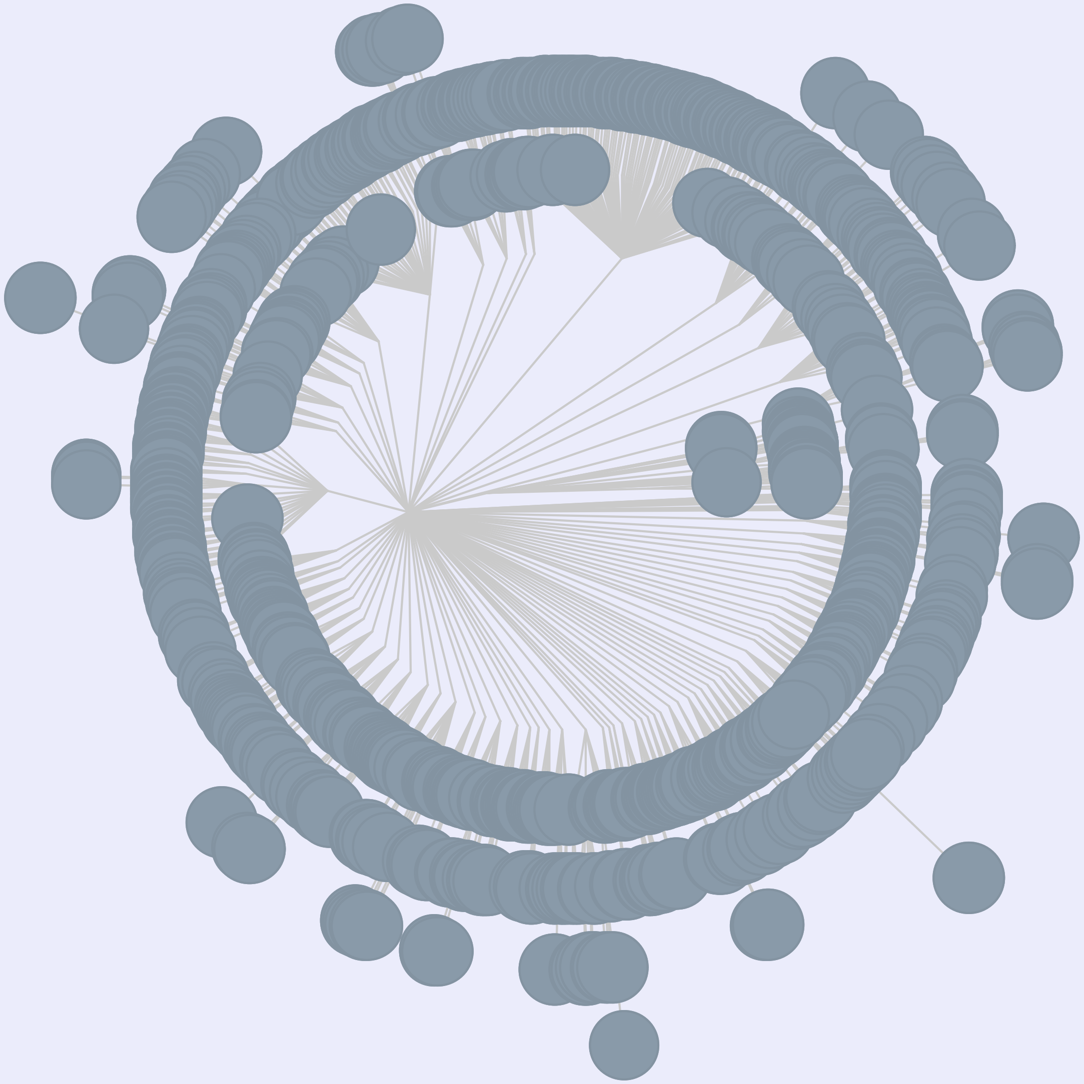
Macros matter

infrastructure for building new PLs

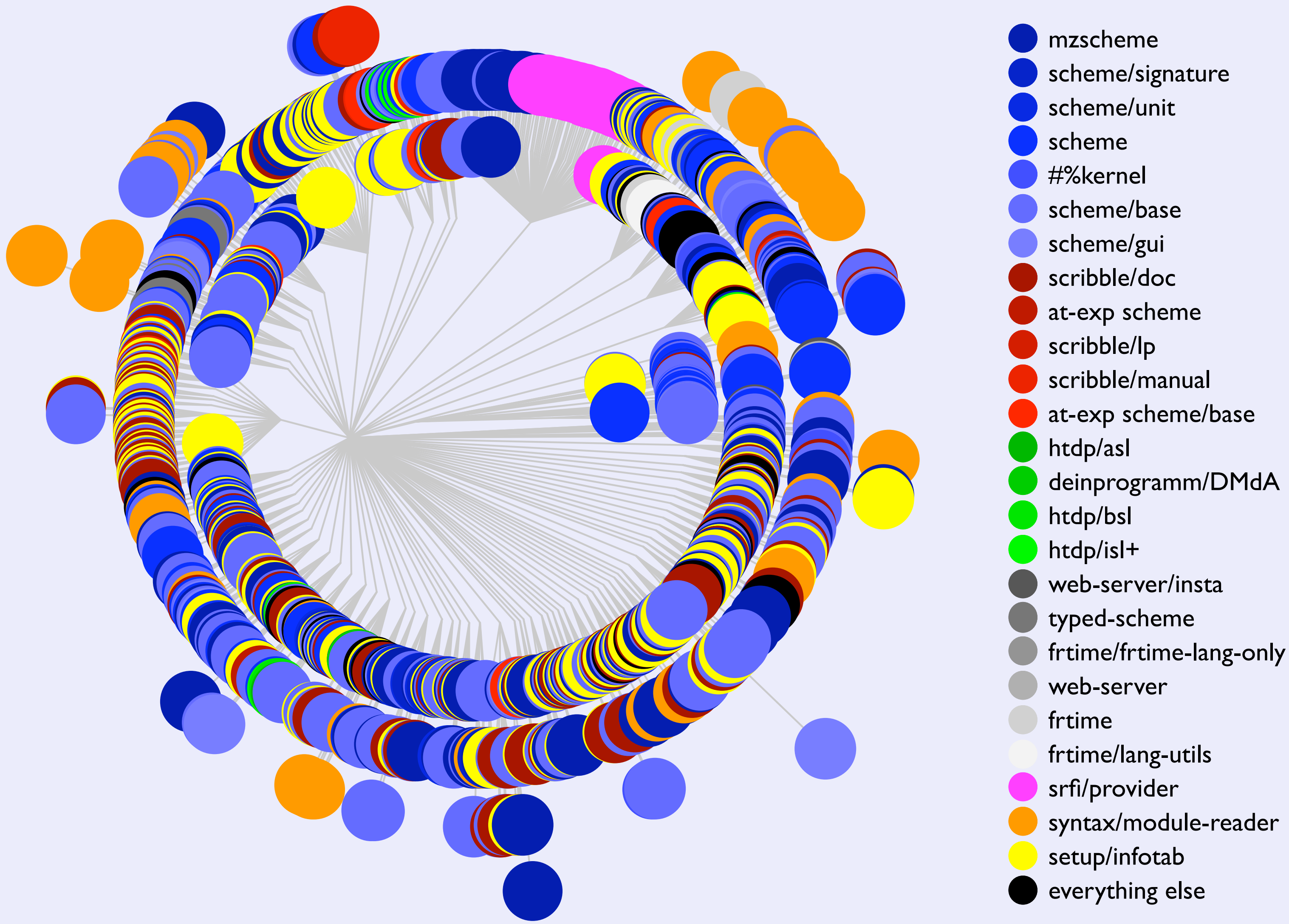
Robby Findler
Northwestern
PLT

Files in PLT



Files in PLT

"A domain specific language is the ultimate abstraction." — Paul Hudak



Macro systems

- A **`macro`** extends a language by specifying how to compile a new feature into existing features

- The macro is itself implemented in the programming language, not an external tool.

Good macros are not salad bars†

```
#define foo "salad

int main(){
  printf(foo bar\n");
}
```

†With thanks (apologies) to Will Clinger and Jonathan Rees

Good macros are not salad bars†

```c
#define sqr(x) x*x

int main(){
  printf("%i\n",sqr(3+2));
}
```

†With thanks (apologies) to Will Clinger and Jonathan Rees

Good macros are not salad bars[†]

```
#define sqr(x) x*x

int main(){
  printf("%i\n",sqr(3+2));
}
```

⇒ 11

[†]With thanks (apologies) to Will Clinger and Jonathan Rees

Good macros are not salad bars[†]

```
#define sqr(x) x*x

int main(){
  printf("%i\n",sqr(3+2));
}
```

$\not\Rightarrow$ `(3+2)*(3+2)`

[†]With thanks (apologies) to Will Clinger and Jonathan Rees

Good macros are not salad bars[†]

```
#define sqr(x) x*x

int main(){
  printf("%i\n",sqr(3+2));
}
```

