

**The**  
**Guaranteed Optimization**  
**Clause**  
  
**of the**  
**Macro-Writer's Bill of Rights**

# Macro-Writer's Bill of Rights

## A macro system must

- provide a powerful pattern language
- allow arbitrary transformations
- unify high- and low-level macros
- respect lexical scoping
- permit controlled capture
- support local macros
- support modular use of macros
- correlate source and object code
- guarantee certain optimizations

# KFFD plus extend-syntax

KFFD\* plus extend-syntax confers some rights

- provide a powerful pattern language
- allow arbitrary transformations
- unify high- and low-level macros
- respect lexical scoping

\*KFFD = Kohlbecker, Friedman, Felleisen, Duba  
Hygienic expansion algorithm

# syntax-rules

R5RS macros (syntax-rules): a different set

- provide a powerful pattern language
- respect lexical scoping
- support local macros\*

\*but no internal define-syntax

# syntax-case

syntax-case: all but guaranteed optimization

- provide a powerful pattern language
- allow arbitrary transformations
- unify high- and low-level macros
- respect lexical scoping
- permit controlled capture
- support local macros
- support modular use of macros
- correlate source and object code

# syntax-case + “cp0”

syntax-case + cp0: all incl. guaranteed optimization

What is cp0?

- Chez Scheme “compiler pass 0”
- also employed by Petite Chez Scheme’s interpreter
- performs various source optimizations
- described in SAS ’97, Waddell dissertation
- several improvements since

# Guaranteed Optimization

Reduce abstraction overhead

⇒ encourage use of syntactic abstraction

Eliminate incentive for using macros for optimization

⇒ discourage abuse of syntactic abstraction

Obviate special-case macro code

⇒ keep macro-writer focus on general case

# Example: R5RS Letrec

```
(letrec ([ $x_1$   $e_1$ ] ... [ $x_n$   $e_n$ ]) body)
```

- binds  $x_1 \dots x_n$  to new locations
- evaluates  $e_1 \dots e_n$  in the scope of these bindings
- assigns  $x_1 \dots x_n$  to the resulting values
- evaluates *body* in the scope of the new bindings

# R5RS syntax-rules transformation

```
(define-syntax letrec
  (syntax-rules ()
    [(_ ([x e] ...) b1 b2 ...)
     (L (x ...) () ((x e) ...) b1 b2 ...) ]))
```

```
(define-syntax L
  (syntax-rules ()
    [(_ () (t ...) ((x e) ...) b1 b2 ...)
     (let ([x #f] ...)
       (let ([t e] ...)
         (set! x t) ...
         (let () b1 b2 ...)))]
    [(_ (x1 x2 ...) (t0 ...) ((x e) ...) b1 b2 ...)
     (L (x2 ...) (t t0 ...) ((x e) ...) b1 b2 ...) ]))
```

# R5RS syntax-rules transformation

```
(define-syntax letrec
  (syntax-rules ()
    [(_ ([x e] ...) b1 b2 ...)
     (L (x ...) () ((x e) ...) b1 b2 ...) ]))
```

```
(define-syntax L
  (syntax-rules ()
    [(_ () (t ...) ((x e) ...) b1 b2 ...)
     (let ([x #f] ...)
       (let ([t e] ...)
         (set! x t) ...
         (let () b1 b2 ...))) ]
    [(_ (x1 x2 ...) (t0 ...) ((x e) ...) b1 b2 ...)
     (L (x2 ...) (t t0 ...) ((x e) ...) b1 b2 ...) ]))
```

# syntax-case transformation

```
(define-syntax letrec
  (lambda (x)
    (syntax-case x ()
      [(_ ([x e] ...) b1 b2 ...)
       (with-syntax ([t ...] (gen-temps #'(x ...)))]
         (let ([x #f] ...)
           (let ([t e] ...)
             (set! x t) ...
             (let () b1 b2 ...)))))))))
```

# Expansion

```
(expand '(letrec ([f (lambda (x) (g x))]
                  [g (lambda (x) (f x))])
        (f (g 0))))
```

```
→ (let ([f #f] [g #f])
      (let ([t1 (lambda (x) (g x))]
            [t2 (lambda (x) (f x))])
        (set! f t1)
        (set! g t2)
        (let () (f (g 0))))))
```

# Petrofsky's transformation

```
(define-syntax letrec
  (syntax-rules ()
    [(_ ((x e) ...) b1 b2 ...)
      (let ([x #f] ...)
        (let ([x (let ([t e]
                      (lambda () (set! x t)))]
              ...
              [body (lambda () b1 b2 ...)])
          (x) ...
          (body) ) ) ]))
```

# Petrofsky's transformation

```
(expand ' (letrec ([f (lambda (x) (g x))]
                  [g (lambda (x) (f x))])
        (f (g 0))))
```

```
→ (let ([f #f] [g #f])
      (let ([f1 (let ([t1 (lambda (x) (g x))])
                  (lambda () (set! f t1)))]
            [g1 (let ([t2 (lambda (x) (f x))])
                  (lambda () (set! g t2)))]
            [body (lambda () (f (g 0)))]])
        (f1)
        (g1)
        (body)))
```

