# Indian Institute of Technology, Kharagpur

*Department of Computer Science and Engineering*

End-Semester Examination, Spring 2016-17

**Principles of Programming Languages (CS 40032):**

**Students**: 96 — **Date**: 21-Apr-17 (AN)

**Full marks**: 100 — **Time**: 3 hours

---

**Instructions:**

1. Marks for every question is shown with the question.
2. No further clarification to any question will be provided. Make and state your assumptions, if any.

---

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

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

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

     where $C \in \mathcal{TC}$, an arbitrary collection of type constants (which may include *Integer*, *Boolean*, etc.)
   - The set $\mathcal{TLCE}$ (*Type 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 given by:

| | |
|---|---|
| *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* | $\dfrac{\mathcal{E} \vdash M{:}T}{\mathcal{E} \vdash (M){:}T}$ |

Now answer the following questions:

(a) Explain the difference between the *pre-expressions* and *expressions* in $\Lambda^{\rightarrow}$ with examples. [2 + 2 = 4]

(b) Justify the following type-checking rules with examples: [2 + 2 = 4]

i. *Function Rule*

ii. *Application Rule*

(c) Determine the types of the following expressions using the type-checking rules (assume $\mathcal{E}_0 = \phi$): [2 + 3 + 2 = 7]

i. $\lambda(g : A \rightarrow B).\ \lambda(x : A).\ g\ x$

ii. $\lambda(x : Integer).\ (\underline{plus}\ x)\ x$, where $\underline{plus} : Integer \rightarrow Integer \rightarrow Integer \in \mathcal{CE}$

iii. $\lambda(f : Int \rightarrow Int).\ \lambda(y : Int).\ f\ (f\ (f\ y))$

In every case clearly show the use of respective rules in every case.

2. Let us extend Simply-Typed $\lambda$-Calculus from Question 1 to $\Lambda_{rr}^{\rightarrow}$ by adding a *Sum Type*

$$T_1 + \cdots + T_n$$

that represents a disjoint union of the types, where each element contains information to indicate which summand it comes from, even if several of the $T_i$'s are identical. If $M$ is an expression from a type $T_i$, the expression

$$in_i^{T_1,\cdots,T_n}(M)$$

injects the value $M$ into the $i^{th}$ component of the sum $T_1 + \cdots + T_n$. If $M$ is an expression of type $T_1 + \cdots + T_n$, then an expression of the form

$$case\ M\ of\ x_1 : T_1\ then\ E_1\ ||\ \cdots\ ||\ x_n : T_n\ then\ E_n$$

represents a statement listing the possible expressions to evaluate depending on which summand $M$ is a part of. Thus if M was created by

$$in_i^{T_1,\cdots,T_n}(M')$$

for some $M'$ of type $T_i$ then evaluating the *case* statement will result in evaluating $E_i$ using $M'$ as the value of $x_i$.

Now answer the following questions:

(a) Consider: [1 + 2 = 3]

$isFirst = \lambda(y : Integer + (Boolean \rightarrow Boolean)).\ case\ y\ of$

$\quad x : Integer\ then\ x\ +\ \underline{1}\ ||\ f : Boolean \rightarrow Boolean\ then\ f(false)$

Using the application of sum type, evaluate:

i. $isFirst\ M_1$ and

ii. $isFirst\ M_2$

where $M_1 \equiv in_1^{Integer, Boolean \rightarrow Boolean}(\underline{12})$, $M_2 \equiv in_2^{Integer, Boolean \rightarrow Boolean}(neg)$, and $neg : Boolean \rightarrow Boolean$ such that $neg(true) = false\ \&\ neg(false) = true$.

(b) Extend the type expressions and pre-expressions for $\Lambda_{rr}^{\rightarrow}$ to add the sum type. [1 + 1 = 2]

(c) Add *Sum Rule* and *Case Rule* to the type-checking rules in $\Lambda_{rr}^{\rightarrow}$ as given in Question 1. [2 + 3 = 5]

(d) Using the extended set of rules from Question 2b, determine the type of: [5]

$\lambda(k : (A \rightarrow B) + (B \rightarrow B)).\ case\ k\ of$

$\quad M : A \rightarrow B\ then\ (\ (\lambda(p : A).\ M)\ p\ )\ p\ ||$

$\quad N : B \rightarrow B\ then\ (\lambda(r : B).\ N\ (N\ (N\ r)))\ r$

**Note**: *You may make assumptions as you need. State your assumptions clearly.*

3. The abstract syntax of Binary Numerals with Addition is given by:

$$B = 0 \mid 1 \mid B0 \mid B1 \mid B \oplus B$$

(a) Write the semantics of this language in: [3 * 3 = 9]

i. Denotational Semantics

ii. Axiomatic Semantics

iii. Operational Semantics

(b) Derive the axiomatic semantics of this language from its denotational semantics. [5]