⇒

```
3+2*3+2
```

[†]With thanks (apologies) to Will Clinger and Jonathan Rees

Good macros are not salad bars†

```
#define sqr(x) x*x

int main(){
  printf("%i\n",sqr(3+2));
}
```

⇒

```
3+2*3+2 = 3+(2*3)+2
```

†With thanks (apologies) to Will Clinger and Jonathan Rees

Good macros are not salad bars[†]

```
#define sqr(x) x*x

int main(){
  printf("%i\n",sqr(3+2));
}
```

⇒

```
3+2*3+2 = 3+(2*3)+2
        = 11
```

[†]With thanks (apologies) to Will Clinger and Jonathan Rees

Outline:

- A challenge
- Academic landmarks
- mini-hdl

Challenge

Design an **`or`** operation:

$(\mathtt{or}\ \mathtt{exp}_a\ \mathtt{exp}_b)$

that returns the first "true" result and is short-circuiting

Challenge

Design an **or** operation:

(**or** exp$_a$ exp$_b$)

that returns the first "true" result and is short-circuiting

```
(define (01-list? x)
  (or (null? x)
      (null? (cdr x))))
```

Non-solution 1: function

```
(define (or x y)
  (if x
      x
      y))

(define (01-list? x)
  (or (null? x)
      (null? (cdr x))))
```

Non-solution 1: function

```
(define (or x y)
  (if x
      x
      y))

(define (01-list? x)
  (or (null? x)
      (null? (cdr x))))

(01-list '()) ⇒ cdr: given '()
```

Non-solution 2: duplicate code

```
(define-syntax-rule
  (or x-exp y-exp)
  (if x-exp x-exp y-exp))


(define (01-list? x)
  (or (null? x)
      (null? (cdr x))))
```

Non-solution 2: duplicate code

Tells the compiler to rewrite the first pattern to the second

```
(define-syntax-rule
  (or x-exp y-exp)
  (if x-exp x-exp y-exp))


(define (01-list? x)
  (or (null? x)
      (null? (cdr x))))
```

Non-solution 2: duplicate code

```
(define-syntax-rule
  (or x-exp y-exp)
  (if x-exp x-exp y-exp))
```

```
(define (01-list? x)       ⇒ (define (01-list? x)
  (or (null? x)                (if (null? x)
      (null? (cdr x))))            (null? x)
                                   (null? (cdr x))))
```

Non-solution 2: duplicate code

```
(define-syntax-rule
  (or x-exp y-exp)
  (if x-exp x-exp y-exp))


(define (012-list? x)
  (or (or (null? x)
          (null?
           (cdr x)))
      (null? (cddr x))))
```

Non-solution 2: duplicate code

```
(define-syntax-rule
  (or x-exp y-exp)
  (if x-exp x-exp y-exp))
```

```
(define (012-list? x)      ⇒ (define (012-list? x)
  (or (or (null? x)            (if (or (null? x)
          (null?                       (null?
           (cdr x)))                    (cdr x)))
      (null? (cddr x))))           (or (null? x)
                                       (null?
                                        (cdr x)))
                                   (null? (cddr x))))
```

Non-solution 2: duplicate code

```
(define-syntax-rule
  (or x-exp y-exp)
  (if x-exp x-exp y-exp))
```

```
(define (012-list? x)
  (or (or (null? x)
          (null?
           (cdr x)))
      (null? (cddr x))))
```

⇒

```
(define (012-list? x)
  (if (if (null? x)
          (null? x)
          (null?
           (cdr x)))
      (if (null? x)
          (null? x)
          (null?
           (cdr x)))
      (null? (cddr x))))
```

Non-solution 2: duplicate code

```
(define-syntax-rule
  (or x-exp y-exp)
  (if x-exp x-exp y-exp))


(define (0123-list? x)
  (or (or (or (null? x)
              (null? (cdr x)))
          (null? (cddr x)))
      (null? (cdddr x))))
```

Non-solution 2: duplicate code

```
(define-syntax-rule
  (or x-exp y-exp)
  (if x-exp x-exp y-exp))
```

```
(or (test-and-set 'x)  ⇒ (if (test-and-set 'x)
    (test-and-set 'y))         (test-and-set 'x)
                               (test-and-set 'y))
```

Non-solution 3: variable capture

```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x x y-exp)))

(define (01-list? x)
  (or (null? x)
      (null? (cdr x))))
```

Non-solution 3: variable capture

```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x x y-exp)))
```

```
(define (01-list? x)
  (or (null? x)
      (null? (cdr x))))
```

