Intro

Let's take some time to talk about LISP. It stands for LISt Processing — a way of coding using only lists! It sounds pretty radical, and it is. There are lots of cool things to know about LISP; if you're interested you can look up the benefits of functional programming. That's not part of the class, so we won't go over it in this note. We will start out by learning the building blocks of scheme, then we will talk about lists in Scheme, and finally we will try some practice problems. There will also be a list of every Scheme keyword, with examples, at the end of this note.

Scheme Basics

Scheme is one of a few dialects of LISP. If you read the intro, it should then be no surprise that Scheme is all about lists. In fact, *everything* in Scheme is either a primitive or a list! Let's explore the primitives first, and then we will talk more about lists. Don't worry about the `if` and `lambda` syntax in the examples below, we will talk about those later.

```
scm> 5
     5

scm> 3.14
     3.14
```

Numbers are the primitive that we'll be dealing with the most. Unlike in Python, there is no difference between float (decimal) and integer (whole number) values. When a number is evaluated in Scheme, we get the value of the number, as you might expect.

```
scm> #t
     #t

scm> (if nil 5 10)
     5

scm> (if 0 5 10)
     5
```

Booleans are another primitive in Scheme. There are two of them: `#t` and `#f`. These are just like `True` and `False` in Python. Recall that in Python we could also use numbers and strings instead of booleans. Scheme is also like that, but everything actually evaluates to `#t`, except for `#f` itself. Even `0` and `nil` are true values in Scheme.

```
scm> +
     #[+]

scm> fred
     (lambda (x) x)
```

Symbols are the last kind of Scheme primitive. They look a bit like strings, but it's better to think of them like names. Symbols are the names bound to variables, functions, etc. For example, when we evaluate the symbol `+`, Scheme responds by writing `#[+]`. This is it's way of saying the name + is bound to the addition function. If we define a function called `fred` that just returns its input, then evaluating the name `fred` would give us the output in the example to the left. This is Scheme's way of saying `fred` is bound to that lambda function.

Make sure you have a solid understanding of these different Scheme primitives before moving on to the next page.

As it turns out, everything else in Scheme is a list. Take this one for example:

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

It may not be totally obvious, but this really is a list. It's important to understand why, if you want to get to know how Scheme works. Let's break it down into individual elements:

1. The first element in the list is the symbol `+`.
2. The second element in the list is the number `1`.
3. The third element in the list is the number `2`.

So, if this is a list, then how come it returns a number? Well, when Scheme reads a list, it always assumes that the first element is a symbol bound to some function. It also assumes the other elements in the list are arguments for that function. So when it reads `(+ 1 2)`, Scheme looks up the name `+`. It finds the addition function, and then adds the numbers `1` and `2`. This produces the final output `3`.

What if we just want a list of numbers, like `(1 2)`? Let's try it in the interpreter:

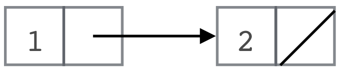
```
scm> (1 2)
     Error: cannot call: 1
```

Remember, Scheme assumes the first element in the list is a symbol bound to a function. When it reads `(1 2)`, it looks for a function bound to the name `1`, but it can't find any because `1` is a number. That's why we get the error message. It's trying to call `1` like a function, but it can't. Okay then, is it possible for us to have Scheme output the list `(1 2)`? Yes, we just need a function that can return the list `(1 2)`. This is where **`cons`** and **`list`** come in.

Understanding Scheme Lists

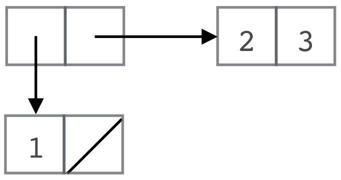
In order to make Scheme lists, we need to know how they work. In fact, they are implemented just like linked lists. Here are a few examples:

`(1 . (2 . ()))`

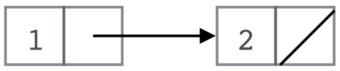


Notice that each dot represents the middle of a link.

