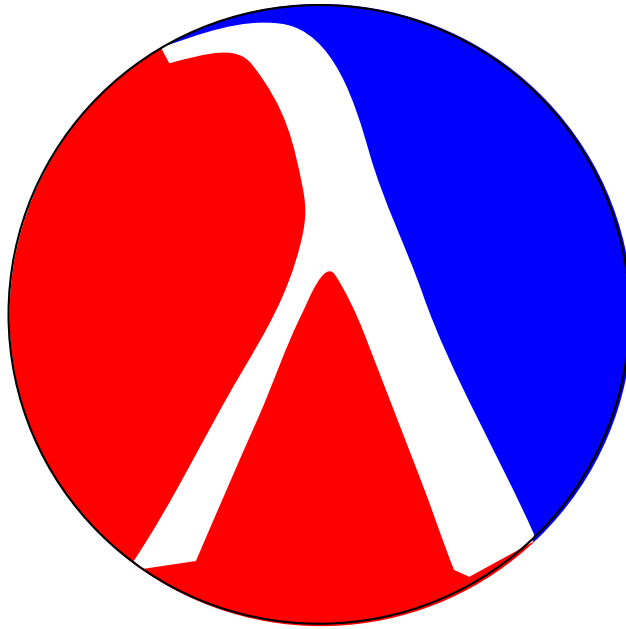


Metaprogramming Time!



Matthew Flatt

PLT and University of Utah

(+ 1 2)

```
(define (+ x y) (* x y))
```

```
(+ 1 2)
```

```
(define-syntax-rule
  (define (id a ...) expr)
  (define-values (id)
    (lambda (a ...) (list a ...)))))
```

```
(define (+ x y) (* x y))
```

```
(+ 1 2)
```

```

(define-syntax define-syntax-rule
  (syntax-rules (...)
    [(define-syntax-rule
      (def-id (id a ...) expr)
      tmpl)
     (define-syntax def-id
      (syntax-rules ()
        [(def-id (id a ...) expr)
         (define-values (id)
          (lambda (a ...) "fish"))]))]))))

```

```

(define-syntax-rule
  (define (id a ...) expr)
  (define-values (id)
    (lambda (a ...) (list a ...))))

```

```

(define (+ x y) (* x y))

```

```

(+ 1 2)

```

#lang htdp/bsl

1 + 2

```
#lang lazy
```

```
(define ones (cons 1 ones))
```

```
(list-ref ones 42)
```

```
#lang plai-typed
```

```
(define-syntax-rule (case expr [(d ...) rhs] ...)  
  (let ([v expr])  
    (cond  
      [(or (equal? v 'd) ...) rhs]  
      ...)))
```

```
(case 'i  
  [(i) 1]  
  [(v) 5]  
  [(x) 10])
```




Part I: Motivation and Approach



A MzScheme Programmer (ca. 2000)...



... Programming in MzScheme



```
(define (parse file)
  ...)
```

```
-----:---- parse.scm
MzScheme version 103
> (parse "test")
```

```
-----:**- *scheme*
```

... in Macro-Extended MzScheme!



```
(load "lex+yacc.scm")
(define (parse file)
  (lex [(+ digit) ...]
    ...))
```

-----:---- **parse.scm**

MzScheme version 103

> (parse "test")

-----:**- ***scheme***

...Trying to Use the Compiler



```
(load "lex+yacc.scm")
(define (parse file)
  (lex [(+ digit) ...]
    ...))
```

-----:---- **parse.scm**

```
$ mzc parse.scm
parse.scm: bad syntax
$ █
```

-----:**- ***shell***

...Accommodating the Compiler



```
(eval-when (compile)
  (load "lex+yacc.scm"))
(define (parse file)
  (lex [(+ digit) ...]
    ----:--- parse.scm
$ mzc parse.scm
$ █

----:**- *shell*
```

...Trying a Complex Library



```
(eval-when ...  
  (load "parselib.scm"))  
(define (parse file)  
  ...)
```

-----:---- **parse.scm**

```
$ mzc parse.scm  
util.scm: bad syntax  
$ █
```

-----:**- ***shell***

... Compiling Remotely



```
(eval-when ...
  (load "gparselib.scm"))
(define (parse file)
  ...)

-----:---- parse.scm
$ mzc parse.scm
Can't open display: :0.0
$ █

-----:**- *shell*
```


REPL-Oriented Program Structure

main.scm

```
(load "grocery.scm")  
(load "top-10.scm")  
.... shop ....  
.... count-down ....
```

grocery.scm

```
(load "list.scm")  
(define (shop l)  
  .... fold ....)
```

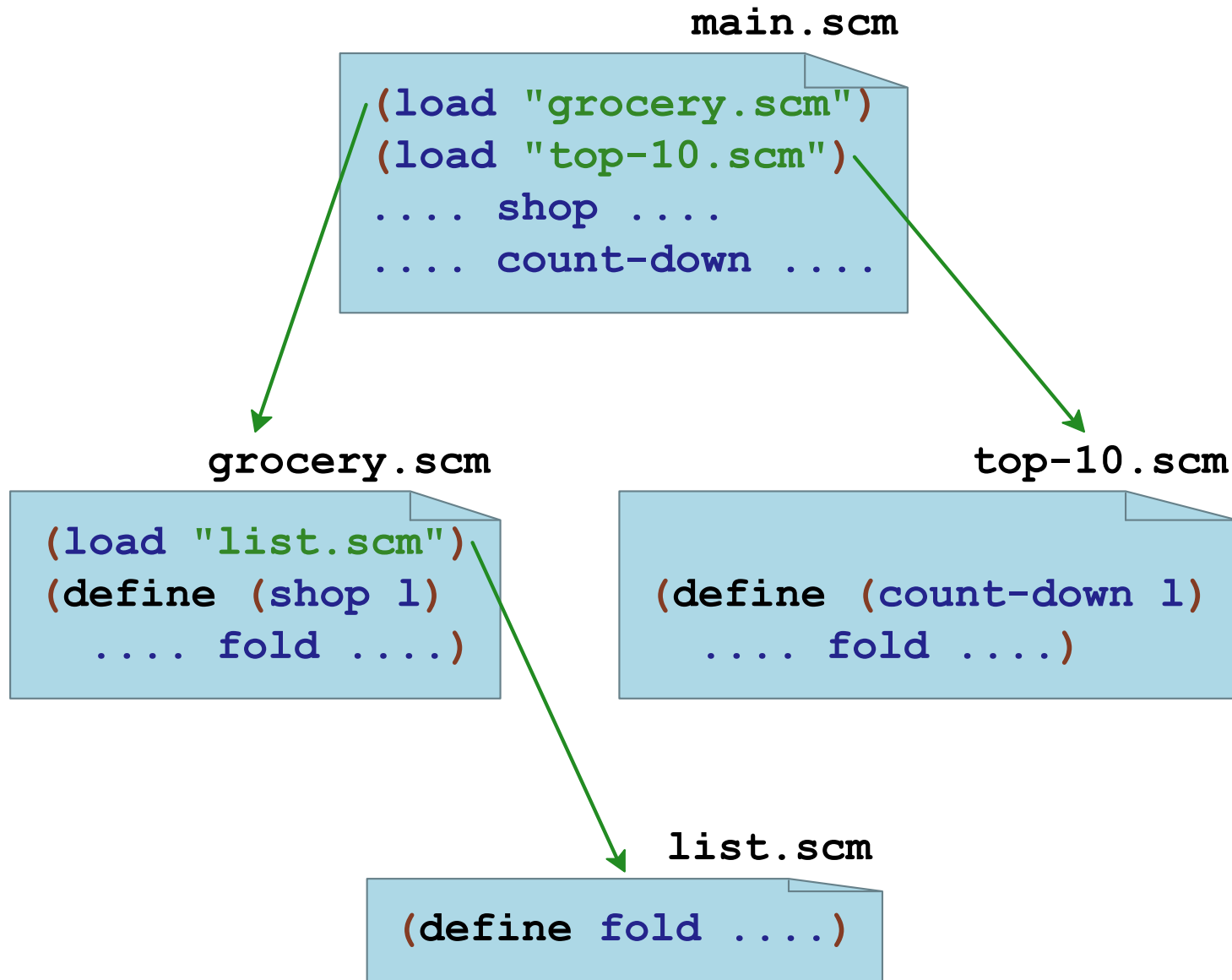
top-10.scm

```
(define (count-down l)  
  .... fold ....)
```

