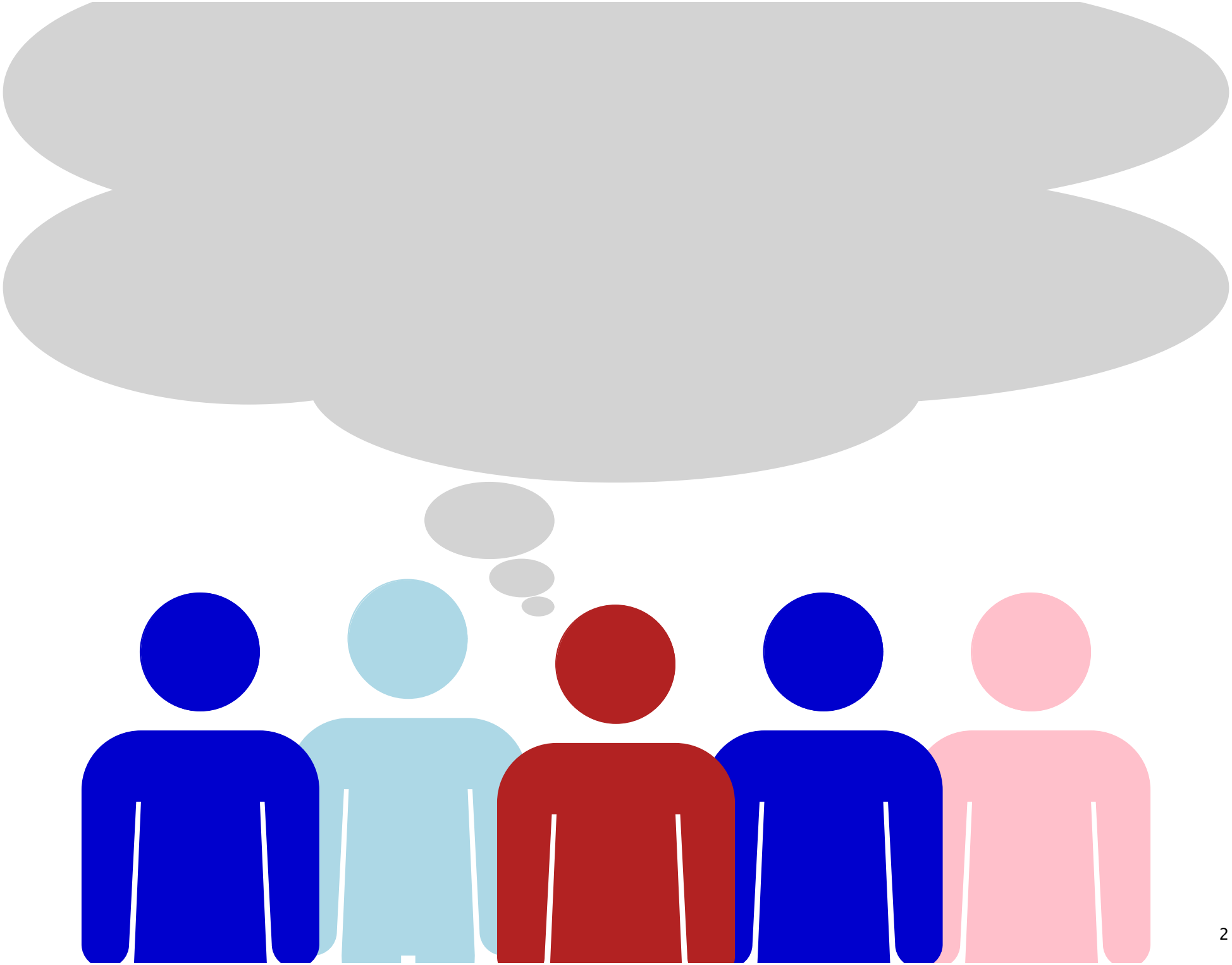
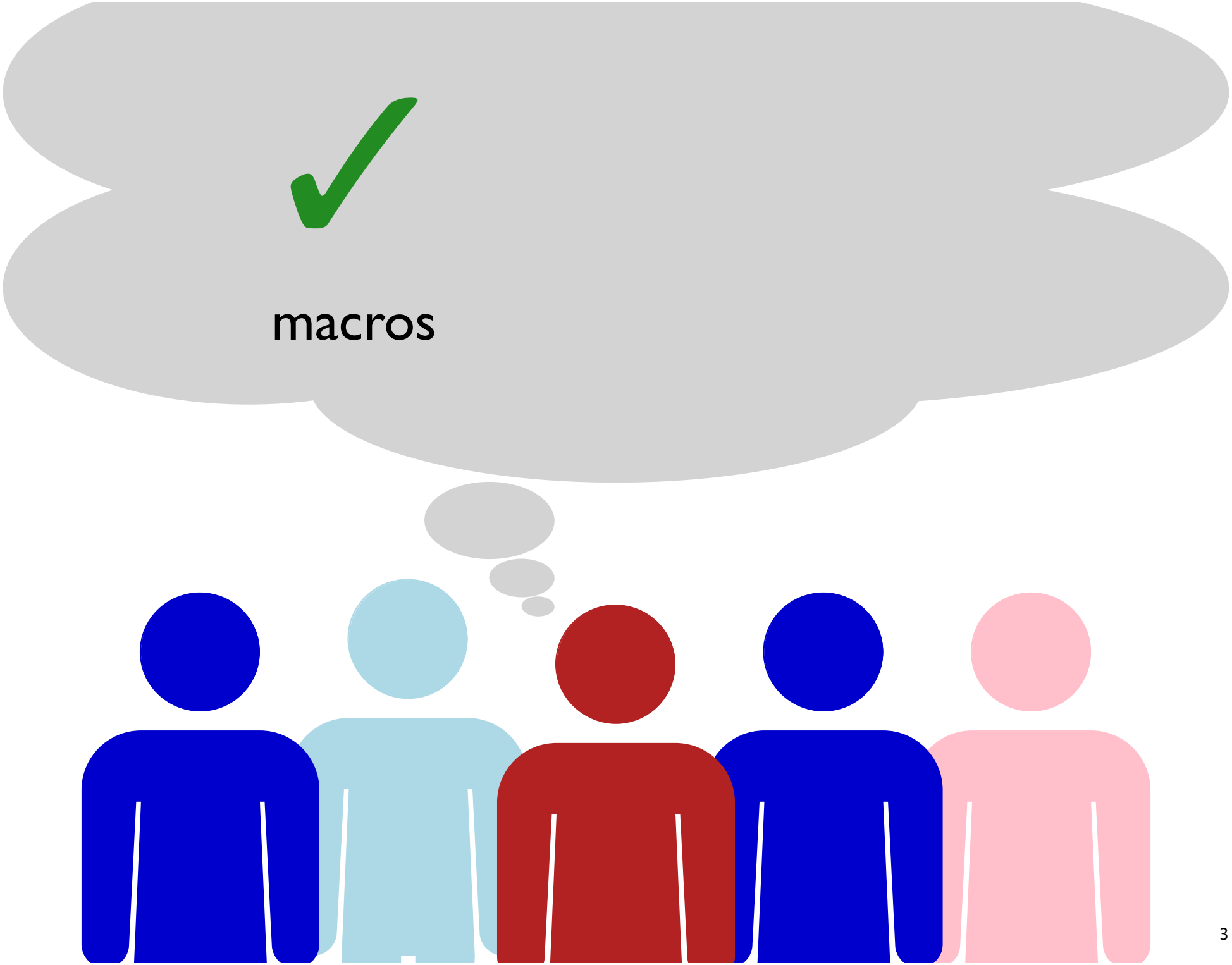
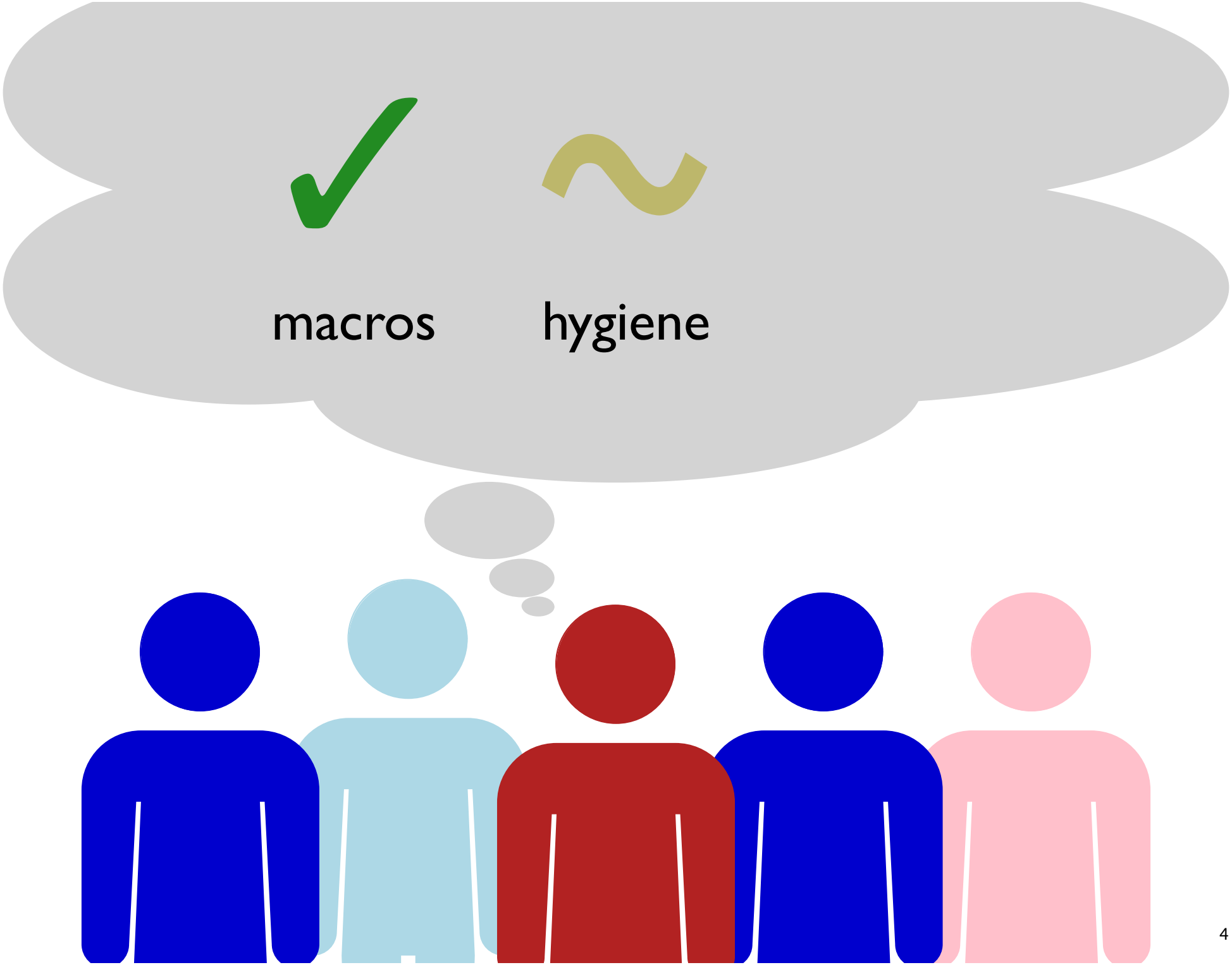
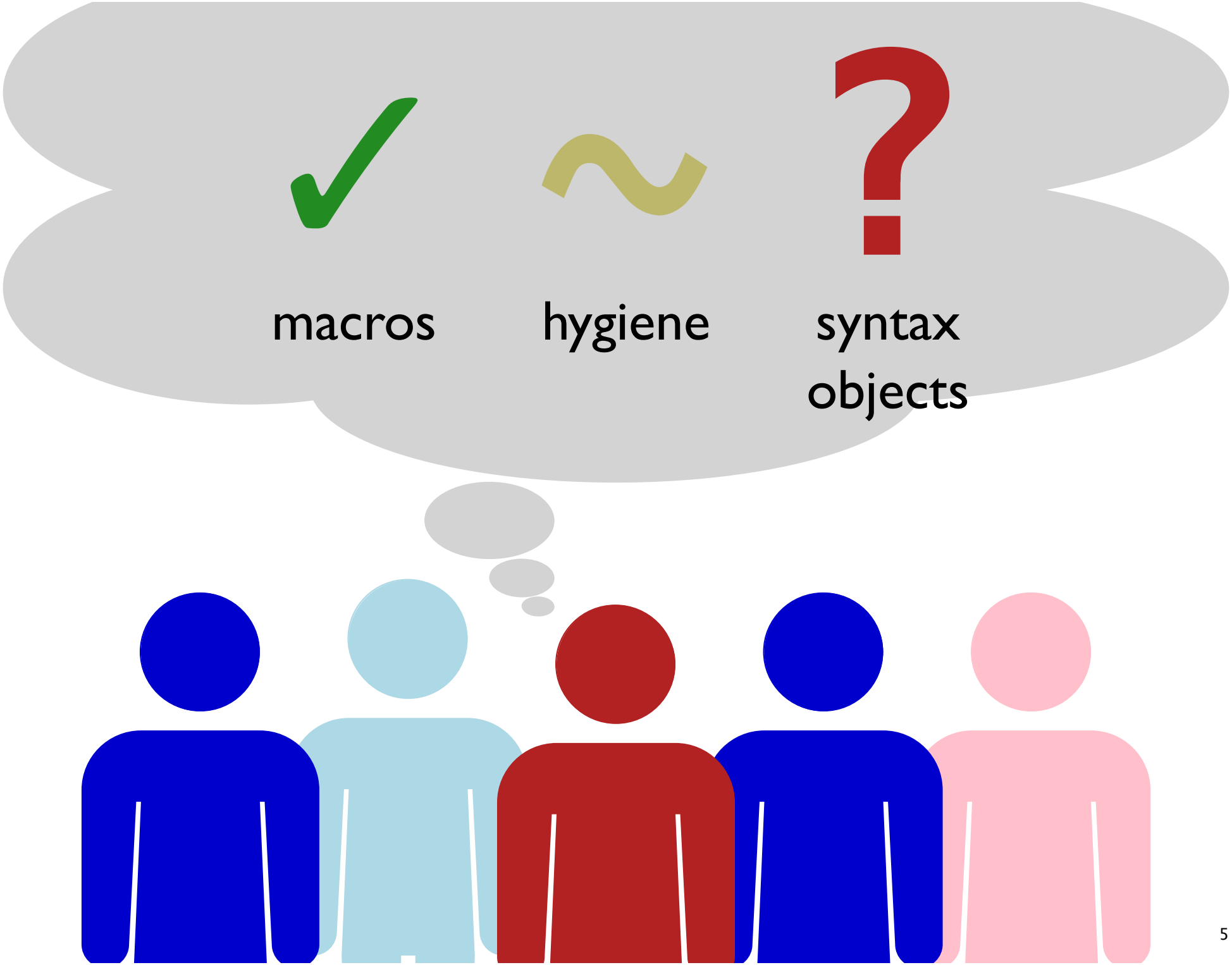
Matthew Flatt




✓
macros


macros
hygiene


macros
hygiene
syntax objects

```
(define x ....)
```

```
; maybe value for "premade":
(define x ....)
```

```
; maybe value for "premade":
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))
```

```
; maybe value for "premade":
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (premade-or (x)))
```

```
; maybe value for "premade":
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))          (premade-or (x))

(let ([x ....])
     ....    )
```

```
; maybe value for "premade":
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))          (premade-or    )
         (x)
(let ([x ....])
     ....    )
```

```
; maybe value for "premade":
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))          (premade-or    )
  (or x (x))
(let ([x ....])
     ....    )
```

```
; maybe value for "premade":
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (or x (x)))
```

