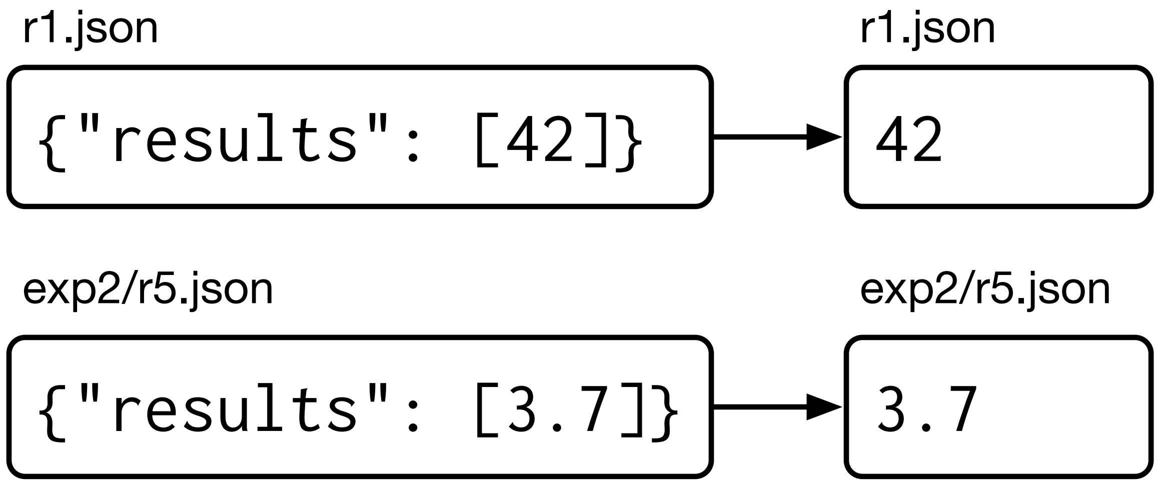
Building a path from language user to sophisticated DSL creator in Racket

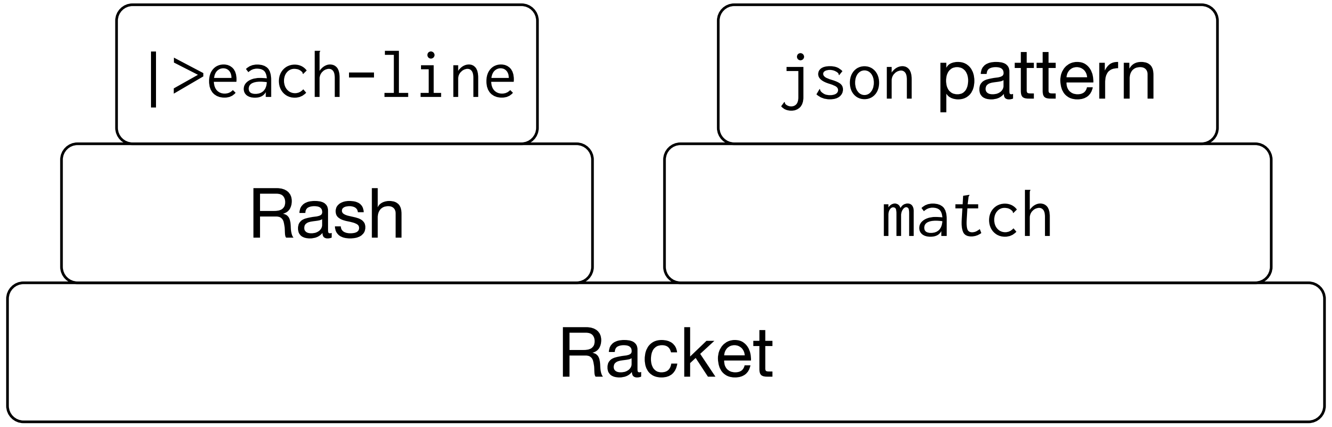
Michael Ballantyne PRL @ Northeastern University

An extensible language
for language-oriented
programming.

```
#lang rash

(require match json)

(define (fix-file f)
  (write-json-file
   (match (read-json-file f)
     [(json { "results" [ v ] }) v]
     [v v])
   f))

find . -name *.json |>each-line fix-file
```



```
#lang rash

(require match json)

(define (fix-file f)
  (write-json-file
   (match (read-json-file f)
     [(json { "results" [ v ] }) v]
     [v v])
   f))

find . -name *.json |>each-line fix-file
```



This talk

How we build DSLs in Racket, via two techniques:

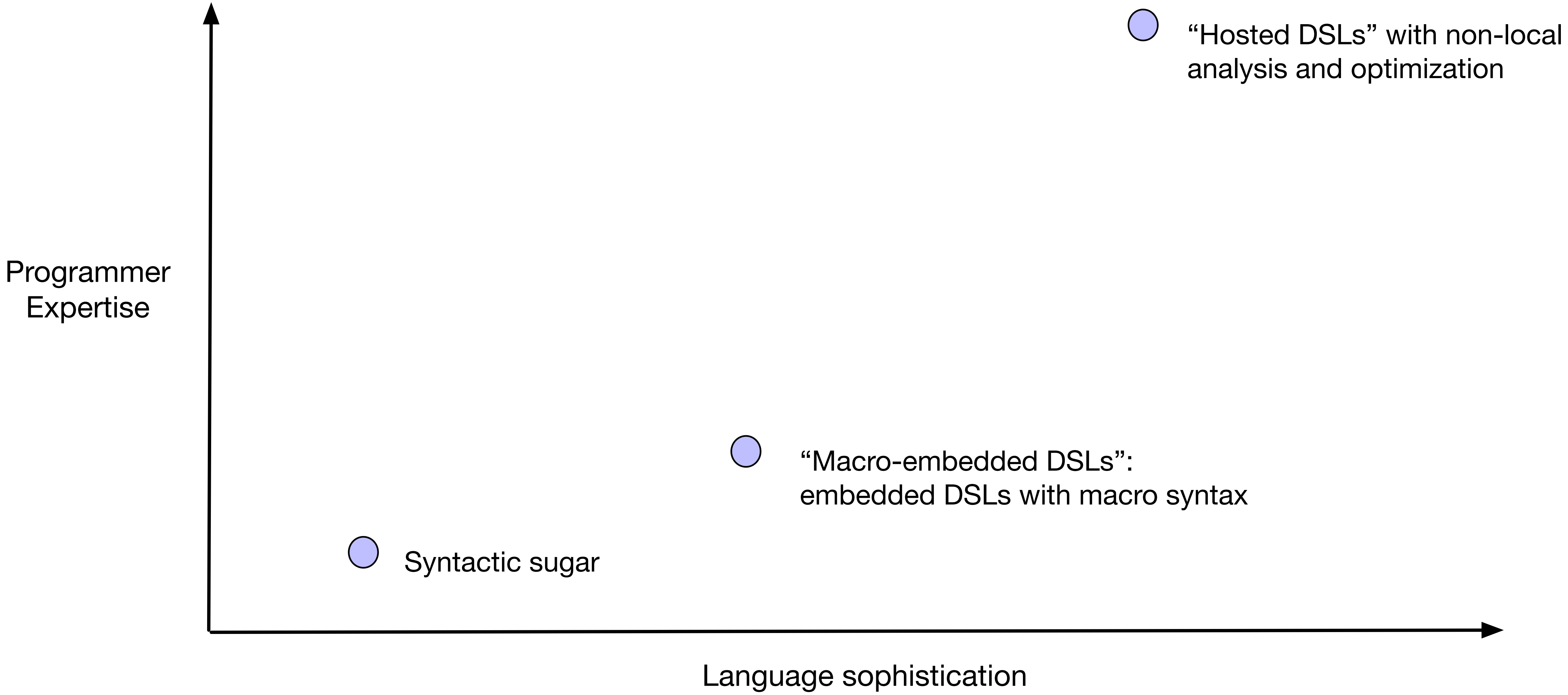
- "Macro-embedded DSLs"
- "Hosted DSLs"

Making sophisticated "hosted" DSLs easier to build:

- A new API for re-using parts of Racket's macro expander to build custom macro expanders for DSLs
- A new meta-language for creating hosted DSL front-ends
  (Work in progress)

Work with Matthias Felleisen and Alexis King

The path from programmer to DSL creator



Syntactic sugar via macros

```
(define (append l1 l2)
  (cond [(null? l1) l2]
        [(pair? l1)
         (let ([head (car l1)] [rest (cdr l1)])
           (cons head (append rest l2)))]))
```

```
(require "match-list.rkt")

(define (append l1 l2)
  (match-list l1
    [() l2]
    [(head rest) (cons head (append rest l2))]))
```

```
(define (append l1 l2)
  (cond [(null? l1) l2]
        [(pair? l1)
         (let ([head (car l1)] [rest (cdr l1)])
           (cons head (append rest l2)))]))
```

Defining `match-list`

```
#lang racket

(provide match-list)
(require (for-syntax syntax/parse))

(define (match-list-error) (error 'match-list […]))

(define-syntax match-list
  (lambda (stx)
    (syntax-parse stx
      [(_ e
          [() null-body ...+]
          [(head tail) pair-body ...+])
       #'(let ([v e])
           (cond [(null? v) null-body ...]
                 [(pair? v) (let ([head (car v)] [tail (cdr v)])
                              pair-body ...)]
                 [else (match-list-error)]))])))
```

