Scheme Review and Lists Solutions

Computer Science Mentors 61A

November 12 – November 15, 2024

Created by Jim Fang, Bryan Adolfo Ventura Benitez. Worksheet feedback: links.jimfang.dev/feedback

1 Scheme Review

1. 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))))
```

2 Scheme Lists

Unlike Python, all Scheme lists are linked lists. Recall that, in Python, a linked list is made up of `Link`s that each have a `first` and a `rest`, where the `rest` is another `Link`. Similarly, each Scheme list is a "pair" where the first element of the pair is the first element of the list, and the second element of the pair is the rest of the list (also a pair).

We use the `cons` procedure to construct Scheme lists, and `nil` to represent empty lists. The sequence $1, 2, 3$ may then be represented as follows:

```
scm> (cons 1 (cons 2 (cons 3 nil)))
```

```
(1 2 3)
```

The `car` and `cdr` procedures are used to access the elements of a Scheme list. `car` gets the first element of a list, while `cdr` gets the rest of the list:

```
scm> (define lst (cons 1 (cons 2 (cons 3 nil))))
lst
scm> (car lst)
1
scm> (cdr lst)
(2 3)
```

You can make the following analogy between linked lists in Python and Scheme:

`Link(1, Link.empty)`	`(cons 1 nil)`
`a = Link(1, Link(2, Link.empty))`	`(define a (cons 1 (cons 2 nil)))`
`a.first`	`(car a)`
`a.rest`	`(cdr a)`

The `list` procedure and quotation give us additional convenient ways to construct lists:

```
scm> (list 1 2 3)
(1 2 3)
scm> '(1 2 3)
(1 2 3)
scm> (list 1 (+ 1 1) 3)
(1 2 3)
scm> '(1 (+ 1 1) 3)
(1 (+ 1 1) 3)
```

Note that quotation will prevent any of the list items from being evaluated, which can occasionally be inconvenient.

2.1 Useful procedures

In addition to the procedures mentioned above, the following procedures are often useful when dealing with Scheme lists:

- `(null? s)`: returns true if `s` is `nil`.
- `(length s)`: returns the length of `s`.
- `(append s1 ... sn)`: returns the result of concatenating lists `s1`, ..., `sn`.
- `(map f s)`: returns the result of applying the procedure `f` to each element of `s`.
- `(filter pred s)`: returns a list containing the elements of `s` for which the single-argument procedure `pred` returns true.
- `(reduce comb s)`: combines the elements of `s` into a single value using the two-argument procedure `comb`.

2.2 Equality testing

Equality testing in Scheme is a bit confusing as it is handled by three separate procedures:

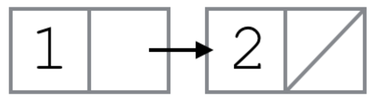
- `(= a b)`: returns true if `a` equals `b`. Both must be numbers.

- `(eq? a b)`: returns true if `a` and `b` are equivalent primitive values. For two objects, `eq?` returns true if both refer to the exactly same object in memory (like `is` in Python).
- `(equal? a b)`: returns true if `a` and `b` are equivalent. Two lists are equivalent if their elements are equivalent.

1. What will Scheme output? Draw box-and-pointer diagrams to help determine this. (Ask your mentor if you're unsure what's going on. You aren't expected to understand this completely on your own.)

```
scm> (cons 1 (cons 2 nil))
```

```
(1 2)
```



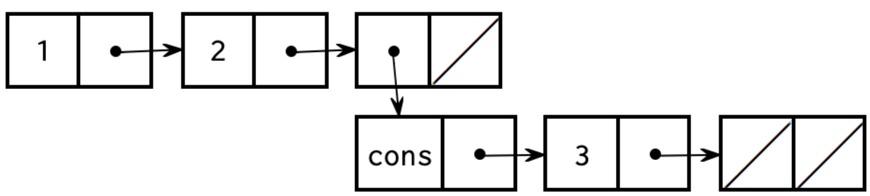

```
scm> (cons 1 '(2 3 4 5))
```

```
(1 2 3 4 5)
```

When we use the quote before the list, we are saying that we should put the literal list (2 3 4 5) in the cdr of this list. So in this case we create a list where the first element (car) is 1, and the cdr is the list (2 3 4 5).

