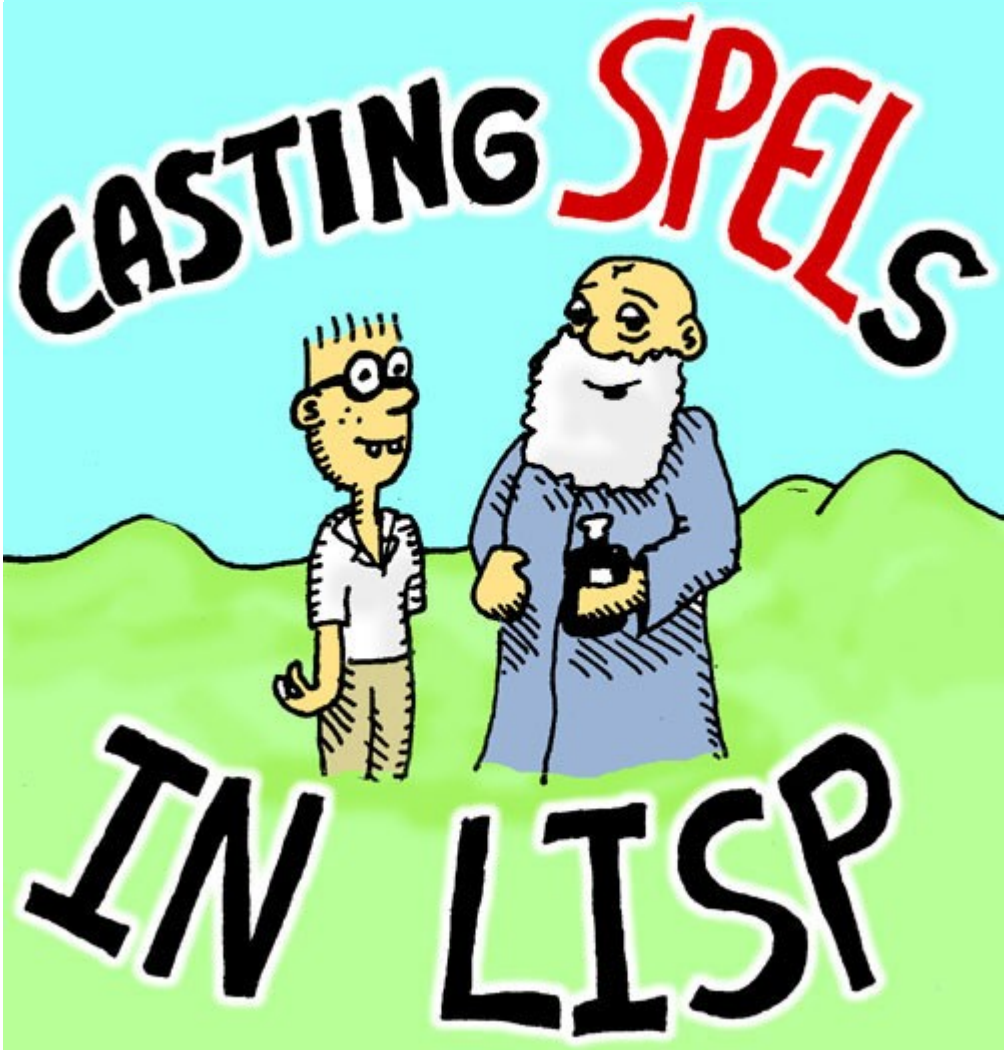
Ever wonder what makes Lisp so powerful?
Now you can find out for yourself-
*And you don't even have to install anything on your computer to do it!*



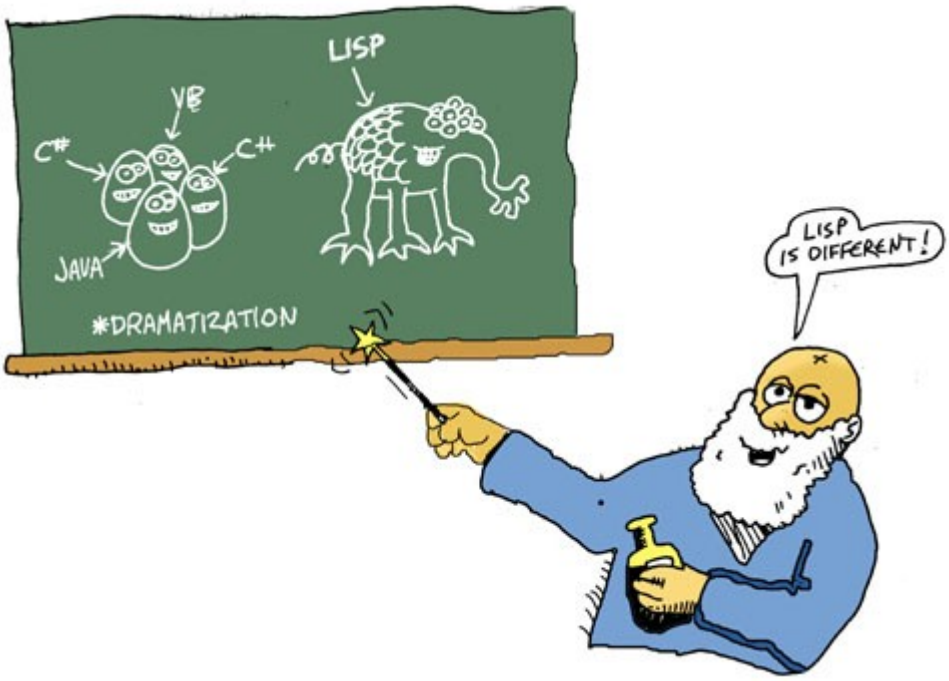

**A COMIC BOOK**
By Conrad Barski, M.D.
LISPERATI.COM

Emacs Lisp Version  Turkish Version  Ruby Version  Haskell Version?!

*1/17/08* ***Breaking News****- Watch out for my new Super Fantastic Expanded Lisp Comic Book/Text Book from No Starch Press later this year! Travel to the Land of Lisp- Soon in book form!!*

Anyone who has ever learned to program in Lisp will tell you it is very different from any other programming language. It is different in lots of surprising ways- This comic book will let

you find out how Lisp's unique design makes it so powerful!

