

Part IV:
Mutation

Marco T.
Morazán

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Part IV: Mutation

Marco T. Morazán

Seton Hall University

Outline

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

1 Sharing Values

2 Mutation Sequencing

3 Vectors

4 In-Place Operations

5 The Chicken and the Egg Paradox

6 Epilogue

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Language: Advanced Student Language (ASL)
- All BSL and ISL programs will still work

Sharing Values

set! and begin Expressions

- It is not uncommon for that do not call each other to have to share values

Sharing Values

set! and begin Expressions

- It is not uncommon for that do not call each other to have to share values
- consider keeping track of your quiz grades

Sharing Values

set! and begin Expressions

- It is not uncommon for that do not call each other to have to share values
- consider keeping track of your quiz grades
- ;; A quiz grade (quiz) is a number in [0..100]

```
;; Sample quiz grade
(define Q1 85)
(define Q2 100)
```

Sharing Values

set! and begin Expressions

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- It is not uncommon for that do not call each other to have to share values
- consider keeping track of your quiz grades
- ;;; A quiz grade (quiz) is a number in [0..100]

```
;; Sample quiz grade
(define Q1 85)
(define Q2 100)
```

- ;;; A list of quizzes (loq) is a (listof quiz)

```
;; Sample loq
(define ALOQ1 (list Q1))
(define MYQZS (list Q2 90 Q2))
```

- Design a program to compute a quiz average

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Design a program to compute a quiz average

- ; ; Sample Expressions for quiz-avg

```
(define ALOQ1-VAL (/ (foldl (λ (q r) (+ q r))  
                                0  
                                ALOQ1)  
                           (length ALOQ1)))  
(define AMYQZS-VAL (/ (foldl (λ (q r) (+ q r))  
                                0  
                                MYQZS)  
                           (length MYQZS))))
```

- Design a program to compute a quiz average
- `;; loq → number Purpose: Compute the quiz avg
(define (quiz-avg a-loq)`

- `;; Sample Expressions for quiz-avg
(define ALOQ1-VAL (/ (foldl (λ (q r) (+ q r))
0
ALOQ1)
(length ALOQ1)))
(define AMYQZS-VAL (/ (foldl (λ (q r) (+ q r))
0
MYQZS)
(length MYQZS)))`

- Design a program to compute a quiz average
- ;; loq → number Purpose: Compute the quiz avg
`(define (quiz-avg a-loq)`

- ;; Sample Expressions for quiz-avg
`(define ALOQ1-VAL (/ (foldl (λ (q r) (+ q r))
 0
 ALOQ1)
 (length ALOQ1)))`
`(define MYQZS-VAL (/ (foldl (λ (q r) (+ q r))
 0
 MYQZS)
 (length MYQZS)))`
- ;; Tests using sample computations for quiz-avg
`(check-within (quiz-avg ALOQ1) ALOQ1-VAL 0.01)`
`(check-within (quiz-avg MYQZS) MYQZS-VAL 0.01)`
;; Tests using sample values for quiz-avg
`(check-within (quiz-avg '(80 80 70 75 70)) 75 0.01)`

- Design a program to compute a quiz average
- ; ; loq → number Purpose: Compute the quiz avg
`(define (quiz-avg a-loq)`
 `(/ (foldl (λ (q r) (+ q r)) 0 a-loq)`
 `(length a-loq)))`
- ; ; Sample Expressions for quiz-avg
`(define ALOQ1-VAL (/ (foldl (λ (q r) (+ q r))`
 `0`
 `ALOQ1)`
 `(length ALOQ1)))`
`(define MYQZS-VAL (/ (foldl (λ (q r) (+ q r))`
 `0`
 `MYQZS)`
 `(length MYQZS)))`
- ; ; Tests using sample computations for quiz-avg
`(check-within (quiz-avg ALOQ1) ALOQ1-VAL 0.01)`
`(check-within (quiz-avg MYQZS) MYQZS-VAL 0.01)`
; ; Tests using sample values for quiz-avg
`(check-within (quiz-avg '(80 80 70 75 70)) 75 0.01)`

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- What needs to happen in the middle of the semester when a new quiz grade is earned?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- What needs to happen in the middle of the semester when a new quiz grade is earned?
- The new quiz needs to be added to MYQZS

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- What needs to happen in the middle of the semester when a new quiz grade is earned?
- The new quiz needs to be added to MYQZS
- A function—a mutator—to add the quiz is needed

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- What needs to happen in the middle of the semester when a new quiz grade is earned?
- The new quiz needs to be added to MYQZS
- A function—a mutator—to add the quiz is needed
- (add-quiz! 90)
MYQZS is mutated to:

(list 90 100 90 100)

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- What needs to happen in the middle of the semester when a new quiz grade is earned?
- The new quiz needs to be added to MYQZS
- A function—a mutator—to add the quiz is needed
- (add-quiz! 90)
MYQZS is mutated to:
`(list 90 100 90 100)`
- How is this done?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Variable that are mutated are called *state variables*

- Variable that are mutated are called *state variables*
- ASL provides a `set!`-expression to do so.
- The syntax for a `set!`-expression is:

`expr ::= (set! <var> expr)`

- Variable that are mutated are called *state variables*
- ASL provides a `set!`-expression to do so.
- The syntax for a `set!`-expression is:
`expr ::= (set! <var> expr)`
- `set!` is **not** a function
 - It is part of an expression like `cond` or `local`
 - Cannot be given as input to or returned as the value of a function
 - Evaluating a `set!`-expression returns `(void)`

- Variable that are mutated are called *state variables*
- ASL provides a `set!`-expression to do so.
- The syntax for a `set!`-expression is:
$$\text{expr} ::= (\text{set! } \langle\text{var}\rangle \text{ expr})$$
- `set!` is **not** a function
 - It is part of an expression like `cond` or `local`
 - Cannot be given as input to or returned as the value of a function
 - Evaluating a `set!`-expression returns `(void)`
- The `!` indicates that it is a mutator
 - Danger warning: mutating a variable changes the behavior of functions
 - Convention in this course: mutator names end with `!`

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- The add-quiz! mutator

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- The add-quiz! mutator
- `;; quiz →(void)`
;; Purpose: Add the given quiz grade to MYQZS
- `(define (add-quiz! q)`

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- The add-quiz! mutator
- `;; quiz →(void)`
`;; Purpose: Add the given quiz grade to MYQZS`
- `;; Effect: Add the given quiz grade to the front of MYQZS`
- `(define (add-quiz! q)`
- Mutators require an *effect statement*
 - Describes how a state variable is mutated

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- The add-quiz! mutator
- `;; quiz →(void)`
`;; Purpose: Add the given quiz grade to MYQZS`
- `;; Effect: Add the given quiz grade to the front of MYQZS`
- `(define (add-quiz! q)`
- `(set! MYQZS (cons q MYQZS)))`
- Mutators require an *effect statement*
 - Describes how a state variable is mutated

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(define ADD97 (set! MYQZS (cons 97 MYQZS)))
```

```
(check-expect (add-quiz! 97) (cons 97 MYQZS))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(define ADD97 (set! MYQZS (cons 97 MYQZS)))
```

```
(check-expect (add-quiz! 97) (cons 97 MYQZS))
```

- Does this test make sense?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(define ADD97 (set! MYQZS (cons 97 MYQZS)))
```

```
(check-expect (add-quiz! 97) (cons 97 MYQZS))
```

- Does this test make sense?
- It does not
 - add-quiz! returns (void)

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(define ADD97 (set! MYQZS (cons 97 MYQZS)))
```

```
(check-expect (add-quiz! 97) (cons 97 MYQZS))
```

- Does this test make sense?
- It does not
 - add-quiz! returns (void)
- The correct test:

```
(check-expect MYQZS (cons 97 MYQZS))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Tests for a mutator must illustrate the effect

- Tests for a mutator must illustrate the effect
- The tested expression must call the mutator and return a value to test.

```
(check-expect
  (local [(define dummyvar (add-quiz! 90))]
    MYQZS)
  (list 90 100 90 100))
```

- Tests for a mutator must illustrate the effect
- The tested expression must call the mutator and return a value to test.
- How can this be done?

```
(check-expect
  (local [(define dummyvar (add-quiz! 90))]
    MYQZS)
  (list 90 100 90 100))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Tests for a mutator must illustrate the effect
- The tested expression must call the mutator and return a value to test.
- How can this be done?
- One option is to use a local-expression:

```
(check-expect
  (local [(define dummyvar (add-quiz! 90))]
    MYQZS)
  (list 90 100 90 100))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

A second test may be written as follows:

```
(check-expect
  (local [(define dummyvar (add-quiz! 92))]
    MYQZS)
  (list 92 90 100 90 100))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

A second test may be written as follows:

```
(check-expect
  (local [(define dummyvar (add-quiz! 92))]
    MYQZS)
  (list 92 90 100 90 100))
```

This may be confusing because the expected value contains the 90 from the first test

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

A second test may be written as follows:

```
(check-expect
  (local [(define dummyvar (add-quiz! 92))]
    MYQZS)
  (list 92 90 100 90 100))
```

This may be confusing because the expected value contains the 90 from the first test

The tests may be combined into a single test:

```
(check-expect
  (local [(define dummyvar1 (add-quiz! 90))
          (define dummyvar2 (add-quiz! 92))]
    MYQZS)
  (list 92 90 100 90 100))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Defining local variables to sequence mutations is rather cumbersome
- ASL provides the programmer with more concise syntax for this in the form of a **begin-expression**.
- A **begin-expression** is used to sequence mutations without having to use a **local-expression**.
- The syntax for a **begin-expression** is:
`expr ::= (begin <expression>*)`
- The value of a **begin-expression** is the value of the last expression

```
(check-expect
  (local [(define dummyvar1 (add-quiz! 90))
          (define dummyvar2 (add-quiz! 92))]
    MYQZS)
  (list 92 90 100 90 100))
```

May be rewritten as:

```
(check-expect
  (begin
    (add-quiz! 90)
    (add-quiz! 92)
    MYQZS)
  (list 92 90 100 90 100))
```

- Should more than one test be written?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Should more than one test be written?
 - Yes!

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Should more than one test be written?
 - Yes!
- To simplify this task for each state variable a mutator to initialize the variable is written

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Should more than one test be written?
 - Yes!
- To simplify this task for each state variable a mutator to initialize the variable is written
- State vars require a signature and purpose
- An initializer mutates a state var to a default value

- Should more than one test be written?
 - Yes!
- To simplify this task for each state variable a mutator to initialize the variable is written
- State vars require a signature and purpose
- An initializer mutates a state var to a default value
- `;; loq`
`;; Purpose: Store the quizzes for this semester`
`(define MYQZS 'uninitialized)`

```
;; → (void)
;; Purpose: Initialize MYQZS
;; Effect: MYQZS is initialized to '(100 90 100)
(define (initialize-myqzs)
  (set! MYQZS '(100 90 100)))
```

Having an initializer for a state variable allows us to easily write multiple tests:

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Having an initializer for a state variable allows us to easily write multiple tests:

```
(check-expect
  (begin
    (initialize-myqzs)
    (add-quiz! 90)
    (add-quiz! 92)
    MYQZS)
  (list 92 90 100 90 100))
```

```
(check-expect
  (begin
    (initialize-myqzs)
    (add-quiz! 88)
    MYQZS)
  (list 88 100 90 100))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Although mutation may prove useful for some applications as problem solvers it is important to realize that it use comes at a cost.

- Although mutation may prove useful for some applications as problem solvers it is important to realize that it use comes at a cost.
- What is the result of the following test:

```
(check-expect
  (begin
    (add-quiz! 88)
    MYQZS)
  (begin
    (add-quiz! 88)
    MYQZS))
```

- Although mutation may prove useful for some applications as problem solvers it is important to realize that it use comes at a cost.
- What is the result of the following test:

```
(check-expect
  (begin
    (add-quiz! 88)
    MYQZS)
  (begin
    (add-quiz! 88)
    MYQZS))
```

- It fails! Alas, the value of two identical expressions is not the same.

- Although mutation may prove useful for some applications as problem solvers it is important to realize that it use comes at a cost.
- What is the result of the following test:

```
(check-expect
  (begin
    (add-quiz! 88)
    MYQZS)
  (begin
    (add-quiz! 88)
    MYQZS))
```

- It fails! Alas, the value of two identical expressions is not the same.
- Programs that use mutation lose referential transparency
- The same expression does not always evaluate to the same value

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

The Design Recipe for Mutators

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

The Design Recipe for Mutators

① Problem and Data Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

The Design Recipe for Mutators

- ➊ Problem and Data Analysis
- ➋ Define the State Variables to be Uninitialized

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

The Design Recipe for Mutators

- ➊ Problem and Data Analysis
- ➋ Define the State Variables to be Uninitialized
- ➌ Design and Define Initializers for Each State Variable

The Design Recipe for Mutators

- ➊ Problem and Data Analysis
- ➋ Define the State Variables to be Uninitialized
- ➌ Design and Define Initializers for Each State Variable
- ➍ Perform Problem Analysis for a Mutator

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

The Design Recipe for Mutators

- ➊ Problem and Data Analysis
- ➋ Define the State Variables to be Uninitialized
- ➌ Design and Define Initializers for Each State Variable
- ➍ Perform Problem Analysis for a Mutator
- ➎ Write the Mutator's Signature, Purpose Statement, Effect Statement, and Function Header

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

The Design Recipe for Mutators

- ➊ Problem and Data Analysis
- ➋ Define the State Variables to be Uninitialized
- ➌ Design and Define Initializers for Each State Variable
- ➍ Perform Problem Analysis for a Mutator
- ➎ Write the Mutator's Signature, Purpose Statement, Effect Statement, and Function Header
- ➏ Write Tests that Illustrate the Effects of the Mutator

The Design Recipe for Mutators

- ➊ Problem and Data Analysis
- ➋ Define the State Variables to be Uninitialized
- ➌ Design and Define Initializers for Each State Variable
- ➍ Perform Problem Analysis for a Mutator
- ➎ Write the Mutator's Signature, Purpose Statement, Effect Statement, and Function Header
- ➏ Write Tests that Illustrate the Effects of the Mutator
- ➐ Write the Body of the Mutator

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

The Design Recipe for Mutators

- ➊ Problem and Data Analysis
- ➋ Define the State Variables to be Uninitialized
- ➌ Design and Define Initializers for Each State Variable
- ➍ Perform Problem Analysis for a Mutator
- ➎ Write the Mutator's Signature, Purpose Statement, Effect Statement, and Function Header
- ➏ Write Tests that Illustrate the Effects of the Mutator
- ➐ Write the Body of the Mutator
- ➑ Run the Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Design a program for a bank account
- Three common services: deposit, withdrawal, and balance inquiry

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Problem and Data Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Problem and Data Analysis
- A bank account has a balance which is a nonnegative number

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Problem and Data Analysis
- A bank account has a balance which is a nonnegative number
- Functions to deposit, withdraw, and get the account's balance have no need to call each other, but they all share the balance

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Problem and Data Analysis
- A bank account has a balance which is a nonnegative number
- Functions to deposit, withdraw, and get the account's balance have no need to call each other, but they all share the balance
- Sharing a value but not calling each other means that the balance must be represented using a state variable

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

State variable definition

`;; number \geq 0`

`;; Purpose: Store the balance of the account
(define mybalance 'uninitialized)`

Initializer for mybalance

```
;; number → (void) throws error
;; Purpose: Initialize mybalance
;; Effect: mutate mybalance to be the given number
(define (initialize-mybalance! init-balance)
  (if (or (not (number? init-balance))
          (< init-balance 0))
      (error
       'initialize-mybalance!
       "Cannot be initialized to the given value")
      (set! mybalance init-balance)))
```

- Testing the initializer: error and effect

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Testing the initializer: error and effect
- ; ; Tests for initialize-mybalance!
(check-error

```
(initialize-mybalance! "-200")
"initialize-mybalance!:
Cannot be initialized to the given value")
```

(check-error

```
(initialize-mybalance! -10)
"initialize-mybalance!:
Cannot be initialized to the given value")
```

- Testing the initializer: error and effect
- ; ; Tests for initialize-mybalance!
 - (check-error
 - (initialize-mybalance! "-200")
 - "initialize-mybalance! :
 - Cannot be initialized to the given value")
 - (check-error
 - (initialize-mybalance! -10)
 - "initialize-mybalance! :
 - Cannot be initialized to the given value")
- (check-expect
 - (begin
 - (initialize-mybalance! 75)
 - mybalance)
 - 75)
- (check-within
 - (begin
 - (initialize-mybalance! 43.54)
 - mybalance)
 - 43.54

The Mutator for Deposits

Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Expects a number for the deposit amount
- Guarded mutator: Throw error if input is not a number or is a number ≤ 0
- $\text{mybalance} \leftarrow \text{mybalance} + \text{given amount.}$

The Mutator for Deposits

Signature, Statements, and Function Header

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; number → (void) throws error
;; Purpose: To make a deposit
;; Effect: The given amount is added to mybalance
(define (deposit! amt)
```

The Mutator for Deposits

Error Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(check-error
  (begin
    (initialize-mybalance! 100)
    (deposit! #t))
  "deposit!: Deposit not a positive number")
```

```
(check-error
  (begin
    (initialize-mybalance! 100)
    (deposit! -100))
  "deposit!: A deposit cannot be <= 0")
```

The Mutator for Deposits

Effect Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(check-expect
  (begin
    (initialize-mybalance! 100)
    (deposit! 100)
    mybalance)
  200)
```

```
(check-expect
  (begin
    (initialize-mybalance! 5000)
    (deposit! 700)
    (deposit! 1300)
    mybalance)
  7000)
```

The Mutator for Deposits

Function Body

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(cond
  [(not (number? amt))
   (error 'deposit! "Deposit not a positive number")]
  [(<= amt 0)
   (error 'deposit! "A deposit cannot be <= 0")]
  [else (set! mybalance (+ mybalance amt))]))
```

The Mutator for Withdrawals

Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- To withdraw a given amount several conditions need to be met:
 - ① The given amount must be > 0
 - ② The given amount must be $\leq \text{mybalance}$.
- Guarded mutator that throws an error
- $\text{mybalance} \leftarrow \text{mybalance} - \text{given amount.}$

The Mutator for Withdrawals

Signature, Statements, and Function Header

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; number → (void) throws error
;; Purpose: To make a withdrawal
;; Effect: Subtract given amount from mybalance
(define (withdraw! amt)
```

The Mutator for Withdrawals

Error Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; Tests for withdraw!
(check-error
  (begin
    (initialize-mybalance! 22)
    (withdraw! (make-posn 0 0)))
  "withdraw!: Amount must be positive.")

(check-error
  (begin
    (initialize-mybalance! 50)
    (withdraw! -30))
  "withdraw!: Amount must be a positive.")
```

The Mutator for Withdrawals

Effect Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(check-expect
  (begin
    (initialize-mybalance! 667)
    (withdraw! 500)
    mybalance)
  167)
```

```
(check-expect
  (begin
    (initialize-mybalance! 1500)
    (withdraw! 800)
    (withdraw! 200)
    mybalance)
  500)
```

The Mutator for Withdrawals

Function Body

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(cond
  [(or (not (number? amt)) (<= amt 0))
     (error 'withdraw! "Amount must be a positive.")]
  [(< mybalance amt)
   (error 'withdraw! "Insufficient funds")]
  [else (set! mybalance (- mybalance amt))]))
```

The Observer for Balance Inquiries

```
;; → number
```

```
;; Purpose: Return the current balance
```

```
(define (get-balance) mybalance)
```

```
;; Tests for get-balance
```

```
(check-expect
```

```
  (begin
```

```
    (initialize-mybalance! 250)
```

```
    (get-balance))
```

```
  250)
```

```
(check-expect
```

```
  (begin
```

```
    (initialize-mybalance! 335)
```

```
    (deposit! 665)
```

```
    (withdraw! 200)
```

```
    (get-balance))
```

```
  800)
```

HOMEWORK

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Problems: 3-7

Abstraction Over State Variables

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A single bank account is of limited use
- Banks maintain thousands of accounts

Abstraction Over State Variables

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A single bank account is of limited use
- Banks maintain thousands of accounts
- How are thousands of bank accounts created and maintained?
 - Do we define thousands of state variables using `define`?
 - Do we write thousands of mutators?

Abstraction Over State Variables

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A single bank account is of limited use
- Banks maintain thousands of accounts
- How are thousands of bank accounts created and maintained?
 - Do we define thousands of state variables using `define`?
 - Do we write thousands of mutators?
- Let's implement a second bank account

Abstraction Over State Variables

```
(define mybalance2 'uninitialized)

(define (initialize-mybalance2! init-balance)
  (if (or (not (number? init-balance))
          (< init-balance 0))
      (error 'initialize-mybalance2!
            "Cannot be initialized to the given value")
      (set! mybalance2 init-balance)))

(define (deposit2! amt)           Is this screaming for abstraction?
  (cond [(not (number? amt))
         (error 'deposit2! "A deposit must be a positive number")]
        [(<= amt 0)
         (error 'deposit2! "A deposit cannot be <= 0")]
        [else (set! mybalance2 (+ mybalance2 amt))]))
  The only difference is the state variable shared!

(define (withdraw2! amt)
  (cond [(or (not (number? amt)) (<= amt 0))
         (error 'withdraw! "Amount must be positive.")]
        [(< mybalance2 amt)
         (error 'withdraw! "Insufficient funds")]
        [else (set! mybalance2 (- mybalance2 amt))]))

(define (get-balance2) mybalance2)
```

Abstraction Over State Variables

- Functions for a given state variable must still share it
- A state variable should not be shared with functions for a different state variable

Abstraction Over State Variables

- Functions for a given state variable must still share it
- A state variable should not be shared with functions for a different state variable
- Encapsulation is used to hide state variables.

Abstraction Over State Variables

- Functions for a given state variable must still share it
- A state variable should not be shared with functions for a different state variable
- Encapsulation is used to hide state variables.
- Such a function is called a *class* and implements an *interface*.
- An interface defines the services offered
- Class are used to construct *objects*.

Abstraction Over State Variables

- Functions for a given state variable must still share it
- A state variable should not be shared with functions for a different state variable
- Encapsulation is used to hide state variables.
- Such a function is called a *class* and implements an *interface*.
- An interface defines the services offered
- Class are used to construct *objects*.
- Objects can perform all the services in the interface
- Every object has a *message-passing* function
 - Input: a message
 - Output: Depends on the service requested
 - If a service may be performed without any further input it is performed
 - If further input is required a function that consumes the extra input is returned.
 - An object is a curried function—a function that consumes its input in stages.

Abstraction Over State Variables

Bank Account State Variables and Interface

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Single difference: the state variable that is manipulated.

Abstraction Over State Variables

Bank Account State Variables and Interface

- Single difference: the state variable that is manipulated.
- A bank account is an interface, `ba`, that has a single state variable and offers the following services:

```
'get-balance: number
  'init: number →(void)
  'deposit: number →(void)
  'withdraw: number →(void)
```

Abstraction Over State Variables

Bank Account State Variables and Interface

- Single difference: the state variable that is manipulated.
- A bank account is an interface, `ba`, that has a single state variable and offers the following services:
 - '`get-balance`: number
 - '`init`: number →(void)
 - '`deposit`: number →(void)
 - '`withdraw`: number →(void)
- Simultaneously define interface and message
- A message is either:
 - '`get-balance`
 - '`init`
 - '`deposit`
 - '`withdraw`

Abstraction Over State Variables

Class Template

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
#| ;; --> ba Purpose: To construct a bank account
(define (make-bankaccount)
  (local [;; number ≥ 0 Purpose: Store account balance
         (define mybalance 'uninitialized)

         ;; number → (void) throws error
         ;; Purpose: Initialize mybalance
         ;; Effect: mybalance is mutated to the given number
         (define (initialize-mybalance! init-balance) ...)

         ;; number → (void) throws error
         ;; Purpose: To make a deposit
         ;; Effect: The given amount is added to mybalance
         (define (deposit! amt) ...)

         ;; number → (void) throws error
         ;; Purpose: To make a withdrawal
         ;; Effect: The given amount is subtracted from mybalance
         (define (withdraw! amt) ...))
```

Abstraction Over State Variables

Class Template

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; → number
;; Purpose: Return the current balance
(define (get-balance) ...)

;; message → ba
;; Purpose: To manage bank account services
(define (bank-account-object a-message)
  (cond [(eq? a-message 'get-balance) ...]
        [(eq? a-message 'init) ...]
        [(eq? a-message 'deposit) ...]
        [(eq? a-message 'withdraw) ...]
        [else
         (error 'bank-account-object
                (format "Unknown message received: ~s"
                       a-message)))]))

bank-account-object)
```

Abstraction Over State Variables

Bank Account Message-Passing Function Design

```
;; message → bank account service throws error
;; Purpose: To manage bank account services
(define (bank-account-object a-message)
  (cond [(eq? a-message 'get-balance) mybalance]
        [(eq? a-message 'init) initialize-mybalance!]
        [(eq? a-message 'deposit) deposit!]
        [(eq? a-message 'withdraw) withdraw!]
        [else
         (error
          'bank-account-object
          (format "Unknown message received: ~s"
                 a-message))]))
```

Abstraction Over State Variables

Bank Account Auxiliary Function Design

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; number → (void) throws error Purpose: Initialize mybalance
;; Effect: mybalance is mutated to the given number
(define (initialize-mybalance! init-balance)
  (if (or (not (number? init-balance)) (< init-balance 0))
      (error 'initialize-mybalance! "Cannot be initialized to the given value")
      (set! mybalance init-balance)))

;; number → (void) throws error Purpose: To make a deposit
;; Effect: The given amount is added to mybalance
(define (deposit! amt)
  (cond [(not (number? amt))
         (error 'deposit! "A deposit must be a positive number")]
        [(<= amt 0)
         (error 'deposit! "A deposit cannot be <= 0")]
        [else (set! mybalance (+ mybalance amt))]))

;; number → (void) throws error
;; Purpose: To make a withdrawal
;; Effect: The given amount is subtracted from mybalance
(define (withdraw! amt)
  (cond [(or (not (number? amt)) (<= amt 0))
         (error 'withdraw! "The amount must be a positive number.")]
        [(< mybalance amt)
         (error 'withdraw! "Insufficient funds")]
        [else (set! mybalance (- mybalance amt))]))
```

Abstraction Over State Variables

Bank Account Auxiliary Function Design

- We create bank accounts as follows:

```
; ; Sample ba
(define acct1 (make-bankaccount))
(define acct2 (make-bankaccount))
```

Abstraction Over State Variables

Bank Account Auxiliary Function Design

- We create bank accounts as follows:

```
; ; Sample ba
(define acct1 (make-bankaccount))
(define acct2 (make-bankaccount))
```

- Bank accounts are initialized using message passing :

```
((acct1 'init) 500)
((acct2 'init) 200)
```

Abstraction Over State Variables

Bank Account Auxiliary Function Design

- We create bank accounts as follows:

```
; ; Sample ba
(define acct1 (make-bankaccount))
(define acct2 (make-bankaccount))
```

- Bank accounts are initialized using message passing :

```
((acct1 'init) 500)
((acct2 'init) 200)
```

- Balance inquiries are done as follows:

```
(acct1 'get-balance)
(acct2 'get-balance)
```

Abstraction Over State Variables

Bank Account Auxiliary Function Design

- We create bank accounts as follows:

```
; ; Sample ba
(define acct1 (make-bankaccount))
(define acct2 (make-bankaccount))
```

- Bank accounts are initialized using message passing :

```
((acct1 'init) 500)
((acct2 'init) 200)
```

- Balance inquiries are done as follows:

```
(acct1 'get-balance)
(acct2 'get-balance)
```

- Deposits are done as follows:

```
((acct1 'deposit) 100)
((acct2 'deposit) 525)
```

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

- Any user will find the message-passing cumbersome
- Why would any user want to learn the messages?

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

- Any user will find the message-passing cumbersome
- Why would any user want to learn the messages?
- We should not expose the implementation
- To hide these implementation details write a *wrapper function* for each service m
- A wrapper function hides the implementation details and presents a user-friendly interface.

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

- Any user will find the message-passing cumbersome
- Why would any user want to learn the messages?
- We should not expose the implementation
- To hide these implementation details write a *wrapper function* for each service m
- A wrapper function hides the implementation details and presents a user-friendly interface.
- The user provides the object to manipulate and extra inputs
- The wrapper function sends the message for the proper service and, if necessary, provides the extra inputs to the returned function.
- An interface's implementation is tested by writing tests for the wrapper functions.

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
; ; ba → number
;; Purpose: To get the balance of the given bank account
(define (get-balance ba)
  (ba 'get-balance))
```

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; ba → number
;; Purpose: To get the balance of the given bank account
(define (get-balance ba)
  (ba 'get-balance))

;; Tests using sample computations for get-balance
  (check-expect (begin
                  (initialize! acct1 500)
                  (get-balance acct1))
                500)

  (check-expect (begin
                  (initialize! acct2 200)
                  (get-balance acct2))
                200)
```

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; ba number → (void)
;; Purpose: To initialize the given bank account with the
;;           given balance
;; Effect: Change the given ba's balance to the given number
(define (initialize! ba amt)
  ((ba 'init) amt))
```

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; ba number → (void)
;; Purpose: To initialize the given bank account with the
;;           given balance
;; Effect: Change the given ba's balance to the given number
(define (initialize! ba amt)
  ((ba 'init) amt))

;; Tests using sample computations for initialize
(check-error
  (initialize! acct1 -78)
  "initialize-mybalance!: Balance cannot be initialized to the given val"
(check-error
  (initialize! acct1 #t)
  "initialize-mybalance!: Balance cannot be initialized to the given val"
(check-expect (begin
                (initialize! acct1 3400)
                (get-balance acct1))
              3400)
  (check-expect (begin
                  (initialize! acct2 788)
                  (get-balance acct2))
                788))
```

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

```
;; ba number → (void)
;; Purpose: To deposit the given amount in the given
;;           bank account
;; Effect: Increase the balance of the given ba by the
;;           given amount
(define (make-deposit! ba amt)
  ((ba 'deposit) amt))
```

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

```
;; ba number → (void)
;; Purpose: To deposit the given amount in the given
;;           bank account
;; Effect: Increase the balance of the given ba by the
;;           given amount
(define (make-deposit! ba amt)
  ((ba 'deposit) amt))

;; Tests using sample computations for make-deposit
(check-error
  (begin
    (initialize! acct1 500)
    (make-deposit! acct1 'a))
  "deposit!: A deposit must be a positive number")
(check-error (begin
  (initialize! acct1 500)
  (make-deposit! acct1 -300))
  "deposit!: A deposit cannot be <= 0")
(check-expect (begin
  (initialize! acct1 500)
  (make-deposit! acct1 300)
  (get-balance acct1))
  800)
```

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

```
;; bankaccount number → (void)
;; Purpose: To withdraw the given amount from the given
;;           bank account
;; Effect: Decrease the balance of the given ba by the
;;           given amount
(define (make-withdrawal! ba amt) ((ba 'withdraw) amt))
```

Abstraction Over State Variables

Bank Account Wrapper Functions and Tests

```
;; bankaccount number → (void)
;; Purpose: To withdraw the given amount from the given
;;           bank account
;; Effect: Decrease the balance of the given ba by the
;;           given amount
(define (make-withdrawal! ba amt) ((ba 'withdraw) amt))

;; Tests using sample computations for make-withdrawal
(check-error
  (begin
    (initialize! acct1 10)
    (make-withdrawal! acct1 'a))
  "withdraw!: The amount must be a positive number.")
(check-error (begin
  (initialize! acct1 750)
  (make-withdrawal! acct1 800))
  "withdraw!: Insufficient funds")
(check-expect (begin
  (initialize! acct1 500)
  (make-withdrawal! acct1 250)
  (get-balance acct1))
  250)
```

Abstraction Over State Variables

Design Recipe for Interfaces

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- ① Identify the values that must be stored and the services that must be provided.

Abstraction Over State Variables

Design Recipe for Interfaces

- ① Identify the values that must be stored and the services that must be provided.
- ② Develop an interface data definition and a data definition for messages.

Abstraction Over State Variables

Design Recipe for Interfaces

- ① Identify the values that must be stored and the services that must be provided.
- ② Develop an interface data definition and a data definition for messages.
- ③ Develop a function template for the class that consumes the values that must be stored and whose body is a local-expression returning the message-processing function.

Abstraction Over State Variables

Design Recipe for Interfaces

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- ① Identify the values that must be stored and the services that must be provided.
- ② Develop an interface data definition and a data definition for messages.
- ③ Develop a function template for the class that consumes the values that must be stored and whose body is a local-expression returning the message-processing function.
- ④ Specialize the signature, purpose, class header, and message-processing function.

