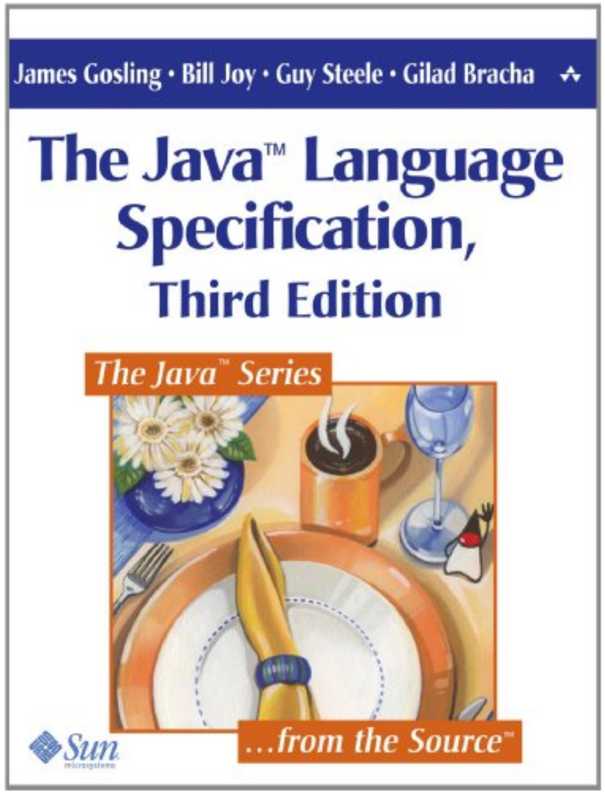
Languages in Racket

Matthew Flatt

University of Utah

Why Language Extensibility?

Your programming language isn't good enough, yet



684 pages

[insert demo here]

example languages

Different levels of language extension...

- Syntactic abstraction

```
(define (roman->number str)
  (rx-case str
   ["([XV]*)(I{1,4})" (xv i) (+ (roman->number xv)
                                (string-length i))]
   ....))
```

- New language constructs

```
(class object% (define/public method ....) ....)
```

- New languages

```
(: factorial (Number -> Number))

@section{Hello}

int f(int n) { return n+1; }
```

... in one framework

Implementing a Text Adventure Game

```
You're standing in a field.
There is a house to the north.
> north
You are standing in front of a house.
There is a door here.
> open door
The door is locked.
>
```

[insert demo here]

play the game

Implementing a Text Adventure Game

- **Places**
- **Things**
- **Verbs**
  - global intransitive verbs
  - place-local intransitive verbs
  - thing-specific transitive verbs

Implementing a Text Adventure Game

- **Places**
- **Things**
- **Verbs**
  - global i… ansitive verbs
  - place-loca… bs
  - thing-spec…

Objects?

Need not only serialize, but save & restore variables

Methods?

Must convert between string command and method call

Domain-Specific Programming

The **programming language** approach:

- Provide expressive constructs

Domain-Specific Programming

The **Lisp** approach:

- Provide expressive constructs
- Enable syntactic abstraction

Domain-Specific Programming

The **Scheme** approach:

- Provide expressive constructs
- Enable syntactic abstraction
- Make syntactic abstraction *easy*

Domain-Specific Programming

The **Racket** approach:

- Provide expressive constructs
- Enable syntactic abstraction
- Make syntactic abstraction *easy*
- Smooth the path from syntactic abstraction to language construction

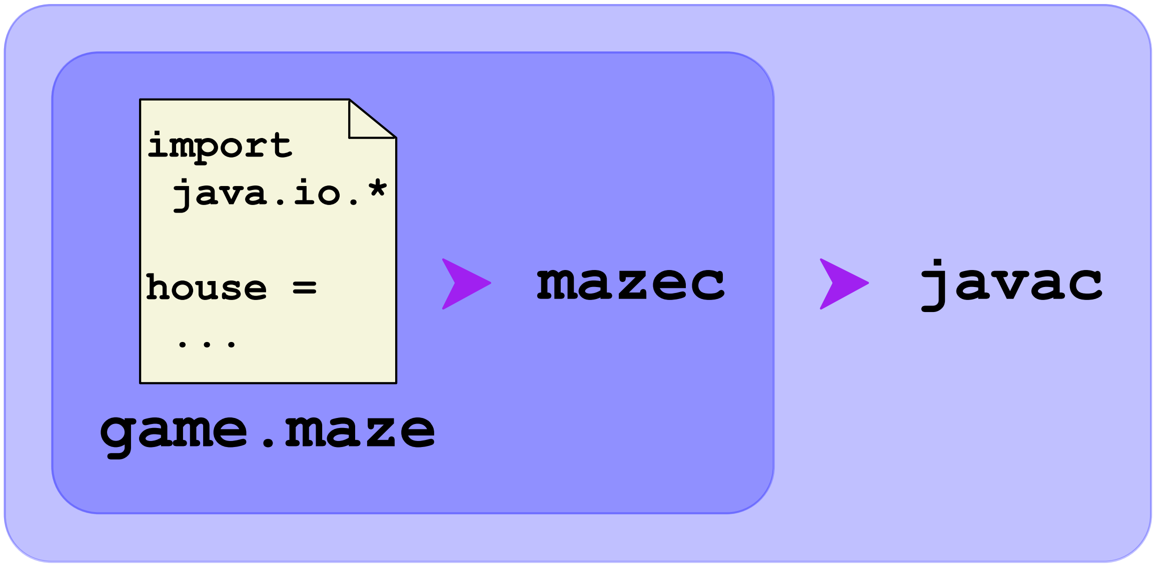
Implementing a Text Adventure Game

```
(define-verbs all-verbs
  [north (n) "go north"]
  [get _ (grab take) "take"]
  ....)

(define-actions everywhere-actions
  ([quit (printf "Bye!\n") (exit)]
   [look (show-current-place)]
   ....))

(define-thing cactus
  [get "Ouch!"])
....

(define-place desert
  "You're in a desert. There is nothing for miles around."
  (cactus key)
  ([north start]
   [south desert] ....))
....
```

