CS1102: Macros and Recursion

Kathi Fisler, WPI

September 28, 2007

This lecture looks at several more macro examples. It aims to show you when you can use recursion safely with macros and when you can't.

1 Multi-Argument Or

We talked about how to define **or** as a macro, and why it needed to be defined as a macro instead of as a function. As a reminder, we developed the following macro for or (calling it myor to avoid a name clash within Scheme):

```
(define-syntax myor
  (syntax-rules ()
    [(myor e1 e2) (cond [e1 true]
                        [else e2])]))
```

This macro limited or to two arguments. The real or macro in Scheme accepts arbitrary numbers of arguments. Here are some examples:

```
> (or (= 3 2))
false

> (or (= 3 4) (> 6 9) (< 2 7))
true
```

Let's change **myor** to accept an arbitrary number of arguments. How might we write that? First, we need to change the input pattern to expect an arbitrary number of arguments. Then, if *e1* isn't true, check **myor** on the rest of the inputs:

```
(define-syntax myor
  (syntax-rules ()
    [(myor e1 ...)
     (cond [e1 true]
           [else (myor ...)])]))
```

Scheme rejects this with an error saying "syntax: missing ellipses with pattern variable in template in: e1". The problem is in the else clause: you can't have ... without the variable that goes with it being nearby. We don't want to insert *e1* into the **myor** in the else clause, so we need to introduce a second pattern variable:

```
(define-syntax myor
  (syntax-rules ()
    [(myor e1 e2 ...)
     (cond [e1 true]
           [else (myor e2 ...)])]))
```

Trying this on input (**myor** (= 2 2) (> 3 4)) yields an error "myor: bad syntax in: (myor)". What happened? Let's look at the sequence of expansions that Scheme performs to expand uses of **myor** according to the pattern:

(**myor** (= 2 2) (> 3 4))

= (**cond** [(= 2 2) *true*]
[**else** (**myor** (> 3 4))])

= (**cond** [(= 2 2) *true*]
[**else** (**cond** [(> 3 4) *true*]
[**else** (**myor**)])])

= *error*

Notice that the second else clause needs to expand (**myor**), but the input pattern requires at least one argument (*e1*). To handle this, add another pattern to the macro that defines **myor** with no arguments.
then we need to adjust the output pattern. How about the following?

```
(define-syntax myor
  (syntax-rules ()
    [(myor) false]
    [(myor e1 e2 ...)
     (cond [e1 true]
           [else (myor e2 ...)])]))
```

The first pattern is like the base case in a recursive function – it gives a concrete answer on a specific number of arguments. The second is the recursive case. As in the recursions we've written before, we reduce the number of arguments on the recursive call.

2 Real For-loops

A friend is fed-up with the lack of for and while loops in Scheme and decides to implement his own for-loop construct. As an example, your friend wants to be able to type

(*for i* 0 5 (*printf* "~a " *i*))

and have Scheme produce

0 1 2 3 4 5

(with *void* returned as the result of the expression).

Should you implement **for** as a function or a macro? Why?

For looks like a macro because of the expression you want to iterate over. If you implement **for** as a function, you'll run the *printf* when you call **for**, and you'll get an unbound identifier error because you have no value for *i*. With that in mind, write a macro for **for**.

```
(define-syntax for
  (syntax-rules ()
    [(for var low high expr)
     (local [(define (loop var)
               (cond [(> var high) void]
                     [else (begin expr
                                  (loop (+ 1 var)))]))]
       (loop low))]))
```

Notice that this for-loop does not use assignment (as you are accustomed to using with for-loops). There's no need to assign an index variable – you can just use recursion. Isn't this limiting though, because the macro fixes how much to increase *i* by each time? We could relax that by adding another pattern to the macro that supplies that information:

```
(define-syntax for
  (syntax-rules ()
    [(for var low high expr)
     (local [(define (loop var)
               (cond [(> var high) void]
                     [else (begin expr
                                  (loop (+ 1 var)))]))]
       (loop low))]
     [(for var low high expr incr)
     (local [(define (loop var)
               (cond [(> var high) void]
                     [else (begin expr
                                  (loop (+ var incr)))]))]
       (loop low))]))
```

Note that this macro duplicates a lot of code across the two cases. We can fix that by converting the case with no increment supplied into the case with an increment supplied.

```
(define-syntax for
  (syntax-rules ()
    [(for var low high expr)
     (for var low high expr 1)]
    [(for var low high expr incr)
     (local [(define (loop var)
               (cond [(> var high) void]
                     [else (begin expr
                                  (loop (+ var incr)))]))]
       (loop low))]))
```

But wait! This looks recursive, like what we had tried for the multi-armed or. Why does this one work, when the recursive or did not? This macro isn't recursive. The first version rewrites into the second *one time only*. The recursion is in the loop function, which does not use **for** again. In the case of **or**, the recursion invoked the macro itself; that created the infinite loop.

As an aside, you may have wondered why we didn't try to break up the **for** macro into two cases as we did for the multi-armed or. For that, you might have tried the following:

```
(define-syntax for
  (syntax-rules ()
    [(for var low var expr) expr]
    [(for var low high expr)
     (begin expr (for var low (+ 1 var)))]))
```

This isn't a valid macro, since the first cases tries to use var twice and Scheme doesn't support that. You can't split up these cases in this manner.

3 Map

Since we've written a recursive macro for **or**, why don't we write one for map? How about this?

```
(define-syntax mymap
  (syntax-rules ()
    [(mymap func lst)
     (cond [(empty? lst) empty]
           [(cons? lst)
```

```
        (cons (func (first lst))
              (mymap func (rest lst)))])))
```

If we test this macro, we find we go into an infinite loop. Why? Recursion worked for **or**, so why doesn't it work for map?

There's a big difference between the two macros. With **myor**, notice that the recursion has a base case *as one of the patterns in the macro*. In **mymap**, there's only one case. Macro-expansion works by replacing every use of a macro with its output pattern until no more uses of macros remain. Furthermore, since macro-expansion takes place without evaluating expressions (the whole point of having them!), there's no way to hit a base case within the expanded code. In other words, macro-expansion of a call to **mymap** proceeds in several steps, ad infinitum:

```
  (mymap square (list 1 2 3))

= (cond [(empty? (list 1 2 3)) empty]
        [(cons? (list 1 2 3))
         (cons (square (first (list 1 2 3)))
               (mymap square (rest (list 1 2 3))))])

= (cond [(empty? (list 1 2 3)) empty]
        [(cons? (list 1 2 3))
         (cons (square (first (list 1 2 3)))
               (cond [(empty? (rest (list 1 2 3))) empty]
                     [(cons? (rest (list 1 2 3)))
                      (cons (square (first (rest (list 1 2 3))))
                            (mymap square (rest (rest (list 1 2 3)))))])))])

= ...
```

Without evaluating the lists, there's no way to stop the expansion, hence the infinite loop.

4 Summary

Macros can be recursive, but the base case must be one of the (multiple) patterns, not buried *within* one of the output patterns. This is necessary for macro expansion to terminate.