Modeling Macro Hygiene with Scope Graphs

Michael Ballantyne

University of Utah

```
(require racket/match)

(define (eval exp env)
  (match exp
    [`(lambda (,(? symbol? arg)) ,body)
     (closure exp env)]
    [(? symbol?)
     (hash-ref env exp)]
    [`(,e1 ,e2)
     (apply (eval e1 env)
            (eval e2 env))]))
```

```
(let ((exp216 exp))
  (define (fail217)
    (match:error exp216 (syntax-srclocs (quote-syntax srcloc)) 'match))
  (let* ((f218
          (lambda ()
            (cond
             ((pair? exp216)
              (let ((unsafe-car219 (unsafe-car exp216))
                    (unsafe-cdr220 (unsafe-cdr exp216)))
                (cond
                 ((pair? unsafe-cdr220)
                  (let ((unsafe-car223 (unsafe-car unsafe-cdr220))
                        (unsafe-cdr224 (unsafe-cdr unsafe-cdr220)))
                    (cond
                     ((null? unsafe-cdr224)
                      (syntax-parameterize
                       ((fail
                         (make-rename-transformer (quote-syntax fail217))))
                       (let ((e2 unsafe-car223))
                         (let ((e1 unsafe-car219)) (let () 'app)))))
                     (else (fail217)))))
                 (else (fail217)))))
             (else (fail217)))))
         (f229
          (lambda ()
            (cond
             ((symbol? exp216)
              (syntax-parameterize
               ((fail (make-rename-transformer (quote-syntax f218)))
                (let () 'ref)))
              (else (f218))))))
    (cond
     ((pair? exp216)
      (let ((unsafe-car233 (unsafe-car exp216))
            (unsafe-cdr234 (unsafe-cdr exp216)))
        (cond
         ((equal? unsafe-car233 'lambda)
          (cond
           ((pair? unsafe-cdr234)
            (let ((unsafe-car237 (unsafe-car unsafe-cdr234))
                  (unsafe-cdr238 (unsafe-cdr unsafe-cdr234)))
              (cond
               ((pair? unsafe-car237)
                (let ((unsafe-car240 (unsafe-car unsafe-car237))
                      (unsafe-cdr241 (unsafe-cdr unsafe-car237)))
                  (cond
                   ((symbol? unsafe-car240)
                    (cond
                     ((null? unsafe-cdr241)
                      (cond
                       ((pair? unsafe-cdr238)
                        (let ((unsafe-car250 (unsafe-car unsafe-cdr238))
                              (unsafe-cdr251 (unsafe-cdr unsafe-cdr238)))
                          (cond
                           ((null? unsafe-cdr251)
                            (syntax-parameterize
                             ((fail
                               (make-rename-transformer (quote-syntax f229))))
                             (let ((body unsafe-car250))
                               (let ((arg unsafe-car240)) (let () 'lambda)))))
                           (else (f229)))))
                       (else (f229))))
                     (else (f229))))
                   (else (f229))))
                (else (f229))))
              (else (f229)))))
           (else (f229)))))
         (else (f229))))
     (else (f229)))))
```

- Classes
- Higher order modules
- Typed Racket

DSLs for:

- Typesetting
- Logic programming

… etc

```
(define-syntax or
  (syntax-rules ()
    [(_ a b)
     (let ([tmp a])
       (if tmp
           tmp
           b))]))
```

```
(define (lookup key default)
  (or (hash-ref m key)
      default))
```

=>

```
(define (lookup key default)
  (let ([tmp (hash-ref m key)])
    (if tmp
        tmp
        default)))
```

```
(define-syntax or
  (syntax-rules ()
    [(_ a b)
     (let ([tmp a])
       (if tmp
           tmp
           b))]))
```