`((1 . ()) . (2 . 3))`



You might have noticed that the Scheme lists written on the previous page were a little different from what the interpreter might produce. For reference, here's the first example from before:



Our answer: `(1 . (2 . ()))`
Scheme's answer: `(1 2)`

Why doesn't our answer match the interpreter? Well, it turns out that both are correct. The reason Scheme writes `(1 2)` is because that version is a lot shorter and easier to read. However, the answer that we came up with is what it would look like, if written out completely. When you're writing Scheme and you are confused about what a list should look like, I recommend writing out the long version first. Students tend to make fewer mistakes that way. Then, you can convert to Scheme's shorter version by cancelling each dot with the pair of parentheses immediately after it. This cancellation process is drawn below, for both examples from the previous page:

```
(1 . (2 . ()))        —> (1 2 . ())     —> (1 2)
((1 . ()) . (2 . 3))  —> ((1) . (2 . 3)) —> ((1) 2 . 3)
```

Notice there is still a dot in the second expression. That's because there is no pair of parentheses immediately after, so we can't cancel it with anything. Make sure you understand everything up to this point, before continuing.

So, how do `cons` and `list` fit into all of this? They are just built-in functions that return lists. But they're very important and they work a little differently from one another, so let's go over them in some detail.

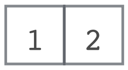
Continue on the next page.

`cons` always takes exactly two arguments.

In terms of Scheme lists, you can think of `cons` like this*: `(cons ‘a ‘b) —> (a . b)`

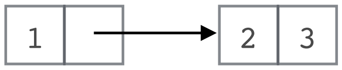
Or in terms of box-and-pointer diagrams, `cons` returns a single link where the first argument is in the first box, and the second argument is in the second box. Make sure you understand the examples below:

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



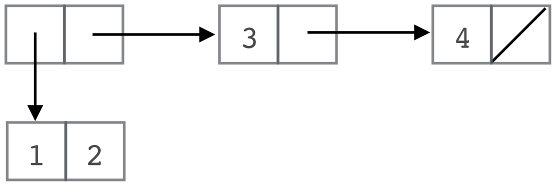
```
scm> (cons 1 (cons 2 3))
     (1 2 . 3)
```

long version is `(1 . (2 . 3))`



```
scm> (cons (cons 1 2) (cons 3 (cons 4 nil)))
     ((1 . 2) 3 4)
```

long version is `((1 . 2) . (3 . (4 . ())))`

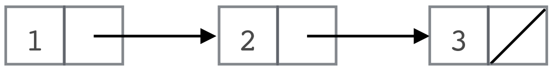


`list` supports any number of arguments.

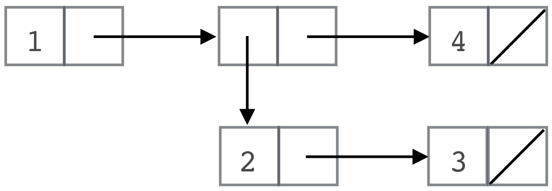
In terms of Scheme lists, you can think of `list` like this*: `(list ‘a ‘b ‘c) —> (a b c)`

Or in terms of box-and-pointer diagrams, `list` returns a sequence of links where each argument is the first element in one of the links. Make sure you understand the examples below:

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



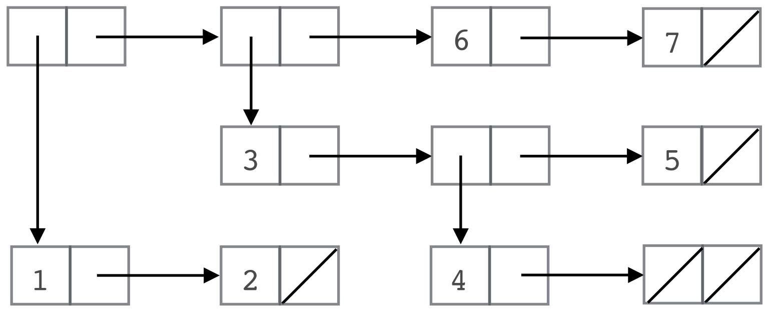
```
scm> (list 1 (list 2 3) 4)
     (1 (2 3) 4)
```



* Don’t worry about the quotes in this expression for now, we’ll talk about those later.

