Module Title: ELEMENTS OF PROGRAMMING LANGUAGES
Exam Diet (Dec/April/Aug): MOCK EXAM 2015
Brief notes on answers:

1. (a)

$$e ::= x \mid e_1\ e_2 \mid \lambda x.e$$

(b) (1): FALSE, (2): TRUE, (3): TRUE, (4): FALSE

(c) The problem is that the term substituted for x contains a free variable y, which is *captured* by the lambda-bound variable y. This is bad because it means that $y+1$ will evaluate to one plus the argument passed in for y, rather than one plus whatever y referred to outside of the function. The way to fix this is to rename the bound variable y before performing the substitution, e.g.:

$$(\lambda x.\lambda y.x + y)\ (y+1) \mapsto (\lambda y'.(y+1) + y')$$

(d) (i). Call-by-value Call-by-value means that arguments are evaluated to values before being substituted in for variables. One advantage of call-by-value is that the program's behavior and performance is more predictable, i.e. we can more easily determine when an expression is evaluated, and it is evaluated at most once.

(ii). Call-by-name Call-by-name means that arguments are substituted without being computed, and are only evaluated if they are needed inside the function call. One advantage of call-by-name is that expressions whose value is never needed are not evaluated. Another advantage is that it can be easier to reason about the correctness of programs since equational principles hold.

(iii). Call-by-need Call-by-need means that arguments are not evaluated until their values are needed, but once they are evaluated, the value is recorded so that it can be reused. One advantage of call-by-need is that expressions are evaluated at most once: if the expression is not needed, it isn't evaluated. Another advantage is (like call-by-name) equational reasoning is valid, and lazy evaluation can be used to implement infinite data structures such as streams.

(e)

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{}{(\lambda x.\ e_1)\ e_2 \mapsto e_1[e_2/x]}$$

2. (a) (i).

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{array}(e_1, e_2) : \texttt{array}[\tau]}$$

$$\frac{\Gamma \vdash e_1 : \texttt{array}[\tau] \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1[e_2] : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{array}[\tau] \quad \Gamma \vdash e_2 : \texttt{int} \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1[e_2] := e_3 : \texttt{unit}}$$

(ii).

$$\frac{\tau <: \tau'}{\texttt{array}[\tau] <: \texttt{array}[\tau']}\ \textit{covariant} \qquad \frac{\tau' <: \tau}{\texttt{array}[\tau] <: \texttt{array}[\tau']}\ \textit{contravariant}$$

2 marks for each rule, correctly labeled.

(iii). Arrays shouldn't be contravariant because this would mean we can cast an array of Objects to an array of Integers and so treat arr[i] as an integer even if it is actually a string. Arrays shouldn't be covariant because this would mean we can cast an array of Integers to and array of Objects, and so it would be allowed to update an array of Integers with an arbitrary Object (e.g. a String), violating the type of the array. Therefore, arrays should be invariant (that is, neither contravariant nor covariant): that is, the subtyping relation only relates array types whose arguments are equal.

(b) (i). First, mutable variable x is initialized to value 0. Then we define a new exception type MyException by making it a subclass of Throwable. We then start a try block, which consists of another try block. In the inner try block, we first add 1 to x, then we raise an exception. The statement adding 10 to x is never executed because of the exception. The exception is not caught by the inner catch block because the type of exception doesn't match, so the statement adding 100 to x is not executed. The finally block is executed so 1000 is added to x. The thrown exception MyException therefore exits the inner try/catch/finally block and is handled by the outer catch block, which does have a case matching it. So, 10000 is added to x.

(ii). type tests and coercion, i.e. `isInstanceof` and `asInstanceOf`. One mark for mentioning each feature.

(iii). The value of x will be 11001. (1 mark for each correct digit.)

3. (a)
```
Line 1: y is binding
Line 2: A is binding and x is binding
Line 3: z is binding and x,y are bound
Line 4: f, x are binding and z is bound
Line 5: A, y, f are bound
```

(b)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{x{:}\texttt{bool}, y{:}A, z{:}A \vdash x : \texttt{bool}} \quad \overline{x{:}\texttt{bool}, y{:}A, z{:}A \vdash y : A} \quad \overline{x{:}\texttt{bool}, y{:}A, z{:}A \vdash z : A}}{x{:}\texttt{bool}, y{:}A, z{:}A \vdash \texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z : A}}{x{:}\texttt{bool}, y{:}A \vdash \lambda z{:}A.\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z : A \to A}}{x{:}\texttt{bool} \vdash \lambda y{:}A.\lambda z{:}A.\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z : A \to A \to A}}{\vdash \lambda x{:}\texttt{bool}.\lambda y{:}A.\lambda z{:}A.\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z : \texttt{bool} \to A \to A \to A}}{\vdash \Lambda A.\lambda x{:}\texttt{bool}.\lambda y{:}A.\lambda z{:}A.\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z : \forall A.\texttt{bool} \to A \to A \to A}$$

(c) (i).

$$\frac{\sigma, s \Downarrow \sigma' \quad \sigma', e \Downarrow \texttt{true} \quad \sigma', s\ \texttt{while}\ e \Downarrow \sigma''}{\sigma, \texttt{do}\ s\ \texttt{while}\ e \Downarrow \sigma''}$$

$$\frac{\sigma, s \Downarrow \sigma' \quad \sigma', e \Downarrow \texttt{false}}{\sigma, \texttt{do}\ s\ \texttt{while}\ e \Downarrow \sigma'}$$

(ii). This should look like:

```
stmt; while (exp) { stmt }
```