Syntax export

Syntax definition

```
#lang racket

(provide match-list)
(require (for-syntax syntax/parse))

(define (match-list-error) (error 'match-list […]))

(define-syntax match-list
  (lambda (stx)
    (syntax-parse stx
      [(_ e
         [() null-body ...+]
         [(head tail) pair-body ...+])
       #'(let ([v e])
           (cond [(null? v) null-body ...]
                 [(pair? v) (let ([head (car v)] [tail (cdr v)])
                              pair-body ...)]
                 [else (match-list-error)]))])))
```

`Syntax -> Syntax` transformer function

```
#lang racket

(provide match-list)
(require (for-syntax syntax/parse))

(define (match-list-error) (error 'match-list […]))

(define-syntax match-list
  (lambda (stx)
    (syntax-parse stx
      [(_ e
         [() null-body ...+]
         [(head tail) pair-body ...+])
       #'(let ([v e])
           (cond [(null? v) null-body ...]
                 [(pair? v) (let ([head (car v)] [tail (cdr v)])
                              pair-body ...)]
                 [else (match-list-error)]))])))
```

```racket
#lang racket

(provide match-list)
(require (for-syntax syntax/parse))

(define (match-list-error) (error 'match-list […]))

(define-syntax match-list
  (lambda (stx)
    (syntax-parse stx
      [(_ e
          [() null-body ...+]
          [(head tail) pair-body ...+])
       #'(let ([v e])
           (cond [(null? v) null-body ...]
                 [(pair? v) (let ([head (car v)] [tail (cdr v)])
                              pair-body ...)]
                 [else (match-list-error)]))])))
```

Import the `syntax-parse` meta-language for compile-time

```
#lang racket

(provide match-list)
(require (for-syntax syntax/parse))

(define (match-list-error) (error 'match-list […]))

(define-syntax match-list
  (lambda (stx)
    (syntax-parse stx
      [(_ e
         [() null-body ...+]
         [(head tail) pair-body ...+])
       #'(let ([v e])
           (cond [(null? v) null-body ...]
                 [(pair? v) (let ([head (car v)] [tail (cdr v)])
                              pair-body ...)]
                 [else (match-list-error)]))])))
```

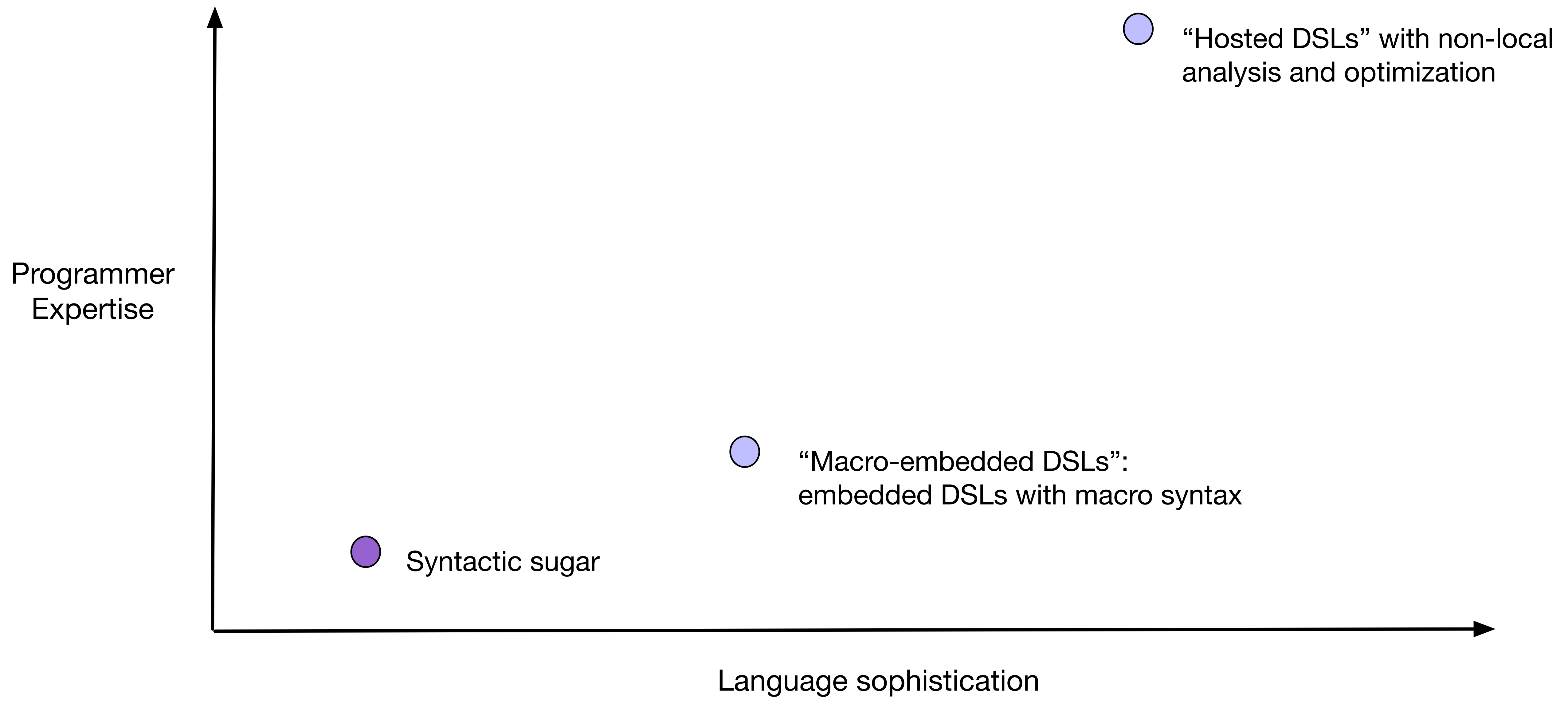
Pattern

Template

```
#lang racket

(provide match-list)
(require (for-syntax syntax/parse))

(define (match-list-error) (error 'match-list […]))

(define-syntax match-list
  (lambda (stx)
    (syntax-parse stx
      [(_ e
         [() null-body ...+]
         [(head tail) pair-body ...+])
       #'(let ([v e])
           (cond [(null? v) null-body ...]
                 [(pair? v) (let ([head (car v)] [tail (cdr v)])
                              pair-body ...)]
                 [else (match-list-error)]))])))
```

Runtime support for the language feature

```
#lang racket

(provide match-list)
(require (for-syntax syntax/parse))

(define (match-list-error) (error 'match-list […]))

(define-syntax match-list
  (lambda (stx)
    (syntax-parse stx
      [(_ e
         [() null-body ...+]
         [(head tail) pair-body ...+])
       #'(let ([v e])
           (cond [(null? v) null-body ...]
                 [(pair? v) (let ([head (car v)] [tail (cdr v)])
                              pair-body ...)]
                 [else (match-list-error)]))])))
```


"Hosted DSLs" with non-local
analysis and optimization
Programmer
Expertise
"Macro-embedded DSLs":
embedded DSLs with macro syntax
Syntactic sugar
Language sophistication

Macro-embedded DSLs

An example DSL: miniKanren

```
(run 3 (l1 l2)
  (append l1 l2 '("a" "b")))
=> ;; evaluates to
((("a" "b") ())
 (("a") ("b"))
 (() ("a" "b")))
```

```
#lang racket

(require minikanren)

(define-relation (append l1 l2 l3)
  (conde
    [(== l1 '())
     (== l2 l3)]
    [(fresh (first rest result)
       (== (cons first rest) l1)
       (== (cons first result) l3)
       (append rest l2 result))]))
```

Embedding

Functions

```
#lang racket

(require minikanren)

(define-relation (append l1 l2 l3)
  (conde
    [(== l1 '())
     (== l2 l3)]
    [(fresh (first rest result)
       (== (cons first rest) l1)
       (== (cons first result) l3)
       (append rest l2 result))]))
```

Embedding

```
#lang racket

(require minikanren)

(define-relation (append l1 l2 l3)
  (conde
    [(== l1 '())
     (== l2 l3)]
    [(fresh (first rest result)
       (== (cons first rest) l1)
       (== (cons first result) l3)
       (append rest l2 result))]))
```

Reuse of Racket

Embedding

Macros

```
#lang racket

(require minikanren)

(define-relation (append l1 l2 l3)
  (conde
    [(== l1 '())
     (== l2 l3)]
    [(fresh (first rest result)
       (== (cons first rest) l1)
       (== (cons first result) l3)
       (append rest l2 result))]))
```

```
(define-syntax conde
  (lambda (stx)
    (syntax-parse stx
      [(_ [g:goal ...+] ...+)
       #'(disj
          (lambda ()
            (conj g ...))
          …)])))
```

```
#lang racket

(require minikanren)

(define-relation (append l1 l2 l3)
  (conde
    [(== l1 '())
     (== l2 l3)]
    [(fresh (first rest result)
       (== (cons first rest) l1)
       (== (cons first result) l3)
       (append rest l2 result))]))
```

Consequences of macro-embedding

Mixing with host-language code

```
(define-relation (naturals v)
  (loop recur ([n 0])
    (conde
      [(== v n)]
      [(recur (+ n 1))])))
```

Mixing with host-language code

```
(define-relation (naturals v)
  (loop recur ([n 0])
    (conde
      [(== v n)]
      [(recur (+ n 1))])))
```

Useful when exploring extensions…

But can easily break the DSL’s semantics.