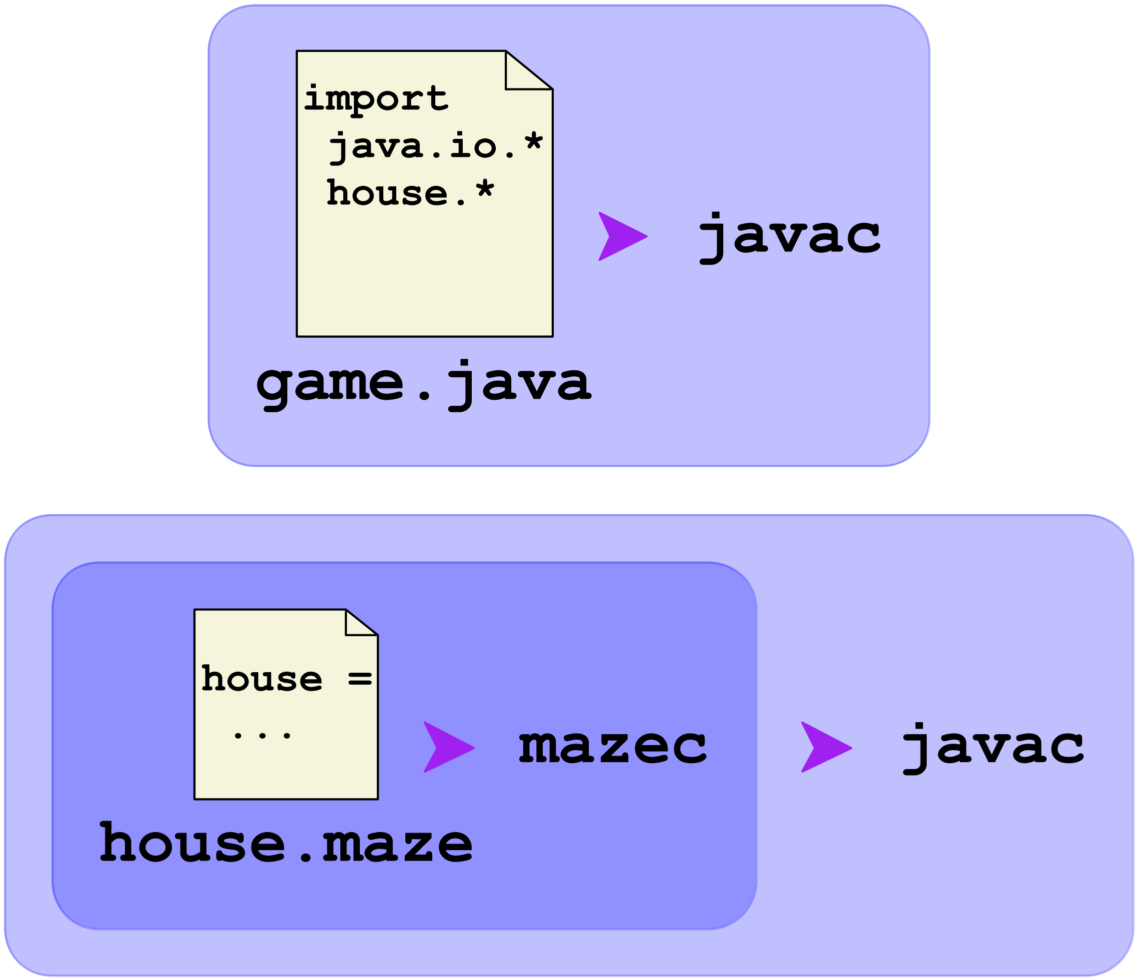
Implementing a Text Adventure Game

```
===VERBS===
north, n
"go north"
....
===EVERYWHERE===
quit
(begin (printf "Bye!\n") (exit))
....
===THINGS===
---cactus---
get
"Ouch!"
....
===PLACES===
---desert---
"You're in a desert. There is nothing for miles around."
[cactus, key]
north start
south desert
....
```