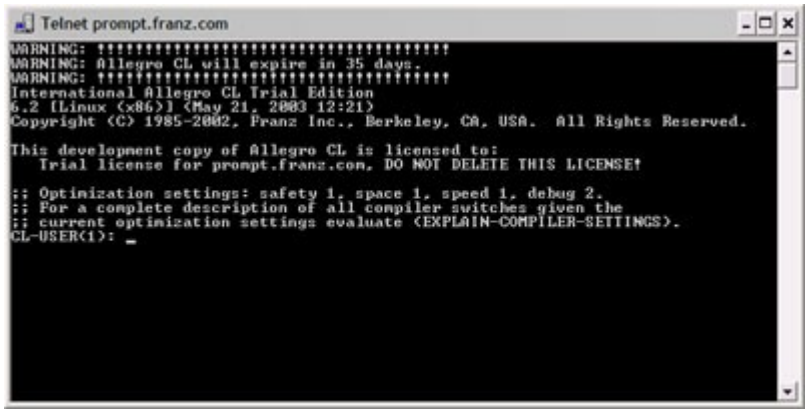



This tutorial has small bits of Lisp code written in

```
this font and color
```

...Simply copy these snippets into the prompt of a Lisp compiler and by the time the tutorial is done you will have **your own text adventure game!**

There are many great Lisp compilers out there that you can use, but the easiest to use for *this* tutorial is Franz's *Allegro Common Lisp*, a very powerful commercial Lisp that Franz, Inc. has graciously made available through a telnet environment **HERE** so you can try it out without installing any software on your PC. Simply click on the link and you browser will launch a telnet window that gives you everything you need!

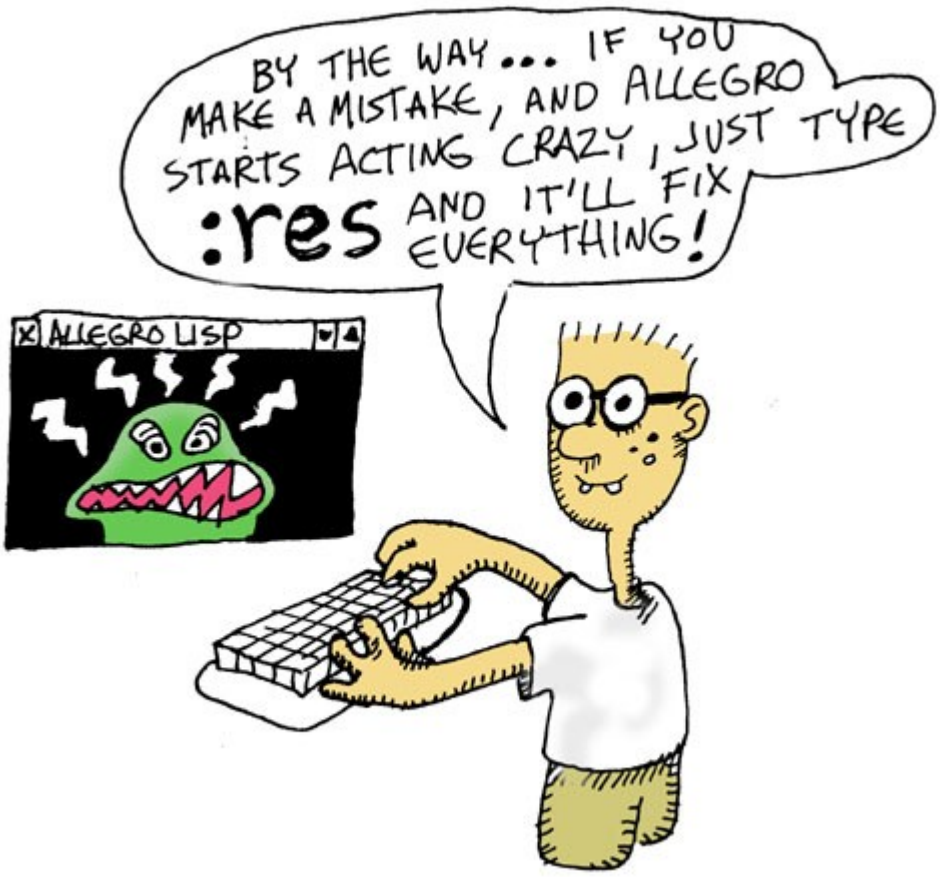



(Another Lisp I really like to use is the open-source CLISP, which you might want to download if you know you'll be spending more time with Lisp)

If you're using Allegro Common Lisp, please execute the following from the Lisp prompt:

```
(setf tpl:*print-length* nil)
```

...Just copy the text from your browser, then right-click on the telnet window and choose *Paste*. Don't worry too much about this line- it just prevents Allegro from chopping off messages printed by our text adventure game.



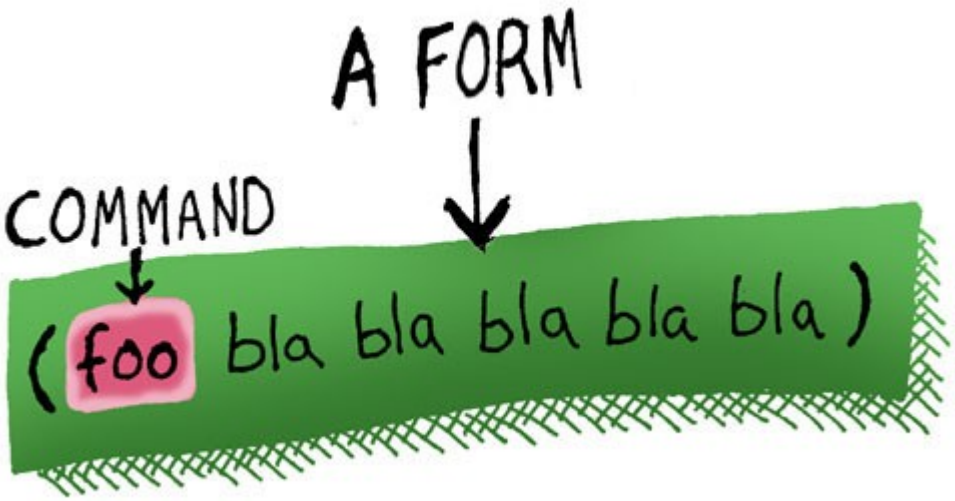

NEXT >>

## Syntax and Semantics

Every computer language has code that is made up of *syntax* and *semantics*. The *syntax* of a programming language is the basic skeleton your program has to follow so that the compiler knows *what's what* in your program, so it can tell what's a function, a variable, etc. The *semantics* of a program is the more "random" stuff, like the different commands you have available and what variables you're allowed to look at at any point in the program. The first thing that is special about Lisp is that it has the simplest syntax of any major programming language.

