MORE SCHEME AND INTERPRETERS

COMPUTER SCIENCE MENTORS

November 9, 2020 - November 12, 2020

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))

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

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

scm> (and 1 2 3)

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

scm> (and (and) (or))

scm> (define a 4)

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

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

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

2. What will Scheme output?

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

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

scm> (boom2)
```

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

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

3. What will Scheme output?

```
scm> (define c 2)

scm> (eval 'c)

scm> '(cons 1 nil)

scm> (eval '(cons 1 nil))

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

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?

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

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

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?

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

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

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

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

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

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

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

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

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)









)
```

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)

)
```

3. Finish the functions **max** and **max**-depth. **max** takes in two numbers and returns the larger. Function **max**-depth takes in a list lst and returns the maximum depth of the list. In a nested scheme list, we define the depth as the number of scheme lists a sublist is nested within. A scheme list with no nested lists has a **max**-depth of 0.

```
;doctests
scm> (max 1 5)
5
scm> (max-depth '(1 2 3))
0
scm> (max-depth '(1 2 (3 (4) 5)))
2
scm> (max-depth '(0 (1 (2 (3 (4) 5) 6) 7))
4

(define (max x y) ______________________________________)

(define (max-depth lst)
    (define (helper lst curr)
        (cond
            ((__________) __________________________)
            ((__________) (max __________________________________
                               __________________________________))
            (else (helper __________________________))
        )
    )
    (_____________________________)
)
```