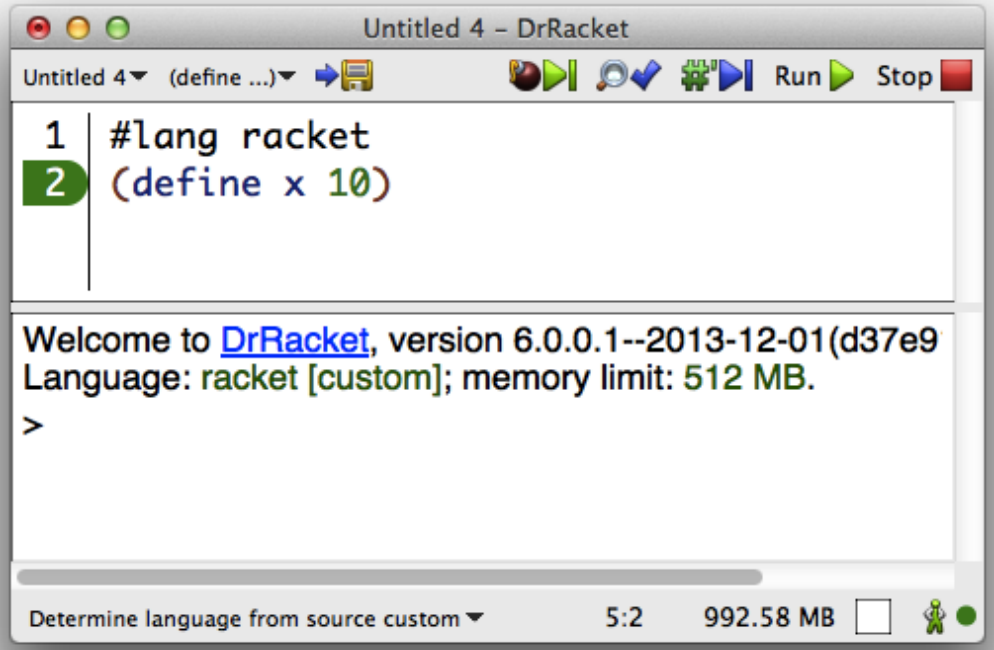
Part I

DrRacket is an Interpreter



eval

The **eval** function is a built-in **parse** plus **interp**:

```
> (eval '(+ 1 2))
3
> (eval '((lambda (x) (+ x x))
          3))
6
> (define op '+)
> (eval `(,op 1 2))
3
> (define name 'x)
> (eval `(define ,name 3))
> x
3
```

Misuse of `eval`

A bad use of `eval`:

```
(define (three-times f-name arg-expr)
  (eval `(,f-name
          (,f-name
           (,f-name ,arg-expr)))))

(three-times 'sqrt '256)
; = (eval '(sqrt (sqrt (sqrt 256))))
```

Instead:

```
(define (three-times f arg)
  (f (f (f arg))))

(three-times sqrt 256)
```

Misuse of `eval`

Another bad use, though less trivial:

```
(struct posn (x y))

(posn-x (posn 1 2))
(posn-y (posn 1 2))

(define (get p field)
  (define getter
    (eval (string->symbol
           (string-append
            "posn-"
            (symbol->string field)))))
  (getter p))

(get (posn 1 2) 'x)
```

Misuse of `eval`

Another bad use, though less trivial:

```
(struct posn (x y))

(posn-x (posn 1 2))
(posn-y (posn 1 2))

(define (get p field)
  (define getter
    (eval (string->symbol
           (string-append
            "posn-"
            (symbol->string field)))))
  (getter p))

(get (posn 1 2) 'x)
```

Alternative: a macro

Misuse of `eval`

A use of `eval` that doesn't work:

```
(define (make-table expr)
  (map (lambda (x)
         (map (lambda (y) (eval expr))
              (list 1 2 3)))
       (list 1 2 3)))

(make-table '(- x y))
```

If you think of `eval` as `parse` plus `interp`, the call to `interp` starts with an empty environment