Basically, the Lisp syntax dictates that any text you give the Lisp compiler has to be arranged into lists, which can be nested into lists of lists or such as needed. The parenthesis mark the beginning and end of each list:


Additionally, the Lisp compiler uses two modes when it reads your code: A *Code Mode* and a *Data Mode*. When you're in *Data Mode*, you can put anything you want into your lists. However, the compiler first starts off reading your code in *Code Mode*- In *Code Mode*, your lists need to be a special type of list called a *form*:




A form is a list where the first symbol in the list has to be a special word that the compiler can understand- Usually the name of a function. In this case, the compiler will send the other items of the list to the function as parameters. When it reads the text for these parameters, it will usually assume that they are *also* in *Code Mode*, unless you tell it to flip into data mode.

[<< PREVIOUS]() [NEXT >>]()

## Defining the Data for our Game World

In order to learn some more about forms, let's create a some forms that create the data for our game world. First of all, our game is going to have some objects in it that the player can pick up and use- Let's define those objects:

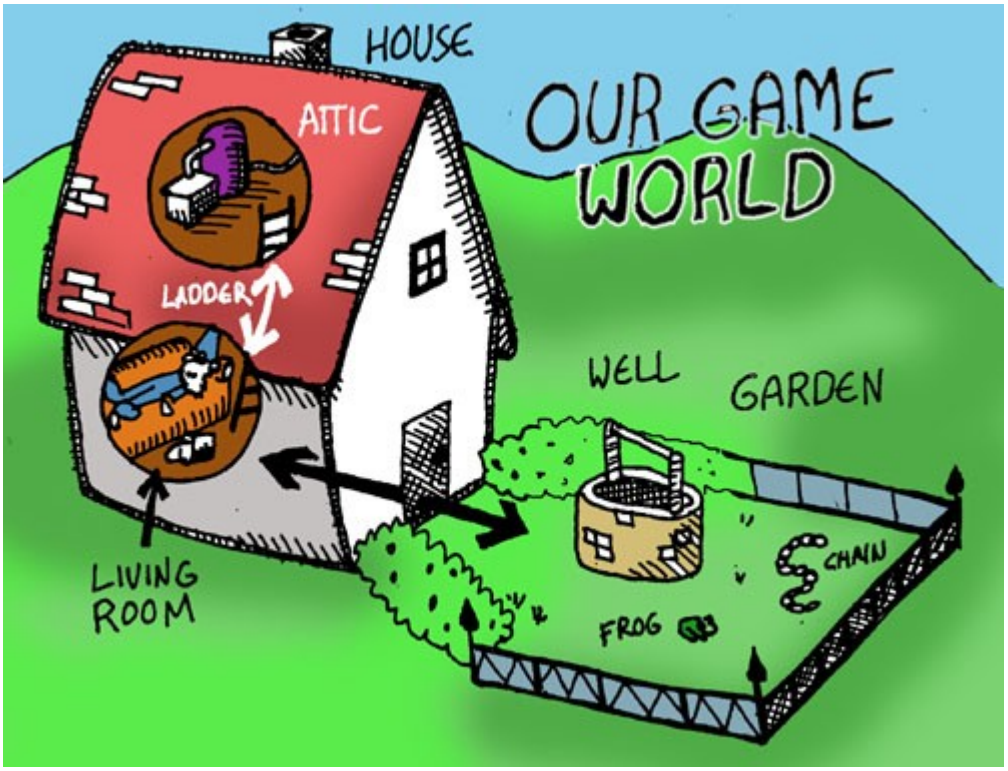
```
(setf *objects* '(whiskey-bottle bucket frog chain))
```

Ok, now let's dissect this line an see what it means: Since a Lisp compiler always starts reading things in *Code Mode* and expects a form, the first symbol, *setf*, must be a command. In this case, the command sets a variable to a value: The variable is *objects* (Lispers like to put stars around the names for global variables as a convention) and the value we are setting it to is a list of the four objects. Now, since the list is data (i.e. we don't want the compiler to try and call a function called *whiskey-bottle*) we need to "flip" the compiler into *Data Mode* when reading the list. The single quote in front of the list is the command that tells the compiler to flip:


You may be wondering why the command is called *setf*... I'm not sure why, actually, but you'll find that a lot of the commands in Lisp have quirky names, since Lisp is such an ancient language. This is actually somewhat useful, since the Lisp versions of common commands have all kind of elegant powers unique to Lisp and therefore the wacky names prevent confusing vocabulary when comparing command in Lisp to commands in other languages. The *setf* command, for instance, has all kinds of clever abilities that we won't even have a chance to touch on in this tutorial.

Now that we've defined some objects in our world, let's ramp

it up a step and define a map of the actual world itself. Here is a picture of what our world looks like:




In this simple game, there will only be three different locations: A house with a living room and an attic, along with a garden. Let's define a new variable, called *map* that describes this mini world:

```
(setf *map* '((living-room (you are in the living-room of a wizards house. there is a
wizard snoring loudly on the couch.)
                           (west door garden)
                           (upstairs stairway attic))
              (garden (you are in a beautiful garden. there is a well in front of you.)
                      (east door living-room))
              (attic (you are in the attic of the wizards house. there is a giant
welding torch in the corner.)
                     (downstairs stairway living-room))))
```

This map contains everything important that we'd like to know about our three locations: a unique name for the location (i.e. *house*, *garden*, and *attic*) a short description of what we can see from there (stored in its own list within the bigger list) , plus the *where and how* of each path into/out of that place. Notice how information-rich this one variable is and how it describes all we need to know but not a thing

more- Lispers love to create small, concise pieces of code that leave out any fat and are easy to understand just by looking at them.

