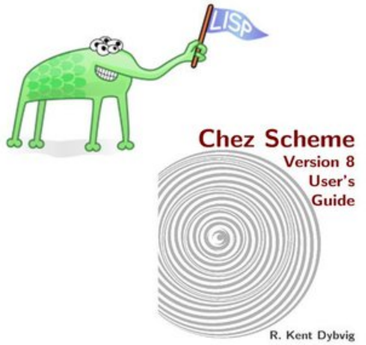
Clojure isn’t Lisp enough

How to use the benefit of s-expression

林子篆

Lisp 方言阵营图	语法纯粹派	语法中立派	语法自由派
语义纯粹派	 Common Lisp 和 Scheme 显然是 Lisp	 language-oriented programming 才是 Lisp 的精髓	julia JMC LISP 2 精神传人
语义中立派	 动态作用域的 Lisp 才是原教旨 Lisp	 JMC 又没说 Lisp 一定要有宏	 有 GC 有 CLOS-Style OO 系统有宏怎么就不是 Lisp 了
语义自由派	 有基于 F-Expr 的宏才是真的 Lisp	 Lisp 又不一定要有 Cons Pair	JS JS 就是换皮 Scheme 啦!

Benefit of S-expression

))

Case 1

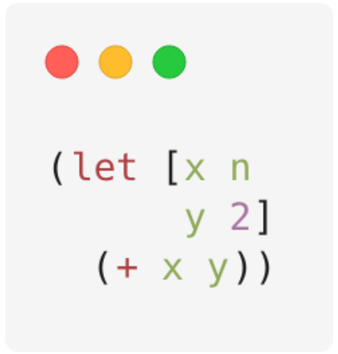
conditional branch

```
(condp = n
  0 0
  1 1
  (+ (fib (- n 1))
     (fib (- n 2))))
```

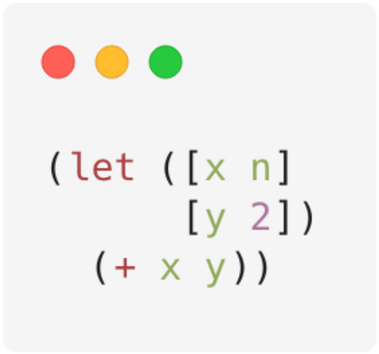
```
(match n
  [0 0]
  [1 1]
  [else (+ (fib (- n 1))
           (fib (- n 2)))])
```

Case 2

let binding



```
(let [x n
      y 2]
  (+ x y))
```

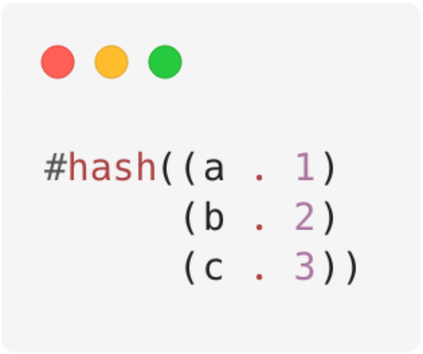


```
(let ([x n]
      [y 2])
  (+ x y))
```

Case 3

map

```
{:a 1
 :b 2
 :c 3}
```



```
#hash((a . 1)
      (b . 2)
      (c . 3))
```

What's wrong?

Case 1(Clojure)

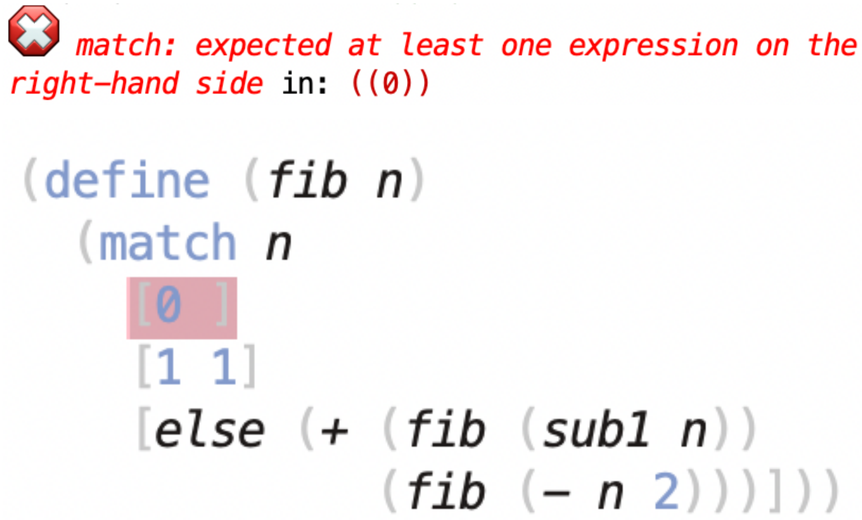
Where is my error reporting?

