Programming Languages
CSCI-GA.2110.001 Fall 2014

Homework 2
Due Thursday, December 11

ANSWERS

1. Write the following in Scheme (you don't have to implement it on a computer):

   (a) The function `(listfromTo a b)` which, given two integers `a` and `b`, returns a list of all the numbers between `a` and `b`, inclusive.

```
(define (listFromTo a b)
  (cond ((> a b) '())
        (else (cons a (listFromTo (+ a 1) b)))))
```

   (b) The function `(removeMult a L)` which removes all multiples of the integer `a` from the list `L` of integers. You can use the built-in function `(modulo x y)` which returns the remainder of `x`/`y`.

```
(define (removeMult a L)
  (cond ((null? L) '())
        ((= (modulo (car L) a) 0) (removeMult a (cdr L)))
        (else (cons (car L) (removeMult a (cdr L))))
        ))
```

   (c) The function `(sieve n)` which returns the list of all prime numbers less than or equal to `n` using the Sieve of Eratosthenes. The algorithm starts with the list of numbers from 2 to `n` and, starting from the beginning of the list, for each element of the list encountered, it removes all subsequent multiples of that element from the list (before progressing to the next element). Be sure to write a purely functional version of it.

```
(define (sieve n)
  (letrec
      ((do-sieve
          (lambda (L)
            (cond ((null? L) '())
                  (else (cons (car L) (do-sieve (removeMult (car L) (cdr L)))))))))
    (do-sieve (listFromTo 2 n))))
```

   It would have also been fine to use a separate helper function outside of `sieve`.

2. Suppose you have written a Scheme interpreter where the main evaluation function is called `my-eval`.

   (a) Suppose the following new construct was added to Scheme, which your interpeter had to interpret:

   `(longest (`exp_1 $result_1$`) ... (`exp_N $result_N$`))`

The `longest` construct has to evaluate each of exp_1 through exp_N, which should each return a list. If expression exp_i returns the longest list (as determined by the usual `length` function), then the value of $result_i$ is returned (and none of the other *result* expressions are evaluated). For example,

```
(longest ('(1 2 3) (+ 1 2 3))
         ((cdr '(1 2 3 4 5)) 'yes)
         ((cons 8 (list 9 10)) 'no) )
```

would return the symbol `yes`, because `(cdr '(1 2 3 4 5))` is a longer list than `(1 2 3)` or the result of `(cons 8 (list 9 10))`. Write a function `handle-longest`, which you could insert into your interpreter, which takes an entire expression `(longest ....)` and an environment and interprets the expression.

```
; This could be done any number of ways.

(define (handle-longest exp env)
  (let* ((pairs (cdr exp))
         (evald-pairs (map (lambda (pair) (list (my-eval (car pair) env) (cadr pair)))
                           pairs)))
    (my-eval (return-longest-exp evald-pairs 0 '()) env)
    ))

(define (return-longest-exp pairs n exp)
  (cond ((null? pairs) exp)
        ((> (length (caar pairs)) n)
         (return-longest-exp (cdr pairs) (length (caar pairs)) (cadr (car pairs))))
        (else
         (return-longest-exp (cdr pairs) n exp))
        ))
```

(b) Explain why, in your Scheme interpreter, you were able to use `(let ...)` in your code that handled `(let ..)` (e.g. in your `handle-let` function) and were able to use `(cond ...)` in your `handle-cond` function. Even if you didn't, explain why you could have. In other words, explain why, if your interpreter needed to implement some construct (such as `let` and `cond`), you were still able to use that construct in your interpreter.

The implementation of let, cond, if, etc. in each level of the Scheme interpreter relies on the let, cond, if, etc. constructs of the level below it. That is, your Scheme interpreter relied on the let, cond, if, etc. supported by the base Scheme implementation (e.g. Racket). When you loaded your Scheme interpreter into itelf and executed (repl), the let, cond, if, etc. of the second level of your interpreter relied on the let, cond, if, etc. of the first level of your interpreter, which in turn relied on the let, cond, if, etc. of the base Scheme implementation.

