Part III: Accumulative Recursion

Marco T. Morazán

Seton Hall University

Outline

Accumulators

- How can functions remember?

Accumulators

- How can functions remember?
- One must also know *what* a function needs to remember

Accumulators

- How can functions remember?
- One must also know *what* a function needs to remember
- Remembering is quite natural in every day activities
- You either memorize or write down the data to remember
- This means you are storing the data to remember in brain cells or on paper.

Accumulators

- How can functions remember?
- One must also know *what* a function needs to remember
- Remembering is quite natural in every day activities
- You either memorize or write down the data to remember
- This means you are storing the data to remember in brain cells or on paper.
- Where can programs store data that needs to be remembered?

Accumulators

- How can functions remember?
- One must also know *what* a function needs to remember
- Remembering is quite natural in every day activities
- You either memorize or write down the data to remember
- This means you are storing the data to remember in brain cells or on paper.
- Where can programs store data that needs to be remembered?
- Programs can remember data by using variables

Accumulators

- How can functions remember?
- One must also know *what* a function needs to remember
- Remembering is quite natural in every day activities
- You either memorize or write down the data to remember
- This means you are storing the data to remember in brain cells or on paper.
- Where can programs store data that needs to be remembered?
- Programs can remember data by using variables
- Variables that help a program or a function remember data are called *accumulators*
- Clearly identify what value the accumulator stores and how the accumulator is exploited

Accumulators

- How can functions remember?
- One must also know *what* a function needs to remember
- Remembering is quite natural in every day activities
- You either memorize or write down the data to remember
- This means you are storing the data to remember in brain cells or on paper.
- Where can programs store data that needs to be remembered?
- Programs can remember data by using variables
- Variables that help a program or a function remember data are called *accumulators*
- Clearly identify what value the accumulator stores and how the accumulator is exploited
- Clearly identifying what an accumulator stores is done by developing an *accumulator invariant*
- An accumulator invariant is an assertion that must always be true about the variable when a function is called
- Identifying how the accumulator is exploited is done by explaining how the accumulator is used during the computation

Accumulators

- Accumulators may be added as parameters to functions that use structural or generative recursion

Accumulators

- Accumulators may be added as parameters to functions that use structural or generative recursion
- When may a function benefit from an accumulator?
  - structural recursion: the result of a recursive call is input to another function
  - generative recursion: the function may fail to produce a result
  - There are other more uncommon situations

Accumulators

- Accumulators may be added as parameters to functions that use structural or generative recursion
- When may a function benefit from an accumulator?
  - structural recursion: the result of a recursive call is input to another function
  - generative recursion: the function may fail to produce a result
  - There are other more uncommon situations
- Recursion that uses accumulators is called *accumulative recursion*.

Running Totals

- Consider the problem of computing the running totals of a list of numbers
- The running total for the $\texttt{i}^{\texttt{th}}$ element of the list is the sum of all the elements up to and including the $\texttt{i}^{\texttt{th}}$ element

Running Totals

Problem Analysis

- The first running total is the sum that includes the first number
- The second running total is the sum that includes the first two numbers
- The third running total is the sum that includes the first three numbers

Running Totals

Problem Analysis

- The first running total is the sum that includes the first number
- The second running total is the sum that includes the first two numbers
- The third running total is the sum that includes the first three numbers
- In general, the $\texttt{k}^{\texttt{th}}$ running total is the sum that includes the first `k` numbers

Running Totals

Problem Analysis

- The first running total is the sum that includes the first number
- The second running total is the sum that includes the first two numbers
- The third running total is the sum that includes the first three numbers
- In general, the `k`$^{\texttt{th}}$ running total is the sum that includes the first `k` numbers
- The number of elements in the sum starts at 1 and increases by 1 for the next running sum
- Suggests traversing the interval `[0..(sub1 (lenght a-lon))]` using an auxiliary function

Running Totals

Problem Analysis

- The first running total is the sum that includes the first number
- The second running total is the sum that includes the first two numbers
- The third running total is the sum that includes the first three numbers
- In general, the `k`$^{\texttt{th}}$ running total is the sum that includes the first `k` numbers
- The number of elements in the sum starts at 1 and increases by 1 for the next running sum
- Suggests traversing the interval `[0..(sub1 (lenght a-lon))]` using an auxiliary function
- Testing `lons`:

```
(define L0 '())
(define L1 (list 1 2 3 4 5 6))
(define L2 (build-list 2500 (λ (n) (random 100000))))
```

Running Totals

Sample Expressions

```
;; Sample expressions for lon-running-totals
(define LRS-L0 (lon-running-totals-helper
                L0
                0
                (sub1 (length L0))))

(define LRS-L1 (lon-running-totals-helper
                L1
                0
                (sub1 (length L1))))

(define LRS-L2 (lon-running-totals-helper
                L2
                0
                (sub1 (length L2))))
```

Running Totals

Sample Expressions

- ```
  ;; Sample expressions for lon-running-totals
  (define LRS-L0 (lon-running-totals-helper
                  L0
                  0
                  (sub1 (length L0))))

  (define LRS-L1 (lon-running-totals-helper
                  L1
                  0
                  (sub1 (length L1))))

  (define LRS-L2 (lon-running-totals-helper
                  L2
                  0
                  (sub1 (length L2))))
  ```
- The only difference among the sample expressions is the `lon` processed
  ```

Running Totals

Signature, Statments, Function Header, and Tests

- ```
  ;; lon → lon
  ;; Purpose: Return list of running totals for the
  ;;          given lon
  (define (lon-running-totals a-lon)
    (lon-running-totals-helper a-lon
                               0
                               (sub1 (length a-lon))))
  ```
  ```

Running Totals

Signature, Statments, Function Header, and Tests

- ```
  ;; lon → lon
  ;; Purpose: Return list of running totals for the
  ;;          given lon
  (define (lon-running-totals a-lon)
    (lon-running-totals-helper a-lon
                               0
                               (sub1 (length a-lon))))
  ```
- ```
  ;; Tests using sample computations for lon-running-totals
  (check-expect (lon-running-totals L0) LRS-L0)
  (check-expect (lon-running-totals L1) LRS-L1)
  (check-expect (lon-running-totals L2) LRS-L2)

  ;; Tests using sample values for lon-running-totals
  (check-expect (lon-running-totals '(-1 0 1)) '(-1 -1 0))
  (check-expect (lon-running-totals '(-5 2 4 0))
                                    '(-5 -3 1 1))
  ```

lon-running-totals-helper

Problem Analysis

- How should the interval be processed?

lon-running-totals-helper

Problem Analysis

- How should the interval be processed?
- Process the interval from `low` to `high`
- If the given interval is empty the answer is the empty list
- If the given interval is not empty the sum of the numbers indexed by `[0..low]` is added to the front of the list obtained by processing the rest of the interval.

lon-running-totals-helper

Problem Analysis

- How should the interval be processed?
- Process the interval from `low` to `high`
- If the given interval is empty the answer is the empty list
- If the given interval is not empty the sum of the numbers indexed by `[0..low]` is added to the front of the list obtained by processing the rest of the interval.
- Computing the sum of the numbers indexed by `[0..low]` is a different problem and an auxiliary function is needed

lon-running-totals-helper

Sample Expressions

- The answer is empty list when the given interval is empty:

```
;; Sample expressions for lon-running-totals-helper
(define LORSH-L0-5  '())
```

lon-running-totals-helper

Sample Expressions

- The answer is empty list when the given interval is empty:

```
;; Sample expressions for lon-running-totals-helper
(define LORSH-L0-5  '())
```

- When the given interval is not empty an auxiliary function to compute the next running total is called to process `[0..low]`
- This sum is added to the front of the list obtained from processing the rest of the interval:

```
(define LORSH-L1-0  (cons (lon-sum L1 0 0)
                          (lon-running-totals-helper
                            L1
                            (add1 0)
                            5)))

(define LORSH-L2-75 (cons (lon-sum L2 0 75)
                          (lon-running-totals-helper
                            L2
                            (add1 75)
                            2499)))
```

lon-running-totals-helper

Sample Expressions

- The answer is empty list when the given interval is empty:

```
;; Sample expressions for lon-running-totals-helper
(define LORSH-L0-5  '())
```

- When the given interval is not empty an auxiliary function to compute the next running total is called to process `[0..low]`
- This sum is added to the front of the list obtained from processing the rest of the interval:

```
(define LORSH-L1-0  (cons (lon-sum L1 0 0)
                          (lon-running-totals-helper
                            L1
                            (add1 0)
                            5)))

(define LORSH-L2-75 (cons (lon-sum L2 0 75)
                          (lon-running-totals-helper
                            L2
                            (add1 75)
                            2499)))
```

- The differences among the sample expressions are the list and the interval processed

lon-running-totals-helper

Signature, Function Definition, and Tests

- ```
  ;; lon [int..int] → lon
  ;; Purpose: Return list of running totals for list and interval.
  ;; Assumption: Given interval contains valid indices into given lis
  (define (lon-running-totals-helper a-lon low high)
    (if (> low high)
        '()
        (cons (lon-sum a-lon 0 low)
              (lon-running-totals-helper a-lon (add1 low) high))))
  ```
  ```

lon-running-totals-helper

Signature, Function Definition, and Tests

- ```
  ;; lon [int..int] → lon
  ;; Purpose: Return list of running totals for list and interval.
  ;; Assumption: Given interval contains valid indices into given lis
  (define (lon-running-totals-helper a-lon low high)
    (if (> low high)
        '()
        (cons (lon-sum a-lon 0 low)
              (lon-running-totals-helper a-lon (add1 low) high))))
  ```
- ```
  ;; Tests using sample computations for
  ;;   lon-running-totals-helper
  (check-expect (lon-running-totals-helper L0 0 -1)
                LORSH-L0-5)
  (check-expect (lon-running-totals-helper L1 0 5)
                LORSH-L1-0)
  (check-expect (lon-running-totals-helper L2 75 2499)
                LORSH-L2-75)
  ;; Tests using sample values for lon-running-totals-helper
  (check-expect
    (lon-running-totals-helper '(-2 -1 0 1 2) 0 4)
    '(-2 -3 -3 -2 0))
  (check-expect
    (lon-running-totals-helper '(50 25 -40) 1 2)
    '(75 35))
  ```

The lon-sum Function

- Structural recursion:

```
;; lon [int..int] → lon
;; Purpose: Return the list element sum for the
;;          given interval
;; Assumption: The given interval only contains valid
;;             indices into the given lon
(define (lon-sum a-lon low high)
  (if (> low high)
      0
      (+ (list-ref a-lon high)(lon-sum a-lon low (sub1 high)))))
;; Sample expressions for lon-sum
(define SFN-L0-VAL 0)
(define SFN-L1-VAL (+ (list-ref L1 3) (lon-sum L1 1 (sub1 3))))
(define SFN-L2-VAL (+ (list-ref L2 99) (lon-sum L2 0 (sub1 99))))

;; Tests using sample computations for lon-sum
(check-expect (lon-sum L0 0 -1) SFN-L0-VAL)
(check-expect (lon-sum L1 1  3) SFN-L1-VAL)
(check-expect (lon-sum L2 0 99) SFN-L2-VAL)

;; Tests using sample values for lon-sum
(check-expect (lon-sum '(10 20 30 40) 0 1) 30)
(check-expect (lon-sum '(0 1 2 3 4 5) 0 5) 15)
```

Running Totals Using an Accumulator

Problem Analysis

- A lot of repetitive work
- `L = '(1 2 3 4 5)`:

low	New Running Total Computed
0	(+ 1 0)
1	(+ 1 2 0)
2	(+ 1 2 3 0)
3	(+ 1 2 3 4 0)
4	(+ 1 2 3 4 5 0)

- At each step the computation of the previous running total is repeated

Running Totals Using an Accumulator

Problem Analysis

- A lot of repetitive work
- `L = '(1 2 3 4 5)`:

low	New Running Total Computed
0	(+ 1 0)
1	(+ 1 2 0)
2	(+ 1 2 3 0)
3	(+ 1 2 3 4 0)
4	(+ 1 2 3 4 5 0)

- At each step the computation of the previous running total is repeated
- Suggests that computing running totals may benefit from using an accumulator to store the previous running total

Running Totals Using an Accumulator

Problem Analysis

- A lot of repetitive work
- `L = '(1 2 3 4 5)`:

low	New Running Total Computed
0	(+ 1 0)
1	(+ 1 2 0)
2	(+ 1 2 3 0)
3	(+ 1 2 3 4 0)
4	(+ 1 2 3 4 5 0)

- At each step the computation of the previous running total is repeated
- Suggests that computing running totals may benefit from using an accumulator to store the previous running total
- At the beginning the running total starts at 0

Running Totals Using an Accumulator

Problem Analysis

- A lot of repetitive work
- `L = '(1 2 3 4 5)`:

low	New Running Total Computed
0	(+ 1 0)
1	(+ 1 2 0)
2	(+ 1 2 3 0)
3	(+ 1 2 3 4 0)
4	(+ 1 2 3 4 5 0)

- At each step the computation of the previous running total is repeated
- Suggests that computing running totals may benefit from using an accumulator to store the previous running total
- At the beginning the running total starts at 0
- We may refactor `lon-running-totals` to call an auxiliary function that processes the given list that using an accumulator with an initial value of 0

Running Totals Using an Accumulator

Sample Expressions and Differences

- ```
  ;; Sample expressions for lon-running-totals2
  (define LRS2-L0 (lon-running-totals-helper-v2 L0 0))
  (define LRS2-L1 (lon-running-totals-helper-v2 L1 0))
  (define LRS2-L2 (lon-running-totals-helper-v2 L2 0))
  ```
- The only difference among the sample expressions is the list processed
  ```

Running Totals Using an Accumulator

Tests and Function Definition

- ```
  ;; lon → lon
  ;; Purpose: Return the list of running totals for the
  ;;          given lon
  (define (lon-running-totals-v2 a-lon)
  ```
  ```

Running Totals Using an Accumulator

Tests and Function Definition

- ```
  ;; lon → lon
  ;; Purpose: Return the list of running totals for the
  ;;          given lon
  (define (lon-running-totals-v2 a-lon)
  ```

- ```
  ;; Tests using sample computations for lon-running-totals2
  (check-expect (lon-running-totals-v2 L0) LRS2-L0)
  (check-expect (lon-running-totals-v2 L1) LRS2-L1)
  (check-expect (lon-running-totals-v2 L2) LRS2-L2)

  ;; Tests using sample values for lon-running-totals2
  (check-expect (lon-running-totals-v2 '(-1 0 1))
                '(-1 -1 0))
  (check-expect (lon-running-totals-v2 '(-5 2 4 0))
                '(-5 -3 1 1))
  ```

Running Totals Using an Accumulator

Tests and Function Definition

```
;; lon → lon
;; Purpose: Return the list of running totals for the
;;          given lon
(define (lon-running-totals-v2 a-lon)
  (lon-running-totals-helper-v2 a-lon 0))
```

```
;; Tests using sample computations for lon-running-totals2
(check-expect (lon-running-totals-v2 L0) LRS2-L0)
(check-expect (lon-running-totals-v2 L1) LRS2-L1)
(check-expect (lon-running-totals-v2 L2) LRS2-L2)

;; Tests using sample values for lon-running-totals2
(check-expect (lon-running-totals-v2 '(-1 0 1))
              '(-1 -1 0))
(check-expect (lon-running-totals-v2 '(-5 2 4 0))
              '(-5 -3 1 1))
```

lon-running-totals-helper-v2

Problem Analysis

- If the given list is empty the empty list is the answer

lon-running-totals-helper-v2

Problem Analysis

- If the given list is empty the empty list is the answer
- If the given list is not empty then the next running total is obtained by adding the first list element to the accumulator
- This value is both the next value to add to the result and the new value of the accumulator

lon-running-totals-helper-v2

Problem Analysis

- If the given list is empty the empty list is the answer
- If the given list is not empty then the next running total is obtained by adding the first list element to the accumulator
- This value is both the next value to add to the result and the new value of the accumulator
- We must also clearly identify the value that is stored in the accumulator by developing an accumulator invariant.

lon-running-totals-helper-v2

Problem Analysis

- If the given list is empty the empty list is the answer
- If the given list is not empty then the next running total is obtained by adding the first list element to the accumulator
- This value is both the next value to add to the result and the new value of the accumulator
- We must also clearly identify the value that is stored in the accumulator by developing an accumulator invariant.
- Think of the list given by `lon-running-totals-v2` as divided in two parts: the processed part and the unprocessed part
- `L = '(1 2 3 4 5)`:

Processed	Unprocessed	Accumulator
'()	(1 2 3 4 5)	0
(1)	(2 3 4 5)	1
(1 2)	(3 4 5)	3
(1 2 3)	(4 5)	6
(1 2 3 4)	(5)	10
(1 2 3 4 5)	'()	15

- Ask yourself what is true about the accumulator at each step captured by the table

lon-running-totals-helper-v2

Problem Analysis

- If the given list is empty the empty list is the answer
- If the given list is not empty then the next running total is obtained by adding the first list element to the accumulator
- This value is both the next value to add to the result and the new value of the accumulator
- We must also clearly identify the value that is stored in the accumulator by developing an accumulator invariant.
- Think of the list given by `lon-running-totals-v2` as divided in two parts: the processed part and the unprocessed part
- `L = '(1 2 3 4 5)`:

Processed	Unprocessed	Accumulator
'()	(1 2 3 4 5)	0
(1)	(2 3 4 5)	1
(1 2)	(3 4 5)	3
(1 2 3)	(4 5)	6
(1 2 3 4)	(5)	10
(1 2 3 4 5)	'()	15

- Ask yourself what is true about the accumulator at each step captured by the table
- ```
  Sum of L's elements =
    Accumulator + Sum of L's unprocessed elements
  ```
  ```

lon-running-totals-helper-v2

Problem Analysis

- We can more concisely state the accumulator invariant as:

```
Accumulator = Sum of L's processed elements
            = the previous running total
```

lon-running-totals-helper-v2

Problem Analysis

- We can more concisely state the accumulator invariant as:

```
Accumulator = Sum of L's processed elements
            = the previous running total
```

- The accumulator invariant is true when `lon-running-totals-helper-v2` is called by `lon-running-totals-v2`:

```
Accumulator = the previous running total
          0 = the previous running total
          0 = 0
```

lon-running-totals-helper-v2

Sample Expressions and Differences

- ```
  ;; Sample expressions for lon-running-totals-helper-v2
  (define LRTHV2-L0-0 '())
  ```
  ```

lon-running-totals-helper-v2

Sample Expressions and Differences

```
;; Sample expressions for lon-running-totals-helper-v2
(define LRTHV2-L0-0 '())
```

```
(define LRTHV2-L1-0
        (local [(define new-accum (+ (first L1) 0))]
          (cons new-accum
                (lon-running-totals-helper-v2
                  (rest L1) new-accum))))

(define LRTHV2-L2-5  first of L2 is already processed
        (local [(define new-accum (+ (first (rest L2))
                                     (first L2)))]
          (cons new-accum
                (lon-running-totals-helper-v2
                  (rest (rest L2)) new-accum))))
```

lon-running-totals-helper-v2

Sample Expressions and Differences

- ```
  ;; Sample expressions for lon-running-totals-helper-v2
  (define LRTHV2-L0-0 '())
  ```
- ```
  (define LRTHV2-L1-0
          (local [(define new-accum (+ (first L1) 0))]
            (cons new-accum
                  (lon-running-totals-helper-v2
                    (rest L1) new-accum))))

  (define LRTHV2-L2-5  first of L2 is already processed
          (local [(define new-accum (+ (first (rest L2))
                                       (first L2)))]
            (cons new-accum
                  (lon-running-totals-helper-v2
                    (rest (rest L2)) new-accum))))
  ```
- The values provided as arguments to the call to `lon-running-totals-helper-v2` make the accumulator invariant true

lon-running-totals-helper-v2

Sample Expressions and Differences

- ```
  ;; Sample expressions for lon-running-totals-helper-v2
  (define LRTHV2-L0-0 '())
  ```
- ```
  (define LRTHV2-L1-0
          (local [(define new-accum (+ (first L1) 0))]
            (cons new-accum
                  (lon-running-totals-helper-v2
                    (rest L1) new-accum))))

  (define LRTHV2-L2-5  first of L2 is already processed
          (local [(define new-accum (+ (first (rest L2))
                                       (first L2)))]
            (cons new-accum
                  (lon-running-totals-helper-v2
                    (rest (rest L2)) new-accum))))
  ```
- The values provided as arguments to the call to `lon-running-totals-helper-v2` make the accumulator invariant true
- Two differences: the `lon` processed and the value of the accumulator

lon-running-totals-helper-v2

Tests and Function Definition

- ```
  ;; lon number → lon
  ;; Purpose: Return list of running totals
  ;; Accumulator Invaraint: acc = previous running total
  (define (lon-running-totals-helper-v2 a-lon acc)
  ```
  ```

lon-running-totals-helper-v2

Tests and Function Definition

- ```
  ;; lon number → lon
  ;; Purpose: Return list of running totals
  ;; Accumulator Invaraint: acc = previous running total
  (define (lon-running-totals-helper-v2 a-lon acc)
  ```

- ```
  ;; Tests using sample computations for lon-running-totals-helper-v2
  (check-expect (lon-running-totals-helper-v2 L0 0) LRTHV2-L0-0)
  (check-expect (lon-running-totals-helper-v2 L1 0) LRTHV2-L1-0)
  (check-expect (lon-running-totals-helper-v2 (rest L2) (first L2))
                LRTHV2-L2-5)
  ```

lon-running-totals-helper-v2

Tests and Function Definition

- ```
  ;; lon number → lon
  ;; Purpose: Return list of running totals
  ;; Accumulator Invaraint: acc = previous running total
  (define (lon-running-totals-helper-v2 a-lon acc)
  ```

- ```
  ;; Tests using sample computations for lon-running-totals-helper-v2
  (check-expect (lon-running-totals-helper-v2 L0 0) LRTHV2-L0-0)
  (check-expect (lon-running-totals-helper-v2 L1 0) LRTHV2-L1-0)
  (check-expect (lon-running-totals-helper-v2 (rest L2) (first L2))
                LRTHV2-L2-5)
  ```

- ```
  ;; Tests using sample values for
  ;;   lon-running-totals-helper-v2
  (check-expect (lon-running-totals-helper-v2 '(1 2 3)  0)
                '(1 3 6))
  (check-expect (lon-running-totals-helper-v2
                  (rest (rest '(10 4 5 6))) 14)
                '(19 25))
  ```
  ```

lon-running-totals-helper-v2

Tests and Function Definition

```
;; lon number → lon
;; Purpose: Return list of running totals
;; Accumulator Invaraint: acc = previous running total
(define (lon-running-totals-helper-v2 a-lon acc)
  (if (empty? a-lon)
      '()
      (local [(define new-accum (+ (first a-lon) acc))]
        (cons new-accum
              (lon-running-totals-helper-v2 (rest a-lon) new-accum)
```

```
;; Tests using sample computations for lon-running-totals-helper-v2
(check-expect (lon-running-totals-helper-v2 L0 0) LRTHV2-L0-0)
(check-expect (lon-running-totals-helper-v2 L1 0) LRTHV2-L1-0)
(check-expect (lon-running-totals-helper-v2 (rest L2) (first L2))
              LRTHV2-L2-5)
```

```
;; Tests using sample values for
;;   lon-running-totals-helper-v2
(check-expect (lon-running-totals-helper-v2 '(1 2 3)  0)
              '(1 3 6))
(check-expect (lon-running-totals-helper-v2
                (rest (rest '(10 4 5 6))) 14)
              '(19 25))
```

lon-running-totals-helper-v2

Empirical Analysis

- Two solutions: Is there any reason to prefer one over another?

lon-running-totals-helper-v2

Empirical Analysis

- Two solutions: Is there any reason to prefer one over another?
- To avoid bias use randomly generated `L2`

lon-running-totals-helper-v2

Empirical Analysis

- Two solutions: Is there any reason to prefer one over another?
- To avoid bias use randomly generated `L2`
- Run experiment 5 times

```
(time (lon-running-totals    L2))
(time (lon-running-totals-v2 L2))
```

lon-running-totals-helper-v2

Empirical Analysis

- Two solutions: Is there any reason to prefer one over another?
- To avoid bias use randomly generated `L2`
- Run experiment 5 times

```
(time (lon-running-totals    L2))
(time (lon-running-totals-v2 L2))
```

- CPU times in milliseconds (after subtracting garbage collection time):

	`lon-running-totals`	`lon-running-totals-v2`
`Run 1`	21406	0
`Run 2`	21563	0
`Run 3`	21860	0
`Run 4`	13562	15
`Run 5`	14031	0

- The average CPU time decreased from about 18.5 seconds without using an accumulator to 3 milliseconds when using an accumulator

lon-running-totals-helper-v2

Complexity

- `lon-running-totals-helper` is called `n + 1` times, where `n` is the size of the list given to `lon-running-totals`.
- Number of times it is called is $O(n)$

`lon-running-totals-helper-v2`

Complexity

- `lon-running-totals-helper` is called `n + 1` times, where `n` is the size of the list given to `lon-running-totals`.
- Number of times it is called is $O(n)$
- For `n` of those calls `lon-sum` is called
- In the worst case, the function `lon-sum` is called `n + 1` times
- This makes the calls to `lon-sum` $O(n)$.

`lon-running-totals-helper-v2`

Complexity

- `lon-running-totals-helper` is called `n + 1` times, where `n` is the size of the list given to `lon-running-totals`.
- Number of times it is called is $O(n)$
- For `n` of those calls `lon-sum` is called
- In the worst case, the function `lon-sum` is called `n + 1` times
- This makes the calls to `lon-sum` $O(n)$.
- The complexity of `lon-running-totals` is $O(n) * O(n) = O(n^2)$.

lon-running-totals-helper-v2

Complexity

- `lon-running-totals-helper` is called `n + 1` times, where `n` is the size of the list given to `lon-running-totals`.
- Number of times it is called is $O(n)$
- For `n` of those calls `lon-sum` is called
- In the worst case, the function `lon-sum` is called `n + 1` times
- This makes the calls to `lon-sum` $O(n)$.
- The complexity of `lon-running-totals` is $O(n) * O(n) = O(n^2)$.
- The function `lon-running-totals-helper-v2` is called `n + 1` times in the worst case, where `n` is the size of the list given to `lon-running-totals-v2`
- For each of those calls a constant number of operations are performed (i.e., `empty?`, `+`, `cons`, and `rest`)
- This makes the number of operations performed by `lon-running-totals-v2` $O(k * n) = O(n)$

lon-running-totals-helper-v2

Complexity

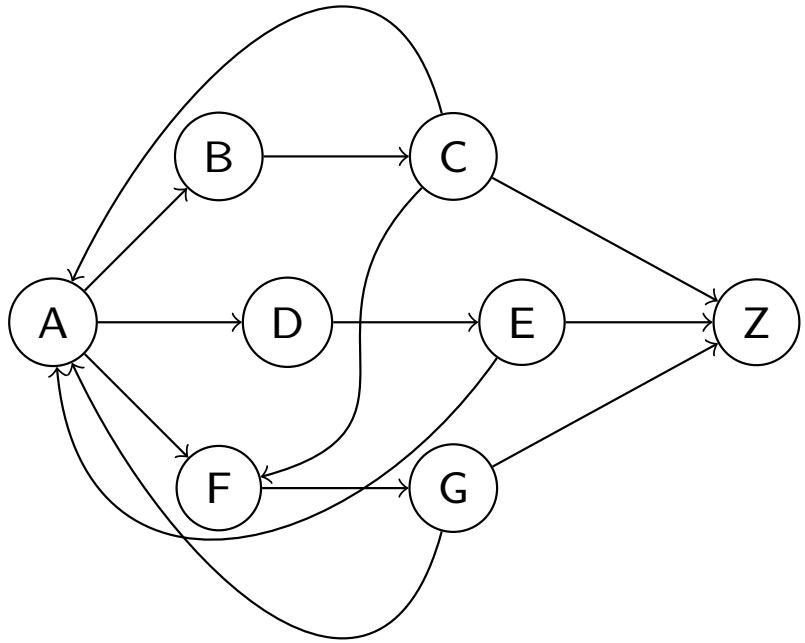
- `lon-running-totals-helper` is called `n + 1` times, where `n` is the size of the list given to `lon-running-totals`.
- Number of times it is called is $O(n)$
- For `n` of those calls `lon-sum` is called
- In the worst case, the function `lon-sum` is called `n + 1` times
- This makes the calls to `lon-sum` $O(n)$.
- The complexity of `lon-running-totals` is $O(n) * O(n) = O(n^2)$.
- The function `lon-running-totals-helper-v2` is called `n + 1` times in the worst case, where `n` is the size of the list given to `lon-running-totals-v2`
- For each of those calls a constant number of operations are performed (i.e., `empty?`, `+`, `cons`, and `rest`)
- This makes the number of operations performed by `lon-running-totals-v2` $O(k * n) = O(n)$
- The complexity of the solution has been reduced from quadratic to linear.
- This explains the impressive performance improvement.

HOMEWORK

- HOMEWORK: 4-7
- QUIZ: 8 (due in one week)

Finding a Path in a Directed Graph

- A directed graph is a representation of a relation or of a map
- Every graph has a finite set of nodes and arrows
- 8 nodes and 13 edges:



- A common problem involving a graph is finding a path between two nodes

Finding a Path in a Directed Graph

Data Analysis

- We need a representation for a graph

Finding a Path in a Directed Graph

Data Analysis

- We need a representation for a graph
- *adjaceny list* representation: a graph is a list of nodes
- Every node has a name and a list of the names of its neighbors

Finding a Path in a Directed Graph

Data Analysis

- We need a representation for a graph
- *adjaceny list* representation: a graph is a list of nodes
- Every node has a name and a list of the names of its neighbors
- For our purposes, the data definition of a node is:

```
#|  A node is a structure: (make-node symbol (listof symbol)) |#
```

Finding a Path in a Directed Graph

Data Analysis

- We need a representation for a graph
- *adjaceny list* representation: a graph is a list of nodes
- Every node has a name and a list of the names of its neighbors
- For our purposes, the data definition of a node is:

```
#|  A node is a structure: (make-node symbol (listof symbol)) |#
```

- Sample nodes:

```
;; Nodes for G1
(define NODEA (make-node 'A '(B D F)))
(define NODEB (make-node 'B '(C)))
(define NODEC (make-node 'C '(Z)))
(define NODED (make-node 'D '(E)))
(define NODEE (make-node 'E '(Z)))
(define NODEF (make-node 'F '(G)))
(define NODEG (make-node 'G '(Z)))
(define NODEZ (make-node 'Z '()))
;; Nodes for G2
(define NODEC2 (make-node 'C '()))
;; Nodes for G3
(define NODEC3 (make-node 'C '(A F Z)))
(define NODEE3 (make-node 'E '(A Z)))
(define NODEG3 (make-node 'G '(A Z)))
```

Finding a Path in a Directed Graph

Data Analysis

```
#| Template for functions on a node

node ... → ...
Purpose:
(define (f-on-node a-node ...)
  ...(node-name a-node)...(node-neighs a-node)...)