Abstraction Over State Variables

Design Recipe for Interfaces

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- ① Identify the values that must be stored and the services that must be provided.
- ② Develop an interface data definition and a data definition for messages.
- ③ Develop a function template for the class that consumes the values that must be stored and whose body is a local-expression returning the message-processing function.
- ④ Specialize the signature, purpose, class header, and message-processing function.
- ⑤ Write and make local the auxiliary functions needed by the message-passing function.

Abstraction Over State Variables

Design Recipe for Interfaces

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- ➊ Identify the values that must be stored and the services that must be provided.
- ➋ Develop an interface data definition and a data definition for messages.
- ➌ Develop a function template for the class that consumes the values that must be stored and whose body is a local-expression returning the message-processing function.
- ➍ Specialize the signature, purpose, class header, and message-processing function.
- ➎ Write and make local the auxiliary functions needed by the message-passing function.
- ➏ Write and test a wrapper function for each service.

Abstraction Over State Variables

Design Recipe for Interfaces

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

HOMEWORK: Problems 10, 11

Mutation and Structures

- ASL structures are mutable.

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation and Structures

- ASL structures are mutable.
- When you define a structure a constructor, an observer for each field, and a mutator for each field is created.

```
(define-struct student (name gpa credits))  
make-student  
student-name  
student-gpa  
student-credits
```

Mutation and Structures

- ASL structures are mutable.
- When you define a structure a constructor, an observer for each field, and a mutator for each field is created.

```
(define-struct student (name gpa credits))  
make-student  
student-name  
student-gpa  
student-credits
```

- ASL also creates a mutator for each field:

```
set-student-name!  
set-student-gpa!  
set-student-credits!
```

- These mutators take as input a student and the new field value.
- Mutators do not create a new student

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(define S1 (make-student "Dr. Strange" 4.0 120))  
(define S2 (make-student "Tony Stark" 4.0 160))
```

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(define S1 (make-student "Dr. Strange" 4.0 120))
(define S2 (make-student "Tony Stark" 4.0 160))

(set-student-credits! S1 150)

(check-expect (student-credits S1) 150)
(check-expect (student-credits S2) 160)
```

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Why is this important or useful?

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Why is this important or useful?
- Memory is allocated when a constructor is used

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Why is this important or useful?
- Memory is allocated when a constructor is used
- A program that allocates too much memory may be inefficient or the computer may run out of allocatable memory.

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Why is this important or useful?
- Memory is allocated when a constructor is used
- A program that allocates too much memory may be inefficient or the computer may run out of allocatable memory.
- A universe program may allocate a lot of memory when a new world is created after each clock tick
- To avoid such allocations the given world may be mutated instead of creating a new world.

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(define H 650)
(define W 600)
(define E-SCENE (empty-scene W H))

;; A world is a structure, (make-world posn number color), with
;; a posn, a radius, and a color representing a circle.
(define-struct world (p radius color))

;; Sample worlds
(define INIT-WORLD (make-world (make-posn (/ W 2) (/ H 2)) 5 'green))
(define WORLD2 (make-world (make-posn 20 20) 10 'blue))
(define WORLD3 (make-world (make-posn 400 300) 40 'yellow))
(define WORLD4 (make-world (make-posn 200 400) 60 'red))
(define WORLD5 (make-world (make-posn 1 1) 400 'red))
```

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; world arrow image      Purpose: Draw the circle for the given world
(define (draw-world w)
  (place-image (circle (world-radius w) 'solid (world-color w))
               (posn-x (world-p w))
               (posn-y (world-p w))
               E-SCENE))

;; Sample expressions for draw world
(define WIMG1 (place-image (circle (world-radius INIT-WORLD)
                                     'solid (world-color
                                             INIT-WORLD))
                            (posn-x (world-p INIT-WORLD))
                            (posn-y (world-p INIT-WORLD))
                            E-SCENE))

(define WIMG2 (place-image (circle (world-radius WORLD2)
                                     'solid
                                     (world-color WORLD2))
                            (posn-x (world-p WORLD2))
                            (posn-y (world-p WORLD2))
                            E-SCENE))
```

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

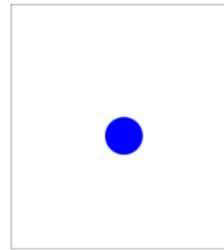
```
; ; Tests using sample computations for draw-world
(check-expect (draw-world INIT-WORLD) WIMG1)
(check-expect (draw-world WORLD2) WIMG2)
```

```
; ; Tests using sample values for draw-world
(check-expect (draw-world (make-world (make-posn 60 60) 30 'yellow))
```



)

```
(check-expect (draw-world (make-world (make-posn 300 350) 50 'blue)))
```



)

Mutation and Structures

```
;; world → world
;; Purpose: To process a clock tick.
;; Effect: The given world's radius is incremented by 1
(define (process-tick w)
  (begin
    (set-world-radius! w (add1 (world-radius w)))
    w))

;; Sample expressions for process-tick
(define PT1 (begin
  (set-world-radius! WORLD3 (add1 (world-radius WORLD3)))
  WORLD3))

(define PT2 (begin
  (set-world-radius! WORLD4 (add1 (world-radius WORLD4)))
  WORLD4))
;; Tests using sample computations for process-tick
(check-expect (process-tick WORLD3) PT1)
(check-expect (process-tick WORLD4) PT2)
;; Tests using sample values for process-tick
(check-expect (process-tick (make-world (make-posn 10 40) 5 'yellow))
  (make-world (make-posn 10 40) 6 'yellow))
```

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
; world arrow Boolean
;; Purpose: To determine if the simulation is over
(define (done? w) (>= (world-radius w) (/ W 2)))

;; Sample expressions for done?
(define DONE1 (>= (world-radius INIT-WORLD) (/ W 2)))
(define DONE2 (>= (world-radius WORLD5)
                  (/ W 2)))

;; Tests using sample computations for done?
(check-expect (done? INIT-WORLD) #false)
(check-expect (done? WORLD5) #true)
;; Tests using sample values for done?
(check-expect (done? (make-world (make-posn 0 0) 5 'green)) #false)
(check-expect (done? (make-world (make-posn 6 2) 350 'pink)) #true)

;; arrow world
;; Purpose: To run the simulation
(define (run)
  (big-bang
    INIT-WORLD
    (on-draw draw-world)
    (on-tick process-tick)
    (stop-when done?)))
```

Mutation and Structures

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

HOMEWORK: 12, 13

The Concept of Equality

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- What does it mean for two values to be equal?

The Concept of Equality

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- What does it mean for two values to be equal?
- Everyone agrees that two values are equal if they are the same
- This is called *extensional equality*.

The Concept of Equality

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- What does it mean for two values to be equal?
- Everyone agrees that two values are equal if they are the same
- This is called *extensional equality*.
- This is not the only definition of equality that exists in the presence of mutation.

The Concept of Equality

- Consider

```
(define P1 (make-posn 10 15))  
(define P2 (make-posn 10 15))  
(define P3 P1)
```

- Are these three posns equal?

The Concept of Equality

- Consider

```
(define P1 (make-posn 10 15))  
(define P2 (make-posn 10 15))  
(define P3 P1)
```

- Are these three posns equal?
- Clearly, there is extensional equality.

The Concept of Equality

- Consider

```
(define P1 (make-posn 10 15))  
(define P2 (make-posn 10 15))  
(define P3 P1)
```

- Are these three posns equal?
- Clearly, there is extensional equality.
- Are P1, P2, and P3 the same?

The Concept of Equality

- Consider

```
(define P1 (make-posn 10 15))  
(define P2 (make-posn 10 15))  
(define P3 P1)
```

- Are these three posns equal?
- Clearly, there is extensional equality.
- Are P1, P2, and P3 the same?
- Run the following code:

```
(set-posn-x! P1 0)
```

```
(check-expect P1 P2)  
(check-expect P1 P3)  
(check-expect P3 P2)
```

The Concept of Equality

- Consider

```
(define P1 (make-posn 10 15))  
(define P2 (make-posn 10 15))  
(define P3 P1)
```

- Are these three posns equal?
- Clearly, there is extensional equality.
- Are P1, P2, and P3 the same?
- Run the following code:

```
(set-posn-x! P1 0)
```

```
(check-expect P1 P2)  
(check-expect P1 P3)  
(check-expect P3 P2)
```

- The first and third test fail
- In some sense our three posns are not equal or the same

The Concept of Equality

```
(define P1 (make-posn 10 15))  
(define P2 (make-posn 10 15))  
(define P3 P1)
```

What is going on?

The Chicken
and the Egg
Paradox

Epilogue

The Concept of Equality

Sharing Values

Mutation
Sequencing

Vectors

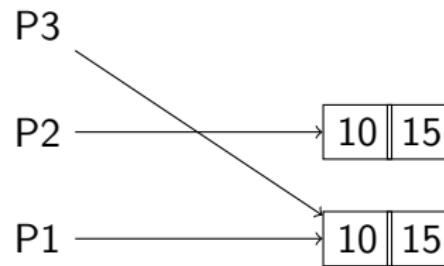
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(define P1 (make-posn 10 15))  
(define P2 (make-posn 10 15))  
(define P3 P1)
```

What is going on?



P1 and P3 refer to the same memory location while P2 refers to a different memory location

The Concept of Equality

Sharing Values

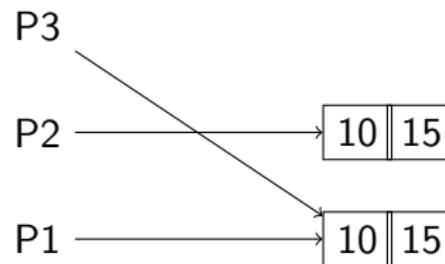
Mutation
Sequencing

Vectors

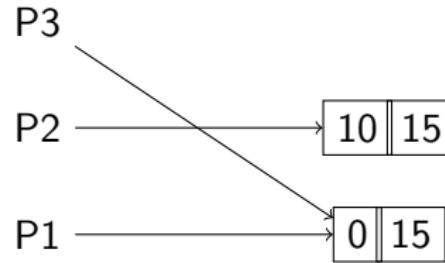
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue



When `(set-posn-x! P1 0)` is evaluated the picture changes to:



The Concept of Equality

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Only P1 is mutated but the value of P3 also changes
- This occurs because P1 and P3 are the same value in memory
- This is called *intensional equality*. Two values have intensional equality when they refer to the same values in memory
- ASL offers two predicates to test for equality.
 - To test for extensional equality use `equal?`
 - To test for intensional equality use `eq?`
- A major pitfall of *imperative programming* (programming using mutation)
- A common source of bugs in modern software.

The Concept of Equality

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

HOMEWORK: 14, 15

Mutation Sequencing

- We studied the design of tail-recursive functions that do not have delayed operations
- Values in one recursive call are not needed in the future of the computation after a recursive call is made.

Mutation Sequencing

- We studied the design of tail-recursive functions that do not have delayed operations
- Values in one recursive call are not needed in the future of the computation after a recursive call is made.

- ```
;; natnum → natnum
;; Purpose: Compute the factorial of the given natural number
(define (fact2 n)
 (local [;; natnum natnum → natnum
 ;; Purpose: Compute the factorial of the first
 ;; given natural number
 ;; Accumulator invariant
 ;; accum = product of the natnums in [k+1..n]
 (define (fact-accum k accum)
 (if (= k 0)
 accum
 (fact-accum (sub1 k) (* k accum))))]
 (fact-accum n 1)))
;; Tests using sample values for fact2
(check-expect (fact2 0) 1)
(check-expect (fact2 5) 120)
(check-expect (fact2 10) 3628800)
```

# Mutation Sequencing

- ```
(fact2 6) = (fact-accum 6 1)
              = (fact-accum 5 6)
              = (fact-accum 4 30)
              = (fact-accum 3 120)
              = (fact-accum 2 360)
              = (fact-accum 1 720)
              = (fact-accum 0 720)
              = 720
```
- The values of `k` and `accum` for one call are not needed in future calls

Mutation Sequencing

- ```
(fact2 6) = (fact-accum 6 1)
 = (fact-accum 5 6)
 = (fact-accum 4 30)
 = (fact-accum 3 120)
 = (fact-accum 2 360)
 = (fact-accum 1 720)
 = (fact-accum 0 720)
 = 720
```

- The values of `k` and `accum` for one call are not needed in future calls
- Suggests that state variables may be used to represent `k` and `accum`. At each step
  - `k` is mutated to be decremented by 1
  - `accum` is mutated to be the product of itself and the value of `k`

# Mutation Sequencing

- ```
(fact2 6) = (fact-accum 6 1)
              = (fact-accum 5 6)
              = (fact-accum 4 30)
              = (fact-accum 3 120)
              = (fact-accum 2 360)
              = (fact-accum 1 720)
              = (fact-accum 0 720)
              = 720
```
- The values of `k` and `accum` for one call are not needed in future calls
- Suggests that state variables may be used to represent `k` and `accum`. At each step
 - `k` is mutated to be decremented by 1
 - `accum` is mutated to be the product of itself and the value of `k`
- $$\begin{array}{ccccccccc} k & = & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ \text{accum} & = & 1 & 6 & 30 & 120 & 360 & 720 & \end{array}$$

Mutation Sequencing

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
; ; natnum → natnum
;; Purpose: To compute n!
(define (fact2-v1 n)
  (local
    [;; natnum
     ;; Purpose: Next accum factor
     (define k (void))]
```

```
; ; natnum → natnum
;; Purpose: To compute n!
(define (fact2-v2 n)
  (local
    [;; natnum
     ;; Purpose: Next accum factor
     (define k (void))])
```

Mutation Sequencing

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
; ; natnum → natnum
;; Purpose: To compute n!
(define (fact2-v1 n)
  (local
    [;; natnum
     ;; Purpose: Next accum factor
     (define k (void))

    ; ; natnum
    ; ; Purpose: The product so far
    ; ; Invariant: accum=Πni=k+1 i
    (define accum (void))
```

```
; ; natnum → natnum
;; Purpose: To compute n!
(define (fact2-v2 n)
  (local
    [;; natnum
     ; ; Purpose: Next accum factor
     (define k (void))

    ; ; natnum
    ; ; Purpose: The product so far
    ; ; Invariant: accum=Πn-ii=k+1
    (define accum (void))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

```
;; → natnum
;; Purpose: Compute n!
;; Effect: k is decremented and
;; accum is multiplied by k
(define (fact-state!)
  (if (= k 0)
      accum
      (begin
        (set! k (sub1 k))
        (set! accum (* k accum))
        (fact-state))))]
```

```
;; → natnum
;; Purpose: Compute n!
;; Effect: k is decremented and
;; accum is multiplied by k
(define (fact-state!)
  (if (= k 0)
      accum
      (begin
        (set! accum (* k accum))
        (set! k (sub1 k))
        (fact-state))))]
```

Mutation Sequencing

```
;; → natnum
;; Purpose: Compute n!
;; Effect: k is decremented and
;; accum is multiplied by k
(define (fact-state!)
  (if (= k 0)
      accum
      (begin
        (set! k (sub1 k))
        (set! accum (* k accum))
        (fact-state))))]
```

```
(begin
  (set! k n)
  (set! accum 1)
  (fact-state!)))
```

```
(check-expect (fact2-v1 0) 1)
(check-expect (fact2-v1 5) 120)
(check-expect (fact2-v1 10) 3628800)
(check-expect (fact2-v1 2) 2)
(check-expect (fact2-v1 6) 720)
```

```
;; → natnum
;; Purpose: Compute n!
;; Effect: k is decremented and
;; accum is multiplied by k
(define (fact-state!)
  (if (= k 0)
      accum
      (begin
        (set! accum (* k accum))
        (set! k (sub1 k))
        (fact-state))))]
```

```
(begin
  (set! k n)
  (set! accum 1)
  (fact-state!)))
```

```
(check-expect (fact2-v2 0) 1)
(check-expect (fact2-v2 5) 120)
(check-expect (fact2-v2 10) 3628800)
(check-expect (fact2-v2 2) 2)
(check-expect (fact2-v2 6) 720)
```

Mutation Sequencing

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- How do we know `accum` is equal to $n!$?

Mutation Sequencing

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- How do we know `accum` is equal to $n!$?
- Consider the assertions that must hold:

$$\text{accum} = \prod_{i=k+1}^n i \wedge k = 0$$

Mutation Sequencing

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- How do we know `accum` is equal to $n!$?
- Consider the assertions that must hold:

$$\text{accum} = \prod_{i=k+1}^n i \wedge k = 0$$

$$\text{accum} = \prod_{i=k+1}^n i \wedge k = 0 \Rightarrow \text{accum} = \prod_{i=0+1}^n i$$

- $\Rightarrow \text{accum} = \prod_{i=1}^n i$
 $\Rightarrow \text{accum} = n!$

Mutation Sequencing

- All that is required is that the state variables be correctly mutated when k is not 0
- Which one should be mutated first?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

- All that is required is that the state variables be correctly mutated when k is not 0
- Which one should be mutated first?
- Sequencing mutations is an extra problem we must solve

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

- All that is required is that the state variables be correctly mutated when k is not 0
- Which one should be mutated first?
- Sequencing mutations is an extra problem we must solve
- Run the two versions of fact-state!
 - 3 tests for fact2-v1 fail
 - 0 tests for fact2-v2 fail

Mutation Sequencing

- All that is required is that the state variables be correctly mutated when k is not 0
- Which one should be mutated first?
- Sequencing mutations is an extra problem we must solve
- Run the two versions of fact-state!
 - 3 tests for fact2-v1 fail
 - 0 tests for fact2-v2 fail
- Is fact2-v2 correct?

Mutation Sequencing

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- All that is required is that the state variables be correctly mutated when k is not 0
- Which one should be mutated first?
- Sequencing mutations is an extra problem we must solve
- Run the two versions of fact-state!
 - 3 tests for fact2-v1 fail
 - 0 tests for fact2-v2 fail
- Is fact2-v2 correct?
- Conclusion from this exercise:
 - A programmer must solve a problem *and* must determine the correct sequencing of the mutations
 - The approach used to determine which program we believe is correct is not scalable

Mutation Sequencing

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- All that is required is that the state variables be correctly mutated when k is not 0
- Which one should be mutated first?
- Sequencing mutations is an extra problem we must solve
- Run the two versions of fact-state!
 - 3 tests for fact2-v1 fail
 - 0 tests for fact2-v2 fail
- Is fact2-v2 correct?
- Conclusion from this exercise:
 - A programmer must solve a problem *and* must determine the correct sequencing of the mutations
 - The approach used to determine which program we believe is correct is not scalable
- A better approach is needed

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Hoare Logic

- *Hoare logic* is used to properly sequence assignment statements and to establish the correctness of imperative programs

Mutation Sequencing

Hoare Logic

- *Hoare logic* is used to properly sequence assignment statements and to establish the correctness of imperative programs
- A Hoare triple consists of two assertions and a mutation or a call to a mutator.

$\text{;; } P$

M

$\text{;; } Q$

- P is an assertion that describes the value of one or more state variables before the mutation(s) performed by M called the *precondition*.
- Q is an assertion that describes the value of one or more state variables after the mutation(s) performed by M called the *postcondition*

Mutation Sequencing

Hoare Logic

- ```
;; X = 2
(set! X (add1 X))
;; X = 3
```

# Mutation Sequencing

## Hoare Logic

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- ```
;; X is a number
(if (< X 0)
    (set! Y (abs X))
    (set! Y X))
;; X is a number ∧ Y = |X| ← How is this proven?
```

Mutation Sequencing

Hoare Logic

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- ```
;; X is a number
(if (< X 0)
 ;; X < 0
 (set! Y (abs X))
 ;; X is a number ∧ Y = |X|
 ;; X ≥ 0
 (set! Y X)
 ;; X is a number ∧ Y = |X|
;; X is a number ∧ Y = |X|
```

# Mutation Sequencing

## Hoare Logic

The use of Hoare logic frequently involves logical implication:

$$A \Rightarrow B$$

- States that A implies B
- A is the *antecedent* and B the *consequent*
- An assertion that means that if A is true then B is true

# Mutation Sequencing

## Hoare Logic

The use of Hoare logic frequently involves logical implication:

$$A \Rightarrow B$$

- States that A implies B
- A is the *antecedent* and B the *consequent*
- An assertion that means that if A is true then B is true

The truth table for an implication is the following:

| A     | B     | $A \Rightarrow B$ |
|-------|-------|-------------------|
| false | false | true              |
| false | true  | true              |
| true  | false | false             |
| true  | true  | true              |

- Always true except when the antecedent is true and the consequent is false
- To prove that an implication is true we only need to prove that the consequent is true when the antecedent is true

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

## Hoare Logic

Prove:

$$x = 3 \Rightarrow x^2 = 9$$

# Mutation Sequencing

## Hoare Logic

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

Prove:

$$x = 3 \Rightarrow x^2 = 9$$

Assume that  $x$  is 3

We must show that  $x^2$  is 9.

# Mutation Sequencing

## Hoare Logic

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

Prove:

$$x = 3 \Rightarrow x^2 = 9$$

Assume that  $x$  is 3

We must show that  $x^2$  is 9.

$$x^2 = 9$$

$3^2 = 9$  by plugging in the value of  $x$

$9 = 9$  by definition of squaring

true

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

## Hoare Logic

- Hoare logic is used to guide imperative code development
- Eliminates guessing the correct sequence of mutations.

# Mutation Sequencing

## Hoare Logic

- Hoare logic is used to guide imperative code development
- Eliminates guessing the correct sequence of mutations.
- $\text{;; } A = n \wedge B = j$   
    ???  
 $\text{;; } A = 2n \wedge B = 3j$

# Mutation Sequencing

## Hoare Logic

- Hoare logic is used to guide imperative code development
- Eliminates guessing the correct sequence of mutations.

- ```
;; A = n ∧ B = j
(set! A (* 2 A))
;; A = 2n ∧ B = j
???
;; A = 2n ∧ B = 3j
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Hoare Logic

- Hoare logic is used to guide imperative code development
- Eliminates guessing the correct sequence of mutations.

- ```
;; A = n ∧ B = j
(set! A (* 2 A))
;; A = 2n ∧ B = j
(set! B (* 3 B))
;; A = 2n ∧ B = 3j
```

# Mutation Sequencing

## Hoare Logic

- $; ; \ A = n$   
 $???$

$; ; \ A = n + 1 \wedge B = 2n$   $\leftarrow$  There is a dependence

# Mutation Sequencing

## Hoare Logic

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- ;; A = n  
(set! B (\* 2 A))  
;; A = n  $\wedge$  B = 2n  
???  
;; A = n + 1  $\wedge$  B = 2n

# Mutation Sequencing

## Hoare Logic

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- ;; A = n  
(set! B (\* 2 A))  
;; A = n  $\wedge$  B = 2n  
(set! A (add1 A))  
;; A = n + 1  $\wedge$  B = 2n

# Mutation Sequencing

Hoare Logic

- $\text{;; } A = n \wedge B = m$   
    ???  
 $\text{;; } A = m \wedge B = n$   $\leftarrow$  There is a circular dependency

# Mutation Sequencing

## Hoare Logic

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- ```
;; A = n ∧ B = m
(local [(define temp A)]
      ???)
;; A = m ∧ B = n
```

Mutation Sequencing

Hoare Logic

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- ```
;; A = n ∧ B = m
(local [(define temp A)]
 (begin
 ;; A = n ∧ B = m ∧ temp = n
 (set! A B)
 ;; A = m ∧ B = m ∧ temp = n
 ???))
;; A = m ∧ B = n
```

# Mutation Sequencing

## Hoare Logic

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- ```
;; A = n ∧ B = m
(local [(define temp A)]
  (begin
    ;; A = n ∧ B = m ∧ temp = n
    (set! A B)
    ;; A = m ∧ B = m ∧ temp = n
    (set! B temp)
    ;; A = m ∧ B = n))
;; A = m ∧ B = n
```

Mutation Sequencing

Developing fact-state!

- Design using the invariant

```
; ; accum = Πni=k+1 i  
<body for fact-state!>
```

Mutation Sequencing

Developing fact-state!

- Design using the invariant

```
; ; accum = Πni=k+1 i  
<body for fact-state!>
```

- Initialize state variables to make the invariant hold
- k is initialized to n and accum is initialized to 1?
- Let us plug-in these values into the invariant to determine if it holds:

Mutation Sequencing

Developing fact-state!

- Design using the invariant

$$\text{;; } \text{accum} = \prod_{i=k+1}^n i$$

<body for fact-state!>

- Initialize state variables to make the invariant hold
- k is initialized to n and accum is initialized to 1?
- Let us plug-in these values into the invariant to determine if it holds:
 - $\text{;; } \text{accum} = \prod_{i=k+1}^n i$
 $\text{;; } 1 = \prod_{i=n+1}^n i$
 $\text{;; } 1 = 1$
 - The invariant holds!

Mutation Sequencing

Developing fact-state!

- ;; natnum → natnum Purpose: Compute n!
(define (fact2-v3 n)
 (local [;; natnum Purpose: The next possible accum factor
 (define k (void))
 ;; natnum Purpose: The current approximation of n!
 (define accum (void))
 ;; natnum natnum → natnum

- (begin
 (set! k n)
 (set! accum 1)
 (fact-state!))))

Mutation Sequencing

Developing fact-state!

- ```
;; natnum → natnum Purpose: Compute n!
(define (fact2-v3 n)
 (local [;; natnum Purpose: The next possible accum factor
 (define k (void))
 ;; natnum Purpose: The current approximation of n!
 (define accum (void))
 ;; natnum natnum → natnum
 ;; Purpose: Compute the factorial of the first
 ;; given natural number
 ;; Effect: k is decremented and accum is multiplied by k
 ;; Accumulator invariant: accum = Πni=k+1 i
 (define (fact-state!))
 ;; accum = Πni=k+1 i
```
- ```
(begin
  (set! k n)
  (set! accum 1)
  (fact-state!))))
```

Mutation Sequencing

Developing fact-state!

- ```
;; natnum → natnum Purpose: Compute n!
(define (fact2-v3 n)
 (local [;; natnum Purpose: The next possible accum factor
 (define k (void))
 ;; natnum Purpose: The current approximation of n!
 (define accum (void))
 ;; natnum natnum → natnum
 ;; Purpose: Compute the factorial of the first
 ;; given natural number
 ;; Effect: k is decremented and accum is multiplied by k
 ;; Accumulator invariant: accum = $\prod_{i=k+1}^n i$
 (define (fact-state!)
 ;; accum = $\prod_{i=k+1}^n i$
 (if (= k 0)
 ;; accum = $\prod_{i=k+1}^n i \wedge k = 0 \Rightarrow accum = n!$
 accum

```
- ```
(begin
  (set! k n)
  (set! accum 1)
  (fact-state!))))
```

Mutation Sequencing

Developing fact-state!

- ```
;; natnum → natnum Purpose: Compute n!
(define (fact2-v3 n)
 (local [;; natnum Purpose: The next possible accum factor
 (define k (void))
 ;; natnum Purpose: The current approximation of n!
 (define accum (void))
 ;; natnum natnum → natnum
 ;; Purpose: Compute the factorial of the first
 ;; given natural number
 ;; Effect: k is decremented and accum is multiplied by k
 ;; Accumulator invariant: accum = Πni=k+1 i
 (define (fact-state!)
 ;; accum = Πni=k+1 i
 (if (= k 0)
 ;; accum = Πni=k+1 i ∧ k = 0 ⇒ accum = n!
 accum
 (begin
 ;; accum = Πni=k+1 i ∧ k ≠ 0
 ;; accum = Πni=k+1 i
 (fact-state!))))
 (begin
 (set! k n)
 (set! accum 1)
 (fact-state!))))
```

# Mutation Sequencing

## Developing fact-state!

- ```
;; natnum → natnum Purpose: Compute n!
(define (fact2-v3 n)
  (local [;; natnum Purpose: The next possible accum factor
         (define k (void))
         ;; natnum Purpose: The current approximation of n!
         (define accum (void))
         ;; natnum natnum → natnum
         ;; Purpose: Compute the factorial of the first
         ;; given natural number
         ;; Effect: k is decremented and accum is multiplied by k
         ;; Accumulator invariant: accum = Πni=k+1 i
         (define (fact-state!)
           ;; accum = Πni=k+1 i
           (if (= k 0)
               ;; accum = Πni=k+1 i ∧ k = 0 ⇒ accum = n!
               accum
               (begin
                 ;; accum = Πni=k+1 i ∧ k ≠ 0
                 (set! accum (* k accum))
                 ;; accum = Πni=k i
               )
             )
           ;; accum = Πni=k+1 i
           (fact-state!)))
         )
       )
     )
   )
 )
```

Mutation Sequencing

Developing fact-state!

- ```
;; natnum → natnum Purpose: Compute n!
(define (fact2-v3 n)
 (local [;; natnum Purpose: The next possible accum factor
 (define k (void))
 ;; natnum Purpose: The current approximation of n!
 (define accum (void))
 ;; natnum natnum → natnum
 ;; Purpose: Compute the factorial of the first
 ;; given natural number
 ;; Effect: k is decremented and accum is multiplied by k
 ;; Accumulator invariant: accum = $\prod_{i=k+1}^n i$
 (define (fact-state!)
 ;; accum = $\prod_{i=k+1}^n i$
 (if (= k 0)
 ;; accum = $\prod_{i=k+1}^n i \wedge k = 0 \Rightarrow accum = n!$
 accum
 (begin
 ;; accum = $\prod_{i=k+1}^n i \wedge k \neq 0$
 (set! accum (* k accum))
 ;; accum = $\prod_{i=k}^n i$
 (set! k (sub1 k))
 ;; accum = $\prod_{i=k+1}^n i$
 (fact-state!))))
 (begin
 (set! k n)
 (set! accum 1)
 (fact-state!))))
```

# Mutation Sequencing

## Developing fact-state!

- ```
;; natnum → natnum Purpose: Compute n!
(define (fact2-v3 n)
  (local [;; natnum Purpose: The next possible accum factor
         (define k (void))
         ;; natnum Purpose: The current approximation of n!
         (define accum (void))
         ;; natnum natnum → natnum
         ;; Purpose: Compute the factorial of the first
         ;; given natural number
         ;; Effect: k is decremented and accum is multiplied by k
         ;; Accumulator invariant: accum = Πni=k+1 i
         (define (fact-state!)
           ;; accum = Πni=k+1 i
           (if (= k 0)
               ;; accum = Πni=k+1 i ∧ k = 0 ⇒ accum = n!
               accum
               (begin
                 ;; accum = Πni=k+1 i ∧ k ≠ 0
                 (set! accum (* k accum))
                 ;; accum = Πni=k i
                 (set! k (sub1 k))
                 ;; accum = Πni=k+1 i
                 (fact-state!)))
           ;;
           ;; Termination argument:
           ;; Natnum k is decremented every recursive call.
           ;; Eventually, k becomes 0 and the recursion stops.
           ])
         (begin
           (set! k n)
           (set! accum 1)
           (fact-state!))))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Imperative Code Debugging

- Can the state variables be mutated in reversed order?

