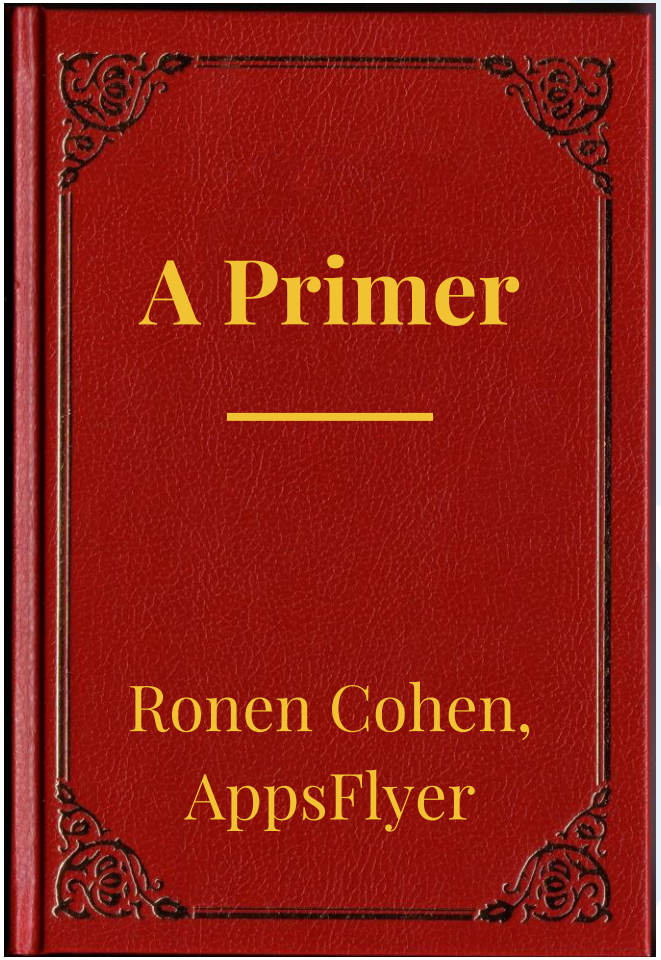
Thinking Functionally



AppsFlyer

Ronen Cohen

Senior Tech Lead

Data Platform, AppsFlyer

@nenorbot

Meet AppsFlyer

Mobile Analytics & Attribution Platform

15
Offices Worldwide

650+
Employees

70k+
Clients

3500+
Integrated Partners

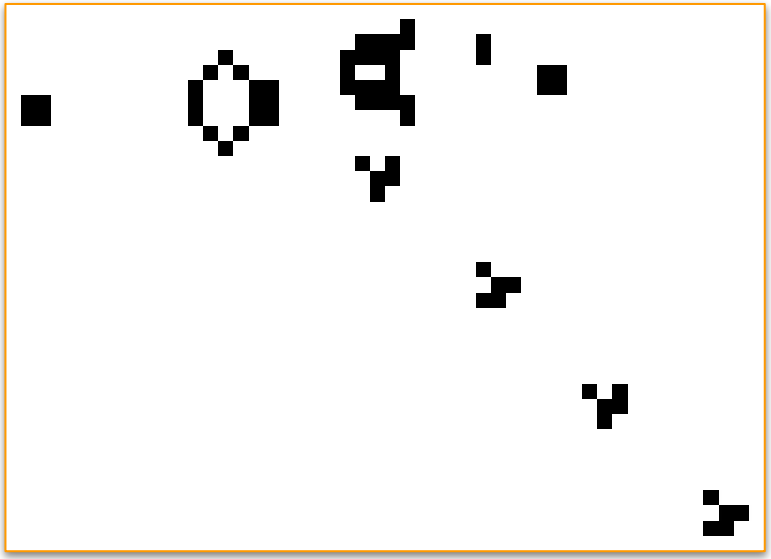
70%
Market share

Thinking Functionally

- Declarative over Imperative
- Pure functions
- Immutability
- Functions as first-class citizens
- Composition

AppsFlyer

Conway's Game of Life



AppsFlyer

Rules

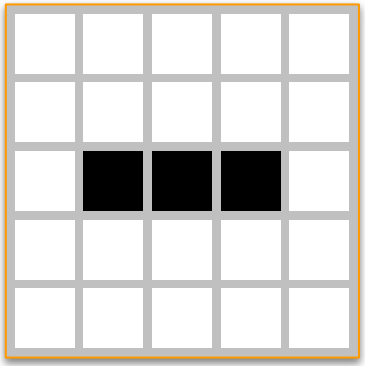
Any Live Cell With:

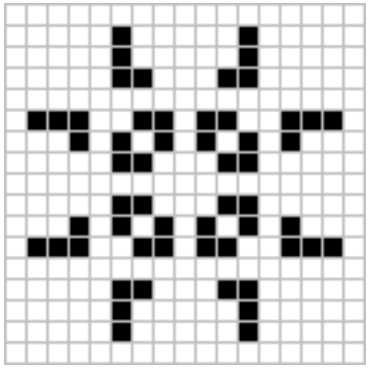
- **Fewer** than two live neighbours dies, as if by underpopulation.
- **Two or three** live neighbours lives on to the next generation.
- **More than three** live neighbours dies, as if by overpopulation.

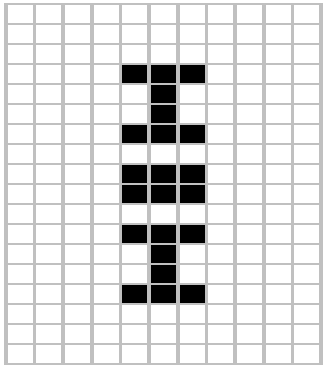
OR

- **Any dead cell** with exactly three live neighbours becomes a live cell, as if by reproduction.

AppsFlyer







AppsFlyer

Rules, as Code

AppsFlyer

```python
DEAD = False

ALIVE = True

def get_next_cell_value(is_cell_alive, live_neighbors_count):
    is_alive = is_cell_alive and (live_neighbors_count == 2 or live_neighbors_count == 3)
               or live_neighbors_count == 3

    return is_alive
```

Pure Functions

- Always return the same result given the same arguments
- No side-effects:
  - No I/O
    - No Files / devices / network / etc
    - No Randomness
    - No Time
  - No dependence on state

AppsFlyer

Pure Functions

- Simple to reason about
- Simple to test
- Parallelizable
- Composable

AppsFlyer

```python
def to_uppercase(s):
    return s.upper()

def reverse(s):
    return s[::-1]

reverse_and_uppercase = lambda s: reverse(to_uppercase(s))

reverse_and_uppercase("hello") => "OLLEH"
```

Transitioning the Board

AppsFlyer

```python
DEAD = False

ALIVE = True

def get_next_cell_value(is_cell_alive, live_neighbors_count):
    is_alive = is_cell_alive and (live_neighbors_count == 2 or live_neighbors_count == 3)
               or live_neighbors_count == 3

    return is_alive
```

```
def transition(board):
    for x in xrange(BOARD_SIZE):
        for y in xrange(BOARD_SIZE):
            curr_cell_value = board[x][y]
            live_neighbors_count = get_live_neighbors_count(board, x, y)
            is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
            board[x][y] = is_alive
```

```
def transition(board):
    for x in xrange(BOARD_SIZE):
        for y in xrange(BOARD_SIZE):
            curr_cell_value = board[x][y]
            live_neighbors_count = get_live_neighbors_count(board, x, y)
            is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
            board[x][y] = is_alive
```

```
def transition(board):
    for x in xrange(BOARD_SIZE):
        for y in xrange(BOARD_SIZE):
            curr_cell_value = board[x][y]
            live_neighbors_count = get_live_neighbors_count(board, x, y)
            is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
            board[x][y] = is_alive
```

```
def transition(board):
    for x in xrange(BOARD_SIZE):
        for y in xrange(BOARD_SIZE):
            curr_cell_value = board[x][y]
            live_neighbors_count = get_live_neighbors_count(board, x, y)
            is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
            board[x][y] = is_alive
```

```
def transition(board):
    for x in xrange(BOARD_SIZE):
        for y in xrange(BOARD_SIZE):
            curr_cell_value = board[x][y]
            live_neighbors_count = get_live_neighbors_count(board, x, y)
            is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
            board[x][y] = is_alive
```

```
def transition(board):
    for x in xrange(BOARD_SIZE):
        for y in xrange(BOARD_SIZE):
            curr_cell_value = board[x][y]
            live_neighbors_count = get_live_neighbors_count(board, x, y)
            is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
            board[x][y] = is_alive
```