Now that we have a map and a bunch of objects, it makes sense to create another variable that says where each of these object is on the map:

```
(setf *object-locations* '((whiskey-bottle living-room)
                           (bucket living-room)
                           (chain garden)
                           (frog garden)))
```

Here we have associated each object with a location. Lists such as this are, not surprisingly, called "association lists" in Lisp. An association list is simply a list of lists where the first item in each inside list is a "key" symbol that is associated with a bunch of other data. Our *map* variable was also an association list- The three keys in that case were *living-room*, *garden*, and *attic*.

Now that we have defined our world and the objects in the world, the only thing left to do is describe the location of the player of the game:

```
(setf *location* 'living-room)
```

Now let's begin making some game commands!

## Looking Around in our Game World

The first command we'd want to have is a command that tells us about the location we're standing in. So what would a function need to describe a location in a world? Well, it would need to know the location we want to describe and would need to be able to look at a map and find that location on the map. Here's our function, and it does exactly that:

```
(defun describe-location (location map)
  (second (assoc location map)))
```

The word *defun* means, as you'd expect, that we're defining a function. The name of the function is *describe-location* and it takes two parameters: a location and a map. Since these variables do not have stars around them, it means they are local and hence unrelated to the global **location** and **map** variables. Note that functions in Lisp are often more like functions in math than in other programming languages: Just like in math, this function doesn't print stuff for the user to read or pop up a message box: All it does is return a value as a result of the function that contains the description. Let's imagine our location is in the living-room (which, indeed, it is...).


To find the description for this, it first needs to look up the spot in the map that points to the living-room. The *assoc* command does this and then returns the data describing the living-room. Then the command *second* trims out the second item in that list, which is the description of the living-room (If you look at the *map* variable we had created, the snippet of text describing the living-room was the second item in the list that contained *all* the data about the living room...)

Now let's use our Lisp prompt to test our function- Again, like all the text in

```
this font and color
```

in the tutorial, paste the following text into your Lisp prompt:

```
(describe-location 'living-room *map*)

==> (YOU ARE IN THE LIVING-ROOM OF A WIZARD'S HOUSE. THERE IS A WIZARD SNORING LOUDLY
ON THE COUCH.)
```

Perfect! Just what we wanted... Notice how we put a quote in

front of the symbol *living-room*, since this symbol is just a piece of data naming the location (i.e. we want it read in *Data Mode*) , but how we didn't put a quote in front of the symbol *map*, since in this case we want the list compiler to hunt down the data stored in the *map* variable (i.e. we want the compiler to be in *Code Mode* and not just look at the word ***map*** as a chunk of raw data)

## The Functional Programming Style

You may have noticed that our describe-location function seems pretty awkward in several different ways. First of all, why are we passing in the variables for location and map as parameters, instead of just reading our global variables directly? The reason is that Lispers often like to write code in the *Functional Programming Style* (To be clear, this is completely unrelated in any way to the concept called "procedural programming" or "structural programming" that you might have learned about in high school...). In this style, the goal is to write functions that always follow the following rules:

**1.** You only read variables that are passed into the function or are created by the function (So you don't read any global variables)
**2.** You never change the value of a variable that has already been set (So no incrementing variables or other such foolishness)
**3.** You never interact with the outside world, besides returning a result value. (So no writing to files, no writing messages for the user)

You may be wondering if you can actually write any code like this that actually does anything useful, given these brutal restrictions... the answer is *yes*, once you get used to the style... Why would anyone bother following these rules? One very important reason: Writing code in this style gives your program *referential transparency*: This means that a given piece of code, called with the same parameters, always positively returns the same result and does exactly the same thing no matter when you call it- This can reduce programming errors and is believed to improve programmer productivity in many cases.

Of course, you'll always have some functions that are not *functional* in style or you couldn't communicate with the user or other parts of the outside world. Most of the functions later in this tutorial do not follow these rules.

Another problem with our *describe-location* function is that it doesn't tells us about the paths in and out of the location to other locations. Let's write a function that describes these paths:

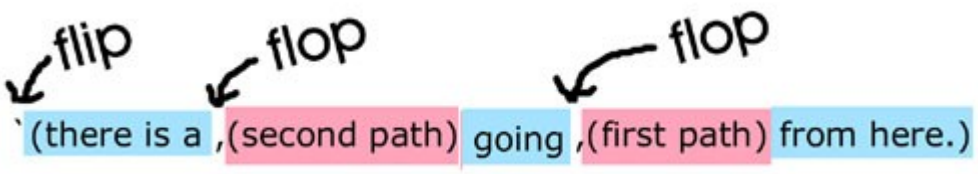
```
(defun describe-path (path)
  `(there is a ,(second path) going ,(first path) from here.))
```

Ok, now this function looks pretty strange: It almost looks more like a piece of data than a function. Let's try it out first and figures out *how* it does what it does later:

```
(describe-path '(west door garden))

==> (THERE IS A DOOR GOING WEST FROM HERE.)
```

So now it's clear: This function takes a list describing a path (just like we have inside our *map* variable) and makes a nice sentence out of it. Now when we look at the function again, we can see that the function "looks" a lot like the data it produces: It basically just splices the first and second item from the path into a declared sentence. How does it do this? It uses ***back-quoting***!

Remember that we've used a quote before to flip the compiler from *Code Mode* to *Data Mode*- Well, by using the the back-quote (the quote in the upper left corner of the keyboard) we can not only *flip*, but then also *flop* back into *Code Mode* by using a comma:




This "back-quoting" technique is a great feature in Lisp- it lets us write code that looks just like the data it creates. This happens frequently with code written in a functional style: By building functions that *look* like the data they create, we can make our code easier to understand and also build for longevity: As long as the data doesn't change, the functions will probably not need to be refactored or otherwise changed, since they mirror the data so closely. Imagine how you'd write a function like this in VB or C: You would probably chop the path into pieces, then append the text snippets and the pieces together again- A more haphazard process that "looks" totally different from the data that is created and probably less likely to have longevity.

