Veit's Blog

Lets Build a Quasiquoter

01/25/2021

For a very long time, Carp didn't have a mechanism for quasiquotation. As a macro writer, this frustrated me to no end, until, finally, I broke down and wrote an implementation of quasiquotation, in macros. It received a lot of helpful feedback from the community, and everyone was very excited when it got merged!

In this blog post I want to both explain what quasiquotation is and how it is useful, and build a mechanism for quasiquotation with you. While I was initially afraid it might be a lot of work to build this feature, I actually was able to get to a working prototype within a few hours. If we work together, it will hopefully take us even less time!

Let's get this party started!

Why quasi? And what is quoting?

If you've ever talked to anyone who likes Lisp, you've probably heard that its killer feature is that "code is data". If your eyes—or ears—start to glaze over when someone tries to explain the concept, you're not alone. Still, let me try and make a short introduction:

```
(defn adder [x y] (+ x y))
```

Fig. 1: An addition function.

In Figure 1, we build a function `adder` that takes two arguments, `x` and `y`, and adds them together using `+`. But this is also a list! Its first two elements are the symbols `defn` and `adder`, its third the argument array, and the last element is the body, which is in itself a list. If we wanted to treat this list as a data structure rather than as code, all we'd have to do is prefix it with a quote (`'`).

```
'(defn adder [x y] (+ x y))
```

Fig. 2: Not a function anymore.

We can traverse that list, take elements, rearrange it, all the things we usually do with our data! This is exciting, because it means that writing a macro in Lisp is just building a list like that and telling the environment to evaluate it. We can programmatically generate code!

To do that, however, we often need to make some sections of the structrue fixed and others flexible. Let's think of a macro that creates an alias for a function without arguments, and you will see what I mean:

```
(defmacro alias [new-name original]
  ; we want to emit (defn new-name [] (or
  (list 'defn new-name [] (list original)

(defn x [] 1)
(alias y x)
(y) ; => 1
```

Fig. 3: Aliasing functions.

While the code in Figure 3 isn't terrible, all the calls to `list` get in the way of understanding the final structure of the code we emit. It doesn't at all look like what we want to produce, and it's very verbose. Still, this is as good as it gets if we use `quote`, because it's an all-or-nothing situation: either this is code, or it's data.

This is where quasiquotation comes in. Rather than explaining what it does, let me just show you, and hopefully it's somewhat clear:

```
(defmacro alias [new-name original]
  ; we want to emit (defn new-name [] (or
  `(defn %new-name [] (%original)))

(defn x [] 1)
(alias y x)
(y) ; => 1
```

Fig. 4: Aliasing functions, quasiquote style.

In short, quasiquotation uses `` ` `` as a glyph, and you can intersperse unquoted values using `%`. There is also another operator, `unquote-splicing` or `%@`, which will unfold a list of expressions and "splice" it into the parent list. Again, let's look at an example:

```
(defdynamic x '(1 2))

; using unquote is not quite enough
`(+ %x) ; => (+ (1 2))

; unquote-splicing works!
`(+ %@x) ; => (+ 1 2)
```

Fig. 5: `unquote-splicing` in action.

This should hopefully make quotation and quasiquotation quite clear. You can refer to the Carp documentation or the API docs to see an alternative explanation, although as of now they've all been written by yours truly.

We are almost ready to try and implement this ourselves, but one piece is missing: how do we implement these weird glyphs? They don't seem to work like regular functions!

All the glyphs we looked at are "reader macros". This means that all Carp does is rewrite the expressions to `quasiquote`, `unquote`, and `unquote-splicing` forms. What would this look like? Let's look at the code from Figure 5 again:

```
`(+ %x)
; expands to (quasiquote (+ (unquote x)))

`(+ %@x)
; expands to (quasiquote (+ (unquote-spli
```

Fig. 6: The reader macros.

That means we can implement this in regular Carp after all, by implementing normal macros and functions.

Let's get to work then!

Implementing `quasiquote`

Before we start, I'd like to point out that this is not a tutorial for writing macros. While it's certainly possible to follow along without really knowing how macros work, it might be a little much, and I don't want to frustrate you. There are many good resources on macros, I personally like Colin Jones' book on Clojure macros. If you are a little more advanced and want some more mind-bending stuff, I'm quite proud of my little series of blog posts on language features implemented as Scheme macros.

Now let's write some code, shall we?

```
(defmacro unquote [form]
  (macro-error "unquotes need to be wrapp

(defmacro unquote-splicing [form]
  (macro-error "unquotes need to be wrapp
```

Fig. 7: Baby's first error.

This might be a little anti-climactic, but we need to ensure that `unquote` and `unquote-splicing` forms are always wrapped inside `quasiquote`s. The two macros above will be called when lone unquotes are found, and report this as an error.

The implementation of `quasiquote` is where it's at, so let's write a little skeleton

```
(defndynamic quasiquote- [form]
  ; magic happens here
)

(defmacro quasiquote [form]
 (quasiquote- form))
```

Fig. 8: A skeleton for `quasiquote`.

