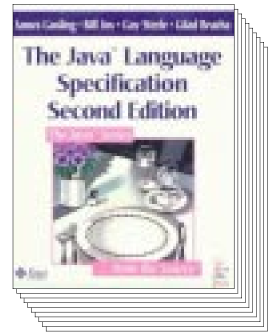
Scheme-Style Macros: Patterns and Lexical Scope

Matthew Flatt

University of Utah

Why Macros?

Language designers have to stop somewhere



(544 pages)

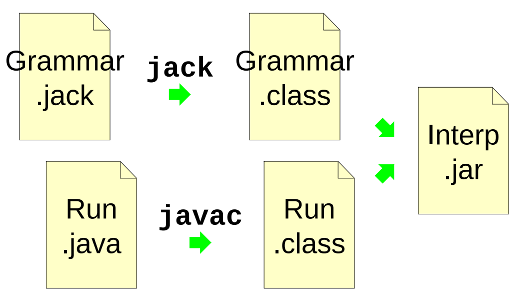
No language can provide every possible useful construct

Macros let a programmer fill in gaps

Macros versus Arbitrary Program Generators

Macros extend the language without extending the tool chain

Jack (YACC for Java) requires a new tool chain:

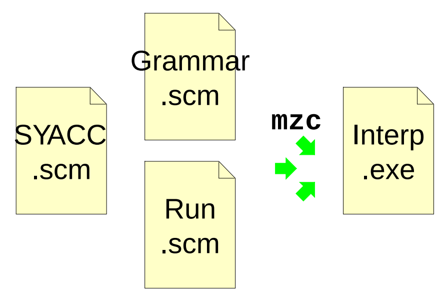


⇒Jack doesn't play nice with all Java environments

Macros versus Arbitrary Program Generators

Macros extend the language without extending the tool chain

Scheme-YACC is a macro:



⇒SYACC automatically plays nice with all Scheme environments

... in principle

Macros and Libraries

- Macros = hook in tool chain to extend a language

  Scheme ensures that macros play nice with the tool chain

- Some libraries include macros

  Scheme ensures that library macros play nice with each other

Pattern-Based Macros

Most popular API for macros: ***patterns***

```
#define swap(x, y) (tmp=y, y=x, x=tmp)

swap(c.red, d->blue)
⇒ (tmp=d->blue, d->blue=c.red, c.red=tmp)
```

\+ Relatively easy for programmers to understand

\+ Obvious hook into the tool chain

\- Pure patterns can't easily express much

...but possibly more than you think

Scheme Macro Basics

```
(define-syntax swap
   )
```

- **`define-syntax`** indicates a macro definition

Scheme Macro Basics

```
(define-syntax swap
  (syntax-rules ()
     ))
```

- **syntax-rules** means a pattern-matching macro
- () means no keywords in the patterns

Scheme Macro Basics

```
(define-syntax swap
  (syntax-rules ()
    (pattern template)
    ...
    (pattern template)))
```

- Any number of *pattern*s to match
- Produce result from *template* of first match

Scheme Macro Basics

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b)  )))
```

- Just one pattern for this macro: (swap a b)
- Each identifier matches anything in use

```
(swap x y)          ⇒ a is x
                      b is y

(swap 9 (+ 1 7))    ⇒ a is 9
                      b is (+ 1 7)
```

Scheme Macro Basics

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let ((tmp b))
                  (set! b a)
                  (set! a tmp)))))
```

- Bindings substituted into template to generate the result

```
(swap x y)         ⇒ (let ((tmp y))
                       (set! y x)
                       (set! x tmp))

(swap 9 (+ 1 7))   ⇒ (let ((tmp (+ 1 7)))
                       (set! (+ 1 7) 9)
                       (set! 9 tmp))
```

Lexical Scope

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let ((tmp b))
                  (set! b a)
                  (set! a tmp)))))
```

- What if we **swap** a variable named **tmp**?

