DEFINING MACROS

JARED GENTNER

JAN 29, 2021

(1) What is a macro?
(2) How did old macro definitions fail programmers
(3) Can declarative macros work with procedural code?
(4) What belongs in the pattern of a macro
(5) How can we leverage the macro expander itself?

Hart (1) 1963
Kohlbecker & Wand (2) 1987
Clinger et al (2) 1991
Dybvig (3) 1992
Culpepper & Fellersen (4) 2010
Flatt et al (5) 2012

ORIGIN

LISP automatically evaluates a function's arguments

eg. (+ (- 4 5) 6)

=> (+ -1 6)

=> 5

if/quote do not evaluate their args...

∴ they can't be functions

What are they? SPECIAL FORMS -

they receive args uneval'd

What if we could have...

USER defined SPECIAL FORMS?

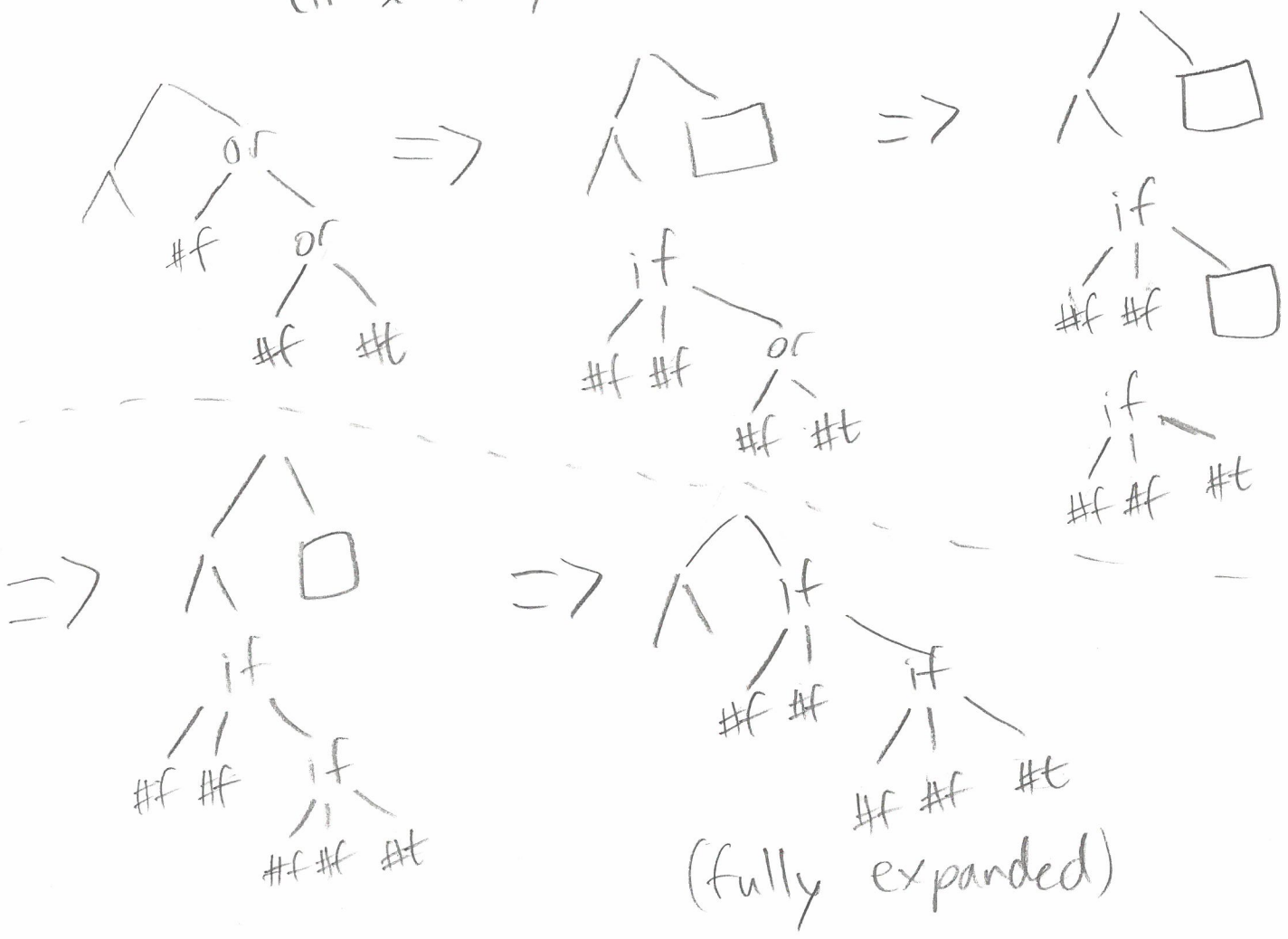
[Hart 63]

or, and, when, loop, cond, case, match...