... sort of

Misuse of `eval`

Yet another a bad use of `eval`:

```
(define (make-table expr)
  (map (lambda (x)
         (map (lambda (y) (eval `(let ([x ,x]
                                       [y ,y])
                                   ,expr)))
              (list 1 2 3)))
       (list 1 2 3)))

(make-table '(- x y))
```

Instead:

```
(define (make-table f)
  (map (lambda (x)
         (map (lambda (y)
                (f x y))
              (list 1 2 3)))
       (list 1 2 3)))

(make-table (lambda (x y) (- x y)))
```

Misuse of `eval`

Fails:

```
(define (three-times f-name arg-expr)
  (eval `(,f-name
          (,f-name
           (,f-name ,arg-expr)))))

(let ([double (lambda (x) (+ x x))])
  (three-times 'double '2))
```

Instead:

```
(define (three-times f arg)
  (f (f (f arg))))

(let ([double (lambda (x) (+ x x))])
  (three-times double 2))
```

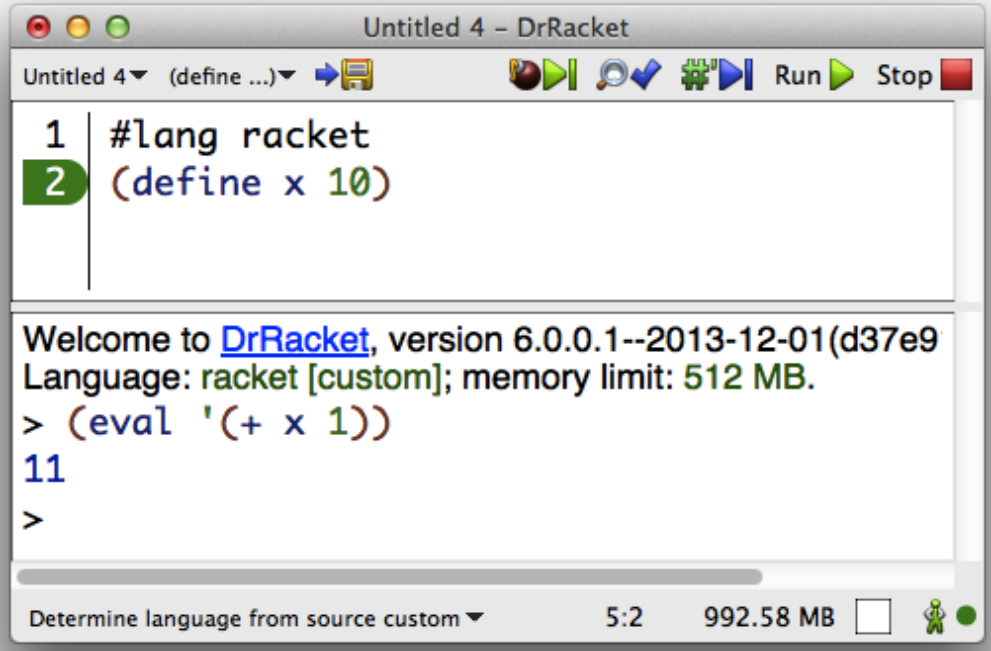
First, do no `eval`

Part 2

Namespaces

`eval` starts with the current ***namespace***

- Interactions area: module's top-level definitions



- During module body: empty

```
#lang racket
(eval '(+ 1 2)) ; fails
```

Namespaces

Using an explicit namespace:

```
#lang racket

(define ns (make-base-namespace))
(eval '(+ 1 2) ns) ; ⇒ 3

(eval '(define x 10) ns)
(eval 'x ns) ; ⇒ 10

(define y 11)
(eval 'y ns) ; ⇒ error
```

Misuse of `eval`

A good use of `eval`:

```
(define (check-handin e)
  ....
  (define ns (make-base-namespace))
  (eval e ns)
  ....)
```

The Problem of `eval`

- First, do no `eval`
- A necessary `eval` ⇒ use an explicit namespace

See also Richards et al., "The Eval that Men Do"

Part 3

Expressive Macros

```
(struct posn (x y))

(posn-x (posn 1 2))
(posn-y (posn 1 2))

(define-syntax-rule (get p field)
  ; Doesn't work:
  ((string->symbol
    (string-append
     "posn-"
     (symbol->string 'field)))
   p))

(get (posn 1 2) x)
```