```
(let ((tmp 5)          ?  (let ((tmp 5)
      (other 6))       ⇒        (other 6))
  (swap tmp other))         (let ((tmp other))
                              (set! other tmp)
                              (set! tmp tmp)))
```

Lexical Scope

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let ((tmp b))
                  (set! b a)
                  (set! a tmp)))))
```

- What if we **swap** a variable named **tmp**?

```
(let ((tmp 5)          ?  (let ((tmp 5)
      (other 6))       ⇒        (other 6))
  (swap tmp other))         (let ((tmp other))
                              (set! other tmp)
                              (set! tmp tmp)))
```

This expansion would violate lexical scope

Lexical Scope

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let ((tmp b))
                  (set! b a)
                  (set! a tmp)))))
```

- What if we **swap** a variable named **tmp**?

```
(let ((tmp 5)          ⇒ (let ((tmp 5)
      (other 6))               (other 6))
  (swap tmp other))        (let ((tmp_1 other))
                             (set! other tmp)
                             (set! tmp tmp_1)))
```

Scheme renames the introduced binding

Details later...

Lexical Scope: Local Bindings

Lexical scope means that local macros work, too:

```
(define (f x)
  (define-syntax swap-with-arg
    (syntax-rules ()
      ((swap-with-arg y) (swap x y))))

  (let ((z 12)
        (x 10))
    ; Swaps z with original x:
    (swap-with-arg z))
                              )
```

Details later...

Matching Sequences

Some macros need to match sequences

```
(rotate x y)

(rotate red green blue)

(rotate front-left
        rear-right
        front-right
        rear-left)
```

Matching Sequences

```
(define-syntax rotate
  (syntax-rules ()
    ((rotate a) (void))
    ((rotate a b c ...) (begin
                          (swap a b)
                          (rotate b c ...)))))
```

- `...` in a pattern: multiple of previous sub-pattern

  `(rotate x y z w)` ⇒ `c` is `z w`

- `...` in a template: multiple instances of previous sub-template

```
(rotate x y z w) ⇒ (begin
                     (swap x y)
                     (rotate y z w))
```

Matching Sequences

```
(define-syntax rotate
  (syntax-rules ()
    ((rotate a c ...)
     (shift-to (c ... a) (a c ...)))))

(define-syntax shift-to
  (syntax-rules ()
    ((shift-to (from0 from ...) (to0 to ...))
     (let ((tmp from0))
       (set! to from) ...
       (set! to0 tmp))    )))
```

- `...` maps over same-sized sequences
- `...` duplicates constants paired with sequences

Identifier Macros

The `swap` and `rotate` names work only in an "application" position

```
(swap x y)  ⇒ (let ((tmp y)) ...)
(+ swap 2)  ⇒ syntax error
```

An ***identifier macro*** works in any expression position

```
clock          ⇒ (get-clock)
(+ clock 10)   ⇒ (+ (get-clock) 10)
(clock 5)      ⇒ ((get-clock) 5)
```

...or as a `set!` target

```
(set! clock 10)  ⇒ (set-clock! 10)
```

Identifier Macros

```
(define-syntax clock
  (syntax-id-rules (set!)
    ((set! clock e) (put-clock! e))
    ((clock a ...) ((get-clock) a ...))
    (clock (get-clock))))
```

- **`set!`** is designated as a keyword
- **`syntax-rules`** is a special case of **`syntax-id-rules`** with errors in the first and third cases

Macro-Generating Macros

If we have many identifiers like **`clock`**...

```
(define-syntax define-get/put-id
  (syntax-rules ()
    ((define-get/put-id id get put!)
     (define-syntax id
       (syntax-id-rules (set!)
         ((set! id e) (put! e))
         ((id a (... ...))
          ((get) a (... ...)))
         (id (get)))))))
```

```
(define-get/put-id clock get-clock put-clock!)
```

- `(... ...)` in a template gets replaced by `...`

Extended Example

Let's add call-by-reference definitions to Scheme

```
(define-cbr (f a b)
  (swap a b))

(let ((x 1) (y 2))
  (f x y)
  x)
; should produce 2
```

Extended Example

Expansion of first half:

```
(define-cbr (f a b)
  (swap a b))

⇒
(define (do-f get-a get-b put-a! put-b!)
  (define-get/put-id a get-a put-a!)
  (define-get/put-id b get-b put-b!)
  (swap a b))
```

Extended Example

Expansion of second half:

```
(let ((x 1) (y 2))
  (f x y)
  x)

⇒
(let ((x 1) (y 2))
  (do-f (lambda () x)
        (lambda () y)
        (lambda (v) (set! x v))
        (lambda (v) (set! y v)))
  x)
```

Call-by-Reference Setup

How the first half triggers the second half:

```
(define-syntax define-cbr
  (syntax-rules ()
    ((_ (id arg ...) body)
     (begin
       (define-for-cbr do-f (arg ...)
         () body)
       (define-syntax id
         (syntax-rules ()
           ((id actual (... ...))
            (do-f (lambda () actual)
                  (... ...)
                  (lambda (v)
                    (set! actual v))
                  (... ...)))))))))
```

Call-by-Reference Body

Remaining expansion to define:

```
(define-for-cbr do-f (a b)
  () (swap a b))

⇒
(define (do-f get-a get-b put-a! put-b!)
  (define-get/put-id a get-a put-a!)
  (define-get/put-id b get-b put-b!)
  (swap a b))
```

How can **define-for-cbr** make **get-** and **put-!** names?

Call-by-Reference Body

A name-generation trick:

```
(define-syntax define-for-cbr
  (syntax-rules ()
    ((define-for-cbr do-f (id0 id ...)
       (gens ...) body)
     (define-for-cbr do-f (id ...)
       (gens ... (id0 get put)) body))
    ((define-for-cbr do-f ()
       ((id get put) ...) body)
     (define (do-f get ... put ...)
       (define-get/put-id id get put) ...
       body))))
```

Call-by-Reference Body

More accurate description of the expansion:

```
(define-for-cbr do-f (a b)
  () (swap a b))

⇒
(define (do-f get^1 get^2 put^1 put^2)
  (define-get/put-id a get^1 put^1)
  (define-get/put-id b get^2 put^2)
  (swap a b))
```

Complete Code to Add Call-By-Reference

```
(define-syntax define-cbr
  (syntax-rules ()
    ((_ (id arg ...) body)
     (begin
       (define-for-cbr do-f (arg ...)
         () body)
       (define-syntax id
         (syntax-rules ()
           ((id actual (... ...))
            (do-f (lambda () actual)
                  (... ...)
                  (lambda (v)
                    (set! actual v))
                  (... ...))))))))

(define-syntax define-get/put-id
  (syntax-rules ()
    ((define-get/put-id id get put!)
     (define-syntax id
       (syntax-id-rules (set!)
         ((set! id e) (put! e))
         ((id a (... ...))
          ((get) a (... ...)))
         (id (get)))))))

(define-syntax define-for-cbr
  (syntax-rules ()
    ((define-for-cbr do-f (id0 id ...)
       (gens ...) body)
     (define-for-cbr do-f (id ...)
       (gens ... (id0 get put)) body))
    ((define-for-cbr do-f ()
       ((id get put) ...) body)
     (define (do-f get ... put ...)
       (define-get/put-id id get put) ...
       body))))
```

Relies on lexical scope and macro-generating macros

- **Macros In General**
- **Pattern-Based Macros**
- **Extended Example**
- **Lexical Scope**
  - Making it work
- **General Transformers**
- **State of the Art**

Lexical Scope

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let ((tmp b))
                  (set! b a)
                  (set! a tmp)))))
```

- What if we **swap** a variable named **tmp**?

```
(let ((tmp 5)           ⇒ (let ((tmp 5)
      (other 6))                (other 6))
  (swap tmp other))        (let ((tmp₁ other))
                             (set! other tmp)
                             (set! tmp tmp₁)))
```

Scheme renames the introduced binding

Reminder: Lexical Scope for Functions

```
(define (app-it f)
  (let ((x 12))
    (f x)))

(let ((x 10))
  (app-it (lambda (y) (+ y x))))

→
```

Reminder: Lexical Scope for Functions

```
(define (app-it f)
  (let ((x 12))
    (f x)))

(let ((x 10))
  (app-it (lambda (y) (+ y x))))

↛
(let ((x 10))
  (let ((x 12))
    ((lambda (y) (+ y x)) x)))
```