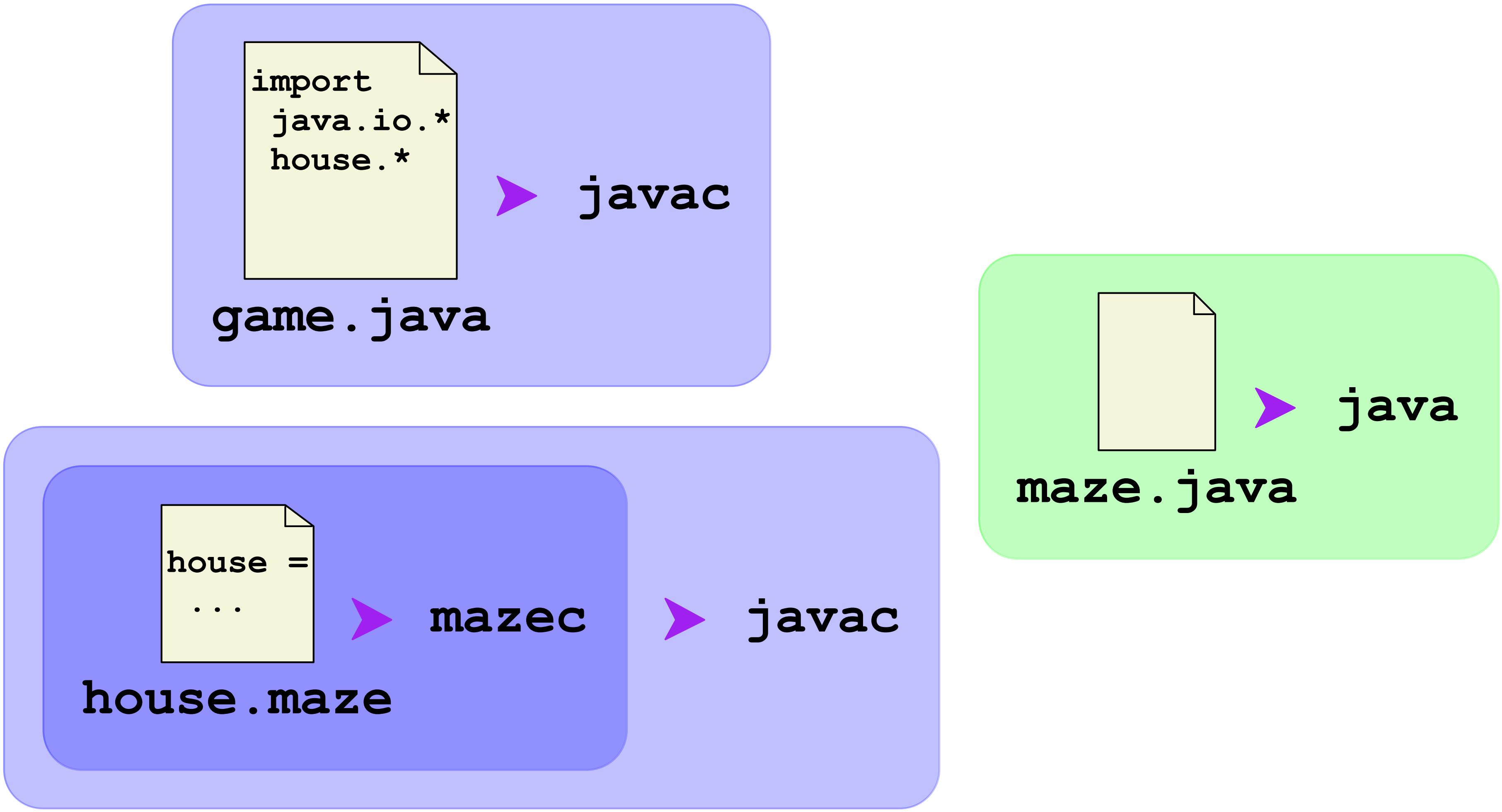
Preprocessors vs. Macros



Preprocessors vs. Macros



Preprocessors vs. Macros



Preprocessors vs. Macros

```
(require "io.rkt")

(define-syntax define-place
  ...)

(define-place house ....)
```

game.rkt

➤ **raco make**

Preprocessors vs. Macros

```
(require "io.rkt"
         "maze.rkt")

(define-place house ....)
```

game.rkt

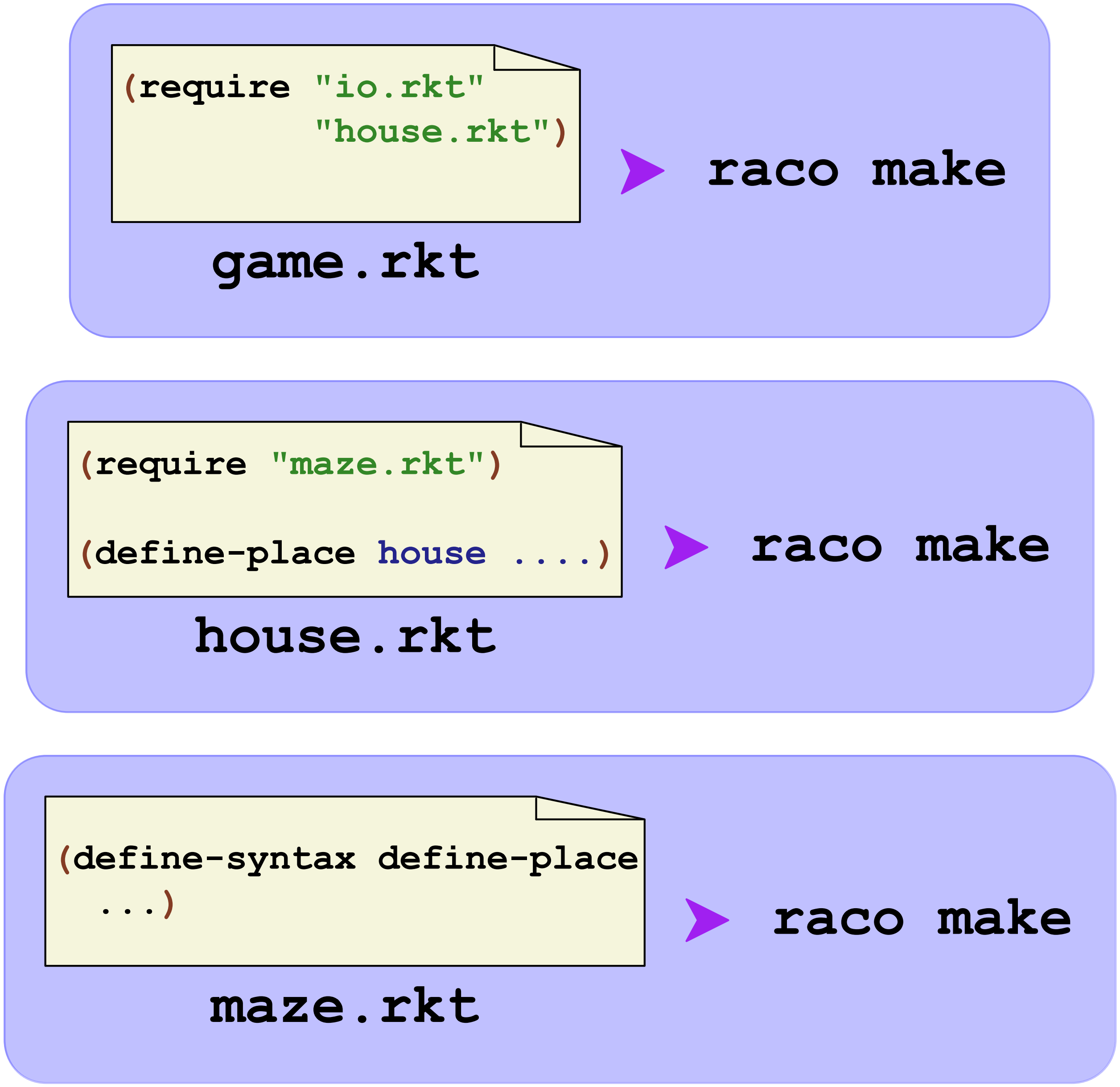
➤ raco make

```
(define-syntax define-place
  ...)
```

maze.rkt

➤ raco make

Preprocessors vs. Macros



[insert programming here]

game implementation overview

Simple Pattern-Based Macros

```
(define-syntax-rule
  )
```

- **`define-syntax-rule`** indicates a simple-pattern macro definition

Simple Pattern-Based Macros

```
(define-syntax-rule pattern
  template)
```

- A ***pattern*** to match
- Produce result from ***template***

Simple Pattern-Based Macros

```
(define-syntax-rule (swap a b)
        )
```

- Pattern for this macro: `(swap a b)`
- Each identifier matches anything in use

`(swap x y)` ⇒ `a` is `x`
`b` is `y`

`(swap 9 (+ 1 7))` ⇒ `a` is `9`
`b` is `(+ 1 7)`

Simple Pattern-Based Macros

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

Bindings substituted into template to generate the result

```
(swap x y)          ⇒  (let ([tmp y])
                          (set! y x)
                          (set! x tmp))

(swap 9 (+ 1 7))    ⇒  (let ([tmp (+ 1 7)])
                          (set! (+ 1 7) 9)
                          (set! 9 tmp))
```