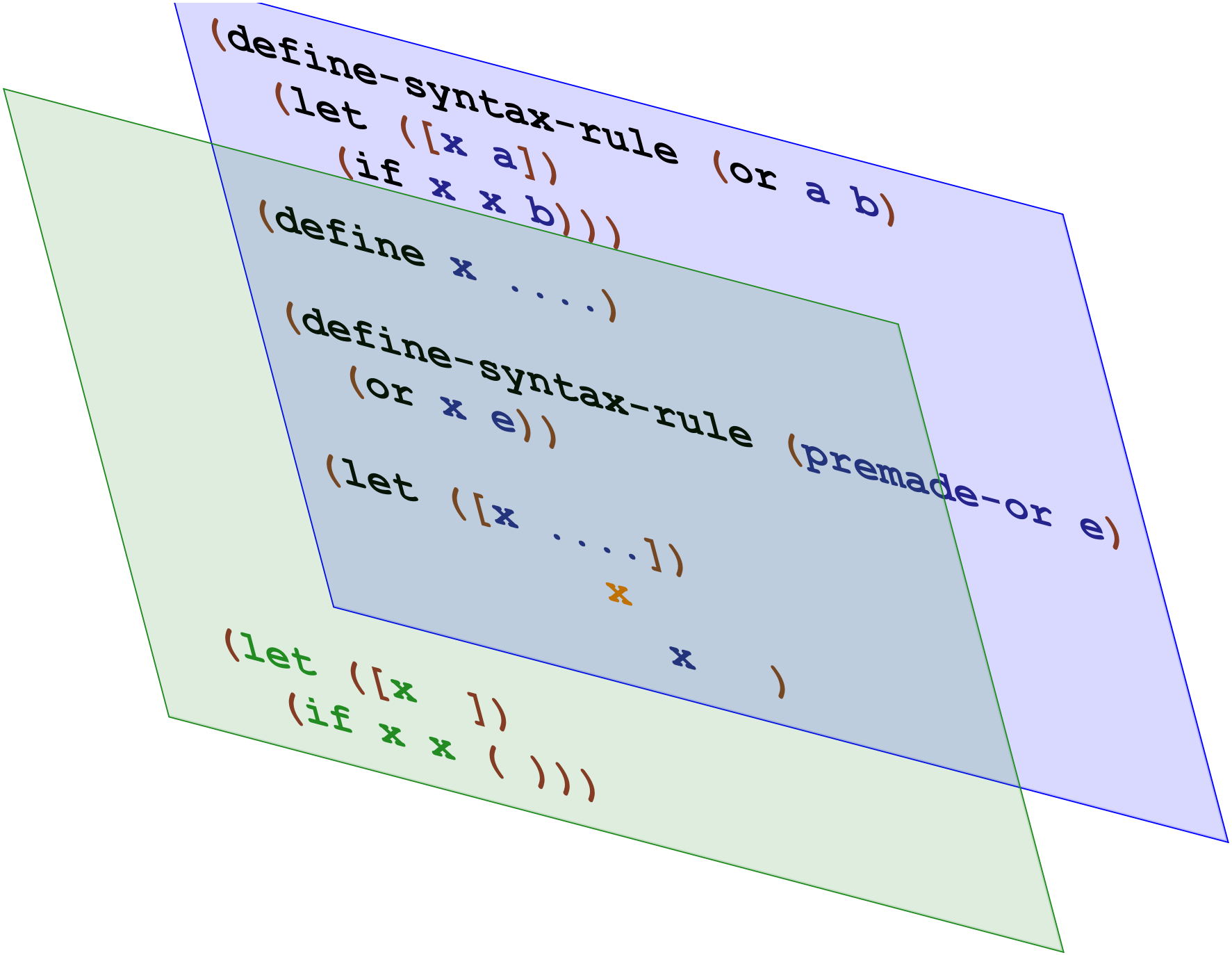
```
(define x ....)

(define (f x)
  (define x ....)

  (define-syntax-rule (premade-or e)
    (or x e))

  (let ([x ....])
    (or x (x)))
  ....)

(define y ....)
```

```
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (or x (x)))
```

```
(define-syntax-rule (or a b)
  (let ([x a])
    (if x x b)))

(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (or x (x)))
```

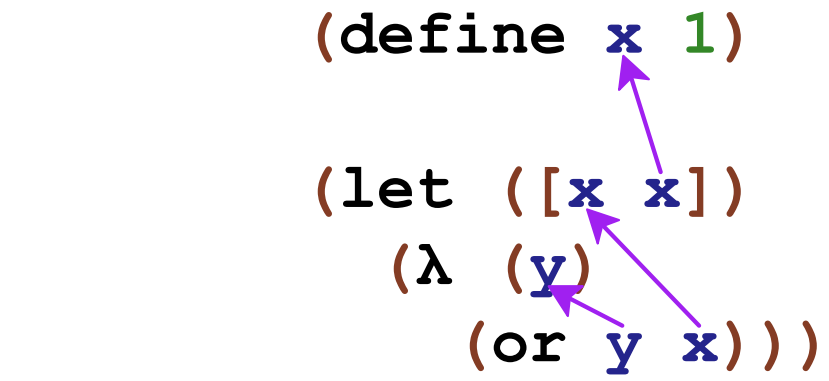
```
(define-syntax-rule (or a b)
  (let ([x a])
    (if x x b)))

(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (let ([x x])
    (if x x (x))))
```

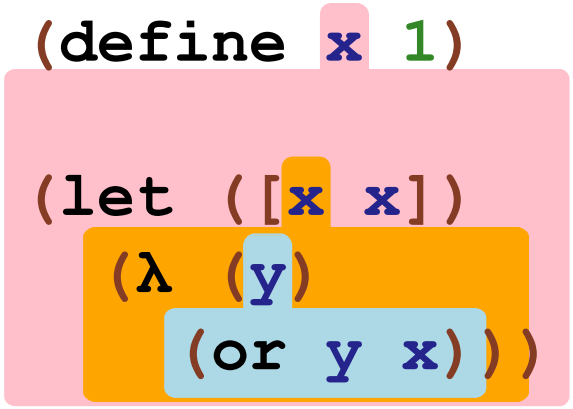
```
(define-syntax-rule (or a b)
  (let ([x a])
    (if x x b)))

(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (let ([x x])
    (if x x (x))))
```

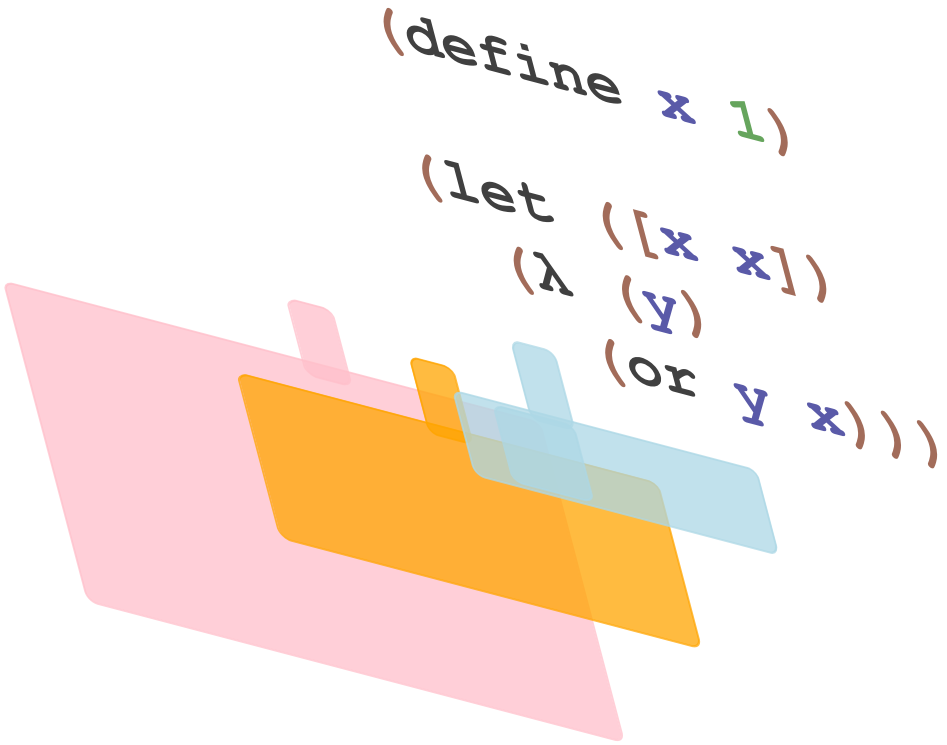

(define-syntax-rule (or a b)
(let ([x a])
(if x x b)))
(define x ....)
(define-syntax-rule (premade-or e)
(or x e))
(let ([x ....])
x
x)
(let ([x])
(if x x ( )))

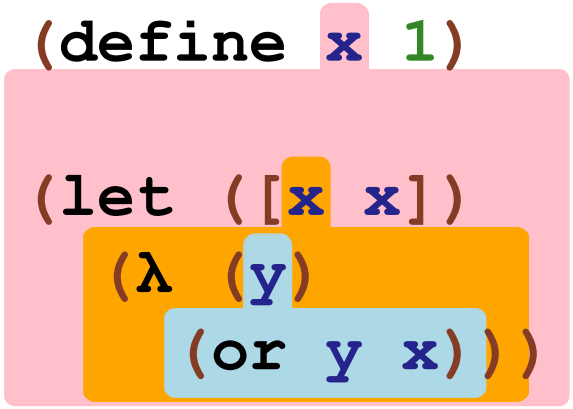
```
(define-syntax-rule (or a b)
  (let ([x a])
    (if x x b)))

(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (let ([x x])
    (if x x (x))))
```

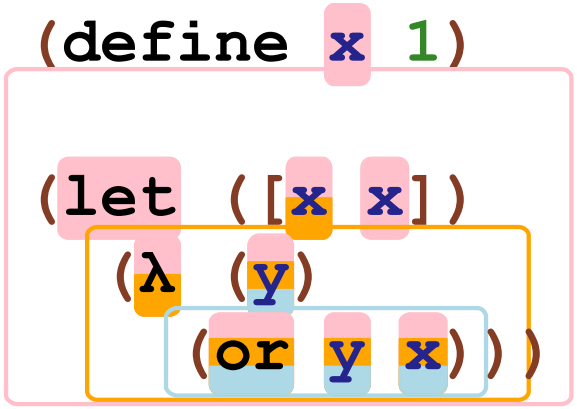
```
(define x 1)

(let ([x x])
  (λ (y)
    (or y x)))
```

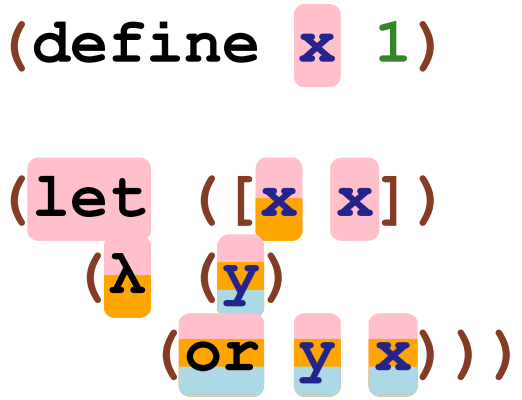

(define x 1)
(let ([x x])
(λ (y)
(or y x)))

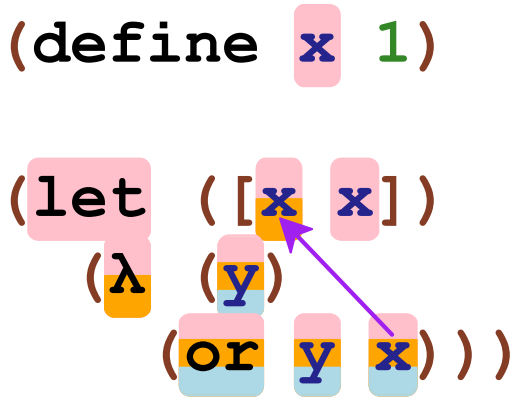

(define x 1)
(let ([x x])
(λ (y)
(or y x)))

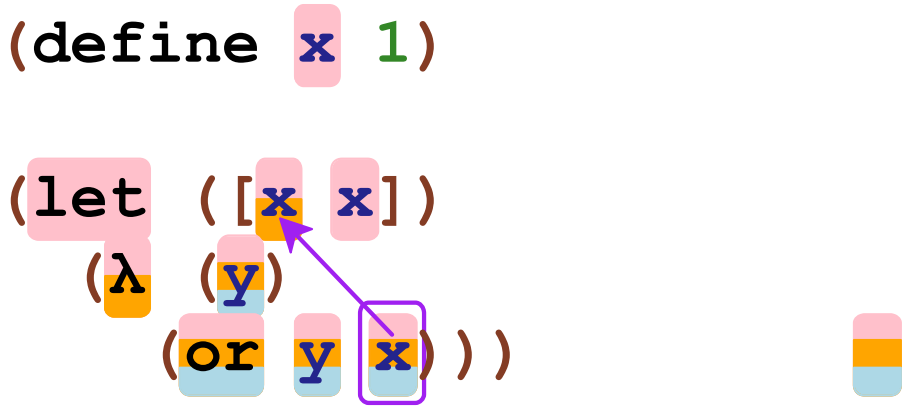

(define x 1)
(let ([x x])
(λ (y)
(or y x)))

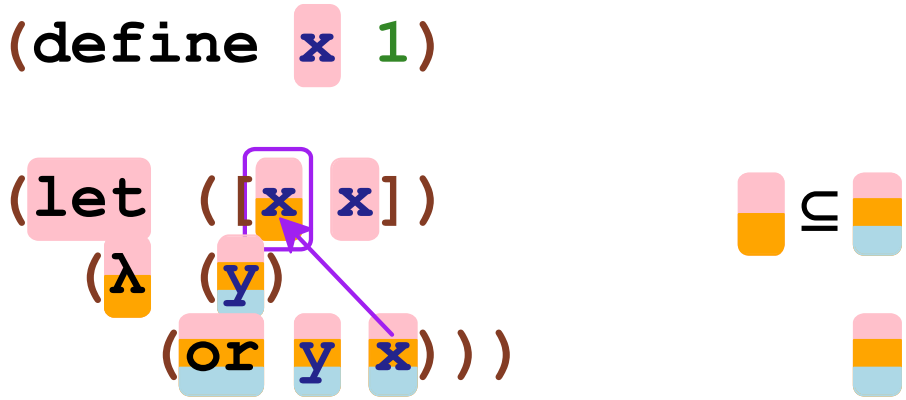

(define x 1)
(let ([x x])
(λ (y)
(or y x)))

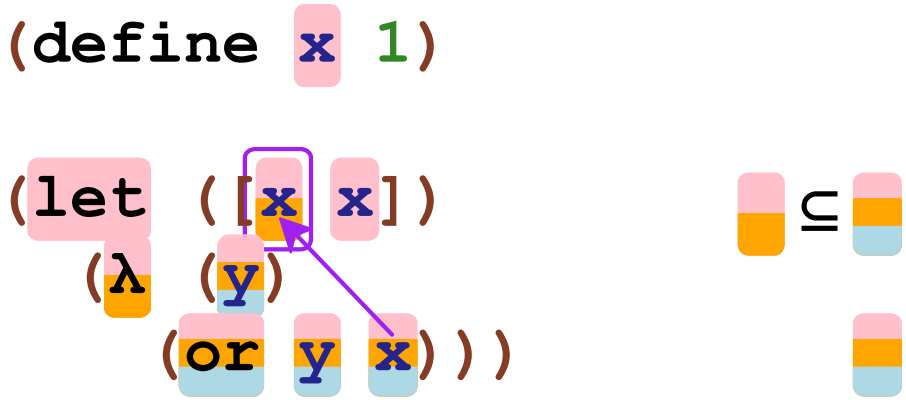

(define x 1)
(let ([x x])
(λ (y)
(or y x)))


(define x 1)
(let ([x x])
(λ (y)
(or y x)))

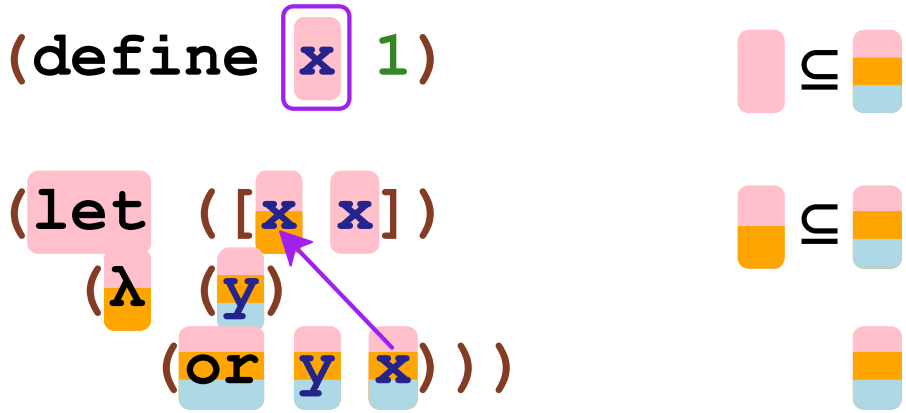

(define x 1)
(let ([x x])
(λ (y)
(or y x)))

