Part I

Interpreters

```
{{lambda {mkrec}
   {{lambda {fib}
      {fib 4}}
    {mkrec
     {lambda {fib}
       {lambda {n}
         {if {zero? n}
             1
             {if {zero? {+ n -1}}
                 1
                 {+ {fib {+ n -1}}
                    {fib {+ n -2}}}}}}}}}}}
 {lambda {body-proc}
   {{lambda {fX}
      {fX fX}}
    {lambda {fX}
      {body-proc {lambda {x} {{fX fX} x}}}}}}}
```

Language Variants

```
{{lambda {mkrec}
   {{lambda {fib}
      {fib 4}}
    {mkrec
     {lambda {fib}
       {lambda {n}
         {if0 n
              1
              {if0 {+ n -1}
                   1
                   {+ {fib {+ n -1}}
                      {fib {+ n -2}}}}}}}}}}
 {lambda {body-proc}
   {{lambda {fX}
      {fX fX}}
    {lambda {fX}
      {body-proc {lambda {x} {{fX fX} x}}}}}}}
```

Language Extension

```
(define-syntax-rule {if0 tst thn els}
  (if (zero? tst)
      thn
      els))
```

- `if0` is a ***pattern-based macro***

- Macros are a form of ***language extension***

- ***Domain-specific languages*** can be extensions

- ***Compile-time reflection ≈ macros***

... in the Lisp/Scheme/Racket sense

Racket macros can express many languages and extensions

Part 2

Simple Pattern-Based Macros

```
(define-syntax-rule ☁
  ☁)
```

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)
          )
```

Simple Pattern-Based Macros

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

- Pattern for this macro: `(swap a b)`

Simple Pattern-Based Macros

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

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

`(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)))
```

Simple Pattern-Based Macros

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

Matches 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))
```

Part 3

General Pattern-Based Macros

```
(define-syntax shift
          )
```

```
(let ([x 0]                    (let ([x 0]
      [y 1]                          [y 1]
      [z 2])                         [z 2])
  (shift x y z))                 (shift back x y z))
```

General Pattern-Based Macros

```
(define-syntax shift
          )
```

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

General Pattern-Based Macros

```
(define-syntax shift
  (syntax-rules (back)
                 ))
```

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

General Pattern-Based Macros

```
(define-syntax shift
  (syntax-rules (back)
    [pattern template]
    ...
    [pattern template]))
```

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

General Pattern-Based Macros

```
(define-syntax shift
  (syntax-rules (back)
    [(shift a b c)         ]
    [(shift back a b c)         ]))
```

Two patterns for this macro

- `(shift x y z)` matches first pattern
- `(shift back x y z)` matches second pattern
- `(shift rev x y z)` does not match

General Pattern-Based Macros

```
(define-syntax shift
  (syntax-rules (back)
    [(shift a b c) (begin
                     (swap a b)
                     (swap b c))]
    [(shift back a b c) (begin
                          (swap c b)
                          (swap b a))]))
```

```
(shift x y z)        ⇒  (begin
                          (swap x y)
                          (swap y z))

(shift back x y z)   ⇒  (begin
                          (swap z y)
                          (swap y x))
```

Part 4

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: 0 or more of previous sub-pattern

  `(rotate x y z w)` ⇒ `c` is `z w`

- `...` in a template: 0 or more 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 (to0 to ...) (from0 from ...))
     (let ([tmp from0])
       (set! to from) ...
       (set! to0 tmp))   ]))
```

- . . . maps over same-sized sequences

- . . . duplicates constants paired with sequences

Part 5

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))
```

$\overset{?}{\Rightarrow}$

```
(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]                ⇒  (let ([tmp 5]
      [other 6])                      [other 6])
  (swap tmp other))               (let ([tmp_1 other])
                                    (set! other tmp)
                                    (set! tmp tmp_1)))
```

Hygienic macros rename the introduced binding

Hygienic Macros: Local Bindings

Hygiene 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 Hygiene Works

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

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

How Hygiene Works

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

How Hygiene 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 Hygiene Works

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

Cannot rename `tmp`:

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

How Hygiene Works

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

Cannot rename `tmp`:

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

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

How Hygiene 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 Hygiene 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 ([set! 5]
      [let 6])                    [let 6])
  (swap set! let))           (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}

Part 6

Identifier Macros

The **swap** and **rotate** names work only in an “application” position