(c) Can we derive the denotational semantics of this language from its axiomatic semantics? If this needs additional assumptions, justify. [3]

(d) "*Different styles for expressing semantics of programming languages are not competitive, rather complementary*" – Justify with examples. [3]

4. Consider an *Simple Calculator* that accepts programs in a simple language as input and produces simple, tangible output. The programs are entered by pressing buttons on the device, and the output appears on a display screen as depicted below:

*Simple Calculator*

| *display* | | | | |
|---|---|---|---|---|
| ON | OFF | MR | | |
| 1 | 2 | 3 | ( | + |
| 4 | 5 | 6 | ) | * |
| 7 | 8 | 9 | IF | , |
| | 0 | | | = |

It is an inexpensive model with a single $Memory\_Cell$ for retaining a numeric value. An expression can be input with numbers, parentheses, and operators. It is evaluated when the "=" button is pressed and the result is shown on the *display*. The $Memory\_Cell$ is set with the computed value every time "=" button is pressed and can be recalled by pressing the **MR** button (reads, *Memory Recall*). There is also a conditional evaluation feature, which allows the user to enter a form of if-then-else expression.

A sample session, the abstract syntax, the semantic algebras, and the valuations functions are shown below for quick reference.

- **Sample Session:**

  *press* **ON**
  *press* $(4 + 1\ 2) * 2$
  *press* $=$ (*the calculator prints* 32)
  *press* $1 +$ **MR**
  *press* $=$ (*the calculator prints* 33)
  *press* **IF MR** $+\ 1, 0, 2 +\ 4$
  *press* $=$ (*the calculator prints* 6)
  *press* **OFF**

- **Abstract Syntax:**

  $P \in Program$
  $S \in Expr\_sequence$
  $E \in Expression$
  $N \in Numeral$

$P ::= \mathbf{ON}\ S$
$S ::= E\ =\ S \mid E = \mathbf{OFF}$
$E ::= E_1 + E_2 \mid E_1 * E_2 \mid \mathbf{IF}\ E_1, E_2, E_3 \mid \mathbf{MR} \mid (E) \mid N$

- **Semantic Algebras:**
  I. Truth values
  Domain: $t \in Tr = B$
  Operations: $true$, $false$: $Tr$

  II. Natural numbers
  Domain: $n \in Nat$
  Operations: $zero$, $one$, $two$, $\cdots$ : $Nat$
  $plus, times : Nat \times Nat \rightarrow Nat$
  $equals : Nat \times Nat \rightarrow Tr$

- **Valuation Functions:**
  $\mathbf{P} : Program \rightarrow Nat^*$
  $\mathbf{P}[[\mathbf{ON}\ S]] = \mathbf{S}[[S]](zero)$ (memory cell is initialized to $zero$)

  $\mathbf{S} : Expr\_sequence \rightarrow Nat \rightarrow Nat^*$
  $\mathbf{S}[[E\ =\ S]](n) = let\ n' = \mathbf{E}[[E]](n)\ in\ n'\ cons\ \mathbf{S}[[S]](n')$
  $\mathbf{S}[[E\ \ \mathbf{OFF}]](n) = \mathbf{E}[[E]](n)\ cons\ nil$

  $\mathbf{E} : Expression \rightarrow Nat \rightarrow Nat$
  $\mathbf{E}[[E_1 + E_2]](n) = \mathbf{E}[[E_1]](n)\ plus\ \mathbf{E}[[E_2]](n)$
  $\mathbf{E}[[E_1 * E_2]](n) = \mathbf{E}[[E_1]](n)\ times\ \mathbf{E}[[E_2]](n)$
  $\mathbf{E}[[\mathbf{IF}\ E_1, E_2, E_3]](n) = \mathbf{E}[[E_1]](n)\ equals\ zero \rightarrow \mathbf{E}[[E_2]](n)\ []\ \mathbf{E}[[E_3]](n)$
  $\mathbf{E}[[\mathbf{MR}]](n) = n$
  $\mathbf{E}[[(E)]](n) = \mathbf{E}[[E]](n)$
  $\mathbf{E}[[N]](n) = \mathbf{N}[[N]]$

  $\mathbf{N} : Numeral \rightarrow Nat$ (maps numeral $\mathcal{N}$ to corresponding $n \in Nat$)

Now we introduce two new buttons **MC** and **MS** for an *Advanced Calculator* as follows:

*Advanced Calculator*

| *display* | | | | |
|---|---|---|---|---|
| ON | OFF | MC | MR | MS |
| 1 | 2 | 3 | ( | + |
| 4 | 5 | 6 | ) | * |
| 7 | 8 | 9 | IF | , |
| | 0 | | | = |

The semantics of the *Advanced Calculator* differs from the semantics of *Simple Calculator* as follows:

- The single $Memory\_Cell$ is replaced with a stack $Memory\_Stack$ of $Memory\_Cell$'s.
- Initially, when **ON** button is pressed, the $Memory\_Stack$ is empty.
- After the "=" button is pressed, the newly computed value that is output on the *display* is *not* set to the $Memory\_Stack$. Note this change in the semantics from the *Simple Calculator*.