;; Sample expressions for f-on-node
(define NODEA-VAL ...)
        :

;; Tests using sample computations for f-on-node
(check-expect (f-on-node NODEA ...) NODEA-VAL)
        :

;; Tests using sample values for f-on-node
(check-expect (f-on-node ... ...) ...)
        :
|#
```

Finding a Path in a Directed Graph

Data Analysis

- `#| A graph is a nonempty (listof node). |#`

Finding a Path in a Directed Graph

Data Analysis

- `#| A graph is a nonempty (listof node). |#`
- Sample graphs:

```
(define G1 (list NODEA NODEB NODEC NODED NODEE NODEF NODEG NODEZ))
(define G2 (list NODEA NODEB NODEC2 NODED NODEE NODEF NODEG  NODEZ)
(define G3 (list NODEA  NODEB NODEC3 NODED NODEE3 NODEF NODEG3 NODE
```

Finding a Path in a Directed Graph

Data Analysis

- `#| A graph is a nonempty (listof node). |#`
- Sample graphs:

```
(define G1 (list NODEA NODEB NODEC NODED NODEE NODEF NODEG NODEZ))
(define G2 (list NODEA NODEB NODEC2 NODED NODEE NODEF NODEG  NODEZ)
(define G3 (list NODEA  NODEB NODEC3 NODED NODEE3 NODEF NODEG3 NODE
```

- Template for functions on a graph is:

```
#| graph ... → ...    Purpose:
   (define (f-on-graph a-graph ...)
       (if (empty? (rest a-graph))
           (f-on-node (first a-graph) ...)
           (...(f-on-node (first a-graph)...)...
            ...(f-on-graph (rest a-graph)...)...)))
   ;; Sample expressions for f-on-graph
   (define FONG-G1-VAL ...) (define FONG-G2-VAL ...)
   (define FONG-G13-VAL ...) ...
   ;; Tests using sample computations for f-on-graph
   (check-expect (f-on-graph G1 ...) FONG-G1-VAL)
   (check-expect (f-on-graph G2 ...) FONG-G2-VAL)
   (check-expect (f-on-graph G3 ...) FONG-G3-VAL) ...
   ;; Tests using sample values for f-on-graph
   (check-expect (f-on-graph ... ...) ...) ...  |#
```

Finding a Path in a Directed Graph

Data Analysis

- `#| A graph is a nonempty (listof node). |#`
- Sample graphs:

```
(define G1 (list NODEA NODEB NODEC NODED NODEE NODEF NODEG NODEZ))
(define G2 (list NODEA NODEB NODEC2 NODED NODEE NODEF NODEG  NODEZ)
(define G3 (list NODEA  NODEB NODEC3 NODED NODEE3 NODEF NODEG3 NODE
```

- Template for functions on a graph is:

```
#| graph ... → ...    Purpose:
   (define (f-on-graph a-graph ...)
       (if (empty? (rest a-graph))
           (f-on-node (first a-graph) ...)
           (...(f-on-node (first a-graph)...)...
            ...(f-on-graph (rest a-graph)...)...)))
   ;; Sample expressions for f-on-graph
   (define FONG-G1-VAL ...) (define FONG-G2-VAL ...)
   (define FONG-G13-VAL ...) ...
   ;; Tests using sample computations for f-on-graph
   (check-expect (f-on-graph G1 ...) FONG-G1-VAL)
   (check-expect (f-on-graph G2 ...) FONG-G2-VAL)
   (check-expect (f-on-graph G3 ...) FONG-G3-VAL) ...
   ;; Tests using sample values for f-on-graph
   (check-expect (f-on-graph ... ...) ...) ...  |#
```

- `#| A path is either: 1. (listof symbol) 2. 'no-path  |#`

find-path

Problem Analysis

- To find a path in a given graph the starting and ending nodes' are needed
- How can a graph be traversed?

find-path

Problem Analysis

- To find a path in a given graph the starting and ending nodes' are needed
- How can a graph be traversed?
- A nonlinear data structure may be traversed using depth-first search or breadth-first search
- Without loss of generality we explore a design based on depth-first search

find-path

Problem Analysis

- To find a path in a given graph the starting and ending nodes' are needed
- How can a graph be traversed?
- A nonlinear data structure may be traversed using depth-first search or breadth-first search
- Without loss of generality we explore a design based on depth-first search
- We must careful about repetitions in the search space to avoid an infinite recursion
- Using first neighbor in our sample graph:

```
path from A to Z = A → path from one in (B D F) to Z
                 = A → path from B to Z
                 = A → B → from one in (C) to Z
                 = A → B → path from C to Z
                 = A → B → C → from one in (A F Z)
                 = A → B → C → path from A to Z
                 ⋮
```

- How can we prevent an infinite recursion?

find-path

Problem Analysis

- To find a path in a given graph the starting and ending nodes' are needed
- How can a graph be traversed?
- A nonlinear data structure may be traversed using depth-first search or breadth-first search
- Without loss of generality we explore a design based on depth-first search
- We must careful about repetitions in the search space to avoid an infinite recursion
- Using first neighbor in our sample graph:

```
path from A to Z = A → path from one in (B D F) to Z
                 = A → path from B to Z
                 = A → B → from one in (C) to Z
                 = A → B → path from C to Z
                 = A → B → C → from one in (A F Z)
                 = A → B → C → path from A to Z
                 ⋮
```

- How can we prevent an infinite recursion?
- Introduce an accumulator to remember the nodes that have been visited
- If a node has been visited during a search it is not used to find a path again

find-path

Sample Expressions and Differences

- ```
  ;; Sample expressions for find-path
  (define FP-AZ-G1 (find-path-acc G1 'A 'Z '()))
  (define FP-AZ-G2 (find-path-acc G2 'C 'Z '()))
  (define FP-AZ-G3 (find-path-acc G3 'A 'Z '()))
  ```
- Three differences: the graph to search, symbol for the starting node, and a symbol for the destination node
  ```

find-path

Tests and Function Definition

- ```
  ;; graph symbol symbol → path
  ;; Purpose: Find a path between nodes with the given names
  ;;          in the given graph
  ;; Assumption: Given node names are in the given graph
  (define (find-path a-graph start end)
  ```
  ```

find-path

Tests and Function Definition

- ```
  ;; graph symbol symbol → path
  ;; Purpose: Find a path between nodes with the given names
  ;;          in the given graph
  ;; Assumption: Given node names are in the given graph
  (define (find-path a-graph start end)
  ```

- ```
  ;; Test using sample computations for find-path
  (check-expect (find-path G1 'A 'Z) FP-AZ-G1)
  (check-expect (find-path G2 'C 'Z) FP-AZ-G2)
  (check-expect (find-path G3 'A 'Z) FP-AZ-G3)
  ```

find-path

Tests and Function Definition

- ```
  ;; graph symbol symbol → path
  ;; Purpose: Find a path between nodes with the given names
  ;;          in the given graph
  ;; Assumption: Given node names are in the given graph
  (define (find-path a-graph start end)
  ```

- ```
  ;; Test using sample computations for find-path
  (check-expect (find-path G1 'A 'Z) FP-AZ-G1)
  (check-expect (find-path G2 'C 'Z) FP-AZ-G2)
  (check-expect (find-path G3 'A 'Z) FP-AZ-G3)
  ```

- Care about which path is found

  ```
  ;; Test using sample values for find-path
  (check-expect (find-path G1 'Z 'A) 'no-path)
  (check-expect (find-path G2 'F 'Z) '(F G Z))
  (check-expect (find-path G3 'C 'Z) '(A B C F G Z))
  ```

find-path

Tests and Function Definition

- ```
  ;; graph symbol symbol → path
  ;; Purpose: Find a path between nodes with the given names
  ;;          in the given graph
  ;; Assumption: Given node names are in the given graph
  (define (find-path a-graph start end)
  ```
- ```
    (find-path-acc a-graph start end '()))
  ```
- ```
  ;; Test using sample computations for find-path
  (check-expect (find-path G1 'A 'Z) FP-AZ-G1)
  (check-expect (find-path G2 'C 'Z) FP-AZ-G2)
  (check-expect (find-path G3 'A 'Z) FP-AZ-G3)
  ```
- Care about which path is found

  ```
  ;; Test using sample values for find-path
  (check-expect (find-path G1 'Z 'A) 'no-path)
  (check-expect (find-path G2 'F 'Z) '(F G Z))
  (check-expect (find-path G3 'C 'Z) '(A B C F G Z))
  ```
  ```

find-path-acc

Problem Analysis

- Find a path using DFS without going through visited nodes

find-path-acc

Problem Analysis

- Find a path using DFS without going through visited nodes
- A node is visited if its neighbors are part of the search

find-path-acc

Problem Analysis

- Find a path using DFS without going through visited nodes
- A node is visited if its neighbors are part of the search
- Informally the accumulator invariant is:

```
Accumulator = nodes whose neighbors are part of the search
```

- Initially, the invariant is true because `find-path` provides the empty list as its value

`find-path-acc`

Problem Analysis

- Find a path using DFS without going through visited nodes
- A node is visited if its neighbors are part of the search
- Informally the accumulator invariant is:

```
Accumulator = nodes whose neighbors are part of the search
```

- Initially, the invariant is true because `find-path` provides the empty list as its value
- If the given node names are the same then there is a path between them that only contains the given name

`find-path-acc`

Problem Analysis

- Find a path using DFS without going through visited nodes
- A node is visited if its neighbors are part of the search
- Informally the accumulator invariant is:

```
Accumulator = nodes whose neighbors are part of the search
```

- Initially, the invariant is true because `find-path` provides the empty list as its value
- If the given node names are the same then there is a path between them that only contains the given name
- If the given node names are not equal then `start`'s unvisited neighbors are computed
- Search for a path from any of these neighbors

find-path-acc

Problem Analysis

- Find a path using DFS without going through visited nodes
- A node is visited if its neighbors are part of the search
- Informally the accumulator invariant is:

```
Accumulator = nodes whose neighbors are part of the search
```

- Initially, the invariant is true because `find-path` provides the empty list as its value
- If the given node names are the same then there is a path between them that only contains the given name
- If the given node names are not equal then `start`'s unvisited neighbors are computed
- Search for a path from any of these neighbors
- If a path does not exist then the answer is `'no-path`
- If there is a path, then the path is obtained by adding `start` to the front of it

find-path-acc

Problem Analysis

- Find a path using DFS without going through visited nodes
- A node is visited if its neighbors are part of the search
- Informally the accumulator invariant is:

```
Accumulator = nodes whose neighbors are part of the search
```

- Initially, the invariant is true because `find-path` provides the empty list as its value
- If the given node names are the same then there is a path between them that only contains the given name
- If the given node names are not equal then `start`'s unvisited neighbors are computed
- Search for a path from any of these neighbors
- If a path does not exist then the answer is `'no-path`
- If there is a path, then the path is obtained by adding `start` to the front of it
- Finding a path from one of `start`'s unvisited neighbors to `end` is a different problem and an auxiliary function is needed

find-path-acc

Sample Expressions and Differences

- Find a path from `A` to `A` in `G1` when the accumulator is empty:

```
;; Sample expressions for find-path-acc
(define FPACC-AA-G1 (list 'A))
```

find-path-acc

Sample Expressions and Differences

- Find a path from `A` to `A` in `G1` when the accumulator is empty:

```
;; Sample expressions for find-path-acc
(define FPACC-AA-G1 (list 'A))
```

- Otherwise, the unvisited neighbors from the starting node and the result of searching for a path from any of these neighbors may be locally defined

```
(define FPACC-AZ-G1
        (local [(define new-visited (cons 'A '(C)))
                (define start-unvisited-neighs
                        (filter
                          (λ (s) (not (member? s new-visited)))
                          (node-neighs
                            (first
                              (filter
                                (λ (n) (eq? (node-name n) 'A))
                                G1)))))
                (define path-from-neigh
                        (find-path-from-neighbors
                          G1 start-unvisited-neighs 'Z new-visited))]
          (if (eq? path-from-neigh 'no-path)
              'no-path
              (cons 'A path-from-neigh))))
```

find-path-acc

Signature and Statements

- ```
  ;; graph symbol symbol (listof symbol) → path
  ;; Purpose: Find a path between the given node names in
  ;;          the given graph without going through any
  ;;          visited neighbors
  ```
  ```

find-path-acc

Signature and Statements

- ```
  ;; graph symbol symbol (listof symbol) → path
  ;; Purpose: Find a path between the given node names in
  ;;          the given graph without going through any
  ;;          visited neighbors
  ```
- ```
  ;; Accumulator Invariant:
  ;;   visited = nodes whose neighbors are part of the search
  ;; Assumption: Given node names are in the given graph
  ```

find-path-acc

Tests and Function Definition

- `(define (find-path-acc a-graph start end visited)`

find-path-acc

Tests and Function Definition

- ```
  (define (find-path-acc a-graph start end visited)
  ```

- ```
  ;; Test using sample computations for find-path-acc
  (check-expect (find-path-acc G1 'A 'A '())  FPACC-AA-G1)
  (check-expect (find-path-acc G1 'A 'Z '(C)) FPACC-AZ-G1)
  (check-expect (find-path-acc G3 'Z 'B '())  FPACC-ZB-G3)
  ```

find-path-acc

Tests and Function Definition

- ```
  (define (find-path-acc a-graph start end visited)
  ```

- ```
  ;; Test using sample computations for find-path-acc
  (check-expect (find-path-acc G1 'A 'A '())  FPACC-AA-G1)
  (check-expect (find-path-acc G1 'A 'Z '(C)) FPACC-AZ-G1)
  (check-expect (find-path-acc G3 'Z 'B '())  FPACC-ZB-G3)
  ```

- ```
  ;; Test using sample values for find-path-acc
  (check-expect (find-path-acc G2 'A 'Z '(B D))
                (list 'A 'F 'G 'Z))
  (check-expect (find-path-acc G3 'F 'Z '(G)) 'no-path)
  ```
  ```

find-path-acc

Tests and Function Definition

```
(define (find-path-acc a-graph start end visited)
(if (eq? start end) (list end)
    (local
      [(define start-unvisited-neighs
         (filter (λ (s) (not (member? s visited)))
                 (node-neighs
                  (first (filter (λ (n)
                                   (eq? (node-name n) start))
                                 a-graph)))))
       (define path-from-neigh
         (find-path-from-neighbors
           a-graph start-unvisited-neighs end (cons start visi
       (if (eq? path-from-neigh 'no-path) 'no-path
           (cons start path-from-neigh))))
```

```
;; Test using sample computations for find-path-acc
(check-expect (find-path-acc G1 'A 'A '())  FPACC-AA-G1)
(check-expect (find-path-acc G1 'A 'Z '(C)) FPACC-AZ-G1)
(check-expect (find-path-acc G3 'Z 'B '())  FPACC-ZB-G3)
```

```
;; Test using sample values for find-path-acc
(check-expect (find-path-acc G2 'A 'Z '(B D))
              (list 'A 'F 'G 'Z))
(check-expect (find-path-acc G3 'F 'Z '(G)) 'no-path)
```

find-path-from-neighbors

Problem Analysis

- Find a path from any given neighbor using DFS with backtracking

`find-path-from-neighbors`

Problem Analysis

- Find a path from any given neighbor using DFS with backtracking
- If the neighbor list is empty then the answer is `'no-path`

`find-path-from-neighbors`

Problem Analysis

- Find a path from any given neighbor using DFS with backtracking
- If the neighbor list is empty then the answer is `'no-path`
- If the neighbor list is not empty then the path from the first neighbor to the end node is determined
- Call `find-path-acc`
- The first neighbor is the argument for `start`
- The accumulator is passed unchanged because finding the path from the first neighbor does not add any nodes to the search

`find-path-from-neighbors`

Problem Analysis

- Find a path from any given neighbor using DFS with backtracking
- If the neighbor list is empty then the answer is `'no-path`
- If the neighbor list is not empty then the path from the first neighbor to the end node is determined
- Call `find-path-acc`
- The first neighbor is the argument for `start`
- The accumulator is passed unchanged because finding the path from the first neighbor does not add any nodes to the search
- If there is a `path` it is returned
- If there is no path then backtracking is used to explore the possible paths from the rest of the neighbors

find-path-from-neighbors

Sample Expressions and Differences

- Search for a path to `'A` in `G1` with an empty list of neighboring names and an empty accumulator:

```
;; Sample Expressions for find-path-from-neighbors
(define FPN-ZA-G1 'no-path)
```

find-path-from-neighbors

Sample Expressions and Differences

- Search for a path to 'A in G1 with an empty list of neighboring names and an empty accumulator:

```
;; Sample Expressions for find-path-from-neighbors
(define FPN-ZA-G1 'no-path)
```

- Search for a path in G2 from the neighbors of A to Z with A visited

```
(define FPN-AZ-G2
        (local [(define path-from-first
                        (find-path-acc
                          G2
                          (first '(B D F))
                          'Z
                          '(A)))]
          (if (not (eq? path-from-first 'no-path))
              path-from-first
              (find-path-from-neighbors G2
                                        (rest '(B D F))
                                        'Z
                                        '(A)))))
```

find-path-from-neighbors

Tests and Function Definition

- ```
  ;; graph (listof symbol) symbol (listof symbol) → path
  ;; Purpose: Find a path from any neighbor to the given
  ;;          node in the given graph without going through
  ;;          any visited neighbors
  ;; Accumulator Invariant:
  ;;  visited = nodes whose neighbors are part of the search
  ;; Assumptions: Given neighbors and node name are in the
  ;;                given graph
  ;;              None of the given neighbors are in visited
  ```
  ```

textttfind-path-from-neighbors

Tests and Function Definition

- `(define (find-path-from-neighbors a-graph neighs end visited)`

textttfind-path-from-neighbors

Tests and Function Definition

- ```
  (define (find-path-from-neighbors a-graph neighs end visited)
  ```

- ```
  ;; Tests using sample computations for find-path-from-neighbors
  (check-expect (find-path-from-neighbors G1 '() 'A '()) FPN-ZA-G1)
  (check-expect (find-path-from-neighbors G2 '(B D E) 'Z '(A)) FPN-AZ
  (check-expect (find-path-from-neighbors G3 '(E) 'A '(E)) FPN-FA-G3)
  ```

textttfind-path-from-neighbors

Tests and Function Definition

- ```
  (define (find-path-from-neighbors a-graph neighs end visited)
  ```

- ```
  ;; Tests using sample computations for find-path-from-neighbors
  (check-expect (find-path-from-neighbors G1 '() 'A '()) FPN-ZA-G1)
  (check-expect (find-path-from-neighbors G2 '(B D E) 'Z '(A)) FPN-AZ
  (check-expect (find-path-from-neighbors G3 '(E) 'A '(E)) FPN-FA-G3)
  ```

- ```
  ;; Tests using sample values for find-path-from-neighbors
  (check-expect (find-path-from-neighbors G1 '(C) 'E '(B))
                'no-path)
  (check-expect (find-path-from-neighbors G3 '(G) 'C '(F))
                '(G A B C))
  ```
  ```

textttfind-path-from-neighbors

Tests and Function Definition

- ```
  (define (find-path-from-neighbors a-graph neighs end visited)
  ```
- ```
    (if (empty? neighs)
        'no-path
        (local [(define path-from-first
                  (find-path-acc a-graph (first neighs)
                                 end     visited))]
          (if (not (eq? path-from-first 'no-path))
              path-from-first
              (find-path-from-neighbors a-graph (rest neighs)
                                        end     visited))))
  ```
- ```
  ;; Tests using sample computations for find-path-from-neighbors
  (check-expect (find-path-from-neighbors G1 '() 'A '()) FPN-ZA-G1)
  (check-expect (find-path-from-neighbors G2 '(B D E) 'Z '(A)) FPN-AZ
  (check-expect (find-path-from-neighbors G3 '(E) 'A '(E)) FPN-FA-G3)
  ```
- ```
  ;; Tests using sample values for find-path-from-neighbors
  (check-expect (find-path-from-neighbors G1 '(C) 'E '(B))
                'no-path)
  (check-expect (find-path-from-neighbors G3 '(G) 'C '(F))
                '(G A B C))
  ```

`find-path-from-neighbors`

Termination Argument

- We can observe that `find-path-acc` is never called with the same start node name because this name is placed in the accumulator to prevent following a loop to it

`find-path-from-neighbors`

Termination Argument

- We can observe that `find-path-acc` is never called with the same start node name because this name is placed in the accumulator to prevent following a loop to it
- This means that any path explored does not contain repeated nodes and depth-first search is not caught in an infinite loop due to cycles in the given graph

`find-path-from-neighbors`

Termination Argument

- We can observe that `find-path-acc` is never called with the same start node name because this name is placed in the accumulator to prevent following a loop to it
- This means that any path explored does not contain repeated nodes and depth-first search is not caught in an infinite loop due to cycles in the given graph
- As the path explored gets larger the names in the accumulator increase
  - the neighbors of the start node are all visited
  - the end node is reached
- In both cases the program terminates.

HOMEWORK

- Problem 11 (hard)

Revisiting Insertion Sort

```
;; insert: a-num lon → lon
;; Purpose: To insert a num into a lon sorted in
;; non-decreasing order
(define (insert a-num a-lon)
  (cond [(empty? a-lon) (cons a-num '())]
        [(<= a-num (first a-lon)) (cons a-num a-lon)]
        [else (cons (first a-lon)
                    (insert a-num (rest a-lon)))]))
;; sort: lon → lon
;; Purpose: Sort given lon in nondecreasing order
(define (insertion-sorting a-lon)
  (cond [(empty? a-lon) '()]
        [else (insert (first a-lon)
                      (insertion-sorting (rest a-lon)))]))
```

Revisiting Insertion Sort

- ```
  ;; insert: a-num lon → lon
  ;; Purpose: To insert a num into a lon sorted in
  ;; non-decreasing order
  (define (insert a-num a-lon)
    (cond [(empty? a-lon) (cons a-num '())]
          [(<= a-num (first a-lon)) (cons a-num a-lon)]
          [else (cons (first a-lon)
                      (insert a-num (rest a-lon)))]))
  ;; sort: lon → lon
  ;; Purpose: Sort given lon in nondecreasing order
  (define (insertion-sorting a-lon)
    (cond [(empty? a-lon) '()]
          [else (insert (first a-lon)
                        (insertion-sorting (rest a-lon)))]))
  ```
- In both functions the result of a recursive call is input to another function
  ```

Revisiting Insertion Sort

- ```
  ;; insert: a-num lon → lon
  ;; Purpose: To insert a num into a lon sorted in
  ;; non-decreasing order
  (define (insert a-num a-lon)
    (cond [(empty? a-lon) (cons a-num '())]
          [(<= a-num (first a-lon)) (cons a-num a-lon)]
          [else (cons (first a-lon)
                      (insert a-num (rest a-lon)))]))
  ;; sort: lon → lon
  ;; Purpose: Sort given lon in nondecreasing order
  (define (insertion-sorting a-lon)
    (cond [(empty? a-lon) '()]
          [else (insert (first a-lon)
                        (insertion-sorting (rest a-lon)))]))
  ```
- In both functions the result of a recursive call is input to another function
- Suggests that these functions may benefit from the use of accumulators
  ```

Revisiting Insertion Sort

- ```
  ;; insert: a-num lon → lon
  ;; Purpose: To insert a num into a lon sorted in
  ;; non-decreasing order
  (define (insert a-num a-lon)
    (cond [(empty? a-lon) (cons a-num '())]
          [(<= a-num (first a-lon)) (cons a-num a-lon)]
          [else (cons (first a-lon)
                      (insert a-num (rest a-lon)))]))
  ;; sort: lon → lon
  ;; Purpose: Sort given lon in nondecreasing order
  (define (insertion-sorting a-lon)
    (cond [(empty? a-lon) '()]
          [else (insert (first a-lon)
                        (insertion-sorting (rest a-lon)))]))
  ```
- In both functions the result of a recursive call is input to another function
- Suggests that these functions may benefit from the use of accumulators
- For sample expressions and tests LON0–LON6 and:

  ```
  (define LON7 '(1 2 3 4 5))
  (define LON8 '(-10 -5 0 5 10))
  ```
  ```

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

- Traverses the given sorted list to find the position of the given number

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

- Traverses the given sorted list to find the position of the given number
- As the search progresses a new list is built
- This list may be accumulated as the list traversal progresses and at each step the next element of the sorted list is added to the accumulator.

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

- Traverses the given sorted list to find the position of the given number
- As the search progresses a new list is built
- This list may be accumulated as the list traversal progresses and at each step the next element of the sorted list is added to the accumulator.
-

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)

- In essence, the accumulator is a copy of the processed sublist
- Process continues until the position of the given number is found or the sorted list is empty

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

- Traverses the given sorted list to find the position of the given number
- As the search progresses a new list is built
- This list may be accumulated as the list traversal progresses and at each step the next element of the sorted list is added to the accumulator.
-

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)

- In essence, the accumulator is a copy of the processed sublist
- Process continues until the position of the given number is found or the sorted list is empty
- If the sorted list is empty add the given number at the end of the accumulator
- If the given number is less than or equal to the first number in the sorted list append the accumulator and the consing of the given number and the sorted list
- If the given number is greater than the first number in the sorted list then process the rest of the sorted list and add the first element of the sorted list to the end of the accumulator

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)
- What is true for each row in the table?

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)

- What is true for each row in the table?
- Accumulator contains all the processed numbers in nondecreasing order
- Is this invariant strong enough?

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)

- What is true for each row in the table?
- Accumulator contains all the processed numbers in nondecreasing order
- Is this invariant strong enough?
- Consider the following list: L = '(1 2 3 40 50)
- Assume these values: a-num = 35 a-slon = '(50) acc = '(1 2 3 40)

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)

- What is true for each row in the table?
- Accumulator contains all the processed numbers in nondecreasing order
- Is this invariant strong enough?
- Consider the following list: L = '(1 2 3 40 50)
- Assume these values: a-num = 35 a-slon = '(50) acc = '(1 2 3 40)
- Result: '(1 2 3 40 35 50)

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)

- What is true for each row in the table?
- Accumulator contains all the processed numbers in nondecreasing order
- Is this invariant strong enough?
- Consider the following list: L = '(1 2 3 40 50)
- Assume these values: a-num = 35 a-slon = '(50) acc = '(1 2 3 40)
- Result: '(1 2 3 40 35 50)
- We need a stronger accumulator invariant
- You may argue that the above scenario should never happen because 35 is less than 40 (dynamic reasoning)

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)

- What is true for each row in the table?
- Accumulator contains all the processed numbers in nondecreasing order
- Is this invariant strong enough?
- Consider the following list: L = '(1 2 3 40 50)
- Assume these values: a-num = 35 a-slon = '(50) acc = '(1 2 3 40)
- Result: '(1 2 3 40 35 50)
- We need a stronger accumulator invariant
- You may argue that the above scenario should never happen because 35 is less than 40 (dynamic reasoning)
- You need: *static reasoning* about the values of the variables

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)

- What is true for each row in the table?
- Accumulator contains all the processed numbers in nondecreasing order
- Is this invariant strong enough?
- Consider the following list: L = '(1 2 3 40 50)
- Assume these values: a-num = 35 a-slon = '(50) acc = '(1 2 3 40)
- Result: '(1 2 3 40 35 50)
- We need a stronger accumulator invariant
- You may argue that the above scenario should never happen because 35 is less than 40 (dynamic reasoning)
- You need: *static reasoning* about the values of the variables
- What else is invariant?

Revisiting Insertion Sort

Redesign of `insert`: Problem Analysis

number	Unprocessed list	Accumulator
5	(-4 1 3 4 8)	'()
5	(1 3 4 8)	'(-4)
5	(3 4 8)	'(-4 1)
5	(4 8)	'(-4 1 3)
5	(8)	'(-4 1 3 4)

- What is true for each row in the table?
- Accumulator contains all the processed numbers in nondecreasing order
- Is this invariant strong enough?
- Consider the following list: L = '(1 2 3 40 50)
- Assume these values: a-num = 35 a-slon = '(50) acc = '(1 2 3 40)
- Result: '(1 2 3 40 35 50)
- We need a stronger accumulator invariant
- You may argue that the above scenario should never happen because 35 is less than 40 (dynamic reasoning)
- You need: *static reasoning* about the values of the variables
- What else is invariant?
- For every row accumulator elements are less than the given number
- This makes the accumulator invariant:

```
Accumulator = traversed numbers in nondecreasing order
                        ∧
∀i (list-ref i accum) < a-num, i is a valid index into accum
```

Revisiting Insertion Sort

Redesign of `insert`: Sample Expressions and Differences

- Inserting a number into an empty sorted list means that the answer is built by appending the given number to the accumulator:

```
;; Sample expressions for insert-accum
(define INSERTACC-LON0 (append '()      (list 10)))
(define INSERTACC-EXMP (append '(5 6 7) (list 20)))
```

Revisiting Insertion Sort

Redesign of `insert`: Sample Expressions and Differences

- Inserting a number into an empty sorted list means that the answer is built by appending the given number to the accumulator:

```
;; Sample expressions for insert-accum
(define INSERTACC-LON0 (append '()      (list 10)))
(define INSERTACC-EXMP (append '(5 6 7) (list 20)))
```

- Given number is less than or equal to the first number in the sorted list:

```
(define INSERTACC-LON7 (append '() (cons 0 LON7)))
(define INSERTACC-LON8 (append '(-10 -5)
                               (cons -2 (rest (rest LON8)))))
```

Revisiting Insertion Sort

Redesign of `insert`: Sample Expressions and Differences

- Inserting a number into an empty sorted list means that the answer is built by appending the given number to the accumulator:

```
;; Sample expressions for insert-accum
(define INSERTACC-LON0 (append '()      (list 10)))
(define INSERTACC-EXMP (append '(5 6 7) (list 20)))
```

- Given number is less than or equal to the first number in the sorted list:

```
(define INSERTACC-LON7 (append '() (cons 0 LON7)))
(define INSERTACC-LON8 (append '(-10 -5)
                               (cons -2 (rest (rest LON8)))))
```

- When the given number is greater than the first number in the sorted list:

```
(define INSERTACC-LON7-2
        (insert-accum 7
                      (rest LON7)
                      (append '() (list (first LON7)))))

(define INSERTACC-LON8-2
        (insert-accum 8
                      (rest '(0 5 10))
                      (append '(-10 -5)
                              (list (first '(0 5 10))))))
```

Revisiting Insertion Sort

Redesign of `insert`: Sample Expressions and Differences

- Inserting a number into an empty sorted list means that the answer is built by appending the given number to the accumulator:

```
;; Sample expressions for insert-accum
(define INSERTACC-LON0 (append '()      (list 10)))
(define INSERTACC-EXMP (append '(5 6 7) (list 20)))
```

- Given number is less than or equal to the first number in the sorted list:

```
(define INSERTACC-LON7 (append '() (cons 0 LON7)))
(define INSERTACC-LON8 (append '(-10 -5)
                               (cons -2 (rest (rest LON8)))))
```

- When the given number is greater than the first number in the sorted list:

```
(define INSERTACC-LON7-2
        (insert-accum 7
                      (rest LON7)
                      (append '() (list (first LON7)))))

(define INSERTACC-LON8-2
        (insert-accum 8
                      (rest '(0 5 10))
                      (append '(-10 -5)
                              (list (first '(0 5 10))))))
```

- Differences: number to insert, the sorted list, and the accumulator

Revisiting Insertion Sort

Redesign of `insert`: Signature and Statements

```
;; number lon lon → lon
;; Purpose: To insert the given number in the given
;;          sorted lon
;; Accumulator Invariant:
;;  accum = traversed numbers in nondecreasing order AND
;;  For all i (list-ref i accum) < a-num, where i is a
;;    valid index into accum
;; Assumption: a-slon is sorted in nondecreasing order
(define (insert-accum a-num a-slon accum)
```

Revisiting Insertion Sort

Redesign of insert: Tests

```
;; Tests using sample computations for insert-accum
(check-expect (insert-accum 10 '()  '())
              INSERTACC-LON0)
(check-expect (insert-accum 20 '()  '(5 6 7))
              INSERTACC-EXMP)
(check-expect (insert-accum  0 LON7 '())
              INSERTACC-LON7)
(check-expect (insert-accum -2
                            (rest (rest LON8))
                            '(-10 -5))
              INSERTACC-LON8)
(check-expect (insert-accum  7 LON7 '())
        INSERTACC-LON7-2)
(check-expect (insert-accum  8
                             (rest (rest LON8))
                             '(-10 -5))
              INSERTACC-LON8-2)
```

Revisiting Insertion Sort

Redesign of insert: Tests

```
;; Tests using sample computations for insert-accum
(check-expect (insert-accum 10 '()  '())
              INSERTACC-LON0)
(check-expect (insert-accum 20 '()  '(5 6 7))
              INSERTACC-EXMP)
(check-expect (insert-accum  0 LON7 '())
              INSERTACC-LON7)
(check-expect (insert-accum -2
                            (rest (rest LON8))
                            '(-10 -5))
              INSERTACC-LON8)
(check-expect (insert-accum  7 LON7 '())
        INSERTACC-LON7-2)
(check-expect (insert-accum  8
                             (rest (rest LON8))
                             '(-10 -5))
              INSERTACC-LON8-2)
```

```
;; Tests using sample values for insert-accum
(check-expect (insert-accum 23 '()  '())
              '(23))
(check-expect (insert-accum 31 '(50 50)  '())
              '(31 50 50))
(check-expect (insert-accum 87 '(50 78 90)  '(20 30))
              '(20 30 50 78 87 90))
```

Revisiting Insertion Sort

Redesign of `insert`: Function Body

```
(cond [(empty? a-slon) (append accum (list a-num))]
      [(<= a-num (first a-slon))
       (append accum (cons a-num a-slon))]
      [else (insert-accum a-num
                          (rest a-slon)
                          (append accum (list (first a-slon))))])
```

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Problem Analysis

- Sorts the rest of the list and then inserts the first element
- Does it have to be done in this order?

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Problem Analysis

- Sorts the rest of the list and then inserts the first element
- Does it have to be done in this order?
- Instead, the traversed elements may be accumulated in nondecreasing order
- If the given list is empty then the answer is the accumulator
- If the given list is not empty then the first list element is inserted into the accumulator and the rest of the list is processed recursively

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Problem Analysis

Unprocessed list	Accumulator
(7 9 1 8 6)	'()
(9 1 8 6)	'(7)
(1 8 6)	'(7 9)
(8 6)	'(1 7 9)
(6)	'(1 7 8 9)
()	'(1 6 7 8 9)

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Problem Analysis

Unprocessed list	Accumulator
(7 9 1 8 6)	'()
(9 1 8 6)	'(7)
(1 8 6)	'(7 9)
(8 6)	'(1 7 9)
(6)	'(1 7 8 9)
()	'(1 6 7 8 9)

- Ask yourself what is true about the accumulator at each step
- It may be tempting to state the invariant as follows:

```
Accumulator = The sorted list in nondecreasing order
```

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Problem Analysis

Unprocessed list	Accumulator
(7 9 1 8 6)	'()
(9 1 8 6)	'(7)
(1 8 6)	'(7 9)
(8 6)	'(1 7 9)
(6)	'(1 7 8 9)
()	'(1 6 7 8 9)

- Ask yourself what is true about the accumulator at each step
- It may be tempting to state the invariant as follows:

  `Accumulator = The sorted list in nondecreasing order`

- Only true for the final step

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Problem Analysis

Unprocessed list	Accumulator
(7 9 1 8 6)	'()
(9 1 8 6)	'(7)
(1 8 6)	'(7 9)
(8 6)	'(1 7 9)
(6)	'(1 7 8 9)
()	'(1 6 7 8 9)

- Ask yourself what is true about the accumulator at each step
- It may be tempting to state the invariant as follows:

```
Accumulator = The sorted list in nondecreasing order
```

- Only true for the final step
- The accumulator is not all elements sorted, but the traversed elements sorted:

```
Accumulator = Traversed elements in nondecreasing order
```

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Sample Expressions and Differences

- When the sorted list is empty the accumulator is the answer

```
;; Sample expressions for insertion-sorting-accum
(define INSALON0-VAL '()) acc must be empty to sort LON0
(define INSALON6-VAL LON6) acc must be LON6 (sorted)
```

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Sample Expressions and Differences

- When the sorted list is empty the accumulator is the answer

```
;; Sample expressions for insertion-sorting-accum
(define INSALON0-VAL '()) acc must be empty to sort LON0
(define INSALON6-VAL LON6) acc must be LON6 (sorted)
```

- When the sorted list is not empty the answer is obtained by inserting its first element into the accumulator and recursively process the rest of the list

```
(define INSALON1-VAL (insertion-sorting-accum
                      (rest LON1)
                      (insert-accum (first LON1) '() '())))
(define INSALON4-VAL (insertion-sorting-accum
                      (rest LON4)
                      (insert-accum (first LON4) '() '())))
```

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Sample Expressions and Differences

- When the sorted list is empty the accumulator is the answer

```
;; Sample expressions for insertion-sorting-accum
(define INSALON0-VAL '()) acc must be empty to sort LON0
(define INSALON6-VAL LON6) acc must be LON6 (sorted)
```

- When the sorted list is not empty the answer is obtained by inserting its first element into the accumulator and recursively process the rest of the list

```
(define INSALON1-VAL (insertion-sorting-accum
                      (rest LON1)
                      (insert-accum (first LON1) '() '())))
(define INSALON4-VAL (insertion-sorting-accum
                      (rest LON4)
                      (insert-accum (first LON4) '() '())))
```

- Differences: the `lon` to sort and the accumulator `lon`

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Signature and Statements

```
;; lon lon → lon
;; Purpose: Sort the first given lon in nondecreasing order
;; Accumulator Invariant:
;;  accum = traversed elements in nondecreasing order
(define (insertion-sorting-accum a-lon accum)
```

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Problem Analysis

- ```
  ;; Tests using sample computations for
  ;;   insertion-sorting-accum
  (check-expect (insertion-sorting-accum LON0 '())
                INSALON0-VAL)
  (check-expect (insertion-sorting-accum '() LON6)
                INSALON6-VAL)
  (check-expect (insertion-sorting-accum LON1 '())
                INSALON1-VAL)
  (check-expect (insertion-sorting-accum LON4 '())
                INSALON4-VAL)
  ```
  ```

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Problem Analysis

- ```
  ;; Tests using sample computations for
  ;;   insertion-sorting-accum
  (check-expect (insertion-sorting-accum LON0 '())
                INSALON0-VAL)
  (check-expect (insertion-sorting-accum '() LON6)
                INSALON6-VAL)
  (check-expect (insertion-sorting-accum LON1 '())
                INSALON1-VAL)
  (check-expect (insertion-sorting-accum LON4 '())
                INSALON4-VAL)
  ```
- ```
  ;; Tests using sample values for insertion-sorting-accum
  (check-expect (insertion-sorting-accum '(3 1 8) '())
                '(1 3 8))
  (check-expect (insertion-sorting-accum '(3 1 8) '(1 8 9))
                '(1 1 3 8 8 9))
  ```

Revisiting Insertion Sort

Redesign of `insertion-sorting`: Function Body

```
(cond [(empty? a-lon) accum]
      [else
       (insertion-sorting-accum
         (rest a-lon)
         (insert-accum (first a-lon) accum '()))])
```

Revisiting Insertion Sort

Performance and Complexity Analysis

- Has anything been gained by using accumulators?

Revisiting Insertion Sort

Performance and Complexity Analysis

- Has anything been gained by using accumulators?
- Experiments executed five times:

```
(time (insertion-sorting       LON5)))
(time (insertion-sorting-accum LON5 '()))
```

- CPU times in milliseconds (after subtracting garbage collection time):

	Structural Recursion	Accumulative Recursion
Run 1	1203	6844
Run 2	1515	7701
Run 3	1375	6422
Run 4	1375	8687
Run 5	1375	7859

- Alas, using accumulators has made insertion sorting slower.

Revisiting Insertion Sort

Performance and Complexity Analysis

- Has anything been gained by using accumulators?
- Experiments executed five times:

```
(time (insertion-sorting       LON5)))
(time (insertion-sorting-accum LON5 '()))
```

- CPU times in milliseconds (after subtracting garbage collection time):

	Structural Recursion	Accumulative Recursion
Run 1	1203	6844
Run 2	1515	7701
Run 3	1375	6422
Run 4	1375	8687
Run 5	1375	7859

- Alas, using accumulators has made insertion sorting slower.
- Without accumulators: $O(n^2)$

Revisiting Insertion Sort

Performance and Complexity Analysis

- Has anything been gained by using accumulators?
- Experiments executed five times:

```
(time (insertion-sorting       LON5)))
(time (insertion-sorting-accum LON5 '()))
```

- CPU times in milliseconds (after subtracting garbage collection time):

	Structural Recursion	Accumulative Recursion
Run 1	1203	6844
Run 2	1515	7701
Run 3	1375	6422
Run 4	1375	8687
Run 5	1375	7859

- Alas, using accumulators has made insertion sorting slower.
- Without accumulators: $O(n^2)$
- In the worst case `insert-accum` is called `n + 1` and append traverses an accumulator of size `n`: $O(n^2)$.

Revisiting Insertion Sort

Performance and Complexity Analysis

- Has anything been gained by using accumulators?
- Experiments executed five times:

```
(time (insertion-sorting       LON5)))
(time (insertion-sorting-accum LON5 '()))
```

- CPU times in milliseconds (after subtracting garbage collection time):

	Structural Recursion	Accumulative Recursion
Run 1	1203	6844
Run 2	1515	7701
Run 3	1375	6422
Run 4	1375	8687
Run 5	1375	7859

- Alas, using accumulators has made insertion sorting slower.
- Without accumulators: $O(n^2)$
- In the worst case `insert-accum` is called `n + 1` and append traverses an accumulator of size `n`: $O(n^2)$.
- `insertion-sorting-accum` is called `n + 1` and for each call but the last `insert-accum` is called: $O(n * O(n^2)) = O(n^3)$

Revisiting Insertion Sort

Performance and Complexity Analysis

- Has anything been gained by using accumulators?
- Experiments executed five times:

```
(time (insertion-sorting       LON5)))
(time (insertion-sorting-accum LON5 '()))
```

- CPU times in milliseconds (after subtracting garbage collection time):

	Structural Recursion	Accumulative Recursion
Run 1	1203	6844
Run 2	1515	7701
Run 3	1375	6422
Run 4	1375	8687
Run 5	1375	7859

- Alas, using accumulators has made insertion sorting slower.
- Without accumulators: $O(n^2)$
- In the worst case `insert-accum` is called `n + 1` and append traverses an accumulator of size `n`: $O(n^2)$.
- `insertion-sorting-accum` is called `n + 1` and for each call but the last `insert-accum` is called: $O(n * O(n^2)) = O(n^3)$
- **Accumulative recursion is not always faster**

Revisiting Insertion Sort

Performance and Complexity Analysis

- Has anything been gained by using accumulators?
- Experiments executed five times:

```
(time (insertion-sorting       LON5)))
(time (insertion-sorting-accum LON5 '()))
```

- CPU times in milliseconds (after subtracting garbage collection time):

	Structural Recursion	Accumulative Recursion
Run 1	1203	6844
Run 2	1515	7701
Run 3	1375	6422
Run 4	1375	8687
Run 5	1375	7859

- Alas, using accumulators has made insertion sorting slower.
- Without accumulators: $O(n^2)$
- In the worst case `insert-accum` is called `n + 1` and append traverses an accumulator of size `n`: $O(n^2)$.
- `insertion-sorting-accum` is called `n + 1` and for each call but the last `insert-accum` is called: $O(n * O(n^2)) = O(n^3)$
- **Accumulative recursion is not always faster**
- Studies have shown that problem solvers that take the time to determine accumulator invariants tend to write more bug-free and understandable code.

HOMEWORK and QUIZ

- HOMEWORK: 12, 13
- QUIZ: 14 (due in 1 week)

N-Puzzle Version 4

- DFS may fall into an infinite recursion when repeatedly visiting the same `worlds` (i.e., a path that has a loop)

N-Puzzle Version 4

- DFS may fall into an infinite recursion when repeatedly visiting the same `worlds` (i.e., a path that has a loop)
- BFS solved this problem but repeatedly searches for solutions starting from the same world

N-Puzzle Version 4

- DFS may fall into an infinite recursion when repeatedly visiting the same `worlds` (i.e., a path that has a loop)
- BFS solved this problem but repeatedly searches for solutions starting from the same world
- Finding a solution for the N-puzzle problem may benefit from using accumulators
- Remember the `worlds` whose successors are part of the search
- Data of arbitrary size suggests using a `(listof world)`

N-Puzzle Version 4

- DFS may fall into an infinite recursion when repeatedly visiting the same `worlds` (i.e., a path that has a loop)
- BFS solved this problem but repeatedly searches for solutions starting from the same world
- Finding a solution for the N-puzzle problem may benefit from using accumulators
- Remember the `worlds` whose successors are part of the search
- Data of arbitrary size suggests using a `(listof world)`
- For DFS this means that backtracking is needed to search for a different solution
- For BFS this means that not all paths in the search space are explored

N-Puzzle Version 4

Problem Analysis

- `make-move` refactored to call `find-solution` with an initial accumulator

N-Puzzle Version 4

Problem Analysis

- `make-move` refactored to call `find-solution` with an initial accumulator
- `find-solution` must be redesigned to exploit the accumulator and maintain the accumulator invariant
- The test that led to discovering that using depth-first search may lead to an infinite recursion is added to the test suite:

```
(check-expect (process-key (make-world 2 3 0
                                       1 5 6
                                       4 7 8)
                           HKEY)
              (make-world 2 0 3
                          1 5 6
                          4 7 8))
```

N-Puzzle Version 4

Problem Analysis

- `make-move` refactored to call `find-solution` with an initial accumulator
- `find-solution` must be redesigned to exploit the accumulator and maintain the accumulator invariant
- The test that led to discovering that using depth-first search may lead to an infinite recursion is added to the test suite:

```
(check-expect (process-key (make-world 2 3 0
                                       1 5 6
                                       4 7 8)
                           HKEY)
              (make-world 2 0 3
                          1 5 6
                          4 7 8))
```

- `make-move` may be refactored to:

```
;; world → world   Purpose: Make a move for the player
(define (make-move a-world)
  (if (equal? a-world WIN)
      a-world
      (second (find-solution a-world '()))))
```

- The refined sample expressions are:

```
;; Sample expressions for make-move
(define MM-WIN-VAL   WIN)
(define MM-WRLD1-VAL (second (find-solution WRLD1 '())))
(define MM-WRLD2-VAL (second (find-solution WRLD2 '())))
```

N-Puzzle Version 4

`find-solution`: Problem Analysis

- Input a `world` and the list of `worlds` whose successors are part of the search

N-Puzzle Version 4

`find-solution`: Problem Analysis

- Input a `world` and the list of `worlds` whose successors are part of the search
- The successors of the given `world` must be filtered to eliminate those that are already visited

N-Puzzle Version 4

`find-solution`: Problem Analysis

- Input a `world` and the list of `worlds` whose successors are part of the search
- The successors of the given `world` must be filtered to eliminate those that are already visited
- Process continues by finding a solution from any of the remaining successors

N-Puzzle Version 4

`find-solution`: Problem Analysis

- Input a `world` and the list of `worlds` whose successors are part of the search
- The successors of the given `world` must be filtered to eliminate those that are already visited
- Process continues by finding a solution from any of the remaining successors
- Finding a solution from a set of successor `worlds` is a different problem from finding a solution from a `world`and an auxiliary function is needed

N-Puzzle Version 4

`find-solution`: Problem Analysis

- Input a `world` and the list of `worlds` whose successors are part of the search
- The successors of the given `world` must be filtered to eliminate those that are already visited
- Process continues by finding a solution from any of the remaining successors
- Finding a solution from a set of successor `worlds` is a different problem from finding a solution from a `world`and an auxiliary function is needed
- Accumulator invariant:

```
visited = list of worlds whose successors are part of
          the search
```

- Accumulator is exploited to determine which successors of the given `world` need to be added to the search for a solution

N-Puzzle Version 4

`find-solution`: Problem Analysis

- Input a `world` and the list of `worlds` whose successors are part of the search
- The successors of the given `world` must be filtered to eliminate those that are already visited
- Process continues by finding a solution from any of the remaining successors
- Finding a solution from a set of successor `worlds` is a different problem from finding a solution from a `world`and an auxiliary function is needed
- Accumulator invariant:

```
visited = list of worlds whose successors are part of
          the search
```

- Accumulator is exploited to determine which successors of the given `world` need to be added to the search for a solution
- Solution returned from searching the successors is `'no-solution` then the answer is `'no-solution`
- Otherwise, the solution is obtained by adding the given `world` to the front of the solution returned by searching for a solution from any of the successors

N-Puzzle Version 4

`find-solution`: Sample Expressions and Differences

- If the given `world` is `WIN` then the answer is always the list containing `WIN`:

```
;; Sample expressions for find-solution
(define FS-WIN-VAL  (list WIN))
(define FS-WIN-VAL2 (list WIN))
```

N-Puzzle Version 4

`find-solution`: Sample Expressions and Differences

- If the given `world` is not `WIN` :

```
(define FS-WRLD1-VAL
        (local
         [(define succs
                  (filter (λ (w) (not (member? WRLD1 '())))
                          (map (λ (neigh) (swap-empty WRLD1 neigh))
                               (list-ref neighbors (blank-pos WRLD1
          (define solution-from-any-succ
           (find-solution-from-any-succ succs (cons WRLD1 '())))]
          (if (eq? solution-from-any-succ 'no-solution)
              'no-solution
              (cons WRLD1 solution-from-any-succ))))
```

N-Puzzle Version 4

`find-solution`: Sample Expressions and Differences

- If the given `world` is not `WIN` :

```
(define FS-WRLD2-VAL
        (local
         [(define succ
                  (filter (λ (w) (not (member? w (list WRLD2))))
                   (map (λ (neigh)
                          (swap-empty (make-world 1 3 0 4 2 6 7 5 8)
                                      neigh))
                        (list-ref neighbors (blank-pos
                                             (make-world 1 3 0
                                                         4 2 6
                                                         7 5 8))))))
          (define solution-from-any-succ
                  (find-solution-from-any-succ
                     succs
                     (cons (make-world 1 3 0 4 2 6 7 5 8)
                           (list WRLD2))))]
          (if (eq? solution-from-any-succ 'no-solution)
              'no-solution
              (cons (make-world 1 3 0 4 2 6 7 5 8)
                    solution-from-any-succ))))
```

N-Puzzle Version 4

find-solution: Sample Expressions and Differences

- If the given world is not WIN :

```
(define FS-WRLD2-VAL
        (local
         [(define succ
                  (filter (λ (w) (not (member? w (list WRLD2))))
                   (map (λ (neigh)
                          (swap-empty (make-world 1 3 0 4 2 6 7 5 8)
                                      neigh))
                        (list-ref neighbors (blank-pos
                                             (make-world 1 3 0
                                                         4 2 6
                                                         7 5 8))))))
          (define solution-from-any-succ
                  (find-solution-from-any-succ
                     succs
                     (cons (make-world 1 3 0 4 2 6 7 5 8)
                           (list WRLD2))))]
          (if (eq? solution-from-any-succ 'no-solution)
              'no-solution
              (cons (make-world 1 3 0 4 2 6 7 5 8)
                    solution-from-any-succ))))
```

- Differences: the given world and the given (listof world) for the accumulator

N-Puzzle Version 4

`find-solution`: Signature and Statements

- ```
  ;; world (listof world) → (listof world)
  ;; Purpose: Return a sequence of moves to WIN that does
  ;;          not go through worlds in the accumulator.
  ;; How: The solution is built using the given world
  ;;      and the first successor found by depth-first
  ;;      search that leads to WIN without exploring
  ;;      successors of the given world that are in
  ;;      the accumulator.
  ;; Accumulator Invariant
  ;;  visited = list of worlds whose children are part of
  ;;            the search
  ```
  ```

N-Puzzle Version 4

`find-solution`: Tests and Function Definition

- `(define (find-solution a-world visited)`

N-Puzzle Version 4

`find-solution`: Tests and Function Definition

- `(define (find-solution a-world visited)`

- ```
  ;; Tests using sample computations for find-solution
  (check-expect (find-solution WIN '()) FS-WIN-VAL)
  (check-expect (find-solution WIN (list WRLD1 WRLD2)) FS-WIN-VAL2)
       ...
  ```
  ```

N-Puzzle Version 4

find-solution: Tests and Function Definition

- `(define (find-solution a-world visited)`

- ```
  ;; Tests using sample computations for find-solution
  (check-expect (find-solution WIN '()) FS-WIN-VAL)
  (check-expect (find-solution WIN (list WRLD1 WRLD2)) FS-WIN-VAL2)
       ...
  ```
- ```
  ;; Tests using sample values for find-solution
  (check-expect (find-solution (make-world 1 2 3 4 0 6 7 5 8) '())
                (list (make-world 1 2 3 4 0 6 7 5 8)
                      (make-world 1 2 3 4 5 6 7 0 8)
                      (make-world 1 2 3 4 5 6 7 8 0)))
  ```

N-Puzzle Version 4

find-solution: Tests and Function Definition

```
(define (find-solution a-world visited)
 (if (equal? a-world WIN)
     (list a-world)
     (local
      [(define succs
               (filter (λ (w) (not (member? w accum)))
                       (map (λ (neigh)
                              (swap-empty a-world neigh))
                            (list-ref neighbors blank-pos a-world))))
       (define solution-from-any-succ
               (find-solution-from-any-succ succs
                                            (cons a-world visited)))]
       (if (eq? solution-from-any-succ 'no-solution)
           'no-solution
           (cons a-world solution-from-any-succ)))))
```

```
;; Tests using sample computations for find-solution
(check-expect (find-solution WIN '()) FS-WIN-VAL)
(check-expect (find-solution WIN (list WRLD1 WRLD2)) FS-WIN-VAL2)
     ...
```

```
;; Tests using sample values for find-solution
(check-expect (find-solution (make-world 1 2 3 4 0 6 7 5 8) '())
              (list (make-world 1 2 3 4 0 6 7 5 8)
                    (make-world 1 2 3 4 5 6 7 0 8)
                    (make-world 1 2 3 4 5 6 7 8 0)))
```

N-Puzzle Version 4

`find-solution-from-any-succ`: Problem Analysis

- Searche for a solution by traversing the given list of successor `worlds` using DFS without going through any `world` in the accumulator

N-Puzzle Version 4

`find-solution-from-any-succ`: Problem Analysis

- Searche for a solution by traversing the given list of successor `worlds` using DFS without going through any `world` in the accumulator
- If the given list of successor `worlds` is empty the answer is `'no-solution`

N-Puzzle Version 4

`find-solution-from-any-succ`: Problem Analysis

- Searche for a solution by traversing the given list of successor `worlds` using DFS without going through any `world` in the accumulator
- If the given list of successor `worlds` is empty the answer is `'no-solution`
- If the first successor is `WIN` then the solution is the list containing `WIN`.