In the macro, we will actually just call a dynamic function to do all the work. I find that this is good practice unless a macro is a simple rewrite, since functions are much more reusable, and it's easier to think about recursive functions than about recursive macros[1].

So, what do we have to do? `quasiquote` will need to traverse its body to see what needs to be unquoted and what can remain the same.

```
(defndynamic quasiquote- [form]
  (cond
    (and (list? form) (> (length form) 0)
      (quasiquote-list form)
    (array? form)
      (collect-into (map quasiquote- form
    (list 'quote form)))
```

Fig. 8: Quasiquoting, by case.

Alright, let's look at the cases! If it's a non-empty list, we call `quasiquote-list` on it, a function we haven't defined yet, but that will presumably take care of the list for us. If it's an array, we just call `quasiquote-` on each of the elements and put the result in an array again, and otherwise we quote the argument, since it's not inside an unquoted form. So far, so good!

Our workhorse `quasiquote-list` will have to deal with a little more at once. Let's look at it!

```
(defndynamic quasiquote-list [form]
  (let [app (car form)]
   (cond
     (= app 'quasiquote) form
```

```
    (and (= app 'unquote) (= (length for
      (cadr form)
    (= app 'unquote)
      (macro-error "unquote takes exactl

    (map quasiquote- form))))
```

Fig. 9: Quasiquoting a list.

I've omitted dealing with `unquote-splicing` for now, since it will complicate matters a little bit. For now, we just look at the first element of the list. If it is another `quasiquote` form, we leave it as is. The next time the macro expander runs, it will expand this nested quasiquotation, but it's not our business. If it's an `unquote` form and it only has one element, we just return that element using `cadr`. Otherwise, the `unquote` form is malformed and we return an error. If we haven't encountered a special form, we just run `quasiquote-` on all the forms in `form` to ensure that any nested lists also get expanded.

This works, and is quite elegant and simple. Sadly, this will break down a little once we have to deal with `unquote-splicing`. This is somewhat obvious when you think about the current implementation: we just walk the form recursively, either leaving it untouched or replacing elements, but the structure is unchanged. This is different with splices: we need to change the enclosing structure of the splice.

Where is the crux of the problem then? It's the last line of Figure 9, where we use `map`. We will have to change this to a more involved traversal. In my implementation of quasiquotation, I opted for `reduce`. Let me show you the magic incantation, and then we stick it back into `quasiquote-list`.

```
(reduce
 (fn [acc elem]
   (if (and* (list? elem)
             (= (length elem) 2)
             (= (car elem) 'unquote-splic
     (list 'append acc (cadr elem))
     (list 'cons-last (quasiquote- elem)
 '()
 form))))
```

Fig. 10: Magic.

This is a little much, and I apologize. But all we're really doing is going over a list, and for each element we check for any sublists that are two elements long and start with `unquote-splicing`. If they do, we use `append` to stick them to our result list, which will flatten it (this is the crux!). In all other cases, we essentially resort to what we did with `map`: call `quasiquote-` on the list element and stick it at the end of the list.

All we have to do now is add two error cases for `unquote-splicing` in our list handling function, and stick our magic bit at the end:

```
(defndynamic quasiquote-list [form]
  (let [app (car form)]
   (cond
     (= app 'quasiquote) form
```

```
      (and (= app 'unquote) (= (length for
        (cadr form)
      (= app 'unquote)
        (macro-error "unquote takes exactl

      (and (= app 'unquote-splicing)
           (= (length form) 2))
        (macro-error "unquote-splicing nee
      (= app 'unquote-splicing)
        (macro-error "unquote-splicing tak

      (reduce
        (fn [acc elem]
          (if (and* (list? elem)
                    (= (length elem) 2)
                    (= (car elem) 'unquote
            (list 'append acc (cadr elem))
            (list 'cons-last (quasiquote-
        '()
        form))))
```

Fig. 11: Quasiquoting a list in full.

The two error cases we're dealing with is when we find a lone `unquote-splicing`, in which we report that it needs to be part of a list, and a malformed one, similar to how dealt with `unquote` above.

And that's it! We can `quasiquote` and `unquote` all we like! Pretty excting, huh?

Fin

While this might have been a little dense, I still find it instructive to look at something like this, something that you might have no idea how to implement, and just go through it. It's a fantastic way to learn, and I use it all the time[2]!

I hope this has been as helpful to you as I hoped, and I'll see you soon!

Footnotes

1. If you like recursive macros, my series on Scheme macros is for you!

2. I most recently used it to understand how one would implement something like Datalog (spoiler: it's almost the same as miniKanren), and went through the Racket implementation. It's a fantastically simple little implementation of Datalog, and it took away all the fear and doubt that I had. In fact, just a week later I gave a one hour wealkthrough of the implementation to a paper reading group that got pretty good feedback!

Want to go back to the list of posts?

💻 Want to become a better programmer? Join the Recurse Center!