Now we can describe a path, but a location in our game may

have more than one path, so let's create a function called *describe-path**s***:

```
(defun describe-paths (location map)
  (apply #'append (mapcar #'describe-path (cddr (assoc location map)))))
```

This function uses another common *functional programming* technique: The use of *Higher Order Functions*- This means that the *apply* and *mapcar* functions are taking other functions as parameters so that they can call them themselves- To pass a function, you need to put #' in front of the function name... The *cddr* command chops the first two item from the front of the list (so only the path data remains). *mapcar* simply applies another function to every object in the list, basically causing all paths to be changed into pretty descriptions by the *describe-path* function. The "apply #'append" just cleans out some parenthesis and isn't so important. Let's try this new function:

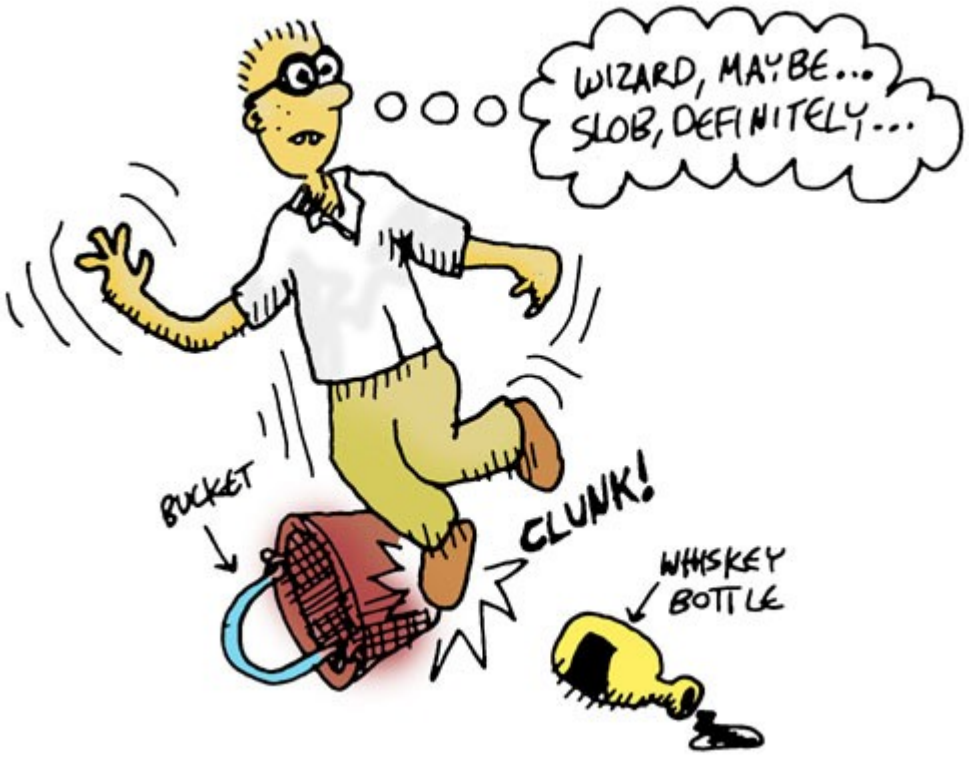
```
(describe-paths 'living-room *map*)

==> (THERE IS A DOOR GOING WEST FROM HERE. THERE IS A STAIRWAY GOING UPSTAIRS FROM
                                            HERE.)
```

Beautiful!

We still have one thing we need to describe: If there are any objects on the floor at the location we are standing in, we'll want to describe them as well. Let's first write a helper function that tells us wether an item is in a given place:

```
(defun is-at (obj loc obj-loc)
  (eq (second (assoc obj obj-loc)) loc))
```

...the *eq* function tells us if the symbol from the object location list is the current location.




Let's try this out:

```
(is-at 'whiskey-bottle 'living-room *object-locations*)

==> T
```

The symbol *t* (or any value other than *nil*) means that it's true that the whiskey-bottle is in living-room.

Now let's use this function to describe the floor:

```
(defun describe-floor (loc objs obj-loc)
  (apply #'append (mapcar (lambda (x)
                            `(you see a ,x on the floor.))
                          (remove-if-not (lambda (x)
                                           (is-at x loc obj-loc))
                                         objs))))
```

This function has a couple of new things: First of all, it has anonymous functions (*lambda* is just a fancy word for this). That first lambda form is just the same as defining a helper function *(defun blabla (x) `(you see a ,x on the floor.))* and then sending *#'blabla* to the mapcar function. The *remove-if-not* function removes any objects from the list that are not at the current location before passing the list on to mapcar to

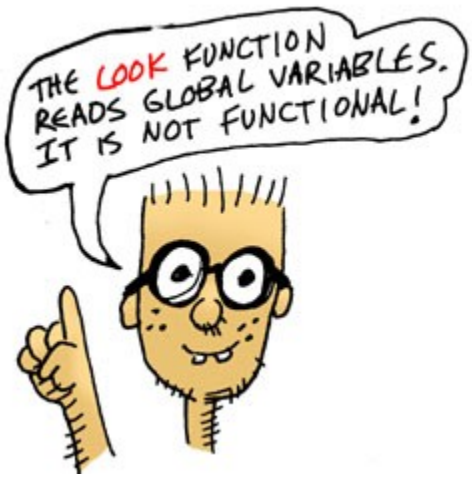
build pretty sentences. Let's try this new function:

```
(describe-floor 'living-room *objects* *object-locations*)
```

```
==> (YOU SEE A WHISKEY-BOTTLE ON THE FLOOR. YOU SEE A BUCKET ON THE FLOOR)
```

Now we can tie all these descriptor functions into a single, easy command called *LOOK* that uses the global variables (therefore this function is not in the *Functional Style*) to feed all the descriptor functions and describes **everything**:

```
(defun look ()
  (append (describe-location *location* *map*)
          (describe-paths *location* *map*)
          (describe-floor *location* *objects* *object-locations*)))
```




Let's try it:

```
(look)
```

```
==> (YOU ARE IN THE LIVING-ROOM OF A WIZARD'S HOUSE.
   THERE IS A WIZARD SNORING LOUDLY ON THE COUCH.
       THERE IS A DOOR GOING WEST FROM HERE.
    THERE IS A STAIRWAY GOING UPSTAIRS FROM HERE.
       YOU SEE A WHISKEY-BOTTLE ON THE FLOOR.
           YOU SEE A BUCKET ON THE FLOOR)
```

pretty cool!