3. (a) In ML, why do all lists have to be homogeneous (i.e. all elements of a list must be of the same type)?

In order to perform static type checking, the compiler has to be able to determine the type of each value at compile time (even if the type of the

value contains a type variable). If lists were allowed to contain elements of different types, then for code that selects an element from a list, the compiler would not be able to determine what the type of that element is.

(b) Write a function in ML whose type is `('a -> 'b) -> ('b -> 'c) -> 'a -> 'c`.

```
fun foo f g x = g (f x)
```

(c) What is the type of the following function (try to answer without running the ML system)?

```
fun foo f (op >) (x,y) z =
   let fun bar a = if x > y then z else a
   in  bar [1,2,3]
   end
```

```
'a -> ('b * 'c -> bool) -> 'b * 'c -> int list -> int list
```

(d) Provide an intuitive explanation of how the ML type inferencer would infer the type that you gave as the answer to the previous question.

Since `f` is never used, its type is unconstrained and can be called `'a`. The only constraint on an infix parameter, such as `(op >)` is that it take a tuple of two elements and return a value. In this case, though, the output of `(op >)`, when it is used in `if x > y then ...`, must be a boolean. So, the type of `(op >)` can be inferred to be `'b * 'c -> bool`. Since `x` and `y` are used as the operands of `(op >)`, their types must be `'b` and `'c`, respectively. Since `z` and `a` are used as the two branches of the conditional within `bar`, `z` and `a` must have the same type – no matter what the type of `a` is. Since `bar` is called on an `int list` (i.e. `a` is an `int list` in that call to `bar`), and `z` must have the same type in all calls to `bar`, `z` must be an `int list`. The result type of `foo` is the result type of `bar [1,2,3]`, which is `int list`. Since the parameters to `foo` are `f`, `(op >)`, `(x,y)`, and `z`, and the return type is `int list`, the type of foo is `'a -> ('b * 'c -> bool) -> 'b * 'c -> int list -> int list`.

4. Consider the following package specification for an Ada package that implements a queue of integers.

```
package queue is
  function extract return integer;
  function insert(x: integer);
end queue;
```

(a) Why would this package not be said to implement an abstract data type (ADT) for a queue?

An ADT must be a type and, here, a queue is not a type (it's a package, or at least some data structure implemented in the package body). That is, you can't declare variables of type `queue`.

(b) Modify the above package specification, and implement a simple package body (that performs no error checking), so that a queue is an ADT.

```
package queue_package is
   type queue is private;
   function extract(q: in out queue) return integer;
   procedure insert(x: Integer; q: in out queue);
private
   type Queue_Array is array(1..100) of Integer;
   type queue is record
      arr: Queue_Array;
      head_index: Integer range 1..100;
      tail_index: Integer range 1..100;
   end record;
end queue_package;

package body queue_package is

-- not really operational, but gives the idea.

  function extract(q: in out queue) return integer is
    x: integer;
  begin
    x := q.arr(q.head_index);
    q.head_index := q.head_index+1;
    return x;
  end extract;

  procedure insert(x: Integer; q: in out queue) is
    begin
      q.arr(q.tail_index) := x;
      q.tail_index := q.tail_index - 1;
    end insert;

end queue_package;
```

5. (a) As discussed in class, what are the three features that a language must have in order to considered object oriented?

    - **Encapsulation of data and code (methods).**
    - **Inheritance.**
    - **Subtyping with dynamic dispatch.**

   (b) i. What is the "subset interpretation of suptyping"?
   It's the interpretation of subtyping such that the set of values defined by a subtype of a parent type is a subset of the set of values defined by the parent type. That is, if type B is a subtype of type A, then any value in the set of values defined by type B is also in the set of values defined by type A.

   ii. Provide an intuitive answer, and give an example, showing why class derivation in Java satisfies the subset interpretation of subtyping.