- A newly computed value (on *display*) can be pushed to the *Memory_Stack* by pressing the button **MS** (reads, *Memory Store*). Naturally, this value remains on top.
- At any time, the topmost value can be recalled from *Memory_Stack* by pressing the **MR** button. This also removes the value from the *Memory_Stack*.
- When the *Memory_Stack* is empty, **MR** returns value 0.
- Pressing **MR** on empty *Memory_Stack* is not an error. It leaves an empty *Memory_Stack*.
- At any time, *Memory_Stack* can be emptied by pressing the button **MC** (reads, *Memory Clear*).

Consider a sample session of the *Advanced Calculator*:

*press* **ON** – *Memory_Stack* = [ ]
*press* $(3+9)*2$
*press* = (*the calculator prints* 24) – *Memory_Stack* = [ ]
*press* **MS** – *Memory_Stack* = [24]
*press* 1 + **MR** – *Memory_Stack* = [ ]
*press* = (*the calculator prints* 25) – *Memory_Stack* = [ ]
*press* **MS** – *Memory_Stack* = [25]
*press* $2*5*3$
*press* = (*the calculator prints* 30) – *Memory_Stack* = [25]
*press* **MS** – *Memory_Stack* = [30 25]
*press* **MR** + **MR** – *Memory_Stack* = [ ]
*press* = (*the calculator prints* 55) – *Memory_Stack* = [ ]
*press* **MS** – *Memory_Stack* = [55]
*press* $(4+1)$
*press* = (*the calculator prints* 5) – *Memory_Stack* = [55]
*press* **MC** – *Memory_Stack* = [ ]
*press* 1 + **MR** – *Memory_Stack* = [ ]
*press* = (*the calculator prints* 1) – *Memory_Stack* = [ ]
*press* **OFF**

Now answer the following:

(a) Update the *Abstract Syntax* and *Semantic Algebra*'s of *Simple Calculator* for the *Advanced Calculator* as needed. [5]

(b) Write the Valuation Functions of the *Advanced Calculator*. [10]

(c) Using the Valuation Functions, write the semantics of the program: [5]

**ON** 2 + 1 = **MS** 3 ∗ 2 = **MS** 4 = **MS MR** + 2 ∗ **MR** = **MC MR** + 2 = **OFF**

**Note:** *You may make assumptions as you need. State your assumptions clearly.*

5. Following questions deal with the semantics of various functional programming languages:

(a) Problems on Haskell Programming

i. Fill in the blanks to get the required output. [2]

```
ghci> ____________________ [1,7,6,4,7]
```

```
Output:  [4,52,39,19,52]
```

*Hint:* Use `map`

ii. Fill in the blanks to get the required output. [2]

```
ghci> ________________ [1..18]
```

```
Output:  [2,4,6,8,10,12,14,16,18]
```

*Hint:* Use `filter` function and required predicates.

iii. Explain the order of evaluation of the following expression in Haskell. [2]

```
Func1 a b (Func 2 4 5)
```

*Hint*: Use curried function.

iv. 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 `merge'`, which takes as input, two sorted lists and merges them. Specify the type of the function `merge'`. **[3 + 1 = 4]**

(b) Problems on Scheme Programming

i. Specify the output of the following code snippets. **[2 * 3 = 6]**

```
a.  ((lambda (x) (+ x x)) (* 4 4))

b.  (let ((x 2))
    (let ((x (+ x 1)))
     (+ x x)))

c.  (quote (quote cons))
```

ii. The definition of the square function using lambda expression in Scheme is given below. [4]

`(define square (lambda (x) (* x x)))`

Using the `square` function, define a function named `pythagoras` which computes the hypotenuse = *hypotenuse* of a right-angled triangle having base = *base* and height = *height*

(c) Problems on Lisp Programming

i. Consider the common syntax of Lisp language as 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

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

   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 the above syntax, define a factorial function using recursion. [6]

ii. Explain the difference between the decision constructs (case, if-else) of Lisp and Haskell with examples. State the reason for the difference. The examples may not be fully correct in terms of syntax. [3 + 1 = 4]