Which program is better? Why?

(\texttt{define} (prime? n)
\hspace{1em} (= n (smallest-divisor n)))

(\texttt{define} (smallest-divisor n)
\hspace{1em} (\texttt{find-divisor} n 2))

(\texttt{define} (find-divisor n d)
\hspace{1em} (\texttt{cond} ((> (square d) n) n)
\hspace{2em} ((\texttt{divides?} d n) d)
\hspace{2em} (else (\texttt{find-divisor} n (+ d 1)))))

(\texttt{define} (divides? a b)
\hspace{1em} (= (remainder b a) 0))

(\texttt{define} (prime? temp1 temp2)
\hspace{1em} (\texttt{cond} ((\geq temp2 temp1) #t)
\hspace{2em} ((= (remainder temp1 temp2) 0) #f)
\hspace{2em} (else (\texttt{prime?} temp1 (+ temp2 1)))))

What do we mean by “better”?

- \textbf{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?

- \textbf{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

- \textbf{Maintainability}
  - Can it be easily changed?

- \textbf{Performance}
  - Algorithm choice: order of growth in time & space
  - Optimization: tweaking of constant factors

Why is optimization last?

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))))
)

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)))))

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))

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

```scheme
(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?

```scheme
(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))
```

Focus on algorithm improvements (order of growth)

Is `square` faster than `sqrt`?

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

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

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

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)))))

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

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

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  
- 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?

“It didn’t work”

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?

- Can we replicate the error?
- We get a different outcome!
- Either this is a different cause, or the same 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)

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

(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)
; => (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 \( n \geq 2 \)
- 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

```
; Tests if n is prime (divisible only by 1 and ; itself)
; n must be \( \geq 2 \)
; 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))
```

Not all comments are good

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

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:

```scheme
(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:

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

Make no assumptions?

All of your bases?

```scheme
(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)
(2 3 4 5 7 9)

(prime? 9) ; -> #t

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

- **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?
- 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

(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)
  (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?
- 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)

“Which did I break this functionality?”

- 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
“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)
- Racket has RackUnit

(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")

(display ...)
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 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.”
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)))
```