Objective Lisp

Todd Newton

March 11, 2018

1 Introduction

When first detailing my goals for this project, I had in mind a program that could parse and compile an object oriented language that simulated dice rolling. Once I set about implementing these ideas, I realized I might have bitten off more than I could chew. A language parser is one thing, but also designing the compiler and the language itself was just too much work for amount of time I had. Not all was lost however. In my planning for compiling the dice language into lisp code, I had laid the foundation for a lisp dialect that could more accurately simulate object oriented language. I reworked my goals, and Objective Lisp became my new target. My goals for this project were to:

1. Create an Object Oriented Domain Specific Language in Lisp by:
2. Implementing macros that convert code into data structures specifying classes and,
3. Implementing functions which enable the construction of objects and the manipulation of the data represented by those objects.
4. Do so without relying on CLOS.

The final goal was to guarantee that the Language I created wasnt just an obfuscation of Common Lisp, and to ensure I learned something while overcoming the challenges I encountered.

2 Implementation

Classes in Objective Lisp are stored in memory as association lists to take advantage of the built in list manipulation functionality of Lisp. All classes share the same basic structure in memory. This structure is created by the `defineclass` macro and is held in a global list called `*class-list*`, which contains a "prototype" of every class that has been defined. Objective Lisp unwraps the code within the `defineclass` block and transforms it into the structure seen below.

```
(*class-name*
  (variables . *list of class variables*) ;if any
  (*variable name* . *variable value*)
  (*variable name* . *variable value*)    ;etc
  ...
  (*method name* . #<Lambda Function>)
  (*method name* . #<Lambda Function>)     ;etc
  ...
)
```

Additional macros used in Objective Lisp are the `vars`, `definemethod`, and `this` macros. `vars` is used to define class variables as well as their initial values. It uses the `pair` function detailed in Conrad Barski's *Land of Lisp* to make an entry in the class structure for each variable. It also creates a list of each variable declared and places it in the "variables" slot of the class structure. The `definemethod` macro is used to specify methods belonging to the enclosing class. It does so by calling a function called `fn`, which build and compiles a lambda function with the arguments and body passed to `definemethod`. In addition, it inserts an argument at the head of the argument list called `self`. "Self" is a special word in Objective Lisp, used to refer to the object a method was called with, if a method needs to refer to another method that object has. Every time a method is called, it is passed a reference to the object that the method was called on, similar to how classes work in the Python programming language. The final macro defined in Objective Lisp is the `this` macro. The `this` macro is used to refer to variables contained in the calling object.

Objects in Objective Lisp are created using the `new` function. This creates an associa-

tion list headed by the name of the object's class-type and followed by a `(*name* . *value*)` pair for each variable contained. To cut down on redundancy in memory, each object does not have a copy of its class functions. Instead, each object refers to the global `*class-list*` to determine the methods it has. The function for calling methods of objects is the `call` function, which retrieves the relevant function from the `*class-list*` and applies it to the calling object and any other arguments. Just like in any other Object Oriented language, objects created with `new` are unique; that is, they are not `eq` with any other object, and modifications to one instance do not affect another instance. Finally, because objects are just association lists, if an exact copy (with the same variable values) of an object is needed it can be copied with `copy-alist`. This will effectively create a new object with its variables initialized to the same values. This is abstracted in Objective Lisp with the `clone` function, which is more intuitive to users with an Object Oriented background.

3 How to use

The syntax of class definitions in Objective Lisp should look familiar to someone who knows other Object Oriented languages like Java or C++. Classes are defined within a `defineclass` expression. Any variables used within the class are defined at the top in a `vars` expression, which consists of a variable's name followed by its initial value. Each variable must be declared in the same `vars` expression, and each one must have an initial value. Normally, class variables are preceded by an asterisk to avoid using them as local variables accidentally. Methods of the class are then defined within `definemethod` expressions. Below is an example of a class with a single class variable and a getter and setter.

```
(defineclass example

  (vars *a nil)

  (definemethod get-a ()
    (this *a))
```

```
  (definemethod set-a (a)
    (setf (this *a) a)))
```

The methods `get-a` and `set-a` both use the form `(this *a)`. This is how a class refers to the class variables belonging to it. Class variables do not have to be preceded by an asterisk, but it is useful for avoiding accidentally shadowing the variable in a method.

Creating an instance of an object is done with `(new '<name of class>)` e.g. `(new 'example)`. This will return a new instance of the named class, which can then be passed to another function or bound with `setf` or `setq`. Calling a class method is done with `(call '<method name> <object> [<args>])`. Example:

```
> (setf e (new 'example))
(example (*a . nil))
> (call 'get-a e)
nil
> (call 'set-a e 1)
1
> (call 'get-a e)
1
```

Class methods can refer to themselves too. Here is an example of a class that accumulates numbers.

```
(defineclass counter
  (vars *i 0)

  (definemethod set-counter (i)
    (setf (this *i) i))

  (definemethod count-by (num)
    (call 'set-counter self (+ (this *i) num))))
```

`Self` is a special word in Objective Lisp. It is used in a method declaration to refer to the object that the method was called on. This is different from `this` in that `this` is used to refer to the object's variables, while `self` refers to the object itself. This can be used to call other methods of that object. Another thing worth note is that multiple methods can have the same name as long as they belong to different classes, because `call` only searches the class in question for the proper method.

4 Summary

This project has taught me several things about symbolic languages, and how lisp handles data. The biggest hurdle for me was setting up the macros to correctly expand into the data structure I wanted. I became very familiar with the concept that macros don't evaluate their arguments. Originally, I implemented the `call` function as a macro. It worked perfectly well as long as the arguments I passed were self-representing, like numbers. I only found a problem when I was working with a `node` class which worked as a linked list (redundant in lisp, but indispensable in other programming languages). I had an object called node1 and an object called node2, and when I did `(call 'append node1 node2)` I got back `(node (*val) (*next node2))`. It had appended the symbol `node2` rather than the value of node2. I now have a greater understanding of how macros function in Lisp (or rather, how they aren't functions). Another recurring issue was with the lambda functions stored in the class definition. Lisp wasn't treating them like functions, it just thought they were a list with a symbol `lambda` followed by two lists. The `fn` function solution was inspired by an academic paper about function composition in lisp that I found while trying to solve this problem. The paper had a macro `fn` which literally built lambda functions piece by piece to create higher-order composed functions (e.g. $(f \circ g)(n)$ is equivalent to $f(g(n))$). This made me realize that the lambda function had to be evaluated before being added to the list to keep its meaning as a function.

I believe this project was a good example of the "code as data" philosophy of Lisp. The structure of the class definition ended up being almost identical to the way the class is stored by the program. Unfortunately (because of the amount of debugging I had to do) I didn't have a lot of time to experiment with new ways this can be used. I wanted to experiment with the idea of mutable classes, where the exact definition of a class can be changed at run time. I also wanted to have a default constructor method called `*init*` which would be called by the `new` function before returning the class, but I couldn't get it to implement properly in time so I scrapped it. But overall, I think I've learned quite a bit about Lisp and functional programming from the problems I had to overcome to realize this project.