# Part I: Introduction

Marco T. Morazán

Seton Hall University

# Outline

# The Science of Problem Solving

- Continues the study of problem solving and program design that began in *Animated Problem Solving*
- We move beyond structural recursion to other forms of recursion that require insights into a problem beyond what may be suggested by the structure of the data

# The Science of Problem Solving

- Continues the study of problem solving and program design that began in *Animated Problem Solving*
- We move beyond structural recursion to other forms of recursion that require insights into a problem beyond what may be suggested by the structure of the data
- The basic concepts you need to be familiar with are:
  - Variables
  - Functions
  - Design Recipe
  - Abstract Functions (generic programming)
  - Abstract Running Time

# The Science of Problem Solving

## The Design Recipe

1. Perform data analysis and outline a design idea.

# The Science of Problem Solving

## The Design Recipe

1. Perform data analysis and outline a design idea.
2. Define constants for the value of sample expressions.

# The Science of Problem Solving

## The Design Recipe

1. Perform data analysis and outline a design idea.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.

# The Science of Problem Solving

## The Design Recipe

1. Perform data analysis and outline a design idea.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature and purpose.

# The Science of Problem Solving

## The Design Recipe

1. Perform data analysis and outline a design idea.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature and purpose.
5. Write the function's header.

# The Science of Problem Solving

## The Design Recipe

1. Perform data analysis and outline a design idea.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature and purpose.
5. Write the function's header.
6. Write tests.

# The Science of Problem Solving

## The Design Recipe

1. Perform data analysis and outline a design idea.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature and purpose.
5. Write the function's header.
6. Write tests.
7. Write the function's body.

# The Science of Problem Solving

## The Design Recipe

1. Perform data analysis and outline a design idea.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature and purpose.
5. Write the function's header.
6. Write tests.
7. Write the function's body.
8. Run the tests and, if necessary, redesign.

# The Science of Problem Solving

## Doubling a (listof number):

- Data of arbitrary size:

```
;; A list of numbers, lon, is either:
;;  1. empty     2. (cons number lon)
```

# The Science of Problem Solving

## Doubling a (listof number):

- Data of arbitrary size:

```
;; A list of numbers, lon, is either:
;;  1. empty     2. (cons number lon)
```

- 
```
;; Sample lon
(define ELON '()) (define LON1 '(2 3 4 5)) (define LON2 '(8 -2 0))
```

# The Science of Problem Solving

## Doubling a (listof number):

- Data of arbitrary size:

```
;; A list of numbers, lon, is either:
;;  1. empty     2. (cons number lon)
```

- 
```
;; Sample lon
(define ELON '()) (define LON1 '(2 3 4 5)) (define LON2 '(8 -2 0))
```

- Function Template:

```
;; lon ... → ... Purpose:
(define (f-on-lon a-lon ...)
  (if (empty? a-lon)
      ...
      ...(first a-lon)...(f-on-lon (rest a-lon))))
;; Sample expressions for f-on-lon
(define ELON-VAL ...) (define LON1-VAL ...) (define LONN-VAL ...)
;; Tests using sample computations for f-on-lon
(check-expect (f-on-lon ELON ...) ...)
(check-expect (f-on-lon LON1 ...) ...) ...
(check-expect (f-on-lon LON2 ...) ...) ...
;; Tests using sample values for f-on-lon
(check-expect (f-on-lon ... ...) ...) ... |#
```

# The Science of Problem Solving

## Doubling a (listof number):

- Data of arbitrary size:

```
;; A list of numbers, lon, is either:
;;  1. empty     2. (cons number lon)
```

- 
```
;; Sample lon
(define ELON '()) (define LON1 '(2 3 4 5)) (define LON2 '(8 -2 0))
```

- Function Template:

```
;; lon ... → ... Purpose:
(define (f-on-lon a-lon ...)
  (if (empty? a-lon)
      ...
      ...(first a-lon)...(f-on-lon (rest a-lon))))
;; Sample expressions for f-on-lon
(define ELON-VAL ...) (define LON1-VAL ...) (define LONN-VAL ...)
;; Tests using sample computations for f-on-lon
(check-expect (f-on-lon ELON ...) ...)
(check-expect (f-on-lon LON1 ...) ...) ...
(check-expect (f-on-lon LON2 ...) ...) ...
;; Tests using sample values for f-on-lon
(check-expect (f-on-lon ... ...) ...) ... |#
```

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

# The Science of Problem Solving

## Doubling a (listof number):

- Data of arbitrary size:

```
;; A list of numbers, lon, is either:
;;  1. empty     2. (cons number lon)
```

- 
```
;; Sample lon
(define ELON '()) (define LON1 '(2 3 4 5)) (define LON2 '(8 -2 0))
```

- Function Template:

```
;; lon ... → ... Purpose:
(define (f-on-lon a-lon ...)
  (if (empty? a-lon)
      ...
      ...(first a-lon)...(f-on-lon (rest a-lon))))
;; Sample expressions for f-on-lon
(define ELON-VAL ...) (define LON1-VAL ...) (define LONN-VAL ...)
;; Tests using sample computations for f-on-lon
(check-expect (f-on-lon ELON ...) ...)
(check-expect (f-on-lon LON1 ...) ...) ...
(check-expect (f-on-lon LON2 ...) ...) ...
;; Tests using sample values for f-on-lon
(check-expect (f-on-lon ... ...) ...) ... |#
```

- If the list is empty the answer is the empty list
- If the `lon` is not empty then cons the doubling of the first element of the given `lon` and recursively double the rest of the given list.

# The Science of Problem Solving

## Doubling a (listof number): Sample Expressions

- ```
  ;; Sample expressions for lon-double
  (define DOUBLE-ELON '())
  (define DOUBLE-LON1 (cons (* 2 (first LON1))
                            (lon-double (rest LON1))))
  (define DOUBLE-LON2 (cons (* 2 (first LON2))
                            (lon-double (rest LON2))))
  ```
  ```

# The Science of Problem Solving

## Doubling a (listof number): Sample Expressions

- ```
  ;; Sample expressions for lon-double
  (define DOUBLE-ELON '())
  (define DOUBLE-LON1 (cons (* 2 (first LON1))
                            (lon-double (rest LON1))))
  (define DOUBLE-LON2 (cons (* 2 (first LON2))
                            (lon-double (rest LON2))))
  ```
- One difference: a `lon`
  ```

# The Science of Problem Solving

## Doubling a (listof number): The Function

- ```
  ;; lon → lon
  ;; Purpose: Double the numbers in the given list of numbers
  (define (lon-double a-lon)
  ```
  ```

# The Science of Problem Solving

## Doubling a (listof number): The Function

- ```
  ;; lon → lon
  ;; Purpose: Double the numbers in the given list of numbers
  (define (lon-double a-lon)
  ```

- ```
  ;; Tests using sample computations
  (check-expect (lon-double ELON) DOUBLE-ELON)
  (check-expect (lon-double LON1) DOUBLE-LON1)
  (check-expect (lon-double LON2) DOUBLE-LON2)
  ```

# The Science of Problem Solving

## Doubling a (listof number): The Function

```
;; lon → lon
;; Purpose: Double the numbers in the given list of numbers
(define (lon-double a-lon)
```

```
;; Tests using sample computations
(check-expect (lon-double ELON) DOUBLE-ELON)
(check-expect (lon-double LON1) DOUBLE-LON1)
(check-expect (lon-double LON2) DOUBLE-LON2)
```

```
;; Tests using sample values
(check-expect (lon-double '(-8 -10)) '(-16 -20))
(check-expect (lon-double '(23 -850 209)) '(46 -1700 418))
```

# The Science of Problem Solving

## Doubling a (listof number): The Function

```
;; lon → lon
;; Purpose: Double the numbers in the given list of numbers
(define (lon-double a-lon)
```

```
(if (empty? a-lon)
    '()
    (cons (* 2 (first a-lon)) (lon-double (rest a-lon)))))
```

```
;; Tests using sample computations
(check-expect (lon-double ELON) DOUBLE-ELON)
(check-expect (lon-double LON1) DOUBLE-LON1)
(check-expect (lon-double LON2) DOUBLE-LON2)
```

```
;; Tests using sample values
(check-expect (lon-double '(-8 -10)) '(-16 -20))
(check-expect (lon-double '(23 -850 209)) '(46 -1700 418))
```

# The Science of Problem Solving

## HOMEWORK

- Problems: 1-7

# The Science of Problem Solving

## Code Refactoring

- Code refactoring restructures existing functions without changing their external behavior

# The Science of Problem Solving

## Code Refactoring

- Code refactoring restructures existing functions without changing their external behavior
- `lon-double` exploits structural recursion
- The given list is traversed one element at a time and ach element is doubled
- A list of the doubled elements is created

# The Science of Problem Solving

## Code Refactoring

- Code refactoring restructures existing functions without changing their external behavior
- `lon-double` exploits structural recursion
- The given list is traversed one element at a time and ach element is doubled
- A list of the doubled elements is created
- The same function is applied to each element of the list
- Recall `map` traverses a given list and applies a given function to every element to create a list of the function application results

# The Science of Problem Solving

## Code Refactoring

- Code refactoring restructures existing functions without changing their external behavior
- `lon-double` exploits structural recursion
- The given list is traversed one element at a time and ach element is doubled
- A list of the doubled elements is created
- The same function is applied to each element of the list
- Recall `map` traverses a given list and applies a given function to every element to create a list of the function application results
- ```
  ;; lon → lon
  ;; Purpose: Double the numbers in the given list of numbers
  (define (lon-double a-lon)
    (map (λ (a-num) (* 2 a-num)) a-lon))
  ```
  ```

# The Science of Problem Solving

## Code Refactoring

- May be refactored to eliminate the use of list selectors:

```
;; lon → lon
;; Purpose: Double the numbers in the given list of numbers
(define (lon-double a-lon)
  (match a-lon
    ['() '()]
    [(cons frst rst) (cons (* 2 frst) (lon-double rst))]))
```

# The Science of Problem Solving

## Code Refactoring

- May be refactored to eliminate the use of list selectors:

```
;; lon → lon
;; Purpose: Double the numbers in the given list of numbers
(define (lon-double a-lon)
  (match a-lon
    ['() '()]
    [(cons frst rst) (cons (* 2 frst) (lon-double rst))]))
```

- Refactor `lon-double` to use a `for`-loop:

```
;; lon → lon
;; Purpose: Double the numbers in the given list of numbers
(define (lon-double a-lon)
  (for/list ([a-num a-lon]) (* 2 a-num)))
```

# The Science of Problem Solving

## HOMEWORK

- Problems: 8-12

# The Science of Problem Solving

## Abstract Running Time

- Abstract running time relates a function's input size with the number of *abstract steps* needed to solve the problem

# The Science of Problem Solving

## Abstract Running Time

- Abstract running time relates a function's input size with the number of *abstract steps* needed to solve the problem
- Count the number of general steps taken like the number of function calls or the number of comparisons made

# The Science of Problem Solving

## Abstract Running Time

- Abstract running time relates a function's input size with the number of *abstract steps* needed to solve the problem
- Count the number of general steps taken like the number of function calls or the number of comparisons made
- How are abstract operations counted?

# The Science of Problem Solving

## Abstract Running Time

- Consider the number of function calls performed for `double-lon`
- If the function is called with the empty list only one function call is made

# The Science of Problem Solving

## Abstract Running Time

- Consider the number of function calls performed for `double-lon`
- If the function is called with the empty list only one function call is made
- When the length of the list is 1 we can observe that two function calls are made

# The Science of Problem Solving

## Abstract Running Time

- Consider the number of function calls performed for `double-lon`
- If the function is called with the empty list only one function call is made
- When the length of the list is 1 we can observe that two function calls are made
- For a list of length 2 the number of function calls is 3

# The Science of Problem Solving

## Abstract Running Time

- Consider the number of function calls performed for `double-lon`
- If the function is called with the empty list only one function call is made
- When the length of the list is 1 we can observe that two function calls are made
- For a list of length 2 the number of function calls is 3
- In general:

```
number-function-calls(L) = (length L) + 1.
```

- If the length of the list is `n` then the number of function calls is `n+1`.

# The Science of Problem Solving

## Abstract Running Time

- Consider the number of function calls performed for `double-lon`
- If the function is called with the empty list only one function call is made
- When the length of the list is 1 we can observe that two function calls are made
- For a list of length 2 the number of function calls is 3
- In general:

```
number-function-calls(L) = (length L) + 1.
```

- If the length of the list is `n` then the number of function calls is `n+1`.
- A program's abstract running time or *complexity* is usually written using Big `O` notation

# The Science of Problem Solving

## Abstract Running Time

- Consider the number of function calls performed for `double-lon`
- If the function is called with the empty list only one function call is made
- When the length of the list is 1 we can observe that two function calls are made
- For a list of length 2 the number of function calls is 3
- In general:

```
number-function-calls(L) = (length L) + 1.
```

- If the length of the list is `n` then the number of function calls is `n+1`.
- A program's abstract running time or *complexity* is usually written using Big `O` notation
- The complexity of `lon-double` is `O(n)`

# The Science of Problem Solving

## Abstract Running Time

- If a program is `O(n)` we say that it is linear
- If the size of the input is doubled then we expect the number of abstract operations performed to be doubled

# The Science of Problem Solving

## Abstract Running Time

- If a program is `O(n)` we say that it is linear
- If the size of the input is doubled then we expect the number of abstract operations performed to be doubled
- If a program is $O(n^2)$ and the size of the input is double we expect the number of abstract operations to be quadrupled.

# The Science of Problem Solving

## Abstract Running Time

- If a program is `O(n)` we say that it is linear
- If the size of the input is doubled then we expect the number of abstract operations performed to be doubled
- If a program is $O(n^2)$ and the size of the input is double we expect the number of abstract operations to be quadrupled.
- It is worth noting that programs may have an exponential running time like $O(2^n)$
- If a program has exponential complexity it is unlikely to be useful

# The Science of Problem Solving

## Abstract Running Time

- If a program is `O(n)` we say that it is linear
- If the size of the input is doubled then we expect the number of abstract operations performed to be doubled
- If a program is $O(n^2)$and the size of the input is double we expect the number of abstract operations to be quadrupled.
- It is worth noting that programs may have an exponential running time like $O(2^n)$
- If a program has exponential complexity it is unlikely to be useful
- Consider a list-processing function that is $O(2^n)$
- For a relatively small list with only 30 elements the expected number of computer operations is proportional to over a billion operations
- If the size of the list is doubled to 60 the number of computer operations is proportional to over a quintillion operations

# The Science of Problem Solving

## Abstract Running Time

- If a program is $O(n)$ we say that it is linear
- If the size of the input is doubled then we expect the number of abstract operations performed to be doubled
- If a program is $O(n^2)$ and the size of the input is double we expect the number of abstract operations to be quadrupled.
- It is worth noting that programs may have an exponential running time like $O(2^n)$
- If a program has exponential complexity it is unlikely to be useful
- Consider a list-processing function that is $O(2^n)$
- For a relatively small list with only 30 elements the expected number of computer operations is proportional to over a billion operations
- If the size of the list is doubled to 60 the number of computer operations is proportional to over a quintillion operations
- A computer that is capable of executing 2 billion (computer) operations per second would roughly take over 160 thousand hours (i.e., over 18 years) to process a list with 60 element
- You can easily see why programs that have an exponential abstract running time are not practical

# The Science of Problem Solving

## Abstract Running Time

- The function for the abstract running time may have more than one component that depends on the size of the input
- `f(n) = 3n^2 + 4n + 10`
- What is the complexity of such a program? `O(n^2 + n)`?

