A language workbench as a library

Michael Ballantyne PRL @ Northeastern University

Parts in collaboration with: Matthias Felleisen, Alexis King, Jason Hemann

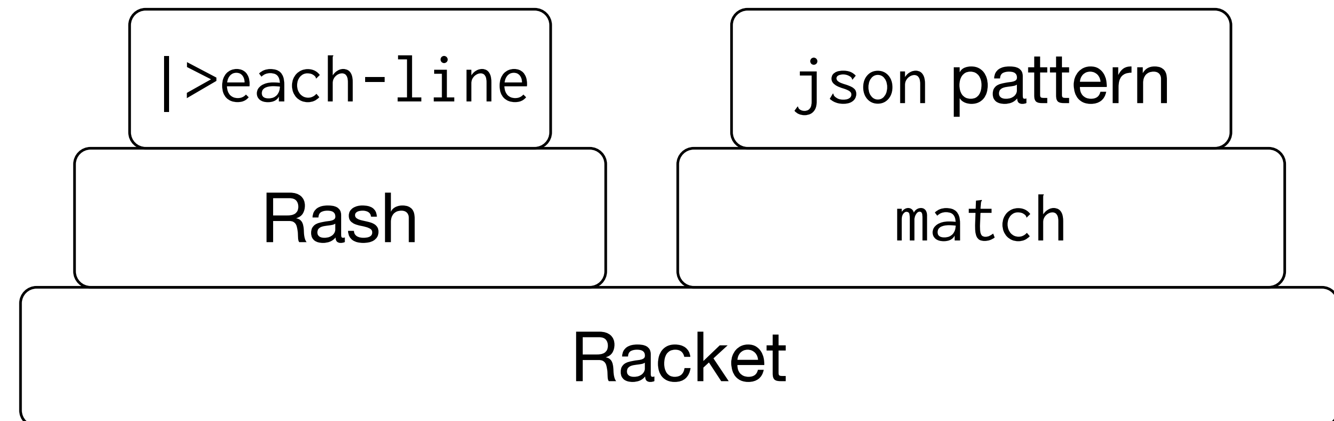
An extensible language
for language-oriented
programming.

```
#lang rash

(require racket/match json-pattern)

(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
```



Many cool LOP systems:

- Spoofax
- Silver
- Jetbrains MPS
- Rascal
- SugarJ
- Racket

But none have taken off.

Preview

Problem: macros and workbenches each have complementary structural problems preventing mainstream adoption.

Solution: layer language workbenches on top of macro systems.

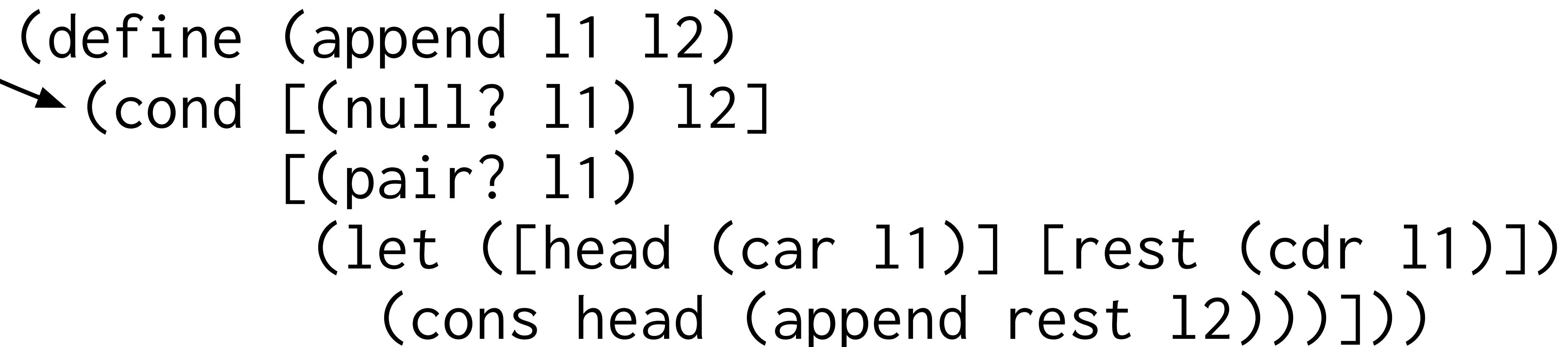
I’ve prototyped this idea with `syntax-spec` in Racket, but the real prize is to bring it to languages such as Rust and Scala.