[enter MACROS stage left.]

Also called "syntax-transformers",
they are functions from Syntax → Syntax

```
(define-simple-macro (or x y)
  (if x x y))
```



(fully expanded)

The EVALUATOR works inside-out
The EXPANDER works outside-in

A common example I will use is a let expanding into a lambda:

let $x_1 = a_1, \ldots, x_n = a_n$ in b

$\Longrightarrow$

$(\lambda x_1 \ldots x_n \cdot b)\ a_1 \ldots a_n$

or in javascript

```
{
  let x1 = a1;
  ...
  let xn = an;
  b
}
```

$\Longrightarrow$

$((x_1, \ldots, x_n) => b)\ (a_1, \ldots, a_n)$

LISP MACROS

```
(defmacro or stx
  (list 'if (second stx) (second stx) (third stx)))
```

```
(defmacro let stx
  (cons (list 'lambda (map first (second stx))
              (third stx))
        (map second (second stx))))
```

with special quote syntax

```
(defmacro let stx
  `((lambda ,(map first (second stx)) ,(third stx))
    ,@(map second (second stx))))
```

Given:

```
(defmacro whoops stx
  `(if ,(second stx) ,(first (third stx))
       ,(second (third stx))))
```

what does

```
(whoops x (5 9))
```

expand to?

Issues

- You have eyes, use 'em

- forgetting to unquote
- what patterns does the macro accept?

Comes down to BOILERPLATE.
MACROS eliminate BOILERPLATE.

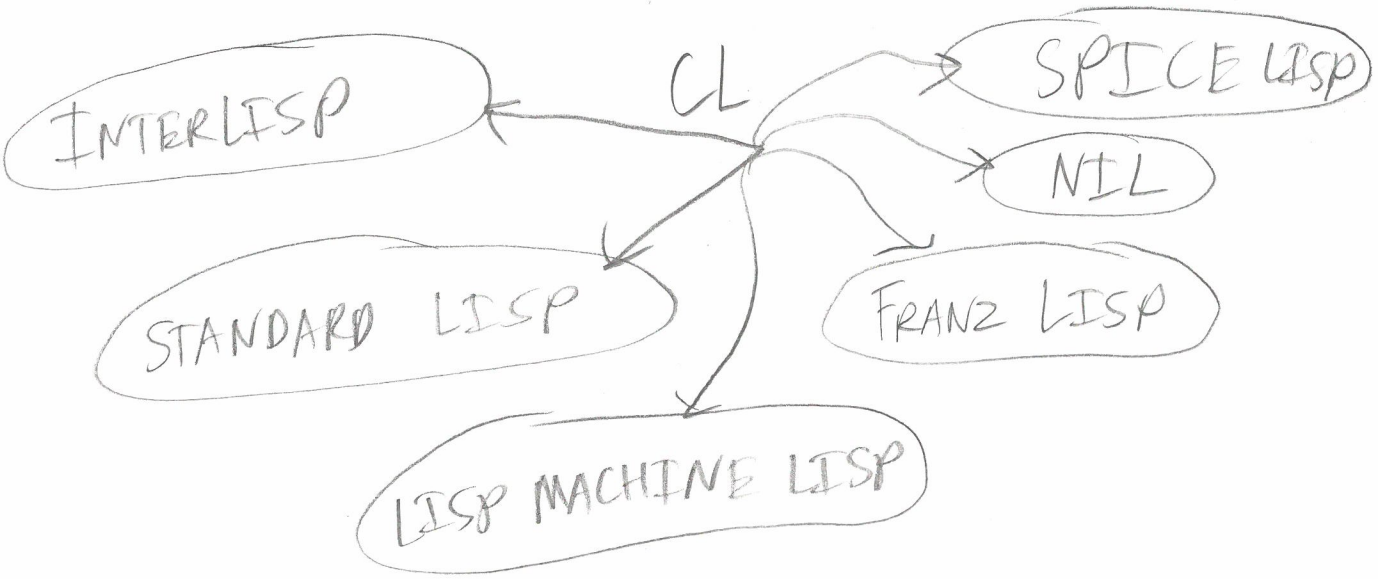
A Macro-Defining Macro

```
(extend-syntax let
  [(let ([x a] ...) b)
   ((λ(x ...) b) a ...)])
```

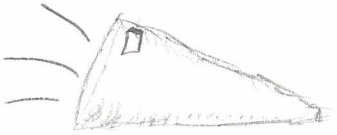
so-called "MACRO-BY-EXAMPLE"
[Kohlbecker & Wand 87]

A sigh of relief.

