# Bugs, crawling all over

## 6.037 - Structure and Interpretation of Computer Programs

Mike Phillips

Massachusetts Institute of Technology

Lecture 5

# Which program is better? Why?

```
(define (prime? n)
 (= n (smallest-divisor n)))

(define (smallest-divisor n)
 (find-divisor n 2))

(define (find-divisor n d)
 (cond ((> (square d) n) n)
       ((divides? d n) d)
       (else (find-divisor n (+ d 1)))))

(define (divides? a b)
  (= (remainder b a) 0))
```

# Which program is better? Why?

```
(define (prime? n)
 (= n (smallest-divisor n)))

(define (smallest-divisor n)
 (find-divisor n 2))

(define (find-divisor n d)
 (cond ((> (square d) n) n)
       ((divides? d n) d)
       (else (find-divisor n (+ d 1)))))

(define (divides? a b)
  (= (remainder b a) 0))
```

---

```
(define (prime? temp1 temp2)
     (cond ((>= temp2 temp1) #t) ((= (remainder
  temp1 temp2) 0) #f) (else (prime? temp1 (+
            temp2 1)))))
```

# What do we mean by “better”?

- Correctness
  - Does the program compute correct results?
  - Programming is about communicating the algorithm to the computer
  - Is it clear what the correct result should be?

# What do we mean by “better”?

- Correctness
  - Does the program compute correct results?
  - Programming is about communicating the algorithm to the computer
  - Is it clear what the correct result should be?
- Clarity
  - Can it be easily read and understood?
  - Programming is also about communicating the algorithm to people!
  - An unreadable program is a useless program
  - Does not benefit from abstraction

# What do we mean by "better"?

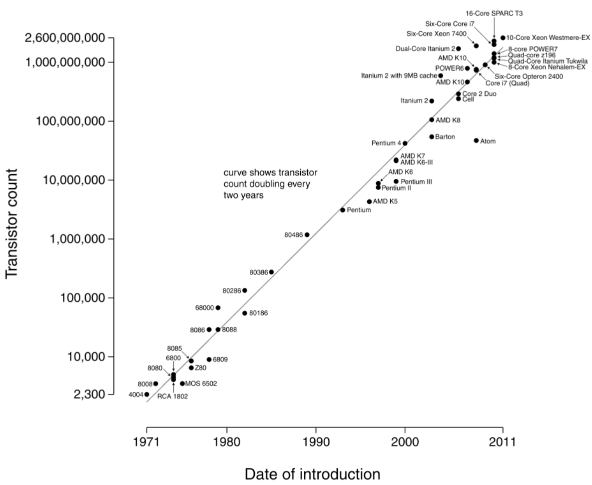
- Correctness
  - Does the program compute correct results?
  - Programming is about communicating the algorithm to the computer
  - Is it clear what the correct result should be?
- Clarity
  - Can it be easily read and understood?
  - Programming is also about communicating the algorithm to people!
  - An unreadable program is a useless program
  - Does not benefit from abstraction
- Maintainability
  - Can it be easily changed?

# What do we mean by "better"?

- Correctness
  - Does the program compute correct results?
  - Programming is about communicating the algorithm to the computer
  - Is it clear what the correct result should be?
- Clarity
  - Can it be easily read and understood?
  - Programming is also about communicating the algorithm to people!
  - An unreadable program is a useless program
  - Does not benefit from abstraction
- Maintainability
  - Can it be easily changed?
- Performance
  - Algorithm choice: order of growth in time & space
  - Optimization: tweaking of constant factors

# Why is optimization last?

Microprocessor Transistor Counts 1971-2011 & Moore's Law



# Making code more readable

```
(define (prime? temp1 temp2)
     (cond ((>= temp2 temp1) #t) ((= (remainder
  temp1 temp2) 0) #f) (else (prime? temp1 (+
           temp2 1)))))
```

# Making code more readable

```
(define (prime? temp1 temp2)
     (cond ((>= temp2 temp1) #t) ((= (remainder
  temp1 temp2) 0) #f) (else (prime? temp1 (+
            temp2 1))))))
```

Use indentation to show structure:

```
(define (prime? temp1 temp2)
 (cond ((>= temp2 temp1) #t)
       ((= (remainder temp1 temp2) 0) #f)
       (else (prime? temp1 (+ temp2 1))))))
```

# Making code more readable

```
(define (prime? temp1 temp2)
 (cond ((>= temp2 temp1) #t)
       ((= (remainder temp1 temp2) 0) #f)
       (else (prime? temp1 (+ temp2 1)))))
```

# Making code more readable

```
(define (prime? temp1 temp2)
 (cond ((>= temp2 temp1) #t)
       ((= (remainder temp1 temp2) 0) #f)
       (else (prime? temp1 (+ temp2 1)))))
```

Don't ask the caller to supply extra arguments for iterative calls:

```
(define (prime? temp1)
 (do-it temp1 2))
(define (do-it temp1 temp2)
 (cond ((>= temp2 temp1) #t)
       ((= (remainder temp1 temp2) 0) #f)
       (else (do-it (+ temp2 1)))))
```

# Making code more readable

```
(define (prime? temp1)
 (do-it temp1 2))
(define (do-it temp1 temp2)
 (cond ((>= temp2 temp1) #t)
       ((= (remainder temp1 temp2) 0) #f)
       (else (do-it (+ temp2 1)))))
```

# Making code more readable

```
(define (prime? temp1)
 (do-it temp1 2))
(define (do-it temp1 temp2)
 (cond ((>= temp2 temp1) #t)
       ((= (remainder temp1 temp2) 0) #f)
       (else (do-it (+ temp2 1)))))
```

Use block structure to hide your helper procedures:

```
(define (prime? temp1)
 (define (do-it temp2)
  (cond ((>= temp2 temp1) #t)
        ((= (remainder temp1 temp2) 0) #f)
        (else (do-it (+ temp2 1)))))
 (do-it 2))
```