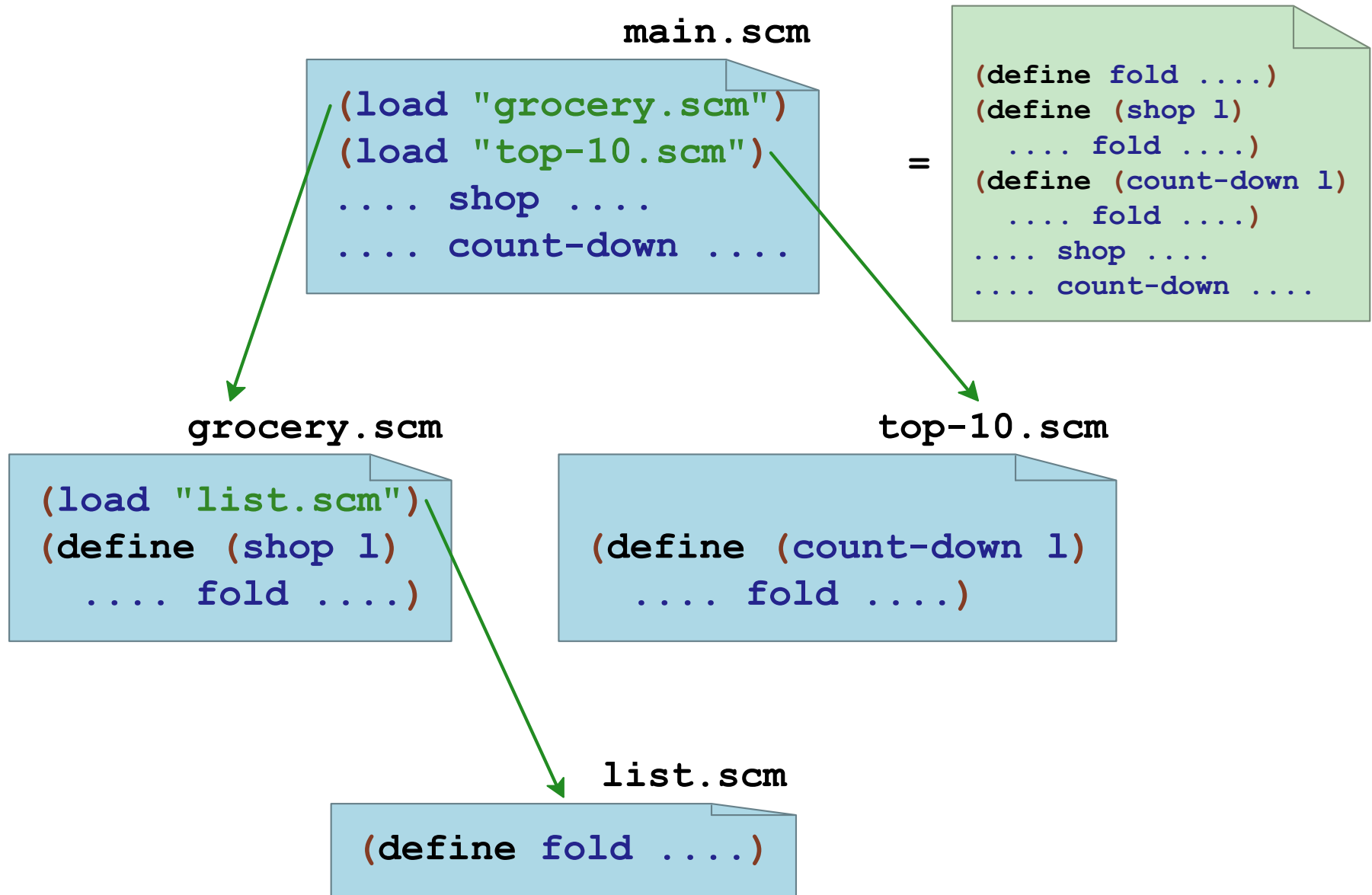
list.scm

```
(define fold ....)
```

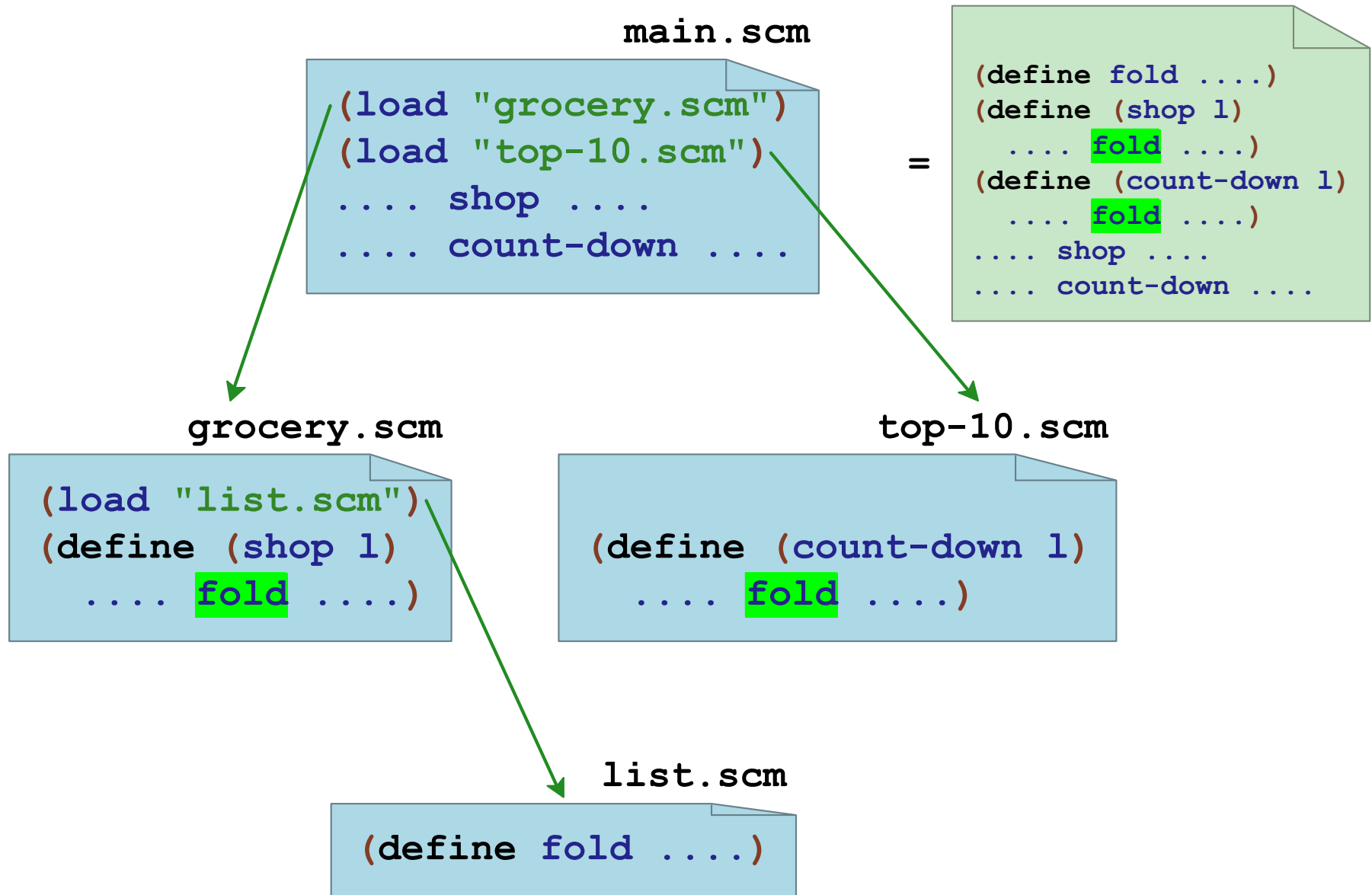
REPL-Oriented Program Structure



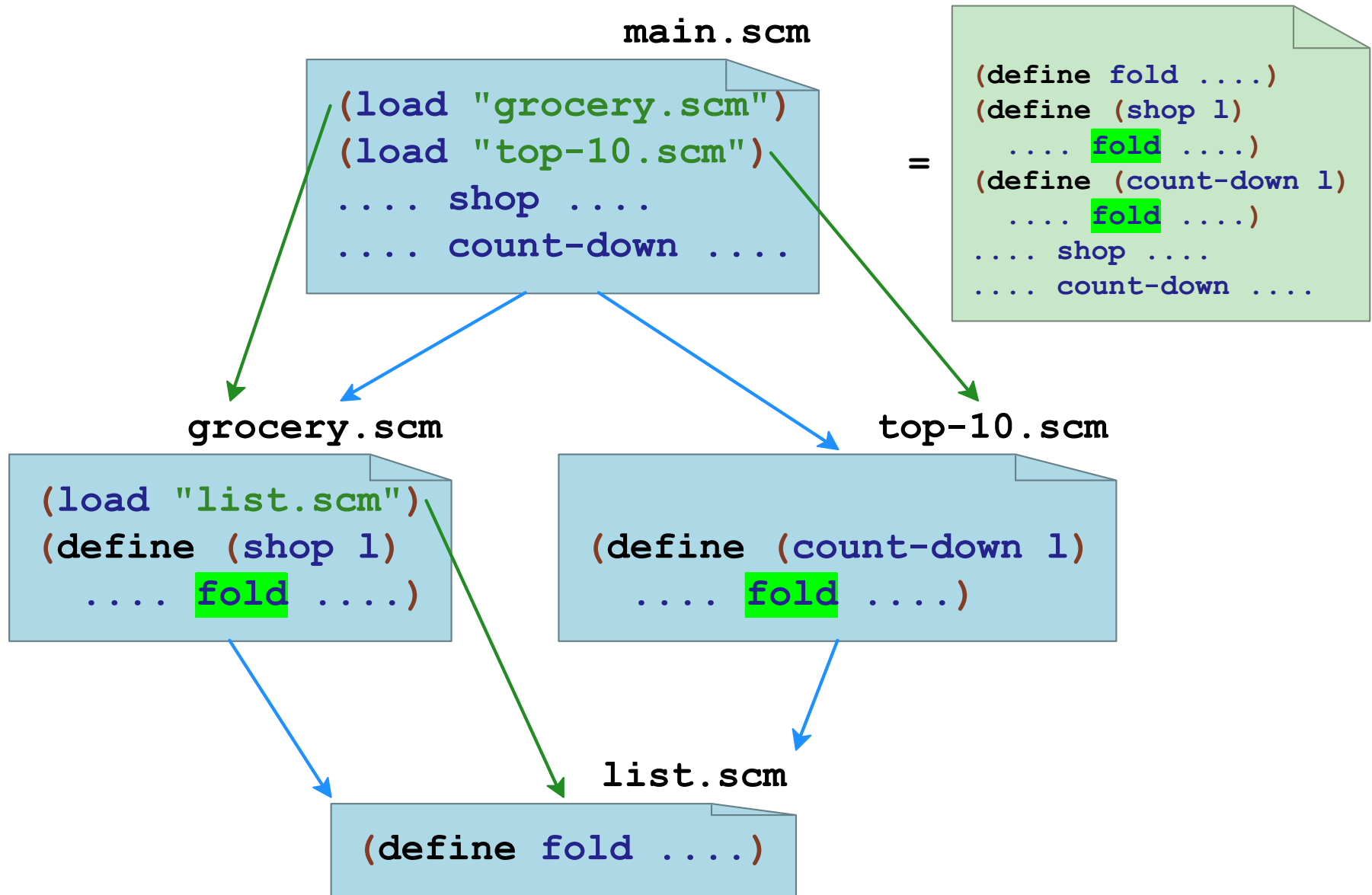
REPL-Oriented Program Structure



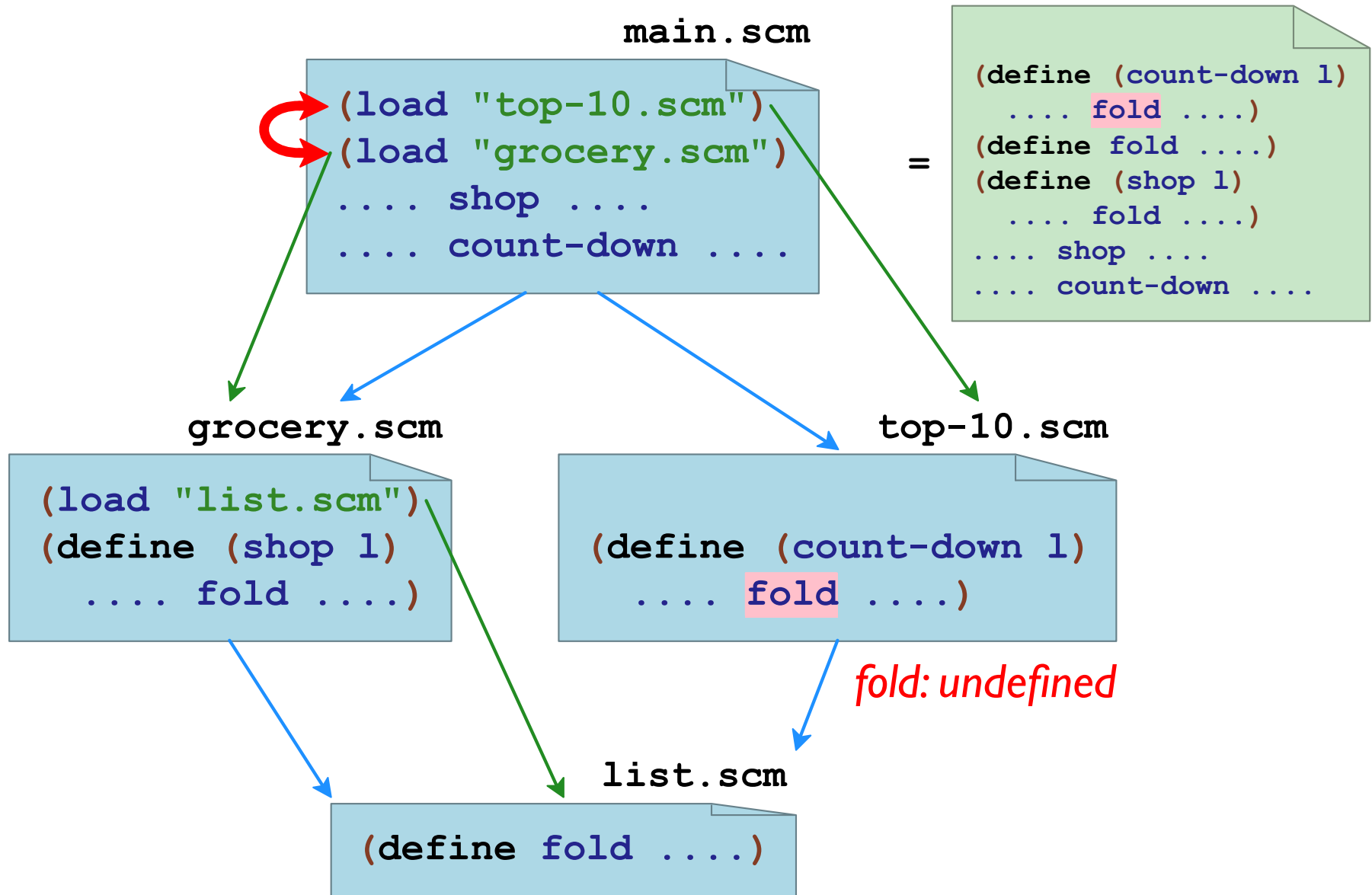
REPL-Oriented Program Structure



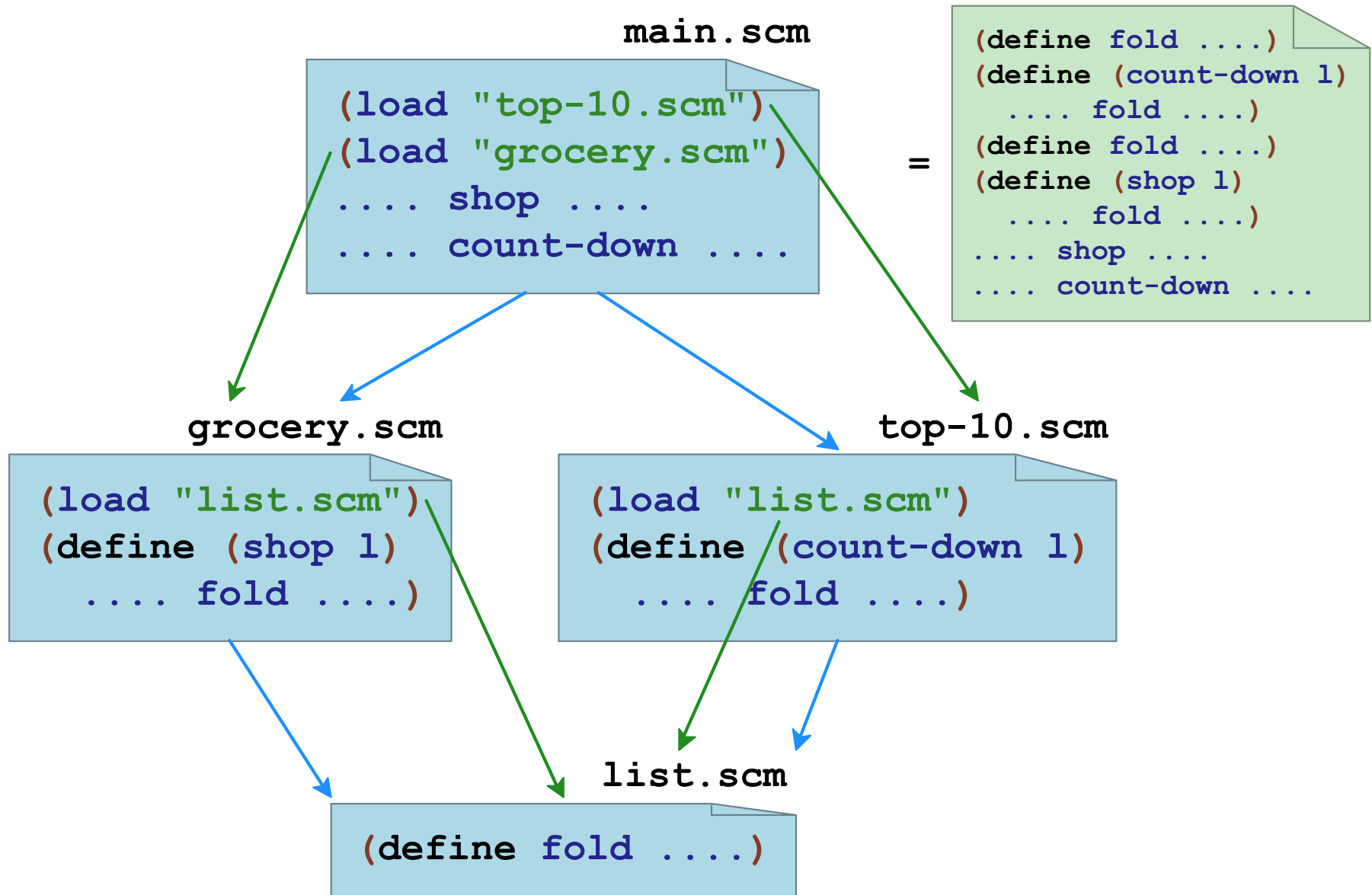
REPL-Oriented Program Structure



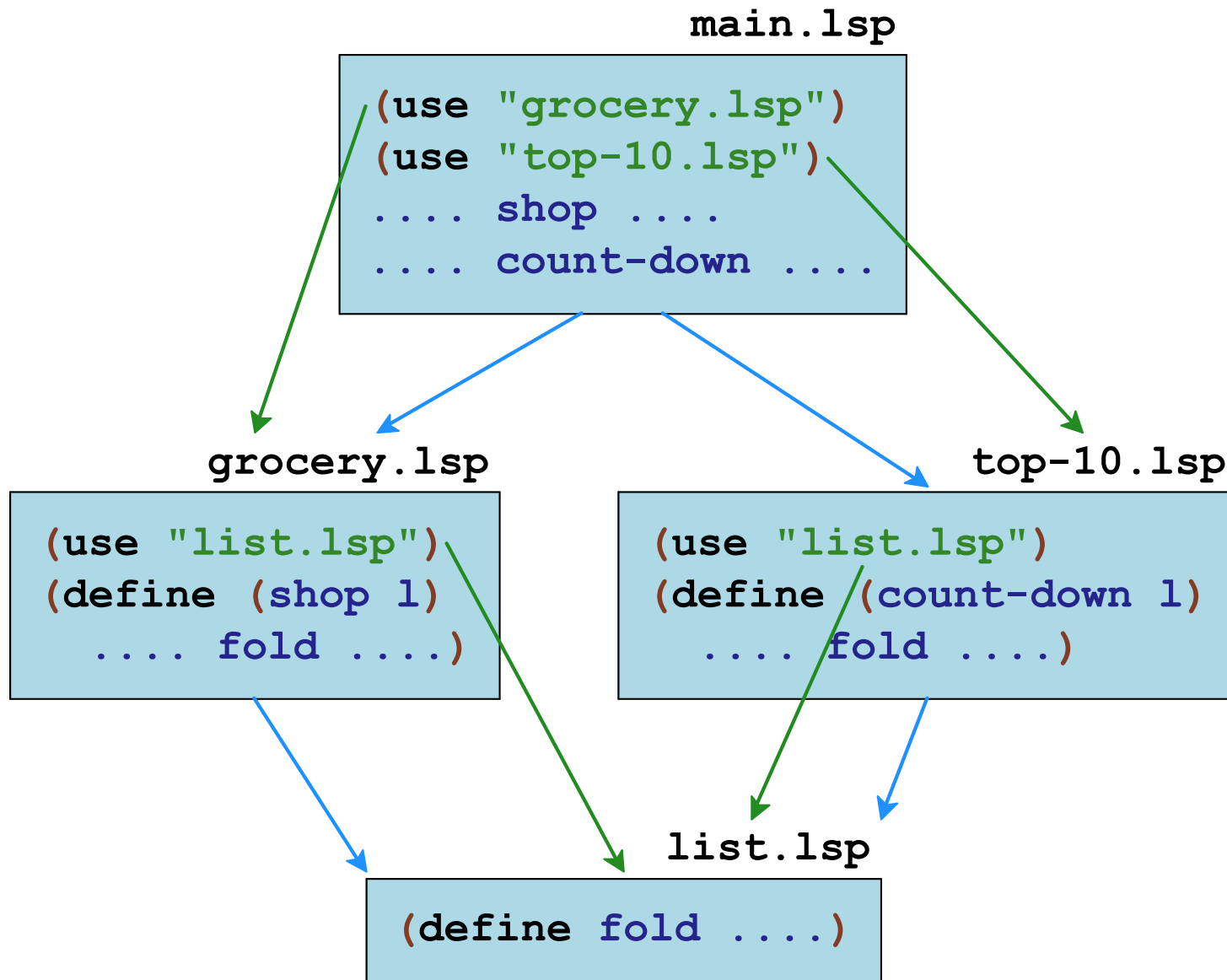
REPL-Oriented Program Structure



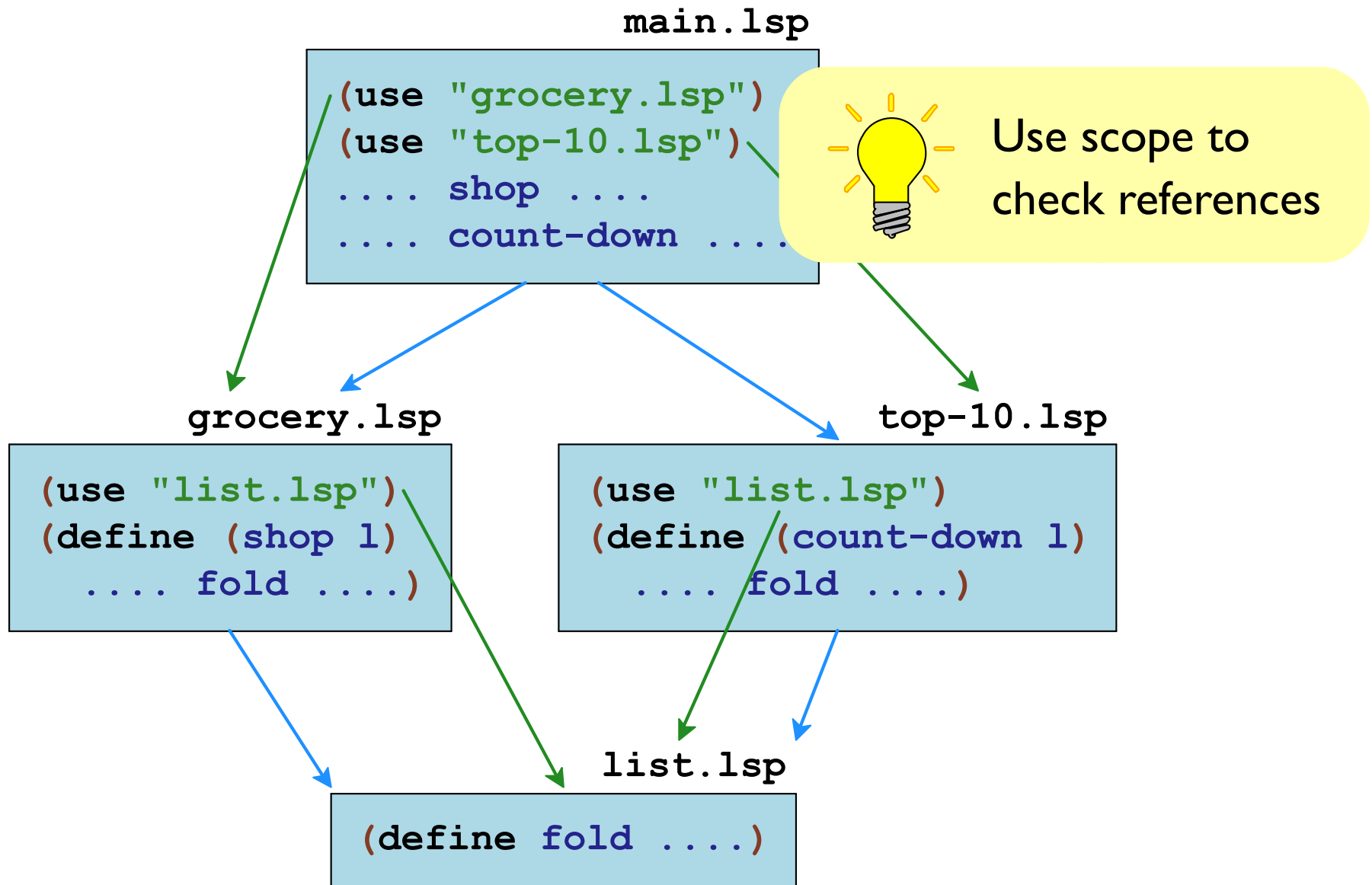
REPL-Oriented Program Structure



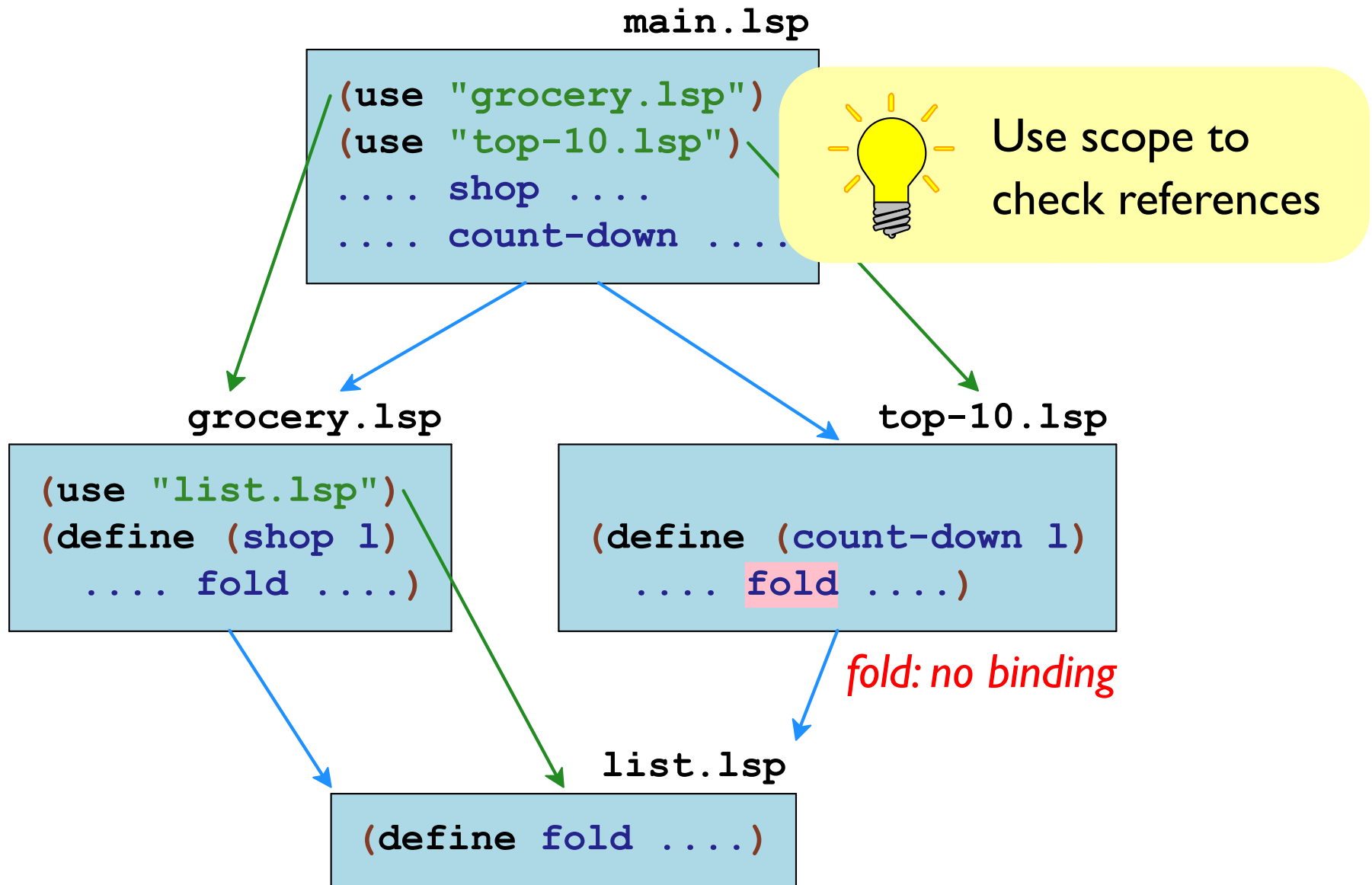
Adding a Namespace Layer



Adding a Namespace Layer



Adding a Namespace Layer



Using Namespaces

kitchen.lsp

```
(use "grocery.lsp")  
  
(shop ....)
```



grocery.lsp

```
(define shop ...)
```

Using Macros

kitchen.lsp



```
(use "grocery.lsp")  
  
(shop (groceries bread  
        [2 milk]  
        apples))
```


grocery.lsp

```
(define-syntax groceries ....)  
(define shop ...)
```

Using Macros

Compile `kitchen.lsp`...

`kitchen.lsp`



```
(use "grocery.lsp")  
  
(shop (groceries bread  
      [2 milk]  
      apples))
```

`grocery.lsp`

```
(define-syntax groceries ....)  
(define shop ...)
```

Using Macros

Compile `kitchen.lsp`...

`kitchen.lsp`

```
(use "grocery.lsp")  
  
(shop (groceries bread  
      [2 milk])
```

If `use` means run-time **load**:
`groceries` macro is not ready

```
(define-syntax groceries ....)  