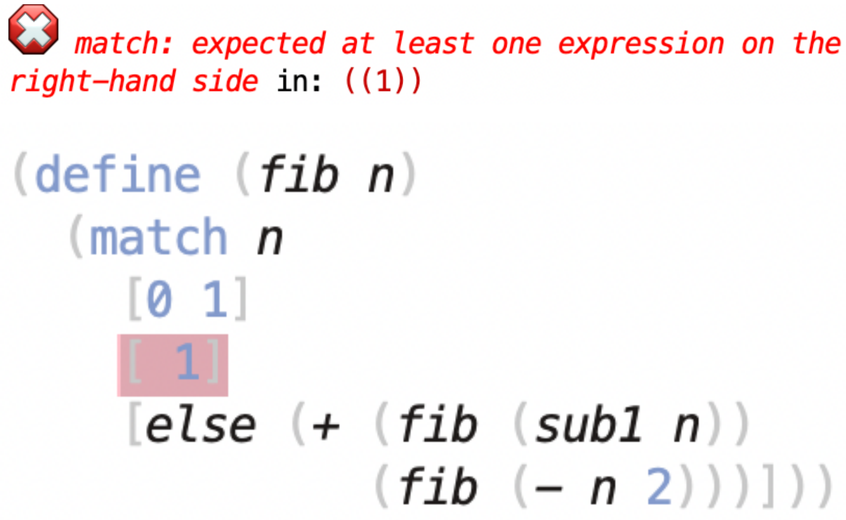
```
(condp = n
  0
  1 1
  (+ (fib (- n 1))
     (fib (- n 2))))
```

```
(condp = n
  0 1
    1
  (+ (fib (- n 1))
     (fib (- n 2))))
```

Case 1(Racket)

Correctly point out error location

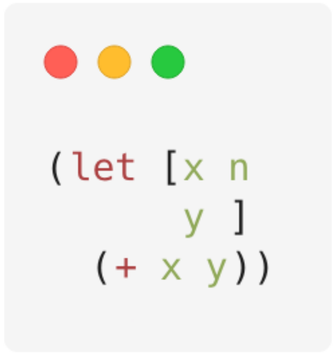




Case 2(Clojure)

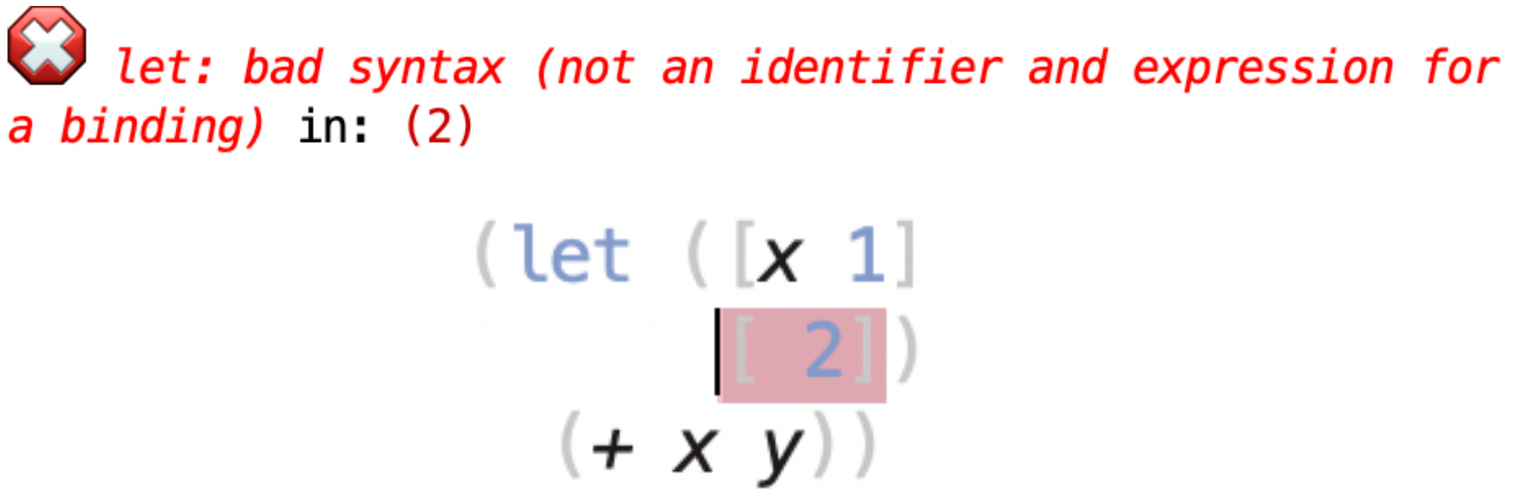
Bad message

```
[x n y] - failed: even-number-of-forms? at: [:bindings] spec: :clojure.core.specs.alpha/bindings
```



