Programming Languages (CSCI-GA.2110.001 Fall 2016)

Final Exam
ANSWERS

Write all answers in the blue book, unless instructed otherwise.

1. True/False. Please Circle the correct answer on this sheet.

   (a) **T** The implementation of a dynamically scoped language does not require a static link.

   (b) **F** The syntax of a language are the rules giving meaning to correct programs.

   (c) **F** In Scala, a generic class declared as "`class C[T] { ... }`" is covariantly subtyped.

   (d) **F** In Scala, a generic class declared as "`class C[T] { ... }`" is contravariantly subtyped.

   (e) **T** In Ada, the task making an entry call waits until the task accepting the entry call finishes executing the code within the accept block.

   (f) **T** Scheme is dynamically typed but statically scoped.

   (g) **T** In order to be considered "object oriented", a programming language must allow the encapsulation of code and data into a single structure, such as an object.

   (h) **F** Because of ML's parametric polymorphism, type checking during execution is required.

   (i) **T** The "erasure" property of Java generics means that an `ArrayList<Vehicle>` object and an `ArrayList<Car>` object are represented the same in memory.

2. (a) What value does the following Scheme code return? You can assume the indenting is correct (so expressions that are indented are at the same level of nesting).

```
(letrec
   ( (f (lambda (L) (cond ((null? L) '0) (else (+ (car L) (f (cdr L)))))))
     (g (lambda (L) (cond ((null? L) '0) (else (+ 1 (g (cdr L)))))))
     (h (cons '3 (cons '4 (append '(0 5 4) '(7 8 1))))) )
   (/ (f h) (g h)))
```

   4, which is the average value in the list h. f computes the sum of the elements of the list, g computes the length of the list.

   (b) Write a regular expression defining the set of strings representing variable names, which start with a letter and contain any number of letters, digits, and _ (underscore).

   Answer:

```
[a-zA-Z][a-zA-Z0-9_]*
```

   (c) What would the following program print if pass by reference parameter passing were used?

```
program main;

y: Integer := 7;

   procedure f(x: Integer)
   begin
     x := x * 2;
     y := y + 1;
   end f;
```

```
begin (* main *)
  f(y);
  print(y);
end main;
```

15

(d) What would the above program print if pass by value-result parameter passing were used?

14

3. Give the results of the following substitutions in the λ-calculus.

(a) $(\lambda z.\ +\ x\ 1)\ [(+\ y\ y)/x]$
Answer: $(\lambda z.\ +\ (+\ y\ y)\ 1)$

(b) $(\lambda x.\ +\ x\ 1)\ [(+\ y\ y)/x]$
Answer: $(\lambda x.\ +\ x\ 1)$

(c) $(\lambda y.\ +x\ y)\ [(+\ y\ y)/x]$
Answer: $(\lambda z.\ +(+\ y\ y)\ z)$

4. (a) In ML, write a *polymorphic* function `sumLists` that takes two lists as parameters and adds the corresponding elements of the two lists together, returning a list of the results. In order to be polymorphic, though, `sumLists` should also take, as its first parameter, the `+` operator to be used to add the elements. For example, calling `sumLists` as follows

```
sumLists (op +) [1,2,3,4] [5,6,7,8]
```

would return `[6,8,10,12]`.
You can assume the two lists are always the same length. You must use pattern matching (no `if..then...else`) and `+` must be used as an *infix* operator within `sumLists` to add two elements together.
Answer:

```
fun sumLists (op +) [] _ = []
 |  sumLists (op +) (x::xs) (y::ys) = (x+y)::sumLists (op +) xs ys
```

(b) What is the type of `sumLists`?
Answer: `('a * 'b -> 'c) -> 'a list -> 'b list -> 'c list`

(c) Explain how the compiler would infer the type of `sumLists`.
The types of the variables `x` and `y` are unconstrained, so we infer that the type of `x` is `'a` and the type of `y` is `'b`. The input type of the infix `+` operator must be a tuple of two values. In this case, since `+` is applied to `x` and `y`, the input type of `+` is `'a * 'b`. The output of `+` is unconstrained, so we infer that `+` has type `('a * 'b -> 'c)`. In this case, `x+y` has type `'c`, so the result of `sumlists` must be `'c list`. Therefore, the type of `sumlists` is: `('a * 'b -> 'c) -> 'a list -> 'b list -> 'c list`.

(d) Suppose you wanted to write a function `longerElements` that took two lists of lists, `L1` and `L2`, as parameters and returned a list of the longer of the corresponding elements of `L1` and `L2`. That is, the i^{th} element of the result list would be the longer of the i^{th} element of `L1` and the i^{th} element of `L2`. For example,

```
longerElements [[1,2,3],[4,5],[6,7,8,9]] [[10,11],[12,13,14],[15,16]]
```

would return `[[1,2,3],[12,13,14],[6,7,8,9]]` because `[1,2,3]` is longer than `[10,11]` and `[12,13,14]` is longer than `[4,5]`, etc.

Fill in the body of `longerElements`,below on this sheet, so that (1) you use `sumLists` and (2) the body of `longerElements` is just one line.

```
fun longerElements L1 L2 =
```

You may use the built-in ML function `length`, such that `length L` returns the length of the list `L` (and you may use `if ... then ... else ..`).

