

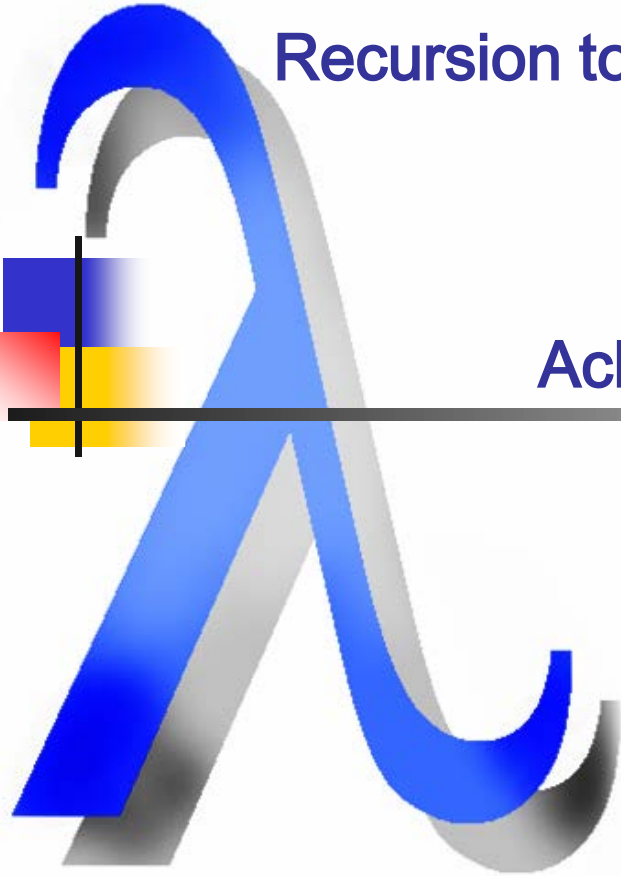
The Time of Space Invaders Will Come to Pass

A CS1 Functional Video Game Journey from Structural
Recursion to Generative and Accumulative Recursion

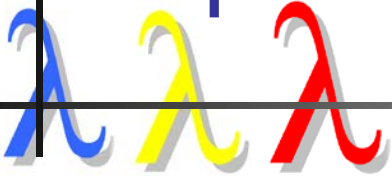
or

Achieving the Impossible!