```
(swap x y)  ⇒  (let ([tmp y]) ☁)
(+ swap 2)  ⇒  syntax error
```

An ***identifier macro*** works in any expression position

```
clock           ⇒  (get-clock)
(+ clock 10)    ⇒  (+ (get-clock) 10)
(clock 5)       ⇒  ((get-clock) 5)
```

...or as a **set!** target

```
(set! clock 10)  ⇒  (set-clock! 10)
```

Identifier Macros

Using **syntax-id-rules**:

```
(define-syntax clock
  (syntax-id-rules (set!)
    [(set! clock e) (put-clock! e)]
    [(clock a ...) ((get-clock) a ...)]
    [clock (get-clock)]))
```

- **set!** is designated as a literal
- **syntax-rules** is a special case of **syntax-id-rules** with errors in the first and third cases

Part 7

Macro-Generating Macros

If we have many identifiers like **clock**...

```
(define-syntax define-get/put-id
  (syntax-rules ()
    [(define-get/put-id id get put!)
     (define-syntax id
       (syntax-id-rules (set!)
        [(set! id e) (put! e)]
        [(id a (... ...)) ((get) a (... ...))]
        [id (get)]))]))

(define-get/put-id clock get-clock put-clock!)
```

where `(... ...)` in a template gets replaced by `...`

Part 8

Extended Example

Let’s add call-by-reference definitions to Racket

```
(define-cbr (f a b)
  (swap a b))

(let ([x 1] [y 2])
  (f x y)
  x)
; should produce 2
```

Extended Example

Expansion of first half:

```
(define-cbr (f a b)
  (swap a b))
```

⇒

```
(define (do-f get-a get-b put-a! put-b!)
  (define-get/put-id a get-a put-a!)
  (define-get/put-id b get-b put-b!)
  (swap a b))
```

Extended Example

Expansion of second half:

```
(let ([x 1] [y 2])
  (f x y)
  x)
```

⇒

```
(let ([x 1] [y 2])
  (do-f (lambda () x)
        (lambda () y)
        (lambda (v) (set! x v))
        (lambda (v) (set! y v)))
  x)
```

Call-by-Reference Setup

How the first half triggers the second half:

```
(define-syntax define-cbr
  (syntax-rules ()
    [(_ (id arg ...) body)
     (begin
       (define-for-cbr do-f (arg ...)
         () body)
       (define-syntax id
         (syntax-rules ()
           [(id actual (... ...))
            (do-f (lambda () actual)
                  (... ...)
                  (lambda (v)
                    (set! actual v))
                  (... ...))        ])))]))
```

Call-by-Reference Body

Remaining expansion to define:

```
(define-for-cbr do-f (a b)
  () (swap a b))
```

⇒

```
(define (do-f get-a get-b put-a! put-b!)
  (define-get/put-id a get-a put-a!)
  (define-get/put-id b get-b put-b!)
  (swap a b))
```

How can `define-for-cbr` make `get-` and `put-!` names?

Call-by-Reference Body

A name-generation trick:

```
(define-syntax define-for-cbr
  (syntax-rules ()
    [(define-for-cbr do-f (id0 id ...)
       (gens ...) body)
     (define-for-cbr do-f (id ...)
       (gens ... (id0 get put)) body)]
    [(define-for-cbr do-f ()
       ((id get put) ...) body)
     (define (do-f get ... put ...)
       (define-get/put-id id get put) ...
       body)]))
```

Call-by-Reference Body

More accurate description of the expansion:

```
(define-for-cbr do-f (a b)
  () (swap a b))
```

⇒

```
(define (do-f get^1 get^2 put^1 put^2)
  (define-get/put-id a get^1 put^1)
  (define-get/put-id b get^2 put^2)
  (swap a b))
```

Complete Code to Add Call-By-Reference

```
(define-syntax define-cbr
  (syntax-rules ()
    [(_ (id arg ...) body)
     (begin
       (define-for-cbr do-f (arg ...)
         () body)
       (define-syntax id
         (syntax-rules ()
           [(id actual (... ...))
            (do-f (lambda () actual)
                  (... ...)
                  (lambda (v)
                    (set! actual v))
                  (... ...))          ])))]))
```