(define shop ...)
```

Using Macros

Compile `kitchen.lsp`...

`kitchen.lsp`

```
(use "grocery.lsp")  
  
(shop (groceries bread  
      [2 milk])
```

If `use` means run-time **load**:
`groceries` macro is not ready

```
(define-syntax groceries ....)  
(define shop ...)
```



Scope should
ensure availability

Using Namespaces

Compile `kitchen.lsp`...

`kitchen.lsp`

```
(use "grocery.lsp")  
  
(shop ....)
```

`grocery.lsp`

```
(use "gui.lsp")  
  
(define shop ...)  
(define list-editor-gui ....)
```

`gui.lsp`

```
(init-gui-application!)  
.....
```


Using Namespaces

Compile `kitchen.lsp`...

`kitchen.lsp`

```
(use "grocery.lsp")  
  
(shop ....)
```

`grocery.lsp`

```
(use "gui.lsp")  
  
(init-gui ....)
```

If **use** means compile-time **load**:

GUI initialized during compile

`gui.lsp`

```
(init-gui-application!)  
....
```

Using Namespaces

Compile `kitchen.lsp`...

`kitchen.lsp`

```
(use "grocery.lsp")  
  
(shop ....)
```

`grocery.lsp`

```
(use "gui.lsp")  
  
(init-gui ....)
```

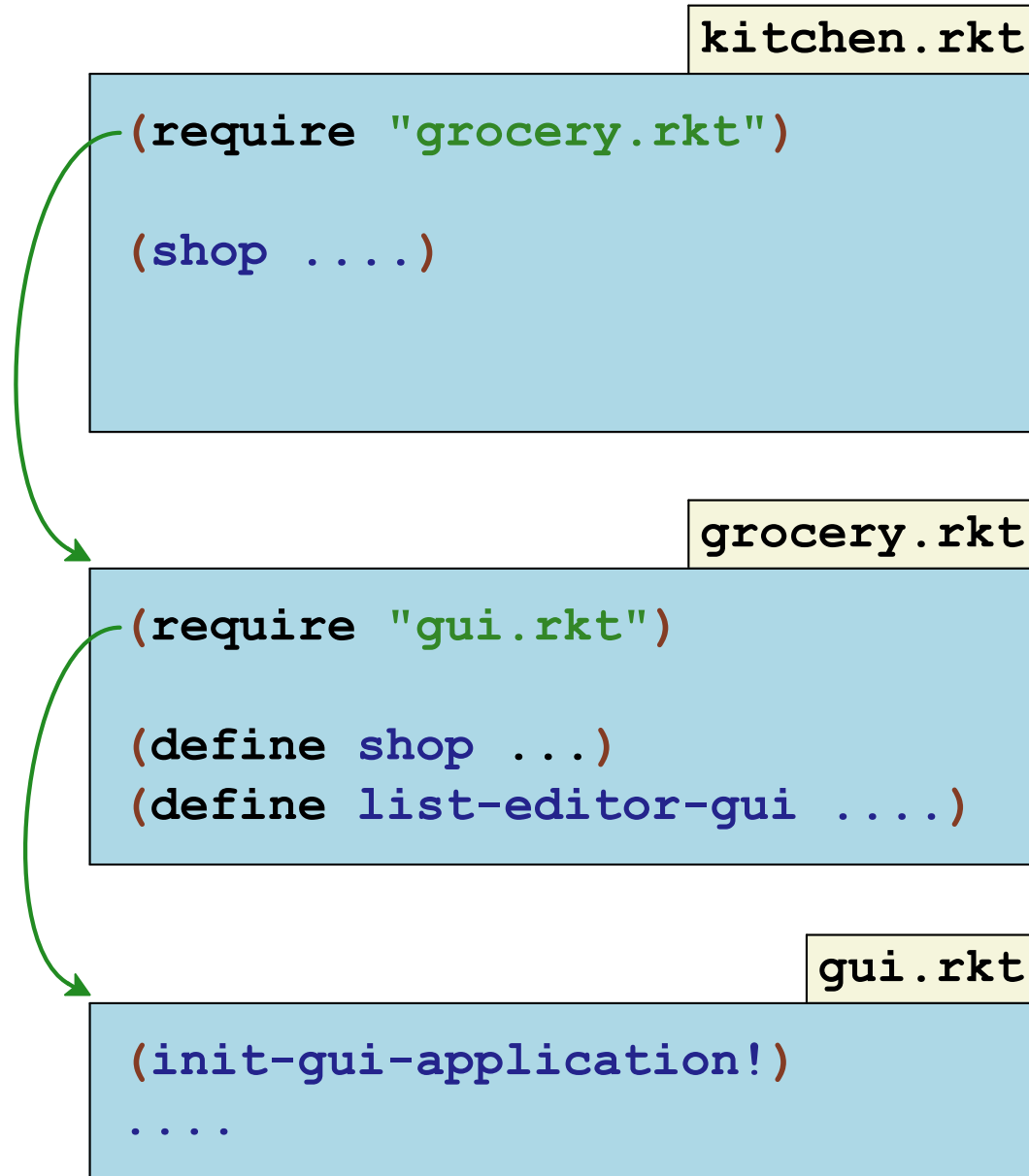
If **use** means compile-time **load**:
GUI initialized during compile

```
(init-gui-application!)  
.....
```

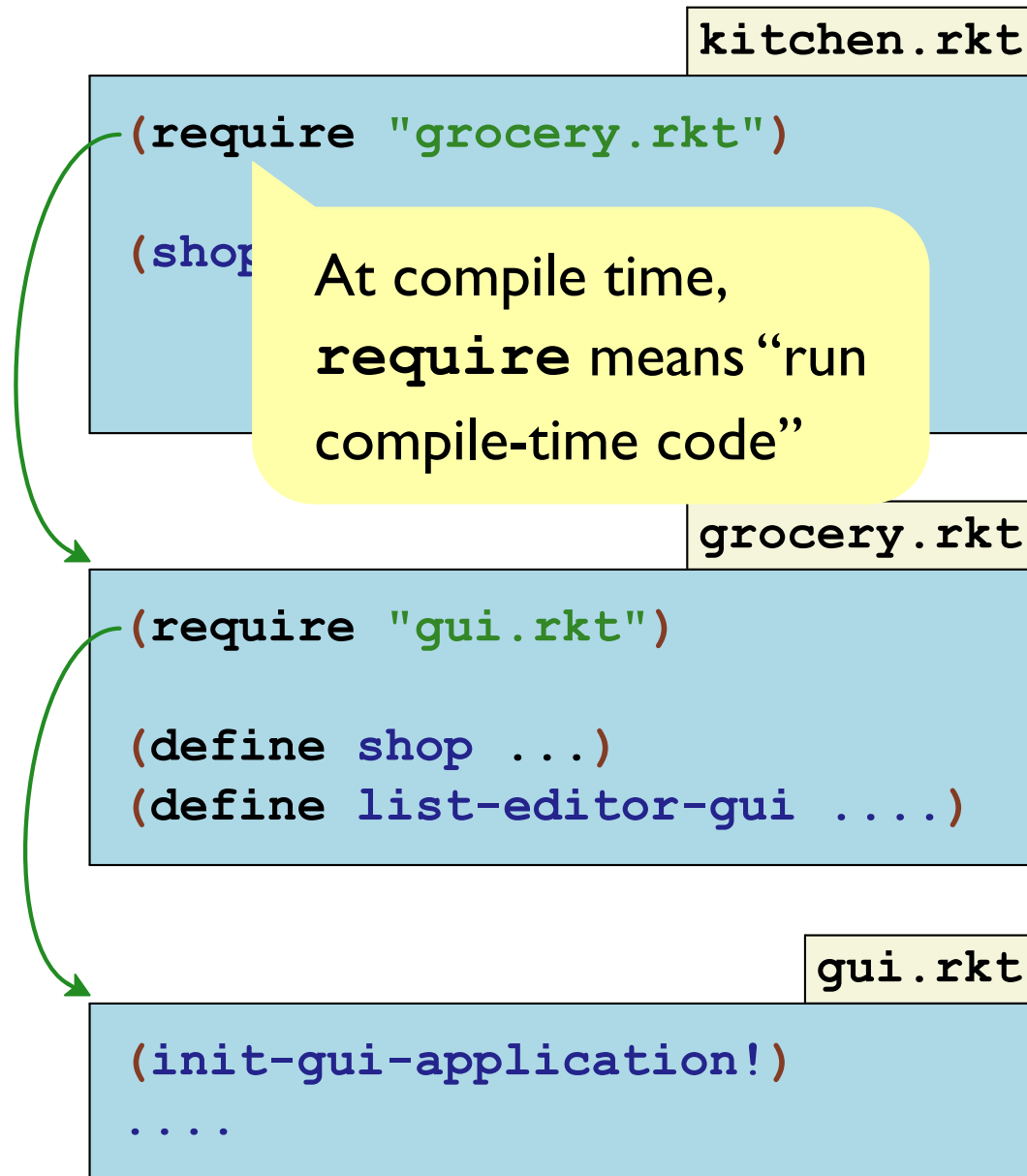


Scope should
make times
apparent

Using Racket Modules



Using Racket Modules



Scope and Phases

kitchen.rkt

```
(require "list.rkt"  
         "grocery.rkt")  
(define weekly (groceries ....))  
(shop .... fold ....)
```