Extension using host-language macros

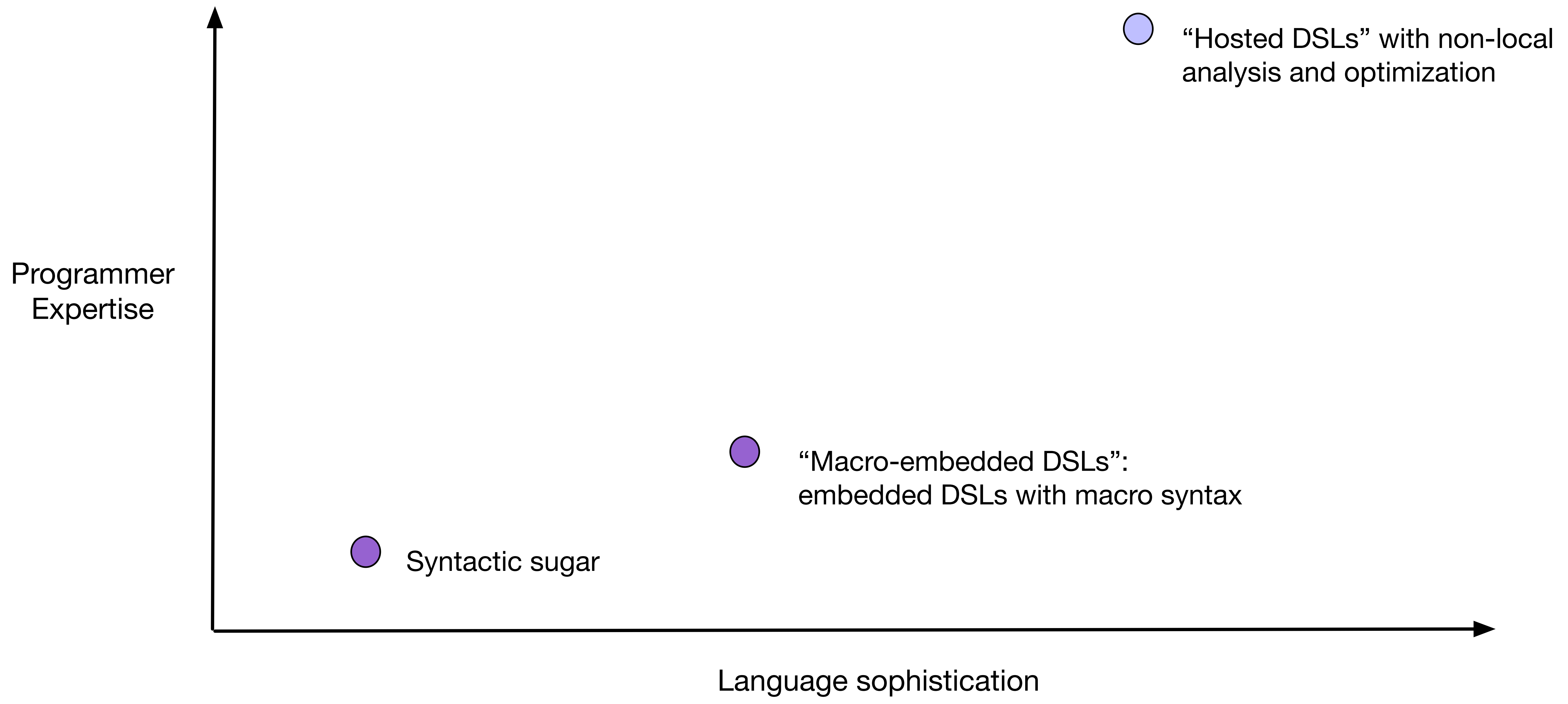
```
#lang racket

(require minikanren)

(define-syntax define-relation/match
  (lambda (stx)
    …))

(define-relation/match (append l1 l2 l3)
  [('() _ _)
   (== l2 l3)]
  [((cons first rest) _ (cons first result))
   (append rest l2 result)])
```

Easy! Just functions and some lightweight macros.

To use the language, just import the library.

Can mix with host-language code.

Can extend with host-language macros.


“Hosted DSLs” with non-local
analysis and optimization
Programmer
Expertise
“Macro-embedded DSLs”:
embedded DSLs with macro syntax
Syntactic sugar
Language sophistication

More sophisticated DSLs

Custom:

- Grammar
- Binding structure
- Static semantics
- Optimizations

All non-local.

Macro-embedding only gives us what we can scrounge from the host language.

More sophisticated DSLs

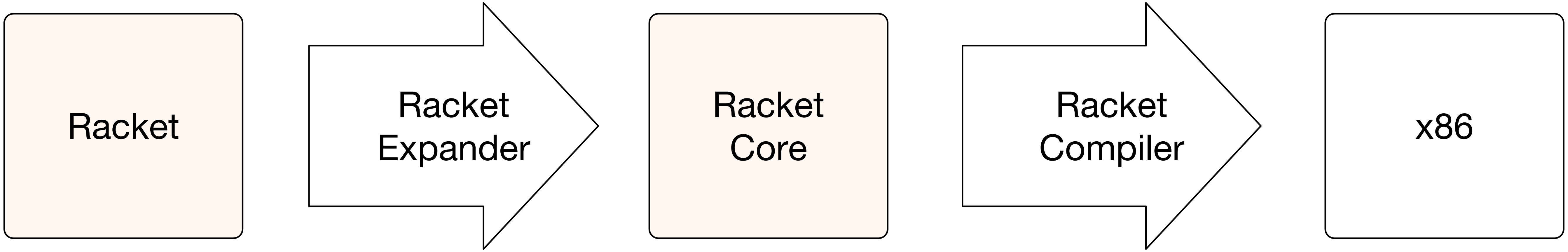
How to support these custom, non-local features…

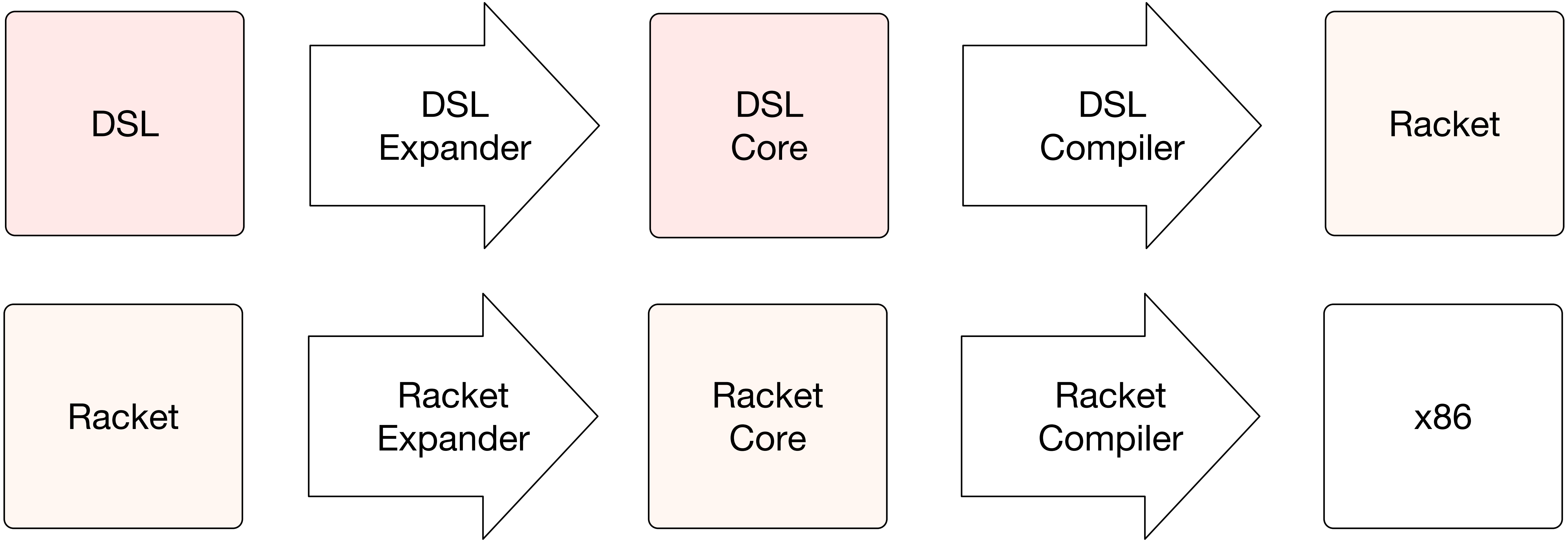
While keeping:

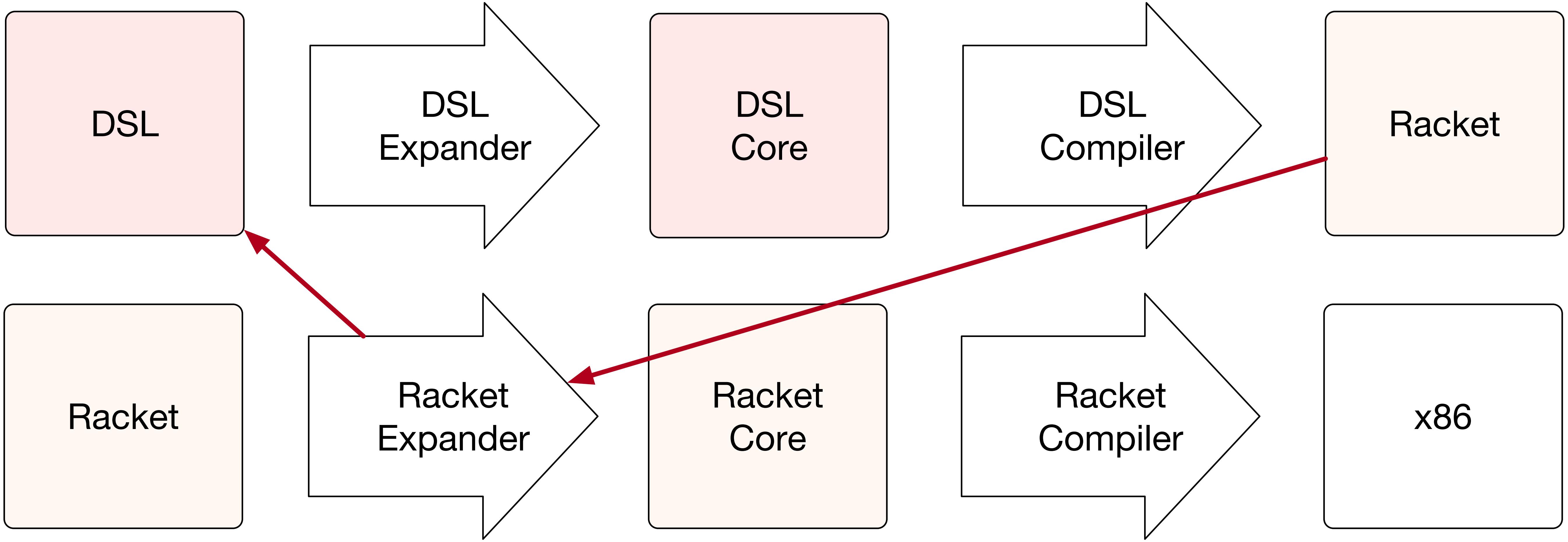
- Integration between DSLs and host
- “Languages as libraries”
- DSL extensibility via macros

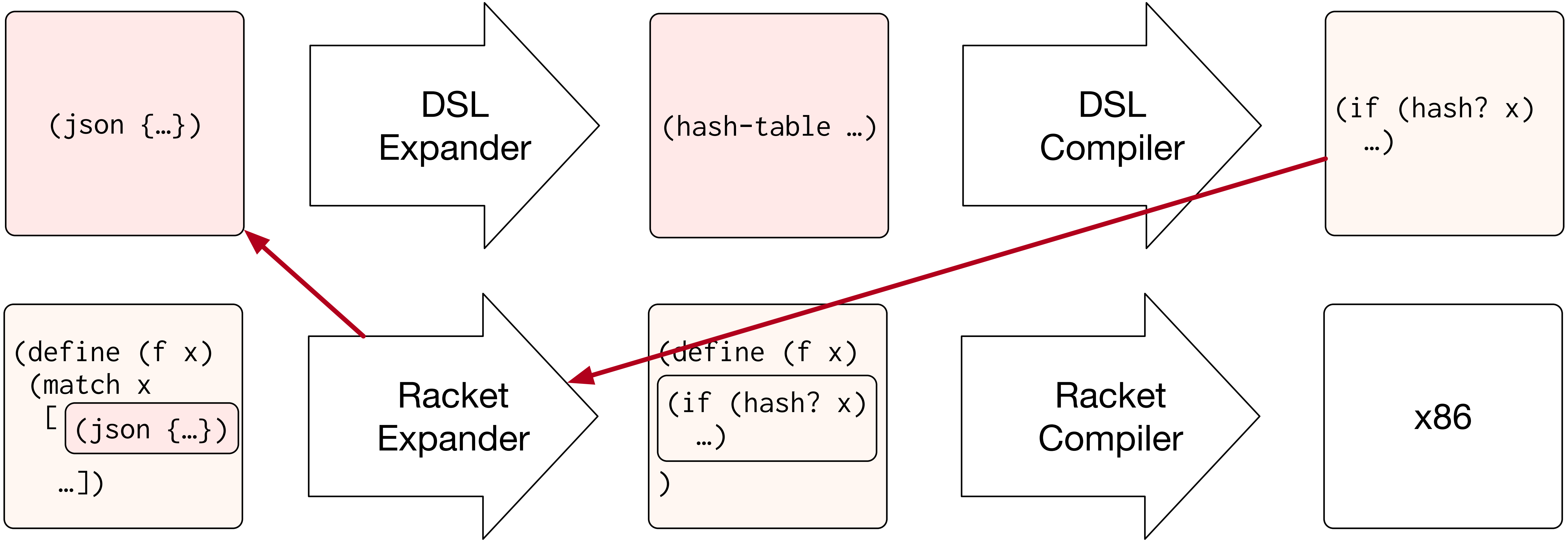
Hosted DSLs

Use a traditional compiler,
but connected to the host-language
macro system.


Racket
Racket
Expander
Racket
Core
Racket
Compiler
x86


DSL
DSL Expander
DSL Core
DSL Compiler
Racket
Racket
Racket Expander
Racket Core
Racket Compiler
x86


DSL
DSL Expander
DSL Core
DSL Compiler
Racket
Racket
Racket Expander
Racket Core
Racket Compiler
x86


(json {…})
DSL Expander
(hash-table …)
DSL Compiler
(if (hash? x) …)
(define (f x) (match x [(json {…}) …])
Racket Expander
(define (f x) (if (hash? x) …))
Racket Compiler
x86

miniKanren core language

```
term := literal
      | lvar
      | (cons term term)

goal := (== term term)
      | (fresh1 (lvar ...) goal)
      | (disj2 goal goal)
      | (conj2 goal goal)
      | (relname term ...)
```

miniKanren core language

```
term := literal
      | lvar
      | (cons term term)

goal := (== term term)
      | (fresh1 (lvar ...) goal)
      | (disj2 goal goal)
      | (conj2 goal goal)
      | (relname term ...)
```

miniKanren syntactic sugar

```
term := ...
      | (quasiquote quoted)

goal := ...
      | (fresh (lvar ...) goal ...)
      | (conde [goal ...] ...)
```

miniKanren core language

```
term := literal
      | lvar
      | (cons term term)

goal := (== term term)
      | (fresh1 (lvar ...) goal)
      | (disj2 goal goal)
      | (conj2 goal goal)
      | (relname term ...)
```

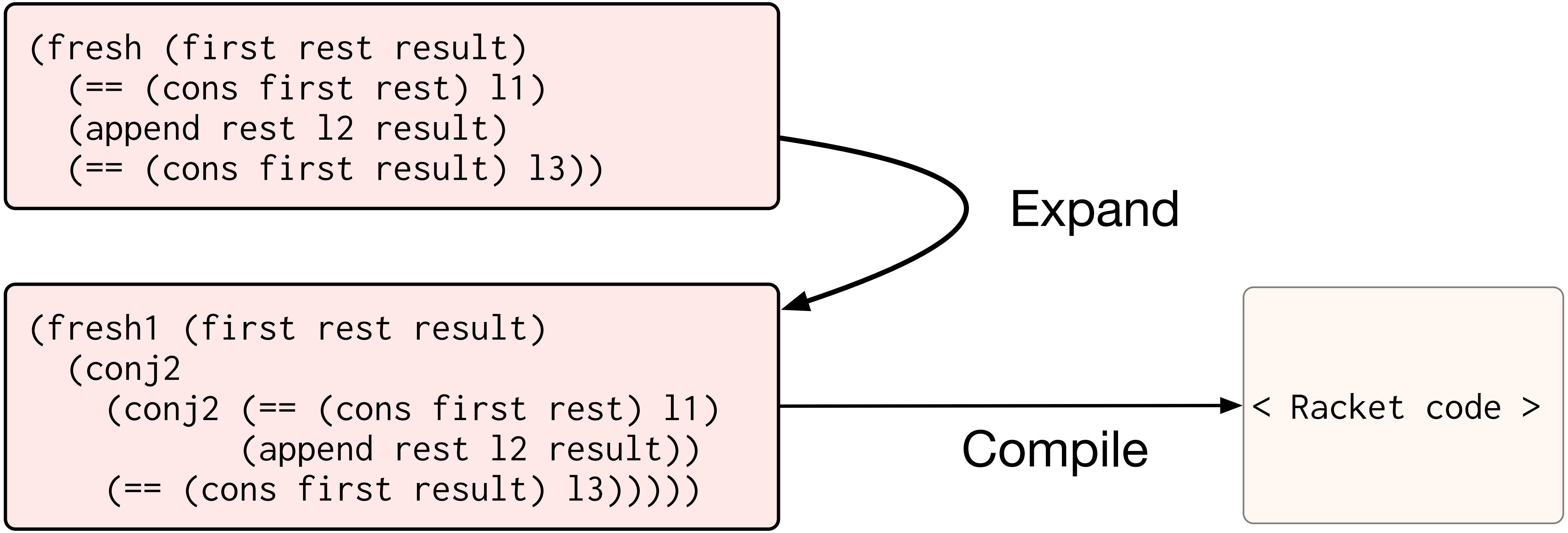
miniKanren syntactic sugar

```
term := ...
      | (quasiquote quoted)

goal := ...
      | (fresh (lvar ...) goal ...)
      | (conde [goal ...] ...)
```

