INTRODUCTION TO SCHEME Solutions

COMPUTER SCIENCE MENTORS

November 2, 2020 - November 5, 2020

Scheme is a *functional* language, as opposed to Python, which is an *imperative* language; whereas a Python program is comprised of *statements* or "instructions" which do not evaluate to a value (an example of a statement would be something like `x = 3` in Python. This does not evaluate to any value but just instructs Python to create a variable `x` with the value 3), a Scheme program is comprised solely of *expressions*, each of which simply evaluates to a value. Remember that the term evaluate means to find the value of something. A variable is evaluated by looking up the name in the current frame, and a function call expression is evaluated using the three steps listed below. The thing to notice is that different types of expressions have different rules for how to evaluate those expressions, and this goes for both Python and Scheme.

The four basic types of expressions in Scheme are literals (i.e., a value itself), call expressions (procedure calls), special forms (language features), and variables. A call expression or special form is denoted by a pair of parentheses and takes prefix notation, i.e., it is formed as so:

```
(operator operand_0, operand_1, ... , operand_n)
```

(Keep in mind each item in a call expression is also an expression)

Evaluation of a call expression progresses so:

1. Evaluate operator (returning a procedure)
2. Evaluate operands
3. Apply operator on operands

If Expression: The **`if`** keyword is similar to **`if/else`** statements in Python. It works as follows:

```
(if <predicate> <do if true> <do if false>)
```

This is similar to the following code in Python:

```
if <predicate>:
    <do if true>
else:
    <do if false>
```

Note that in Python, **`if`** is a statement whereas in Scheme, **`if`** is an expression and evaluates to a value like any other expression would. This means that in Scheme you could write something like this where the **`if`** expression can be placed as an operand in a function call expression:

```
scm> (+ 1 (if #t 9 99))
10
```

Just like in Python, the `<do if false>` (in Python this would be the equivalent to the else clause) is optional. If there is no `<do if false>` and the `<predicate>` evaluates to false then the **`if`** expression evaluates to `undefined`.

Define Expression: `define` does two things in Scheme. The first is that it defines variables using the following syntax:

```
(define <name> <expression>)
```

The way this works is Schme will evaluates `<expression>` and binds the value to `<name>` in the current environment. `<name>` must be a valid Scheme symbol (you can think of a symbol as an identifier or variable name).

`define` is also used to define functions using the following syntax (note that this is different from using `define` to create variables as there is an extra pair of parentheses around `<name> [param] ...`):

```
(define (<name> [param] ...) <body> ...)
```

Either way, after the `<name>` is bound to either a function or value, the define expression evaluates to the symbol `<name>`.

```
scm> (define x 3)
x
```

links.cs61a.org/schemespec will direct you to a page with all of the explanations and syntax descriptions of Scheme. Use it when filling out the WWSD section!

1 What Would Scheme Print?

1. What will Scheme output?

```
scm> (define pi 3.14)
pi
scm> pi
3.14
scm> 'pi
pi
scm> (+ 1 2)
3
scm> (+ 1 (* 3 4))
13
scm> (if 2 3 4)
3
scm> (if 0 3 4)
3
scm> (- 5 (if #f 3 4))
1
scm> (if (= 1 1) 'hello 'goodbye)
hello
scm> (define (factorial n)
       (if (= n 0)
           1
           (* n (factorial (- n 1)))))
factorial
scm> (factorial 5)
120
```

CSM 61A Fall 2020: Jason Chang and Jade Singh, with
Catherine Cang, Alina Trinh, Ethan Chiu, Jamie Ip, Kenneth Chi, and
Aidan Tong, Aman Shah, Bridget Cheng, Cindy Lin, Ethan Yeh, Ivan Penev, Ivy Li, James Fletcher, Jemmy Zhou, Jennifer Huang, Jessica Yu, Joshua Baum, Kevin Moy, Kunal Agarwal, Lauren Meier, Matthew Guo, Mihira Patel, Nikhita Anumukonda, Owen Gong, Richard Roggenkemper, Sean Sananikone, Uma Unni

2 Code Writing in Scheme

2. **Hailstone yet again** Define a program called `hailstone`, which takes in two numbers `seed` and `n`, and returns the `n`*th* hailstone number in the sequence starting at `seed`. Assume the hailstone sequence starting at `seed` is longer or equal to `n`. As a reminder, to get the next number in the sequence, if the number is even, divide by two. Else, multiply by 3 and add 1.

   Useful procedures

   - `quotient`: floor divides, much like `//` in python

     `(quotient 103 10)` outputs 10

   - `remainder`: takes two numbers and computes the remainder of dividing the first number by the second

     `(remainder 103 10)` outputs 3

```
; The hailstone sequence starting at seed = 10 would be
; 10 => 5 => 16 => 8 => 4 => 2 => 1

; Doctests
> (hailstone 10 0)
10
> (hailstone 10 1)
5
> (hailstone 10 2)
16
> (hailstone 5 1)
16

(define (hailstone seed n)









)
```

```
(define (hailstone seed n)
    (if (= n 0)
        seed
        (if (= 0 (remainder seed 2))
            (hailstone
            (quotient seed 2)
            (- n 1))
          (hailstone
          (+ 1 (* seed 3))
          (- n 1)))))
```

3 Scheme Lists

Unlike Python, all Scheme lists are linked lists. Recall a linked list is made up of Links that have a first and a rest, where the rest is another Link. Similarly, Scheme lists are made up of pairs with a first and a rest, where the rest is another pair.