grocery.rkt

```
(require "gui.rkt"  
         (for-syntax "list.rkt"))  
(define-syntax groceries .... fold ....)  
(define shop ....)  
(define list-editor-gui ....)
```

list.rkt

```
(define fold ....)
```

Scope and Phases

 = run time

kitchen.rkt

```
(require "list.rkt"  
         "grocery.rkt")  
(define weekly (groceries ....))  
(shop .... fold ....)
```

grocery.rkt

```
(require "gui.rkt"  
         (for-syntax "list.rkt"))  
(define-syntax groceries .... fold ....)  
(define shop ....)  
(define list-editor-gui ....)
```

list.rkt

```
(define fold ....)
```

Scope and Phases

 = run time

kitchen.rkt

```
(require "list.rkt"  
        "grocery.rkt")  
(define weekly (groceries ....))  
(shop .... fold ....)
```

grocery.rkt

```
(require "gui.rkt"  
        (for-syntax "list.rkt"))  
(define-syntax groceries .... fold ....)  
(define shop ....)  
(define list-editor-gui ....)
```

list.rkt

```
(define fold ....)
```

Scope and Phases

■ = run time ■ = compile time

kitchen.rkt

```
(require "list.rkt"  
        "grocery.rkt")  
(define weekly (groceries ....))  
(shop .... fold ....)
```

grocery.rkt

```
(require "gui.rkt"  
        (for-syntax "list.rkt"))  
(define-syntax groceries .... fold ....)  
(define shop ....)  
(define list-editor-gui ....)
```

list.rkt

```
(define fold ....)
```


Scope and Phases

■ = run time ■ = compile time



Scope and Phases

■ = run time ■ = compile time

kitchen.rkt

```
(require "list.rkt"  
        "grocery.rkt")  
(define weekly (groceries ....))  
(shop .... fold ....)
```

grocery.rkt

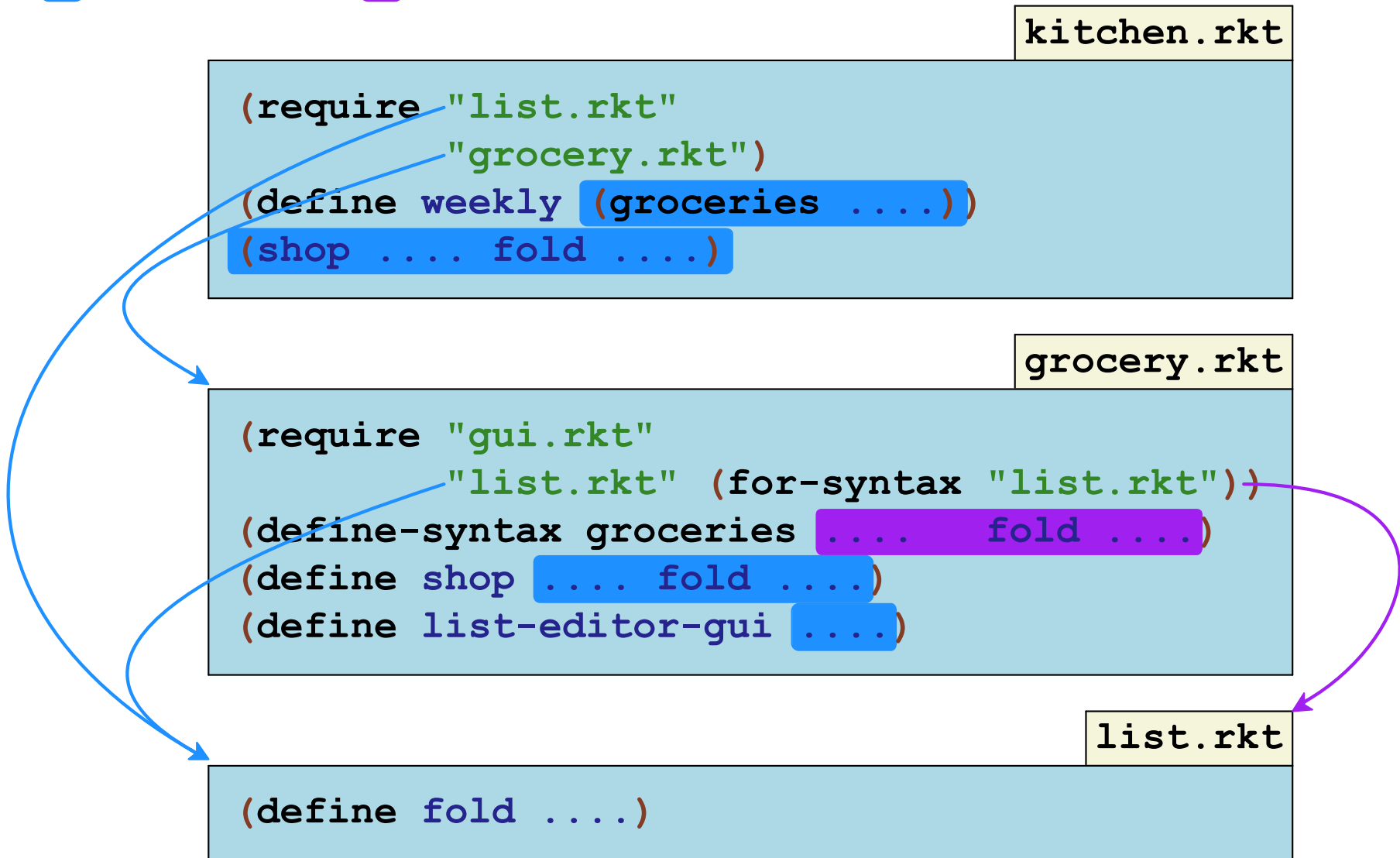
```
(require "gui.rkt"  
        "list.rkt")  
(define-syntax groceries .... #'fold ....)  
(define shop .... fold ....)  
(define list-editor-gui ....)
```

list.rkt

```
(define fold ....)
```

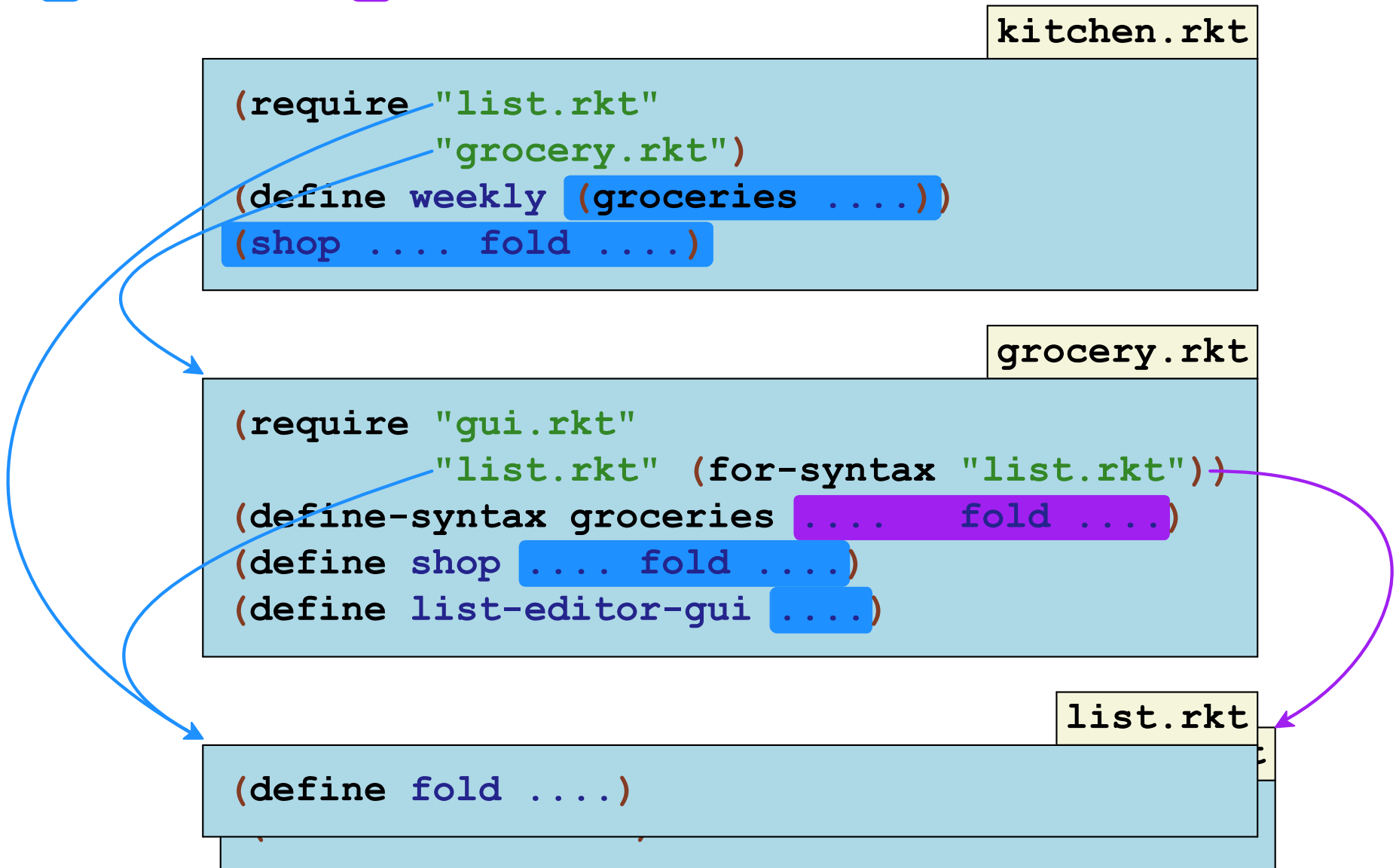
Scope and Phases

■ = run time ■ = compile time

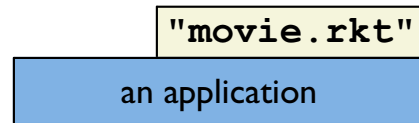


Scope and Phases

 = run time  = compile time



Dependency Chain in a Racket Program

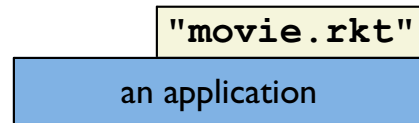


```
#lang racket/base
(require racket/gui/base
         racket/class
         "movie-panel.rkt")

....

(define mp (new movie-panel%
                  [parent f]
                  [file "stay-functional.mp4"])))
```

Dependency Chain in a Racket Program

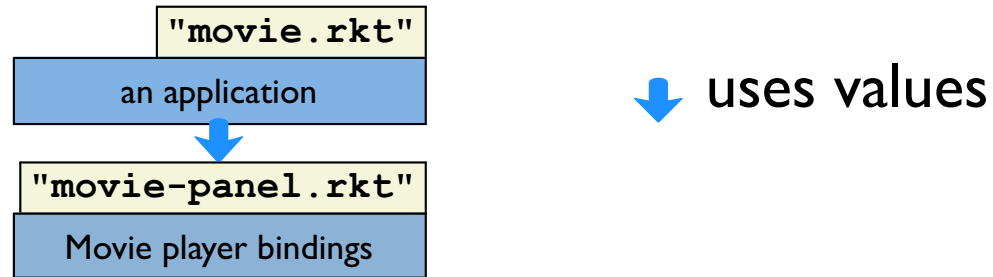


```
#lang racket/base
(require racket/gui/base
 racket/class
 "movie-panel.rkt")

....
(define mp (new movie-panel%
  [parent f]
  [file "stay-functional.mp4"])))
```

A blue arrow points from the text "movie-panel%" in the code block to the "movie.rkt" box in the diagram above, highlighting the dependency.

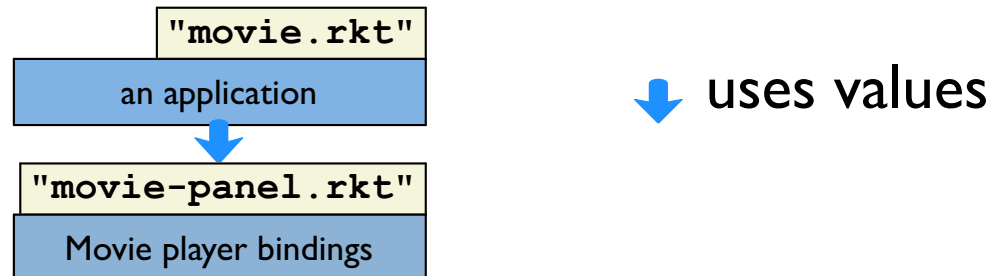
Dependency Chain in a Racket Program



```
#lang racket/base
(require racket/gui/base
 racket/class
 "movie-panel.rkt")

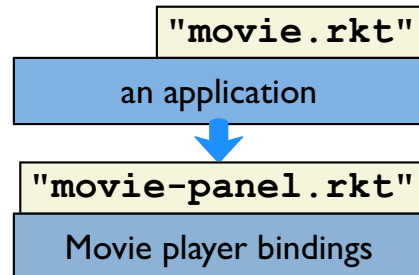
....
(define mp (new movie-panel%
                [parent f]
                [file "stay-functional.mp4"])))
```

Dependency Chain in a Racket Program



```
(define-objc-class MyMovieView QTMovieView
[]
[-a _void (keyDown: evt)
  (when (equal? "a" (event-string evt))
    (tell self gotoBeginning: self))
  (super-tell keyDown: evt)])
```

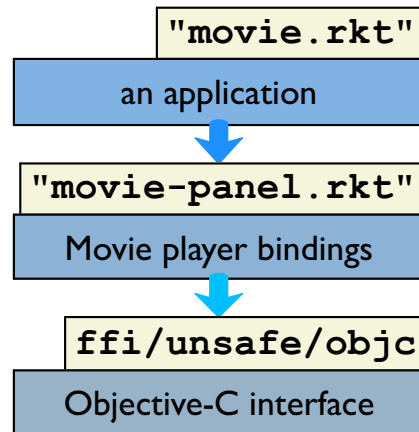

Dependency Chain in a Racket Program



↓ uses values

```
(define-objc-class MyMovieView QMainWindow
[]
[-a _void (keyDown: evt)
  (when (equal? "a" (event-string evt))
    (tell self gotoBeginning: self))
  (super-tell keyDown: evt)])
```

Dependency Chain in a Racket Program

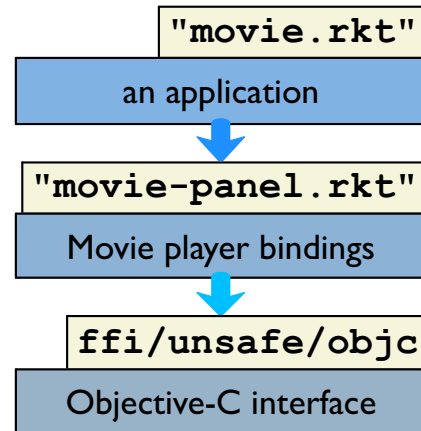


↓ uses values

↓ uses values & macros

```
(define-objc-class MyMovieView QTMovieView
[]
[-a _void (keyDown: evt)
  (when (equal? "a" (event-string evt))
    (tell self gotoBeginning: self))
  (super-tell keyDown: evt)])
```

Dependency Chain in a Racket Program

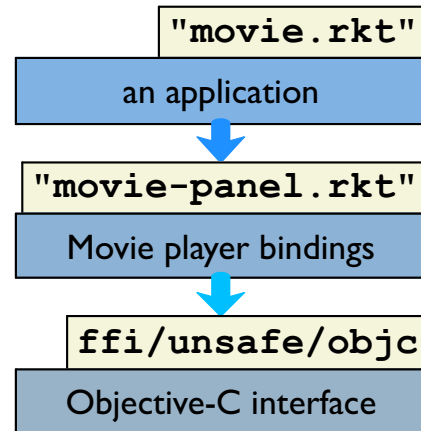


↓ uses values

↓ uses values & macros

```
(define-cstruct _objc_ivar
  ([name _pointer]
   [ivar_type _pointer]
   [ivar_offset _int]))
```