# The Science of Problem Solving

## Abstract Running Time

- The function for the abstract running time may have more than one component that depends on the size of the input
- `f(n) = 3n^2 + 4n + 10`
- What is the complexity of such a program? `O(n^2 + n)`?
- 

| List Length | n | $n^2$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 4 |
| 3 | 3 | 9 |
| 4 | 4 | 16 |
| 5 | 5 | 25 |
| ⋮ | ⋮ | ⋮ |
| 1000000 | 1000000 | 1000000000000 |

- $n^2$ grows faster that `n`
- The value of $n^2$ is much larger and `n` grows

# The Science of Problem Solving

## Abstract Running Time

- The function for the abstract running time may have more than one component that depends on the size of the input
- `f(n) = 3n^2 + 4n + 10`
- What is the complexity of such a program? `O(n^2 + n)`?
- 

| List Length | n | $n^2$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 4 |
| 3 | 3 | 9 |
| 4 | 4 | 16 |
| 5 | 5 | 25 |
| ⋮ | ⋮ | ⋮ |
| 1000000 | 1000000 | 1000000000000 |

- $n^2$ grows faster that `n`
- The value of $n^2$ is much larger and `n` grows
- This means that the number of operations is proportional to $n^2$ and not to `n`
- The complexity is `O(n^2)`
- In general, the complexity of the program is the fastest growing component of its abstract running time function

# The Science of Problem Solving

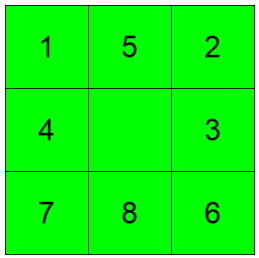
## HOMEWORK

- Problems: 12-15

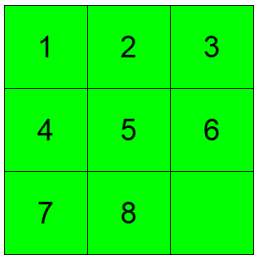
# N-Puzzle Problem

- Goal this semester: implement an N-puzzle video game
- Interesting because it is a problem that allows us to explore Computer Science topics that are usually relegated to advanced courses

# N-Puzzle Problem

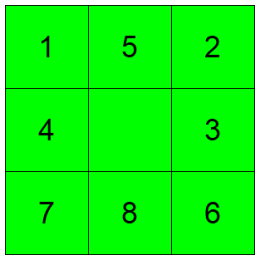
- Goal this semester: implement an N-puzzle video game
- Interesting because it is a problem that allows us to explore Computer Science topics that are usually relegated to advanced courses
- Sliding puzzle that has N tile positions, where N+1 is a square
- 


(a) An Unsolved 8-Puzzle.

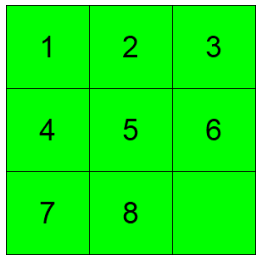



(b) A Solved 8-Puzzle.

# N-Puzzle Problem

- Goal this semester: implement an N-puzzle video game
- Interesting because it is a problem that allows us to explore Computer Science topics that are usually relegated to advanced courses
- Sliding puzzle that has N tile positions, where N+1 is a square
- 

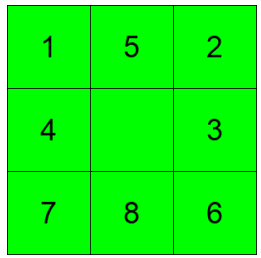

(a) An Unsolved 8-Puzzle.



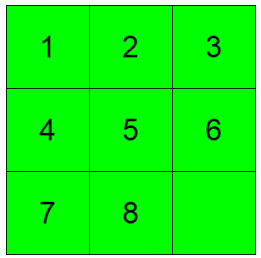

(b) A Solved 8-Puzzle.

- Player may feel she needs help to solve the puzzle
- The game offers a mechanism for the player to ask for help
- Help comes in the form of a single move

# N-Puzzle Problem

- Goal this semester: implement an N-puzzle video game
- Interesting because it is a problem that allows us to explore Computer Science topics that are usually relegated to advanced courses
- Sliding puzzle that has N tile positions, where N+1 is a square
- 


(a) An Unsolved 8-Puzzle.




(b) A Solved 8-Puzzle.

- Player may feel she needs help to solve the puzzle
- The game offers a mechanism for the player to ask for help
- Help comes in the form of a single move
- How does the program decide what move to make?
- How do we make sure that providing help does not make the program slow?

# N-Puzzle Problem

## The `world` and the `run` Function

- The `run` function evaluates a `big-bang` expression
- Decide how to represent the world: What changes?

# N-Puzzle Problem

## The `world` and the `run` Function

- The `run` function evaluates a `big-bang` expression
- Decide how to represent the world: What changes?
- The `world` is the value of the board.
- How can the value of the board be represented?

# N-Puzzle Problem

## The `world` and the `run` Function

- The `run` function evaluates a `big-bang` expression
- Decide how to represent the world: What changes?
- The `world` is the value of the board.
- How can the value of the board be represented?
- 
```
;; A board position, bpos, is an integer in [0..8]
;; Sample bpos
(define TRCRNR 0)  ;; top right corner ...
#| ;; bpos ... → ...   Purpose:
     (define (f-on-bpos a-bpos ...)
       (cond [(= a-bpos 0) ...]
             [(= a-bpos 1) ...]
             [(= a-bpos 2) ...]
             [(= a-bpos 3) ...]
             [(= a-bpos 4) ...]
             [(= a-bpos 5) ...]
             [(= a-bpos 6) ...]
             [(= a-bpos 7) ...]
             [else ...]))
;; Sample expressions for f-on-bpos
(define TRCRNR-VAL ...) ...
;; Tests using sample computations for f-on-bpos
(check-expect (f-on-bpos TRCRNR ...) ...) ...
;; Tests using sample values for f-on-bpos
(check-expect (f-on-bpos ... ...) ...) ...  |#
```

# N-Puzzle Problem

- Each board position may contain a tile or be empty
- A tile may be represented by the number of on the tile: 1–8
- The empty tile space as 0

# N-Puzzle Problem

- Each board position may contain a tile or be empty
- A tile may be represented by the number of on the tile: 1–8
- The empty tile space as 0
- ```
  ;; A tile value, tval, is an integer in [0..8]
  ;; Sample tval
  (define BLNK 0) ...
  #| ;; tval ... → ...
     ;; Purpose:
     (define (f-on-tval a-tval ...)
     (cond [(= a-tval 0) ...]
           [(= a-tval 1) ...]
           [(= a-tval 2) ...]
           [(= a-tval 3) ...]
           [(= a-tval 4) ...]
           [(= a-tval 5) ...]
           [(= a-tval 6) ...]
           [(= a-tval 7) ...]
           [else ...]))
  ;; Sample expressions for f-on-tval
  (define BLNK-VAL ...) ...
  ;; Tests using sample computations for f-on-tval
  (check-expect (f-on-tval BLNK ...) ...) ...
  ;; Tests using sample values for f-on-tval
  (check-expect (f-on-tval ... ...) ...) ...    |#
  ```
- Do not erroneously conclude that bpos and tval are the same
  ```

# N-Puzzle Problem

## The `world` and the `run` Function

- ```
  ;; A world is a structure:
  ;;   (make-world tval tval tval tval tval tval tval tval tval).
  (define-struct world (t0 t1 t2 t3 t4 t5 t6 t7 t8))
  ```
  ```

# N-Puzzle Problem

## The `world` and the `run` Function

- ```
  ;; A world is a structure:
  ;;   (make-world tval tval tval tval tval tval tval tval tval).
  (define-struct world (t0 t1 t2 t3 t4 t5 t6 t7 t8))
  ```
- ```
  ;; Sample worlds
  (define WIN    (make-world 1 2 3 4 5 6 7 8 0))
  (define A-WRLD (make-world 1 5 2 4 0 3 7 8 6))
  #| ;; world ... → ...   Purpose:
     (define (f-on-world a-world ...)
       (...(f-on-tval (world-t0)) (f-on-tval (world-t1))
        ...(f-on-tval (world-t2)) (f-on-tval (world-t3))
        ...(f-on-tval (world-t4)) (f-on-tval (world-t5))
        ...(f-on-tval (world-t6)) (f-on-tval (world-t7))
        ...(f-on-tval (world-t8)) ...))
  ;; Sample expressions for f-on-world
  (define WIN-VAL ...)
  (define A-WORLD-VAL ...)
  ;; Tests using sample computations for f-on-world
  (check-expect (f-on-world WIN ...)     WIN-VAL)
  (check-expect (f-on-world A-WORLD ...) A-WORLD-VAL) ...
  ;; Tests using sample values for f-on-world
  (check-expect (f-on-world ... ...) ...) ...      |#
  ```

# N-Puzzle Problem

## The `world` and the `run` Function

- `run` takes as input a name, represented as a symbol or a string, to be placed on the game's window frame
- Returns a `world`

# N-Puzzle Problem

## The `world` and the `run` Function

- `run` takes as input a name, represented as a symbol or a string, to be placed on the game's window frame
- Returns a `world`
- Handlers:
  - `draw-world`
  - `game-over?`
  - `draw-last-world`
  - `process-key`

# N-Puzzle Problem

## The `world` and the `run` Function

- `run` takes as input a name, represented as a symbol or a string, to be placed on the game's window frame
- Returns a `world`
- Handlers:
  - `draw-world`
  - `game-over?`
  - `draw-last-world`
  - `process-key`
- `run` function:

```
;; A name is either a symbol or a string

;; name → world
;; Purpose: Run the 8-puzzle game
(define (run a-name)
  (big-bang
    A-WRLD
    (on-draw draw-world)
    (on-key process-key)
    (stop-when game-over? draw-last-world)
    (name a-name)))
```

# N-Puzzle Problem

## Useful Constants

```
(define TILE-LEN 100)
(define TILE-COLOR 'green)
(define BORD-COLOR 'black)
(define TEXT-COLOR 'black)
(define TEXT-SIZE 36)
(define SCENE-LEN (* 3 TILE-LEN))
(define e-scene (empty-scene SCENE-LEN SCENE-LEN))
```

# N-Puzzle Problem

## Useful Constants

- ```
  (define TILE-LEN 100)
  (define TILE-COLOR 'green)
  (define BORD-COLOR 'black)
  (define TEXT-COLOR 'black)
  (define TEXT-SIZE 36)
  (define SCENE-LEN (* 3 TILE-LEN))
  (define e-scene (empty-scene SCENE-LEN SCENE-LEN))
  ```
- ```
  (define T0 (overlay (square TILE-LEN 'outline TEXT-COLOR)
                      (square TILE-LEN 'solid   TILE-COLOR)))

  (define T1 (overlay (text (number->string 1)
                            TEXT-SIZE
                            TEXT-COLOR)
                      (square TILE-LEN 'outline TEXT-COLOR)
                      (square TILE-LEN 'solid   TILE-COLOR)))

  (define T2 (overlay (text (number->string 2)
                            TEXT-SIZE
                            TEXT-COLOR)
                      (square TILE-LEN 'outline TEXT-COLOR)
                      (square TILE-LEN 'solid   TILE-COLOR)))
  ```

Screaming for abstraction!

# N-Puzzle Problem

## Useful Constants

- ```
  ;; tval → image
  ;; Purpose: Return the tile image for the given tile number
  (define (make-tile-img a-tval)
    (if (= a-tval 0)
        (overlay (square TILE-LEN 'outline BORD-COLOR)
                 (square TILE-LEN 'solid TILE-COLOR))
        (overlay (text (number->string a-tval) TEXT-SIZE TEXT-COLOR)
                 (square TILE-LEN 'outline BORD-COLOR)
                 (square TILE-LEN 'solid TILE-COLOR))))
  ```
  ```

# N-Puzzle Problem

## Useful Constants

- ```
  ;; tval → image
  ;; Purpose: Return the tile image for the given tile number
  (define (make-tile-img a-tval)
    (if (= a-tval 0)
        (overlay (square TILE-LEN 'outline BORD-COLOR)
                 (square TILE-LEN 'solid TILE-COLOR))
        (overlay (text (number->string a-tval) TEXT-SIZE TEXT-COLOR)
                 (square TILE-LEN 'outline BORD-COLOR)
                 (square TILE-LEN 'solid TILE-COLOR))))
  ```
- ```
  (define T3 (make-tile-img 3))

  (define T4 (make-tile-img 4))

  (define T5 (make-tile-img 5))

  (define T6 (make-tile-img 6))

  (define T7 (make-tile-img 7))

  (define T8 (make-tile-img 8))
  ```

# N-Puzzle Problem

## The `draw-world` Handler

- Think of a board image as having three rows above each other

# N-Puzzle Problem

## The `draw-world` Handler

- Think of a board image as having three rows above each other
- Sample expressions:

```
(above (beside (tile-img (world-t0 WIN))
               (tile-img (world-t1 WIN))
               (tile-img (world-t2 WIN)))
       (beside (tile-img (world-t3 WIN))
               (tile-img (world-t4 WIN))
               (tile-img (world-t5 WIN)))
       (beside (tile-img (world-t6 WIN))
               (tile-img (world-t7 WIN))
               (tile-img (world-t8 WIN))))

(above (beside (tile-img (world-t0 A-WRLD))
               (tile-img (world-t1 A-WRLD))
               (tile-img (world-t2 A-WRLD)))
       (beside (tile-img (world-t3 A-WRLD))
               (tile-img (world-t4 A-WRLD))
               (tile-img (world-t5 A-WRLD)))
       (beside (tile-img (world-t6 A-WRLD))
               (tile-img (world-t7 A-WRLD))
               (tile-img (world-t8 A-WRLD))))
```

- `tile-img` returns the tile image for the given `tval`
- Only difference between the sample expressions is the world drawn

# N-Puzzle Problem

## The `world` and the `run` Function

- `;; draw-world: world → image Purpose: Draw world in empty-scene`
  `(define (draw-world a-world)`

# N-Puzzle Problem

## The `world` and the `run` Function

- `;; draw-world: world → image Purpose: Draw world in empty-scene`
  `(define (draw-world a-world)`

- Tests in the textbook

# N-Puzzle Problem

## The `world` and the `run` Function

- `;; draw-world: world → image Purpose: Draw world in empty-scene`
  `(define (draw-world a-world)`

- ```
      (above (beside (tile-img (world-t0 a-world))
                     (tile-img (world-t1 a-world))
                     (tile-img (world-t2 a-world)))
             (beside (tile-img (world-t3 a-world))
                     (tile-img (world-t4 a-world))
                     (tile-img (world-t5 a-world)))
             (beside (tile-img (world-t6 a-world))
                     (tile-img (world-t7 a-world))
                     (tile-img (world-t8 a-world)))))
  ```

- Tests in the textbook
  ```