Mutation Sequencing

Imperative Code Debugging

- Can the state variables be mutated in reversed order?

- ```
;; accum = Πni=k+1 i
(if (= k 0)
 ;; accum = Πni=k+1 i ∧ k = 0
 ;; ⇒ accum = Πni=1 i
 ;; ⇒ accum = n!
 accum
 ;; else branch
 (begin
 ;; accum = Πni=k+1 i ∧ k ≠ 0
```

- ```
;; accum = Πni=k+1 i
(fact-state!)
))
```

Mutation Sequencing

Imperative Code Debugging

- Can the state variables be mutated in reversed order?

- ```
;; accum = Πni=k+1 i
(if (= k 0)
 ;; accum = Πni=k+1 i ∧ k = 0
 ;; ⇒ accum = Πni=1 i
 ;; ⇒ accum = n!
 accum
 ;; else branch
 (begin
 ;; accum = Πni=k+1 i ∧ k ≠ 0
 (set! k (sub1 k))
 ;; accum = Πni=k+2 i
 ...
 ;; accum = Πni=k+1 i
 (fact-state!)
))
```

# Mutation Sequencing

## Imperative Code Debugging

- Can the state variables be mutated in reversed order?

- ```
;; accum = Πni=k+1 i
(if (= k 0)
    ;; accum = Πni=k+1 i ∧ k = 0
    ;; ⇒ accum = Πni=1 i
    ;; ⇒ accum = n!
    accum
    ;; else branch
    (begin
        ;; accum = Πni=k+1 i ∧ k ≠ 0
        (set! k (sub1 k))
        ;; accum = Πni=k+2 i
        (set! accum (* (add1 k) accum))
        ;; accum = Πni=k+1 i
        (fact-state!)
    ))
```

Mutation Sequencing

HOMEWORK:

- Design and implement an imperative program to sum the elements of a list.
- Design and implement an imperative program to reverse a (listof X).
- Design and implement an imperative program to add all the integer elements in, [low..high], a given interval.
- QUIZ (due in one week)

Refactor the following function to be imperative:

```
;; lon → number Purpose: Find the max of the given list
(define (max-lon a-lon)
  (local [;; lon number → number
         ;; Purpose: Return max(max of given list, given number)
         ;; Accumulator Invariant: accum = maximum in L - lst
         (define (max-helper lst accum)
           (if (empty? lst)
               accum
               (max-helper (rest lst) (max (first lst) accum))))
         (if (empty? a-lon)
             (error 'max-lon "An empty lon does not have a maximum.")
             (max-helper (rest a-lon) (first a-lon))))
         (check-error (max-lon '()))
         "max-lon: An empty lon does not have a maximum.")
  (check-expect (max-lon '(83 77 392 92 832 49 -9837))
                832))
```

Mutation Sequencing

while Loops

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Hoare

Mutation Sequencing

while Loops

- Common practice to package the repeated mutations of state variables using syntax in which the recursive call is not explicit
- Such syntactic constructs are called *loops*
- We shall focus on while loops.
- To use a while loop in ASL you must install a teachpack that extends the language with the proper syntax
- To install the package
 - Go to the File menu in DrRacket
 - Click on Package Manager...
 - For Package Source type:
<https://github.com/morazanm/while-loops-for-Animated-Programming.git>
 - Click on Install
 - When the package finishes installing you may close the package manager window

Mutation Sequencing

while Loops

- To write programs using while loops in ASL the first line in your program must require the while teachpack as follows:

(require while)

Mutation Sequencing

while Loops

- To write programs using while loops in ASL the first line in your program must require the while teachpack as follows:

```
(require while)
```

- The syntax for a while-loop is:

```
(while expr ← driver
      expr+ ← body
    )
```

Mutation Sequencing

while Loops

- To write programs using while loops in ASL the first line in your program must require the while teachpack as follows:

```
(require while)
```

- The syntax for a while-loop is:

```
(while expr ← driver
      expr+ ← body
    )
```

- Semantics

- The driver is evaluated
- If #false the loop ends and returns (void)
- If #true then the body is evaluated and the process is repeated

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

while Loops

- Tail-recursive function a while loop function is mostly rearranging expressions

Mutation Sequencing

while Loops

- Tail-recursive function a while loop function is mostly rearranging expressions
- The following steps are needed:
 - ① Initialize the state variables to achieve the invariant.
 - ② The conjunction (i.e., and-ing) of the negation of the conditions for nonrecursive cases is the loop's driver.
 - ③ The body of the loop is composed of the code in the recursive cases of the conditional.

Mutation Sequencing

while Loops

- Tail-recursive function a while loop function is mostly rearranging expressions
- The following steps are needed:
 - ① Initialize the state variables to achieve the invariant.
 - ② The conjunction (i.e., and-ing) of the negation of the conditions for nonrecursive cases is the loop's driver.
 - ③ The body of the loop is composed of the code in the recursive cases of the conditional.
- Let us refactor imperative factorial

Mutation Sequencing

while Loops

- ```
(require while)
;; natnum → natnum Purpose: Compute n!
(define (fact2-v4 n)
 (local [;; natnum Purpose: Next accum factor
 (define k (void))
 ;; natnum Purpose: Approximation of n!
 (define accum (void))]
```

# Mutation Sequencing

## while Loops

- ```
(require while)
;; natnum → natnum Purpose: Compute n!
(define (fact2-v4 n)
  (local [;; natnum Purpose: Next accum factor
         (define k (void))
         ;; natnum Purpose: Approximation of n!
         (define accum (void))])
```
- ```
(begin
 (set! k n)
 (set! accum 1)
 ;; INV: accum = Π_{i=k+1}^n i
```

# Mutation Sequencing

## while Loops

- ```
(require while)
;; natnum → natnum Purpose: Compute n!
(define (fact2-v4 n)
  (local [;; natnum Purpose: Next accum factor
         (define k (void))
         ;; natnum Purpose: Approximation of n!
         (define accum (void))]

  (begin
    (set! k n)
    (set! accum 1)
    ;; INV: accum = Π_{i=k+1}^n i
    (while (not (= k 0))
      ;; accum = Π_{i=k+1}^n i ∧ k ≠ 0
```

Mutation Sequencing

while Loops

- ```
(require while)
;; natnum → natnum Purpose: Compute n!
(define (fact2-v4 n)
 (local [;; natnum Purpose: Next accum factor
 (define k (void))
 ;; natnum Purpose: Approximation of n!
 (define accum (void))])
```
- ```
(begin
  (set! k n)
  (set! accum 1)
  ;; INV: accum = Π_{i=k+1}^n i
  (while (not (= k 0))
    ;; accum = Π_{i=k+1}^n i ∧ k ≠ 0
    (set! accum (* k accum)))
  ;; accum = Π_{i=k}^n i
```

Mutation Sequencing

while Loops

- ```
(require while)
;; natnum → natnum Purpose: Compute n!
(define (fact2-v4 n)
 (local [;; natnum Purpose: Next accum factor
 (define k (void))
 ;; natnum Purpose: Approximation of n!
 (define accum (void))])
```
- ```
(begin
  (set! k n)
  (set! accum 1)
  ;; INV: accum = Π_{i=k+1}^n i
  (while (not (= k 0))
    ;; accum = Π_{i=k+1}^n i ∧ k ≠ 0
    (set! accum (* k accum)))
  ;; accum = Π_{i=k}^n i
  (set! k (sub1 k))
  ;; accum = Π_{i=k+1}^n i
  ))
```

Mutation Sequencing

while Loops

- ```
(require while)
;; natnum → natnum Purpose: Compute n!
(define (fact2-v4 n)
 (local [;; natnum Purpose: Next accum factor
 (define k (void))
 ;; natnum Purpose: Approximation of n!
 (define accum (void))])
```
- ```
(begin
  (set! k n)
  (set! accum 1)
  ;; INV: accum = Π_{i=k+1}^n i
  (while (not (= k 0))
    ;; accum = Π_{i=k+1}^n i ∧ k ≠ 0
    (set! accum (* k accum)))
  ;; accum = Π_{i=k}^n i
  (set! k (sub1 k))
  ;; accum = Π_{i=k+1}^n i
  )
  ;; accum = Π_{i=k+1}^n i ∧ k = 0 ⇒ accum = n!
  accum)
```

Mutation Sequencing

while Loops

- ```
(require while)
;; natnum → natnum Purpose: Compute n!
(define (fact2-v4 n)
 (local [;; natnum Purpose: Next accum factor
 (define k (void))
 ;; natnum Purpose: Approximation of n!
 (define accum (void))])
```
- ```
(begin
  (set! k n)
  (set! accum 1)
  ;; INV: accum = Π_{i=k+1}^n i
  (while (not (= k 0))
    ;; accum = Π_{i=k+1}^n i ∧ k ≠ 0
    (set! accum (* k accum)))
  ;; accum = Π_{i=k}^n i
  (set! k (sub1 k))
  ;; accum = Π_{i=k+1}^n i
  )
  ;; accum = Π_{i=k+1}^n i ∧ k = 0 ⇒ accum = n!
  accum)
  ;; Termination argument:
  ;; k is decremented each loop iteration
  ;; Eventually, k becomes 0 and the loop stops.
  ))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

while Loops

HOMEWORK: 6-10

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Design Recipe while Loops

1 Problem and Data Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Design Recipe while Loops

- ① Problem and Data Analysis
- ② Write the signature, the purpose, effect, and how statements, and the function header

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Design Recipe while Loops

- ➊ Problem and Data Analysis
- ➋ Write the signature, the purpose, effect, and how statements, and the function header
- ➌ Write Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Design Recipe while Loops

- ➊ Problem and Data Analysis
- ➋ Write the signature, the purpose, effect, and how statements, and the function header
- ➌ Write Tests
- ➍ Develop the Loop Invariant

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Design Recipe while Loops

- ➊ Problem and Data Analysis
- ➋ Write the signature, the purpose, effect, and how statements, and the function header
- ➌ Write Tests
- ➍ Develop the Loop Invariant
- ➎ Begin writing the function's body with a local-expression: state vars and function headers

Mutation Sequencing

Design Recipe while Loops

- ➊ Problem and Data Analysis
- ➋ Write the signature, the purpose, effect, and how statements, and the function header
- ➌ Write Tests
- ➍ Develop the Loop Invariant
- ➎ Begin writing the function's body with a local-expression: state vars and function headers
- ➏ Write the local-expression's body using a begin-expression that:
 - ➐ Initializes the state variables to achieve the invariant
 - ➑ Defines the while loop
 - ➒ Define the driver and write the loop header
 - ➓ Use the invariant to correctly sequence mutations
 - ➔ Make progress towards termination

Mutation Sequencing

Design Recipe while Loops

- ➊ Problem and Data Analysis
- ➋ Write the signature, the purpose, effect, and how statements, and the function header
- ➌ Write Tests
- ➍ Develop the Loop Invariant
- ➎ Begin writing the function's body with a local-expression: state vars and function headers
- ➏ Write the local-expression's body using a begin-expression that:
 - ➐ Initializes the state variables to achieve the invariant
 - ➑ Defines the while loop
 - ➒ Define the driver and write the loop header
 - ➓ Use the invariant to correctly sequence mutations
 - ➔ Make progress towards termination
- ➐ Design the Post-Loop Code

Mutation Sequencing

Design Recipe while Loops

- ➊ Problem and Data Analysis
- ➋ Write the signature, the purpose, effect, and how statements, and the function header
- ➌ Write Tests
- ➍ Develop the Loop Invariant
- ➎ Begin writing the function's body with a local-expression: state vars and function headers
- ➏ Write the local-expression's body using a begin-expression that:
 - ➐ Initializes the state variables to achieve the invariant
 - ➑ Defines the while loop
 - ➒ Define the driver and write the loop header
 - ➓ Use the invariant to correctly sequence mutations
 - ➔ Make progress towards termination
- ➐ Design the Post-Loop Code
- ➑ Design the Auxiliary Functions

Mutation Sequencing

Design Recipe while Loops

- ➊ Problem and Data Analysis
- ➋ Write the signature, the purpose, effect, and how statements, and the function header
- ➌ Write Tests
- ➍ Develop the Loop Invariant
- ➎ Begin writing the function's body with a local-expression: state vars and function headers
- ➏ Write the local-expression's body using a begin-expression that:
 - ➐ Initializes the state variables to achieve the invariant
 - ➑ Defines the while loop
 - ➒ Define the driver and write the loop header
 - ➓ Use the invariant to correctly sequence mutations
 - ➔ Make progress towards termination
- ➐ Design the Post-Loop Code
- ➑ Design the Auxiliary Functions
- ➒ Develop a Termination Argument

Mutation Sequencing

Design Recipe while Loops

- ➊ Problem and Data Analysis
- ➋ Write the signature, the purpose, effect, and how statements, and the function header
- ➌ Write Tests
- ➍ Develop the Loop Invariant
- ➎ Begin writing the function's body with a local-expression: state vars and function headers
- ➏ Write the local-expression's body using a begin-expression that:
 - ➐ Initializes the state variables to achieve the invariant
 - ➑ Defines the while loop
 - ➒ Define the driver and write the loop header
 - ➓ Use the invariant to correctly sequence mutations
 - ➔ Make progress towards termination
- ➐ Design the Post-Loop Code
- ➑ Design the Auxiliary Functions
- ➒ Develop a Termination Argument
- ➓ Run Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Design Recipe while Loops

- Hardest part is developing the invariant

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

Design Recipe while Loops

- Hardest part is developing the invariant
- How do you know if you have the right invariant?

Mutation Sequencing

Design Recipe while Loops

- Hardest part is developing the invariant
- How do you know if you have the right invariant?
- Golden Rules

Invariant \wedge Driver negation \Rightarrow Post condition

Loop maintains invariant and makes progress towards termination

Mutation Sequencing

Design Recipe while Loops

- Hardest part is developing the invariant
- How do you know if you have the right invariant?
- Golden Rules

Invariant \wedge Driver negation \Rightarrow Post condition

Loop maintains invariant and makes progress towards termination

- Function template

```
;; ... → ... Purpose:  
;; Effect:  
(define (f-while ...)  
  (local [;; <type> Purpose:  
         (define state-var1 (void))  
         ...  
         ;; <type> Purpose:  
         (define state-varN (void))  
         <helper functions>]  
  (begin  
    (set! state-var1 ...) ... (set! state-varN ...)  
    ;; <Invariant>  
    (while <driver>  
      <while-body>)  
    ;; <Invariant> and (not <driver>)  
    <return value code>)  
    ;; <Termination argument>
```

Mutation Sequencing

contains-prime?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Write a predicate to determine if a given interval contains a prime number using a `while`-loop.
- Follow the steps of the design recipe to solve this problem.

Mutation Sequencing

contains-prime?: Problem Analysis

- Traverse from low to high
- As the interval is traversed determine if the low element is a prime number

Mutation Sequencing

contains-prime?: Problem Analysis

- Traverse from low to high
- As the interval is traversed determine if the low element is a prime number
- Two state variables:
 - `l` is the next interval element to test
 - `found` is a Boolean accumulator that is true if there is a prime number in the traversed part of the given interval

Mutation Sequencing

contains-prime?: Problem Analysis

- Traverse from low to high
- As the interval is traversed determine if the low element is a prime number
- Two state variables:
 - `l` is the next interval element to test
 - `found` is a Boolean accumulator that is true if there is a prime number in the traversed part of the given interval
- Initial value of `l` is `low`
- Initial value of `found` is `#false`

Mutation Sequencing

contains-prime?: Problem Analysis

- Traverse from low to high
- As the interval is traversed determine if the low element is a prime number
- Two state variables:
 - `l` is the next interval element to test
 - `found` is a Boolean accumulator that is true if there is a prime number in the traversed part of the given interval
- Initial value of `l` is `low`
- Initial value of `found` is `#false`
- $l \leftarrow l+1$
- $found \leftarrow found \vee (\text{prime? } l)$

Mutation Sequencing

contains-prime?: Problem Analysis

- Traverse from low to high
- As the interval is traversed determine if the low element is a prime number
- Two state variables:
 - l is the next interval element to test
 - `found` is a Boolean accumulator that is true if there is a prime number in the traversed part of the given interval
- Initial value of l is `low`
- Initial value of `found` is `#false`
- $l \leftarrow l+1$
- $\text{found} \leftarrow \text{found} \vee (\text{prime? } l)$
- Loop terminates when $[l..high]$ is empty

Mutation Sequencing

contains-prime?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; [integer integer] → Boolean
;; Purpose: Determine if the given interval contain
;; Assumption: The given interval is not empty
(define (contains-prime? low high)
```

Mutation Sequencing

contains-prime?:Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
; ; Tests for contains-prime?  
(check-expect (contains-prime? 3 2) #false)  
(check-expect (contains-prime? 44 46) #false)  
(check-expect (contains-prime? -9 -4) #false)  
(check-expect (contains-prime? -5 5) #true)  
(check-expect (contains-prime? 19 19) #true)
```

Mutation Sequencing

contains-prime? Loop Invariant

- To formulate the loop invariant start with the loop's postcondition

Mutation Sequencing

contains-prime? Loop Invariant

- To formulate the loop invariant start with the loop's postcondition
- - found means the given interval contains a prime
 - (not found) means given interval does contain a prime.
 - The postcondition is:

$\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}] \wedge$
 $(\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$

Mutation Sequencing

contains-prime? Loop Invariant

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- To formulate the loop invariant start with the loop's postcondition
- - found means the given interval contains a prime
 - (not found) means given interval does not contain a prime.
 - The postcondition is:

$\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}] \wedge$
 $(\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$

- Next write the driver:

$(\geq \text{high } 1)$

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Mutation Sequencing

contains-prime?: Loop invariant

- Use the golden rule

Mutation Sequencing

contains-prime?: Loop invariant

- Use the golden rule
- Loop Invariant \wedge (not (\geq high l))
 \Rightarrow
found \Rightarrow prime in [low..high] \wedge
(not found) \Rightarrow no prime in [low..high]

Mutation Sequencing

contains-prime?: Loop invariant

- Use the golden rule
- $\text{Loop Invariant} \wedge (\text{not } (>= \text{high } l))$
 \Rightarrow
 $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}] \wedge$
 $(\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- The loop's postcondition and invariant should be similar

Mutation Sequencing

contains-prime?: Loop invariant

- Use the golden rule
- $\text{Loop Invariant} \wedge (\text{not } (>= \text{high } l))$
 \Rightarrow
 $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}] \wedge$
 $(\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- The loop's postcondition and invariant should be similar
- The invariant ought to refer to the what has been processed
- The invariant needs to refer to the state variables

Mutation Sequencing

contains-prime?: Loop invariant

- Use the golden rule
- $\text{Loop Invariant} \wedge (\text{not } (>= \text{high } l))$
 \Rightarrow
 $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}] \wedge$
 $(\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- The loop's postcondition and invariant should be similar
- The invariant ought to refer to the what has been processed
- The invariant needs to refer to the state variables
 - $\text{found} \Rightarrow \text{prime in } [\text{low}..l-1] \wedge$
 $(\text{not found}) \Rightarrow \text{no prime in } [\text{low}..l-1]$

Mutation Sequencing

contains-prime?: Loop invariant

- `found ⇒ prime in [low..l-1]`
 \wedge (`not found`) \Rightarrow no prime in `[low..l-1]`
- Is the proposed invariant strong enough?

Mutation Sequencing

contains-prime?: Loop invariant

- $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
- Is the proposed invariant strong enough?
- $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{l} > \text{high})$
 \Rightarrow
 $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- Can't conclude that $\text{l-1} = \text{high}$. Proposed INV is too weak.

Mutation Sequencing

contains-prime?: Loop invariant

- $\text{found} \Rightarrow \text{prime in } [\text{low}..l-1]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..l-1]$
- Is the proposed invariant strong enough?
- $\text{found} \Rightarrow \text{prime in } [\text{low}..l-1]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..l-1]$
 $\wedge (l > \text{high})$
 \Rightarrow
 $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- Can't conclude that $l-1 = \text{high}$. Proposed INV is too weak.
- Update the invariant:
 $l \leq \text{high}+1 \wedge \text{found} \Rightarrow \text{prime in } [\text{low}..l-1]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..l-1]$

Mutation Sequencing

contains-prime?: Loop invariant

- $\text{found} \Rightarrow \text{prime in } [\text{low}..l-1]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..l-1]$
- Is the proposed invariant strong enough?
- $\text{found} \Rightarrow \text{prime in } [\text{low}..l-1]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..l-1]$
 $\wedge (l > \text{high})$
 \Rightarrow
 $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- Can't conclude that $l-1 = \text{high}$. Proposed INV is too weak.
- Update the invariant:
 $l \leq \text{high}+1 \wedge \text{found} \Rightarrow \text{prime in } [\text{low}..l-1]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..l-1]$
- Can the consequent can be proved?
 $\text{INV} \wedge (\text{not driver}) \Rightarrow \text{postcondition}$

Mutation Sequencing

contains-prime?: Loop invariant

- $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
- Is the proposed invariant strong enough?
- $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{l} > \text{high})$
 \Rightarrow
 $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- Can't conclude that $\text{l-1} = \text{high}$. Proposed INV is too weak.
- Update the invariant:
 $1 \leq \text{high+1} \wedge \text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
- Can the consequent can be proved?
 $\text{INV} \wedge (\text{not driver}) \Rightarrow \text{postcondition}$
- $1 \leq \text{high+1} \wedge \text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}] \wedge (\text{l} > \text{high})$

Mutation Sequencing

contains-prime?: Loop invariant

- $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
- Is the proposed invariant strong enough?
- $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{l} > \text{high})$
 \Rightarrow
 $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- Can't conclude that $\text{l-1} = \text{high}$. Proposed INV is too weak.
- Update the invariant:
 $1 \leq \text{high+1} \wedge \text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
- Can the consequent can be proved?
 $\text{INV} \wedge (\text{not driver}) \Rightarrow \text{postcondition}$
- $1 \leq \text{high+1} \wedge \text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}] \wedge (\text{l} > \text{high})$
- $\Rightarrow \text{l} = \text{high+1}$

Mutation Sequencing

contains-prime?: Loop invariant

- $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
- Is the proposed invariant strong enough?
- $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{l} > \text{high})$
 \Rightarrow
 $\text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- Can't conclude that $\text{l-1} = \text{high}$. Proposed INV is too weak.
- Update the invariant:
 $1 \leq \text{high+1} \wedge \text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}]$
- Can the consequent can be proved?
 $\text{INV} \wedge (\text{not driver}) \Rightarrow \text{postcondition}$
- $1 \leq \text{high+1} \wedge \text{found} \Rightarrow \text{prime in } [\text{low}..\text{l-1}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{l-1}] \wedge (\text{l} > \text{high})$
- $\Rightarrow \text{l} = \text{high+1}$
- $\Rightarrow \text{found} \Rightarrow \text{prime in } [\text{low}..\text{high}]$
 $\wedge (\text{not found}) \Rightarrow \text{no prime in } [\text{low}..\text{high}]$
- The proposed invariant is strong enough

Mutation Sequencing

contains-prime?: Function Body

- (local [;; integer Purpose: Next integer to test
 (define l (void))
 ;; Boolean Purpose: Indicate if a prime is found
 (define found (void))
 ;; integer → Boolean
 ;; Purpose: Determine if given integer is a prime
 (define (prime? n) ...)]

Mutation Sequencing

contains-prime?: Function Body

- (local [;; integer Purpose: Next integer to test
 (define l (void))
 ;; Boolean Purpose: Indicate if a prime is found
 (define found (void))
 ;; integer → Boolean
 ;; Purpose: Determine if given integer is a prime
 (define (prime? n) ...)]
- (begin
 (set! l low)
 (set! found #false)
 ;; INV: $l \leq \text{high}+1 \wedge \text{found} \Rightarrow \text{prime in } [\text{low}..\text{l}-1] \wedge$
 ;; (not found) $\Rightarrow \text{no prime in } [\text{low}..\text{l}-1]$

Mutation Sequencing

contains-prime?: Function Body

- ```
(local [;; integer Purpose: Next integer to test
 (define l (void))
 ;; Boolean Purpose: Indicate if a prime is found
 (define found (void))
 ;; integer → Boolean
 ;; Purpose: Determine if given integer is a prime
 (define (prime? n) ...))]
```
- ```
(begin
  (set! l low)
  (set! found #false)
  ;; INV: l ≤ high+1 ∧ found ⇒ prime in [low..l-1] ∧
  ;;       (not found) ⇒ no prime in [low..l-1]
  (while (≥ high l)
    ;; l <= high ∧ found ==> prime in [low..l-1] ∧
    ;; (not found) ==> no prime in [low..l-1])
```

Mutation Sequencing

contains-prime?: Function Body

- ```
(local [;; integer Purpose: Next integer to test
 (define l (void))
 ;; Boolean Purpose: Indicate if a prime is found
 (define found (void))
 ;; integer → Boolean
 ;; Purpose: Determine if given integer is a prime
 (define (prime? n) ...))]
```
- ```
(begin
  (set! l low)
  (set! found #false)
  ;; INV: l ≤ high+1 ∧ found ⇒ prime in [low..l-1] ∧
  ;;       (not found) ⇒ no prime in [low..l-1]
  (while (≥ high l)
    ;; l <= high ∧ found ==> prime in [low..l-1] ∧
    ;; (not found) ==> no prime in [low..l-1]
    (set! found (or (prime? l) found)))
  ;; l <= high ∧ found ==> prime in [low..l] ∧
  ;; (not found) ==> no prime in [low..l]
```

Mutation Sequencing

contains-prime?: Function Body

- ```
(local [;; integer Purpose: Next integer to test
 (define l (void))
 ;; Boolean Purpose: Indicate if a prime is found
 (define found (void))
 ;; integer → Boolean
 ;; Purpose: Determine if given integer is a prime
 (define (prime? n) ...))]
```
- ```
(begin
    (set! l low)
    (set! found #false)
    ;; INV: l ≤ high+1 ∧ found ⇒ prime in [low..l-1] ∧
    ;;       (not found) ⇒ no prime in [low..l-1]
```
- ```
(while (>= high l)
 ;; l <= high ∧ found ==> prime in [low..l-1] ∧
 ;; (not found) ==> no prime in [low..l-1]
```
- ```
(set! found (or (prime? l) found))
    ;; l <= high ∧ found ==> prime in [low..l] ∧
    ;; (not found) ==> no prime in [low..l]
```
- ```
(set! l (add1 l))
 ;; l <= high+1 ∧ found ==> prime in [low..l-1] ∧
 ;; (not found) ==> no prime in [low..l-1]
)
```

# Mutation Sequencing

## contains-prime?: Function Body

- ```
(local [;; integer Purpose: Next integer to test
        (define l (void))
        ;; Boolean Purpose: Indicate if a prime is found
        (define found (void))
        ;; integer → Boolean
        ;; Purpose: Determine if given integer is a prime
        (define (prime? n) ...)]
```
- ```
(begin
 (set! l low)
 (set! found #false)
 ;; INV: l ≤ high+1 ∧ found ⇒ prime in [low..l-1] ∧
 ;; (not found) ⇒ no prime in [low..l-1]
```
- ```
(while (>= high l)
    ;; l <= high ∧ found ==> prime in [low..l-1] ∧
    ;; (not found) ==> no prime in [low..l-1]
```
- ```
(set! found (or (prime? l) found))
 ;; l <= high ∧ found ==> prime in [low..l] ∧
 ;; (not found) ==> no prime in [low..l]
```
- ```
(set! l (add1 l))
    ;; l <= high+1 ∧ found ==> prime in [low..l-1] ∧
    ;; (not found) ==> no prime in [low..l-1]
)
```
- ```
found))
```

# Mutation Sequencing

## contains-prime?: Auxiliary Functions

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

```
;; natnum → Boolean
;; Purpose: Determine if given natnum is a prime
(define (prime? n)
 (local [;; [int int] → Boolean
 ;; Purpose: Determine if n is divisible by
 ;; in the given interval
 (define (any-divide? low high)
 (if (< high low)
 #false
 (or (= (remainder n high) 0)
 (any-divide? low (sub1 high))))))
 (if (< n 2)
 #false
 (not (any-divide? 2 (quotient n 2))))))
```

# Mutation Sequencing

## contains-prime?: Termination Argument

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

The state variable `l` starts at low. Each loop iteration increases `l` by 1. Eventually, `[l..high]` becomes empty when `l = high + 1` and the loop terminates.

# Mutation Sequencing

contains-prime?

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

## HOMEWORK: 11

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Problem Analysis

- If the given list is empty throw an error because an empty list does not have a maximum

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Problem Analysis

- If the given list is empty throw an error because an empty list does not have a maximum
- Traverse the given list and use an accumulator to remember the maximum so far

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Problem Analysis

- If the given list is empty throw an error because an empty list does not have a maximum
- Traverse the given list and use an accumulator to remember the maximum so far
- Two state variables are used
  - The unprocessed part of the list
  - An accumulator for the maximum so far

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Problem Analysis

- If the given list is empty throw an error because an empty list does not have a maximum
- Traverse the given list and use an accumulator to remember the maximum so far
- Two state variables are used
  - The unprocessed part of the list
  - An accumulator for the maximum so far
- The unprocessed part of the list may be initialized to the given list
- The accumulator may be initialized to negative infinity

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Problem Analysis

- If the given list is empty throw an error because an empty list does not have a maximum
- Traverse the given list and use an accumulator to remember the maximum so far
- Two state variables are used
  - The unprocessed part of the list
  - An accumulator for the maximum so far
- The unprocessed part of the list may be initialized to the given list
- The accumulator may be initialized to negative infinity
- After traversing the list the accumulator is returned

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Problem Analysis

- If the given list is empty throw an error because an empty list does not have a maximum
- Traverse the given list and use an accumulator to remember the maximum so far
- Two state variables are used
  - The unprocessed part of the list
  - An accumulator for the maximum so far
- The unprocessed part of the list may be initialized to the given list
- The accumulator may be initialized to negative infinity
- After traversing the list the accumulator is returned
- To test the function the following sample lons are defined:

```
(define L0 '())
(define L1 '(1 2 3 4 5 6))
(define L2 '(5 4 3 2 1 0 -1))
(define L3 '(9 0 -22 54 10 -89))
```

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Signature, Statements, and Function  
Header

```
;; lon → number throws error
;; Purpose: Return the maximum of the given list
(define (max-lon L)
```

# Mutation Sequencing: Tests

max-lon

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

