**CSSE 304 Final Exam Part 2 Wednesday evening, Feb 20, 2019 Name\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Allowed resources:** The interpreter printouts that you brought to the exam. No other papers, earbuds, electronic devices, etc.

You must submit this page and your interpreter printouts before you leave the exam room. **Failure to do so will result in an F for the course and a report of academic misconduct to the dean of students. You must also submit it before you use your computer, notes, textbook, etc.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Problem** | **Max score** | **Your score** | **Comments** |
| trace | 1 | 15 |  |  |
| escaper | 2 | 10 |  |  |
|  | **Total** | **25** |  |  |

**Summary: For each problem, please annotate your interpreter printouts with the code changes and/or additions necessary to add this interpreter feature.**

1. Annotate your A17 printout using a **red** pen. Or write in pencil and circle with a red pen.
2. Annotate your A18 printout using a **red** pen. Or write in pencil and circle with a red pen.  
   **---------------------------------------------------------**
3. **(15 points)** Use your A17 interpreter as the starting point for this problem. **Use a red pen, or write in pencil and circle your annotations with a red pen.**

Add trace to the language that your interpreter interprets. (trace proc) causes future applications of the globally-defined procedure proc to be traced. Restrictions to make your code easier to write:

* Your trace syntax is only required to be usable at the top level, as we have typically used it in class.
* You do not have to support the (trace proc1 proc2 proc3) syntax; the same effect can be obtained by three separate calls to trace.
* You do not have to implement untrace.
* You do not have to detect tail-recursion and suppress additional indentation when tail calls of traced procedures happen.

You must support tracing of both user-defined procedures and primitive procedures.   
**Hint: Add to your interpreter code a global variable that keeps track of the current level of indentation.** The value of that variable can be incremented when a traced procedure is applied and decremented when a traced procedure returns.

The output printed when a traced procedure runs does not have to exactly match what Scheme’s trace procedure would output, but it should

(a) when a traced procedure is called, print the name of the procedure and the values of its parameters.

(b) when the traced procedure returns, print the return value.

(c) the indentation level of what gets printed for (a) and (b) should be the same.

(d) A traced call during execution of another traced call should cause more indentation.

So that your output can look a lot like Scheme’s (but not exactly like it), I have provided a procedure called display-traced-output which you can call when you need to produce a single line of tracing output. You will probably want to call it like this:

* when entering a traced procedure (call it with 3 arguments: indent-level proc-name args) and
* when returning from a traced procedure (call it with 2 arguments: indent-level value).

> **(display-traced-output 5 'f '(a b c))**

| | | | | (f a b c)

> **(display-traced-output 3 4)**

| | | 4

You do not have to understand the details of this procedure, but I am providing the code just in case it helps you. You can simply call it when you need it.

**(define display-traced-output**

**(let ([multi-indent-string**

**(lambda (level)**

**(let loop ([level level] [result ""])**

**(if (zero? level)**

**result**

**(loop (- level 1) (string-append result "| ")))))])**

**(lambda args** ; args will be (level proc-name args) or (level answer)

**(let ([indent-string (multi-indent-string (car args))])**

**(display indent-string)**

**(display (if (= 2(length args))**

**(cadr args)**

**(cons (cadr args) (caddr args)))))**

**(newline))))**

The following examples were run in *Chez* Scheme, not in a CSSE 304 interpreter; If you could enter the same code into your interactive interpreter after running (rep), the results should be similar. If you call my display-traced-output procedure, there will be a vertical line for each level of indentation, where *Chez* Scheme only prints a vertical line for every two levels of indentation.

> (define aa (lambda (n) (+ 1 n)))

> (define bb (lambda (n m) (+ m (aa n))))

> (define cc (lambda (n) (+ n (bb n (aa n)))))

> (trace aa)

(aa)

> (trace bb)

(bb)

> (trace cc)

(cc)

> (cc 3)

|(cc 3)

| (aa 3)

| 4

| (bb 3 4)

| |(aa 3)

| |4

| 8

|11

11

> (trace +)

(+)

> (define aa (lambda (n) (+ 1 n)))

> (define bb (lambda (n m) (+ m (aa n))))

> (define cc (lambda (n) (+ n (bb n (aa n)))))

> (trace aa)

(aa)

> (trace bb)

(bb)

> (trace cc)

(cc)

> (cc 3)

|(cc 3)

| (aa 3)

| (+ 1 3)

| 4

| (bb 3 4)

| |(aa 3)

| |(+ 1 3)

| |4

| (+ 4 4)

| 8

|(+ 3 8)

|11

11

>

> (define tree-mult

(lambda (ls)

(cond

[(null? ls) 1]

[(not (list? (car ls)))

(\* (car ls) (tree-mult (cdr ls)))]

[else (\* (tree-mult (car ls)) (tree-mult (cdr ls)))])))

> (trace tree-mult)

(tree-mult)

> (tree-mult '((1 2) (() (3) 4)))

|(tree-mult ((1 2) (() (3) 4)))

| (tree-mult ((() (3) 4)))

| |(tree-mult ())

| |1

| |(tree-mult (() (3) 4))

| | (tree-mult ((3) 4))

| | |(tree-mult (4))

| | | (tree-mult ())

| | | 1

| | |4

| | |(tree-mult (3))

| | | (tree-mult ())

| | | 1

| | |3

| | 12

| | (tree-mult ())

| | 1

| |12

| 12

| (tree-mult (1 2))

| |(tree-mult (2))

| | (tree-mult ())

| | 1

| |2

| 2

|24

24

**2. (10 points)** escaper is a procedure that can be added to Scheme. It was described in several different class meetings. ((escaper f) x y z) returns the same value that (f x y z) returns, but it returns that value as the final answer for the entire expression being evaluated. Modify your **A18** interpreter code to add escaper to the interpreted language. Show the code that you would add and/or change in order to do this. **Use a red pen or write with a pencil and circle your annotations with a red pen.**

(eval-one-exp '(+ 5 ((escaper \*) 6 7))) 🡺 42

(eval-one-exp '(procedure? escaper)) 🡺 #t

(eval-one-exp '(let ([escape-\* (escaper \*)])

(- 6 (\* 4 (\* 2 3)))))) 🡺 -18

(eval-one-exp '(procedure? (escaper +))) 🡺 #t  
  
(eval-one-exp '(procedure? ((escaper +) 3 5)) 🡺 8

(eval-one-exp '(let ([escape-\* (escaper \*)])

(- 6 (escape-\* 4 (escape-\* 7 8)))))) 🡺 56

(eval-one-exp ' (let ([xyz (escaper (lambda (x y)

(\* x (- y 2))))])

(+ 7 (xyz 5 9)))) 🡺 35