⇒

```
(define (01-list? x)
  (let ([x (null? x)])
    (if x
        x
        (null? (cdr x)))))
```

Non-solution 3: variable capture

```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x x y-exp)))
```

```
(define (01-list? x)        ⇒ (define (01-list? x)
  (or (null? x)                 (let ([x (null? x)])
      (null? (cdr x))))           (if x
                                      x
                                      (null? (cdr x)))))
```

`(01-list? (list 1))` ⇒ *cdr: given #f*

Hygiene

```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x x y-exp)))

(define (01-list? x)
  (or (null? x)
      (null? (cdr x))))
```

Hygiene

```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x x y-exp)))

(define (01-list? x^0)
  (or (null? x^0)
      (null? (cdr x^0))))
```

Hygiene

```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x x y-exp)))
```

```
(define (01-list? x^0)
  (or (null? x^0)
      (null? (cdr x^0))))
```

⇒

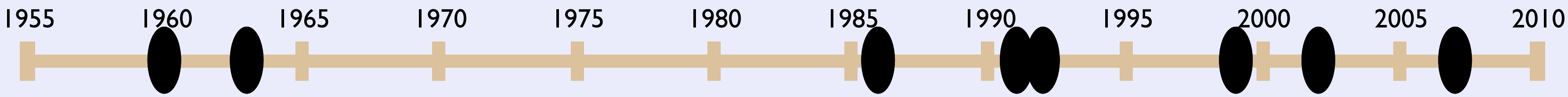
```
(define (01-list? x^0)
  (let ([x^1 (null? x^0)])
    (if x^1
        x^1
        (null?
         (cdr x^0)))))
```

Fix the macro expander:

- Each expansion stage gets its own variables
- Thus variables are safe to use in macros

Academic landmarks

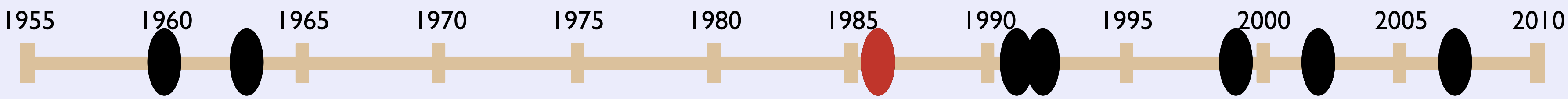
Key macro system developments



Macro Instruction Extensions of Compiler Languages
Doug McIlroy [CACM(3) '60]

MACRO Definitions for LISP.
Timothy P. Hart [AIM-57 '63]

Key macro system developments

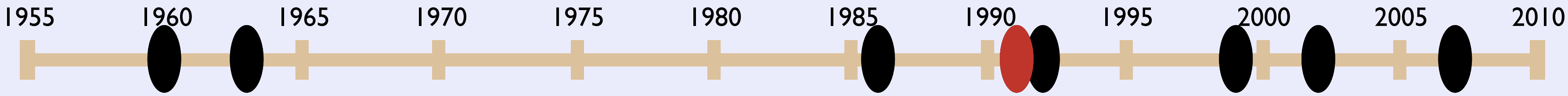


Hygienic macro expansion

Kohlbecker, Friedman, Felleisen, Duba [LFP ’86]

- Introduced hygiene
- Quadratic-time algorithm

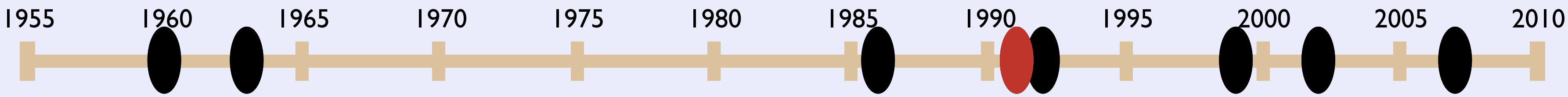
Key macro system developments



Macros that work
Clinger, Rees [POPL ’91]

- Linear time algorithm
- Only pattern-based macros
- Handles free variables in templates properly

Key macro system developments

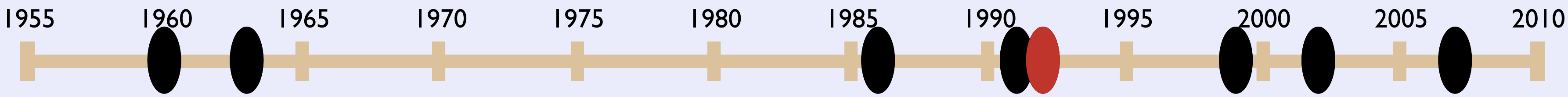


Macros that work
Clinger, Rees [POPL ’91]

```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x
        x
        y-exp)))
```

