CSE 341 - Programming Languages
Final Exam - Autumn 2005 - Answer Key

Open book and notes. No laptop computers, PDAs, or similar devices. (Calculators are OK, although you won't need one.) Please answer the problems on the exam paper — if you need extra space use the back of a page.

110 minutes, 100 points total

1. (14 points)

   Suppose the following ML functions and exception have been defined:

```
exception UnequalLengthLists;

fun zip2 ([], []) = []
| zip2 ((x::xs), (y::ys)) = (x,y) :: zip2 (xs, ys)
| zip2 _ = raise UnequalLengthLists;

fun zip2curried [] [] = []
| zip2curried (x::xs) (y::ys) = (x,y) :: zip2curried xs ys
| zip2curried _ _ = raise UnequalLengthLists;

fun average (x,y) = (x+y)/2.0;
```

   What is the *value* of each of the following expressions? If evaluating it raises an exception, say so. (Hint: this code uses the built-in function `map` in ML, which is curried.)

   (a)
```
zip2([1.0, 2.0, 3.0], [3.0, 4.0, 5.0])
  [(1.0,3.0),(2.0,4.0),(3.0,5.0)]
```

   (b)
```
map average (zip2([1.0, 2.0, 3.0], [3.0, 4.0, 5.0]))
[2.0,3.0,4.0]
```

   (c)
```
zip2([1.0, 2.0, 3.0], [3.0, 4.0])
uncaught exception UnequalLengthLists
```

   What is the *type* of each of the following expressions? Some of them may give type errors — if so, say that.

   (a)
```
zip2
fn : 'a list * 'b list -> ('a * 'b) list
```

   (b)
```
zip2curried
fn : 'a list -> 'b list -> ('a * 'b) list
```

   (c)
```
zip2 average
type error
```

   (d)
```
map average
fn : (real * real) list -> real list
```

2. (8 points - 2 points each for parts a and b; 4 points for part c) What is the result of evaluating each of the following Scheme expressions? If there is more than one expression, just give the result of evaluating the final expression.

   (a)
```
(define a 3)
(define b 4)
(define c 5)
```

```
(let ((b (+ 2 4))
      (c (+ b 10)))
   (+ a b c))

23
```

(b)
```
(define a 3)
(define b 4)
(define c 5)
(let* ((b (+ 2 4))
       (c (+ b 10)))
   (+ a b c))

25
```

(c)
```
(define z 'y)
(define y 'x)
(define x 42)
(define w 'w)
(list
   z
   (eval z)
   (eval (eval z))
   (eval (eval (eval z)))
   w
   (eval w)
   (eval (eval w))
   (eval (eval (eval w))))


(y x 42 42 w w w w)
```

3. (6 points)

   (a) What is the result of evaluating the following expressions in Scheme? (Just give the value of the final expression.)

```
(define y 3)
(define (f x) (+ x y))
(let ((x 10)
      (y 20))
  (f 100))

103
```

   (b) Suppose Scheme used dynamic scoping rather than lexical scoping. In that case, what would be the value of the final expression?

   120

   (This time, when evaluating the function f, x is bound locally to 100 as before, but we search dynamically for y, and find the binding of y to 20.)

4. (12 points) Write a Scheme definition for my-cond, a macro for the built-in Scheme "cond". Use define-syntax. The definition for cond is as follows:

```
(cond <clause1> <clause2> ...)
```

Syntax: Each <clause> should be of the form

```
(<test> <expression1> ...)
```

where <test> is any expression. The last <clause> may be an "else clause," which has the form

```
(else <expression1> <expression2> ...).
```

Semantics: A cond expression is evaluated by evaluating the <test> expressions of successive <clause>s in order until one of them evaluates to a true value. When a <test> evaluates to a true value, then the remaining <expression>s in its <clause> are evaluated in order, and the result of the last <expression> in the <clause> is returned as the result of the entire cond expression. If the selected <clause> contains only the <test> and no <expression>s, then the value of the <test> is returned as the result. If there are no true <clause>s, cond should return nothing. In your macro, you can get this result with the expression `(void)`.

Macros: As a refresher on macro syntax, here are a couple examples we saw in class:

```
(define-syntax my-if              ; macro name
  (syntax-rules (then else)       ; literals it uses, if any
    ((my-if e1 then e2 else e3)   ; pattern
     (if e1 e2 e3))))             ; template

(define-syntax my-or
  (syntax-rules ()
    ((my-or) #f)
    ((my-or e) e)
    ((my-or e1 e2 ...)
     (let ((temp e1))
       (if temp
           temp
           (my-or e2 ...))))))
```

Answer:

```
(define-syntax my-cond
  (syntax-rules (else)
    ((my-cond) (void))
    ((my-cond (else e1 e2 ...)) (begin e1 e2 ...))
    ((my-cond (e1 e2 ...) rest ...)
       (let ((temp e1))
         (if temp
             (begin temp e2 ...)
             (my-cond rest ...))))))
```

5. (6 points)

   (a) Supose that you have a global variable `k` declared in Scheme:

```
(define k #f)
```

What is the value returned by evaluating the following Scheme expression?

```
(+ 10
   (call/cc
     (lambda (c)
       (set! k c)
       (* 5 6)))
   20)
```

60

(b) After the above expression has been evaluated and `k` reassigned, what is the value returned by this expression?

```
(k 100)
```

130

(c) And this one?

```
(* 3 (k 200))
```

230

6. (12 points) Tacky but easy-to-grade true/false questions!

