Variable scope and variable capture

✔ Sometimes it is necessary for a macro to introduce new variables in its macroexpansion

✔ However, these new variables can...
  ✗ “capture” variables in the macroexpanded construct
  ✗ be “captured” by variables in surrounding constructs

✔ This can be a feature or a bug...
Variable scope and variable capture

✔ Suppose you want to call function foo on the result of a big, long calculation, if the result of the calculation is not NIL.

(when (big-long-calculation) (foo (big-long-calculation)))

✔ will work, but might do the big calculation twice. What you want instead is something like this, which does it only once:

(let ((result (big-long-calculation)))
  (when result (foo result)))

✔ So, you can define a macro

(defun foowhen (fcn)
  '(let ((result (funcall ,fcn)))
    (when result (foo result))))

✔ so that (foowhen #'big-long-calculation) macroexpands to

(let ((result (funcall #'big-long-calculation)))
  (when result (foo result)))
Variable scope and variable capture

✔ Now suppose you want to call function \texttt{foo} on the result of a big, long calculation together with some other arguments, if the result of the calculation is not NIL...

\begin{verbatim}
(when (big-long-calculation)
  (foo (big-long-calculation) arg1 arg2))
\end{verbatim}

✔ To avoid doing the big long calculation twice, we could try the macro trick

\begin{verbatim}
(defmacro foowhen (fcn &rest args)
  `(let ((result (funcall ,fcn)))
     (when result (foo result ,@args))))
\end{verbatim}

✔ Now (foowhen #'big-long-calculation arg1 arg2) macroexpands to

\begin{verbatim}
(let ((result (funcall #'big-long-calculation)))
  (when result (foo result arg1 arg2))
\end{verbatim}

✔ But there is a potential variable capture bug here...
Variable capture

✔ Consider evaluating the expressions

USER: (defun foo (arg1 arg2) (list arg1 arg2))

USER: (defun bar () ’hello)

USER: (foowhen #'bar ’world)
(HELLO WORLD)

USER: (setq result ’world)

USER: (foowhen #'bar result)

(HELLO HELLO)

✔ The call to foowhen macroexpands to

(let ((result (funcall #'bar)))
  (when result (foo result result))

✔ The variable result in the macro call has been captured by the variable result introduced in the macroexpansion...
Avoiding variable capture

✔ You could try to get around this problem by rewriting the macro to use a bizarre variable name...

```
(defmacro foowhen (fcn &rest args)
  `(let ((var***123 (funcall ,fcn)))
    (when var***123 (foo var***123 ,@args))))
```

✔ But LISP provides a better way: `gensym`

✗ each time it is called, the function `gensym` returns a symbol which is guaranteed to be unique

```
USER: (gensym)
#:G229
```

✗ there is no way to type in a symbol that `gensym` returns

✔ A version of `foowhen` that uses `gensym` to avoid variable capture:

```
(defmacro foowhen (fcn &rest args)
  (let ((result (gensym)))
    `(let ((,result (funcall ,fcn)))
      (when ,result (foo ,result ,@args))))
```

✔ Now `(foowhen #’bar result)` macroexpands to something like

```
(let ((#:G233 (funcall #’bar)))
  (when #:G233 (foo #:G233 result))
```
Using variable capture

✔️ Now suppose you want to have the effect of `foowhen` on other functions besides `foo`... That is, you would like the effect of

```
(when (big-long-calculation)
  (baz (big-long-calculation)))
```

for any function `bar`, without doing the big long calculation twice.

✔️ A neat way to do this is by defining an *anaphoric macro* `fwhen` which you can call as

```
(fwhen (big-long-calculation) (baz it))
```

✔️ This definition works:

```
(defmacro fwhen (form action)
  '(let ((it ,form))
    (when it ,action)))
```

✔️ The macroexpansion of

```
(fwhen (big-long-calculation) (baz it))
```

is

```
(let ((it (big-long-calculation)))
  (when it (baz it)))
```

✔️ The variable `it` in the macro call is captured by the variable `it` introduced in the macroexpansion... which is a good thing in this case!