Bindings for free variables in macro expansion come from definition site, not use site

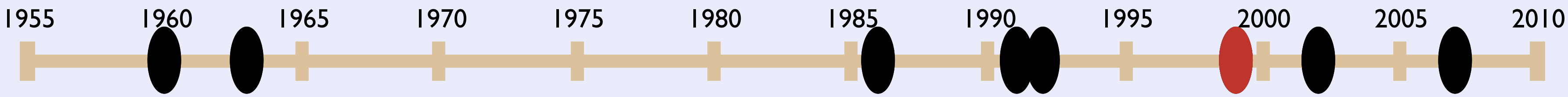
Key macro system developments



Syntactic abstraction in Scheme
Dybvig, Hieb, Bruggeman [LSC ’92]

- Fully general macro transformers
- (Pattern-based macros impl. via a macro)
- Source correlation

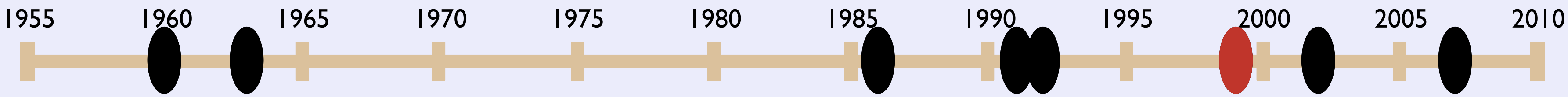
Key macro system developments



Extending the scope of syntactic abstraction
Waddell, Dybvig [POPL '99]

- Module system for macros
- Fine grained control over scope

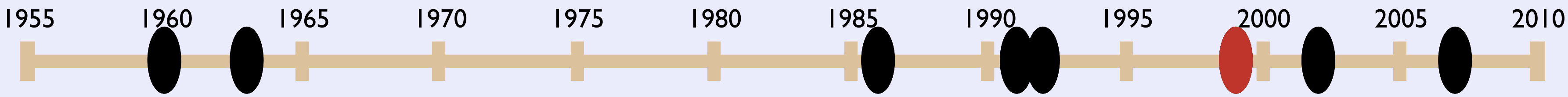
Key macro system developments



Extending the scope of syntactic abstraction
Waddell, Dybvig [POPL ’99]

```
class Super {
    int x=5;
}
class Sub
 extends Super {
    int y=6;
    int m(){
        return x+y;
    }
}
```

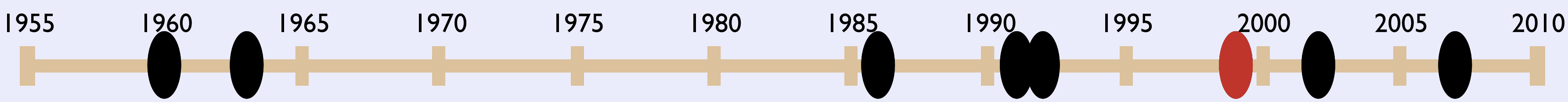
Key macro system developments



Extending the scope of syntactic abstraction
Waddell, Dybvig [POPL '99]

```
class Super {
    int x=5;
}
class Sub
 extends Super {
    int y=6;
    int m(){
        return x+y;
    }
}
```

Key macro system developments



Extending the scope of syntactic abstraction
Waddell, Dybvig [POPL '99]

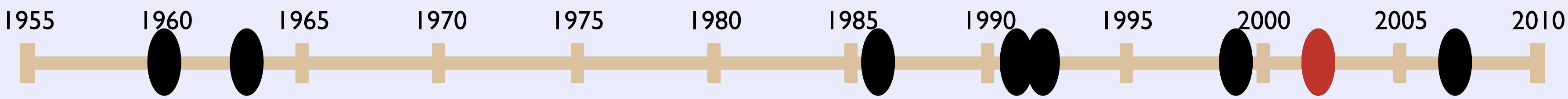
```
class Super {
    int x=5;
}
```

```
class Sub
 extends Super {
    int y=6;
    int m(){
        return x+y;
    }
}
```

⇒

```
class Sub {
    int x=5;
    int y=6;
    int m(){
        return x+y;
    }
}
```