[<< PREVIOUS]  [NEXT >>]

## Walking Around In Our World

Ok, now that we can see our world, let's write some code that lets us walk around in it. The function *walk-direction* (not in the *functional style*) takes a direction and lets us walk there:

```
(defun walk-direction (direction)
  (let ((next (assoc direction (cddr (assoc *location* *map*)))))
    (cond (next (setf *location* (third next)) (look))
          (t '(you cant go that way.)))))
```

The special command *let* allows us to create the local variable *next*, which we set to the path descriptor for the direction the player wants to walk in- *cdr* just chops the first item off of a list. If the user types in a bogus direction, *next* will be *nil*. The *cond* command is like a chain of if-then commands in Lisp: Each row in a cond has a value to check and an action to do. In this case, if the *next location* is not nil, it will setf the player's location to the third item in the path descriptor, which holds the symbol describing the new direction, then gives the user a look of the new place. If the *next* location is *nil*, it falls through to the next line and admonishes the user. Let's try it:

```
(walk-direction 'west)
```

```
==> (YOU ARE IN A BEAUTIFUL GARDEN.
  THERE IS A WELL IN FRONT OF YOU.
THERE IS A DOOR GOING EAST FROM HERE.
  YOU SEE A CHAIN ON THE FLOOR.
  YOU SEE A FROG ON THE FLOOR.)
```

Now, we were able to simplify our description functions by creating a *look* command that is easy for our player to type. Similarly, it would be nice to adjust the *walk-direction* command so that it doesn't have an annoying quote mark in the command that the player has to type in. But, as we have learned, when the compiler reads a form in *Code Mode*, it will read all its parameters in *Code Mode*, unless a quote tells it not to. Is there anything we can do to tell the compiler that *west* is just a piece of data without the quote?

[<< PREVIOUS]  [NEXT >>]

## Casting SPELs

Now we're going to learn an incredibly powerful feature of Lisp: Creating SPELs!

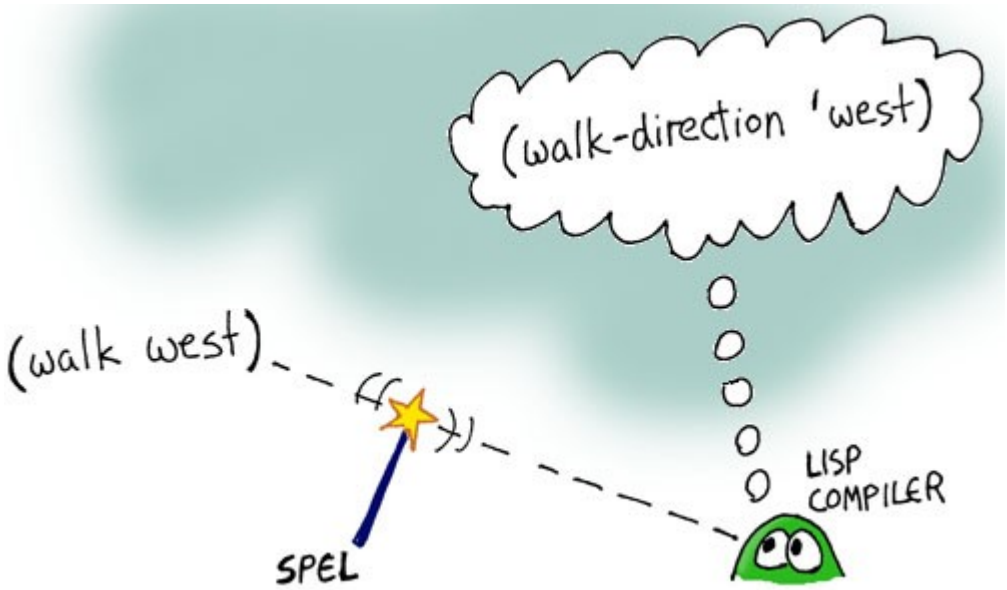
SPEL is short for "Semantic Program Enhancement Logic" and lets us create new behavior inside the world of our computer code that changes the Lisp language at a fundamental level in order to customize its behavior for our needs- It's the part of Lisp that looks most like magic. To enable SPELs, we first need to activate SPELs inside our Lisp compiler (Don't worry about what this line does- Advanced Lispers should click here.)

```
(defmacro defspel (&rest rest) `(defmacro ,@rest))
```

Ok, now that they're enabled, let's cast our first spell, called *walk*:

```
(defspel walk (direction)
  `(walk-direction ',direction))
```

What this code does is it tells the Lisp compiler that the word *walk* is not actually the word *walk* but the word *walk-direction* and that the word *direction* actually has a quote in front of it, even though we can't see it. Basically we can sneak in some special code inbetween our program and the compiler that changes our code into something else before it is compiled:




Notice how similar this function looks to the code we had written before for describe-path: In Lisp, not only do code and data look a lot identical, but code and special commands to the compiler (the SPELs) look identical as well- A very consistent and clean design! Let's try our new spell:

```
(walk east)
```

```
==> (YOU ARE IN THE LIVING ROOM OF A WIZARD'S HOUSE.
    THERE IS A WIZARD SNORING LOUDLY ON THE COUCH.
        THERE IS A DOOR GOING WEST FROM HERE.
    THERE IS A STAIRWAY GOING UPSTAIRS FROM HERE.
        YOU SEE A WHISKEY-BOTTLE ON THE FLOOR.
            YOU SEE A BUCKET ON THE FLOOR)
```

much better!

Now we'll create a command to pickup objects in our world:

```
(defun pickup-object (object)
  (cond ((is-at object *location* *object-locations*) (push (list object 'body) *object-locations*)
                                                      `(you are now carrying the ,object))
        (t '(you cannot get that.))))
```

This function checks to see if the object is indeed on the floor of the current location- If it is, it *pushes* the new location (the player's body) onto the list (*pushing* means to add a new item to the list, in a way that the *assoc* command sees and therefore hides the previous location) and returns a sentence letting us know wether it succeeded.