# N-Puzzle Problem

## The `world` and the `run` Function

- `;; draw-world: world → image Purpose: Draw world in empty-scene`
  `(define (draw-world a-world)`
- `  (local [;; tval → image Purpose: Return tval image`
  `(define (tile-img a-tval)`
  `(cond [(= a-tval 0) T0]`
  `[(= a-tval 1) T1]`
  `[(= a-tval 2) T2]`
  `[(= a-tval 3) T3]`
  `[(= a-tval 4) T4]`
  `[(= a-tval 5) T5]`
  `[(= a-tval 6) T6]`
  `[(= a-tval 7) T7]`
  `[else T8]))]`
- `  (above (beside (tile-img (world-t0 a-world))`
  `(tile-img (world-t1 a-world))`
  `(tile-img (world-t2 a-world)))`
  `(beside (tile-img (world-t3 a-world))`
  `(tile-img (world-t4 a-world))`
  `(tile-img (world-t5 a-world)))`
  `(beside (tile-img (world-t6 a-world))`
  `(tile-img (world-t7 a-world))`
  `(tile-img (world-t8 a-world)))))))`
- Tests in the textbook

# N-Puzzle Problem

## The game-over? Handler

- ```
  ;; Sample expressions for game-over?
  (define WIN-OVER (equal? WIN WIN))
  (define A-WRLD-OVER (equal? A-WRLD WIN))
  ```
  ```

# N-Puzzle Problem

## The game-over? Handler

- ```
  ;; world → Boolean
  ;; Purpose: Determine if the game has ended
  (define (game-over? a-world)
    (equal? a-world WIN))
  ```
- ```
  ;; Sample expressions for game-over?
  (define WIN-OVER (equal? WIN WIN))
  (define A-WRLD-OVER (equal? A-WRLD WIN))
  ```

# N-Puzzle Problem

## The game-over? Handler

```
;; world → Boolean
;; Purpose: Determine if the game has ended
(define (game-over? a-world)
  (equal? a-world WIN))
```

```
;; Sample expressions for game-over?
(define WIN-OVER (equal? WIN WIN))
(define A-WRLD-OVER (equal? A-WRLD WIN))
```

```
;; Tests using sample computations for game-over?
(check-expect (game-over? A-WRLD) A-WRLD-OVER)
(check-expect (game-over? WIN)    WIN-OVER)

;; Tests using sample values for game-over?
(check-expect (game-over? (make-world 5 2 3
                                      1 8 6
                                      4 0 7))
              #false)
```

# N-Puzzle Problem

## The `draw-last-world` Handler

- ```
  ;; Sample expressions for draw-last-world
  (define WIN-FIMG (overlay/xy (text "PUZZLE SOLVED!!!" 28 'brown)
                               -25 -75 (draw-world WIN)))
  (define A-WRD-FIMG (overlay/xy (text "PUZZLE SOLVED!!!" 28 'brown)
                                 -25 -75 (draw-world A-WRLD)))
  ```
  ```

# N-Puzzle Problem

## The `draw-last-world` Handler

- `;; world → image  Purpose: Draw the final world`
  `(define (draw-last-world a-world)`

- ```
  ;; Sample expressions for draw-last-world
  (define WIN-FIMG (overlay/xy (text "PUZZLE SOLVED!!!" 28 'brown)
                               -25 -75 (draw-world WIN)))
  (define A-WRD-FIMG (overlay/xy (text "PUZZLE SOLVED!!!" 28 'brown)
                                 -25 -75 (draw-world A-WRLD)))
  ```
  ```

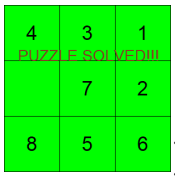
# N-Puzzle Problem

## The draw-last-world Handler

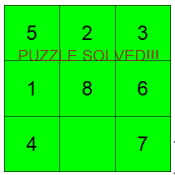
- ;; world → image  Purpose: Draw the final world
```
(define (draw-last-world a-world)
```

- ;; Sample expressions for draw-last-world
```
(define WIN-FIMG (overlay/xy (text "PUZZLE SOLVED!!!" 28 'brown)
                             -25 -75 (draw-world WIN)))
(define A-WRD-FIMG (overlay/xy (text "PUZZLE SOLVED!!!" 28 'brown)
                               -25 -75 (draw-world A-WRLD)))
```

- ;; Tests using sample computations for draw-last-world
```
(check-expect (draw-last-world WIN)    WIN-FIMG)
(check-expect (draw-last-world A-WRLD) A-WRLD-FIMG)
;; Tests using sample values for draw-last-world
(check-expect (draw-last-world (make-world 4 3 1 0 7 2 8 5 6))
```




```
)
(check-expect (draw-last-world (make-world 5 2 3 1 8 6 4 0 7))
```




```
)
```

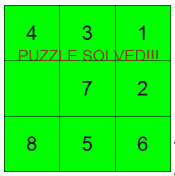
# N-Puzzle Problem

## The draw-last-world Handler

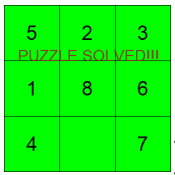
```
;; world → image  Purpose: Draw the final world
(define (draw-last-world a-world)
  (overlay/xy (text "PUZZLE SOLVED!!!" 28 'brown)
              -25 -75 (draw-world a-world)))
```

```
;; Sample expressions for draw-last-world
(define WIN-FIMG (overlay/xy (text "PUZZLE SOLVED!!!" 28 'brown)
                             -25 -75 (draw-world WIN)))
(define A-WRD-FIMG (overlay/xy (text "PUZZLE SOLVED!!!" 28 'brown)
                               -25 -75 (draw-world A-WRLD)))
```

```
;; Tests using sample computations for draw-last-world
(check-expect (draw-last-world WIN)    WIN-FIMG)
(check-expect (draw-last-world A-WRLD) A-WRLD-FIMG)
;; Tests using sample values for draw-last-world
(check-expect (draw-last-world (make-world 4 3 1 0 7 2 8 5 6))
```




```
)
(check-expect (draw-last-world (make-world 5 2 3 1 8 6 4 0 7))
```




```
)
```

# N-Puzzle Problem

## The `process-key` Handler

- Moves the empty tile space
- Makes a move when the player requests help
- In this version of the game the request for help shall not be implemented

# N-Puzzle Problem

## The process-key Handler

- Moves the empty tile space
- Makes a move when the player requests help
- In this version of the game the request for help shall not be implemented
- Game only reacts to keys for moving the empty space and for help:

```
;; A valid key, vk, is either
;; 1. "up"  2. "down"  3. "left"  4. "right"  5. " "
;; Sample vk
(define UP    "up")    (define DOWN  "down") (define LEFT  "left")
(define RIGHT "right") (define HKEY  " ")
#| ;; vk ... → ...  Purpose:
   (define (f-on-vk a-vk ...)
    (cond [(key=? a-vk UP)    ...]
          [(key=? a-vk DOWN)  ...]
          [(key=? a-vk LEFT)  ...]
          [(key=? a-vk RIGHT) ...]
          [else ...]))
   ;; Sample expressions for f-on-vk
   (define UP-VAL    ...) (define DOWN-VAL  ...)
   (define LEFT-VAL  ...) (define RIGHT-VAL ...) (define HKEY-VAL  .
   ;; Tests using sample computations for f-on-vk
   (check-expect (f-on-vk UP    ...) UP-VAL) ...
   ;; Tests using sample values for f-on-vk
   (check-expect (f-on-vk ...  ...) ...)... |#
```

# N-Puzzle Problem

## The `process-key` Handler

- The design idea: distinguish between `vk` and `non-vk` keystrokes

# N-Puzzle Problem

## The process-key Handler

- The design idea: distinguish between `vk` and `non-vk` keystrokes
- Outline:

```
;; world key → world
;; Purpose: Return next world after the given key event
(define (process-key a-world a-key)
  (local [...
          (define (vk? a-key) ...)
          ...
          (define (process-vk a-world a-vk) ...)]
    (if (vk? a-key)
        (process-vk a-world a-key)
        a-world)))
```

# N-Puzzle Problem

## The process-key Handler

```
;; Tests for process-key
(check-expect (process-key A-WRLD "m") A-WRLD)
(check-expect (process-key A-WRLD "d") A-WRLD)
(check-expect (process-key WIN UP)
              (make-world 1 2 3 4 5 0 7 8 6))
(check-expect (process-key (make-world 0 1 2 3 4 5 6 7 8)
                           UP)
              (make-world 0 1 2 3 4 5 6 7 8))
(check-expect (process-key (make-world 0 1 2 3 4 5 6 7 8)
                           DOWN)
              (make-world 3 1 2 0 4 5 6 7 8))
(check-expect (process-key WIN DOWN) WIN)
check-expect (process-key A-WRLD LEFT)
              (make-world 1 5 2 0 4 3 7 8 6))
(check-expect (process-key (make-world 1 5 2 0 4 3 7 8 6)
                           LEFT)
              (make-world 1 5 2 0 4 3 7 8 6))
(check-expect (process-key A-WRLD RIGHT)
              (make-world 1 5 2 4 3 0 7 8 6))
(check-expect (process-key WIN RIGHT) WIN)
(check-expect (process-key A-WRLD HKEY) A-WRLD)
```

# N-Puzzle Problem

## The Design of vk?

```
;; key → Boolean
;; Purpose: Determine in given key is a vk
(define (vk? a-key)
  (or (key=? a-key UP)
      (key=? a-key DOWN)
      (key=? a-key LEFT)
      (key=? a-key RIGHT)
      (key=? a-key HKEY)))
```

# N-Puzzle Problem

## The Design of process-vk

- ```
  ;; world vk → world
  ;; Purpose: Return the next world after given vk
  (define (process-vk a-world a-vk)
      (local [(define (blank-pos a-world) ...)
              (define (get-target-bpos a-world a-vk) ...)
              (define (swap-empty a-world target) ...)
              (define (make-move a-world) ...)]
    (if (or (key=? a-vk UP)   (key=? a-vk DOWN)
            (key=? a-vk LEFT) (key=? a-vk RIGHT))
        (swap-empty a-world (get-target-bpos a-world a-vk))
        (make-move a-world))))
  ```
  ```

# N-Puzzle Problem

## Computing the Position of the Empty Space

- ```
  ;; world → bpos
  ;; Purpose: Return the bpos for the blank
  (define (blank-pos a-world)
    (cond [(= (world-t0 a-world) 0) 0]
          [(= (world-t1 a-world) 0) 1]
          [(= (world-t2 a-world) 0) 2]
          [(= (world-t3 a-world) 0) 3]
          [(= (world-t4 a-world) 0) 4]
          [(= (world-t5 a-world) 0) 5]
          [(= (world-t6 a-world) 0) 6]
          [(= (world-t7 a-world) 0) 7]
          [(= (world-t8 a-world) 0) 8])
  ```
  ```

# N-Puzzle Problem

## The Design of `get-target-bpos`

- Compute the board position to swap the empty space into
- Three questions to answer:
    1. `What is the target board position if the empty space cannot be moved?`
    2. `How can we determine if the empty space may be moved in the given direction?`
    3. `How is the target board position computed when the empty space can be moved?`

# N-Puzzle Problem

## The Design of `get-target-bpos`

- Compute the board position to swap the empty space into
- Three questions to answer:
    1. `What is the target board position if the empty space cannot be moved?`
    2. `How can we determine if the empty space may be moved in the given direction?`
    3. `How is the target board position computed when the empty space can be moved?`
- First question: return the board position of the empty space

# N-Puzzle Problem

## The Design of `get-target-bpos`

- Compute the board position to swap the empty space into
- Three questions to answer:
  1. `What is the target board position if the empty space cannot be moved?`
  2. `How can we determine if the empty space may be moved in the given direction?`
  3. `How is the target board position computed when the empty space can be moved?`
- First question: return the board position of the empty space
- Second question: reason about each possible `vk` values and the position of the blank space
- Given `vk` is `UP`: empty space may move only if it is not in the top row
- Given `vk` is `DOWN`: empty space may move only if it is not in the bottom row
- Given `vk` is `LEFT`: empty space may move only if it is not in leftmost column
- Given `vk` is `RIGHT`: empty space may move only if it is not in rightmost column

# N-Puzzle Problem

## The Design of `get-target-bpos`

- Compute the board position to swap the empty space into
- Three questions to answer:
  1. `What is the target board position if the empty space cannot be moved?`
  2. `How can we determine if the empty space may be moved in the given direction?`
  3. `How is the target board position computed when the empty space can be moved?`
- First question: return the board position of the empty space
- Second question: reason about each possible `vk` values and the position of the blank space
- Given `vk` is `UP`: empty space may move only if it is not in the top row
- Given `vk` is `DOWN`: empty space may move only if it is not in the bottom row
- Given `vk` is `LEFT`: empty space may move only if it is not in leftmost column
- Given `vk` is `RIGHT`: empty space may move only if it is not in rightmost column
- Third question: reason about each possible `vk` values
- Blank moves up: decremented by 3
- Blank moves down: increment by 3
- Blank moves left: decrement by 1
- Blank moves right: increment by 1

# N-Puzzle Problem

## The Design of `get-target-bpos`

- ```
  ;; world vk → bpos
  ;; Purpose: Return the bpos to move the blank into
  (define (get-target-bpos a-world a-vk)
  ```
  ```

# N-Puzzle Problem

## The Design of `get-target-bpos`

- ```
  ;; world vk → bpos
  ;; Purpose: Return the bpos to move the blank into
  (define (get-target-bpos a-world a-vk)
  ```
- ```
    (local [(define BLNK-POS (blank-pos a-world))]
  ```

# N-Puzzle Problem

## The Design of get-target-bpos

```
;; world vk → bpos
;; Purpose: Return the bpos to move the blank into
(define (get-target-bpos a-world a-vk)
  (local [(define BLNK-POS (blank-pos a-world))]
    (cond [(key=? a-vk UP)
           (if (< BLNK-POS 3)
               BLNK-POS
               (- BLNK-POS 3))]
          [(key=? a-vk DOWN)
           (if (> BLNK-POS 5)
               BLNK-POS
               (+ BLNK-POS 3))]
          [(key=? a-vk LEFT)
           (if (= (remainder BLNK-POS 3) 0)
               BLNK-POS
               (sub1 BLNK-POS))]
          [else
           (if (= (remainder BLNK-POS 3) 2)
               BLNK-POS
               (add1 BLNK-POS))])))
```

# N-Puzzle Problem

## The Design of swap-empty

- ```
  ;; world bpos → world
  ;; Purpose: Move the blank to the given bpos
  (define (swap-empty a-world target)
    (local [...
            (define (new-tile-value a-world a-bpos) ...)]
              (make-world (new-tile-value a-world 0)
                          (new-tile-value a-world 1)
                          (new-tile-value a-world 2)
                          (new-tile-value a-world 3)
                          (new-tile-value a-world 4)
                          (new-tile-value a-world 5)
                          (new-tile-value a-world 6)
                          (new-tile-value a-world 7)
                          (new-tile-value a-world 8))))
  ```
  ```

# N-Puzzle Problem

## The Design of swap-empty