# Petrofsky, After Optimization

```
(optimize (expand ' (letrec ([f (lambda (x) (g x))]
                             [g (lambda (x) (f x))])
          (f (g 0))))))
```

```
→ (let ([f #f] [g #f])
      (let ([t1 (lambda (x) (g x))]
            [t2 (lambda (x) (f x))])
        (set! f t1)
        (set! g t2)
        (f (g 0))))
```

# Guaranteed Optimizations

A macro programmer can count on:

- constant folding
- copy propagation
- procedure inlining “when appropriate”
- dead and useless code elimination
- certain degenerate-case optimizations

# Constant Folding

Fold only deterministic, effect-free primitive calls

Cannot fold mutable-object constructors

Must preserve sharing/nonsharing, pointer equality

`(+ 3 4) → 7`

`(= 3 4) → #f`

`(+ 3 (- 6 2)) → 7`

`(car '(a b c)) → a`

`(memq 'b '(a b c)) → '(b c)`

`(random 10) ↛ 6`

`(list 1 2 3) ↛ '(1 2 3)`

`(remove 'b '(a b c)) ↛ '(a c)`

# Copy Propagation

`(let ([x 3]) (+ x x))` → `(let ([x 3]) (+ 3 3))`

`(let ([x e1]  
 (let ([y x])  
 (+ y x))))` → `(let ([x e1]  
 (let ([y x])  
 (+ x x))))`

`(let ([x e1]  
 (let ([y x])  
 (define z  
 (lambda () y))  
 (+ (z) x))))` → `(let ([x e1]  
 (let ([y x])  
 (define z  
 (lambda () x))  
 (+ (z) x))))`

`(let ([x '(a b c)])  
 (car x))` → `(let ([x '(a b c)])  
 (car '(a b c)))`

# Copy Propagation (cont.)

```
(let ([x e1])  
  (let ([y x])  
    (set! x e2)  
    (+ y x)))  
  ↗  
(let ([x e1])  
  (set! x e2)  
  (+ x x))
```

```
(let ([x e1])  
  (let ([y x])  
    (define z  
      (lambda () y))  
    (set! x e2)  
    (+ (z) x)))  
  ↗  
(let ([x e1])  
  (define z  
    (lambda () x))  
  (set! x e2)  
  (+ (z) x))
```

# Procedure Inlining

```
(let ()
  (define min
    (lambda (x y)
      (if (< x y) x y)))
  (min e1 e2))

→ (let ()
    (define min
      (lambda (x y)
        (if (< x y) x y)))
    ((lambda (x y)
       (if (< x y) x y))
     e1 e2))

→ (let ()
    (define min
      (lambda (x y)
        (if (< x y) x y)))
    (let ([x e1] [y e2])
      (if (< x y) x y)))
```

# Dead and useless code elimination

```
(if #t x (+ x 1)) → x
```

```
(begin (car ' (a b c)) #f) → #f
```

```
(let ((x 3)) 6) → 6
```

```
(let ()
```

```
  (define min
```

```
    (lambda (x y)
```

```
      (if (< x y) x y)))
```

```
(let ([x e1])
```

```
  (if (< x 0) x 0)))
```

```
→ (let ([x e1])
```

```
    (if (< x 0) x 0))
```

# Degenerate-case optimizations

`(let ([x e]) x) → e`

`(eq? x x) → #t`

`(list) → '()`

`(list* e) → e`

`(memq e '()) → (begin e #f)`

⋮

# Impact

A macro programmer can freely ...

- introduce let-bindings to avoid possible duplicate evaluation
- introduce lambda abstractions to avoid code duplication
- ignore special cases involving constants
- ignore degenerate cases resulting in dead or useless code

... and count on the compiler to clean it all up

# Petrofsky expansion

```
(let ([f #f] [g #f])
  (let ([f1 (let ([t1 (lambda (x) (g x))])
              (lambda () (set! f t1)))]
        [g1 (let ([t2 (lambda (x) (f x))])
              (lambda () (set! g t2)))]
        [body (lambda () (f (g 0)))]])
    (f1)
    (g1)
    (body) ) )
```

# Petrofsky expansion

```
(let ([f #f] [g #f])
  (let ([t1 (lambda (x) (g x))]
        [t2 (lambda (x) (f x))])
    (let ([f1 (lambda () (set! f t1))]
          [g1 (lambda () (set! g t2))]
          [body (lambda () (f (g 0)))]))
      (f1)
      (g1)
      (body)))
```

# Petrofsky expansion

```
(let ([f #f] [g #f])
  (let ([t1 (lambda (x) (g x))]
        [t2 (lambda (x) (f x))])
    (let ([f1 (lambda () (set! f t1))]
          [g1 (lambda () (set! g t2))]
          [body (lambda () (f (g 0)))]))
      (let () (set! f t1))
      (let () (set! g t2))
      (let () (f (g 0))))))
```