```
;; Tests for max-lon
(check-error
 (max-lon L0)
 "The empty list does not have a maximum element")
(check-expect (max-lon L1) 6)
(check-expect (max-lon L2) 5)
(check-expect (max-lon L3) 54)
```

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.
- Driver
  - (`not (empty? ul)`)

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.
- Driver
  - (`not (empty? ul)`)
- Postcondition:  
`max = maximum(L)`

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.
- Driver
  - (`not (empty? ul)`)
- Postcondition:
  - `max = maximum(L)`
- The processed part of the list:
  - `(L - ul)`

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.
- Driver
  - (`not (empty? ul)`)
- Postcondition:  
`max = maximum(L)`
- The processed part of the list:  
`(L - ul)`
- Example

| <code>ul</code> | <code>max</code> |
|-----------------|------------------|
| '(8 -5 47 8 -1) | -inf.0           |
| '(-5 47 8 -1)   | 8                |
| '(47 8 -1)      | 8                |
| '(8 -1)         | 47               |
| '(-1)           | 47               |
| '()             | 47               |

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.
- Driver
  - (`not (empty? ul)`)
- Postcondition:  
`max = maximum(L)`
- The processed part of the list:  
`(L - ul)`
- Example

| <code>ul</code> | <code>max</code> |
|-----------------|------------------|
| '(8 -5 47 8 -1) | -inf.0           |
| '(-5 47 8 -1)   | 8                |
| '(47 8 -1)      | 8                |
| '(8 -1)         | 47               |
| '(-1)           | 47               |
| '()             | 47               |

- What is invariant?

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.
- Driver
  - (`not (empty? ul)`)
- Postcondition:  
`max = maximum(L)`
- The processed part of the list:  
`(L - ul)`
- Example

| <code>ul</code> | <code>max</code> |
|-----------------|------------------|
| '(8 -5 47 8 -1) | -inf.0           |
| '(-5 47 8 -1)   | 8                |
| '(47 8 -1)      | 8                |
| '(8 -1)         | 47               |
| '(-1)           | 47               |
| '()             | 47               |

- What is invariant?
- `L = (append (L - ul) ul) ∧ max = maximum(L-ul)`

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.
- Driver
  - (`not (empty? ul)`)
- Postcondition:  
`max = maximum(L)`
- The processed part of the list:  
`(L - ul)`
- Example

| <code>ul</code> | <code>max</code> |
|-----------------|------------------|
| '(8 -5 47 8 -1) | -inf.0           |
| '(-5 47 8 -1)   | 8                |
| '(47 8 -1)      | 8                |
| '(8 -1)         | 47               |
| '(-1)           | 47               |
| '()             | 47               |

- What is invariant?
- `L = (append (L - ul) ul) ∧ max = maximum(L-ul)`
- Is the proposed invariant strong enough?

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.
- Driver
  - (`not (empty? ul)`)
- Postcondition:  
`max = maximum(L)`
- The processed part of the list:  
`(L - ul)`
- Example

| <code>ul</code> | <code>max</code> |
|-----------------|------------------|
| '(8 -5 47 8 -1) | -inf.0           |
| '(-5 47 8 -1)   | 8                |
| '(47 8 -1)      | 8                |
| '(8 -1)         | 47               |
| '(-1)           | 47               |
| '()             | 47               |

- What is invariant?
- `L = (append (L - ul) ul) ∧ max = maximum(L-ul)`
- Is the proposed invariant strong enough?
- `L = (append (L - ul) ul) ∧ max = maximum(L-ul) ∧ (empty? ul)`

# Mutation Sequencing

max-lon: Loop invariant

- Call state variables `ul` and `max`.
- Driver
  - (`not (empty? ul)`)
- Postcondition:  
`max = maximum(L)`
- The processed part of the list:  
`(L - ul)`
- Example

| <code>ul</code> | <code>max</code> |
|-----------------|------------------|
| '(8 -5 47 8 -1) | -inf.0           |
| '(-5 47 8 -1)   | 8                |
| '(47 8 -1)      | 8                |
| '(8 -1)         | 47               |
| '(-1)           | 47               |
| '()             | 47               |

- What is invariant?
- `L = (append (L - ul) ul) ∧ max = maximum(L-ul)`
- Is the proposed invariant strong enough?
- `L = (append (L - ul) ul) ∧ max = maximum(L-ul) ∧ (empty? ul)`
- Plug-in `ul`

$$L = L \wedge \text{max} = \text{maximum}(L) \wedge (\text{empty? } ul) \Rightarrow \text{max} = \text{maximum}(L)$$

# Mutation Sequencing

max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))

# Mutation Sequencing

max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))
- (local
  - [;; lon        Purpose: Unprocessed part of L  
          (define ul (void))
  - ;; number     Purpose: The maximum in the processed part of L  
          (define max (void))]

# Mutation Sequencing

max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))
- (local
  - [;; lon        Purpose: Unprocessed part of L  
          (define ul (void))  
          ;; number      Purpose: The maximum in the processed part of L  
          (define max (void))]
  - (begin
    - (set! ul L)
    - (set! max -inf.0)
    - ;; INV: L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)

# Mutation Sequencing

max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))
- (local
  - [;; lon        Purpose: Unprocessed part of L  
          (define ul (void))  
          ;; number      Purpose: The maximum in the processed part of L  
          (define max (void))]
  - (begin
    - (set! ul L)
    - (set! max -inf.0)
    - ;; INV: L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
  - (while (not (empty? ul))  
          ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul))

# Mutation Sequencing

## max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))
- (local
  - [;; lon        Purpose: Unprocessed part of L  
          (define ul (void))  
          ;; number      Purpose: The maximum in the processed part of L  
          (define max (void))]
- (begin
  - (set! ul L)
  - (set! max -inf.0)
  - [;; INV: L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)]
- (while (not (empty? ul))
  - [;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)]
- (if (> (first ul) max)
  - (begin
    - [;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$
    - [;; (> (first ul) max)]

# Mutation Sequencing

## max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))
- (local
  - [;; lon        Purpose: Unprocessed part of L  
          (define ul (void))  
          ;; number      Purpose: The maximum in the processed part of L  
          (define max (void))]
- (begin
  - (set! ul L)
  - (set! max -inf.0)
  - ;; INV: L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
- (while (not (empty? ul))
  - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
- (if (> (first ul) max)
  - (begin
    - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$
    - ;; (> (first ul) max)
    - (set! max (first ul))
    - ;; L = (append (L - ul) ul)  $\wedge$
    - ;; max = maximum(L-(rest ul))

# Mutation Sequencing

## max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))
- (local
  - [;; lon        Purpose: Unprocessed part of L  
          (define ul (void))  
          ;; number      Purpose: The maximum in the processed part of L  
          (define max (void))])
- (begin
  - (set! ul L)
  - (set! max -inf.0)
  - ;; INV: L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul))
- (while (not (empty? ul))
  - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul))
- (if (> (first ul) max)
  - (begin
    - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$
    - ;; (> (first ul) max)
    - (set! max (first ul))
    - ;; L = (append (L - ul) ul)  $\wedge$
    - ;; max = maximum(L-(rest ul)))
  - (set! ul (rest ul))
  - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul))

# Mutation Sequencing

## max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))
- (local
  - [;; lon        Purpose: Unprocessed part of L  
          (define ul (void))  
          ;; number      Purpose: The maximum in the processed part of L  
          (define max (void))]
- (begin
  - (set! ul L)
  - (set! max -inf.0)
  - ;; INV: L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
- (while (not (empty? ul))
  - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
  - (if (> (first ul) max)
    - (begin
      - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$
      - ;; (> (first ul) max)
      - (set! max (first ul))
      - ;; L = (append (L - ul) ul)  $\wedge$
      - ;; max = maximum(L-(rest ul))
    - (set! ul (rest ul))
    - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
  - )
- ;; else  
;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$   
;; (<= (first ul) max)

# Mutation Sequencing

## max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))
- (local
  - [;; lon        Purpose: Unprocessed part of L  
          (define ul (void))  
          ;; number      Purpose: The maximum in the processed part of L  
          (define max (void))]
  - (begin
    - (set! ul L)
    - (set! max -inf.0)
    - ;; INV: L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
  - (while (not (empty? ul))
    - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
    - (if (> (first ul) max)
      - (begin
        - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$
        - ;; (> (first ul) max)
        - (set! max (first ul))
        - ;; L = (append (L - ul) ul)  $\wedge$
        - ;; max = maximum(L-(rest ul))
      - (set! ul (rest ul))
      - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
    - ;; else
      - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$
      - ;; (<= (first ul) max)
      - (set! ul (rest ul))
      - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)

# Mutation Sequencing

## max-lon: Loop invariant

- (if (empty? L)  
    (error "The empty list does not have a maximum element."))
- (local
  - [;; lon        Purpose: Unprocessed part of L  
          (define ul (void))  
          ;; number      Purpose: The maximum in the processed part of L  
          (define max (void))]
  - (begin
    - (set! ul L)
    - (set! max -inf.0)
    - ;; INV: L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
  - (while (not (empty? ul))  
        ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
  - (if (> (first ul) max)  
          (begin
    - ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$   
          ;; (> (first ul) max)
    - (set! max (first ul))
    - ;; L = (append (L - ul) ul)  $\wedge$   
          ;; max = maximum(L-(rest ul))
  - (set! ul (rest ul))  
        ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
  - )
  - ; else  
        ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$   
        ;; (<= (first ul) max)
  - (set! ul (rest ul))  
        ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)
- ))
- ;; L = (append (L - ul) ul)  $\wedge$  max = maximum(L-ul)  $\wedge$  (empty? ul)  
;;  $\Rightarrow$  max = maximum(L)

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Mutation Sequencing

max-lon: Termination argument

```
;; Termination Argument
;; ul starts as a nonempty lon. Each loop iteration
;; decrements ul's length by 1. Eventually, ul
;; becomes empty and the loop terminates.
```

# Mutation Sequencing

max-lon

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

HOMEWORK: 12-15

QUIZ (due in one week): 16

## Vectors

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- Accessing the  $n^{\text{th}}$  element of a list takes time proportional to  $O(n)$

## Vectors

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- Accessing the  $n^{\text{th}}$  element of a list takes time proportional to  $O(n)$
- This is inefficient because when a list is processed its size is fixed
- In essence, it is compound data of fixed, not arbitrary size, akin to a structure

## Vectors

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- Accessing the  $n^{\text{th}}$  element of a list takes time proportional to  $O(n)$
- This is inefficient because when a list is processed its size is fixed
- In essence, it is compound data of fixed, not arbitrary size, akin to a structure
- Why not use a structure?

## Vectors

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- Accessing the  $n^{\text{th}}$  element of a list takes time proportional to  $O(n)$
- This is inefficient because when a list is processed its size is fixed
- In essence, it is compound data of fixed, not arbitrary size, akin to a structure
- Why not use a structure?
- Structures can be cumbersome, if a list has 23 elements:
  - Do programmers want to write a structure with 23 fields?
  - Use 23 different observers?
  - 23 different mutators?

# Vectors

## Basics

- ASL offers a built-in solution: *vectors* (a.k.a arrays)

# Vectors

## Basics

- ASL offers a built-in solution: *vectors* (a.k.a arrays)
- A vector is compound data of fixed size that is allocated in consecutive memory locations
- Any element in the vector may be accessed in,  $O(k)$ , constant time

# Vectors

## Basics

- ASL offers a built-in solution: *vectors* (a.k.a arrays)
- A vector is compound data of fixed size that is allocated in consecutive memory locations
- Any element in the vector may be accessed in,  $O(k)$ , constant time
- Visualize a (vector of  $X$ ) of length 8 as follows:

|       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|
| $X_0$ | $X_1$ | $X_2$ | $X_3$ | $X_4$ | $X_5$ | $X_6$ | $X_7$ |
|-------|-------|-------|-------|-------|-------|-------|-------|

- The valid indices into the vector are: 0, 1, 2, 3, 4, 5 , 6, and 7
- If an attempt is made to access the vector with an index less than 0 or an index greater than 7 an error is thrown

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Basics

(vector  $X_0 \dots X_{n-1}$ ): Builds a size  $n$  vector with  $X_i$  in the  $i^{\text{th}}$  position.

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Basics

`(vector X0...Xn-1)`: Builds a size n vector with X<sub>i</sub> in the i<sup>th</sup> position.

`(build-vector n f)`: Builds a size n vector by applying f to the natural numbers in [0..n-1] and placing (f i) in the i<sup>th</sup> vector position. The signature for f is: natnum → X.

# Vectors

## Basics

`(vector X0...Xn-1)`: Builds a size n vector with X<sub>i</sub> in the i<sup>th</sup> position.

`(build-vector n f)`: Builds a size n vector by applying f to the natural numbers in [0..n-1] and placing (f i) in the i<sup>th</sup> vector position. The signature for f is: natnum → X.

`(vector-length (vectorof X))`: Returns the given vector's length.

# Vectors

## Basics

`(vector X0...Xn-1)`: Builds a size n vector with  $X_i$  in the  $i^{\text{th}}$  position.

`(build-vector n f)`: Builds a size n vector by applying f to the natural numbers in  $[0..n-1]$  and placing  $(f i)$  in the  $i^{\text{th}}$  vector position. The signature for f is:  $\text{natnum} \rightarrow X$ .

`(vector-length (vectorof X))`: Returns the given vector's length.

`(vector-ref (vectorof X) natnum)`: Returns the given vector's value indicated by the given index. The given index must be a natural number in  $[0..v\ell-1]$ , where  $v\ell$  is the given vector's length. If the given index is not in  $[0..v\ell-1]$  then an error is thrown. The  $i^{\text{th}}$  element of a vector is denoted  $V[i]$ .

# Vectors

## Basics

`(vector X0...Xn-1)`: Builds a size n vector with  $X_i$  in the  $i^{\text{th}}$  position.

`(build-vector n f)`: Builds a size n vector by applying f to the natural numbers in  $[0..n-1]$  and placing  $(f i)$  in the  $i^{\text{th}}$  vector position. The signature for f is:  $\text{natnum} \rightarrow X$ .

`(vector-length (vectorof X))`: Returns the given vector's length.

`(vector-ref (vectorof X) natnum)`: Returns the given vector's value indicated by the given index. The given index must be a natural number in  $[0..v\ell-1]$ , where  $v\ell$  is the given vector's length. If the given index is not in  $[0..v\ell-1]$  then an error is thrown. The  $i^{\text{th}}$  element of a vector is denoted  $V[i]$ .

`(vector? X)`: A predicate to determine if the given X is a vector.

# Vectors

## Basics

`(vector X0...Xn-1)`: Builds a size n vector with X<sub>i</sub> in the i<sup>th</sup> position.

`(build-vector n f)`: Builds a size n vector by applying f to the natural numbers in [0..n-1] and placing (f i) in the i<sup>th</sup> vector position. The signature for f is: natnum → X.

`(vector-length (vectorof X))`: Returns the given vector's length.

`(vector-ref (vectorof X) natnum)`: Returns the given vector's value indicated by the given index. The given index must be a natural number in [0..v<sub>l</sub>-1], where v<sub>l</sub> is the given vector's length. If the given index is not in [0..v<sub>l</sub>-1] then an error is thrown. The i<sup>th</sup> element of a vector is denoted V[i].

`(vector? X)`: A predicate to determine if the given X is a vector.

`(vector-set! (vectorof X) natnum X)`: Mutates the vector position indicated by the given index to be the given X. The given index must be in a natural number in [0..v<sub>l</sub>-1], where v<sub>l</sub> is the given vector's length. If the given index is not in [0..v<sub>l</sub>-1] then an error is thrown.

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Basics

- Even though vectors may be accessed in any manner (even randomly) they are usually processed using an interval of valid indices into the vector

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Basics

- Even though vectors may be accessed in any manner (even randomly) they are usually processed using an interval of valid indices into the vector
- All indices into the vector must be in  $[0..(\text{sub1 } (\text{vector-length } V))]$
- A vector interval is a *dependent type*

# Vectors

## Basics

- Even though vectors may be accessed in any manner (even randomly) they are usually processed using an interval of valid indices into the vector
- All indices into the vector must be in  $[0..(\text{sub1 } (\text{vector-length } V))]$
- A vector interval is a *dependent type*
- For a vector,  $V$ , of size  $N$ , a vector interval,  $\text{VINTV1}$ , is two integers,  $\text{low} \geq 0$  and  $-1 \leq \text{high} \leq N-1$ , such that it is either:
  1. empty (i.e.,  $\text{low} > \text{high}$ )
  2.  $[[\text{low}..(\text{sub1 } \text{high})]..\text{high}]$ ,  
where  $[\text{low}..(\text{sub1 } \text{high})]$  is a  $\text{VINTV1}$   
 $\wedge \text{low} \leq \text{high}$
- Suggests processing a vector interval from high to low

# Vectors

## Basics

- Even though vectors may be accessed in any manner (even randomly) they are usually processed using an interval of valid indices into the vector
- All indices into the vector must be in  $[0..(\text{sub1 } (\text{vector-length } V))]$
- A vector interval is a *dependent type*
- For a vector,  $V$ , of size  $N$ , a vector interval,  $\text{VINTV1}$ , is two integers,  $\text{low} \geq 0$  and  $-1 \leq \text{high} \leq N-1$ , such that it is either:
  - empty (i.e.,  $\text{low} > \text{high}$ )
  - $[[\text{low}..(\text{sub1 } \text{high})]..\text{high}]$ ,  
where  $[\text{low}..(\text{sub1 } \text{high})]$  is a  $\text{VINTV1}$   
 $\wedge \text{low} \leq \text{high}$
- Suggests processing a vector interval from high to low
- For a vector,  $V$ , of size  $N$ , a vector interval,  $\text{VINTV2}$ , is two integers,  $\text{high} \geq 0$  and  $0 \leq \text{low} \leq N+1$ , such that it is either:
  - empty (i.e.,  $\text{low} > \text{high}$ )
  - $[\text{low}..[(\text{add1 } \text{low})..\text{high}]]$ ,  
where  $[(\text{add1 } \text{low})..\text{high}]$  is a  $\text{VINTV2}$   
 $\wedge \text{low} \leq \text{high}$
- Suggests processing a vector interval from low to high

# Vectors

## Basics

- Even though vectors may be accessed in any manner (even randomly) they are usually processed using an interval of valid indices into the vector
- All indices into the vector must be in  $[0..(\text{sub1 } (\text{vector-length } V))]$
- A vector interval is a *dependent type*
- For a vector,  $V$ , of size  $N$ , a vector interval,  $\text{VINTV1}$ , is two integers,  $\text{low} \geq 0$  and  $-1 \leq \text{high} \leq N-1$ , such that it is either:
  - empty (i.e.,  $\text{low} > \text{high}$ )
  - $[[\text{low}..(\text{sub1 } \text{high})]..\text{high}]$ ,  
where  $[\text{low}..(\text{sub1 } \text{high})]$  is a  $\text{VINTV1}$   
 $\wedge \text{low} \leq \text{high}$
- Suggests processing a vector interval from high to low
- For a vector,  $V$ , of size  $N$ , a vector interval,  $\text{VINTV2}$ , is two integers,  $\text{high} \geq 0$  and  $0 \leq \text{low} \leq N+1$ , such that it is either:
  - empty (i.e.,  $\text{low} > \text{high}$ )
  - $[\text{low}..[(\text{add1 } \text{low})..\text{high}]]$ ,  
where  $[(\text{add1 } \text{low})..\text{high}]$  is a  $\text{VINTV2}$   
 $\wedge \text{low} \leq \text{high}$
- Suggests processing a vector interval from low to high
- Vector elements, not vector interval elements, are processed

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Basics

```
;; Sample (vectorof X)
(define VECTOR0 ...) ... (define VECTORK ...)
;; f-on-vector: (vector X) ... → ...
;; Purpose:
(define (f-on-vector V ...)
 (local
 [;; f-on-VINTV1: [int int] ... → ...
 ;; Purpose: ...
 (define (f-on-VINTV1 low high ...)
 (if (> low high)
 ...
 ...(vector-ref V high)...(f-on-VINTV1 low (sub1 high)))
);; f-on-VINTV2: [int int] ... → ...
 ;; Purpose:
 (define (f-on-VINTV2 low high ...)
 (if (> low high)
 ...
 ...(vector-ref V low)...(f-on-VINTV2 (add1 low) high)))
 ...))
;; Tests using for f-on-vector
(check-expect (f-on-vector) ...)
```

Part IV:  
Mutation

Marco T.  
Morazán

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Basics

### HOMEWORK: 1, 2

# Vectors

## Vector Processing Using Structural Recursion: Dot Product

### Problem Analysis

- Given two vectors of numbers, V1 and V2, the dot product is defined as:

$$V1 \cdot V2 = \sum_{i=0}^{n-1} V1[i] * V2[i],$$

where n is the length of the vectors.

# Vectors

## Vector Processing Using Structural Recursion: Dot Product

### Problem Analysis

- Given two vectors of numbers, V1 and V2, the dot product is defined as:
$$V1 \cdot V2 = \sum_{i=0}^{n-1} V1[i] * V2[i],$$
where n is the length of the vectors.
- Both vectors must be simultaneously processed
- Design around processing one vector interval

# Vectors

## Vector Processing Using Structural Recursion: Dot Product

### Problem Analysis

- Given two vectors of numbers, V1 and V2, the dot product is defined as:

$$V1 \cdot V2 = \sum_{i=0}^{n-1} V1[i] * V2[i],$$

where n is the length of the vectors.

- Both vectors must be simultaneously processed
- Design around processing one vector interval
- Vector interval may be processed from high to low or from low to high
- Without loss of generality, the vector interval is processed from low to high

# Vectors

## Vector Processing Using Structural Recursion: Dot Product

### Problem Analysis

- Given two vectors of numbers, V1 and V2, the dot product is defined as:
$$V1 \cdot V2 = \sum_{i=0}^{n-1} V1[i]*V2[i],$$
where n is the length of the vectors.
- Both vectors must be simultaneously processed
- Design around processing one vector interval
- Vector interval may be processed from high to low or from low to high
- Without loss of generality, the vector interval is processed from low to high
- Vector interval is processed by calling an auxiliary function as suggested by the function template

# Vectors

## Vector Processing Using Structural Recursion: Dot Product

```
; ; Sample (vectorof X)
(define VECTOR0 (vector)) (define VECTOR1 (vector 1 2 3))
(define VECTOR2 (vector -1 -2 8)) (define VECTOR3 (vector 5 10 20 40))

;; (vector number) (vector number) → number.
;; Purpose: To compute the dot product of the given vectors
;; Assumption: Given vectors have the same length
(define (dot-product V1 V2)
 (local
 [;; [int int] ... → ...
 ;; Purpose: ...
 (define (sum-products low high sum)
 (if (> low high)
 ...
 ... (vector-ref V low)...(f-on-VINTV2 (add1 low) high)))]
 (sum-products 0 (sub1 (vector-length V1)) 0)))

;; Tests using for dot-product
(check-expect (dot-product VECTOR0 VECTOR0) 0)
(check-expect (dot-product VECTOR1 VECTOR2) 19)
(check-expect (dot-product VECTOR3 VECTOR3) 2125)
```

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## sum-products: Problem Analysis

- If the given vector interval, [low..high], is empty then the accumulator is returned

# Vectors

## sum-products: Problem Analysis

- If the given vector interval, `[low..high]`, is empty then the accumulator is returned
- Otherwise, a recursive call incrementing `low` and adding to the accumulator the product of `V1[low]` and `V2[low]`

# Vectors

## sum-products: Problem Analysis

- If the given vector interval, `[low..high]`, is empty then the accumulator is returned
- Otherwise, a recursive call incrementing `low` and adding to the accumulator the product of `V1[low]` and `V2[low]`
- Given vector interval: unprocessed indices

# Vectors

## sum-products: Problem Analysis

- If the given vector interval, [low..high], is empty then the accumulator is returned
- Otherwise, a recursive call incrementing low and adding to the accumulator the product of V1[low] and V2[low]
- Given vector interval: unprocessed indices
- Accumulator invariant is:

$$\text{sum} = \sum_{i=0}^{\text{low}-1} V1[i]*V2[i]$$

# Vectors

## sum-products: Problem Analysis

- If the given vector interval, [low..high], is empty then the accumulator is returned
- Otherwise, a recursive call incrementing low and adding to the accumulator the product of V1[low] and V2[low]
- Given vector interval: unprocessed indices
- Accumulator invariant is:  
$$\text{sum} = \sum_{i=0}^{\text{low}-1} V1[i]*V2[i]$$
- The programmer is responsible for vectors only accessed with valid indices
- The invariant must also help the programmer establish that only valid indices are used to reference any vector

# Vectors

## sum-products: Problem Analysis

- If the given vector interval, [low..high], is empty then the accumulator is returned
- Otherwise, a recursive call incrementing low and adding to the accumulator the product of V1[low] and V2[low]
- Given vector interval: unprocessed indices
- Accumulator invariant is:

$$\text{sum} = \sum_{i=0}^{\text{low}-1} V1[i]*V2[i]$$

- The programmer is responsible for vectors only accessed with valid indices
- The invariant must also help the programmer establish that only valid indices are used to reference any vector
- Invariant  $\wedge$   $\text{low} \leq \text{high} \Rightarrow 0 \leq \text{low} \leq (\text{sub1 } (\text{vector-length } V1))$
- What needs to be added to the invariant?

# Vectors

## sum-products: Problem Analysis

- If the given vector interval, [low..high], is empty then the accumulator is returned

- Otherwise, a recursive call incrementing low and adding to the accumulator the product of V1[low] and V2[low]

- Given vector interval: unprocessed indices

- Accumulator invariant is:

$$\text{sum} = \sum_{i=0}^{\text{low}-1} V1[i]*V2[i]$$

- The programmer is responsible for vectors only accessed with valid indices

- The invariant must also help the programmer establish that only valid indices are used to reference any vector

- Invariant  $\wedge$   $low \leq high \Rightarrow 0 \leq low \leq (\text{sub1 } (\text{vector-length } V1))$

- What needs to be added to the invariant?

- $\text{sum} = \sum_{i=0}^{\text{low}-1} V1[i]*V2[i] \wedge 0 \leq low \leq high+1$

# Vectors

## sum-products: Problem Analysis

- If the given vector interval, [low..high], is empty then the accumulator is returned

- Otherwise, a recursive call incrementing low and adding to the accumulator the product of V1[low] and V2[low]
- Given vector interval: unprocessed indices
- Accumulator invariant is:

$$\text{sum} = \sum_{i=0}^{\text{low}-1} V1[i]*V2[i]$$

- The programmer is responsible for vectors only accessed with valid indices
- The invariant must also help the programmer establish that only valid indices are used to reference any vector
- Invariant  $\wedge$   $0 \leq \text{low} \leq \text{high} \Rightarrow 0 \leq \text{low} \leq (\text{sub1 } (\text{vector-length V1}))$
- What needs to be added to the invariant?
- $\text{sum} = \sum_{i=0}^{\text{low}-1} V1[i]*V2[i] \wedge 0 \leq \text{low} \leq \text{high}+1$
- When the vector interval is not empty low is always a valid index

# Vectors

## sum-products: Signature, Statements, and Function Header

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

```
;; [int int] → number
;; Purpose: Sum the products of corresponding V1
;; elements for indices in the given ve
;; interval
;; ACCUM INV: sum = $\sum_{i=0}^{\text{low}-1} V1[i]*V2[i]$ \wedge $0 \leq \text{low}$
(define (sum-products low high sum)
```

# Vectors

## sum-products: Function Body

```
(if (> low high)
 sum
 (sum-products (add1 low)
 high
 (+ (* (vector-ref V1 low)
 (vector-ref V2 low))
 sum)))
```

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Merging Two Sorted Vectors

- Merge of two (vector of number)s that are sorted in nondecreasing order

# Vectors

## Merging Two Sorted Vectors

- Merge of two (`vector<number>`)s that are sorted in nondecreasing order
- To sharpen our skills the implementation uses a `while`-loop

# Vectors

## Merging Two Sorted Vectors: Problem Analysis

- V1 and V2

|    |   |   |   |
|----|---|---|---|
| 0  | 1 | 4 | 6 |
| -5 | 1 | 3 | 8 |

# Vectors

## Merging Two Sorted Vectors: Problem Analysis

- V1 and V2

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 4 | 6 |
|---|---|---|---|

|    |   |   |   |
|----|---|---|---|
| -5 | 1 | 3 | 8 |
|----|---|---|---|

- res

|    |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|
| -5 | 0 | 1 | 1 | 3 | 4 | 6 | 8 |
|----|---|---|---|---|---|---|---|

- Observe that res is a new vector whose length is the sum of V1's and V2's length.

# Vectors

## sum-products: Problem Analysis

- To populate `res` with the proper values `V1` and `V2` must be traversed. That is, the vector intervals:

```
[0..(sub1 (vector-length res))]
[0..(sub1 (vector-length V1))
[0..(sub1 (vector-length V2))]
```

# Vectors

## sum-products: Problem Analysis

- To populate `res` with the proper values `V1` and `V2` must be traversed. That is, the vector intervals:

```
[0..(sub1 (vector-length res))]
[0..(sub1 (vector-length V1))]
[0..(sub1 (vector-length V2))]
```

- Each vector interval has two parts: the processed part and the unprocessed part:

|    | Processed     | Unprocessed                          |
|----|---------------|--------------------------------------|
| V1 | [0..lowV1-1]  | [lowV1..(sub1 (vector-length V1))]   |
| V2 | [0..lowV2-1]  | [lowV2..(sub1 (vector-length V2))]   |
| V3 | [0..lowres-1] | [lowres..(sub1 (vector-length res))] |

# Vectors

## sum-products: Problem Analysis

- To populate `res` with the proper values `V1` and `V2` must be traversed. That is, the vector intervals:

```
[0..(sub1 (vector-length res))]
[0..(sub1 (vector-length V1))]
[0..(sub1 (vector-length V2))]
```

- Each vector interval has two parts: the processed part and the unprocessed part:

|    | Processed     | Unprocessed                          |
|----|---------------|--------------------------------------|
| V1 | [0..lowV1-1]  | [lowV1..(sub1 (vector-length V1))]   |
| V2 | [0..lowV2-1]  | [lowV2..(sub1 (vector-length V2))]   |
| V3 | [0..lowres-1] | [lowres..(sub1 (vector-length res))] |

- To select the smallest element in the unprocessed parts of `V1` and `V2` there are three conditions that must be distinguished:
  - Unprocessed vector intervals for `V1` and `V2` are not empty
  - Only unprocessed vector interval for `V1` is not empty
  - Only the unprocessed vector interval for `V2` is not empty

# Vectors

## sum-products: Problem Analysis

To test the merging function the following vectors are defined:

```
;; Sample (vectorof number)
(define V0 (vector))
(define V1 (vector 1 2 3))
(define V2 (vector -8 -4 -2 10))
(define V3 (vector 4 5 6))
```

# Vectors

## sum-products: Signature, Statements, and Function Header

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

```
;; (vectorof number) (vectorof number) → (vectorof number)
;; Purpose: To merge the two given vectors
;; Assumption: The given vectors are sorted in
;; nondecreasing order
(define (merge-vectors V1 V2)
```

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

sum-products: Tests

```
;; Tests for merge-vectors
(check-expect (merge-vectors V0 V0) V0)
(check-expect (merge-vectors V0 V1) V1)
(check-expect (merge-vectors V2 V0) V2)
(check-expect (merge-vectors V2 V1) (vector -8 -4 -2 1 2 3 10))
(check-expect (merge-vectors V1 V3) (vector 1 2 3 4 5 6))
```

# Vectors

## sum-products: Loop Invariant

- Postcondition:

```
res[0..(vector-length res)-1] =
 (V1[0..(vector-length V1)-1] ∪ V2[0..(vector-length V2)-1])
 in nondecreasing order
```

# Vectors

## sum-products: Loop Invariant

- Postcondition:

```
res[0..(vector-length res)-1] =
 (V1[0..(vector-length V1)-1] ∪ V2[0..(vector-length V2)-1])
 in nondecreasing order
```

- The loop invariant must assert something about the contents of `res` in relation to the contents of `V1` and `V2`

# Vectors

## sum-products: Loop Invariant

- Postcondition:

```
res[0..(vector-length res)-1] =
 (V1[0..(vector-length V1)-1] ∪ V2[0..(vector-length V2)-1])
 in nondecreasing order
```

- The loop invariant must assert something about the contents of `res` in relation to the contents of `V1` and `V2`
- `res[0..lowres-1] = (V1[0..lowV1-1] ∪ V2[0..lowV2-1])`  
in nondecreasing order

# Vectors

## sum-products: Loop Invariant

- Postcondition:

```
res[0..(vector-length res)-1] =
 (V1[0..(vector-length V1)-1] ∪ V2[0..(vector-length V2)-1])
 in nondecreasing order
```

- The loop invariant must assert something about the contents of `res` in relation to the contents of `V1` and `V2`
- `res[0..lowres-1] = (V1[0..lowV1-1] ∪ V2[0..lowV2-1])`  
in nondecreasing order
- Is the proposed invariant strong enough?

# Vectors

## sum-products: Loop Invariant

- Postcondition:

```
res[0..(vector-length res)-1] =
 (V1[0..(vector-length V1)-1] ∪ V2[0..(vector-length V2)-1])
 in nondecreasing order
```

- The loop invariant must assert something about the contents of `res` in relation to the contents of `V1` and `V2`
- $\text{res}[0..\text{lowres}-1] = (\text{V1}[0..\text{lowV1}-1] \cup \text{V2}[0..\text{lowV2}-1])$   
in nondecreasing order
- Is the proposed invariant strong enough?
- $\text{loop invariant} \wedge (\text{not driver}) \Rightarrow \text{postcondition}$

```
res[0..lowres-1] = (V1[0..lowV1-1] ∪ V2[0..lowV2-1])
 in nondecreasing order
 $\wedge \text{lowres} > (\text{sub1 } (\text{vector-length res}))$
```

≠

