Vocabulary

Categorize each of the four main languages we have worked with (Python, OCaml, C, and Java) according to the following distinctions.

a.) Statically typed vs. dynamically typed
   Static: OCaml, C, Java. Dynamic: Python
b.) Object-oriented vs. non-object-oriented
   Object-oriented: Python, OCaml, Java. Non-object-oriented: C
c.) Active memory management vs. passive management
   Active: C. Passive: Python, OCaml, Java
d.) Primarily imperative vs. primarily functional
   Imperative: Python, C, Java. Functional: OCaml

Type Inference

For each OCaml value below, fill in the blank for the type annotation or else write “ill typed” if there is a type error on that line. Your answer should be the most generic type for the value—i.e. if int list and bool list are both possible types of an expression, you should write 'a list.

The first one is done for you.

let z : _______ 'a list list ________ = [[]]
let a : __________ ill-typed___________ = true::false
let b : __________'a list list__________ = [] :: [] :: []
let c : __________ill-typed___________ = begin match [] with
   | [] -> "foo"
   | h::t -> h+3
end
let d : _______int list -> bool list____ = transform (fun x -> x>2)
let e : _'a list -> 'a list -> 'a list_ = fold (fun x y -> x::y)
let f : _______'a -> int___________ = (fun x -> fun y -> x) 42
let g : ____int list list___________ = let f x = [x] in f (f 5)
let h : __int list list -> (int -> int) list list_ = let sum x y = x+y in
   transform (transform sum)
Note: in the above, recall our definitions of transform and fold:

```ml
let rec transform (f : 'a -> 'b) (ls : 'a list) : 'b list =
  begin match ls with
  | [] -> []
  | h::t -> (f h) :: (transform f t)
  end

let rec fold (combine: 'a -> 'b -> 'b) (base: 'b) (l: 'a list): 'b =
  begin match l with
  | [] -> base
  | h :: t -> combine h (fold combine base t)
  end
```

**Conditional Expressions**

Consider the Java code below. Underline the calls to `getVal()` that will actually be executed when the following code runs.

```java
class Cond {
  private int value;
  public Cond(int v) {
    value = v;
  }
  public int getVal() {
    return value;
  }
}
public static void main(String[] args) {
  Cond c = new Cond(7);
  if ((c.getVal()<8)&&(c.getVal()>6)) {
    System.out.println(c.getVal());
  }
  c = new Cond(5);
  if ((c.getVal()<8)||(c.getVal()>6)) {
    System.out.println(c.getVal());
  }
  c = new Cond(3);
  if ((c.getVal()>8)&&(c.getVal()<6)) {
    System.out.println(c.getVal());
  }
  c = new Cond(6);
  if ((c.getVal()>8)||(c.getVal()<6)) {
    System.out.println(c.getVal());
  }
  c = new Cond(2);
  if (((c.getVal()>8)||(c.getVal()<6))&&((c.getVal()<5)||(c.getVal()<9))) {
    System.out.println(c.getVal());
  }
  c = new Cond(8);
  if (((c.getVal()>2)&&(c.getVal()<6))||(c.getVal()>4)||(c.getVal()<3)) {
    System.out.println(c.getVal());
  }
}
```
Consider the pseudocode below in answering the questions that follow. Assume that the value of n has been set previously. It may have any integer value.

```plaintext
i = 0
print i
while i < n do
    i = i+1
    print i
end
```

a.) Rewrite to get the same behavior using repeat...until

```plaintext
i = 0
repeat
    print i
    i = i+1
until i >= n
```

b.) Rewrite to get the same behavior using do...while

```plaintext
i = 0
do
    print i
    i = i+1
while i < n
```

c.) Rewrite to get the same behavior using a Python-style range loop

```plaintext
for i in range(0,6):
    print i
```

d.) Rewrite to get the same behavior using a while true do... loop with a mid-loop break to exit.

```plaintext
i = 0
while true do
    print i
    if i >= n then break
    i = i+1
end
```

This turned out to be the most difficult question on the exam. Most people correctly figured out the ending condition, making sure that their loops printed numbers up to and including n. However, you also need to pay attention to the beginning of the loop. The code given will print only 0 when n = 0. Many of the responses to this question had two print statements before the first continuation test, meaning that they would always print at least two numbers regardless of the value of n.
Subtypes

Consider the code on the following page in answering the questions below.

a.) What are the subtypes of Alpha?  
   Epsilon, Lambda

b.) What are the subtypes of Beta?  
   Zeta, Iota, Theta, Kappa

c.) What are the subtypes of Gamma?  
   Zeta, Theta, Lambda

d.) What are the subtypes of Delta?  
   Eta, Iota, Theta, Lambda, Kappa

e.) What are the supertypes of Theta?  
   Beta, Gamma, Delta, Zeta

f.) What are the supertypes of Lambda?  
   Alpha, Gamma, Delta, Epsilon

class Alpha {
   // details omitted
}
class Beta {
   // details omitted
}
interface Gamma {
   // details omitted
}
interface Delta {
   // details omitted
}
class Epsilon extends Alpha {
   // details omitted
}
class Zeta extends Beta implements Gamma {
   // details omitted
}
class Eta implements Delta {
   // details omitted
}
class Theta extends Zeta implements Delta {
   // details omitted
}
class Iota extends Beta implements Delta {
   // details omitted
}
class Kappa extends Iota {
   // details omitted
}
class Lambda extends Epsilon implements Gamma, Delta {
   // details omitted
}
a.) The diagram below shows one state of the abstract stack machine during execution of some OCaml code. Circle all the bindings that are accessible in the current scope.

**In OCaml, all bindings in the stack are accessible as long as they are not shadowed by a more recent binding of the same name. Only the most recent bindings for \( n \) and \( a \) (at the bottom) are accessible.**

b.) If the compiler employs a tail recursion optimization, the stack would look different. Draw the state of the stack under this assumption.

Under tail recursion, the stack frames for recursive calls are reused. Thus instead of three frames for factTR we have just one, the most recently called.
c.) The diagram below shows one state of the abstract stack machine during execution of some C code. The current line is underlined. Circle all the bindings that are accessible in the current scope.

In C, only the current function is in scope. Any previous bindings are inaccessible (this is the intended meaning of the thick black bar).