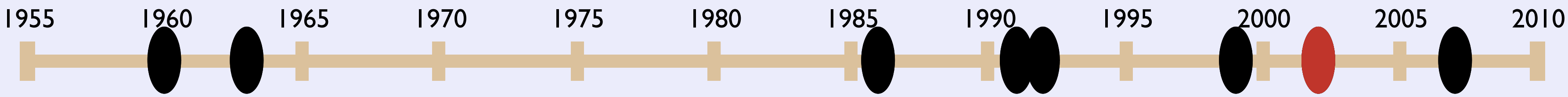
Key macro system developments



Composable and compilable macros
Flatt [ICFP ’02]

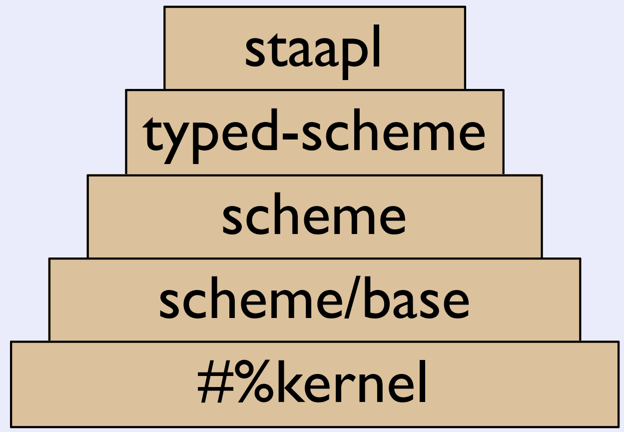
- Separate compilation
- Tower of compile times

Key macro system developments

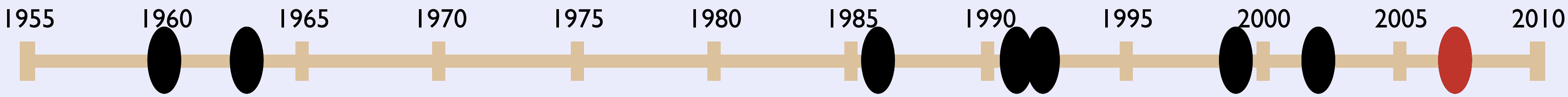


Composable and compilable macros
Flatt [ICFP ’02]

- Separate compilation
- Tower of compile times



Key macro system developments



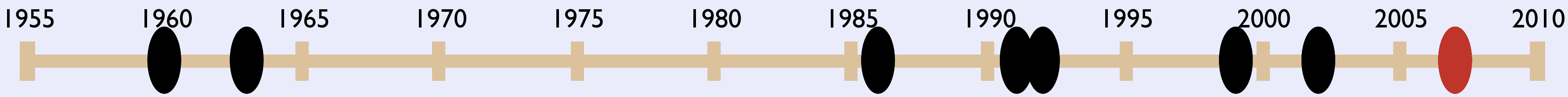
Macro writer's bill of rights
Dybvig [Friedman Feschrift '07]

"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"

Key macro system developments



Macro writer's bill of rights
Dybvig [Friedman Feschrift '07]

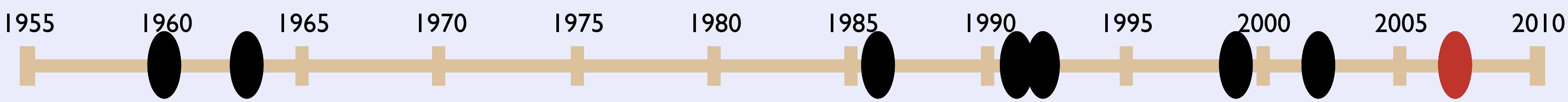
```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x x y-exp)))
```

```
(or z
    #f)
```

```
(let ([x z])
  (if x
      x
      #f))
```

```
z
```

Key macro system developments

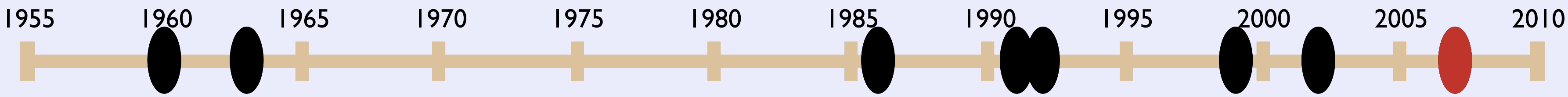


Macro writer's bill of rights
Dybvig [Friedman Feschrift '07]

```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x x y-exp)))
```

```
(or z          (let ([x z])          z
    #f)          (if x
                     x
                     #f))
```