```
res[0..(vector-length res)-1] =
 (V1[0..(vector-length V1)-1] ∪ V2[0..(vector-length V2)-1])
 in nondecreasing order
```

- Antecedent does not contain enough information to conclude what values are stored in `lowres`, `lowV1`, and `lowV2`

# Vectors

## sum-products: Loop Invariant

- Indices must be greater than 0 and less than or equal to the length of the vector they index. Loop invariant may be strengthened as follows:

$$\text{res}[0..\text{lowres}-1] = (\text{V1}[0..\text{lowV1}-1] \cup \text{V2}[0..\text{lowV2}-1]) \\ \text{in nondecreasing order}$$
$$\wedge 0 \leq \text{lowres} \leq (\text{vector-length res})$$
$$\wedge 0 \leq \text{lowV1} \leq (\text{vector-length V1})$$
$$\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$$

- Is the invariant now strong enough?

# Vectors

## sum-products: Loop Invariant

- Indices must be greater than 0 and less than or equal to the length of the vector they index. Loop invariant may be strengthened as follows:

$$\text{res}[0..\text{lowres}-1] = (\text{V1}[0..\text{lowV1}-1] \cup \text{V2}[0..\text{lowV2}-1]) \\ \text{in nondecreasing order}$$
$$\wedge 0 \leq \text{lowres} \leq (\text{vector-length res})$$
$$\wedge 0 \leq \text{lowV1} \leq (\text{vector-length V1})$$
$$\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$$

- Is the invariant now strong enough?

- $\text{res}[0..\text{lowres}-1] = (\text{V1}[0..\text{lowV1}-1] \cup \text{V2}[0..\text{lowV2}-1]) \\ \text{in nondecreasing order}$

$$\wedge 0 \leq \text{lowres} \leq (\text{vector-length res})$$
$$\wedge 0 \leq \text{lowV1} \leq (\text{vector-length V1})$$
$$\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$$
$$\wedge \text{lowres} > (\text{sub1 } (\text{vector-length res}))$$

≠

$$\text{res}[0..\text{(sub1 } (\text{vector-length res}))] \\ = (\text{V1}[0..\text{(sub1 } (\text{vector-length V1}))] \\ \cup \text{V2}[0..\text{(sub1 } (\text{vector-length V2}))]) \\ \text{in nondecreasing order}$$

# Vectors

## sum-products: Loop Invariant

- Indices must be greater than 0 and less than or equal to the length of the vector they index. Loop invariant may be strengthened as follows:

$$\text{res}[0..\text{lowres}-1] = (\text{V1}[0..\text{lowV1}-1] \cup \text{V2}[0..\text{lowV2}-1]) \\ \text{in nondecreasing order}$$

$$\wedge 0 \leq \text{lowres} \leq (\text{vector-length res})$$

$$\wedge 0 \leq \text{lowV1} \leq (\text{vector-length V1})$$

$$\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$$

- Is the invariant now strong enough?

- $\text{res}[0..\text{lowres}-1] = (\text{V1}[0..\text{lowV1}-1] \cup \text{V2}[0..\text{lowV2}-1])$

$$\text{in nondecreasing order}$$

$$\wedge 0 \leq \text{lowres} \leq (\text{vector-length res})$$

$$\wedge 0 \leq \text{lowV1} \leq (\text{vector-length V1})$$

$$\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$$

$$\wedge \text{lowres} > (\text{sub1 } (\text{vector-length res}))$$

≠

$$\text{res}[0..\text{(sub1 } (\text{vector-length res}))]$$

$$= (\text{V1}[0..\text{(sub1 } (\text{vector-length V1}))])$$

$$\cup \text{V2}[0..\text{(sub1 } (\text{vector-length V2}))])$$

$$\text{in nondecreasing order}$$

- There is a clear conclusion we may derive from the antecedent:

$$0 \leq \text{lowres} \leq (\text{vector-length res})$$

$$\wedge \text{lowres} > (\text{sub1 } (\text{vector-length res}))$$

$$\Rightarrow \text{lowres} = (\text{vector-length res})$$

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## sum-products: Loop Invariant

- Is it possible to conclude what values are stored in `lowV1` and `lowV2`?

# Vectors

## sum-products: Loop Invariant

- Is it possible to conclude what values are stored in `lowV1` and `lowV2`?
- We know that the length of `res` is the sum `V1`'s and `V2`'s length. Observe that this allows us to conclude what values are stored in `V1` and `V2`:

$$\begin{aligned} \text{lowV1} &\leq (\text{vector-length V1}) \\ \wedge \text{lowV2} &\leq (\text{vector-length V2}) \\ \wedge \text{lowres} &= (\text{vector-length V1}) + (\text{vector-length V2}) \\ \Rightarrow \quad \text{lowV1} &= (\text{vector-length V1}) \wedge \text{lowV2} = (\text{vector-length V2}) \end{aligned}$$

- The consequent holds because any other values for `lowV1` and `lowV2` make the antecedent false and that contradicts our assumption.

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## sum-products: Problem Analysis

- We must still determine if indexing errors do not occur.

# Vectors

## sum-products: Problem Analysis

- We must still determine if indexing errors do not occur.
- The loop is only entered when the `res`'s vector interval is not empty:

$$\begin{aligned} \text{res}[0..\text{lowres}-1] &= (\text{V1}[0..\text{lowV1}-1] \cup \text{V2}[0..\text{lowV2}-1]) \\ &\quad \text{in nondecreasing order} \\ \wedge 0 \leq \text{lowres} &\leq (\text{vector-length res}) \\ \wedge 0 \leq \text{lowV1} &\leq (\text{vector-length V1}) \\ \wedge 0 \leq \text{lowV2} &\leq (\text{vector-length V2}) \\ \wedge \text{lowres} &\leq (\text{sub1 } (\text{vector-length res})) \end{aligned}$$

$\Rightarrow$

$$0 \leq \text{lowres} \leq (\text{sub1 } (\text{vector-length res}))$$

- This informs us that `lowres` is a valid index into `res`.
- We cannot conclude that `lowV1` and `lowV2` are valid indices
- The invariant must still be strengthened

# Vectors

## sum-products: Loop Invariant

- What's the value in `lowres` inside the loop?
- After loop: `lowres = sum of V1 and V2 lengths`

# Vectors

## sum-products: Loop Invariant

- What's the value in `lowres` inside the loop?
- After loop: `lowres = sum of V1 and V2 lengths`
- Suggests `lowres = lowV1 + lowV2`:

# Vectors

## sum-products: Loop Invariant

- What's the value in `lowres` inside the loop?
- After loop: `lowres = sum of V1 and V2 lengths`
- Suggests `lowres = lowV1 + lowV2`:
  - $\wedge 0 \leq \text{lowres} \leq (\text{sub1}(\text{vector-length res}))$
  - $\wedge 0 \leq \text{lowV1} \leq (\text{vector-length V1})$
  - $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$
  - $\wedge \text{lowres} = \text{lowV1} + \text{lowV2}$
- The invariant is strong enough!

# Vectors

## sum-products: Local Declarations

```
(local [;; (vectorof number)
 ;; Purpose: Stores V1 and V2 elements in
 ;; nondecreasing order
 (define res (build-vector (+ (vector-length V1)
 (vector-length V2))
 (λ (i) (void)))))

 ;; int
 ;; Purpose: The next V1 index to process
 (define lowV1 (void))

 ;; int
 ;; Purpose: The next V2 index to process
 (define lowV2 (void))

 ;; int
 ;; Purpose: The next res index to process
 (define lowres (void))]

 ...)
```

# Vectors

## sum-products: Local Expr Body

```
(begin
 (set! lowV1 0)
 (set! lowV2 0)
 (set! lowres 0)
 #| INV:
 res[0..lowres-1] = (V1[0..lowV1-1] ∪ V2[0..lowV2-1])
 in nondecreasing order
 ∧ 0 ≤ lowres ≤ (vector-length res)
 ∧ 0 ≤ lowV1 ≤ (vector-length V1)
 ∧ 0 ≤ lowV2 ≤ (vector-length V2)
 ∧ lowres = lowV1 + lowV2
 |#
```

# Vectors

## sum-products: Local Expr Body

- ```
(while (<= lowres (vector-length res))
      #| res[0..lowres-1] = (V1[0..lowV1-1] ∪ V2[0..lowV2-1])
         in nondecreasing order
         ∧ 0 ≤ lowres < (vector-length res)
         ∧ 0 ≤ lowV1 ≤ (vector-length V1)
         ∧ 0 ≤ lowV2 ≤ (vector-length V2)
         ∧ lowres = lowV1 + lowV2
      ⇒ lowV1 < (vector-length V1) ∨ lowV2 < (vector-length V2)
      |#)
```

Vectors

sum-products: Local Expr Body

- ```
(while (<= lowres (vector-length res))
 #| res[0..lowres-1] = (V1[0..lowV1-1] ∪ V2[0..lowV2-1])
 in nondecreasing order
 ∧ 0 ≤ lowres < (vector-length res)
 ∧ 0 ≤ lowV1 ≤ (vector-length V1)
 ∧ 0 ≤ lowV2 ≤ (vector-length V2)
 ∧ lowres = lowV1 + lowV2
 ⇒ lowV1 < (vector-length V1) ∨ lowV2 < (vector-length V2)
 |#)
```
- Need to determine if elements from both given vectors, only V1, or only V2 need to be processed

```
(cond [(and (< lowV1 (vector-length V1))
 (< lowV2 (vector-length V2)))
 ...]
 [(< lowV1 (vector-length V1)) ...]
 [else ...])
```

# Vectors

## sum-products: Local Expr Body

- Outline the expression for the first condition:

```
#| res[0..lowres-1] = (V1[0..lowV1-1] ∪ V2[0..lowV2-1])
 in nondecreasing order
 ∧ 0 ≤ lowres < (vector-length res)
 ∧ 0 ≤ lowV1 < (vector-length V1)
 ∧ 0 ≤ lowV2 < (vector-length V2)
 ∧ lowres = lowV1 + lowV2
|#
(if (≤ (vector-ref V1 lowV1) (vector-ref V2 lowV2))
 (begin
 :
)
 (begin
 :
)))

```

# Vectors

## sum-products: Local Expr Body

- $V1[\text{lowV1}] \leq V2[\text{lowV2}]$ 
  - #|  $\text{res}[0..\text{lowres}-1] = (V1[0..\text{lowV1}-1] \cup V2[0..\text{lowV2}-1])$   
in nondecreasing order
    - $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$
    - $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$
    - $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$
    - $\wedge \text{lowres} = \text{lowV1} + \text{lowV2}$
    - $\wedge V1[\text{lowV1}] \leq V2[\text{lowV2}]$
  - |#
    - (vector-set! res lowres (vector-ref V1 lowV1))
    - #|  $\text{res}[0..\text{lowres}] = (V1[0..\text{lowV1}] \cup V2[0..\text{lowV2}-1])$   
in nondecreasing order
      - $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$
      - $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$
      - $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$
      - $\wedge \text{lowres} = \text{lowV1} + \text{lowV2}$
  - |#

# Vectors

## sum-products: Local Expr Body

- $V1[\text{lowV1}] \leq V2[\text{lowV2}]$   
  #|    $\text{res}[0..\text{lowres}-1] = (V1[0..\text{lowV1}-1]$   
       $\cup V2[0..\text{lowV2}-1])$   
      in nondecreasing order  
       $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
       $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$   
       $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
       $\wedge \text{lowres} = \text{lowV1} + \text{lowV2}$   
       $\wedge V1[\text{lowV1}] \leq V2[\text{lowV2}]$   
    |#  
    (vector-set! res lowres (vector-ref V1 lowV1))  
    #|    $\text{res}[0..\text{lowres}] = (V1[0..\text{lowV1}]$   
       $\cup V2[0..\text{lowV2}-1])$   
      in nondecreasing order  
       $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
       $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$   
       $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
       $\wedge \text{lowres} = \text{lowV1} + \text{lowV2}$   
    |#
- (set! lowV1 (add1 lowV1))  
  #|    $\text{res}[0..\text{lowres}] = (V1[0..\text{lowV1}-1]$   
       $\cup V2[0..\text{lowV2}-1])$   
      in nondecreasing order  
       $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
       $\wedge 0 \leq \text{lowV1} \leq (\text{vector-length V1})$   
       $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
       $\wedge \text{lowres} = \text{lowV1-1} + \text{lowV2}$   
    |#

# Vectors

## sum-products: Local Expr Body

- $V1[\text{lowV1}] \leq V2[\text{lowV2}]$   
  #|    $\text{res}[0..\text{lowres}-1] = (V1[0..\text{lowV1}-1]$   
       $\cup V2[0..\text{lowV2}-1])$   
      in nondecreasing order  
       $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
       $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$   
       $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
       $\wedge \text{lowres} = \text{lowV1} + \text{lowV2}$   
       $\wedge V1[\text{lowV1}] \leq V2[\text{lowV2}]$   
    |#  
    ( $\text{vector-set! res lowres } (\text{vector-ref V1 lowV1}))$   
    #|    $\text{res}[0..\text{lowres}] = (V1[0..\text{lowV1}]$   
       $\cup V2[0..\text{lowV2}-1])$   
      in nondecreasing order  
       $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
       $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$   
       $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
       $\wedge \text{lowres} = \text{lowV1} + \text{lowV2}$   
    |#  
  • ( $\text{set! lowV1 } (\text{add1 lowV1}))$   
    #|    $\text{res}[0..\text{lowres}] = (V1[0..\text{lowV1}-1]$   
       $\cup V2[0..\text{lowV2}-1])$   
      in nondecreasing order  
       $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
       $\wedge 0 \leq \text{lowV1} \leq (\text{vector-length V1})$   
       $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
       $\wedge \text{lowres} = \text{lowV1-1} + \text{lowV2}$   
    |#  
  • ( $\text{set! lowres } (\text{add1 lowres}))$   
    ;; INV  
    )

# Vectors

## sum-products: Local Expr Body

- $V1[\text{lowV1}] > V2[\text{lowV2}]$ 

```
(begin
 #| res[0..lowres-1] = (V1[0..lowV1-1]
 ∪ (V2[0..lowV2-1]))
 in nondecreasing order
 ∧ 0 ≤ lowres < (vector-length res)
 ∧ 0 ≤ lowV1 ≤ (vector-length V1)
 ∧ 0 ≤ lowV2 < (vector-length V2)
 ∧ lowres = lowV1 + lowV2
 ∧ V1[lowV1] > V2[lowV2]
 |#
 (vector-set! res lowres (vector-ref V2 lowV2))
 #| res[0..lowres] = (V1[0..lowV1-1]
 ∪ (V2[0..lowV2]))
 in nondecreasing order
 ∧ 0 ≤ lowres < (vector-length res)
 ∧ 0 ≤ lowV1 ≤ (vector-length V1)
 ∧ 0 ≤ lowV2 < (vector-length V2)
 ∧ lowres = lowV1 + lowV2
 |#
```

# Vectors

## sum-products: Local Expr Body

- $V1[lowV1] > V2[lowV2]$   
    (begin  
        #|     res[0..lowres-1] = (V1[0..lowV1-1]  
                              \cup (V2[0..lowV2-1]))  
                              in nondecreasing order  
                          ^ 0 ≤ lowres < (vector-length res)  
                          ^ 0 ≤ lowV1 ≤ (vector-length V1)  
                          ^ 0 ≤ lowV2 < (vector-length V2)  
                          ^ lowres = lowV1 + lowV2  
                          ^ V1[lowV1] > V2[lowV2]  
        |#  
        (vector-set! res lowres (vector-ref V2 lowV2))  
        #|     res[0..lowres] = (V1[0..lowV1-1]  
                              \cup (V2[0..lowV2]))  
                              in nondecreasing order  
                          ^ 0 ≤ lowres < (vector-length res)  
                          ^ 0 ≤ lowV1 ≤ (vector-length V1)  
                          ^ 0 ≤ lowV2 < (vector-length V2)  
                          ^ lowres = lowV1 + lowV2  
        |#  
    )  
    #|     set! lowV2 (add1 lowV2))  
    #|     res[0..lowres] = (V1[0..lowV1-1]  
                              \cup (V2[0..lowV2-1]))  
                              in nondecreasing order  
                          ^ 0 ≤ lowres < (vector-length res)  
                          ^ 0 ≤ lowV1 ≤ (vector-length V1)  
                          ^ 0 ≤ lowV2 ≤ (vector-length V2)  
                          ^ lowres = lowV1 + lowV2-1  
    |#

# Vectors

## sum-products: Local Expr Body

- $V1[lowV1] > V2[lowV2]$   
    (begin  
        #|     res[0..lowres-1] = (V1[0..lowV1-1]  
                              \cup (V2[0..lowV2-1]))  
                              in nondecreasing order  
        ^ 0 ≤ lowres < (vector-length res)  
        ^ 0 ≤ lowV1 ≤ (vector-length V1)  
        ^ 0 ≤ lowV2 < (vector-length V2)  
        ^ lowres = lowV1 + lowV2  
        ^ V1[lowV1] > V2[lowV2]  
    |#  
    (vector-set! res lowres (vector-ref V2 lowV2))  
    #|     res[0..lowres] = (V1[0..lowV1-1]  
                              \cup (V2[0..lowV2]))  
                              in nondecreasing order  
        ^ 0 ≤ lowres < (vector-length res)  
        ^ 0 ≤ lowV1 ≤ (vector-length V1)  
        ^ 0 ≤ lowV2 < (vector-length V2)  
        ^ lowres = lowV1 + lowV2  
    |#  
    #|  
    (set! lowV2 (add1 lowV2))  
    #|     res[0..lowres] = (V1[0..lowV1-1]  
                              \cup (V2[0..lowV2-1]))  
                              in nondecreasing order  
        ^ 0 ≤ lowres < (vector-length res)  
        ^ 0 ≤ lowV1 ≤ (vector-length V1)  
        ^ 0 ≤ lowV2 ≤ (vector-length V2)  
        ^ lowres = lowV1 + lowV2-1  
    |#  
    #|  
    (set! lowres (add1 lowres))  
    ;; INV  
    ))))])

# Vectors

## sum-products: Local Expr Body

- [ $(< \text{lowV1} (\text{vector-length V1}))$   
(begin  
   $\#|\text{res}[0..\text{lowres}-1]=(\text{V1}[0..\text{lowV1}-1] \cup (\text{V2}[0..\text{lowV2}-1])) \text{ in nondecreasing}|$   
   $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
   $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$   
   $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
   $\wedge \text{lowres} = \text{lowV1} + \text{lowV2}$   
     $\wedge [\text{lowV2}..(\text{sub1} (\text{vector-length V2}))] \text{ is empty } | \#$   
   $(\text{vector-set! res lowres} (\text{vector-ref V1 lowV1}))$   
   $\#|\text{res}[0..\text{lowres}]=(\text{V1}[0..\text{lowV1}] \cup (\text{V2}[0..\text{lowV2}-1])) \text{ in nondecreasing}|$   
   $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
   $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$   
   $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
   $\wedge \text{lowres} = \text{lowV1} + \text{lowV2} | \#$

# Vectors

## sum-products: Local Expr Body

- [ $(< \text{lowV1} (\text{vector-length V1}))$   
 $\text{(begin}$   
     $\#|\text{res}[0..\text{lowres}-1]=(\text{V1}[0..\text{lowV1}-1] \cup (\text{V2}[0..\text{lowV2}-1]) \text{ in nondecreas}$   
     $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
     $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$   
     $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
     $\wedge \text{lowres} = \text{lowV1} + \text{lowV2}$   
         $\wedge [\text{lowV2}..(\text{sub1} (\text{vector-length V2}))] \text{ is empty } | \#$   
     $(\text{vector-set! res lowres} (\text{vector-ref V1 lowV1}))$   
     $\#|\text{res}[0..\text{lowres}]=(\text{V1}[0..\text{lowV1}] \cup (\text{V2}[0..\text{lowV2}-1]) \text{ in nondecreasing}$   
     $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
     $\wedge 0 \leq \text{lowV1} < (\text{vector-length V1})$   
     $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
     $\wedge \text{lowres} = \text{lowV1} + \text{lowV2} | \#$
- $(\text{set! lowV1} (\text{add1 lowV1}))$   
     $\#|\text{res}[0..\text{lowres}]=(\text{V1}[0..\text{lowV1}-1] \cup (\text{V2}[0..\text{lowV2}]) \text{ in nondecreasing}$   
     $\wedge 0 \leq \text{lowres} < (\text{vector-length res})$   
     $\wedge 0 \leq \text{lowV1} \leq (\text{vector-length V1})$   
     $\wedge 0 \leq \text{lowV2} \leq (\text{vector-length V2})$   
     $\wedge \text{lowres} = \text{lowV1-1} + \text{lowV2} | \#$   
     $(\text{set! lowres} (\text{add1 lowres}))$   
     $; \text{ INV}$   
]

# Vectors

## sum-products: Local Expr Body

- ```
[else
  (begin
    #| res[0..lowres-1] = (V1[0..lowV1-1] ∪ (V2[0..lowV2-1])
                           in nondecreasing order
    ∧ 0 ≤ lowres < (vector-length res)
    ∧ 0 ≤ lowV1 ≤ (vector-length V1)
    ∧ 0 ≤ lowV2 < (vector-length V2)
    ∧ ∧ lowres = lowV1 + lowV2
    ∧ [lowV1..(sub1 (vector-length V1))] is empty
  |#
  (vector-set! res lowres (vector-ref V2 lowV2))
  #| res[0..lowres] = (V1[0..lowV1-1] ∪ (V2[0..lowV2])
                           in nondecreasing order
    ∧ 0 ≤ lowres < (vector-length res)
    ∧ 0 ≤ lowV1 ≤ (vector-length V1)
    ∧ 0 ≤ lowV2 < (vector-length V2)
    ∧ lowres = lowV1 + lowV2
  |#
```

Vectors

sum-products: Local Expr Body

- ```
[else
 (begin
 #| res[0..lowres-1] = (V1[0..lowV1-1] ∪ (V2[0..lowV2-1])
 in nondecreasing order
 ∧ 0 ≤ lowres < (vector-length res)
 ∧ 0 ≤ lowV1 ≤ (vector-length V1)
 ∧ 0 ≤ lowV2 < (vector-length V2)
 ∧ ∧ lowres = lowV1 + lowV2
 ∧ [lowV1..(sub1 (vector-length V1))] is empty
 |#
 (vector-set! res lowres (vector-ref V2 lowV2))
 #| res[0..lowres] = (V1[0..lowV1-1] ∪ (V2[0..lowV2-1])
 in nondecreasing order
 ∧ 0 ≤ lowres < (vector-length res)
 ∧ 0 ≤ lowV1 ≤ (vector-length V1)
 ∧ 0 ≤ lowV2 < (vector-length V2)
 ∧ lowres = lowV1 + lowV2
 |#
 (set! lowV2 (add1 lowV2))
 #| res[0..lowres] = (V1[0..lowV1-1] ∪ (V2[0..lowV2-1])
 in nondecreasing order
 ∧ 0 ≤ lowres < (vector-length res)
 ∧ 0 ≤ lowV1 ≤ (vector-length V1)
 ∧ 0 ≤ lowV2 ≤ (vector-length V2)
 ∧ lowres = lowV1 + lowV2-1
 |#
 (set! lowres (add1 lowres))
 ;; INV
))])
```

← Post-loop Code

# Vectors

## sum-products: Post loop code

```
#| INV ∧ losres > (sub1 (vector-length res))
⇒ lowres = (vector-length res)
 ∧ lowV1 = (vector-length V1)
 ∧ lowV2 = (vector-length V2)
⇒
 res[0..(vector-length res)-1]
 = (V1[0..(vector-length V1)-1]
 ∪ (V2[0..(vector-length V2)-1]))
 in nondecreasing order
|#
res
```

# Vectors

## sum-products: Termination Argument

#|

Termination argument

The state variable lowres starts at 0. Every time through the loop it is incremented by 1. Eventually, lowres becomes  $> (\text{sub1} (\text{vector-length res}))$  and the loop halts.

|#

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

## HOMEWORK: 4-11

# Vectors

## Vector Processing Using Generative Recursion

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- The Sieve of Eratosthenes is an algorithm to compute all the prime numbers less than or equal to a given natural number

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Vector Processing Using Generative Recursion

- The Sieve of Eratosthenes is an algorithm to compute all the prime numbers less than or equal to a given natural number
- We shall use a vector-based algorithm

# Vectors

## Sieve of Eratosthenes: Problem Analysis

- Start by assuming that all natnums in  $[2..n]$  are prime
- At each step the multiples of the interval's first element are marked as nonprime
- Stops when the interval's first element doubled is larger than  $n$

# Vectors

## Sieve of Eratosthenes: Problem Analysis

- Start by assuming that all natnums in  $[2..n]$  are prime
- At each step the multiples of the interval's first element are marked as nonprime
- Stops when the interval's first element doubled is larger than  $n$
- Represent the numbers that are and are not prime so far as a (vector of Boolean) of size  $n+1$ , `is-prime`, is used as an accumulator
- The vector interval of interest is  $[2..n]$
- The vector interval is processed from left to right
- If the first vector interval element, `low`, is prime then the multiples of `low` in the vector interval  $[2*low..n]$  are mutated to `#false`
- If the first vector interval element is not prime (i.e., `is-prime[low]` is `#false`) then its multiples in the rest of the interval are already marked as not prime and `low` is incremented

# Vectors

## Sieve of Eratosthenes: Problem Analysis

- Start by assuming that all natnums in  $[2..n]$  are prime
- At each step the multiples of the interval's first element are marked as nonprime
- Stops when the interval's first element doubled is larger than  $n$
- Represent the numbers that are and are not prime so far as a (vector of Boolean) of size  $n+1$ , `is-prime`, is used as an accumulator
- The vector interval of interest is  $[2..n]$
- The vector interval is processed from left to right
- If the first vector interval element, `low`, is prime then the multiples of `low` in the vector interval  $[2*low..n]$  are mutated to `#false`
- If the first vector interval element is not prime (i.e., `is-prime[low]` is `#false`) then its multiples in the rest of the interval are already marked as not prime and `low` is incremented
- When the loop terminates the list of prime numbers is created by traversing  $[2..n]$

# Vectors

## Sieve of Eratosthenes: Signature, Statements, and Function Header

```
;; natnum → (listof natnum)
;; Purpose: Return all the primes ≤ to the given natnum
(define (sieve-vector n)
```

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Sieve of Eratosthenes: Tests

```
;; Tests for sieve-vector
(check-expect (sieve-vector 0) '())
(check-expect (sieve-vector 1) '())
(check-expect (sieve-vector 11) '(2 3 5 7 11))
(check-expect (sieve-vector 20) '(2 3 5 7 11 13 17 19))
```

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- A proper divisor of a natural number,  $n$ , is any divisor of  $n$  not equal to  $n$

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- A proper divisor of a natural number,  $n$ , is any divisor of  $n$  not equal to  $n$
- Postcondition

$$\begin{aligned} \forall j \in [2..n] \\ (\text{not is-prime}[j]) &\Rightarrow j \text{ is not a prime} \\ \wedge \text{is-prime}[j] &\Rightarrow j \text{ is prime} \end{aligned}$$

- Plausible part of the loop invariant:

$$\begin{aligned} \forall j \in [2..low-1] \\ (\text{not is-prime}[j]) &\Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..low-1] \\ \wedge \text{is-prime}[j] &\Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..low-1] \end{aligned}$$

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- A proper divisor of a natural number,  $n$ , is any divisor of  $n$  not equal to  $n$
- Postcondition

$$\begin{aligned} \forall j \in [2..n] \\ (\text{not is-prime}[j]) &\Rightarrow j \text{ is not a prime} \\ \wedge \text{is-prime}[j] &\Rightarrow j \text{ is prime} \end{aligned}$$

- Plausible part of the loop invariant:

$$\begin{aligned} \forall j \in [2.. \text{low}-1] \\ (\text{not is-prime}[j]) &\Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2.. \text{low}-1] \\ \wedge \text{is-prime}[j] &\Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2.. \text{low}-1] \end{aligned}$$

- It is necessary to determine the value of  $\text{low}$  after the loop
- Must be able to establish that inside the loop  $\text{low}$  is a valid index into  $\text{is-prime}$
- Need an assertion that bounds  $\text{low}$ 's value

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- A proper divisor of a natural number,  $n$ , is any divisor of  $n$  not equal to  $n$
- Postcondition

$$\begin{aligned} \forall j \in [2..n] \\ (\text{not is-prime}[j]) &\Rightarrow j \text{ is not a prime} \\ \wedge \text{is-prime}[j] &\Rightarrow j \text{ is prime} \end{aligned}$$

- Plausible part of the loop invariant:

$$\begin{aligned} \forall j \in [2..low-1] \\ (\text{not is-prime}[j]) &\Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..low-1] \\ \wedge \text{is-prime}[j] &\Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..low-1] \end{aligned}$$

- It is necessary to determine the value of  $low$  after the loop
- Must be able to establish that inside the loop  $low$  is a valid index into  $\text{is-prime}$
- Need an assertion that bounds  $low$ 's value
- Refined invariant:

$$\begin{aligned} \forall j \in [2..low-1] \\ (\text{not is-prime}[j]) &\Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..low-1] \\ \wedge \text{is-prime}[j] &\Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..low-1] \\ \wedge 2 \leq low \leq high + 1 \end{aligned}$$

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- Is the invariant strong enough?

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- Is the invariant strong enough?
- Inside the loop:

$$\begin{aligned} 2 \leq \text{low} \leq \text{high} + 1 \wedge \text{low} \leq \text{high} \\ \Rightarrow \\ 2 \leq \text{low} \leq \text{high} \end{aligned}$$

- `low` is a valid index

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- After the loop
$$\forall j \in [2..low-1] \quad (\text{not } \text{is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..low-1]$$
$$\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..low-1]$$
$$\wedge 2 \leq low \leq high + 1$$
$$\wedge low > high$$

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- After the loop
$$\forall j \in [2..low-1] \quad (\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..low-1]$$
$$\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..low-1]$$
$$\wedge 2 \leq low \leq high + 1$$
$$\wedge low > high$$
- $\Rightarrow$ 
$$low = high + 1$$

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- After the loop  
 $\forall j \in [2..low-1]$   
 $(\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..low-1]$   
 $\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..low-1]$   
 $\wedge 2 \leq low \leq high + 1$   
 $\wedge low > high$
- $\Rightarrow$   
 $low = high + 1$
- $\Rightarrow$   
 $\forall j \in [2..high]$   
 $(\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..high]$   
 $\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..high]$   
=
- $\forall j \in [2..(\text{quotient } n \ 2)]$   
 $(\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in }$   
 $[2..(\text{quotient } n \ 2)]$   
 $\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in }$   
 $[2..(\text{quotient } n \ 2)]$

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- After the loop
$$\forall j \in [2..low-1] \quad \begin{aligned} & (\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..low-1] \\ & \wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..low-1] \\ & \wedge 2 \leq low \leq high + 1 \\ & \wedge low > high \end{aligned}$$
- $\Rightarrow$ 
$$\text{low} = \text{high} + 1$$
- $\Rightarrow$ 
$$\forall j \in [2..high] \quad \begin{aligned} & (\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..high] \\ & \wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..high] \end{aligned}$$
 $=$ 
$$\forall j \in [2..(\text{quotient } n \ 2)] \quad \begin{aligned} & (\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } \\ & \quad [2..(\text{quotient } n \ 2)] \\ & \wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } \\ & \quad [2..(\text{quotient } n \ 2)] \end{aligned}$$
- $\forall j \in [2..(\text{quotient } n \ 2)] \quad \begin{aligned} & (\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } \\ & \quad [2..(\text{quotient } n \ 2)] \\ & \wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } \\ & \quad [2..(\text{quotient } n \ 2)] \\ & \wedge \nexists \text{ a proper divisor of } j \text{ in } [(\text{quotient } n \ 2)+1..n] \end{aligned}$  $\Rightarrow$ 
$$\forall j \in [2..n] \quad \begin{aligned} & (\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..n] \\ & \wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..n] \end{aligned}$$

- The proposed invariant is strong enough

# Vectors

## Sieve of Eratosthenes: Loop Invariant

- After the loop  
 $\forall j \in [2..low-1]$   
 $(\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..low-1]$   
 $\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..low-1]$   
 $\wedge 2 \leq low \leq high + 1$   
 $\wedge low > high$
- $\Rightarrow$   
 $low = high + 1$
- $\Rightarrow$   
 $\forall j \in [2..high]$   
 $(\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..high]$   
 $\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..high]$   
=
- $\forall j \in [2..(\text{quotient } n \ 2)]$   
 $(\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in }$   
 $[2..(\text{quotient } n \ 2)]$   
 $\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in }$   
 $[2..(\text{quotient } n \ 2)]$
- $\forall j \in [2..(\text{quotient } n \ 2)]$   
 $(\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in }$   
 $[2..(\text{quotient } n \ 2)]$   
 $\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in }$   
 $[2..(\text{quotient } n \ 2)]$   
 $\wedge \nexists \text{ a proper divisor of } j \text{ in } [(\text{quotient } n \ 2)+1..n]$
- $\Rightarrow$   
 $\forall j \in [2..n]$   
 $(\text{not is-prime}[j]) \Rightarrow \exists \text{ a proper divisor of } j \text{ in } [2..n]$   
 $\wedge \text{is-prime}[j] \Rightarrow \nexists \text{ a proper divisor of } j \text{ in } [2..n]$
- $\Rightarrow$   
 $\forall j \in [2..n] (\text{not is-prime}[j]) \Rightarrow j \text{ is not a prime}$   
 $\wedge \text{is-prime}[j] \Rightarrow j \text{ is prime}$
- The proposed invariant is strong enough

# Vectors

## Sieve of Eratosthenes: Local Definitions