Pattern variable reference

Template

Pattern

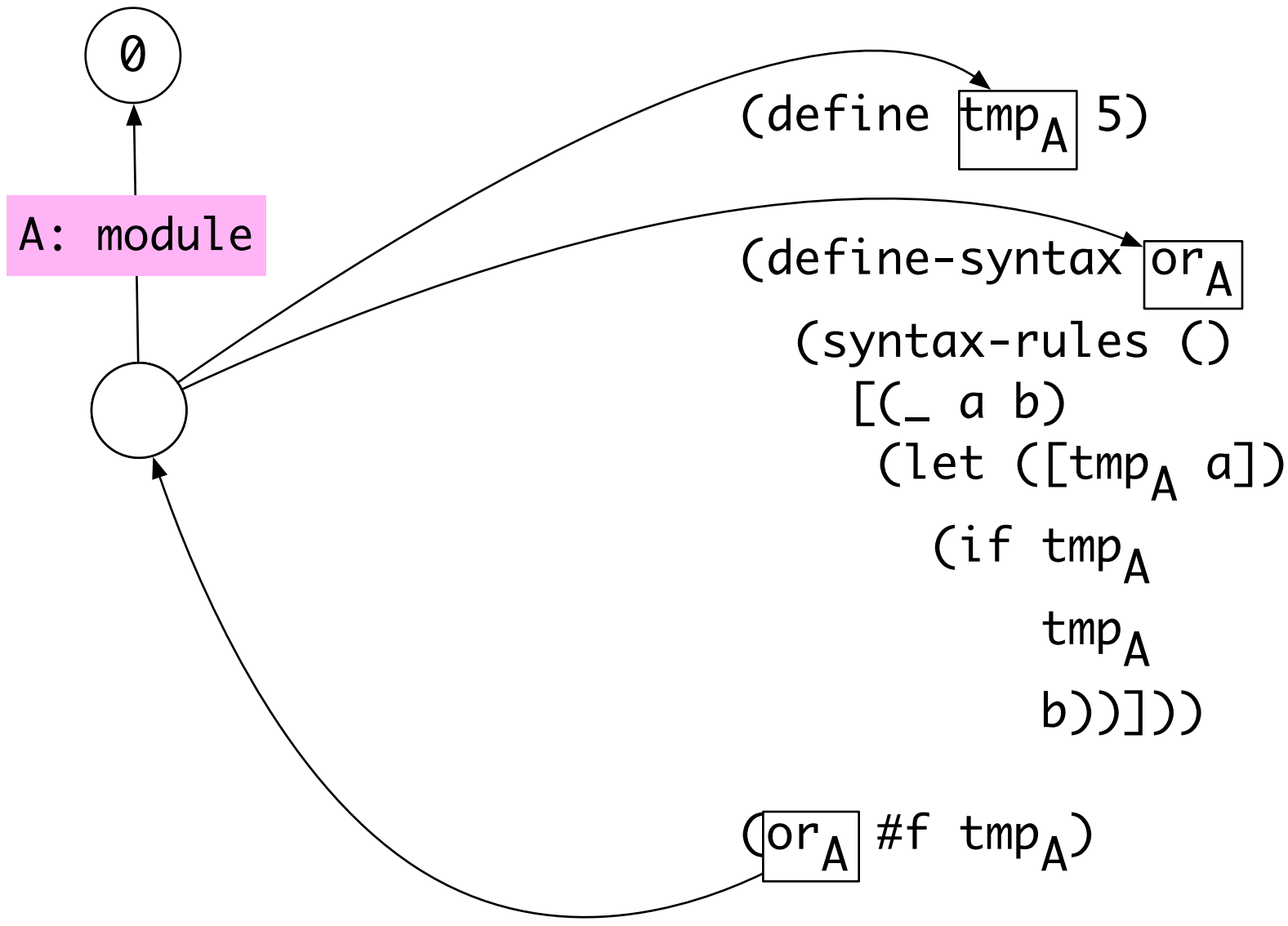
```
(define (lookup key default)
  (or (hash-ref m key)
      default))
```