```
(define-syntax define-for-cbr
  (syntax-rules ()
    [(define-for-cbr do-f (id0 id ...)
       (gens ...) body)
     (define-for-cbr do-f (id ...)
       (gens ... (id0 get put)) body)]
    [(define-for-cbr do-f ()
       ((id get put) ...) body)
     (define (do-f get ... put ...)
       (define-get/put-id id get put) ...
       body)                             ]))
```

```
(define-syntax define-get/put-id
  (syntax-rules ()
    [(define-get/put-id id get put!)
     (define-syntax id
       (syntax-id-rules (set!)
         [(set! id e) (put! e)]
         [(id a (... ...)) ((get) a (... ...))]
         [id (get)]))                         ]))
```

Part 9

Modules

Modules can export and import macros

cbr.rkt

```
(provide define-cbr)

....
```

f.rkt

```
(require "cbr.rkt")

(define-cbr (f x y)
  ....)

(let ([a 0] [b 1])
  (f a b))
```

Modules

Modules can export and import macros

cbr.rkt

```
(provide define-cbr)

....
```

f.rkt

```
(require "cbr.rkt")

(define-cbr (f x y)
  ....)

(provide f)
```

g.rkt

```
(require "f.rkt")

(let ([a 0] [b 1])
  (f a b))
```

Renaming Exports

The **`provide`** form supports renaming

cbr.rkt

```
(provide
 (rename-out
  [define-cbr define]))

....
```

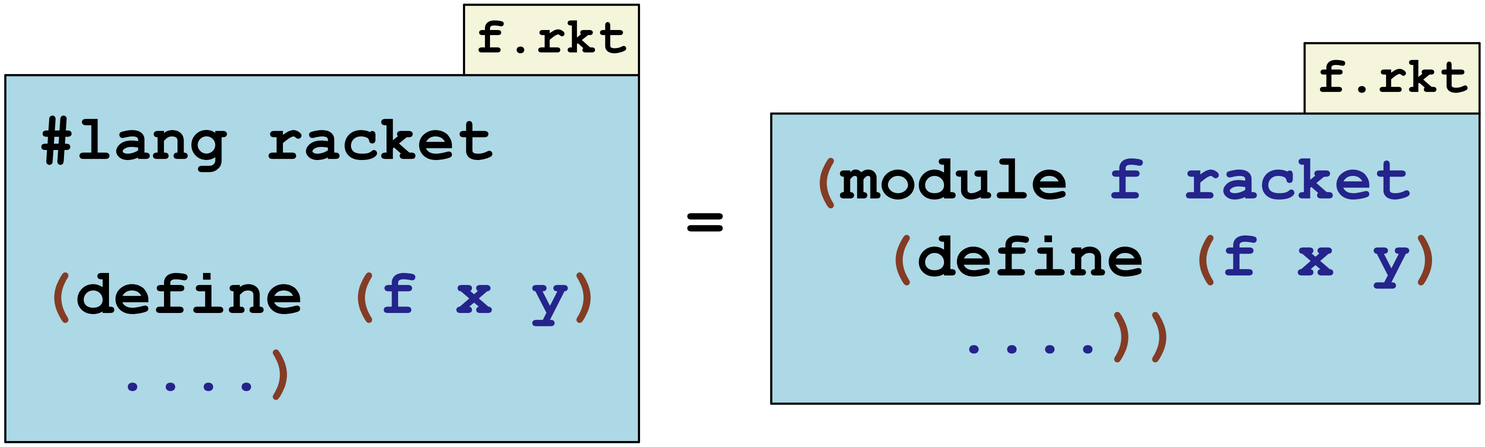
f.rkt

```
(require "cbr.rkt")

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

(let ([a 0] [b 1])
  (f a b))
```

Modules and `#lang`



f.rkt

```
#lang racket

(define (f x y)
  ....)
```

=

f.rkt

```
(module f racket
  (define (f x y)
    ....))
```

Adjusting a Language

The **provide** form has its own sublanguage...

racket-cbr.rkt

```
#lang racket

(provide (except-out (all-from-out racket)
                     define)
         (rename-out [define-cbr define]))

...
```

f.rkt

```
(module f "racket-cbr.rkt"
  (define (f x y)
    ....))
```

Adjusting a Language

The **provide** form has its own sublanguage...

racket-cbr.rkt

