INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Mid-Spring Semester 2018-19

Date of Examination: 21-Feb-19 **Session (FN/AN)**:AN **Duration**: 2 hrs **Full Marks**: 60
Subject No.: CS 40032 **Subject**: Principles of Programming Languages
Department/Center/School: Department of Computer Science and Engineering
Special Instructions (if any): **Attempt *all* questions. Marks are shown against every question**
No clarification to any questions will be provided. Make and state your assumptions if you need.

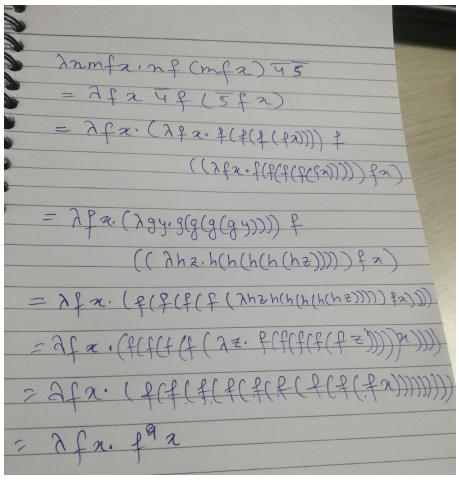
---

1. Consider the following definitions: **[5]**

$$add = \lambda n.\lambda m.\lambda f.\lambda x.\ n\ f\ (m\ f\ x)$$

$$Prove\ that: add\ \overline{4}\ \overline{5}\ =\ 9$$

BEGIN SOLUTION



END SOLUTION

2. **[2 + 3 + 2 + 3 = 10]**

(a) Write the λ-expression for the **Y combinator**.

BEGIN SOLUTION

The λ-expression

$$Y = \lambda t.\ (\lambda x.\ t\ (x\ x))\ (\lambda x.\ t\ (x\ x))$$

is called the **Y combinator**

END SOLUTION

(b) For a function t, show:

$$Y\ t = t\ (Y\ t)$$

BEGIN SOLUTION

$$\begin{aligned} Y\ t &= (\lambda x.\ t\ (x\ x))\ (\lambda x.\ t\ (x\ x)) \\ &= t\ ((\lambda x.\ t\ (x\ x))\ (\lambda x.\ t\ (x\ x))) \\ &= t\ (Y\ t) \end{aligned}$$

END SOLUTION

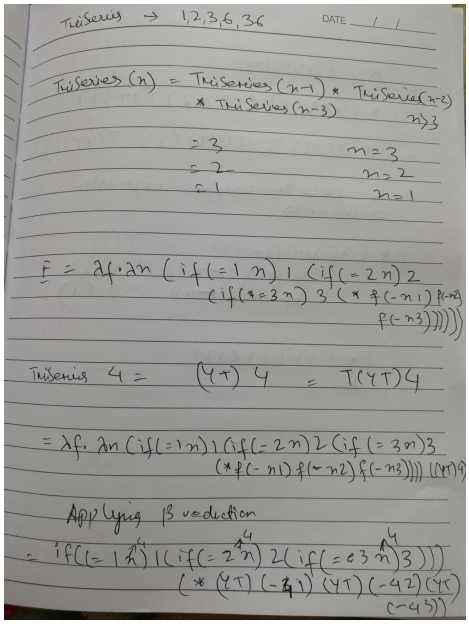
(c) Write the recursive definition for *triSeries* where $triSeries(n)$ can be defined as