Bad capture

Reminder: Lexical Scope for Functions

```
(define (app-it f)
  (let ((x 12))
    (f x)))

(let ((x 10))
  (app-it (lambda (y) (+ y x))))

→
(let ((x 10))
  (let ((z 12))
    ((lambda (y) (+ y x)) z)))
```

Ok with α-rename inside **app-it**

Reminder: Lexical Scope for Functions

```
(define (app-it f)
  (let ((x 12))
    (f x)))

(let ((x 10))
  (app-it (lambda (y) (+ y x))))

→(let ((x 10))
  (let ((z 12))
    ((lambda (y) (+ y x)) z)))
```

Ok with α-rename inside **app-it**

But usual strategy must see the binding...

Bindings in Templates

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let ((tmp b))
                  (set! b a)
                  (set! a tmp)))))
```

Seems obvious that **tmp** can be renamed

Bindings in Templates

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let-one (tmp b)
                  (set! b a)
                  (set! a tmp)) )))
```

- Rename **tmp** if

```
(define-syntax let-one
  (syntax-rules ()
    ((let-one (x v) body)
     (let ((x v)) body))))
```

Bindings in Templates

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let-one (tmp b)
                  (set! b a)
                  (set! a tmp)) )))
```

- *Cannot* rename **tmp** if

```
(define-syntax let-one
  (syntax-rules ()
    ((let-one (x v) body)
     (list 'x v body))))
```

Scheme tracks identifier introductions, then renames only as binding forms are discovered

Lexical Scope via Tracking, Roughly

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let ((tmp b))
                  (set! b a)
                  (set! a tmp)))))
```

- Tracking avoids capture by introduced variables

```
(let ((tmp 5)                ~> (let ((tmp 5)
      (other 6))             =>       (other 6))
  (swap tmp other))               (let^1 ((tmp^1 other))
                                    (set!^1 other tmp)
                                    (set!^1 tmp tmp^1)))
```

^{1} means introduced by expansion

`tmp`^{1} does not capture `tmp`

Lexical Scope via Tracking, Roughly

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let ((tmp b))
                  (set! b a)
                  (set! a tmp)))))
```

- Tracking also avoids capture *of* introduced variables

```
(let ((set! 5)               ~> (let ((set! 5)
      (let 6))               =>       (let 6))
  (swap set! let))                (let^1 ((tmp^1 let))
                                    (set!^1 let set!)
                                    (set!^1 set! tmp^1)))
```

`set!` does not capture `set!`^{1}

`let` does not capture `let`^{1}

Precise Rules for Expansion and Binding

```
(let ((tmp 5)
      (other 6))
  (swap tmp other))
```

Precise Rules for Expansion and Binding

```
(let ((tmp 5)             => (let ((tmp_0 5)
      (other 6))                   (other_0 6))
  (swap tmp other))            (swap tmp_0 other_0))
```

When the expander encounters **`let`**, it renames bindings by adding a subscript

Precise Rules for Expansion and Binding

```
(let ((tmp 5)          ⇒ (let ((tmp_0 5)
      (other 6))               (other_0 6))
  (swap tmp other))      (swap tmp_0 other_0))
```

When the expander encounters **let**, it renames bindings by adding a subscript

If a use turns out to be **quote**d, the subscript will be erased

```
(let ((x 1))  ⇒ (let ((x_1 1))  ⇒ (let ((x_1 1))
  'x)              'x_1)              'x)
```

Precise Rules for Expansion and Binding

```
(let ((tmp 5)          ⇒ (let ((tmp_0 5)
      (other 6))               (other_0 6))
  (swap tmp other))      (swap tmp_0 other_0))

⇒ (let ((tmp_0 5)
        (other_0 6))
    (let^1 ((tmp^1 other_0))
      (set!^1 other_0 tmp_0)
      (set!^1 tmp_0 tmp^1)))
```

Then expansion continues, adding superscripts for introduced identifiers

Precise Rules for Expansion and Binding

```
(let ((tmp 5)          ⇒ (let ((tmp_0 5)
      (other 6))               (other_0 6))
  (swap tmp other))      (swap tmp_0 other_0))