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))]))
```



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)]))])))
```

```
#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

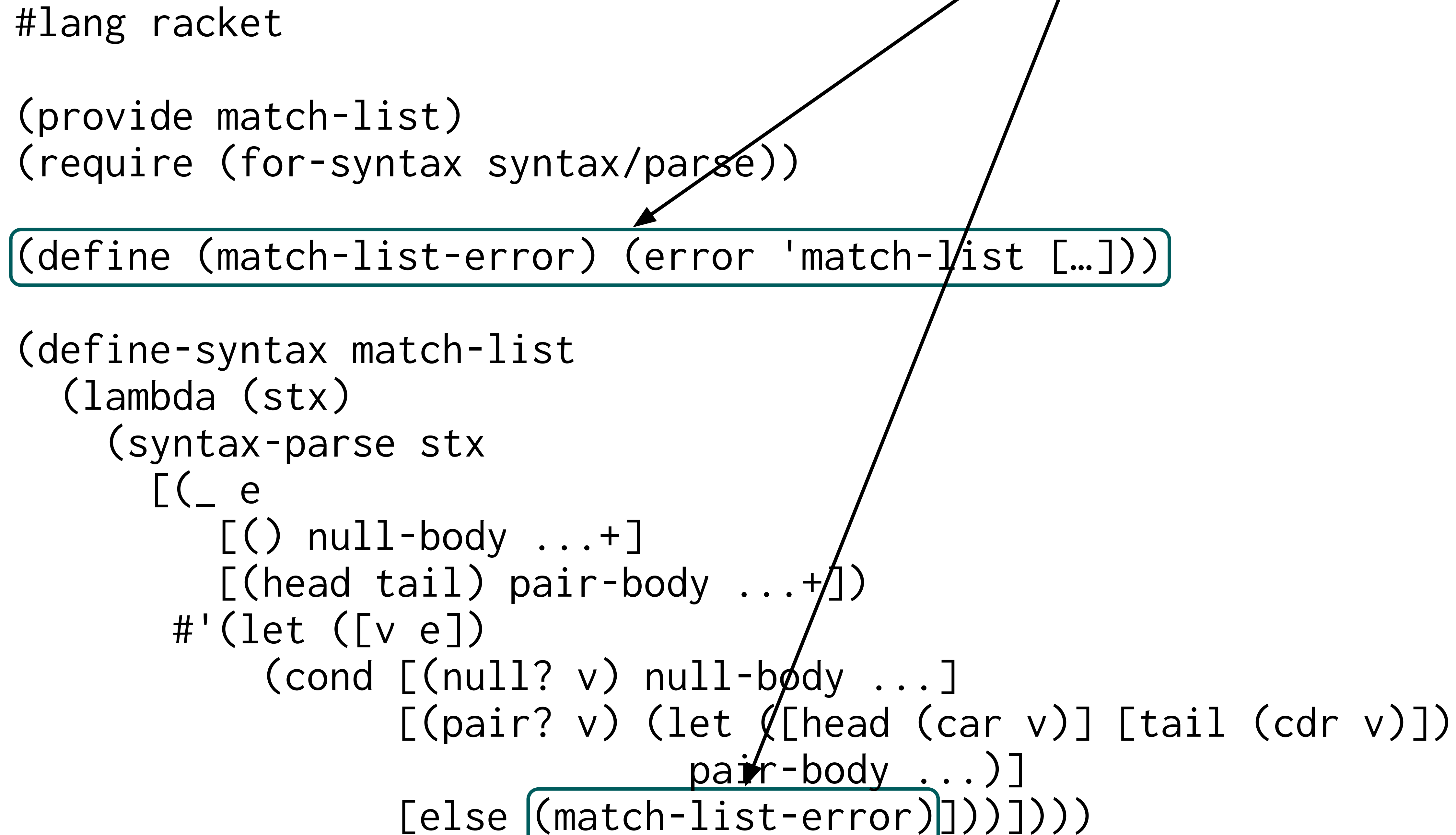
Template