This last example is a bit tricky. It is much easier if you analyze just one element at a time. Start with `(1 2)`. The first element of our linked list is a pointer to that. Next work out the second element: `(3 (4 ()) 5)`. This one on its own can be hard, so do the same thing — break it into one element at a time: `3`, then the list resulting from `(4 ())`, and finally `5`. Keep going until you have the whole thing:

```
scm> (list (list 1 2) (list 3 (list 4 nil) 5) 6 7)
     ((1 2) (3 (4 ()) 5) 6 7)
```



To sum up:
* `cons` takes 2 things, and puts them both in the same link.
* `list` takes any number of things, and puts them in sequential links.

Tip: If you want extra practice converting between Scheme lists and box-and-pointer diagrams, go to scheme.cs61a.org. Type in the command `(demo 'autopair)`. Now try constructing a few lists using the `cons` and `list` commands; you should notice a corresponding box-and-pointer diagram gets drawn to the screen. Try predicting some complicated ones, and check your correctness in the interpreter. If you get something wrong, take the time to figure out why you thought it was something else. That way, you don't make the same mistake again.

Caution: Sometimes students think `list` is necessary because they don't want any dots in their answer. This is the case when we're `cons`-ing two numbers, but remember that applying `cons` on a number and a list will let us cancel the dot with the parentheses. Meanwhile, applying `list` on a number and a list would return a nested list. Always take the time to analyze which one you need, based on your arguments. Write out the long version of the Scheme list if you have to. It definitely matters that you pick the right function for the right situation.

Before going on, make sure you understand everything so far.

Scheme in a Spreadsheet

Even though Scheme is not as big a language as Python, it still has a lot of built-in keywords. These are nothing more than functions the developers of Scheme decided they would implement for you, to save you some time. You will be expected to know the functions that follow, and you should know how to use them. Make note of the subtleties, like the difference between `pair?` and `list?`, or between `cons` and `list`.

If you find yourself wishing for more examples, just go to scheme.cs61a.org and play around with the interpreter. There, you will be able to answer any questions you have about Scheme, simply by writing out some examples and analyzing the results.

With that said, here is Scheme in a spreadsheet. Don't let it intimidate you; work through each function one by one and make sure you get how they all work.

Description	Syntax

`if` takes 3 arguments. If the first one is `#t`, then it returns the second one. Otherwise it returns the third one. Remember all values in Scheme are `#t`, except for the boolean `#f` itself.	`(if` *`condition`*
	    `do-if-true`
	    `do-if-false``)`
	―――――――――――
	`scm> (if #t 5 10)`
	     `5`
	`scm> (if #f 5 10)`
	     `10`
	`scm> (if 0 1 2)`
	     `1`
	`scm> (if nil nil nil)`
	     `()`

`cond` is like an if-elif-else chain. It can take any number of arguments. Each argument has to be a list of 2 elements, where the first element is the condition and the second element is the action to be taken on that condition. Note that only one action can be executed from a `cond`. The keyword `else` is a condition that always evaluates to `#t`; it should be the last condition if you want to use it.	`(cond (`*`case1 action1`*`)`
	      `(`*`case2 action2`*`)`
	      `(`*`case3 action3`*`))`
	―――――――――――
	`scm> (cond (#f nil)`
	`...         ((= 1 2) 5)`
	`...         (else 1))`
	     `1`
	`scm> (cond`
	`...    ((null? '(1)) 5)`
	`...    ((null? '()) 42)`
	`...    ((null? '()) 7))`
	     `42`

`and` in Scheme works just like it does in Python. It returns the first `#f` argument, or if all arguments are `#t`, it returns the last one. Remember, all values in Scheme are `#t` (including `0` and `nil`) except for the boolean `#f` itself.

```
(and bool1
     bool2
     bool3)
```

```
scm> (and 0 nil 5)
     5
scm> (and ‘a #t #f)
     #f
```

