What is (twice 4)	8
And (twice -1)	-2
Can you define twice?	Sure. It's (lambda (x) (* 2 x))
What's (map twice '(4 -1))	(8 -2)
What's (map '(4))	(8)
What should (map '()) be?	() sounds like a good idea.
map is part of the language, but here's a definition anyway. (define map (lambda (f lst) (cond ((null? lst) '()) (else (cons (f (car lst)) (map f (cdr lst)))))))	That's old news, isn't it?
What does (map twice (map twice x)) mean?	Multiply each element of x by 4.
Can you express this formally.	Sure. How about (map (lambda (x) (* 4 x)) x)
Does this work?	Sure. Remember the distinction between free and bound variables. The x inside the lambda expression is bound.
Does (map (lambda (x) (* 4 x)) x) make happy?	No. It's too different than the original expression. It would be nice to use twice in the simplification.
Does this work? (map (twice twice) x)	No! (twice twice) produces an error message.
How come?	Twice expects a number, not a procedure argument.
Here's a solution (map twice (map twice x)) == (map (lambda (x) (twice (twice x))) x)	Much better.
Can you see a difference between the two equivalent expressions?	The first traverses two lists (x, and the result of the first map application). The second expression traverses only one list.

Use the following to express the simplified expression more succinctly.
(define o
  (lambda (f1 f2)
    (lambda (x)
      (f1 (f2 x)))))
We use o to represent function composition.
	(map (o twice twice) x)

Can we use o to simplify (map twice (map twice x))?	Well, it should be ((o (map twice) (map twice)) x)
Does the work?	No!
Why not?	Map expects two arguments, and it is only given one: twice.
Can we fix this problem using the same kind of trick we used before when defining o?	Yes, we can do something similar. Do it!
Is this really elegant?	No, but you asked for it!
OK, forget I asked. Let's go back for a minute. We saw that (map twice (map twice x)) == (map (lambda (x) (twice (twice x))) x) would this hold for any function, or is it specific to twice?	Should work.
Let's try to explain it in words.	Here's our explanation: On the left hand side, we apply the function (let's call it f) to each element of x, and produce an list of results. We then apply f to all the elements in the result list. On the right hand side we apply f twice to each element of x. The result is the same.

And if instead of one function we had two? (map f (map g x)) == (map (lambda (x) (f (g x))) x)	That works too.

Here a new function
(define so-of-twice
  (let ((counter 0))
    (lambda (x)
      (set! counter (+ 1 counter))
      (+ (* 2 x) counter))))
	Looks pretty meaningless to us.

Bare with us, please. What's (map son-of-twice (map son-of-twice '(1 2 3)))	We get (10 17 24)
What's (map (o twice son-of-twice) '(1 2 3))	(8 18 28)
See the problem?	These results should have been equal!
But you promised!	It was you! I am only doing what I am told.
Can we explain what happened?	Son-of-Twice behaves a little differently each time it is invoked, because counter keeps on changing.
Right. This is called a side-effect.	We don't like those, don't we?
Sure don't.	Good, I had a feeling this sort of thing can make a girl cry.
Not to mention grown up programmers. Was this the only problem we had today?	Don't get me started on the traffic.
No, we are talking about our manipulation of map expressions.	Well, there's were these problems with (twice (twice)), and (map twice).
All in a days work.	But can't we do better?

Of course we can, that's why we will use the **Haskell** language.	But only after we had some pizza, right?
And a tall Chocolate Brownie Frappuccino®	Our's was tasty. How was yours?