# Making code more readable

```
(define (prime? temp1)
 (define (do-it temp2)
  (cond ((>= temp2 temp1) #t)
        ((= (remainder temp1 temp2) 0) #f)
        (else (do-it (+ temp2 1)))))
 (do-it 2))
```

# Making code more readable

```
(define (prime? temp1)
 (define (do-it temp2)
  (cond ((>= temp2 temp1) #t)
        ((= (remainder temp1 temp2) 0) #f)
        (else (do-it (+ temp2 1)))))
 (do-it 2))
```

Choose good names for procedures and variables:

```
(define (prime? n)
 (define (find-divisor d)
  (cond ((>= d n) #t)
        ((= (remainder n d) 0) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))
```

# Making code more readable

```
(define (prime? n)
 (define (find-divisor d)
  (cond ((>= d n) #t)
        ((= (remainder n d) 0) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))
```

# Making code more readable

```
(define (prime? n)
 (define (find-divisor d)
  (cond ((>= d n) #t)
        ((= (remainder n d) 0) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))
```

Find useful common patterns:

```
(define (prime? n)
 (define (find-divisor d)
  (cond ((>= d n) #t)
        ((divides? d n) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))
```

```
(define (divides? d n)
 (= (remainder n d) 0))
```

# Performance?

```
(define (prime? n)
 (define (find-divisor d)
  (cond ((>= d n) #t)
        ((divides? d n) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))

(define (divides? d n)
 (= (remainder n d) 0))
```

# Performance?

```
(define (prime? n)
 (define (find-divisor d)
  (cond ((>= d (sqrt n)) #t)
        ((divides? d n) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))

(define (divides? d n)
 (= (remainder n d) 0))
```

Focus on algorithm improvements (order of growth)

# Performance?

```
(cond ((>= d (sqrt n)) #t)
      ((divides? d n) #f)
      (else (find-divisor (+ d 1)))))
```

# Performance?

```
(cond ((>= d (sqrt n)) #t)
      ((divides? d n) #f)
      (else (find-divisor (+ d 1)))))
```

Is `square` faster than `sqrt`?

```
(cond ((>= (square d) n) #t)
      ((divides? d n) #f)
      (else (find-divisor (+ d 1)))))
```

# Performance?

```
(cond ((>= d (sqrt n)) #t)
      ((divides? d n) #f)
      (else (find-divisor (+ d 1)))))
```

Is `square` faster than `sqrt`?

```
(cond ((>= (square d) n) #t)
      ((divides? d n) #f)
      (else (find-divisor (+ d 1)))))
```

What if we inline `square` and `divides`?

```
(cond ((>= (* d d) n) #t)
      ((= (remainder n d) 0) #f)
      (else (find-divisor (+ d 1)))))
```

# Performance?

```
(cond ((>= d (sqrt n)) #t)
      ((divides? d n) #f)
      (else (find-divisor (+ d 1)))))
```

Is `square` faster than `sqrt`?

```
(cond ((>= (square d) n) #t)
      ((divides? d n) #f)
      (else (find-divisor (+ d 1)))))
```

What if we inline `square` and `divides`?

```
(cond ((>= (* d d) n) #t)
      ((= (remainder n d) 0) #f)
      (else (find-divisor (+ d 1)))))
```

Micro-optimizations are generally useless

# Making code more readable

- Indent code for readability
- Find common, **easily-named** patterns in your code, and pull them out as procedures and data abstractions
  - Makes procedures shorter, able to fit more in your head
- Choose good, descriptive names for procedures and variables
- **Clarity first**, then performance
  - If performance matters, focus on the algorithm first
  - Small optimizations are just constant factors

# Finding prime numbers in a range

```
(define (primes-in-range min max)
 (cond ((> min max) '())
       ((prime? min)
        (cons min
              (primes-in-range (+ 1 min)
                               max)))
       (else (primes-in-range (+ 1 min) max))))
```

# Finding prime numbers in a range

```
(define (primes-in-range min max)
 (cond ((> min max) '())
       ((prime? min)
        (cons min
              (primes-in-range (+ 1 min)
                               max)))
       (else (primes-in-range (+ 1 min) max))))
```

# Finding prime numbers in a range

```
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))

(primes-in-range 0 10)    ; expect (2 3 5 7)
```

# Finding prime numbers in a range

```
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))

(primes-in-range 0 10)    ; expect (2 3 5 7)
```

.

# Finding prime numbers in a range

```
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))

(primes-in-range 0 10)    ; expect (2 3 5 7)
..
```

# Finding prime numbers in a range

```
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))

(primes-in-range 0 10)    ; expect (2 3 5 7)
...
```

# Finding prime numbers in a range

```
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))

(primes-in-range 0 10)    ; expect (2 3 5 7)
.......
```

# Dealing with bugs in your code

- We all write perfect code

# Dealing with bugs in your code

- We all write perfect code
- Clearly never any bugs in it

# Dealing with bugs in your code

- We all write perfect code
- Clearly never any bugs in it
- But *other people's code* has bugs in it

# Dealing with bugs in other people's code

- What do you do when you find a bug in a program?

# Dealing with bugs in other people's code

- What do you do when you find a bug in a program?
- **Write a bug report**

# Dealing with bugs in other people's code

- What do you do when you find a bug in a program?
- **Write a bug report**
- Anyone can do this

# Dealing with bugs in other people's code

- What do you do when you find a bug in a program?
- **Write a bug report**
- Anyone can do this
- A lot of people do it *badly*

# Bad bug reports

```
To: Alyssa P. Hacker
From: Ben Bitdiddle

Your prime-finding program doesn't work.

Please advise.

 - Ben
```

# Questions to ask

- What did you do to cause the bug?

# Questions to ask

- What did you do to cause the bug?
- Is it repeatable?

# Questions to ask

- What did you do to cause the bug?
- Is it repeatable?
- What did you expect it to do?