N-Puzzle Version 4

`find-solution-from-any-succ`: Problem Analysis

- Searche for a solution by traversing the given list of successor `worlds` using DFS without going through any `world` in the accumulator
- If the given list of successor `worlds` is empty the answer is `'no-solution`
- If the first successor is `WIN` then the solution is the list containing `WIN`.
- Otherwise, a search for a solution is performed with the first successor and the given accumulator by calling `find-solution`
- If this search returns a sequence of moves then the solution is constructed by adding the first successor to it
- If the search returns `'no-solution` then backtracking must take place and search continues using the rest of the given successors and a new accumulator that includes the first successor

N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- Three conditions: require sample expressions for each

N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- Three conditions: require sample expressions for each
- The answer is always `'no-solution` when the given successor list is empty. Sample expressions to illustrate this condition may be written for an empty successor list and either an empty visited list or a visited list containing `A-WRLD` as follows:

```
;; Sample expressions for find-solution-from-any-succ
(define FSFAS-EMPSUCCS1 'no-solution) successors include is empty
(define FSFAS-EMPSUCCS2 'no-solution) successors include A-WRLD
```

N-Puzzle Version 4

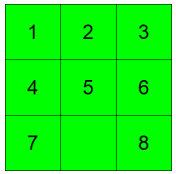
`find-solution-from-any-succ`: Sample Expressions

- When the first successor is `WIN` the answer is always a list containing `WIN`

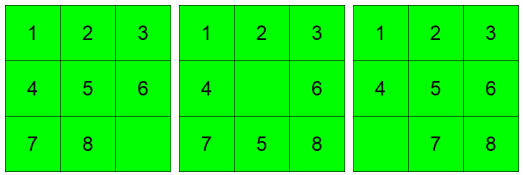
N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- When the first successor is `WIN` the answer is always a list containing `WIN`
- Consider finding a solution starting with the following board:



This `world` is added to the accumulator when its successors:

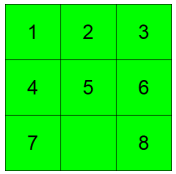


are given as input to `find-solution-from-any-succ`

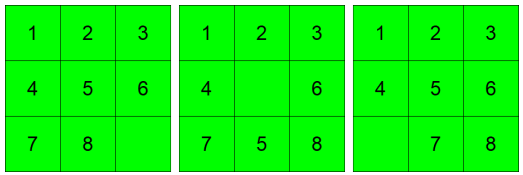
N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- When the first successor is `WIN` the answer is always a list containing `WIN`
- Consider finding a solution starting with the following board:



This `world` is added to the accumulator when its successors:



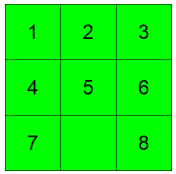
are given as input to `find-solution-from-any-succ`

- Observe that the successors appear in the same order defined by `neighbors` (the first successor is `WIN`)

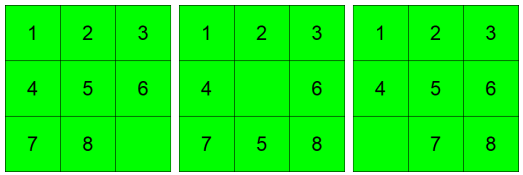
N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- When the first successor is `WIN` the answer is always a list containing `WIN`
- Consider finding a solution starting with the following board:



  This `world` is added to the accumulator when its successors:

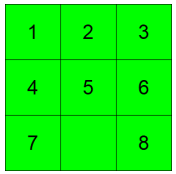


  are given as input to `find-solution-from-any-succ`
- Observe that the successors appear in the same order defined by `neighbors` (the first successor is `WIN`)
- ```
  (define FSFAS-WIN1 (list WIN)) successor are empty
  (define FSFAS-WIN2 (list WIN))
  ```
  ```

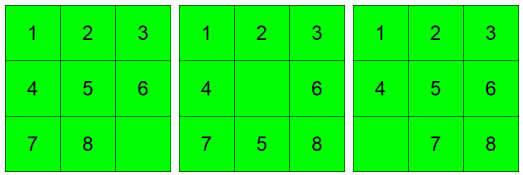
N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- When the first successor is `WIN` the answer is always a list containing `WIN`
- Consider finding a solution starting with the following board:



This `world` is added to the accumulator when its successors:



are given as input to `find-solution-from-any-succ`

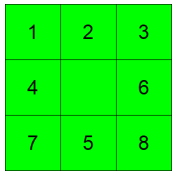
- Observe that the successors appear in the same order defined by `neighbors` (the first successor is `WIN`)
- ```
  (define FSFAS-WIN1 (list WIN)) successor are empty
  (define FSFAS-WIN2 (list WIN))
  ```
- For easy testing:

```
(define SUCCS-WIN2
        (list WIN
              (make-world 1 2 3 4 0 6 7 5 8)
              (make-world 1 2 3 4 5 6 0 7 8)))
(define ACCUM-WIN2
        (list (make-world 1 2 3 4 5 6 7 0 8)))
```
```

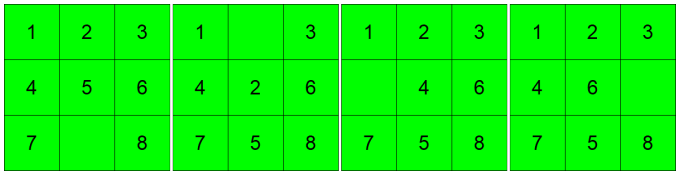
N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- Any sample expression for the third condition requires a nonempty successor list:



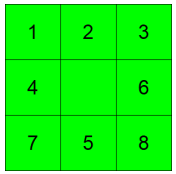
has the following successors:



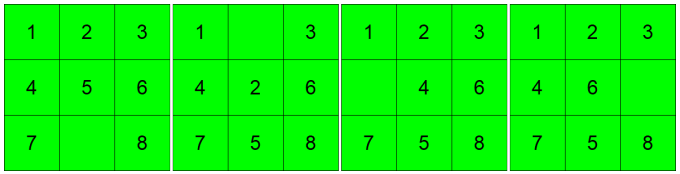
N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- Any sample expression for the third condition requires a nonempty successor list:



has the following successors:



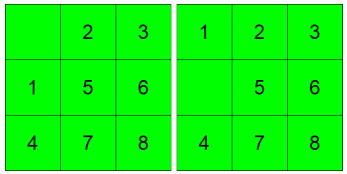
- Variables for the successors and the accumulator are defined:

```
(define SUCCS-WRLD1
        (list (make-world 1 2 3 4 5 6 7 0 8)
              (make-world 1 0 3 4 2 6 7 5 8)
              (make-world 1 2 3 0 4 6 7 5 8)
              (make-world 1 2 3 4 6 0 7 5 8)))
(define VISITED1 (list (make-world 1 2 3 4 0 6 7 5 8)))
```

N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

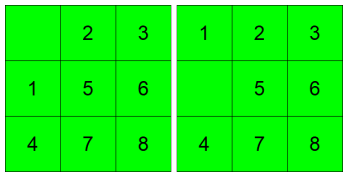
- Consider an accumulator with the following worlds after a move:



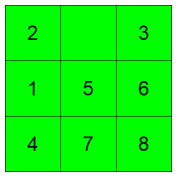
N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- Consider an accumulator with the following worlds after a move:



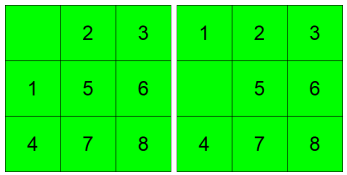
- There is only one unvisited successor for `bpos = 0`:



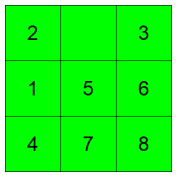
N-Puzzle Version 4

`find-solution-from-any-succ`: Sample Expressions

- Consider an accumulator with the following worlds after a move:



- There is only one unvisited successor for `bpos = 0`:



- Variables for the successor and the accumulator are defined:

```
(define SUCCS-WRLD2 (list (make-world 2 0 3 1 5 6 4 7 8)))

(define VISITED2 (list (make-world 0 2 3 1 5 6 4 7 8)
                       (make-world 1 2 3 0 5 6 4 7 8)))
```

N-Puzzle Version 4

find-solution-from-any-succ: Sample Expressions

```
(define FSFAS-SUCC1
        (local
          [(define solution-from-first-succ
                   (find-solution (first SUCCS-WRLD1)
                                  VISITED1))]
             (if (not (eq? solution-from-first-succ
                           'no-solution))
                 solution-from-first-succ
                 (find-solution-from-any-succ
                   (rest SUCCS-WRLD1)
                   (cons (first SUCCS-WRLD1) VISITED1)))))
(define FSFAS-SUCC2
   (local [(define solution-from-first-succ
                   (find-solution (first SUCCS-WRLD2)
                                  VISITED2))]
             (if (not (eq? solution-from-first-succ
                           'no-solution))
                 solution-from-first-succ
                 (find-solution-from-any-succ
                   (rest SUCCS-WRLD2)
                   (cons (first SUCCS-WRLD2) VISITED2)))))
```

N-Puzzle Version 4

find-solution-from-any-succ: Sample Expressions

```
(define FSFAS-SUCC1
        (local
          [(define solution-from-first-succ
                   (find-solution (first SUCCS-WRLD1)
                                  VISITED1))]
             (if (not (eq? solution-from-first-succ
                           'no-solution))
                 solution-from-first-succ
                 (find-solution-from-any-succ
                   (rest SUCCS-WRLD1)
                   (cons (first SUCCS-WRLD1) VISITED1)))))
(define FSFAS-SUCC2
   (local [(define solution-from-first-succ
                   (find-solution (first SUCCS-WRLD2)
                                  VISITED2))]
             (if (not (eq? solution-from-first-succ
                           'no-solution))
                 solution-from-first-succ
                 (find-solution-from-any-succ
                   (rest SUCCS-WRLD2)
                   (cons (first SUCCS-WRLD2) VISITED2)))))
```

- Two differences: `(listof world)` for successors and for acc

N-Puzzle Version 4

`find-solution-from-any-succ`: Siganture, Statements, and Function Header

- ```
  ;; (listof world) (listof world) → (listof world)
  ;; Purpose: Find a solution from any world in the first
  ;;          given list without going through any world
  ;;          in the second given list.
  ;; Accumulator Invariant:
  ;;  visited = list of worlds whose successors are part
  ;;            of the search
  (define (find-solution-from-any-succ succs visited)
  ```
  ```

N-Puzzle Version 4

find-solution-from-any-succ: Tests

- ```
  ;; Tests using sample computations for find-solution-from-any-succ
  (check-expect (find-solution-from-any-succ '() '()) FSFAS-EMPSUCCS1
  (check-expect (find-solution-from-any-succ '() (list A-WRLD))
                FSFAS-EMPSUCCS2)     ...
  ```
  ```

N-Puzzle Version 4

find-solution-from-any-succ: Tests

- ```
  ;; Tests using sample computations for find-solution-from-any-succ
  (check-expect (find-solution-from-any-succ '() '()) FSFAS-EMPSUCCS1
  (check-expect (find-solution-from-any-succ '() (list A-WRLD))
                FSFAS-EMPSUCCS2)     ...
  ```
- Writing tests using sample values requires determining the solution the function ought to return after a series of moves has been made

```
;; Tests using sample values for find-solution-from-any-succ
(check-expect (find-solution-from-any-succ
               (list (make-world 1 0 3 4 2 6 7 5 8))
               (list (make-world 1 3 0 4 2 6 7 5 8)
                     (make-world 1 3 6 4 2 0 7 5 8)))
              (list
               (make-world 1 0 3 4 2 6 7 5 8)
               (make-world 1 2 3 4 0 6 7 5 8)
               (make-world 1 2 3 4 5 6 7 0 8)
               (make-world 1 2 3 4 5 6 7 8 0)))
(check-expect (find-solution-from-any-succ
                (list (make-world 1 0 3 4 2 6 7 5 8))
                '())
              (list
               (make-world 1 0 3 4 2 6 7 5 8)
               (make-world 1 2 3 4 0 6 7 5 8)
               (make-world 1 2 3 4 5 6 7 0 8)
               (make-world 1 2 3 4 5 6 7 8 0)))
```
```

N-Puzzle Version 4

find-solution-from-any-succ: Function Body

```
(cond
    [(empty? succs) 'no-solution]
    [(equal? (first succs) WIN) (list WIN)]
    [else
     (local [(define solution-from-first-succ
               (find-solution (first succs) visited))]
       (if (not (eq? solution-from-first-succ
                     'no-solution))
           solution-from-first-succ
           (find-solution-from-any-succ
            (rest succs)
            (cons (first succs) visited))))])
```

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Establish that the accumulator invariant is maintained

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Establish that the accumulator invariant is maintained
- Consider the two calls to `find-solution` in the program

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Establish that the accumulator invariant is maintained
- Consider the two calls to `find-solution` in the program
  - Made from `make-move` with an empty accumulator
  - Clearly, the accumulator invariant is true because the successors of no `world` are part of the search

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Establish that the accumulator invariant is maintained
- Consider the two calls to `find-solution` in the program
  - Made from `make-move` with an empty accumulator
  - Clearly, the accumulator invariant is true because the successors of no `world` are part of the search
  - The call from `find-solution-from-any-succ` is made with an unvisited successor and the unchanged list of accumulated visited `worlds`
  - The accumulator invariant is maintained given that no new successor `worlds` are part of the search

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Establish that the accumulator invariant is maintained
- Consider the two calls to `find-solution` in the program
  - Made from `make-move` with an empty accumulator
  - Clearly, the accumulator invariant is true because the successors of no `world` are part of the search
  - The call from `find-solution-from-any-succ` is made with an unvisited successor and the unchanged list of accumulated visited `worlds`
  - The accumulator invariant is maintained given that no new successor `worlds` are part of the search
- Two calls to `find-solution-from-any-succ`

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Establish that the accumulator invariant is maintained
- Consider the two calls to `find-solution` in the program
  - Made from `make-move` with an empty accumulator
  - Clearly, the accumulator invariant is true because the successors of no `world` are part of the search
  - The call from `find-solution-from-any-succ` is made with an unvisited successor and the unchanged list of accumulated visited `worlds`
  - The accumulator invariant is maintained given that no new successor `worlds` are part of the search
- Two calls to `find-solution-from-any-succ`
  - From `find-solution` is made with a list of unvisited successors and an accumulator that contains all the `worlds` whose successors are part of the search including its given `world`

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Establish that the accumulator invariant is maintained
- Consider the two calls to `find-solution` in the program
  - Made from `make-move` with an empty accumulator
  - Clearly, the accumulator invariant is true because the successors of no `world` are part of the search
  - The call from `find-solution-from-any-succ` is made with an unvisited successor and the unchanged list of accumulated visited `worlds`
  - The accumulator invariant is maintained given that no new successor `worlds` are part of the search
- Two calls to `find-solution-from-any-succ`
  - From `find-solution` is made with a list of unvisited successors and an accumulator that contains all the `worlds` whose successors are part of the search including its given `world`
  - Recursive is made with a list of unvisited successors and an accumulator that containing all `worlds` whose successors are part of the search including the `world` that failed to produce a solution

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Establish that the accumulator invariant is maintained
- Consider the two calls to `find-solution` in the program
  - Made from `make-move` with an empty accumulator
  - Clearly, the accumulator invariant is true because the successors of no `world` are part of the search
  - The call from `find-solution-from-any-succ` is made with an unvisited successor and the unchanged list of accumulated visited `worlds`
  - The accumulator invariant is maintained given that no new successor `worlds` are part of the search
- Two calls to `find-solution-from-any-succ`
  - From `find-solution` is made with a list of unvisited successors and an accumulator that contains all the `worlds` whose successors are part of the search including its given `world`
  - Recursive is made with a list of unvisited successors and an accumulator that containing all `worlds` whose successors are part of the search including the `world` that failed to produce a solution
  - Both calls, therefore, maintain the accumulator invariant

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Invariant is maintained means two things:
  - `find-solution` can never loop back to the same `world` and, therefore, fall into an infinite recursion
  - `find-solution-from-any-succ` will not repeat a failed search starting from the given `world`

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Invariant is maintained means two things:
  - `find-solution` can never loop back to the same `world` and, therefore, fall into an infinite recursion
  - `find-solution-from-any-succ` will not repeat a failed search starting from the given `world`
- Given that there is a solution for any given board, the above implies that eventually `find-solution` gets `WIN` as input and the program terminates.

N-Puzzle Version 4

`find-solution-from-any-succ`: Termination Argument

- Invariant is maintained means two things:
  - `find-solution` can never loop back to the same `world` and, therefore, fall into an infinite recursion
  - `find-solution-from-any-succ` will not repeat a failed search starting from the given `world`
- Given that there is a solution for any given board, the above implies that eventually `find-solution` gets `WIN` as input and the program terminates.
- Test the program! Does it work? Does it work well?

N-Puzzle Version 4

HOMEWORK

- Problem 2

N-Puzzle Version 5

Problem Analysis

- Introduce an accumulator to reduce the work done by BFS
- Avoid exploring the same paths repeatedly

N-Puzzle Version 5

Problem Analysis

- Introduce an accumulator to reduce the work done by BFS
- Avoid exploring the same paths repeatedly
- Consider a queue that has the following paths after making one move:

  $(w_1\ w_0)$ $(w_2\ w_0)$ $(w_3\ w_0)$ $(w_4\ w_0)$

N-Puzzle Version 5

Problem Analysis

- Introduce an accumulator to reduce the work done by BFS
- Avoid exploring the same paths repeatedly
- Consider a queue that has the following paths after making one move:

  $(w_1\ w_0)$ $(w_2\ w_0)$ $(w_3\ w_0)$ $(w_4\ w_0)$

- After generating new paths from the first path the queue becomes:

  $(w_2\ w_0)$
  $(w_3\ w_0)$
  $(w_4\ w_0)$
  $(w_5\ w_1\ w_0)$
  $(w_0\ w_1\ w_0)$
  $(w_6\ w_1\ w_0)$

- The next to last path reverses the first move from w_0 to w_1

N-Puzzle Version 5

Problem Analysis

- Introduce an accumulator to reduce the work done by BFS
- Avoid exploring the same paths repeatedly
- Consider a queue that has the following paths after making one move:

  $(w_1\ w_0)$ $(w_2\ w_0)$ $(w_3\ w_0)$ $(w_4\ w_0)$

- After generating new paths from the first path the queue becomes:

  $(w_2\ w_0)$
  $(w_3\ w_0)$
  $(w_4\ w_0)$
  $(w_5\ w_1\ w_0)$
  $(w_0\ w_1\ w_0)$
  $(w_6\ w_1\ w_0)$

- The next to last path reverses the first move from w_0 to w_1
- An accumulator can be added to `find-solution-bfs` to remember the `worlds` whose successors are already part of the search
- Use it to reduce the number of new paths generated
- Only successors not in the accumulator are used to generate new paths. For the example above the new queue is:

  $(w_2\ w_0)$
  $(w_3\ w_0)$
  $(w_4\ w_0)$
  $(w_5\ w_1\ w_0)$
  $(w_6\ w_1\ w_0)$

N-Puzzle Version 5

Sample Expressions

- When the first `world` of the queue's first path is `WIN`:

```
;; Sample expressions for find-solution-bfs
(define FS-QLOW1-VAL
  (local [(define first-path (qfirst QLOW1))]
    (reverse first-path)))

(define FS-QLOW2-VAL
  (local [(define first-path (qfirst QLOW2))]
    (reverse first-path)))
```

N-Puzzle Version 5

Sample Expressions

- When the first `world` of the queue's first path is not `WIN`:
  - Generate new paths after filtering out successors
  - Update the accumulator by adding this first `world`

N-Puzzle Version 5

Sample Expressions

- When the first `world` of the queue's first path is not `WIN`:
  - Generate new paths after filtering out successors
  - Update the accumulator by adding this first `world`
- ```
  (define FS-QLOW3-VAL
    (local
      [(define first-path (qfirst QLOW3))
       (define first-world (first first-path))
       (define successors
               (filter
                 (λ (w)
                   (not (member w
                                (list (make-world 1 2 3 4 5 0 7 8 6)))))
                 (map (λ (neigh)
                        (swap-empty first-world neigh))
                      (list-ref neighbors
                                (blank-pos first-world)))))
       (define new-paths (map (λ (w) (cons w first-path))
                              successors))
       (define new-q (enqueue new-paths (dequeue QLOW3)))]
     (find-solution-bfs
       new-q
       (cons first-world (list (make-world 1 2 3 4 5 0 7 8 6))))))
  ```
  ```

N-Puzzle Version 5

Sample Expressions

- When the first `world` of the queue's first path is not `WIN`:
  - Generate new paths after filtering out successors
  - Update the accumulator by adding this first `world`
- ```
  (define FS-QLOW3-VAL
    (local
      [(define first-path (qfirst QLOW3))
       (define first-world (first first-path))
       (define successors
               (filter
                 (λ (w)
                   (not (member w
                                (list (make-world 1 2 3 4 5 0 7 8 6)))))
                 (map (λ (neigh)
                        (swap-empty first-world neigh))
                      (list-ref neighbors
                                (blank-pos first-world)))))
       (define new-paths (map (λ (w) (cons w first-path))
                              successors))
       (define new-q (enqueue new-paths (dequeue QLOW3)))]
      (find-solution-bfs
        new-q
        (cons first-world (list (make-world 1 2 3 4 5 0 7 8 6))))))
  ```
- Differences: `(qof (listof world))` `(paths)` & `(listof world)` `(for acc)`
  ```

N-Puzzle Version 5

Signature, Statements, and Function Header

```
;; (qof (listof world)) (listof world) → (listof world)
;; Purpose: Return sequence of moves to WIN
;; How: If the first path's first world is WIN then return
;;      the reverse of the first path. Otherwise, continue the
;;      search with:
;;        1. a new queue obtained by removing the first
;;           path and adding paths constructed using the
;;           unvisited successors of the first path's first
;;           world and the first path.
;;        2. an accumulator obtained by adding the first
;;           path's first world to the given accumulator.
;; Accumulator Invariant:
;;   visited = worlds whose successors are part of the search
(define (find-solution-bfs a-qlow visited)
```

N-Puzzle Version 5

Tests

```
;; Tests using sample computations for find-solution
(check-expect (find-solution-bfs QLOW1 '()) FS-QLOW1-VAL)
(check-expect (find-solution-bfs QLOW2 '()) FS-QLOW2-VAL)
     ...
```

N-Puzzle Version 5

Tests

- ```
  ;; Tests using sample computations for find-solution
  (check-expect (find-solution-bfs QLOW1 '()) FS-QLOW1-VAL)
  (check-expect (find-solution-bfs QLOW2 '()) FS-QLOW2-VAL)
      ...
  ```
- Write out the search tree based on `neighbors`:

```
;; Tests using sample values for find-solution
    ...
(check-expect (find-solution-bfs
                (list (list (make-world 1 2 3 4 5 6 7 0 8)
                            (make-world 1 2 3 4 0 6 7 5 8))
                      (list (make-world 1 0 3 4 2 6 7 5 8)
                            (make-world 1 2 3 4 0 6 7 5 8))
                      (list (make-world 1 2 3 0 4 6 7 5 8)
                            (make-world 1 2 3 4 0 6 7 5 8))
                      (list (make-world 1 2 3 4 6 0 7 5 8)
                            (make-world 1 2 3 4 0 6 7 5 8)))
                (list (make-world 1 2 3 4 0 6 7 5 8)))
              (list (make-world 1 2 3 4 0 6 7 5 8)
                    (make-world 1 2 3 4 5 6 7 0 8)
                    (make-world 1 2 3 4 5 6 7 8 0)))
```
```

N-Puzzle Version 5

Function Body

```
(local [(define first-path (qfirst a-qlow))
        (define first-world (first first-path))]
  (if (equal? first-world WIN)
      (reverse first-path)
      (local
        [(define successors
                 (filter
                   (λ (w)
                     (not (member? w visited)))
                   (map (λ (neigh)
                          (swap-empty first-world neigh))
                        (list-ref
                          neighbors
                          (blank-pos first-world)))))
         (define new-paths (map (λ (w)
                                  (cons w first-path))
                                successors))
         (define new-q (enqueue new-paths
                                (dequeue a-qlow)))]
        (find-solution-bfs new-q (cons first-world visited)))))
```

N-Puzzle Version 5

Termination Argument

- When the first path's first `world` in the given queue is `WIN` `find-solution-bfs` halts

N-Puzzle Version 5

Termination Argument

- When the first path's first `world` in the given queue is `WIN` `find-solution-bfs` halts
- For each recursive call one path is taken one step down the search tree using only unvisited successors to avoid following paths with loops

N-Puzzle Version 5

Termination Argument

- When the first path's first `world` in the given queue is `WIN` `find-solution-bfs` halts
- For each recursive call one path is taken one step down the search tree using only unvisited successors to avoid following paths with loops
- Given that a loopless solution exists starting from any valid `world` eventually a path reaching the search tree's height `k` contains `WIN` and the function terminates.

N-Puzzle Version 5

Complexity

- T_1 is `BFS`'s search tree (w/out acc), T_2 is `DFSA`'s search tree (with acc), and T_3 is `BFS`'s search tree (with acc)

N-Puzzle Version 5

Complexity

- T_1 is `BFS`'s search tree (w/out acc), T_2 is `DFSA`'s search tree (with acc), and T_3 is `BFS`'s search tree (with acc)
- Let `h` be the height of the tallest among T_1, T_2, and T_3 and let T_i be the full search subtree of height `h`

N-Puzzle Version 5

Complexity

- T_1 is `BFS`'s search tree (w/out acc), T_2 is `DFSA`'s search tree (with acc), and T_3 is `BFS`'s search tree (with acc)
- Let `h` be the height of the tallest among T_1, T_2, and T_3 and let T_i be the full search subtree of height `h`
- We define `n` as the number of edges and number of nodes in T_i

N-Puzzle Version 5

Complexity

- T_1 is `BFS`'s search tree (w/out acc), T_2 is `DFSA`'s search tree (with acc), and T_3 is `BFS`'s search tree (with acc)
- Let `h` be the height of the tallest among T_1, T_2, and T_3 and let T_i be the full search subtree of height `h`
- We define `n` as the number of edges and number of nodes in T_i
- In the worst case, all three working implementations must traverse T_i in its entirety

N-Puzzle Version 5

Complexity

- T_1 is `BFS`'s search tree (w/out acc), T_2 is `DFSA`'s search tree (with acc), and T_3 is `BFS`'s search tree (with acc)
- Let `h` be the height of the tallest among T_1, T_2, and T_3 and let T_i be the full search subtree of height `h`
- We define `n` as the number of edges and number of nodes in T_i
- In the worst case, all three working implementations must traverse T_i in its entirety
- Visit each node and each edge exactly once
- This means each algorithm performs $O(n)$ steps to complete the traversal

N-Puzzle Version 5

Complexity

- T_1 is `BFS`'s search tree (w/out acc), T_2 is `DFSA`'s search tree (with acc), and T_3 is `BFS`'s search tree (with acc)
- Let `h` be the height of the tallest among T_1, T_2, and T_3 and let T_i be the full search subtree of height `h`
- We define `n` as the number of edges and number of nodes in T_i
- In the worst case, all three working implementations must traverse T_i in its entirety
- Visit each node and each edge exactly once
- This means each algorithm performs $O(n)$ steps to complete the traversal
- For each of these steps, `BFS` performs $O(3n)$ steps (two `map` and one `enqueue` operations)
- `DFSA` performs $O(2n)$ steps (one `map` and one `filter` operations)
- `BFSA` performs $O(4n)$ steps (two `map`, one `filter`, and one `enqueue` operations)

N-Puzzle Version 5

Complexity

- T_1 is `BFS`'s search tree (w/out acc), T_2 is `DFSA`'s search tree (with acc), and T_3 is `BFS`'s search tree (with acc)
- Let `h` be the height of the tallest among T_1, T_2, and T_3 and let T_i be the full search subtree of height `h`
- We define `n` as the number of edges and number of nodes in T_i
- In the worst case, all three working implementations must traverse T_i in its entirety
- Visit each node and each edge exactly once
- This means each algorithm performs $O(n)$ steps to complete the traversal
- For each of these steps, `BFS` performs $O(3n)$ steps (two `map` and one `enqueue` operations)
- `DFSA` performs $O(2n)$ steps (one `map` and one `filter` operations)
- `BFSA` performs $O(4n)$ steps (two `map`, one `filter`, and one `enqueue` operations)
- All three solutions have the same complexity: $O(n^2)$.

N-Puzzle Version 5

Performance

- Run the following timing experiments five times:

```
BFS:
 1: (time (find-solution-bfs
            (list (list (make-world 1 3 8 5 2 0 4 6 7)))))
 2: (time (find-solution-bfs
            (list (list (make-world 0 2 3 1 5 6 4 7 8)))))
 3: (time (find-solution-bfs
            (list (list (make-world 4 1 3 7 2 6 0 5 8)))))
 4: (time (find-solution-bfs
            (list (list (make-world 1 2 3 0 5 6 4 7 8)))))
DFSA:
 1: (time (find-solution (make-world 1 3 8 5 2 0 4 6 7) '()))
 2: (time (find-solution (make-world 0 2 3 1 5 6 4 7 8) '()))
 3: (time (find-solution (make-world 4 1 3 7 2 6 0 5 8) '()))
 4: (time (find-solution (make-world 1 2 3 0 5 6 4 7 8) '()))
BFSA:
 1: (time (find-solution-bfs
            (list (list (make-world 1 3 8 5 2 0 4 6 7))) '()))
 2: (time (find-solution-bfs
            (list (list (make-world 0 2 3 1 5 6 4 7 8))) '()))
 3: (time (find-solution-bfs
            (list (list (make-world 4 1 3 7 2 6 0 5 8))) '()))
 4: (time (find-solution-bfs
            (list (list (make-world 1 2 3 0 5 6 4 7 8))) '()))
```

N-Puzzle Version 5

Performance

	R1	R2	R3	R4	R5	Avg	Sol Len
BFS1	32438	32281	32938	32078	32422	32481.4	12
BFS2	0	0	0	0	0	0	5
BFS3	15	0	0	0	0	3	7
BFS4	0	0	0	0	0	0	4
DFSA1	352032	351187	349938	350891	351703	351150	26282
• DFSA2	328	328	312	343	328	327.8	1109
DFSA3	0	0	0	0	0	0	7
DFSA4	7704	7953	8047	7172	7140	7603.2	5680
BFSA1	843	984	1031	953	985	959.2	12
BFSA2	15	0	0	15	0	6	5
BFSA3	15	0	0	0	0	3	7
BFSA4	0	0	0	0	0	0	4

N-Puzzle Version 5

Performance

	R1	R2	R3	R4	R5	Avg	Sol Len
BFS1	32438	32281	32938	32078	32422	32481.4	12
BFS2	0	0	0	0	0	0	5
BFS3	15	0	0	0	0	3	7
BFS4	0	0	0	0	0	0	4
DFSA1	352032	351187	349938	350891	351703	351150	26282
DFSA2	328	328	312	343	328	327.8	1109
DFSA3	0	0	0	0	0	0	7
DFSA4	7704	7953	8047	7172	7140	7603.2	5680
BFSA1	843	984	1031	953	985	959.2	12
BFSA2	15	0	0	15	0	6	5
BFSA3	15	0	0	0	0	3	7
BFSA4	0	0	0	0	0	0	4

- BFS finds a shorter solution making it significantly faster
- The subtree explored by depth-first search is significantly taller

N-Puzzle Version 5

Performance

	R1	R2	R3	R4	R5	Avg	Sol Len
BFS1	32438	32281	32938	32078	32422	32481.4	12
BFS2	0	0	0	0	0	0	5
BFS3	15	0	0	0	0	3	7
BFS4	0	0	0	0	0	0	4
DFSA1	352032	351187	349938	350891	351703	351150	26282
DFSA2	328	328	312	343	328	327.8	1109
DFSA3	0	0	0	0	0	0	7
DFSA4	7704	7953	8047	7172	7140	7603.2	5680
BFSA1	843	984	1031	953	985	959.2	12
BFSA2	15	0	0	15	0	6	5
BFSA3	15	0	0	0	0	3	7
BFSA4	0	0	0	0	0	0	4

- BFS finds a shorter solution making it significantly faster
- The subtree explored by depth-first search is significantly taller
- `DFSA 3` explores the search tree to the same depth as BFS
- There are conditions under which depth-first search is as fast or faster than breadth-first search

N-Puzzle Version 5

Performance

	R1	R2	R3	R4	R5	Avg	Sol Len
BFS1	32438	32281	32938	32078	32422	32481.4	12
BFS2	0	0	0	0	0	0	5
BFS3	15	0	0	0	0	3	7
BFS4	0	0	0	0	0	0	4
DFSA1	352032	351187	349938	350891	351703	351150	26282
DFSA2	328	328	312	343	328	327.8	1109
DFSA3	0	0	0	0	0	0	7
DFSA4	7704	7953	8047	7172	7140	7603.2	5680
BFSA1	843	984	1031	953	985	959.2	12
BFSA2	15	0	0	15	0	6	5
BFSA3	15	0	0	0	0	3	7
BFSA4	0	0	0	0	0	0	4

- BFS finds a shorter solution making it significantly faster
- The subtree explored by depth-first search is significantly taller
- `DFSA 3` explores the search tree to the same depth as BFS
- There are conditions under which depth-first search is as fast or faster than breadth-first search
- The cost of an accumulator paid off in our experiments

N-Puzzle Version 5

Performance

	R1	R2	R3	R4	R5	Avg	Sol Len
BFS1	32438	32281	32938	32078	32422	32481.4	12
BFS2	0	0	0	0	0	0	5
BFS3	15	0	0	0	0	3	7
BFS4	0	0	0	0	0	0	4
DFSA1	352032	351187	349938	350891	351703	351150	26282
DFSA2	328	328	312	343	328	327.8	1109
DFSA3	0	0	0	0	0	0	7
DFSA4	7704	7953	8047	7172	7140	7603.2	5680
BFSA1	843	984	1031	953	985	959.2	12
BFSA2	15	0	0	15	0	6	5
BFSA3	15	0	0	0	0	3	7
BFSA4	0	0	0	0	0	0	4

- BFS finds a shorter solution making it significantly faster
- The subtree explored by depth-first search is significantly taller
- `DFSA 3` explores the search tree to the same depth as BFS
- There are conditions under which depth-first search is as fast or faster than breadth-first search
- The cost of an accumulator paid off in our experiments
- Combine these two searching algorithms to get the best of both?

