

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Optional: Tokenization, Parenthesization, and Scope

Tokenization

First question for a macro system: How does it tokenize?

- Macro systems generally work at the level of *tokens* not sequences of characters
 - So must know how programming language tokenizes text
- Example: “macro expand **head** to **car**”
 - Would not rewrite **(+ headt foo)** to **(+ cart foo)**
 - Would not rewrite **head-door** to **car-door**
 - But would in C where **head-door** is subtraction

Parenthesization

Second question for a macro system: How does associativity work?

C/C++ basic example:

```
#define ADD(x,y) x+y
```

Probably *not* what you wanted:

`ADD(1,2/3)*4` means `1 + 2 / 3 * 4` not `(1 + 2 / 3) * 4`

So C macro writers use lots of parentheses, which is fine:

```
#define ADD(x,y) ((x)+(y))
```

Racket won't have this problem:

- Macro use: `(macro-name ...)`
- After expansion: `(something else in same parens)`

Local bindings

Third question for a macro system: Can variables shadow macros?

Suppose macros also apply to variable bindings. Then:

```
(let ([head 0] [car 1]) head) ; 0  
(let* ([head 0] [car 1]) head) ; 0
```

Would become:

```
(let ([car 0] [car 1]) car) ; error  
(let* ([car 0] [car 1]) car) ; 1
```

This is why C/C++ convention is all-caps macros and non-all-caps for everything else

Racket does *not* work this way – it gets scope “right”!