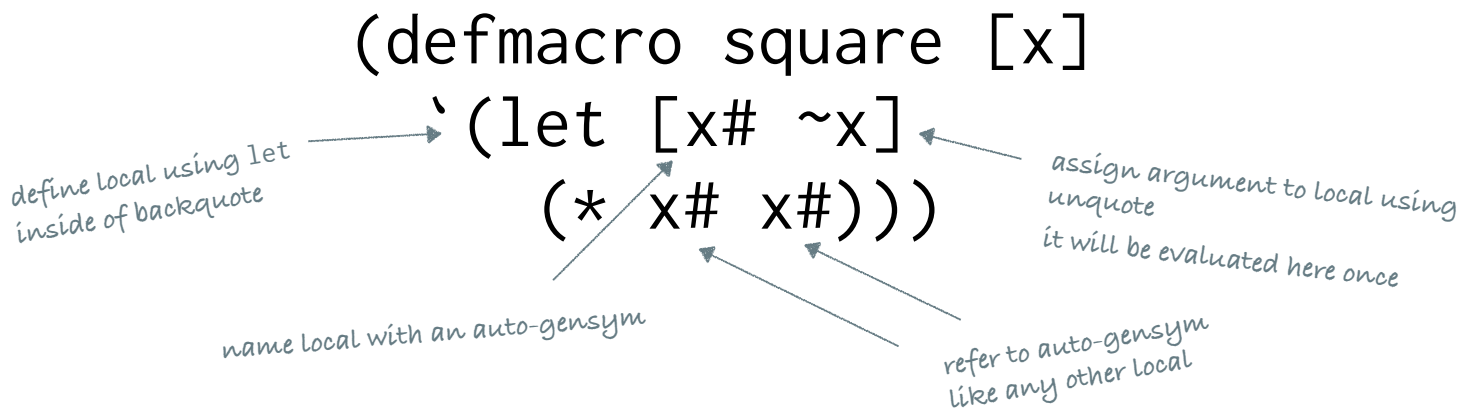


Auto-Gensym Local



Use auto-gensyms to define locals to prevent evaluating arguments multiple times.

Body of Expressions

```
(defmacro when* [test & body]
  `(if ~test
      (do
        ~@body)))
```

*we need a do here because
you can only put one
expression in the then
branch*

*use a rest argument to capture a body
it will be a list of expressions*

splice the body in

Use a rest argument to capture a "body" and splice the body into place. You may need to wrap multiple expressions in a do.

Unquote Quote Binding

nice auto-gensym!

check out the body of expressions pattern!

```
(defmacro regex [re s & body]
  `(let [match# (re-find ~re ~s)]
    (when match#
      (let [[~'%0 ~'%1 ~'%2]
        (if (string? match#)
            [match#]
            match#))
        ~@body))))
```

use unquote quote to bind specific symbols in a let these symbols will shadow ones defined in a scope around the macro call the body can refer to them, but it might shadow them as well

expressions in body can refer to the bindings use well-known bindings they are part of the interface of the macro

Use unquote quote to get an namespace-unqualified symbol and bind it in a let. That way, you can refer to the local in your body.

Macro with Helper Function

define a helper function to do the heavy lifting
as a bonus, this function may be useful to call
on its own

helper fn should take a
function to call
the function is made in the
macro using the body

applying f to these is
equivalent to a let since the
binding names were
turned into arguments

```
(defn with-open*-fn [to-close f]
  (try
    (apply f to-close)
    (finally
      (doseq [c (reverse to-close)]
        (.close c))))))
```

macro will do mostly
syntactic transformation

```
(defmacro with-open* [bindings & body]
  (let [pairs (partition 2 bindings)]
    `(with-open*-fn ~(mapv second pairs)
      (fn ~(mapv first pairs)
        ~@body))))
```

call the function in
a backquote for it to
be called at runtime

create a function
using the body

construct the argument vector to
correspond to the binding names

in this case we construct a vector of
streams to close that we pass to the
function
these are also the parameters for the
function

**Separate a complicated macro
into a macro and a helper
function. The helper function will
be easier to debug.**

Binding Macro

define a helper function to do the heavy lifting
as a bonus, this function may be useful to call
on its own

accept a binding vector just
like in a let

```
(defmacro with-open* [bindings & body]
  `(let ~bindings
    (try
      ~@body
      (finally
        ~@(for [[sym _]
                (reverse (partition 2 bindings))]
              `(.close ~sym))))))
```

you can unquote the
bindings right in the let

bindings will be available
inside the body

unquote splice is great for
use with for
comprehensions
the for will be evaluated at
compile time

you can nest another level of
backquote here to make sure this
evaluates at runtime

Allow macro user to bind locals by using a binding vector, just like let.

Manual Gensyms

```
(defmacro mf [m]
  (let [letters "abcdefghijklmnop"
        syms (map (comp gensym str) letters)]
    `(fn
       ~@(for [n (range (inc (count letters)))]
            `([~@(take n syms)]
              (~m ~@(take n syms)))))))
```

we can build our own gensyms out of letters if we need lots of them

map gensym over strings and you can make as many unique symbols as you need

generate your symbols at compile time

make an argument vector with all of those symbols

you can refer to those arguments in the body of the function

Use manual gensyms to create as many symbols as you need. They are useful for long argument lists.