   (a) An advantage of static typing over dynamic typing is that it is more flexible — more programs can execute correctly. **False**

   (b) An advantage of static typing over dynamic typing is that more errors can be caught at compile time rather than run time. **True**

   (c) An advantage of static typing over dynamic typing is that type declarations provide machine-checkable documentation. **True**

   (d) After a continuation has been resumed in Scheme, it is used up — it can't be resumed again. **False**

   (e) In a pure functional language, any function will return the same answers if call-by-name is used instead of lazy evaluation. **True**

   (f) In a pure functional language, any function will return the same answers if call-by-name is used instead of call-by-value. **False**

7. (6 points) Discuss two reasons **inner classes** are useful for defining iterators in Java.

   An inner class is a class declared within the scope of another class. One reason they are useful for defining Java iterators is that an instance of a (non-static) inner class has direct access to the fields of the parent instance. These fields would hold the state of the collection. Another reason is that the name of the iterator is known only within the parent class, so that it doesn't clutter up the package name space.

8. (8 points) Consider the following Smalltalk class definitions.

```
Object subclass: #Animal
  classVariableNames: ''
  poolDictionaries: ''

describe
  Transcript show: 'An animal'.
```

```
  self additionalDescription.

additionalDescription
  Transcript show: ' - no further information available'.
```

```
Animal subclass: #SeaCreature
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

additionalDescription
  Transcript show: ' that lives in the sea'.
  Transcript cr.
  self yetMoreDescription.

yetMoreDescription
  Transcript show: 'Also a source of 341 variable names'.
```

```
SeaCreature subclass: #Octopus
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''


additionalDescription
  Transcript show: ' that has tentacles and'.
  super additionalDescription.
```

```
Octopus subclass: #GiantOctopus
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

yetMoreDescription
  Transcript show: 'It can grow up to 300 pounds'.
```

What is printed when each of the following expressions is evaluated?

- `Animal new describe`

  `An animal - no further information available`

- `SeaCreature new describe`

  `An animal that lives in the sea`

```
Also a source of 341 variable names
```

- `Octopus new describe`

```
An animal that has tentacles and that lives in the sea
Also a source of 341 variable names
```

- `GiantOctopus new describe`

```
An animal that has tentacles and that lives in the sea
It can grow up to 300 pounds
```

9. (16 points)

Consider the following Java code fragments. In each case, does the code compile correctly? If so, does it execute without error, or is there an exception? (Hints: the `List` interface includes an `add` method. The class `ArrayList` implements the `List` interface.)

(a)
```
List<String> s = new ArrayList<String>();
s.add("squid");
```
Executes without error. (`ArrayList<String>` is a subtype of `List<String>`.)

(b)
```
ArrayList<Object> s = new ArrayList<String>();
s.add("squid");
```
Compile time error. (`ArrayList<String>` is *not* a subtype of `ArrayList<Object>`.)

(c)
```
ArrayList<String> s = new ArrayList<Object>();
s.add("squid");
```
Compile time error. (`ArrayList<Object>` is certainly not a subtype of `ArrayList<String>`.)

(d)
```
String [] s = new String [10];
s[0] = "squid";
```
Executes without error.

(e)
```
Object [] s = new String[10];
s[0] = "squid";
```
Executes without error. Arrays obey the covariant typing rule, so that in this case `String[]` *is* a subtype of `Object[]`.

(f)
```
String [] s = new Object[10];
s[0] = "squid";
```
Compile time error. `Object[]` is not a subtype of `String[]`.

(g)
```
String [] s = new String [10];
s[0] = "squid";
s[1] = new Point(10,20);
```
Compile time error. Since `Point` is not a subtype of `String`, the Java compiler flags the store into `s[1]` as a type error.

(h)
```
Object [] s = new String[10];
s[0] = "squid";
s[1] = new Point(10,20);
```
Run time exception. This type checks, since `String[]` *is* a subtype of `Object[]`. However, the array is noted at runtime as holding strings, so the assignment of a point to `s[1]` raises an ArrayStoreException.

10. (12 points) Suppose that we have two definitions of a static method `shapes` in Java:

```
public static boolean shapes1
    (ArrayList<? extends RectangularShape> s) {
     ....
}


public static boolean shapes2
    (ArrayList<RectangularShape> s) {
     ....
}
```

Suppose we also have some variables declared:

```
ArrayList<RectangularShape> a;
ArrayList<Rectangle2D> b;
ArrayList<Ellipse2D> c;
ArrayList<Object> d;
```

(`Rectangle2D` and `Ellipse2D` are both subclasses of `RectangularShape`.)

(a) Which of the following method calls are legal? Circle the legal ones:

```
shapes1(a);  LEGAL

shapes1(b);  LEGAL

shapes1(c);  LEGAL

shapes1(d);  NOT LEGAL

shapes2(a);  LEGAL

shapes2(b);  NOT LEGAL

shapes2(c);  NOT LEGAL

shapes2(d);  NOT LEGAL
```

(b) Within the method `shapes1`, would it be legal to add a new `Rectangle2D` to the array list `s`?
No - because we could be passing in an array of ellipses instead (for example).

(c) Within the method `shapes2`, would it be legal to add a new `Rectangle2D` to the array list `s`?
Yes.