Macros

Pascal Costanza
Programming Technology Lab
Vrije Universiteit Brussel

...not yet...

- First something completely unrelated.

- (let ((a 1) (b 2) (c 3) (d 4))
  (list a b c d))

  => (1 2 3 4) ; everything is evaluated

- (let ((a 1) (b 2) (c 3) (d 4))
  (list ‘a b c d))

  => (A 2 3 4) ; not everything is evaluated

...not yet...

- (let ((a 1) (b 2) (c 3) (d 4))
  (list ‘a ‘b ‘c d))

  => (A B C 4) ; very little is evaluated

- Here is a more concise way to write this:

(let ((a 1) (b 2) (c 3) (d 4))
  `(a b c ,d))

=> (A B C 4) ; very little is evaluated

Backquote

- `(a b c ,d) uses backquote
- ‘(a b c d) uses quote
- backquote allows evaluating parts of an expression explicitly marked with a comma
- you can’t do this with quote

Backquote

- `(a b c) <=> ‘(a b c)
- `(a ,b c ,d) <=> (list ‘a b ‘c d)
- (let ((b 2)) `(a (,b c))) => (A (2 C))
- (let ((a 1) (b 2) (c 3))
  `(a b ,c (‘,(+ a b c)) (+ a b) ‘c ‘((,a ,b))))

  => (A B 3 (‘6) (+ A B) ‘C ‘((1 2)))

More Backquote

- (let ((list ‘(1 2 3)))
  `(a b ,@list c d))

  => (a b 1 2 3 c d)

- ,@ (comma-at) splices into the surrounding list
  (so there must be a surrounding list!)

Macros

- This is code: (+ 1 2 3)
- This is data: ‘(+ 1 2 3)
- Macros are like functions,
but they take code as arguments and return new code.

Macro Example

- (defun while-fun (predicate thunk)
    (when (funcall predicate)
      (funcall thunk)
      (while-fun predicate thunk)))

- (defmacro while (expression &body body)
  (list ‘while-fun (list ‘lambda ‘() expression)
                   (list* ‘lambda ‘() body)))

Macro Example

```
• (defun qsort (vector low high)
      (let* ((left low)
             (right high)
             (pivot ...))
          ...
            (while (< (svref vector left) pivot)
              (incf left))
...))
```

Macro Example

- (funcall (macro-function ‘while)
    ‘(while (< (svref vector left) pivot)
        (incf left))
    environment)
  => (WHILE-FUN
        (LAMBDA NIL
    (< (SVREF VECTOR LEFT) PIVOT))
(LAMBDA NIL (INCF LEFT)))

Macro Example

- (defun qsort (vector low high)
  (let* (...)
  ...
  (WHILE-FUN
  (LAMBDA NIL
  (< (SVREF VECTOR LEFT) PIVOT))
  (LAMBDA NIL (INCF LEFT)))
  ...))

Macros + Backquote

- This looks ugly:

```
(defmacro while (expression &body body)
    (list ‘while-fun (list ‘lambda ‘() expression)
                     (list* ‘lambda ‘() body)))
```

- You can also write that:

```
(defmacro while (expression &body body)
    `(while-fun (lambda () ,expression)
                (lambda () ,@body)))
```

Remember!

- Backquote is independent from macros!

- (defun greet (name)
    `(hello ,name))

  ...is a function!

Why Macros?

- Use macros for syntactic abstractions.

- Question: Why not just say this?

```
(while (lambda () (< (svref ...) pivot))
    (lambda () (incf left)))
```

Syntactic Abstractions

- The while function leaks: You need to know details about its implementation!

- That is, the fact that it uses closures.

Alternative implementations of while.

- (defmacro while (expression &body body)
  `(do () ((not ,expression)) ,@body))

- (defmacro while (expression &body body)
  `(tagbody
    start (unless ,expression (go end))
    ,@body
    (go start)
end))

Abstractions

- Syntactic abstractions hide implementation details,
  just like functional abstractions.

- Hiding implementation details allows you to change your mind later on,
  and allows the users of your library to think purely in terms of what they care about.