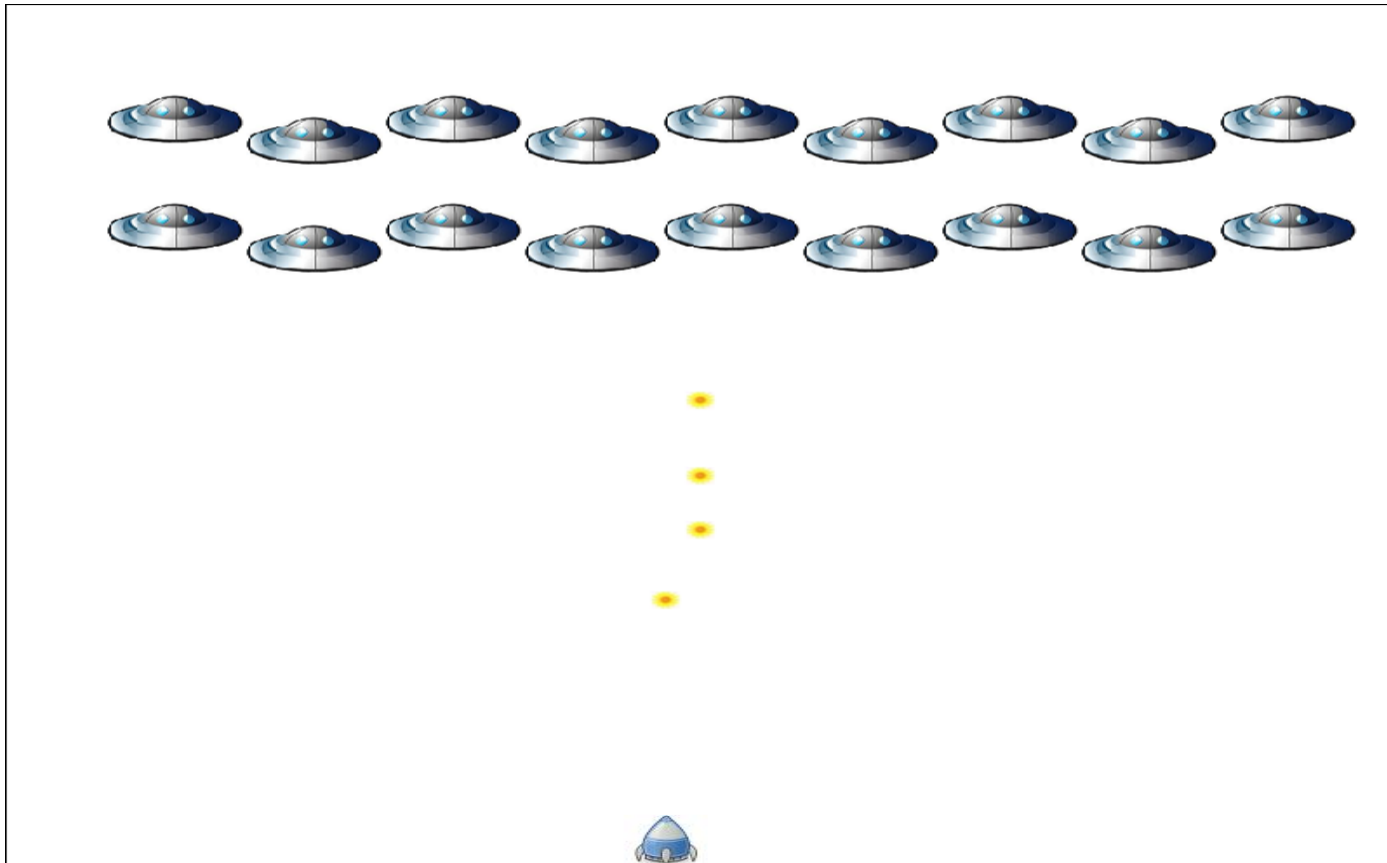
Marco T. Morazán
Seton Hall University



Space Invaders by CS1 Students



- What do we do after students program Space Invaders?



Functional Video Games & CS1



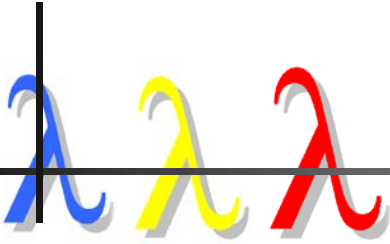
- Functional Video Games in CS1
 - Flourishing trend
 - Used by universities and high schools
 - Program by Design using *How to Design Programs*
- Why is this successful?
 - Students get excited and can be creative
 - Students can go home and brag about what they have done!
 - No need to reason about state
 - Students learn to reason algorithms into existence using *Design Recipes*

Functional Video Games & CS1



- Design Recipe
 - Steps to follow to design functions based
 - From blank screen to working solution
- A great deal of examples using structural recursion
 - Space Invaders, Snake, Putting out fires
- There is more than structural recursion
- How to transition to Generative and Accumulative Recursion?
 - Harness the enthusiasm for video games
 - Reinforce lessons on structural recursion and abstraction
 - Few examples using video games in the literature

The N-Puzzle



Example

1	3	7
4		6
2	5	8

HELP ME!

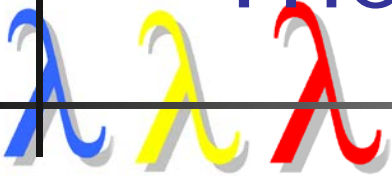


1	2	3
4	5	6
7	8	

HELP ME!

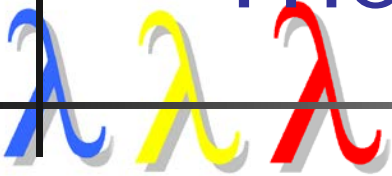
- Universal, easy to understand, easy to scale
- Help button to rescue those that are stuck
- Demonstrate that informed heuristic search strategies are within the grasp of CS1 students

The First Encounter in Class



- Students have studied
 - primitive data
 - structures
 - structural recursion (e.g., on lists, trees, and natural numbers)
 - abstraction (e.g. map, filter, build-list, and other basic HOFs)
 - Have implemented Space Invaders or Snake or ...