- ```
  ;; world bpos → world
  ;; Purpose: Move the blank to the given bpos
  (define (swap-empty a-world target)
    (local [...
            (define (new-tile-value a-world a-bpos) ...)]
              (make-world (new-tile-value a-world 0)
                          (new-tile-value a-world 1)
                          (new-tile-value a-world 2)
                          (new-tile-value a-world 3)
                          (new-tile-value a-world 4)
                          (new-tile-value a-world 5)
                          (new-tile-value a-world 6)
                          (new-tile-value a-world 7)
                          (new-tile-value a-world 8))))
  ```
- ```
  ;; world bpos → bval Purpose: Return new value of given bpos
  (define (new-tile-value a-world a-bpos)
    (local [ ...
            (define (get-tile-value a-world a-bpos) ...)]
              (cond [(= target a-bpos) 0]
                    [(blank-pos a-world)
                     (get-tile-value a-world target)]
                    [else (get-tile-value a-world a-bpos)])))
  ```

# N-Puzzle Problem

## The Design of `swap-empty`

- ```
  ;; world bpos → bval
  ;; Purpose: Return the given bpos' bval
  (define (get-tile-value a-world a-bpos)
    (cond [(= a-bpos 0) (world-t0 a-world)]
          [(= a-bpos 1) (world-t1 a-world)]
          [(= a-bpos 2) (world-t2 a-world)]
          [(= a-bpos 3) (world-t3 a-world)]
          [(= a-bpos 4) (world-t4 a-world)]
          [(= a-bpos 5) (world-t5 a-world)]
          [(= a-bpos 6) (world-t6 a-world)]
          [(= a-bpos 7) (world-t7 a-world)]
          [else (world-t8 a-world)]))
  ```
  ```

# N-Puzzle Problem

## The Design of `make-move`

- In this version of the game do nothing

# N-Puzzle Problem

## The Design of `make-move`

- In this version of the game do nothing
- ```
  ;; world → world
  ;; Purpose: Make a move on behalf of the player
  (define (make-move a-world) a-world)
  ```
- A function like `make-move` above is called a `stub` (or `function stub`)
- A stub is a function written to temporarily fill-in for some programming functionality that is not yet developed
  ```

# Randomness

- How to make a move for the player in the 8-puzzle game? How do you decide to make a move?

# Randomness

- How to make a move for the player in the 8-puzzle game? How do you decide to make a move?
- Difficult questions to answer

# Randomness

- How to make a move for the player in the 8-puzzle game? How do you decide to make a move?
- Difficult questions to answer
- Given our inability to express how a human solves the puzzle randomly pick among the possible moves?

# Randomness

- How to make a move for the player in the 8-puzzle game? How do you decide to make a move?
- Difficult questions to answer
- Given our inability to express how a human solves the puzzle randomly pick among the possible moves?
- Attractive because of its simplicity
- Determine the empty space's neighbors
- Randomly pick one to make a move for the player.

# Randomness

- How to make a move for the player in the 8-puzzle game? How do you decide to make a move?
- Difficult questions to answer
- Given our inability to express how a human solves the puzzle randomly pick among the possible moves?
- Attractive because of its simplicity
- Determine the empty space's neighbors
- Randomly pick one to make a move for the player.
- Randomly picking a move `make-move` becomes a *nondeterministic function*
- A function whose result cannot be predicted
- Has profound implications for writing tests and guaranteeing that our programs terminate

# Randomness

## ISL+'s `random` Function

- Randomness requires the generation of random numbers
- The computer needs to generate a sequence numbers that cannot be predicted
- The level of unpredictability of the next number to be generated is called *entropy*
- A random number generator ought to maximize the level of entropy
- The generation of a random integer in `[0..n-1]` ought to make it equally likely that any integer in the given range is the next number generated: $\frac{1}{n}$

# Randomness

## ISL+'s `random` Function

- Randomness requires the generation of random numbers
- The computer needs to generate a sequence numbers that cannot be predicted
- The level of unpredictability of the next number to be generated is called *entropy*
- A random number generator ought to maximize the level of entropy
- The generation of a random integer in `[0..n-1]` ought to make it equally likely that any integer in the given range is the next number generated: $\frac{1}{n}$
- Generating random numbers is an incredibly difficult problem
- Computers can produce long sequences of numbers that appear random
- Use a mathematical formula that requires an input value that is called a *seed*
- The numbers generated are not truly random
- Anyone that knows the formula used and the seed value can predict the sequence of numbers generated
- This is called a *pseudo random number generator*

# Randomness

## ISL+'s `random` Function

- In `ISL+`, the `random` function may be used to generate pseudo random natural numbers
- Input: a natural number, `n`
- Output: a natural number in `[0..n-1]`
-
```
> (random 10)
8
> (random 10)
9
> (random 10)
8
> (random 10)
3
```

# Randomness

## ISL+'s `random` Function

- In `ISL+`, the `random` function may be used to generate pseudo random natural numbers
- Input: a natural number, `n`
- Output: a natural number in `[0..n-1]`
- ```
  > (random 10)
  8
  > (random 10)
  9
  > (random 10)
  8
  > (random 10)
  3
  ```
- The good news is that we can simulate randomness in `ISL+`
  ```

# Randomness

## ISL+'s `random` Function

- In ISL+, the `random` function may be used to generate pseudo random natural numbers
- Input: a natural number, `n`
- Output: a natural number in `[0..n-1]`
- ```
  > (random 10)
  8
  > (random 10)
  9
  > (random 10)
  8
  > (random 10)
  3
  ```
- The good news is that we can simulate randomness in ISL+
- The bad news is that randomness breaks our mental model of what a function is
- We usually assume that the functions we write are *deterministic*
- When this condition holds functions exhibit what is called *referential transparency*
- Referential transparency is the property that an expression may always be substituted with its value
- `(f x)` is always equal to `(f x)`.
  ```

# Randomness

## ISL+'s `random` Function

- Run the following test:

```
(check-expect (random 10) (random 10))
```

- The test fails (fairly frequently)!
- When `random` is used we lose referential transparency

# Randomness

## ISL+'s `random` Function

- Run the following test:

```
(check-expect (random 10) (random 10))
```

- The test fails (fairly frequently)!
- When `random` is used we lose referential transparency
- The use of randomness makes the job of a problem solver more difficult
- How can the function be tested?

# Randomness

## ISL+'s `random` Function

- Run the following test:

```
(check-expect (random 10) (random 10))
```

- The test fails (fairly frequently)!
- When `random` is used we lose referential transparency
- The use of randomness makes the job of a problem solver more difficult
- How can the function be tested?
- Property-based testing:

```
(check-expect (<= 0 (random 10) 9) #true)
(check-expect (<= 0 (random 10) 9) #true)
```

# Randomness

## ISL+'s `random` Function

- Run the following test:

```
(check-expect (random 10) (random 10))
```

- The test fails (fairly frequently)!
- When `random` is used we lose referential transparency
- The use of randomness makes the job of a problem solver more difficult
- How can the function be tested?
- Property-based testing:

```
(check-expect (<= 0 (random 10) 9) #true)
(check-expect (<= 0 (random 10) 9) #true)
```

- `ISL+` provides a more elegant way of testing properties using `check-satisfied`
- This type of test checks if the value of a given expression satisfies a given one-input predicate
-

```
(check-satisfied (random 10) (λ (n) (<= 0 n 9)))
(check-satisfied (random 10) (λ (n) (<= 0 n 9)))
```

- Why is the same test written more than once?

# Randomness

## ISL+'s `random` Function

- Run the following test:

```
(check-expect (random 10) (random 10))
```

- The test fails (fairly frequently)!
- When `random` is used we lose referential transparency
- The use of randomness makes the job of a problem solver more difficult
- How can the function be tested?
- Property-based testing:

```
(check-expect (<= 0 (random 10) 9) #true)
(check-expect (<= 0 (random 10) 9) #true)
```

- `ISL+` provides a more elegant way of testing properties using `check-satisfied`
- This type of test checks if the value of a given expression satisfies a given one-input predicate
- 

```
(check-satisfied (random 10) (λ (n) (<= 0 n 9)))
(check-satisfied (random 10) (λ (n) (<= 0 n 9)))
```

- Why is the same test written more than once?
- Remember that they are not the same test because the value of `(random 10)` is not predictable.

# Randomness

## ISL+'s `random` Function

- A second technique involves making sure that the two expressions tested use the same seed
- If the same seed is used then the pseudo random number generator produces the same numbers for both tested expressions

# Randomness

## ISL+'s `random` Function

- A second technique involves making sure that the two expressions tested use the same seed
- If the same seed is used then the pseudo random number generator produces the same numbers for both tested expressions
- Use `check-random`:

```
(check-random (random 10) (random 10))
(check-random (random 50) (random 50))
```

- Both tests pass because the same seed is used in both expressions

# Randomness

## ISL+'s `random` Function

- A second technique involves making sure that the two expressions tested use the same seed
- If the same seed is used then the pseudo random number generator produces the same numbers for both tested expressions
- Use `check-random`:

```
(check-random (random 10) (random 10))
(check-random (random 50) (random 50))
```

- Both tests pass because the same seed is used in both expressions
- Both expressions must request random numbers in the same order in the same interval
- The following tests fail:

```
(check-random (random 20) (random 60))
(check-random (+ (random 10) (random 20) (random 30))
              (+ (random 20) (random 10) (random 30)))
```

- In the first test both expressions request a random number in the same order but use different intervals
- In the second test the same intervals are used but the random numbers are requested in a different order.

# Randomness

## N-Puzzle Version 1

- A random move made when the player requests help
- The only function that requires redesign is `make-move` (which means the behavior of `process-key` in changed) Why is this important?

# Randomness

## N-Puzzle Version 1

- A random move made when the player requests help
- The only function that requires redesign is `make-move` (which means the behavior of `process-key` in changed) Why is this important?
- Board positions neighboring the empty tile space need to be known
- A neighboring board position is randomly chosen and swapped with the empty tile space

# Randomness

## N-Puzzle Version 1

- The neighbors of every board position are fixed:

```
(define neighbors '((1 3)
                    (4 0 2)
                    (1 5)
                    (0 4 6)
                    (7 1 3 5)
                    (2 4 8)
                    (3 7)
                    (8 4 6)
                    (5 7)))
```

- $\texttt{i}^{\texttt{th}}$ element contains the neighbors of the $\texttt{i}^{\texttt{th}}$ board position

# Randomness

## N-Puzzle Version 1

- The neighbors of every board position are fixed:

```
(define neighbors '((1 3)
                    (4 0 2)
                    (1 5)
                    (0 4 6)
                    (7 1 3 5)
                    (2 4 8)
                    (3 7)
                    (8 4 6)
                    (5 7)))
```

- $\texttt{i}^{\texttt{th}}$ element contains the neighbors of the $\texttt{i}^{\texttt{th}}$ board position
- The neighbors of the empty space position may be defined as follows:

```
(define BLNK-NEIGHS (list-ref neighbors
                              (blank-pos a-world)))
```

# Randomness

## N-Puzzle Version 1

- The neighbors of every board position are fixed:

```
(define neighbors '((1 3)
                    (4 0 2)
                    (1 5)
                    (0 4 6)
                    (7 1 3 5)
                    (2 4 8)
                    (3 7)
                    (8 4 6)
                    (5 7)))
```

- $\texttt{i}^{\texttt{th}}$ element contains the neighbors of the $\texttt{i}^{\texttt{th}}$ board position
- The neighbors of the empty space position may be defined as follows:

```
(define BLNK-NEIGHS (list-ref neighbors
                              (blank-pos a-world)))
```

- To swap:

```
(swap-empty (list-ref BLNK-NEIGHS
                      (random (length BLNK-NEIGHS))))
```

# Randomness

## N-Puzzle Version 1

- The complete function to make local inside of `process-key`:

```
;; world arrow world
;; Purpose: Make a move for the player
(define (make-move a-world)
 (local [(define neighbors '((1 3)
                             (4 0 2)
                             (1 5)
                             (0 4 6)
                             (7 1 3 5)
                             (2 4 8)
                             (3 7)
                             (8 4 6)
                             (5 7)))

         (define BLNK-NEIGHS (list-ref neighbors
                                       (blank-pos a-world)))]
   (swap-empty (list-ref BLNK-NEIGHS
                         (random (length BLNK-NEIGHS))))))
```

# Randomness

## N-Puzzle Version 1

- Changed the design of `process-key`
- The processing `HKEY` now creates a new `world`.
- The following test fails:

```
(check-expect (process-key A-WRLD HKEY) A-WRLD)
```

- How do we write tests for the processing of `HKEY`?

# Randomness

## N-Puzzle Version 1

- Changed the design of `process-key`
- The processing `HKEY` now creates a new `world`.
- The following test fails:

```
(check-expect (process-key A-WRLD HKEY) A-WRLD)
```

- How do we write tests for the processing of `HKEY`?
- We cannot write tests that look as follows: Why?

```
(check-expect (process-key ... HKEY) (make-world ...))
```

# Randomness

## N-Puzzle Version 1

- Changed the design of `process-key`
- The processing `HKEY` now creates a new `world`.
- The following test fails:

```
(check-expect (process-key A-WRLD HKEY) A-WRLD)
```

- How do we write tests for the processing of `HKEY`?
- We cannot write tests that look as follows: Why?

```
(check-expect (process-key ... HKEY) (make-world ...))
```

- The best we can do is to check properties of the value returned
- Worlds ought to be different `worlds` by a single away

# Randomness

## N-Puzzle Version 1

- Changed the design of `process-key`
- The processing `HKEY` now creates a new `world`.
- The following test fails:

```
(check-expect (process-key A-WRLD HKEY) A-WRLD)
```

- How do we write tests for the processing of `HKEY`?
- We cannot write tests that look as follows: Why?

