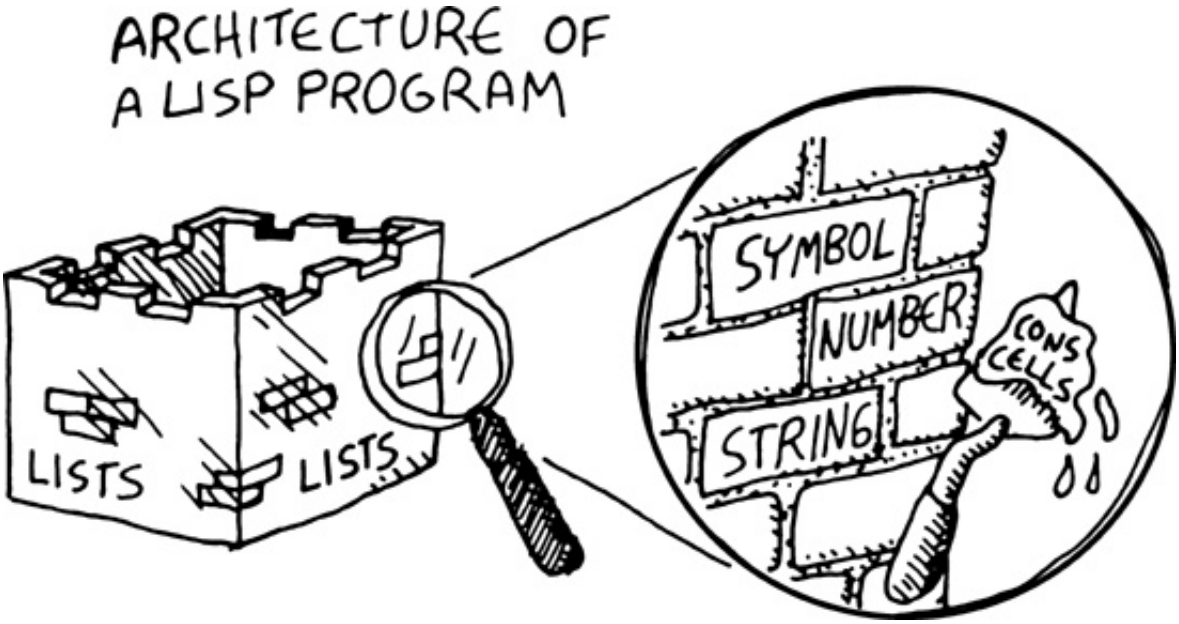

Lists in Lisp

Lists are a crucial feature in Lisp. They are what hold all your Lisp code (as well as data) together. Take any basic piece of Lisp code, such as the following:

```
(expt 2 3)
```

This piece of code contains a symbol (`expt`) and two numbers, all tied together as a list, indicated by the parentheses.