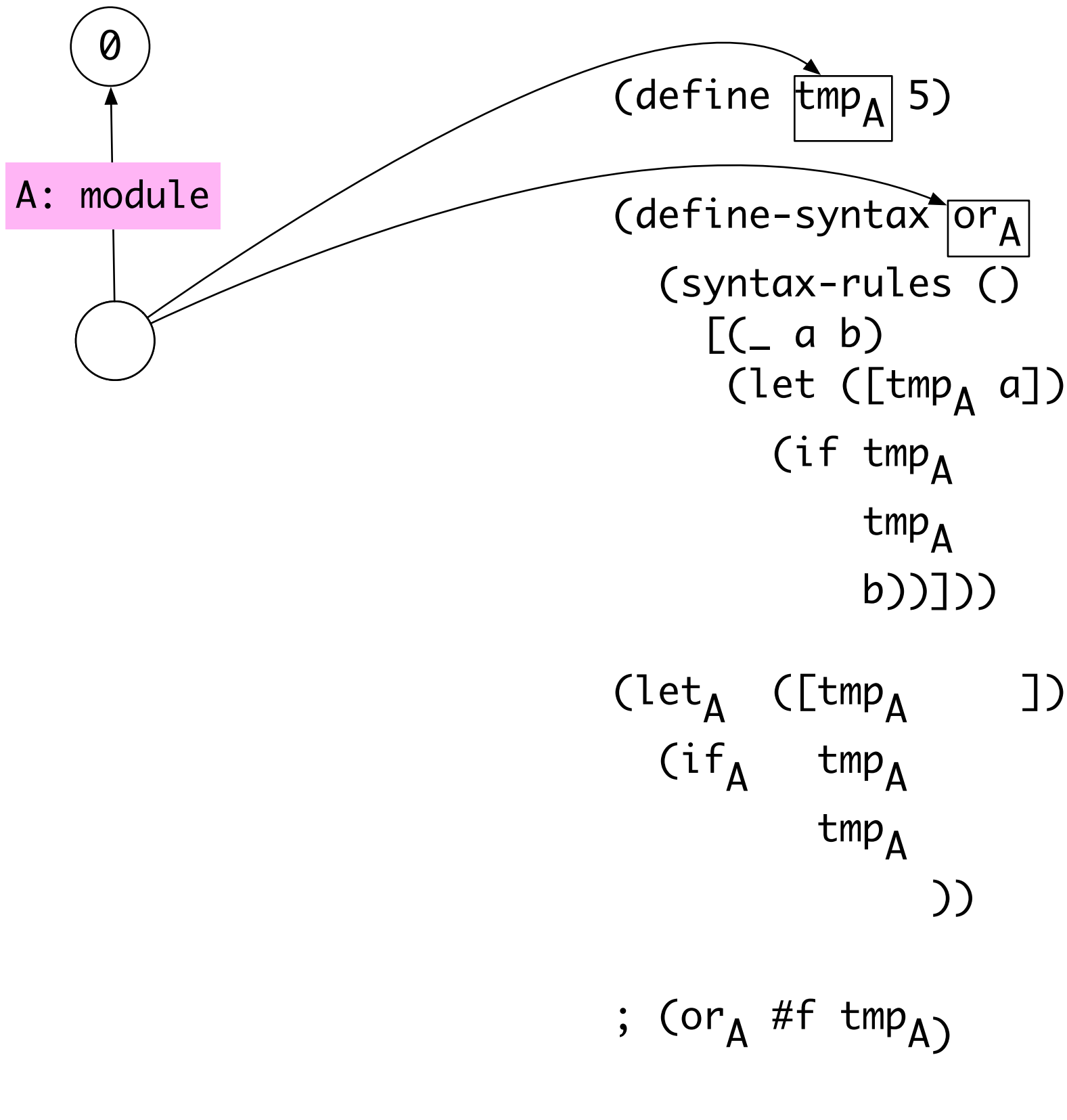
=>

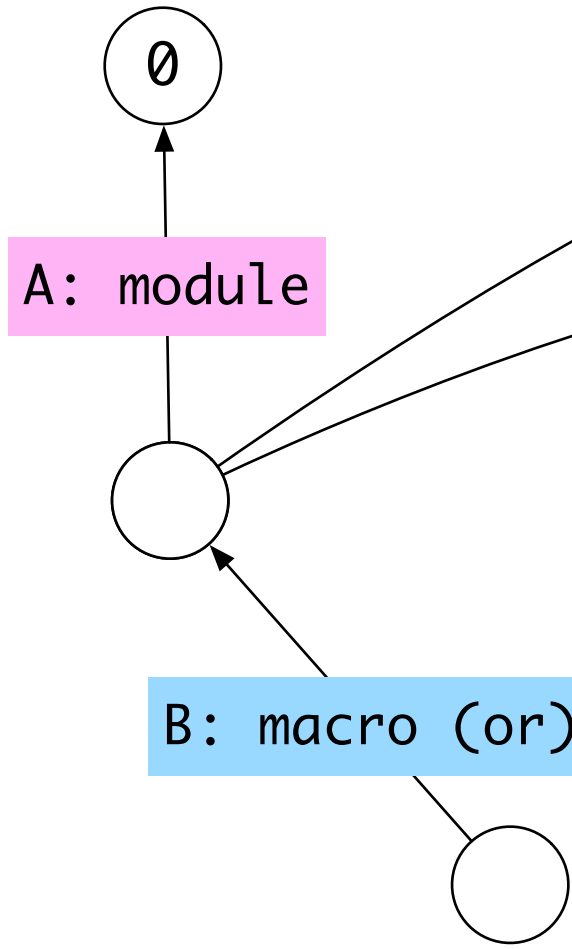
```
(define (lookup key default)
  (let ([tmp (hash-ref m key)])
    (if tmp
        tmp
        default)))
```

```
(define-syntax or
  (syntax-rules ()
    [(_ a b)
     (let ([tmp a])
       (if tmp
           tmp
           b))]))
```

