Macros

Pat Hanrahan

CS448h
Fall 2015

Macro Definitions in Lisp

Timothy Harris

Abstract

In LISP 1.5 special forms are used for three logically separate purposes: a) to reach the alist, b) to allow functions to have an indefinite number of arguments, and c) to keep arguments from being evaluated. New LISP interpreters can easily satisfy need (a) by making the alist a SPECIAL-type or APVAL-type entity. Uses (b) and (c) can be replaced by incorporating a MACRO instruction expander in define. I am proposing such an expander.

AI Memo 57, 1963

Relevance

Text macros are widely used (cpp)

Macros are evaluated at compile-time, not run-time (staged program)

Text macros are limited compared to lisp macros

Resurgence in interest in providing macros for programming languages (Terra, Rust, ...)

Last feature of lisp to not be widely used!

```
%cpp | gcc -E
#define NULL 0
#define SQUARE(x) x*x

SQUARE(1+2)

// results in
1+2*1+2

// defensive programming
#define SQUARE(x) ((x)*(x))

// cpp is not “aware” of C
```

```
// macro passed a string argument

// string operations?


#define STR(s) #s

#define CONCAT(a,b) a##b

// not complete set of string ops …
```

```
// conditional macros

// expressions?

#ifdef LINUX
…
#endif

// expressions in predicate?
#if defined(LINUX) …
#if VERSION > 1.0 …
```

```
# prep: use python mako templating engine
<%
n = 10
ns = range(10)
%>\
#include <stdio.h>

void main(void) {
  int a = ${n};
% for i in ns:
%   if i != 5:
  ${i};
%   endif
% endfor
}
```

Lisp Macros

```
# lisp/scheme s-expressions

% racket
> (+ 1 2)
3
> (* (+ 1 2) 4)
12
> (define x 5)
> (/ 10 x)
2
> (define (square x) (* x x))
> (square 5)
25
```

```
; special forms

; normally function arguments
; are evaluated left-to-right
; before the function is called

; sometimes function arguments
; are evaluated differently.
; these functions are special forms
(define x 2)
(if cond true-expr false-expr)
(or expr1 expr2)
(for [(i 10)] (displayln i))
```

```
; homoiconic: lists = code|data

> (define l (list 1 2 3))
'(1 2 3)
> (car l)
1
> (cdr l)
'(2 3)
> (cadr l)
2

; languages with free-form syntax
;   smalltalk, ruby ..
```

```
; quote

> (+ 1 2)
3
> (quote (+ 1 2))
'(+ 1 2)
> (list '+ 1 2)
'(+ 1 2)

; notation
> '(+ 1 2)
'(+ 1 2)
```

```
; code = (eval list)

> (eval '(+ 1 2 3))
6
> (eval '(if #t 1 0))
1
```

```
# meta-programming
#  implement (when pred expr)

> (define (convert whenlist)
   (list 'if (nth whenlist 1)
             (nth whenlist 2)
             (void)))
> (define s '(when (> 2 1)
              (display "true\n")))
> (convert s)
'(if (> 2 1) (display "true\n") #<void>)
> (eval (convert s))
true
```

```
# quasiquote

> (define x 2)
> (quasiquote (+ 1 x))
'(+ 1 x)
> (quasiquote (+ 1 (unquote x)))
'(+ 1 2)
> (quasiquote (+ 1 (unquote x)
                   (unquote-splicing '(2 2))))
'(+ 1 2 2 2)

; short-hand (note backquote `)
> `(+ 1 ,x ,@(list 2 2))
'(+ 1 2 2 2)

; should remind you of terra
```

```
# meta-programming
#  implement (when pred expr)

> (define (convert when)
   `(if ,(nth when 1)
        ,(nth when 2)
        ,(void)))
> (define s '(when (> 2 1)
                (display "true\n")))
> (convert s)
'(if (> 2 1) (display "true\n") #<void>)
> (eval (convert s))
true
```

```
# macros

> (define-macro (when test expr)
    `(if ,test ,expr ,(void)))

> (when (> 2 1) 1)
1

; 1. the arguments to the macro
;    are NOT evaluated (they are quoted)
; 2. The returned list is evaluated


; it’s that simple!
```

```
# macros

> (define-macro (or x y)
    `(if ,x ,x ,y))

> (or 1 2)
1
> (or #f 1)
1

; what’s wrong with this macro?
```

```
# macros

> (define-macro (or x y)
    `(let ((t ,x))
       (if t t ,y)))

> (or 1 2)
1
> (or #f 2)
2

; what's wrong with this macro?
```

```
; problem : variable capture

> (define t 2)
> (or #f t)
#f

; Variable capture "rarely" happens,
; but when it does,
; it creates insidious bugs

; sometimes you want variable capture
```

macros

```
> (define-macro (or x y)
    (let ((t (gensym)))
      `(let ((,t ,x))
         (if ,t ,t ,y))))

> (define t 2)
> (or #f 2)
2
```

Hygienic Macros

Approach: test whether there exists an variable in the environment with the same name. Generate unique name.

Replace argument list with "syntax objects = list + lexical environment)

define-syntax-rules

define-syntax-rule

```
> (define-syntax-rule (or x y)
    (let ((t x)) (if t t y)))

> (or 1 2)
1
> (or #f 2)
2
> (define t 2)
> (or #f t)
2
```

define-syntax
and
syntax-rules

syntax-case

```
> (define-syntax or
   (syntax-rules ()
    [(_) #f]
    [(_ e) e]
    [(_ e1 e2 e3 ...)
     (let ([t e1])
           (if t t (or e2 e3 …)))] ))


> (define t 2)
> (or #f t)
1
```

```
# define-syntax

> (define-syntax (or stx)
   (datum->syntax stx
    (let* ((l (syntax->datum stx))
           (x (list-ref l 1))
           (y (list-ref l 2)))
     `(let ((t ,x)) (if t t ,y)))))


> (define t 1)
> (or #f t)
1
```

Summary

History contains many cool ideas

- Mine the past!

Lisp macros

- Homoiconic language
- Macros act as syntax "transformers"
- Macros can be any lisp function
- Software that makes it easier to make DSLs