⇒ (let ((tmp_0 5)              ⇒ (let ((tmp_0 5)
        (other_0 6))                   (other_0 6))
    (let^1 ((tmp^1 other_0))       (let^1 ((tmp_2 other_0))
      (set!^1 other_0 tmp_0)         (set!^1 other_0 tmp_0)
      (set!^1 tmp_0 tmp^1)))         (set!^1 tmp_0 tmp_2)))
```

Again, rename for **let** —but only where superscripts match

Precise Rules for Expansion and Binding

```
(let ((set! 5)
      (let 6))
  (swap set! let))
```

Precise Rules for Expansion and Binding

```
(let ((set! 5)         ⇒ (let ((set!_0 5)
      (let 6))                  (let_0 6))
  (swap set! let))         (swap set!_0 let_0))
```

Precise Rules for Expansion and Binding

```
(let ((set! 5)         ⇒ (let ((set!_0 5)
      (let 6))                  (let_0 6))
  (swap set! let))         (swap set!_0 let_0))

⇒ (let ((set!_0 5)
         (let_0 6))
     (let^1 ((tmp^1 let_0))
       (set!^1 let_0 set!_0)
       (set!^1 set!_0 tmp^1)))
```

Precise Rules for Expansion and Binding

```
(let ((set! 5)         ⇒ (let ((set!_0 5)
      (let 6))                  (let_0 6))
  (swap set! let))         (swap set!_0 let_0))

⇒ (let ((set!_0 5)              ⇒ (let ((set!_0 5)
         (let_0 6))                      (let_0 6))
     (let^1 ((tmp^1 let_0))          (let^1 ((tmp_2 let_0))
       (set!^1 let_0 set!_0)           (set!^1 let_0 set!_0)
       (set!^1 set!_0 tmp^1)))         (set!^1 set!_0 tmp_2)))
```

Superscript does not count as a rename, so **`let`** and **`let^1`** refer to the usual **`let`**

Local Macros

```
(define (run-clock get put!)
  (define-get/put-id clock get put!)
  (set! clock (add1 clock)))

⇒⇒
(define (run-clock get_0 put!_0)
  (define-get/put-id clock_1 get_0 put!_0)
  (set! clock_1 (add1 clock_1)))

⇒⇒
(define (run-clock get_0 put!_0)
  (define-get/put-id clock_1 get_0 put!_0)
  (put_0^2 (add1 (get_0^3))))
```

Local Macros

(define (run-clock get put!)
  (define-get/put-id clock get put!)
    (let ((get))
      (set! clock (get clock)))
)

⇒
(define (run-clock get_0 $put!_0$)
  (define-get/put-id $clock_1$ get_0 $put!_0$)
    (let ((get_0))
      (set! $clock_1$ (get_0 $clock_1$)))
)

Local Macros

(define (run-clock get_0 $put!_0$)
  (define-get/put-id $clock_1$ get_0 $put!_0$)
    (let ((get_0))
      (set! $clock_1$ (get_0 $clock_1$)))
)

⇒
(define (run-clock get_0 $put!_0$)
  (define-get/put-id $clock_1$ get_0 $put!_0$)
    (let ((get_2))
      (set! $clock_1$ (get_2 $clock_1$)))
)

Local Macros

(define (run-clock get_0 $put!_0$)
  (define-get/put-id $clock_1$ get_0 $put!_0$)
    (let ((get_2))
      (set! $clock_1$ (get_2 $clock_1$)))
)

⇒⇒
(define (run-clock get_0 $put!_0$)
  (define-get/put-id $clock_1$ get_0 $put!_0$)
    (let ((get_2))
      ($put!_0^3$ (get_2 (get_0^4))))
)

General Strategy Summarized

While expanding

- Primitive binding form:
  - Change subscript in scope for matching names, subscript, and superscripts
- When looking for binders of a use:
  - Check for matching name and subscript, only
- After expanding a macro use:
  - Add a superscript to introduced identifiers

(macro-generating macros can stack superscripts)

Terminology

Avoid capture *by* introduced: ***hygiene***

Avoid capture *of* introduced: ***referential transparency***

Together ⇒***lexical scope***

Lexically scoped macros play nice together

- **Macros In General**
- **Pattern-Based Macros**
- **Extended Example**
- **Lexical Scope**
- **General Transformers**
  - Beyond patterns and templates
- **State of the Art**

Transformer Definitions

In general, **`define-syntax`** binds a transformer procedure