Pattern-Based Macros

```
(define-syntax flip
          )
```

```
(let ([x 0]
      [y 1])
  (flip x y))
```

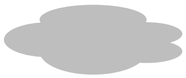
```
(let ([xb (box 0)]
      [yb (box 1)])
  (f xb yb))

(define (f xb yb)
  (flip in xb yb))
```

Pattern-Based Macros

```
(define-syntax flip
       )
```

- **`define-syntax`** indicates a macro definition

Pattern-Based Macros

```
(define-syntax flip
  (syntax-rules (in)
```

```
    ))
```

- **`syntax-rules`** means a pattern-matching macro
- **`(in)`** means that **`in`** is literal in patterns

Pattern-Based Macros

```
(define-syntax flip
  (syntax-rules (in)
    [pattern template]
    ...
    [pattern template]))
```

- Any number of ***pattern***s to match
- Produce result from ***template*** of first match

Pattern-Based Macros

```
(define-syntax flip
  (syntax-rules (in)
    [(flip in a b)  ....]
    [(flip a b)         ]))
```

Two patterns for this macro

- `(flip in xb yb)` matches first pattern
- `(flip x y)` falls through to second pattern

Pattern-Based Macros

```
(define-syntax flip
  (syntax-rules (in)
    [(flip in a b)  (let ([tmp (unbox b)])
                      (set-box! b (unbox a))
                      (set-box! a tmp))]
    [(flip a b)  (swap a b)]))
```

```
(flip in xb yb)  ⇒  (let ([tmp (unbox yb)])
                      (set-box! yb (unbox xb))
                      (set-box! xb tmp))
```

```
(flip x y)       ⇒  (swap x y)
```

Matching Sequences

Some macros need to match sequences

```
(rotate x y)

(rotate red green blue)

(rotate front-left
        rear-right
        front-right
        rear-left)
```

Matching Sequences

```
(define-syntax rotate
  (syntax-rules ()
    [(rotate a) (void)]
    [(rotate a b c ...) (begin
                          (swap a b)
                          (rotate b c ...))]))
```

- `...` in a pattern: multiple of previous sub-pattern

  `(rotate x y z w)` ⇒ `c` is `z w`

- `...` in a template: multiple instances of previous sub-template

```
(rotate x y z w)  ⇒  (begin
                        (swap x y)
                        (rotate y z w))
```

Matching Sequences

```
(define-syntax rotate
  (syntax-rules ()
    [(rotate a c ...)
     (shift-to (c ... a) (a c ...))]))

(define-syntax shift-to
  (syntax-rules ()
    [(shift-to (from0 from ...) (to0 to ...))
     (let ([tmp from0])
       (set! to from) ...
       (set! to0 tmp))    ]))
```

- ... maps over same-sized sequences
- ... duplicates constants paired with sequences

[insert programming here]

complete game implementation

Macro Scope

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

What if we **swap** a variable named **tmp**?

```
(let ([tmp 5]
      [other 6])
  (swap tmp other))
```

$\overset{?}{\Rightarrow}$

```
(let ([tmp 5]
      [other 6])
  (let ([tmp other])
    (set! other tmp)
    (set! tmp tmp)))
```

Macro Scope

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

What if we **swap** a variable named **tmp**?

```
(let ([tmp 5]
      [other 6])
  (swap tmp other))
```

$\stackrel{?}{\Rightarrow}$

```
(let ([tmp 5]
      [other 6])
  (let ([tmp other])
    (set! other tmp)
    (set! tmp tmp)))
```

This expansion would break scope

Macro Scope

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

What if we `swap` a variable named `tmp`?

```
(let ([tmp 5]
      [other 6])
  (swap tmp other))
```

$\Rightarrow$

```
(let ([tmp 5]
      [other 6])
  (let ([tmp₁ other])
    (set! other tmp)
    (set! tmp tmp₁)))
```

Rename the introduced binding

Macro Scope: Local Bindings

Macro scope means that local macros work, too:

```
(define (f x)
  (define-syntax swap-with-arg
    (syntax-rules ()
      [(swap-with-arg y) (swap x y)]))

  (let ([z 12]
        [x 10])
    ; Swaps z with original x:
    (swap-with-arg z))
                                          )
```

How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

Seems obvious that `tmp` can be renamed...

How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let-one [tmp b]
    (set! b a)
    (set! a tmp)))
```

How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let-one [tmp b]
    (set! b a)
    (set! a tmp)))
```

Can rename **tmp**:

```
(define-syntax-rule (let-one (x v) body)
  (let ([x v]) body))
```

How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let-one [tmp b]
    (set! b a)
    (set! a tmp)))
```

Cannot rename **tmp**:

```
(define-syntax (let-one (x v) body)
  (list 'x v body))
```

How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let-one [tmp b]
    (set! b a)
    (set! a tmp)))
```

Cannot rename **tmp**:

```
(define-syntax (let-one (x v) body)
  (list 'x v body))
```

Track identifier introductions, then rename only as binding forms are discovered

How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

Tracking avoids capture by introduced variables

```
(let ([tmp 5]                ⇒  (let ([tmp 5]
      [other 6])                       [other 6])
  (swap tmp other))               (let^1 ([tmp^1 other])
                                     (set!^1 other tmp)
                                     (set!^1 tmp tmp^1)))
```

^{1} means introduced by expansion

`tmp`^{1} does not capture `tmp`

How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

Tracking also avoids capture *of* introduced variables

```
(let ([set! 5]
      [let 6])
  (swap set! let))
```

$\Rightarrow$

```
(let ([set! 5]
      [let 6])
  (let^1 ([tmp^1 let])
    (set!^1 let set!)
    (set!^1 set! tmp^1)))
```

`set!` does not capture **`set!`**^{1}

`let` does not capture **`let`**^{1}

[insert programming here]

modular game implementation

Implicit Syntactic Forms

To change functions:

```
(define-syntax-rule (lambda ....) ....)
```

To change function calls?

```
(define-syntax-rule (#%app ....) ....)
```

($expr_1$ ... $expr_N$)

is implicitly

(#%app $expr_1$ ... $expr_N$)

Implicit Syntactic Forms

```
#lang s-exp path
form₁
...
formN
```

is implicitly

```
#lang s-exp path
(#%module-begin
 form₁
 ...
 formN)
```

[insert programming here]

game module language

Transformer Definitions

In general, **`define-syntax`** binds a transformer procedure:

```
(define-syntax swap
  (syntax-rules .....))
```