# Petrofsky expansion

```
(let ([f #f] [g #f])
  (let ([t1 (lambda (x) (g x))]
        [t2 (lambda (x) (f x))])
    (set! f t1)
    (set! g t2)
    (f (g 0))))
```

# Assembler

Provides two syntactic forms:

```
(emit opname arg ...)  
(reg regname)
```

```
(define test  
  (lambda (r)  
    (emit ld (reg r0) (reg r1) (reg r2))  
    (emit addi (reg r2) 320 (reg r2))  
    (emit add (reg r2) r (reg r2))))
```

# Assembler

Provides two syntactic forms:

```
(emit opname arg ...)  
(reg regname)
```

```
(define test  
  (lambda (r)  
    (emit ld (reg r0) (reg r1) (reg r2))  
    (emit addi (reg r2) 320 (reg r2))  
    (emit add (reg r2) r (reg r2))))
```

**Highlighted** instructions are constant

Parts of other instruction also constant

# So do we write ...

```
(define-syntax emit
  (lambda (x)
    (syntax-case x ()
      [(_ op (reg s1) (reg s2) (reg d))
       (with-syntax ([inst (+ (ash (op->code (syntax-object->datum x))
                                   (ash (reg->code (syntax-object->datum s1))
                                       (ash (reg->code (syntax-object->datum s2))
                                           (ash (reg->code (syntax-object->datum d))))
                                   #'(emit-word! inst)))]
         [(_ op (reg s1) n (reg d))
          (integer? (syntax-object->datum #'n))
          (with-syntax ([inst (+ (ash (op->code (syntax-object->datum x))
                                   (ash (reg->code (syntax-object->datum s1))
                                       (ash (syntax-object->datum #'n))
                                           (ash (reg->code (syntax-object->datum d))))
                                   #'(emit-word! inst)))]
            [(_ op (reg s1) (reg s2) expr)
```

# ... or do we write

```
(define-syntax emit
  (syntax-rules ()
    [(_ op a1 a2 a3)
     ($emit! 'op a1 a2 a3)]))
```

and let the optimizer do its job?

```
(define opcode-pos 27)
(define src1-pos 22)
(define src2-pos 0)
(define dst-pos 17)
(define imm-bit (ash 1 16))

(define regops '(ld . #b10110) (add . #b11100))
(define immops '(addi . #b11100))
(define regcodes '(r0 . 0) (r1 . 1) (r2 . 2))

(define-syntax reg
  (syntax-rules ()
    [(_ r) (cdr (assq 'r regcodes))]))
```

```

(define $emit!
  (lambda (op a1 a2 a3)
    (emit-word!
      (+ (cond
          [(assq op regops) =>
           (lambda (a) (ash (cdr a) opcode-pos))]
          [(assq op immops) =>
           (lambda (a)
             (+ (ash (cdr a) opcode-pos) imm-bit))]
          [else
           (error 'emit "invalid operator" op)])
        (ash a1 src1-pos)
        (ash a2 src2-pos)
        (ash a3 dst-pos))))))

(define-syntax emit
  (syntax-rules ()
    [(_ op a1 a2 a3)
     ($emit! 'op a1 a2 a3)]))

```

# After optimization

```
(define test
  (lambda (r)
    (emit ld (reg r0) (reg r1) (reg r2))
    (emit addi (reg r2) 320 (reg r2))
    (emit add (reg r2) r (reg r2))))
```

```
→ (define test
    (lambda (r)
      (emit-word! 2953052161)
      (emit-word! 3766812992)
      (emit-word! (#3%+ 3766747136 r))))
```

# Inlining Caveat

## Inlining guarantee is a bit weak

- always when single call site
- always for “trivial” procedures
- otherwise depends on
  - size of residual code
  - compile-time inlining cost

# Other Caveats

Propagation/inlining take place only within a single top-level expression

⇒ wrap program in `let` or `module`

⇒ use `include` to combine files

Constant folding takes place only when primitives are known

⇒ use `(import scheme)`

# Sadly, no eta reduction

`(lambda (x1 ... xn) (e x1 ... xn))`  $\rightarrow$  *e*

obvious problems:

side effects, errors, nontermination

subtle problems:

types, argument counts, continuations

`(procedure? (lambda () (3)))`  $\Rightarrow$  `#t`

`(procedure 3)`  $\Rightarrow$  `#f`

`((lambda (x) (list x)) 'a 'b)`  $\Rightarrow$  *error*

`(list 'a 'b)`  $\Rightarrow$  `(a b)`

`(lambda (x) ((call/cc values)))` ; *caller's k*

`(call/cc values)` ; *current k*

# Remarks

Guaranteed optimization really does help

- for syntactic abstraction
- also for procedural abstraction

Fully effective only if well understood

- guarantees must be well documented
- need way to see what's been done

Wish list:

- way to specify domain-specific optimizations