CS2135: Recursive Macros

Kathi Fisler, WPI

February 5, 2003

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. The question arose about how to define an **or** macro that accepts arbitrary numbers of arguments. Scheme **or** can take any number of arguments. Here are some examples:

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

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

Try writing a macro for **or** that takes an arbitrary number of arguments. Hint: think of the multiple-case *timecond* macro from the previous set of notes. Also think about recursion.

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

Here's the macro. What do the two patterns do? The first is like the base case in the recursion – 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 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.

3 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.