Case 2(Racket)

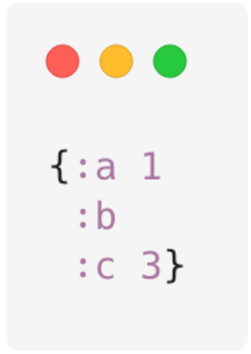
Better message



Case 3(Clojure)

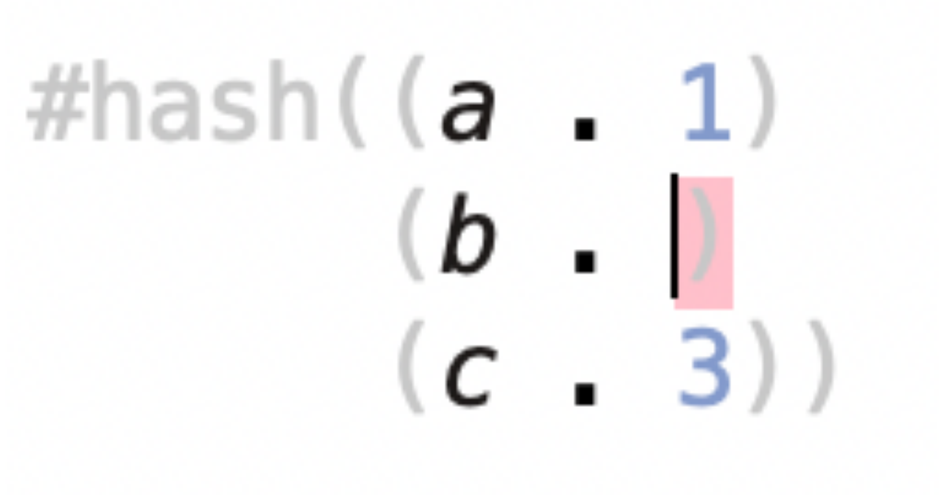
Bad message

```
Map literal must contain an even number of forms
```



Case 3(Racket)

Point out syntax error



Now we know what go wrong

What can we do?

Rewrite: A new let form

```clojure
(defmacro let-sexp
  [bindings & body]
  (doseq [bind bindings]
    (assert (= (-> bind count) 2) "invalid binding"))
  `((fn [~@(map first bindings)]
     ~@body)
    ~@(map second bindings)))
```

New let form example

```
(let-sexp ([x 1]
           [y 2])
  (+ x y))
```

What’s macro?

Macro is...

- Text substitution(C)
- Meta substitution(C++ Template)
- Compile-time function, Syntax validator(Clojure)
- A function in higher phase(Racket)
- A function with dynamic scope(f-expr: vau operator)
- Maybe more?

Racket macro: Evolution

syntax-rules (1975)

```
(define-syntax let1
  (syntax-rules ()
    [(_ ([var rhs] ...) body)
     ((λ (var ...)
        body)
      rhs ...)]))
```

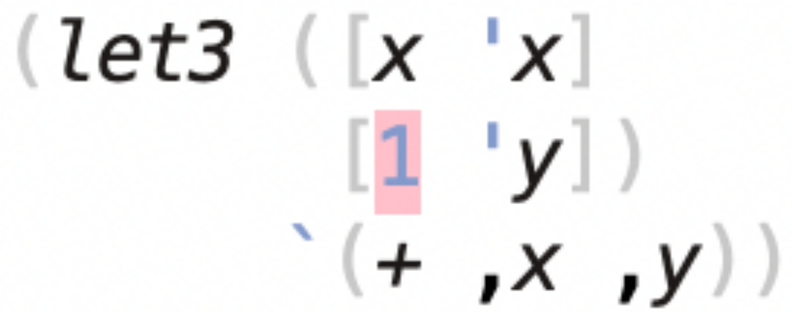
syntax-case (1988)

