Quoting

We've thrown around the `'` construct on a number of occasions, but we have not formally defined it. What does it do, and when it is appropriate?

`'` is a shorthand for the `(quote ...)` function. `quote` instructs the Lisp runtime not to evaluate its argument. Instead, the argument should be taken at "face value." In many languages, including Lisp, face values are known as literals.

What does this mean? We've seen `'` used to create lists, as in `'(a b c d)`. When an expression is given to the Lisp interpreter, it is normally evaluated to yield a value. Quoting tells Lisp to skip the evaluation step: the value is the quoted expression. In other words, the meaning of `'(a b c d)` is whatever the expression `(a b c d)` has in the CLISP REPL.

Be careful with this rule. Just because quoting tells the Lisp interpreter not to evaluate the expression, this does not mean that the quoted expression is never evaluated. Take the expression `(cons 'a '(b c d))`. When evaluating this expression, the interpreter first attempts to evaluate `cons`. `cons` has two arguments: `'a` and `'(b c d)`. The interpreter evaluates each in turn. When evaluating `'a`, the value is the literal `a`, which is a symbol. Likewise, `'(b c d)` is the literal `(b c d)` which is a list of symbols. `cons` now knows that it must create a `cons` cell consisting of `a` on the left and the list `(b c d)` on the right; chains of `cons` cells of this form are lists, so the result is the list `(a b c d)`.

To prove the equivalence of quoted expressions to their evaluated counterparts, consider the following expression: `(equal '(a . b) (cons 'a 'b))`. The result of this expression is `T` because `(a.b)` is the result of `(cons 'a 'b)`.

`mapcar` with fancy lists

The activity in class asked you to write a function called `firsts` that returns the left element of every `cons` in a list of `cons` cells.

Using the above information about quoting, let's first create a list of `cons` cells:

```
'( (a.b) (c.d) (e.f) (g.h) )
```

Since the left element of a `cons` is the `car`, the `car` of one of our `cons` cells, say `(a.b)` is `a`. Thus applying `firsts` to the entire list will produce:

```
(a c e g)
```

This sounds like a job for `mapcar`, which applies a function to each element of a list. But what's our function? We already know that we need to get the `car`. So all we really need to do is combine the two:

```
(mapcar #'car '( (a.b) (c.d) (e.f) (g.h) )
```

What's the deal with `'#`?

`'#` is shorthand for the function `(function ...)`. But why do we need it?

It turns out that variable names in Common Lisp (and CLISP in particular) are looked up differently depending on whether they store ordinary values (like `1`) or function values.

`(function ...)` or `#'` tells Lisp to retrieve the function value for the given variable. If you omit `#'` when you need a function value, Lisp will go looking for the ordinary value for the variable.

The rules are a tad more subtle than this, depending on whether Lisp is expecting an ordinary value or a function value. There's a nice explanation here: `https://stackoverflow.com/a/665673/480764`.