`or` in Scheme works just like it does in Python. It returns the first `#t` argument, or if all arguments are `#f`, it returns the last one.

```
(or bool1
    bool2
    bool3)
```

```
scm> (or nil (/ 1 0))
     ()
scm> (or (/ 1 0) ‘a)
     error
```

`not` simply returns `#t` if its argument is `#f`, or `#f` if its argument is `#t`.

```
(not bool1)
```

```
scm> (not (and (+ 5 7) nil))
     #f
scm> (not #f)
     #t
```

`define` takes 2 arguments. The first one is a list where the first element is the name of the function and all other elements are the function’s parameters. The second argument is what that function should evaluate to, in terms of its parameters.
If the first element is not a list, then you will define a variable instead of a function.

```
(define (f p1 p2 p3)
        do-something)
```

```
scm> (define a 10)
     a
scm> a
     10
scm> (define (b) 10)
     b
scm> b
     (lambda () 10)
scm> (b)
     10
```

`lambda` is just like `define`, except the first argument doesn’t contain a function name. That is — the first argument in the `lambda` is just a list of parameters, and the second argument in the `lambda` is what to return in terms of those parameters.

```
(lambda (p1 p2 p3)
        do-something)
```

```
scm> ((lambda (x) (+ x 1)) 5)
     6
scm> (define a (lambda (x)
...                (cons x nil)))
     a
scm> (a 5)
     (5)
```

`let` can be confusing, so make sure you practice it! It takes two arguments:

* The first argument in the `let` is a list, where each element is a pair containing a variable name and a value for that variable.
* The second argument in the `let` is just some expression, in terms of the variables you defined.

Note that the variables computed in the second argument *cannot* rely on each other! See the example at right, where **b** is defined in terms of the global value of **a** instead of the local value of a.

```
(let ((var1 value1)
      (var2 value2)
      (var3 value3))
     body)
```

```
scm> (define a 5)
     a
scm> (let ((a 50)
...         (b (+ a 1)))
...        (- a b))
     44
scm> (let ((x (lambda (y) y))
...         (y (lambda (x) x)))
...        (+ (x 5) (y 10)))
     15
```

car takes a list as its argument. It returns the first element of that list.

```
(car lst)
```

```
scm> (car (list (cons 1 2)
...             nil ‘(3)))
     (1 . 2)
```

cdr takes a list as its argument. It returns the second element of that list. Since Scheme lists are linked lists, keep in mind this second element will also be a list, if the argument is well-formed.

```
(cdr lst)
```

```
scm> (cdr (list (cons 1 2)
...             nil ‘(3)))
     (() (3))
```

cons takes 2 arguments, and puts them both in the same link. See the previous section of this note for more details and examples.

```
(cons elem1 elem2)
```

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

`list` takes any number of arguments, and puts them in sequential links. See the previous section of this note for more details and examples.

```
(list el1 el2 el3)
```

```
scm> (list 1 2)
     (1 2)
scm> (list)
     ()
```

`length` takes in a list and returns the length of that list.

```
(length lst)
```

```
scm> (length nil)
     0
scm> (length ‘(1 2 3))
     3
scm> (length ‘(1 2 . 3))
     error
```

`append` takes any number of arguments. It returns all its arguments joined into one list, and it errors if the resulting list is not a valid Scheme list.

```
(append el1 el2 el3)
```

```
scm> (append (list 1 2 3)
...          (cons 4 5))
     (1 2 3 4 . 5)
scm> (append (list 1 2 3)
...          (cons 4 nil)
...          5)
     (1 2 3 4 5)
scm> (append (list 1 2 3)
...          (cons 4 5)
...          6)
     error
scm> (append (list 1 2 3)
...          (cons 4 5)
...          '(1))
     error
```

`boolean?` returns `#t` if its argument is a boolean, otherwise it returns `#f`.

```
(boolean? arg)
```

```
scm> (boolean? 5)
     #f
scm> (boolean? #f)
     #t
```

`integer?` returns `#t` if its argument is a whole number, otherwise it returns `#f`.

```
(integer? arg)
```

```
scm> (integer? 'a)
     #f
scm> (integer? (cons 1 nil))
     #f
scm> (integer? 3.14)
     #f
```