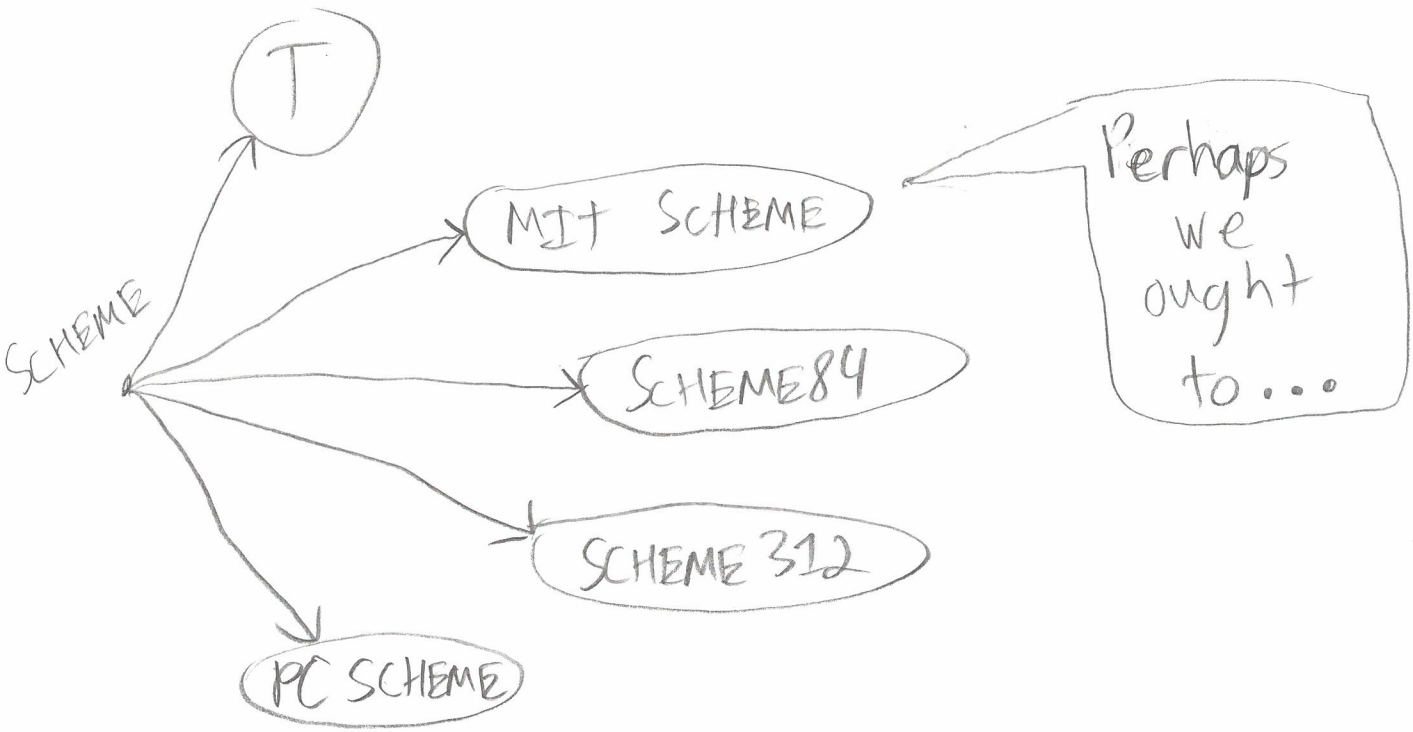
STANDARDIZING
SCHEME



"YOU MUST STANDARDIZE"



ARPA



No macros in R3RS [1986]

Hot-button issue

R4RS subcommittee for macros:

Dybvig + Hieb

Indiana

pro-Kohlbecker, enhancing extend-syntax

anti-Kohlbecker syntactic closures

MIT

Rees + Bawden

Clinger ends up working w/ Rees to produce **syntax-rules**

Dybvig and Hieb make **syntax-case**

syntax-rules:
- A version of extend-syntax without the 'with' expression
- purely declarative, only facility is pattern matching

syntax-case
- A more powerful version of extend-syntax
- Must be implemented in the compiler
- Can be used to define 'syntax-rules'

```
(define-syntax let
  (syntax-rules ()
    [(_ ([x a] ...) b)
     ((λ(x ...) b) a ...)]))
```

```
(define-syntax (let stx)
  (syntax-case stx ()
    [(_ ([x a] ...) b)
     #'((λ(x ...) b) a ...)]))
```