```
(check-expect (process-key ... HKEY) (make-world ...))
```

- The best we can do is to check properties of the value returned
- Worlds ought to be different `worlds` by a single away
- Only have different tiles in two board positions
- The tile value in the first world at the blank's board position in the second world must equal the tile value in the second world at the blank's board position in the first world

# Randomness

## N-Puzzle Version 1

```
;; world world → Boolean Purpose: Determine if worlds one move awa
(define (one-move-away? w1 w2)
  (local [;; world → bpos Purpose: Return the blank's bpos
          (define (get-empty-bpos a-world)
            (cond ...))
          ;; world bpos → bval Purpose: Return given bpos' bval
          (define (get-tile-value a-world a-bpos)
            (cond ...))
          (define BLNK-W1 (get-empty-bpos w1))
          (define BLNK-W2 (get-empty-bpos w2))
          ;; world world → Boolean Purpose: Count differences
          (define (cnt-diffs w1 w2)
            (+ (if (= (world-t0 w1)(world-t0 w2)) 0 1)
               (if (= (world-t1 w1)(world-t1 w2)) 0 1)
               (if (= (world-t2 w1)(world-t2 w2)) 0 1)
               (if (= (world-t3 w1)(world-t3 w2)) 0 1)
               (if (= (world-t4 w1)(world-t4 w2)) 0 1)
               (if (= (world-t5 w1)(world-t5 w2)) 0 1)
               (if (= (world-t6 w1)(world-t6 w2)) 0 1)
               (if (= (world-t7 w1)(world-t7 w2)) 0 1)
               (if (= (world-t8 w1)(world-t8 w2)) 0 1)))]
    (and (= (cnt-diffs w1 w2) 2)
         (= (get-tile-value w1 BLNK-W2) (get-tile-value w2 BLNK-W1)
```

# Randomness

## N-Puzzle Version 1

- ```
  ;; Tests using sample values for one-move-away?
  (check-expect (one-move-away? WIN    (make-world 1 2 3 4 5 6 7 0 8)
                #true)
  (check-expect (one-move-away? A-WRLD (make-world 1 5 2 4 8 3 7 0 6)
                #true)
  (check-expect (one-move-away? A-WRLD (make-world 0 1 2 4 5 3 7 8 6)
                #false)
  ```
  ```

# Randomness

## N-Puzzle Version 1

- ```
  ;; Tests using sample values for one-move-away?
  (check-expect (one-move-away? WIN    (make-world 1 2 3 4 5 6 7 0 8))
                #true)
  (check-expect (one-move-away? A-WRLD (make-world 1 5 2 4 8 3 7 0 6))
                #true)
  (check-expect (one-move-away? A-WRLD (make-world 0 1 2 4 5 3 7 8 6))
                #false)
  ```
- The `process-key` tests using `HKEY` are updated as follows:

  ```
  (check-expect
    (one-move-away? A-WRLD (process-key A-WRLD HKEY))
    #true)
  (check-expect (one-move-away? WIN (process-key WIN HKEY))
                #true)
  ```
  ```

# Randomness

## N-Puzzle Version 1

- Run the game several times using `(make-world 1 2 3 4 5 6 7 0 8)` as the initial `world` only request help

# Randomness

## N-Puzzle Version 1

- Run the game several times using `(make-world 1 2 3 4 5 6 7 0 8)` as the initial `world` only request help
- The program does not always make the right move
- Are you satisfied with this implementation to help the player?

# Randomness

## N-Puzzle Version 1

- Run the game several times using `(make-world 1 2 3 4 5 6 7 0 8)` as the initial `world` only request help
- The program does not always make the right move
- Are you satisfied with this implementation to help the player?
- There is another more subtle problem with our program
- Imagine a player that only wants to see the moves to solve a puzzle
- Repeatedly requests help until the puzzle is solved
- The player may request help for millennia if she could live that long!
- We cannot argue that the puzzle will be solved
- An algorithm that may run forever to find the solution to a problem is not a practical solution
- This observation suggests that we need to learn techniques to approximate answers when finding a solution may take too long.

# Randomness

## N-Puzzle Version 1

- Run the game several times using `(make-world 1 2 3 4 5 6 7 0 8)` as the initial `world` only request help
- The program does not always make the right move
- Are you satisfied with this implementation to help the player?
- There is another more subtle problem with our program
- Imagine a player that only wants to see the moves to solve a puzzle
- Repeatedly requests help until the puzzle is solved
- The player may request help for millennia if she could live that long!
- We cannot argue that the puzzle will be solved
- An algorithm that may run forever to find the solution to a problem is not a practical solution
- This observation suggests that we need to learn techniques to approximate answers when finding a solution may take too long.
- Randomness must be used with care and as problem solvers you must recognize when using randomness is a good choice
- Randomness ought to be considered when it is important not to be able to predict a function's value

# Randomness

## Generating Random Passwords

- Cybersecurity experts always remind us that we ought to use strong unique passwords that are not easy to predict
- A strong password is a string of at least 10 characters that includes a combination of at least one uppercase letter, one lowercase letter, one number, and one special character
- Special characters include, for example, `$`, `&`, and `*`.

# Randomness

## Generating Random Passwords

- Cybersecurity experts always remind us that we ought to use strong unique passwords that are not easy to predict
- A strong password is a string of at least 10 characters that includes a combination of at least one uppercase letter, one lowercase letter, one number, and one special character
- Special characters include, for example, `$`, `&`, and `*`.
- A password character, `pwdchar`, may be represented as a string of length 1
- The four sets of password characters needed to create a strong password may be represented using `(listof pwdchar)`s
- We may define these four sets as follows:

```
(define UCASE (map symbol->string
                   '(A B C D E F G H I J K L M
                     N O P Q R S T U V W X Y Z)))
(define LCASE (map string-downcase UCASE))
(define NUMBS (map number->string '(0 1 2 3 4 5 6 7 8 9)))
(define SCHRS '("$" "&" "*" "+" "@" "#" "_" "!" "?"))
```

- A password is defined as follows:

```
#|
A password, pw, is a string that has a minimum length of 10
and contains at least one of each: lower case letter, upper
case letter, number, and special character.
|#
```

# Randomness

## Generating Random Passwords

- A strong password ought to be difficult to predict
- This suggests the use of randomness to
- User may specify the length of a password using a natural number

# Randomness

## Generating Random Passwords

- A strong password ought to be difficult to predict
- This suggests the use of randomness to
- User may specify the length of a password using a natural number
- Characters used from each set are randomly chosen
- The chosen elements may be randomly used to generate a password

# Randomness

## Generating Random Passwords

- Outline a random password generating function as follows:

```
;; natnum → pw throws error
;; Purpose: To generate a random password of the given length
(define (generate-password passwd-len)
```

# Randomness

## Generating Random Passwords

- Outline a random password generating function as follows:

```
;; natnum → pw throws error
;; Purpose: To generate a random password of the given length
(define (generate-password passwd-len)
```

- 
```
  (if (< passwd-len 10)
      (error "The password length must be at least 10.")
```

# Randomness

## Generating Random Passwords

- Outline a random password generating function as follows:

```
;; natnum → pw throws error
;; Purpose: To generate a random password of the given length
(define (generate-password passwd-len)
```

- 
```
  (if (< passwd-len 10)
      (error "The password length must be at least 10.")
```

- 
```
      (local
        [(define UC-NUM (add1 (random 7)))
         (define LC-NUM (add1 (random (- passwd-len UC-NUM 2))))
         (define NB-NUM (add1 (random (- passwd-len UC-NUM LC-NUM 1
         (define SC-NUM (- passwd-len UC-NUM LC-NUM NB-NUM))

         ;; (listof X) natnum → (listof X)
         ;; Purpose: Randomly pick n of elements from given list
         (define (pick lst n)
           (build-list n (λ (i) (list-ref lst (random (length lst))

         ;; ...  Purpose: ...
         (define (generate ...) ...) ]
```

# Randomness

## Generating Random Passwords

- Outline a random password generating function as follows:

```
;; natnum → pw throws error
;; Purpose: To generate a random password of the given length
(define (generate-password passwd-len)
  (if (< passwd-len 10)
      (error "The password length must be at least 10.")
      (local
        [(define UC-NUM (add1 (random 7)))
         (define LC-NUM (add1 (random (- passwd-len UC-NUM 2))))
         (define NB-NUM (add1 (random (- passwd-len UC-NUM LC-NUM 1
         (define SC-NUM (- passwd-len UC-NUM LC-NUM NB-NUM))

         ;; (listof X) natnum → (listof X)
         ;; Purpose: Randomly pick n of elements from given list
         (define (pick lst n)
           (build-list n (λ (i) (list-ref lst (random (length lst))

         ;; ...  Purpose: ...
         (define (generate ...) ...) ]
       (generate (pick UCASE UC-NUM)
                 (pick LCASE LC-NUM)
                 (pick NUMBS NB-NUM)
                 (pick SCHRS SC-NUM)))))
```

# Randomness

## Generating Random Passwords

- The next task is to design the function that generates a random string using the elements from each set
- Input 4 `(listof string)` (i.e., the sets to use)
- Output: a `string`

# Randomness

## Generating Random Passwords

- The next task is to design the function that generates a random string using the elements from each set
- Input 4 `(listof string)` (i.e., the sets to use)
- Output: a `string`
- If all 4 given lists are empty then the empty string is returned

# Randomness

## Generating Random Passwords

- The next task is to design the function that generates a random string using the elements from each set
- Input 4 (`listof string`) (i.e., the sets to use)
- Output: a `string`
- If all 4 given lists are empty then the empty string is returned
- Otherwise, a list is randomly chosen.
- If the chosen list is empty then a recursive call is made without changing any of the given lists
- If the chosen list is not empty then its first element is appended with the result of recursively processing the rest of the chosen list and the other lists

# Randomness

## Generating Random Passwords

- ```
  ;(listof str) (listof str) (listof str) (listof str) → string
  ;; Purpose: Generate string using all strings in the given lists
  (define (generate L0 L1 L2 L3)
   (if (and (empty? L0) (empty? L1) (empty? L2) (empty? L3))
    ""
    (local [(define lst-num (random 4))]
     (cond
      [(= lst-num 0)
       (if (empty? L0)
         (generate L0 L1 L2 L3)
         (string-append (first L0) (generate (rest L0) L1 L2 L3)))]
      [(= lst-num 1)
       (if (empty? L1)
          (generate L0 L1 L2 L3)
          (string-append (first L1) (generate L0 (rest L1) L2 L3)))]
      [(= lst-num 2)
       (if (empty? L2)
          (generate L0 L1 L2 L3)
          (string-append (first L2) (generate L0 L1 (rest L2) L3)))]
      [(= lst-num 3)
       (if (empty? L3)
          (generate L0 L1 L2 L3)
          (string-append (first L3) (generate L0 L1 L2 (rest L3))))])))
  ```
  ```

# Randomness

## Generating Random Passwords

- There are recursive calls with the given arguments
- This suggest that there may be an *infinite recursion*
- An infinite recursion occurs when a function fails to make progress towards termination
- In this case, it means that it fails to move all four arguments towards becoming empty

# Randomness

## Generating Random Passwords

- There are recursive calls with the given arguments
- This suggest that there may be an *infinite recursion*
- An infinite recursion occurs when a function fails to make progress towards termination
- In this case, it means that it fails to move all four arguments towards becoming empty
- Essentially, we hope that each given list is chosen enough times to become empty
- Although extremely unlikely with a good pseudo random number generator it is possible for this program to run forever because one of the lists is never chosen
- This possibility ought to make you very uncomfortable because it means this solution may be impractical.

# Randomness

## Generating Random Passwords

- Don't forget tests must be written!

# Randomness

## Generating Random Passwords

- Don't forget tests must be written!
- ```
  (check-error (generate-password 6)
               "The password length must be at least 10.")
  ```
  ```

# Randomness

## Generating Random Passwords

- Don't forget tests must be written!
- ```
  (check-error (generate-password 6)
               "The password length must be at least 10.")
  ```
- How do we test the function when a password is actually generated?
  ```

# Randomness

## Generating Random Passwords

- Don't forget tests must be written!
- ```
  (check-error (generate-password 6)
               "The password length must be at least 10.")
  ```
- How do we test the function when a password is actually generated?
- `generate-password` is nondeterministic and, therefore, we may employ property-based testing
- Sample tests:

  ```
  (check-satisfied (generate-password 10) valid-passwd?)
  (check-satisfied (generate-password 12) valid-passwd?)
  ```
- We need to design a predicate to determine if a given string is a valid password
  ```

# Randomness

## Generating Random Passwords

- How do we determine if a given string is a valid password?

# Randomness

## Generating Random Passwords

- How do we determine if a given string is a valid password?
- The string must:
  - Have a minimum length of 10
  - Have at least one uppercase letter
  - Have at least one lowercase letter
  - Have at least one number
  - Have at least one special character
  - Contain no other characters

# Randomness

## Generating Random Passwords

- How do we determine if a given string is a valid password?
- The string must:
  - Have a minimum length of 10
  - Have at least one uppercase letter
  - Have at least one lowercase letter
  - Have at least one number
  - Have at least one special character
  - Contain no other characters
- A local variable may be used to store the value of converting the string into a list
- Filter list 4 times to define a local variable for the elements of each of the four sets
- Each of these filtered lists must have a length greater than 0 and summing their lengths must equal the length of the given string

# Randomness

## Generating Random Passwords

- ```
  ;; string natnum → Boolean
  ;; Purpose: Determine of the given string is a pw
  (define (valid-passwd? str)
    (local [(define lststr (explode str))
            (define uc (filter (λ (s) (member s UCASE)) lststr))
            (define lc (filter (λ (s) (member s LCASE)) lststr))
            (define nc (filter (λ (s) (member s NUMBS)) lststr))
            (define sc (filter (λ (s) (member s SCHRS)) lststr))]
      (and (>= (string-length str) 10)
           (= (length lststr)
              (+ (length uc) (length lc)
                 (length nc) (length sc)))
           (> (length uc) 0)
           (> (length lc) 0)
           (> (length nc) 0)
           (> (length sc) 0))))

  ;; Tests using sample values for valid-passwd?
  (check-expect (valid-passwd? "?$Isdwer67G")   #true)
  (check-expect (valid-passwd? "Z!jwqx8t2sY32") #true)
  (check-expect (valid-passwd? "abcdefgh")      #false)
  (check-expect (valid-passwd? "a2cZdEfghJ8")   #false)
  ```
  ```

# Randomness

## HOMEWORK

- Problems: 3, 4

# Randomness

## Distributed Fortune Teller Game

- The fortune teller game provides a random fortune every time a player requests it
- Design and implement a distributed fortune teller game that allows multiple players to requests fortunes

# Randomness

## Distributed Fortune Teller Game

- The fortune teller game provides a random fortune every time a player requests it
- Design and implement a distributed fortune teller game that allows multiple players to requests fortunes
- The design recipe for distributed programming is:
  1. Divide the problem into components.
  2. Draft data definitions for the different components.
  3. Design a communication protocol.
  4. Design marshalling and unmarshalling functions.
  5. Design and implement the components.
  6. Test your program.