# Questions to ask

- What did you do to cause the bug?
- Is it repeatable?
- What did you expect it to do?
- What did it actually do?

# What did you do?

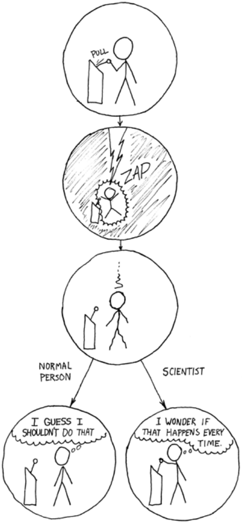
- Precise instructions are important

# What did you do?

- Precise instructions are important
- *Simple* precise instructions are even better

# What did you do?

- Precise instructions are important
- *Simple* precise instructions are even better
- *Repeatability* is key



# What were you expecting?

- State and re-check your assumptions

# What were you expecting?

- State and re-check your assumptions
- Your belief of the right answer may differ from the specification of the author's

# What were you expecting?

- State and re-check your assumptions
- Your belief of the right answer may differ from the specification of the author's

```
; Dividing by zero is always an error
(/ 5 0)
```

# What were you expecting?

- State and re-check your assumptions
- Your belief of the right answer may differ from the specification of the author's

```
; Dividing by zero is always an error
(/ 5 0)  ; error
```

# What were you expecting?

- State and re-check your assumptions
- Your belief of the right answer may differ from the specification of the author's

```
; Dividing by zero is always an error
(/ 5 0)  ; error
(/ 5 0.)
```

# What were you expecting?

- State and re-check your assumptions
- Your belief of the right answer may differ from the specification of the author's

```
; Dividing by zero is always an error
(/ 5 0)  ; error
(/ 5 0.) ; +inf.0
```

# What were you expecting?

- State and re-check your assumptions
- Your belief of the right answer may differ from the specification of the author's

```
; Dividing by zero is always an error
(/ 5 0)  ; error
(/ 5 0.) ; +inf.0
```

- Sometimes the bug is in the user

# What were you expecting?

- State and re-check your assumptions
- Your belief of the right answer may differ from the specification of the author's

```
; Dividing by zero is always an error
(/ 5 0)  ; error
(/ 5 0.) ; +inf.0
```

- Sometimes the bug is in the user
- Read the documentation

# What were you expecting?

- State and re-check your assumptions
- Your belief of the right answer may differ from the specification of the author's

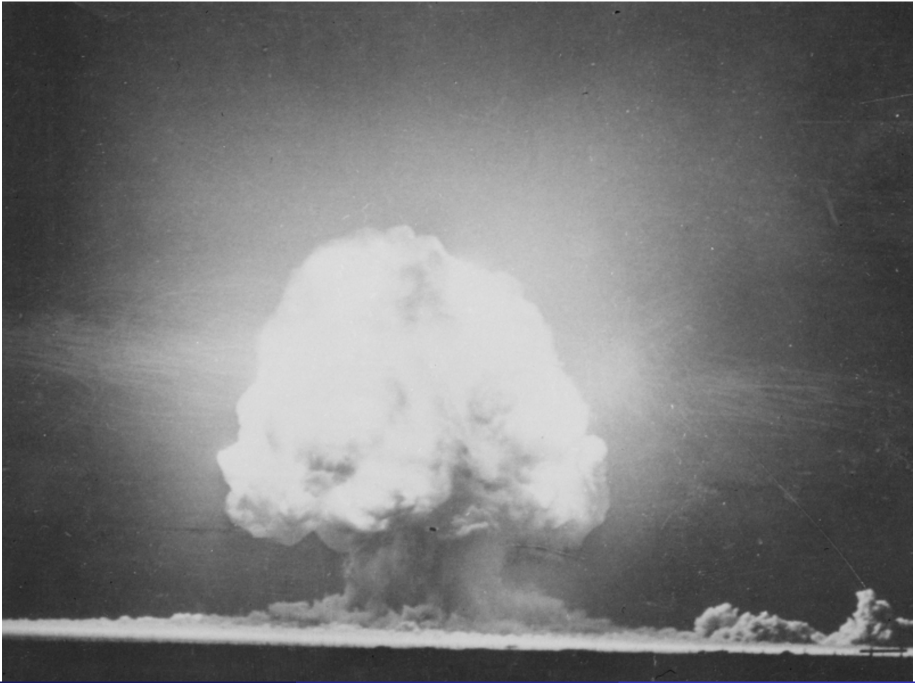
```
; Dividing by zero is always an error
(/ 5 0)  ; error
(/ 5 0.) ; +inf.0
```

- Sometimes the bug is in the user
- Read the documentation
- Leave open the possibility of PEBKAC

# What happened?

# What happened?

"It didn't work"



# The many flavors of failure

- **“Nothing happens”**

# The many flavors of failure

- **“Nothing happens”**
- . . . or is it just very slow?

# The many flavors of failure

- **“Nothing happens”**
- . . . or is it just very slow?
- . . . does it pinwheel?

# The many flavors of failure

- **“Nothing happens”**
- . . . or is it just very slow?
- . . . does it pinwheel?
- . . . does it consume all of your CPU?

# The many flavors of failure

- **“Nothing happens”**
- . . . or is it just very slow?
- . . . does it pinwheel?
- . . . does it consume all of your CPU?
- . . . does it consume all of your memory?

# The many flavors of failure

- **“Nothing happens”**
- . . . or is it just very slow?
- . . . does it pinwheel?
- . . . does it consume all of your CPU?
- . . . does it consume all of your memory?
- **“The answer is not what I expect”**

# The many flavors of failure

- **“Nothing happens”**
- . . . or is it just very slow?
- . . . does it pinwheel?
- . . . does it consume all of your CPU?
- . . . does it consume all of your memory?
- **“The answer is not what I expect”**
- . . . what is the significant way in which it differs from your expectations?