Racket “interface macros”

```
racket-def := ...
            | (define-relation (relname lvar ...) goal)

racket-exp := ...
            | (run n (lvar ...) goal)
```

```
(define-relation (append l1 l2 l3)
  (conde
    [(== l1 '())
     (== l2 l3)]
    [(fresh (first rest result)
       (== (cons first rest) l1)
       (append rest l2 result)
       (== (cons first result) l3))]))
```


(fresh (first rest result)
  (== (cons first rest) l1)
  (append rest l2 result)
  (== (cons first result) l3))
Expand
(fresh1 (first rest result)
  (conj2
    (conj2 (== (cons first rest) l1)
           (append rest l2 result))
    (== (cons first result) l3))))
Compile
< Racket code >

```
(fresh (first rest result)
  (== (cons first rest) l1)
  (append rest l2 result)
  (== (cons first result) l3))
```

Expand

```
(fresh1 (first rest result)
  (conj2
    (conj2 (== (cons first rest) l1)
           (append rest l2 result))
    (== (cons first result) l3)))))
```

Transform

(Improve search behavior)

```
(fresh1 (first rest result)
  (conj2
    (conj2 (== (cons first rest) l1)
           (== (cons first result) l3))
    (append rest l2 result)))))
```

Compile

< Racket code >

Benefits:

- Enforced DSL grammar
  miniKanren terms and goals are separated
- Enforced DSL static semantics
  Relation arity
- Domain-specific analysis and transformation
  Unification lifting

Integration Between Languages

```
(define-relation (naturals v)
  (loop recur ([n 0])
    (conde
      [(== v n)]
      [(recur (+ n 1))])))
```

Integration Between Languages

```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))])))))
```

Integration Between Languages

```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))])))))
```

`loop` body scope

Relation body scope

Integration Between Languages

```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))])))))
```

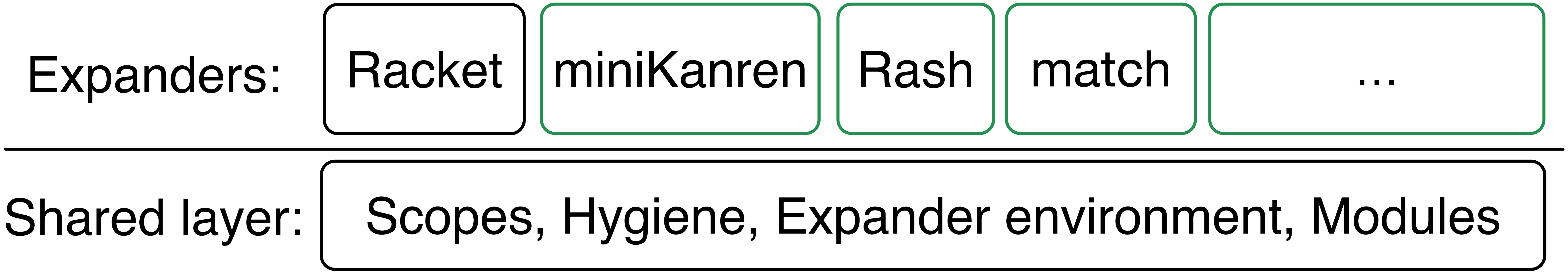
Cross-language name reference

`loop` body scope

Relation body scope

Separate out and expose the language-independent parts of Racket's expander, and reuse them in DSL expanders.

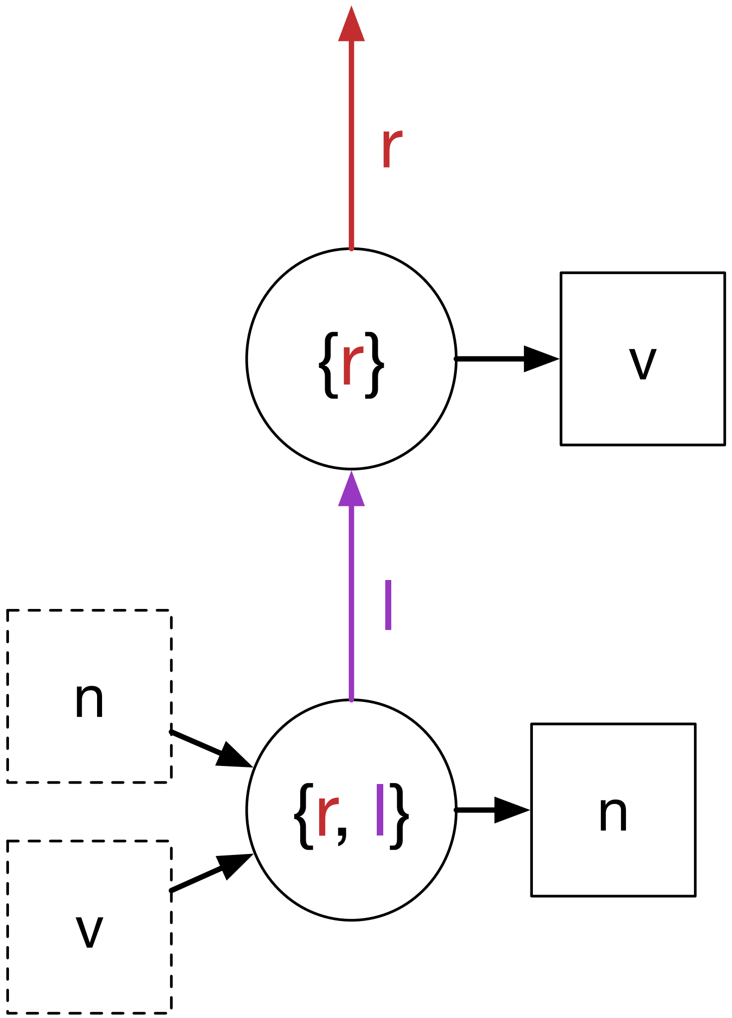
Expanders:	Racket	miniKanren	Rash	match	...
Shared layer:	Scopes, Hygiene, Expander environment, Modules				



New API to make this shared layer easy to reuse.

Reuse: Scope

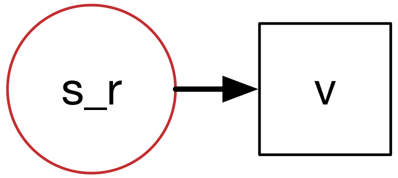
```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))])))))
```



```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))])))))
```

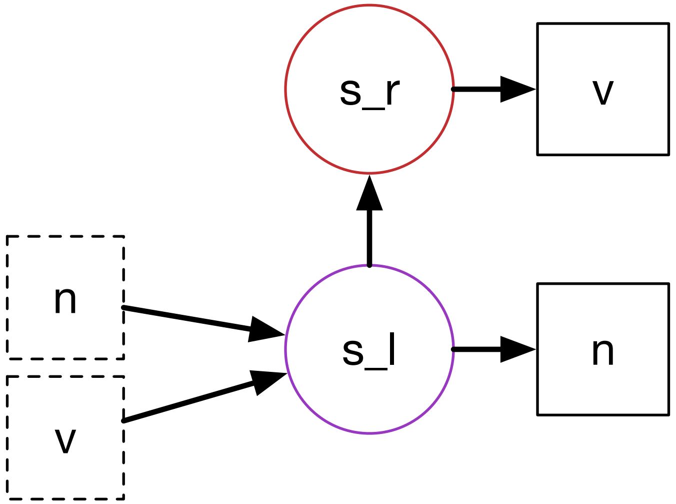
```
(define-relation (naturals v)
  (racket
   (loop recur ([n 0])
     (mkgoal
       (conde
         [(== v n)]
         [(racket (recur (+ n 1)))])))))
```

```
expOk(s_r, (loop recur ([n 0])
             (mkgoal
               (conde
                 [(== v n)]
                 [(racket (recur (+ n 1)))]))))
```



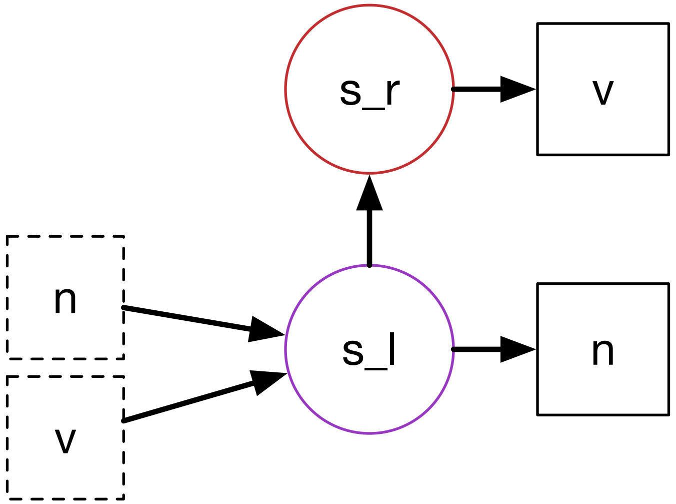
```
(define-relation (naturals v)
  (racket
   (loop recur ([n 0])
     (mkgoal
       (conde
         [(== v n)]
         [(racket (recur (+ n 1)))])))))
```

```
expOk(s_r, (loop recur ([n 0])
             (mkgoal
               (conde
                 [(== v n)]
                 [(racket (recur (+ n 1)))]))))
```

