CSSE 304 Exam 2 Jan 22, 2019 (day 26.5) Name_____ Section 01(9:00) 02(9:55) 03(10:50)

Part 3, programming (interpreter). For this part, you may use the resources from Part 1, plus TSPL, EoPL, your notes, and a Scheme programming environment , plus the PLC grading server and any materials that I provided online for the course. You may not use any other web or network resources. You may look at and use any Scheme code that you have previously written.

Submit your code to the E2-part3-201920 **assignment on the PLC server.**

Mutation is allowed (in fact, required). Efficiency and elegance will not affect your score.

Problem	Possible	Earned	Comments
I	12		

In the Pascal programming language, a `for` loop has two possible forms:

for <var> : = <expression> to <expression> do <statement>
for <var> : = <expression> downto <expression> do <statement>

In each form, the loop variable gets its initial value from the first expression, and then is incremented (in the first form) or decremented (in the second form) by 1 each time through the loop. In our semantics (unlike Pascal's semantics), the loop variable does not have to have been declared before the loop, so this *for* loop construct creates a binding for this variable that shadows any outer bindings of the same variable name. As in Pascal, the body of the loop is not allowed to mutate the loop variable. Some particular Pascal compilers allowed the programmer to specify that the increment should be by a different number than 1, but this was never part of the Pascal standard, as far as I know. The expression that determines the stopping condition for the loop (the one before the DO) is only evaluated once, before the loop begins.

For example,
- `for i := 3 to 7 do p(i)` calls the procedure `p` five times.
- `for i := 6 downto 3 do p(i)` calls the procedure `p` four times.
- `for i := 6 to 3 do p(i)` never calls the procedure `p`, because 6 is already larger than 3.

If we want to add this loop construct to our interpreted language, we can use the following syntax additions:

```
<expression> ::= (for <var> := <expression> to <expression> do <expression>)
             ::= (for <var> := <expression> downto <expression> do <expression>)
```

Like a `while` loop, a `for` loop is executed for effect; it does not return a value.

Example. Your interpreter is not required to execute this example, since you are not required to have `set!` working yet. But it is still an instructive example to see how `for` works.

```
> (rep)
--> (let ([i 9] [x 4])
      (begin (for i := (+ 3 5) to (+ x 10) do (set! x (+ x 2)))
             (list x i))
(18 9)
```

Continued on the back of this page

A `for` loop is not very useful unless it mutates something, and at this point, `set-car!` is the simplest way for your interpreted language to do that. So all of my test cases use `set-car!` None of the test cases use the "variable lambda" syntaxes, so it is okay if your interpreter does not implement them. Chances are good that you can start with your A13 interpreter and solve this problem, which means that you do not have to bother with `syntax-expand`.

It is conceivable that you can implement `for` as a syntax-expansion, but without `set!` in your interpreted language, I think it will be much easier to interpret a parsed `for-exp` directly in `eval-exp`.

It is likely that your new code in `eval-exp` will need to use `let*` and `extend-env`, and it certainly will need to do recursive calls (perhaps using named `let`).

While your interpreter cannot evaluate a `set!` expression in the interpreted language, your implementation code for `for` can use `set!` for the Scheme variable that represents the current value of the `for` variable.

Hint: You may need to use `extend-env` in a different circumstance than we have used it before.

You do not have to check for syntax errors; I will only put valid code in my test-cases.

---------- some of the test cases ----------

```
> (eval-one-exp
 '(let ([x (list 2)])
    (for j := -1 to 1 do
        (set-car! x (* (car x) (car x))))
    (car x)))
256

> (eval-one-exp
 '(let ([LoL (list (list 1 2 3))])
    (for i := 5 to 7 do
        (set-car! LoL (map add1 (car LoL))))
    LoL))
((4 5 6))

> (eval-one-exp
 '(let ([list-of-num (list 0)])
    (begin
      (for i := 1 to 3 do
          (set-car! list-of-num
               (+ (car list-of-num)
                  (- (* 2 i) 1))))
      list-of-num)))
(9)

> (eval-one-exp
 '(let ([list-of-num (list 0)])
    (begin
      (for i := 12 to 1 do
          (set-car! list-of-num
               (+ (car list-of-num)
                  (- (* 2 i) 1))))
      list-of-num))))
(0)

> (eval-one-exp
 '(let ([list-of-num (list 0)]
        [start 1])
    (begin
      (for i := (+ start 3) downto 1 do
          (set-car! list-of-num
               (+ (car list-of-num)
                  (- (* 2 i) 1))))
      list-of-num)))
(144)
```

```
> (eval-one-exp
 '(let ([list-of-num (list 0)]
        [i 1000])
    (begin
      (for i := 12 downto 1 do
          (set-car! list-of-num
               (+ (car list-of-num)
                  (- (* 2 i) 1))))
      (list (car list-of-num) i))))
(144 1000)

> (eval-one-exp
 '(let ([list-of-num (list 0)])
    (begin
      (for i := (let ([list-of-num (list 0)])
                  (begin
                    (for i := 5 downto 1 do
                    (set-car! list-of-num
                         (+ (car list-of-num)
                            (- (* 2 i) 1))))
                    (car list-of-num)))
          downto 1 do
          (set-car! list-of-num
               (+ (car list-of-num)
                  (- (* 2 i) 1))))
      list-of-num)))
(625)
```