N-Puzzle Version 5

Performance

	R1	R2	R3	R4	R5	Avg	Sol Len
BFS1	32438	32281	32938	32078	32422	32481.4	12
BFS2	0	0	0	0	0	0	5
BFS3	15	0	0	0	0	3	7
BFS4	0	0	0	0	0	0	4
DFSA1	352032	351187	349938	350891	351703	351150	26282
DFSA2	328	328	312	343	328	327.8	1109
DFSA3	0	0	0	0	0	0	7
DFSA4	7704	7953	8047	7172	7140	7603.2	5680
BFSA1	843	984	1031	953	985	959.2	12
BFSA2	15	0	0	15	0	6	5
BFSA3	15	0	0	0	0	3	7
BFSA4	0	0	0	0	0	0	4

- BFS finds a shorter solution making it significantly faster
- The subtree explored by depth-first search is significantly taller
- `DFSA 3` explores the search tree to the same depth as BFS
- There are conditions under which depth-first search is as fast or faster than breadth-first search
- The cost of an accumulator paid off in our experiments
- Combine these two searching algorithms to get the best of both?
- The goal is to find a short solution like breadth-first search by exploring fewer paths in the search tree like depth-first search

N-Puzzle Version 5

HOMEWORK

- Problems: 3, 4

Iteration

- Recursive functions are a form of iteration
- Repeatedly perform the same evaluation using different values until a condition is met

Iteration

- Recursive functions are a form of iteration
- Repeatedly perform the same evaluation using different values until a condition is met
- A recursive function may contain one or more *delayed operations*
- A delayed operation is a function application that cannot be evaluated until the evaluation of a different function call is finished

Iteration

- Recursive functions are a form of iteration
- Repeatedly perform the same evaluation using different values until a condition is met
- A recursive function may contain one or more *delayed operations*
- A delayed operation is a function application that cannot be evaluated until the evaluation of a different function call is finished
- The delayed operation and the values it needs must be remembered to complete the computation in the future of the computation
- Allocating memory to remember needed values and returning to finish a delayed operation take time

Iteration

- Recursive functions are a form of iteration
- Repeatedly perform the same evaluation using different values until a condition is met
- A recursive function may contain one or more *delayed operations*
- A delayed operation is a function application that cannot be evaluated until the evaluation of a different function call is finished
- The delayed operation and the values it needs must be remembered to complete the computation in the future of the computation
- Allocating memory to remember needed values and returning to finish a delayed operation take time
- Accumulators may be used to make program evaluation more efficient by eliminating delayed operations
- Use one or more accumulators to remember a partially computed value that is used to finish the computation.

Iteration

```
;; natnum → natnum
;; Purpose: Compute the factorial of the given natural number
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (sub1 n)))))
```

Iteration

- ```
  ;; natnum → natnum
  ;; Purpose: Compute the factorial of the given natural number
  (define (fact n)
    (if (= n 0)
        1
        (* n (fact (sub1 n)))))
  ```
- ```
  (fact 6) = (* 6 (fact 5))
           = (* 6 (* 5 (fact 4)))
           = (* 6 (* 5 (* 4 (fact 3))))
           = (* 6 (* 5 (* 4 (* 3 (fact 2)))))
           = (* 6 (* 5 (* 4 (* 3 (* 2 (fact 1))))))
           = (* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (fact 0)))))))
           = (* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1))))))
           = (* 6 (* 5 (* 4 (* 3 (* 2 1)))))
           = (* 6 (* 5 (* 4 (* 3 2))))
           = (* 6 (* 5 (* 4 6)))
           = (* 6 (* 5 24))
           = (* 6 120)
           = 720
  ```
- The "bump" in the trace indicates that more memory is required at each step to store information for delayed operations until `n` is 0

Iteration

- Partial product stored in an accumulator to eliminate the delayed operation

Iteration

- Partial product stored in an accumulator to eliminate the delayed operation
- ```
  ;; natnum → natnum  Purpose: Compute n!
  (define (fact2 n)
   (local [;; natnum natnum → natnum Purpose: Compute n!
           ;; Acc invariant: accum=the product of nat nums in [k+1..n
           (define (fact-accum n accum)
             (if (= n 0) accum
                 (fact-accum (sub1 n) (* n accum))))]
     (fact-accum n 1)))
  ```
- Final action of a function is a function call we say that it is a *tail call*
- If a recursive call we say that the function is *tail recursive*
  ```

Iteration

- Partial product stored in an accumulator to eliminate the delayed operation
- ```
  ;; natnum → natnum  Purpose: Compute n!
  (define (fact2 n)
   (local [;; natnum natnum → natnum Purpose: Compute n!
           ;; Acc invariant: accum=the product of nat nums in [k+1..n
           (define (fact-accum n accum)
             (if (= n 0) accum
                 (fact-accum (sub1 n) (* n accum))))]
     (fact-accum n 1)))
  ```
- Final action of a function is a function call we say that it is a *tail call*
- If a recursive call we say that the function is *tail recursive*
- ```
  (fact2 6) = (fact-accum 6   1)   Trace has no bump
            = (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
  ```

Iteration

- Partial product stored in an accumulator to eliminate the delayed operation
- ```
  ;; natnum → natnum  Purpose: Compute n!
  (define (fact2 n)
   (local [;; natnum natnum → natnum Purpose: Compute n!
           ;; Acc invariant: accum=the product of nat nums in [k+1..n
           (define (fact-accum n accum)
             (if (= n 0) accum
                 (fact-accum (sub1 n) (* n accum))))]
     (fact-accum n 1)))
  ```
- Final action of a function is a function call we say that it is a *tail call*
- If a recursive call we say that the function is *tail recursive*
- ```
  (fact2 6) = (fact-accum 6   1)   Trace has no bump
            = (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
  ```
	Run 1	Run 2	Run 3	Run 4	Run 5	Average
fact	516	484	515	485	485	497
fact2	609	610	593	594	594	600
- `fact` is slightly faster
- Does this mean that tail-recursion is not an efficient form of iteration?

Iteration

Summing a List of Numbers: Problem Analysis

- ```
  ;; (listof number) → number
  ;; Purpose: Add the numbers in the given lon
  (define (sum-lon a-lon)
    (if (empty? a-lon)
        0
        (+ (first a-lon)
           (sum-lon (rest a-lon)))))
  ```
  ```

Iteration

Summing a List of Numbers: Problem Analysis

- ```
  ;; (listof number) → number
  ;; Purpose: Add the numbers in the given lon
  (define (sum-lon a-lon)
    (if (empty? a-lon)
        0
        (+ (first a-lon)
           (sum-lon (rest a-lon)))))
  ```
- The result of the recursive call is input to + making it a delayed operation
- Introduce an accumulator to eliminate the delayed operation.
  ```

Iteration

Summing a List of Numbers: Problem Analysis

- ```
  ;; (listof number) → number
  ;; Purpose: Add the numbers in the given lon
  (define (sum-lon a-lon)
    (if (empty? a-lon)
        0
        (+ (first a-lon)
           (sum-lon (rest a-lon)))))
  ```
- The result of the recursive call is input to + making it a delayed operation
- Introduce an accumulator to eliminate the delayed operation.
- `accum = the sum of list elements so far`
- Initially `accum` must be 0
- When the given list of numbers is empty the accumulator contains the completed sum and it is returned as the value of the function.
- When the list of numbers is not empty a recursive call is made with the rest of the list and the first list element is folded into the accumulator using +
  ```

Iteration

Summing a List of Numbers: Refactoring sum-lon

```
;; (listof number) → number
;; Purpose: Add the numbers in the given lon
(define (sum-lon2 a-lon)
  (sum-lon-accum a-lon 0))

;; Sample expressions for sum-lon2
(define SUM2-ELON (sum-lon-accum ELON 0))
(define SUM2-LON1 (sum-lon-accum LON1 0))
(define SUM2-LON2 (sum-lon-accum LON2 0))

;; Tests using sample computations for sum-lon2
(check-expect (sum-lon2 ELON) SUM2-ELON)
(check-expect (sum-lon2 LON1) SUM2-LON1)
(check-expect (sum-lon2 LON2) SUM2-LON2)

;; Tests using sample values for sum-lon2
(check-expect (sum-lon2 '(1 2 3 4)) 10)
(check-expect (sum-lon2 '(7 31 8)) 46)
```

Iteration

sum-lon-accum's Sample Expressions and Differences

- The sample expressions we use the same sample `lons` defined in Figure 39 in the textbook
- The answer is always the accumulator when the given `lon` is `ELON`
- For example, means the answer is 0 and processing `ELON` after traversing '(10 6 4) means the answer is 20. Sample expressions for these are written as follows:

```
;; Sample expressions for sum-lon-accum
(define SUMACCUM-ELON1 0) processing ELON  and 0

(define SUMACCUM-ELON2 20) after traversing '(10 6 4)
```

Iteration

sum-lon-accum's Sample Expressions and Differences

- The sample expressions we use the same sample `lons` defined in Figure 39 in the textbook
- The answer is always the accumulator when the given `lon` is `ELON`
- For example, means the answer is 0 and processing `ELON` after traversing '(10 6 4) means the answer is 20. Sample expressions for these are written as follows:

```
;; Sample expressions for sum-lon-accum
(define SUMACCUM-ELON1 0) processing ELON  and 0

(define SUMACCUM-ELON2 20) after traversing '(10 6 4)
```

-
```
(define SUMACCUM-LON1 (sum-lon-accum
                       (rest LON1)
                       (+ (first LON1) 0)))

(define SUMACCUM-RLON2 (sum-lon-accum
                        (rest (rest LON2))
                        (+ (second LON2) (first LON2))))
```

Iteration

sum-lon-accum's Sample Expressions and Differences

- The sample expressions we use the same sample `lons` defined in Figure 39 in the textbook
- The answer is always the accumulator when the given `lon` is `ELON`
- For example, means the answer is 0 and processing `ELON` after traversing '(10 6 4) means the answer is 20. Sample expressions for these are written as follows:

```
;; Sample expressions for sum-lon-accum
(define SUMACCUM-ELON1 0) processing ELON  and 0

(define SUMACCUM-ELON2 20) after traversing '(10 6 4)
```

- ```
  (define SUMACCUM-LON1 (sum-lon-accum
                         (rest LON1)
                         (+ (first LON1) 0)))

  (define SUMACCUM-RLON2 (sum-lon-accum
                          (rest (rest LON2))
                          (+ (second LON2) (first LON2))))
  ```
- Differences: a list of numbers and a number
  ```

Iteration

Summing a List of Numbers:

```
;; (listof number) number → number
;; Purpose: Add the numbers in the given lon to the
;;          given accumulator
;; Accumulator invariant
;;  accum = sum of list elements so far
(define (sum-lon-accum a-lon accum)
```

Iteration

Summing a List of Numbers:

```
;; (listof number) number → number
;; Purpose: Add the numbers in the given lon to the
;;          given accumulator
;; Accumulator invariant
;;  accum = sum of list elements so far
(define (sum-lon-accum a-lon accum)
```

```
;; Tests using sample computations for sum-lon-accum
(check-expect (sum-lon-accum ELON 0)  SUMACCUM-ELON1)
(check-expect (sum-lon-accum ELON 20) SUMACCUM-ELON2)
(check-expect (sum-lon-accum LON1 0)  SUMACCUM-LON1)
(check-expect (sum-lon-accum (rest LON2) (first LON2))
              SUMACCUM-RLON2)

;; Tests using sample values for sum-lon-accum
(check-expect (sum-lon-accum '(1 2 3 4)  0) 10)
(check-expect (sum-lon-accum '(7 31 8)   0) 46)
```

Iteration

Summing a List of Numbers:

```
;; (listof number) number → number
;; Purpose: Add the numbers in the given lon to the
;;          given accumulator
;; Accumulator invariant
;;  accum = sum of list elements so far
(define (sum-lon-accum a-lon accum)
  (if (empty? a-lon)
      accum
      (sum-lon-accum
        (rest a-lon)
        (+ (first a-lon) accum))))
```

```
;; Tests using sample computations for sum-lon-accum
(check-expect (sum-lon-accum ELON 0)  SUMACCUM-ELON1)
(check-expect (sum-lon-accum ELON 20) SUMACCUM-ELON2)
(check-expect (sum-lon-accum LON1 0)  SUMACCUM-LON1)
(check-expect (sum-lon-accum (rest LON2) (first LON2))
              SUMACCUM-RLON2)

;; Tests using sample values for sum-lon-accum
(check-expect (sum-lon-accum '(1 2 3 4)  0) 10)
(check-expect (sum-lon-accum '(7 31 8)   0) 46)
```

Iteration

Performance

- ```
  (define L (build-list 1000000 (λ (i) (random 100000))))

  (time (sum-lon L))
  (time (sum-lon2 L))
  ```
  ```

Iteration

Performance

- ```
  (define L (build-list 1000000 (λ (i) (random 100000))))

  (time (sum-lon L))
  (time (sum-lon2 L))
  ```

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4

- The tail-recursive version is always faster
  ```

Iteration

Performance

- ```
  (define L (build-list 1000000 (λ (i) (random 100000))))

  (time (sum-lon L)))
  (time (sum-lon2 L)))
  ```

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4

- The tail-recursive version is always faster
- To compare the averages we may use their *relative difference*
- The relative difference is a means to compare two numbers taking into account their sizes. It is a unitless ratio that uses one of the measurements as a reference value and informs us by how much the measurements are proportionally different:

$$\frac{x - x_{reference}}{x_{reference}}$$

- If the relative difference is positive then the reference performs better
  ```

Iteration

Performance

- ```
  (define L (build-list 1000000 (λ (i) (random 100000))))

  (time (sum-lon L))
  (time (sum-lon2 L))
  ```

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4

- The tail-recursive version is always faster
- To compare the averages we may use their *relative difference*
- The relative difference is a means to compare two numbers taking into account their sizes. It is a unitless ratio that uses one of the measurements as a reference value and informs us by how much the measurements are proportionally different:

$$\frac{x - x_{reference}}{x_{reference}}$$

- If the relative difference is positive then the reference performs better
- Using `sum-lon2`'s average as the reference measurement we obtain a relative difference that is approximately 0.11
- This informs us that the tail-recursive version is about 11% faster
  ```

Iteration

Reversing a List

- ```
  (define ELOX '())
  (define LOX1 '(d c b a))
  (define LOX2 '(99 #true "Hi!"))

  ;; (listof X) → (listof X)
  ;; Purpose: Reverse the given list
  (define (rev-lox a-lox)
    (if (empty? a-lox)
        '()
        (append (rev-lox (rest a-lox)) (list (first a-lox)))))
  ```
- Candidate for delayed operation elimination
  ```

Iteration

Reversing a List: Problem Analysis

- Add an accumulator that stores the reverse of the list traversed so far
- Initially, the accumulator is `'()`

Iteration

Reversing a List: Problem Analysis

- Add an accumulator that stores the reverse of the list traversed so far
- Initially, the accumulator is `'()`
- If the list is `'()` the accumulator is returned
- Otherwise, recursive call with the rest of the given list and an accumulator that has the first element of the given list

Iteration

Reversing a List: Problem Analysis

- Add an accumulator that stores the reverse of the list traversed so far
- Initially, the accumulator is `'()`
- If the list is `'()` the accumulator is returned
- Otherwise, recursive call with the rest of the given list and an accumulator that has the first element of the given list

a-lox	accum
(d c b a)	'()
(c b a)	(d)
(b a)	(c d)
(a)	(b c d)
()	(a b c d)

Iteration

Reversing a List: Problem Analysis

- Add an accumulator that stores the reverse of the list traversed so far
- Initially, the accumulator is `'()`
- If the list is `'()` the accumulator is returned
- Otherwise, recursive call with the rest of the given list and an accumulator that has the first element of the given list
-

a-lox	accum
(d c b a)	'()
(c b a)	(d)
(b a)	(c d)
(a)	(b c d)
()	(a b c d)

- Invariant property: accum = the list of traversed elements reversed

Iteration

Reversing a List: Problem Analysis

- Add an accumulator that stores the reverse of the list traversed so far
- Initially, the accumulator is `'()`
- If the list is `'()` the accumulator is returned
- Otherwise, recursive call with the rest of the given list and an accumulator that has the first element of the given list
-

a-lox	accum
(d c b a)	'()
(c b a)	(d)
(b a)	(c d)
(a)	(b c d)
()	(a b c d)

- Invariant property: accum = the list of traversed elements reversed
-

```
;; (listof X) → (listof X)
;; Purpose: Reverse the given list
;; INV: accum = the list of traversed elements reversed
(define (rev-lox2 a-lox) (rev-lox-accum a-lox '()))
;; Tests using sample computations for rev-lox2
(check-expect (rev-lox2 ELOX) REV2-ELOX)
     ...
;; Tests using sample values for rev-lox2
(check-expect (rev-lox2 '(1 2 3 4 5)) '(5 4 3 2 1))
(check-expect (rev-lox2 '(red white blue)) '(blue white red))
```

Iteration

Sample Expressions and Differences for `rev-lox-accum`

- When the given list is empty the accumulator is returned
- ```
  ;; Sample expressions for rev-lox-accum
  (define SUMACCUM-ELOX1 '())  acc must be empty
  (define SUMACCUM-ELOX2 '(blue white red)) acc is the list reversed
  ```
  ```

Iteration

Sample Expressions and Differences for `rev-lox-accum`

- When the given list is empty the accumulator is returned
- ```
  ;; Sample expressions for rev-lox-accum
  (define SUMACCUM-ELOX1 '())  acc must be empty
  (define SUMACCUM-ELOX2 '(blue white red)) acc is the list reversed
  ```
- When the given list is not empty a recursive call is made with the rest of the list and the first element of the given list is added to the accumulator:

```
(define SUMACCUM-LOX1 (rev-lox-accum
                       (rest LOX1)
                       (cons (first LOX1) '())))
(define SUMACCUM-RLOX2 (rev-lox-accum
                        (rest (rest LOX2))
                        (cons (first (rest LOX2))
                              (list (first LOX2)))))
```

- Observe that there are no delayed operations
```

Iteration

Sample Expressions and Differences for `rev-lox-accum`

- When the given list is empty the accumulator is returned
-
```
;; Sample expressions for rev-lox-accum
(define SUMACCUM-ELOX1 '())  acc must be empty
(define SUMACCUM-ELOX2 '(blue white red)) acc is the list reversed
```
- When the given list is not empty a recursive call is made with the rest of the list and the first element of the given list is added to the accumulator:

```
(define SUMACCUM-LOX1 (rev-lox-accum
                       (rest LOX1)
                       (cons (first LOX1) '())))
(define SUMACCUM-RLOX2 (rev-lox-accum
                        (rest (rest LOX2))
                        (cons (first (rest LOX2))
                              (list (first LOX2)))))
```

- Observe that there are no delayed operations
- Differences: `(listof X)` to reverse and the given `(listof X)` for the accumulator

Iteration

Reversing a List

```
;; (listof X) → (listof X)
;; Purpose: Reverse the first given list
;; Accumulator invariant:
;;  accum = the list of traversed elements reversed
(define (rev-lox-accum a-lox accum)
```

Iteration

Reversing a List

```
;; (listof X) → (listof X)
;; Purpose: Reverse the first given list
;; Accumulator invariant:
;;  accum = the list of traversed elements reversed
(define (rev-lox-accum a-lox accum)
```

```
;; Tests using sample computations for rev-lox-accum
(check-expect (rev-lox-accum ELOX '())
              SUMACCUM-ELOX1)
(check-expect (rev-lox-accum ELOX '(blue white red))
              SUMACCUM-ELOX2)
(check-expect (rev-lox-accum LOX1 '())
              SUMACCUM-LOX1)
(check-expect (rev-lox-accum (rest LOX2)
                             (list (first LOX2)))
              SUMACCUM-RLOX2)

;; Tests using sample values for rev-lox-accum
(check-expect (rev-lox-accum '(1 2 3 4) '())
              '(4 3 2 1))
(check-expect (rev-lox-accum '(#false #true)  '())
              '(#true #false))
```

Iteration

Reversing a List

```
;; (listof X) → (listof X)
;; Purpose: Reverse the first given list
;; Accumulator invariant:
;;  accum = the list of traversed elements reversed
(define (rev-lox-accum a-lox accum)
  (if (empty? a-lox)
      accum
      (rev-lox-accum (rest a-lox) (cons (first a-lox) accum))))

;; Tests using sample computations for rev-lox-accum
(check-expect (rev-lox-accum ELOX '())
              SUMACCUM-ELOX1)
(check-expect (rev-lox-accum ELOX '(blue white red))
              SUMACCUM-ELOX2)
(check-expect (rev-lox-accum LOX1 '())
              SUMACCUM-LOX1)
(check-expect (rev-lox-accum (rest LOX2)
                             (list (first LOX2)))
              SUMACCUM-RLOX2)

;; Tests using sample values for rev-lox-accum
(check-expect (rev-lox-accum '(1 2 3 4) '())
              '(4 3 2 1))
(check-expect (rev-lox-accum '(#false #true)  '())
              '(#true #false))
```

Iteration

Reversing a List: Performance

- ```
  (define L (build-list 50000 (λ (i) (random 100000))))

  (time (rev-lox  L)))
  (time (rev-lox2 L)))
  ```
  ```

Iteration

Reversing a List: Performance

- ```
  (define L (build-list 50000 (λ (i) (random 100000))))

  (time (rev-lox  L)))
  (time (rev-lox2 L)))
  ```

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
rev-lox	12172	12313	12187	12562	11829	12212.6
rev-lox2	15	15	15	0	0	9

- An accumulator has a tremendous impact on performance
  ```

Iteration

Reversing a List: Performance

- ```
  (define L (build-list 50000 (λ (i) (random 100000))))

  (time (rev-lox  L)))
  (time (rev-lox2 L)))
  ```
-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
rev-lox	12172	12313	12187	12562	11829	12212.6
rev-lox2	15	15	15	0	0	9

- An accumulator has a tremendous impact on performance
- Such a dramatic impact on performance suggests that the use of an accumulator has done much more than simply eliminate the delayed operations
- `rev-lox` is called `n + 1` times for each call except the last `append` is called which we know is $O(n)$
- $O(n * n) = O(n^2)$
  ```

Iteration

Reversing a List: Performance

- ```
  (define L (build-list 50000 (λ (i) (random 100000))))

  (time (rev-lox  L)))
  (time (rev-lox2 L)))
  ```
	Run 1	Run 2	Run 3	Run 4	Run 5	Average
rev-lox	12172	12313	12187	12562	11829	12212.6
rev-lox2	15	15	15	0	0	9
- An accumulator has a tremendous impact on performance
- Such a dramatic impact on performance suggests that the use of an accumulator has done much more than simply eliminate the delayed operations
- `rev-lox` is called `n + 1` times for each call except the last `append` is called which we know is $O(n)$
- $O(n * n) = O(n^2)$
- `rev-lox-accum` is also called `n + 1`
- For each call except the last a `rest`, a `first`, and a `cons` are executed
- $O(3 * n) = O(n)$
  ```

HOMEWORK

- Problems: 3-6

List-Folding from the Right

- Sometimes processing list elements in the reversed order followed by structural recursion yields the wrong result
- Sometimes it is necessary to process list elements in the same order as structural recursion: right to left
- Other times an operation is associative and it may be natural to follow a design that processes list elements from right to left

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

- Structural Recursion

```
(define ELOSTR '())
(define LOSTR1 '("a" "b" "c"))
(define LOSTR2 '("Program" "design" "is" "awesome!"))

;; (listof string) → (listof natnum)
;; Purpose: Return the lengths of the strings in the given
;;          list of strings
(define (lengths-lostr a-lostr)
  (if (empty? a-lostr)
      '()
      (cons (string-length (first a-lostr))
            (lengths-lostr (rest a-lostr)))))
```

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

- Structural Recursion

```
(define ELOSTR '())
(define LOSTR1 '("a" "b" "c"))
(define LOSTR2 '("Program" "design" "is" "awesome!"))

;; (listof string) → (listof natnum)
;; Purpose: Return the lengths of the strings in the given
;;          list of strings
(define (lengths-lostr a-lostr)
  (if (empty? a-lostr)
      '()
      (cons (string-length (first a-lostr))
            (lengths-lostr (rest a-lostr)))))
```

- An accumulator to store the string lengths of the traversed part

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

- Structural Recursion

```
(define ELOSTR '())
(define LOSTR1 '("a" "b" "c"))
(define LOSTR2 '("Program" "design" "is" "awesome!"))

;; (listof string) → (listof natnum)
;; Purpose: Return the lengths of the strings in the given
;;          list of strings
(define (lengths-lostr a-lostr)
  (if (empty? a-lostr)
      '()
      (cons (string-length (first a-lostr))
            (lengths-lostr (rest a-lostr)))))
```

- An accumulator to store the string lengths of the traversed part
- Initially, the accumulator must be `'()` given that none of the list of strings has been traversed.

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

- Structural Recursion

```
(define ELOSTR '())
(define LOSTR1 '("a" "b" "c"))
(define LOSTR2 '("Program" "design" "is" "awesome!"))

;; (listof string) → (listof natnum)
;; Purpose: Return the lengths of the strings in the given
;;          list of strings
(define (lengths-lostr a-lostr)
  (if (empty? a-lostr)
      '()
      (cons (string-length (first a-lostr))
            (lengths-lostr (rest a-lostr)))))
```

- An accumulator to store the string lengths of the traversed part
- Initially, the accumulator must be `'()` given that none of the list of strings has been traversed.
- If we give the auxiliary function the given list of strings as input every new length must be added to the end of the accumulator
- append must be used $\Rightarrow$ solution is $O(n^2)$

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

- Structural Recursion

```
(define ELOSTR '())
(define LOSTR1 '("a" "b" "c"))
(define LOSTR2 '("Program" "design" "is" "awesome!"))

;; (listof string) → (listof natnum)
;; Purpose: Return the lengths of the strings in the given
;;          list of strings
(define (lengths-lostr a-lostr)
  (if (empty? a-lostr)
      '()
      (cons (string-length (first a-lostr))
            (lengths-lostr (rest a-lostr)))))
```

- An accumulator to store the string lengths of the traversed part
- Initially, the accumulator must be `'()` given that none of the list of strings has been traversed.
- If we give the auxiliary function the given list of strings as input every new length must be added to the end of the accumulator
- append must be used $\Rightarrow$ solution is $O(n^2)$
- Can we avoid using append?