We'll get to it later,

but first...

```
(define-syntax grug
  (syntax-rules ()
    [(grug (acc ...)) (acc ...)]
    [(grug id more ... (acc ...) (grug more ... (id acc ...))]
    [(grug id more ...) (grug more ... (id))]))
```

What does this do?

```
(define-syntax (grug stx)
  (syntax-case stx ()
    [(grug id ...)
     (with-syntax ([(id* ...)
                    (reverse (syntax->list #'(id ...)))])
       #'(id* ...))]))
```

How about this?

Syntax Objects

#' = quote syntax

#` = quasiquote syntax

#, = unquote syntax

similar to `+, from LISP macros

Big Idea:

Is CODE DATA?

Syntax objects store:

- lexical information
  VERY IMPORTANT
- source locations
- hints for your editor of choice, DrRacket

Syntax objects do not store:

- your credit card number
- your bank account information
- your social security

BAD SYNTAX

Often we want to validate things about syntax.
eg. all bindings in let are identifiers.

```
(define-syntax (let stx)
  (unless (and (map identifier? (syntax-e #'(x ...)))
               (raise-syntax-error "non-identifier" stx))
    #'((λ (x ...) b) a ...)))
```

more specific:

```
(for ([id (syntax-e #'(x ...))])
  (unless (identifier? id)
    (raise-syntax-error "non-identifier" stx id)))
```

imagine additional check for duplicate bindings...

see Culpepper & Felleisen 2010

SYNTAX-PARSE

allows for describing syntax as a grammar

BNF:

<EXPR> ::= (let ([<id> <EXPR>] ...) <EXPR>)

```
syntax-parse:
 [(let ([x:id a:expr] ...) b:expr) ...]
```

defining custom classes, with attributes

```
(define-syntax-class binding
  #:description "binding pair"
  (pattern [var:id rhs]))
```

which can be used as attribute grammar-like rules!

```
[(let (bi:bind ...) b)
#'((λ (bi.var ...) b) bi.rhs ...)]
```

Syntax classes are also an abstraction. say we implemented the distinct-bindings class from Fortifying Macros.

```
(let ([a 42] [b 5]) (+ a b))
```

distinct-bindings

```
(for ([l (in-list '(1 2 3))] [i (in-naturals)])
  ...)
```

distinct-bindings

or perhaps

```
(define (f [x : Int]) ...)
```

typed-bind

```
(λ ([f : (-> Int Int)]) ...)
```

typed-bind

```
(λ ([f : (→ Int Int)] [x : Int]) ...)
```

distinct-typed-bindings??

The Expander

- Recall outside-in nature of expander
- As a result, macros can output new macro definitions

e.g.

```
(define-simple-macro (x name)
  (define-simple-macro (name n) (+ n 2)))
```

defines a macro called x which defines another macro.

```
(x test)
(test 4)
= 6
```

define-syntax doesn't have to bind an identifier to a macro.

These are called *compile-time*, or *transformer*, bindings

transformer bindings can be produced in between macro expansions

they contain all kinds of static info about an identifier.

ex: the struct macro in racket generates a struct type containing accessors, superstructs, etc.

another use is for custom macros

e.g.
```
(struct my-cool-lang-macro-rep [transform])
(define-simple-macro (define-my-cool-lang-macro n t)
  (define-syntax n (my-cool-lang-macro t)))
```

then expand in a loop

local-expand

- point of contention between Common Lispers and schemers

"where is your macroexpand?"

Here it is.

Typically the expander only runs on output

local-expand provides capability to expand subforms

give it a stop list to tell it when to... stop.

e.g.

```
(define-simple-macro (a n)
  (b n))
(define-simple-macro (b n)
  42)
(define-syntax test
  (syntax-parse
    [(_ n)
     (print (local-expand n 'expression (list #'b)))
     #'(void)]))
(test (a 42))
prints <syntax (b 42)>
```

Case-study: classes

Flatt et al. 2012

- classes have form

```
(class n (...)
  (public <method-name>)
  (define ...))
```

- reuses define!

- forms expand to define

- giving a class a local definition context

WHAT HAVE WE LEARNED?

- Macros are user defined special forms
- They work outside in, giving unique properties
- Macros can remove boilerplate that obscures the meaning of programs
- They routinely use this power on their own defining forms
- Declaratvity can be misleading
  It can also be highly rewarding