Expressive Macros

The **`define-syntax`** form binds a compile-time function as a macro:

```
(define-syntax get
  (lambda (stx)
    ....))
```

The **`syntax-rules`** and **`syntax-id-rules`** forms actually produce functions

Expressive Macros

The **`define-syntax`** form binds a compile-time function as a macro:

```
(define-syntax get
  (lambda (stx)
    ....))
```

With

```
(get (posn 1 2) x)
```

the **`stx`** argument is something like

```
'(get (posn 1 2) x)
```

and the result is something like

```
'(posn-x (posn 1 2))
```

Macro Results

x.rkt

```
#lang racket

(define x 1)

(define-syntax-rule (get)
  x)

(provide get)
```

y.rkt

```
#lang racket

(require "x.rkt")

(define x 2)

(get)
```

Macro Results

x.rkt

```
#lang racket

(define x 1)

(define-syntax-rule (get)
  x)

(provide get)
```

y.rkt

```
#lang racket

(require "x.rkt")

(define x 2)

(get)
```

' produces an ***s-expression***

#' produces a ***syntax object***

Part 4

Syntax Objects

- Variable reference:

```
> x
```

- Symbol s-expression:

```
> 'x
```

- Syntax object:

```
> #'x
```

Syntax Objects

- Function call:

```
> (+ 1 2)
```

- List s-expression:

```
> '(+ 1 2)
```

- Syntax object:

```
> #'(+ 1 2)
```

Syntax Objects

Part 5

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")
```

Part 6

Macros

now.rkt

```
#lang racket

(define now
  (current-seconds))

now
```

Compile-Time Expressions

then.rkt

```
#lang racket

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

then
```

Part 7

Compile-Time Imports

then.rkt

```
#lang racket

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

then
```

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
```

Part 8

Macro-Generating Macros

main.rkt

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

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

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

at-this-moment (sleep 1) now
```

Macro-Generating Macros and Imports

recent.rkt

```
#lang racket/base

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

(provide recent-seconds)
```

main.rkt

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

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

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

at-this-moment (sleep 1) now
```

Part 9

Pattern Matching vs. Arbitrary Expressions

```
(define-five x)
x ; ⇒ 5
```

Easy:

```
(define-syntax-rule (define-five name)
  (define name 5))
```

Painful and incomplete:

```
(define-syntax define-five
  (lambda (stx)
    (define id (second (syntax-e stx)))
    #`(define #,id 5)))
```

Pattern Matching plus Arbitrary Expressions

```
(require (for-syntax syntax/parse))

(define-syntax define-five
  (lambda (stx)
    (syntax-parse stx
      [(define-five name)
       (unless (symbol? (syntax-e #'name))
         (raise-syntax-error 'define-now-alias
                             "not an identifier"))
       #'(define name 5)])))
```

- **`syntax-parse`** patterns bind pattern variables
- `#'` templates can use pattern variables

`syntax-rules` and `syntax-parse`

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

is the same as

```
(define-syntax (rotate stx)
  (syntax-parse stx
    [(rotate a) #'(void)]
    [(rotate a b c ...) #'(begin
                            (swap a b)
                            (rotate b c ...))]))
```

Error Checking

```
(define-syntax (rotate stx)
  (syntax-parse stx
    [(rotate a)
     (unless (symbol? (syntax-e #'a))
       (raise-syntax-error 'rotate
                           "not an identifier"))
     #'(void)]
    [(rotate a b c ...)
     (unless (symbol? (syntax-e #'a))
       (raise-syntax-error 'rotate
                           "not an identifier"))
     #'(begin
         (swap a b)
         (rotate b c ...))]))
```

Simpler Error Checking

```
(define-syntax (rotate stx)
  (syntax-parse stx
    [(rotate a:id)
     #'(void)]
    [(rotate a:id b:id c:id ...)
     #'(begin
         (swap a b)
         (rotate b c ...))]))
```

- `id` is a pre-defined ***syntax class***
- ***`sym:class`*** constrains ***`sym`*** to match ***`class`***