# Randomness

## Distributed Fortune Teller Game

- The fortune teller game provides a random fortune every time a player requests it
- Design and implement a distributed fortune teller game that allows multiple players to requests fortunes
- The design recipe for distributed programming is:
  1. Divide the problem into components.
  2. Draft data definitions for the different components.
  3. Design a communication protocol.
  4. Design marshalling and unmarshalling functions.
  5. Design and implement the components.
  6. Test your program.
- A message to be transmissible it must be a symbolic expressions:

```
A symbolic expression, sexpr, is either a:
  1. string
  2. symbol
  3. number
  4. Boolean
  5. character
  6 (listof sexpr)
```

# Randomness

## Distributed Fortune Teller Game: The Components

- Arbitrary number of players and a server
- All the players execute a copy of the same code but have a unique name
- Players: maintain world, draw world, process keystrokes, detect game end & process server messages

# Randomness

## Distributed Fortune Teller Game: The Components

- Arbitrary number of players and a server
- All the players execute a copy of the same code but have a unique name
- Players: maintain world, draw world, process keystrokes, detect game end & process server messages
- Player's `run` function is:

```
;; A name is a string or a symbol
;; name → world
(define (run a-name)
  (big-ban INIT-WORLD
           (on-draw draw-world) (on-key process-key)
           (stop-when finished? draw-final-world)
           (on-receive process-message)
           (register LOCALHOST) (name a-name)))
```

# Randomness

## Distributed Fortune Teller Game: The Components

- Arbitrary number of players and a server
- All the players execute a copy of the same code but have a unique name
- Players: maintain world, draw world, process keystrokes, detect game end & process server messages
- Player's `run` function is:

```
;; A name is a string or a symbol
;; name → world
(define (run a-name)
  (big-ban INIT-WORLD
           (on-draw draw-world) (on-key process-key)
           (stop-when finished? draw-final-world)
           (on-receive process-message)
           (register LOCALHOST) (name a-name)))
```

- Server: add players, remove players, and process player messages

# Randomness

## Distributed Fortune Teller Game: The Components

- Arbitrary number of players and a server
- All the players execute a copy of the same code but have a unique name
- Players: maintain world, draw world, process keystrokes, detect game end & process server messages
- Player's `run` function is:

```
;; A name is a string or a symbol
;; name → world
(define (run a-name)
  (big-ban INIT-WORLD
           (on-draw draw-world) (on-key process-key)
           (stop-when finished? draw-final-world)
           (on-receive process-message)
           (register LOCALHOST) (name a-name)))
```

- Server: add players, remove players, and process player messages
- Server's `run` function is:

```
;; Any → universe
;; Purpose: Run the universe server
(define (run-server a-z)
  (universe
    INIT-UNIV
    (on-new add-new-iworld)
    (on-msg process-message)
    (on-disconnect rm-iworld)))
```

# Randomness

## Distributed Fortune Teller Game: Player Definitions

- Scene constants:

```
(define ESCN-LEN   700)       (define ESCN-HEIGHT 200)
(define ESCN-COLOR 'darkblue) (define TXT-SIZE    36)
(define TXT-COLOR  'gold)     (define TXT-SIZE2   16)  (define TXT-COLOR2 'pink)
```

# Randomness

## Distributed Fortune Teller Game: Player Definitions

- Scene constants:

```
(define ESCN-LEN   700)       (define ESCN-HEIGHT 200)
(define ESCN-COLOR 'darkblue) (define TXT-SIZE    36)
(define TXT-COLOR  'gold)     (define TXT-SIZE2   16)  (define TXT-COLOR2 'pink)
```

- String constants for: instructions, when the player quits, and connection reject:

```
(define INSTR-STR "Press q to quit or any other key to get a fortune")
(define DONE-STR   "Bye...")
(define DENIED-STR "Connection denied: name is already in use.")
```

# Randomness

## Distributed Fortune Teller Game: Player Definitions

- Scene constants:

```
(define ESCN-LEN   700)       (define ESCN-HEIGHT 200)
(define ESCN-COLOR 'darkblue) (define TXT-SIZE    36)
(define TXT-COLOR  'gold)     (define TXT-SIZE2   16)  (define TXT-COLOR2 'pink)
```

- String constants for: instructions, when the player quits, and connection reject:

```
(define INSTR-STR "Press q to quit or any other key to get a fortune")
(define DONE-STR   "Bye...")
(define DENIED-STR "Connection denied: name is already in use.")
```

- Background and message images:

```
(define ESCN-BCKGRND (rectangle ESCN-LEN ESCN-HEIGHT 'solid ESCN-COLOR))
(define INSTR-IMG (text INSTR-STR TXT-SIZE2 TXT-COLOR2))
(define BYE-IMG   (text DONE-STR TXT-SIZE TXT-COLOR))
(define RJCT-IMG  (text DENIED-STR TXT-SIZE TXT-COLOR))
```

# Randomness

## Distributed Fortune Teller Game: Player Definitions

- Scene constants:

```
(define ESCN-LEN   700)       (define ESCN-HEIGHT 200)
(define ESCN-COLOR 'darkblue) (define TXT-SIZE    36)
(define TXT-COLOR  'gold)     (define TXT-SIZE2   16)  (define TXT-COLOR2 'pink)
```

- String constants for: instructions, when the player quits, and connection reject:

```
(define INSTR-STR "Press q to quit or any other key to get a fortune")
(define DONE-STR   "Bye...")
(define DENIED-STR "Connection denied: name is already in use.")
```

- Background and message images:

```
(define ESCN-BCKGRND (rectangle ESCN-LEN ESCN-HEIGHT 'solid ESCN-COLOR))
(define INSTR-IMG (text INSTR-STR TXT-SIZE2 TXT-COLOR2))
(define BYE-IMG   (text DONE-STR TXT-SIZE TXT-COLOR))
(define RJCT-IMG  (text DENIED-STR TXT-SIZE TXT-COLOR))
```

- Empty scenes: running, quitting, and rejecting

```
(define E-SCN
 (place-image INSTR-IMG (/ ESCN-LEN 2) (* 0.8 ESCN-HEIGHT) ESCN-BCKGRND))
(define E-SCN2
 (place-image BYE-IMG   (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) ESCN-BCKGRND))
(define E-SCN3
 (place-image RJCT-IMG  (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) ESCN-BCKGRND))
```

# Randomness

## Distributed Fortune Teller Game: Player Definitions

```
;; A world is a string of length less than 45.
;; Sample worlds
(define W1 "You will live long and prosper.")
(define W2 "Logic is the beginning of wisdom, not the end.")
(define W3 "The trial never ends")
(define W4 DONE-STR)  (define W5 DENIED-STR)
(define INIT-WORLD "")
```

# Randomness

## Distributed Fortune Teller Game: Player Definitions

```
;; A world is a string of length less than 45.
;; Sample worlds
(define W1 "You will live long and prosper.")
(define W2 "Logic is the beginning of wisdom, not the end.")
(define W3 "The trial never ends")
(define W4 DONE-STR)  (define W5 DENIED-STR)
(define INIT-WORLD "")
```

```
;; A game key, gk, is either: 1. "q"   2. any other key
;; sample gk
(define QUIT-GK "q")     (define OTHR-GK "z")
```

# Randomness

## Distributed Fortune Teller Game: Player Definitions

```
;; A world is a string of length less than 45.
;; Sample worlds
(define W1 "You will live long and prosper.")
(define W2 "Logic is the beginning of wisdom, not the end.")
(define W3 "The trial never ends")
(define W4 DONE-STR)  (define W5 DENIED-STR)
(define INIT-WORLD "")
```

```
;; A game key, gk, is either: 1. "q"   2. any other key
;; sample gk
(define QUIT-GK "q")     (define OTHR-GK "z")
```

```
#| Template for functions on a gk
;; gk ... → ...   Purpose:
(define (f-on-gk a-gk ...)
  (if (key=? a-gk QUIT-GK)
      ...
      ...))
;; Sample expressions for f-on-gk
(define QGK-VAL ...)  (define OGK-VAL ...)
;; Tests using sample computations for f-on-gk
(check-expect (f-on-gk QUIT-GK ...) QGK-VAL)
(check-expect (f-on-gk OTHR-GK ...) OGK-VAL) ...
;; Tests using sample values for f-on-gk
(check-expect (f-on-gk ... ...) QGK-VAL) ...  |#
```

# Randomness

## Distributed Fortune Teller Game: Server Definitions

- Database of fortunes

```
(define FORTUNES
        '("Do not violate the prime directive"
          "0.67 seconds is an eternity for an android"
          "Don't be left with nothing but your bones"
          "Stay down, honor has been served"
          "Reach for the final frontier"
          "I canna' change the laws of physics"
          "No one can summon the future"
          "You can use logic to justify almost anything"
          "There is a way out of every box"
          "Make it so"
          "Don't grieve if it is logical"
          "Live long and prosper"
          "Resistance is futile"
               .
               .))
```

# Randomness

## Distributed Fortune Teller Game: Server Definitions

- Database of fortunes

```
(define FORTUNES
        '("Do not violate the prime directive"
          "0.67 seconds is an eternity for an android"
          "Don't be left with nothing but your bones"
          "Stay down, honor has been served"
          "Reach for the final frontier"
          "I canna' change the laws of physics"
          "No one can summon the future"
          "You can use logic to justify almost anything"
          "There is a way out of every box"
          "Make it so"
          "Don't grieve if it is logical"
          "Live long and prosper"
          "Resistance is futile"
                .
                :))
```

- The server must track an arbitrary number of players:

```
;; A universe is a (listof iworld)

(define INIT-UNIV '())
(define UNIV2 (list iworld1))
```

# Randomness

## Distributed Fortune Teller Game: Communication Protocol

- Ask yourself when messages must be exchanged between a player and the server

# Randomness

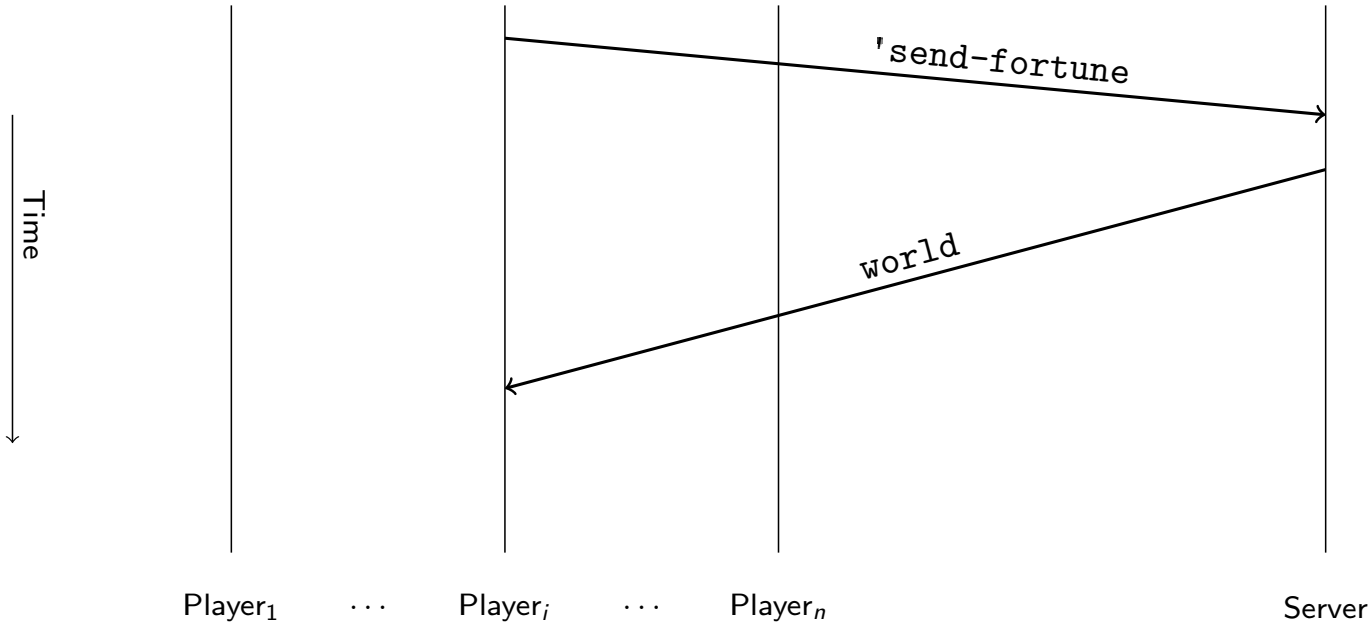
## Distributed Fortune Teller Game: Communication Protocol

- Ask yourself when messages must be exchanged between a player and the server
- An event that starts a communication chain is $\texttt{Player}_\texttt{i}$ requesting a fortune
- A message must be sent to the server indicating that a fortune is requested
- In turn, the server must send a fortune to $\texttt{Player}_\texttt{i}$

# Randomness

## Distributed Fortune Teller Game: Communication Protocol

- Ask yourself when messages must be exchanged between a player and the server
- An event that starts a communication chain is `Player`$_i$ requesting a fortune
- A message must be sent to the server indicating that a fortune is requested
- In turn, the server must send a fortune to `Player`$_i$
- Protocol Diagram




- The request from `Player`$_i$ is communicated by sending to the server the symbol `'send-fortune`
- The server communicates back by sending a fortune to `Player`$_i$

# Randomness

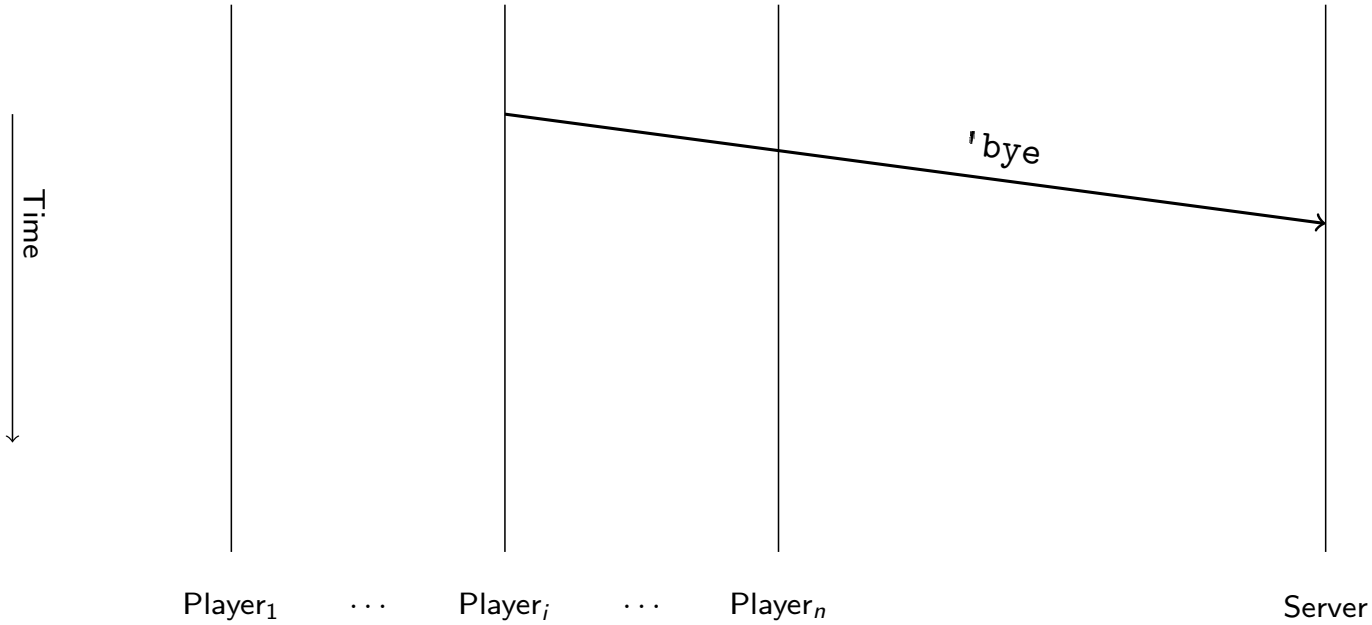
## Distributed Fortune Teller Game: Communication Protocol

- A communication chain is sparked when $\texttt{Player}_i$ quits the game
- A message informing the server must be sent