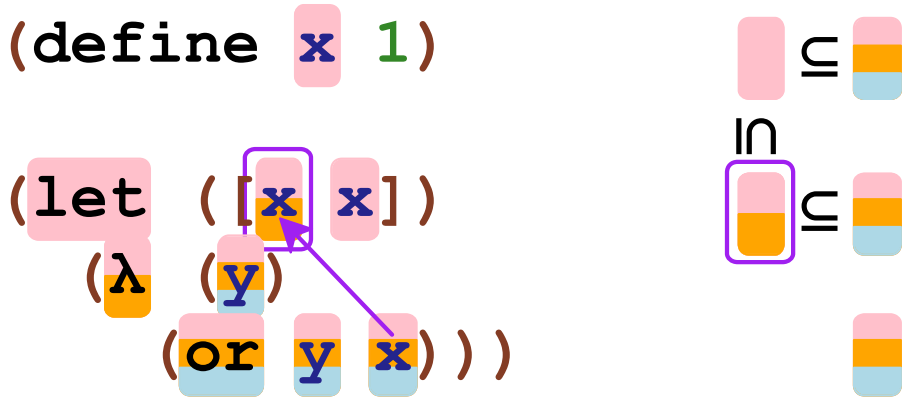

(define x 1)
(let ([x x])
(λ (y)
(or y x)))

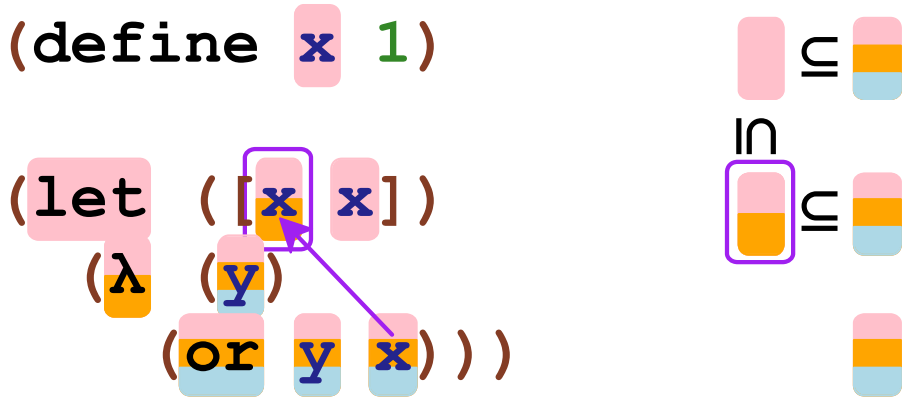

(define x 1)
(let ([x x])
(λ (y)
(or y x)))
⊆



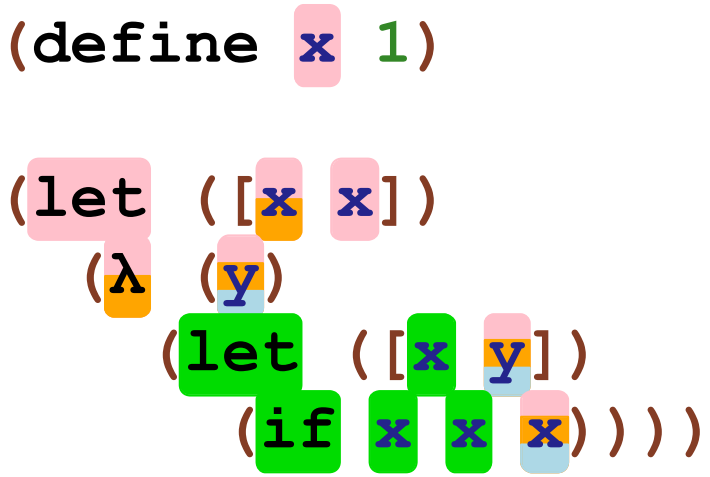
binding scopes ⊆ reference scopes

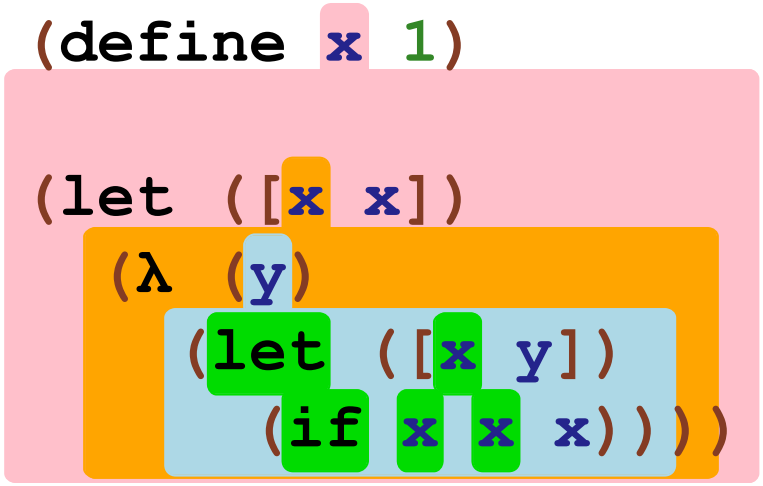

(define x 1)
(let ([x x])
(λ (y)
(or y x)))
⊆
⊆

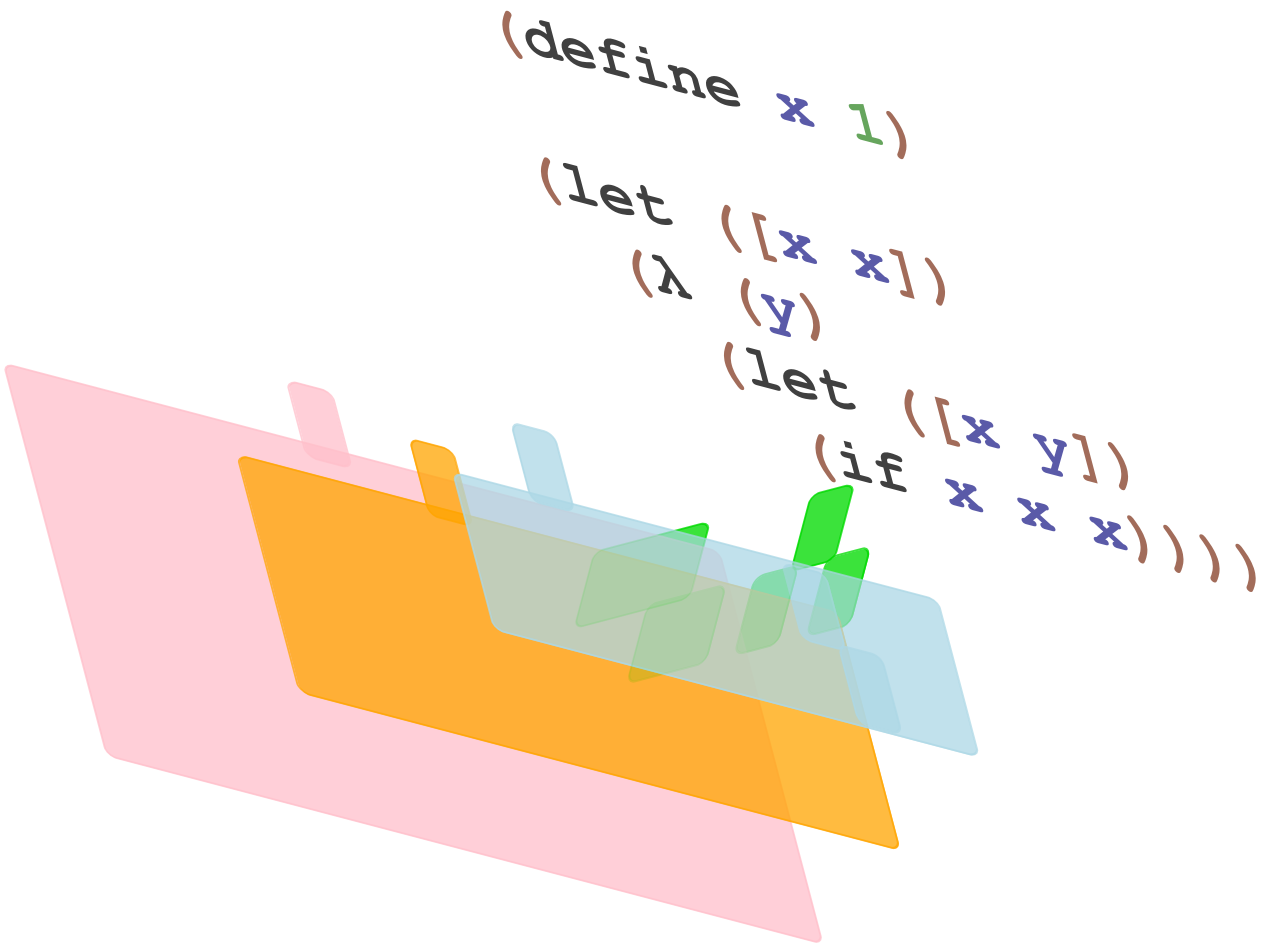

(define x 1)
(let ([x x])
(λ (y)
(or y x)))
⊆
∪
⊆

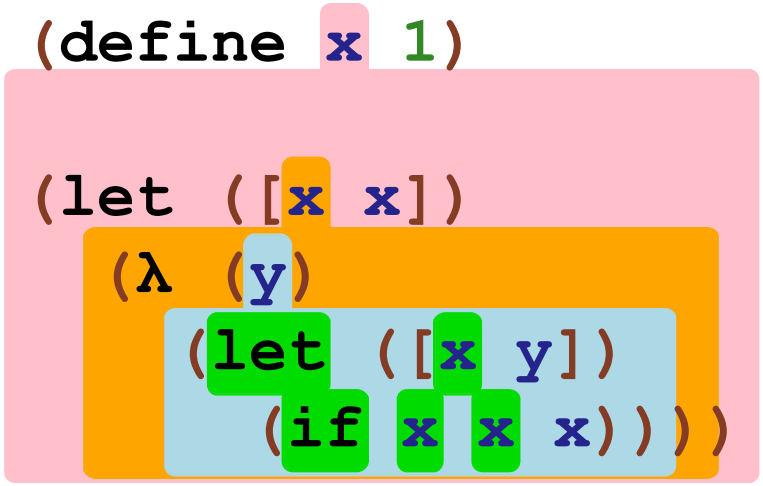


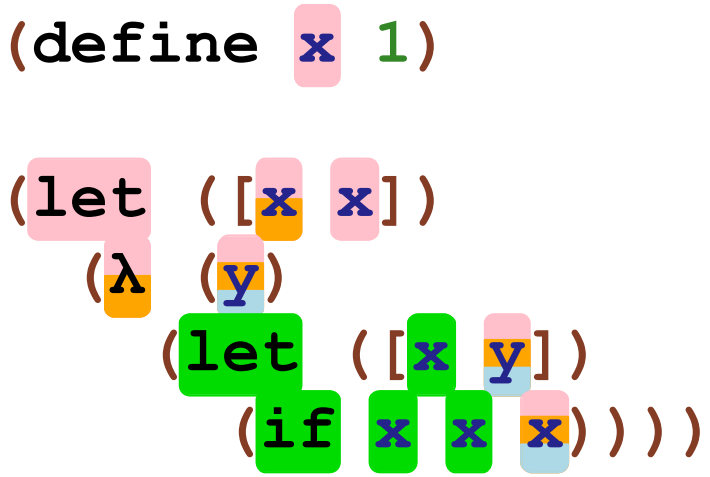
use candidate with *biggest* subset

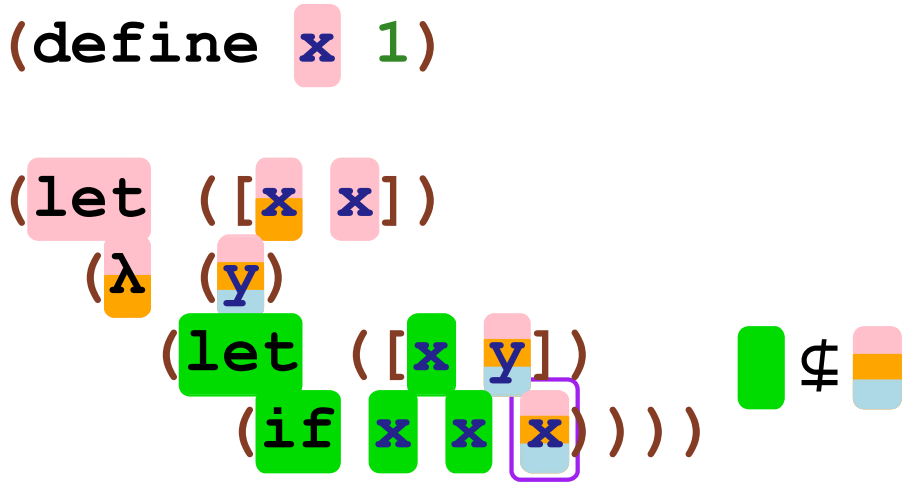

(define x 1)
(let ([x x])
(λ (y)
(let ([x y])
(if x x x))))

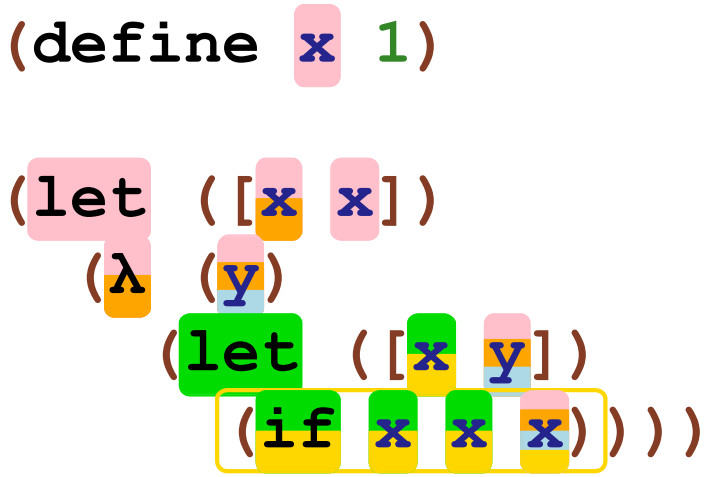

(define x 1)
(let ([x x])
(λ (y)
(let ([x y])
(if x x x))))


(define x 1)
(let ([x x])
(λ (y)
(let ([x y])
(if x x x))))


(define x 1)
(let ([x x])
(λ (y)
(let ([x y])
(if x x x))))


(define x 1)
(let ([x x])
(λ (y)
(let ([x y])
(if x x x))))