- **Mutation:**
  - Introduces a layer of complexity
  - Concurrency is harder
- **Immutability**
  - Inefficient?
    - Depends on implementation
    - Persistent data-structures (i.e, Clojure)

AppsFlyer

Transitioning the Board

 Take Two

AppsFlyer

```
def transition(board):
    new_board = []
    for x in xrange(BOARD_SIZE):
        row = []
        new_board.append(row)
        for y in xrange(BOARD_SIZE):
            curr_cell_value = board[x][y]
            live_neighbors_count = get_live_neighbors_count(board, x, y)
            is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
            row.append(is_alive)
    return new_board
```

```
def transition(board):
    new_board = []
    for x in xrange(BOARD_SIZE):
        row = []
        new_board.append(row)
        for y in xrange(BOARD_SIZE):
            curr_cell_value = board[x][y]
            live_neighbors_count = get_live_neighbors_count(board, x, y)
            is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
            row.append(is_alive)
    return new_board
```

Transitioning the Board

Take Three

AppsFlyer

```
def make_board(board_size, cell_value_fn):
    new_board = []
    for x in xrange(board_size):
        row = []
        new_board.append(row)
        for y in xrange(board_size):
            is_alive = cell_value_fn(x, y)
            row.append(is_alive)
    return new_board

def transition(board):
    def next_cell_value_fn(x, y):
        curr_cell_value = board[x][y]
        live_neighbors_count = get_live_neighbors_count(board, x, y)
        is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
        return is_alive

    return make_board(BOARD_SIZE, next_cell_value_fn)
```

```
def make_board(board_size, cell_value_fn):
    new_board = []
    for x in xrange(board_size):
        row = []
        new_board.append(row)
        for y in xrange(board_size):
            is_alive = cell_value_fn(x, y)
            row.append(is_alive)
    return new_board

def transition(board):
    def next_cell_value_fn(x, y):
        curr_cell_value = board[x][y]
        live_neighbors_count = get_live_neighbors_count(board, x, y)
        is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
        return is_alive

    return make_board(BOARD_SIZE, next_cell_value_fn)
```

```
def make_board(board_size, cell_value_fn):
    new_board = []
    for x in xrange(board_size):
        row = []
        new_board.append(row)
        for y in xrange(board_size):
            is_alive = cell_value_fn(x, y)
            row.append(is_alive)
    return new_board

def transition(board):
    def next_cell_value_fn(x, y):
        curr_cell_value = board[x][y]
        live_neighbors_count = get_live_neighbors_count(board, x, y)
        is_alive = get_next_cell_value(curr_cell_value, live_neighbors_count)
        return is_alive

    return make_board(BOARD_SIZE, next_cell_value_fn)
```

Functions are First-Class citizens

- Functions are first-class citizens:
  - Can be passed as arguments.
  - Can be returned from other functions.
- Declarative rather than imperative.

```
map(lambda x: x*x, [1,2,3,4,5]) => [1, 4, 9, 16, 25]

filter(lambda x: x % 2 == 0, [1, 4, 9, 16, 25]) => [4, 16]
```

```
def game(init_board, num_of_generations):
    board = init_board
    for i in xrange(num_of_generations):
        print_board(board)
        board = transition(board)
```

```
def game(init_board, num_of_generations):
    boards = []
    board = init_board
    for i in xrange(num_of_generations):
        boards.append(board)
        board = transition(board)
    return boards

def print_game(boards):
    for b in boards:
        print_board(b)
```

```
def game(init_board, num_of_generations):
    boards = []
    board = init_board
    for i in xrange(num_of_generations):
        boards.append(board)
        board = transition(board)
    return boards

def find_overpopulated_boards(boards):
    overpopulated = filter(lambda board: get_live_cells_count(board) > N, boards)
    return overpopulated
```

```
def game(init_board, num_of_generations):
    boards = []
    board = init_board
    for i in xrange(num_of_generations):
        boards.append(board)
        board = transition(board)
    return boards

def find_overpopulated_boards(boards):
    overpopulated = filter(lambda board: get_live_cells_count(board) > N, boards)
    return overpopulated

def print_game(boards): ...
def save_game_to_file(boards): ...
def send_game_to_phone(boards): ...
def flip_game(boards): ...
```

- Declarative over Imperative
- Pure functions
- Immutability
- Functions as first-class citizens
- Composition

AppsFlyer

Thank You!

ronen.cohen@appsflyer.com
@nenorbot

https://github.com/nenorbot/devweeknyc2019