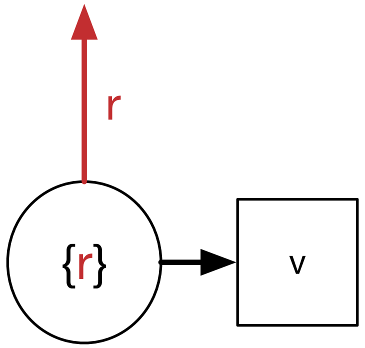


```
(define-relation (naturals v)
  (racket
   (loop recur ([n 0])
     (mkgoal
       (conde
         [(== v n)]
         [(racket (recur (+ n 1)))])))))
```

```
expOk(s_r, (loop recur ([n 0])
             (mkgoal
               (conde
                 [(== v n)]
                 [(racket (recur (+ n 1)))]))))
```

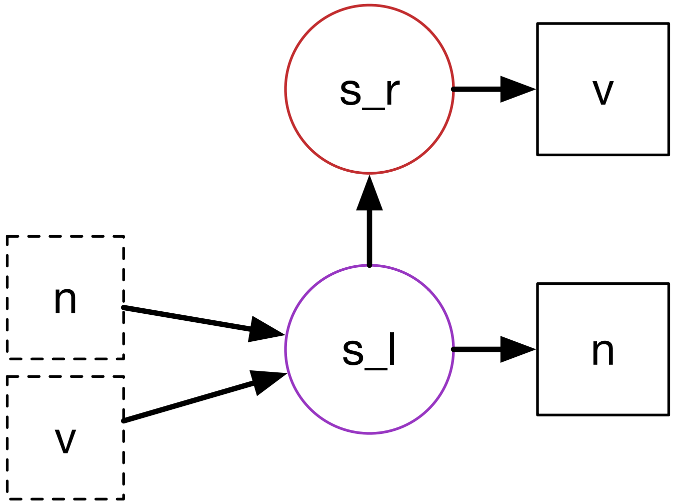



```
(expand #'(loop recur ([n_{r} 0])
            (mkgoal
              (conde
                [(== v_{r} n)]
                [(racket (recur (+ n_{r} 1)))]))))
```

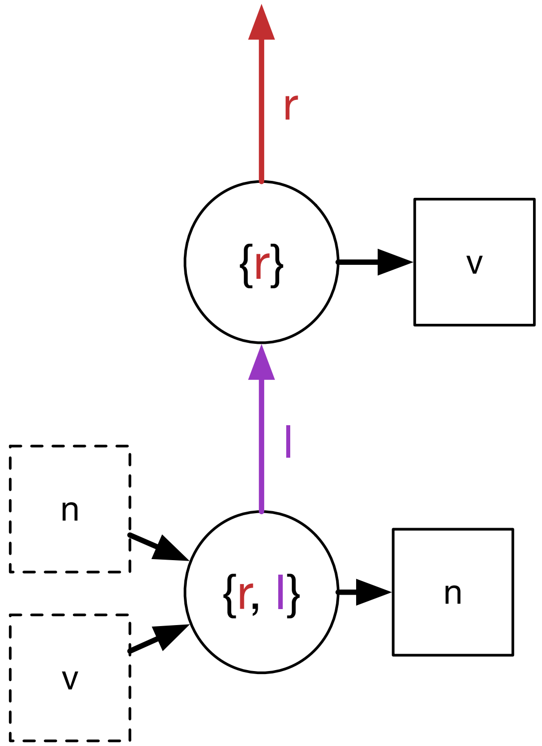


```
(define-relation (naturals v)
  (racket
   (loop recur ([n 0])
     (mkgoal
       (conde
         [(== v n)]
         [(racket (recur (+ n 1)))])))))
```

```
expOk(s_r, (loop recur ([n 0])
             (mkgoal
               (conde
                 [(== v n)]
                 [(racket (recur (+ n 1)))]))))
```




```
(expand #'(loop recur ([n_{r} 0])
            (mkgoal
              (conde
                [(== v_{r} n)]
                [(racket (recur (+ n_{r} 1)))]))))
=>
#'(loop recur ([n_{r,l} 0])
    (mkgoal
     (conde
       [(== v_{r,l} n)]
       [(racket (recur (+ n_{r,l} 1)))])))
```



Scope in the presence of macros: hygiene

We want programmers to be able to extend both Racket and DSLs with macros.

```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))]))))))
```

Scope in the presence of macros: hygiene

We want programmers to be able to extend both Racket and DSLs with macros.

```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))])))))
```

Expansion…

- Moves syntax between scopes and modules
- Combines syntax from different origins

Like the need to avoid capture in λ substitution, but more subtle.

Scope in the presence of macros: hygiene

We want programmers to be able to extend both Racket and DSLs with macros.

```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))])))))
```

Expansion…

- Moves syntax between scopes and modules
- Combines syntax from different origins

Like the need to avoid capture in λ substitution, but more subtle.

Automatic hygiene motivates Racket’s model of scope.

Scope in the presence of macros: hygiene

What if `loop` is defined by a macro?

```
(loop recur ([x init]) b)
->
(let ([v init])
  (letrec ([recur (lambda (x) b)])
    (recur v)))
```

```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))]))))))
```

Scope in the presence of macros: hygiene

What if `loop` is defined by a macro?

```
(loop recur ([x init]) b)
->
(let ([v init])
  (letrec ([recur (lambda (x) b)])
    (recur v)))
```

```
(define-relation (naturals v)
 (racket
  (loop recur ([n 0])
    (mkgoal
      (conde
        [(== v n)]
        [(racket (recur (+ n 1)))])))))
```

Expands to

```
(define-relation (naturals v)
 (racket
  (let ([v 0])
    (letrec ([recur
              (lambda (n)
                (mkgoal
                  (conde
                    [(== v n)]
                    [(racket (recur (+ n 1)))])))])
    (recur v)))))
```

Scope in the presence of macros: hygiene

What if `loop` is defined by a macro?

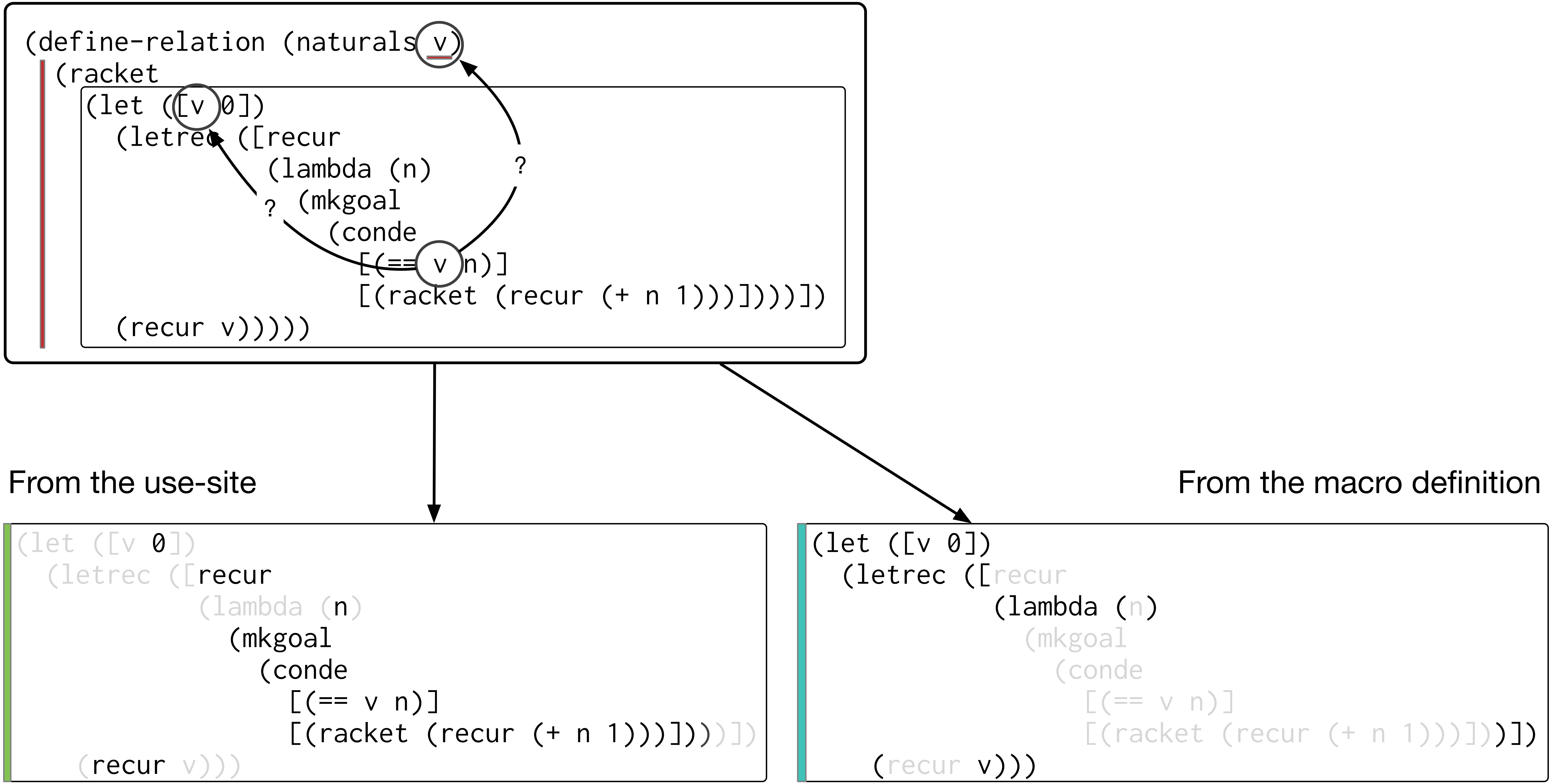
```
(loop recur ([x init]) b)
->
(let ([v init])
  (letrec ([recur (lambda (x) b)])
    (recur v)))
```

```
(define-relation (naturals v)
  (racket
    (loop recur ([n 0])
      (mkgoal
        (conde
          [(== v n)]
          [(racket (recur (+ n 1)))])))))
```