Dependency Chain in a Racket Program

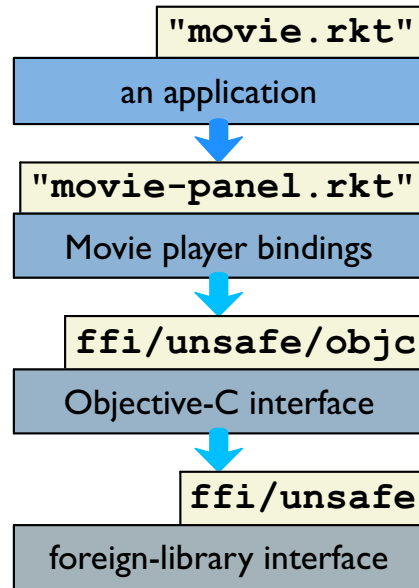


↓ uses values

↓ uses values & macros

```
(define-cstruct _objc_ivar
  ([name _pointer]
   [ivar_type _pointer]
   [ivar_offset _int]))
```

Dependency Chain in a Racket Program

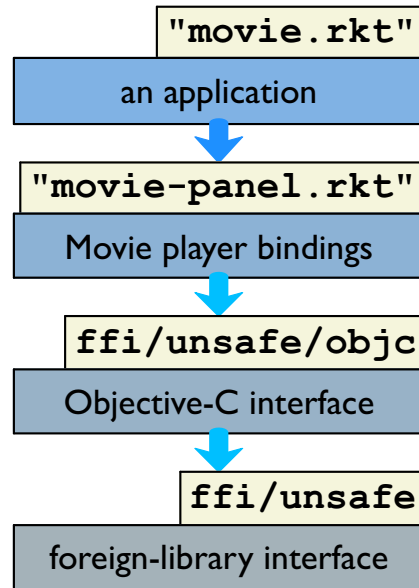


↓ uses values

↓ uses values & macros

```
(define-cstruct _objc_ivar  
  ([name _pointer]  
   [ivar_type _pointer]  
   [ivar_offset _int]))
```

Dependency Chain in a Racket Program

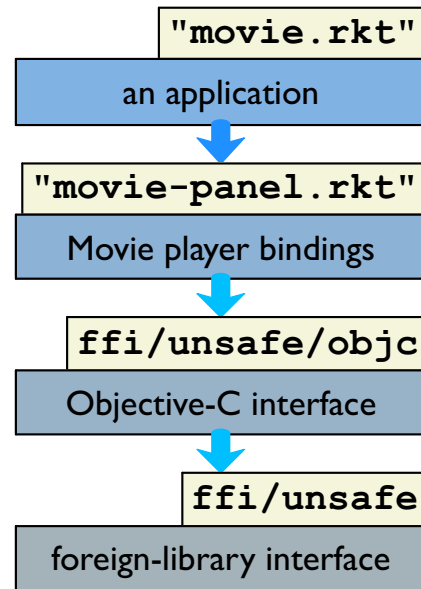


↓ uses values

↓ uses values & macros

```
(define (get-ffi-lib name
  [version/s ""]
  #:fail [fail #f]
  ....)
....)
```

Dependency Chain in a Racket Program

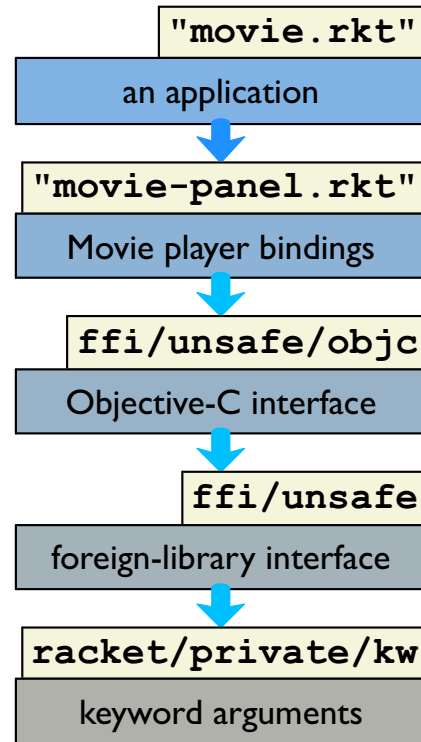


↓ uses values

↓ uses values & macros

```
(define (get-ffi-lib name
  [version/s ""]
  #:fail [fail #f]
  ....)
....)
```

Dependency Chain in a Racket Program

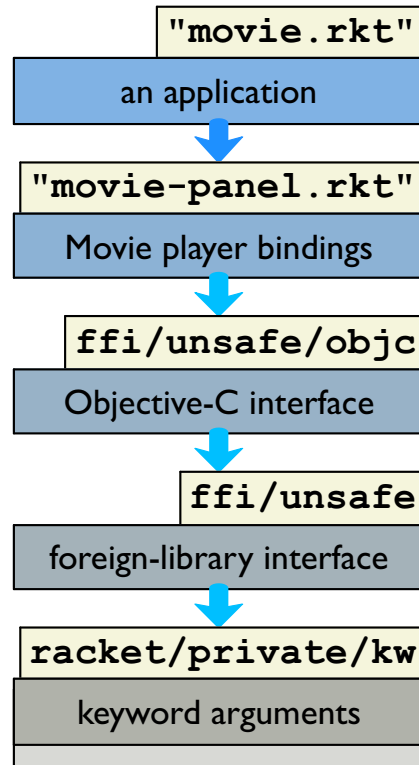


↓ uses values

↓ uses values & macros

```
(define (get-ffi-lib name
  [version/s ""]
  #:fail [fail #f]
  ....)
....)
```


Dependency Chain in a Racket Program

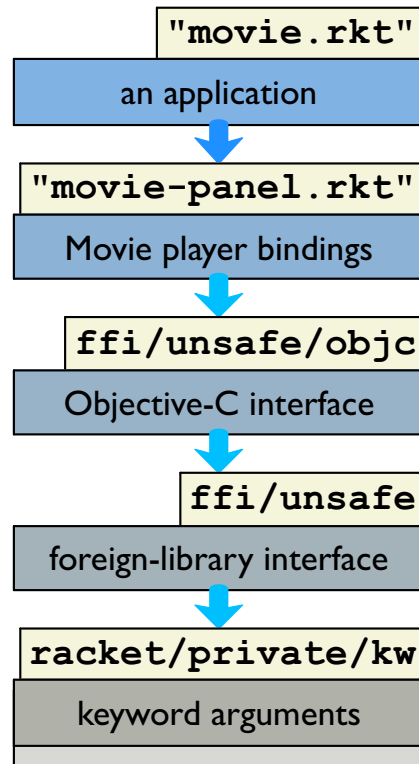


↓ uses values

↓ uses values & macros

```
(define-syntax (new-lambda stx)
  (syntax-case stx ()
    [(_ args body1 body ...)
     (if (simple-args? #'args)
         ....)]))
```

Dependency Chain in a Racket Program

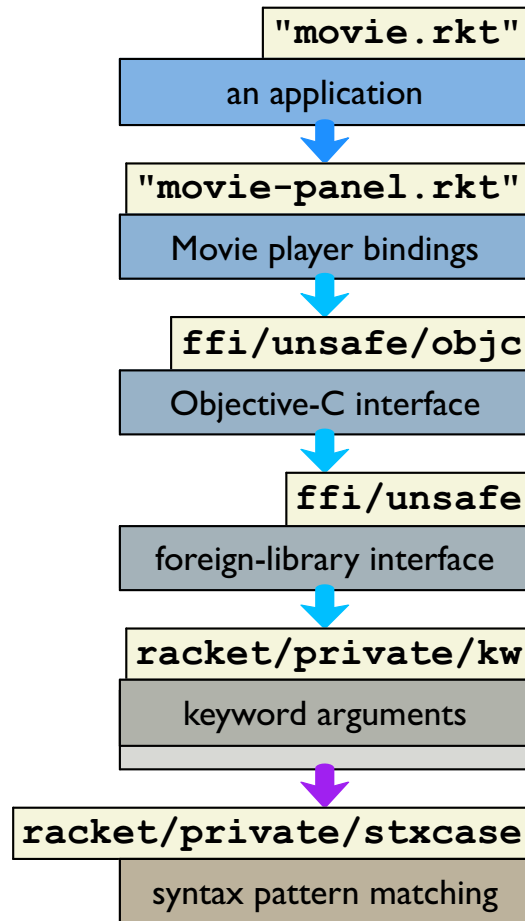


↓ uses values

↓ uses values & macros

```
(define-syntax (new-lambda stx)
  (syntax-case stx ()
    [(_ args body1 body ...)
     (if (simple-args? #'args)
         ....)]))
```

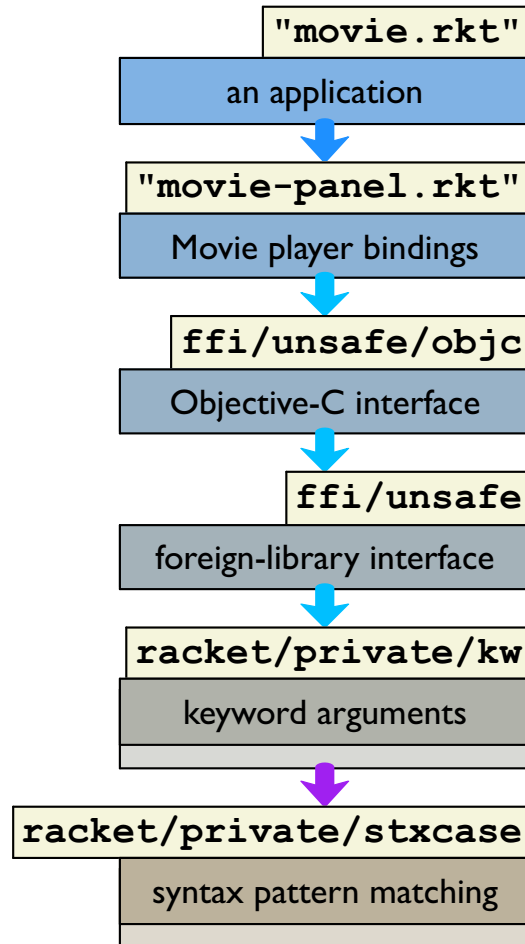
Dependency Chain in a Racket Program



- ↓ uses values
- ↓ uses values & macros
- ↓ macros use values & macros

```
(define-syntax (new-lambda stx)
  (syntax-case stx ()
    [(_ args body1 body ...)
     (if (simple-args? #'args)
         ....)]))
```

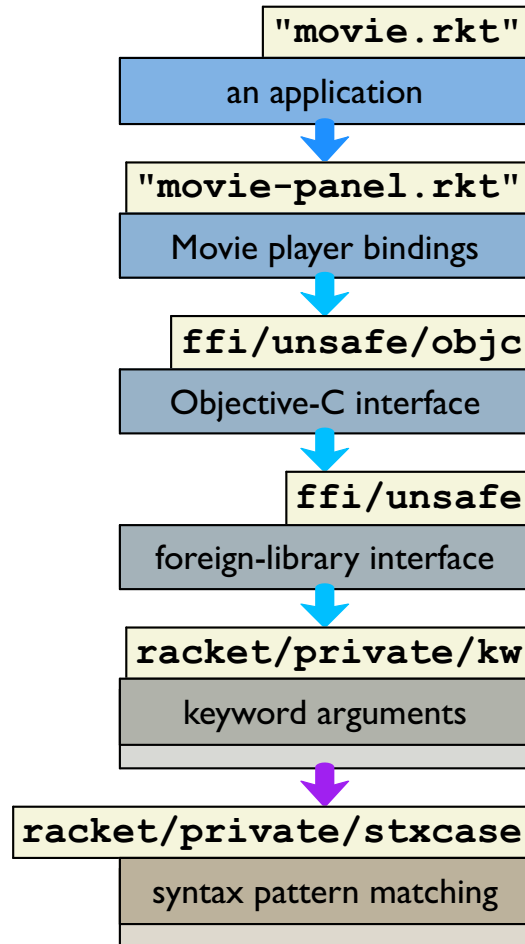
Dependency Chain in a Racket Program



- ↓ uses values
- ↓ uses values & macros
- ↓ macros use values & macros

```
(-define-syntax syntax-case**  
  (lambda (x)  
    (unless (stx-list? x)  
      ....)))
```

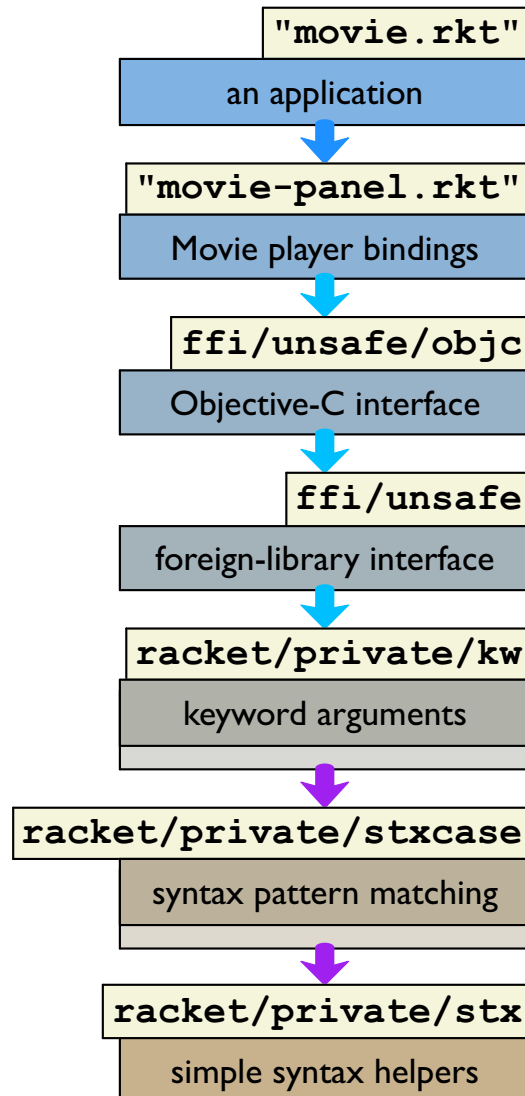
Dependency Chain in a Racket Program



- ↓ uses values
- ↓ uses values & macros
- ↓ macros use values & macros

```
(-define-syntax syntax-case**  
  (lambda (x)  
    (unless (stx-list? x)  
      ....)))
```

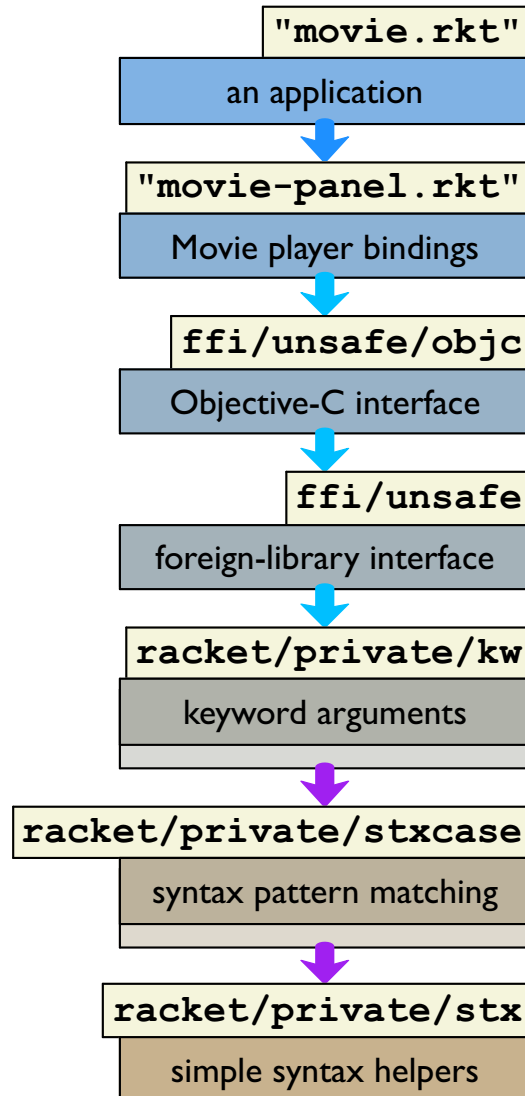
Dependency Chain in a Racket Program



- ↓ uses values
- ↓ uses values & macros
- ↓ macros use values & macros

```
(-define-syntax syntax-case**  
  (lambda (x)  
    (unless (stx-list? x)  
      ....)))
```