`equal?` returns `#t` if its 2 arguments evaluate to the same thing, otherwise it returns `#f`.

```
(equal? arg1 arg2)
```

```
scm> (equal? (list 1 2 3)
...          (cons 1 (list 2 3)))
     #t
scm> (equal? 5 (+ 4 1))
     #t
scm> (equal? + +)
     #t
```

= is the same as `equal?` except its arguments have to be numbers. Otherwise it errors.

```
(= num1 num2)
```

```
scm> (= (list 1 2 3)
...     (cons 1 (list 2 3)))
     #f
scm> (= 5 (+ 4 1))
     #t
```

`even?` returns `#t` if its argument is an even number, otherwise it returns `#f`.	`(even?` *`num`*`)`
	`scm> (even? 0)`
	`     #t`

`odd?` returns `#t` if its argument is an even number, otherwise it returns `#f`.	`(odd?` *`num`*`)`
	`scm> (odd? nil)`
	`     error`
	`scm> (odd? 0)`
	`     #f`

`null?` returns `#t` if its argument is `nil`, a.k.a. `()`, otherwise it returns `#f`.	`(null?` *`arg`*`)`
	`scm> (null? ‘())`
	`     #t`
	`scm> (null? nil)`
	`     #t`

`list?` returns `#t` if its argument is a well-formed list (i.e., no dots), otherwise it returns `#f`.	`(list?` *`arg`*`)`
	`scm> (list? (cons 1 nil))`
	`     #t`
	`scm> (list? (cons 1 1))`
	`     #f`

`pair?` is like `list?` except it allows malformed lists too (i.e., dots are fine).	`(pair?` *`arg`*`)`
	`scm> (pair? (cons 1 nil))`
	`     #t`
	`scm> (pair? (cons 1 1))`
	`     #t`

`quote` simply tells Scheme not to evaluate its argument, so it returns the unevaluated argument without the quote. Notably, Scheme *will* try to make the argument a well-formed list if possible. It will also error if there is more than one element after any dot in the expression, since that's not a valid Scheme list.	`(quote` *`arg`*`)` OR `‘`*`arg`*
	`scm> (quote a)`
	`     a`
	`scm> (quote (cons 1 2 3 hi oops))`
	`     (cons 1 2 3 hi oops)`
	`scm> ‘(1 . (2 . (five hello)))`
	`     (1 2 five hello)`
	`scm> ‘(1 . () . . . 5)`
	`     error`
	`scm> ‘(1 . (2 . ()) five)`
	`     error`

`eval` returns what Scheme *would* return, if Scheme were to evaluate the argument.	`(eval arg)`
	`scm> (eval '(list 1 2))`
	`     (1 2)`
	`scm> (eval ''(even? 5))`
	`     (even? 5)`
	`scm> (eval (eval ''(even? 5)))`
	`     #f`
	`scm> ((eval '+) 1 2)`
	`     3`

Building Scheme Lists Recursively

A *lot* of the Scheme problems you see — probably most of them — will ask you to return a list. That sounds like a handy thing to know how to do, so let's practice. In general, we want to stick to this procedure:

1. Pick a base case. Since we're building a list, the condition should be whatever assures us that we should return the empty list, `nil`. Most of the time `(null? lst)` is a good guess, where `lst` is some list we got as a parameter, but that's not always the case.
2. Now we build our list recursively. If one of our arguments is a list, then each recursive call should only take any action with respect to *one* element in that list — the first one. We will take care of all the other elements when we recurse on the rest of the list. Note that there are many ways to "take action" with respect to the first element in the list. We could want to add it to a list of our own, or we might want to apply some function on it first. We might even use it to change the rest of the list before recursing. The point is, when we're in a recursive call, we pretend the first element is the only value we care about.