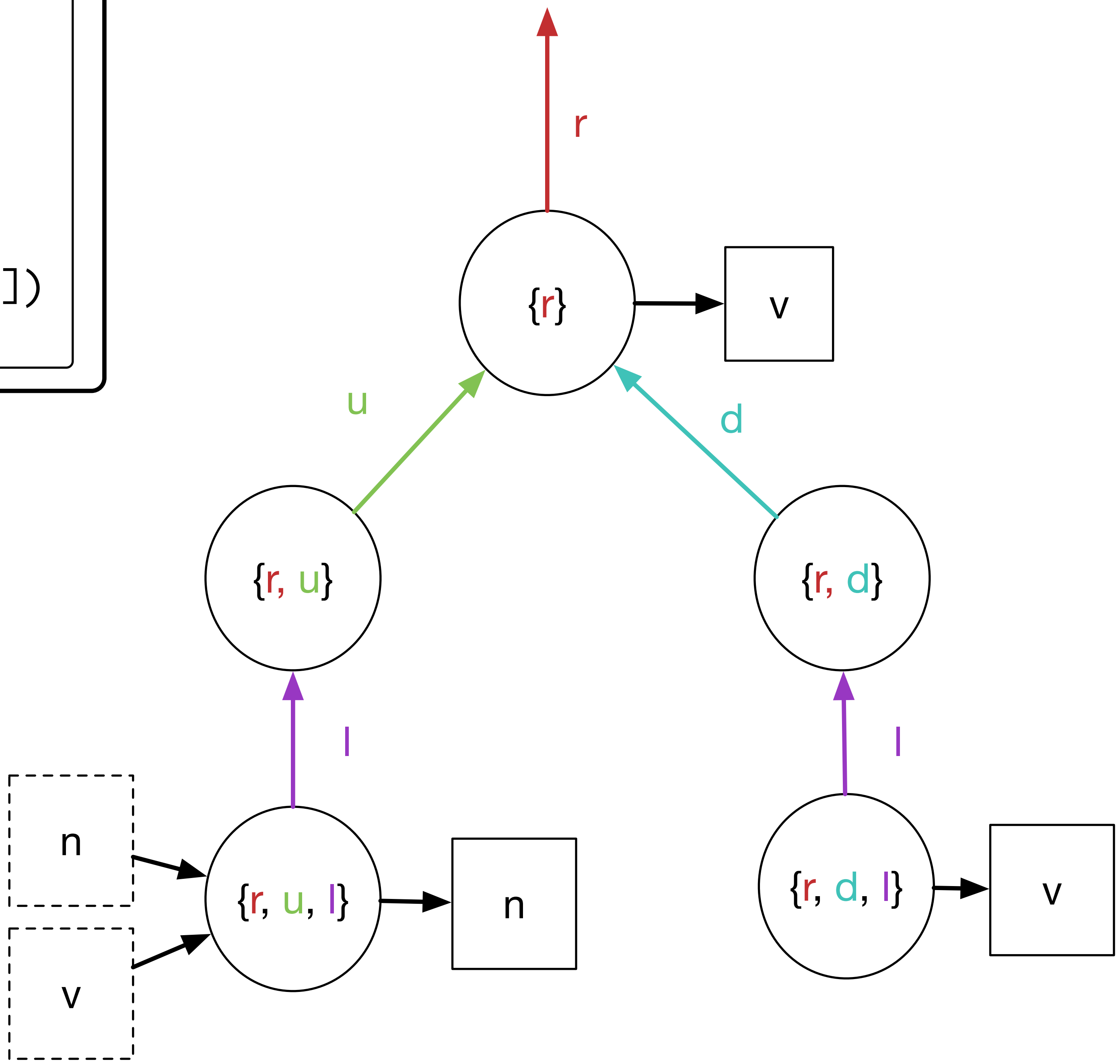
Expands to

```
(define-relation (naturals v)
  (racket
    (let ([v 0])
      (letrec ([recur
                 (lambda (n)
                   (mkgoal
                     (conde
                       [(== v n)]
                       [(racket (recur (+ n 1)))])))])
      (recur v)))))
```

```
(define-relation (naturals v)
  (racket
    (let ([v 0])
      (letrec ([recur
                 (lambda (n)
                   (mkgoal
                     (conde
                       [(== v n)]
                       [(racket (recur (+ n 1)))])))])
        (recur v)))))
```


(define-relation (naturals v)
(racket
(let ([v 0])
(letrec ([recur
(lambda (n)
(mkgoal
(conde
[(== v n)]
[(racket (recur (+ n 1)))])))])
(recur v)))))
?
?
From the use-site
From the macro definition
(let ([v 0])
(letrec ([recur
(lambda (n)
(mkgoal
(conde
[(== v n)]
[(racket (recur (+ n 1)))])))])
(recur v)))
(let ([v 0])
(letrec ([recur
(lambda (n)
(mkgoal
(conde
[(== v n)]
[(racket (recur (+ n 1)))])))])
(recur v)))

```
(define-relation (naturals v)
  (racket
    (let ([v 0])
      (letrec ([recur
                (lambda (n)
                  (mkgoal
                    (conde
                      [(== v n)]
                      [(racket (recur (+ n 1)))])))])
      (recur v)))))
```



Statix

- Scope:
  ```
  new s_let
  s_let -P-> s
  ```
- Binding:
  ```
  !lvar[x] in s_let
  x in s_let |-> [(_,_)]
  ```

extDSL API

- Scope:
  ```
  (with-scope s_let …)
  (add-scope #'body s_let)
  ```
- Binding:
  ```
  (bind! #'x (lvar))
  (lookup #'x)
  ```
- Hygiene:
  ```
  (apply-as-transformer f arg …)
  ```

(Break and questions)

Reuse: Expander Environment

```
define ->
  (rkt-macro #<procedure>)
x ->
  (racket-var)
```

```
(define x 5)
; ->
(define-values (x) 5)
x
```

Reuse: Expander Environment

```
define ->
  (rkt-macro #<procedure>)
x ->
  (racket-var)
conde ->
  (goal-macro #<procedure>)
append ->
  (relation 3)
l1, l2, l3 ->
  (lvar)
```

```
(define x 5)
; ->
(define-values (x) 5)
x

(define-relation (append l1 l2 l3)
  (conde
     …))

(run* (l1 l2)
  (append l1 l2 '(1 2)))
```

Reuse: Modules and Separate Compilation

Expander Environment

```
append ->
  (relation 3)
```

mk/lists

```
(provide append)
(define-relation (append l1 l2 l3)
  …)
```

Reuse: Modules and Separate Compilation

Expander Environment

```
append ->
  (relation 3)
```

`mk/lists`

```
(provide append)
(define-relation (append l1 l2 l3)
  …)
```

`mk/tests/lists`

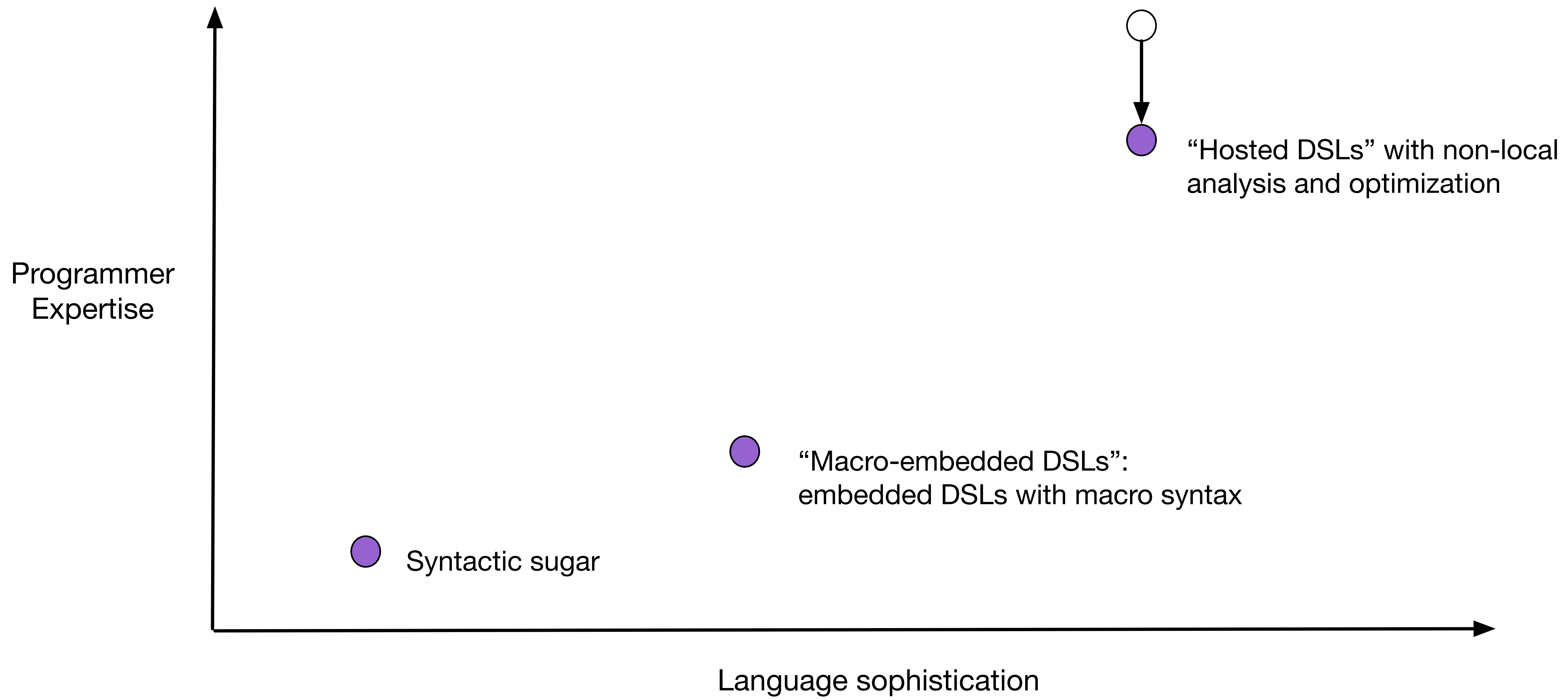
```
(require mk/lists)
(run* (l1 l2)
  (append l1 l2 '(1 2)))
```