(define x 1)
(let ([x x])
(λ (y)
(let ([x y])
(if x x x))))
∉

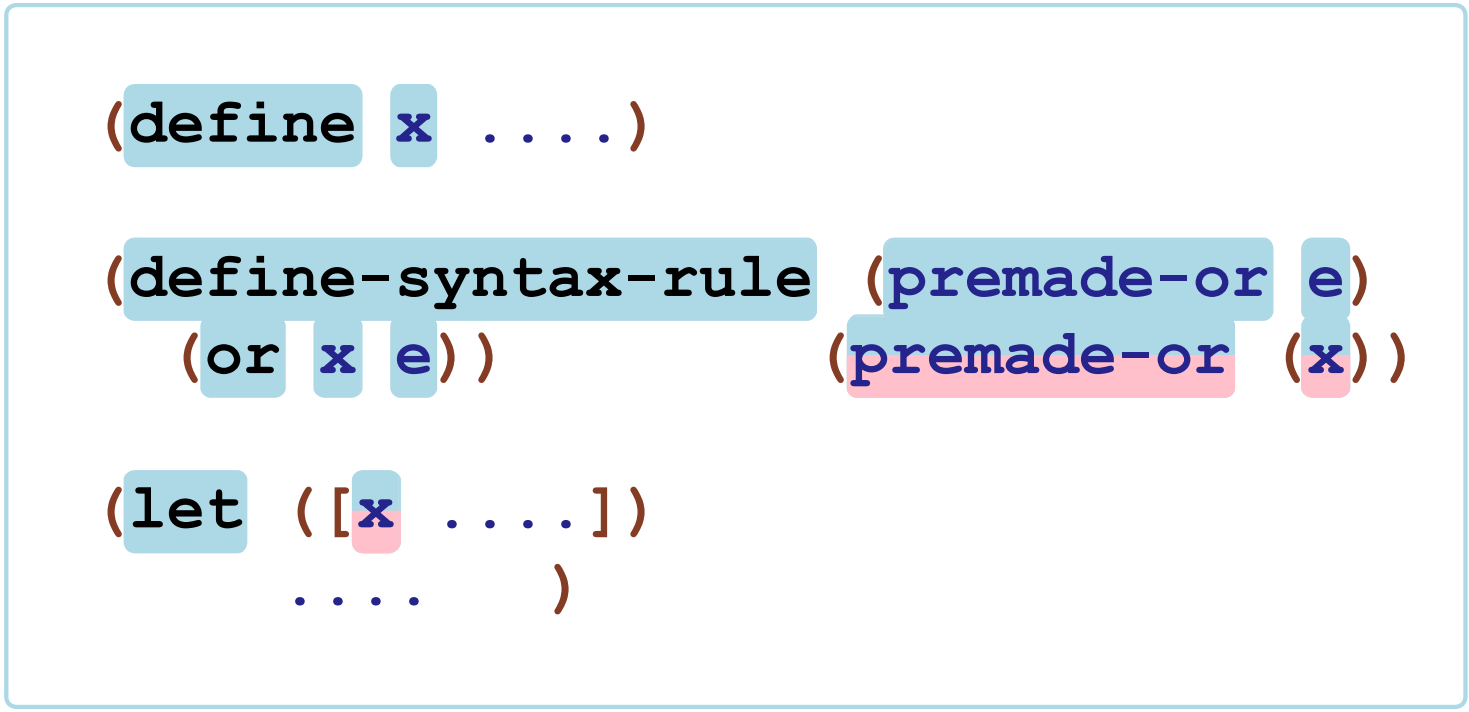


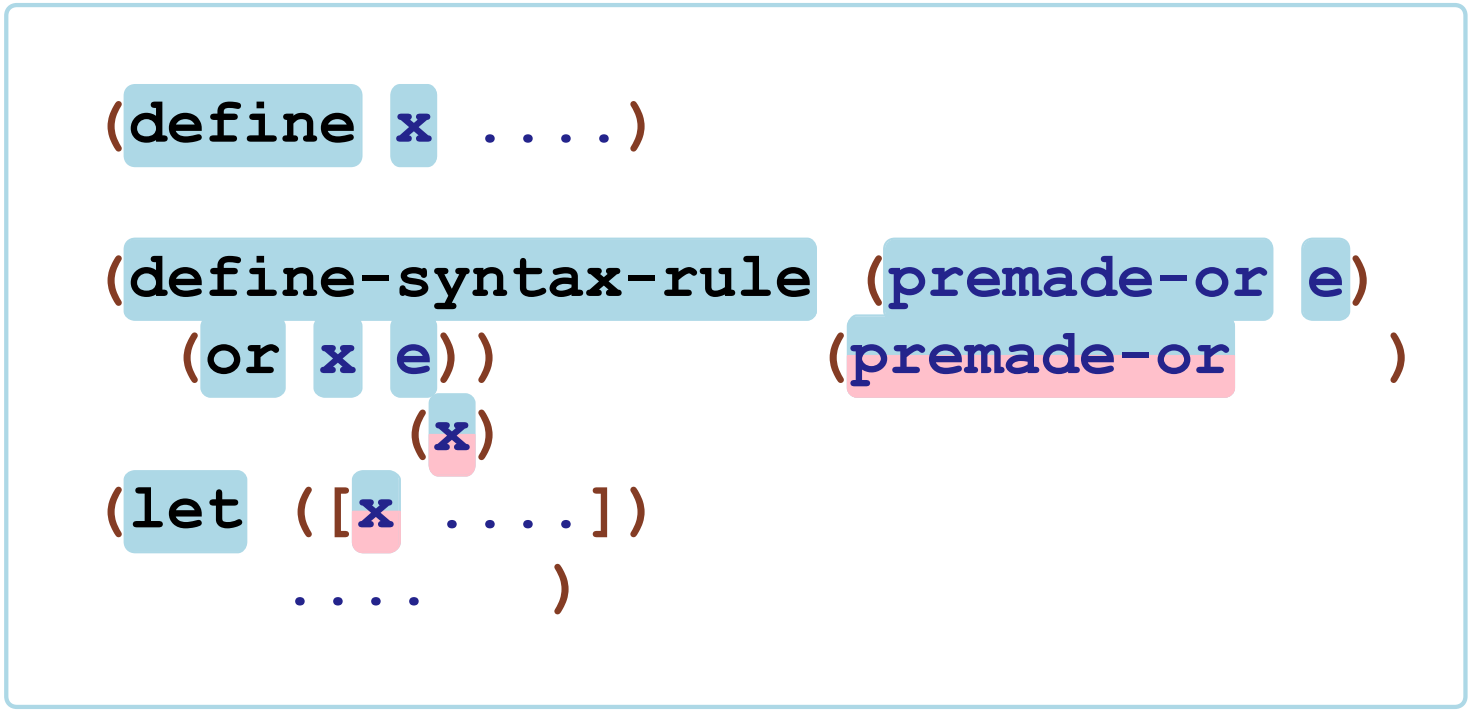
Expander introduces a fresh scope when it

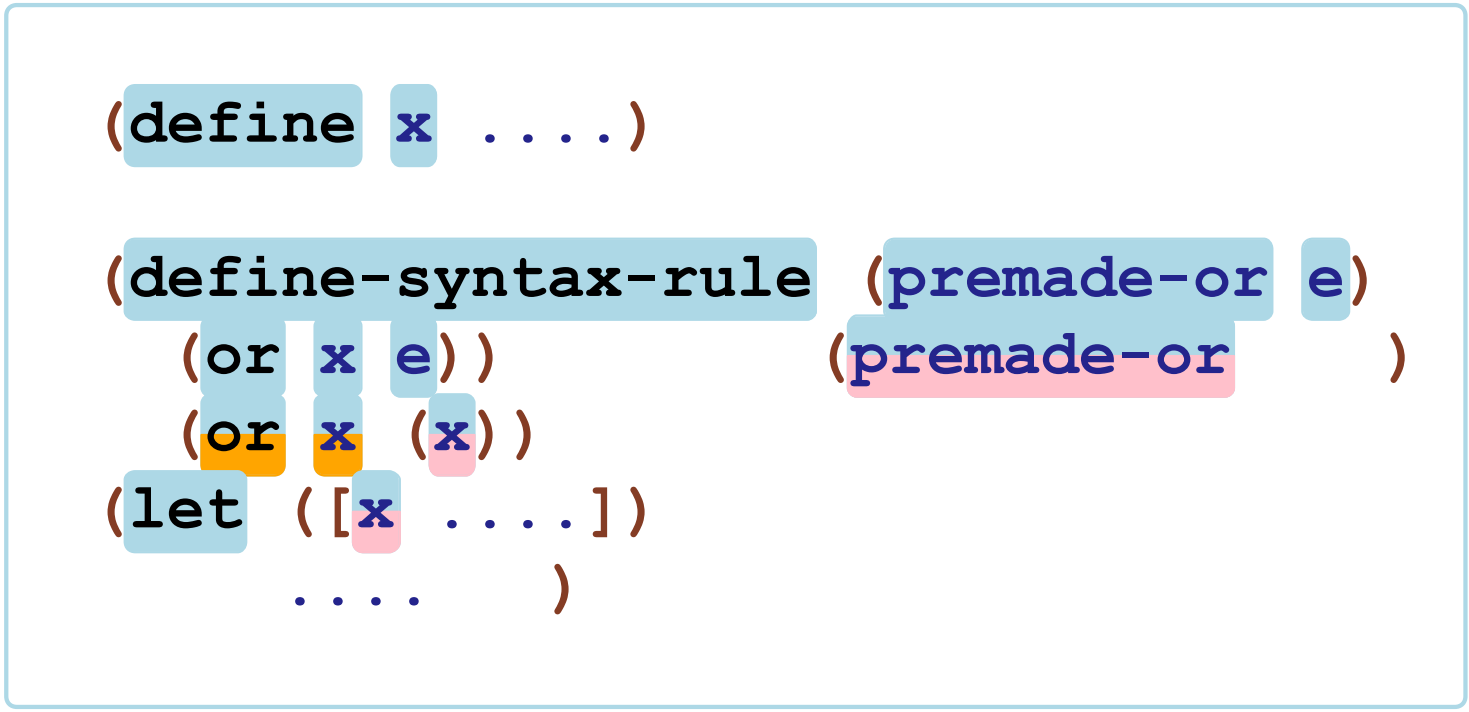
- expands a macro (e.g.,)
- finds a binding form (e.g., , , ,)

```
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (premade-or (x)))
```

```
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (premade-or (x)))
```


(define x ....)
(define-syntax-rule (premade-or e)
(or x e)) (premade-or (x))
(let ([x ....])
....)


(define x ....)
(define-syntax-rule (premade-or e)
(or x e))
(premade-or)
(x)
(let ([x ....])
....
)


(define x ....)
(define-syntax-rule (premade-or e)
(or x e))
(or x (x))
(let ([x ....])
....)
(premade-or)

```
(define x ....)

(define-syntax-rule (premade-or e)
  (or x e))

(let ([x ....])
  (or x (x)))
```

= scope

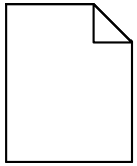
= scope

= scope set

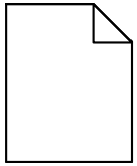
= scope

= scope set

= syntax object

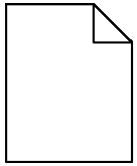


```
⟨expr⟩ ::= (lambda (⟨id⟩) ⟨expr⟩)                    function
         | ⟨id⟩                                      variable
         | (⟨expr⟩ ⟨expr⟩ ...)                       function call
         | (quote ⟨datum⟩)                           literal data
         | (let-syntax ([⟨id⟩ ⟨expr⟩])               macro binding
             ⟨expr⟩)
         | (quote-syntax ⟨datum⟩)                    literal syntax
```



```
⟨expr⟩ ::= (lambda (⟨id⟩) ⟨expr⟩)          function
         |  ⟨id⟩                            variable
         |  (⟨expr⟩ ⟨expr⟩ ...)             function call
         |  (quote ⟨datum⟩)                 literal data
         |  (let-syntax ([⟨id⟩ ⟨expr⟩])     macro binding
              ⟨expr⟩)
         |  (quote-syntax ⟨datum⟩)          literal syntax
```

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```



⟨expr⟩ ::= **(lambda (**⟨id⟩**)** ⟨expr⟩**)** *function*

| ⟨id⟩ *variable*

| **(**⟨expr⟩ ⟨expr⟩ **...)** *function call*

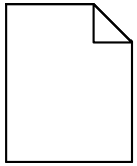
| **(quote** ⟨datum⟩**)** *literal data*

| **(let-syntax ([**⟨id⟩ ⟨expr⟩**])** *macro binding*
⟨expr⟩**)**

| **(quote-syntax** ⟨da *literal syntax*

(one)

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```



⟨expr⟩ ::= **(lambda (**⟨id⟩**)** ⟨expr⟩**)** *function*

| ⟨id⟩ *variable*

| **(**⟨expr⟩ ⟨expr⟩ **...)** *function call*

| **(quote** ⟨datum⟩**)** *literal data*

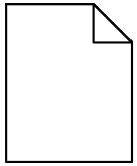
| **(let-syntax ([**⟨id⟩ ⟨expr⟩**])** *macro binding*
⟨expr⟩**)**

| **(quote-syntax** ⟨da *literal syntax*

(one)

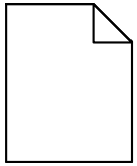
```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

expands to `'1`



⟨expr⟩ ::=	**(lambda (**⟨id⟩**)** ⟨expr⟩**)**	*function*
|	⟨id⟩	*variable*
|	**(**⟨expr⟩ ⟨expr⟩ **...)**	*function call*
|	**(quote** ⟨datum⟩**)**	*literal data*
|	**(let-syntax ([**⟨id⟩ ⟨expr⟩**])** ⟨expr⟩**)**	*macro binding*
|	**(quote-syntax** ⟨datum⟩**)**	*literal syntax*

```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk '1))
```



⟨expr⟩ ::= **(lambda (⟨id⟩) ⟨expr⟩)** *function*

| ⟨id⟩ *variable*

| **(⟨expr⟩ ⟨expr⟩ ...)** *function call*

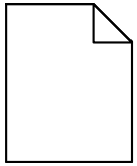
| **(quote ⟨datum⟩)** *literal data*

| **(let-syntax ([⟨id⟩ ⟨expr⟩]) ⟨expr⟩)** *macro binding*

| **(quote-syn** *literal syntax*

(thunk '1)

```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk '1))
```



⟨expr⟩ ::= **(lambda (**⟨id⟩**)** ⟨expr⟩**)** *function*

| ⟨id⟩ *variable*

| **(**⟨expr⟩ ⟨expr⟩ **...)** *function call*

| **(quote** ⟨datum⟩**)** *literal data*

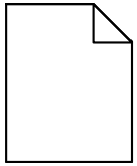
| **(let-syntax ([**⟨id⟩ ⟨expr⟩**])** *macro binding*
⟨expr⟩**)**

| **(quote-syn** *literal syntax*

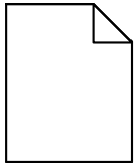
(thunk '1)

```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk '1))
```

expands to **(lambda (x) '1)**



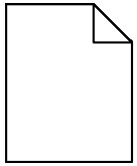
Part 1 - Representing Syntax



Syntax Objects

Combine a symbol with a set of scopes

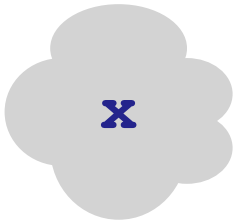
```
(struct syntax (e scopes) #:transparent)
```

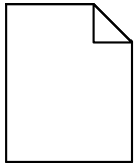


Syntax Objects

Combine a symbol with a set of scopes

```
(struct syntax (e scopes) #:transparent)
```

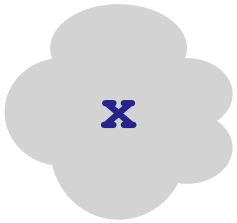




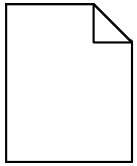
Syntax Objects

Combine a symbol with a set of scopes

```
(struct syntax (e scopes) #:transparent)
```



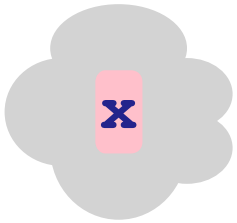

```
(syntax 'x (set))
```



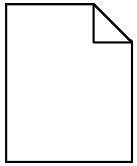
Syntax Objects

Combine a symbol with a set of scopes

```
(struct syntax (e scopes) #:transparent)
```



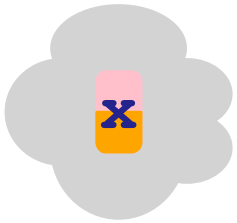

```
(syntax 'x (set sc1))
```



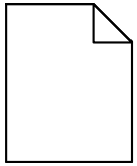
Syntax Objects

Combine a symbol with a set of scopes

```
(struct syntax (e scopes) #:transparent)
```




```
(syntax 'x (set sc1 sc2))
```

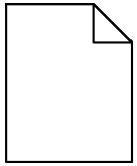


Syntax Objects

Combine a symbol with a set of scopes

```
(struct syntax (e scopes) #:transparent)
```

```
(syntax? (syntax 'x (set)))
⇒ #t
```

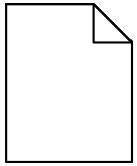


Syntax Objects

Combine a symbol with a set of scopes

```
(struct syntax (e scopes) #:transparent)
```

```
(syntax? 'x)
⇒ #f
```

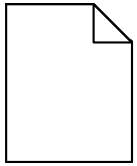


Syntax Objects

Combine a symbol with a set of scopes