```
(local [;; (vectorof Boolean) Purpose: Track primes identified
 (define is-prime (build-vector (add1 n) (λ (i) #true)))
 ;; natnum Purpose: Index of next potential prime
 (define low (void))
 (define high (quotient n 2))
 ;; [int int] → (void)
 ;; Purpose: Mark multiples of the given low as not
 ;; prime
 ;; How: ...
 ;; Effect: In is-primes multiples of given i in
 ;; [low..high] are set to #false
 ;; Assumption: low is a multiple of i
 (define (mark-multiples! i low high)
 ...)
 ;; [int int] → (listof natnum)
 ;; Purpose: Return the list of primes ≤ n
 (define (extract-primes low high)
 ...)
]
 :
)
```

# Vectors

## Sieve of Eratosthenes: Local Expr Body

- (if (< n 2)  
      '()  
      (begin  
          (set! low 2)  
          ;; INV  
          ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
          ;;  $\wedge$  is-prime[j]  $\Rightarrow$   $\nexists$  proper divisor of j in [2..low-1]  
          ;;  $\wedge$  2  $\leq$  low  $\leq$  high + 1

# Vectors

## Sieve of Eratosthenes: Local Expr Body

- (if (< n 2)  
      '()  
      (begin  
          (set! low 2)  
          ;; INV  
          ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
          ;;  $\wedge$  is-prime[j]  $\Rightarrow$  # proper divisor of j in [2..low-1]  
          ;;  $\wedge$  2  $\leq$  low  $\leq$  high + 1  
      • (while (<= low high)  
          ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
          ;; is-prime[j]  $\Rightarrow$  # proper divisor of j in [2..low-1]  
          ;;  $\wedge$  2  $\leq$  i < high + 1

# Vectors

## Sieve of Eratosthenes: Local Expr Body

- (if (< n 2)  
      '()  
      (begin  
          (set! low 2)  
          ;; INV  
          ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
          ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
          ;;  $\wedge 2 \leq low \leq high + 1$   
      • (while (<= low high)  
          ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
          ;; is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
          ;;  $\wedge 2 \leq i < high + 1$   
      • (if (vector-ref is-prime low)  
              (begin  
                ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
                ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
                ;;  $\wedge 2 \leq low < high + 1$   
                ;;  $\wedge low$  is a prime

# Vectors

## Sieve of Eratosthenes: Local Expr Body

- (if (< n 2)  
    '()  
    (begin  
        (set! low 2)  
        ;; INV  
        ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
        ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
        ;;  $\wedge 2 \leq low \leq high + 1$   
    • (while (<= low high)  
        ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
        ;; is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
        ;;  $\wedge 2 \leq i < high + 1$   
    • (if (vector-ref is-prime low)  
        (begin  
            ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
            ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
            ;;  $\wedge 2 \leq low < high + 1$   
            ;;  $\wedge$  low is a prime  
        • (mark-multiples! low (\* 2 low) n)  
            ;; For j in [2..low], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
            ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
            ;;  $\wedge 2 \leq low < high + 1$

# Vectors

## Sieve of Eratosthenes: Local Expr Body

- (if (< n 2)  
    '()  
    (begin  
        (set! low 2)  
        ;; INV  
        ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
        ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
        ;;  $\wedge 2 \leq low \leq high + 1$   
    • (while (<= low high)  
        ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
        ;; is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
        ;;  $\wedge 2 \leq i < high + 1$   
    • (if (vector-ref is-prime low)  
        (begin  
            ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
            ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
            ;;  $\wedge 2 \leq low < high + 1$   
            ;;  $\wedge$  low is a prime  
        • (mark-multiples! low (\* 2 low) n)  
            ;; For j in [2..low], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low]  
            ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low]  
            ;;  $\wedge 2 \leq low < high + 1$   
        • (set! low (add1 low))  
            ;; INV  
    )

# Vectors

## Sieve of Eratosthenes: Local Expr Body

- (if (< n 2)  
    '()  
    (begin  
        (set! low 2)  
        ;; INV  
        ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
        ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
        ;;  $\wedge 2 \leq low \leq high + 1$   
    • (while (<= low high)  
        ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
        ;; is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
        ;;  $\wedge 2 \leq i < high + 1$   
    • (if (vector-ref is-prime low)  
        (begin  
            ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
            ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
            ;;  $\wedge 2 \leq low < high + 1$   
            ;;  $\wedge$  low is a prime  
        • (mark-multiples! low (\* 2 low) n)  
            ;; For j in [2..low], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low]  
            ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low]  
            ;;  $\wedge 2 \leq low < high + 1$   
        • (set! low (add1 low))  
            ;; INV  
        )  
    •  
        ;; For j in [2..low-1], (not is-prime[j])  $\Rightarrow \exists$  a proper divisor of j in [2..low-1]  
        ;;  $\wedge$  is-prime[j]  $\Rightarrow \#$  proper divisor of j in [2..low-1]  
        ;;  $\wedge 2 \leq low < high + 1$   
        ;;  $\wedge$  low is not prime  
        (set! low (add1 low))  
        ;; INV  
    ))

# Vectors

## Sieve of Eratosthenes: Local Expr Post-Loop Code

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- ```
;; For j in [2..n],  (not is-prime[j]) ⇒ ∃ a proper divisor of j in [2..low-1]
;;                                     ∧ is-prime[j] ⇒ # proper divisor of j in [2..low-1]
;; ∧ 2 <= low <= high + 1
;; ∧ low > high
;; ⇒ low = high + 1
;; ⇒ For j in [2..n],  (not is-prime[j]) ⇒ ∃ a proper divisor of j in [2..high]
;;                                     ∧ is-prime[j] ⇒ # proper divisor of j in [2..high]
;; For j in [2..n], (not is-prime[j]) ⇒ j is not a prime
;;                      ∧ is-prime[j] ⇒ j is prime
(extract-primes 2 n)))
```

Vectors

Sieve of Eratosthenes: Auxiliary Functions

Sharing Values

Mutation

Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; natnum [int int] → (void)
;; Purpose: Marks multiples of the given number as not
;;           prime in the given is-prime vector interval
;; How: Mutate is-prime[low] to be false and create a new
;;       vector interval to traverse starting with the next
;;       multiple of i by adding i to low
;; Effect: In is-primes multiples of given i in [low..high]
;;          are set to #false
;; Assumption: low is a multiple of i
(define (mark-multiples! i low high)
  (if (> low high)
      (void)
      (begin
        (vector-set! is-prime low #false)
        (mark-multiples! i (+ i low) high))))
;; Termination Argument
;; The given vector interval is [low..high]. Each recursive
;; call is made with a smaller interval: [i+low..high].
;; Eventually, the given vector interval becomes empty and
;; the function halts.
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Vectors

Sieve of Eratosthenes: Auxiliary Functions

```
;; [int int] → (listof natnum)
;; Purpose: Return the list of primes ≤ n
(define (extract-primes low high)
  (cond [(> low high) '()]
        [(vector-ref is-prime low)
         (cons low (extract-primes (add1 low) high))]
        [else (extract-primes (add1 low) high)]))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Vectors

Sieve of Eratosthenes: Termination Argument

The loop traverses $[2..(\text{quotient } n \ 2)]$ from the low end to the high end. The traversal starts with low equal to 2. Every time the body of the loop is evaluated, regardless of which path is taken in the if-expression, low is incremented reducing the size of the vector interval to traverse by 1. Eventually, the vector interval becomes empty when $\text{low} = (\text{quotient } n \ 2) + 1$ and the loop halts.

Vectors

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

HOMEWORK: 12, 13

QUIZ: 14 (due in 1 week)

In-Place Operations

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- We have studied sorting algorithms for lists
- New lists are created
 - insertion-sorting creates a new list every time a number is inserted into a sorted list
 - quick-sorting creates new lists for the numbers less than or equal to the pivot and for the numbers greater than the pivot.

In-Place Operations

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- We have studied sorting algorithms for lists
- New lists are created
 - insertion-sorting creates a new list every time a number is inserted into a sorted list
 - quick-sorting creates new lists for the numbers less than or equal to the pivot and for the numbers greater than the pivot.
- When the list being sorted is short the allocation of new lists is unlikely to be of concern
- When a list is large the amount of memory allocated may become a concern

In-Place Operations

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- We have studied sorting algorithms for lists
- New lists are created
 - insertion-sorting creates a new list every time a number is inserted into a sorted list
 - quick-sorting creates new lists for the numbers less than or equal to the pivot and for the numbers greater than the pivot.
- When the list being sorted is short the allocation of new lists is unlikely to be of concern
- When a list is large the amount of memory allocated may become a concern
- An alternative to allocating auxiliary data structures is to use mutation to perform operations in place
- Memory allocation is lessened or eliminated
- The danger is the loss of the original data structure and of referential transparency

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

Reversing a Vector

- Consider the problem of reversing an arbitrary number of elements
- A natural representation for the elements is a (listof X)
- If the given list is empty then the reversed list is empty
- Otherwise, append the reverse of the rest of the list and a list that only contains the first element
- Three intermediate lists are constructed

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

Reversing a Vector

- Consider the problem of reversing an arbitrary number of elements
- A natural representation for the elements is a (listof X)
- If the given list is empty then the reversed list is empty
- Otherwise, append the reverse of the rest of the list and a list that only contains the first element
- Three intermediate lists are constructed
- A design using an accumulator eliminates the delayed operation
- Significantly reduces the amount of memory allocated
- Only `cons` first element and the accumulator

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

Reversing a Vector

- Consider the problem of reversing an arbitrary number of elements
- A natural representation for the elements is a `(listof X)`
- If the given list is empty then the reversed list is empty
- Otherwise, append the reverse of the rest of the list and a list that only contains the first element
- Three intermediate lists are constructed
- A design using an accumulator eliminates the delayed operation
- Significantly reduces the amount of memory allocated
- Only `cons` first element and the accumulator
- To eliminate this memory allocation the representation use a `(vectorof X)`
- Reversing in-place is possible
- Safe as long as the original vector is not needed

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

Reversing a Vector: Problem Analysis

- The first and last elements are swapped, the second and the next to last elements are swapped, and so on

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

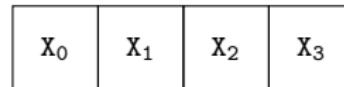
Reversing a Vector: Problem Analysis

- The first and last elements are swapped, the second and the next to last elements are swapped, and so on
- Done by traversing the vector interval `[0..(sub1 (quotient (vector-length V) 2))]`

In-Place Operations

Reversing a Vector: Problem Analysis

- The first and last elements are swapped, the second and the next to last elements are swapped, and so on
- Done by traversing the vector interval `[0..(sub1 (quotient (vector-length V) 2))]`
- Consider reversing the following vector of even length:



- The vector interval processed is `[0..1]`
- $V[0]$ is swapped with $V[3]$ and $V[1]$ is swapped with $V[2]$

In-Place Operations

Reversing a Vector: Problem Analysis

- The first and last elements are swapped, the second and the next to last elements are swapped, and so on
- Done by traversing the vector interval `[0..(sub1 (quotient (vector-length V) 2))]`
- Consider reversing the following vector of even length:

x_0	x_1	x_2	x_3
-------	-------	-------	-------

- The vector interval processed is `[0..1]`
- $V[0]$ is swapped with $V[3]$ and $V[1]$ is swapped with $V[2]$
- Consider reversing the following vector of odd length:

x_0	x_1	x_2	x_3	x_4
-------	-------	-------	-------	-------

- The vector interval processed is `[0..1]`
- $V[0]$ is swapped with $V[4]$, $V[1]$ is swapped with $V[3]$, and $V[2]$. is not swapped
- $V[2]$ does not need to be swapped because it is the middle element

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

Reversing a Vector: Problem Analysis

- The vector interval may be traversed from the low end to the high end
- If the vector interval is empty (`void`) is returned
- Otherwise, `V[low]` is swapped with `V[(sub1 (- (vector-length V) low))]` and the rest of the vector interval is processed recursively.

In-Place Operations

reverse-vector-in-place!: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Call a function to process a vector interval
- The vector interval to process is [0..(sub1 (quotient (vector-length v) 2))]

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

reverse-vector-in-place!: Problem Analysis

- Call a function to process a vector interval
- The vector interval to process is `[0..(sub1 (quotient (vector-length v) 2))]`
- To test the mutator the following vectors are defined:

```
(define V0 (vector))
(define V1 (vector 1 2 3 4 5))
(define V2 (build-vector 5000 (λ (i) (make-posn i i))))
```

In-Place Operations

reverse-vector-in-place!: Signature and Statements

```
;; (vectorof X) → (void)
;; Purpose: Reverse the elements in the given vector
;; Effect: Vector elements are rearranged such that
;;           ∀ i ∈ [0..(quotient n 2)] V[i] is
;;           swapped with V[(vector-length V)-i-1]
(define (reverse-vector-in-place! V)
```

In-Place Operations

reverse-vector-in-place!: Tests

```
(check-expect (begin
                  (reverse-vector-in-place! V0)
                  V0)
                  (vector))

(check-expect (begin
                  (reverse-vector-in-place! V1)
                  V1)
                  (vector 5 4 3 2 1))

(check-expect
  (begin
    (reverse-vector-in-place! V2)
    V2)
  (build-vector 5000
    (λ (i)
      (make-posn (- 5000 i 1) (- 5000 i 1)))))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

reverse-vector-in-place!: Function Body

```
(rev-vector! 0 (sub1 (quotient (vector-length V) 2)))))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

rev-vector!: Problem Analysis

- Traverse the given vector interval [low..high]
- If the vector interval is empty (void) is returned
- $v[low]$ is swapped with $v[(sub1 (- (vector-length v) low))]$ and the rest of the vector interval is recursively processed

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

rev-vector!: Signature and Statements

```
;; [int int] → (void)
;; Purpose: Reverse V's elements
;; Effect: Vector elements in the given vector interval
;;          are rearranged such that V[i] is swapped
;;          with V[(vector-length V)-i-1]
;; Assumption: low ≥ 0 ∧ 2*high is a valid index into V
(define (rev-vector! low high)
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

`rev-vector!:` Tests

- `(define VV0 (vector))
(define VV1 (vector 1 2 3 4))
(define VV2 (vector 1 2 3 4 5 6 7 8 9))`

In-Place Operations

rev-vector!: Tests

- ```
(define VV0 (vector))
(define VV1 (vector 1 2 3 4))
(define VV2 (vector 1 2 3 4 5 6 7 8 9))
```
- ```
(check-expect (begin
                    (rev-vector! VV0 0 -1)
                    VV0)
                    (vector))
(check-expect (begin
                    (rev-vector! VV1 0 1)
                    VV1)
                    (vector 4 3 2 1))

(check-expect (begin
                    (rev-vector! VV2 0 3)
                    VV2)
                    (vector 9 8 7 6 5 4 3 2 1)))
```

In-Place Operations

`rev-vector!`: Tests

- ```
(define VV0 (vector))
(define VV1 (vector 1 2 3 4))
(define VV2 (vector 1 2 3 4 5 6 7 8 9))
```
- ```
(check-expect (begin
                    (rev-vector! VV0 0 -1)
                    VV0)
                    (vector))
(check-expect (begin
                    (rev-vector! VV1 0 1)
                    VV1)
                    (vector 4 3 2 1))

(check-expect (begin
                    (rev-vector! VV2 0 3)
                    VV2)
                    (vector 9 8 7 6 5 4 3 2 1)))
```

- Passing tests ought suggests the function is properly implemented
- It is at this point that the function may be encapsulated
- Once encapsulated `V` may be removed as an input to obtain the signature and function header from the previous step of the design recipe

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

rev-vector!: Function Body

```
(if (> low high)
    (void)
    (begin
        (swap! low (sub1 (- (vector-length V) low)))
        (rev-vector! (add1 low) high))))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

swap!

```
;; natnum natnum → (void)
;; Purpose: Swap the elements at the given indices
;; Effect: V is mutated by swapping elements at given indices
(define (swap! i j)
  (local [(define temp (vector-ref V i))]
    (begin
      (vector-set! V i (vector-ref V j))
      (vector-set! V j temp))))
```

Run all the tests and make sure they pass

In-Place Operations

reverse-vector-in-place!: Performance

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Has anything been gained?

In-Place Operations

reverse-vector-in-place!: Performance

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Has anything been gained?
- To answer this question empirical data is collected from running the following experiments 5 times:

```
(define L4rev-lox (build-list 50000 (λ (i) i)))
(define L4rev-lox2 (build-list 50000 (λ (i) i)))
(define V4rv-in-p (build-vector 50000 (λ (i) i)))

(define T1 (time (rev-lox L4rev-lox)))
(define T2 (time (rev-lox2 L4rev-lox2)))
(define T3 (time (reverse-vector-in-place! V4rv-in-p)))
```

In-Place Operations

reverse-vector-in-place!: Performance

Sharing Values

Mutation
Sequencing

Vectors

In-Place
OperationsThe Chicken
and the Egg
Paradox

Epilogue

- Has anything been gained?
- To answer this question empirical data is collected from running the following experiments 5 times:

```
(define L4rev-lox (build-list 50000 (λ (i) i)))
(define L4rev-lox2 (build-list 50000 (λ (i) i)))
(define V4rv-in-p (build-vector 50000 (λ (i) i)))

(define T1 (time (rev-lox L4rev-lox)))
(define T2 (time (rev-lox2 L4rev-lox2)))
(define T3 (time (reverse-vector-in-place! V4rv-in-p)))
```

-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
rev-lox	8984	9234	9141	12125	9188	9734.4
rev-lox2	15	15	15	15	15	15
rev-in-place!	0	0	0	0	0	0

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

Reversing a Vector

HOMEWORK: 1-5

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Quick Sorting

- If the list is empty then the result is the empty list
- If the list is not empty:
 - Pick a pivot element
 - Create a list with the elements less than or equal to the pivot
 - Create a list with the elements greater than the pivot
 - Recursively sort the two lists
 - Appending the sorted smaller elements and the list containing the pivot and the sorted greater elements

In-Place Operations

In-Place Quick Sorting

- To avoid the allocation of intermediate lists the representation of the numbers to sort may be changed to be a (vector of number)
- In-place sort the elements in [low..high]:

```
If the vector interval is empty
    return (void)
else
    Pick a pivot (say, V[low])
    Partition the vector so that
        VINTV[low..i] contains the elements ≤ pivot
        VINTV[i+1..high] contains the elements > vector
    Swap V[low] and V[i]
    Recursively sort VINTV[low..i-1] and VINTV[i+1..high]
```

In-Place Operations

In-Place Quick Sorting

- To avoid the allocation of intermediate lists the representation of the numbers to sort may be changed to be a (vectorof number)
- In-place sort the elements in [low..high]:

```
If the vector interval is empty
    return (void)
else
    Pick a pivot (say, V[low])
    Partition the vector so that
        VINTV[low..i] contains the elements ≤ pivot
        VINTV[i+1..high] contains the elements > vector
    Swap V[low] and V[i]
    Recursively sort VINTV[low..i-1] and VINTV[i+1..high]
```

- To test in-place quick sorting the following sample vectors are defined:

```
(define V0 (vector))
(define V1 (vector 10 3 7 17 11))
(define V2 (vector 31 46 60 22 74 22 27 60 20 44
                  23 85 86 67 12 75 80 77 62 37))
(define V3 (build-vector 10 (λ (i) i)))
(define V4 (build-vector 10 (λ (i) (sub1 (- 10 i)))))
```

In-Place Operations

qs-in-place!: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Process [0..(sub1 (vector-length V))]
- Call auxiliary function is used

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

qs-in-place!: Signature, Statements, Function
Header

```
;; (vectorof number) → (void)
;; Purpose: Sort the given vector in nondecreasing order
;; Effect: Vector elements are rearranged in place in
;;           nondecreasing order
(define (qs-in-place! V)
```

In-Place Operations

qs-in-place!: Tests

```
(check-expect (begin
                  (qs-in-place! V0)
                  V0)
                  (vector))
(check-expect (begin
                  (qs-in-place! V1)
                  V1)
                  (vector 3 7 10 11 17))
(check-expect (begin
                  (qs-in-place! V2)
                  V2)
                  (vector 12 20 22 22 23 27 31 37 44 46
                         60 60 62 67 74 75 77 80 85 86))
(check-expect (begin
                  (qs-in-place! V3)
                  V3)
                  (vector 0 1 2 3 4 5 6 7 8 9))
(check-expect (begin
                  (qs-in-place! V4)
                  V4)
                  (vector 0 1 2 3 4 5 6 7 8 9))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

qs-in-place!: Function Body

```
(local [...]
  (qs-aux! 0 (sub1 (vector-length V)))))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

qs-aux!: Problem Analysis

- If the given vector interval is empty then the function returns (`void`)

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

qs-aux!: Problem Analysis

- If the given vector interval is empty then the function returns (void)
- $v[low]$ is chosen as the pivot

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

qs-aux!: Problem Analysis

- If the given vector interval is empty then the function returns (`void`)
- $v[low]$ is chosen as the pivot
- Vector is partitioned
 - Elements less than or equal to the pivot at the beginning
 - Elements greater than the pivot at the end of the vector interval

In-Place Operations

qs-aux!: Problem Analysis

- If the given vector interval is empty then the function returns (`void`)
- $V[\text{low}]$ is chosen as the pivot
- Vector is partitioned
 - Elements less than or equal to the pivot at the beginning
 - Elements greater than the pivot at the end of the vector interval
- Let the largest index of the elements less than or equal to the pivot be `pivot-pos`
- `pivot-pos` is the position of the pivot in the sorted vector
- The pivot is swapped with $V[\text{pivot}-\text{pos}]$

In-Place Operations

qs-aux!: Problem Analysis

- If the given vector interval is empty then the function returns (`void`)
- $V[low]$ is chosen as the pivot
- Vector is partitioned
 - Elements less than or equal to the pivot at the beginning
 - Elements greater than the pivot at the end of the vector interval
- Let the largest index of the elements less than or equal to the pivot be `pivot-pos`
- `pivot-pos` is the position of the pivot in the sorted vector
- The pivot is swapped with $V[pivot\text{-}pos]$
- $V[low..(sub1\ pivot\text{-}pos)]$ and $V[(add1\ pivot\text{-}pos)..high]$, are recursively sorted.

In-Place Operations

qs-aux!: Problem Analysis

- Sort:

V:	low	high
	10	2

- The pivot, $V[low]$, is 10

In-Place Operations

qs-aux!: Problem Analysis

- Sort:

V:	low	10	2	30	1	10	16	20	high

- The pivot, $V[low]$, is 10
- After partitioning the vector:

V:	low	10	2	pivot-pos	10	1	30	16	20	high

In-Place Operations

qs-aux!: Problem Analysis

- Sort:

V:	low	high
	10 2 30 1 10 16 20	

- The pivot, $V[\text{low}]$, is 10
- After partitioning the vector:

V:	low	pivot-pos	high
	10 2 10 1 30 16 20		

- The pivot's position in the sorted vector is where the 1 is because 1 has the largest index among the elements less than or equal to the pivot.
- The pivot and 1 are swapped to obtain:

V:	low	pivot-pos	high
	1 2 10 10 30 16 20		

- 10 is in the position it shall have when the vector is sorted

In-Place Operations

qs-aux!: Problem Analysis

- Sort:

V:	low	high
	10 2 30 1 10 16 20	

- The pivot, $V[\text{low}]$, is 10
- After partitioning the vector:

V:	low	pivot-pos	high
	10 2 10 1 30 16 20		

- The pivot's position in the sorted vector is where the 1 is because 1 has the largest index among the elements less than or equal to the pivot.
- The pivot and 1 are swapped to obtain:

V:	low	pivot-pos	high
	1 2 10 10 30 16 20		

- 10 is in the position it shall have when the vector is sorted
- The smaller elements, before the pivot, and the larger elements, after the pivot, are recursively sorted.

In-Place Operations

qs-aux!: Signature, Statements, and Function Header

```
;; [int int] → (void)
;; Purpose: Sort V's elements in the given vector interval
;;           in nondecreasing order
;; How: The vector is partitioned in two. The first element
;;       is placed in the vector position between the
;;       elements ≤ to it and the elements > than it. The
;;       The vector intervals for the two parts of the
;;       partition are recursively sorted
;; Effect: Vector elements in the given vector interval are
;;           rearranged in nondecreasing order.
(define (qs-aux! low high)
```

In-Place Operations

qs-aux!: Function Body

```
(if (> low high)
    (void)
    (local [(define pivot (vector-ref V low))
            (define pivot-pos (partition! pivot low high))]
      (begin
        (swap! low pivot-pos)
        (qs-aux! low (sub1 pivot-pos))
        (qs-aux! (add1 pivot-pos) high))))
```

In-Place Operations

qs-aux!: Termination Argument

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; Termination Argument
;; A given nonempty vector interval is divided into two
;; smaller vector intervals and these are recursively
;; processed. Eventually, the given vector interval
;; becomes empty and the function halts.
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

partition!: Problem Analysis

- Rearrange the vector elements in a given vector interval such that all the numbers less than or equal to the pivot are before the numbers greater than the pivot

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

`partition!:` Problem Analysis

- Rearrange the vector elements in a given vector interval such that all the numbers less than or equal to the pivot are before the numbers greater than the pivot
- Return the largest index of a number less than or equal to the pivot after partitioning the elements

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

partition!: Problem Analysis

- Rearrange the vector elements in a given vector interval such that all the numbers less than or equal to the pivot are before the numbers greater than the pivot
- Return the largest index of a number less than or equal to the pivot after partitioning the elements
- Any two numbers that are relatively out of order must be swapped
- Two numbers are relatively out of order if a vector element greater than the pivot has an index that is smaller than another vector element that is less than or equal to the pivot

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

`partition!:` Problem Analysis

- Rearrange the vector elements in a given vector interval such that all the numbers less than or equal to the pivot are before the numbers greater than the pivot
- Return the largest index of a number less than or equal to the pivot after partitioning the elements
- Any two numbers that are relatively out of order must be swapped
- Two numbers are relatively out of order if a vector element greater than the pivot has an index that is smaller than another vector element that is less than or equal to the pivot
- Search, from low to high, for the first index, i , for a number $>$ the pivot

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

partition!: Problem Analysis

- Rearrange the vector elements in a given vector interval such that all the numbers less than or equal to the pivot are before the numbers greater than the pivot
- Return the largest index of a number less than or equal to the pivot after partitioning the elements
- Any two numbers that are relatively out of order must be swapped
- Two numbers are relatively out of order if a vector element greater than the pivot has an index that is smaller than another vector element that is less than or equal to the pivot
- Search, from low to high, for the first index, i , for a number $>$ the pivot
- Search, from high to low, for the first index, j , for a number \leq the pivot

In-Place Operations

partition!: Problem Analysis

- Rearrange the vector elements in a given vector interval such that all the numbers less than or equal to the pivot are before the numbers greater than the pivot
- Return the largest index of a number less than or equal to the pivot after partitioning the elements
- Any two numbers that are relatively out of order must be swapped
- Two numbers are relatively out of order if a vector element greater than the pivot has an index that is smaller than another vector element that is less than or equal to the pivot
- Search, from low to high, for the first index, i , for a number $>$ the pivot
- Search, from high to low, for the first index, j , for a number \leq the pivot
- If $[i..j]$ is empty then V's elements in the given vector interval are partitioned and j is returned as it is the position of the pivot

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

partition!: Problem Analysis

- Consider partitioning the following vector interval when the pivot is 7:

V:	low	4	-3	87	10	31	24	78	high

- The values for i and j, respectively, are low + 2 and low + 1
- [low+2..low+1] is empty
- The elements are partitioned
- low + 1 is the position of the pivot

In-Place Operations

partition!: Problem Analysis

- Consider partitioning the following vector interval when the pivot is 10:

	low					high	
V:	9	11	87	0	4	15	-3

In-Place Operations

partition!: Problem Analysis

- Consider partitioning the following vector interval when the pivot is 10:

	low					high	
V:	9	11	87	0	4	15	-3

•

	low	i			j	high	
V:	9	11	87	0	4	15	-3

In-Place Operations

partition!: Problem Analysis

- Consider partitioning the following vector interval when the pivot is 10:

	low						high
V:	9	11	87	0	4	15	-3

•

	low	i				j	high
V:	9	11	87	0	4	15	-3

•

	low					high	
V:	9	-3	87	0	4	15	11

In-Place Operations

partition!: Problem Analysis

- Consider partitioning the following vector interval when the pivot is 10:

	low						high
V:	9	11	87	0	4	15	-3

•

	low	i					high
V:	9	11	87	0	4	15	-3

•

	low					high	
V:	9	-3	87	0	4	15	11

•

	low	i	j			high	
V:	9	-3	87	0	4	15	11

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

partition!: Problem Analysis

•

V:	9	-3	87	0	4	15	11	low	i	j	high
•											

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

partition!: Problem Analysis

•

V:	9	-3	87	0	4	15	11
	low	i		j		high	

•

V:	9	-3	4	0	87	15	11
	low			high			

In-Place Operations

partition!: Problem Analysis

- | | | | | | | | | |
|----|---|----|-------|-----|---|-----|----|--------|
| | | | low | i | | j | | $high$ |
| V: | 9 | -3 | 87 | 0 | 4 | 15 | 11 | |
- | | | | | | | | | |
|----|---|----|-------|---|----|--------|----|--|
| | | | low | | | $high$ | | |
| V: | 9 | -3 | 4 | 0 | 87 | 15 | 11 | |
- | | | | | | | | | |
|----|---|----|-------|-----|-----|--------|----|--|
| | | | low | j | i | $high$ | | |
| V: | 9 | -3 | 4 | 0 | 87 | 15 | 11 | |

In-Place Operations

partition!:

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; number [int int] → natnum
;; Purpose: Return the position of the pivot in the
;;           sorted V
;; How: The smallest index of a vector element > pivot
;;       and the largest index of an element ≤ to the
;;       pivot are found. If they form an empty vector
;;       interval the largest index of an element ≤ to
;;       the pivot is returned. Otherwise, the two
;;       indexed values are swapped and the partitioning
;;       process continues with the vector interval
;;       formed by the two indices.
;; Effect: V's elements are rearranged so that all
;;           elements ≤ to the pivot are at the beginning
;;           of the vector interval and all elements > the
;;           pivot are at the end of the vector interval.
(define (partition! pivot low high)
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

partition!: Function Body

```
(local [(define first>pivot  (find> pivot low high))
         (define first<=pivot (find<= pivot low high))]
  (if (> first>pivot first<=pivot)
      first<=pivot
      (begin
        (swap! first>pivot first<=pivot)
        (partition! pivot
                    first>pivot
                    first<=pivot))))
```

In-Place Operations

partition!: Termination Argument

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

;; Termination Argument

;; Every recursive call is made with a smaller vector
;; interval that does not contain the beginning numbers
;; <= to the pivot and the ending numbers > pivot.
;; Eventually, the recursive call is made with an empty
;; vector interval and the function halts.

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

`first<=:` Problem Analysis

- Search the given vector interval for the largest index of a number less than or equal to the given pivot

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

`first<=: Problem Analysis`

- Search the given vector interval for the largest index of a number less than or equal to the given pivot
- Traverse the given vector interval from right to left

In-Place Operations

first \leq : Problem Analysis

- Search the given vector interval for the largest index of a number less than or equal to the given pivot
- Traverse the given vector interval from right to left
- To guarantee that a valid index into V is returned this function assumes that $V[low]$ is the pivot
- In this manner, the search is always successful
- This assumption is safe because `partition!` calls this function with a vector interval that starts with the index for the pivot.

In-Place Operations

first \leq : Problem Analysis

- Search the given vector interval for the largest index of a number less than or equal to the given pivot
- Traverse the given vector interval from right to left
- To guarantee that a valid index into V is returned this function assumes that $V[low]$ is the pivot
- In this manner, the search is always successful
- This assumption is safe because `partition!` calls this function with a vector interval that starts with the index for the pivot.
- If the given vector interval is not empty then there are two cases that must be distinguished:
 - If $V[high]$ is less than or equal to the given pivot then $high$ is returned
 - Otherwise, the search continues with the rest of the vector interval

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

partition!

- ```
;; number [int int] → natnum
;; Purpose: Find the largest index, i, such that
;; V[i] ≤ to the given pivot
;; Assumption: V[low] = pivot
(define (find≤= pivot low high))
```

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

# In-Place Operations

## partition!

- ```
;; number [int int] → natnum
;; Purpose: Find the largest index, i, such that
;;           V[i] ≤ to the given pivot
;; Assumption: V[low] = pivot
(define (find≤= pivot low high)
  (if (or (> low high)
          (≤= (vector-ref V high) pivot))
      high
      (find≤= pivot low (sub1 high))))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

first>: Problem Analysis

- Find, if it exists, the smallest index, i , in the given vector interval such that $V[i]$ is greater than the given pivot

In-Place Operations

first>: Problem Analysis

- Find, if it exists, the smallest index, i , in the given vector interval such that $v[i]$ is greater than the given pivot
- Traverse the given interval from left to right

In-Place Operations

`first>: Problem Analysis`

- Find, if it exists, the smallest index, i , in the given vector interval such that $v[i]$ is greater than the given pivot
- Traverse the given interval from left to right
- If vector interval is empty low is greater than high
- There are no elements that are greater than the given pivot
- The elements are partitioned
- A value that makes `[first>pivot..find<= pivot]` in `partition!` empty must be returned
- Such a value is low.