$\Longrightarrow$

```
(define-syntax swap
  (lambda (stx)
```

use syntax-object primitives to match **`stx`** and generate result

```
  ))
```

Representing Code

#'lambda

syntax-object
datum
srcloc
context

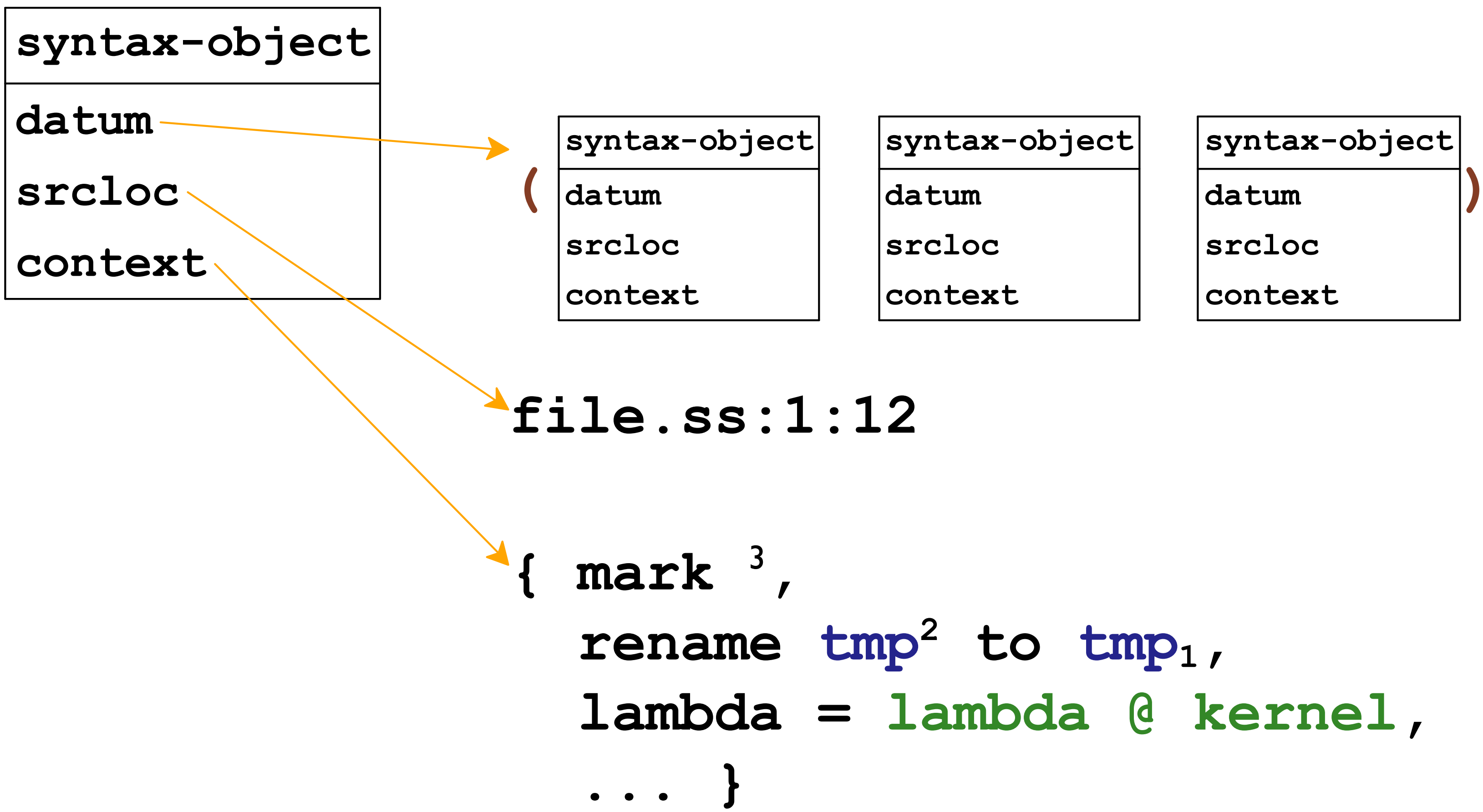
'lambda

file.ss:1:13

{ mark ^{3},
rename $\texttt{tmp}^{2}$ to $\texttt{tmp}_{1}$,
lambda = lambda @ kernel,
... }

Representing Code

```
#'(lambda (x) x)
```



Expressions, Bindings, and Phases

```
(define-syntax three
  (lambda (stx) #'3))

(+ 1 (three))
```

Expressions, Bindings, and Phases

```
(define-syntax three
  (lambda (stx) #'3))

(+ 1 (three))
```

Expressions, Bindings, and Phases

```
(define-syntax three
  (lambda (stx) #'3))

(+ 1 (three))
```

Expressions, Bindings, and Phases

```
(require (for-syntax "roman-numerals.rkt"))

(define-syntax three
  (lambda (stx)
    #`(+ 1 #,(roman->number "II"))))

(+ 1 (three))
```

Expressions, Bindings, and Phases

```
(require (for-syntax "roman-numerals.rkt"))

(define-syntax three
  (lambda (stx)
    #`(+ 1 #,(roman->number "II"))))

(+ 1 (three))
```

Expressions, Bindings, and Phases

```
(begin-for-syntax
 (define (roman->number str) ....))

(define-syntax three
  (lambda (stx)
    #`(+ 1 #,(roman->number "II"))))

(+ 1 (three))
```

Expressions, Bindings, and Phases

```
(begin-for-syntax
 (define (roman->number str) ....))

(define-syntax three
  (lambda (stx)
    #`(+ 1 #,(roman->number "II"))))

(+ 1 (three))
```

Matching Syntax and Having It, Too

`syntax-case` and **`#'`** combine patterns and computation

```
(syntax-case stx-expr ()
  [pattern result-expr]
  ...
  [pattern result-expr])

#'template
```

Matching Syntax and Having It, Too

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

$\Longrightarrow$

```
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(swap₁ a b) #'(let ([tmp b])
                       (set! b a)
                       (set! a tmp))])))
```