Expander Environment

```
append ->
  (relation 3)
```

Benefits of reuse

- Scope and binding work across languages
- DSL macros behave like host-language macros
- DSLs reuse host’s module system
- IDE understands DSL scope and binding



“Hosted DSLs” with non-local analysis and optimization
Programmer Expertise
“Macro-embedded DSLs”: embedded DSLs with macro syntax
Syntactic sugar
Language sophistication

Remaining problem: DSL expanders are low-level, procedural, and demand deep understanding of Racket’s macro system.

Declaring a language.

```
(define-variable-class lvar)
(define-variable-class relname)

(define-nonterminal term
  literal
  lvar
  (cons term term))

(define-nonterminal goal
  (== term term)
  (fresh (v:lvar ...+) g:goal)
    #:binding { (! v) g }
  (disj2 goal goal)
  (conj2 goal goal)
  (relname term ...+))
```

Binding specifications

```
(define-variable-class lvar)
(define-variable-class relname)

(define-nonterminal term
  literal
  lvar
  (cons term term))

(define-nonterminal goal
  (== term term)
  (fresh (v:lvar ...+) g:goal)
    #:binding { (! v) g }
  (disj2 goal goal)
  (conj2 goal goal)
  (relname term ...+))
```

Declaring extension points

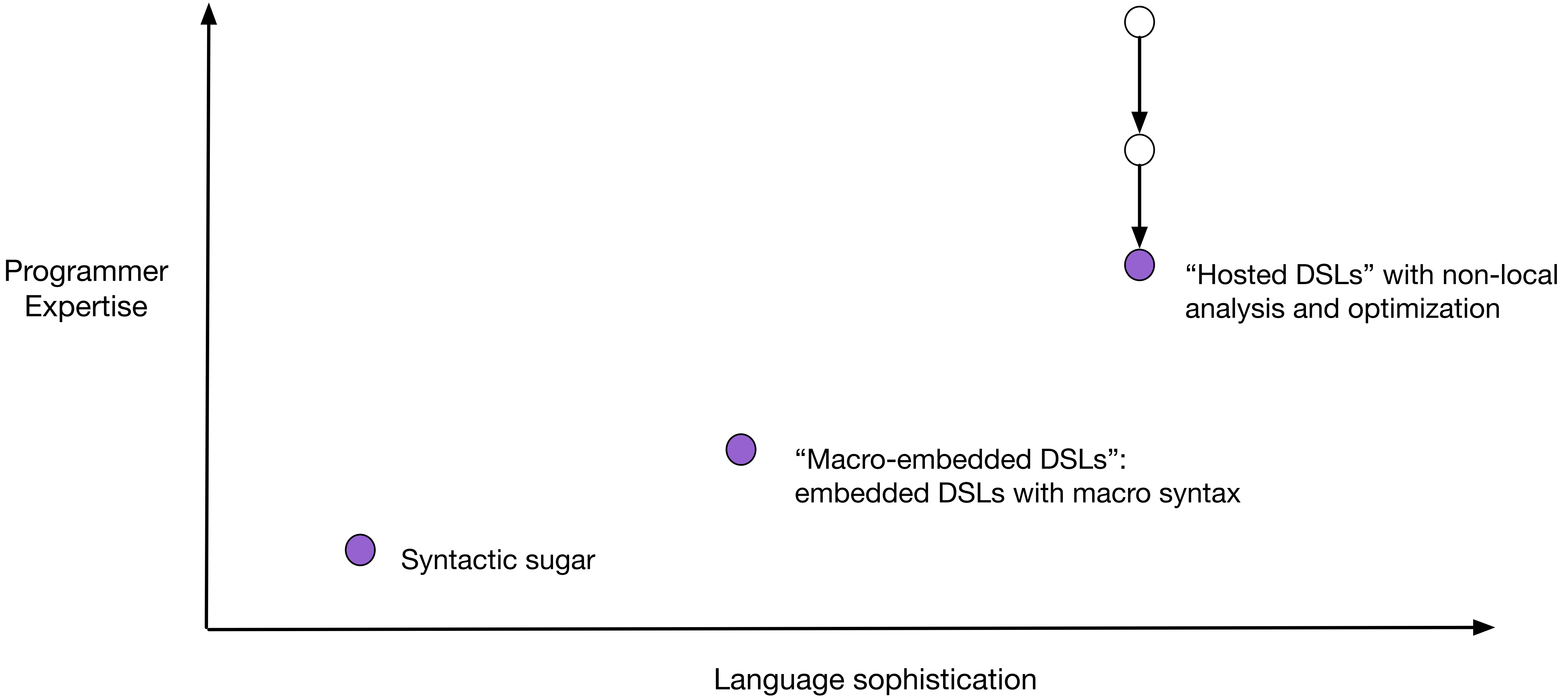
```
(define-variable-class lvar)
(define-variable-class relname)

(define-extension-class goal-macro)

(define-nonterminal term
  (quote datum)
  literal
  lvar
  (cons term term))

(define-nonterminal goal
  #:allow-extension goal-macro
  (== term term)
  (fresh (lvar ...+) goal ...+)
    #:binding { (! lvar) goal }
  (disj goal ...+)
  (conj goal ...+)
  (relname term ...+))
```

Key goal: make simple, untyped languages easy.

How can we

- integrate type rules
- handle dependent binding structures

without making the common case complicated?


Programmer Expertise
“Hosted DSLs” with non-local analysis and optimization
“Macro-embedded DSLs”: embedded DSLs with macro syntax
Syntactic sugar
Language sophistication

This talk:

- Macro-embedding is easy and great for simple DSLs and design exploration.
- Custom DSL expanders and compilers enable more sophisticated features, and can integrate with the host via our new API.

- In progress: Declarative definition of extensible hosted DSLs, using language workbench ideas.

This talk:

- Macro-embedding is easy and great for simple DSLs and design exploration.
- Custom DSL expanders and compilers enable more sophisticated features, and can integrate with the host via our new API.

- In progress: Declarative definition of extensible hosted DSLs, using language workbench ideas.

Questions?

Details of binding specification…

Binding specifications

```
(fresh (first rest result)
  (conj2
    (conj2
      (== (cons first rest) l1)
      (== (cons first result) l3))
    (append rest l2 result)))
```

Binding specifications

```
(fresh (first rest result)
  (conj2
    (conj2
      (== (cons first rest) l1)
      (== (cons first result) l3))
    (append rest l2 result)))
```

```
(fresh (v:lvar ...+) g:goal)
  #:binding { (! v) g }
```

Exported bindings

```
(match '(1 2 3)
  [(cons first (cons second tail))
   second]
  [_ (error)])
```

Exported bindings

```
(match '(1 2 3)
  [(cons first (cons second tail))
   second]
  [_ (error)])
```

```
(define-nonterminal match-clause
  [p:pat e:racket-expr]
    #:binding { (! p) e })
```

Exported bindings

```
(match '(1 2 3)
  [(cons first (cons second tail))
   second]
  [_ (error)])
```

```
(define-nonterminal match-clause
  [p:pat e:racket-expr]
    #:binding { (! p) e })

(define-nonterminal pat
  literal
  v:pvar
    #:binding (^ v)
  (cons p1:pat p2:pat)
    #:binding (^ p1 p2))
```

Mutually recursive bindings

```
(block
  (define f
    (lambda (x) (g x)))
  (define g
    (lambda (x) (f x)))
  (f 5))
```

Mutually recursive bindings

```
(block
  (define f
    (lambda (x) (g x)))
  (define g
    (lambda (x) (f x)))
  (f 5))
```

```
(define-nonterminal def-or-expr
  (define v:rlvar e:expr)
    #:binding (^ v)
  e:expr)
```

Mutually recursive bindings

```
(block
  (define f
    (lambda (x) (g x)))
  (define g
    (lambda (x) (f x)))
  (f 5))
```

```
(define-nonterminal def-or-expr
  (define v:rlvar e:expr)
    #:binding (^ v)
  e:expr)

(define-nonterminal expr
  …
  (block body:def-or-expr ...)
    #:binding { (! body) body })
```