```
#lang racket

(provide (except-out (all-from-out racket)
                     define)
         (rename-out [define-cbr define]))

...
```

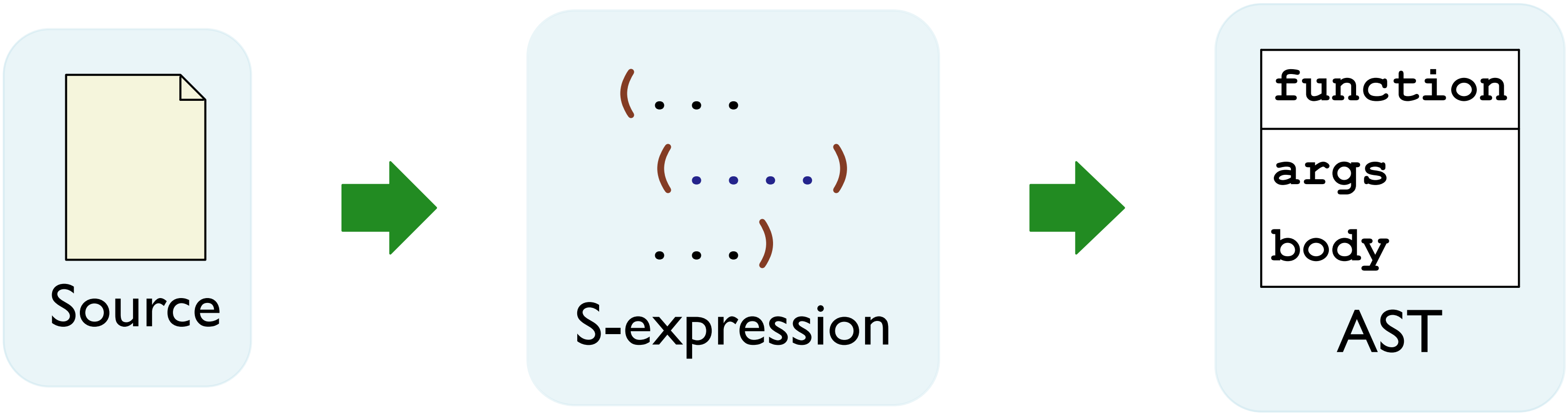
f.rkt

```
#lang s-exp "racket-cbr.rkt"