```
(struct syntax (e scopes) #:transparent)
```

```
(syntax-e (syntax 'x (set)))
⇒ 'x
```

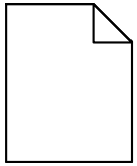


Syntax Objects

Combine a symbol with a set of scopes

```
(struct syntax (e scopes) #:transparent)
```

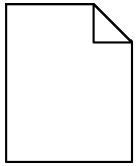
```
(syntax-scopes (syntax 'x (set)))
⇒ (set)
```



Syntax Objects

All syntax object are identifiers

```
(define (identifier? s)
  (syntax? s))
```

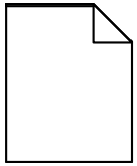


Syntax Objects

All syntax object are identifiers

```
(define (identifier? s)
  (syntax? s))
```

```
(identifier? (syntax 'x (set)))
⇒ #t
```

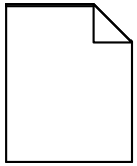


Syntax Objects

`datum->syntax` coerces to syntax with no scopes

leaving existing syntax as-is

```
(define (datum->syntax v)
  (cond
   [(syntax? v) v]
   [(symbol? v) (syntax v (set))]
   [(list? v) (map datum->syntax v)]
   [else v]))
```



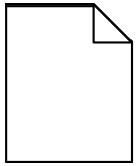
Syntax Objects

`datum->syntax` coerces to syntax with no scopes

leaving existing syntax as-is

```
(define (datum->syntax v)
  (cond
   [(syntax? v) v]
   [(symbol? v) (syntax v (set))]
   [(list? v) (map datum->syntax v)]
   [else v]))
```

```
(datum->syntax 'a)
⇒ (syntax 'a (set))
```



Syntax Objects

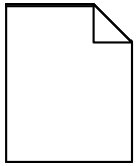
`datum->syntax` coerces to syntax with no scopes

leaving existing syntax as-is

```
(define (datum->syntax v)
  (cond
   [(syntax? v) v]
   [(symbol? v) (syntax v (set))]
   [(list? v) (map datum->syntax v)]
   [else v]))
```

```
(datum->syntax 1)
```

⇒ `1`



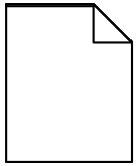
Syntax Objects

`datum->syntax` coerces to syntax with no scopes

leaving existing syntax as-is

```
(define (datum->syntax v)
  (cond
   [(syntax? v) v]
   [(symbol? v) (syntax v (set))]
   [(list? v) (map datum->syntax v)]
   [else v]))
```

```
(datum->syntax '(a b c))
⇒ (list (syntax 'a (set))
        (syntax 'b (set))
        (syntax 'c (set)))
```



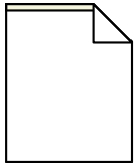
Syntax Objects

`datum->syntax` coerces to syntax with no scopes

leaving existing syntax as-is

```
(define (datum->syntax v)
  (cond
   [(syntax? v) v]
   [(symbol? v) (syntax v (set))]
   [(list? v) (map datum->syntax v)]
   [else v]))
```

```
(datum->syntax (list 'a
                     (syntax 'b (set sc1))
                     'c))
⇒ (list (syntax 'a (set))
        (syntax 'b (set sc1))
        (syntax 'c (set)))
```

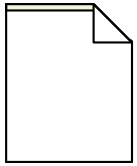


Syntax Objects

`syntax->datum` discards scopes

produces a plain S-expression

```
(define (syntax->datum s)
  (cond
   [(syntax? s) (syntax-e s)]
   [(list? s) (map syntax->datum s)]
   [else s]))
```



Syntax Objects

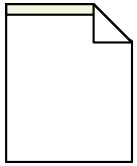
`syntax->datum` discards scopes

produces a plain S-expression

```
(define (syntax->datum s)
  (cond
   [(syntax? s) (syntax-e s)]
   [(list? s) (map syntax->datum s)]
   [else s]))
```

```
(syntax->datum (datum->syntax '(a b c)))
```

⇒ `'(a b c)`

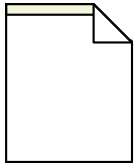


Scopes

Scope is an empty record

identity is based on **`eq?`**

```
(struct scope ())
```



Scopes

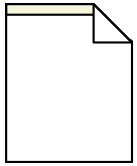
Scope is an empty record

identity is based on **eq?**

```
(struct scope ())
```

```
(define sc1 (scope))
(define sc2 (scope))
```

```
(eq? sc1 sc2)
⇒ #f
```



Scopes

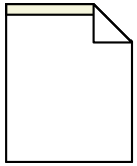
Scope is an empty record

identity is based on **eq?**

```
(struct scope ())
```

```
(define sc1 (scope))
(define sc2 (scope))
```

```
(eq? sc1 sc1)
⇒ #t
```



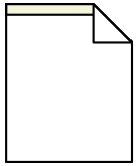
Scopes

Scope is an empty record

identity is based on **`eq?`**

```
(struct scope ())
```

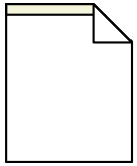
```
(define sc1 (scope))
(define sc2 (scope))

(set sc1 sc2)
```



Scopes

Add a scope everywhere, including in nested

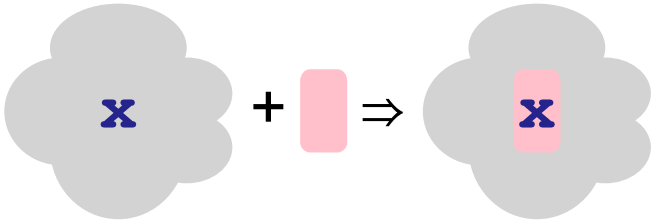
```
(define (add-scope s sc)
  (cond
   [(syntax? s)
    (syntax (syntax-e s)
            (set-add (syntax-scopes s) sc))]
   [(list? s)
    (map (lambda (e) (add-scope e sc)) s)]
   [else s]))
```



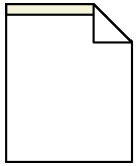
Scopes

Add a scope everywhere, including in nested

```
(define (add-scope s sc)
  (cond
   [(syntax? s)
    (syntax (syntax-e s)
            (set-add (syntax-scopes s) sc))]
   [(list? s)
    (map (lambda (e) (add-scope e sc)) s)]
   [else s]))
```




```
(add-scope (syntax 'x (set)) sc1)
⇒ (syntax 'x (set sc1))
```

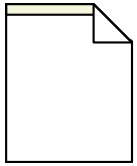


Scopes

Add a scope everywhere, including in nested

```
(define (add-scope s sc)
  (cond
   [(syntax? s)
    (syntax (syntax-e s)
            (set-add (syntax-scopes s) sc))]
   [(list? s)
    (map (lambda (e) (add-scope e sc)) s)]
   [else s]))
```

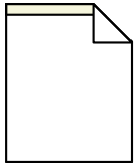
```
(add-scope (datum->syntax '(x (y))) sc1)
⇒ (list (syntax 'x (set sc1))
        (list (syntax 'y (set sc1))))
```



Scopes

Add a scope everywhere, including in nested

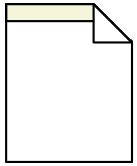
```
(define (add-scope s sc)
  (cond
   [(syntax? s)
    (syntax (syntax-e s)
            (set-add (syntax-scopes s) sc))]
   [(list? s)
    (map (lambda (e) (add-scope e sc)) s)]
   [else s]))
```



Scopes

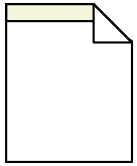
Adjust a scope everywhere, including in nested

```
(define (adjust-scope s sc op)
  (cond
   [(syntax? s)
    (syntax (syntax-e s)
            (op (syntax-scopes s) sc))]
   [(list? s)
    (map (lambda (e) (adjust-scope e sc op)) s)]
   [else s]))
```



Scopes

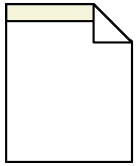
```
(define (add-scope s sc)
  (adjust-scope s sc set-add))

(define (flip-scope s sc)
  (adjust-scope s sc set-flip))

(define (set-flip s e)
  (if (set-member? s e)
      (set-remove s e)
      (set-add s e)))
```



Scopes

```
(define (add-scope s sc)
  (adjust-scope s sc set-add))

(define (flip-scope s sc)
  (adjust-scope s sc set-flip))

(define (set-flip s e)
  (if (set-member? s e)
      (set-remove s e)
      (set-add s e)))
```

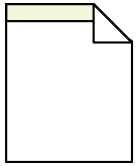
```
(add-scope (syntax 'x (set sc1))
           sc2)
⇒ (syntax 'x (set sc1 sc2))
```



Scopes

```
(define (add-scope s sc)
  (adjust-scope s sc set-add))

(define (flip-scope s sc)
  (adjust-scope s sc set-flip))

(define (set-flip s e)
  (if (set-member? s e)
      (set-remove s e)
      (set-add s e)))
```

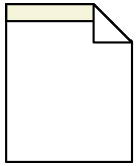
```
(add-scope (syntax 'x (set sc1))
           sc1)
⇒ (syntax 'x (set sc1))
```



Scopes

```
(define (add-scope s sc)
  (adjust-scope s sc set-add))

(define (flip-scope s sc)
  (adjust-scope s sc set-flip))

(define (set-flip s e)
  (if (set-member? s e)
      (set-remove s e)
      (set-add s e)))
```

```
(flip-scope (syntax 'x (set sc1))
            sc2)
⇒ (syntax 'x (set sc1 sc2))
```

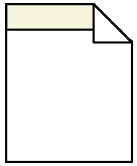


Scopes

```
(define (add-scope s sc)
  (adjust-scope s sc set-add))

(define (flip-scope s sc)
  (adjust-scope s sc set-flip))

(define (set-flip s e)
  (if (set-member? s e)
      (set-remove s e)
      (set-add s e)))
```

```
(flip-scope (syntax 'x (set sc1 sc2))
            sc2)
```

⇒ `(syntax 'x (set sc1))`

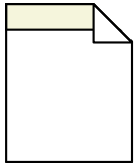


Global Binding Table

A binding is either

- symbol = core form or primitive
- gensym = local binding

```
(define all-bindings (make-hash))

(define (add-binding! id binding)
  (hash-set! all-bindings id binding))
```



Global Binding Table

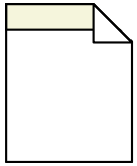
A binding is either
- symbol = core form or primitive
- gensym = local binding

```
(define all-bindings (make-hash))

(define (add-binding! id binding)
  (hash-set! all-bindings id binding))
```

```
(let ([a '1])
  (let ([z '2])
    ....))
```

```
(define loc/a (gensym))
```

```
(add-binding! (syntax 'a (set sc1)) loc/a)
```



Global Binding Table

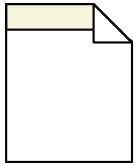
A binding is either
- symbol = core form or primitive
- gensym = local binding

```
(define all-bindings (make-hash))

(define (add-binding! id binding)
  (hash-set! all-bindings id binding))
```

```
(let ([b '1])
  (let ([b '2])
    ....))
```

```
(define loc/b-out (gensym))
(define loc/b-in (gensym))
```

```
(add-binding! (syntax 'b (set sc1)) loc/b-out)
(add-binding! (syntax 'b (set sc1 sc2)) loc/b-in)
```



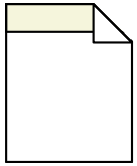
Global Binding Table

A binding is either
- symbol = core form or primitive
- gensym = local binding

```
(define all-bindings (make-hash))

(define (add-binding! id binding)
  (hash-set! all-bindings id binding))
```

```
(list
 (let ([c '1]) ...)
 (let ([c '2]) ...))
```

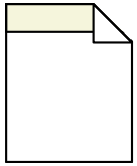
```
(define loc/c1 (gensym))
(define loc/c2 (gensym))

(add-binding! (syntax 'c (set sc1)) loc/c1)
(add-binding! (syntax 'c (set sc2)) loc/c2)
```



Global Binding Table

`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
    (check-unambiguous max-id candidate-ids)
    (hash-ref all-bindings max-id)]
   [else #f]))
```



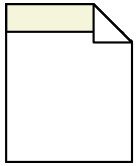
Global Binding Table

`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
    (unambiguous max-id candidate-ids)
    (binding-bindings max-id)]
```

```
(let ([a '1])
  (let ([z '2])
    ....))
```

```
(resolve (syntax 'a (set sc1)))
⇒ loc/a
```



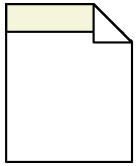
Global Binding Table

`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
                    us max-id candidate-ids)
                    ndings max-id)]
```

```
(let ([a '1])
  (let ([z '2])
    ....))
```

```
(resolve (syntax 'a (set sc1 sc2)))
⇒ loc/a
```



Global Binding Table

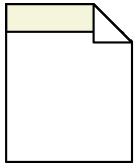
`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
    (unambiguous max-id candidate-ids)
    (binding-of (find-bindings max-id)]
```

```
(let ([a '1])
  (let ([z '2])
    ....))
```

```
(resolve (syntax 'a (set sc2)))
```

⇒ `#f`



Global Binding Table

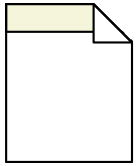
`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
    (unambiguous max-id candidate-ids)
    (binding-in (find-bindings max-id)]
```

```
(let ([b '1])
  (let ([b '2])
    ....))
```

```
(resolve (syntax 'b (set sc1)))
```

⇒ `loc/b-out`



Global Binding Table

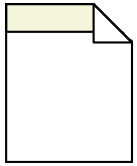
`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
                    us max-id candidate-ids)
                    ndings max-id)]
```

```
(let ([b '1])
  (let ([b '2])
    ....))
```

```
(resolve (syntax 'b (set sc1 sc2)))
```

⇒ `loc/b-in`



Global Binding Table

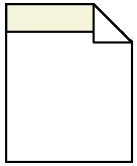
`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
    (unambiguous max-id candidate-ids)
    (binding-bindings max-id)]
```

```
(let ([b '1])
  (let ([b '2])
    ....))
```

```
(resolve (syntax 'b (set sc2)))
```

⇒ `#f`



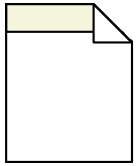
Global Binding Table

`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
                    -id candidate-ids)
                    max-id)]
```

```
(list
 (let ([c '1]) ...)
 (let ([c '2]) ...))
```

```
(resolve (syntax 'c (set sc1)))
⇒ loc/c1
```



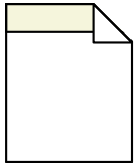
Global Binding Table

`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
                    -id candidate-ids)
                    max-id)]
```

```
(list
 (let ([c '1]) ...)
 (let ([c '2]) ...))
```

```
(resolve (syntax 'c (set sc2)))
⇒ loc/c2
```



Global Binding Table

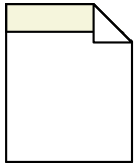
`resolve` finds the binding for an identifier

```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
                    -id candidate-ids)
                    max-id)]
```

```
(list
 (let ([c '1]) ...)
 (let ([c '2]) ...))
```

```
(resolve (syntax 'c (set sc1 sc2)))
```

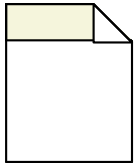
⇒ *error: ambiguous*



Global Binding Table

`resolve` finds the binding for an identifier

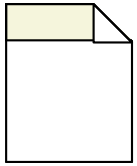
```
(define (resolve id)
  (define candidate-ids
    (find-all-matching-bindings id))
  (cond
   [(pair? candidate-ids)
    (define max-id
      (argmax (compose set-count syntax-scopes)
              candidate-ids))
    (check-unambiguous max-id candidate-ids)
    (hash-ref all-bindings max-id)]
   [else #f]))
```



Global Binding Table

Helper: find candidates as bindings with a subset of scopes

```
(define (find-all-matching-bindings id)
  (for/list ([c-id (in-hash-keys all-bindings)]
             #:when (and (eq? (syntax-e c-id) (syntax-e id))
                         (subset? (syntax-scopes c-id)
                                  (syntax-scopes id))))
    c-id))
```



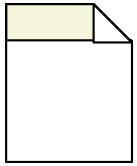
Global Binding Table

Helper: find candidates as bindings with a subset of scopes

```
(define (find-all-matching-bindings id)
  (for/list ([c-id (in-hash-keys all-bindings)]
             #:when (and (eq? (syntax-e c-id) (syntax-e id))
                         (subset? (syntax-scopes c-id)
                                  (syntax-scopes id))))
    c))
```

```
(let ([a '1])
  (let ([z '2])
    ....))
```

```
(find-all-matching-bindings
 (syntax 'a (set sc1)))
⇒ (list (syntax 'a (set sc1)))
```