```
(define-syntax swap
  (lambda (stx)
     ))
```

Argument to transformer is a ***syntax object***: like an S-expression, but with context info

---

Primitives for Transformers

Primitives deconstruct and construct syntax objects:

```
(stx-car stx) -> stx
(stx-cdr stx) -> stx
(stx-pair? stx) -> bool
(identifier? stx) -> bool

(quote-syntax datum) -> stx

(bound-identifier=? stx1 stx2) -> bool
(free-identifier=? stx1 stx2) -> bool
(datum->syntax-object stx v) -> stx
```

Syntax-Rules as a Transformer

`syntax-rules` is actually a macro

```
(define-syntax swap
  (syntax-rules .....))
```

⇒

```
(define-syntax swap
  (lambda (stx)
     use transformer primitives to
     match stx and generate result
                                       ))
```

Pattern-Matching Syntax and Having It, Too

The **`syntax-case`** and **`#'`** forms combine patterns and arbitrary computation

```
(syntax-case stx-expr ()
  (pattern result-expr)
  ...
  (pattern result-expr))
```

```
#'template
```

`syntax-case` and **`#'`** work anywhere

- useful for sub-expression matches

Pattern-Matching Syntax and Having It, Too

Actually, **`syntax-rules`** is implemented in terms of **`syntax-case`**

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b) (let ((tmp b))
                  (set! b a)
                  (set! a tmp)))))
```

⇒

```
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      ((swap_1 a b) #'(let ((tmp b))
                        (set! b a)
                        (set! a tmp)))))
```

Syntax-Case for a Better Swap Macro

Check for identifiers before expanding:

```
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      ((_ a b)
       (if (and (identifier? #'a)
                (identifier? #'b))
           #'(let ((tmp b))
               (set! b a)
               (set! a tmp))
           (raise-syntax-error
            'swap "needs identifiers"
            stx))))))
```

Syntax-case for a Better Call-by-Ref Macro

Use **generate-temporaries** to produce a list ids:

```
(define-syntax (define-for-cbr stx)
  (syntax-case stx ()
    ((_ id (arg ...) body)
     (with-syntax (((get ...)
                    (generate-temporaries #'(arg ...)))
                   ((put ...)
                    (generate-temporaries #'(arg ...))))
       #'(define (do-f get ... put ...)
           (define-get/put-id id get put) ...
           body)))))
```


Scheme Today

- Standard Scheme (R5RS) provides only **syntax-rules**
- Most implementations also provide **syntax-case**
  - Public expander implementation in R5RS
  - Syntax-object primitives vary
  - Separation of compile-time and run-time code varies greatly
- Some implementations support identifier macros

Code in these slides is somewhat specific to PLT Scheme...

Slide Language

... actually, it's PLT Scheme plus

```
(define-syntax syntax-id-rules
  (syntax-rules ()
    ((_ kws (pat tmpl) ...)
     (make-set!-transformer
      (lambda (stx)
        (syntax-case stx kws
          (pat #'tmpl)
          ...))))))
```

in a **module** loaded with **require-for-syntax**

Scheme in the Future

There's no one Scheme

or

Scheme is a langauge for defining practical languages

- Standardized language-declaration syntax may be the way to tame implementation differences
- In DrX, we intend to push the limits of these ideas

References, Abridged

hygiene — Kohlbecker, Friedman, Felleisen, and Duba
"Hygienic Macro Expansion"
LFP 1986

patterns — Clinger and Rees
"Macros That Work"
POPL 1991

lexical scope — Dybvig, Hieb, and Bruggeman
"Syntactic Abstraction in Scheme"
Lisp and Symbolic Computation 1993

splicing scope — Waddell and Dybvig
"Extending the Scope of Syntactic Abstraction"
POPL 1999

phases — Flatt
"Composable and Compilable Macros"
ICFP 2002