1 What Would Scheme Print?

Solution: Solutions begin on the following page.

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

Solution:
4

(c) ((lambda (x y z) (y x)) 2 / 2)

Solution:
0.5

(d) ((lambda (x) (x x)) (lambda (y) 4))

Solution: 4

(e) (define boom1 (/ 1 0))

Solution: Error: Zero Division

(f) boom1

Solution: Error: boom1 not defined

(g) (define boom2 (lambda () (/ 1 0)))

Solution: boom2

(h) (boom2)

Solution: Error: Zero Division
(i) Why/How are the two “boom” definitions above different?

**Solution:** 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 \( (/ 1 0) \).

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

**Solution:**

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

2. What will Scheme output?

(a) \((\text{if} \ (\text{/} \ 1 \ 0) \ 1 \ 0)\)

**Solution:**

Error: Zero Division

(b) \((\text{if} \ 1 \ 1 \ (\text{/} \ 1 \ 0))\)

**Solution:**

1

(c) \((\text{if} \ 0 \ (\text{/} \ 1 \ 0) \ 1)\)

**Solution:**

Error: Zero Division

(d) \((\text{and} \ 1 \ #f \ (\text{/} \ 1 \ 0))\)

**Solution:**

#f

(e) \((\text{and} \ 1 \ 2 \ 3)\)

**Solution:**

3
(f) \(\text{or } #f #f 0 #f (/ 1 0)\)

Solution:
0

(g) \(\text{or } #f #f (/ 1 0) 3 4\)

Solution:
Error: Zero Division

(h) \((\text{and } (\text{and}) (\text{or}))\)

Solution:
#f

(i) Given the lines above, what can we say about interpreting \textit{if} expressions and booleans in Scheme?

Solution: \textit{if} functions and boolean expressions will short-circuit, just like in Python. All values have a boolean value of \#t unless they are specifically \#f. This means that unlike in Python, 0 and 1 are both considered \#t!

3. The following line of code does not work. Why? Write the lambda equivalent of the \textit{let} expressions.
\(\text{let } ((\text{foo } 3)
   \text{(bar } (+ \text{foo } 2))\)
\(+ \text{foo } \text{bar})\)

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

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 \textit{lambda}, it will call a function to start a new frame).

If we had the line \((\text{define foo } 3)\) before the call to \textit{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.
4. 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).

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

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

Solution:

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

6. 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.
7. Implement `waldo`. `waldo` returns `#t` if the symbol `waldo` is in a list. You may assume that the list passed in is well-formed.

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

**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`).

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

**Solution:**

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

**Challenge solution:**

```scheme
(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))
)
3 Challenge Question

8. (Optional) From CS61A Fall 2017 Discussion 6: 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.

```
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))
)
```