```
scm> (cons 1 '(2 (cons 3 nil)))
```

```
(1 2 (cons 3 ()))
```



Since we also used a quote here, we do not evaluate the `(cons 3 nil)`. We keep everything inside the quotes the same so the `cdr` of this list is the list `(2 (cons 3 nil))`. That means that we add the element 2, and then the nested list `(cons 3 nil)`.

```
scm> (cons 1 (2 (cons 3 nil)))
```

```
eval: bad function in : (2 (cons 3 nil))
```

While evaluating the operands, Scheme will try to evaluate the expression `(2 (cons 3 nil))`. Since 2 is not a valid operator, this expression Errors.

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

```
(3 (4))
```

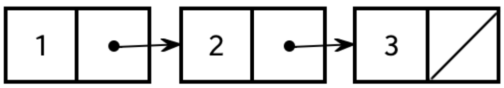
```
scm> (define a '(1 2 3))
```

```
a
```

Defines a list of elements of `(1 2 3)` and binds the list to the variable `a`. Recall that define returns the name of the symbol.

```
scm> a
```

```
(1 2 3)
```




```
scm> (car a)
```

```
1
```

```
scm> (cdr a)
```

```
(2 3)
```

```
scm> (car (cdr a))
```

```
2
```

From above, we know that `(cdr a)` is `(2 3)`. From that, we can evaluate `(car (cdr a))` to 2.

How can we get the 3 out of a?

```
(car (cdr (cdr a)))
```

To get to the pair that contains 3, we need to call `(cdr (cdr a))`. To get the element 3, we need the `car` of `(cdr (cdr a))`.

2. Define `sixty-ones`. Return the number of times that `1` follows `6` in the list.

```
> (sixty-ones '(4 6 1 6 0 1))
1
> (sixty-ones '(1 6 1 4 6 1 6 0 1))
2
> (sixty-ones '(6 1 6 1 4 6 1 6 0 1))
3
```

```
(define (sixty-ones lst)
    (cond (______________________________________________)
          (__________________________________________________________________________)
          (else ________________________________)))
```

```
(define (sixty-ones lst)
    (cond ((or (null? lst) (null? (cdr lst))) 0)
          ((and (= 6 (car lst)) (= 1 (cadr lst))) (+ 1 (sixty-ones (cddr
             lst))))
          (else (sixty-ones (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)

        ______________________________

        ______________________________

        ______________________________

        ______________________________

        ______________________________

        ______________________________

)
```

```
; 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. Implement `argmax`, a function that takes in a list, `lst`, and returns the index of the largest element in `lst`. If there are two or more elements that are the largest element, return the index of the one that appears first in `lst`.

   You can assume all elements of `lst` are non-negative integers, and `lst` has at least 1 element and no nested lists.

```
(define (argmax lst)
    (define (max-helper lst max-so-far max-index curr-index)
        (cond

            ((________________________) ______________________________)

            ((________________________) ______________________________

                _________________________________________________________________________)

            (else
                _________________________________________________________________________)
        )
    )

    (max-helper _____________________________)
)
```

```
(define (argmax lst)
    (define (max-helper lst max-so-far max-index curr-index)
        (cond
            ((null? lst) max-index)
            ((> (car lst) max-so-far)
                (max-helper (cdr lst) (car lst) curr-index (+ curr-index
                    1)))
            (else
                (max-helper (cdr lst) max-so-far max-index (+ curr-index
                    1)))
        )
    )
    (max-helper lst 0 0 0)
)
```