In-Place Operations

`first>: Problem Analysis`

- Find, if it exists, the smallest index, i , in the given vector interval such that $V[i]$ is greater than the given pivot
- Traverse the given interval from left to right
- If vector interval is empty low is greater than high
- There are no elements that are greater than the given pivot
- The elements are partitioned
- A value that makes `[first>pivot..find<= pivot]` in `partition!` empty must be returned
- Such a value is low.
- If the given interval is not empty then there are two cases:
 - If $V[low]$ is greater than the given pivot then low is the smallest index of a number greater than the given pivot and it is returned
 - Otherwise, the search continues with the rest of the vector interval

In-Place Operations

first>

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- ```
;; number [int int] → natnum
;; Purpose: If it exists, find smallest index \geq low,
;; i, such that V[i] > given pivot if it exists.
;; Otherwise, return a value greater than high.
(define (find> pivot low high)
```

# In-Place Operations

first>

Sharing Values

Mutation  
Sequencing

Vectors

In-Place  
Operations

The Chicken  
and the Egg  
Paradox

Epilogue

- ```
;; number [int int] → natnum
;; Purpose: If it exists, find smallest index  $\geq$  low,
;;           i, such that V[i] > given pivot if it exists.
;;           Otherwise, return a value greater than high.
(define (find> pivot low high)
  (if (or (> low high)
          (> (vector-ref V low) pivot))
      low
      (find> pivot (add1 low) high)))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Quick Sorting

- `swap!` previously designed
- Run the tests and make sure they pass.

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

- The complexity of quick sorting is $O(n^2)$

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

- The complexity of quick sorting is $O(n^2)$
- Recall that the problem is a linear traversal when elements are sorted or in reversed sorted order
- Takes the algorithm from $O(n * \lg(n))$ to $O(n^2)$

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

- The complexity of quick sorting is $O(n^2)$
- Recall that the problem is a linear traversal when elements are sorted or in reversed sorted order
- Takes the algorithm from $O(n * \lg(n))$ to $O(n^2)$
- At each sorting step we would like to, at most, traverse half of the remaining elements to sort
- If this is always the case then sorting is made $O(n * \lg(n))$.
- This can only be guaranteed if the elements must have a relative order to each other

In-Place Operations

Heaps

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- The elements to sort may be represented as a *heap*
- A heap is a binary tree in which only the last two levels may have empty subheaps
- If a heap is not empty then the root value is larger than or equal to the root value of either subheap

In-Place Operations

Heaps

Sharing Values

Mutation
Sequencing

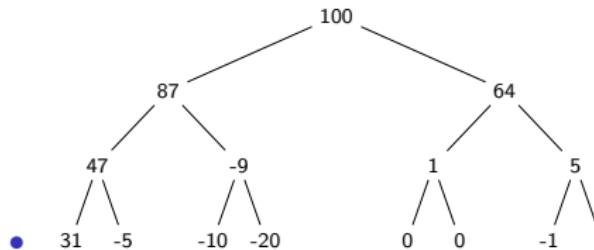
Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- The elements to sort may be represented as a *heap*
- If a heap is not empty then the root value is larger than or equal to the root value of either subheap



In-Place Operations

Heaps

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A heap is either:
 1. empty
 2. (list number heap heap)

In-Place Operations

Heaps

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A heap is either:
 1. empty
 2. (list number heap heap)
- How is a heap constructed?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

Heaps

- A heap is either:
 1. empty
 2. (list number heap heap)
- How is a heap constructed?
- How are heaps used to sort?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

- The maximum element is at the root of the heap

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

- The maximum element is at the root of the heap
- The root of the heap may be removed and added as the next largest element to the result of sorting

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

- The maximum element is at the root of the heap
- The root of the heap may be removed and added as the next largest element to the result of sorting
- The root is substituted with the rightmost leaf at the deepest level

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

- The maximum element is at the root of the heap
- The root of the heap may be removed and added as the next largest element to the result of sorting
- The root is substituted with the rightmost leaf at the deepest level
- Substituting the root means that the heap must be restored by trickling the new root value down the binary tree until the heap is restored

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

- The maximum element is at the root of the heap
- The root of the heap may be removed and added as the next largest element to the result of sorting
- The root is substituted with the rightmost leaf at the deepest level
- Substituting the root means that the heap must be restored by trickling the new root value down the binary tree until the heap is restored
- A root value must be trickled down when it is not larger than the root of any subheaps below
- Exchange with largest child
- Trickling down continues until it is larger than any roots below it or there are no nonempty subheaps below it.

In-Place Operations

In-Place Heap Sorting

Sharing Values

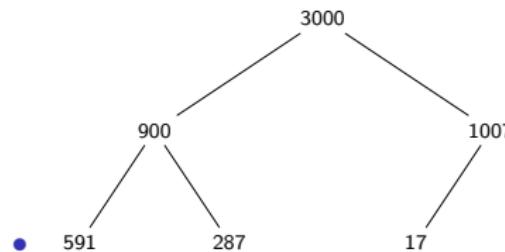
Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue



Sharing Values

Mutation
Sequencing

Vectors

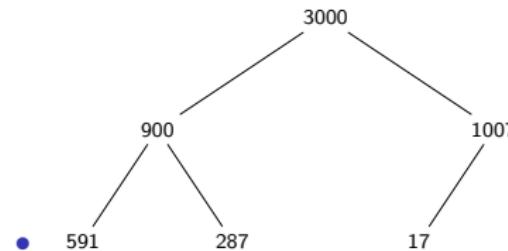
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting



- Result: (3000)

Sharing Values

Mutation
Sequencing

Vectors

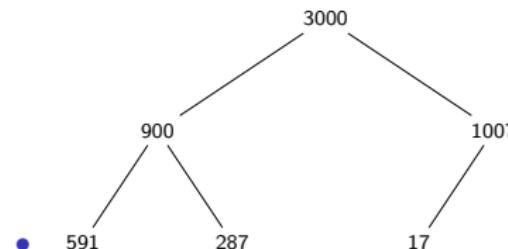
In-Place
Operations

The Chicken
and the Egg
Paradox

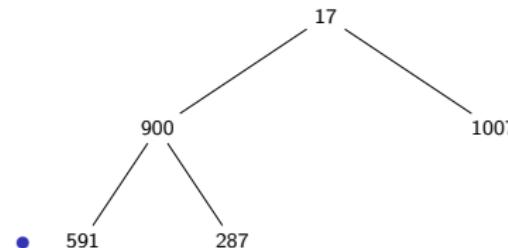
Epilogue

In-Place Operations

In-Place Heap Sorting



- Result: (3000)



In-Place Operations

In-Place Heap Sorting

Sharing Values

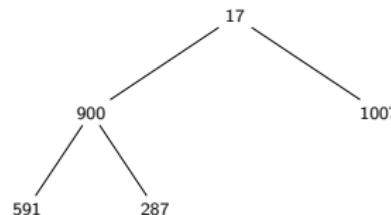
Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue



In-Place Operations

In-Place Heap Sorting

Sharing Values

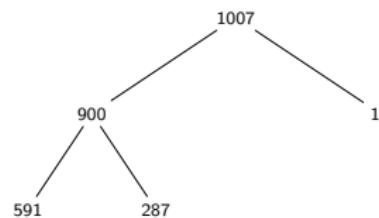
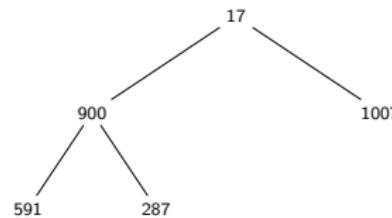
Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue



Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

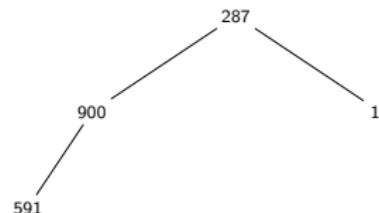
The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (1007 3000)



Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

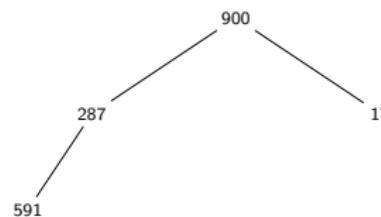
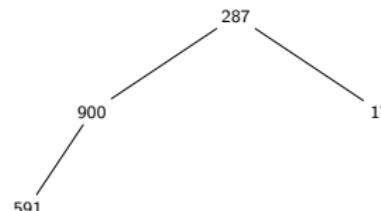
The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (1007 3000)



Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

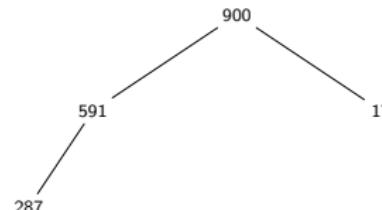
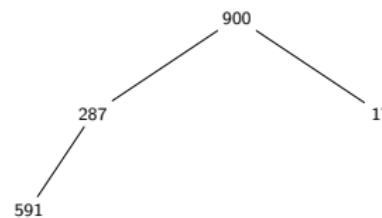
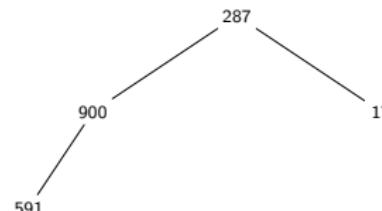
The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (1007 3000)



Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (900 1007 3000)

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

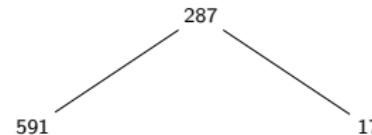
The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (900 1007 3000)



Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

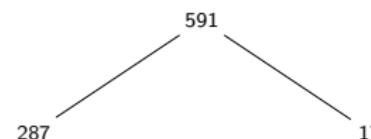
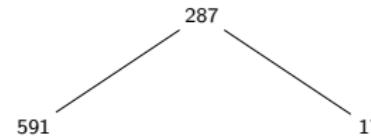
The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (900 1007 3000)



Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (591 900 1007 3000)

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (591 900 1007 3000)



Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (591 900 1007 3000)



Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (591 900 1007 3000)



(287 591 900 1007 3000)

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (591 900 1007 3000)



(287 591 900 1007 3000)

17

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

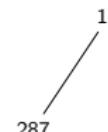
The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

In-Place Heap Sorting

Result: (591 900 1007 3000)



(287 591 900 1007 3000)

17

(17 287 591 900 1007 3000) The heap is empty and the process stops

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

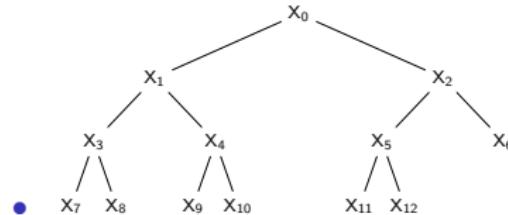
Mapping a Heap to a Vector

- heap elements numbered level by level and from left to right

In-Place Operations

Mapping a Heap to a Vector

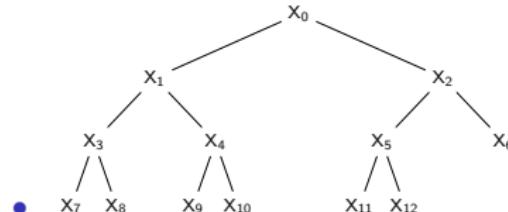
- heap elements numbered level by level and from left to right



In-Place Operations

Mapping a Heap to a Vector

- heap elements numbered level by level and from left to right



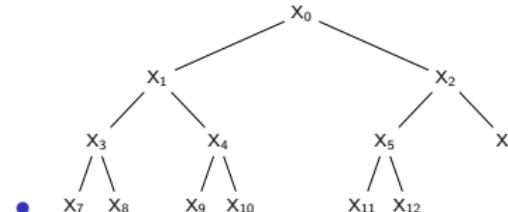
- The vector elements are placed in the vector as follows:

x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	...
----	----	----	----	----	----	----	----	----	----	-----	-----	-----

In-Place Operations

Mapping a Heap to a Vector

- heap elements numbered level by level and from left to right



- The vector elements are placed in the vector as follows:

X ₀	X ₁	X ₂	X ₃	X ₄	X ₅	X ₆	X ₇	X ₈	X ₉	X ₁₀	X ₁₁	...
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	-----------------	-----------------	-----

- ;; natnum → natnum Purpose: Return index of left root
(define (left-heap-root parent-index) (+ 1 (* 2 parent-index)))
- ;; natnum → natnum Purpose: Return of right root
(define (right-heap-root parent-index) (+ (* 2 parent-index) 2))
- ;; natnum → natnum Purpose: Return the parent index in the heap
;; Assumption: the given index is in [1..(sub1 (vector-length V))]
(define (parent i)
 (if (even? i)
 (quotient (sub1 i) 2)
 (quotient i 2))))

In-Place Operations

heap-sort-in-place!: Problem Analysis

- Two steps:

- Mutates given vector into a heap
- Sort the vector that represents

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heap-sort-in-place!: Problem Analysis

- Two steps:
 - Mutates given vector into a heap
 - Sort the vector that represents
- Think about the vector as being divided in two parts:
 - The elements that must still be sorted
 - The elements already sorted That is, it contains the larger elements that are already in the correct place

In-Place Operations

heap-sort-in-place!: Problem Analysis

- Two steps:
 - Mutates given vector into a heap
 - Sort the vector that represents
- Think about the vector as being divided in two parts:
 - The elements that must still be sorted
 - The elements already sorted That is, it contains the larger elements that are already in the correct place
- Sorting function needs the indices of the elements that still need to be sorted

In-Place Operations

heap-sort-in-place!: Problem Analysis

- Two steps:
 - Mutates given vector into a heap
 - Sort the vector that represents
- Think about the vector as being divided in two parts:
 - The elements that must still be sorted
 - The elements already sorted That is, it contains the larger elements that are already in the correct place
- Sorting function needs the indices of the elements that still need to be sorted
- Initially, this vector interval includes the indices for all the elements in the given vector.

In-Place Operations

heap-sort-in-place!: Problem Analysis

- Two steps:
 - Mutates given vector into a heap
 - Sort the vector that represents
- Think about the vector as being divided in two parts:
 - The elements that must still be sorted
 - The elements already sorted That is, it contains the larger elements that are already in the correct place
- Sorting function needs the indices of the elements that still need to be sorted
- Initially, this vector interval includes the indices for all the elements in the given vector.
- To mutate a vector to represent a heap elements must be rearranged such that a parent value is always larger than its children's value

In-Place Operations

heap-sort-in-place!: Problem Analysis

- Two steps:
 - Mutates given vector into a heap
 - Sort the vector that represents
- Think about the vector as being divided in two parts:
 - The elements that must still be sorted
 - The elements already sorted That is, it contains the larger elements that are already in the correct place
- Sorting function needs the indices of the elements that still need to be sorted
- Initially, this vector interval includes the indices for all the elements in the given vector.
- To mutate a vector to represent a heap elements must be rearranged such that a parent value is always larger than its children's value
- All children values must be processed

In-Place Operations

heap-sort-in-place!: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Two steps:
 - Mutates given vector into a heap
 - Sort the vector that represents
- Think about the vector as being divided in two parts:
 - The elements that must still be sorted
 - The elements already sorted That is, it contains the larger elements that are already in the correct place
- Sorting function needs the indices of the elements that still need to be sorted
- Initially, this vector interval includes the indices for all the elements in the given vector.
- To mutate a vector to represent a heap elements must be rearranged such that a parent value is always larger than its children's value
- All children values must be processed
- The only value that does not have a parent and, therefore, is not a child is the root

In-Place Operations

heap-sort-in-place!: Problem Analysis

- Two steps:
 - Mutates given vector into a heap
 - Sort the vector that represents
- Think about the vector as being divided in two parts:
 - The elements that must still be sorted
 - The elements already sorted That is, it contains the larger elements that are already in the correct place
- Sorting function needs the indices of the elements that still need to be sorted
- Initially, this vector interval includes the indices for all the elements in the given vector.
- To mutate a vector to represent a heap elements must be rearranged such that a parent value is always larger than its children's value
- All children values must be processed
- The only value that does not have a parent and, therefore, is not a child is the root
- Initial vector interval for unprocessed children is from 1 to the length of the given vector minus 1.

In-Place Operations

heap-sort-in-place!: Signature, Statements, and Function Header

```
;; (vectorof number) → (void)
;; Purpose: Sort the given vector in nondecreasing order
;; Effect: The given vector's elements are rearranged in
;;         nondecreasing order
(define (heap-sort-in-place! V)
```

In-Place Operations

heap-sort-in-place!: Tests

```
(check-expect (begin
                  (heap-sort-in-place! V0)
                  V0)
                  (vector))

(check-expect (begin
                  (heap-sort-in-place! V1)
                  V1)
                  (vector 3 7 10 11 17))

(check-expect (begin
                  (heap-sort-in-place! V2)
                  V2)
                  (vector 12 20 22 22 23 27 31 37 44 46
                         60 60 62 67 74 75 77 80 85 86))

(check-expect (begin
                  (heap-sort-in-place! V3)
                  V3)
                  (vector 0 1 2 3 4 5 6 7 8 9))

(check-expect (begin
                  (heap-sort-in-place! V4)
                  V4)
                  (vector 0 1 2 3 4 5 6 7 8 9))
```

In-Place Operations

heap-sort-in-place!: Function Body

```
(local [;; natnum natnum → (void) Purpose: Swap the elements at the given indices
       ;; Effect: V is mutated by swapping elements at given indices
       (define (swap! i j)
         (local [(define temp (vector-ref V i))]
           (begin
             (vector-set! V i (vector-ref V j))
             (vector-set! V j temp))))
       ;; natnum → natnum
       ;; Purpose: Return the index for the right subheap's root
       (define (right-heap-root parent-index)
         (+ (* 2 parent-index) 2))
       ;; natnum → natnum
       ;; Purpose: Return the index for the left subhead's root
       (define (left-heap-root parent-index)
         (+ add1 (* 2 parent-index)))
       ;; natnum → natnum Purpose: Return the parent index in the heap
       ;; Assumption: the given index is in [1..(sub1 (vector-length V))]
       (define (parent i)
         (if (even? i)
             (quotient (sub1 i) 2)
             (quotient i 2)))
       ;; ... Purpose:
       (define (sorter! low high) ...)
       ;; ... Purpose:
       (define (heapify! low high) ...
         .
         .
         .
         :     ])
```

In-Place Operations

heap-sort-in-place!: Function Body

```
(local [;; natnum natnum → (void) Purpose: Swap the elements at the given indices
       ;; Effect: V is mutated by swapping elements at given indices
       (define (swap! i j)
           (local [(define temp (vector-ref V i))]
               (begin
                   (vector-set! V i (vector-ref V j))
                   (vector-set! V j temp))))
       ;; natnum → natnum
       ;; Purpose: Return the index for the right subheap's root
       (define (right-heap-root parent-index)
           (+ (* 2 parent-index) 2))
       ;; natnum → natnum
       ;; Purpose: Return the index for the left subhead's root
       (define (left-heap-root parent-index)
           (add1 (* 2 parent-index)))
       ;; natnum → natnum Purpose: Return the parent index in the heap
       ;; Assumption: the given index is in [1..(sub1 (vector-length V))]
       (define (parent i)
           (if (even? i)
               (quotient (sub1 i) 2)
               (quotient i 2)))
       ;; ... Purpose:
       (define (sorter! low high) ...)
       ;; ... Purpose:
       (define (heapify! low high) ...
             .
             .
             ]
             (begin
                 (heapify! 1 (sub1 (vector-length V)))
                 (sorter! 0 (sub1 (vector-length V)))))))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

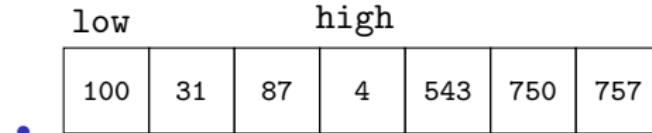
sorter!: Problem Analysis

- Think of the vector as divided into two parts: sorted and unsorted

In-Place Operations

sorter!: Problem Analysis

- Think of the vector as divided into two parts: sorted and unsorted



The two parts of the vector are described as follows:

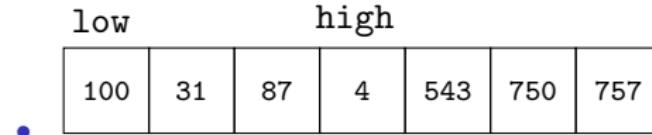
$V[low..high]$ is a heap of unsorted elements

$V[high+1..(sub1 (vector-length V))]$ contains V 's largest numbers in nondecreasing order

In-Place Operations

sorter!: Problem Analysis

- Think of the vector as divided into two parts: sorted and unsorted



The two parts of the vector are described as follows:

$V[low..high]$ is a heap of unsorted elements

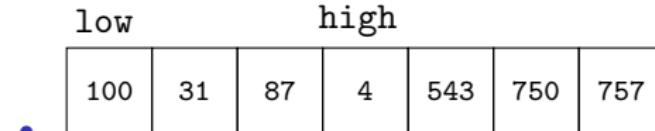
$V[high+1..(sub1 (vector-length V))]$ contains V 's largest numbers in nondecreasing order

- Process the part containing the heap from right to left
- At each step high is added to the vector interval indexing the sorted elements

In-Place Operations

sorter!: Problem Analysis

- Think of the vector as divided into two parts: sorted and unsorted



The two parts of the vector are described as follows:

`V[low..high]` is a heap of unsorted elements

`V[high+1..(sub1 (vector-length V))]` contains V's largest numbers in nondecreasing order

- Process the part containing the heap from right to left
- At each step high is added to the vector interval indexing the sorted elements
- If the given vector interval is empty (`void`) is returned
- If the vector interval for the unsorted elements, `[low..high]`, is not empty
 - Swap `V[low]` with `V[high]`
 - `V[high..(sub1 (vector-length V))]` contains V's largest elements in nondecreasing order
 - Trickle down the new root value to restore the heap
 - Continue with the rest of the vector interval

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg

Paradox

Epilogue

In-Place Operations

sorter!: Signature, Statements, and Function Header

```
;; [int int] → (void)
;; Purpose: For the given VINTV, sort the vector elements
;; Effect: V's elements in the given VINTV are rearranged
;;           in nondecreasing order
;; Assumption: V[low..high] is a heap
(define (sorter! low high)
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

sorter!: Function Body

```
(cond [(> low high) (void)]
      [else (begin
              (swap! low high)
              (trickle-down! low (sub1 high))
              (sorter! low (sub1 high))))])
```

In-Place Operations

trickle-down!: Problem Analysis

- Restores a heap in rooted at low stored in V[low..high]

In-Place Operations

trickle-down!: Problem Analysis

- Restores a heap in rooted at low stored in V[low..high]
- Assume that all the elements in [low+1..high] are heap roots

In-Place Operations

trickle-down!: Problem Analysis

- Restores a heap in rooted at low stored in V[low..high]
- Assume that all the elements in [low+1..high] are heap roots
- Swap the root element with the largest, if any, subheap root

In-Place Operations

trickle-down!: Problem Analysis

- Restores a heap in rooted at low stored in $V[\text{low}..\text{high}]$
- Assume that all the elements in $[\text{low}+1..\text{high}]$ are heap roots
- Swap the root element with the largest, if any, subheap root
- The trickling down process continues with the mutated subheap

In-Place Operations

trickle-down!: Problem Analysis

- Restores a heap in rooted at low stored in $V[low..high]$
- Assume that all the elements in $[low+1..high]$ are heap roots
- Swap the root element with the largest, if any, subheap root
- The trickling down process continues with the mutated subheap
- Stops when $V[low]$ is the root of a heap

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

trickle-down!: Problem Analysis

- Restores a heap in rooted at low stored in $V[\text{low}..\text{high}]$
- Assume that all the elements in $[\text{low}+1..\text{high}]$ are heap roots
- Swap the root element with the largest, if any, subheap root
- The trickling down process continues with the mutated subheap
- Stops when $V[\text{low}]$ is the root of a heap
- The potential indices for the left child, $lc\text{-index}$, and for the right child, $rc\text{-index}$, are locally defined

In-Place Operations

trickle-down!: Problem Analysis

- Restores a heap in rooted at low stored in $V[low..high]$
- Assume that all the elements in $[low+1..high]$ are heap roots
- Swap the root element with the largest, if any, subheap root
- The trickling down process continues with the mutated subheap
- Stops when $V[low]$ is the root of a heap
- The potential indices for the left child, $lc\text{-index}$, and for the right child, $rc\text{-index}$, are locally defined
- Determine how many children the root, $V[low]$:
 - left-child index $> high$ then the root has no children and $V[low]$ is the root of a heap and (void) is returned
 - If only the right-child index $> high$ then the root only has a left child
 - Determine if $V[lc\text{-index}] < V[low]$
 - If not return (void)
 - Otherwise, $V[low]$ and $V[lc\text{-index}]$ are swapped and the trickling down process continues with $[lc\text{-index}..high]$.
 - If the root has two children
 - Determine, $mc\text{-index}$, maximum subheap root
 - If not out of order return (void)
 - Otherwise, swap it with the root and the trickling down process continues with $[mc\text{-index}..high]$

In-Place Operations

trickle-down!: Function Definition

- ```
;; trickle-down!: [int int] → (void)
;; Purpose: For the given VINTV, reestablish a heap rooted at low
;; Effect: Vector elements are rearranged to have a heap rooted at low
;; Assumption: V[low+1..high] are all heap roots
(define (trickle-down! low high))
```

# In-Place Operations

## trickle-down!: Function Definition

- ```
;; trickle-down!: [int int] → (void)
;; Purpose: For the given VINTV, reestablish a heap rooted at low
;; Effect: Vector elements are rearranged to have a heap rooted at low
;; Assumption: V[low+1..high] are all heap roots
(define (trickle-down! low high))
```
- ```
(local [(define rc-index (right-heap-root low))
 (define lc-index (left-heap-root low))])
```

# In-Place Operations

## trickle-down!: Function Definition

- ```
;; trickle-down!: [int int] → (void)
;; Purpose: For the given VINTV, reestablish a heap rooted at low
;; Effect: Vector elements are rearranged to have a heap rooted at
;; Assumption: V[low+1..high] are all heap roots
(define (trickle-down! low high))
```
- ```
(local [(define rc-index (right-heap-root low))
 (define lc-index (left-heap-root low))])
```
- ```
(cond [(> lc-index high) (void)];; root has no children
```

In-Place Operations

trickle-down!: Function Definition

- `;; trickle-down!: [int int] → (void)`
;; Purpose: For the given VINTV, reestablish a heap rooted at low
;; Effect: Vector elements are rearranged to have a heap rooted at low
;; Assumption: V[low+1..high] are all heap roots
`(define (trickle-down! low high)`
- `(local [(define rc-index (right-heap-root low))
 (define lc-index (left-heap-root low))])`
- `(cond [(> lc-index high) (void)];; root has no children`
- `[(> rc-index high) ;; root only has a left child
 (if (<= (vector-ref V lc-index) (vector-ref V low))
 (void)
 (begin (swap! low lc-index)
 (trickle-down! lc-index high))))]`

In-Place Operations

trickle-down!: Function Definition

- `;; trickle-down!: [int int] → (void)`
;; Purpose: For the given VINTV, reestablish a heap rooted at low
;; Effect: Vector elements are rearranged to have a heap rooted at low
;; Assumption: V[low+1..high] are all heap roots
`(define (trickle-down! low high)`
- `(local [(define rc-index (right-heap-root low))
 (define lc-index (left-heap-root low))])`
- `(cond [(> lc-index high) (void)];; root has no children`
- `[(> rc-index high) ;; root only has a left child
 (if (<= (vector-ref V lc-index) (vector-ref V low))
 (void)
 (begin (swap! low lc-index)
 (trickle-down! lc-index high))))]`
- `[else ;; root has two children
 (local
 [(define mc-index (if (>= (vector-ref V lc-index)
 (vector-ref V rc-index))
 lc-index
 rc-index))]
 (cond [(>= (vector-ref V low) (vector-ref V mc-index))
 (void)]
 [else (begin
 (swap! low mc-index)
 (trickle-down! mc-index high))]))])])])`

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

trickle-down!: Termination Argument

A recursive call is only made when $V[\text{low}]$ has 1 or 2 children. That is, the index of any child must be less than or equal to high . Every recursive call is made with an interval formed by a child index and high . Eventually, the given vector interval becomes empty or $V[\text{low}]$ is a heap root and the mutator terminates.

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heapify!: Problem Analysis

- Rearrange V's elements into a heap rooted at low.