# The many flavors of failure

- **“Nothing happens”**
- . . . or is it just very slow?
- . . . does it pinwheel?
- . . . does it consume all of your CPU?
- . . . does it consume all of your memory?
- **“The answer is not what I expect”**
- . . . what is the significant way in which it differs from your expectations?
- **“It gives an error message”**

# The many flavors of failure

- **"Nothing happens"**
- . . . or is it just very slow?
- . . . does it pinwheel?
- . . . does it consume all of your CPU?
- . . . does it consume all of your memory?
- **"The answer is not what I expect"**
- . . . what is the significant way in which it differs from your expectations?
- **"It gives an error message"**
- . . . and what does that message say?

# The many flavors of failure

- **"Nothing happens"**
- . . . or is it just very slow?
- . . . does it pinwheel?
- . . . does it consume all of your CPU?
- . . . does it consume all of your memory?
- **"The answer is not what I expect"**
- . . . what is the significant way in which it differs from your expectations?
- **"It gives an error message"**
- . . . and what does that message say?
- . . . and is there anything in the error log?

# Better bug reports

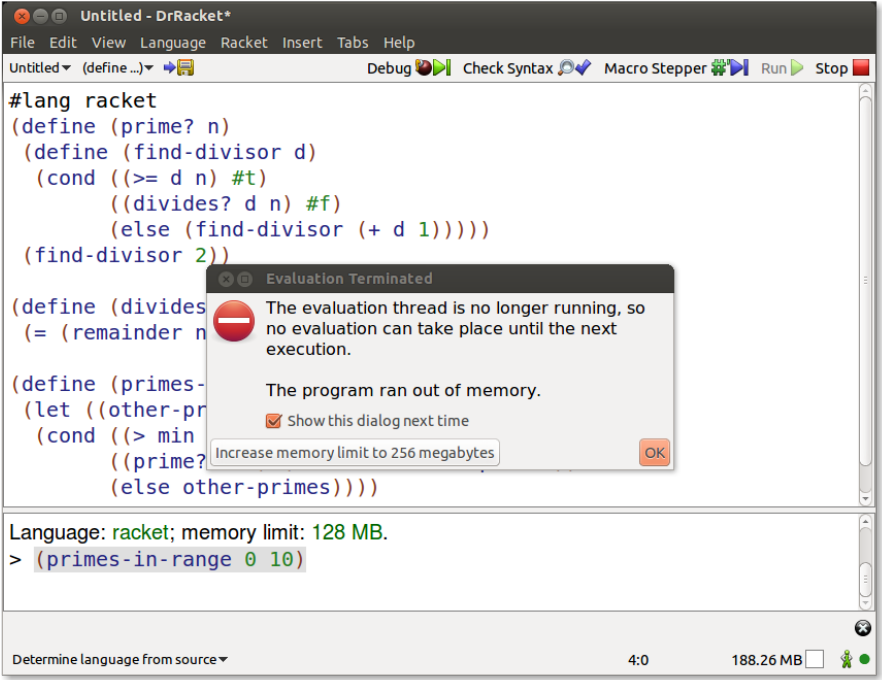
```
To: Alyssa P. Hacker
From: Ben Bitdiddle

primes-in-range appears to never halt.  I ran:

  (primes-in-range 0 10)

...and it just kept going, never outputting anything;
I'd expect it to return (1 2 3 5 7).  I waited for 10
minutes, but it appeared to just make my laptop hot.

 - Ben
```

# Check expectations

- As the author, do we agree that `(primes-in-range 0 10)` should halt?

# Replicate the error

- Can we replicate the error?


Untitled - DrRacket*
File Edit View Language Racket Insert Tabs Help
Untitled (define ...)
Debug Check Syntax Macro Stepper Run Stop
#lang racket
(define (prime? n)
(define (find-divisor d)
(cond ((>= d n) #t)
((divides? d n) #f)
(else (find-divisor (+ d 1)))))
(find-divisor 2))
(define (divides
(= (remainder n
(define (primes-
(let ((other-pr
(cond ((> min
((prime?
(else other-primes))))
Evaluation Terminated
The evaluation thread is no longer running, so no evaluation can take place until the next execution.
The program ran out of memory.
Show this dialog next time
Increase memory limit to 256 megabytes
OK
Language: racket; memory limit: 128 MB.
> (primes-in-range 0 10)
Determine language from source
4:0
188.26 MB

# Replicate the error

- Can we replicate the error?

# Replicate the error

- Can we replicate the error?
- We get a different outcome!

# Replicate the error

- Can we replicate the error?
- We get a different outcome!
- Either this is a <u>different</u> cause, or the <u>same</u> cause with a different symptom

# Replicate the error

- Can we replicate the error?
- We get a different outcome!
- Either this is a <u>different</u> cause, or the <u>same</u> cause with a different symptom
- Always re-check you actually fixed the relevant bug at the end

# Is this the simplest error case?

```
;; Out of memory; test from user
(primes-in-range 0 10)
```

# Is this the simplest error case?

```
;; Out of memory; test from user
(primes-in-range 0 10)

;; Ditto; so 0 not at fault
(primes-in-range 9 10)
```

# Is this the simplest error case?

```
;; Out of memory; test from user
(primes-in-range 0 10)

;; Ditto; so 0 not at fault
(primes-in-range 9 10)

;; Simpler upper bound
(primes-in-range 0 1)
```

# Use abstraction barriers to your advantage

- There appears to be nothing special about `0` or `10`
- All calls to `primes-in-range` run out of memory

# Use abstraction barriers to your advantage

- There appears to be nothing special about `0` or `10`
- All calls to `primes-in-range` run out of memory
- **Divide and conquer** – verify that lower abstractions work
- Abstractions (procedural and structural) are good points to check

# Check the lower abstractions

```
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))
```

# Check the lower abstractions

```
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))

;; Check that our prime? code works!
(prime? 1)
```