Key macro system developments



Macro writer's bill of rights
Dybvig [Friedman Feschrift '07]

```
(define-syntax-rule
  (or x-exp y-exp)
  (let ([x x-exp])
    (if x x y-exp)))
```

```
(or z            (let ([x z])          z
    #f)            (if x
                       x
                       #f))
```

mini-hdl

Working through mini-hdl

```
inputs a1,a0 = 2;
inputs b1,b0 = 1;
s0 = a0 ⊕ b0;
c0 = a0 ∧ b0;
s1 = a1 ⊕ b1 ⊕ c0;
c1 = (a1 ∧ b1) ∨
     (c0 ∧ (a1 ⊕ b1));
showint(c1,s1,s0);
```

Working through mini-hdl

```
(inputs (a1 a0) 2)
(inputs (b1 b0) 1)
(= s0 (⊕ a0 b0))
(= c0 (∧ a0 b0))
(= s1 (⊕ a1 (⊕ b1 c0)))
(= c1 (∨ (∧ a1 b1)
         (∧ c0 (⊕ a1 b1))))
(showint c1 s1 s0)
```

```
#lang scheme/base
(provide hdl-read hdl-read-syntax)
(require parser-tools/lex parser-tools/yacc
         (prefix-in : parser-tools/lex-sre)
         schemeunit)

(define-tokens tokens (id kwd num op))
(define-empty-tokens mt-tokens (semi comma equal inputs eoft open close showint))

(define lex
  (lexer-src-pos
   ["inputs" (token-inputs)]
   ["showint" (token-showint)]
   [(:or #\⊕ #\∧ #\∨) (token-op (string->symbol lexeme))]
   [#\; (token-semi)]
   [#\, (token-comma)]
   [#\= (token-equal)]
   [#\( (token-open)]
   [#\) (token-close)]
   [(:: (:/ #\a #\z #\A #\Z)
        (:* (:/ #\a #\z #\A #\Z #\0 #\9)))
    (token-id (string->symbol lexeme))]
   [(:+ (:/ #\0 #\9))
    (token-num (string->number lexeme))]
   [(:+ whitespace) (return-without-pos (lex input-port))]
   [(eof) (token-eoft)]))

(define (str->toks str)
  (let ([p (open-input-string str)])
    (let loop ()
      (let ([next (lex p)])
        (cons (token-name (position-token-token next))
              (if (eq? 'eoft (token-name (position-token-token next)))
                  '()
                  (loop)))))))


(check-equal? (str->toks "ab0 ∨  1234; inputs")
              '(id op num semi inputs eoft))
(check-equal? (str->toks "showint(x);")
              '(showint open id close semi eoft))

(define current-source (make-parameter #f))

(define parse
  (parser
   [grammar
    (start [(decls) $1])
    (decls [(decl) (list (add-srcloc $1 $1-start-pos $1-end-pos))]
           [(decl decls) (cons (add-srcloc $1 $1-start-pos $1-end-pos) $2)])
    (decl [(inputs ids equal num semi) `(inputs ,$2 ,$4)]
          [(inputs ids semi) `(inputs ,$2)]
          [(id equal expr semi) `(= ,(add-srcloc $1 $1-start-pos $1-end-pos) ,$3)]
          [(showint open ids close semi) `(showint ,@$3)])
    (expr [(expr op expr) (prec op) (add-srcloc `(,$2 ,$1 ,$3) $1-start-pos $n-end-pos)]
          [(num) (add-srcloc $1 $1-start-pos $n-end-pos)]
          [(id) (add-srcloc $1 $1-start-pos $n-end-pos)]
          [(open expr close) (add-srcloc $2 $1-start-pos $n-end-pos)])
    (ids [(id) (list (add-srcloc $1 $1-start-pos $1-end-pos))]
         [(id comma ids) (cons (add-srcloc $1 $1-start-pos $1-end-pos) $3)])]
   [precs (right op)]
   [tokens mt-tokens tokens]
   [src-pos]
   [start start]
   [end eoft]
   [error
    (lambda (tok-ok? tok-name tok-value start-pos end-pos)
      (raise-syntax-error
       'parse-error
       (format "~s" (if tok-ok?
                        tok-value
                        'unknown))
       (add-srcloc (if tok-ok?
                       tok-value
                       'unknown)
                   start-pos
                   end-pos)))]))

(define (add-srcloc stuff start-pos end-pos)
  (datum->syntax #f stuff
                 (vector
                  (current-source)
                  (position-line start-pos)
                  (position-col start-pos)
                  (position-offset start-pos)
                  (- (position-offset end-pos)
                     (position-offset start-pos)))))

(define (run-p src p)
  (parameterize ([current-source src])
    (parse (λ () (lex p)))))

(define (hdl-read [port (current-input-port)])
  (syntax->datum (run-p #f port)))

(define (hdl-read-syntax [name #f] [port (current-input-port)])
  (run-p (or name (object-name port))
         port))
```