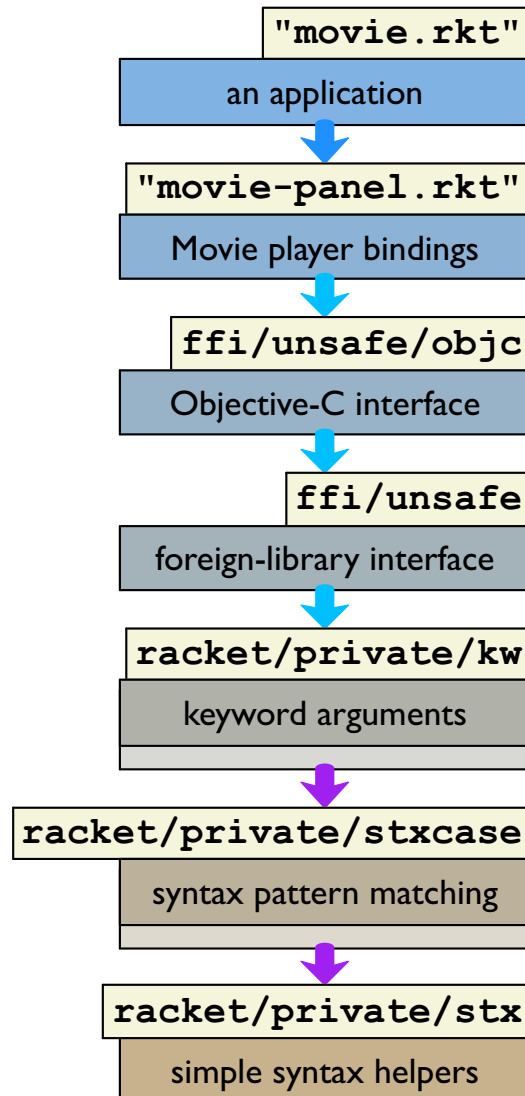
Dependency Chain in a Racket Program



- ↓ uses values
- ↓ uses values & macros
- ↓ macros use values & macros

```
(define-values (stx-list?)  
  (lambda (p)  
    (if (list? p)  
        #t  
        ...)))
```

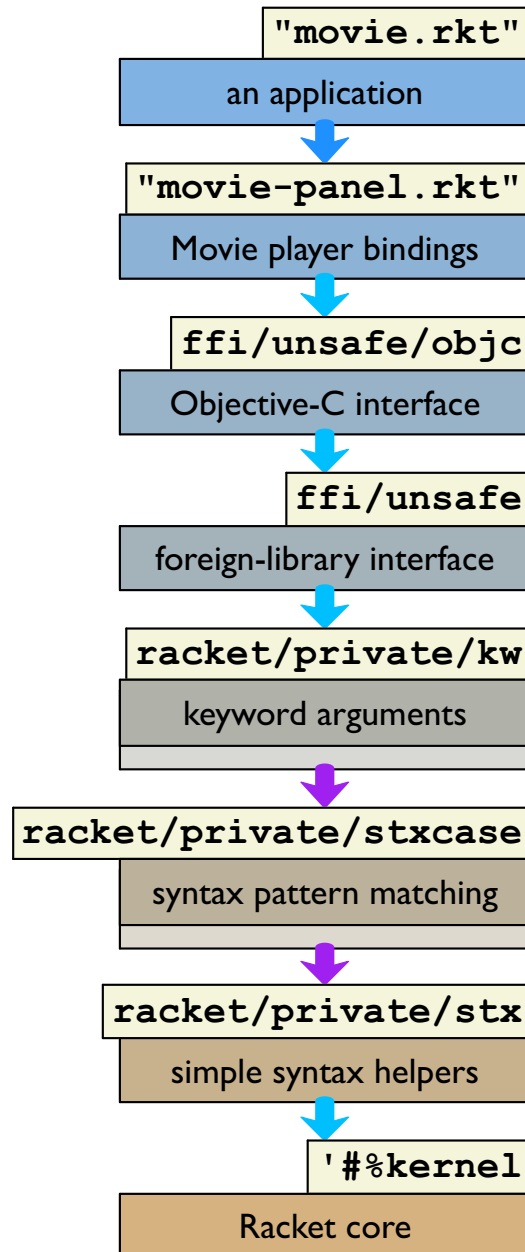
Dependency Chain in a Racket Program



- ↓ uses values
- ↓ uses values & macros
- ↓ macros use values & macros

```
(define-values (stx-list?)  
  (lambda (p)  
    (if (list? p)  
        #t  
        ...)))
```

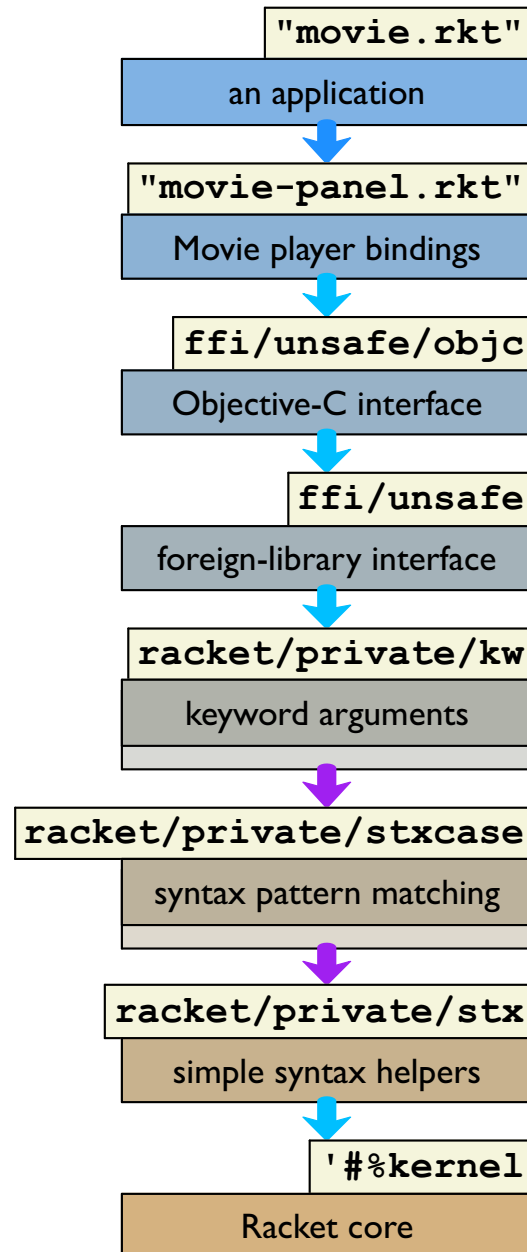

Dependency Chain in a Racket Program



- ↓ uses values
- ↓ uses values & macros
- ↓ macros use values & macros

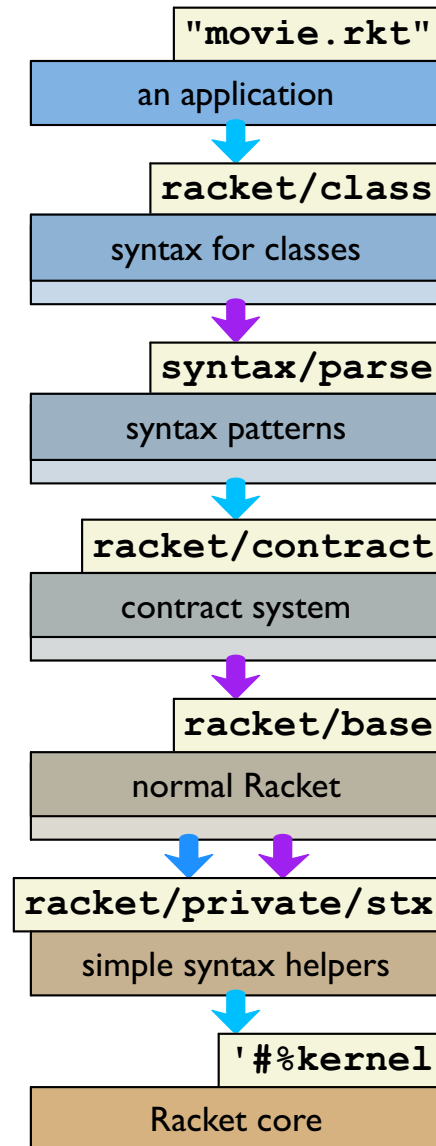
```
(define-values (stx-list?)  
  (lambda (p)  
    (if (list? p)  
        #t  
        ...)))
```

Dependency Chain in a Racket Program



- ↓ uses values
- ↓ uses values & macros
- ↓ macros use values & macros

Another Dependency Chain in a Racket Program



↓ uses values & macros

↓ macros use values & macros



Use Lexical Scope



Use Lexical Scope

Scope is declarative

```
(let ([x 5])  
  x)
```



Use Lexical Scope

Scope is declarative

```
(let ([x 5])  
  x)
```

main.rkt

```
(require "list.rkt"  
         "grocery.rkt")  
(shop (fold (groceries ....)))
```



Use Lexical Scope

Binding implies availability

```
(let ([x 5])  
  (+ x x))
```



Use Lexical Scope

Binding implies availability

```
(let ([x 5])  
  (+ x x))
```

grocery.rkt

```
(require "gui.rkt")  
.... (new frame%) ....
```




Use Lexical Scope

Scope need not imply dynamic extent

```
(lambda (x)
  (lambda (y)
    (+ x y)))
```



Use Lexical Scope

Scope need not imply dynamic extent

```
(lambda (x)
  (lambda (y)
    (+ x y)))
```

program.rkt

```
(module test ....)
```



Use Lexical Scope

Escape hatches are available

```
(parameterize ([current-output-port  
               (open-log-file)])  
  ....)
```



Use Lexical Scope

Escape hatches are available

```
(parameterize ([current-output-port  
                (open-log-file)])  
  ....)
```

gui.rkt

```
(dynamic-require  
  (case (system-type)  
    [(unix) "gtk.rkt"]  
    [(macosx) "cocoa.rkt"]  
    [(windows) "win32.rkt"])  
  ....)
```

Scope and Tests

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))
```

External Tests

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

(provide seconds->hours)
```

hours-test.rkt

```
(require rackunit)
(check-equal 0 (seconds->hours 0))
(check-equal 1 (seconds->hours 3600))
(check-equal 42 (seconds->hours 151200))
```

External Tests

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

(provide seconds->hours)
```

Forces export of internal function

hours-test.rkt

```
(require rackunit)
(check-equal 0 (seconds->hours 0))
(check-equal 1 (seconds->hours 3600))
(check-equal 42 (seconds->hours 151200))
```

Internal Tests Create Dependencies

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

; tests:
(require rackunit)
(check-equal 0 (seconds->hours 0))
(check-equal 1 (seconds->hours 3600))
(check-equal 42 (seconds->hours 151200))
```


Internal Tests Create Dependencies

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s 3600))

; tests:
(require rackunit)
(check-equal 0 (seconds->hours 0))
(check-equal 1 (seconds->hours 3600))
(check-equal 42 (seconds->hours 151200))
```

Creates a library dependency on testing framework

Internal Tests Create Dependencies

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

; tests:
(require rackunit)
(check-equal 0 (seconds->hours 0))
(check-equal 1 (seconds->hours 3600))
(check-equal 42 (seconds->hours 151200))
```

Test run every time library is used

Internal Tests Create Dependencies

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

; tests:
(require rackunit)
(check-equal 0 (seconds->hours 0))
(check-equal 1 (seconds->hours 3600))
(check-equal 42 (seconds->hours 151200))
```

Submodule Tests

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

(module+ test
  (require rackunit)
  (check-equal 0 (seconds->hours 0))
  (check-equal 1 (seconds->hours 3600))
  (check-equal 42 (seconds->hours 151200)))
```

Submodule Tests

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

(module+ test
  (require rackunit)
  (check-equal 0 (seconds->hours 0))
  (check-equal 1 (seconds->hours 3600))
  (check-equal 42 (seconds->hours 151200)))
```

```
> (require (submod "hours.rkt" tests))
```

Submodule Tests

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

(module+ test
  (require rackunit)
  (check-equal 0 (seconds->hours 0))
  (check-equal 1 (seconds->hours 3600))
  (check-equal 42 (seconds->hours 151200)))
```

```
% raco test hours.rkt
```

Submodule Tests

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

(module+ main
  (require rackunit)
  (check-equal 0 (seconds->hours 0))
  (check-equal 1 (seconds->hours 3600))
  (check-equal 42 (seconds->hours 151200)))
```

Submodule Tests

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

(module+ main
  (require rackunit)
  (check-equal 0 (seconds->hours 0))
  (check-equal 1 (seconds->hours 3600))
  (check-equal 42 (seconds->hours 151200)))
```

% racket hours.rkt

Submodule Tests

hours.rkt

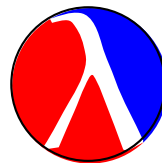
```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(module+ test
  (define-module+ test-module
    (require rackunit)
    (check-equal 0 (seconds->hours 0))
    (check-equal 1 (seconds->hours 3600))
    (check-equal 42 (seconds->hours 151200))))
```

module+ is a macro that generates
a module

Part 2: Modules and Syntax Objects



Module

now.rkt

```
#lang racket
```

```
(define now  
  (current-seconds))
```

```
now
```

1363840882

Module

now.rkt

```
#lang racket

(define now
  (current-seconds))

now
(sleep 1)
now
```

1363840882

1363840882

Modules

now.rkt

```
#lang racket  
  
(define now  
  (current-seconds))  
  
now
```

today.rkt

```
#lang racket  
(require "now.rkt")  
  
now
```

1363840883

now: unbound identifier in module

Modules

now.rkt

```
#lang racket
```

```
(define now  
  (current-seconds))
```

```
(provide now)
```

Modules

now.rkt

```
#lang racket  
  
(define now  
  (current-seconds))  
  
(provide now)
```

today.rkt

```
#lang racket  
(require "now.rkt")  
  
now
```

1363840883

Modules

today.rkt

```
#lang racket
(require "now.rkt")

now
```

now.rkt

```
#lang racket
(require "today.rkt")
(define now
  (current-seconds))

(provide now)
```

cycle in loading modules

The module Form

clock.rkt

```
(module clock.rkt racket  
  "tick")
```

"tick"

The module Form

clock.rkt

```
(module clock.rkt racket

  (module tick racket
    "tick")

)
```

The module Form

clock.rkt

```
(module clock.rkt racket  
  (module tick racket  
    "tick")  
  (require (submod "." tick))  
)
```

"tick"

The module Form

clock.rkt

```
(module clock.rkt racket

  (module tick racket
    "tick")
  (module tock racket
    "tock")

  (require (submod "." tick))
)
```

"tick"

The module Form

clock.rkt

```
(module clock.rkt racket

  (module tick racket
    "tick")
  (module tock racket
    "tock")

  (require (submod "." tock))
)
```

"tock"

The module Form

clock.rkt

```
(module clock.rkt racket

  (module tick racket
    "tick")
  (module tock racket
    (require (submod ".." tick))
    "tock")

  (require (submod "." tock))
)
```

"tick"

"tock"

The module Form

clock.rkt

```
(module clock.rkt racket
  (define sound "tick")

  (module tick racket
    sound)

  (require (submod "." tick))
)
```

sound: unbound identifier in module

The module Form

clock.rkt

```
(module clock.rkt racket

  (define sound "tick")

  (module* tick #f
    sound)
)
```


The module Form

clock.rkt

```
(module clock.rkt racket

  (define sound "tick")

  (module* test #f
    sound)
)
```

The module Form

clock.rkt

```
(module clock.rkt racket

  (define sound "tick")

  (module* test #f
    sound)
)
```

Symbols and Syntax Objects

tick.rkt

```
#lang racket  
  
(define sound "tick")  
  
(define id 'sound)  
(provide id)  
sound  
id
```

"tick"
'sound

Symbols and Syntax Objects

tick.rkt

```
#lang racket  
  
(define sound "tick")  
  
(define id 'sound)  
(provide id)
```

tock.rkt

```
#lang racket  
  
(define sound "tock")  
  
(require "tick.rkt")  
sound  
id
```

"tock"
'sound

Symbols and Syntax Objects

tick.rkt

```
#lang racket  
  
(define sound "tick")  
  
(define id 'sound)  
(provide id)
```

tock.rkt

```
#lang racket  
  
(define sound "tock")  
  
(require "tick.rkt")  
'sound  
id
```

'sound

'sound

Symbols and Syntax Objects

tick.rkt

```
#lang racket  
  
(define sound "tick")  
  
(define id 'sound)  
(provide id)
```

tock.rkt

```
#lang racket  
  
(define sound "tock")  
  
(require "tick.rkt")  
(eval 'sound)
```