# Check the lower abstractions

```
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))

;; Check that our prime? code works!
(prime? 1)    ; -> #t
```

```scheme
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))
```

```
(define (primes-in-range min max)
 (let ((other-primes (primes-in-range (+ 1 min) max)))
  (cond ((> min max) '())
        ((prime? min) (cons min other-primes))
        (else other-primes))))
```

```
(define (primes-in-range min max)
 (if (> min max)
     '()
     (let ((other-primes (primes-in-range (+ 1 min))))
      (if (prime? min)
          (cons min other-primes)
          other-primes))))
```

```
(define (primes-in-range min max)
 (if (> min max)
     '()
     (let ((other-primes (primes-in-range (+ 1 min))))
      (if (prime? min)
          (cons min other-primes)
          other-primes))))

(primes-in-range 0 10)  ;; expect (2 3 5 7)
```

```
(define (primes-in-range min max)
 (if (> min max)
     '()
     (let ((other-primes (primes-in-range (+ 1 min))))
      (if (prime? min)
          (cons min other-primes)
          other-primes))))

(primes-in-range 0 10)  ;; expect (2 3 5 7)
; => (0 1 2 3 4 5 7 9)
```

```
(define (primes-in-range min max)
 (if (> min max)
     '()
     (let ((other-primes (primes-in-range (+ 1 min))))
      (if (prime? min)
          (cons min other-primes)
          other-primes))))

(primes-in-range 0 10)  ;; expect (2 3 5 7)
; => (0 1 2 3 4 5 7 9)
```

# Assumptions

```
(define (prime? n)
 (define (find-divisor d)
  (cond ((>= d (sqrt n)) #t)
        ((divides? d n) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))
```

- Only works on $n \geq 2$

# Assumptions

```
(define (prime? n)
 (define (find-divisor d)
  (cond ((>= d (sqrt n)) #t)
        ((divides? d n) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))
```

- Only works on $n \geq 2$
- Everything has hidden assumptions

# Assumptions

```
(define (prime? n)
 (define (find-divisor d)
  (cond ((>= d (sqrt n)) #t)
        ((divides? d n) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))
```

- Only works on $n \geq 2$
- Everything has hidden assumptions
- Document them!

# Documenting code

- Documentation improves **readability**, allows for **maintenance**, and supports **reuse.**
- Describe input and output
- Any assumptions about inputs or internal state
- Interesting decisions or algorithms

# Documenting code

```
(define (prime? n)
 ; Tests if n is prime (divisible only by 1 and
 ; itself)
 ; n must be >= 2

 ; Test each divisor from 2 to sqrt(n),
 ; since if a divisor > sqrt(n) exists,
 ; there must be another divisor < sqrt(n)
 (define (find-divisor d)
  (cond ((>= d (sqrt n)) #t)
        ((divides? d n) #f)
        (else (find-divisor (+ d 1)))))
 (find-divisor 2))

(define (divides? d n)
 ; Tests if d is a factor of n (i.e. n/d is an integer)
 ; d cannot be 0
 (= (remainder n d) 0))
```

# Not all comments are good

Horrid comment:

```
(define k 2)   ;; set k to 2
```

# Not all comments are good

Horrid comment:

```
(define k 2)  ;; set k to 2
```

Better comment:

```
(define k 2)  ;; 2 is the smallest prime
```

# Not all comments are good

Horrid comment:

```
(define k 2)   ;; set k to 2
```

Better comment:

```
(define k 2)   ;; 2 is the smallest prime
```

Better yet, obviate the need for the comment:

```
(define smallest-prime 2)
```

# The how and why of comments

- Comments should explain “how” or “why”
- “What” is almost never useful

# Make no assumptions?

Use assertions to check assumptions and provide good errors:

```
(define (prime? n)
 ; Tests if n is prime (divisible only by 1 and
 ; itself)
 ; n must be >= 2

 (find-divisor 2))
```

# Make no assumptions?

Use assertions to check assumptions and provide good errors:

```
(define (prime? n)
 ; Tests if n is prime (divisible only by 1 and
 ; itself)
 (if (< n 2)
     (error "prime? requires n >= 2")
     (find-divisor 2)))
```

# Make no assumptions?

Or, better, cover all of your bases:

```
(define (prime? n)
 ; Tests if n is prime (divisible only by 1 and
 ; itself)
 ; n must be >= 2

 (find-divisor 2))
```

# Make no assumptions?

Or, better, cover all of your bases:

```
(define (prime? n)
 ; Tests if n is prime (divisible only by 1 and
 ; itself)
 (if (< n 2)
     #f
     (find-divisor 2)))
```

# Make no assumptions?

All of your bases?

```
(prime? "5")
```

# Make no assumptions?

All of your bases?

```
(prime? "5")
(if (<= "5" 1) #f (find-divisor 2))
```

# Make no assumptions?

All of your bases?

```
(prime? "5")
(if (<= "5" 1) #f (find-divisor 2))
(<= "5" 1)
```

# Make no assumptions?

All of your bases?

```
(prime? "5")
(if (<= "5" 1) #f (find-divisor 2))
(<= "5" 1)
<=: expected argument of type <real number>;
    given "5"
```

# Make no assumptions?

All of your bases?

```
(prime? "5")
(if (<= "5" 1) #f (find-divisor 2))
(<= "5" 1)
<=: expected argument of type <real number>;
    given "5"
```

Include input/output types in a comment

# All better!

```
(primes-in-range 0 10)   ; (expect 2 3 5 7)
```

# All better!

```
(primes-in-range 0 10)  ; (expect 2 3 5 7)
(2 3 4 5 7 9)
```

# All better!

```
(primes-in-range 0 10)   ; (expect 2 3 5 7)
(2 3 4 5 7 9)

(prime? 9)
```

# All better!

```
(primes-in-range 0 10)   ; (expect 2 3 5 7)
(2 3 4 5 7 9)

(prime? 9) ; => #t
```

# How do you know what works?…

- Assume you get a *good* bug report

# How do you know what works?...

- Assume you get a *good* bug report
- With simple, precise instructions that allow you to repeat it

# How do you know what works?...

- Assume you get a *good* bug report
- With simple, precise instructions that allow you to repeat it
- Would be good if we never had this bug again...

