Macros and Metaprogramming

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

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.

Three Ways to Write Programs That Write Programs

1: Compose strings!

```
ELISP> (defun str-make-inc (name delta)
          (concat "(defun " name
                  " (x) (+ x " delta "))"))
str-make-inc
ELISP> (str-make-inc "five" "5")
"(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

Three Ways to Write Programs That Write Programs

2: Build ASTs!

```
ELISP> (defun ast-make-inc (name delta)
             `(defun ,name (x) (+ x ,delta)))
ast-make-inc
ELISP> (ast-make-inc 'five 5)
(defun five (x) (+ x 5))
ELISP> (eval (ast-make-inc 'five 5))
five
ELISP> (five 23)
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.

Three Ways to Write Programs That Write Programs

3: Use a macro!

```
ELISP> (defmacro make-inc (name delta)
             `(defun ,name (x) (+ x ,delta)))
make-inc
ELISP> (make-inc ten 10)
ten
ELISP> (ten 123)
133 (#o205, #x85, ?)
ELISP>
```

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

Macros Are Lazy, Functions Are Usually Not

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

but ...

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

We Hate Boilerplate

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

- Most Lisps have macros to abstract this.

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

Domain Specific Languages

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

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

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!

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

Unintended Capture

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

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

Gensym

- gensym to the rescue!

```
ELISP> (gensym)
G99398
ELISP> (defmacro mk-sum (a b)
           (let ((sum (gensym)))
              `(let ((,sum (+ ,a ,b)))
                    (list ,a ,b ,sum))))
mk-sum
ELISP> (mk-sum 2 3)
(2 3 5)
ELISP> (mk-sum 2 sum)
(2 10 12)
```

Anaphoric Macros

- Here is a pattern you see a lot.

```
ELISP> (defun open-exists (fname)
             (if (file-exists-p fname)
                 (find-file fname)))
open-exists
ELISP> (open-exists "/asdf")
nil
ELISP> (open-exists "/tmp")
#<buffer tmp>
ELISP> (let ((the-buffer (open-exists "/tmp")))
            (if the-buffer (buffer-name the-buffer)
                "none")))
"tmp"
```

Anaphoric if

```
ELISP> (defmacro a-if (cond then else)
               `(let ((it ,cond))
                     (if it ,then ,else)))
ELISP> (a-if (open-exists "/tmp")
             (buffer-name it) "nope.")
"tmp"
ELISP> (a-if (open-exists "/tm4444p")
             (buffer-name it) "nope.")
"nope."
```

Pattern Matching

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

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

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.