```
#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)]))])))

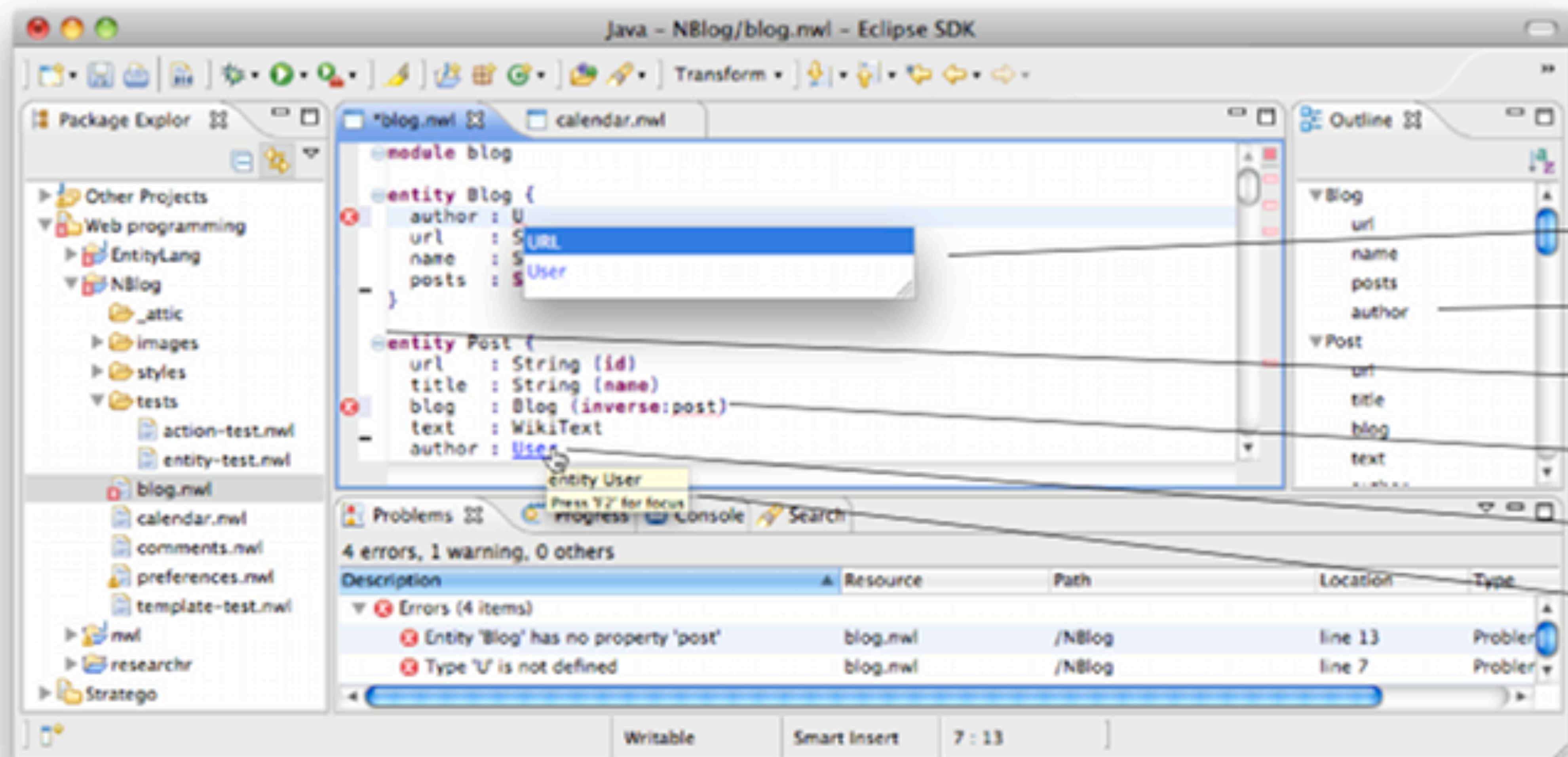
Language workbenches

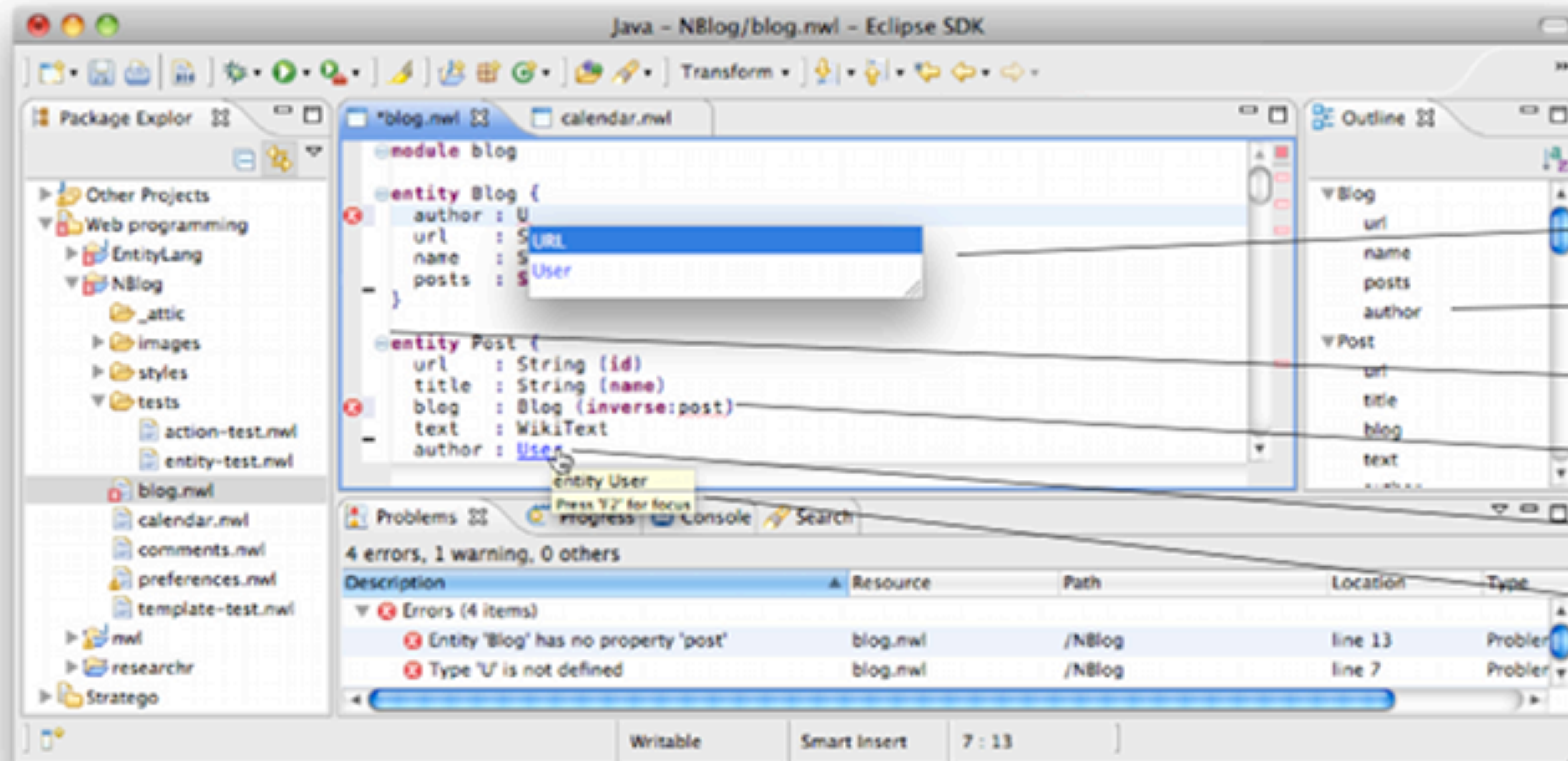
Functions.sdf3

```
6 context-free syntax
7
8   Dec.FunDecs = <<{FunDec "\n"}+>> {longe
9
10  FunDec.ProcDec = <
11    function <Occ>(<{FArg ", "}*>) =
12      <Exp>
13  >
14
15  FunDec.FunDec = <
16    function <Occ>(<{FArg ", "}*>) : <Typ
17      <Exp>
18  >
19
20  FArg.FArg = <<Occ> : <Type>>
21
```