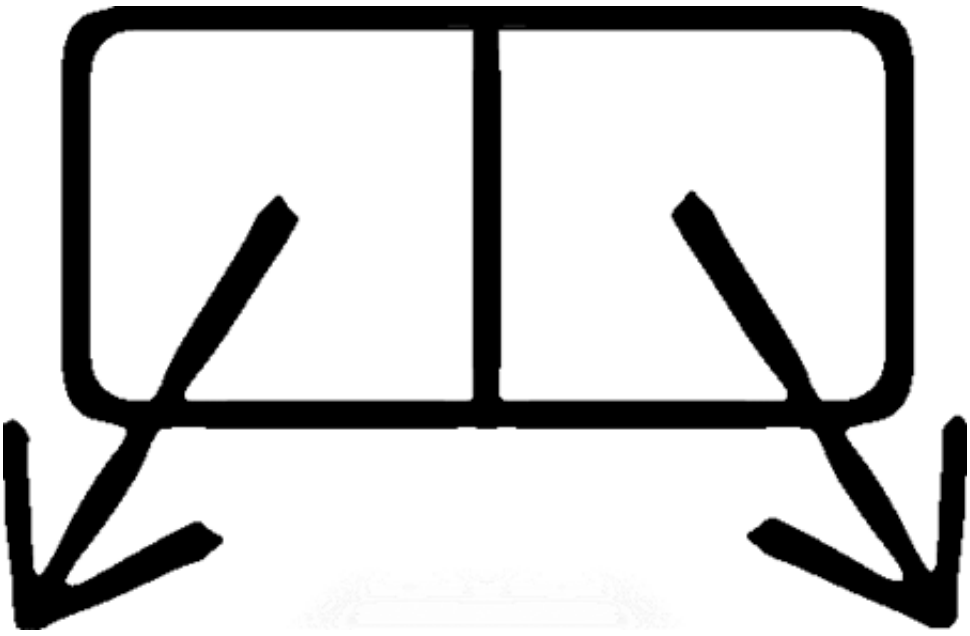


You can think of a Lisp program as a house. If you were to build a house in Lisp, your walls would be made out of lists. The bricks would be made out of symbols, numbers, and strings. However, a wall needs mortar to hold it together. In the same way, lists in Lisp are held together by structures called *cons cells*.

Cons Cells

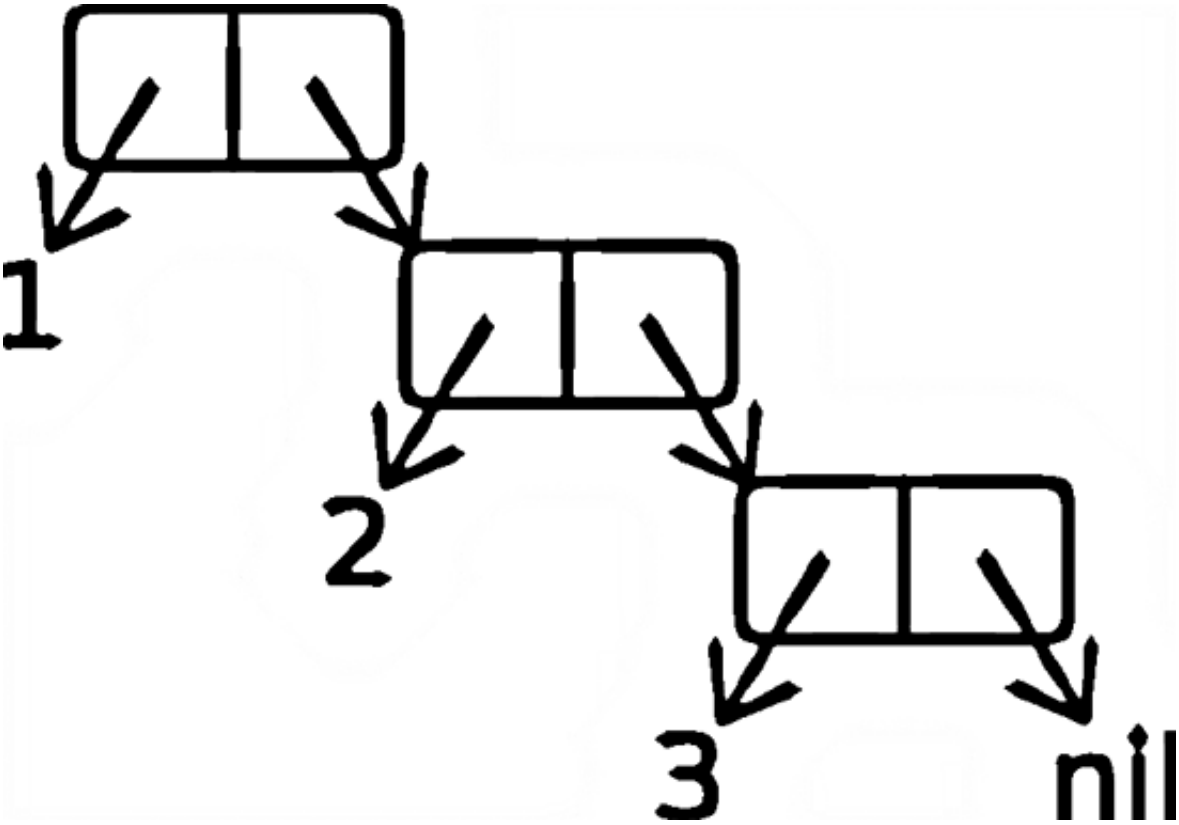
Lists in Lisp are held together with cons cells. Understanding the relationship between cons cells and lists will give you a better idea of how Lisp works.

