# INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

## Mid-Spring Semester 2017-18

**Date of Examination**: 20-Feb-18 **Session (FN/AN)**: FN **Duration**: 2 hrs **Full Marks**: 60
**Subject No.**: CS 40032 **Subject**: Principles of Programming Languages
**Department/Center/School**: Department of Computer Science and Engineering
**Specific charts, graph paper, log book etc., required**: None
**Special Instructions (if any)**: Marks for every question is shown with the question.
No further clarification to any question will be provided. Make and state your assumptions, if any.

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

$$1 = \lambda f.\ \lambda y.\ f\ y$$
$$2 = \lambda f.\ \lambda y.\ f\ (f\ y)$$
$$3 = \lambda f.\ \lambda y.\ f\ (f\ (f\ y))$$
$$M\ +\ N = \lambda x.\ \lambda y.\ (M\ x)\ ((N\ x)\ y)$$

Prove that: $(+\ 2\ 1) = 3$

2. **[2 + 3 + 1 + 1 + (1 + 1 + 1) = 10]**

(a) Write the $\lambda$-expression for the **Y combinator**.

(b) For a function $t$, show:

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

(c) Write the recursive definition for $fibo$ where $fibo(n)$ computes the $n^{th}$ Fibonacci number.

(d) Using Y combinator, encode the above recursive definition of $fibo$ as $\lambda$-expressions.

(e) Reduce $fibo$ 3. Show every step of $\beta$- and $\delta$- reductions. You may skip $\alpha$-reduction steps with a mention of the step.

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

(a) Consider the $\lambda$ expression

$$E = (\lambda s.\ \lambda t.\ (*\ t\ ((\lambda p.\ (-\ s\ p))\ 5)))\ 1\ 8$$

where $-$ and $*$ are predefined subtraction and multiplication operators.

i. Build the AST (Abstract Syntax Tree) of $E$.

ii. Evaluate $E$ by:

A. Normal Order

B. Applicative Order

and represent in Normal Form.

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

4. Following questions are based on the semantics of respective functional programming languages as marked:

(a) Haskell

i. What is the output of the following command in Haskel? **[1]**

```
ghci> [3,2,1] > [2,10,200]
```

ii. What is the output of the following command in Haskell? **[1]**

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
```

iii. Fill in the command to get the required output.
*Note*: Use `cycle` function of Haskell. **[2]**

```
ghci> _____________________________

Output: COMPUTERCOMP
```

iv. Explain the order of evaluation of the functions `foo` and `bar` in Haskell.
*Hint*: Use curried function to explain. **[2]**

```
foo 1 2 (bar 3 2 7) 6
```

v. We can write a maximum function (`maximum'`) in Haskell in the following manner using recursion.

```
maximum' :: (Ord a) => [a] -> a  // specifying the type of the function
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

Following the syntax, write a function named `replicate'`, which takes two inputs of type integer and returns a list of integers.
The first argument specifies how many times an element has to be repeated and the second argument specifies the element. If the first argument is 0, or less than 0, then an empty list will be returned. Specify the type of the function `replicate'`. **[3+1]**

(b) MIT-Scheme

i. Specify the output of the following code snippets written in Scheme. **[5]**

A. `((lambda (x y) (+ x y)) 3 4)`

B.
```
let ((z 4))
          (let (z (+ z 1))
          (+ z z))
```

C. `(quote (quote cons))`

D. `(car (cdr '(a b c d e f)))`

E. `(map (* 2) '(1 2 3 4))`

(c) Lisp

i. The syntax for defining functions, lambdas in Lisp is given below.

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

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

Some of the common predicates used in LISP are:

```
(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 are:

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

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

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

Using the above specification, define a power function (*Power* $x$ $y$ *implies* $x^y$) using recursion constructs of Lisp. **[5]**

5. A Simply-Typed $\lambda$-Calculus, $\Lambda^{\rightarrow}$ comprises:

- The set, *Type*, of *type expressions* is given by:

$$T \in Type ::= C \mid T_1 \rightarrow T_2 \mid (T)$$

where $C \in \mathcal{TC}$, an arbitrary collection of type constants (which may include *Integer*, *Boolean*, etc.)

- The set $\mathcal{TLCE}$ (*Typed Lambda Calculus Expressions*) of *pre-expressions* are given with respect to:
  - a collection of type constants, $\mathcal{TC}$,
  - a collection of expression identifiers, $\mathcal{EI}$, and
  - a collection of expression constants, $\mathcal{EC}$:

as

$$M, N \in \mathcal{TLCE} ::= c \mid x \mid \lambda(x:T).\ M \mid M\ N \mid (M)$$

where $x \in \mathcal{EI}$ and $c \in \mathcal{EC}$

- A static type environment, $\mathcal{E}$, is defined as a finite set of associations between identifiers and type expressions of the form $x : T$, where each $x$ is unique in $\mathcal{E}$ and $T$ is a type. If $x : T \in \mathcal{E}$, then we sometimes write $\mathcal{E}(x) = T$.
- The Type-Checking Rules are:

| | |
|---|---|
| *Identifier Rule* | $\dfrac{}{\mathcal{E} \cup \{x{:}T\} \vdash x{:}T}$ |
| *Constant Rule* | $\dfrac{}{\mathcal{E} \vdash c \in C}$ |
| *Function Rule* | $\dfrac{\mathcal{E} \cup \{x{:}T\} \vdash M{:}T'}{\mathcal{E} \vdash \lambda(x{:}T).M{:}T \rightarrow T'}$ |
| *Application Rule* | $\dfrac{\mathcal{E} \vdash M{:}T \rightarrow T',\ \mathcal{E} \vdash N{:}T}{\mathcal{E} \vdash M\ N{:}T'}$ |
| *Paren Rule* | $\dfrac{\mathcal{E} \vdash M{:}T}{\mathcal{E} \vdash (M){:}T}$ |

Answer the following questions based on Simply Typed Lambda Calculus. **[2 + (3 + 5 + 5) = 15]**

(a) Explain the difference between the *pre-expressions* and *expressions* in $\Lambda^{\rightarrow}$ with examples.

(b) Derive the types of the following expressions: $\varepsilon_0 = \phi$. Show the derivation trees and steps of applications of type checking rules for each of them.

i.

$$(\lambda(x:\ Float).(mult\ x)\ x\ )\ \underline{40.5}$$

Let `mult` be a constant of type `Float` $\rightarrow$ `Float` $\rightarrow$ `Float` and let $\underline{40.5}$ be a constant of type `Float`.

ii.

$$\lambda(g:\ Bool\ \rightarrow Char).\ \lambda(x:\ Bool).\ g\ (\ x\ \&\ \underline{true}\ )$$

Let `&` be the constant with the type `Bool` $\rightarrow$ `Bool` $\rightarrow$ `Bool`. The type of <u>`true`</u> is `Bool`

iii.

$$\lambda(p:\ Float\ \rightarrow Integer).\ \lambda(f:\ Float\ \rightarrow Float).\ \lambda(y:\ Float).\ p\ (f\ (f\ y))$$