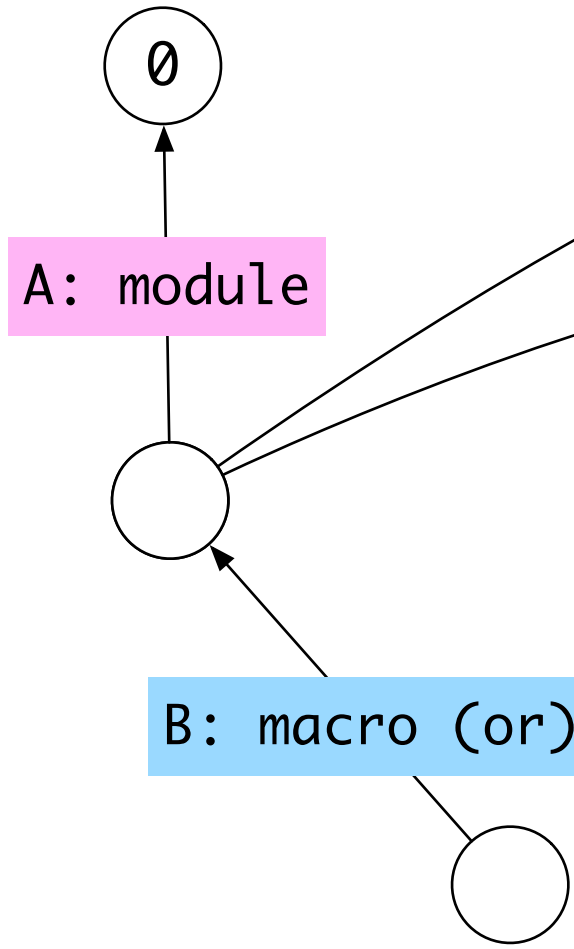
```
(define (lookup key if)
  (or (hash-ref m key)
      if))

=>

(define (lookup key if)
  (let ([tmp (hash-ref m key)])
    (if tmp
        tmp
        if)))
```


0
A: module
(define tmp_A 5)
(define-syntax or_A
(syntax-rules ()
[(_ a b)
(let ([tmp_A a])
(if tmp_A
tmp_A
b))]))
(or_A #f tmp_A)