Now let's cast another SPEL to make the command easier to use:

```
(defspel pickup (object)
  `(pickup-object ',object))
```

Now let's try our new SPEL:

```
(pickup whiskey-bottle)

==> (YOU ARE NOW CARRYING THE WHISKEY-BOTTLE)
```

Now let's add a couple more useful commands- First, a command that lets us see our current inventory of items we're carrying:

```
(defun inventory ()
  (remove-if-not (lambda (x)
                   (is-at x 'body *object-locations*))
                 *objects*))
```

Now a function that tells us if he have a certain object on us:

```
(defun have (object)
  (member object (inventory)))
```

[<< PREVIOUS]  [NEXT >>]

## Creating Special Actions in Our Game

We have only one more thing to do now and our game will be complete: Add some special actions to the game that the player has to do to win in the game. The first command will let the player weld the chain to the bucket in the attic:

```
(setf *chain-welded* nil)
```

```
(defun weld (subject object)
  (cond ((and (eq *location* 'attic)
              (eq subject 'chain)
              (eq object 'bucket)
              (have 'chain)
              (have 'bucket)
              (not *chain-welded*))
         (setf *chain-welded* 't)
         '(the chain is now securely welded to the bucket.))
        (t '(you cannot weld like that.))))
```

So first we created a new global variable that lets us tell whether we've done this action already. Next, we create a weld function that makes sure all the right conditions are in place for welding and lets us weld.


Let's try our new command:

```
(weld 'chain 'bucket)
```

```
==> (YOU CANNOT WELD LIKE THAT.)
```

Oops... we're don't have a bucket or chain, do we? ...and there's no welding machine around... oh well...

Now let's create a command for dunking the chain and bucket in the well:

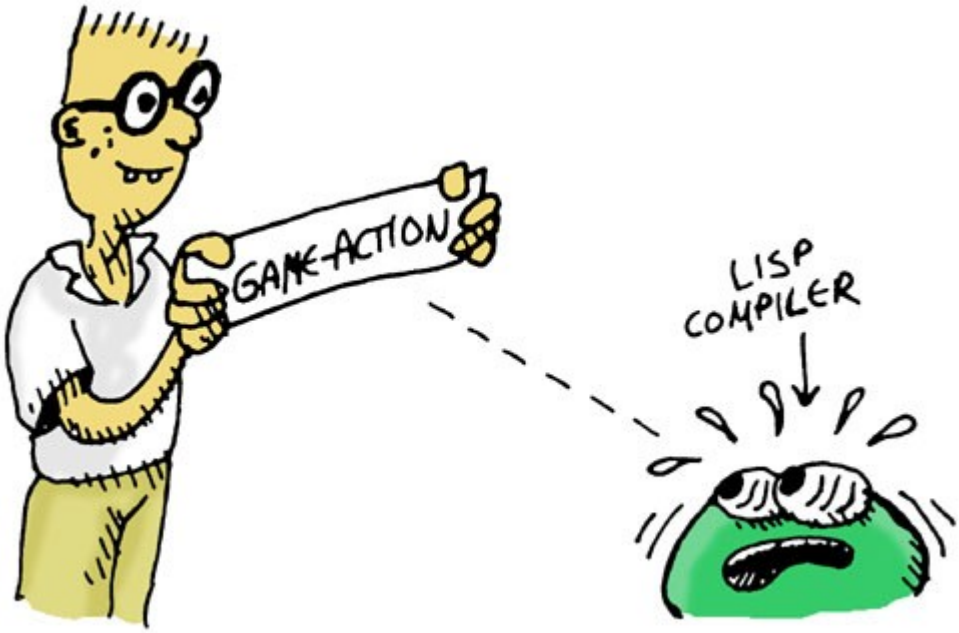
```
(setf *bucket-filled* nil)

(defun dunk (subject object)
  (cond ((and (eq *location* 'garden)
              (eq subject 'bucket)
              (eq object 'well)
              (have 'bucket)
              *chain-welded*)
         (setf *bucket-filled* 't) '(the bucket is now full of water))
        (t '(you cannot dunk like that.))))
```

Now if you paid attention, you probably noticed that this command looks *a lot* like the *weld* command... Both commands need to check the location, subject, and object- But there's enough making them different enough so that we can't combine the similarities into a single function. Too bad...

...but since this is Lisp, we can do more than just write functions, we can cast SPELs! Let's create the following SPEL:

```
(defspel game-action (command subj obj place &rest rest)
  `(defspel ,command (subject object)
     `(cond ((and (eq *location* ',',place)
                  (eq ',subject ',',subj)
                  (eq ',object ',',obj)
                  (have ',',subj))
             ,@',rest)
            (t '(i cant ,',command like that.)))))
```

Notice how ridiculously complex this SPEL is- It has more weird quotes, backquotes, commas and other weird symbols than you can shake a list at. More than that it is a SPEL that actually cast ANOTHER SPEL! Even experienced Lisp programmers would have to put some thought into create a monstrosity like this (and in fact they would consider this SPEL to be inelegant and would go through some extra esoteric steps to make it better-behaved that we won't worry about here...)




The point of this SPEL is to show you just how sophisticated and clever you can get with these SPELs. Also, the ugliness doesn't really matter much if we only have to write it once and then can use it to make hundreds of commands for a bigger adventure game.

Let's use our new SPEL to replace our ugly *weld* command:

```
(game-action weld chain bucket attic
             (cond ((and (have 'bucket) (setf *chain-welded* 't))
                    '(the chain is now securely welded to the bucket.))
                   (t '(you do not have a bucket.))))
```

Look at how much easier it is to understand this command- The game-action SPEL lets us write exactly what we want to say without a lot of fat- It's almost like we've created our own computer language just for creating game commands. Creating your own pseudo-language with SPELs is called *Domain Specific Language Programming*, a very powerful way to program very quickly and elegantly.

```
(weld chain bucket)

==> (YOU DO NOT HAVE THE CHAIN.)
```

...we still aren't in the right situation to do any welding, but the command is doing its job!


