SCHEME LISTS

CS 61A GROUP MENTORING

July 16th, 2018

1 What Would Scheme Display

1. What will Scheme output? If it outputs a list, draw a box-and-pointer diagram. If a part contains multiple expressions, only give the output of the final expression. Assume each expression is evaluated sequentially.

(a) `scm> (cons 3 4)`

Solution:
`(3 . 4)`

(b) `scm> (cons 3 (cons (cons 4 5) nil))`

Solution: `(3 (4 . 5))`

(c) `scm> (3 . 4)`

Solution: `Error`

(d) `scm> '(3 . 4)`

Solution: `(3 . 4)`

(e) `scm> (list 3 4 5)`

Solution: `(3 4 5)`

(f) `scm> (list 3 '(4 5) 6)`

Solution: `(3 (4 5)6)`

(g)
```
scm> (define a '(5 (6) . (7)))
scm> a
```

Solution: `(5 (6)7)`

(h)
```
scm> (set-car! a 2)
scm> (define b (list 3 a))
scm> b
```

Solution: `(3 (2 (6)7))`

(i)
```
scm> (set-cdr! (cdr (cdr a)) 8)
scm> b
```

Solution: `(3 (2 (6)7 . 8))`

(j)
```
scm> (define c 2)
```

Solution: `c`

(k)
```
scm> (eval 'c)
```

Solution: `2`

(l)
```
scm> '(cons 1 2)
```

Solution: `(cons 1 2)`

(m)
```
scm> (eval '(cons 1 2))
```

Solution: `(1 . 2)`

2 Code Writing

2. Define `well-formed`, which determines whether `lst` is a well-formed list or not. Assume that `lst` only contains numbers and no nested lists.

```
; Doctests
scm> (well-formed '())
#t
scm> (well-formed '(1 2 3))
#t
; List doesn't end in nil
scm> (well-formed (cons 1 2))
#f

(define (well-formed lst)









)
```

Solution:

```
; well-formed with nested if statements
(define (well-formed lst)
    (if (null? lst)
        #t
        (if (number? lst)
            #f
            (well-formed (cdr lst)))))

; well-formed with a cond statement
(define (well-formed lst)
    (cond ((null? lst) #t)
        ((number? lst) #f)
        (else (well-formed (cdr lst)))))
```

3. 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)









)
```

Solution:

```
; 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))))))
```

4. Define `waldo` which takes in a list. If that list contains the symbol waldo, it returns the index where waldo first appears. Otherwise, it returns `#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))
```

5. Implement `double-link`, which takes in a list and replaces the second in each pair of two consecutive items with the first using mutation and returns the mutated list. The first of each pair of consecutive items is unchanged.

```
scm> (define a '(1 2 3 4))
a
scm> (double-link a)
(1 1 3 3)
scm> a
(1 1 3 3)
scm> (double-link '(c s 6 1 a))
(c c 6 6 a)
```

```
(define (double-link lst)

    (if _____________________________________

        _________________________________

        (begin

            _____________________________________

            _____________________________________

            lst)))
```

Solution:

```
(define (double-link lst)
    (if (or (null? lst) (null? (cdr lst)))
        lst
        (begin
            (set-car! (cdr lst) (car lst))
            (double-link (cdr (cdr lst)))
            lst)))
```