SCHEME LISTS Meta

COMPUTER SCIENCE MENTORS 61A

April 24–May 5, 2023

1 Scheme Review

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

Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang
Provided by CSM 61A contributors under CC BY 4.0/3-clause BSD.

2. 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))))
```

Teaching Tips

- Functions can get a little confusing to work with in Scheme so it helps to remind your students that they are all just pieces of data.
- Thinking about the base case first may be more helpful with this problem. When do you stop applying your function to the input? How do you know/which input will tell you when to stop? This will then provide a good idea of what the recursive calls should be.
- There are two alternate solutions to this issue that differ in what you call your function f on. Go along with whatever your students share and if possible share the alternate solutions so students can see different ways of recursing in Scheme.

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)
```

It's worth pointing out to your students that, unlike with the `Link` class, the `nil` must be explicitly provided at the end of the linked list.

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.

If relevant, I like to discuss when it makes the most sense to use the different ways of constructing a list.

- `cons` is useful when you have a way to construct the first element and rest of the list, e.g. in recursive problem solving,
- `list` and quotation are useful when you want to hardcode a list into your code beforehand, but typically aren't that useful if you want to dynamically create a list based on program input.

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.

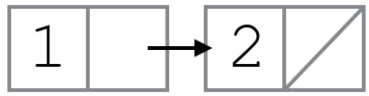
Teaching Tips

- For the love of God, please do not mini-lecture all of this stuff. This information is presented as a reference for you, and you should ask your students what they would like to go over so that you do not waste their time.
- Emphasize to students that Scheme lists are linked lists and NOT Python lists
  - Discuss the limitations (e.g. no indexing) and capabilities (e.g. recursion)
- If you're an old bear™, keep in mind that dotted lists (thank god) have been removed from the curriculum, so Scheme lists have the same functionality as linked lists
- The 61A Scheme Web interpreter is **very useful** for visualizing lists!
- If you choose to give a mini-lecture on Scheme list syntax, try using each keyword in an example instead of just talking about them!

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)`



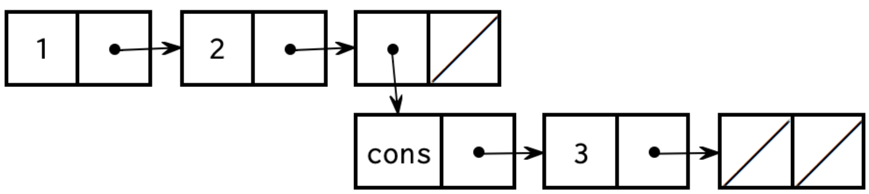

```
scm> (cons 1 '(2 3 4 5))
```

`(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))`

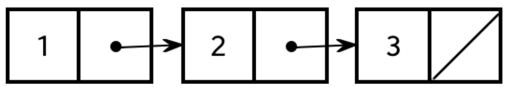
```
scm> (define a '(1 2 3))
```

```
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))`.

Teaching Tips

- Draw diagrams or use the 61A Scheme Web interpreter for visualizing lists
- Encourage students to ask questions and experiment with extra `cons`, `car`, and `cdr` statements to see how they change the outputs of statments!
- While unrelated to the problem, it may be helpful to teach students these keywords:
  - `(pair? arg)`, which checks if arg has a first and rest
  - `(list? arg)`, which returns true if arg is a well-formed list

2. 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
      ____________________________________
    )
  )
)
```

```
(define (binary bin-list)
  (cond
    ((null? bin-list)
      0
    )
    ((car bin-list)
      (+ 1 (* 2 (binary (cdr bin-list))))
    )
    (else
      (* 2 (binary (cdr bin-list)))
    )
  )
)
```

3. 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.

```
;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 ____________________)
)
```

```
(define (binary-tail bin-list)
  (define (helper bin-list i sum)
    (cond
      ((null? bin-list)
        sum
      )
      ((car bin-list)
        (helper
          (cdr bin-list) (+ 1 i) (+ sum (expt 2 i))
        )
      )
      (else
        (helper
          (cdr bin-list) (+ 1 i) sum
        )
      )
    )
  )
  (helper bin-list 0 0)
)
```

4. 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)

)
```

```scheme
; 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))))))
```

Teaching Tips

- Encourage students to think about how they would solve this problem without starter code. How would you determine if a given input matches the first part of another input? Iteration! Then translate this iteration into Scheme.
- Be sure to check for null cases or edge cases, keep track of parentheses, and keep in mind how true and false are represented in Scheme.
- Remind students also that there are two ways to go about checking different cases in Scheme: nested ifs or a cond statement.
- As a hint, also consider suggesting figuring out the logic with pseudo or Python code, then translating into Scheme.

5. 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.

```scheme
(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)
)
```

It's important that students learn how to use helper functions in Scheme; since there is no iteration, most anything that cannot be done through pure recursion will have to be done via a helper function.