(define (f x y)
  ....)
```

Part 10

Parsing in Racket



Source

```
(...
  (....)
 ...)
```

S-expression

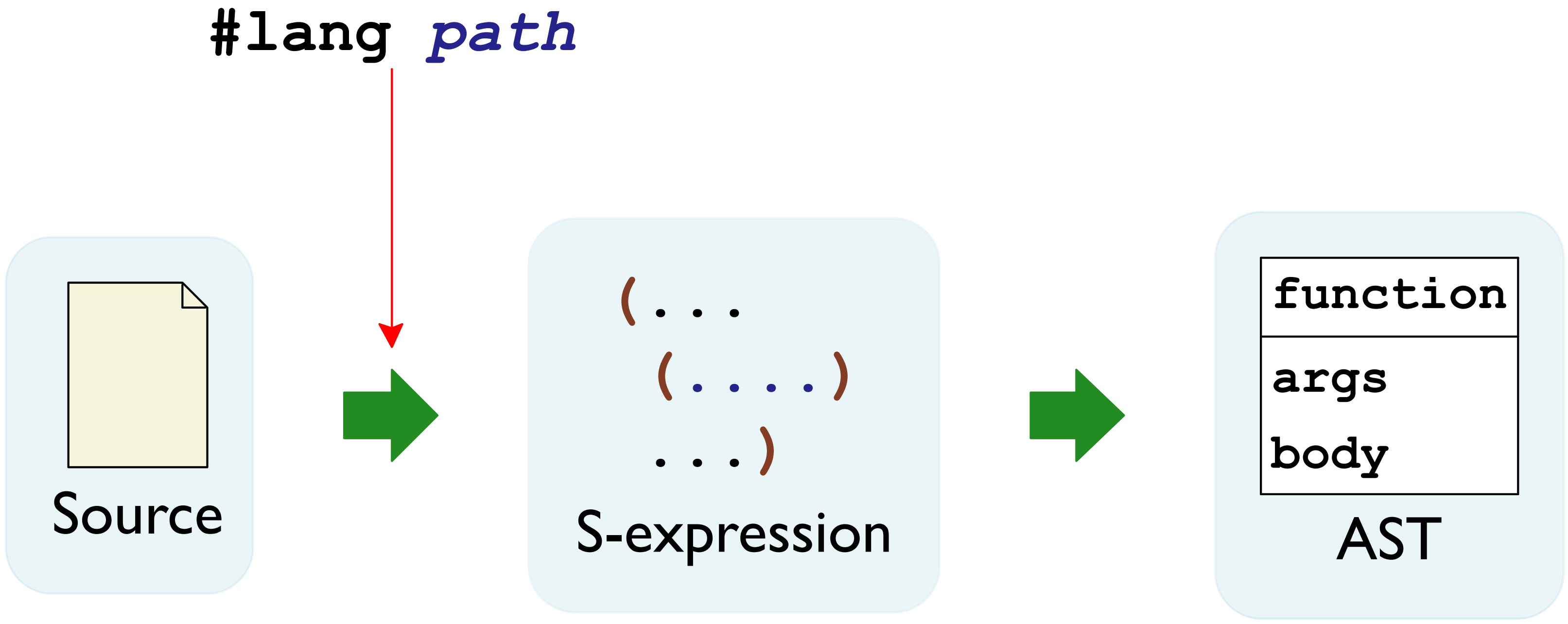
function

args

body

AST

Parsing in Racket



