MORE SCHEME AND INTERPRETERS Solutions

COMPUTER SCIENCE MENTORS

April 5, 2021 - April 8, 2021

Call expressions follow *prefix* notation, i.e. `(<operator> <operand1> <operand2> ... <operandN>)`

Evaluating a call expressions closely mirrors Python:

- Evaluate the operator, yielding a procedure p
- Evaluate each operand, each yielding a value argi
- Apply the procedure p with arguments arg1, arg2, . . . , argN

Special forms *look* like call expressions but aren't – they implement Scheme language features and follow special evaluation rules (e.g., short-circuiting).

(Aside: Note that you're free to use a special form name as a variable name, but the name will be looked up *only* in a non-operator position; when used as an operator, it will always refer to the original special form.)

Notable Special Forms:

behavior	syntax
variable assignment	`(define <variable-name> <value>)`
function defining	`(define (<function> <op1>...<opN>) <body>)`
if / else	`(if <condition> <true-expr> <else-expr>)`
if / elif / else	`(cond (<cond1> <expr1>) ... (else <else-expr>))`
and	`(and <operand1> ... <operandN>)`
or	`(or <operand1> ... <operandN>)`
quote	`(quote <operand1>)`
begin	`(begin <expr1> <expr2> ... <exprN>)`
lambdas	`(lambda (<operand1> ... <operandN>) <body>)`
let / execute many lines	`(let ((<var1> <val1>) ... (<varN> <valN>)) body)`

1 What Would Scheme Print?

1. What will Scheme output?

```
scm> (if 1 1 (/ 1 0))
```

```
1
```

If statements in Scheme have the form (if cond true-result false-result). Since 1 is a truthy value, we return the second argument, 1, and never evaluate the third argument due to short circuiting.

```
scm> (if 0 (/ 1 0) 1)
```

```
Error: Zero Division
```

Recall that 0 is a Truth-y value in Scheme. Thus (/ 1 0) evaluates to a Zero Division Error

```
scm> (and 1 #f (/ 1 0))
```

```
#f
```

And statements return the first falsey value it encounters. In this case, the second argument is #f, so it returns that and never reaches the third argument.

```
scm> (and 1 2 3)
```

```
3
```

Short-circuiting rules apply. This means that and returns the first False-y value or the last Truth-y value. In this case, the last Truth-y value is 3.

```
scm> (or #f #f 0 #f (/ 1 0))
```

```
0
```

On the other hand, or statements return the first truthy value it encounters. In this case, that is the third argument, 0, and does not evaluate anything after.

```
scm> (and (and) (or))
```

```
#f
```

The special form or without any arguments evaluates to #f. The special form and without any arguments evaluates to #t. Also, short-circuiting rules apply. This means that and returns the first False-y value or the last Truth-y value. In this case, the first False-y value is #f.

```
scm> (define a 4)
```

```
a
```

The define statement in Scheme always returns the name of the variable defined, in this case, a.

```
scm> ((lambda (x y) (+ a x y)) 1 2)
```

```
7
```

We make a new lambda that takes in two arguments, x and y. The set of parentheses around the lambda evaluates it with the arguments 1 and 2, so we evaluate the body of the lambda with arguments 1 and 2, which adds them together.

```
scm> ((lambda (x y z) (y x z)) 2 / 2)
```

```
1
```

We make a new lambda that takes in three arguments, x, y and z. The set of parentheses around the lambda evaluates it with the arguments 2, / and 2 respectively, so we evaluate the body of the lambda with x as 2, y as /, and z as 2. The body of the lambda is equivalent to (/ 2 2), and that's what gets evaluated and returned.

```
scm> ((lambda (x) (x x)) (lambda (y) 4))
```

```
4
```

We make a new lambda that takes in one argument x. The set of parentheses around the lambda evaluates it with an argument of another lambda function, so we run the body (x x) with x as the (lambda (y) 4). Recall that evaluating an operator happens by putting parentheses around the operator and passing in an operand. When we do (x x), we're evaluating the operator x with the parameter x. Therefore we're now doing (lambda (y) 4) evaluated with the argument y as (lambda (y) 4). Since we evaluate the lambda, we evaluate the body and return it, which is 4.

2. What will Scheme output?

```
scm> (define boom1 (/ 1 0))
```

Error: Zero Division

Unlike Python, when we define in Scheme, we have to evaluate the body expression and bind it to the name boom1. Evaluating the body gives us an error.

```
scm> (define boom2 (lambda () (/ 1 0)))
```

boom2

We are defining another variable boom2. The value of boom2 is a lambda expression. Note that since boom2 is only being defined, not called, we do not evaluate the body of the lambda expression. So, this line will return the function name, boom2.

```
scm> (boom2)
```

Error: Zero Division

Now we are trying to call the function boom2, and so we are evaluating the body of the lambda expression. This expression is (/ 1 0), so just as we had before, we get an error from attempting to divide by 0.

