# Final Exam: COSE212 Programming Languages, Fall 2016

Instructor: Hakjoo Oh

Korea University

***Problem 1 (40pts)*** O/X questions. Leave a blank when you are uncertain; each correct answer gets you 2 points but you lose 2 points for each wrong answer.

1. Consider a set $S$ of natural numbers that satisfies the two conditions:

   (a) $0 \in S$, and

   (b) if $n \in S$, then $n + 2 \in S$.

   Such a set $S$ is unique.

2. The following inductive definition

   $$\frac{}{\text{leaf}} \qquad \frac{t_1 \quad t_2}{(n, t_1, t_2)}\ n \in \mathbb{Z}$$

   defines the set of balanced binary trees. (A binary tree is balanced if the depth of the two subtrees of every node never differ by more than 1.)

3. C is a statically typed language with automatic type inference.

4. In C++, compiled programs do not get stuck.

5. In C, variables are first-class objects.

6. Consider the OCaml code:

   ```
   let f a b = a + b
   let g = f 1
   ```

   The type of g is `int -> int`.

7. With static scoping, the program

   ```
   let a = 1 in
     let p = proc (b) (a+b) in
       let f = proc (a) (p a) in
         let a = 5 in
           (f 2)
   ```

   evaluates to 3.

8. With dynamic scoping, the previous program evaluates to 4.

9. Consider the semantics of procedure calls:

   $$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad [x \mapsto v]\rho \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

   The semantics describes the dynamic scoping rule.

10. The nameless representation of the program

    ```
    (let a = 5 in proc (x) (let y = x-a in x-y)) 7
    ```

    is `(let 5 in proc (let (#0-#1) in (#2-#1))) 7`.

11. With static scoping, the following program

    ```
    let f = proc (x) (
        let counter = 0 in
          (set counter = counter + 1; counter)) in
      let a = (f 0) in
        let b = (f 0) in
          (a-b)
    ```

    evaluates to $-1$.

12. With static scoping, the following program

    ```
    let x = 0 in
      let f = proc (x) (set x = 44; x) in
        let g = proc (y) ((f <y>) + x) in
          let z = 55 in
            ((g <z>); z)
    ```

    evaluates to 44.

13. With static scoping, the following program

    ```
    let b = 3 in
    let p = proc (x) proc (y) (set x = 4; b) in
      ((p <b>) <b>)
    ```

    evaluates to 4.

14. Lazy evaluation is always faster than eager evaluation.

15. The static type system in OCaml accepts a program if and only if the program has no type errors at runtime.

16. Our static type system discussed in class accepts the program:

    ```
    (proc (x) (x 1)) ((proc y y) (proc z z))
    ```

17. Our static type system discussed in class accepts the program:

    ```
    let id x = x in
      let x = id 1 in
        let y = id true in
          if y then x else 2
    ```

18. For any Turing-complete language, it is impossible to design a sound and complete static type system.

19. Recall the Church encoding of natural numbers:

    $$c_i = \lambda s.\lambda z.s^i\ z.$$

    The multiplication function mult for Church numerals can be defined as follows:

    $$\mathsf{mult} = \lambda m.\lambda n.\lambda s.m\ (n\ s).$$

20. In program synthesis, the state space of programs is defined by the grammar of the target programming language.

***Problem 2 (15pts)*** Consider the function definition in OCaml:

```
let f xs ys =
  fold (fun x pairs ->
    fold (fun y l -> (x,y) :: l) ys pairs
  ) xs []
```

where `fold` is defined as follows:

```
let rec fold f l a =
  match l with
  | [] -> a
  | hd::tl -> f hd (fold f tl a)
```

1. (5pts) Write the type of the function `f`.
2. (10pts) What is the result of evaluating the following expression?

   ```
   f [1;2] ['a';'b';'c']
   ```

***Problem 3 (20pts)*** Complete the definitions of the functions `zip` and `unzip`.

1. (10pts) The function `zip` receives two lists and pairs corresponding members of the lists:

   `zip` $[x_1; \ldots; x_n]$ $[y_1; \ldots; y_n]$ = $[(x_1, y_1); \ldots; (x_n, y_n)]$

   If the two lists differ in length, ignore surplus elements. For example,

   - `zip [1;2;3] [4;5;6] = [(1,4);(2,5);(3,6)]`
   - `zip [1;2] [4;5;6] = [(1,4);(2,5)]`
   - `zip [1;2;3] [4;5] = [(1,4);(2,5)]`

   Fill in the holes (1) and (2) in the following definition:

   ```
   let rec zip l1 l2 =
   match l1, l2 with
    | x::xs, y::ys -> (1)
    | any -> (2)
   ```

2. (10pts) The function `unzip` is the inverse of `zip`. It takes a list of pairs and returns a pair of lists. For example,

   ```
   unzip [(1,4);(2,5)] = ([1; 2],[4; 5])
   ```

   Complete the following definition:

   ```
   let rec unzip l =
   let conspair (x,y) (xs,ys) = (x::xs, y::ys) in
   match l with
   | [] -> (1)
   | (x,y)::pairs -> (2)
   ```