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

- Structural Recursion

```
(define ELOSTR '())
(define LOSTR1 '("a" "b" "c"))
(define LOSTR2 '("Program" "design" "is" "awesome!"))

;; (listof string) → (listof natnum)
;; Purpose: Return the lengths of the strings in the given
;;          list of strings
(define (lengths-lostr a-lostr)
  (if (empty? a-lostr)
      '()
      (cons (string-length (first a-lostr))
            (lengths-lostr (rest a-lostr)))))
```

- An accumulator to store the string lengths of the traversed part
- Initially, the accumulator must be `'()` given that none of the list of strings has been traversed.
- If we give the auxiliary function the given list of strings as input every new length must be added to the end of the accumulator
- append must be used $\Rightarrow$ solution is $O(n^2)$
- Can we avoid using append?
- Provide the given list of strings reversed

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

```
  (lengths-lostr2 '("Oct" "31" "2352"))
= (lengths-lostr-accum '("2352" "31" "Oct") '())
= (lengths-lostr-accum '("31" "Oct") '(4))
= (lengths-lostr-accum '("Oct") '(2 4))
= (lengths-lostr-accum '() '(3 2 4))
= '(3 2 4)
```

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

- ```
    (lengths-lostr2 '("Oct" "31" "2352"))
  = (lengths-lostr-accum '("2352" "31" "Oct") '())
  = (lengths-lostr-accum '("31" "Oct") '(4))
  = (lengths-lostr-accum '("Oct") '(2 4))
  = (lengths-lostr-accum '() '(3 2 4))
  = '(3 2 4)
  ```
- What is the accumulator invariant?
  ```

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

- ```
  (lengths-lostr2 '("Oct" "31" "2352"))
= (lengths-lostr-accum '("2352" "31" "Oct") '())
= (lengths-lostr-accum '("31" "Oct") '(4))
= (lengths-lostr-accum '("Oct") '(2 4))
= (lengths-lostr-accum '() '(3 2 4))
= '(3 2 4)
  ```
- What is the accumulator invariant?
- `accum = traversed list string lengths in reversed order`
  ```

List-Folding from the Right

Computing String Lengths From a List of Strings: Problem Analysis

```
  (lengths-lostr2 '("Oct" "31" "2352"))
= (lengths-lostr-accum '("2352" "31" "Oct") '())
= (lengths-lostr-accum '("31" "Oct") '(4))
= (lengths-lostr-accum '("Oct") '(2 4))
= (lengths-lostr-accum '() '(3 2 4))
= '(3 2 4)
```

- What is the accumulator invariant?
- accum = traversed list string lengths in reversed order

```
;; (listof string) → (listof natnum)
;; Purpose: Return the lengths of the strings in the given
;;          list of strings
(define (lengths-lostr2 a-lostr)
  (lengths-lostr-accum (reverse a-lostr) '()))

;; Sample expressions for lengths-lostr2
(define LENS2-ELOSTR (lengths-lostr-accum (reverse ELOSTR) '()))
(define LENS2-LOSTR1 (lengths-lostr-accum (reverse LOSTR1) '()))
(define LENS2-LOSTR2 (lengths-lostr-accum (reverse LOSTR2) '()))

;; Tests using sample computations for lengths-lostr2
(check-expect (lengths-lostr2 ELOSTR) LENS2-ELOSTR)
(check-expect (lengths-lostr2 LOSTR1) LENS2-LOSTR1)
(check-expect (lengths-lostr2 LOSTR2) LENS2-LOSTR2)
```

List-Folding from the Right

Sample Expressions and Differences for `lengths-lostr-accum`

- Whenever the given list is empty the answer is the accumulator

```
;; Sample expressions for lengths-lostr-accum
(define LACC2-ELOSTR1 '()) process '()
(define LACC2-ELOSTR2 '("1" "1" "1")) process '("c" "b" "a")
```

List-Folding from the Right

Sample Expressions and Differences for `lengths-lostr-accum`

- Whenever the given list is empty the answer is the accumulator

```
;; Sample expressions for lengths-lostr-accum
(define LACC2-ELOSTR1 '()) process '()
(define LACC2-ELOSTR2 '("1" "1" "1")) process '("c" "b" "a")
```

- When the given list is not empty a recursive tail-call is needed with the rest of the given list and the length of the first string added to the front of the given accumulator

```
(define LACC2-LOSTR1 (lengths-lostr-accum
                      (rest LOSTR1)
                       (cons (string-length (first LOSTR1))
                             '())))
(define LACC2-LOSTR2 (lengths-lostr-accum
                      (rest (rest LOSTR2))
                      (cons (string-length (second LOSTR2))
                            (list (first LOSTR2)))))
```

List-Folding from the Right

Sample Expressions and Differences for `lengths-lostr-accum`

- Whenever the given list is empty the answer is the accumulator

```
;; Sample expressions for lengths-lostr-accum
(define LACC2-ELOSTR1 '()) process '()
(define LACC2-ELOSTR2 '("1" "1" "1")) process '("c" "b" "a")
```

- When the given list is not empty a recursive tail-call is needed with the rest of the given list and the length of the first string added to the front of the given accumulator

```
(define LACC2-LOSTR1 (lengths-lostr-accum
                      (rest LOSTR1)
                       (cons (string-length (first LOSTR1))
                             '())))
(define LACC2-LOSTR2 (lengths-lostr-accum
                      (rest (rest LOSTR2))
                      (cons (string-length (second LOSTR2))
                            (list (first LOSTR2)))))
```

- Differences: `(listof string)` and the given `(listof natnum)` for the accumulator

List-Folding from the Right

Computing String Lengths From a List of Strings: Code

- ```
  ;; (listof string) (listof natnum) → (listof natnum)
  ;; Purpose: ...
  ;;  accum = traversed list string lengths in reversed order
  (define (lengths-lostr-accum a-lostr accum)
  ```
  ```

List-Folding from the Right

Computing String Lengths From a List of Strings: Code

- ```
  ;; (listof string) (listof natnum) → (listof natnum)
  ;; Purpose: ...
  ;;  accum = traversed list string lengths in reversed order
  (define (lengths-lostr-accum a-lostr accum)
  ```

- ```
  ;; Tests using sample computations for lengths-lostr-accum
  (check-expect (lengths-lostr-accum ELOSTR '()) LACC2-ELOSTR1)
  (check-expect (lengths-lostr-accum ELOSTR '("1" "1" "1")) LACC2-ELO
  (check-expect (lengths-lostr-accum LOSTR1 '()) LACC2-LOSTR1)
  (check-expect (lengths-lostr-accum (rest LOSTR2)
                                     (list (first LOSTR2)))
                LACC2-LOSTR2)
  ;; Tests using sample values for lengths-lostr-accum
  (check-expect (lengths-lostr-accum '("invariants." "love" "I") '())
                '(1 4 11))
  (check-expect (lengths-lostr-accum '("Rock!" "Accumulators") '())
                '(12 5))
  ```

List-Folding from the Right

Computing String Lengths From a List of Strings: Code

```
;; (listof string) (listof natnum) → (listof natnum)
;; Purpose: ...
;;  accum = traversed list string lengths in reversed order
(define (lengths-lostr-accum a-lostr accum)
  (if (empty? a-lostr)
      accum
      (lengths-lostr-accum
        (rest a-lostr)
        (cons (string-length (first a-lostr)) accum))))
```

```
;; Tests using sample computations for lengths-lostr-accum
(check-expect (lengths-lostr-accum ELOSTR '()) LACC2-ELOSTR1)
(check-expect (lengths-lostr-accum ELOSTR '("1" "1" "1")) LACC2-ELO
(check-expect (lengths-lostr-accum LOSTR1 '()) LACC2-LOSTR1)
(check-expect (lengths-lostr-accum (rest LOSTR2)
                                   (list (first LOSTR2)))
              LACC2-LOSTR2)
;; Tests using sample values for lengths-lostr-accum
(check-expect (lengths-lostr-accum '("invariants." "love" "I") '())
              '(1 4 11))
(check-expect (lengths-lostr-accum '("Rock!" "Accumulators") '())
              '(12 5))
```

List-Folding from the Right

Computing String Lengths From a List of Strings: Performance

- ```
  (define L (build-list
              50000 (λ (i) (generate-password (+ 10 (random 20))))))
  (time (lengths-lostr  L)))  (time (lengths-lostr2 L)))
  ```
  ```

List-Folding from the Right

Computing String Lengths From a List of Strings: Performance

- ```
  (define L (build-list
              50000 (λ (i) (generate-password (+ 10 (random 20))))))
  (time (lengths-lostr  L)))  (time (lengths-lostr2 L)))
  ```

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
lengths-lostr	78	78	109	62	47	74.8
lengths-lostr2	78	63	47	78	78	68.8

- At a first glance there is no clear winner here
  ```

List-Folding from the Right

Computing String Lengths From a List of Strings: Performance

- `(define L (build-list 50000 (λ (i) (generate-password (+ 10 (random 20))))))`
  `(time (lengths-lostr L)) (time (lengths-lostr2 L))`

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
lengths-lostr	78	78	109	62	47	74.8
lengths-lostr2	78	63	47	78	78	68.8

- At a first glance there is no clear winner here
- If we use `lengths-lostr2`'s average CPU time as the reference measurement the relative difference is approximately 0.087: about 8% faster

List-Folding from the Right

Computing String Lengths From a List of Strings: Performance

- ```
  (define L (build-list
              50000 (λ (i) (generate-password (+ 10 (random 20))))))
  (time (lengths-lostr  L)))  (time (lengths-lostr2 L)))
  ```
	Run 1	Run 2	Run 3	Run 4	Run 5	Average
lengths-lostr	78	78	109	62	47	74.8
lengths-lostr2	78	63	47	78	78	68.8
- At a first glance there is no clear winner here
- If we use `lengths-lostr2`'s average CPU time as the reference measurement the relative difference is approximately 0.087: about 8% faster
- Be careful of *outliers*
- An outlier measurement is one that is significantly larger or smaller than the rest of the measurements
- May have a disproportional impact on the average
- Statisticians suggest removing outliers
  ```

List-Folding from the Right

Computing String Lengths From a List of Strings: Performance

- ```
  (define L (build-list
              50000 (λ (i) (generate-password (+ 10 (random 20))))))
  (time (lengths-lostr  L)))  (time (lengths-lostr2 L)))
  ```
-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
lengths-lostr	78	78	109	62	47	74.8
lengths-lostr2	78	63	47	78	78	68.8

- At a first glance there is no clear winner here
- If we use `lengths-lostr2`'s average CPU time as the reference measurement the relative difference is approximately 0.087: about 8% faster
- Be careful of *outliers*
- An outlier measurement is one that is significantly larger or smaller than the rest of the measurements
- May have a disproportional impact on the average
- Statisticians suggest removing outliers
- In the data collected 47 and 109 may be considered outliers
- Remove them: `lengths-lostr` avg is 72.6 and `lengths-lostr2` is 74.25.
- Suggests a very small performance gap between the two implementations: relative difference is -0.02
- `lengths-lostr` is ever so slightly faster on average
- Certainly, reversing the list in `lengths-lostr2` does not have a significant impact on performance.
  ```

List-Folding from the Right

Summing a List of Numbers Revisited

- Summing a list of numbers is an associative operation
- If `RL` = `(reverse L)` then this means that:

```
(sum-lon L) = (sum-lon RL)                    when L = '()

(+ (first L)  (sum-lon (rest L)))
= (+ (first RL) (sum-lon (rest RL)))          when L ≠ '()
```

List-Folding from the Right

Summing a List of Numbers Revisited

- Summing a list of numbers is an associative operation
- If `RL` = `(reverse L)` then this means that:

```
(sum-lon L) = (sum-lon RL)                    when L = '()

(+ (first L)  (sum-lon (rest L)))
= (+ (first RL) (sum-lon (rest RL)))   when L ≠ '()
```

- Refactor `sum-lon2` to pass `(reverse a-lon)`

```
;; (listof number) → number
;; Purpose: Add the numbers in the given list of numbers
(define (sum-lon3 a-lon) (sum-lon-accum (reverse a-lon) 0))

;; Sample expressions for sum-lon3
(define SUM3-ELON (sum-lon-accum (reverse ELON) 0))
(define SUM3-LON1 (sum-lon-accum (reverse LON1) 0))
(define SUM3-LON2 (sum-lon-accum (reverse LON2) 0))
;; Tests using sample computations for sum-lon3
(check-expect (sum-lon3 ELON) SUM3-ELON)
(check-expect (sum-lon3 LON1) SUM3-LON1)
(check-expect (sum-lon3 LON2) SUM3-LON2)
;; Tests using sample values for sum-lon3
(check-expect (sum-lon3 '(1 2 3 4)) 10)
(check-expect (sum-lon3 '(7 31 8)) 46)
```

- No changes required for `sum-lon-accum`

List-Folding from the Right

Computing String Lengths From a List of Strings: Performance

- Compare the performance with previous versions:

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4
sum-lon3	308	312	313	313	328	314.8

- The table suggests that reversing the list has a small impact on performance

List-Folding from the Right

Computing String Lengths From a List of Strings: Performance

- Compare the performance with previous versions:

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4
sum-lon3	308	312	313	313	328	314.8

- The table suggests that reversing the list has a small impact on performance
- Accumulator still has a positive impact on performance but not as accentuated as that obtained from `sum-lon2`
- Using `sum-lon2`'s average running time as the reference measurement against `sum-lon3`'s average running time we observe a relative difference of approximately 0.027
- The price paid to reverse the list is about 3%
- Better to avoid reversing the list whenever possible.

List-Folding from the Right

Computing String Lengths From a List of Strings: Performance

- Compare the performance with previous versions:

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4
sum-lon3	308	312	313	313	328	314.8

- The table suggests that reversing the list has a small impact on performance

List-Folding from the Right

Computing String Lengths From a List of Strings: Performance

- Compare the performance with previous versions:

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4
sum-lon3	308	312	313	313	328	314.8

- The table suggests that reversing the list has a small impact on performance
- Accumulator still has a positive impact on performance but not as accentuated as that obtained from `sum-lon2`
- Using `sum-lon2`'s average running time as the reference measurement against `sum-lon3`'s average running time we observe a relative difference of approximately 0.027
- The price paid to reverse the list is about 3%
- Better to avoid reversing the list whenever possible

List-Folding from the Right

HOMEWORK

- Problems: 7-11

Functional Abstraction

```
(define (sum-lon2 a-lon)
 (local
  [(define
    (sum-lon-accum a-lon accum)
     (if (empty? a-lon)
         accum
         (sum-lon-accum
          (rest a-lon)
          (+ (first a-lon)
             accum))))]
  (sum-lon-accum a-lon 0)))
```

```
(define (rev-lox2 a-lox)
 (local
  [(define
    (rev-lox-accum a-lox accum)
     (if (empty? a-lox)
         accum
         (rev-lox-accum
          (rest a-lox)
          (cons (first a-lox)
                accum))))]
  (rev-lox-accum a-lox '())))
```

- Very similar $\Rightarrow$ good candidates for abstraction

Functional Abstraction

```
(define (sum-lon2 a-lon)
 (local
  [(define
    (sum-lon-accum a-lon accum)
     (if (empty? a-lon)
         accum
         (sum-lon-accum
          (rest a-lon)
          (+ (first a-lon)
             accum))))]
  (sum-lon-accum a-lon 0)))
```

```
(define (rev-lox2 a-lox)
 (local
  [(define
    (rev-lox-accum a-lox accum)
     (if (empty? a-lox)
         accum
         (rev-lox-accum
          (rest a-lox)
          (cons (first a-lox)
                accum))))]
  (rev-lox-accum a-lox '())))
```

- Very similar $\Rightarrow$ good candidates for abstraction
- Recall the design recipe for abstracting over functions from *Animated Problem Solving*:

  1. Compare and mark the differences in the bodies of the functions.
  2. Define the abstract function that takes as additional input the differences.
  3. Refactor the functions abstracted over to use the abstract function.

Abstraction Over L2R Accumulating Folding Functions

```
(define (sum-lon2 a-lon)
 (local
  [(define
    (sum-lon-accum a-lon accum)
     (if (empty? a-lon) accum
         (sum-lon-accum
          (rest a-lon)
          (+ (first a-lon)
             accum))))]
  (sum-lon-accum a-lon 0)))
```

```
(define (rev-lox2 a-lox)
 (local
  [(define
    (rev-lox-accum a-lox accum)
     (if (empty? a-lox) accum
         (rev-lox-accum
          (rest a-lox)
          (cons (first a-lox)
                accum))))]
  (rev-lox-accum a-lox '())))
```

Abstraction Over L2R Accumulating Folding Functions

```
(define (sum-lon2 a-lon)
 (local
  [(define
    (sum-lon-accum a-lon accum)
     (if (empty? a-lon) accum
         (sum-lon-accum
          (rest a-lon)
          (+ (first a-lon)
             accum))))]
  (sum-lon-accum a-lon 0)))
```

```
(define (rev-lox2 a-lox)
 (local
  [(define
    (rev-lox-accum a-lox accum)
     (if (empty? a-lox) accum
         (rev-lox-accum
          (rest a-lox)
          (cons (first a-lox)
                accum))))]
  (rev-lox-accum a-lox '())))
```

-
-
```
(local [;; (listof X) Y arrow Y
        ;; Purpose: Fold from left to right into given accumulato
```
-
```
         (define (aux-f a-lox accum)
```

Abstraction Over L2R Accumulating Folding Functions

```
(define (sum-lon2 a-lon)
 (local
  [(define
    (sum-lon-accum a-lon accum)
     (if (empty? a-lon) accum
         (sum-lon-accum
          (rest a-lon)
          (+ (first a-lon)
             accum))))]
  (sum-lon-accum a-lon 0)))
```

```
(define (rev-lox2 a-lox)
 (local
  [(define
    (rev-lox-accum a-lox accum)
     (if (empty? a-lox) accum
         (rev-lox-accum
          (rest a-lox)
          (cons (first a-lox)
                accum))))]
  (rev-lox-accum a-lox '())))
```

-
```
(local [;; (listof X) Y arrow Y
        ;; Purpose: Fold from left to right into given accumulato
```
-
```
        ;; Accumulator invariant:
        ;;   accum = Y value for traversed list elements so far
```
-
```
        (define (aux-f a-lox accum)
```

Abstraction Over L2R Accumulating Folding Functions

```
(define (sum-lon2 a-lon)
 (local
  [(define
    (sum-lon-accum a-lon accum)
     (if (empty? a-lon) accum
         (sum-lon-accum
          (rest a-lon)
          (+ (first a-lon)
             accum))))]
  (sum-lon-accum a-lon 0)))
```

```
(define (rev-lox2 a-lox)
 (local
  [(define
    (rev-lox-accum a-lox accum)
     (if (empty? a-lox) accum
         (rev-lox-accum
          (rest a-lox)
          (cons (first a-lox)
                accum))))]
  (rev-lox-accum a-lox '())))
```

```
(local [;; (listof X) Y arrow Y
        ;; Purpose: Fold from left to right into given accumulato
        ;; Accumulator invariant:
        ;;   accum = Y value for traversed list elements so far
        (define (aux-f a-lox accum)
          (if (empty? a-lox) accum
              (aux-f (rest a-lox) (comb (first a-lox) accum))))]
```

Abstraction Over L2R Accumulating Folding Functions

```
(define (sum-lon2 a-lon)
 (local
  [(define
    (sum-lon-accum a-lon accum)
     (if (empty? a-lon) accum
         (sum-lon-accum
          (rest a-lon)
          (+ (first a-lon)
             accum))))]
  (sum-lon-accum a-lon 0)))
```

```
(define (rev-lox2 a-lox)
 (local
  [(define
    (rev-lox-accum a-lox accum)
     (if (empty? a-lox) accum
         (rev-lox-accum
          (rest a-lox)
          (cons (first a-lox)
                accum))))]
  (rev-lox-accum a-lox '())))
```

```
;; Y (X Y → Y) (listof X) → Y
;; Purpose: Fold the given list from left to right using the given
;;      initial accumulator value and given combinator function.
(define (fold-from-left base comb a-lox)
  (local [;; (listof X) Y arrow Y
          ;; Purpose: Fold from left to right into given accumulato
          ;; Accumulator invariant:
          ;;   accum = Y value for traversed list elements so far
          (define (aux-f a-lox accum)
            (if (empty? a-lox) accum
                (aux-f (rest a-lox) (comb (first a-lox) accum))))]
```

Abstraction Over L2R Accumulating Folding Functions

```
(define (sum-lon2 a-lon)
 (local
  [(define
    (sum-lon-accum a-lon accum)
     (if (empty? a-lon) accum
         (sum-lon-accum
          (rest a-lon)
          (+ (first a-lon)
             accum))))]
  (sum-lon-accum a-lon 0)))
```

```
(define (rev-lox2 a-lox)
 (local
  [(define
    (rev-lox-accum a-lox accum)
     (if (empty? a-lox) accum
         (rev-lox-accum
          (rest a-lox)
          (cons (first a-lox)
                accum))))]
  (rev-lox-accum a-lox '())))
```

```
;; Y (X Y → Y) (listof X) → Y
;; Purpose: Fold the given list from left to right using the given
;;     initial accumulator value and given combinator function.
(define (fold-from-left base comb a-lox)
  (local [;; (listof X) Y arrow Y
          ;; Purpose: Fold from left to right into given accumulato
          ;; Accumulator invariant:
          ;;   accum = Y value for traversed list elements so far
          (define (aux-f a-lox accum)
            (if (empty? a-lox) accum
                (aux-f (rest a-lox) (comb (first a-lox) accum))))]
          (aux-f a-lox base)))
```

Abstraction Over L2R Accumulating Folding Functions

- Refactor the original functions

```
;; (listof number) → number
;; Purpose: Add the numbers in the given lon
(define (sum-lon4 a-lon) (fold-from-left 0 + a-lon))

;; (listof X) → (listof X)
;; Purpose: Reverse the given list
(define (rev-lox3 a-lox) (fold-from-left '() cons a-lox))
```

Abstraction Over L2R Accumulating Folding Functions

- Refactor the original functions

```
;; (listof number) → number
;; Purpose: Add the numbers in the given lon
(define (sum-lon4 a-lon) (fold-from-left 0 + a-lon))

;; (listof X) → (listof X)
;; Purpose: Reverse the given list
(define (rev-lox3 a-lox) (fold-from-left '() cons a-lox))
```

- Abstraction is so powerful that in ISL+: `foldl`
- The signature for `foldl` is:

```
(X Y → Y) Y (listof X) → Y
```

- Refactor using

```
(define (sum-lon5 a-lon) (foldl + 0 a-lon))

(define (rev-lox3 a-lox) (foldl cons '() a-lox))
```

Abstraction Over L2R Accumulating Folding Functions

Performance

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4
sum-lon3	308	312	313	313	328	314.8
sum-lon5	15	15	15	15	15	15

- CPU times are significantly better for `sum-lon5`
- Programming languages like `ISL+` highly optimize

Abstraction Over L2R Accumulating Folding Functions

Performance

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4
sum-lon3	308	312	313	313	328	314.8
sum-lon5	15	15	15	15	15	15

- CPU times are significantly better for `sum-lon5`
- Programming languages like `ISL+` highly optimize

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
rev-lox	12172	12313	12187	12562	11829	12212.6
rev-lox2	15	15	15	0	0	9
rev-lox3	0	0	0	0	0	0

- The abstraction yields the best CPU times

Abstraction Over R2L Accumulating Folding Functions

```
(define (sum-lon2 a-lon)
  (local
   [(define
     (aux-f a-lon accum)
      (if (empty? a-lon)
          accum
          (aux-f
           (rest a-lon)
           (+ (first a-lon)
              accum))))]
   (aux-f (reverse a-lon) 0)))
```

```
(define (lengths-lostr2 a-lostr)
 (local
  [(define
    (aux-f a-lostr accum)
     (if (empty? a-lostr)
         accum
         (aux-f
          (rest a-lostr)
          (cons (string-length
                 (first a-lostr))
                accum))))]
  (aux-f (reverse a-lostr) '())))
```

Abstraction Over R2L Accumulating Folding Functions

```
(define (sum-lon2 a-lon)
  (local
   [(define
     (aux-f a-lon accum)
      (if (empty? a-lon)
          accum
          (aux-f
           (rest a-lon)
           (+ (first a-lon)
              accum))))]
   (aux-f (reverse a-lon) 0)))
```

```
(define (lengths-lostr2 a-lostr)
 (local
  [(define
    (aux-f a-lostr accum)
     (if (empty? a-lostr)
         accum
         (aux-f
          (rest a-lostr)
          (cons (string-length
                 (first a-lostr))
                accum))))]
  (aux-f (reverse a-lostr) '())))
```

```
;; Y (X Y → Y) (listof X) → Y
;; Purpose: Fold list R2L with given acc val and function
(define (fold-from-right base comb a-lox)
          (aux-f (reverse a-lox) base)))
```

Abstraction Over R2L Accumulating Folding Functions

```
(define (sum-lon2 a-lon)
  (local
   [(define
     (aux-f a-lon accum)
      (if (empty? a-lon)
          accum
          (aux-f
           (rest a-lon)
           (+ (first a-lon)
              accum))))]
   (aux-f (reverse a-lon) 0)))
```

```
(define (lengths-lostr2 a-lostr)
 (local
  [(define
    (aux-f a-lostr accum)
     (if (empty? a-lostr)
         accum
         (aux-f
          (rest a-lostr)
          (cons (string-length
                 (first a-lostr))
                accum))))]
  (aux-f (reverse a-lostr) '())))
```

```
;; Y (X Y → Y) (listof X) → Y
;; Purpose: Fold list R2L with given acc val and function
(define (fold-from-right base comb a-lox)
  (local [;;(listof X) Y → Y Purpose:Fold list R2L into given acc
          ;;Acc invt: accum = Y value for traversed list elements
          (define (aux-f a-lox accum)
            (if (empty? a-lox)
                accum
                (aux-f (rest a-lox)
                       (comb (first a-lox) accum))))]
    (aux-f (reverse a-lox) base)))
```

Abstraction Over R2L Accumulating Folding Functions

Performance

- `(define (sum-lon6 a-lon) (foldr + 0 a-lon))`

Abstraction Over R2L Accumulating Folding Functions

Performance

- `(define (sum-lon6 a-lon) (foldr + 0 a-lon))`

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
sum-lon	344	390	328	329	312	340.6
sum-lon2	297	313	297	328	297	306.4
sum-lon3	308	312	313	313	328	314.8
sum-lon5	15	15	15	15	15	15
sum-lon6	78	78	32	47	78	47

- `ISL+`'s abstract function performs significantly better than the solutions using structural recursion and accumulative recursion
- `foldl` always performs slightly better because folding a list from the right requires a little more work than folding a list from the left

Abstraction Over R2L Accumulating Folding Functions

Performance

- `lengths-lostr2`'s refactoring requires slightly more care
- A function is applied to the first element of the list before it is folded into the accumulator

Abstraction Over R2L Accumulating Folding Functions

Performance

- `lengths-lostr2`'s refactoring requires slightly more care
- A function is applied to the first element of the list before it is folded into the accumulator
- ```
  (define (lengths-lostr3 a-lostr)
    (foldr (λ (a-str acc)
              (cons (string-length a-str) acc))
           '()
           a-lostr))
  ```
  ```

Abstraction Over R2L Accumulating Folding Functions

Performance

- `lengths-lostr2`'s refactoring requires slightly more care
- A function is applied to the first element of the list before it is folded into the accumulator
- ```
  (define (lengths-lostr3 a-lostr)
    (foldr (λ (a-str acc)
              (cons (string-length a-str) acc))
           '()
           a-lostr))
  ```
-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
lengths-lostr	78	78	109	62	47	74.8
lengths-lostr2	78	63	47	78	78	68.8
lengths-lostr3	15	0	0	0	0	3

- Same pattern as with other benchmarks
  ```

Abstraction Over Right to Left Accumulating Folding Functions

Conclusions

- Accumulators provide significant gains in performance for list folding operations
- This stems from the functions using accumulators being tail-recursive or efficient iteration

Abstraction Over Right to Left Accumulating Folding Functions

Conclusions

- Accumulators provide significant gains in performance for list folding operations
- This stems from the functions using accumulators being tail-recursive or efficient iteration
- Caution: not all programming languages implement tail-recursive programs efficiently
- Inform yourself if tail-recursion is optimized and run empirical experiments when performance is important

Abstraction Over Right to Left Accumulating Folding Functions

Conclusions

- Accumulators provide significant gains in performance for list folding operations
- This stems from the functions using accumulators being tail-recursive or efficient iteration
- Caution: not all programming languages implement tail-recursive programs efficiently
- Inform yourself if tail-recursion is optimized and run empirical experiments when performance is important
- Many programming languages optimize built-in functions
- This is not to say that it is impossible for you to write a more efficient function than one provided by the programming language
- If you need better performance write your own function and conduct empirical experiments

Abstraction Over Right to Left Accumulating Folding Functions

Conclusions

- Accumulators provide significant gains in performance for list folding operations
- This stems from the functions using accumulators being tail-recursive or efficient iteration
- Caution: not all programming languages implement tail-recursive programs efficiently
- Inform yourself if tail-recursion is optimized and run empirical experiments when performance is important
- Many programming languages optimize built-in functions
- This is not to say that it is impossible for you to write a more efficient function than one provided by the programming language
- If you need better performance write your own function and conduct empirical experiments
- When an abstraction is not provided by the programming language then you must write your own

Abstraction Over Right to Left Accumulating Folding Functions

HOMEWORK

- Problems: 13-15
- QUIZ: Problem 2 (due in 1 week)

N-Puzzle Version 6

- N-Puzzle solved using depth-first search and breadth-first search
- Both benefit from the use of an accumulator
- DFS can be very fast but does not always find the shortest solution
- BFS always finds the shortest solution but can be slow

N-Puzzle Version 6

- N-Puzzle solved using depth-first search and breadth-first search
- Both benefit from the use of an accumulator
- DFS can be very fast but does not always find the shortest solution
- BFS always finds the shortest solution but can be slow
- Neither strategy makes an effort to intelligently decide which paths to explore

N-Puzzle Version 6

- N-Puzzle solved using depth-first search and breadth-first search
- Both benefit from the use of an accumulator
- DFS can be very fast but does not always find the shortest solution
- BFS always finds the shortest solution but can be slow
- Neither strategy makes an effort to intelligently decide which paths to explore
- Don't blindly explore paths by picking the "best" path to explore

N-Puzzle Version 6

- N-Puzzle solved using depth-first search and breadth-first search
- Both benefit from the use of an accumulator
- DFS can be very fast but does not always find the shortest solution
- BFS always finds the shortest solution but can be slow
- Neither strategy makes an effort to intelligently decide which paths to explore
- Don't blindly explore paths by picking the "best" path to explore
- Use a *heuristic* to guide the search
- A heuristic is a function that ranks the alternatives in a search
- `A*` is a *best-first search* algorithm used in AI

N-Puzzle Version 6

- N-Puzzle solved using depth-first search and breadth-first search
- Both benefit from the use of an accumulator
- DFS can be very fast but does not always find the shortest solution
- BFS always finds the shortest solution but can be slow
- Neither strategy makes an effort to intelligently decide which paths to explore
- Don't blindly explore paths by picking the "best" path to explore
- Use a *heuristic* to guide the search
- A heuristic is a function that ranks the alternatives in a search
- A* is a *best-first search* algorithm used in AI
- Caution: a heuristic may not always converge to a solution faster.
- It may not converge to a solution

N-Puzzle Version 6

Manhattan Distance

- Insight into the problem must be developed as for generative recursion

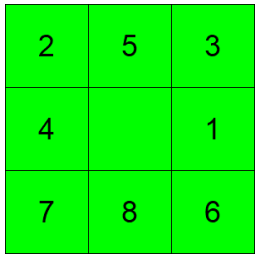
N-Puzzle Version 6

Manhattan Distance

- Insight into the problem must be developed as for generative recursion
- For the N-puzzle approximate how many moves are needed to solve it
- The path with the smallest number of approximate moves is explored

N-Puzzle Version 6

Manhattan Distance

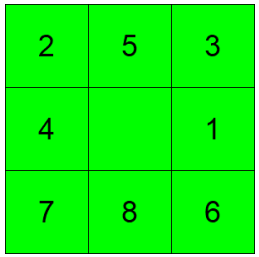
- Insight into the problem must be developed as for generative recursion
- For the N-puzzle approximate how many moves are needed to solve it
- The path with the smallest number of approximate moves is explored
- 

- The distance of each tile to its correct position in `WIN`:

Tile	Distance
1	3
2	1
3	0
4	0
5	1
6	1
7	0
8	0

N-Puzzle Version 6

Manhattan Distance

- Insight into the problem must be developed as for generative recursion
- For the N-puzzle approximate how many moves are needed to solve it
- The path with the smallest number of approximate moves is explored
- 

- The distance of each tile to its correct position in `WIN`:

Tile	Distance
1	3
2	1
3	0
4	0
5	1
6	1
7	0
8	0

- The *Manhattan distance* of a `world` is the sum of all the tile distances. For the `world` above we have:

```
manhattan distance = 3 + 1 + 0 + 0 + 1 + 1 + 0 + 0 = 6
```

- Use the Manhattan distance as an approximation for the number of moves required to solve the puzzle

N-Puzzle Version 6

- ```
  ;; world → natnum  Purpose: Compute the Manhattan
  (define (manhattan-distance a-world)
   (local
  ```

- ```
        (sum-distances 8)))
  ```

N-Puzzle Version 6

```
;; world → natnum  Purpose: Compute the Manhattan
(define (manhattan-distance a-world)
 (local
  [;; bpos → natnum Purpose: Sum the tile distances
   (define (sum-distances a-bpos)
    (local [;; world bpos → bval Purpose: Return given bpos' bval
            (define (tile-value a-bpos) (cond ...))
      (sum-distances 8)))
```

N-Puzzle Version 6

```
;; world → natnum  Purpose: Compute the Manhattan
(define (manhattan-distance a-world)
 (local
  [;; bpos → natnum Purpose: Sum the tile distances
   (define (sum-distances a-bpos)
    (local [;; world bpos → bval Purpose: Return given bpos' bval
            (define (tile-value a-bpos) (cond ...))
            ;; tval → bpos Purpose: Return WIN bpos for given tval
            (define (final-bpos a-tval)
             (if (= a-tval 0) 8 (sub1 a-tval)))
      (sum-distances 8)))
```

N-Puzzle Version 6

```
;; world → natnum  Purpose: Compute the Manhattan
(define (manhattan-distance a-world)
 (local
  [;; bpos → natnum Purpose: Sum the tile distances
   (define (sum-distances a-bpos)
    (local [;; world bpos → bval Purpose: Return given bpos' bval
            (define (tile-value a-bpos) (cond ...))
            ;; tval → bpos Purpose: Return WIN bpos for given tval
            (define (final-bpos a-tval)
             (if (= a-tval 0) 8 (sub1 a-tval)))
            ;; bpos → natnum  Purpose: Return bpos' row
            (define (get-row a-bpos) (quotient a-bpos 3))
            ;; bpos → natnum  Purpose: Return bpos' column
            (define (get-col a-bpos) (remainder a-bpos 3))
      (sum-distances 8)))
```

N-Puzzle Version 6

```
;; world → natnum  Purpose: Compute the Manhattan
(define (manhattan-distance a-world)
 (local
  [;; bpos → natnum Purpose: Sum the tile distances
   (define (sum-distances a-bpos)
    (local [;; world bpos → bval Purpose: Return given bpos' bval
            (define (tile-value a-bpos) (cond ...))
            ;; tval → bpos Purpose: Return WIN bpos for given tval
            (define (final-bpos a-tval)
             (if (= a-tval 0) 8 (sub1 a-tval)))
            ;; bpos → natnum  Purpose: Return bpos' row
            (define (get-row a-bpos) (quotient a-bpos 3))
            ;; bpos → natnum  Purpose: Return bpos' column
            (define (get-col a-bpos) (remainder a-bpos 3))
            ;; bpos bpos → natnum Purpose: Return distance
            (define (distance bpos1 bpos2)
             (if (= (tile-value bpos1) 0) 0
                 (+ (abs (- (get-row bpos1) (get-row bpos2)))
                    (abs (- (get-col bpos1) (get-col bpos2))))))
```

```
      (sum-distances 8)))
```

N-Puzzle Version 6

```
;; world → natnum  Purpose: Compute the Manhattan
(define (manhattan-distance a-world)
 (local
  [;; bpos → natnum Purpose: Sum the tile distances
   (define (sum-distances a-bpos)
    (local [;; world bpos → bval Purpose: Return given bpos' bval
            (define (tile-value a-bpos) (cond ...))
            ;; tval → bpos Purpose: Return WIN bpos for given tval
            (define (final-bpos a-tval)
             (if (= a-tval 0) 8 (sub1 a-tval)))
            ;; bpos → natnum  Purpose: Return bpos' row
            (define (get-row a-bpos) (quotient a-bpos 3))
            ;; bpos → natnum  Purpose: Return bpos' column
            (define (get-col a-bpos) (remainder a-bpos 3))
            ;; bpos bpos → natnum Purpose: Return distance
            (define (distance bpos1 bpos2)
             (if (= (tile-value bpos1) 0) 0
                 (+ (abs (- (get-row bpos1) (get-row bpos2)))
                    (abs (- (get-col bpos1) (get-col bpos2))))))
            (define win-bpos (final-bpos (tile-value a-bpos)))]
      (sum-distances 8)))
```

N-Puzzle Version 6

```
;; world → natnum  Purpose: Compute the Manhattan
(define (manhattan-distance a-world)
 (local
  [;; bpos → natnum Purpose: Sum the tile distances
   (define (sum-distances a-bpos)
    (local [;; world bpos → bval Purpose: Return given bpos' bval
            (define (tile-value a-bpos) (cond ...))
            ;; tval → bpos Purpose: Return WIN bpos for given tval
            (define (final-bpos a-tval)
             (if (= a-tval 0) 8 (sub1 a-tval)))
            ;; bpos → natnum  Purpose: Return bpos' row
            (define (get-row a-bpos) (quotient a-bpos 3))
            ;; bpos → natnum  Purpose: Return bpos' column
            (define (get-col a-bpos) (remainder a-bpos 3))
            ;; bpos bpos → natnum Purpose: Return distance
            (define (distance bpos1 bpos2)
             (if (= (tile-value bpos1) 0) 0
                 (+ (abs (- (get-row bpos1) (get-row bpos2)))
                    (abs (- (get-col bpos1) (get-col bpos2))))))
            (define win-bpos (final-bpos (tile-value a-bpos)))]
      (if (= a-bpos 0)
          (distance a-bpos win-bpos)
          (+ (distance a-bpos win-bpos) (sum-distances (sub1 a-bpos
    (sum-distances 8)))
```

N-Puzzle Version 6

HOMEWORK

- Problem: 1

N-Puzzle Version 6

Problem Analysis

- Explore the most promising paths first: fast + path found guarantee
- The best path is the path with the smallest Manhattan distance
- Paths may not be stored in a queue
- Paths are stored in a `(listof (listof world))`

N-Puzzle Version 6

Problem Analysis

- Explore the most promising paths first: fast + path found guarantee
- The best path is the path with the smallest Manhattan distance
- Paths may not be stored in a queue
- Paths are stored in a `(listof (listof world))`
- Previously defined sample queues are recast as sample `(listof (listof world))`:

```
;; Sample (listof (listof world))
(define LLOW1 (list (list WIN)))
(define LLOW2 (list (list WIN
                          (make-world 1 2 3
                                      4 5 0
                                      7 8 6))))
(define LLOW3 (list ...))

(define LLOW4 (list (list (make-world 1 2 3
                                      0 5 6
                                      4 7 8)
                          ...)
                    (list (make-world 1 2 3
                                      4 5 6
                                      7 0 8)
                          ...)))
```

N-Puzzle Version 6

Problem Analysis

- An accumulator to remember the `worlds` whose successors are part of the search is still needed
- Accumulator is used to reduce the number of paths generated
- Accumulator invariant also remains unchanged:

```
;; Accumulator Invariant:
;;  visited = worlds whose successors are part of the search
```

N-Puzzle Version 6

Problem Analysis

- An accumulator to remember the `worlds` whose successors are part of the search is still needed
- Accumulator is used to reduce the number of paths generated
- Accumulator invariant also remains unchanged:

```
;; Accumulator Invariant:
;;  visited = worlds whose successors are part of the search
```

- At each step the best path is extracted
- If first world of best path is `WIN` then the reverse of the best path

N-Puzzle Version 6

Problem Analysis

- An accumulator to remember the `worlds` whose successors are part of the search is still needed
- Accumulator is used to reduce the number of paths generated
- Accumulator invariant also remains unchanged:

```
;; Accumulator Invariant:
;;  visited = worlds whose successors are part of the search
```

- At each step the best path is extracted
- If first world of best path is `WIN` then the reverse of the best path
- Otherwise, a recursive call is made with new paths and a new accumulator
- New paths: remove best path and new paths using new successors
- The new accumulator is obtained by adding the best path's first `world`

N-Puzzle Version 6

Problem Analysis

- An accumulator to remember the `worlds` whose successors are part of the search is still needed
- Accumulator is used to reduce the number of paths generated
- Accumulator invariant also remains unchanged:

```
;; Accumulator Invariant:
;;  visited = worlds whose successors are part of the search
```

- At each step the best path is extracted
- If first world of best path is `WIN` then the reverse of the best path
- Otherwise, a recursive call is made with new paths and a new accumulator
- New paths: remove best path and new paths using new successors
- The new accumulator is obtained by adding the best path's first `world`
- Refactor `make-move`:

```
;; world → world
;; Purpose: Make a move for the player
(define (make-move a-world)
  (if (equal? a-world WIN)
      a-world
      (second (find-solution-a* (list (list a-world))
                                '()))))
```

N-Puzzle Version 6

Sample Expressions and Differences

- For each path compute first world Manhattan distance
- First path with the smallest Manhattan distance is best path
- Suggests traversing the list from left to right using `foldl`
- Accumulated value is path with the minimum Manhattan distance so far

N-Puzzle Version 6

Sample Expressions and Differences

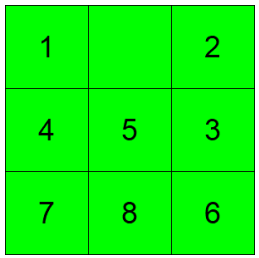
- For each path compute first world Manhattan distance
- First path with the smallest Manhattan distance is best path
- Suggests traversing the list from left to right using `foldl`
- Accumulated value is path with the minimum Manhattan distance so far
-

```
(define FS-LLOW1-VAL        Exprs best path's first world is WIN
  (local [(define best-path
            (foldl (λ (p accum)
                     (if (< (manhattan-distance (first p))
                            (manhattan-distance (first accum)))
                         p
                         accum))
                   (first LLOW1)
                   (rest  LLOW1)))]
    (reverse best-path)))
(define FS-LLOW2-VAL
  (local [(define best-path
            (foldl (λ (p accum)
                     (if (< (manhattan-distance (first p))
                            (manhattan-distance (first accum)))
                         p
                         accum))
                   (first LLOW2)
                   (rest  LLOW2)))]
    (reverse best-path)))
```

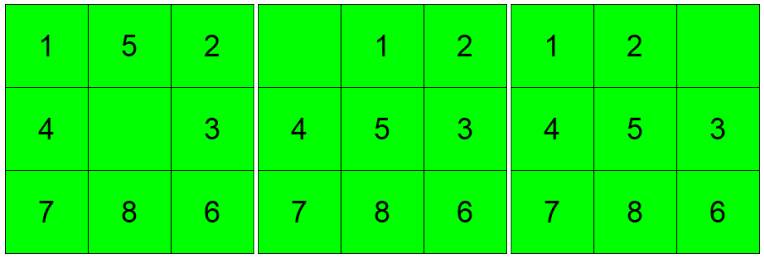
N-Puzzle Version 6

Sample Expressions and Differences

- Exprs best path's first `world` is not `WIN`
- pick a `world` that may be solved in a small number of steps. For example, consider solving the following puzzle:



Its successors are:

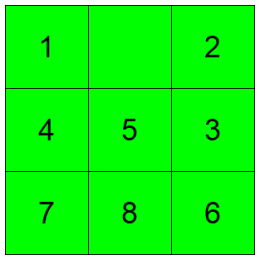


- Successors do not include `WIN`

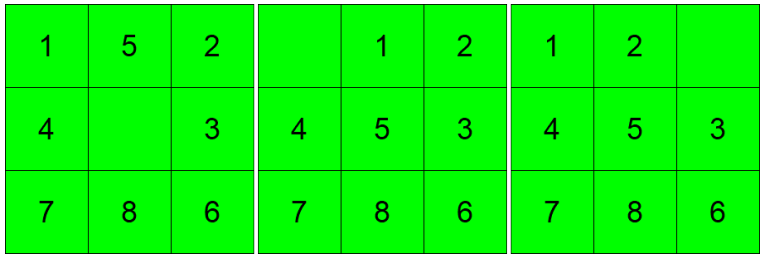
N-Puzzle Version 6

Sample Expressions and Differences

- Exprs best path's first `world` is not `WIN`
- pick a `world` that may be solved in a small number of steps. For example, consider solving the following puzzle:



Its successors are:



- Successors do not include `WIN`
- We may define a `(listof (listof world))` with 3 paths:

```
(define LLOW5 (list (list (make-world 1 5 2 4 0 3 7 8 6)
                          (make-world 1 0 2 4 5 3 7 8 6))
                    (list (make-world 0 1 2 4 5 3 7 8 6)
                          (make-world 1 0 2 4 5 3 7 8 6))
                    (list (make-world 1 2 0 4 5 3 7 8 6)
                          (make-world 1 0 2 4 5 3 7 8 6)))))
```

N-Puzzle Version 6

Sample Expressions and Differences

- Exprs best path's first `world` is not `WIN`

```
(define FS-LLOW5-VAL
  (local [(define best-path
            (foldl
             (λ (p accum)
               (if (< (manhattan-distance (first p))
                      (manhattan-distance (first accum)))
                   p
                   accum))
             (first LLOW5)
             (rest  LLOW5)))
          (define first-world (first best-path))
```

N-Puzzle Version 6

Sample Expressions and Differences

- Exprs best path's first `world` is not `WIN`

```
(define FS-LLOW5-VAL
  (local [(define best-path
            (foldl
             (λ (p accum)
               (if (< (manhattan-distance (first p))
                      (manhattan-distance (first accum)))
                   p
                   accum))
             (first LLOW5)
             (rest  LLOW5)))
          (define first-world (first best-path))
```

-

```
          (define successors
            (map (λ (neigh) (swap-empty first-world neigh))
                 (filter (λ (w)
                           (not (member? w (list (make-world 1 0 2 4 5 3 7 8 6)))))
                         (list-ref neighbors (blank-pos first-world)))))
          (define new-paths (map (λ (w) (cons w best-path))
                                 successors))
          (define new-llow (append (remove best-path LLOW5) new-paths))]
```

N-Puzzle Version 6

Sample Expressions and Differences

- Exprs best path's first `world` is not `WIN`

```
(define FS-LLOW5-VAL
  (local [(define best-path
            (foldl
             (λ (p accum)
               (if (< (manhattan-distance (first p))
                      (manhattan-distance (first accum)))
                   p
                   accum))
             (first LLOW5)
             (rest  LLOW5)))
          (define first-world (first best-path))
```

-
```
          (define successors
            (map (λ (neigh) (swap-empty first-world neigh))
                 (filter (λ (w)
                           (not (member? w (list (make-world 1 0 2 4 5 3 7 8 6)))))
                         (list-ref neighbors (blank-pos first-world)))))
          (define new-paths (map (λ (w) (cons w best-path))
                                 successors))
          (define new-llow (append (remove best-path LLOW5) new-paths))]
```

-
```
    (find-solution-a* new-llow
                      (cons first-world
                            (list (make-world 1 0 2 4 5 3 7 8 6))))))
```

N-Puzzle Version 6

Sample Expressions and Differences

- Exprs best path's first `world` is not `WIN`

```
(define FS-LLOW4-VAL
  (local [(define best-path
                  (foldl
                   (λ (p accum)
                    (if (< (manhattan-distance (first p))
                           (manhattan-distance (first accum)))
                        p
                        accum))
                   (first LLOW4)
                   (rest  LLOW4)))
          (define first-world (first best-path))
```

N-Puzzle Version 6

Sample Expressions and Differences

- Exprs best path's first `world` is not `WIN`

```
(define FS-LLOW4-VAL
  (local [(define best-path
                  (foldl
                   (λ (p accum)
                    (if (< (manhattan-distance (first p))
                           (manhattan-distance (first accum)))
                        p
                        accum))
                   (first LLOW4)
                   (rest  LLOW4)))
          (define first-world (first best-path))
```

-

```
          (define successors
           (filter (λ (w)
                    (not (member? w (list (make-world 1 2 3 4 5 6 0 7 8)))))
                   (map (λ (neigh)
                          (swap-empty first-world neigh))
                        (filter (λ (w)
                                  (not (member?
                                        w
                                        (list (make-world 1 2 3 4 5 6 0 7 8)))))
                                (list-ref neighbors (blank-pos first-world))))))
          (define new-paths (map (λ (w) (cons w best-path))
                                 successors))
          (define new-llow (append (remove best-path LLOW4) new-paths))]
```

N-Puzzle Version 6

Sample Expressions and Differences

- Exprs best path's first world is not WIN

```
(define FS-LLOW4-VAL
  (local [(define best-path
                  (foldl
                   (λ (p accum)
                    (if (< (manhattan-distance (first p))
                           (manhattan-distance (first accum)))
                        p
                        accum))
                   (first LLOW4)
                   (rest  LLOW4)))
          (define first-world (first best-path))
```

-
```
          (define successors
           (filter (λ (w)
                    (not (member? w (list (make-world 1 2 3 4 5 6 0 7 8)))))
                   (map (λ (neigh)
                     (swap-empty first-world neigh))
                     (filter (λ (w)
                               (not (member?
                                     w
                                     (list (make-world 1 2 3 4 5 6 0 7 8)))))
                             (list-ref neighbors (blank-pos first-world))))))
          (define new-paths (map (λ (w) (cons w best-path))
                                 successors))
          (define new-llow (append (remove best-path LLOW4) new-paths))]
```

-
```
    (find-solution-a* new-llow
                      (cons first-world
                            (list (make-world 1 2 3 4 5 6 0 7 8))))))
```

N-Puzzle Version 6

Sample Expressions and Differences

- Differences: `(listof (listof world))` for the paths and `(listof world)` for the accumulator

N-Puzzle Version 6

Sample Expressions and Differences

- ```
  ;; (listof (listof world)) (listof world) → (listof world)
  ;; Purpose: Return sequence of moves to WIN
  ;; How: If the best path's first world is WIN then return
  ;;      the reverse of the best path. Otherwise, continue
  ;;      the search with:
  ;;        1. a new list of paths is obtained by removing the
  ;;           best path and adding paths constructed using
  ;;           the unvisited successors of the best path's
  ;;           first world and the best path.
  ;;        2. an accumulator obtained by adding the best
  ;;           path's first world to the given accumulator.
  ;; Accumulator Invariant:
  ;;  visited = worlds whose successors are part of the search
  ```
  ```

N-Puzzle Version 6

Function Definition

- `(define (find-solution-a* a-llow visited)`

N-Puzzle Version 6

Function Definition

```
(define (find-solution-a* a-llow visited)
  (local [(define best-path   Common to all sample exprs
                   (foldl
                    (λ (p accum)
                     (if (< (manhattan-distance (first p))
                            (manhattan-distance (first accum)))
                         p
                         accum))
                    (first a-llow)
                    (rest a-llow)))
          (define first-world (first best-path))]
```

N-Puzzle Version 6

Function Definition

-
```
(define (find-solution-a* a-llow visited)
```
-
```
  (local [(define best-path   Common to all sample exprs
                  (foldl
                   (λ (p accum)
                    (if (< (manhattan-distance (first p))
                           (manhattan-distance (first accum)))
                        p
                        accum))
                   (first a-llow)
                   (rest a-llow)))
          (define first-world (first best-path))]
```
-
```
    (if (equal? first-world WIN)
        (reverse best-path)
        (local   Common to sample exprs for which WIN is not first
           [(define successors
                    (filter
                      (λ (w) (not (member? w visited)))
                      (map (λ (neigh) (swap-empty first-world neigh))
                           (list-ref neighbors
                                     (blank-pos first-world)))))
            (define new-paths (map (λ (w) (cons w best-path))
                                   successors))
            (define new-llow (append (remove best-path a-llow)
                                     new-paths))]
          (find-solution-a* new-llow (cons first-world visited)))))
```

N-Puzzle Version 6

Tests

```
;; Tests using sample computations for find-solution-a*
(check-expect (find-solution-a* LLOW1 '()) FS-LLOW1-VAL)
(check-expect (find-solution-a* LLOW2 '()) FS-LLOW2-VAL)
(check-expect (find-solution-a*
               LLOW4
               (list (make-world 1 2 3 4 5 6 0 7 8)))
              FS-LLOW4-VAL)
(check-expect (find-solution-a*
               LLOW5
               (list (make-world 1 0 2 4 5 3 7 8 6)))
              FS-LLOW5-VAL)
;; Tests using sample values for find-solution-a*
(check-expect
  (find-solution-a* (list (list (make-world 0 2 3 1 5 6 4 7 8))) '())
  (list (make-world 0 2 3 1 5 6 4 7 8)
        (make-world 1 2 3 0 5 6 4 7 8)
        (make-world 1 2 3 4 5 6 0 7 8)
        (make-world 1 2 3 4 5 6 7 0 8)
        (make-world 1 2 3 4 5 6 7 8 0)))
(check-expect
  (find-solution-a*
    (list (list (make-world 1 2 3 4 5 6 7 0 8) (make-world 1 2 3 4 0 6 7 5 8))
          (list (make-world 1 0 3 4 2 6 7 5 8) (make-world 1 2 3 4 0 6 7 5 8))
          (list (make-world 1 2 3 0 4 6 7 5 8) (make-world 1 2 3 4 0 6 7 5 8))
          (list (make-world 1 2 3 4 6 0 7 5 8) (make-world 1 2 3 4 0 6 7 5 8)))
    (list (make-world 1 2 3 4 0 6 7 5 8)))
  (list (make-world 1 2 3 4 0 6 7 5 8)
        (make-world 1 2 3 4 5 6 7 0 8)
        (make-world 1 2 3 4 5 6 7 8 0)))
```

N-Puzzle Version 6

Termination Argument

- Just like breadth-first search, the A* algorithm exhaustively traverses all paths rooted at a given `world` once
- Exhaustive search is guaranteed because the successors of each `world` encountered are only used to create new paths and the path used to create them is eliminated from the search
- Every new path is explored if and when it has the smallest Manhattan distance
- Given that there is a solution from every (valid) `world` with the smallest Manhattan distance eventually a path reaches `WIN` and the process halts

N-Puzzle Version 6

Performance

-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Sol Len
BFSA1	843	984	1031	953	985	959.2	12
BFSA2	15	0	0	15	0	6	5
BFSA3	15	0	0	0	0	3	7
BFSA4	0	0	0	0	0	0	4
BFSA*1	0	0	0	0	0	0	12
BFSA*2	0	0	0	0	0	0	5
BFSA*3	0	0	0	0	0	0	7
BFSA*4	0	0	0	0	0	0	4

N-Puzzle Version 6

Performance

-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Sol Len
BFSA1	843	984	1031	953	985	959.2	12
BFSA2	15	0	0	15	0	6	5
BFSA3	15	0	0	0	0	3	7
BFSA4	0	0	0	0	0	0	4
BFSA*1	0	0	0	0	0	0	12
BFSA*2	0	0	0	0	0	0	5
BFSA*3	0	0	0	0	0	0	7
BFSA*4	0	0	0	0	0	0	4

- A* is always faster or just as fast
- `BFSA*` finds the shortest solution faster than `BFSA` Why?

N-Puzzle Version 6

Performance

-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Sol Len
BFSA1	843	984	1031	953	985	959.2	12
BFSA2	15	0	0	15	0	6	5
BFSA3	15	0	0	0	0	3	7
BFSA4	0	0	0	0	0	0	4
BFSA*1	0	0	0	0	0	0	12
BFSA*2	0	0	0	0	0	0	5
BFSA*3	0	0	0	0	0	0	7
BFSA*4	0	0	0	0	0	0	4

- A* is always faster or just as fast
- `BFSA*` finds the shortest solution faster than `BFSA` Why?
- Look at the number of paths being explored
- 603 paths in the queue when the solution is found by `BFSA 1`
- When `BFSA* 1` finds a solution there are 14 paths in the given `(listof (listof world))`
- `BFSA*` is searching fewer paths
- `BFSA*` does not invest as much effort as `BFSA` searching paths that are not promising
- Illustrates that the Manhattan distance is an effective heuristic

N-Puzzle Version 6

HOMEWORK

- Problems:4-6
- MIDTERM (due in 1 week): Problem 7

Continuation-Passing Style

- Use accumulators to make programs tail-recursive

Continuation-Passing Style

- Use accumulators to make programs tail-recursive
- Delayed operation: the function must remember to return to complete it
- The information that must be remembered is *control information*
- Accumulator eliminates the need to remember control information

Continuation-Passing Style

- Use accumulators to make programs tail-recursive
- Delayed operation: the function must remember to return to complete it
- The information that must be remembered is *control information*
- Accumulator eliminates the need to remember control information
- How are delayed operations eliminated when there is more than on function call in an argument position?

Continuation-Passing Style

- Use accumulators to make programs tail-recursive
- Delayed operation: the function must remember to return to complete it
- The information that must be remembered is *control information*
- Accumulator eliminates the need to remember control information
- How are delayed operations eliminated when there is more than on function call in an argument position?
- ```
  (define (quick-sorting a-lon)
    (if (empty? a-lon)
        '()
        (local [(define SMALLER= (filter (λ (i) (<= i (first a-lon)))
                                         (rest a-lon)))
                (define GREATER  (filter (λ (i) (> i (first a-lon)))
                                         (rest a-lon)))]
          (append (quick-sorting SMALLER=)
                  (cons (first a-lon) (quick-sorting GREATER))))))
  ```
- Three steps are needed:
  1. Evaluate `(quick-sorting SMALLER=)` remember two more things to do
  2. Evaluate `(cons (first a-lon) (quick-sorting GREATER))`

     remember one more thing to do
  3. Evaluate append
  ```

Continuation-Passing Style

- Use accumulators to make programs tail-recursive
- Delayed operation: the function must remember to return to complete it
- The information that must be remembered is *control information*
- Accumulator eliminates the need to remember control information
- How are delayed operations eliminated when there is more than on function call in an argument position?
- ```
  (define (quick-sorting a-lon)
    (if (empty? a-lon)
        '()
        (local [(define SMALLER= (filter (λ (i) (<= i (first a-lon)))
                                         (rest a-lon)))
                (define GREATER  (filter (λ (i) (> i (first a-lon)))
                                         (rest a-lon)))]
          (append (quick-sorting SMALLER=)
                  (cons (first a-lon) (quick-sorting GREATER))))))
  ```
- Three steps are needed:
  1. Evaluate `(quick-sorting SMALLER=)` remember two more things to do
  2. Evaluate `(cons (first a-lon) (quick-sorting GREATER))` remember one more thing to do
  3. Evaluate append
- It is difficult to see how an accumulator may be introduced
  ```

Continuation-Passing Style

- Use accumulators to make programs tail-recursive
- Delayed operation: the function must remember to return to complete it
- The information that must be remembered is *control information*
- Accumulator eliminates the need to remember control information
- How are delayed operations eliminated when there is more than on function call in an argument position?
- ```
  (define (quick-sorting a-lon)
    (if (empty? a-lon)
        '()
        (local [(define SMALLER= (filter (λ (i) (<= i (first a-lon)))
                                         (rest a-lon)))
                (define GREATER  (filter (λ (i) (> i (first a-lon)))
                                         (rest a-lon)))]
          (append (quick-sorting SMALLER=)
                  (cons (first a-lon) (quick-sorting GREATER))))))
  ```
- Three steps are needed:
  1. Evaluate `(quick-sorting SMALLER=)` remember two more things to do
  2. Evaluate `(cons (first a-lon) (quick-sorting GREATER))` remember one more thing to do
  3. Evaluate append
- It is difficult to see how an accumulator may be introduced
- Control information must be accumulated!
  ```

Continuation-Passing Style

Accumulating Control

- To finish the computation the result obtained from one or more previous steps and control information are used to finish the computation.
- Consider sorting `'(5 8 2 9 1 8 4)`:

```
  (quick-sort '(5 8 2 9 1 8 4))
= (append (quick-sorting '(2 1 4))
          (cons 5 (quick-sorting '(8 9 8))))
```

Continuation-Passing Style

Accumulating Control

- To finish the computation the result obtained from one or more previous steps and control information are used to finish the computation.
- Consider sorting `'(5 8 2 9 1 8 4)`:

```
  (quick-sort '(5 8 2 9 1 8 4))
= (append (quick-sorting '(2 1 4))
          (cons 5 (quick-sorting '(8 9 8))))
```

- Let us assume `(quick-sorting '(2 1 4))` is evaluated first
- Call this value `ssm` (for sorted smaller)
- How is the computation completed?

Continuation-Passing Style

Accumulating Control

- To finish the computation the result obtained from one or more previous steps and control information are used to finish the computation.
- Consider sorting `'(5 8 2 9 1 8 4)`:

```
  (quick-sort '(5 8 2 9 1 8 4))
= (append (quick-sorting '(2 1 4))
          (cons 5 (quick-sorting '(8 9 8))))
```

- Let us assume `(quick-sorting '(2 1 4))` is evaluated first
- Call this value `ssm` (for sorted smaller)
- How is the computation completed?
- The expression that needs to be evaluated becomes:

```
(append ssm (cons 5 (quick-sorting '(8 9 8))))
```

- Control information must once again be accumulated to finish the computation after (the second) call to `quick-sorting`
- Let us name the value of this call `sgr` (for sorted greater)

Continuation-Passing Style

Accumulating Control

- To finish the computation the result obtained from one or more previous steps and control information are used to finish the computation.
- Consider sorting `'(5 8 2 9 1 8 4)`:

```
  (quick-sort '(5 8 2 9 1 8 4))
= (append (quick-sorting '(2 1 4))
          (cons 5 (quick-sorting '(8 9 8))))
```

- Let us assume `(quick-sorting '(2 1 4))` is evaluated first
- Call this value `ssm` (for sorted smaller)
- How is the computation completed?
- The expression that needs to be evaluated becomes:

```
(append ssm (cons 5 (quick-sorting '(8 9 8))))
```

- Control information must once again be accumulated to finish the computation after (the second) call to `quick-sorting`
- Let us name the value of this call `sgr` (for sorted greater)
- The expression that must be evaluated becomes:

```
(append ssm (cons 5 sgr))
```

- At this point `append` may be called and its value returned.

Continuation-Passing Style

Accumulating Control

- The evaluation described above is an iteration

Continuation-Passing Style

Accumulating Control

- The evaluation described above is an iteration
- If there are no function calls in an argument position return the value of the expression
- If there is a function call in an argument position:
  1. Evaluate a function call in an argument position.
  2. Use the value obtained to evaluate a specialized expression to complete the computation.

Continuation-Passing Style

Accumulating Control

- The evaluation described above is an iteration
- If there are no function calls in an argument position return the value of the expression
- If there is a function call in an argument position:
  1. Evaluate a function call in an argument position.
  2. Use the value obtained to evaluate a specialized expression to complete the computation.
- This suggests that a computation may be divided into two parts
- The first evaluates a function call in an argument position
- The second part is a function that takes as input an intermediate result and its body is an expression specialized by substituting the evaluated function call with the function's parameter
- This function, in essence, knows how to finish the computation and is called a *continuation*
- A continuation is a function that accumulates all the control information needed to complete a computation.

Continuation-Passing Style

Accumulating Control

- Consider quick sorting a list of numbers using a continuation

Continuation-Passing Style

Accumulating Control

- Consider quick sorting a list of numbers using a continuation
- The sorting function must take an additional input: the continuation to finish the computation

Continuation-Passing Style

Accumulating Control

- Consider quick sorting a list of numbers using a continuation
- The sorting function must take an additional input: the continuation to finish the computation
- Instead of returning a value the function must pass the returned value to the continuation

Continuation-Passing Style

Accumulating Control

- Consider quick sorting a list of numbers using a continuation
- The sorting function must take an additional input: the continuation to finish the computation
- Instead of returning a value the function must pass the returned value to the continuation
- Important to determine the initial value of the continuation
- When an evaluation is started the continuation only needs to return the value of the expression
- The continuation is a function that simply returns its input:

```
(define (endk v) v)
```

Continuation-Passing Style

Accumulating Control

- Consider quick sorting a list of numbers using a continuation
- The sorting function must take an additional input: the continuation to finish the computation
- Instead of returning a value the function must pass the returned value to the continuation
- Important to determine the initial value of the continuation
- When an evaluation is started the continuation only needs to return the value of the expression
- The continuation is a function that simply returns its input:

```
(define (endk v) v)
```

- Consider sorting the empty list
- Given that the list is empty `quick-sorting` returns `'()`

Continuation-Passing Style

Accumulating Control

- Consider quick sorting a list of numbers using a continuation
- The sorting function must take an additional input: the continuation to finish the computation
- Instead of returning a value the function must pass the returned value to the continuation
- Important to determine the initial value of the continuation
- When an evaluation is started the continuation only needs to return the value of the expression
- The continuation is a function that simply returns its input:

```
(define (endk v) v)
```

- Consider sorting the empty list
- Given that the list is empty `quick-sorting` returns `'()`
- To finish the computation using a continuation `'()` is given as input to the continuation:

```
(quick-sorting/k '() endk) = (endk '())
```

Continuation-Passing Style

Accumulating Control

- Now consider sorting an nonempty list:

```
(quick-sorting/k '(5 8 2 9 1 8 4) endk) = ???
```

Continuation-Passing Style

Accumulating Control

- Now consider sorting an nonempty list:

```
(quick-sorting/k '(5 8 2 9 1 8 4) endk) = ???
```

- Intuitively, like to apply the continuation, `endk`, to the result of `append`:

```
(endk (append (quick-sorting '(2 1 4))
              (cons 5 (quick-sorting '(8 9 8)))))
```

- This would defeat our purpose because the expression is not in *tail-form*

Continuation-Passing Style

Accumulating Control

- Now consider sorting an nonempty list:

```
(quick-sorting/k '(5 8 2 9 1 8 4) endk) = ???
```

- Intuitively, like to apply the continuation, `endk`, to the result of `append`:

```
(endk (append (quick-sorting '(2 1 4))
              (cons 5 (quick-sorting '(8 9 8)))))
```

- This would defeat our purpose because the expression is not in *tail-form*
- Refactor to eliminate the function calls in an argument position
-

```
  (quick-sorting/k '(5 8 2 9 1 8 4) endk)
= (quick-sorting/k
    '(2 1 4)
    (λ (ssm)
      (endk (append ssm
                    (cons 5 (quick-sorting '(8 9 8)))))))
```

Continuation-Passing Style

Accumulating Control

- Now consider sorting an nonempty list:

```
(quick-sorting/k '(5 8 2 9 1 8 4) endk) = ???
```

- Intuitively, like to apply the continuation, `endk`, to the result of `append`:

```
(endk (append (quick-sorting '(2 1 4))
              (cons 5 (quick-sorting '(8 9 8)))))
```

- This would defeat our purpose because the expression is not in *tail-form*
- Refactor to eliminate the function calls in an argument position
-

```
  (quick-sorting/k '(5 8 2 9 1 8 4) endk)
= (quick-sorting/k
    '(2 1 4)
    (λ (ssm)
      (endk (append ssm
                    (cons 5 (quick-sorting '(8 9 8)))))))
```

- There is a function call in an argument position
- Need a new continuation to put this call in tail-position:

```
  (quick-sorting/k '(5 8 2 9 1 8 4) endk)
= (quick-sorting/k
    '(2 1 4)
    (λ (ssm)
      (quick-sorting/k '(8 9 8)
                       (λ (sgr) (endk (append ssm (cons 5 sgr)))))))
```

- Control context doesn't grow calling ISL+ built-in functions

Continuation-Passing Style

Accumulating Contro

- The transformation may be done with any given `lon` and any given continuation.

Continuation-Passing Style

Accumulating Contro

- The transformation may be done with any given `lon` and any given continuation.
- This means we can refactor `quick-sorting`:

```
;; lon (lon → lon) → lon Purpose: Sort in nondecreasing order
(define (quick-sorting/k a-lon k)
 (if (empty? a-lon)
     (k '())
     (local [(define SMALLER= (filter (λ (i) (<= i (first a-lon)))
                                      (rest a-lon)))
             (define GREATER  (filter (λ (i) (> i (first a-lon)))
                                      (rest a-lon)))]
             (quick-sorting/k
               SMALLER=
               (λ (ssm) (quick-sorting/k
                          GREATER
                          (λ (sgr)
                           (k (append ssm
                                      (cons (first a-lon) sgr)))))
```

- We say that the function is in *continuation-passing style* (CPS)
- A function is in CPS if the control context is passed as an argument in the form of a continuation and it ends by passing the continuation a value.

Continuation-Passing Style

Accumulating Contro

- The transformation may be done with any given `lon` and any given continuation.
- This means we can refactor `quick-sorting`:

```
;; lon (lon → lon) → lon Purpose: Sort in nondecreasing order
(define (quick-sorting/k a-lon k)
 (if (empty? a-lon)
     (k '())
     (local [(define SMALLER= (filter (λ (i) (<= i (first a-lon)))
                                      (rest a-lon)))
             (define GREATER  (filter (λ (i) (> i (first a-lon)))
                                      (rest a-lon)))]
             (quick-sorting/k
               SMALLER=
               (λ (ssm) (quick-sorting/k
                          GREATER
                          (λ (sgr)
                           (k (append ssm
                                      (cons (first a-lon) sgr)))))
```

- We say that the function is in *continuation-passing style* (CPS)
- A function is in CPS if the control context is passed as an argument in the form of a continuation and it ends by passing the continuation a value.
- `quick-sorting` has been *linearized* (or *serialized*)
- The order in which ops are done is completely specified

Continuation-Passing Style

Accumulating Control

- Natural to ask is if transforming the function to continuation-passing style has a positive impact on running time

Continuation-Passing Style

Accumulating Control

- Natural to ask is if transforming the function to continuation-passing style has a positive impact on running time
- Experiments executed five times:

```
(define L (build-list 100000 (λ (i) (random 1000000))))

(time (quick-sorting L))
(time (quick-sorting/k L endk))
```

Continuation-Passing Style

Accumulating Control

- Natural to ask is if transforming the function to continuation-passing style has a positive impact on running time
- Experiments executed five times:

```
(define L (build-list 100000 (λ (i) (random 1000000))))

(time (quick-sorting L))
(time (quick-sorting/k L endk))
```

-

	quick-sorting	quick-sorting/k
Run 1	1344	1328
Run 2	1328	1312
Run 3	1297	1157
Run 4	1281	1140
Run 5	1250	1093
Average	1300	1206

- Suggests that the continuation-passing style version is faster
- The average's relative difference suggests that `quick-sorting/k` is about 8% faster

Continuation-Passing Style

The CPS Design Recipe

- 1. Add a continuation parameter to each programmer-written function.
  2. If the value returned is the result of a programmer-written function call in tail-position add the given continuation as an argument.
  3. If the value returned is computed without calling a programmer-written function then apply the given continuation to it.
  4. Whenever a function call occurs in an expression, e, in an argument position evaluate the function call using a new one-input continuation whose body is obtained by substituting e with the continuation's parameter.

Continuation-Passing Style

Computing Fibonacci Numbers

-

n	pairs
0	1
1	1
2	2
3	3
4	5
5	8
...	...

- Italian mathematician L. Bonacci calculated growth of a rabbit population
- The number of pairs after `n` months in table

Continuation-Passing Style

Computing Fibonacci Numbers

-

n	pairs
0	1
1	1
2	2
3	3
4	5
5	8
...	...

- Italian mathematician L. Bonacci calculated growth of a rabbit population
- The number of pairs after `n` months in table
- If `n` is less than 2 the number of pairs is 1
- If `n` is greater than or equal to 2 then the number of pairs is equal to the number of pairs after `n - 1` months plus the number of pairs after `n - 2` months

Continuation-Passing Style

Computing Fibonacci Numbers

-

n	pairs
0	1
1	1
2	2
3	3
4	5
5	8
...	...

- Italian mathematician L. Bonacci calculated growth of a rabbit population
- The number of pairs after `n` months in table
- If `n` is less than 2 the number of pairs is 1
- If `n` is greater than or equal to 2 then the number of pairs is equal to the number of pairs after `n - 1` months plus the number of pairs after `n - 2` months
- Mathematical definition:

$$fibonacci(n) = \begin{cases} 1 & \text{if } n < 2 \\ fibonacci(n-1) + fibonacci(n-2) & \text{if } n \geq 2 \end{cases}$$

Continuation-Passing Style

Computing Fibonacci Numbers

-

n	pairs
0	1
1	1
2	2
3	3
4	5
5	8
...	...

- Italian mathematician L. Bonacci calculated growth of a rabbit population
- The number of pairs after `n` months in table
- If `n` is less than 2 the number of pairs is 1
- If `n` is greater than or equal to 2 then the number of pairs is equal to the number of pairs after `n - 1` months plus the number of pairs after `n - 2` months
- Mathematical definition:

$$fibonacci(n) = \begin{cases} 1 & \text{if } n < 2 \\ fibonacci(n-1) + fibonacci(n-2) & \text{if } n \geq 2 \end{cases}$$

- ISL+:

```
;; natnum → natnum Purpose: Compute the nth Fibonacci number
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (sub1 n)) (fib (- n 2)))))
```

Continuation-Passing Style

Computing Fibonacci Numbers

```
(define (fib n k)
  (if (< n 2)
      1
      (+ (fib (sub1 n)) (fib (- n 2)))))
```

Continuation-Passing Style

Computing Fibonacci Numbers

- ```
  (define (fib n k)
    (if (< n 2)
        1
        (+ (fib (sub1 n)) (fib (- n 2)))))
  ```
- ```
  (define (fib/k n k)
    (if (< n 2)
        (k 1)
        (+ (fib/k (sub1 n)) (fib/k (- n 2)))))
  ```

Continuation-Passing Style

Computing Fibonacci Numbers

```
(define (fib/k n k)
  (if (< n 2)
      (k 1)
      (+ (fib/k (sub1 n)) (fib/k (- n 2)))))
```

Continuation-Passing Style

Computing Fibonacci Numbers

- ```
  (define (fib/k n k)
    (if (< n 2)
        (k 1)
        (+ (fib/k (sub1 n)) (fib/k (- n 2)))))
  ```
- ```
  (define (fib/k n k)
    (if (< n 2)
        (k 1)
        (fib/k (sub1 n)
               (λ (f1) (k (+ f1 (fib/k (- n 2))))))))
  ```

Continuation-Passing Style

Computing Fibonacci Numbers

- ```
  (define (fib/k n k)
    (if (< n 2)
        (k 1)
        (fib/k (sub1 n)
               (λ (f1) (k (+ f1 (fib/k (- n 2))))))))
  ```
  ```

Continuation-Passing Style

Computing Fibonacci Numbers

- ```
  (define (fib/k n k)
    (if (< n 2)
        (k 1)
        (fib/k (sub1 n)
               (λ (f1) (k (+ f1 (fib/k (- n 2))))))))
  ```
- ```
  (define (fib/k n k)
    (if (< n 2)
        (k 1)
        (fib/k (sub1 n)
               (λ (f1)
                 (fib/k (- n 2)
                        (λ (f2) (+ f1 f2)))))))
  ```

Continuation-Passing Style

Computing Fibonacci Numbers

- ```
  (define (fib/k n k)
    (if (< n 2)
        (k 1)
        (fib/k (sub1 n)
               (λ (f1)
                 (fib/k (- n 2)
                        (λ (f2) (+ f1 f2)))))))
  ```
  ```

Continuation-Passing Style

Computing Fibonacci Numbers

- ```
  (define (fib/k n k)
    (if (< n 2)
        (k 1)
        (fib/k (sub1 n)
               (λ (f1)
                 (fib/k (- n 2)
                        (λ (f2) (+ f1 f2)))))))
  ```
- ```
  (define (fib/k n k)
    (if (< n 2)
        (k 1)
        (fib/k (sub1 n)
               (λ (f1)
                 (fib/k (- n 2)
                        (λ (f2) (k (+ f1 f2))))))))
  ```

Continuation-Passing Style

The CPS Design Recipe

- Running the following experiments 5 times:

```
(define N 35)

(time (fib N))
(time (fib/k N endk))
```

Continuation-Passing Style

The CPS Design Recipe

- Running the following experiments 5 times:

```
(define N 35)

(time (fib N))
(time (fib/k N endk))
```

-

	fib	fib/k
Run 1	1250	1437
Run 2	1282	1391
Run 3	1265	1437
Run 4	1219	1343
Run 5	1235	1375
Average	1250.2	1396.6

Continuation-Passing Style

The CPS Design Recipe

- Running the following experiments 5 times:

```
(define N 35)

(time (fib N))
(time (fib/k N endk))
```

-

	fib	fib/k
Run 1	1250	1437
Run 2	1282	1391
Run 3	1265	1437
Run 4	1219	1343
Run 5	1235	1375
Average	1250.2	1396.6

- Clearly suggests that `fib` is faster
- The relative difference for the average running time is about -0.1
- `fib` is 10% faster than `fib/k`
- Alas, accumulating control context is not always faster.

Continuation-Passing Style

Beyond the Design Recipe

- Transforming `fib` to `CPS` resulted in a slower program
- Does this mean that `fib` cannot be made faster?
- In general, it is very difficult to answer such a question

Continuation-Passing Style

Beyond the Design Recipe

- Transforming `fib` to CPS resulted in a slower program
- Does this mean that `fib` cannot be made faster?
- In general, it is very difficult to answer such a question
- Observe that in `fib/k` the two continuations receive as input the `(n - 1)`$^{\texttt{th}}$ and `(n - 2)`$^{\texttt{th}}$ Fibonacci numbers
- What if the continuations are represented as two natural numbers instead of two functions?

Continuation-Passing Style

Beyond the Design Recipe

-

n	pairs
0	1
1	1
2	2
3	3
4	5
5	8
...	...

- If the given natural number, `n`, is less than 2 then the function returns 1

Continuation-Passing Style

Beyond the Design Recipe

-

n	pairs
0	1
1	1
2	2
3	3
4	5
5	8
...	...

- If the given natural number, `n`, is less than 2 then the function returns 1
- `[2..n]` needs to be processed from from `low` to `high`
- Use an auxiliary function that takes as input an interval and the two accumulators

Continuation-Passing Style

Beyond the Design Recipe

-

n	pairs
0	1
1	1
2	2
3	3
4	5
5	8
...	...

- If the given natural number, `n`, is less than 2 then the function returns 1
- `[2..n]` needs to be processed from from `low` to `high`
- Use an auxiliary function that takes as input an interval and the two accumulators
- Accumulator invariants:

  `f1 = the (low - 1)`$^{\text{th}}$ `Fibonacci number`
  `f2 = the (low - 2)`$^{\text{th}}$ `Fibonacci number`

- Both accumulators start at 1
- At each step, `f1` is the new value for `f2` and the sum of `f1` and `f2` is the new value for `f1`
- What value ought to be returned when the interval is empty?

Continuation-Passing Style

Beyond the Design Recipe

-

n	pairs
0	1
1	1
2	2
3	3
4	5
5	8
...	...

- If the given natural number, `n`, is less than 2 then the function returns 1
- `[2..n]` needs to be processed from from `low` to `high`
- Use an auxiliary function that takes as input an interval and the two accumulators
- Accumulator invariants:

  `f1 = the (low - 1)`$^{\texttt{th}}$ `Fibonacci number`
  `f2 = the (low - 2)`$^{\texttt{th}}$ `Fibonacci number`

- Both accumulators start at 1
- At each step, `f1` is the new value for `f2` and the sum of `f1` and `f2` is the new value for `f1`
- What value ought to be returned when the interval is empty?
- When the interval is empty `low` is `high + 1 = n + 1`
- According to the invariant this means that `f1` = `n`$^{\texttt{th}}$ Fibonacci number

Continuation-Passing Style

Beyond the Design Recipe

```
;; natnum → natnum
;; Purpose: Compute the nth Fibonacci number
(define (fib-acc n)
  (local [;; [natnum natnum] natnum natnum → natnum
          ;; Purpose: Compute the nth Fibinacci number
          ;; Accumulator Invariants
          ;;  f1 = the (low-1)th Fibonacci number
          ;;  f2 = the (low-2)th Fibobacci number
          (define (fib-helper low high f1 f2)
            (if (< high low)
                f1
                (fib-helper (add1 low) high (+ f1 f2) f1)))]
    (if (< n 2)
        1
        (fib-helper 2 n 1 1))))

;; Tests using sample values for fib-acc
(check-expect (fib-acc 0) FIB0)
(check-expect (fib-acc 1) FIB1)
(check-expect (fib-acc 5) FIB5)
(check-expect (fib-acc 9) FIB9)
(check-expect (fib-acc 4)  5)
(check-expect (fib-acc 6) 13)
```

Continuation-Passing Style

Beyond the Design Recipe

Continuation-Passing Style

Beyond the Design Recipe

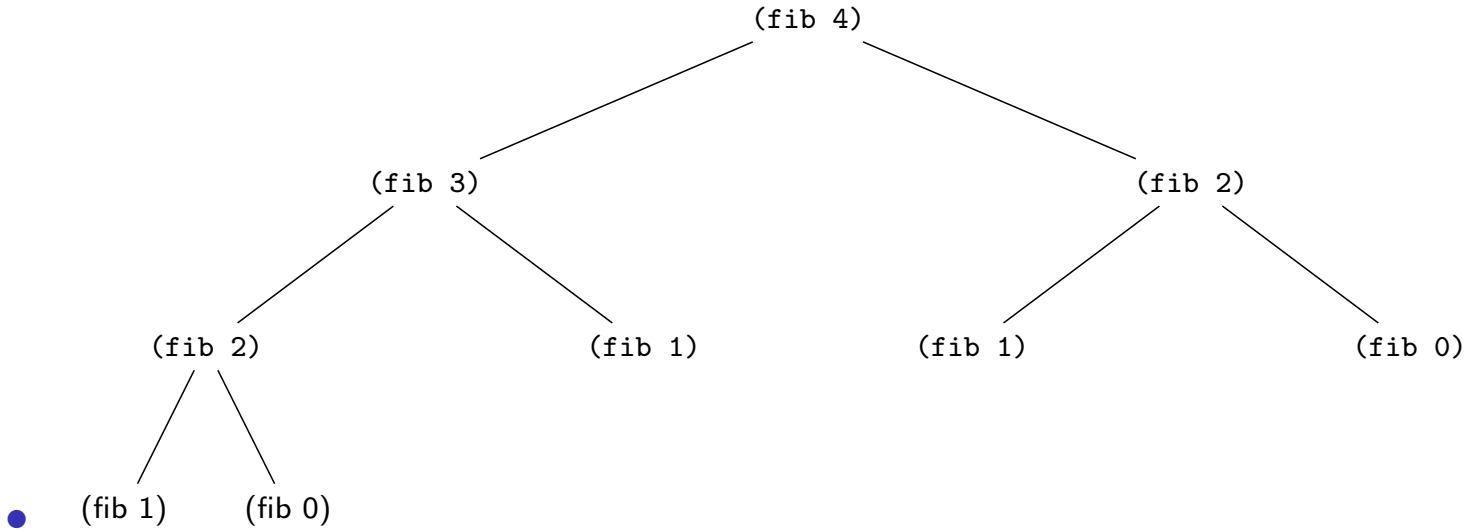
- Run same experiments:

	`fib`	`fib/k`	`fib-acc`
`Run 1`	1250	1437	0
`Run 2`	1282	1391	0
`Run 3`	1265	1437	0
`Run 4`	1219	1343	0
`Run 5`	1235	1375	0
`Average`	1250.2	1396.6	0

- Using two accumulators is significantly faster
- Why is `fib-acc` significantly faster?

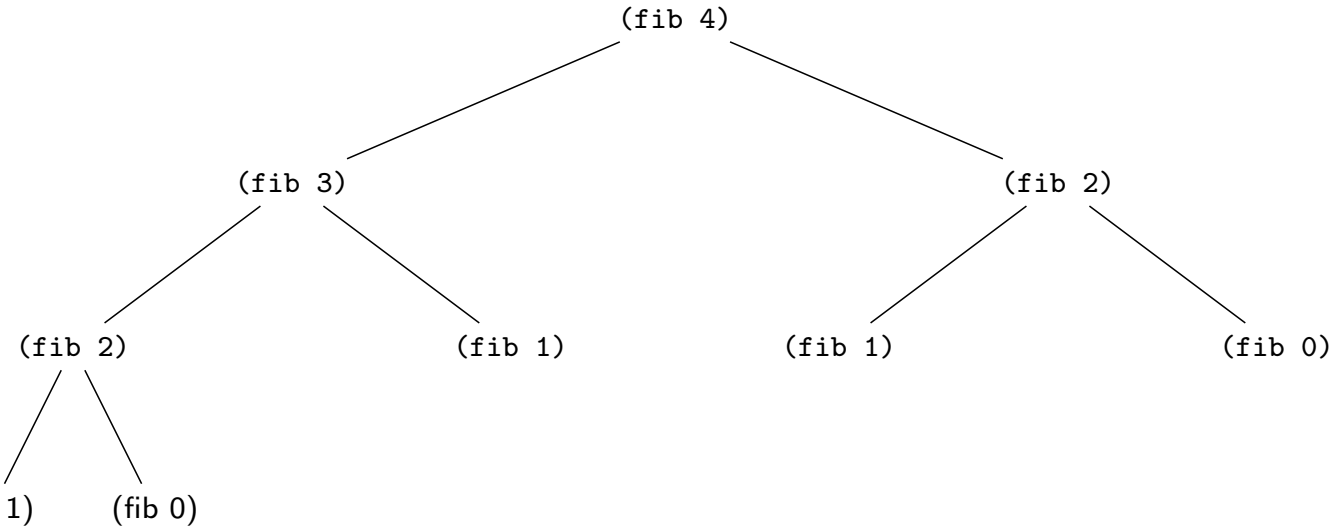
Continuation-Passing Style

Beyond the Design Recipe



Continuation-Passing Style

Beyond the Design Recipe




- ```
  (fib-acc 4) = (fib-helper 2 4 1 1)
              = (fib-helper 3 4 2 1)
              = (fib-helper 4 4 3 2)
              = (fib-helper 5 4 5 3)
  ```
- There are no calls to `fib-helper` with the same `n` value
- No repeated work done
- This is why `fib-acc` is significantly faster.
  ```

Continuation-Passing Style

Lessons

- Writing programs in continuation-passing style is another tool in the problem-solver's toolbox
- Joins other tools like structural recursion, generative recursion, accumulative recursion, the use of a heuristic, the use of a `local`-expression, and code refactoring.
- Like any other tool continuation-passing style will not always the best tool to use

Continuation-Passing Style

Revisiting List Reversal

- Reversing a list benefitted from n accumulator
- Reduced the complexity

Continuation-Passing Style

Revisiting List Reversal

- Reversing a list benefitted from n accumulator
- Reduced the complexity
- Can transforming to `CPS` achieve or surpass the improvement obtained using an accumulator to store the reversed list so far?

Continuation-Passing Style

Revisiting List Reversal

- ```
  ;; (listof X) → (listof X)
  ;; Purpose: Reverse the given list
  (define (rev-lox a-lox)
    (if (empty? a-lox)
        '()
        (append (rev-lox (rest a-lox)) (list (first a-lox)))))
  ```
  ```

Continuation-Passing Style

Revisiting List Reversal

- ```
  ;; (listof X) → (listof X)
  ;; Purpose: Reverse the given list
  (define (rev-lox a-lox)
    (if (empty? a-lox)
        '()
        (append (rev-lox (rest a-lox)) (list (first a-lox)))))
  ```
- ```
  ;; X → X
  ;; Purpose: Return the given value
  (define (endk v) v)

  ;; (listof X) ((listof X) → (listof X)) → (listof X)
  ;; Purpose: Reverse the given list
  (define (rev-lox/k a-lox k)
    (if (empty? a-lox)
        (k '())
        (rev-lox/k
          (rest a-lox)
            (λ (revr)
              (k (append revr (list (first a-lox))))))))
  ```

Continuation-Passing Style

Revisiting List Reversal

- `(define L (build-list 50000 (λ (i) (random 100000))))`
- Execute five times: `(time (rev/k L endk))`

Continuation-Passing Style

Revisiting List Reversal

- `(define L (build-list 50000 (λ (i) (random 100000))))`
- Execute five times: `(time (rev/k L endk))`
-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
rev-lox	12172	12313	12187	12562	11829	12212.6
rev-lox2	15	15	15	0	0	9
rev-lox/k	12485	12640	12875	13032	10313	12269

- `rev-lox/k`'s performance is about the same as the performance as `rev-lox`
- `rev-lox` may be slightly faster but for `Run 5` `rev-lox/k` performed better
- There is not a clear winner among them

Continuation-Passing Style

Revisiting List Reversal

- `(define L (build-list 50000 (λ (i) (random 100000))))`
- Execute five times: `(time (rev/k L endk))`
-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
rev-lox	12172	12313	12187	12562	11829	12212.6
rev-lox2	15	15	15	0	0	9
rev-lox/k	12485	12640	12875	13032	10313	12269

- `rev-lox/k`'s performance is about the same as the performance as `rev-lox`
- `rev-lox` may be slightly faster but for `Run 5` `rev-lox/k` performed better
- There is not a clear winner among them
- `rev-lox2` is significantly faster than `rev-lox/k`
- How are these empirical observations explained?

Continuation-Passing Style

Revisiting List Reversal

- `(define L (build-list 50000 (λ (i) (random 100000))))`
- Execute five times: `(time (rev/k L endk))`
-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
rev-lox	12172	12313	12187	12562	11829	12212.6
rev-lox2	15	15	15	0	0	9
rev-lox/k	12485	12640	12875	13032	10313	12269

- `rev-lox/k`'s performance is about the same as the performance as `rev-lox`
- `rev-lox` may be slightly faster but for `Run 5` `rev-lox/k` performed better
- There is not a clear winner among them
- `rev-lox2` is significantly faster than `rev-lox/k`
- How are these empirical observations explained?
- The transformation to `CPS` does not change the algorithm
- Both `rev-lox` and `rev-lox/k` use `append` making the abstract running time $O(n^2)$
- In contrast, `rev-lox2` uses `cons` to accumulate the reversed list, thus, making its complexity $O(n)$.

Continuation-Passing Style

Revisiting List Reversal

- `(define L (build-list 50000 (λ (i) (random 100000))))`
- Execute five times: `(time (rev/k L endk))`
-

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
rev-lox	12172	12313	12187	12562	11829	12212.6
rev-lox2	15	15	15	0	0	9
rev-lox/k	12485	12640	12875	13032	10313	12269

- `rev-lox/k`'s performance is about the same as the performance as `rev-lox`
- `rev-lox` may be slightly faster but for `Run 5` `rev-lox/k` performed better
- There is not a clear winner among them
- `rev-lox2` is significantly faster than `rev-lox/k`
- How are these empirical observations explained?
- The transformation to `CPS` does not change the algorithm
- Both `rev-lox` and `rev-lox/k` use `append` making the abstract running time $O(n^2)$
- In contrast, `rev-lox2` uses `cons` to accumulate the reversed list, thus, making its complexity $O(n)$.
- Use an accumulator to eliminate delayed operations and build the result one step at a time
- If it is not clear how to use an accumulator to eliminate a delayed operation then try transforming the program to `CPS`
- Empirical experimentation is necessary to determine if rewriting using an accumulator or `CPS` yields any gains in performance

Continuation-Passing Style

HOMEWORK

- Problems: 1, 2, 4, 6, 9