# How do you know what works?…

- Assume you get a *good* bug report
- With simple, precise instructions that allow you to repeat it
- Would be good if we never had this bug again…
- Hey, computers are good at executing simple, precise instructions

# How do you know what works?. . .

- Assume you get a *good* bug report
- With simple, precise instructions that allow you to repeat it
- Would be good if we never had this bug again. . .
- Hey, computers are good at executing simple, precise instructions
- **Write a test case** for the bug

# When to write tests

- When should you write tests?

# When to write tests

- When should you write tests?
- **ALL OF THE TIME**.

# When to write tests

- When should you write tests?
- **ALL OF THE TIME**.
- Mostly after a bug is found

# When to write tests

- When should you write tests?
- **ALL OF THE TIME**.
- Mostly after a bug is found
- You can also write tests *before* a feature is added – "test-first methodology"

# When to write tests

- When should you write tests?
- **ALL OF THE TIME.**
- Mostly after a bug is found
- You can also write tests *before* a feature is added – "test-first methodology"
- But at least a tests-sometime methodology is key

# When to write tests

- When should you write tests?
- **ALL OF THE TIME.**
- Mostly after a bug is found
- You can also write tests *before* a feature is added – “test-first methodology”
- But at least a tests-sometime methodology is key
- Test each moving part before you use it elsewhere

# Choosing good test cases

- How do you choose what to test?

# Choosing good test cases

- How do you choose what to test?
- Start with simple cases

# Choosing good test cases

- How do you choose what to test?
- Start with simple cases
- Test the boundaries of your data and recursive cases

# Choosing good test cases

- How do you choose what to test?
- Start with simple cases
- Test the boundaries of your data and recursive cases
- Check a variety of kinds of input (empty list, single element, many)

# Choosing good test cases

# Choosing good test cases

```
(prime? 0)  ;; Test the lower limits
(prime? 1)
(prime? 2)
(prime? 3)
```

# Choosing good test cases

```
(prime? 0)   ;; Test the lower limits
(prime? 1)
(prime? 2)
(prime? 3)
(prime? 7)   ;; Simple should-be-true test
```

# Choosing good test cases

```
(prime? 0)  ;; Test the lower limits
(prime? 1)
(prime? 2)
(prime? 3)
(prime? 7)  ;; Simple should-be-true test
(prime? 10) ;; Simple should-be-false test
```

# Choosing good test cases

```
(prime? 0)  ;; Test the lower limits
(prime? 1)
(prime? 2)
(prime? 3)
(prime? 7)  ;; Simple should-be-true test
(prime? 10) ;; Simple should-be-false test
(prime? 9)  ;; Square numbers should be false
```

# Boundary cases

```
(define (prime? n)
 ; Tests if n is prime (divisible only by 1 and
 ; itself)

 ; Test each divisor from 2 to sqrt(n),
 ; since if a divisor > sqrt(n) exists,
 ; there must be another divisor < sqrt(n)
 (define (find-divisor d)
  (cond ((>= d (sqrt n)) #t)
        ((divides? d n) #f)
        (else (find-divisor (+ d 1)))))
 (if (< n 2)
     #f
     (find-divisor 2)))
```

# Boundary cases

```
(define (prime? n)
 ; Tests if n is prime (divisible only by 1 and
 ; itself)

 ; Test each divisor from 2 to sqrt(n),
 ; since if a divisor > sqrt(n) exists,
 ; there must be another divisor < sqrt(n)
 (define (find-divisor d)
  (cond ((> d (sqrt n)) #t)
        ((divides? d n) #f)
        (else (find-divisor (+ d 1)))))
 (if (< n 2)
     #f
     (find-divisor 2)))
```

# “What will this change break?”

- “Did I actually fix the bug?”

# “What will this change break?”

- “Did I actually fix the bug?”
- Having tests means not needing to know all of the code

# “What will this change break?”

- “Did I actually fix the bug?”
- Having tests means not needing to know all of the code
- Small changes can have far-reaching impacts

# “What will this change break?”

- “Did I actually fix the bug?”
- Having tests means not needing to know all of the code
- Small changes can have far-reaching impacts
- You can keep maybe about 50k LOC in your head at once

# "What will this change break?"

- "Did I actually fix the bug?"
- Having tests means not needing to know all of the code
- Small changes can have far-reaching impacts
- You can keep maybe about 50k LOC in your head at once
- Tests keep the proper functionality on disk, not in your head

# "When did I break this functionality?"

- Tests written now are like debugging in the past

# “When did I break this functionality?”

- Tests written now are like debugging in the past
- Run your test against old versions of your code

# “When did I break this functionality?”

- Tests written now are like debugging in the past
- Run your test against old versions of your code
- If it ever worked, you’ll find what change broke it

# “When did I break this functionality?”

- Tests written now are like debugging in the past
- Run your test against old versions of your code
- If it ever worked, you’ll find what change broke it
- *Bisection* in time is awesome

# "When did I break this functionality?"

- Tests written now are like debugging in the past
- Run your test against old versions of your code
- If it ever worked, you'll find what change broke it
- *Bisection* in time is awesome
- (but only as awesome as your ability to use your version control)

# “Why did I do it that way?”

- Store your code in “version control”

# “Why did I do it that way?”

- Store your code in “version control”
- Git, Subversion, Mercurial, Bazaar, DARCS, CVS, RCS, SCCS,…

# “Why did I do it that way?”

- Store your code in “version control”
- Git, Subversion, Mercurial, ~~Bazaar, DARCS, CVS, RCS, SCCS,...~~

# “Why did I do it that way?”

- Store your code in “version control”
- Git, Subversion, Mercurial, ~~Bazaar, DARCS, CVS, RCS, SCCS,...~~
- Version control lets you group a set of changes into a chunk