static-semantics.stx

```
236 rules // function declarations
237
238   // Parameters: In [function id(... id1: id2 ...) = exp]
239   // scope of the parameter id1 lasts throughout the func
240   // body exp
241
242   decOk(s, s_outer, FunDecs(fdecs)) :-
243     fdecsOk(s, s_outer, fdecs).
244
245   fdecsOk        maps fdecOk(*, *, list(*))
246   fdecOk       : scope * scope * FunDec
247
248   fdecOk(s, s_outer, d@ProcDec(f, args, e)) :- {s_fun Ts}
249     new s_fun, s_fun -P-> s,
250     typesOfArgs(s_fun, s_outer, args) == Ts,
251     declareVar(s, f, FUN(Ts, UNIT())),
```


Java – NBlog/blog.nwl – Eclipse SDK
Content completion
Outline view
Code folding
Error markers
Reference resolving
Hover help



With error recovery, incrementally!

A provocation: Neither has what it takes for wide adoption.

A provocation: Neither has what it takes for wide adoption.

	Macros	Workbenches
In-language	✓	✗
Simple to start	✓	✗
Scales well	✗	✓
Good IDE services	✗	✓

```
#lang racket

(define-syntax SDF …)
(define-syntax Statix …)

(SDF
  Dec.FunDecs = <<{FunDec "\n"}+>> {longe

  FunDec.ProcDec = <
    function <Occ>(<{FArg ", "}*>) =
      <Exp>
  >

  FunDec.FunDec = <
    function <Occ>(<{FArg ", "}*>) : <Typ
      <Exp>
  >

  FArg.FArg = <<Occ> : <Type>>
)

(Statix
  decOk(s, s_outer, FunDecs(fdecs)) :-
    fdecsOk(s, s_outer, fdecs).

  fdecsOk      maps fdecOk(*, *, list(*))
  fdecOk      : scope * scope * FunDec

  fdecOk(s, s_outer, d@ProcDec(f, args, e))
    new s_fun, s_fun -P-> s,
    typesOfArgs(s_fun, s_outer, args) == Ts,
    declareVar(s, f, FUN(Ts, UNIT())),
)

(Function Distance ([p1 : Point] [p2 : Point]) …)
```

	Workbench as a library
In-language	✓
Simple to start	✓
Scales well	✓
Good IDE services	?

`syntax-spec`

A language workbench as a macro library with:
- DSL grammar and boundary with Racket
- Binding specification
- DSL extensibility

But…
- Not yet great IDE support
- In Racket, not a mainstream language.

Demo

```
#;(-> (listof number?) number?)
(define-flow root-mean-square
  (-< (~> length (as l))
      (~> Δ (>< sqr) + (/ l) sqrt)))
```

Qi

```
(define (modal-mixin base%)
  (class base%
    (super-new)
    (field [mode 'command])
    (define/public (toggle-mode)...)
    (define/override (on-key key)...)))
```

racket/class

```
#; (-> Circle Circle)
(define (shift-left circle)
  (update circle
    [(list color radius (cons x y))
     (modify! x sub1)]))
```

Lens-match

The expressiveness of the macro system's reflective API determines how workbench-specified DSLs can interact with the host.

Racket's macro API:
- Cross-language binding
- Hygienic macro expansion.