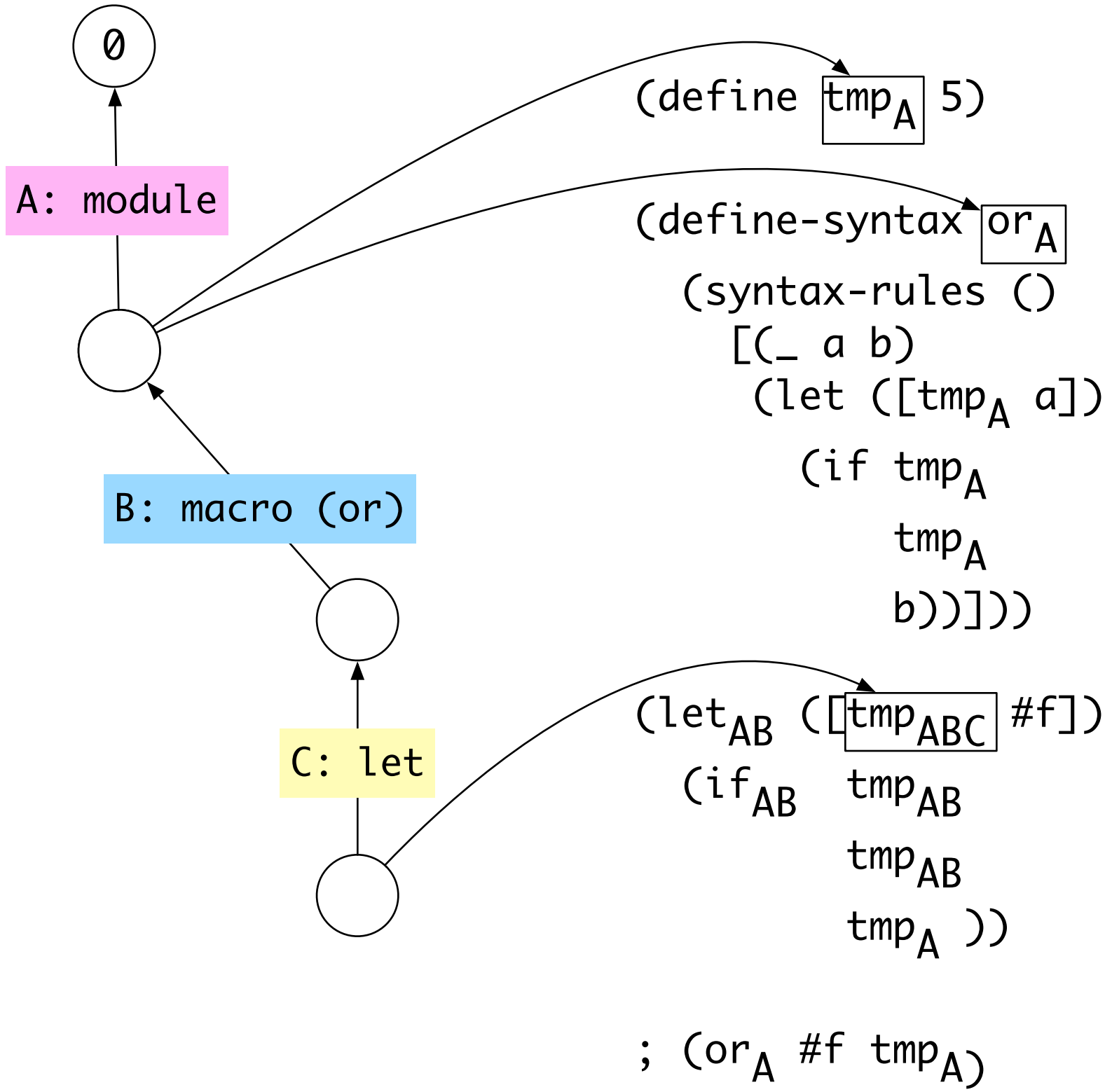

0
A: module
(define tmp_A 5)
(define-syntax or_A
(syntax-rules ()
[(_ a b)
(let ([tmp_A a])
(if tmp_A
tmp_A
b))]))
(let_A ([tmp_A])
(if_A tmp_A
tmp_A
))
; (or_A #f tmp_A)

