CS 345 - Programming Languages
Fall 2010

MIDTERM #2

November 9, 2010

DO NOT OPEN UNTIL INSTRUCTED

YOUR NAME: ______________________________

Collaboration policy

No collaboration is permitted on this midterm. Any cheating (*e.g.*, submitting another person's work as your own, or permitting your work to be copied) will automatically result in a failing grade. The Computer Sciences department code of conduct can be found at `http://www.cs.utexas.edu/academics/conduct/`.

Midterm #2 (85 points)

Problem 1 (15 points)

Circle only <u>one</u> of the choices **(3 points each)**.

1. **TRUE** **FALSE** Mark-sweep garbage collection is incremental, performed every time a reference is updated.
2. **TRUE** **FALSE** With polymorphic functions in ML, a separate copy of the function is generated for each type with which the function is used.
3. **TRUE** **FALSE** Each Java object is associated with a monitor.
4. **TRUE** **FALSE** Deadlock cannot occur in a Java program that does not use synchronization.
5. **TRUE** **FALSE** Closures are necessary in any Scheme implementation.

Problem 2 (20 points)

Define the following terms:

Overloading:

Parametric polymorphism:

Race condition:

No-Side-Effects (Declarative, Pure Functional) Language Test:

Horn clause:

Problem 3 (6 points)

Which two features of functional programming languages are highlighted by John Hughes as contributing significantly to modularity?

Which of these features is not supported by Scheme?

Problem 4

Consider the following recursively defined Scheme function, where `list2` is a function that returns a 2-element list:

```
(set zip (lambda (l1 l2)
     (if (or (null? l1) (null? l2)) nil
         (cons (list2 (car l1) (car l2))
               (zip   (cdr l1) (cdr l2))))))
```

The `zip` function takes two lists and returns a list of 2-element lists. For example,

```
(zip '(3 4 5) '(hi there sue sam)) => '((3 hi) (4 there) (5 sue))
```

Problem 4a (6 points)

Write `zip` in ML using pattern matching. The result should be a list of 2-element tuples. You may assume that the input lists are of equal length. Use the following implementation of `length` as a guide:

```
fun length [] = 0
  | length (x::xs) = 1 + length xs;
```

Problem 4b (6 points)

The type of `length` is `'a list -> int`. What is the type of `zip`?

Problem 5 (5 points)

Describe how reference counting could be used for garbage collection in evaluating the following Scheme expression:

```
(car (cdr (cons (cons a (cons b c)) (cons d e))))
```

where `a,b,c,d,e` are previously defined names for cells whose reference counts are greater than 0 (*i.e.*, they do not become garbage). Assume that the final result of evaluation is not garbage, either. How many of the four `cons` cells can be garbage-collected?

Problem 6

Problem 6a (8 points)

Evaluate the following Scheme expressions:

```
(car (car (cdr (cdr (a b (c d) e (f g))))))
```

```
((lambda (f x y) (f x y)) * 2 (+ 3 2))
```

Problem 6b (8 points)

Redefine the following `let` and `let*` expressions using `lambda`, and evaluate the resulting `lambda` expressions.

```
(define x 4)
(define y 2)
(let  ((x 6) (y x) (z (+ x y))) (+ x y z))
```

```
(let* ((x 6) (y x) (z (+ x y))) (+ x y z))
```

Problem 6c (6 points)

Rewrite the following function using `foldl/foldr`. **Be sure that the order of the result list is the same as for the original function**. You can assume a function `reverse` is already defined if you need it.

```
(define (map f lst)
    (if (empty? lst) '()
        (cons (f (car lst)) (map f (cdr lst))))))
```

Problem 7 (5 points)

Consider the following Prolog implementation of `append`:

```
append([X|Xs], Ys, [Z|Zs]) :- append(Xs,Ys,Zs), X=Z.
append([], Ys, Ys).
```

Why is this implementation potentially problematic?