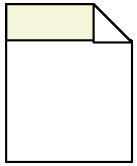
Global Binding Table

Helper: find candidates as bindings with a subset of scopes

```
(define (find-all-matching-bindings id)
  (for/list ([c-id (in-hash-keys all-bindings)]
             #:when (and (eq? (syntax-e c-id) (syntax-e id))
                         (subset? (syntax-scopes c-id)
                                  (syntax-scopes id))))
    c-id))
```

```
(let ([a '1])
  (let ([z '2])
    ....))
```

```
(find-all-matching-bindings
 (syntax 'a (set sc2)))
⇒ (list)
```



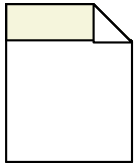
Global Binding Table

Helper: find candidates as bindings with a subset of scopes

```
(define (find-all-matching-bindings id)
  (for/list ([c-id (in-hash-keys all-bindings)]
             #:when (and (eq? (syntax-e c-id) (syntax-e id))
                         (subset? (syntax-scopes c-id)
                                  (syntax-scopes id))))
    c-id))
```

```
(let ([a '1])
  (let ([z '2])
    ....))
```

```
(find-all-matching-bindings
 (syntax 'a (set sc1 sc2)))
⇒ (list (syntax 'a (set sc1)))
```



Global Binding Table

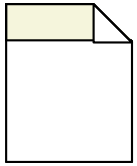
Helper: find candidates as bindings with a subset of scopes

```
(define (find-all-matching-bindings id)
  (for/list ([c-id (in-hash-keys all-bindings)]
             #:when (and (eq? (syntax-e c-id) (syntax-e id))
                         (subset? (syntax-scopes c-id)
                                  (syntax-scopes id))))
    c))
```

```
(let ([b '1])
  (let ([b '2])
    ....))
```

```
(list->set
 (find-all-matching-bindings
  (syntax 'b (set sc1 sc2))))
```

```
⇒ (set (syntax 'b (set sc1))
       (syntax 'b (set sc1 sc2)))
```



Global Binding Table

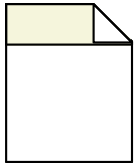
Helper: find candidates as bindings with a subset of scopes

```
(define (find-all-matching-bindings id)
  (for/list ([c-id (in-hash-keys all-bindings)]
             #:when (and (eq? (syntax-e c-id) (syntax-e id))
                         (subset? (syntax-scopes c-id)
                                  (syntax-scopes id))))
    c))
```

```
(list
 (let ([c '1]) ...)
 (let ([c '2]) ...))
```

```
(list->set
 (find-all-matching-bindings
  (syntax 'c (set sc1 sc2))))
```

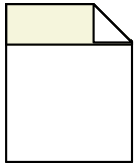
```
⇒ (set (syntax 'c (set sc1))
       (syntax 'c (set sc2)))
```



Global Binding Table

Helper: check that max has a superset for each candidate

```
(define (check-unambiguous max-id candidate-ids)
  (for ([c-id (in-list candidate-ids)])
    (unless (subset? (syntax-scopes c-id)
                     (syntax-scopes max-id))
      (error "ambiguous:" max-id))))
```

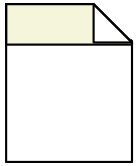


Global Binding Table

Helper: check that max has a superset for each candidate

```
(define (check-unambiguous max-id candidate-ids)
  (for ([c-id (in-list candidate-ids)])
    (unless (subset? (syntax-scopes c-id)
                     (syntax-scopes max-id))
      (error "ambiguous:" max-id))))
```

```
(check-unambiguous
 (syntax 'b (set sc1 sc2))
 (list (syntax 'b (set sc1))
       (syntax 'b (set sc1 sc2))))
```



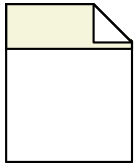
Global Binding Table

Helper: check that max has a superset for each candidate

```
(define (check-unambiguous max-id candidate-ids)
  (for ([c-id (in-list candidate-ids)])
    (unless (subset? (syntax-scopes c-id)
                     (syntax-scopes max-id))
      (error "ambiguous:" max-id))))
```

```
(check-unambiguous
 (syntax 'c (set sc2))
 (list (syntax 'c (set sc1))
       (syntax 'c (set sc2))))
```

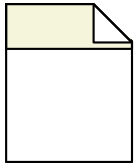
⇒ *error: ambiguous*



Core Forms and Primitives

All core bindings in `core-scope`

```
(define core-scope (scope))

(define core-forms
  (set 'lambda 'let-syntax 'quote 'quote-syntax))
(define core-primitives
  (set 'datum->syntax 'syntax->datum 'syntax-e
       'list 'cons 'first 'second 'rest 'map))

(for ([sym (set-union core-forms core-primitives)])
  (add-binding! (syntax sym (set core-scope)) sym))
```

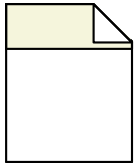


Core Forms and Primitives

All core bindings in `core-scope`

```
(define core-scope (scope))

(define core-forms
  (set 'lambda 'let-syntax 'quote 'quote-syntax))
(define core-primitives
  (set 'datum->syntax 'syntax->datum 'syntax-e
       'list 'cons 'first 'second 'rest 'map))

(for ([sym (set-union core-forms core-primitives)])
  (add-binding! (syntax sym (set core-scope)) sym))
```

```
(resolve (datum->syntax 'lambda))
⇒ #f
```



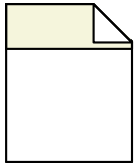
Core Forms and Primitives

All core bindings in **`core-scope`**

```
(define core-scope (scope))

(define core-forms
  (set 'lambda 'let-syntax 'quote 'quote-syntax))
(define core-primitives
  (set 'datum->syntax 'syntax->datum 'syntax-e
       'list 'cons 'first 'second 'rest 'map))

(for ([sym (set-union core-forms core-primitives)])
  (add-binding! (syntax sym (set core-scope)) sym))
```

```
(resolve (add-scope (datum->syntax 'lambda)
                    core-scope))
```

⇒ `'lambda`



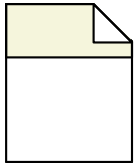
Core Forms and Primitives

All core bindings in `core-scope`

```
(define core-scope (scope))

(define core-forms
  (set 'lambda 'let-syntax 'quote 'quote-syntax))
(define core-primitives
  (set 'datum->syntax 'syntax->datum 'syntax-e
       'list 'cons 'first 'second 'rest 'map))

(for ([sym (set-union core-forms core-primitives)])
  (add-binding! (syntax sym (set core-scope)) sym))
```

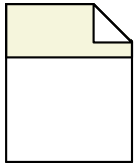
```
(resolve (add-scope (datum->syntax 'cons)
                    core-scope))
```

⇒ `'cons`



Importing Core Bindings

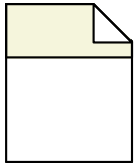
```
(define (introduce s)
  (add-scope s core-scope))
```



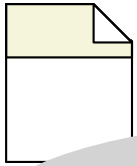
Importing Core Bindings

```
(define (introduce s)
  (add-scope s core-scope))
```

```
(introduce
 (datum->syntax 'cons))
⇒ (syntax 'cons (set core-scope))
```



Part 2 - Expander Dispatch

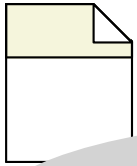


Expanding Macros

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

expands to

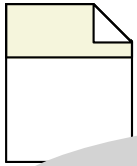
```
'1
```



Expanding Macros

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

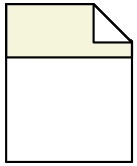
```
(define one-prog
  (introduce
   (datum->syntax
    '(let-syntax ([one (lambda (stx)
                         (quote-syntax '1))])
       (one)))))

(syntax->datum (expand one-prog))
⇒ '(quote 1)
```



Expanding Macros

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

```
(define one-prog
  (introduce
   (datum->syntax
    '(let-syntax ([one (lambda (stx)
                         (quote-syntax '1))])
       (one)))))

(expand one-prog)
⇒ (list (syntax 'quote ....core-scope....) 1)
```



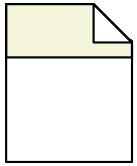
Expanding Function Calls

```
(list (one) '2)
```

expands to

```
(list '1 '2)
```

```
(expand (introduce
         (datum->syntax '(list '1 '2))))
⇒ (list
   (syntax 'list (set core-scope))
   (list (syntax 'quote (set core-scope)) 1)
   (list (syntax 'quote (set core-scope)) 2))
```



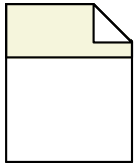
Expanding Binding Forms

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

expands to

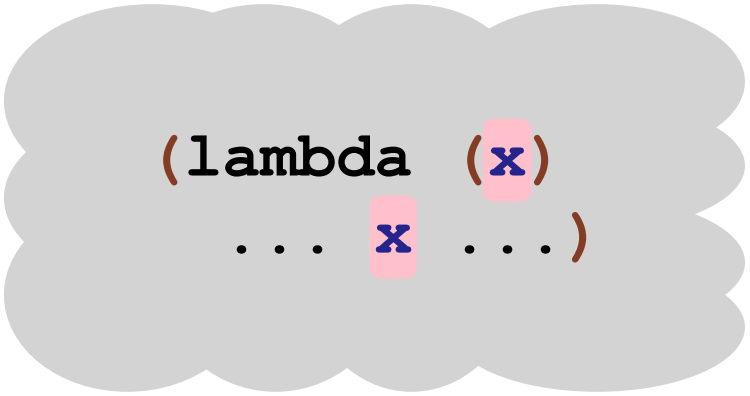
```
(lambda (x) x)
```

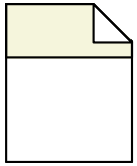
```
(expand (introduce
          (datum->syntax '(lambda (x) x))))
⇒ (list
    (syntax 'lambda (set core-scope))
    (list (syntax 'x (set core-scope sc1)))
    (syntax 'x (set core-scope sc1)))
```



Expander and Bindings

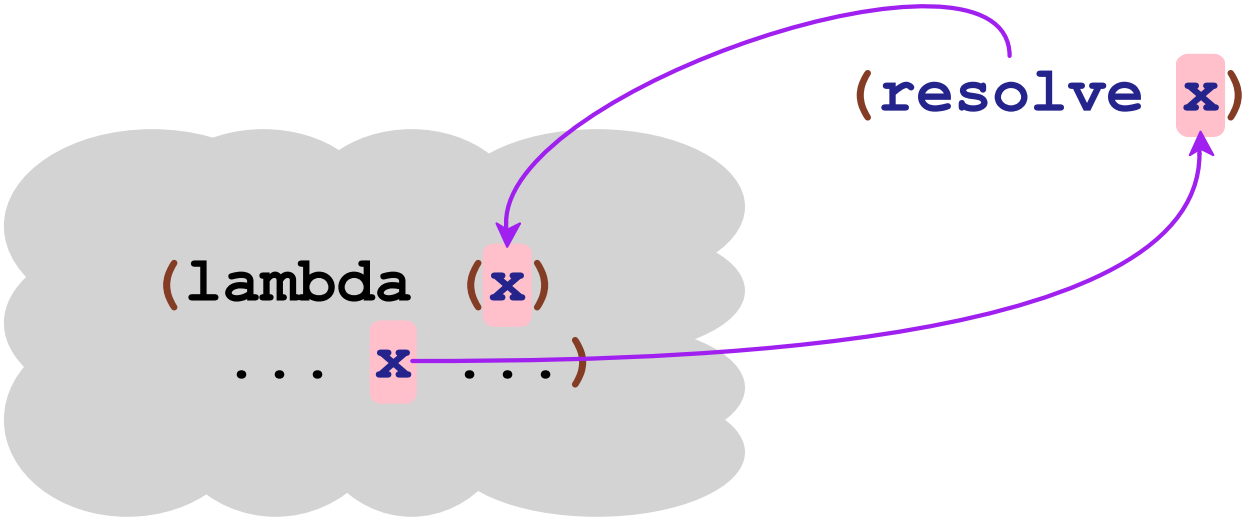
Binding table helps the expander connect *use* to *binding*

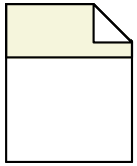




Expander and Bindings

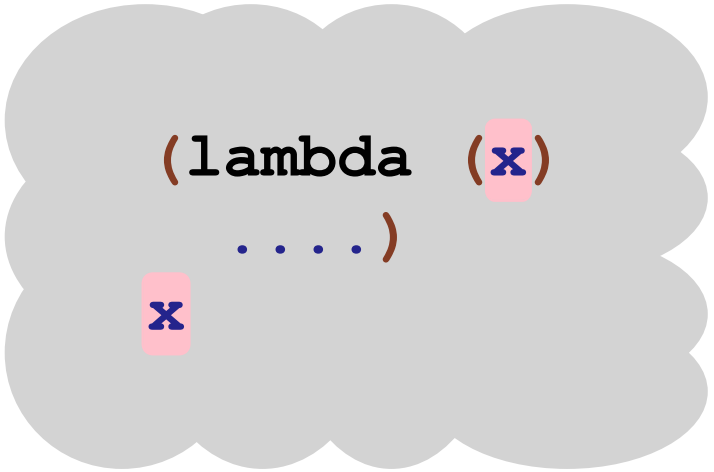
Binding table helps the expander connect *use* to *binding*

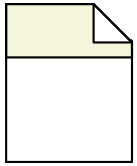




Expander and Bindings

Expander must still check whether a *use* makes sense





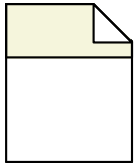
Expander and Bindings

Expander must still check whether a *use* makes sense

```
(let-syntax ([m (lambda (stx) ...)])
  ....)
m
```

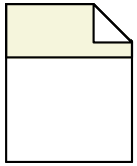
Compile-time environment maps a binding to either

- the constant `variable`
- a macro-transformer function



Compile-time Environment

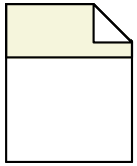
```
(define empty-env (hash))

(define (env-extend env key val)
  (hash-set env key val))

(define (env-lookup env binding)
  (hash-ref env binding #f))

(define variable (gensym 'variable))
```



Compile-time Environment

```
(define empty-env (hash))

(define (env-extend env key val)
  (hash-set env key val))

(define (env-lookup env binding)
  (hash-ref env binding #f))

(define variable (gensym 'variable))
```

```
(env-lookup empty-env loc/a)
⇒ #f
```



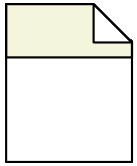
Compile-time Environment

```
(define empty-env (hash))

(define (env-extend env key val)
  (hash-set env key val))

(define (env-lookup env binding)
  (hash-ref env binding #f))

(define (new-loc/a) (gensym 'variable))
```

```
(lambda (a)
  ....)
```

```
(env-lookup
 (env-extend empty-env loc/a variable)
 loc/a)
```

⇒ `variable`

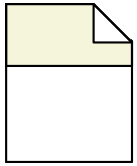


Compile-time Environment

```
(define empty-env (hash))

(define (env-extend env key val)
  (hash-set env key val))

(define (env-lookup env binding)
  (hash-ref env binding #f))

(let-syntax ([a macro-expr])
  ....)
```

```
(env-lookup
 (env-extend empty-env loc/a macro-function)
 loc/a)
```

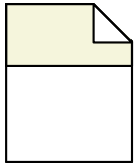
⇒ *macro-function*



Expansion Dispatch

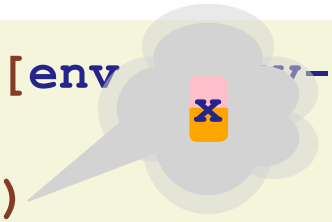
Main **expand** function

```
(define (expand s [env empty-env])
  (cond
   [(identifier? s)
    ; an identifier by itself
    (expand-identifier s env)]
   [(and (pair? s)
         (identifier? (first s)))
    ; "application" of an identifier; maybe a form
    (expand-id-application-form s env)]
   [(list? s)
    ; application of non-identifier
    (expand-app s env)]
   [else
    ; anything else: error
    (error "bad syntax:" s)]))
```