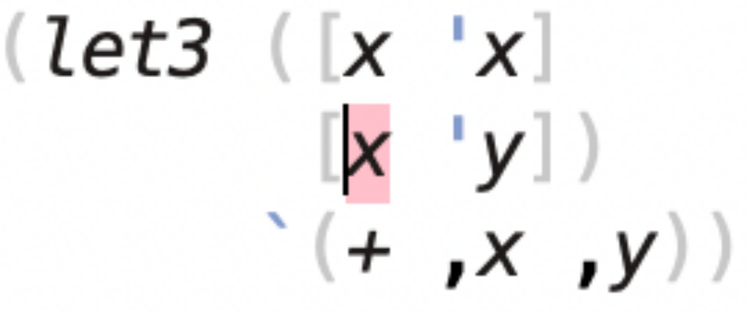
```
(define-syntax (let2 stx)
  (syntax-case stx ()
    [(_ ([var rhs] ...) body)
     ; guard
     (not (check-duplicate-identifier (syntax->list #'(var ...))))
     ; result
     #'((λ (var ...)
          body)
        rhs ...)]))
```

syntax-case (improved)

```
(define-syntax (let3 stx)
  (syntax-case stx ()
    [(_ ([var rhs] ...) body)
     ; guard
     (begin
       (for-each (λ (var)
                   (unless (identifier? var)
                     (raise-syntax-error 'not-identifier "expected identifier"
                                         stx var)))
                 (syntax->list #'(var ...)))
       (let ([dup (check-duplicate-identifier (syntax->list #'(var ...)))])
         (when dup
           (raise-syntax-error 'dup "duplicate variable name" stx dup))))
     ; result
     #'((λ (var ...)
          body)
        rhs ...)]))
```

syntax-case (improved): show error





syntax-parse (1993)

```
(define-syntax (let4 stx)
  (syntax-parse stx
    [(let4 ([var:id rhs:expr] ...) body:expr)
     #:fail-when (check-duplicate-identifier (syntax->list #'(var ...))) "duplicate variable name"
     #'((λ (var ...) body) rhs ...)]))
```

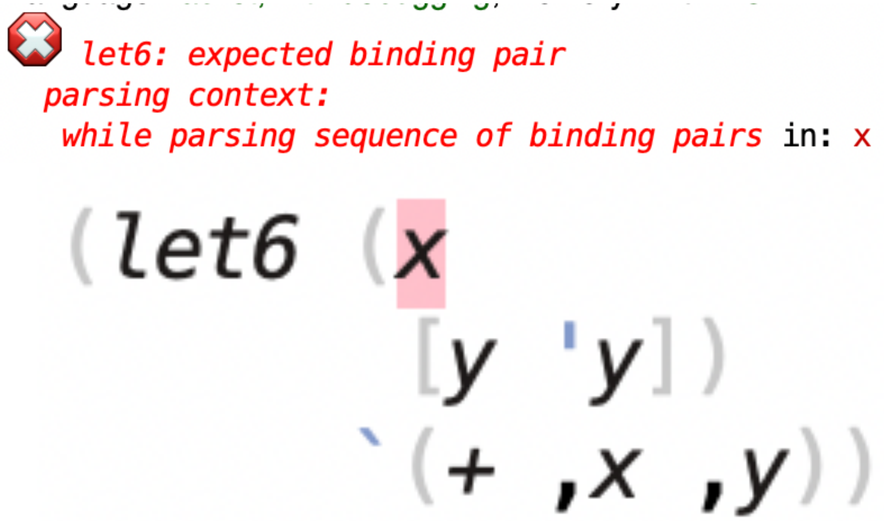
syntax-class

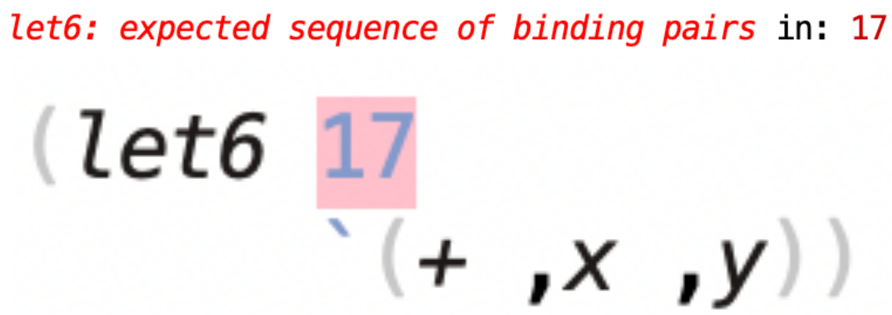
```
(define-syntax (let5 stx)
  (define-syntax-class binding
    #:description "binding pair"
    (pattern [var:id rhs:expr]))
  (syntax-parse stx
    [(let5 (b*:binding ...) body:expr)
     #:fail-when (check-duplicate-identifier (syntax->list #'(b*.var ...)))
                 "duplicate variable name"
     #'((λ (b*.var ...) body) b*.rhs ...)]))
```