A cons cell looks like this:



It's made of two little connected boxes, both of which can point at other things. A cons cell can point to another cons cell or another type of Lisp data. By being able to point to two different things, it's possible to link cons cells together into lists. In fact, lists in Lisp are just an abstract illusion—all of them are actually composed of cons cells.

For instance, suppose we create the list '(1 2 3). Here's how this list is represented in computer memory:



It's created using three cons cells. Each cell points to a number, as well as the next cons cell for the list. The final cons cell then points at `nil`, to terminate the list. (If you've ever used a linked list in another programming language, this is the same basic idea.) You can think of this arrangement like a calling chain for your friends: "When I know about a party this weekend, I'll call Bob, and then Bob will call Lisa, who will call . . ." Each person in a calling chain is responsible for only one phone call, which activates the next call in the list.

List Functions

Manipulating lists is extremely important in Lisp programming. There are three basic functions for manipulating cons cells (and hence lists) in Lisp: `cons`, `car`, and `cdr`.

The cons Function

If you want to link any two pieces of data in your Lisp program (regardless of type), the usual way to do that is with the `cons` function. When you call `cons`, the Lisp compiler typically allocates a small chunk of memory, the cons cell, that can hold two references to the objects being linked. (Usually, the second of the two items being linked will be a list.) For example, let's link the symbol `chicken` to the symbol `cat`:

```
> (cons 'chicken 'cat)
(CHICKEN . CAT)
```

As you can see, `cons` returns a single object, the cons cell, represented by parentheses and a dot between the two connected items. Don't confuse this with a regular list. The dot in the middle makes this a cons cell, just linking those two items together.

Notice how we prefix our two pieces of data with a single quote to make sure that Lisp sees them as just data and doesn't try to evaluate them as code.

If instead of another piece of data, we attach the symbol `nil` on the right side of the list, something special happens:

```
> (cons 'chicken 'nil)
(CHICKEN)
```

Unlike with our `cat`, the `nil` does not show in the output this time. There's a simple reason for this: `nil` is a special

symbol that is used to terminate a list in Lisp. That said, the Lisp REPL is taking a shortcut and just saying that we created a list with one item, our `chicken`. It could have displayed the result by explicitly showing our cons cell and printing `(CHICKEN . NIL)`. However, because this result is coincidentally also a list, it instead will show the list notation.

The lesson here is that Lisp will always go out of its way to "hide" the cons cells from you. When it can, it will show your results using lists. It will show a cons cell (with the dot between the objects) only if there isn't a way to show your result using lists.

The previous example can also be written like this:

```
> (cons 'chicken ())
(CHICKEN)
```

The *empty list*, `()`, can be used interchangeably with the `nil` symbol in Common Lisp. Thinking of the terminator of a list as an empty list makes sense. What do you get when you add a chicken to an empty list? Just a list with a chicken in it. The `cons` function also can add a new item to the front of the list. For example, to add `pork` to the front of a list containing `(beef chicken)`, use `cons` like so:

```
> (cons 'pork '(beef chicken))
(PORK BEEF CHICKEN)
```

When Lispers talk about using `cons`, they say they are *consing* something. In this example, we consed pork to a list containing beef and chicken.

Since all lists are made of cons cells, our `(beef chicken)` list must have been created from its own two cons cells, perhaps like this:

```
> (cons 'beef (cons 'chicken ()))
(BEEF CHICKEN)
```

Combining the previous two examples, we can see what all the lists look like when viewed as conses. This is what is *really* happening:

```
> (cons 'pork (cons 'beef (cons 'chicken ())))
(PORK BEEF CHICKEN)
```

Basically, this is telling us that when we cons together a list of three items, we get a list of three items. No wholesale copying or deleting of data ever needs to take place.

The REPL echoed back to us our entered items as a list, `(pork beef chicken)`, but it could just as easily (though a little less conveniently) have reported back the items exactly as we entered them: `(cons 'pork (cons 'beef (cons 'chicken ())))`. Either response would have been perfectly correct. *In Lisp, a chain of cons cells and a list are exactly the same thing.*

The car and cdr Functions

Lists are just long chains of two-item cells.