Matching Syntax and Having It, Too

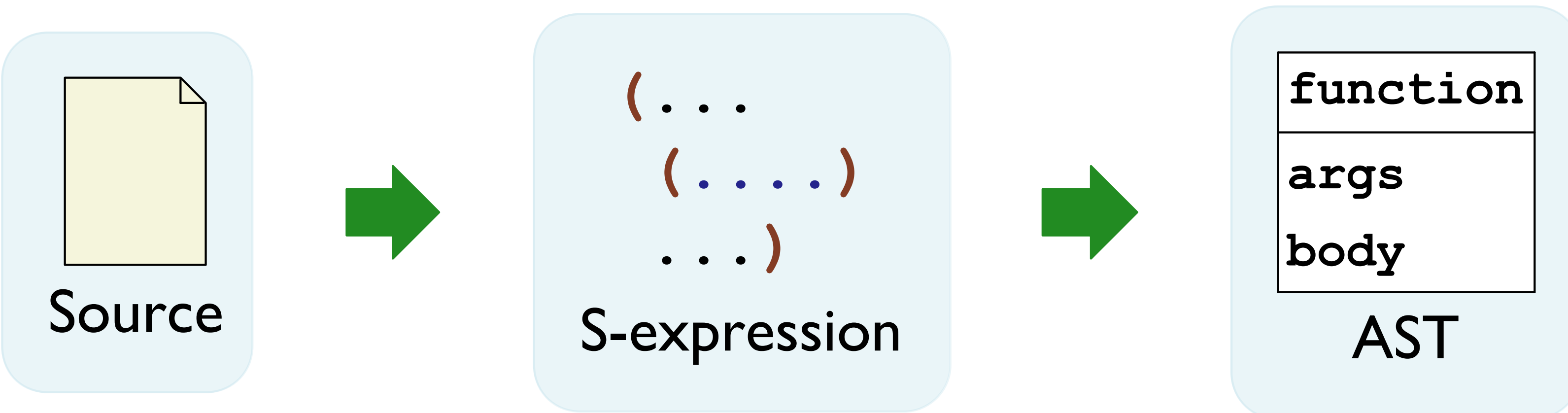
Check for identifiers before expanding:

```
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(swap a b)
       (if (and (identifier? #'a)
                (identifier? #'b))
           #'(let ([tmp b])
               (set! b a)
               (set! a tmp))
           (raise-syntax-error
            'swap "needs identifiers"
            stx))])))
```

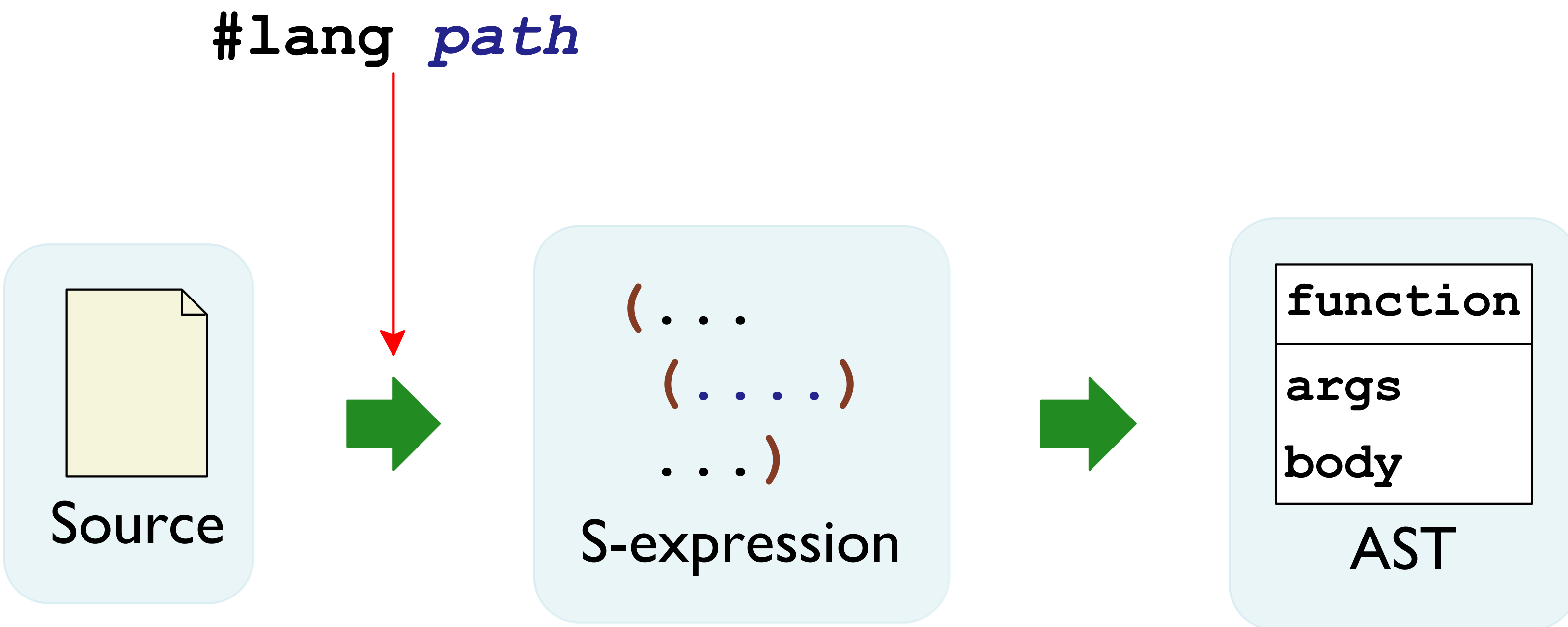
[insert programming here]