Ways to make scheme lists:

- Cons
  Syntax: `(cons <car-elem> <cdr-elem>)`
  Takes in a pair of two elements; similar to how a python linked list has 2 elements as well- first and rest
- List
  Syntax: `(list <elem1> <elem2> ...)`
  Takes in an arbitrary number of elements/arguments, and constructs a list where each elem is the first of its own pair. Note how this differs from `cons` where you specify a first and rest rather than just specifying the first of each pair. All the arguments will be evaluated before being collected into the scheme list.
- ' (aka single quote)
  Syntax: `'(<elem1> <elem2> ...)`
  Also takes in an arbitrary number of elements and construct a list out of the elements, but the arguments are not evaluated.

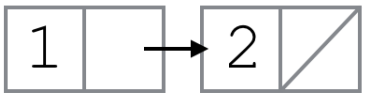
Ways to access list items:

- Car
  Syntax: `(car <pair>)`
  Gets you the first item of a pair
- Cdr
  Syntax: `(cdr <pair>)`

Gets you the second item of a pair

- Cadr
  Syntax: `(cadr <pair>)`
  Gets you the `car` of the `cdr`
- Cddr
  Syntax: `(cddr <pair>)`
  Gets you the `cdr` of the `cdr`

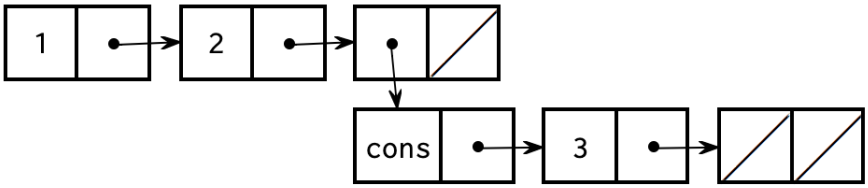
```
scm> (cons 1 (cons 2 nil))
(1 2)
```




```
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)`.

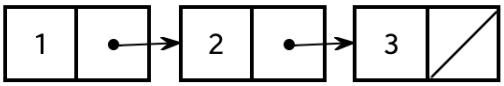
```
scm> (cons 1 (2 (cons 3 nil)))
eval: bad function in : (2 (cons 3 nil))
```

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> (cadr a)
2
```

Recall that cadr is short for `(car (cdr a))`. 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))`.

4 More Code Writing in Scheme

3. Implement `waldo`. `waldo` returns `#t` if the symbol waldo is in a list.

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

(define (waldo lst)
```

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

Alternate solution:

```
(define (waldo lst)
    (if (null? lst)
        #f
        (if (eq? (car lst) 'waldo)
            #t
            (waldo (cdr lst))
        )
    )
)
```

Similar to how we focus just think about first and rest when working with linked lists, we want to think about solving this question in terms of car and cdr in scheme. At a high level, we can check we can check (car lst) is equal to waldo. If it is, we can return true. Otherwise, we check the rest of the list: (cdr lst). Last, we conclude that waldo is not in the list when we have checked every element but still not found at match.

Observe that the second doctest is essentially the simplest input we can give. From here, we get our first base case– if we are given an empty Scheme list, then return `#f`. Otherwise, notice that regardless of where in the list we find 'waldo', our function should return `#t`. So our function can just stop after we find the first instance of 'waldo in the list, if it exists. Therefore, our second base case checks if our current element (the car of lst) is 'waldo and returns `#t` if this is true. Finally, if neither base case is satisfied, then we must search through the rest of lst, excluding the first element we just checked, for the string 'waldo. So, we make a recursive call using the cdr of lst.

Many problems that involve traversing through a Scheme list will have a similar structure to this solution. Recursive calls on (cdr lst) are also common, similar to recursive calls on link.rest. Note that you can also write a solution using two nested ifs instead of cond, but cond is often a little cleaner to read.

4. **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`).

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

(define (waldo lst)









)
```

```
(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)
  )
```

Recall HW3, problem 2, "pingpong" without assignment. To get around the assignment restriction, we introduced a new variable to keep track of state. We passed some state into a helper function to keep track of our iteration. In this problem, we will use the same idea for our implementation.

Inside our recursive helper function, we use the same logic as the original waldo question. Specifically, if the list is empty, we return false. If the first element we see is equal to waldo, we now return the index we have been tracking instead of true. Finally, to recursively search, we call the helper function on the rest of the list and increment the index by 1, returning the result of that call. We call this helper function initially with arguments 'lst' and index 0 to start from the beginning of the list.

A logical alternate solution would be to increment 1 after the recursive call to helper (without incrementing index), instead of adding one to the index, and then recursing. This solution would actually be incorrect. When we increment by one, if the element is not found, `#f` would not be properly returned. At the end of recursion, we would try to add 1 to `#f`, which results in an error.