# “Why did I do it that way?”

- Store your code in “version control”
- Git, Subversion, Mercurial, ~~Bazaar, DARCS, CVS, RCS, SCCS,...~~
- Version control lets you group a set of changes into a chunk
- And then write a message about the how and why of the change

# “Why did I do it that way?”

- Store your code in “version control”
- Git, Subversion, Mercurial, ~~Bazaar, DARCS, CVS, RCS, SCCS,...~~
- Version control lets you group a set of changes into a chunk
- And then write a message about the how and why of the change

# "Why did I do it that way?"

- Store your code in "version control"
- Git, Subversion, Mercurial, ~~Bazaar, DARCS, CVS, RCS, SCCS,...~~
- Version control lets you group a set of changes into a chunk
- And then write a message about the how and why of the change
- Commit messages are like comments – the intended audience is you in the future

# How to write tests

- Languages have test frameworks
- JUnit (Java), PyUnit (Python), Test::Unit (Ruby), Test::More (Perl)

# How to write tests

- Languages have test frameworks
- JUnit (Java), PyUnit (Python), Test::Unit (Ruby), Test::More (Perl)
- Racket has `RackUnit`

```
(require rackunit)
```

```
(require rackunit)

(check-false (prime? 0) "0 is composite")
(check-false (prime? 1) "1 is composite")
(check-true  (prime? 2) "2 is the smallest prime")
(check-true  (prime? 3) "3 is also prime")
```

```
(require rackunit)

(check-false (prime? 0) "0 is composite")
(check-false (prime? 1) "1 is composite")
(check-true  (prime? 2) "2 is the smallest prime")
(check-true  (prime? 3) "3 is also prime")
(check-true  (prime? 7) "Larger prime")
```

```
(require rackunit)

(check-false (prime? 0) "0 is composite")
(check-false (prime? 1) "1 is composite")
(check-true  (prime? 2) "2 is the smallest prime")
(check-true  (prime? 3) "3 is also prime")
(check-true  (prime? 7) "Larger prime")
(check-false (prime? 10) "Divisible by 2 is composite")
```

```
(require rackunit)

(check-false (prime? 0) "0 is composite")
(check-false (prime? 1) "1 is composite")
(check-true  (prime? 2) "2 is the smallest prime")
(check-true  (prime? 3) "3 is also prime")
(check-true  (prime? 7) "Larger prime")
(check-false (prime? 10) "Divisible by 2 is composite")
(check-false (prime? 9) "Square means composite")
```

# Debugging 101

# Debugging 101

`(display ...)`

# Reasons why display is awesome

- Learn the name of one function, and you can debug in a new language

# Reasons why display is awesome

- Learn the name of one function, and you can debug in a new language
- Faster to implement than learning a new debugger

# Reasons why display is awesome

- Learn the name of one function, and you can debug in a new language
- Faster to implement than learning a new debugger
- Provides written log of code decisions

# Reasons why display is awesome

- Learn the name of one function, and you can debug in a new language
- Faster to implement than learning a new debugger
- Provides written log of code decisions
- Find out which branch the code took?

# Reasons why display is awesome

- Learn the name of one function, and you can debug in a new language
- Faster to implement than learning a new debugger
- Provides written log of code decisions
- Find out which branch the code took?

```
(display "No fallback value found!")
```

# Reasons why display is awesome

- Learn the name of one function, and you can debug in a new language
- Faster to implement than learning a new debugger
- Provides written log of code decisions
- Find out which branch the code took?

  ```
  (display "No fallback value found!")
  ```
- Find out the return value of a function?

# Reasons why display is awesome

- Learn the name of one function, and you can debug in a new language
- Faster to implement than learning a new debugger
- Provides written log of code decisions
- Find out which branch the code took?
  ```
  (display "No fallback value found!")
  ```
- Find out the return value of a function?
  ```
  (display retval)
  ```

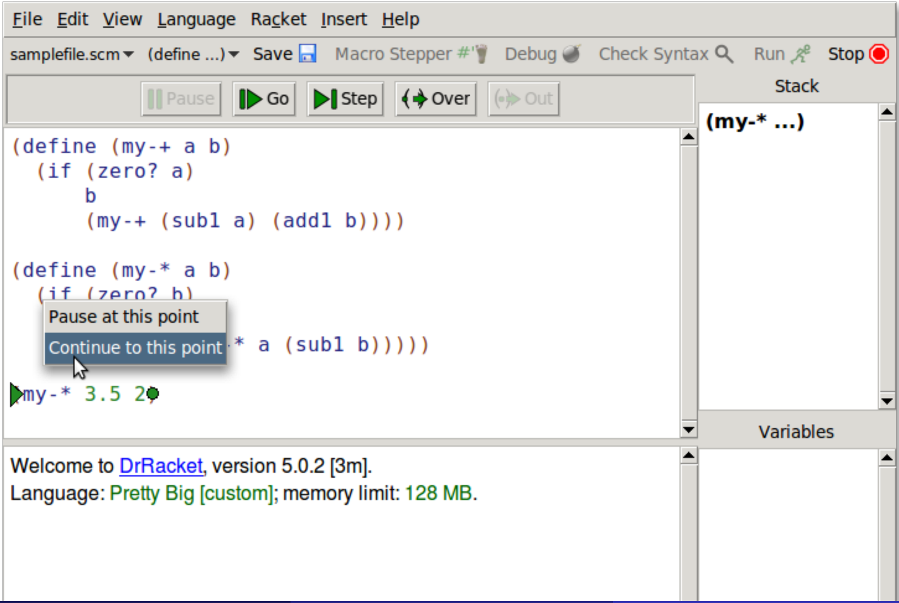
# Reasons why display is awesome

- Learn the name of one function, and you can debug in a new language
- Faster to implement than learning a new debugger
- Provides written log of code decisions
- Find out which branch the code took?
  ```
  (display "No fallback value found!")
  ```
- Find out the return value of a function?
  ```
  (display retval)
  ```
- Find if a function is called?

