Introduction
0000

Macro systems
0000

Variable Capture
00
0000

Macros and Metaprogramming

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Introduction
●○○○

Macro systems
○○○○

Variable Capture
○○
○○○○

Objectives
You should be able to ...

- ▶ See three methods for making programs that write other programs.
- ▶ Understand the syntax of the defmacro form.
- ▶ Compare Lisp's defmarco to C's #define.
- ▶ Use defmacro to extend a language.
- ▶ Explain the concept of variable capture, both accidental and intentional.
- ▶ Explain why Haskell doesn't have macros.

Introduction
○●○○

Macro systems
○○○○

Variable Capture
○○
○○○○

Three Ways to Write Programs That Write Programs

1: Compose strings!

```
1 ELISP> (defun str-make-inc (name delta)
2           (concat "(defun " name
3                   " (x) (+ x " delta "))"))
4 str-make-inc
5 ELISP> (str-make-inc "five" "5")
6 "(defun five (x) (+ x 5))"
```

- ▶ The code examples are in Emacs Lisp, using the IELM repl.
  Use M-x ielm to start it.
- ▶ Advantages: easy to get started; cross-language support
- ▶ Disadvantages: very easy to break
- ▶ Quine – a program that, when run, outputs its own source code

Introduction
○○●○

Macro systems
○○○○

Variable Capture
○○
○○○○

Three Ways to Write Programs That Write Programs

2: Build ASTs!

```
1 ELISP> (defun ast-make-inc (name delta)
2               `(defun ,name (x) (+ x ,delta)))
3 ast-make-inc
4 ELISP> (ast-make-inc 'five 5)
5 (defun five (x) (+ x 5))
6 ELISP> (eval (ast-make-inc 'five 5))
7 five
8 ELISP> (five 23)
9 28 (#o34, #x1c, ?\C-\\)
```

▶ The eval function compiles ASTs.
▶ The read function (not showns) converts strings to ASTs.
▶ Advantages: much simpler to manipulate code
▶ But you need language support for manipulaing the syntax tree.

Introduction
○○○●

Macro systems
○○○○

Variable Capture
○○
○○○○

Three Ways to Write Programs That Write Programs

3: Use a macro!

```
1 ELISP> (defmacro make-inc (name delta)
2              `(defun ,name (x) (+ x ,delta)))
3 make-inc
4 ELISP> (make-inc ten 10)
5 ten
6 ELISP> (ten 123)
7 133 (#o205, #x85, ?)
8 ELISP>
```

► This skips the eval step.
► But you need language support for macros.

Introduction
0000

Macro systems
●000

Variable Capture
00
0000

Macros Are Lazy, Functions Are Usually Not

```
1 E> (defun my-if (test true false)
2           (if test true false))
3 my-if
4 E> (defun fact (n) (my-if (> n 0) (* n (fact (- n 1))) 1))
5 fact
6 E> (fact 4) ;; Runs out of stack space
```

but ...

```
1 E> (defmacro my-if (test true false)
2           `(if ,test ,true ,false))
3 my-if
4 E> (defun fact (n) (my-if (> n 0) (* n (fact (- n 1))) 1))
5 fact
6 E> (fact 4)
7 24 (#o30, #x18, ?\C-x)
```

Introduction
0000

Macro systems
0●00

Variable Capture
00
0000

We Hate Boilerplate

```
1 (let ((handle (fopen "file.txt")))
2     (try
3         ... do stuff with file ...
4         (catch e (print "Yikes! and Error!"))
5         (finally (close handle))))
```

▶ Most Lisps have macros to abstract this.

```
1 (with-open handle "file.txt"
2     ... do stuff with file ...)
```

Introduction
0000

Macro systems
0000

Variable Capture
00
0000

Domain Specific Languages

- ▶ Macros are used extensively in DSLs.
- ▶ Here is the html macro from Clojure's hiccup package.
- ▶ Can handle

```
1 user> (html [:p [:a {:href "http://google.com"} "Google"]
2                  "is not a verb."])
3 "<p><a href=\"http://google.com\">Google</a>is not a verb.</p>"
4 user> (html [:ul (for [i (range 3)] [:li i])])
5 "<ul><li>0</li><li>1</li><li>2</li></ul>"
```

Introduction
0000

Macro systems
0000●

Variable Capture
00
0000

We Like to Rewrite Code

▶ Lisp style macros are more powerful than C style macros.

