SCHEME LISTS

COMPUTER SCIENCE MENTORS 61A

April 17–April 21, 2023

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)




)
```

Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang

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.)

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



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



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



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



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




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



scm> a



scm> (car a)



scm> (cdr a)



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

How can we get the 3 out of 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
```

3. You are creating a computer from scratch. In their rawest form, computers use 0s and 1s to compose commands and data. Fill in a function that takes a list of boolean values representing an **unsigned binary number** and returns its **decimal representation**. Each `#t` in the list represents a 1 and each `#f` represents a 0, with the **first** element in the list being the **rightmost** (smallest) binary digit and the **last** element being the **leftmost** (largest) binary digit.

```
;Doctests
scm> (binary (list #f #t)) ; 10
2
scm> (binary (list #t #f #t #t)) ; 1101
13
scm> (binary (list #t #t #f #f #t)) ; 10011
19
scm> (binary (list #f)) ; 0
0

(define (binary bin-list)
  (cond
    ((null? _____)
     _____
    )
    ((_____)
     _____
    )
    (else
     _____
    )
  )
)
```

4. Now, write the binary to decimal function, but in tail recursive form. Note that the `expt` function takes in a base and an exponent. For example, `(expt 2 3)` raises 2 to the third power, returning 8.

```scheme
;Doctests
scm> (binary-tail (list #f #t)) ; 10
2
scm> (binary-tail (list #t #f #t #t)) ; 1101
13
scm> (binary-tail (list #t #t #f #f #t)) ; 10011
19
scm> (binary-tail (list #f)) ; 0
0

(define (binary-tail bin-list)
  (define (helper bin-list i sum)
    (cond
      ((null? _____)
       _____
      )
      ((_____)
       _____
      )
      (else
       _____
      )
    )
  )
  (helper _____)
)
```

5. 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)




)
```

6. 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
```

```scheme
      ( (_____)  _____)

      ( (_____)  _____

        _____)

      (else _____)
    )
  )

  (max-helper _____)
)
```