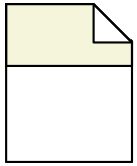


Expansion Dispatch

Main **expand** function




```
(define (expand s [env      -env])
  (cond
   [(identifier? s)
    ; an identifier by itself
    (expand-identifier s env)]
   [(and (pair? s)
         (identifier? (first s)))
    ; "application" of an identifier; maybe a form
    (expand-id-application-form s env)]
   [(list? s)
    ; application of non-identifier
    (expand-app s env)]
   [else
    ; anything else: error
    (error "bad syntax:" s)]))
```

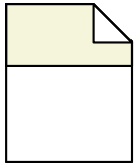


Expansion Dispatch

Main **expand** function

```
(define (expand s [env empty-env])
  (cond
   [(identifier? s)
    ; an identifier by itself
    (expand-identifier s env)]
   [(and (pair? s)
         (identifier? (first s)))
    ; "application" of an identifier; maybe a form
    (expand-id-application-form s env)]
   [(list? s)
    ; application of non-identifier
    (expand-app s env)]
   [else
    ; anything else: error
    (error "bad syntax:" s)]))
```

```
(f '1)
```

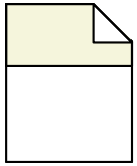


Expansion Dispatch

Main **expand** function

```
(define (expand s [env empty-env])
  (cond
   [(identifier? s)
    ; an identifier by itself
    (expand-identifier s env)]
   [(and (pair? s)
         (identifier? (first s)))
    ; "application" of an identifier; maybe a form
    (expand-id-application-form s env)]
   [(list? s)
    ; application of non-identifier
    (expand-app s env)]
   [else
    ; anything else: error
    (error "bad syntax:" s)]))
```

(one)

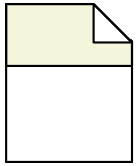


Expansion Dispatch

Main **expand** function

```
(define (expand s [env empty-env])
  (cond
   [(identifier? s)
    ; an identifier by itself
    (expand-identifier s env)]
   [(and (pair? s)
         (identifier? (first s)))
    ; "application" of an identifier; maybe a form
    (expand-id-application-form s env)]
   [(list? s)
    ; application of non-identifier
    (expand-app s env)]
   [else
    ; anything else: error
    (error "bad syntax:" s)]))
```

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

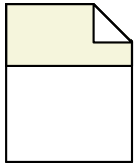


Expansion Dispatch

Main **expand** function

```
(define (expand s [env empty-env])
  (cond
   [(identifier? s)
    ; an identifier by itself
    (expand-identifier s env)]
   [(and (pair? s)
         (identifier? (first s)))
    ; "application"                    maybe a form
    (expand-id-a                      )]
   [(list? s)
    ; application of non-identifier
    (expand-app s env)]
   [else
    ; anything else: error
    (error "bad syntax:" s)]))
```

```
((curried '1) '2)
```

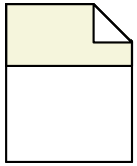


Expansion Dispatch

Main **expand** function

```
(define (expand s [env empty-env])
  (cond
   [(identifier? s)
    ; an identifier by itself
    (expand-identifier s env)]
   [(and (pair? s)
         (identifier? (first s)))
    ; "application" of an identifier; maybe a form
    (expand-id-application-form s env)]
   [(list? s)
    ; applic      of non-identifier
    (expand      env)]
   [else
    ; anything else: error
    (error "bad syntax:" s)]))
```

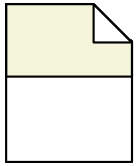
1



Expansion Dispatch

Main **expand** function

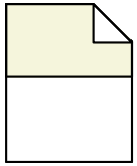
```
(define (expand s [env empty-env])
  (cond
   [(identifier? s)
    ; an identifier by itself
    (expand-identifier s env)]
   [(and (pair? s)
         (identifier? (first s)))
    ; "application" of an identifier; maybe a form
    (expand-id-application-form s env)]
   [(list? s)
    ; application of non-identifier
    (expand-app s env)]
   [else
    ; anything else: error
    (error "bad syntax:" s)]))
```



Expansion Dispatch

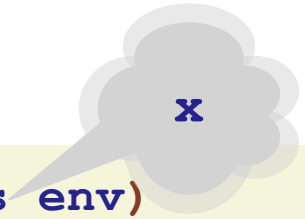
Expand an identifier by itself

```
(define (expand-identifier s env)
  (define binding (resolve s))
  (cond
   [(not binding) (error "free variable:" s)]
   [(set-member? core-primitives binding) s]
   [(set-member? core-forms binding) (error "bad syntax:" s)]
   [else
    (define v (env-lookup env binding))
    (cond
     [(eq? v variable) s]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])]))
```

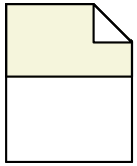


Expansion Dispatch

Expand an identifier by itself



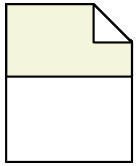

```
(define (expand-identifier s env)
  (define binding (resolve s))
  (cond
   [(not binding) (error "free variable:" s)]
   [(set-member? core-primitives binding) s]
   [(set-member? core-forms binding) (error "bad syntax:" s)]
   [else
    (define v (env-lookup env binding))
    (cond
     [(eq? v variable) s]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])]))
```



Expansion Dispatch

Expand an identifier by itself

```
(define (expand-identi      env)
  (define binding (r      conz      )
  (cond
   [(not binding) (error "free variable:" s)]
   [(set-member? core-primitives binding) s]
   [(set-member? core-forms binding) (error "bad syntax:" s)]
   [else
    (define v (env-lookup env binding))
    (cond
     [(eq? v variable) s]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])]))
```

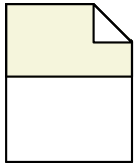


Expansion Dispatch

Expand an identifier by itself

```
(define (expand-identifier s env)
  (define binding (resolve s))
  (cond
   [(not binding) (error "free variable:" s)]
   [(set-member? core-primitives binding) s]
   [(set-member? core-forms binding) (error "bad syntax:" s)]
   [else
    (define v (env-lookup env binding))
    (cond
     [(eq? v variable) s]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])]))
```

cons

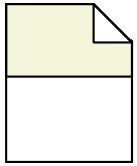


Expansion Dispatch

Expand an identifier by itself

```
(define (expand-identifier s env)
  (define binding (resolve s))
  (cond
   [(not binding) (error "free variable
   [(set-member? core-primitives bindi g) s]
   [(set-member? core-forms binding) (error "bad syntax:" s)]
   [else
    (define v (env-lookup env binding))
    (cond
     [(eq? v variable) s]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])]))
```

lambda

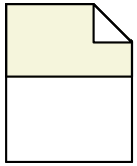


Expansion Dispatch

Expand an identifier by itself

```
(define (expand-identifier s env)
  (define binding (resolve s))
  (cond
   [(not binding) (error "free variable:" s)]
   [(set-member? core-primitives binding) s]
   [(set-member? core-forms binding) (error "bad syntax:" s)]
   [else
    (define v (env-lookup env binding))
    (cond
     [(eq? v variable) s]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])]))
```

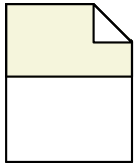
must be a local binding



Expansion Dispatch

Expand an identifier by itself

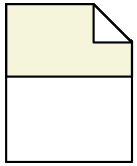
```
(define (expand-identifier s env)
  (define binding (resolve s))
  (cond
   [(not binding) (error "free variable:" s)]
   [(set-member? core-primitives binding) s]
   [(set-member? core-for                 r "bad syntax:" s)]
   [else                    (lambda (x)
                              x)
    (define v (env-lookup
    (cond
     [(eq? v variable) s]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])])))
```



Expansion Dispatch

Expand an identifier by itself

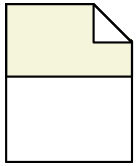
```
(define (expand-identifier s env)
  (define binding (resolve s))
  (cond
   [(not binding) (error "free variable:" s)]
   [(set-member? core-primitives binding) s]
   [(set-member? core-fo                  or "bad syntax:" s)]
   [else
    (define v (env-looku      ....)
    (cond                  x
     [(eq? v variable)  ]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])]))
```



Expansion Dispatch

Expand an identifier by itself

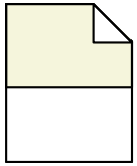
```
(define (expand-identifier s env)
  (define binding (resolve s))
  (cond
   [(not binding) (error "free variable:" s)]
   [(set-member? core-primitives binding) s]
   [(set-member? core-forms binding) (error "bad syntax:" s)]
   [else
    (define v (env-lookup env binding))
    (cond
     [(eq? v one) s]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])]))
```



Expansion Dispatch

Expand an identifier by itself

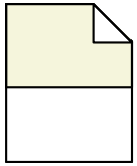
```
(define (expand-identifier s env)
  (define binding (resolve s))
  (cond
   [(not binding) (error "free variable:" s)]
   [(set-member? core-primitives binding) s]
   [(set-member? core-forms binding) (error "bad syntax:" s)]
   [else
    (define v (env-lookup env binding))
    (cond
     [(eq? v variable) s]
     [(not v) (error "out of context:" s)]
     [else (error "bad syntax:" s)])]))
```



Expansion Dispatch

Expand an identifier in “application” position

```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```

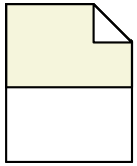


Expansion Dispatch

Expand an identifier in “application” position

(f '1)

```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```

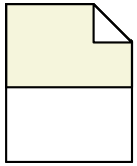


Expansion Dispatch

Expand an identifier in “application” position

(one)

```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```

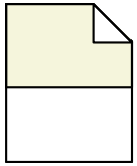


Expansion Dispatch

Expand an identifier in “application” position

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

```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```

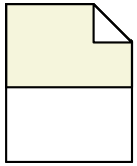


Expansion Dispatch

Expand an identifier in “application” position

core forms

```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```

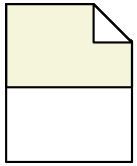


Expansion Dispatch

Expand an identifier in “application” position

core forms

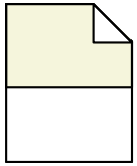
```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```



Expansion Dispatch

Expand an identifier in “application” position

```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (exp      mbda s env)]
    [(let-syntax)      d-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```

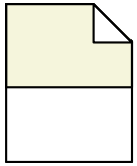


Expansion Dispatch

Expand an identifier in “application” position

```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```

```
(quote-syntax x)
```

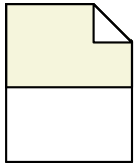


Expansion Dispatch

Expand an identifier in “application” position

```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand                    ]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```

must be a local binding

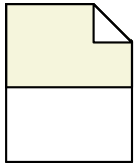


Expansion Dispatch

Expand an identifier in “application” position

```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup s binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```

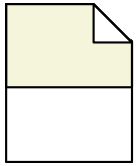
(one)



Expansion Dispatch

Expand an identifier in “application” position

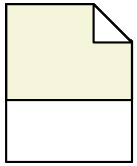
```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply transformer, then recur
       (expand (v (f '1) s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```



Expansion Dispatch

Expand an identifier in “application” position

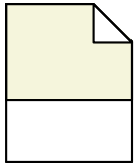
```
(define (expand-id-application-form s env)
  (define binding (resolve (first s)))
  (case binding
    [(lambda) (expand-lambda s env)]
    [(let-syntax) (expand-let-syntax s env)]
    [(quote) s]
    [(quote-syntax) s]
    [else
     (define v (env-lookup env binding))
     (cond
      [(procedure? v)
       ; apply a macro, then recur
       (expand (apply-transformer v s) env)]
      [else
       ; anything else: a function call
       (expand-app s env)])]))
```



Applying a Macro

Apply a macro transformer to syntax

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

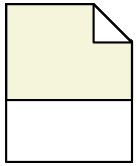


Applying a Macro

Apply a macro transformer to syntax

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```



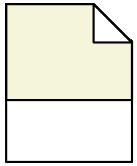
Applying a Macro

Apply a macro transformer to syntax

(one)

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```



Applying a Macro

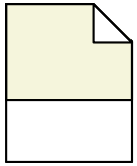
Apply … yntax

procedure that returns '1

(one)

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```



Applying a Macro

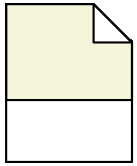
…yntax

procedure that returns `(quote 1)`

`(one)`

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```



Applying a Macro

...yntax

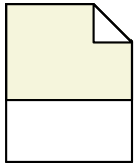
procedure that returns `(quote 1)`

`(one)`

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add                    the input
  (define intro-s (a                  ntro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

call procedure

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```



Applying a Macro

...yntax

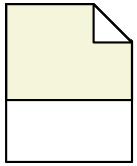
procedure that returns `(quote 1)`

`(one)`

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentativel...pe to the input
  (define intro-...pe s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

`(quote 1)`

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```



Applying a Macro

...yntax

procedure that returns `(quote 1)`

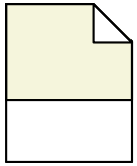
`(one)`

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentative... ...pe to the input
  (define intro... ...pe s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

`(quote 1)`

`(quote 1)`

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```



Applying a Macro

…yntax

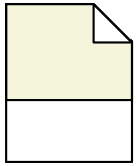
procedure that returns `(quote 1)`

`(one)`

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

`(quote 1)`

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

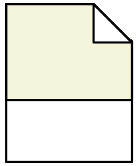


Applying a Macro

Apply a macro transformer to syntax

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk x))
```



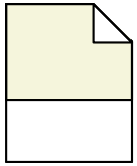
Applying a Macro

Apply a macro transformer to syntax

```
(thunk x)
```

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk x))
```



Applying a Macro

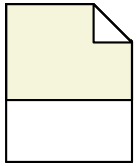
App…yntax

procedure that expands **thunk**

(thunk x)

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk x))
```



Applying a Macro

App…yntax

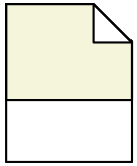
procedure that expands **thunk**

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(thunk x)
```

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

```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk x))
```



Applying a Macro

App…yntax

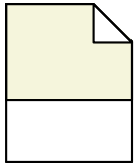
procedure that expands **thunk**

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(thunk x)
```

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

```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk x))
```



Applying a Macro

App…yntax

procedure that expands **thunk**

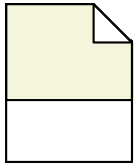
```
(define (apply-transformer t s)
  ; Create … to represent the macro step
  (define in… (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

(thunk x)

(thunk x)

(lambda (x) x)

```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk x))
```



Applying a Macro

App…yntax

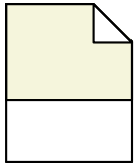
procedure that expands **thunk**

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Ten…  (lambda (x) x)  …cope to the input
  (define …                …ope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```

```
(thunk x)
```

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

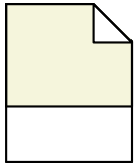
```
(let-syntax ([thunk (lambda (stx)
                      (list (quote-syntax lambda)
                            (list (quote-syntax x))
                            (second stx)))])
  (thunk x))
```



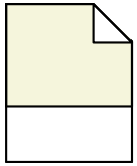
Applying a Macro

Apply a macro transformer to syntax

```
(define (apply-transformer t s)
  ; Create a scope to represent the macro step
  (define intro-scope (scope))
  ; Tentatively add the scope to the input
  (define intro-s (add-scope s intro-scope))
  ; Call the transformer
  (define transformed-s (t intro-s))
  ; Flip intro scope to get final result
  (flip-scope transformed-s intro-scope))
```



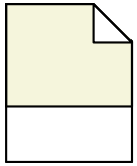
Part 3 - Core Forms



Primitive Syntactic Forms

Expand a function call

```
(define (expand-app s env)
  (map (lambda (sub-s) (expand sub-s env))
       s))
```

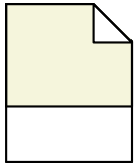


Primitive Syntactic Forms

Expand a function call

(f (one))

```
(define (expand-app s env)
  (map (lambda (sub-s) (expand sub-s env))
       s))
```



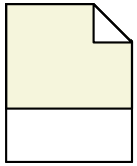
Primitive Syntactic Forms

Expand a function call

```
(define (expand-app s env)
  (map (lambda (sub-s) (expand sub-s env))
       s))
```

(f (one))

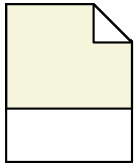
(f '1)



Primitive Syntactic Forms

Expand a **`lambda`** form

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add binding to the environment
  (define body-env (env-extend env binding variable))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

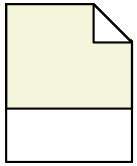


Primitive Syntactic Forms

Expand a **`lambda`** form