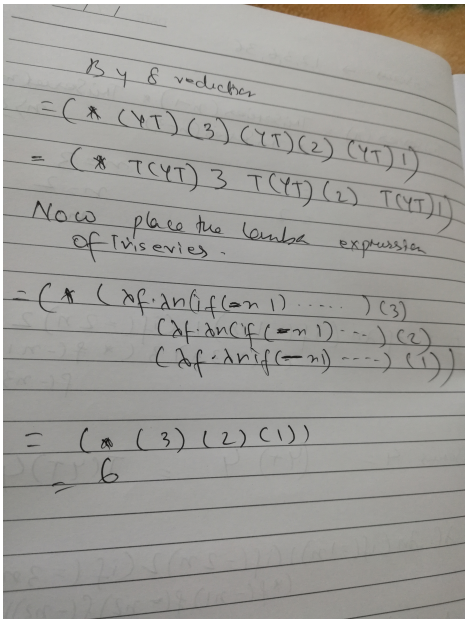
$$\begin{aligned} triSeries(n) &= triSeries(n-1) * triSeries(n-2) * triSeries(n-3), && if\ n > 3 \\ &= 3, && if\ n = 3 \\ &= 2, && if\ n = 2 \\ &= 1 && if\ n = 1 \end{aligned}$$

n is a natural number.
Hence the series starts with 1,2,3,6,36..
Using Y combinator, encode the above recursive definition of *triSeries* as λ-expressions

(d) Reduce *triSeries* 4. Show every step of β- and δ- reductions. You may skip α-reduction steps with a mention of the step.

BEGIN SOLUTION





END SOLUTION

3. **[2 + (2 + 2) + 4 = 10]**

(a) Consider the λ expression

$$L = (\lambda F.\ \lambda W.\ (*\ W\ ((\lambda J.\ (+\ F\ J))\ 3)))\ 4\ 6$$

where $+$ and $*$ are predefined addition and multiplication operators.

i. Build the AST (Abstract Syntax Tree) of L.

ii. Evaluate L by:

A. Normal Order

B. Applicative Order

and represent in Normal Form.

BEGIN SOLUTION

The expression L evaluates to 42.

λ expression $L = (\lambda W.\ (*\ W\ ((\lambda J.\ (+\ 4\ J))\ 3)))\ 6 \Rightarrow$

$L = (\ (*\ 6\ ((\lambda J.\ (+\ 4\ J))\ 3))) \Rightarrow$

$L = (\ (*\ 6\ ((\ (+\ 4\ 3)))))$

Applicative Order:

$L = (\lambda F.\ \lambda W.\ (*\ W\ ((\ (+\ F\ 3))))\ 4\ 6 \Rightarrow$

$L = (\lambda F.\ (*\ 6\ ((\ (+\ F\ 3))))\ 4 \Rightarrow$

$L = (\ (*\ 6\ ((\ (+\ 4\ 3)))))$

END SOLUTION

(b) Show that $((\lambda x.\ y)((\lambda x.\ x\ x)(\lambda x.\ x\ x)))$does not have a Normal Form.

BEGIN SOLUTION

Using applicative order, we cannot reduce the expression

Using normal order it should reduce to y.

END SOLUTION

4. Following questions deal with the semantics of Haskell Language **[2 + 2 + (3+1) + 1 + 1 = 10]**

(a) What is the output of the following command in Haskell.

```
hci> [x*y | x <- [1..5], y <- take 2 (cycle [1,2]) ]
```

Begin Solution

```
[1,2,2,4,3,6,4,8,5,10]
```

End Solution

(b) Write the corresponding eager evaluation version of the && operator of Haskell. The definition of the && operator is given below.

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
```

Name the eager evaluation version as &&!.

Begin Solution

```
(&&!) :: Bool -> Bool -> Bool
True  &&! True  = True
True  &&! False = False
False &&! True  = False
False &&! False = False
```

OR

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```

End Solution

(c) Write a function *extractNonUppercase* which extracts the uppercase letters from a string. *extractNonUppercase* "TYuiJ" would produce "TYJ". Mention the type of the function. Two example functions are written in Haskell notation below for your reference of the syntax.

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z

factorial :: Integer -> Integer
factorial n = product [1..n]
```

Begin Solution

```
extractNonUppercase :: [Char] -> [Char]
extractUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

End Solution

(d) A function names *ElementChecker* checks if an element is present in a list. What is the type of the function?

Begin Solution

```
a -> [a] -> Bool
```

End Solution

(e) What is the type of == operator?

Begin Solution

```
a -> a -> Bool
```

End Solution

5. Following questions deal with the semantics of Scheme Language **[(2 + 2) +4 = 8]**

(a) Specify the output of the following code snippets written in Scheme.