# Randomness

## Distributed Fortune Teller Game: Communication Protocol

- A communication chain is sparked when `Player`$_i$ quits the game
- A message informing the server must be sent



# Randomness

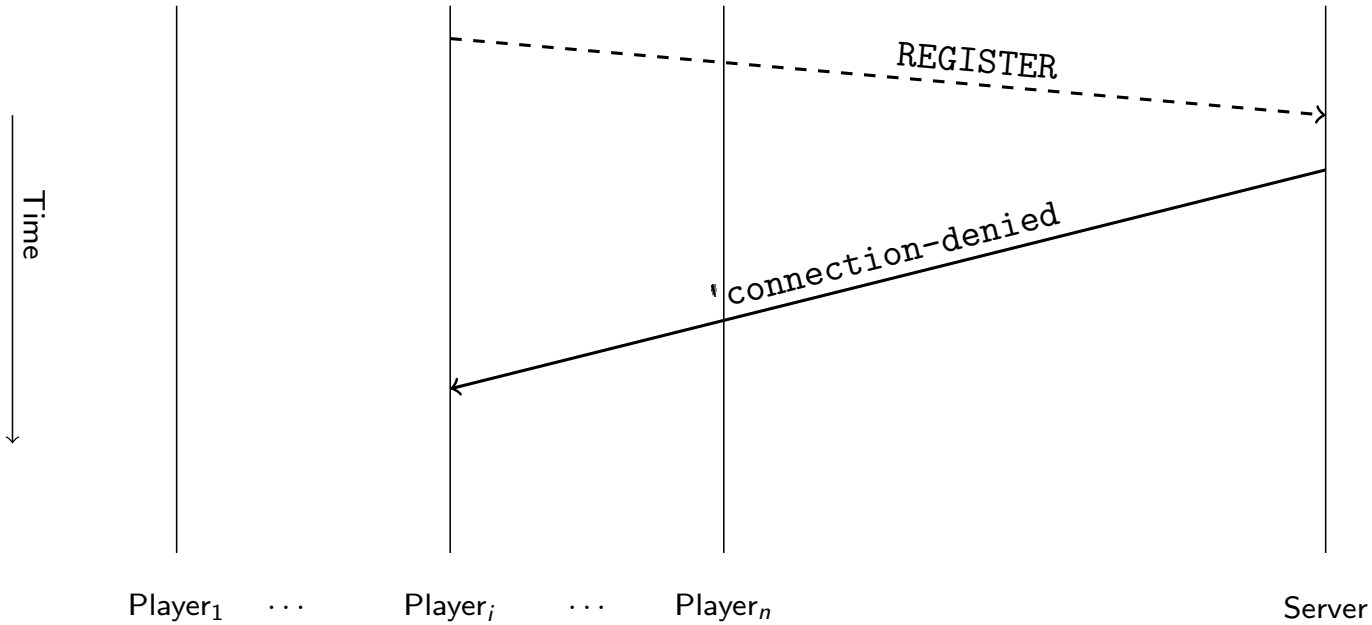
## Distributed Fortune Teller Game: Communication Protocol

- A communication chain is started when `Player`$_i$'s request to join the game is rejected
- This occurs when `Player`$_i$'s name is already in use by another player

# Randomness

## Distributed Fortune Teller Game: Communication Protocol

- A communication chain is started when `Player`$_i$'s request to join the game is rejected
- This occurs when `Player`$_i$'s name is already in use by another player



# Randomness

## Distributed Fortune Teller Game: Communication Protocol

- A to-server message is an enumerated type:

```
#| A to-server message, tsm, is either: 1. 'bye
                                         2. 'send-fortune  |#
```

```
;; Sample tsm
(define BYE 'bye)
(define SFT 'send-fortune)
```

# Randomness

## Distributed Fortune Teller Game: Communication Protocol

- A to-server message is an enumerated type:

```
#| A to-server message, tsm, is either: 1. 'bye
                                        2. 'send-fortune  |#


;; Sample tsm
(define BYE 'bye)
(define SFT 'send-fortune)
```

- The template for a function on a tsm is:

```
#|   ;; tsm ... → ...   Purpose:
     (define (f-on-tsm a-tsm ...)
       (if (eq? a-tsm 'bye)
           ...
           ...))
     ;; Sample expressions for f-on-tsm
     (define BYE-VAL ...)
     (define SFT-VAL ...)

     ;; Tests using sample computations for f-on-tsm
     (check-expect (f-on-tsm BYE ...) BYE-VAL)
     (check-expect (f-on-tsm SFT ...) SFT-VAL)
     ;; Tests using sample values for f-on-tsm
     (check-expect (f-on-tsm ... ...) ...)
```

# Randomness

## Distributed Fortune Teller Game: Communication Protocol

- A to-player message is an enumerated type:

```
#| A to-player message, tpm, is either: 1. 'connection-denied
                                        2. world              |#

   ;; Sample tpm
   (define CDEN 'connection-denied)
   (define WTPM W3)
```

# Randomness

## Distributed Fortune Teller Game: Communication Protocol

- A to-player message is an enumerated type:

```
#| A to-player message, tpm, is either: 1. 'connection-denied
                                        2. world              |#

;; Sample tpm
(define CDEN 'connection-denied)
(define WTPM W3)
```

- The template for a function on a tpm is:

```
#|   ;; tpm ... → ...
     ;; Purpose:
     (define (f-on-tpm a-tpm ...)
       (if (eq? a-tpm 'connection-denied)
           ...
           ...))
     ;; Sample expressions for f-on-tpm
     (define CDEN-VAL ...)
     (define WTFP-VAL ...)

     ;; Tests using sample computations for f-on-tpm
     (check-expect (f-on-tpm CDEN ...) CDEN-VAL)
     (check-expect (f-on-tpm WTPM ...) WTPM-VAL)
     ;; Tests using sample values for f-on-tpm
     (check-expect (f-on-tpm ... ...) ...)
```

# Randomness

## Distributed Fortune Teller Game Marshaling

- All `tsms` and `tpms` are `sexprs`

# Randomness

## Distributed Fortune Teller Game Marshaling

- All `tsms` and `tpms` are `sexprs`
- There is no need for marshalling and unmarshalling functions.

# Randomness

## Player Component: `draw-world`

- ```
  ;; Sample expressions for draw-world
  (define W1-IMG (place-image (text W1 TXT-SIZE TXT-COLOR)
                              (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
  (define W2-IMG (place-image (text W2 TXT-SIZE TXT-COLOR)
                              (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
  (define W3-IMG (place-image (text W3 TXT-SIZE TXT-COLOR)
                              (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
  ```
  ```

# Randomness

## Player Component: `draw-world`

- `;; world → image  Purpose: Draw the given world`
  `(define (draw-world a-world)`

- ```
  ;; Sample expressions for draw-world
  (define W1-IMG (place-image (text W1 TXT-SIZE TXT-COLOR)
                              (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
  (define W2-IMG (place-image (text W2 TXT-SIZE TXT-COLOR)
                              (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
  (define W3-IMG (place-image (text W3 TXT-SIZE TXT-COLOR)
                              (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
  ```
  ```

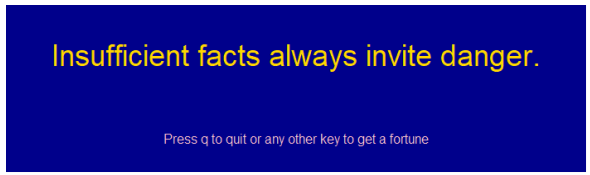
# Randomness

## Player Component: draw-world

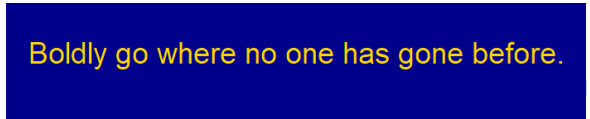
- ```
  ;; world → image    Purpose: Draw the given world
  (define (draw-world a-world)
  ```

- ```
  ;; Sample expressions for draw-world
  (define W1-IMG (place-image (text W1 TXT-SIZE TXT-COLOR)
                              (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
  (define W2-IMG (place-image (text W2 TXT-SIZE TXT-COLOR)
                              (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
  (define W3-IMG (place-image (text W3 TXT-SIZE TXT-COLOR)
                              (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
  ```

- ```
  ;; Tests using sample expressions for draw-world
  (check-expect (draw-world W1) W1-IMG) (check-expect (draw-world W2)
  (check-expect (draw-world W3) W3-IMG)
  ;; Tests using sample values for draw-world
  (check-expect (draw-world "Insufficient facts always invite danger.
  ```




```
  )
  (check-expect (draw-world "Boldly go where no one has gone before."
```

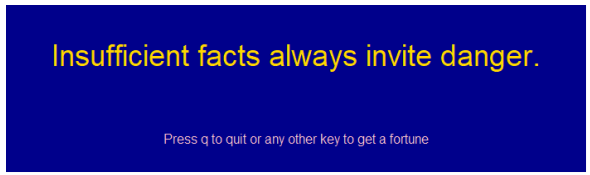

```

# Randomness

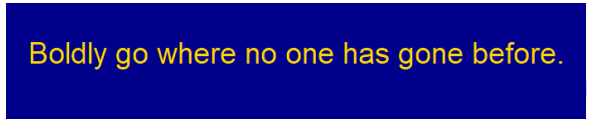
## Player Component: draw-world

```
;; world → image    Purpose: Draw the given world
(define (draw-world a-world)
  (place-image (text a-world TXT-SIZE TXT-COLOR)
               (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
;; Sample expressions for draw-world
(define W1-IMG (place-image (text W1 TXT-SIZE TXT-COLOR)
                            (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
(define W2-IMG (place-image (text W2 TXT-SIZE TXT-COLOR)
                            (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
(define W3-IMG (place-image (text W3 TXT-SIZE TXT-COLOR)
                            (/ ESCN-LEN 2) (* 0.3 ESCN-HEIGHT) E-SCN))
```

```
;; Tests using sample expressions for draw-world
(check-expect (draw-world W1) W1-IMG) (check-expect (draw-world W2)
(check-expect (draw-world W3) W3-IMG)
;; Tests using sample values for draw-world
(check-expect (draw-world "Insufficient facts always invite danger.
```




```
)
(check-expect (draw-world "Boldly go where no one has gone before."
```



# Randomness

## Distributed Fortune Teller Game: `process-key`

```
;; Sample expressions for process-key
(define W1Q-PK (make-package DONE-STR 'bye)) ;; "q"
(define W2Q-PK (make-package DONE-STR 'bye)) ;; "q"
(define W2*-PK (make-package W2       'send-fortune)) ;; "*"
(define W1a-PK (make-package W1       'send-fortune)) ;; "a"
```

# Randomness

## Distributed Fortune Teller Game: `process-key`

- ```
  ;; world key → package Purpose: Create next world after key e
  (define (process-key a-world a-key)
  ```

- ```
  ;; Sample expressions for process-key
  (define W1Q-PK (make-package DONE-STR 'bye)) ;; "q"
  (define W2Q-PK (make-package DONE-STR 'bye)) ;; "q"
  (define W2*-PK (make-package W2       'send-fortune)) ;; "*"
  (define W1a-PK (make-package W1       'send-fortune)) ;; "a"
  ```

# Randomness

## Distributed Fortune Teller Game: process-key

```
;; world key → package Purpose: Create next world after key e
(define (process-key a-world a-key)
```

```
;; Sample expressions for process-key
(define W1Q-PK (make-package DONE-STR 'bye)) ;; "q"
(define W2Q-PK (make-package DONE-STR 'bye)) ;; "q"
(define W2*-PK (make-package W2       'send-fortune)) ;; "*"
(define W1a-PK (make-package W1       'send-fortune)) ;; "a"
```

```
;; Tests using sample computations for process-key
(check-expect (process-key W1 "q") W1Q-PK)
(check-expect (process-key W2 "q") W1Q-PK)
(check-expect (process-key W2 "*") W2*-PK)
(check-expect (process-key W1 "a") W1a-PK)
;; Tests using sample values for process-key
(check-expect (process-key W3 "q")
              (make-package DONE-STR 'bye))
(check-expect (process-key W1 "z")
              (make-package W1 'send-fortune))
```

# Randomness

## Distributed Fortune Teller Game: process-key

```
;; world key → package Purpose: Create next world after key e
(define (process-key a-world a-key)
```

```
  (if (key=? a-key "q")
      (make-package DONE-STR 'bye)
      (make-package a-world  'send-fortune)))
```

```
;; Sample expressions for process-key
(define W1Q-PK (make-package DONE-STR 'bye)) ;; "q"
(define W2Q-PK (make-package DONE-STR 'bye)) ;; "q"
(define W2*-PK (make-package W2       'send-fortune)) ;; "*"
(define W1a-PK (make-package W1       'send-fortune)) ;; "a"
```

```
;; Tests using sample computations for process-key
(check-expect (process-key W1 "q") W1Q-PK)
(check-expect (process-key W2 "q") W1Q-PK)
(check-expect (process-key W2 "*") W2*-PK)
(check-expect (process-key W1 "a") W1a-PK)
;; Tests using sample values for process-key
(check-expect (process-key W3 "q")
              (make-package DONE-STR 'bye))
(check-expect (process-key W1 "z")
              (make-package W1 'send-fortune))
```

# Randomness

## Distributed Fortune Teller Game: process=message

```
;; Sample expression for process-message
(define PM-DN DENIED-STR)
(define PM-W2 "Warp speed now")
(define PM-W3 "This far and no further")
```

# Randomness

## Distributed Fortune Teller Game: process=message

- ```
  ;; world tpm → world
  ;; Purpose: Create new world after receiving the given to player me
  (define (process-message a-world a-tpm)
  ```

- ```
  ;; Sample expression for process-message
  (define PM-DN DENIED-STR)
  (define PM-W2 "Warp speed now")
  (define PM-W3 "This far and no further")
  ```

# Randomness

## Distributed Fortune Teller Game: process=message

- ```
  ;; world tpm → world
  ;; Purpose: Create new world after receiving the given to player me
  (define (process-message a-world a-tpm)
  ```

- ```
  ;; Sample expression for process-message
  (define PM-DN DENIED-STR)
  (define PM-W2 "Warp speed now")
  (define PM-W3 "This far and no further")
  ```

- ```
  ;; Test using sample computation for process-message
  (check-expect (process-message INIT-WORLD DENIED-STR) PM-DN)
  (check-expect (process-message W2 "Warp speed now") PM-W2)
  (check-expect (process-message W3 "This far and no further") PM-W3)
  ;; Test using sample value for process-message
  (check-expect (process-message "Make it so" "Resistance is futile")
                "Resistance is futile")
  (check-expect (process-message "It will all work out"
                                 "Time to let it go")
                "Time to let it go")
  ```
  ```

