Macros are an alternative to functions.

We've seen several macros already:
- We've noted that they do not have to evaluate all of their arguments.
- Common macros:
  - if, and, or, do, setf...

Macros do not have their own lexical context. Instead, they are replacement code, replaced at compile time—much like #defines in C, but more frequently used.

 Definition

Before we show how to write Macros, we should look at what they actually do.
- We can look at functions as being statements of code executed within a new lexical context.
- Macros do not have their own lexical context.
- Instead, they are replacement code, replaced at compile time—much like #defines in C, but more frequently used.

```
(defun macro-name (args) (macro-expansion))
```

This is not as simple as it looks.
- Macro expansion should expand to look just like Lisp code. We'll see an example of this in a second.
Macro Example

> (defmacro me (x)
  (list 'setf x ''Kennen))
> (setf f '(1 2 3))
  (1 2 3)
> (me f)
  KENN
> f
  KENN

Macro Example II

> (macroexpand '(me f))
  (SETQ F 'KENN)
  T
> (macroexpand) takes a macro call, expands it and returns what it looks like. This is useful for debugging.

Macro Example With A Micro Error

(defun me-bad (x)
  (setf x 'Kenn))
> (me-bad x)
  ;; Error: Unbound variable KENN in #<function 0 #xE8AA0C>
  ;; Returning to Top Level
  WRONG!!!

A Macro to Make Macros Easier: Backquote

Nifty!
` by itself is identical to 
However, within a backquoted form, you can turn evaluation back on using the , prefix
You can also use ,@ to turn evaluation of a list on, with splicing (each element is inserted).

Backquote Examples

> (setf x 1)
  1
> `(You are number ,x)
  (YOU ARE NUMBER 1)
> (defmacro me-back (x)
  `(setf ,x 'Kenn))
> (macroexpand `(me-back x))
  (SETQ X 'KENN)

Common Macro Errors (Read "Big Mistakes")

Variable Capture
  Shadowing a variable with a new lexical variable
Multiple Evaluation
  Evaluating an argument to a macro more than once

Kenneth W. Flynn (7-12)
09/30/98
A Complex Example

- Suppose we want to write a repeat-until macro.
- Let's pass it a function to evaluate as the until, which takes no arguments (lambda function?).
- Let's also pass it a maximum number of times to loop (to prevent infinite loops).
- Let's finally pass it some expressions.

Repeat-Until, Pass 1

```lisp
(defmacro repeat-until-or-max
  (done-p max &rest body)
  `(progn
    ,@body
    `(do ((numtimes 1 (+ numtimes 1)))
      ((or (funcall ,done-p)
          (= ,max numtimes))
       ,@body)
    ))
```

Example Call

```lisp
(defun tester ()
  (let ((x 0)
         (y 5))
    (repeat-until-or-max
     #'(lambda () (> x y))
     10
     (setf x (+ x 1)))
  )
)\n```

```lisp
> (tester)
6
```

Problem #1

```lisp
(defun tester2 ()
  (let ((x 0)
         (y 5)
         (numtimes 10))
    (repeat-until-or-max #'(lambda () (> x y))
                          numtimes
                          (setf x (+ x 1)))
  )
)\n```

```lisp
> (tester2)
1
```

Problem #2

```lisp
(defun tester3 ()
  (let ((x 0)
         (y 5)
         (z 6))
    (repeat-until-or-max #'(lambda () (> x y))
                         (setf z (- z 1))
                         (setf x (+ x 1)))
  )
)\n```

```lisp
> (tester3)
3
```

gensym

- Generates "uninterned" symbol - a symbol that is not part of any package.
- Cannot conflict with any of your symbols.
- Can be used to avoid Variable Capture.

```lisp
> (gensym)
#:C40
```
Avoiding Multiple Evaluation

- Use a gensym and bind it to that which you don't want to keep evaluating.
- See the example...

Correct Repeat

```lisp
(defun repeat-until-or-max-1
  (done-p max &rest body)
  (let ((numtimes (gensym))
        (g-max (gensym)))
    `(let ((,g-max ,max))
        ,@body
        (do ((,numtimes 1 (+ ,numtimes 1)))
            ((or (funcall ,done-p)
                 (= ,max ,numtimes))
             ,numtimes
             )
          ,@body)))
```

 Trying It Out

```lisp
> (tester-1)
6
> (tester2-1)
6
> (tester3-1)
6
```

That's It!

- For next time
  - Work on Project #2
  - Read Chapter #10 on Macros
  - Try it out!
- Next class
  - CLOS!