`#lang path`

Source

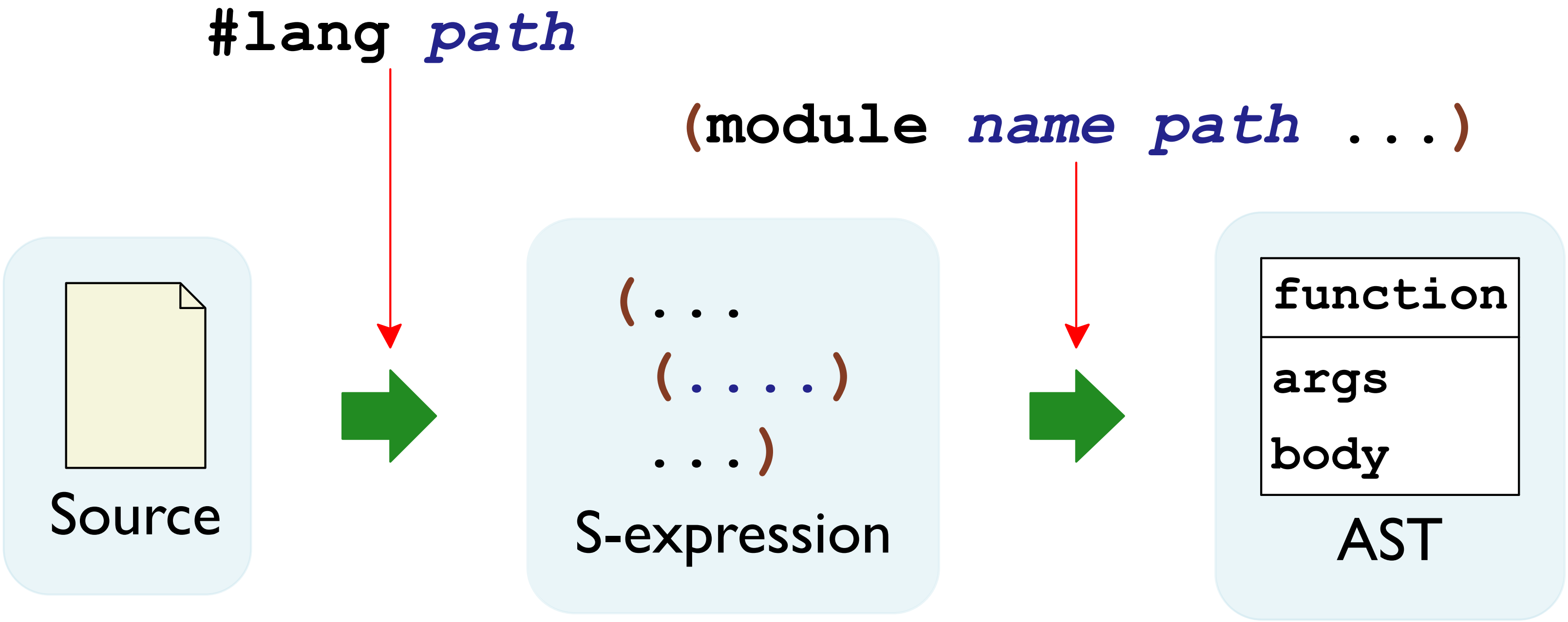
```
(...
 (....)
 ...)
```

S-expression

function
args body

AST

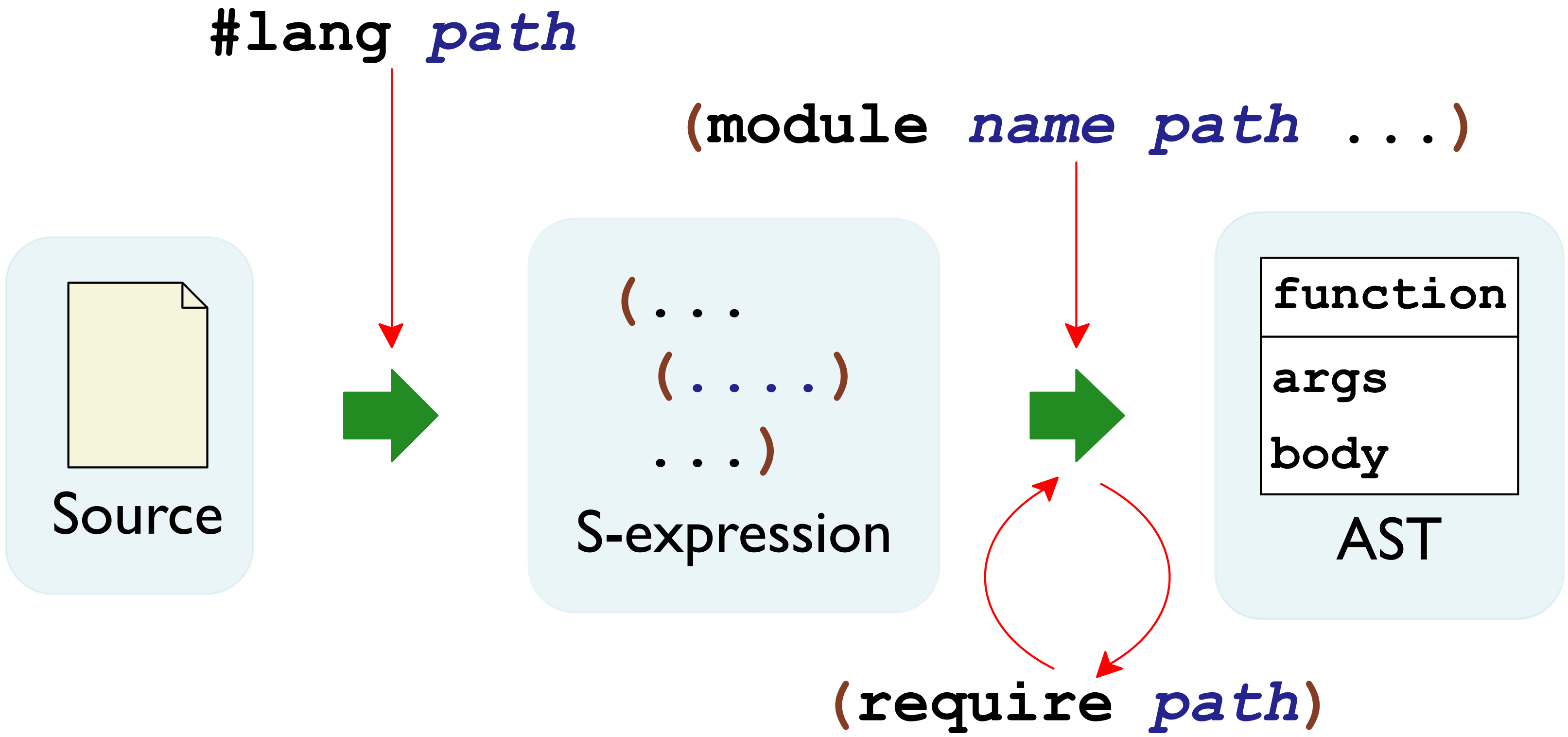
Parsing in Racket



