Solutions to CS 61A Challenge Problems: Basic Scheme

Other worksheets and solutions at https://alextseng.net/teaching/cs61a/

Alex Tseng

1 Constructing Pairs and Lists

For each of the following lines of code, determine what Scheme will print out and draw the corresponding box-and-pointer diagram.
See last page for box and pointer diagrams

```
(define a '(1 2 3))
(define b '(4 5 6))

(cons a b) ; 1
((1 2 3) 4 5 6)

(cons 10 20) ; 2
(10 . 20)

(cons '(a b) '(a)) ; 3
((a b) a)

(cons (list a b) (list a)) ; 4
(((1 2 3) (4 5 6)) (1 2 3))

(list a b) ; 5
((1 2 3) (4 5 6))

(list (list (list a)) b) ; 6
((((1 2 3))) (4 5 6))

(append a b) ; 7
(1 2 3 4 5 6)

(append (cons a 'foo) (list b)) ; 8
Error

(append a b 42) ; 9
(1 2 3 4 5 6 . 42)
```

Make sure you understand the *mechanisms* of `cons`, `list`, and `append`, as they are the 3 main ways of creating pairs and lists. Also make sure you understand `car`, `cdr`, and everything in between. Remember, read the letters right to left: `(caddr a) ---> 3`. There can be up to 4 letters between the `c` and `r`.
`cons` takes in exactly 2 arguments, and creates a single new pair with the first element being the first argument and the second element being the second argument. It doesn't matter if these arguments are symbols or lists.
`list` takes in any number arguments and puts them into a new list. You can imagine that `list` takes its n arguments and creates a new set of n pairs pointing in a line (linked-list style), and then fills in the first element of each pair with the arguments. Again, it doesn't matter if these arguments are symbols or lists.
`append` takes at least 2 lists and concatenates them together. That is, it goes through the lists and makes the last element of the ith list point to the first element of the $i+1$th list. Notice that `append` must be given

lists. Scheme will complain otherwise. The exception is that the very last argument to `append` doesn't have to be a list. The mechanism is the same, but now the list is not well-formed, like in number 9.

2 Functions on Lists

(a) Write a function `last` that takes in a list and returns the last element in the list.
```
(last '(1 2 3 4 (5 6) 7)) ---> 7
(last '(1 2 3 (4 5)) ---> (4 5))
```

```
(define (last lst)
    (cond ((null? (cdr lst)) (car lst))
          (else (last (cdr lst)))))
```

This is fairly simple recursion. If we're at the end of the list (the second element of the pair is null), then we are done. Otherwise, we must go further down the list and call it on cdr of the list.

(b) Write a function `double` that takes in a list and returns a list with every element duplicated. Assume that every element in the list is a single token, and not another list.
```
(double '(1 2 3 4)) ---> (1 1 2 2 3 3 4 4)
```

```
(define (double lst)
    (cond ((null? lst) lst)
          (else (append (list (car lst) (car lst)) (double (cdr lst))))))
```

Again, this is simple recursion. We simply go through and duplicate the car of the list and put them into a new list. Notice that we use `list` to put the duplicates in a new list, and then we use `append` to concatenate the result of the rest of the list.

(c) *Challenge* You may be familiar with the function that reverses a shallow list. That is, if the list has elements that are also lists, those inner lists are not reversed themselves. Write a function `deep-reverse` that reverses all elements of the list, including sublists.
```
(deep-rev '(1 2 (3 4 5) ((6)) 7 (8 9) 10)) ---> (10 (9 8) 7 ((6)) (5 4 3) 2 1)
```

```
(define (deep-rev lst)
    (cond ((null? lst) lst)
          ((list? (car lst)) (append (deep-rev (cdr lst)) (list (deep-rev (car lst)))))
          (else (append (deep-rev (cdr lst)) (list (car lst))))))
```

This one has two recursive cases. If the first element is a list, then we must deep reverse that list before appending it to the end of the result of deep reversing the rest of the list. Notice that we must wrap this in a `list` call, as well. This is because `append` will concatenate the input lists, so if we were to forget to wrap `(deep-rev (car lst))` in another list, we would lose that level of nesting completely. In the case that the first element is a simple token, then we simply need to append it to the end of the result of deep reversing the rest of the list. Again, we need to wrap this first element in a list so `append` does the right thing.

