CSSE304 Exam 1 Paper Review

Instructions

Please complete the following problems. If you have questions come up and ask. I will send out an answer key after the review session is over.

Conceptual Questions

1. **Problem 1:** Describe the Racket programming language and explain some of its use cases. Racket is a multi-paradigm, recursive heavy language, Meaning it supports imperative functional, and OO programming. Racket is used for creating domain specific languages, It is also used for scripting.

2. **Problem 2:** Describe a meta-function and give an example of a use case of one.
Meta-functions are procedures that operate on, generate, or transform other functions.

An example would be $n listrecur in your notes

3. **Problem 3:** What is a first class procedure?
a first class procedure is where procedures are treated like all other data. So they can be arguments to or returned from procedures.

4. **Problem 4:** What does it mean if a procedure is "Short Circuiting"?
If the function reaches an end condition it will stop running and return. It won't run code that comes after the condition causing the "short circuit".

5. **Problem 5:** How is Racket scoped? Explain what that means in terms of running code.
Racket is statically or lexically scoped. This means that when running code Racket will search up the known environments and find the most recent binding of a variable.

Code Based Questions

1. **Problem 1:** Create the contour diagram for the following code snippet:

```
(define silly-proc
  (lambda (a b c)
    (let ([d (* a c)])
      (lambda (e)
        (let ([a (let ([a (+ d a e)]
                       [f 11])
                   (+ a f c))])
          (+ e a c)))))))
```

2. **Problem 2:** What does running (silly-proc 1 2 3) return?

A procedure

3. **Problem 3:** Evaluate the following code. Show the values stored in each let and lambda expression.

```
(define silly-proc
  (lambda (a b c) 3,4,5
    (let ([d (* a c)]) 15
      (lambda (e) 2
        (let ([a (let ([a (+ d a e)] 20
                       [f 11]) 11
                   (+ a f c))])
          43 (+ e a c)))))))

((silly-proc 3 4 5) 2)
```

43

4. **Problem 4:** Convert the following let-expression to a lambda-application expression and evaluate its output.

```
((lambda (x)
   (let ([a (+ x 1)])
     (let ([b (+ a 2)])
       ((lambda (y)
          (+ a b y)) b)))) 5)
```

((lambda (x)
((lambda (a)
((lambda (b)
((lambda (y)
(+ a b y))
b))
(+ a 2)))
(+ x 1)))
5)

22

5. **Problem 5:** Write a function find-valid-data that takes in a list of lists of numbers and returns a list of only the positive averages. Use some combination of map, apply, and filter to complete the problem (no explicit recursion).

(find-valid-data '((1 2 3) (-2 -4 -2) (0 3 5 4))) -> '(2 3)

```
(define find-valid-data
 (lambda (lst)
  (filter positive?
    (map (lambda (internal-lst)
           (/ (apply + internal-lst) (length internal-lst)))
         lst))))
```

6. **Problem 6:** Here is some code that uses let and lambda to show closures. Write what will be displayed by the code when it is run.

```
(define factory
  (let ((global-val 100))
    (lambda ()
      (let ((local-val 0))
        (lambda ()
          (let ((temp-val 5))
            (set! global-val (- global-val 5))
            (set! local-val (+ local-val 10))
            (set! temp-val (+ temp-val 1))
            (display (list global-val local-val temp-val))
            (newline)))))))

(define f1 (factory))
(define f2 (factory))
(f1)
(f1)
(f2)
(f1)
```

'(95 10 6)
'(90 20 6)
'(85 10 6)
'(80 30 6)

7. **Problem 7:** Use the following grammar and expression to create a grammar tree. Use E for expr, B for base_val, S for symbol, and I for integer.

```
<expr> ::= (<symbol> <expr> <expr>) | [<expr> <expr>]
         | <base_val>
<base_val> ::= <integer> | <symbol>

(+ a [3 (* 4 5)])
```