```
a.  (let ((list1 '(a b c)) (list2 '(d e f)))
  (cons (cons (cdr list1)
              (car list2))
        (cons (car (cdr list1))
              (cdr (cdr list2)))))

b.  (cons (quote x) (quote (y z)))
```

Begin Solution

```
a.(((b c) . d) b f)
b. xyz
```

End Solution

(b) Write a function *Counter* using concepts of lambdas and recursion in Scheme to *Count the number of elements in a list.* For example (*Counter* '(a b c b a)) returns 5. We provide the syntax for lambdas in Scheme below.

```
Syntax for lambdas and recursion in Scheme

(define Function_name (lambda (x)  (* x x)))

(define Function_name
```

```
(lambda (n)
        (if (= n 0)
               1
               (* n (Function_name (- n 1))))))
```

Begin Solution

```
Case  1
: List is empty => return 0
Case  2
: List is not empty =>
it has a first element that is an atom =>
return  1 + number of atoms in cdr of list
(define  count1  (lambda (L)
(cond
((null? L)  0)
(else  (add1 (count1 (cdr L)))) )))
```

End Solution

6. The syntaxes for defining functions, lambdas in Lisp is given below. **[5 + 2 = 7]**

```
// function definition
(defun name (parameter-list) "Optional documentation string." body)

(lambda (parameters) body) // anonymous function definition

Some of the common predicates used in LISP

(write (atom 'abcd))
(terpri)
(write (equal 'a 'b))
(terpri)
(write (evenp 10))
(terpri)
(write (evenp 7 ))
(terpri)
(write (oddp 7 ))
(terpri)
(write (zerop 0.0000000001))
(terpri)
(write (null nil ))

Output of the predicate snippets in sequence
T
NIL
T
NIL
T
NIL
T

Decision constructs of Lisp
```

```
(cond   (test1    action1)
(test2    action2)
...
(testn   actionn))

(if (test-clause) (action1) (action2))

(case  (keyform)
((key1)   (action1   action2 ...) )
((key2)   (action1   action2 ...) )
...
((keyn)   (action1   action2 ...) ))
```

Using recursion , define a function *ListAppend* to append two lists. Also show the step by step working of the function *ListAppend* with two lists '(a b c) '(c d e).

Begin Solution

```
(defun listappend (L1 L2)
  "Append L1 by L2."
  (if (null L1)
      L2
    (cons (first L1) (list-append (rest L1) L2))))


 (listappend '(a b c) '(c d e))
 0: (LIST-APPEND (A B C) (C D E))
   1: (LIST-APPEND (B C) (C D E))
     2: (LIST-APPEND (C) (C D E))
       3: (LIST-APPEND NIL (C D E))
       3: returned (C D E)
     2: returned (C C D E)
   1: returned (B C C D E)
 0: returned (A B C C D E)
(A B C C D E)
```

End Solution

7. Write a C++ code to compute gcd of 2 integers **[5 + 5 = 10]**

   (a) using Functors

   (b) using Lambda Expressions

Begin Solution

```
Functor-

#include <iostream>
```

```
#include <functional>

using namespace std;

class GcdFunctor{

public:

int operator()(int a, int b){

return b==0? a: (*this)(b, a%b);

}

};


int main(){

GcdFunctor gcdf;

cout << gcdf(25, 15);

return 0;

}
```

Lambda

```
#include <iostream>

#include <functional>

using namespace std;


int main() {

std::function<int(int, int)> gcd = [&](int a, int b){

    return b == 0 ? a : gcd(b, a%b);

};

cout << gcd(25, 15);

return 0;

}
```

End Solution