Other possible features:
- Parsing
- Type checking
- Autocompletion

$\text{enter-scope} : \text{exp-st} \to \langle \text{scp}, \text{exp-st} \rangle$

$\text{add-scope} : \text{stx}, \text{scp} \to \text{stx}$

$\text{bind} : \text{exp-st}, \text{id}, \text{env-val} \to \langle \text{id}, \text{exp-st} \rangle$

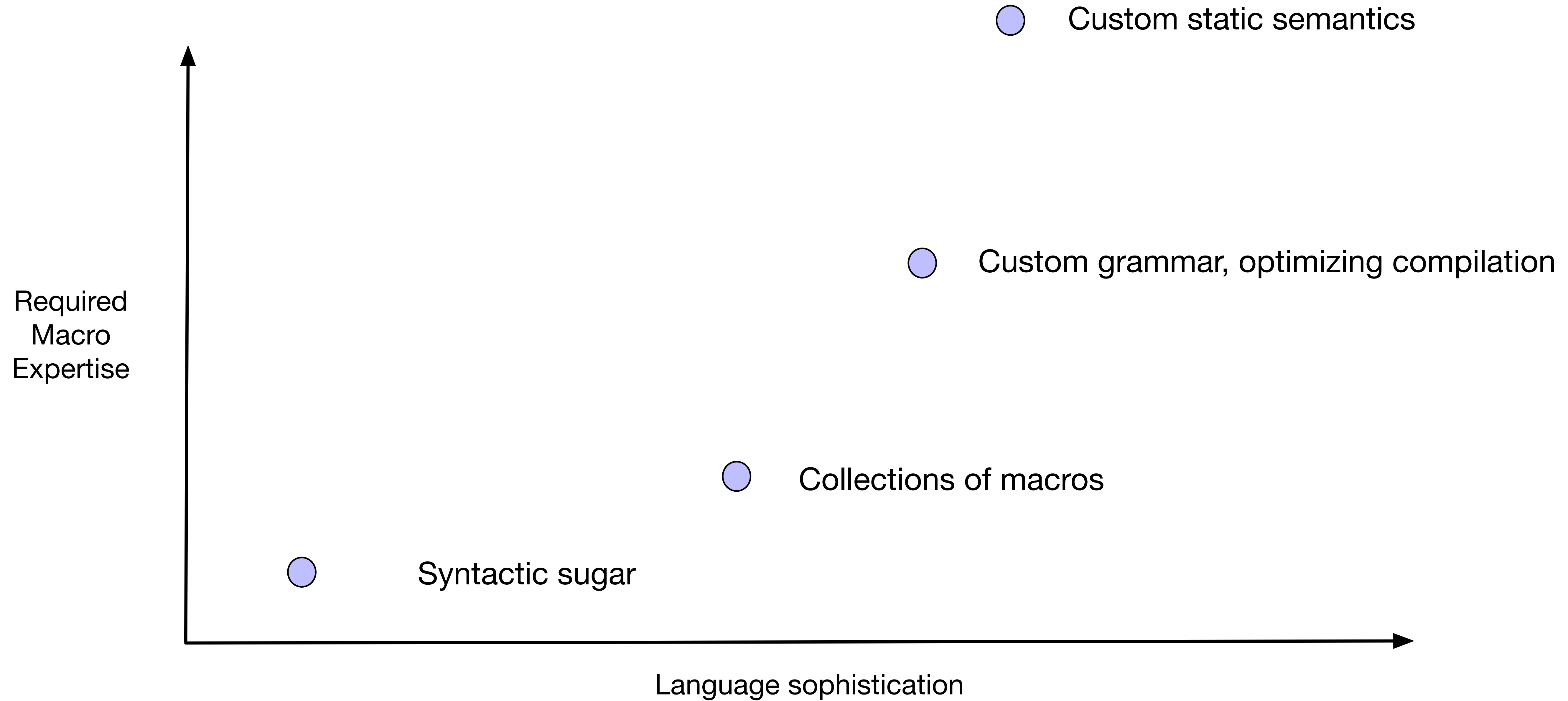
$\text{lookup} : \text{exp-st}, \text{id} \to \text{env-val} \cup \text{False}$

$\text{eval-transformer} : \text{exp-st}, \text{stx} \to \text{transformer}$

$\text{apply-transformer} : \text{exp-st}, \text{transformer}, \text{stx} \to \langle \text{stx}, \text{exp-st} \rangle$

A reflective macro API is a smaller ask of mainstream language implementers than an integrated language workbench.

The path from programmer to DSL creator



End

Scaling up

Improving expressivity
- Binding structures
- Type systems

IDE support
- Autocompletion
- Error recovery
- Incrementality

Applying the idea in other languages
- Rust, Julia, Clojure, Lean 4?