Next, let's rewrite the *dunk* command as well:

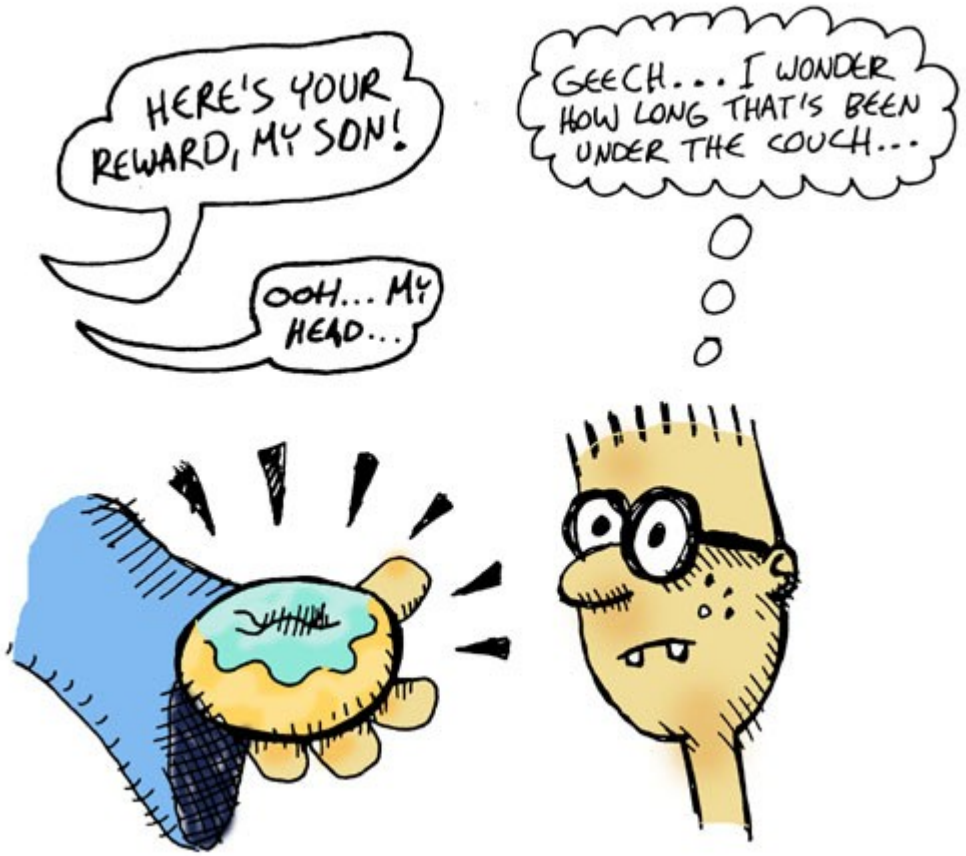
```
(game-action dunk bucket well garden
             (cond (*chain-welded* (setf *bucket-filled* 't) '(the bucket is now full of water))
                   (t '(the water level is too low to reach.))))
```

Notice how the *weld* command had to check whether we *have* the subject, but that the dunk command skips that step- Our new game-action SPEL makes the code easy to write and understand.


And now our last code for splashing water on the wizard:

```
(game-action splash bucket wizard living-room
             (cond ((not *bucket-filled*) '(the bucket has nothing in it.))
```

```
((have 'frog) '(the wizard awakens and sees that you stole his frog.
                he is so upset he banishes you to the
                netherworlds- you lose! the end.))
(t '(the wizard awakens from his slumber and greets you warmly.
     he hands you the magic low-carb donut- you win! the end.))))
```




**You have now written a complete text adventure game!**

Click HERE for a complete walkthrough of the game,
Click HERE for a copy of the source code you can copy & paste into your Lisp prompt in a single step.

In order to make this tutorial as simple as possible, many details about how Lisp works have been glossed or fudged over, so let's look at what those details are...


<< PREVIOUS NEXT >>


**Addendum**

Ok, now lets talk about some issues that were glossed over in this tutorial...

First of all, Lisp code is often interpreted as well as compiled, often right within the same program... Because of this, the Lisp environments are not usually referred to as compilers but just as *implementations*.

(although even interpreted Lisp code is usually first compiled into byte code, but that's another story...)

Additionally, there are other great Lisp implementations that are worth mentioning- There is a detailed list available that Rainer Joswig and Bill Clementson have put together.

One major *cheat* that we made in this tutorial is that we wrote our game sentences using symbols

```
'(this is not how Lispers usually write text)

"Lispers write text using double quotes"
```

Symbols have a special meaning in Lisp and are used to store unique names of functions, variables, and other things. Because of this, Lisp treats symbols in special ways that are awkward for text messages (such as making them ALL CAPS...). Using strings instead of symbols allows text we work with to not be affected by any such quirks, but requires more esoteric commands for manipulating text. Also, working with strings is not so relevant to teaching the far more important symbol manipulation commands in Lisp.

One more thing we left out of the code is the *defparameter* command for creating global variables- Instead, we just used setf to declare variables (which works, but is considered bad style...)

Another simplification is that association lists (also called alists) are usually written using a *dotted list*, because it is slighlty more efficient and elegant to an experienced Lisper. This is confusing to beginners, however, because it requires an understanding of Cons Cells, which you can read about here.

Another glossed over issue is that SPELs are more commonly referred to as "Lisp true macros" and are created with the *defmacro* command, which is very confusing for teaching purposes. Read the following short essay as to why I think this name distinction is beneficial. And finally, there are ugly name collisions that can happen when a SPEL is written in the style of the *game-action* SPEL. If you read more

advanced lisp materials this will be explained in greater detail.

**Q.** What should I read next to expand my knowledge of lisp?

**A.**

There are many great (and some downloadable) Lisp books available at the cliki website.
If you're interested in the most intense theoretical text, I would recommend the free ebook version of On Lisp by Paul Graham. The other books he has written and the essays on his website are also fantastic.

If you're interested in a more pragmatic tack, many Lispers are currently excited about the book Practical Common Lisp by Peter Seibel. Some chapters of this book are available on-line.