A class with certain properties (fields and methods) can be thought of as defining the set of all values that have at least those properties. Since any subclass derived from that orginal class also has at least those properties, the values in the set defined by the subclass will also be in the set defined by the original class. For example, given

```
class A {
  int x;
  int value() { return x+1; }
}
```

class A can be thought of as defining the set of all objects with at least an integer `x` field and a method `value()` that returns an integer. The definition

```
class B extends A {
  int y;
  int value() { return x+y+1; }
}
```

defines a subclass B of A, which can be thought of as defining the set of all objects with at least an integer `x` field, a method `value()` that returns an integer, and an integer y field. This set (defined by class B) is clearly a subset of the set defined by class A.

iii. Provide an intuitive answer, and give an example, showing why subtyping of functions in Scala satisfies the subset interpretation of subtyping.

As discussed in class, function subtyping is contravariant in the input type. Because a variable of type `A->Integer` can be assigned any function of type `T->Integer`, for any supertype `T` of `A` (including `A` itself), the type `A->Integer` can be thought of as defining the set of all functions that take any supertype of `A` (including `A` itself) and return an Integer. If `B` is a subtype of `A`, then type `B->Integer` defines the set of all functions that take any supertype of `B` (including `B` itself) and return an `Integer` – which is clearly a superset of the set defined by `A->Integer`.

Also as discussed in class, function subtyping is covariant in the output type. Because a variable of type `Integer->A` can be assigned any function of type `Integer->T`, for any subtype `T` of `A` (including A itself), the type `Integer->A` can be thought of as defining the set of all functions that take an Integer and return a value of a subtype `T` of `A` (including `A` itself). If `B` is a subtype of A, then type `Integer->B` defines the set of all functions that take an Integer and return a value of a subtype of `B` (including `B` itself) – which is clearly a subset of the set defined by `Integer->A`.

An example showing the subtyping of functions in Scala is the following.

```
class A(z: Integer) {
  val x = z
}

class B(z: Integer, w: Integer) extends A(z) {
  val y = w
}
```

```
object hw2 {

  def fab(a: A): B = new B(a.x, 3)  //of type A->B
  def fba(b: B): A = new A(b.x+b.y)  //of type B->A

  var ba: B=>A = fab   //fine, since A->B is a subtype of B->A
  var ab: A=>B = fba   //error, since B->A is not a subtype of A->B

  def main(args: Array[String]) {
  }
}
```

(c) Consider the following Scala definition of a tree type, where each node contains a value.

```
abstract class Tree[T <: Ordered[T]]
case class Node[T <: Ordered[T]](v:T, l:Tree, r:Tree) extends Tree[T]
case class Leaf[T <: Ordered[T]](v:T) extends Tree[T]
```

Ordered is a built-in trait in Scala (see `http://www.scala-lang.org/api/current/index.html#scala.math.Ordered`). Write a Scala function that takes a Tree[T], for any ordered T, and returns the maximum value in the tree. Be sure to use good Scala programming style.

```
//helper function
def max[T <: Ordered[T]](a: T, b: T, c: T) = {
  if (a > b)
    if (a > c) a else c
 else
  if (b > c) b else c
}

def maxTree[T <: Ordered[T]](t: Tree[T]): T = t match {
  case Leaf(value) => value
  case Node(value, left, right) => max(value, maxTree(left), maxTree(right))
}
```

(d) In Java generics, subtyping on instances of generic classes is invariant. That is, two different instances `C<A>` and `C<B>` of a generic class `C` have no subtyping relationship, regardless of a subtyping relationship between `A` and `B` (unless, of course, A and B are the same class).

i. Write a function (method) in Java that illustrates why, even if `B` is a subtype of `A`, `C<B>` should not be a subtype of `C<A>`. That is, write some Java code that, if the compiler allowed such covariant subtyping among instances of a generic class, would result in a run-time type error.

```
//Assuming B is a subclass of A.

void f(ArrayList<A> L) {
  A a = new A;
  L.add(a);  //if L is actually an ArrayList<B>, this
             //would stick an A object into L, which
```

```
            //would create an error if L's elements
            //are treated elsewhere as Bs.
}
```

ii. Modify the code you wrote for the above question that illustrates how Java allows a form of polymorphism among instances of generic classes, without allowing subtyping. That is, make the function you wrote above be able to be called with many different instances of a generic class.

```
void f(ArrayList<? super A> L) {
  A a = new A;
  L.add(a);   //Adding an A to an ArrayList of elements of a
              //supertype of A (or A itself). That's fine.
}
```