(a) Why/How are the two `boom` definitions above different?

The first line is setting boom1 to be equal to the value `(/ 1 0)`, which turns out to be an error. On the other hand, boom2 is defined as a lambda that takes in no arguments that, when called, will evaluate `(/ 1 0)`.

(b) How can we rewrite `boom2` without using the **`lambda`** operator?

```
(define (boom2) (/ 1 0))
```

boom2's initial definition defines a variable that evaluates to a function. Instead, let's define boom2 as a function. When we do this, then the body of the function is not evaluated upon definition, but rather, when we call the function.

3. What will Scheme output?

```
scm> (define c 2)

c

scm> (eval 'c)

2

scm> '(cons 1 nil)

(cons 1 nil)

scm> (eval '(cons 1 nil))

(1)

scm> (eval (list 'if '(even? c) 1 2))

1
```

2 Interpreters

The following questions refer to the Scheme interpreter. Assume we're using the implementation seen in lecture and in the Scheme project.

1. What's the purpose of the read stage in a Read-Eval-Print Loop? For our Scheme interpreter, what does it take in, and what does it return?

   The read stage returns a representation of the code that is easier to process later in the interpreter by putting it in a new data structure. In our interpreter, it takes in a string of code, and outputs a Pair representing an expression (which is really just the same as a Scheme list).

2. What are the two components of the read stage? What do they do?

   The read stage consists of

   1. The lexer, which breaks the input string and breaks it up into tokens (individual characters or symbols)
   2. The parser, which takes that string of tokens and puts it into the data structure that the read stage outputs (in our case, a Pair).

3. Write out the constructor for the Pair object the read stage creates with the input string `(define (foo x) (+ x 1))`

   Pair("define", Pair(Pair("foo", Pair("x", nil)), Pair(Pair("+", Pair("x", Pair(1, nil))), nil)))

4. For the previous example, imagine we saved that Pair object to the variable `p`. How could we check that the expression is a `define` special form? How would we access the name of the function and the body of the function?

   We could check to see that it's a define special form by checking if `p.first == "define"`.
   We could get its name by accessing `p.second.first.first` and get the body of the function with `p.second.second.first`.

5. Circle or write the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
(if 1 (+ 2 3) (/ 1 0))
```

```
scheme_eval   1  3  4  6
scheme_apply  1  2  3  4
```

6 `scheme_eval`, 1 `scheme_apply`.

```
(or #f (and (+ 1 2) 'apple) (- 5 2))
```

```
scheme_eval   6  8  9  10
scheme_apply  1  2  3   4
```

8 `scheme_eval`, 1 `scheme_apply`.

```
(define (square x) (* x x))

(+ (square 3) (- 3 2))
```

```
scheme_eval   2  5  14  24
scheme_apply  1  2   3   4
```

14 `scheme_eval`, 4 `scheme_apply`.

```
(define (add x y) (+ x y))

(add (- 5 3) (or 0 2))
```

13 `scheme_eval`, 3 `scheme_apply`.

3 Code Writing

1. Define **is**-prefix, which takes in a list p and a list lst and determines if p is a prefix of lst. That is, it determines if lst starts with all the elements in p.

```
; Doctests:
scm> (is-prefix '() '())
#t
scm> (is-prefix '() '(1 2))
#t
scm> (is-prefix '(1) '(1 2))
#t
scm> (is-prefix '(2) '(1 2))
#f
; Note here p is longer than lst
scm> (is-prefix '(1 2) '(1))
#f

(define (is-prefix p lst)




)
```

```
; is-prefix with nested if statements
(define (is-prefix p lst)
    (if (null? p)
        #t
        (if (null? lst)
            #f
            (and
                (= (car p) (car lst))
                (is-prefix (cdr p) (cdr lst))))))

; is-prefix with a cond statement
(define (is-prefix p lst)
    (cond
        ((null? p) #t)
        ((null? lst) #f)
        (else (and (= (car p) (car lst))
            (is-prefix (cdr p) (cdr lst))))))
```

2. Define `apply-multiple` which takes in a single argument function `f`, a nonnegative integer `n`, and a value `x` and returns the result of applying `f` to `x` a total of `n` times.

```
;doctests
scm> (apply-multiple (lambda (x) (* x x)) 3 2)
256
scm> (apply-multiple (lambda (x) (+ x 1)) 10 1)
11
scm> (apply-multiple (lambda (x) (* 1000 x)) 0 5)
5
```

```
(define (apply-multiple f n x)






)
```

```
(define (apply-multiple f n x)
    (if (= n 0)
        x
        (f (apply-multiple f (- n 1) x))))
```

Alternate solution:

```
(define (apply-multiple f n x)
    (if (= n 0)
        x
        (apply-multiple f (- n 1) (f x))))
```