```
(lambda (x) (f x))
```

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add binding to the environment
  (define body-env (env-extend env binding variable))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

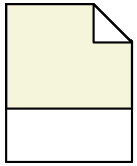


Primitive Syntactic Forms

Expand a **`lambda`** form

```
(lambda (x) (f x))
```

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add binding to the environment
  (define body-env (env-extend env binding variable))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```



Primitive Syntactic Forms

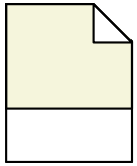
Expand a **`lambda`** form

```
(lambda (x) (f x))
```

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add binding to the environment
  (define body-env (env-extend env binding variable))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

x

(f x)

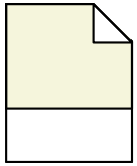


Primitive Syntactic Forms

Expand a **`lambda`** form

```
(lambda (x) (f x))
```

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add binding to the environment
  (define body-env (env-extend env binding variable))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

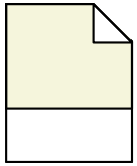


Primitive Syntactic Forms

Expand a **`lambda`** form

```
(lambda (x) (f x))
```

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (s  x
  ; Add new sc pe  o the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add binding to the environment
  (define body-env (env-extend env binding variable))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```



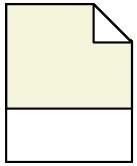
Primitive Syntactic Forms

Expand a **`lambda`** form

`(lambda (x) (f x))`

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope                    ntifier
  (define id (add-
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add binding to the environment
  (define body-env (env-extend env binding variable))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

fresh local binding for `x`



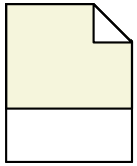
Primitive Syntactic Forms

Expand a **`lambda`** form

`(lambda (x) (f x))`

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding
  (add-binding! id
  ; Add binding to the environment
  (define body-env (env-extend env binding variable))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

fresh binding mapped to **`variable`**



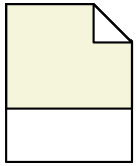
Primitive Syntactic Forms

Expand a **`lambda`** form

```
(lambda (x) (f x))
```

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add binding to the environment
  (define body-env (env-extend env bi      riable))
  ; Expand the function body after addi   the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

```
(f x)
```



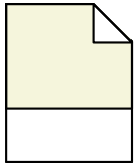
Primitive Syntactic Forms

Expand a **`lambda`** form

`(lambda (x) (f x))`

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add binding to the environment
  (define body-env (env-extend env binding
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

`(f x)`



Primitive Syntactic Forms

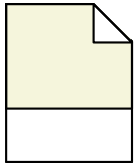
Expand a **`lambda`** form

```
(lambda (x) (f x))
```

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add b                    the environment
  (define b                  (env-extend env binding          ))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

```
(f x)
```

```
(f x)
```



Primitive Syntactic Forms

Expand a **`lambda`** form

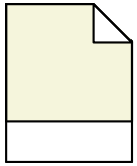
```
(lambda (x) (f x))
```

```
(define (expand-lambda s env)
  (match-define `(,lambda-id (,arg-id) ,body) s)
  ; Create a scope for this lambda
  (define sc (scope))
  ; Add new scope to the argument identifier
  (define id (add-scope arg-id sc))
  ; Bind the argument identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Add b                    the environment
  (define b                  (env-extend env binding          ))
  ; Expand the function body after adding the scope
  (define exp-body (expand (add-scope body sc)
                           body-env))
  ; Rebuild expanded form
  `(,lambda-id (,id) ,exp-body))
```

```
(f x)
```

```
(f x)
```

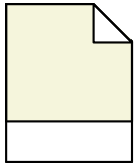
```
(lambda (x) (f x))
```



Primitive Syntactic Forms

Expand a local macro-binding form

```
(define (expand-let-syntax s env)
  (match-define `(,let-syntax-id ([,lhs-id ,rhs])
                                 ,body)
                s)
  ; Create a scope for this let-syntax
  (define sc (scope))
  ; Add new scope to the identifier
  (define id (add-scope lhs-id sc))
  ; Bind the identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Evaluate compile-time expressions
  (define rhs-val (eval-for-syntax-binding rhs))
  ; Map binding to its value
  (define body-env (env-extend env binding rhs-val))
  ; Expand body
  (expand (add-scope body sc) body-env))
```

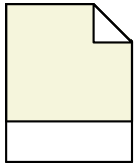


Primitive Syntactic Forms

Expand a `let-syntax`

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

```
(define (expand-let-syntax s env)
  (match-define `(,let-syntax-id ([,lhs-id ,rhs])
                   ,body)
                s)
  ; Create a scope for this let-syntax
  (define sc (scope))
  ; Add new scope to the identifier
  (define id (add-scope lhs-id sc))
  ; Bind the identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Evaluate compile-time expressions
  (define rhs-val (eval-for-syntax-binding rhs))
  ; Map binding to its value
  (define body-env (env-extend env binding rhs-val))
  ; Expand body
  (expand (add-scope body sc) body-env))
```



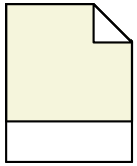
Primitive Syntactic Forms

Expand a

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

```
(define (expand-let-syntax s env)
  (match-define `(,let-syntax-id ([,lhs-id ,rhs])
                   ,body)
                s)
  ; Create a scope for this let-sy
  (define sc (scope))
  ; Add new scope to the identifier
  (define id (add-scope lhs-id sc))
  ; Bind the identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Evaluate compile-time expressions
  (define rhs-val (eval-for-syntax-binding rhs))
  ; Map binding to its value
  (define body-env (env-extend env binding rhs-val))
  ; Expand body
  (expand (add-scope body sc) body-env))
```

```
(lambda (stx)
  (quote-syntax '1))
```



Primitive Syntactic Forms

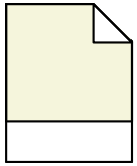
Expand a

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

```
(define (expand-let-syntax s env)
  (match-define `(,let-syntax-id ([,lhs-id ,rhs])
                   ,body)
                 s)
  ; Create a scope for this let-sy
  (define sc (scope))
  ; Add new scope to the identifier
  (define id (add-scope lhs-id sc))
  ; Bind the ident
  (define binding
  (add-binding! id binding)
  ; Evaluate compile-time expressions
  (define rhs-val (eval-for-syntax-binding rhs))
  ; Map binding to its value
  (define body-env (env-extend env binding rhs-val))
  ; Expand body
  (expand (add-scope body sc) body-env))
```

```
(lambda (stx)
  (quote-syntax '1))
```

a procedure



Primitive Syntactic Forms

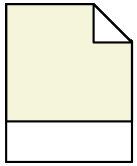
Expand a

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

```
(define (expand-let-syntax s env)
  (match-define `(,let-syntax-id ([,lhs-id ,rhs])
                  ody)
                s
  ; Create a scope for this let-sy
  (define sc (scope))
  ; Add new scope to the identifier
  (define id (add-scope lhs-id sc))
  ; Bind the ident
  (define binding
  (add-binding! id binding)
  ; Evaluate compile-time expressions
  (define rhs-val (eval-for-syntax-binding rhs))
  ; Map binding to its value
  (define body-env (env-extend env binding rhs-val))
  ; Expand body
  (expand (add-scope body sc) body-env))
```

```
(lambda (stx)
  (quote-syntax '1))
```

a procedure



Primitive Syntactic Forms

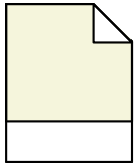
Expand a

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

```
(define (expand-let-syntax s env)
  (match-define `(,let-syntax-id ([,lhs-id ,rhs])
                  ody)
                 s
  ; Create a scope for this let-sy
  (define sc (scope))
  ; Add new scope to the identifier
  (define id (add-scope lhs-id sc))
  ; Bind the identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Evaluate compi
  (define rhs-val
  ; Map binding to its value
  (define body-env (env-extend env binding rhs-val))
  ; Expand body
  (expand (add-scope body sc) body-env))
```

```
(lambda (stx)
  (quote-syntax '1))
```

fresh binding for one mapped to procedure



Primitive Syntactic Forms

Expand a

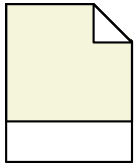
```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

```
(define (expand-let-syntax s env)
  (match-define `(,let-syntax-id ([,lhs-id ,rhs])
                  ody)
                s
  ; Create a scope for this let-sy
  (define sc (scope))
  ; Add new scope to the identifier
  (define id (add-scope lhs-id sc))
  ; Bind the identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Evaluate compi
  (define rhs-val
  ; Map binding to
  (define body-env (
  ; Expand body
  (expand (add-scope body sc) body-env))
```

```
(lambda (stx)
  (quote-syntax '1))
```

fresh binding for one mapped to procedure

```
(one)
```



Primitive Syntactic Forms

Expand a

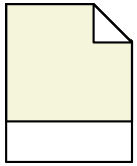
```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

```
(define (expand-let-syntax s env)
  (match-define `(,let-syntax-id ([,lhs-id ,rhs])
                  ody)
                s
  ; Create a scope for this let-sy
  (define sc (scope))
  ; Add new scope to the identifier
  (define id (add-scope lhs-id sc))
  ; Bind the identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Evaluate compi
  (define rhs-val
  ; Map binding to its v
  (define body-env (env-ex  binding rhs-val))
  ; Expand body
  (expand (add-scope body sc) body-env))
```

```
(lambda (stx)
  (quote-syntax '1))
```

fresh binding for one mapped to procedure

one



Primitive Syntactic Forms

Expand a

```
(let-syntax ([one (lambda (stx)
                    (quote-syntax '1))])
  (one))
```

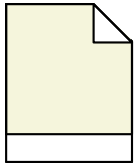
```
(define (expand-let-syntax s env)
  (match-define `(,let-syntax-id ([,lhs-id ,rhs])
                  ody)
                s
  ; Create a scope for this let-sy
  (define sc (scope))
  ; Add new scope to the identifier
  (define id (add-scope lhs-id sc))
  ; Bind the identifier
  (define binding (gensym))
  (add-binding! id binding)
  ; Evaluate compi
  (define rhs-val
  ; Map binding to its v
  (define body-env (env-ex  binding rhs-val))
  ; Expand body
  (expand (add-scope body sc) body-env))
```

```
(lambda (stx)
  (quote-syntax '1))
```

fresh binding for one mapped to procedure

```
(one)
```

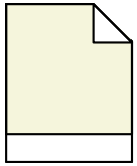
```
(quote 1)
```



Primitive Syntactic Forms

Helper: expand and eval for compile time

```
(define (eval-for-syntax-binding rhs)
  (eval-compiled (compile (expand rhs empty-env))))
```

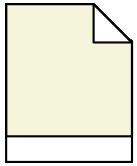


Primitive Syntactic Forms

Helper: expand and eval for co

```
(lambda (stx)
  (quote-syntax '1))
```

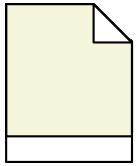
```
(define (eval-for-syntax-binding rhs)
  (eval-compiled (compile (expand rhs empty-env))))
```



Bridge to the Host

Compile expanded to a host-Racket S-expression

```
(define (compile s)
  (cond
   [(identifier? s) (resolve s)]
   [else
    (case (and (identifier? (first s)) (resolve (first s)))
      [(lambda)
       (match-define `(,lambda-id (,id) ,body) s)
       `(lambda (,(resolve id)) ,(compile body))]
      [(quote)
       (match-define `(,quote-id ,datum) s)
       `(quote ,(syntax->datum datum))]
      [(quote-syntax)
       (match-define `(,quote-syntax-id ,datum) s)
       `(quote ,datum)]
      [else
       ; Anything else is a function call
       (map compile s)])]))
```

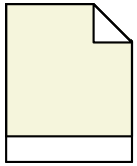


Bridge to the Host

Compile expanded to a host-Racket S-expression

```
(define (compile s)
  (cond
   [(identifier? s) (resolve s)]
   [else
    (case (and (identifier? (first s)) (length s))
      [(lambda)
       (match-define `(,lambda-id (,id) ,body) s)
       `(lambda (,(resolve id)) ,(compile body))]
      [(quote)
       (match-define `(,quote-id ,datum) s)
       `(quote ,(syntax->datum datum))]
      [(quote-syntax)
       (match-define `(,quote-syntax-id ,datum) s)
       `(quote ,datum)]
      [else
       ; Anything else is a function call
       (map compile s)])]))
```

use **gensym** for variable

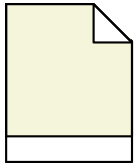


Bridge to the Host

Compile expanded to a host-Racket S-expression

```
(define (compile s)
  (cond
   [(identifier? s) (resolve s)]
   [else
    (case (and (identifier? (first s)) (resolve (first s)))
      [(lambda)
       (match-define `(,lambda-id (,id) ,
       `(lambda (,(resolve id)) ,(compile
      [(quote)
       (match-define `(,quote-id ,datum) s)
       `(quote ,(syntax->datum datum))]
      [(quote-syntax)
       (match-define `(,quote-syntax-id ,datum) s)
       `(quote ,datum)]
      [else
       ; Anything else is a function call
       (map compile s)])]))
```

strip scopes for **quote**

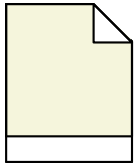


Bridge to the Host

Compile expanded to a host-Racket S-expression

```
(define (compile s)
  (cond
   [(identifier? s) (resolve s)]
   [else
    (case (and (identifier? (first s)) (resolve (first s)))
      [(lambda)
       (match-define `(,lambda-id (,id) ,body) s)
       `(lambda (,(resolve id)) ,(compile body))]
      [(quote)
       (match-define `(,qu                    )
       `(quote ,(syntax->d
      [(quote-syntax)
       (match-define `(,quote-syntax-id ,datum) s)
       `(quote ,datum)]
      [else
       ; Anything else is a function call
       (map compile s)])]))
```

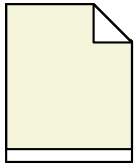
preserve scopes for `quote-syntax`



Bridge to the Host

Compile expanded to a host-Racket S-expression

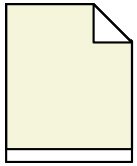
```
(define (compile s)
  (cond
   [(identifier? s) (resolve s)]
   [else
    (case (and (identifier? (first s)) (resolve (first s)))
      [(lambda)
       (match-define `(,lambda-id (,id) ,body) s)
       `(lambda (,(resolve id)) ,(compile body))]
      [(quote)
       (match-define `(,quote-id ,datum) s)
       `(quote ,(syntax->datum datum))]
      [(quote-syntax)
       (match-define `(,quote-syntax-id ,datum) s)
       `(quote ,datum)]
      [else
       ; Anything else is a function call
       (map compile s)])]))
```



Bridge to the Host

Set up a host-Racket evaluation environment

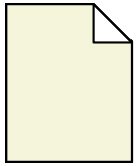
```
(define namespace (make-base-namespace))
(eval '(require racket/list) namespace)

(namespace-set-variable-value! 'datum->syntax
                               datum->syntax
                               #t namespace)
(namespace-set-variable-value! 'syntax->datum
                               syntax->datum
                               #t namespace)
(namespace-set-variable-value! 'syntax-e
                               syntax-e
                               #t namespace)

(define (eval-compiled s)
  (eval s namespace))
```



Bridge to the Host

Set up a host-Racket evaluation environment

```
(define namespace (make-base-namespace))
(eval '(require racket/list) namespace)

(namespace-set-variable-value! 'datum->syntax
                               datum->syntax
                               #t namespace)
(namespace-set-variable-value! 'syntax->datum
                               syntax->datum
                               #t namespace)
(namespace-set-variable-value! 'syntax-e
                               s
                               (lambda (g42) g42)

(define (eval-compiled s)
  (eval s namespace))
```



Done!

Copy Code Copy Code + Examples

Explore More

https://github.com/mflatt/expander

branch	description	LoC*
pico	*We just built it*	**~250**
nano	*Implicit quote and multi-arg λ*	**~300**
micro	*Split into modules*	**~700**
mini	*Definition contexts*	**~1,300**
demi	*Modules & phases*	**~3,000**
master	*Full Racket expander*	**~20,000†**

* without examples/tests

† without bootstrap & extract

Thanks!