parser.ss
98 lines

Working through mini-hdl

```
(define a1 (nth-bit 1 2))
(define a0 (nth-bit 0 2))
(define b1 (nth-bit 1 1))
(define b0 (nth-bit 0 1))
(define s0 (⊕ a0 b0))
(define c0 (∧ a0 b0))
(define s1 (⊕ a1 (⊕ b1 c0)))
(define c1 (∨ (∧ a1 b1)
               (∧ c0 (⊕ a1 b1))))
(showint c1 s1 s0)
```

⇒ 3

```
#lang scheme
(require schemeunit)

(define-syntax-rule (= a b) (define a b))

(define (∧ a b) (and a b))
(define (∨ a b) (or a b))
(define (⊕ a b) (not (equal? a b)))

(define-syntax (inputs stx)
  (syntax-case stx ()
    [(inputs (a ...))
     (raise-syntax-error 'inputs "expected an initial value" stx)]
    [(inputs (a ...) n-exp)
     (with-syntax ([(i ...)
                    (let ([count (length (syntax->list #'(a ...)))])
                      (build-list count (λ (x) (- count x 1))))])
       #'(define-values (a ...)
           (let ([n n-exp])
             (values (extract-bit i n) ...))))]))

;; extract-bit : number[index] number[bits] -> boolean
;; returns the ith bit in n (indexing from least significant)
(define (extract-bit i n)
  (let ([bits (n->bits n)])
    (cond
      [(< i (length bits))
       (list-ref bits i)]
      [else #f])))

;; n->bits : positive-integer -> (listof boolean)
;; returns the bits of n from least significant to most significant
(define (n->bits n)
  (cond
    [(zero? n) '()]
    [(zero? (modulo n 2)) (cons #f (n->bits (quotient n 2)))]
    [else (cons #t (n->bits (quotient n 2)))]))

(check-equal? (n->bits 1) (list #t))
;; 11 = 1 + 2 + 8
(check-equal? (n->bits 11) (list #t #t #f #t))
(check-equal? (extract-bit 0 1) #t)
(check-equal? (extract-bit 0 0) #f)
(check-equal? (extract-bit 0 11) #t)
(check-equal? (extract-bit 2 11) #f)
(check-equal? (extract-bit 111 11) #f)
(check-equal? (let ()
                (inputs (a3 a2 a1 a0) 7)
                (list a3 a2 a1 a0))
              (list #f #t #t #t))

(define (showint . args)
  (let loop ([args args]
             [n 0])
    (cond
      [(null? args) n]
      [else
       (loop (cdr args)
             (+ (* n 2)
                (if (car args) 1 0)))])))

(check-equal? (showint #t #t #f #t) 13)

(provide
 #%module-begin
 #%datum
 #%app
 #%top-interaction
 #%top
 extract-bit
 showint
 inputs
 = ⊕ ∧ ∨)
```

runtime.ss
73 lines

Working through mini-hdl

```
(define (iterate a1 a0 b1 b0
                 s0 c0 s1 c1)
 (let ((a1 (nth-bit 1 2))
       (a0 (nth-bit 0 2))
       (b1 (nth-bit 1 1))
       (b0 (nth-bit 0 1))
       (s0 (⊕ a0 b0))
       (c0 (∧ a0 b0))
       (s1 (⊕ a1 (⊕ b1 c0)))
       (c1 (∨ (∧ a1 b1)
              (∧ c0 (⊕ a1 b1)))))
  (values a1 a0 b1 b0
          s0 c0 s1 c1)))
```

gc-runtime.ss
107 lines

Working through mini-hdl

```
#lang s-exp syntax/module-reader
"gc-runtime.ss"
#:read hdl-read
#:read-syntax hdl-read-syntax
#:whole-body-readers? #t
(require "parser.ss")
```

```
#lang s-exp syntax/module-reader
"runtime.ss"
#:read hdl-read
#:read-syntax hdl-read-syntax
#:whole-body-readers? #t
(require "parser.ss")
```

Conclusions

- Macros matter
- Need a new language? Try PLT

Thanks

With help from Matthew Flatt, Eli Barzilay, Matthias Felleisen, Jay McCarthy, and all of PLT.