*sound: undefined;
cannot reference undefined identifier*

Symbols and Syntax Objects

tick.rkt

```
#lang racket  
  
(define sound "tick")  
  
(define id 'sound)  
(provide id)
```

tock.rkt

```
#lang racket  
  
(define sound "tock")  
  
(require "tick.rkt")  
(eval #'sound)
```

"tock"

Symbols and Syntax Objects

tick.rkt

```
#lang racket  
  
(define sound "tick")  
  
(define id #'sound)  
(provide id)
```

tock.rkt

```
#lang racket  
  
(define sound "tock")  
  
(require "tick.rkt")  
(eval id)
```

"tick"

Symbols and Syntax Objects

tick.rkt

```
#lang racket  
  
(define sound "tick")  
  
(define id #'sound)  
(provide id)
```

tock.rkt

```
#lang racket  
  
(define sound "tock")  
  
(require "tick.rkt")  
#'sound  
id
```

```
#<syntax:6:2 sound>  
#<syntax:5:13 sound>
```

Symbols and Syntax Objects

tick.rkt

```
#lang racket

(define sound "tick")

(define id #'sound)
(provide id)
```

tock.rkt

```
#lang racket

(define sound "tock")

(require "tick.rkt")
(eval #'(list sound
               "!"))
```

```
' ("tock"  "!")
```

Symbols and Syntax Objects

tick.rkt

```
#lang racket

(define sound "tick")

(define id #'sound)
(provide id)
```

tock.rkt

```
#lang racket

(define sound "tock")

(require "tick.rkt")
(eval #`(list #,id
              "!"))
```

```
'("tick" "!")
```

Macros

now.rkt

```
#lang racket

(define now
  (current-seconds))

now
(sleep 1)
now
```

1363840887

1363840887

Macros

now.rkt

```
#lang racket

(define-syntax now
  (lambda (stx)
    #'(current-seconds)))

now
(sleep 1)
now
```

1363840888

1363840889

Macros

now.rkt

```
#lang racket

(define-syntax now
  (lambda (stx)
    #'(current-seconds)))

now
```

1363840889

Macros

now.rkt

```
#lang racket

(define-syntax now
  (lambda (stx)
    #'(current-seconds)))

(now 1 2 3 whatever)
```

1363840889

Macros

now.rkt

```
#lang racket

(define-syntax now
  (lambda (stx)
    #'(current-seconds)))

(now 1 2 3 whatever)
now
```

1363840889

1363840889

Macros

now.rkt

```
#lang racket

(define-syntax now
  (lambda (stx)
    #'(current-seconds)))

#' (now 1 2 3 whatever)
#' now
```

```
#<syntax:7:2 (now 1 2 3 whatever)>
#<syntax:8:2 now>
```

Macros

now.rkt

```
#lang racket

(define-syntax now
  (lambda (stx)
    #'(current-seconds)))

(syntax-e #'now)

(syntax-e #'(now 1 2 3 whatever))
```

'now

'(#<syntax:9:13 now> #<syntax:9:17 1> #<s

Macros

now.rkt

```
#lang racket

(define-syntax now
  (lambda (stx)
    (if (symbol? (syntax-e stx))
        #'(current-seconds)
        (raise-syntax-error
         #f
         "bad syntax"
         stx))))

now
(now 1 2 3 something)
```

now: bad syntax

Macros

now.rkt

```
#lang racket

(define-syntax now
  (lambda (stx)
    #'(current-seconds)))

now
```

1363840890

Macros

now.rkt

```
#lang racket
```

```
(define-syntax (now stx)  
  #' (current-seconds))
```

```
now
```

1363840890

Compile-Time Expressions

then.rkt

```
#lang racket

(define-syntax (then stx)
  (current-seconds))

then
```

*then: received value from syntax expander was not syntax
received: 1363840890*

Compile-Time Expressions

then.rkt

```
#lang racket

(define-syntax (then stx)
  #`#, (current-seconds))

then
```

1363840890

Compile-Time Expressions

then.rkt

```
#lang racket

(define-syntax (about-then stx)
  #`(- #, (current-seconds)
        10))

about-then
```

1363840880

Compile-Time Imports

then.rkt

```
#lang racket/base

(define-syntax (about-then stx)
  #`(- #, (current-seconds)
      10))

about-then
```

*quasisyntax: unbound identifier in the transformer environment;
also, no `#%app` syntax transformer is bound*

Compile-Time Imports

then.rkt

```
#lang racket/base
(require (for-syntax racket/base))

(define-syntax (about-then stx)
  #`(- #, (current-seconds)
      10))

about-then
```

1363840880

Compile-Time Imports

recent.rkt

```
#lang racket/base

(define (recent-seconds)
  (- (current-seconds) 10))

(provide recent-seconds)
```

about-then.rkt

```
#lang racket/base
(require (for-syntax racket/base
                      "recent.rkt"))

(define-syntax (about-then stx)
  #`#, (recent-seconds))

about-then
```

1363840880

Compile-Time Imports

recent.rkt

```
#lang racket/base

(define (recent-seconds)
  (- (current-seconds) 10))

(provide recent-seconds)
```

about-then.rkt

```
#lang racket/base
(require (for-syntax racket/base)
         "recent.rkt")

(define-syntax (about-then stx)
  #`#, (recent-seconds))

about-then
```

*recent-seconds: undefined;
cannot reference an identifier before its definition
phase: 1*

Macro-Generating Macros

main.rkt

```
#lang racket

(define-syntax (define-now-alias stx)
  (define id (cadr (syntax-e stx)))
  #`(define-syntax (#,id stx)
      #'(current-seconds)))

(define-now-alias now)
(define-now-alias at-this-moment)

at-this-moment
```

1363840891

Macro-Generating Macros and Imports

recent.rkt

```
#lang racket/base

(define (recent-seconds)
  (- (current-seconds) 10))

(provide recent-seconds)
```

main.rkt

```
#lang racket
(require "recent.rkt")

(define-syntax (define-now-alias stx)
  (define id (cadr (syntax-e stx)))
  #`(define-syntax (#,id stx)
      #'(recent-seconds)))

(define-now-alias now)
(define-now-alias at-this-moment)

at-this-moment
```

1363840881

Everything-Generating Macros

times.rkt

```
#lang racket

(define-syntax (define-times stx)
  (define rt-id (cadr (syntax-e stx)))
  (define ct-id (caddr (syntax-e stx)))
  (define proc (caddr (syntax-e stx)))
  #'(begin
      (define-syntax (#,rt-id stx)
        #'(#,proc (current-seconds)))
      (define-syntax (#,ct-id stx)
        #'(#,proc (current-seconds)))
      (provide #,rt-id #,ct-id)
      (module+ test
        (require rackunit)
        (check-equal? #,rt-id (begin (sleep 1) (- #,rt-id 1)))
        (check-equal? #,ct-id (begin (sleep 1) #,ct-id))))))

(provide define-times)
```

main.rkt

```
#lang racket
(require "times.rkt")
(define-times recently about-then
  (lambda (t) (- t 10)))
recently
```

1363840881

Everything-Generating Macros

times.rkt

```
#lang racket

(define-syntax (define-times stx)
  (define rt-id (cadr (syntax-e stx)))
  (define ct-id (caddr (syntax-e stx)))
  (define proc (caddr (syntax-e stx)))
  #'(begin
      (define-syntax (#,rt-id stx)
        #'(#,proc (current-seconds)))
      (define-syntax (#,ct-id stx)
        #'(#,proc (current-seconds)))
      (provide #,rt-id #,ct-id)
      (module+ test
        (require rackunit)
        (check-equal? #,rt-id (begin (sleep 1) (- #,rt-id 2)))
        (check-equal? #,ct-id (begin (sleep 1) #,ct-id))))))

(provide define-times)
```

main.rkt

```
#lang racket
(require "times.rkt")
(define-times recently about-then
  (lambda (t) (- t 10)))
recently
```

1363840881

FAILURE

name: check-equal?

location: (#<procedure:t> 15 8 414 54)

expression: (check-equal? recently (begin (sleep 1) (- recently 2)))

actual: 1363840881

expected: 1363840880

Check failure

Bindings and Phases

recent.rkt

```
#lang racket/base

(define (recent-seconds)
  (- (current-seconds) 10))

(provide recent-seconds)
```

main.rkt

```
#lang racket
(require "recent.rkt")

(define id #'recent-seconds)

(identifier-binding id)
(identifier-transformer-binding id)
```

```
' (#<module-path-index> recent-seconds #<m
#f
```

Bindings and Phases

recent.rkt

```
#lang racket/base

(define (recent-seconds)
  (- (current-seconds) 10))

(provide recent-seconds)
```

main.rkt

```
#lang racket
(require (for-syntax "recent.rkt"))

(define id #'recent-seconds)

(identifier-binding id)
(identifier-transformer-binding id)
```

#f

' (#<module-path-index> recent-seconds #<m

Bindings and Phases

recent.rkt

```
#lang racket/base

(define (recent-seconds)
  (- (current-seconds) 10))

(provide recent-seconds)
```

main.rkt

```
#lang racket
(require "recent.rkt"
         (for-syntax "recent.rkt"))

(define id #'recent-seconds)

(identifier-binding id)
(identifier-transformer-binding id)
```

```
' (#<module-path-index> recent-seconds #<m
' (#<module-path-index> recent-seconds #<m
```

Bindings and Phases

recent.rkt

```
#lang racket/base

(define (recent-seconds)
  (- (current-seconds) 10))

(provide recent-seconds)
```

main.rkt

```
#lang racket
(require (rename-in racket/base
                    [current-seconds
                     recent-seconds])
         (for-syntax "recent.rkt"))

(define id #'recent-seconds)

(identifier-binding id)
(identifier-transformer-binding id)
```

```
' (#<module-path-index> current-seconds #<
' (#<module-path-index> recent-seconds #<m
```

Documentation

```
#lang scribble/manual
@(require (for-label lang/htdp-beginner))

@(define (step n)
  @section{Step @(format "~a" n)})

@step[1]

Define the function @racket[seconds->days]
using @racket[define].

@step[2]

Define the function @racket[current-days]
using @racket[define].
```

Documentation

```
#lang scribble/manual
@(require (for-label lang/htdp-beginner))
```

```
@(define (step n)
  @section{Step @(format "~a" n)})
```

```
@step[1]
```

```
Define the function @racket[seconds->days]
using @racket[define].
```

```
@step[2]
```

```
Define the function @racket[current-days]
using @racket[define].
```

Documentation

normal Racket **define**

```
#lang scribble/manual  
@(require (for-label lang/htdp-beginner))
```

```
@(define (step n)  
  @section{Step @(format "~a" n)}))
```

```
@step[1]
```

Define the function `@racket[seconds->days]`
using `@racket[define]`.

```
@step[2]
```

Define the function `@racket[current-days]`
using `@racket[define]`.

Documentation

normal Racket `define`

```
#lang scribble/manual  
@(require (for-label lang/htdp-beginner))
```

```
@(define (step n)  
  @section{Step @ (format "~a" n) })
```

```
@step[1]
```

Define the function `@racket[seconds->days]`
using `@racket[define]`.

```
@step[2]
```

Define the function `@racket[current-days]`
using `@racket[define]`.

Form that hyperlinks code based on
`for-label` bindings

Docstrings

```
(define current-seconds  
  (lambda ()  
    "reports the time in seconds since the Epoch"  
    ....))
```

```
(define current-days  
  (lambda ()  
    "reports the time in days since the Epoch"  
    ....))
```

```
(define current-years  
  (lambda ()  
    "reports the time in years since the Epoch"  
    ....))
```

Abstraction Over Docstrings

```
(define (docs-for-current what)
  (format "reports the time in ~a since the Epoch"
    what))
```

```
(define current-seconds
  (lambda ()
    (docs-for-current "seconds")
    ....))
```

```
(define current-days
  (lambda ()
    (docs-for-current "days")
    ....))
```

```
(define current-years
  (lambda ()
    (docs-for-current "years")
    ....))
```

Rackety Docstrings

program.rkt

```
(begin-for-doc
  (define (docs-for-current what)
    (format
      "reports the time in ~a since the Epoch"
      what)))

(define current-seconds
  (lambda ()
    #:doc (docs-for-current "seconds")
    ....)))
```

Rackety Docstrings

program.rkt

```
(module+ doc
  (define (docs-for-current what)
    (format
      "reports the time in ~a since the Epoch"
      what)))

(define current-seconds
  (lambda ()
    ....))

(module+ doc
  (provide current-seconds)
  (define current-seconds
    (docs-for-current "seconds")))
```

In-Source Scribble Documentation

time.rkt

```
(define (seconds->hours secs)
  #:contract (exact-integer? . -> . exact-integer?)
  #:docs @{\Takes @racket[secs], a number of seconds
           since the Epoch, and converts it to a number
           of days since the Epoch, rounding down.

           For example, compose with with the
           @racket[current-seconds] function to get
           @racket[current-hours].}
  (quotient secs (* 60 60)))
```

No Project Files

movie.rkt

```
#lang racket  
....
```

To run:

```
% racket movie.rkt
```

No Project Files

movie.rkt

```
#lang racket  
....
```

To compile:

```
% raco make movie.rkt
```

No Project Files

movie.rkt

```
#lang racket  
....
```

To create an executable:

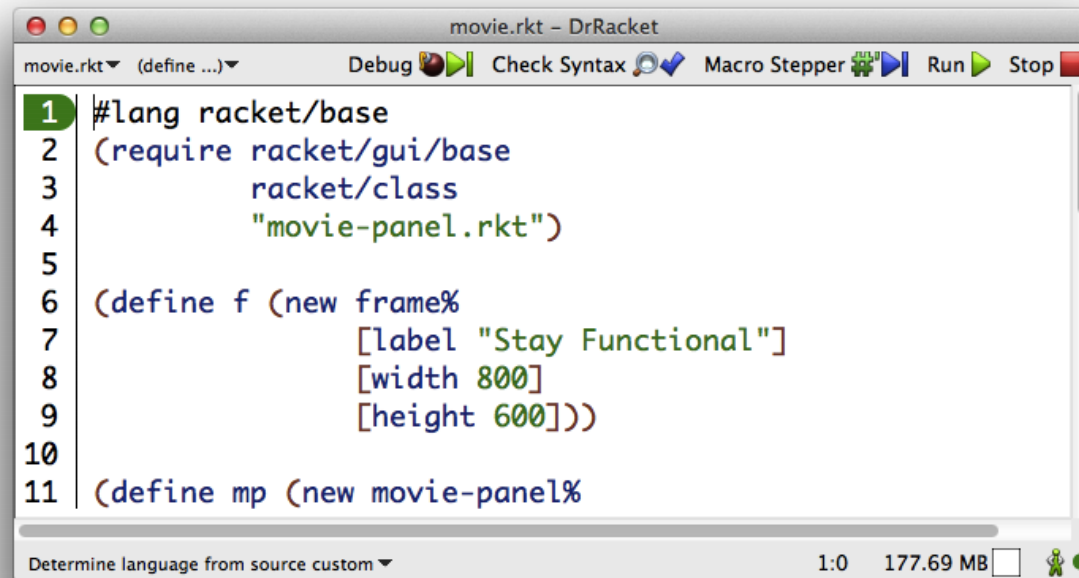
```
% raco exe movie.rkt
```


No Project Files

```
movie.rkt  
#lang racket  
....
```

To use the IDE:

```
% drracket movie.rkt
```



The screenshot shows the DrRacket IDE window titled "movie.rkt - DrRacket". The menu bar includes "movie.rkt", "(define ...)", "Debug", "Check Syntax", "Macro Stepper", "Run", and "Stop". The code editor displays the following Racket code:

```
1 #lang racket/base  
2 (require racket/gui/base  
3         racket/class  
4         "movie-panel.rkt")  
5  
6 (define f (new frame%  
7         [label "Stay Functional"]  
8         [width 800]  
9         [height 600]))  
10  
11 (define mp (new movie-panel%
```

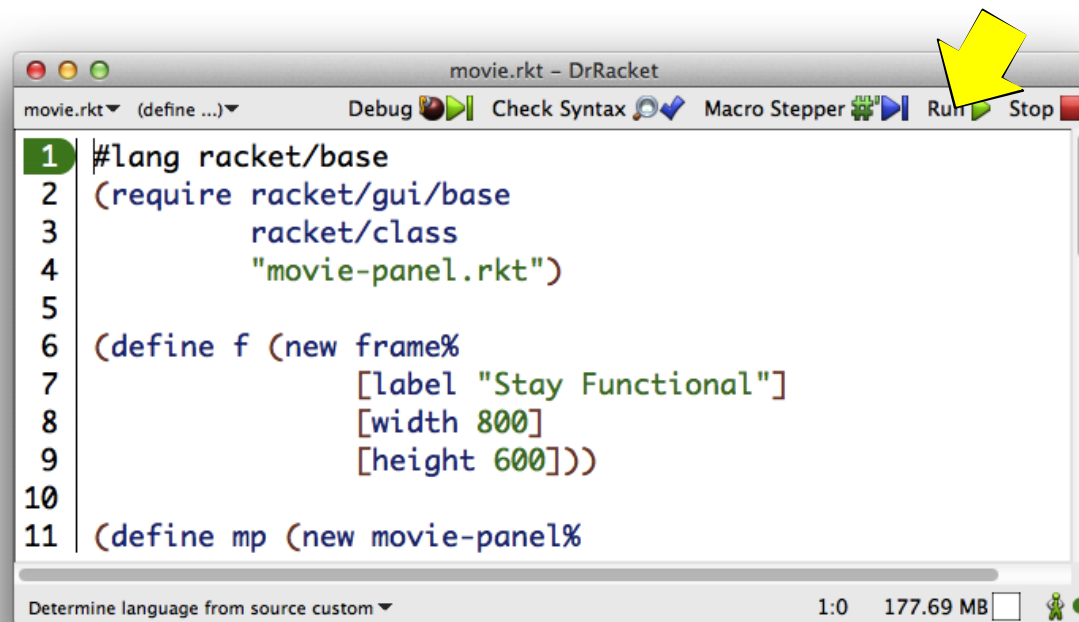
The status bar at the bottom shows "Determine language from source custom", "1:0", and "177.69 MB".

