Do not cause memory leak or use uninitialized memory. Do not use recursion. Write a loop which does nothing but manipulate pointers. Return a pointer to the first node in the reversed list. **[2✓]**

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

```
node* reverse (node* head) {
[illegible]
}
```

```
node* old = head;
head = nullptr;
while (old != nullptr) {
   node* t = old;
   old = old -> link;
   t -> link = head;
   head = t;
}
return head
```

11. *Scheme.* Define the functions `map` and `filter`. Do not use higher-order functions.

(a) `map` **[1✓]**

```
> (map (lambda (n) (+ 1 n)) '(3 6 9))
(4 7 10)
```

```
(define (map f l)
  (if (null? l) '()
      (cons (f (car l)) (map f (cdr l)))))
```

(b) `filter` **[2✓]**

```
> (filter (lambda (n) (< n 4)) '(1 2 3 4 5 6 7))
(1 2 3)
```

```
(define (filter p? l)
  (if (null? l) '()
      (let ((a (car l))
            (fd (filter p? (cdr l))))
        (if (p? a) (cons a fd) fd))))
```

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. **[6✓]**

number of correct answers		× 1 =	$= a$
number of wrong answers		× ½ =	$= b$
number of missing answers		× 0 =	0
column total $c = \max(a - b, 0)$	6		$= c$

1. "Go To Statement Considered Harmful", *Communications of the ACM*, Vol. 11, No. 3, March 1968.
   B
   (A) John Backus
   (B) Edsger Dijkstra
   (C) Donald Knuth
   (D) John McCarthy

2. Static type inference is a ***major*** feature of:
   C
   (A) C
   (B) Java
   (C) Ocaml
   (D) Scheme

3. In smalltalk code is executed by:
   C
   (A) calling functions which are static members of classes.
   (B) making use of the standard template library.
   (C) sending messages to objects.
   (D) using higher-order functions.

4. In the expression `(lambda (x) (+ x y))`
   B
   (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.

5. The `make` language can be referred to as:
   B
   (A) a functional language.
   (B) a "little" language.
   (C) a logic language.
   (D) an object-oriented language.

6. In Smalltalk, the expression `3+4.` means:
   B
   (A) The message `+` is sent to the number `3`, the result of which is a function that accepts the message `4`.
   (B) The message `+4` is sent to the number `3`.
   (C) The message `3+` is sent to the number `4`.
   (D) The messages `3` and `4` are sent to the operator `+`.

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 is the Scheme value of:
```
(caddr '(1 2 3 4 5))
```
D
(A) `(3 4 5)`
(B) `(4 5)`
(C) `2`
(D) `3`

2. How much stack space does the following function use?
```
let rec f n = match n with
    | 0 -> 0
    | 1 -> 1
    | n -> f (n - 1) + f (n - 2)
```
C
(A) $O(1)$
(B) $O(\log n)$
(C) $O(n)$
(D) $O(2^n)$

3. What is the Ocaml type signature of:
```
(/);;
```
D
(A) `- : int * int * int`
(B) `- : int * int -> int`
(C) `- : int -> int * int`
(D) `- : int -> int -> int`

4. Which function can be written in a tail recursive purely functional manner?
B
(A) filter
(B) fold_left
(C) fold_right
(D) map

5. Lisp (McCarthy) and Scheme (Steele and Sussman), in general form, are based on a form of mathematics first formulated by Alonzo Church.
A
(A) λ-calculus
(B) μ-calculus
(C) π-calculus
(D) ψ-calculus

6. Which line is a comment in Scheme?
D
(A) `(*...*)`
(B) `/*...*/`
(C) `//...`
(D) `;;...`

7. What feature of imperative languages is typically missing from functional languages?
C
(A) conditionals
(B) functions
(C) loops
(D) recursion

8. What is the signature of the Ocaml function `List.hd` (equivalent to `car`)?
D
(A) `'a -> 'a list`
(B) `'a -> 'a`
(C) `'a list -> 'a list`
(D) `'a list -> 'a`

9. Given:
```
# List.map ((+)3) [1;2;3];;
- : int list = [4; 5; 6]
```
what is the type of `List.map ((+)3)`?
D
(A) `int -> int`
(B) `int -> int list`
(C) `int list -> int`
(D) `int list -> int list`

10. In both Java and C++, what keyword is used to restrict access to a class itself but allow access to classes derived from it?
C
(A) inheritance
(B) private
(C) protected
(D) public

11. Backus-Naur format describes what about a language?
D
(A) environment
(B) linkage
(C) semantics
(D) syntax

12. What language was designed by John Kemeny and Thomas Kurtz in 1965?
A
(A) BASIC
(B) COBOL
(C) FORTRAN
(D) LISP