Practice: repeat

This one is from last semester's discussion worksheet. Define a function `repeat` that takes in parameters `x` and `n`. It should return a list of the number `n`, repeated `x` times. For example:

```
scm> (repeat 3 5)
     (5 5 5)
```

The first step is to determine a base case. Following the procedure above, let's think about what parameters would make us return `nil`. This is the same as asking what parameters would make us return a list of length 0. That happens when we repeat `n` a total of 0 times, so our base case is `(= x 0)`. We have the following:

```
(define (repeat x n)
        (if (= x 0)
            nil
            ...
        )
)
```

How about when `x` is not `0`? Then we need a list containing `n`, and a recursive call to give us the number `n`, repeated `x-1` more times. `n` is a number, and the recursive call will give us a list, so should we use `cons` or `list`? `cons` would give us something like `(n .` (recursive call)`)`, but the dot cancels with the parenthesis so we end up with a flat list. `list`, on the other hand, gives us `(n` (recursive call)`)`, so we end up with a nested list. We choose `cons`, since we desire a flat list. Here is the final scheme program, side by side with the Python equivalent. Check that you understand both:

```
(define (repeat x n)                      def repeat(x, n):
        (if (= x 0)                           if x == 0:
            nil                                   return []
            (cons n (repeat (- x 1) n))       return [n]+repeat(x-1,n)
        )
)
```

Caution: Sometimes it can be helpful to write the Python version of a program, as we have done here. However, a lot of the time this can just be more confusing. It happens to have worked out well in this case, but don't rely on that being so.

Practice: filter
We have a function `f` that takes a single number as its argument, and it returns either `#t` or `#f`. We also have a list of numbers, called `lst`. How can we get *only* the elements of `lst` for which `f` returns `#t`?

The first step is to determine a base case. We should ask ourselves, when do we want to return `nil`? The answer is when `lst` itself is `nil`, because filtering an empty list should give us an empty list. So that makes our base case `(null? lst)`. Here's what we have so far:

```
(define (filter f lst)
        (if (null? lst)
            nil
            ...
        )
)
```

Now, recall from the procedure above that we want to build our desired list *one* element at a time. That means that right now we're only concerned with the first element in `lst`. Either we want it, or we don't. Let's think about each case:

* If we want it, then we should have a list where the first element is `(car lst)` and the rest of the elements come from a recursive call on the rest of `lst`.
* If we don't want it, that means we only want the recursive call. `(car lst)` should *not* be at the beginning.

But how do we know if we want `(car lst)`? From the description of `filter`, we should only use it if `(f (car lst))` is `#t`. A few of my students have suggested writing the following:

```
(if (= (car lst) #t)
    ... ; we want (car lst)
    ... ; we don't want (car lst)
)
```

This works, but notice that `(f (car lst))` itself evaluates to a boolean, so we can directly use it as the condition in the `if` statement, like this:

```
(define (filter f lst)
        (if (null? lst)
            nil
            (if (f (car lst))
                ... ; we want (car lst)
                ... ; we don't want (car lst)
            )
        )
)
```

This is a valid Scheme expression, but it might be cumbersome to read. Let's try converting to a `cond` statement. Look at the previous section if you need to review the correct syntax.

```
(define (filter f lst)
        (cond ((null? lst) nil)
              ((f (car lst)) ...) ; we want (car lst)
              (else ...)          ; we don't want (car lst)
        )
)
```

Check to see you agree this is the same as what we had before. Now all there's left to do is fill out each of the conditional actions. If we want `(car lst)`, then we'll need to join a number to the list we get from our recursive call. As in the previous example, we will need to use `cons` in order to get a flat list. If you don't understand why, take a look back to *Example 1*.
On the other hand, when we *don't* want to use `(car lst)`, then we somehow have to get a list from only our recursive call. But the recursive call *is* a list, so that's not a problem. If you ever find yourself thinking, "I want a `cons` but I don't know what to `cons` this with …", chances are you don't need to `cons`. This is a common mistake when you're just learning Scheme — remember, if you already have the list you want there's no need to join it with anything else.
Putting that all together, we get this:

```
(define (filter f lst)
        (cond ((null? lst) nil)
              ((f (car lst)) (cons (car lst) (filter f (cdr lst))))
              (else (filter f (cdr lst)))
        )
)
```

If this is a bit confusing, try writing out a few iterations on paper. You can also try drawing an example using the `(demo 'autopair)` command on scheme.cs61a.org.