▶ `#define` can only rearrange text.

▶ `defmacro` can perform arbitrary code rewrites!

```
1 ELISP> (subst '- '+ '(* 2 (+ 3 4)))
2 (* 2 (- 3 4))
3 ELISP> (defmacro unplus (tr) (subst '- '+ tr))
4 unplus
5 ELISP> (unplus (* 2 (+ 10 9)))
6 2
```

Introduction
0000

Macro systems
0000

Variable Capture
●○
0000

Unintended Capture

```
1 ELISP> (setq sum 10)
2 10 (#o12, #xa, ?\C-j)
3 ELISP> (defmacro mk-sum (a b)
4          `(let ((sum (+ ,a ,b)))
5             (list ,a ,b sum)))
6 mk-sum
7 ELISP> (mk-sum 2 3)
8 (2 3 5)
9 ELISP> (mk-sum 2 sum)
10 (2 12 12)
```

▶ We want to store the sum of the arguments, but we need a fresh variable.

Introduction
0000

Macro systems
0000

Variable Capture
○●
0000

Gensym

▶ gensym to the rescue!

```
1 ELISP> (gensym)
2 G99398
3 ELISP> (defmacro mk-sum (a b)
4          (let ((sum (gensym)))
5            `(let ((,sum (+ ,a ,b)))
6               (list ,a ,b ,sum))))
7 mk-sum
8 ELISP> (mk-sum 2 3)
9 (2 3 5)
10 ELISP> (mk-sum 2 sum)
11 (2 10 12)
```

Introduction
OOOO

Macro systems
OOOO

Variable Capture
OO
●OOO

Anaphoric Macros

▶ Here is a pattern you see a lot.

```
1 ELISP> (defun open-exists (fname)
2            (if (file-exists-p fname)
3                (find-file fname)))
4 open-exists
5 ELISP> (open-exists "/asdf")
6 nil
7 ELISP> (open-exists "/tmp")
8 #<buffer tmp>
9 ELISP> (let ((the-buffer (open-exists "/tmp")))
10           (if the-buffer (buffer-name the-buffer)
11               "none")))
12 "tmp"
```

Introduction
0000

Macro systems
0000

Variable Capture
00
0●00

Anaphoric `if`

```elisp
1 ELISP> (defmacro a-if (cond then else)
2                `(let ((it ,cond))
3                     (if it ,then ,else)))
4 ELISP> (a-if (open-exists "/tmp")
5              (buffer-name it) "nope.")
6 "tmp"
7 ELISP> (a-if (open-exists "/tm4444p")
8              (buffer-name it) "nope.")
9 "nope."
```

Introduction
0000

Macro systems
0000

Variable Capture
00
000●

Pattern Matching

▶ More frequently it's better that we chose the variable names ourselves.

```
1 ELISP> x
2 (6 . 7)
3 ELISP> (defmacro match (thing pattern body)
4                `(let (( ,(car pattern) (car ,thing))
5                      ( ,(cdr pattern) (cdr ,thing)))
6                   ,body))
7 match
8 ELISP> (match x (a . b) (+ a b))
9 13 (#o15, #xd, ?\C-m)
```

Introduction
0000

Macro systems
0000

Variable Capture
00
000●

Conclusions

- ▶ Most languages do not have a macro system!
- ▶ Haskell "doesn't need one."
  - ▶ Monads / type classes wrap boilerplate.
  - ▶ Laziness is already built in.
  - ▶ There is a template Haskell though.
- ▶ Macros are difficult to reason about.
- ▶ Most programmers were never taught them.
- ▶ Work best in a homoiconic language.