The `car` function is used for getting the thing out of the *first* slot of a cell:

```
> (car '(pork beef chicken))
PORK
```

The `cdr` function is used to grab the value out of the *second* slot, or the remainder of a list:

```
> (cdr '(pork beef chicken))
(BEEF CHICKEN)
```

You can string together `car` and `cdr` into new functions like `cadr`, `cdar`, or `cadadr`. This lets you succinctly extract specific pieces of data out of complex lists. Entering `cadr` is the same as using `car` and `cdr` together—it returns the second item from a list. (The first slot of the second cons cell would contain that item.) Take a look at this example:

```
❶ > (cdr '(pork beef chicken))
(BEEF CHICKEN)
```

```
❷ > (car '(beef chicken))
BEEF
❸ > (car (cdr '(pork beef chicken)))
BEEF
❹ > (cadr '(pork beef chicken))
BEEF
```

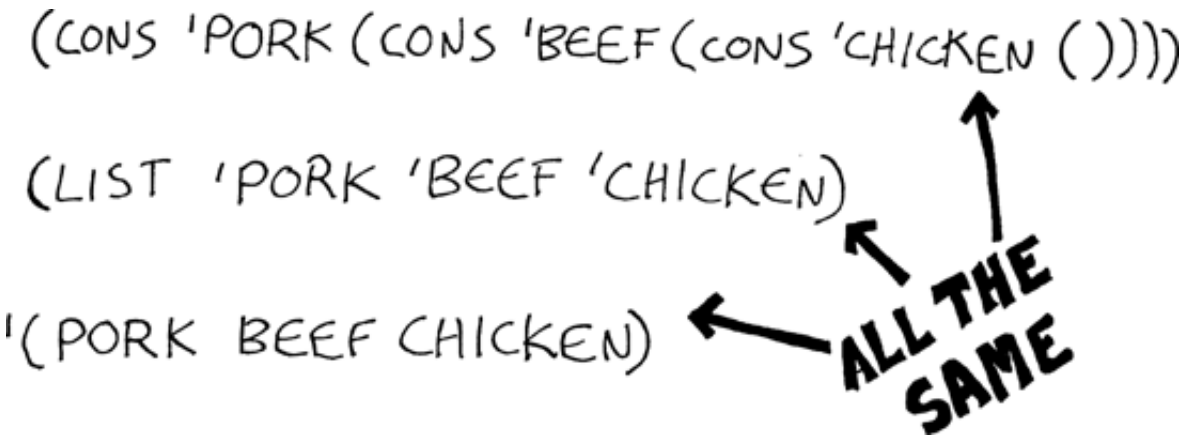
We know that `cdr` will take away the first item in a list ❶. If we then take that shortened list and use `car`, we'll get the first item in the new list ❷. Then, if we use these two commands together, we'll get the second item in the original list ❸. Finally, if we use the `cadr` command, it gives us the same result as using `car` and `cdr` together ❹. Essentially, using the `cadr` command is the same as saying that you want the second item in the list.

The list Function

For convenience, Common Lisp has many functions built on top of the basic three— `cons`, `car`, and `cdr`. A useful one is the `list` function, which does the dirty work of creating all the cons cells and builds our list all at once:

```
> (list 'pork 'beef 'chicken)
(PORK BEEF CHICKEN)
```

Remember that there is no difference between a list created with the `list` function, one created by specifying individual cons cells, or one created in data mode using the single quote. They're all the same animal.



Nested Lists

Lists can contain other lists. Here's an example:

```
'(cat (duck bat) ant)
```

This is a list containing three items. The second item of this list is `(duck bat)`, which is a list itself. This is an example of a *nested list*.

However, under the hood, these nested lists are still just made out of cons cells. Let's look at an example where we pull items out of nested lists. Here, the first item is `(peas carrots tomatoes)` and the second item is `(pork beef chicken)`:

```
❶ > (car '((peas carrots tomatoes) (pork beef chicken)))
(PEAS CARROTS TOMATOES)
❷ > (cdr '(peas carrots tomatoes))
(CARROTS TOMATOES)
❸ > (cdr (car '((peas carrots tomatoes) (pork beef chicken))))
(CARROTS TOMATOES)
❹ > (cdar '((peas carrots tomatoes) (pork beef chicken)))
(CARROTS TOMATOES)
```

The `car` function gives us the first item in the list, which is a list in this case ❶. Next, we use the cdr command to chop off the first item from this inner list, leaving us with (`CARROTS TOMATOES`) ❷. Using these commands together gives this same result ❸. Finally, using `cdar` gives the same result as using `cdr` and `car` separately ❹.