Practice: no-repeats
Once we have defined a function like `filter`, we can use it to do more complicated things. For example, imagine I have a list called `lst`, but there are a lot of repeated elements in it. `lst` might look like `(1 1 4 3 2 5 2 4 1)`. How do we get the same list, but with all the repeats eliminated? For this example I would be looking for the list `(1 4 3 2 5)`. We are allowed to use the `filter` function. Here's the signature:

```
(define (no-repeats lst) ...)
```

Step one is the base case. When should we return `nil`? Just like in the last example, we return `nil` if `(null? lst)`, so we'll make that our base case. For reference, here's the function:

```
(define (no-repeats lst)
        (if (null? lst)
            nil
            ...
        )
)
```

Now let's think about how we plan to build our list of non-repeated elements. From the procedure we came up with earlier, we know that we should *only* care about the first element right now. That means the problem is no longer finding *all* the repeats in `lst`; rather we are now only trying to find all the repeats of `(car lst)` in `lst`. The rest will be handled when we recurse on the rest of the elements. Take a moment to think about the content of this paragraph — really, *really* understand why it works. This is important.

Okay, so then how do we get rid of all the repeats of `(car lst)` from `lst`? Recall that we have the `filter` function from before. Perhaps try doing this part on your own; keep reading once you've given it a go.

We're trying to filter out all the duplicates of `(car lst)` from `(cdr lst)`. So, how do make that call to `filter`? We will have to come up with the right arguments. `filter` takes two of them: a function, and a list on which to apply the function. We should be applying our function on `(cdr lst)`, since we've decided that's what we're filtering. That gives us the following code, where the filtering function will go on the blank line when we're done:

```
(define (no-repeats lst)
        (if (null? lst)
            nil
            (cons (car lst)
                  (filter _____ (cdr lst))
            )
        )
)
```

Check that this all makes sense so far.

We need to filter for duplicates of `(car lst)`. That means our function has to return `#t` for things that don't equal `(car lst)`, and `#f` for things that equal `(car lst)`. We could achieve this relatively easily by nesting a `define` up at the top of `no-repeats`, but let's do it with a `lambda` instead to get some practice. It will take in one argument, a number, and it will output a boolean. The syntax looks like this:

```
(lambda (x) _____) ; x is a number
```

The rest is just converting from English to Scheme: to check that `x` does not equal `(car lst)`, we simply write `(not (= x (car lst)))`. Make sure the final `lambda` expression makes sense to you:

```
(lambda (x) (not (= x (car lst))))
```

Then we just have to plug it in:

```
(define (no-repeats lst)
        (if (null? lst)
            nil
            (cons (car lst)
                  (filter (lambda (x) (not (= x (car lst)))) (cdr lst)
                  )
            )
        )
)
```

Let's walk through what this does. First, we check that our list has at least one element. This was our base case. Next we decide that the first element must not be a repeat, since it is the first one. That's why we include it in the list we're returning, using `cons`. Then the `filter` function will check that this first element never occurs again in `(cdr lst)`. It mostly works, but something is missing. What is it?

Remember how we only want to take action regarding *one* element at a time, when we're recursively building a list? There was an assumption there, which was that we'd get to all the other elements recursively. Thus, a quick check on the correctness of your program is just to look for a recursive call. If there isn't one, the odds are you missed something. Where should we put that recursive call in `no-repeats`? It has to operate on all the elements that are after the first element, so we should recursively call on the result from `filter`. It is equally good to call it on `(cdr lst)` inside of `filter`. Here's our finished product, with the recursive call included:

```
(define (no-repeats lst)
        (if (null? lst)
            nil
            (cons (car lst)
                  (no-repeats
                      (filter (lambda (x) (not (= x (car lst))))
                              (cdr lst)
                      )
                  )
            )
        )
)
```

Always remember your recursive call when you're doing problems like this one. Now try writing out a few iterations of this function, to convince yourself that it works. Also go back and identify how we were able to arrive at the solution by following the procedure detailed at the beginning of this section.

Happy scheming.