**CSSE 304 Final Exam Part 3 (computer/interpreter)** Nov 18, 2019 **(Monday evening) Name\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

You may not use code written by other CSSE 304 students (from this term or any term), except code that you and your partner wrote for the project. **No headphones or earbuds.**

You may receive some partial credit based on your code and comments. If your code passes the test cases but does not implement the required features according to the specification, you may receive less than full credit.   
**Caution!**  Do not write code that is specific to my test-cases but does not solve the general problem.

You can use any course materials posted on or linked from Moodle or the schedule page, the textbooks, your printed notes, and code that you wrote before the exam.

**Please submit this paper as soon as you have finished this part.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Max score** | **Your score** | **Instructor/grader Comments** |
| P - int | 30 |  |  |

The following is an excerpt from *The Scheme Programming Language*:

A case-lambda expression consists of a set of clauses, each resembling a lambda expression. Each *clause* has the form below.

[*formals* *body1* *body2* ...]

The formal parameters of a clause are defined by *formals* in the same manner as for a lambda expression. The number of arguments accepted by the procedure value of a case-lambda expression is determined by the numbers of arguments accepted by the individual clauses.

When a procedure created with case-lambda is invoked, the clauses are considered in order. The first clause that accepts the given number of actual parameters is selected, the formal parameters defined by its *formals* are bound to the corresponding actual parameters, and the body is evaluated as described for lambda above. If *formals* in a clause is a proper list of identifiers, then the clause accepts exactly as many actual parameters as there are formal parameters (identifiers) in *formals*. As with a lambda *formals*, a case-lambda clause *formals* may be a single identifier, in which case the clause accepts any number of arguments, or an improper list of identifiers terminated by an identifier, in which case the clause accepts any number of arguments greater than or equal to the number of formal parameters excluding the terminating identifier. If no clause accepts the number of actual parameters supplied, an exception with condition type &assertion is raised.

The following definition for make-list uses case-lambda to support an optional fill parameter.

(define make-list  
  (case-lambda  
    [(n) (make-list n #f)]  
    [(n x)  
     (do ([n n (- n 1)] [ls '() (cons x ls)])  
         ((zero? n) ls))]))

There are more case-lambda examples in the Day 8 slides (near the end).

The substring procedure may be extended with case-lambda to accept either no *end* index, in which case it defaults to the end of the string, or no *start* and *end* indices, in which case substring is equivalent to string-copy:

(define substring1  
  (case-lambda  
    [(s) (substring1 s 0 (string-length s))]  
    [(s start) (substring1 s start (string-length s))]  
    [(s start end) (substring s start end)]))

Additional case-lambda examples are in the Day 8 slides.  
Add case-lambda to the language interpreted by your interpreter. 22 of the 30 points will be for case-lambda with only the “normal” proper lists of lambda arguments; the other 8 points will be for test cases that include the “special” cases (single variable and improper list of arguments). You may begin with either your A17a or A18 interpreter. A17 will probably be easier.

**Additional examples:**

**> (eval-one-exp '**

**(let ([f (case-lambda**

**[(a b) (- a b)]**

**[(a b c) (+ a b c)])])**

**(cons (f 4 1) (f 1 2 3))))**

(3 . 6)

**> (eval-one-exp '**

**(map (lambda (x) (apply**

**(case-lambda**

**[(a b) (- a b)]**

**[(a b c) (+ a b c)]**

**[() 7])**

**x))**

**'(() (10 12) (5 6) (7 8 9))))**

(7 -2 -1 24)

**> (eval-one-exp '**

**(let ([x 5])**

**(let ([y (map (begin**

**(set! x (+ 1 x))**

**(lambda (x)**

**(apply**

**(case-lambda**

**[(a b) (- a b)]**

**[(a b c) (+ a b c)]**

**[() 7])**

**x)))**

**'(() (10 12) (5 6) (7 8 9)))])**

**(cons x y))))**

(6 7 -2 -1 24)

**> (eval-one-exp '**

**((case-lambda**

**[(a b c) (- a b c)]**

**[(a b c d) (+ a b c d)]**

**[(a b . c) (+ a b (apply \* c))]**

**[a (apply \* a)])**

**3 4 5)**

**)**

-6

**> (eval-one-exp '**

**((case-lambda**

**[(a b c) (- a b c)]**

**[(a b c d) (+ a b c d)]**

**[(a b . c) (+ a b (apply \* c))]**

**[a (apply \* a)])**

**3 4 5 6)**

**)**

18

**> (eval-one-exp '**

**((case-lambda**

**[(a b c) (- a b c)]**

**[(a b c d) (+ a b c d)]**

**[(a b . c) (+ a b (apply \* c))]**

**[a (apply \* a)])**

**)**

**)**

1

**> (eval-one-exp '**

**((case-lambda**

**[(a b c) (- a b c)]**

**[(a b c d) (+ a b c d)]**

**[(a b . c) (+ a b (apply \* c))]**

**[a (apply \* a)])**

**2 3)**

**> (eval-one-exp '**

**((case-lambda**

**[(a b c) (- a b c)]**

**[(a b c d) (+ a b c d)]**

**[(a b . c) (+ a b (apply \* c))]**

**[a (apply \* a)])**

**7 6 5 4 3 2 1)**

**)**

133