Abstractions

- (while-fun (lambda () (< (svref ...) pivot))
  (lambda () (incf left)))

  vs.

- (while (< (svref ...) pivot)
  (incf left))

How to write macros

- Decide if the macro is really necessary.
- Write down the syntax of the macro.
- Figure out what the macro should expand into.
- Use defmacro to implement the syntax/expansion correspondence.

Is a macro necessary?

- (defun square (x) (* x x))

  vs.

- (defmacro square (x) `(* ,x ,x))

- (defun avg (&rest args)
    (/ (apply #'+ args) (length args)))

  vs.

- (defmacro avg (&rest args)
    `(/ (+ ,@args) ,(length args)))

Write down the syntax

- (nif 28
    (print ‘positive)
    (print ‘zero)
    (print ‘negative)) => POSITIVE

- (nif expression
    if-positive
    if-zero
    if-negative)

What is the expansion?

- (nif expression
    if-positive
    if-zero
    if-negative)

- (let ((value expression))
  (cond ((> value 0) if-positive)
        ((= value 0) if-zero)
        ((< value 0) if-negative)))

Implement the macro

- (let ((value expression))
    (cond ((> value 0) if-positive)
          ((= value 0) if-zero)
          ((< value 0) if-negative)))

- (defmacro nif (expression pos zero neg)
    `(let ((value ,expression))
       (cond ((> value 0) ,pos)
             ((= value 0) ,zero)
             ((< value 0) ,neg))))

Test the macro

- (macroexpand
    ‘(nif 28
        (print ‘positive)
        (print ‘zero)
        (print ‘negative))))

Examples

- dovector, scheme-let, when-let, when-let*, with-open-file

destructuring-bind

- (destructuring-bind
    (a (b) c)
    ‘(1 (2) 3)
  (list a b c))

  => (1 2 3)

- (destructuring-bind
    (x (y) . z)
    ‘(a (b) c d)
  (list x y z))

  => (A B (C D))

What’s wrong?

- (defmacro our-dotimes (n &body body)
  `(progn
  ,@(do ((i 1 (1+ i))
  (form body (append form body)))
  ((>= i n) form))))

- (our-dotimes 3 (print ‘hi)) => prints ‘hi 3 times

- (let ((n 3))
  (our-dotimes n (print ‘hi))) => error?!?

What’s wrong?

- (defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
  (limit ,stop))
  ((> ,var limit))
  ,@body))

- > (for (x 1 5)
  (print x))
  12345
  NIL

- > (for (limit 1 5) ; Macro-expand it to see the resulting code
  (print limit)

What’s wrong?

- (defmacro swap (ref1 ref2)
  `(let (temp)
    (setq temp ,ref1)
    (setq ,ref1 ,ref2)
    (setq ,ref2 temp)))

Variable Capture

- Assume the following definitions:

- (defmacro push (obj-exp list-var)
  `(setq ,list-var (cons ,obj-exp ,list-var)))

- (defmacro or (exp1 exp2)
  `(let ((temp ,exp1))
    (if temp temp ,exp2)))

Are these problems?

- (let ((cons 5))
    (push ‘foo stack))

- (defvar temp 1000)
  (or (member x y) temp))

- (macrolet ((setq (flag) `(setf-flag ‘,flag)))
    (push ‘foo stack))

Two kinds of capture

- Macro argument capture
- Free symbol capture

Macro argument capture

- (defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))

- e.g., (for (limit 1 5) (princ limit)) =>
  (DO ((LIMIT 1 (1+ LIMIT))
       (LIMIT 5))
      ((> LIMIT LIMIT))
    (PRINC LIMIT))

Macro argument capture

- e.g.:
  (let ((limit 5))
    (for (i 1 10)
      (when (> i limit)
        (princ i))))

Free symbol capture

- (defvar w '())
  (defmacro gripe (warning)
    `(progn
       (setq w (nconc w (list ,warning)))
       nil))

- (defun sample-ratio (v w)
    (let ((vn (length v))
          (wn (length w)))
      (if (or (< nv 2) (< wn 2))
          (gripe "sample < 2")
          (/ vn wn))))

Free symbol capture

- (defun sample-ratio (v w)
  (let ((vn (length v))
  (wn (length w)))
  (if (or (< nv 2) (< wn 2))
  (PROGN (SETQ W (NCONC W ...))
  NIL)
  (/ vn wn))))

- e.g., (let ((list (list ‘b)))
  (sample-ratio nil list)
  list)

When does capture occur?