```
#lang path
```

```
(module name path ...)
```

Parsing in Racket



`#lang path`

`(module name path ...)`

Source

```
(...
 (....)
 ...)
```

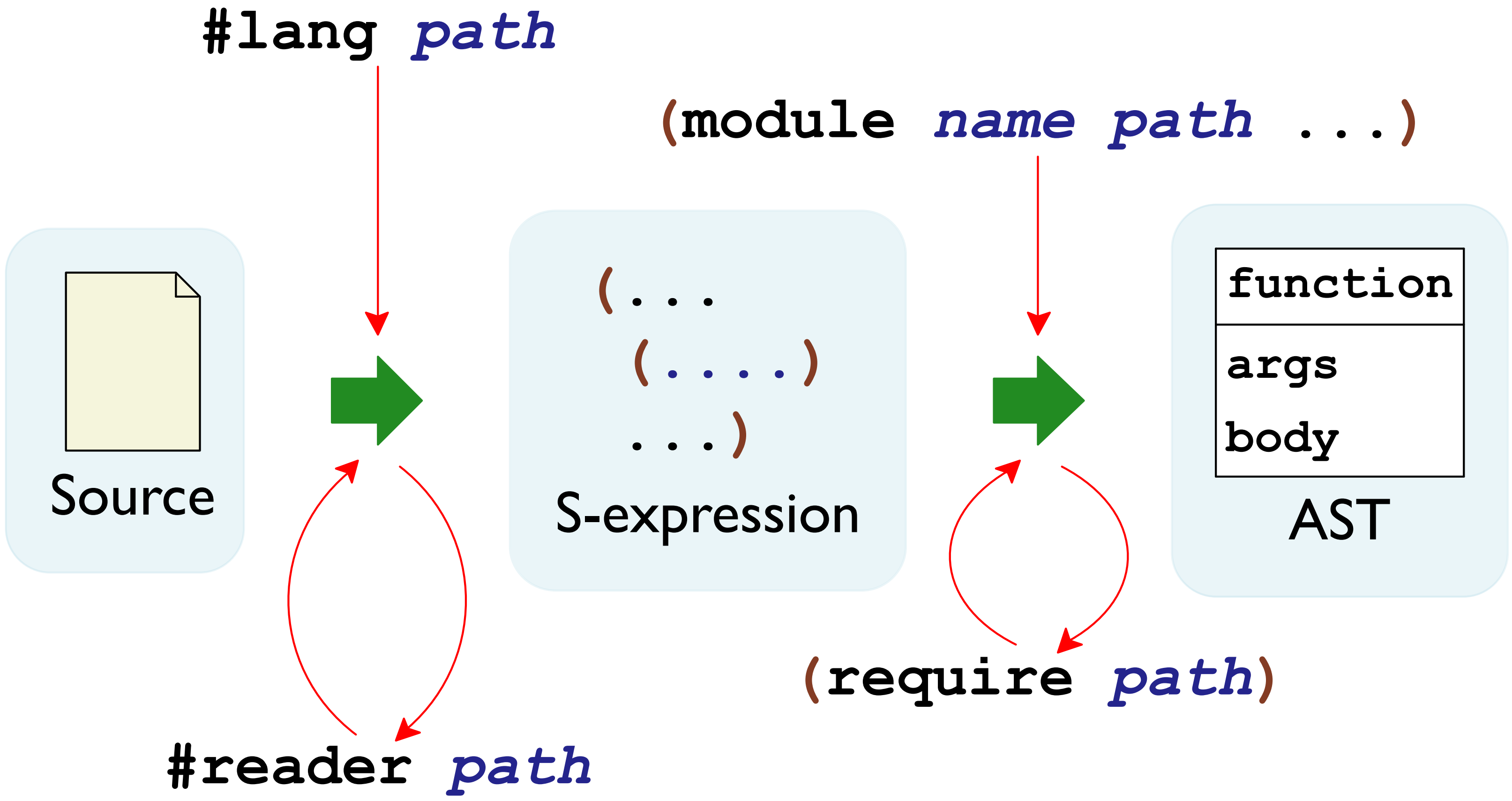
S-expression

```
function
args
body
```

AST

`(require path)`

Parsing in Racket



Parsing in Racket

```
#lang racket
⇒
(module name racket ....)
```

```
#lang s-exp path
⇒
(module name path ....)
```