The First Encounter in Class



- What is changing in the game? How can it be represented?

A board is either:

(use BNF grammar????)

1. empty
2. (cons number b), where b is a board

Template for functions on boards:

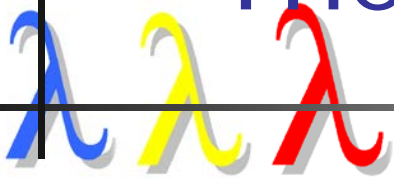
```
(define (f-on-board a-board)
```

```
  (cond [(empty? a-board) ...]
```

```
        [else ...(first a-board)...(rest a-board)]))
```

- Brings the game into familiar territory!

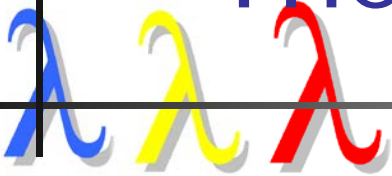
The First Encounter in Class



- To get started ask students to perform task that are familiar
 - reinforce lessons on structural recursion and abstraction

```
(define WIN (build-list N (lambda (n)
                            (cond [(< n (- N 1)) (+ n 1)]
                                  [else 0]))))
```

The First Encounter in Class



- To get started ask students to perform task that are familiar
 - reinforce lessons on structural recursion and abstraction

; get-blank-pos: board → number

; Purpose: To find the position of the blank

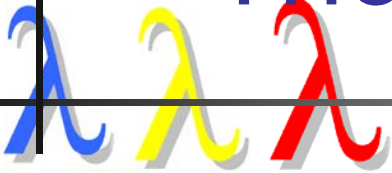
```
(define (get-blank-pos l)
```

```
  (cond [(empty? l) (error 'get-blank-pos "Blank not found")]
```

```
        [(= (car l) BLANK) 0]
```

```
        [else (add1 (get-blank-pos (cdr l)))]))
```

The First Encounter in Class



- To get started ask students to perform task that are familiar
 - reinforce lessons on structural recursion and abstraction

; swap-tiles: board natnum natnum → board

; Purpose: To swap the given tiles in the given board

```
(define (swap-tiles w i j)
```

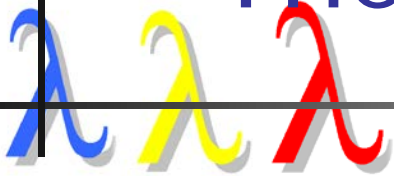
```
  (build-list N (lambda (n)
```

```
    (cond [(= n i) (list-ref w j)]
```

```
          [(= n j) (list-ref w i)]
```

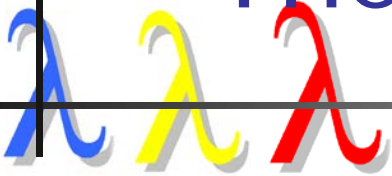
```
          [else (list-ref w n)]))))
```

The First Encounter in Class



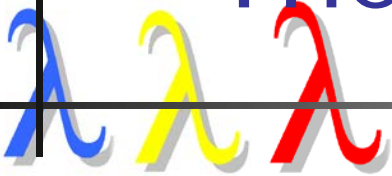
- What does it mean to find a solution when the help button is hit?

The First Encounter in Class



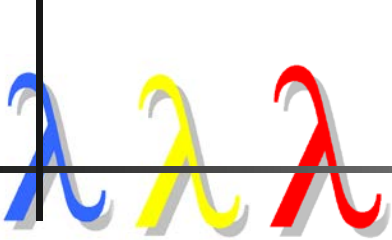
- What does it mean to find a solution when the help button is hit?
 - Find a sequence of moves from **b** to **WIN**
 - Find a solution from a successor of **b** to **WIN** and add move from **b** to the successor of **b**
 - Students easily see that recursion is required

The First Encounter in Class



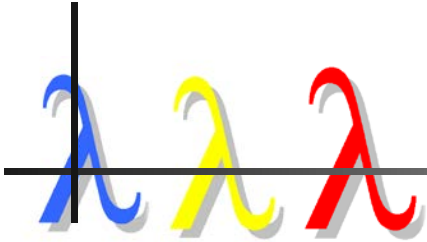
- What does it mean to find a solution when the help button is hit?
 - Find a sequence of moves from **b** to **WIN**
 - Find a solution from a successor of **b** to **WIN** and add move from **b** to the successor of **b**
 - Students easily see that recursion is required
- How do you select a successor of **b**?
 - Students have reasoned their way into generative recursion
 - The sub-problem is not based on the structure of **b** (nor is smaller)