As demonstrated in this example, cons cells allow us to create complex structures, and we use them here to build a nested list. To prove that our nested list consists solely of cons cells, here is how we could create this nested list using only the `cons` command:

```
> (cons (cons 'peas (cons 'carrots (cons 'tomatoes ())))
        (cons (cons 'pork (cons 'beef (cons 'chicken ()))) ()))
((PEAS CARROTS TOMATOES) (PORK BEEF CHICKEN))
```

Here are some more examples of functions based on `car` and `cdr` that we could use on our data structure:

```
> (cddr '((peas carrots tomatoes) (pork beef chicken) duck))
(DUCK)
> (caddr '((peas carrots tomatoes) (pork beef chicken) duck))
DUCK
> (cddar '((peas carrots tomatoes) (pork beef chicken) duck))
(TOMATOES)
> (cadadr '((peas carrots tomatoes) (pork beef chicken) duck))
BEEF
```

Common Lisp already defines all these functions for you. You can use any function with the name `c*r` right out of the box, up to four levels deep. In other words, `cadadr` will already exist for you to use, whereas `cadadar` (which is five levels deep) does not (you would need to write that function yourself). These functions make it easy to manipulate cons cells-based structures in Lisp, no matter how complicated they might be.


BILLY & BOB
LISP PROGRAMMERS BY DAY
HIGH TECH BANDITS BY NIGHT
ALRIGHT BILLY! I'VE GOT A NEW JOB FOR US!
I DON'T KNOW, BOB... WE'RE MAKING GOOD MONEY LISP CONSULTING.
WHY BOTHER WITH CRIME ANYMORE?
WAIT TILL YOU HEAR MY PLAN!
THE "EMERALD OF GIBRALTAR" IS BEING MOVED BY BULLET TRAIN THIS WEEKEND...
IT'LL BE ON THE 2ND CARRIAGE FROM THE FRONT...
CONTROLS
CAMERA
GUARDS
I'M CONFUSED... IS THAT COUNTING THE LOCOMOTIVE?
NO...
WHAT ABOUT THAT THINGY WITH COAL IN IT??
DUDE, THIS IS A BULLET TRAIN... IT DOESN'T HAVE A "THINGY WITH COAL" IN IT!!!
LET ME PUT IT THIS WAY... IT'S IN THE "CADDR" OF THE TRAIN... ALSO, THERE'S A SECURITY CAMERA ABOVE THE 2ND SEAT OF THE FIRST CARRIAGE ... ER, I MEAN "IN THE CADADR OF THE TRAIN"


I'M GOING TO JUMP ONTO THE LOCOMOTIVE ...ER... "CAR" WHERE THE CAMERA IS CONTROLLED FROM AND SET IT RUNNING ON A LOOP.
TICKET
CONTROLS
CAMERA
GUARDS
FOR YOU, ON THE OTHER HAND, I BOUGHT A REGULAR PASSENGER TICKET FOR A SEAT IN THE CADDDDR... YOU HEAD TO THE FRONT AND STEAL THE EMERALD!
BUT WHAT ABOUT THESE GUARDS IN THE CADADDDR AND CADDDDADDDR???
WHEN YOU GET TO THE CADDDR, YOU'LL JUST CLIMB OVER THE ROOF...
BUT THIS IS A 200 MPH BULLET TRAIN - AND WE'RE SUPPOSED TO HAVE RAIN THIS WEEKEND!
THAT'S THE GENIUS OF THIS PLAN... YOU'RE GOING TO BE WEARING THIS WET SUIT, WHICH WILL GIVE YOU THE AERODYNAMIC PROPERTIES NEEDED TO CLING TO THE TRAIN AND AVOID HYPOTHERMIA... ...HEY, WHAT ARE YOU DOING??
CONSING YOUR PLAN TO THE TRASH!!
JERK!!