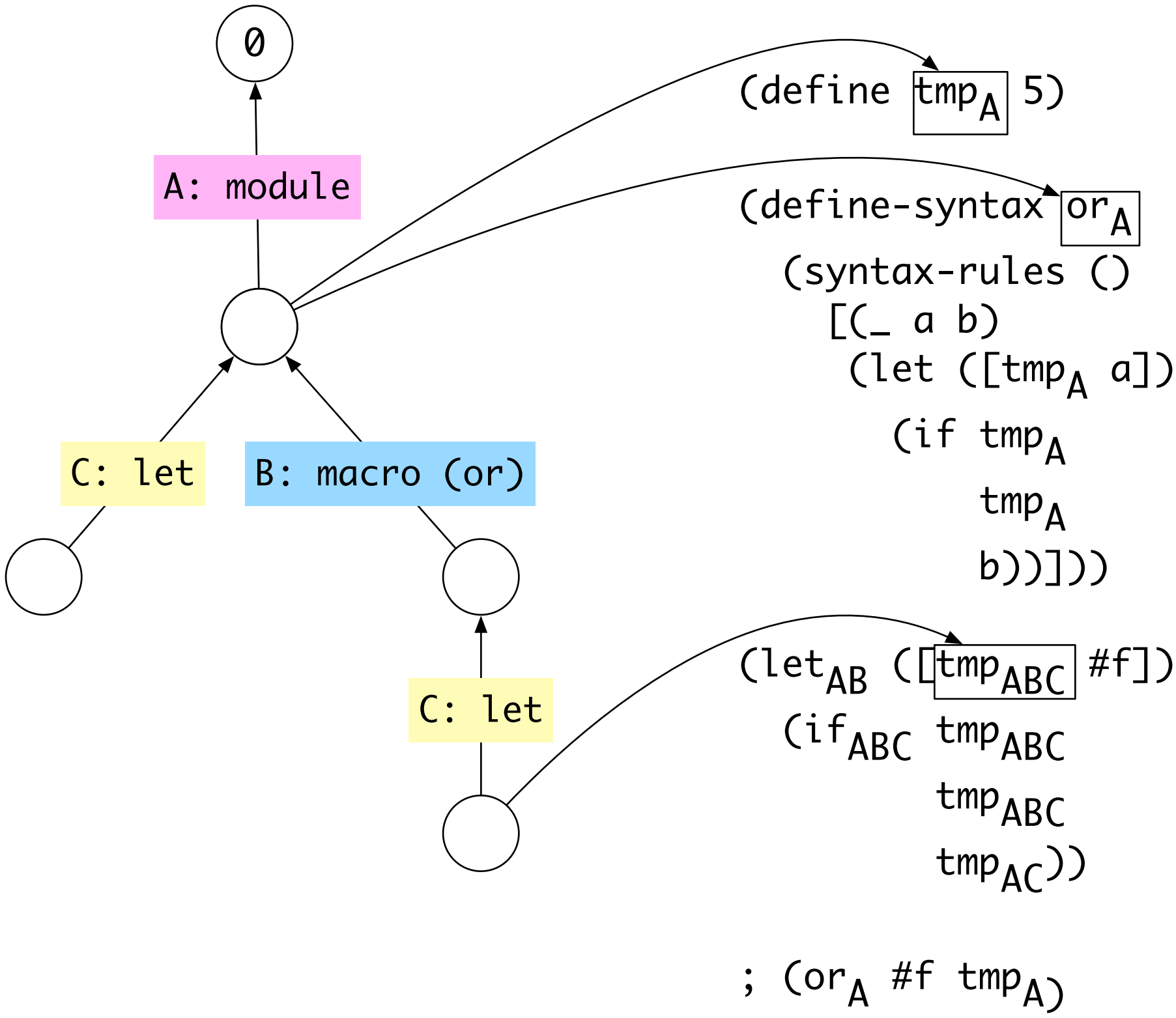


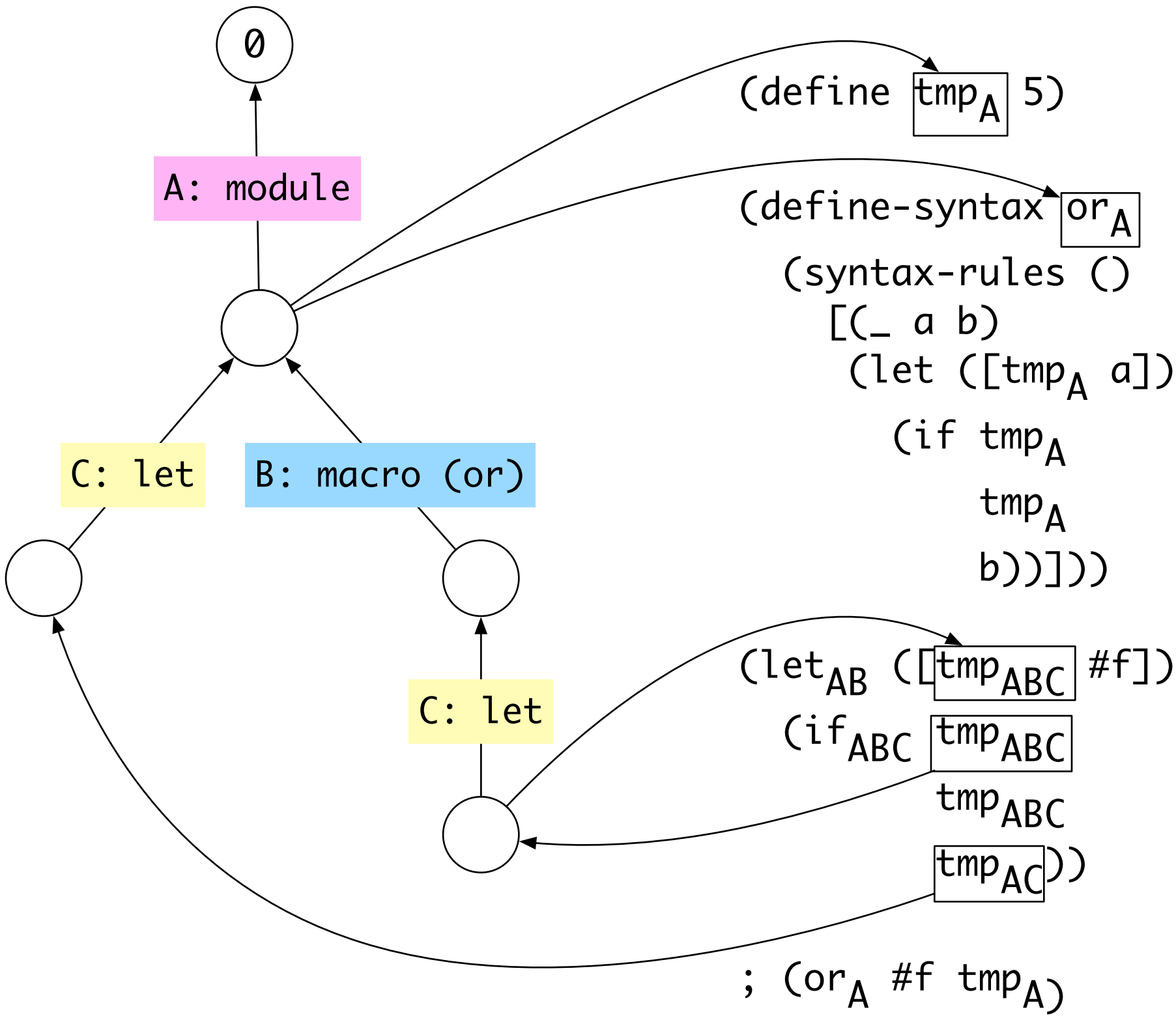

```
(define tmp_A 5)

(define-syntax or_A
  (syntax-rules ()
    [(_ a b)
     (let ([tmp_A a])
       (if tmp_A
           tmp_A
           b))]))

(let_AB ([tmp_AB    ])
  (if_AB  tmp_AB
          tmp_AB
              ))

; (or_A #f tmp_A)
```




```
(define tmp_A 5)

(define-syntax or_A
  (syntax-rules ()
    [(_ a b)
     (let ([tmp_A a])
       (if tmp_A
           tmp_A
           b))]))

(let_AB ([tmp_AB  #f])
  (if_AB  tmp_AB
          tmp_AB
          tmp_A ))

; (or_A #f tmp_A)
```


0
A: module
B: macro (or)
C: let
(define tmp_A 5)
(define-syntax or_A
(syntax-rules ()
[(_ a b)
(let ([tmp_A a])
(if tmp_A
tmp_A
b))]))
(let_AB ([tmp_ABC #f])
(if_AB tmp_AB
tmp_AB
tmp_A))
; (or_A #f tmp_A)


0
A: module
C: let
B: macro (or)
C: let
(define tmp_A 5)
(define-syntax or_A
(syntax-rules ()
[(_ a b)
(let ([tmp_A a])
(if tmp_A
tmp_A
b))]))
(let_AB ([tmp_ABC #f])
(if_ABC tmp_ABC
tmp_ABC
tmp_AC))
; (or_A #f tmp_A)


0
A: module
C: let
B: macro (or)
C: let
(define tmp_A 5)
(define-syntax or_A
(syntax-rules ()
[(_ a b)
(let ([tmp_A a])
(if tmp_A
tmp_A
b))]))
(let_AB ([tmp_ABC #f])
(if_ABC tmp_ABC
tmp_ABC
tmp_AC))
; (or_A #f tmp_A)

Simple notion of scope.

Trivially scales up to complex scenarios:

- block scope with mutually recursive definitions
- macro-defining macros
- procedural macros