Answer:

```
fun longerElements L1 L2 =
     sumLists (fn (x,y) => if length x >= length y then x else y) L1 L2
```

5. As you know, the Java API defines the generic class `ArrayList<T>`, which provides the following methods (among many others):

```
void add(T e);  // inserts e at the end of the ArrayList
T get(int i);   // returns the i'th element of the ArrayList
```

A convenient way to iterate over the elements of an `ArrayList<T>` is:

```
for(T e: A) { ... }  // where A is an ArrayList<T>
```

(a) Suppose classes `A` and `B` are defined in Java as follows:

```
class A { int x; }
class B extends A { int y; }
```

Suppose also that covariant subtyping between instances of a generic class were allowed in Java (which it's not). Give a very short example of some Java code, using `A` and `B` above, as well as `ArrayList<>`, of how covariant subtyping of generics would allow a program to try to access a field of an object, even though the field doesn't exist.

Answer:

```
ArrayList<B> ab = new ArrayList<B>();
ArrayList<A> aa = ab;  // allowed if ArrayList<B> is a subtype
                       // of ArrayList<A>.
aa.add(new A());
B b = ab.get(0);  // returns an A object
b.y = 3;          // attempts to access the y field of an A object,
                  // which doesn't exist.
```

(b) As you know, the Java API defines the generic interface `Comparable<T>`, which requires any class implementing `Comparable<T>` to provide the following method:

```
int compareTo(T x); // a return value < 0 represents "less than",
                    // = 0 represents "equal to",
                    // > 0 represents "greater than"
```

Define a Java generic class `Set<>`, representing a set of elements, which is parameterized by an element type `T` and implements the following methods:

```
T getFrom();   // returns an element of the set (doesn't matter which)
void addTo(T e);  // adds a new element e to the "this" set, but only
                  // if the element is not already in the set.
boolean member(T e);  // returns true if e is an element of the
                      // "this" set, false otherwise.
Set<T> union(Set<T> other);  // returns a new Set<T> representing the
                             // union of the "this" set and the "other" set.
Set<T> difference(Set<T> other); // returns a new Set<T> containing all the
                                 // elements of the "this" set that are
                                 // not in the "other" set.
```

Because a `Set<T>` must not contain duplicates, values of type `T` should be able to be compared to each other, so you need to specify a constraint on `T`. Hint: `union()` and `difference()` are easily implemented using the other methods. You can use the `ArrayList<>` class in your definition of `Set<>` however you like.

Answer:

```
class Set<T extends Comparable<T>> extends ArrayList<T> {

    T getFrom() { return get(0); }

    public boolean addTo(T e) {
        if (!member(e))
            return add(e);
        else return false;
    }

    boolean member(T e) {
        for (T elt: this) {
            if (elt.compareTo(e)==0) return true;
        }
        return false;
    }

    Set<T> union(Set<T> other) {
        Set<T> s = new Set<T>();
        for (T e: this) s.add(e);
        for (T e: other) s.add(e);
        return s;
    }

    Set<T> difference(Set<T> other) {
        Set<T> s = new Set<T>();
        for (T e: this) {
            if (!other.member(e)) s.add(e);
        }
        return s;
    }
}
```

6. (a) Describe why function subtyping in Scala, which is *contravariant* on the input types, satisfies the subset interpretation of subtyping. That is, explain why, if `B` is a subtype

of A, then the set denoted by A->Int is a subset of the set denoted by B->Int.

The type A->Int denotes the set of all functions that can be applied to an A object and return an Int. The type B->Int denotes the set of all functions that can be applied to an B object and returns an Int. Any function that can be applied to an A object, so is in A->Int, can also be applied to a B object, so is in B->Int. Thus, since every element of A->Int is also an element of B->Int, A->Int is a subset of B->Int.

(b) Define in Scala an abstract class and some case classes for representing a simple tree structure, where a tree is either (1) a node that has a left and right subtree and (2) a leaf that has an Int label.

Answer:

```
abstract class Tree

case class Node(l:Tree, r:Tree) extends Tree

case class Leaf(x:Int) extends Tree
```

(c) Write the function fringe(t) in Scala, where t is a tree represented by your tree type in the previous question, that returns a list of the labels found at the leaves. In Scala, List() represents an empty list, the expression x :: L returns a new list containing x and all the values in the list L, and L1 ++ L2 returns a new list containing all the elements of the lists L1 and L2. In writing fringe, you must use pattern matching (do not use a conditional).

Answer:

```
def fringe(t: Tree):List[Int] = t match {
  case Node(l,r) => fringe(l) ++ fringe(r)
  case Leaf(x) => List(x)
}
```

7. (a) When space is allocated for a data structure (array, object, etc.), under what circumstances can the data structure be allocated on the stack and under what circumstances must the data structure be allocated in the heap?

If a data structure does not outlive the function that created it, then the data structure can be allocated in the stack frame for that function. However, if the data structure might outlive the function that created it (i.e. the data structure can be referenced after the function returns), then the data structure would, in general, need to be allocated in the heap.

(b) Explain why garbage collection algorithms that work in conjunction with a heap pointer use forwarding addresses, while garbage collection algorithms that work with a free list don't use forwarding addresses.

The use of a heap pointer requires that live objects been compacted during garbage collection. In order to perform compaction, live objects must be moved during garbage collection, in which case a forwarding address is needed to indicate where the object has moved to. The use of a free list does not require compaction, so live objects can be left in place and no forwarding address is needed.