(e) i. In Scala, write a generic class definition that supports covariant subtyping among instances of the class. For example, define a generic class C[E] such that if class B is a subtype of class A, then C[B] is a subtype of C[A].
In the definition of the generic class C[E], you have to be careful that type E is only used in covariant position (as field types or the the output types of methods, not as variables to be assigned to or as values to be inserted).

```
class C[+E](x:E) {
  val y:E = x
  def value():E = y
}
```

ii. Give an example of the use of your generic class.

```
class A {}

class B extends A {}

object foo2 {

  def test(m: C[A]):A = m.value()

  def main(args: Array[String]) {
    val L1 = new C[A](new A)
    val a1: A = test(L1)
    println(L1)

    val L2 = new C[B](new B)
    val a2: A = test(L2)   //use of coviarant subtyping
    println(L2)
  }
}
```

(f) i. In Scala, write a generic class definition that supports contravariant subtyping among instances of the class. For example, define a generic class C[E] such that if class B is a subtype of class A, then C[A] is a subtype of C[B].

In the definition of the generic class C[E], you have to be careful that type E is only used in contravariant position (as the input types of methods and not as the output types nor as the types of fields).

```
class C[-T]() {

  var v: List[Any] = List()

  def insert(x: T) {
    v = x::v
  }
}
```

ii. Give an example of the use of your generic class.

```
class A {}

class B extends A {}

object foo {

  def test(m: C[B], z: B) {
    m.insert(z)
  }

  def main(args: Array[String]) {
    val L1 = new C[B]()
    test(L1, new B)

    val L2 = new C[A]()
    test(L2, new B)   //use of contravariant subtyping in first argument
  }
}
```

6. (a) What is the advantage of a mark-and-sweep garbage collector over a reference counting collector? **Mark-and-sweep GC is able to collect cyclical structures that are no longer reachable. Reference counting will not collect cycles, even if they are not accessible by the program.**

(b) What is the advantage of a copying garbage collector over a mark and sweep garbage collector?

Copying GC performs compaction of the live objects, so that a simple heap pointer can be used for storage allocation (no free list needed). The cost of copying GC is also proportional to the total size of live objects, whereas the cost of a mark-and-sweep GC is proportional to total size of the heap.

(c) Write a brief description of generational copying garbage collection.

Generational GC uses a series of heaps, each heap representing a "generation" of objects. All new objects are allocated in the "youngest" heap. When the youngest heap fills up, a copying collector copies all live objects to the heap representing the next generation (the second youngest generation). At this point, the youngest heap is empty and the program can continue exe-

cuting. When the second youngest generation fills up (as a result of copying from the youngest generation during GC), the live objects are copied from the second youngest generation to the next generation (the third youngest), and so forth.

The advantage of generational copying GC is that GC is rarely performed on older generations, which contain objects that have survived multiple GCs and are highly likely to be live. The youngest generation undergoes the most frequent GC, but the youngest objects are generally temporary and are unlikely to be live when GC occurs. Since the cost of copying GC is proportional to the total size of the live objects encountered, the cost of performing GC on the youngest generation will be low.

(d) Write, in the language of your choice, the procedure `delete(x)` in a reference counting GC system, where `x` is a pointer to a structure (e.g. object, struct, etc.) and `delete(x)` reclaims the structure that `x` points to. Assume that there is a free list of available blocks and `addToFreeList(x)` puts the structure that `x` points to onto the free list.

```
void delete(obj *x)
{
  x->refCount--;
  if (x->refCount == 0) {
    for(i=0; i < x->num_children; i++)
        delete(x->child[i]);  //assuming each child is a pointer.
    addToFreeList(x);
  }
}
```