MORE SCHEME

COMPUTER SCIENCE MENTORS 61A

April 2 to April 4, 2018

1 Scheme

1. What will Scheme output? Draw box-and-pointer diagrams to help determine this.

(a) `(cons (cons 1 nil) (cons 2 (cons (cons 3 (cons 4 5)) (cons 6 nil))))`

Solution:
`((1) 2 (3 4 . 5) 6)`

(b) `(cons (cons (car '(1 2 3)) (list 2 3 4)) (cons 2 3))`

Solution: `((1 2 3 4)2 . 3)`

(c)
```
(define a 4)
((lambda (x y) (+ a)) 1 2)
```

Solution:
`4`

(d) `((lambda (x y z) (y x)) 2 / 2)`

Solution:
`0.5`

(e) `((lambda (x) (x x)) (lambda (y) 4))`

Solution: 4

(f) `(define boom1 (/ 1 0))`

Solution: Error: Zero Division

(g) `boom1`

Solution: Error: boom1 not defined

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

Solution: boom2

(i) `(boom2)`

Solution: Error: Zero Division

(j) How can we rewrite boom2 without using the lambda operator?

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

2. What will Scheme output?.

(a) `(if 0 (/ 1 0) 1)`

Solution:
```
Error: Zero Division
```

(b) `(and 1 #f (/ 1 0))`

Solution:
```
#f
```

(c) `(and 1 2 3)`

Solution:
```
3
```

(d) `(or #f #f 0 #f (/ 1 0))`

Solution:
```
0
```

(e) `(or #f #f (/ 1 0) 3 4)`

Solution:
```
Error: Zero Division
```

(f) `(and (and) (or))`

Solution:
```
#f
```

3. **let** is a special form in Scheme which allows you to create local bindings. Consider the example

```
(let ((x 1)) (+ x 1))
```

Here, we assign `x` to `1`, and then evaluate the expression `(x + 1)` using that binding, returning `2`. However, outside of this expression, `x` would not be bound to anything.
Each `let` special form has a corresponding lambda equivalent. The equivalent lambda expression for the above example is

```
((lambda (x) (+ x 1)) 1)
```

The following line of code does not work. Why? Write the lambda equivalent of the `let` expressions.

```
(let ((foo 3)
      (bar (+ foo 2)))
    (+ foo bar))
```

Solution: The above function will error because it is equivalent to:
`((lambda (foo bar) (+ foo bar)) 3 (+ foo 2))`

In other words, foo has not been defined in the global frame. When bar is being assigned to `(+ foo 2)`, it will error. The assignment of foo to 3 happens in the lambda?s frame when it's called, not the global frame (this is relevant to the Scheme project – when the interpreter sees `lambda`, it will call a function to start a new frame).

If we had the line `(define foo 3)` before the call to let, then it would return 8, because within let, foo would be 3 and bar would be `(+ 3 2)`, since it would use the foo in the Global frame.

2 Scoping

1. What is the difference between dynamic and lexical scoping?

   Solution:

   - **Lexical:** The parent of a frame is the frame in which a procedure was defined (used in Python).
   - **Dynamic:** The parent of a frame is the frame in which a procedure is called (keep an eye out for this in the Scheme project).

2. What would this print using lexical scoping? What would it print using dynamic scoping?

```
a = 2
def foo():
    a = 10
    return lambda x: x + a
bar = foo()
bar(10)
```

   Solution:

   - **Lexical:** 20
   - **Dynamic:** 12

3. How would you modify an environment diagram to represent dynamic scoping?

   Solution: Assign parents when you create a frame (do not set parents when defining functions!). The parent in this case is the frame in which you called this function.

3 Code-Writing

1. Implement `waldo`. `waldo` returns `#t` if the symbol waldo is in a list. You may assume that the list passed in is well-formed.

```
scm> (waldo '(1 4 waldo))
#t
scm> (waldo '())
#f
scm> (waldo '(1 4 9))
#f
```

```
(define (waldo lst)









)
```

Solution:

```
(define (waldo lst)
    (cond ((null? lst) #f)
          ((eq? (car lst) 'waldo) #t)
          (else (waldo (cdr lst)))
      )
  )
```

2. **Extra challenge:** Define `waldo` so that it returns the index of the list where the symbol waldo was found (if waldo is not in the list, return `#f`).

```
scm> (waldo '(1 4 waldo))
2
scm> (waldo '())
#f
scm> (waldo '(1 4 9))
#f
```

```
(define (waldo lst)









)
```

Solution:

```
(define (waldo lst)
    (define (helper lst index)
        (cond ((null? lst) #f)
              ((eq? (car lst) 'waldo) index)
              (else (helper (cdr lst) (+ index 1)))
        )
      )
    (helper lst 0)
  )
```

4 Challenge Question

3. **(Optional)** The quicksort sorting algorithm is an efficient and commonly used algorithm to order the elements of a list. We choose one element of the list to be the pivot element and partition the remaining elements into two lists: one of elements less than the pivot and one of elements greater than the pivot. We recursively sort the two lists, which gives us a sorted list of all the elements less than the pivot and all the elements greater than the pivot, which we can then combine with the pivot for a completely sorted list.

   Implement `quicksort` in Scheme. Choose the first element of the list as the pivot. You may assume that all elements are distinct. Hint: you may want to use a helper function.

   You may additionally want to use the built-in `append` function, which takes in two lists and returns a new list containing the elements of the first list followed by the elements of the second list. You can also use `filter` procedure, which takes in a one-argument function and a list and returns a new list containing only the elements of the original list for which the function returns true, although it is not required.

```
scm> (quicksort (list 5 2 4 3 12 7))
(2 3 4 5 7 12)
```

Solution:

```
(define (quicksort lst)
    (define (helper lst pivot less greater )
        (cond
            ((null? lst)
                (append (qs less) (list pivot) (qs
                   greater)))
            ((> pivot (car lst))
                (helper (cdr lst) pivot (append (list (car
                   lst)) less) greater))
            ((< pivot (car lst)
                (helper (cdr lst) pivot less (append (list
                   (car lst)) greater)))))
    (if (or (null? lst) (null? (cdr lst)))
                        lst
            (helper (cdr lst) (car lst) nil nil)))
```

Alternate solution using filter:

```
(define (quicksort lst)
   (if (null? lst)
       nil
       (let ((less (filter (lambda (x) (< x (car lst)))
          lst))
          (greater (filter (lambda (x) (> x (car lst)))
             lst)))
          (append (append (quicksort less) (list (car
             lst))) (quicksort greater)))))
```