(d) *Challenge* Write a function `flatten` that flattens a list, bringing all elements in sublists to the top level.
```
(flatten '(a b (c d) (((e)) f) (g (h (i) j k) l))) ---> (a b c d e f g h i j k l)
```

```
(define (flatten lst)
    (cond ((null? lst) lst)
          ((list? (car lst)) (append (flatten (car lst)) (flatten (cdr lst))))
          (else (append (list (car lst)) (flatten (cdr lst))))))
```

This is a classic Scheme problem. Again, our two recursive cases are if the first element is a list or if it is not. If the first element is not a list (`else` case), then we simply append the result of flattening the rest of the list to a new list with the first element. We use `append` and put the first element in a list. In the case that the first element was a list, we must flatten this first element, as well. Notice that `flatten` returns a list, and we wish to bring everything to the top nesting level, so we don't wrap the results of either of these calls in lists before appending them together. We could have also used `cons` for this case: `(cons (car lst) (flatten (cdr lst))`. It may behoove you to run through an example.

3 Iteration to Recursion in Scheme

(a) Write a function `prime` that tests if a number is prime. You may find the function `mod` useful, which is the equivalent to the `%` operator to find the remainder in Python.
Hint: consider writing a helper function

```
(define (prime-helper num factor)
    (cond ((= factor 1) #t)
          ((= (mod num factor) 0) #f)
          (else (prime-helper num (- factor 1)))))

(define (prime? num)
    (prime-helper num (- num 1)))
```

Scheme doesn't have any iterators, like `for` or `while` like in Python and many other languages. So everything must be naturally recursive. A common trick we use in Scheme is to write a helper function whose argument stores some state. For example, we would normally use a variable to keep track of numbers and check if they are factors. But since we cannot directly store them, we must keep them around as arguments. So `prime-helper` will simply run through all possible factors and return whether or not `num` is prime. We then write a wrapper function `prime` that calls our helper with the appropriate starting values. This is a very common pattern in Scheme. There are other ways of solving this problem, too.

(b) *Challenge* Write a function `fibo` that returns the `n`th Fibonacci number in $\Theta(n)$ time (so no tree recursion).

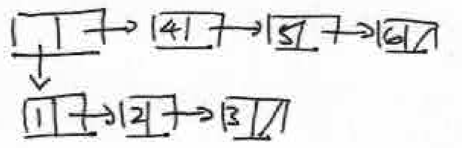
```
(define (fibo-helper n prev curr)
  (cond ((= n 0) curr)
        (else (fibo-helper (- n 1) curr (+ prev curr)))))

(define (fibo n)
  (fibo-helper n 0 1))
```

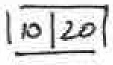
Again, we use a helper function to store state. Here, we keep track of our previous and current values and increment them appropriately. In each subsequent level of recursion, previous becomes current, and current becomes the sum of the two. We define `fibo` to call our helper function with the appropriate starting values.

Note: For number 4, there are two copies of the list a shown for clarity's sake. In reality, they are one copy, so any arrows pointing to them really point to the same object in memory.

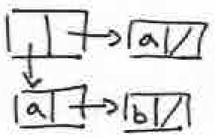
1



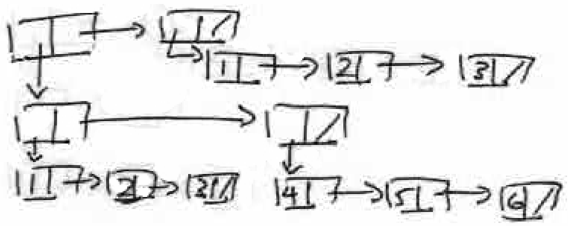
2



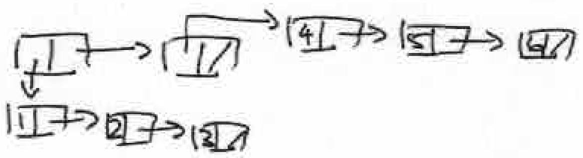
3



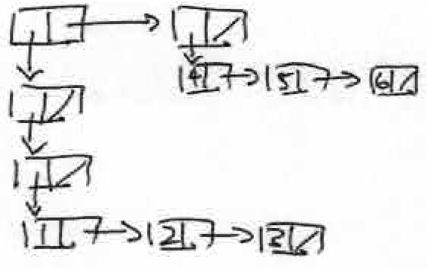
4



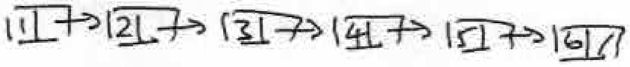
5



6



7



8 Error

9