No Project Files

```
movie.rkt  
#lang racket  
....
```

To use the IDE:

```
% drracket movie.rkt
```

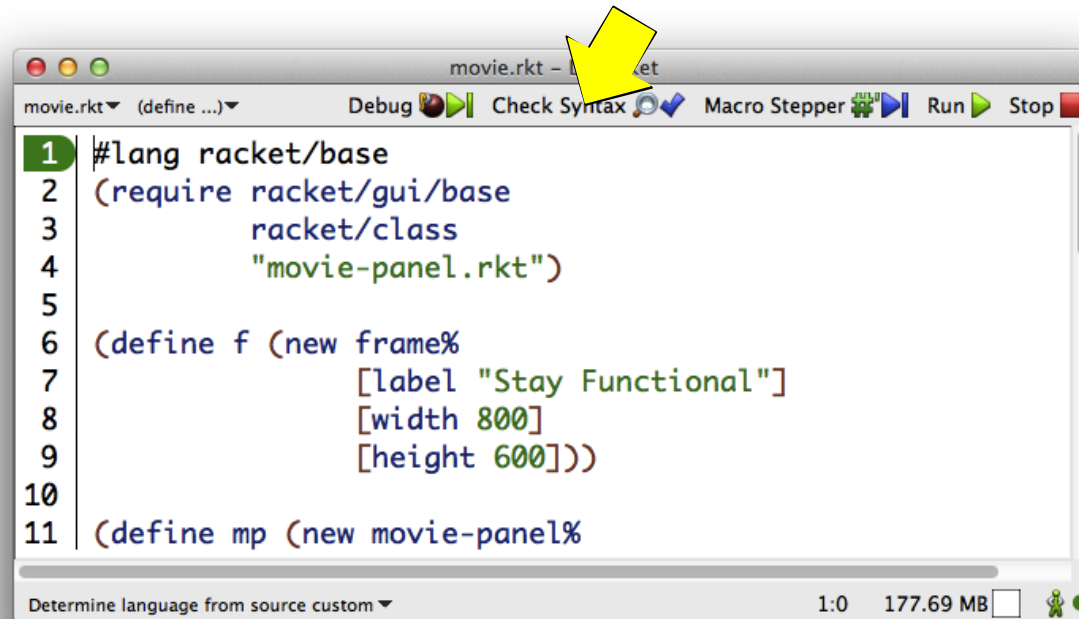


No Project Files

```
movie.rkt  
#lang racket  
....
```

To use the IDE:

```
% drracket movie.rkt
```

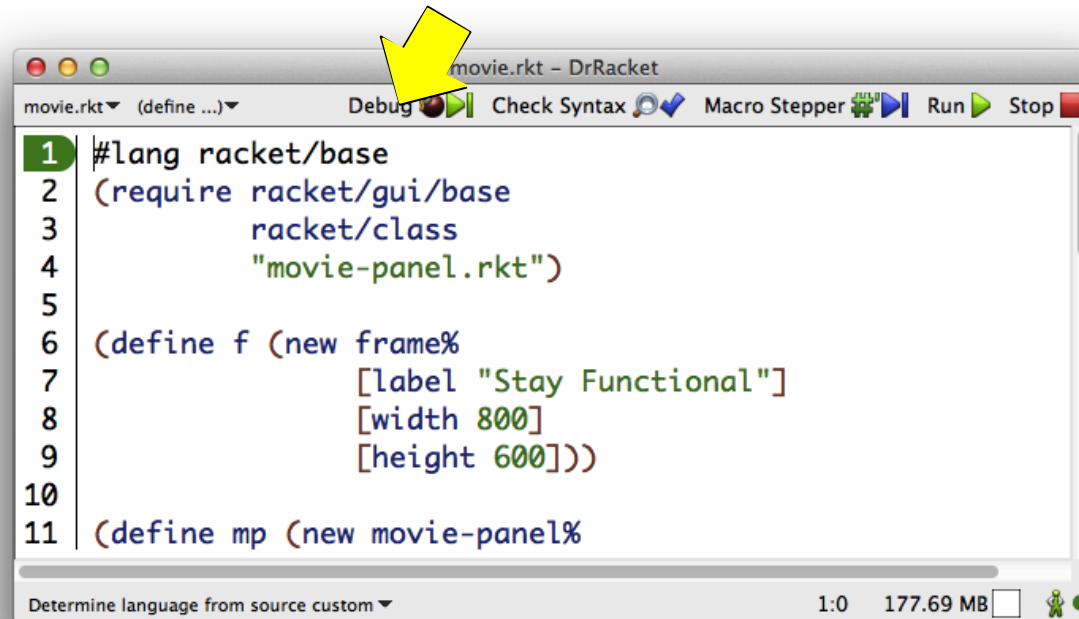


No Project Files

```
movie.rkt  
#lang racket  
....
```

To use the IDE:

```
% drracket movie.rkt
```

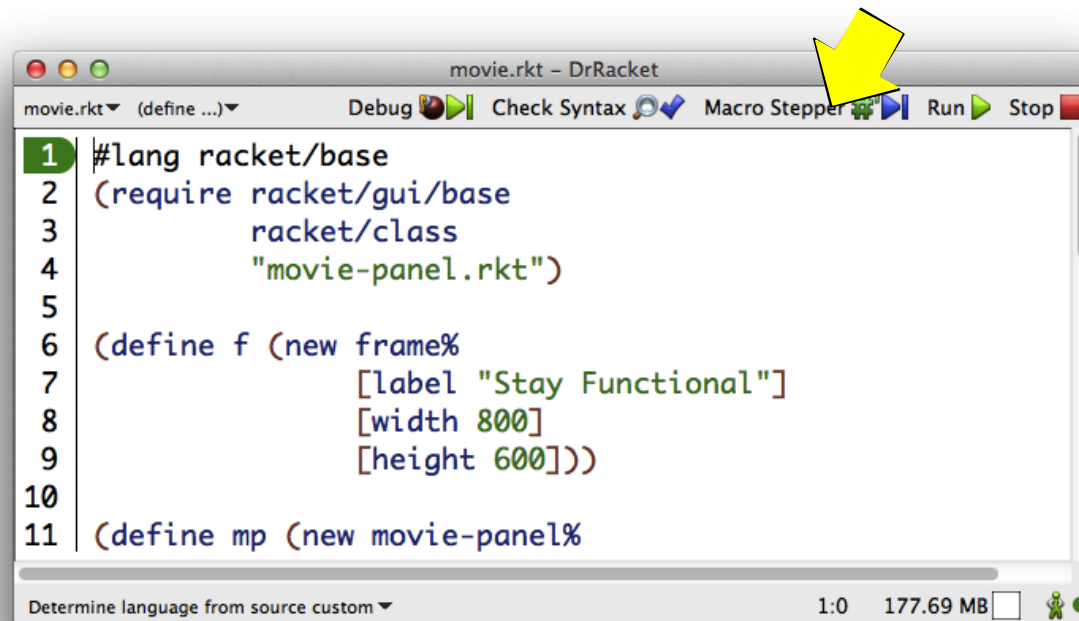


No Project Files

```
movie.rkt  
#lang racket  
....
```

To use the IDE:

```
% drracket movie.rkt
```



Learning More

- <http://docs.racket-lang.org>

- “Fear of Macros”

<http://www.greghendershott.com/fear-of-macros/>
Hendershott

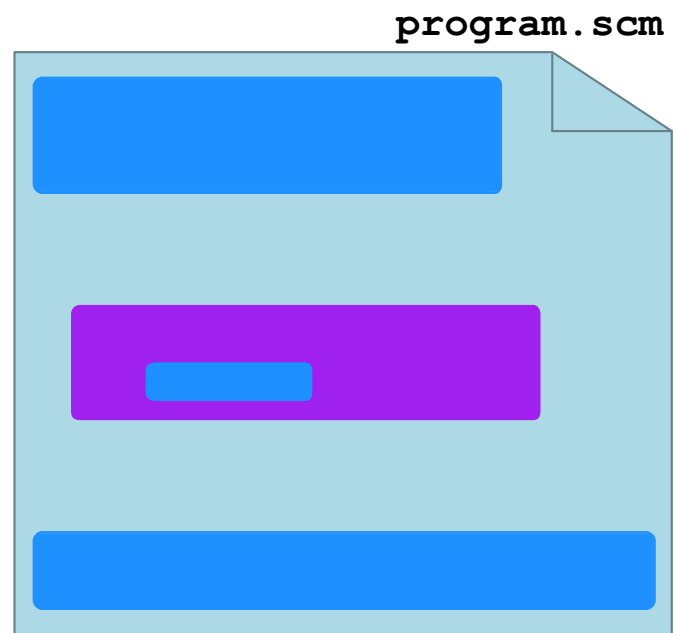
- “Composable and Compilable Macros”

Flatt, ICFP’02

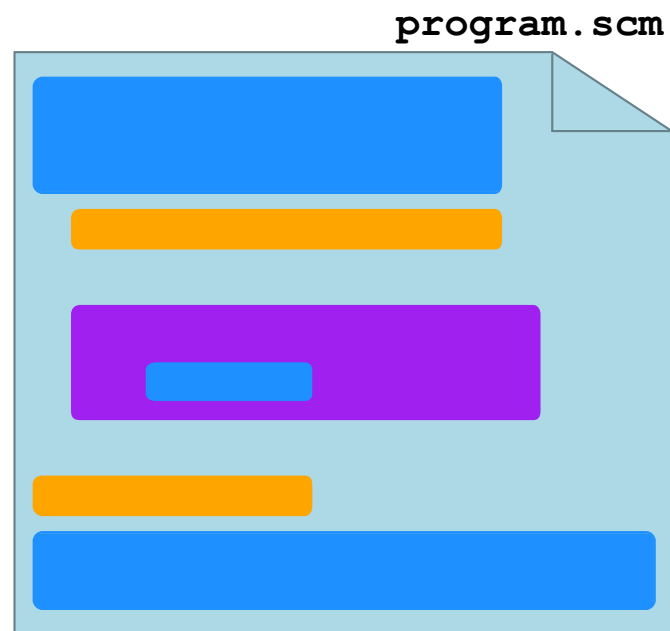
- ◆◆ “Macros that Work Together”

Flatt, Culpepper, Darais, and Findler, JFP’12

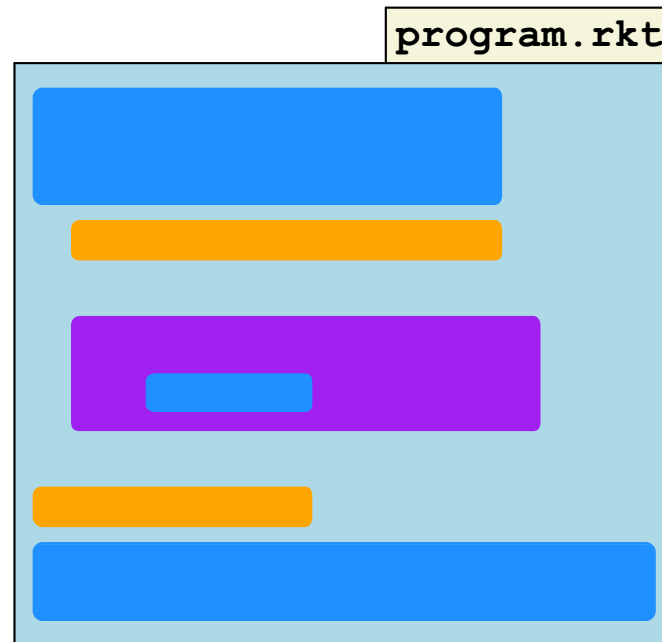
≡ Phases



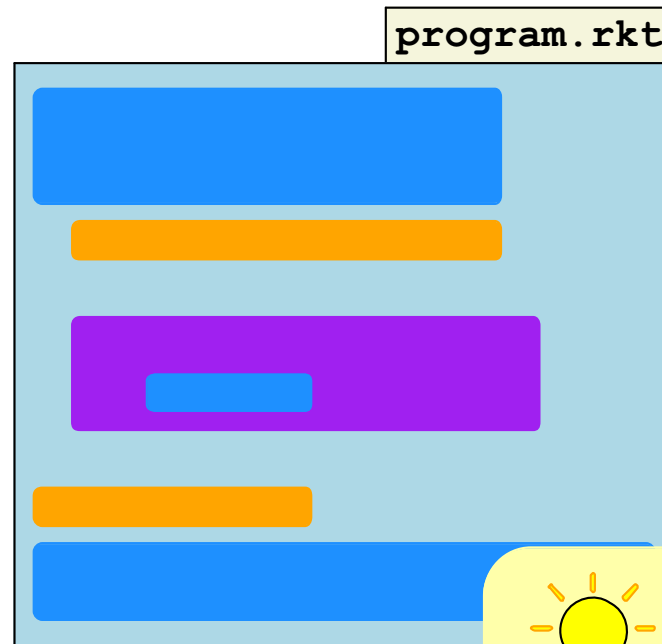
≡ More Phases



Racket Modules Tame Phases

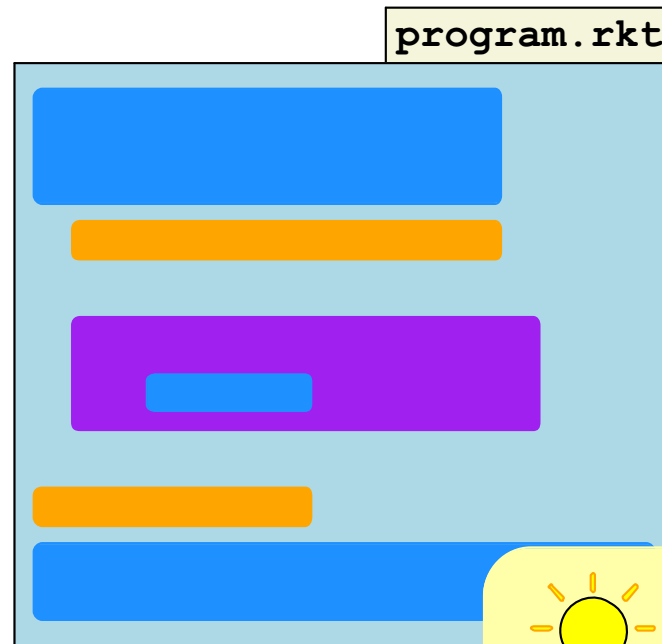


Racket Modules Tame Phases

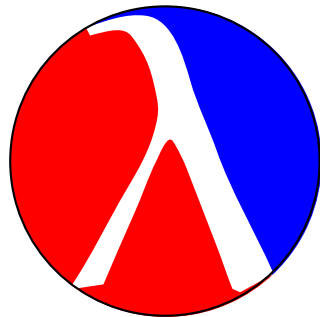


Use Lexical Scope

Racket Modules Tame Phases



Use Lexical Scope



Racket

www.racket-lang.org