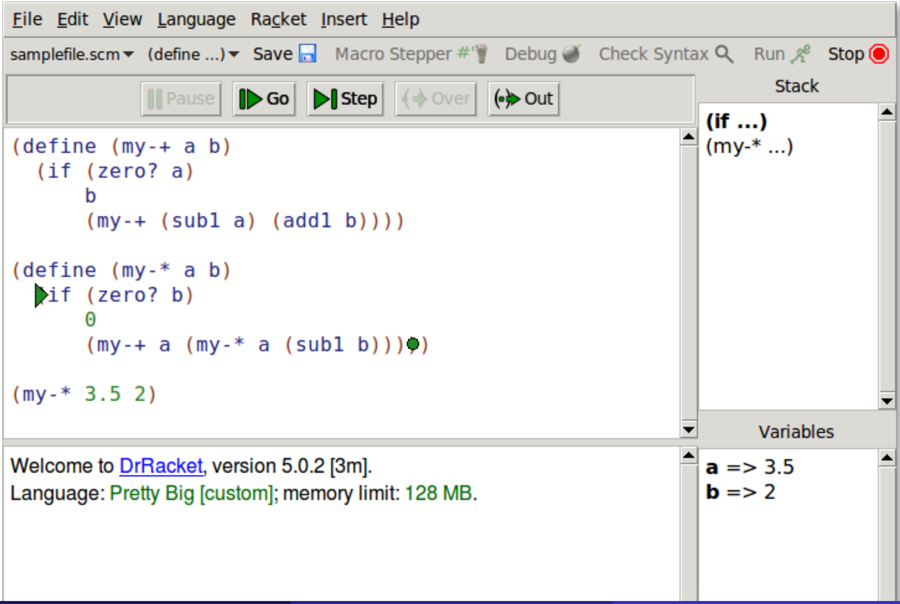
# Reasons why display is awesome

- Learn the name of one function, and you can debug in a new language
- Faster to implement than learning a new debugger
- Provides written log of code decisions
- Find out which branch the code took?
  ```
  (display "No fallback value found!")
  ```
- Find out the return value of a function?
  ```
  (display retval)
  ```
- Find if a function is called?
  ```
  (display "IaIaCthuluFtagn() called!")
  ```

# Interactive debuggers



# Interactive debuggers



# Interactive debugger glossary

Go – Continue until you hit a breakpoint

# Interactive debugger glossary

**Go** – Continue until you hit a breakpoint

**Breakpoint** – Function or line to stop at

# Interactive debugger glossary

**Go** – Continue until you hit a breakpoint

**Breakpoint** – Function or line to stop at

**Watch** – Value or expression to continuously display

# Interactive debugger glossary

**Go** – Continue until you hit a breakpoint

**Breakpoint** – Function or line to stop at

**Watch** – Value or expression to continuously display

**Step** – Proceed to next expression

# Interactive debugger glossary

**Go** – Continue until you hit a breakpoint

**Breakpoint** – Function or line to stop at

**Watch** – Value or expression to continuously display

**Step** – Proceed to next expression

**Step over** – Run until we have the value of the current expression, or hit a breakpoint

# Interactive debugger glossary

**Go** – Continue until you hit a breakpoint

**Breakpoint** – Function or line to stop at

**Watch** – Value or expression to continuously display

**Step** – Proceed to next expression

**Step over** – Run until we have the value of the current expression, or hit a breakpoint

**Out** – Run until we have the value of the surrounding expression, or hit a breakpoint

# Interactive debugger glossary

Go – Continue until you hit a breakpoint

Breakpoint – Function or line to stop at

Watch – Value or expression to continuously display

Step – Proceed to next expression

Step over – Run until we have the value of the current expression, or hit a breakpoint

Out – Run until we have the value of the surrounding expression, or hit a breakpoint

Call stack – Nested list of function calls that we are in; also, “backtrace.”

# Heisenbugs

- Some bugs go away when you examine them

# Heisenbugs

- Some bugs go away when you examine them
- Debugging statements can have side effects

# Heisenbugs

- Some bugs go away when you examine them
- Debugging statements can have side effects

```
(define foo 0)
(define (new-foo) (set! foo (add1 foo)) foo)

(define sum 0)
(display
 (let loop ()
   (if (< foo 10)
       (begin

         (set! sum (+ sum (new-foo)))
         (loop))
      sum)))
```

# Heisenbugs

- Some bugs go away when you examine them
- Debugging statements can have side effects

```
(define foo 0)
(define (new-foo) (set! foo (add1 foo)) foo)

(define sum 0)
(display
 (let loop ()
   (if (< foo 10)
       (begin
         (display (new-foo))(newline)
         (set! sum (+ sum (new-foo)))
         (loop))
      sum)))
```

# Common failure paradigms

- Some error messages tell you immediately what you should be looking for

# Common failure paradigms

- Some error messages tell you immediately what you should be looking for
- `application: not a procedure; expected a procedure that can be applied to arguments, given: 6; arguments were: 7 8`

# Common failure paradigms

- Some error messages tell you immediately what you should be looking for
- `application: not a procedure; expected a procedure that can be applied to arguments, given: 6; arguments were: 7 8`
- `cdr: expects argument of type <pair>; given ()`

# Common failure paradigms

- Some error messages tell you immediately what you should be looking for
- `application: not a procedure; expected a procedure that can be applied to arguments, given: 6; arguments were: 7 8`
- `cdr: expects argument of type <pair>; given ()`
- `cannot reference an identifier before its definition: paramter`

# Common failure paradigms

- Some error messages tell you immediately what you should be looking for
- `application: not a procedure; expected a procedure that can be applied to arguments, given: 6; arguments were: 7 8`
- `cdr: expects argument of type <pair>; given ()`
- `cannot reference an identifier before its definition: paramter`
- Learn them for your given language (ConcurrentModificationException, null pointer dereference, etc)