***Problem 4 (25pts)*** Let us design a C-like imperative programming language. The syntax of the language is defined by the grammar:

| | | | |
|---|---|---|---|
| $S$ | $\to$ | $x := A$ | assignment |
| | $\mid$ | $\{\texttt{var}\ x; S\}$ | block |
| | $\mid$ | $\texttt{skip}$ | skip |
| | $\mid$ | $S_1; S_2$ | sequence |
| | $\mid$ | $\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2$ | conditional |
| | $\mid$ | $\texttt{while}\ B\ \texttt{do}\ S$ | while loop |
| $A$ | $\to$ | $n \mid x \mid A_1 + A_2 \mid A_1 - A_2$ | arithmetic exp. |
| $B$ | $\to$ | $\texttt{true} \mid \texttt{false} \mid A_1 = A_2 \mid A_1 < A_2$ | boolean exp. |

A program is a statement ($S$). A statement is an assignment, local block with variable declaration, skip, sequence, conditional statement, or while loop. An expression is either an arithmetic expression ($E$) or a boolean expression ($B$).

The semantics of the language is defined in a standard way with static scoping and explicit variable initialization. For example,

```
1: { var x; // x is initialized to 0
2:   x := x + 1; // x is 1
3:   { var x; // x is initialized to 0
4:     while (x < 10) x := x + 1;
5:     // x is 10
6:   };
7:   x := x + 1; // x is 2
8: }
```

Note that the variable definition at line 3 is only valid inside the local block at lines 3–5.

To formally define the semantics, we need environments and memory states:

$$\begin{array}{rcl} \sigma \in \mathit{Mem} & = & \mathit{Loc} \to \mathbb{Z} \\ \rho \in \mathit{Env} & = & \mathit{Var} \to \mathit{Loc} \end{array}$$

A memory state ($\sigma$) is a function from locations ($\mathit{Loc}$) to integer values ($\mathbb{Z}$). An environment ($\rho$) maps variables ($\mathit{Var}$) to their locations ($\mathit{Loc}$).

1. (10pts) The semantics $\mathcal{A}(A) : \mathit{Env} \times \mathit{Mem} \to \mathbb{Z}$ and $\mathcal{B}(B) : \mathit{Env} \times \mathit{Mem} \to \{\mathit{true}, \mathit{false}\}$ of arithmetic and boolean expressions are defined as follows:

   $$\begin{array}{rcl} \mathcal{A}(n)(\rho,\sigma) & = & n \\ \mathcal{A}(x)(\rho,\sigma) & = & \boxed{(1)} \\ \mathcal{A}(A_1 + A_2)(\rho,\sigma) & = & \mathcal{A}(A_1)(\rho,\sigma) + \mathcal{A}(A_2)(\rho,\sigma) \\ \mathcal{A}(A_1 - A_2)(\rho,\sigma) & = & \mathcal{A}(A_1)(\rho,\sigma) - \mathcal{A}(A_2)(\rho,\sigma) \\ \mathcal{B}(\texttt{true})(\rho,\sigma) & = & \mathit{true} \\ \mathcal{B}(\texttt{false})(\rho,\sigma) & = & \mathit{false} \\ \mathcal{B}(A_1 = A_2)(\rho,\sigma) & = & \boxed{(2)} \\ \mathcal{B}(A_1 < A_2)(\rho,\sigma) & = & \boxed{(3)} \end{array}$$

   Complete the definition.

2. (15pts) The semantics of statements is defined by the relation

   $$\rho, \sigma \vdash S \Rightarrow \sigma'$$

   which means that given environment $\rho$, executing $S$ on the input memory $\sigma$ produces the output memory $\sigma'$. Complete the definition:

   $$\frac{}{\rho, \sigma \vdash x := A \Rightarrow \boxed{(1)}}$$

   $$\frac{\boxed{(2)}}{\rho, \sigma \vdash \{\texttt{var}\ x; S\} \Rightarrow \sigma_1}\ \boxed{(3)}$$

   $$\frac{}{\rho, \sigma \vdash \texttt{skip} \Rightarrow \sigma}$$

   $$\frac{\boxed{(4)}}{\rho, \sigma \vdash S_1; S_2 \Rightarrow \sigma_2}$$

   $$\frac{\rho, \sigma \vdash S_1 \Rightarrow \sigma_1}{\rho, \sigma \vdash \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \Rightarrow \sigma_1}\ [\![B]\!](\rho,\sigma) = \mathit{true}$$

   $$\frac{\rho, \sigma \vdash S_2 \Rightarrow \sigma_1}{\rho, \sigma \vdash \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \Rightarrow \sigma_1}\ [\![B]\!](\rho,\sigma) = \mathit{false}$$

   $$\frac{}{\rho, \sigma \vdash \texttt{while}\ B\ \texttt{do}\ S \Rightarrow \sigma}\ [\![B]\!](\rho,\sigma) = \mathit{false}$$

   $$\frac{\boxed{(5)}}{\rho, \sigma \vdash \texttt{while}\ B\ \texttt{do}\ S \Rightarrow \sigma_2}\ [\![B]\!](\rho,\sigma) = \mathit{true}$$