- Free: A symbol s occurs free in an expression when it is used as a variable in that expression, but the expression does not create a binding for it.

- e.g., (let ((x y) (z 10))
        (list w x z))

- e.g., (let ((x x)) x)

When does capture occur?

- Skeleton: The skeleton of a macro expansion is the whole expansion, minus anything which was part of an argument in the macro call.

- (defmacro foo (x y)
  `(/ (+ ,x 1) ,y))

- (foo (- 5 2) 6) => (/ (+ (- 5 2) 1) 6)

- skeleton: (/ (+ 1))

When does capture occur?

- Capturable: A symbol is capturable in some macro expansion if
  - (a) it occurs free in the skeleton of the macro expansion, or
  - (b) it is bound by a part of the skeleton in which arguments passed to the macro are either bound or evaluated.

Examples

- (defmacro cap1 () ‘(+ x 1))

- (defmacro cap2 (var)
  `(let ((x ...)
  (,var ...))
  ...))

- (defmacro cap3 (var)
  `(let ((x ...))
  (let ((,var ...)) ...)))

- (defmacro cap4 (var)
  `(let ((,var ...))
  (let ((x ...)) ...)))

Examples

- (defmacro safe1 (var)
  `(progn
  (let ((x 1)) (print x))
  (let ((,var 1)) (print ,var))))

- (defmacro cap5 (&body body)
  `(let ((x ...)) ,@body))

- (defmacro safe2 (expr)
  `(let ((x ,expr)) (cons x 1)))

- (defmacro safe3 (var &body body)
  `(let ((,var ...)) ,@body))

What is the case here?

- (defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))

Warning: This is vague!

- (defmacro let ((&rest bindings) &body body)
  `((lambda ,(mapcar #'first bindings) ,@body)
  ,@(mapcar #'second bindings)))

- (let ((x 1)) (list x))

- => This intentionally captures a variable!

Avoiding Capture

- Better names
- Prior evaluation
- Generated symbols
- Packages

Better Names

- (defvar *warnings*)

- (defmacro gripe (warning)
    `(progn
        (setq *warnings* (nconc *warnings* (list ,warning)))
        nil))

Prior Evaluation

- Wrong:
  (defmacro before (x y seq)
    `(let ((seq ,seq))
       (< (position ,x seq)
          (position ,y seq))))

- e.g. (before (progn (setq seq ‘(b a)) ‘a)
                ‘b ‘(a b))

Prior Evaluation

- Correct:

```
(defmacro before (x y seq)
    `(let ((xval ,x) (yval ,y) (seq ,seq))
        (< (position xval seq)
            (position yval seq)))))
```

Prior Evaluation

- Only applicable when...
  - ...all arguments are evaluated exactly once.
  - ...no arguments need to be evaluated in the scope of new bindings.

Prior Evaluation

- (defmacro for ((var start stop) &body body)
  `(do ((b (lambda (,var) ,@body))
        (count ,start (1+ count))
        (limit ,stop))
       ((> count limit))
     (funcall b count)))

- Beware: additional call and space overhead!

Unlikely Names

- (defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
  (xsf2jsh ,stop))
  ((> ,var xsf2jsh))
  ,@body))

- Not a solution because may still be accidentally captured.

Generated Symbols

- (defmacro for ((var start stop) &body body)
    (let ((gstop (gensym)))
      `(do ((,var ,start (1+ ,var))
            (,gstop ,stop))
           ((> ,var ,gstop))
         ,@body)))

Packages

- (in-package :my-iteration-macros)

```
(defmacro for ((var start stop) &body)
    `(do ((,var ,start (1+ ,var))
          (limit ,stop))
         ((> ,var limit))
       ,@body))
```