“typed” game language

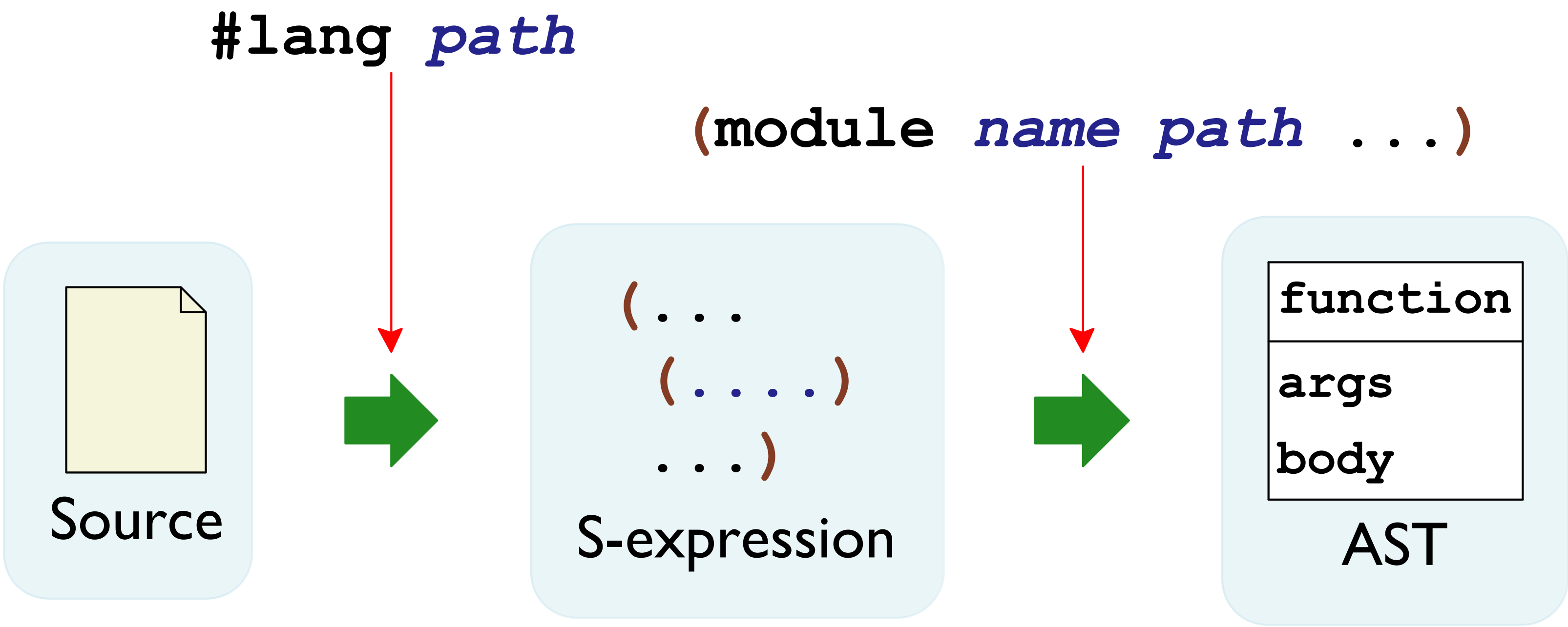
Parsing



Parsing

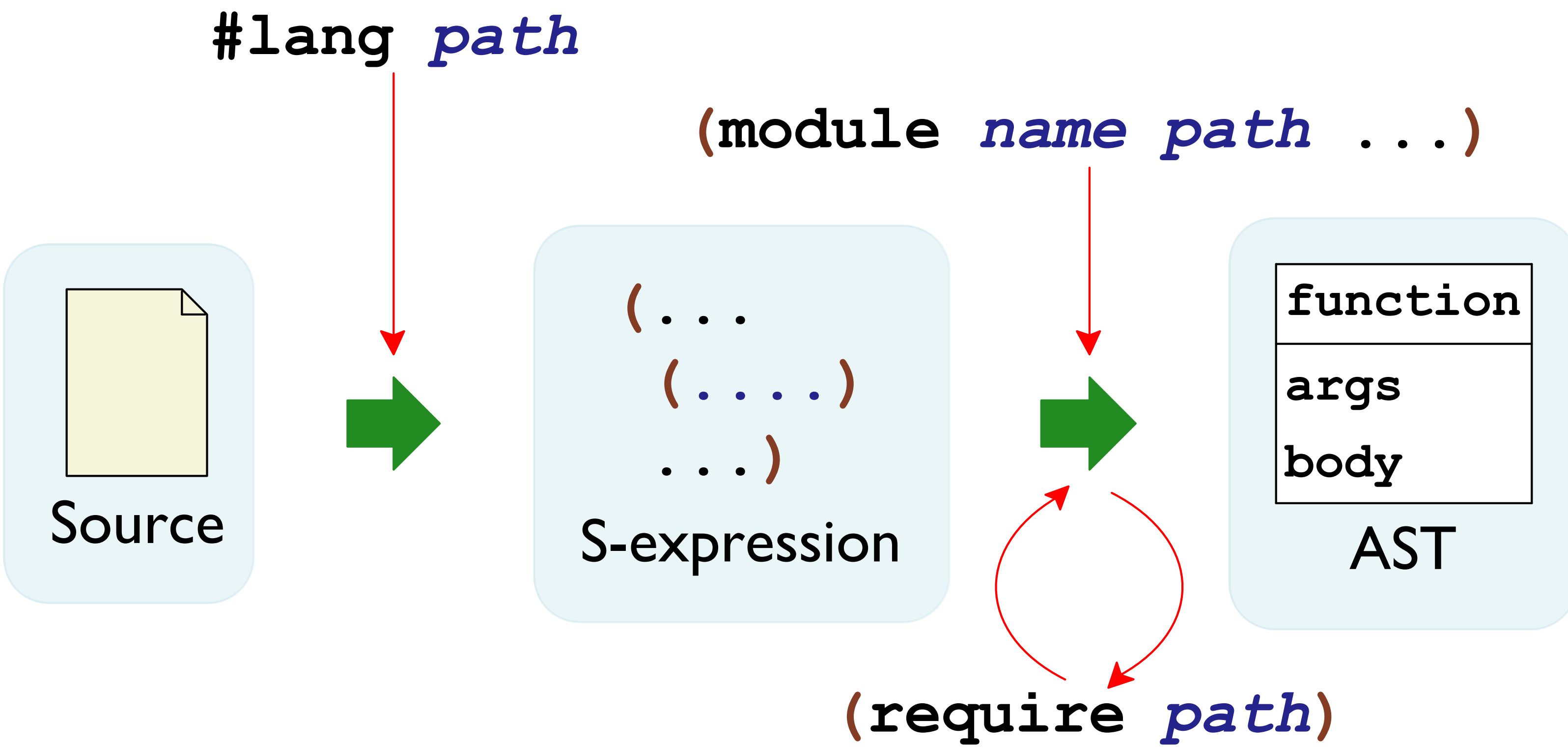


Parsing

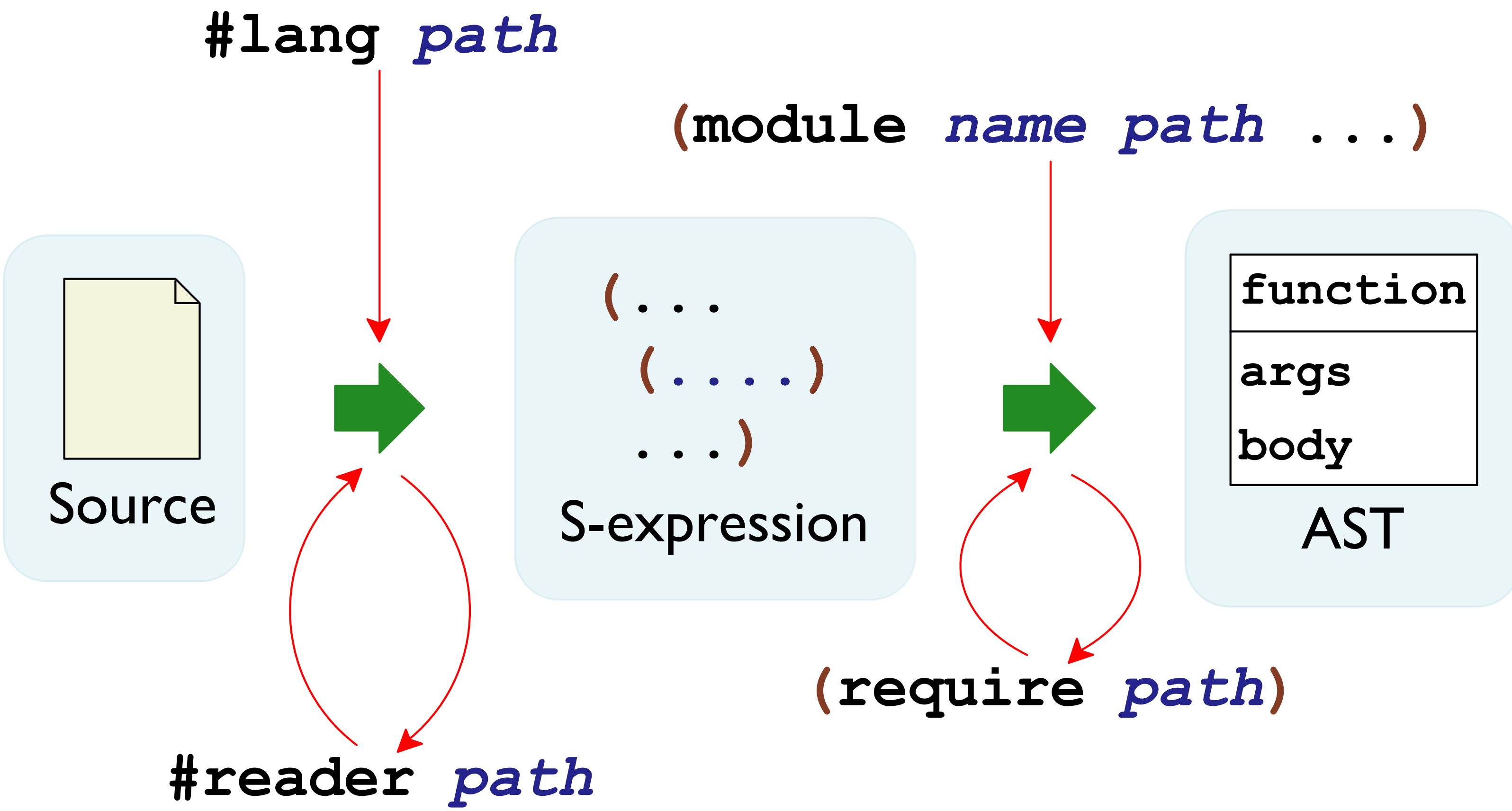


#lang path
(module name path ...)
Source
(...
(....)
...)
S-expression
function
args
body
AST

Parsing


#lang path
(module name path ...)
(...
(....)
...)
Source
S-expression
function
args
body
AST
(require path)

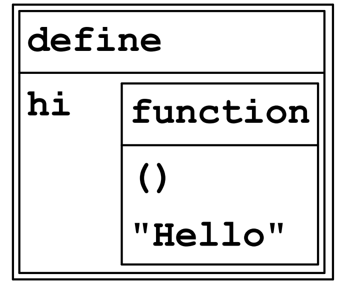
Parsing


#lang path
(module name path ...)
Source
(...
(....)
...)
S-expression
function
args
body
AST
(require path)
#reader path

Parsing

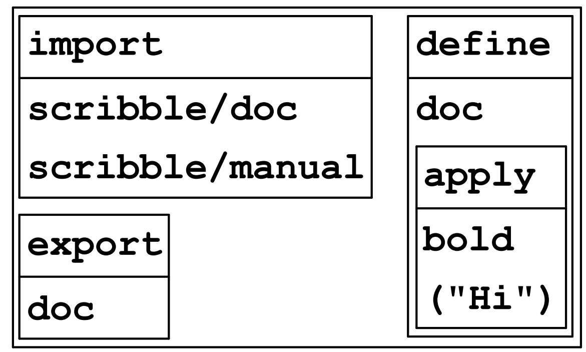
```
#lang scheme
(define (hi)
  "Hello")
```

```
(module m scheme
  (define (hi)
    "Hello"))
```



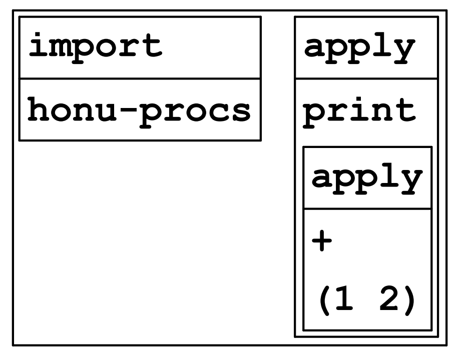

```
#lang scribble/doc
@(require
  scribble/manual)
@bold{Hi}
```

```
(module m doclang
  (require
   scribble/manual)
  (bold "Hi"))
```




```
#lang honu
1+2;
```

```
(module m honu
  1 + 2 |;|)
```



Parsing

Parsing

Read layer provides absolute control

```
(+ 1 2)     @bold{Hi}     1+2
```

Parsing

Expand layer can delay “inside” until after “outside”

```
(define-place start ....
  ([north house-front]
   [south desert]))

(define-place house-front ....
  ([in room]
   [south start]))
```

```
int is_odd(int x) {
   ... is_even(x-1);
}

int is_even(int x) {
   ... is_odd(x-1);
}
```

[insert programming here]

non-S-expression game language

Environment Support

Support at S-expression level is free

- Error source locations
- Check Syntax

Source-editing support requires more

- On-the-fly coloring

[insert demo here]

DrRacket editor support

Languages in Racket