The Littlest Macro

Harry R. Schwartz

February 13, 2015

Functions are extraordinarily useful, but there are certain things they can't do. For example, when a function is evaluated in a language with eager evaluation (that is, most popular languages), every argument to the function is recursively evaluated before the function runs. Generally this is what we'd want, but in certain cases we want to delay the evaluation of the function's arguments until later. This isn't possible with a regular function.

For example, some languages (Ruby, for instance) feature an `unless` operator. This operator works a lot like `if`. It takes a condition, a consequent, and an alternative, just like an `if`, but it negates the condition before picking a branch.

Suppose we want to implement an `unless` operator in Emacs Lisp. Our first inclination might be to implement this as a function:

```
(defun unless (condition consequent alternative)
  (if (not condition) consequent alternative))
```

This seems reasonable, right? Let's try running it with `C-x C-e`:

```
(unless t
  (print "evaluating the consequent!")
  (print "evaluating the alternative!"))
```

```
"evaluating the consequent!"
"evaluating the alternative!"
"evaluating the alternative!"
```

Oops! We evaluated both the consequent *and* the alternative.

The second alternative is the return value of the whole `unless` expression; it's harmless.

That's definitely not what we wanted, and it happened because the arguments to the function were evaluated before the conditional was called.

Let's rewrite this as a macro:

```
(defmacro unless (condition consequent alternative)
  `(if (not ,condition)
       ,consequent
       ,alternative))
```

We can use `macroexpand` to display the code that our macro generates:

```
(macroexpand '(unless t (+ 1 2) (- 1 2)))
```

Which yields:

```
(if (not t) (+ 1 2) (- 1 2))
```

Now that we've reimplemented `unless` as a macro, let's try running it again:

```
(unless t
  (print "evaluating the consequent!")
  (print "evaluating the alternative!"))
```

```
"evaluating the alternative!"
"evaluating the alternative!"
```

Again, that second alternative is the return value of the whole `unless` expression. But, woo! We've effectively added a whole new construct to Emacs Lisp. Users can now use `unless` in the same way that they could use `if`.

Because they let you add new constructs to the language, macros are extremely powerful. In fact, most of the features of Emacs lisp *are* implemented as macros. Just `macro-expand` them and see for yourself!

```
(macroexpand '(when t foo bar))
```

```
(if t (progn foo bar))
```

Macros allow you to arbitrarily manipulate a program's abstract syntax tree. This manipulation is usually centered around the idea of delaying a chunk of computation. This delay is done by treating a statement as a data structure, which is easy because Lisp is homoiconic. Some non-Lisps are homoiconic, too: Julia and Elixir are among them.

This isn't something that you'll often need in your everyday programming, but once it's part of your arsenal it becomes pretty useful. By manipulating the syntax tree directly you're effectively extending the language, and that's an awfully powerful idea.