# Randomness

## Distributed Fortune Teller Game: process=message

```
;; world tpm → world
;; Purpose: Create new world after receiving the given to player me
(define (process-message a-world a-tpm)
```

```
  (if (eq? a-tpm 'connection-denied)
      DENIED-STR
      a-tpm))
```

```
;; Sample expression for process-message
(define PM-DN DENIED-STR)
(define PM-W2 "Warp speed now")
(define PM-W3 "This far and no further")
```

```
;; Test using sample computation for process-message
(check-expect (process-message INIT-WORLD DENIED-STR) PM-DN)
(check-expect (process-message W2 "Warp speed now") PM-W2)
(check-expect (process-message W3 "This far and no further") PM-W3)
;; Test using sample value for process-message
(check-expect (process-message "Make it so" "Resistance is futile")
              "Resistance is futile")
(check-expect (process-message "It will all work out"
                               "Time to let it go")
              "Time to let it go")
```

# Randomness

## Distributed Fortune Teller Game: `finished?`

```
;; Sample expressions for finished?
(define NFNSHD (string=? W1 DONE-STR))
(define FNSHD  (string=? W4 DONE-STR))
(define DND    (string=? W5 DENIED-STR))
(define NDND   (string=? W3 DENIED-STR))
```

# Randomness

## Distributed Fortune Teller Game: `finished?`

- ```
  ;; world → Boolean
  ;; Purpose: Determine if the simulation has ended
  (define (finished? a-world)
  ```

- ```
  ;; Sample expressions for finished?
  (define NFNSHD (string=? W1 DONE-STR))
  (define FNSHD  (string=? W4 DONE-STR))
  (define DND    (string=? W5 DENIED-STR))
  (define NDND   (string=? W3 DENIED-STR))
  ```

# Randomness

## Distributed Fortune Teller Game: `finished?`

- ```
  ;; world → Boolean
  ;; Purpose: Determine if the simulation has ended
  (define (finished? a-world)
  ```

- ```
  ;; Sample expressions for finished?
  (define NFNSHD (string=? W1 DONE-STR))
  (define FNSHD  (string=? W4 DONE-STR))
  (define DND    (string=? W5 DENIED-STR))
  (define NDND   (string=? W3 DENIED-STR))
  ```

- ```
  ;; Tests using sample computations for finished?
  (check-expect (finished? W1) NFNSHD)
  (check-expect (finished? W4) FNSHD)
  (check-expect (finished? W5) DND)
  (check-expect (finished? W3) NDND)

  ;; Tests using sample values for finished?
  (check-expect (finished? DONE-STR)   #true)
  (check-expect (finished? "Stop being highly illogical") #false)
  (check-expect (finished? DENIED-STR) #true)
  ```
  ```

# Randomness

## Distributed Fortune Teller Game: `finished?`

```
;; world → Boolean
;; Purpose: Determine if the simulation has ended
(define (finished? a-world)
  (or (string=? a-world DONE-STR)
      (string=? a-world DENIED-STR)))
```

```
;; Sample expressions for finished?
(define NFNSHD (string=? W1 DONE-STR))
(define FNSHD  (string=? W4 DONE-STR))
(define DND    (string=? W5 DENIED-STR))
(define NDND   (string=? W3 DENIED-STR))
```

```
;; Tests using sample computations for finished?
(check-expect (finished? W1) NFNSHD)
(check-expect (finished? W4) FNSHD)
(check-expect (finished? W5) DND)
(check-expect (finished? W3) NDND)

;; Tests using sample values for finished?
(check-expect (finished? DONE-STR)   #true)
(check-expect (finished? "Stop being highly illogical") #false)
(check-expect (finished? DENIED-STR) #true)
```

# Randomness

## Distributed Fortune Teller Game: `draw-final-world`

- ```
  ;; Sample expression for draw-final-world
  (define DFW1 E-SCN2) ;; W4 player quit world
  (define DFW2 E-SCN3) ;; W5 connection denied world
  ```
  ```

# Randomness

## Distributed Fortune Teller Game: `draw-final-world`

- ```
  ;; world → image   Purpose: To draw the last world
  ;; ASSUMPTION: Given world is either W4 or W5
  (define (draw-final-world a-world)
  ```

- ```
  ;; Sample expression for draw-final-world
  (define DFW1 E-SCN2) ;; W4 player quit world
  (define DFW2 E-SCN3) ;; W5 connection denied world
  ```

# Randomness

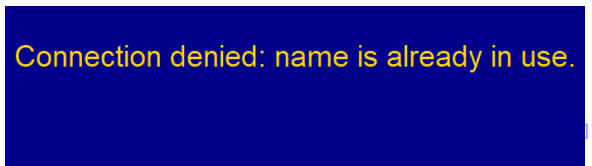
## Distributed Fortune Teller Game: draw-final-world

```
;; world → image   Purpose: To draw the last world
;; ASSUMPTION: Given world is either W4 or W5
(define (draw-final-world a-world)
```

```
;; Sample expression for draw-final-world
(define DFW1 E-SCN2) ;; W4 player quit world
(define DFW2 E-SCN3) ;; W5 connection denied world
```

```
;; Tests using sample computations for draw-final-world
(check-expect (draw-final-world W4) DFW1)
(check-expect (draw-final-world W5) DFW2)
;; Tests using sample values for draw-final-world
(check-expect (draw-final-world W4)
```


```
)
(check-expect (draw-final-world W5)
```




```
)
```

# Randomness

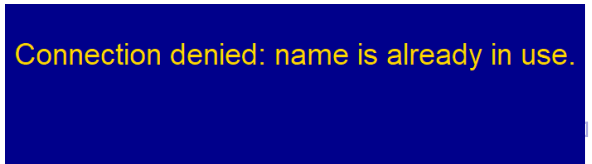
## Distributed Fortune Teller Game: draw-final-world

```
;; world → image   Purpose: To draw the last world
;; ASSUMPTION: Given world is either W4 or W5
(define (draw-final-world a-world)
  (if (string=? DONE-STR a-world)
      E-SCN2
      E-SCN3))
```

```
;; Sample expression for draw-final-world
(define DFW1 E-SCN2) ;; W4 player quit world
(define DFW2 E-SCN3) ;; W5 connection denied world
```

```
;; Tests using sample computations for draw-final-world
(check-expect (draw-final-world W4) DFW1)
(check-expect (draw-final-world W5) DFW2)
;; Tests using sample values for draw-final-world
(check-expect (draw-final-world W4)
```


```
)
(check-expect (draw-final-world W5)
```




```
)
```

# Randomness

## Distributed Fortune Teller Game: `add-new-player`

- ```
  ;; Sample expressions for add-new-world
  (define ADD1 (make-bundle (cons iworld1 INIT-UNIV) '() '()))
  (define ADD2 (make-bundle (cons iworld2 UNIV2) '() '()))
  (define ADD3 (make-bundle UNIV2
                            (list (make-mail iworld1 'connection-deni
                            '()))
  ```
  ```

# Randomness

## Distributed Fortune Teller Game: add-new-player

- ```
  ;; universe iworld → bundle Purpose: Add world universe
  (define (add-new-iworld u iw)
  ```

- ```
  ;; Sample expressions for add-new-world
  (define ADD1 (make-bundle (cons iworld1 INIT-UNIV) '() '()))
  (define ADD2 (make-bundle (cons iworld2 UNIV2) '() '()))
  (define ADD3 (make-bundle UNIV2
                            (list (make-mail iworld1 'connection-deni
                            '())))
  ```

# Randomness

## Distributed Fortune Teller Game: add-new-player

- ```
  ;; universe iworld → bundle Purpose: Add world universe
  (define (add-new-iworld u iw)
  ```

- ```
  ;; Sample expressions for add-new-world
  (define ADD1 (make-bundle (cons iworld1 INIT-UNIV) '() '()))
  (define ADD2 (make-bundle (cons iworld2 UNIV2) '() '()))
  (define ADD3 (make-bundle UNIV2
                            (list (make-mail iworld1 'connection-deni
                            '())))
  ```

- ```
  ;; Tests using sample computation for add-new-world
  (check-expect (add-new-iworld INIT-UNIV iworld1) ADD1)
  (check-expect (add-new-iworld UNIV2     iworld2) ADD2)
  (check-expect (add-new-iworld UNIV2     iworld1) ADD3)
  ;; Tests using sample values for add-new-world
  (check-expect (add-new-iworld (list iworld2) iworld3)
                (make-bundle (list iworld3 iworld2) '() '()))
  (check-expect (add-new-iworld (list iworld2 iworld3) iworld3)
                (make-bundle (list iworld2 iworld3)
                             (list (make-mail iworld3 'connection-der
                             '()))
  ```
  ```

# Randomness

## Distributed Fortune Teller Game: add-new-player

```
;; universe iworld → bundle Purpose: Add world universe
(define (add-new-iworld u iw)
  (if (member? (iworld-name iw) (map iworld-name u))
      (make-bundle u
                   (list (make-mail iw 'connection-denied))
                   '())
      (make-bundle (cons iw u) '() '())))
```

```
;; Sample expressions for add-new-world
(define ADD1 (make-bundle (cons iworld1 INIT-UNIV) '() '()))
(define ADD2 (make-bundle (cons iworld2 UNIV2) '() '()))
(define ADD3 (make-bundle UNIV2
                          (list (make-mail iworld1 'connection-deni
                          '())))
```

```
;; Tests using sample computation for add-new-world
(check-expect (add-new-iworld INIT-UNIV iworld1) ADD1)
(check-expect (add-new-iworld UNIV2     iworld2) ADD2)
(check-expect (add-new-iworld UNIV2     iworld1) ADD3)
;; Tests using sample values for add-new-world
(check-expect (add-new-iworld (list iworld2) iworld3)
              (make-bundle (list iworld3 iworld2) '() '()))
(check-expect (add-new-iworld (list iworld2 iworld3) iworld3)
              (make-bundle (list iworld2 iworld3)
                           (list (make-mail iworld3 'connection-der
                           '())))
```

# Randomness

## Distributed Fortune Teller Game: `rm-iworld`

- ```
  ;; Sample expressions for rm-iworld
  (define RM1 (filter (λ (w) (not (equal? w iworld2))) UNIV2))
  (define RM2 (filter (λ (w) (not (equal? w iworld1))) INIT-UNIV))
  ```
  ```

# Randomness

## Distributed Fortune Teller Game: `rm-iworld`

- ```
  ;; universe iworld → universe
  ;; Purpose: Remove the given iworld from the given universe
  (define (rm-iworld u iw)
  ```

- ```
  ;; Sample expressions for rm-iworld
  (define RM1 (filter (λ (w) (not (equal? w iworld2))) UNIV2))
  (define RM2 (filter (λ (w) (not (equal? w iworld1))) INIT-UNIV))
  ```

# Randomness

## Distributed Fortune Teller Game: `rm-iworld`

- ```
  ;; universe iworld → universe
  ;; Purpose: Remove the given iworld from the given universe
  (define (rm-iworld u iw)
  ```

- ```
  ;; Sample expressions for rm-iworld
  (define RM1 (filter (λ (w) (not (equal? w iworld2))) UNIV2))
  (define RM2 (filter (λ (w) (not (equal? w iworld1))) INIT-UNIV))
  ```

- ```
  ;; Tests using sample computations for rm-iworld
  (check-expect (rm-iworld UNIV2 iworld2) RM1)
  (check-expect (rm-iworld UNIV2 iworld1) RM2)

  ;; Tests using sample values for rm-iworld
  (check-expect (rm-iworld (list iworld1 iworld2 iworld3) iworld2)
                (list iworld1 iworld3))
  ```
  ```

# Randomness

## Distributed Fortune Teller Game: `rm-iworld`

- ```
  ;; universe iworld → universe
  ;; Purpose: Remove the given iworld from the given universe
  (define (rm-iworld u iw)
  ```
- ```
    (filter (λ (w) (not (equal? w iw))) u))
  ```
- ```
  ;; Sample expressions for rm-iworld
  (define RM1 (filter (λ (w) (not (equal? w iworld2))) UNIV2))
  (define RM2 (filter (λ (w) (not (equal? w iworld1))) INIT-UNIV))
  ```
- ```
  ;; Tests using sample computations for rm-iworld
  (check-expect (rm-iworld UNIV2 iworld2) RM1)
  (check-expect (rm-iworld UNIV2 iworld1) RM2)

  ;; Tests using sample values for rm-iworld
  (check-expect (rm-iworld (list iworld1 iworld2 iworld3) iworld2)
                (list iworld1 iworld3))
  ```

# Randomness

## Distributed Fortune Teller Game: process-message

- ```
  ;; Sample expressions for process-message
  (define PM1 (make-bundle (filter (λ (w)
                                     (not (equal? (iworld-name w)
                                                  (iworld-name iworld
                                   UNIV2)
                           '()
                           (list iworld1)))
  (define PM2 (make-bundle UNIV2
                           (list (make-mail
                                  iworld1
                                  (list-ref FORTUNES
                                            (random (length FORTUNES
                           '())))
  ```
  ```

# Randomness

## Distributed Fortune Teller Game: process-message

- ```
  ;; universe iworld tsm → bundle  Purpose: Process to-server messag
  (define (process-message u iw m)
  ```

- ```
  ;; Sample expressions for process-message
  (define PM1 (make-bundle (filter (λ (w)
                                     (not (equal? (iworld-name w)
                                                  (iworld-name iworld
                                   UNIV2)
                           '()
                           (list iworld1)))
  (define PM2 (make-bundle UNIV2
                           (list (make-mail
                                  iworld1
                                  (list-ref FORTUNES
                                            (random (length FORTUNES
                           '()))
  ```

# Randomness

## Distributed Fortune Teller Game: process-message

```
;; universe iworld tsm → bundle  Purpose: Process to-server messag
(define (process-message u iw m)
  (if (eq? m 'bye)
      (make-bundle (rm-iworld u iw)
                   '()
                   (list iw))
      (make-bundle u
                   (list (make-mail
                          iw (list-ref FORTUNES
                                       (random (length FORTUNES))))
                   '())))
;; Sample expressions for process-message
(define PM1 (make-bundle (filter (λ (w)
                                   (not (equal? (iworld-name w)
                                                (iworld-name iworld
                                 UNIV2)
                         '()
                         (list iworld1)))
(define PM2 (make-bundle UNIV2
                         (list (make-mail
                                iworld1
                                (list-ref FORTUNES
                                          (random (length FORTUNES
                         '())))
```

# Randomness

## Distributed Fortune Teller Game: process-message

```
;; Tests using sample computations for process-message
(check-expect (process-message UNIV2 iworld1 'bye) PM1)
(check-random (process-message UNIV2 iworld1 'send-fortune)
              (make-bundle UNIV2                    Run the game!
                           (list (make-mail
                                  iworld1
                                  (list-ref FORTUNES
                                            (random (length FORTUN
                           '())))
;; Tests using sample values for process-message
(check-random
  (process-message (list iworld3 iworld2) iworld2 'send-fortune)
  (make-bundle (list iworld3 iworld2)
               (list (make-mail iworld2
                                (list-ref FORTUNES
                                          (random (length FORTUNES)
               '())))
(check-random
  (process-message (list iworld1 iworld2) iworld1 'send-fortune)
  (make-bundle (list iworld1 iworld2)
               (list (make-mail iworld1
                                (list-ref FORTUNES
                                          (random (length FORTUNES)
               '())))
```

# Randomness

## HOMEWORK

- Problem: 7
- Bonus Double Quiz: 6 (due in one week)