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heapify!: Problem Analysis

- Rearrange V's elements into a heap rooted at low.
- If a parent element is less than a child element then the parent and the child elements are swapped

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heapify!: Problem Analysis

- Rearrange V's elements into a heap rooted at low.
- If a parent element is less than a child element then the parent and the child elements are swapped
- Process from high to low

Sharing Values

Mutation
Sequencing

Vectors

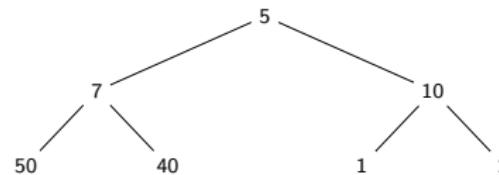
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heapify!: Problem Analysis



Sharing Values

Mutation
Sequencing

Vectors

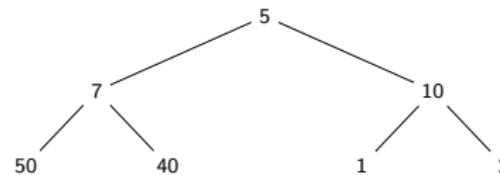
In-Place
Operations

The Chicken
and the Egg
Paradox

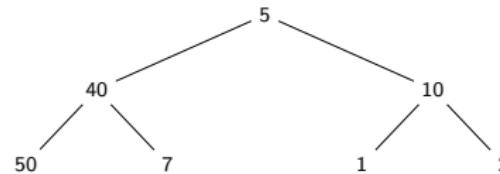
Epilogue

In-Place Operations

heapify!: Problem Analysis



2 and 1 OK; swap 7 and 40



Sharing Values

Mutation
Sequencing

Vectors

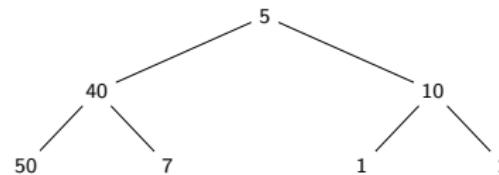
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heapify!: Problem Analysis



Sharing Values

Mutation
Sequencing

Vectors

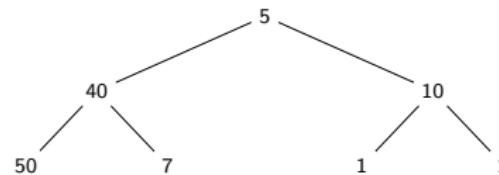
In-Place
Operations

The Chicken
and the Egg
Paradox

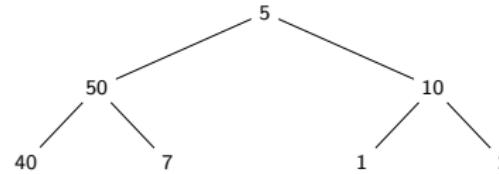
Epilogue

In-Place Operations

heapify!: Problem Analysis



$50 > 40$ so they are swapped



Sharing Values

Mutation
Sequencing

Vectors

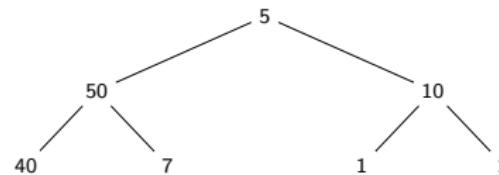
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heapify!: Problem Analysis



In-Place Operations

heapify!: Problem Analysis

Sharing Values

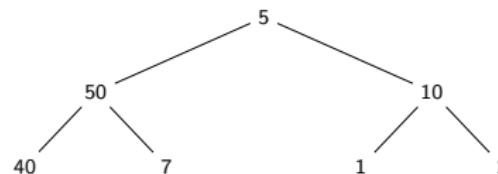
Mutation
Sequencing

Vectors

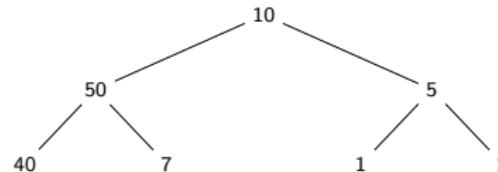
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue



$10 > 5$ so swap them; no need to trickle down



Sharing Values

Mutation
Sequencing

Vectors

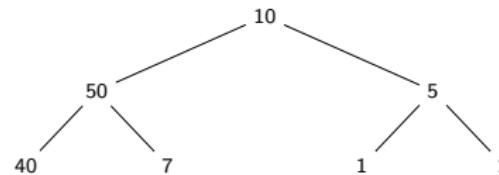
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heapify!: Problem Analysis



Sharing Values

Mutation
Sequencing

Vectors

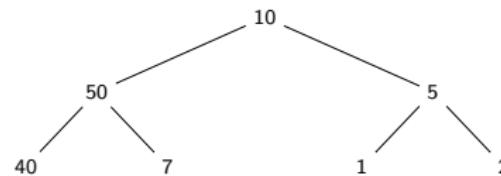
In-Place
Operations

The Chicken
and the Egg
Paradox

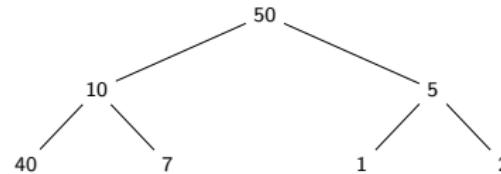
Epilogue

In-Place Operations

heapify!: Problem Analysis



$50 > 10$ so swap them; need to trickle down



Sharing Values

Mutation
Sequencing

Vectors

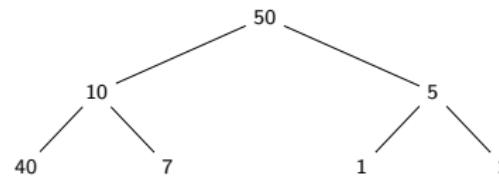
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heapify!: Problem Analysis



In-Place Operations

heapify!: Problem Analysis

Sharing Values

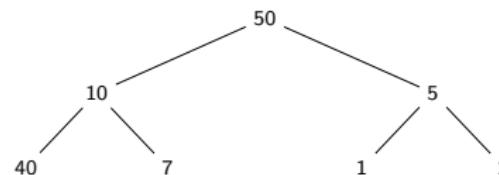
Mutation
Sequencing

Vectors

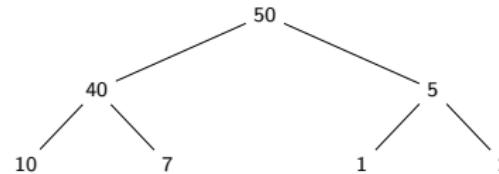
In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue



$40 > 10$ so swap them; no need to further trickle down



In-Place Operations

heapify!: Signature, Statements, and Function Header

```
;; heapify!: [int int] → (void)
;; Purpose: Transform the elements in the given VINTV
;;           into a heap
;; Effect: Rearrange the vector elements to form a heap
;;           rooted at low
;; Assumptions:
;;   low > 0 Given VINTV is valid for V
;;
;; Given VINTV is valid for V
;;
;; Elements indexed in [high+1..(sub1 (vector-length V))]
;; are heap roots
(define (heapify! low high)
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

heapify!: Function Body

```
(cond [(> low high) (void)]
      [else
       (local [(define parent-index (parent high))]
         (cond [(>= (vector-ref V parent-index)
                  (vector-ref V high))
                (heapify! low (sub1 high))])
         [else
          (begin
            (swap! parent-index high)
            (trickle-down! high
                           (sub1 (vector-length V)))
            (heapify! low (sub1 high))))]))])
```

In-Place Operations

`heapify!`: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: `heapify` and `sort`.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus `sorter`'s complexity.

In-Place Operations

`heapify!`: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: `heapify` and `sort`.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus `sorter`'s complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.

In-Place Operations

heapify!: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: heapify and sort.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus sorter's complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.
- Half of the elements are in the left subheap and half of the elements are in the right subheap

In-Place Operations

heapify!: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: heapify and sort.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus sorter's complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.
- Half of the elements are in the left subheap and half of the elements are in the right subheap
- The difference in height between the two subheaps is at most 1.

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

`heapify!`: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: `heapify` and `sort`.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus `sorter`'s complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.
- Half of the elements are in the left subheap and half of the elements are in the right subheap
- The difference in height between the two subheaps is at most 1.
- In the worst case, `trickle-down!` always performs a recursive to process a subheap half the size

In-Place Operations

heapify!: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: heapify and sort.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus sorter's complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.
- Half of the elements are in the left subheap and half of the elements are in the right subheap
- The difference in height between the two subheaps is at most 1.
- In the worst case, `trickle-down!` always performs a recursive to process a subheap half the size
- `trickle-down!`'s complexity $O(\lg(n))$

In-Place Operations

heapify!: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: heapify and sort.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus `sorter!`'s complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.
- Half of the elements are in the left subheap and half of the elements are in the right subheap
- The difference in height between the two subheaps is at most 1.
- In the worst case, `trickle-down!` always performs a recursive to process a subheap half the size
- `trickle-down!`'s complexity $O(\lg(n))$
- `sorter!`'s complexity is $O(n * \lg(n))$

In-Place Operations

heapify!: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: `heapify` and `sort`.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus `sorter`'s complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.
- Half of the elements are in the left subheap and half of the elements are in the right subheap
- The difference in height between the two subheaps is at most 1.
- In the worst case, `trickle-down!` always performs a recursive to process a subheap half the size
- `trickle-down!`'s complexity $O(\lg(n))$
- `sorter!`'s complexity is $O(n * \lg(n))$
- `heapify!` makes n recursive calls

In-Place Operations

heapify!: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: `heapify` and `sort`.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus `sorter`'s complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.
- Half of the elements are in the left subheap and half of the elements are in the right subheap
- The difference in height between the two subheaps is at most 1.
- In the worst case, `trickle-down!` always performs a recursive to process a subheap half the size
- `trickle-down!`'s complexity $O(\lg(n))$
- `sorter!`'s complexity is $O(n * \lg(n))$
- `heapify!` makes n recursive calls
- In the worst case every recursive call requires a call to `trickle-down!`

In-Place Operations

heapify!: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: `heapify` and `sort`.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus `sorter`'s complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.
- Half of the elements are in the left subheap and half of the elements are in the right subheap
- The difference in height between the two subheaps is at most 1.
- In the worst case, `trickle-down!` always performs a recursive to process a subheap half the size
- `trickle-down!`'s complexity $O(\lg(n))$
- `sorter!`'s complexity is $O(n * \lg(n))$
- `heapify!` makes n recursive calls
- In the worst case every recursive call requires a call to `trickle-down!`
- Its complexity is $O(n * \lg(n))$.

In-Place Operations

heapify!: Complexity

- The mutator, `heap-sort-in-place!`, performs two actions: `heapify` and `sort`.
- The complexity of the `heap-sort-in-place!` is `heapify!`'s complexity plus `sorter`'s complexity.
- The mutator `sorter!` is called with a vector interval of size n
- Makes n recursive calls
- In the worst case, for each recursive call two elements are swapped and a call to `trickle-down!` is made
- The complexity of `sorter!` is given by the product of n and the complexity of `trickle-down!`.
- Half of the elements are in the left subheap and half of the elements are in the right subheap
- The difference in height between the two subheaps is at most 1.
- In the worst case, `trickle-down!` always performs a recursive to process a subheap half the size
- `trickle-down!`'s complexity $O(\lg(n))$
- `sorter!`'s complexity is $O(n * \lg(n))$
- `heapify!` makes n recursive calls
- In the worst case every recursive call requires a call to `trickle-down!`
- Its complexity is $O(n * \lg(n))$.
- `heap-sort-in-place!`'s complexity is $O(n * \lg(n))$

In-Place Operations

heapify!: Performance

Is it better than quick sort when the vector is sorted?

```
(define N 10000)
(define TV (build-vector N (λ (i) i)))
;; (vectorof number)
;; Purpose: Vector to test quick sorting
(define TV1 (void))

;; (vectorof number)
;; Purpose: Vector to test heap sorting
(define TV2 (void))
```

In-Place Operations

heapify!: Performance

Is it better than quick sort when the vector is sorted?

```
(define N 10000)
(define TV (build-vector N (λ (i) i)))
;; (vectorof number)
;; Purpose: Vector to test quick sorting
(define TV1 (void))

;; (vectorof number)
;; Purpose: Vector to test heap sorting
(define TV2 (void))
```

The following experiment is executed 5 five times:

```
(begin
  (set! TV1 (build-vector N (λ (i) (vector-ref TV i))))
  (set! TV2 (build-vector N (λ (i) (vector-ref TV i))))
  (time (qs-in-place! TV1))
  (time (heap-sort-in-place! TV2)))
```

In-Place Operations

heapify!: Performance

Is it better than quick sort when the vector is sorted?

```
(define N 10000)
(define TV (build-vector N (λ (i) i)))
;; (vectorof number)
;; Purpose: Vector to test quick sorting
(define TV1 (void))

;; (vectorof number)
;; Purpose: Vector to test heap sorting
(define TV2 (void))
```

The following experiment is executed 5 five times:

```
(begin
  (set! TV1 (build-vector N (λ (i) (vector-ref TV i))))
  (set! TV2 (build-vector N (λ (i) (vector-ref TV i))))
  (time (qs-in-place! TV1))
  (time (heap-sort-in-place! TV2)))
```

CPU times:

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
quick sorting	2828	2984	3031	3187	3094	3024.8
heap sorting	47	31	46	31	31	37.2

In-Place Operations

FINAL EXAM PROJECT

```
;; natnum (listof ((vectorof number) → (void))) → (void)
;; Purpose: Run empirical study with multiples of 1000 vector lens in [1000..natnum*1000]
(define (empirical-study factor lst-of-sorters)
  (local
    [(define NUM-RUNS 5)
     (define V (build-vector (* factor 1000) (λ (i) (random 10000000))))
     ;; (vectorof number) natnum (listof ((vectorof number) → (void)))
     ;; → (void)
     ;; Purpose: Time the given sorters using the given vector the given number of times
     (define (run-experiments V runs sorters)
       (local [;; (listof ((vectorof number)→(void)))→(void)
              ;; Purpose: Run n experiments
              (define (run sorters)
                (if (empty? sorters) (void)
                    (local [;; (vectorof number) Purpose: Copy of V to sort
                           (define V1 (build-vector (vector-length V)
                                                     (λ (i) (vector-ref V i)))]
                     (begin
                       (display (format "Sorter ~s: " (add1 (- (length lst-of-sorters)
                                                               (length sorters)))))
                       (time ((first sorters) V1))
                       (run (rest sorters))))])
                  (if (= runs 0) (void)
                      (begin (display (format "      RUN ~s\n" (add1 (- NUM-RUNS runs))))
                             (run sorters)
                             (run-experiments V (sub1 runs) sorters))))))]
       (if (= factor 0) (void)
           (begin
             (display (format "Experiments for length ~s \n" (* factor 1000)))
             (run-experiments V NUM-RUNS lst-of-sorters)
             (newline) (newline)
             (empirical-study (sub1 factor) lst-of-sorters))))]))
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

FINAL EXAM PROJECT

Empirical data to determine if in-place quick sorting or in-place heap sorting is better in practice may be gathered as follows:

```
(empirical-study 15 (list qs-in-place! heap-sort-in-place!))
```

In-Place Operations

FINAL EXAM PROJECT



Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

- Radix sorting orders the numbers using their digits from least significant to most significant
- To sort the numbers by a given digit position 10 buckets, [0..9], are needed
- The numbers are placed in a bucket based on the value of the digit being processed.
- After all the numbers have been “bucketized” they are dumped back into the vector starting with bucket 0.

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

918	82	87	31	780	103	4
-----	----	----	----	-----	-----	---

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

918	82	87	31	780	103	4
-----	----	----	----	-----	-----	---

The numbers are bucketized based on the ones digit

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

918	82	87	31	780	103	4

The numbers are bucketized based on the ones digit

B0	780
B1	31
B2	82
B3	103
B4	4
B5	
B6	
B7	87
B8	918
B9	

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

918	82	87	31	780	103	4

The numbers are bucketized based on the ones digit

B0	780
B1	31
B2	82
B3	103
B4	4
B5	
B6	
B7	87
B8	918
B9	

The contents of each bucket is dumped back into the vector starting with the 0 bucket

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

918	82	87	31	780	103	4

The numbers are bucketized based on the ones digit

B0	780
B1	31
B2	82
B3	103
B4	4
B5	
B6	
B7	87
B8	918
B9	

The contents of each bucket is dumped back into the vector starting with the 0 bucket

780	31	82	103	4	87	918
-----	----	----	-----	---	----	-----

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

780	31	82	103	4	87	918
-----	----	----	-----	---	----	-----

The numbers are bucketized by the tens digit:

B0	103	4
B1	918	
B2		
B3	31	
B4		
B5		
B6		
B7		
B8	780	82
B9		87

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

780	31	82	103	4	87	918
-----	----	----	-----	---	----	-----

The numbers are bucketized by the tens digit:

B0	103	4
B1	918	
B2		
B3	31	
B4		
B5		
B6		
B7		
B8	780	82
B9		87

The numbers are dumped into the vector starting with bucket 0:

103	4	918	31	780	82	87
-----	---	-----	----	-----	----	----

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

103	4	918	31	780	82	87
-----	---	-----	----	-----	----	----

Numbers are bucketized by the hundreds digit:

B0	4	31	82	87
B1	103			
B2				
B3				
B4				
B5				
B6				
B7	780			
B8				
B9	918			

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

103	4	918	31	780	82	87
-----	---	-----	----	-----	----	----

Numbers are bucketized by the hundreds digit:

B0	4	31	82	87
B1	103			
B2				
B3				
B4				
B5				
B6				
B7	780			
B8				
B9	918			

The numbers are dumped into vector starting with bucket 0 to obtain:

4	31	82	87	103	780	918
---	----	----	----	-----	-----	-----

In-Place Operations

FINAL EXAM PROJECT: Radix Sort

103	4	918	31	780	82	87
-----	---	-----	----	-----	----	----

Numbers are bucketized by the hundreds digit:

B0	4	31	82	87
B1	103			
B2				
B3				
B4				
B5				
B6				
B7	780			
B8				
B9	918			

The numbers are dumped into vector starting with bucket 0 to obtain:

4	31	82	87	103	780	918
---	----	----	----	-----	-----	-----

There are no more digits to process and the sorting process stops

Observe that the vector is sorted.

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

In-Place Operations

FINAL EXAM PROJECT

THE EXAM QUESTIONS: 7-12

The Chicken and the Egg Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Which came first the chicken or the egg?

The Chicken and the Egg Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Which came first the chicken or the egg?
- Every chicken is born from an egg
- Every egg is laid by a chicken

The Chicken and the Egg Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Which came first the chicken or the egg?
- Every chicken is born from an egg
- Every egg is laid by a chicken
- The circular relationship between chickens and eggs makes it a problem to determine how chickens and eggs got started

The Chicken and the Egg Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- Which came first the chicken or the egg?
- Every chicken is born from an egg
- Every egg is laid by a chicken
- The circular relationship between chickens and eggs makes it a problem to determine how chickens and eggs got started
- Can the paradox be solved?

The Chicken and the Egg Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- In the world around us many things are related in a circular manner

The Chicken and the Egg Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- In the world around us many things are related in a circular manner
 - Bank clients and bank accounts
 - Employees and employers
 - Books and authors

The Chicken and the Egg Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- In the world around us many things are related in a circular manner
 - Bank clients and bank accounts
 - Employees and employers
 - Books and authors
- How can circular data be represented in a program?

The Chicken and the Egg Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- In the world around us many things are related in a circular manner
 - Bank clients and bank accounts
 - Employees and employers
 - Books and authors
- How can circular data be represented in a program?
- To make the discussion concrete the representation of a bank is used:
`A bank is a (listof client)`

The Chicken and the Egg Paradox

The Paradox in Programming

- #| A bank client is a structure:
(make-client string (listof account))
with a name and a nonempty list of accounts |#

(define-struct client (name accounts))

The Chicken and the Egg Paradox

The Paradox in Programming

- #| A bank client is a structure:
(make-client string (listof account))
with a name and a nonempty list of accounts |#

(define-struct client (name accounts))
- #| A bank account is a structure:
(make-account number client)
with a nonnegative balance and an owner |#

(define-struct account (balance owner))

The Chicken and the Egg Paradox

The Paradox in Programming

- #| A bank client is a structure:
(make-client string (listof account))
with a name and a nonempty list of accounts |#

(define-struct client (name accounts))
- #| A bank account is a structure:
(make-account number client)
with a nonnegative balance and an owner |#

(define-struct account (balance owner))
- How is a client created?

The Chicken and the Egg Paradox

The Paradox in Programming

- #| A bank client is a structure:
(make-client string (listof account))
with a name and a nonempty list of accounts |#
- (define-struct client (name accounts))
- #| A bank account is a structure:
(make-account number client)
with a nonnegative balance and an owner |#
- (define-struct account (balance owner))
- How is a client created?
- Just use make-client, right?

```
(define CLIENT1 (make-client
                    "Ada Lovelace"
                    (list (make-account
                            1000
                            CLIENT1)))) ← Oops!
```

The Chicken and the Egg Paradox

The Paradox in Programming

- #| A bank client is a structure:
(make-client string (listof account))
with a name and a nonempty list of accounts |#
- (define-struct client (name accounts))
- #| A bank account is a structure:
(make-account number client)
with a nonnegative balance and an owner |#
- (define-struct account (balance owner))
- How is a client created?
- Just use make-client, right?
(define CLIENT1 (make-client
"Ada Lovelace"
(list (make-account
1000
CLIENT1)))) ← Ooops!
- Well, create the account first
(define ACCT1 (make-account
1000
(make-client "Ada Lovelace"
(list ACCT1))) ← Ooops!

The Chicken and the Egg Paradox

The Paradox in Programming

- ```
#| A bank client is a structure:
 (make-client string (listof account))
 with a name and a nonempty list of accounts |#
```
- ```
(define-struct client (name accounts))
```
- ```
#| A bank account is a structure:
 (make-account number client)
 with a nonnegative balance and an owner |#
```
- ```
(define-struct account (balance owner))
```
- How is a client created?
- Just use make-client, right?

```
(define CLIENT1 (make-client  
                  "Ada Lovelace"  
                  (list (make-account  
                         1000  
                         CLIENT1)))) ← Ooops!
```
- Well, create the account first

```
(define ACCT1 (make-account  
                  1000  
                  (make-client "Ada Lovelace"  
                             (list ACCT1)))) ← Ooops!
```
- We have the chicken and the egg paradox in programming!

The Chicken and the Egg Paradox

Solving the Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A new type of constructor is needed

The Chicken and the Egg Paradox

Solving the Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A new type of constructor is needed
- A *generalized constructor* builds incorrect structure instances

The Chicken and the Egg Paradox

Solving the Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A new type of constructor is needed
- A *generalized constructor* builds incorrect structure instances
- Mutation is used to correct values in the instances

The Chicken and the Egg Paradox

Solving the Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A new type of constructor is needed
- A *generalized constructor* builds incorrect structure instances
- Mutation is used to correct values in the instances
- As problem solvers we need to decide which structure type is returned by a general constructor

The Chicken and the Egg Paradox

Solving the Paradox

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A new type of constructor is needed
- A *generalized constructor* builds incorrect structure instances
- Mutation is used to correct values in the instances
- As problem solvers we need to decide which structure type is returned by a general constructor
- A generalized constructor may be written for any structure in a circular dependency and later fields are mutated to correct the structure instances
- To build a client, the client's name and the initial balance of the first account may be used

The Chicken and the Egg Paradox

Client Constructor: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- To build a new client an incorrect client is constructed with an empty list of accounts

The Chicken and the Egg Paradox

Client Constructor: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- To build a new client an incorrect client is constructed with an empty list of accounts
- The incorrect client is used to build a new account

The Chicken and the Egg Paradox

Client Constructor: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- To build a new client an incorrect client is constructed with an empty list of accounts
- The incorrect client is used to build a new account
- Upon building the account the incorrect client's accounts field is mutated to be a list containing the constructed account
- The mutation corrects the client and the account

The Chicken and the Egg Paradox

Client Constructor: Sample Expressions and Differences

```
;; Sample expressions for build-new-client
(define HOPPER (local [(define new-client (make-client
                                         "Grace Hopper"
                                         '()))
                         (begin
                           (set-client-accounts! new-client
                                                 (list (make-account
                                                       847
                                                       new-client)))
                           new-client)]))
```

The Chicken and the Egg Paradox

Client Constructor: Sample Expressions and Differences

```
;; Sample expressions for build-new-client
(define HOPPER (local [(define new-client (make-client
                                              "Grace Hopper"
                                              '())))
                         (begin
                           (set-client-accounts! new-client
                                                 (list (make-account
                                                       847
                                                       new-client)))
                           new-client)))

(define RHODES (local [(define new-client (make-client
                                              "Ida Rhodes"
                                              '())))
                         (begin
                           (set-client-accounts! new-client
                                                 (list (make-account
                                                       1301
                                                       new-client)))
                           new-client)))
```

The Chicken and the Egg Paradox

Client Constructor: Signature, Statements, and
Function Header

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; string number → client
;; Purpose: To build a new client with given name
;;           and a single new account that has the
;;           given balance
;; Assumption: The given initial balance is positive
(define (build-new-client name init-balance)
```

The Chicken and the Egg Paradox

Client Constructor: Tests

```
;; Tests using sample computations for build-new-client  
(check-expect (build-new-client "Grace Hopper" 847) HOPPER)  
(check-expect (build-new-client "Ida Rhodes" 1301) RHODES)
```

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

The Chicken and the Egg Paradox

Client Constructor: Tests

```
; ; Tests using sample computations for build-new-client
(check-expect (build-new-client "Grace Hopper" 847) HOPPER)
(check-expect (build-new-client "Ida Rhodes" 1301) RHODES)
```

To write tests using sample values we need to understand a little about how ASL internally represents circular data

```
> (build-new-client "Ada Lovelace" 1000)
(shared ((-0- (make-client "Ada Lovelace"
                           (list (make-account 1000 -0-)))))

      -0-)
```

The Chicken and the Egg Paradox

Client Constructor: Tests

```
; ; Tests using sample computations for build-new-client  
(check-expect (build-new-client "Grace Hopper" 847) HOPPER)  
(check-expect (build-new-client "Ida Rhodes" 1301) RHODES)
```

To write tests using sample values we need to understand a little about how ASL internally represents circular data

```
> (build-new-client "Ada Lovelace" 1000)  
(shared ((-0- (make-client "Ada Lovelace"  
                           (list (make-account 1000 -0-))))  
         -0-))  
  
;; Tests using sample values for build-new-client  
(check-expect  
  (build-new-client "Ada Lovelace" 1000)  
  (shared ((-0- (make-client  
                  "Ada Lovelace"  
                  (list (make-account 1000 -0-))))  
         -0-)))  
  
(check-expect  
  (build-new-client "Mary Kenneth Keller" 431)  
  (shared ((-0- (make-client  
                  "Mary Kenneth Keller"  
                  (list (make-account 431 -0-))))))> ◀ ▶ ⏪ ⏩ 🔍
```

The Chicken and the Egg Paradox

Client Constructor: Function Body

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(local [(define new-client (make-client name '()))]
  (begin
    (set-client-accounts!
      new-client
      (list (make-account init-balance new-client)))
    new-client))
```

The Chicken and the Egg Paradox

Adding Clients to a Bank: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- A bank is represented as a state variable whose value is a list of clients
- Every time an account is constructed for a new client the bank is mutated to add the client

The Chicken and the Egg Paradox

Adding Clients to a Bank: State Variable Definition

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; (listof client)
;; Purpose: Store the clients of a bank
;; Assumption: Every client has a unique name
(define bank (void))
```

The Chicken and the Egg Paradox

Adding Clients to a Bank: bank initializer

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; → (void)
;; Purpose: Initialize bank
(define (initialize-bank!) (set! bank '()))
```

The Chicken and the Egg Paradox

`add-account!`: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- To add an account the client's name and the new account's initial balance are needed

The Chicken and the Egg Paradox

`add-account!`: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- To add an account the client's name and the new account's initial balance are needed
- The mutator must determine if the given client already exists

The Chicken and the Egg Paradox

add-account!: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- To add an account the client's name and the new account's initial balance are needed
- The mutator must determine if the given client already exists
- If so then a new account is constructed using the given initial balance and the existing client and the new account is added to the front of the existing client's account list

The Chicken and the Egg Paradox

add-account!: Problem Analysis

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

- To add an account the client's name and the new account's initial balance are needed
- The mutator must determine if the given client already exists
- If so then a new account is constructed using the given initial balance and the existing client and the new account is added to the front of the existing client's account list
- Otherwise, a new client is constructed using the given initial balance and the given name. The new client is then added to the front of bank.

The Chicken and the Egg Paradox

add-account!: Signature, Statements, and Function
Header

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
;; string number → (void)
;; Purpose: Add an account for the given client's name
;;           with the given initial balance
;; Effect: A new client, if necessary, is added to the
;;         front of bank and a new account is added to
;;         the front of the client's accounts
;; Assumption: The given balance is positive
(define (add-account! name balance)
```

The Chicken and the Egg Paradox

add-account!: Tests

Convert a bank into a list for testing purposes

```
; ; → (listof (cons string (listof number)))
; ; Purpose: Transform bank information into a list
(define (bank->list)
  (map (λ (client)
         (cons (client-name client)
               (map (λ (acct) (account-balance acct))
                    (client-accounts client))))
        bank))
```

The Chicken and the Egg Paradox

add-account!: Tests

Convert a bank into a list for testing purposes

```
;; → (listof (cons string (listof number)))
;; Purpose: Transform bank information into a list
(define (bank->list)
  (map (λ (client)
         (cons (client-name client)
               (map (λ (acct) (account-balance acct))
                     (client-accounts client))))
        bank))

;; Tests for bank->list
(check-expect (begin
                  (initialize-bank!)
                  (bank->list))
               '())

(check-expect
  (begin
    (set! bank (shared ((-1- (make-client
                                "Frances Allen"
                                (list (make-account 500 -1-)))))
                      (list -1-)))
    (bank->list))
  (list (list "Frances Allen" 500)))
```

The Chicken and the Egg Paradox

add-account!: Tests

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

```
(check-expect (begin
                  (initialize-bank!)
                  (bank->list))
                  ' ())

(check-expect (begin
                  (initialize-bank!)
                  (add-account! "Joan Clarke" 1000)
                  (bank->list))
                  (list (list "Joan Clarke" 1000)))

(check-expect (begin
                  (initialize-bank!)
                  (add-account! "Joan Clarke" 1000)
                  (add-account! "Katherine Johnson" 100)
                  (add-account! "Joan Clarke" 6500)
                  (add-account! "Jean E. Sammet" 830)
                  (bank->list))
                  (list (list "Jean E. Sammet" 830)
                        (list "Katherine Johnson" 100)
                        (list "Joan Clarke" 6500 1000)))
```

The Chicken and the Egg Paradox

add-account!: Function Body

```
(local [(define client-search (filter
                                (λ (c)
                                    (string=? name (client-name c)))
                                bank))

        ;; → client Purpose: To build a new client with name and
        ;;                      a single new account that has init-balance
(define (build-new-client)
  (local [(define new-client (make-client name '()))]
    (begin
      (set-client-accounts! new-client
                            (list (make-account balance
                                                new-client)))
      new-client))))]
```

The Chicken and the Egg Paradox

add-account!: Function Body

```
(local [(define client-search (filter
                                (λ (c)
                                    (string=? name (client-name c))))
                                bank))

        ;; → client Purpose: To build a new client with name and
        ;;                      a single new account that has init-balance
(define (build-new-client)
    (local [(define new-client (make-client name '()))]
        (begin
            (set-client-accounts! new-client
                (list (make-account balance
                    new-client)))
            new-client)))

(if (not (empty? client-search))
    (local [(define the-client (first client-search))
            (define new-acct (make-account balance the-client))]
        (set-client-accounts!
            the-client
            (cons new-acct (client-accounts the-client))))
    (set! bank (cons (build-new-client) bank))))
```

Run the tests and make sure they all pass

The Chicken and the Egg Paradox

Adding Clients to a Bank: bank initializer

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

HOMEWORK: 3, 4

Quiz: 2 (due in 1 week)

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Epilogue

- Congratulations! You have taken your next step into the exciting and wonderful world of problem solving and programming

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Epilogue

- Congratulations! You have taken your next step into the exciting and wonderful world of problem solving and programming
- Hopefully, you have felt your mind expand and your problem solving skills grow as you progressed

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Epilogue

- Congratulations! You have taken your next step into the exciting and wonderful world of problem solving and programming
- Hopefully, you have felt your mind expand and your problem solving skills grow as you progressed
- You are well-prepared to continue your journey honing problem-solving skills in the realm of programming and beyond

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Epilogue

- Congratulations! You have taken your next step into the exciting and wonderful world of problem solving and programming
- Hopefully, you have felt your mind expand and your problem solving skills grow as you progressed
- You are well-prepared to continue your journey honing problem-solving skills in the realm of programming and beyond
- There is still much to learn and the next steps ought to be as or more exciting

Epilogue

- Congratulations! You have taken your next step into the exciting and wonderful world of problem solving and programming
- Hopefully, you have felt your mind expand and your problem solving skills grow as you progressed
- You are well-prepared to continue your journey honing problem-solving skills in the realm of programming and beyond
- There is still much to learn and the next steps ought to be as or more exciting
- Remember that all good things must...continue!

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Epilogue

- Where should you go from here?

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Epilogue

- Where should you go from here?
- Many paths you may follow but make sure that you always design the solutions to problems

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Epilogue

- Where should you go from here?
- Many paths you may follow but make sure that you always design the solutions to problems
- Do not shy away from recursion

Epilogue

- Where should you go from here?
- Many paths you may follow but make sure that you always design the solutions to problems
- Do not shy away from recursion
- Always remember that mutation is like a hammer that clobbers everything it touches Sometimes problem solving requires more finesse and a screwdriver, not a hammer, ought to be used

Epilogue

- Where should you go from here?
- Many paths you may follow but make sure that you always design the solutions to problems
- Do not shy away from recursion
- Always remember that mutation is like a hammer that clobbers everything it touches Sometimes problem solving requires more finesse and a screwdriver, not a hammer, ought to be used
- Sometimes problem solving requires more finesse and a screwdriver, not a hammer, ought to be used

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Epilogue

For the more immediate future here are some suggestions for you:

- Pursue excellence. Strive to become the best problem solver you can.

Epilogue

For the more immediate future here are some suggestions for you:

- Pursue excellence. Strive to become the best problem solver you can.
- Learn or teach yourself a new programming language every semester and summer. Learn in this context refers to mastering the abstractions (not the syntax) offered. Some suggestions:
 - Learn about lazy evaluation by writing programs using Haskell or Clean.
 - Learn about object-oriented programming by writing programs using Java or Scala.
 - Learn about macros by writing programs using Racket.
 - Learn about logic programming by writing programs using Prolog.

Epilogue

For the more immediate future here are some suggestions for you:

- Pursue excellence. Strive to become the best problem solver you can.
- Learn or teach yourself a new programming language every semester and summer. Learn in this context refers to mastering the abstractions (not the syntax) offered. Some suggestions:
 - Learn about lazy evaluation by writing programs using Haskell or Clean.
 - Learn about object-oriented programming by writing programs using Java or Scala.
 - Learn about macros by writing programs using Racket.
 - Learn about logic programming by writing programs using Prolog.
- Learn about the implementation of programming languages!

Epilogue

For the more immediate future here are some suggestions for you:

- Pursue excellence. Strive to become the best problem solver you can.
- Learn or teach yourself a new programming language every semester and summer. Learn in this context refers to mastering the abstractions (not the syntax) offered. Some suggestions:
 - Learn about lazy evaluation by writing programs using Haskell or Clean.
 - Learn about object-oriented programming by writing programs using Java or Scala.
 - Learn about macros by writing programs using Racket.
 - Learn about logic programming by writing programs using Prolog.
- Learn about the implementation of programming languages!
- Do not shy away from Math!

Epilogue

For the more immediate future here are some suggestions for you:

- Pursue excellence. Strive to become the best problem solver you can.
- Learn or teach yourself a new programming language every semester and summer. Learn in this context refers to mastering the abstractions (not the syntax) offered. Some suggestions:
 - Learn about lazy evaluation by writing programs using Haskell or Clean.
 - Learn about object-oriented programming by writing programs using Java or Scala.
 - Learn about macros by writing programs using Racket.
 - Learn about logic programming by writing programs using Prolog.
- Learn about the implementation of programming languages!
- Do not shy away from Math!
- Attend talks by invited speakers in your department

Sharing Values

Mutation
Sequencing

Vectors

In-Place
Operations

The Chicken
and the Egg
Paradox

Epilogue

Epilogue

It has been a pleasure!



Live long and prosper!