syntax-class(improved)

```
(define-syntax (let6 stx)
  (define-syntax-class binding
    #:description "binding pair"
    (pattern [var:id rhs:expr]))
  (define-syntax-class distinct-bindings
    #:description "sequence of binding pairs"
    (pattern (b*:binding ...)
             #:fail-when (check-duplicate-identifier (syntax->list #'(b*.var ...)))
                         "duplicate variable name"
             #:with (var ...) #'(b*.var ...)
             #:with (rhs ...) #'(b*.rhs ...)))
  (syntax-parse stx
    [(let6 b*:distinct-bindings body:expr)
     #'((λ (b*.var ...) body) b*.rhs ...)]))
```

syntax-class(improved): error message





Error selection

```
(define-syntax (let7 stx)
  (define-syntax-class binding
    #:description "binding pair"
    (pattern [var:id rhs:expr]))
  (define-syntax-class distinct-bindings
    #:description "sequence of binding pairs"
    (pattern (b*:binding ...)
             #:fail-when (check-duplicate-identifier (syntax->list #'(b*.var ...)))
                         "duplicate variable name"
             #:with (var ...) #'(b*.var ...)
             #:with (rhs ...) #'(b*.rhs ...)))
  (syntax-parse stx
    [(let7 loop:identifier bs:distinct-bindings body:expr)
     #'(letrec ([loop (λ (bs.var ...) body)]) (loop bs.rhs ...))]
    [(let7 b*:distinct-bindings body:expr)
     #'((λ (b*.var ...) body) b*.rhs ...)]))
```

Error selection: report message

```
let7: expected identifier or expected sequence of binding pairs in: 17
```

```
(let7 17
      `(+ ,x ,y))
```

Macro: implementation

Naive Hygienic Expansion(1986)

The problem of naive Hygienic Expansion

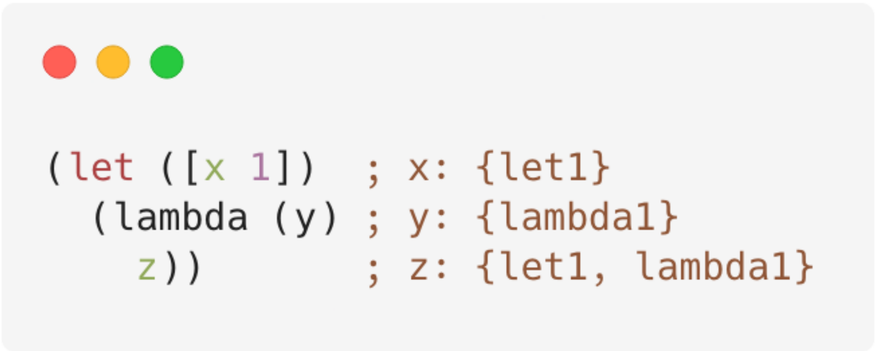
- Unwieldy for recursive definition contexts
- Inefficiently and hard to get correct with “hygiene bending”(e.g. `datum->syntax`)

Binding as sets of scopes

Sets of Scopes: Binding Rule

We can define binding based on *subsets*: A reference's binding is found as one whose set of scopes is the largest subset of the reference's own scopes (in addition to having the same symbolic name).

Sets of Scopes



```
(let ([x 1])  ; x: {let1}
  (lambda (y) ; y: {lambda1}
    z))       ; z: {let1, lambda1}
```

Sets of Scopes: complex

```
(let ([x 1])                              ; x: {let1}
  (let-syntax ([m (syntax-rules ()        ; m: {let1, let-syntax1}
                    [(m) #'x])])          ; #'x: {let1}
    (lambda (x)                           ; x: {let1, let-syntax1, lambda1}
      (m))))                              ; m: {let1, let-syntax1, lambda1}
```

Sets of Scopes: complex (m) expanded

```
(let ([x 1])                              ; x: {let1}
  (let-syntax ([m (syntax-rules ()        ; m: {let1, let-syntax1}
                    [(m) #'x])])          ; #'x: {let1}
    (lambda (x)                           ; x: {let1, let-syntax1, lambda1}
      x)))                                ; x: {let1, macro-intro}
```

Macro: development

New definition

```
(define-syntax-parser data
  [(name:id c*:constructor ...)
   (language-server/new-def name this-syntax)
   #'(begin
       (define name Type)
       c*.define ...)])
```

Auto complete

```
(define-syntax-parser data
  [(name:id c*:constructor ...)
   (language-server/new-complete 'data '(data !name !constructor))
   #'(begin
       (define name Type)
       c*.define ...)])
```