Finding a Solution



- Selecting a successor
 - introduce students to heuristics
 - estimate how many moves to **WIN**
 - pick best successor
 - hope it leads to **WIN**
- The Manhattan distance of a board is the sum of how far away each tile is from its correct position
 - structural recursion on natural numbers

Finding a Solution



; manhattan-distance: board --> number

; Purpose: To compute the Manhattan distance of the given board

(define (manhattan-distance b)

(local

[; distance: number number --> number

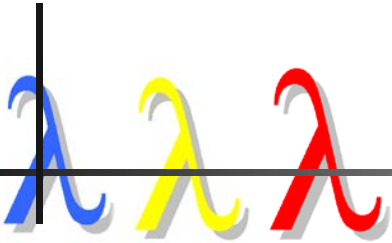
; Purpose: To compute the distance between the two tile positions

(define (distance curr corr)

(+ (abs (- (quotient curr (sqrt N)) (quotient corr (sqrt N))))

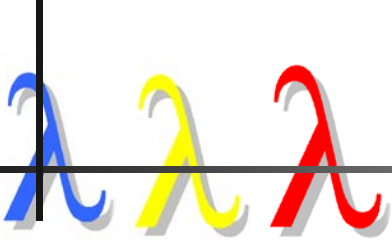
(abs (- (remainder curr (sqrt N)) (remainder corr (sqrt N))))))

Finding a Solution



```
; manhattan-distance: board --> number
; Purpose: To compute the Manhattan distance of the given board
(define (manhattan-distance b)
  (local
    [ ...
      ; correct-pos: number --> number
      ; Purpose: To determine the correct position of the given tile
      (define (correct-pos n)
        (cond [(= n 0) (sub1 N)]
              [else (sub1 n)]))])
```

Finding a Solution



; manhattan-distance: board --> number

; Purpose: To compute the Manhattan distance of the given board

```
(define (manhattan-distance b)
```

```
  (local
```

```
    [...
```

```
    ...
```

```
    ; adder: number --> number
```

```
    ; Purpose: To add all the distances of each tile
```

```
    (define (adder pos)
```

```
      (cond [(= pos 0) 0]
```

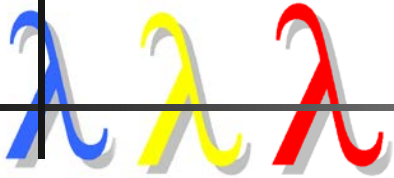
```
            [else (+ (distance (sub1 pos)
```

```
                      (correct-pos (list-ref b (sub1 pos))))
```

```
                      (adder (sub1 pos))))]))
```

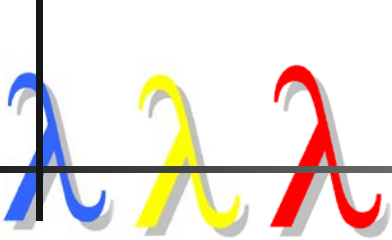
```
(adder N)))
```

Finding a Solution



- The Solver
 - given a board return a sequence (non-empty list of boards)
 - leads naturally to a depth-first search algorithm
 - If the given board is the winning board, then the solution is trivial
 - Otherwise, create sequence from the given board and the solution generated starting from the best child of the given board

Finding a Solution



```
; find-solution-dfs: board --> (listof boards)
```

```
; Purpose: To find a solution to the given board using DFS
```

```
(define (find-solution-dfs b)
```

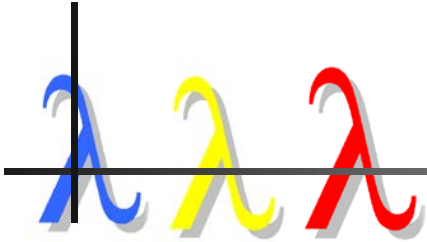
```
  (cond [(equal? b WIN) (list b)]
```

```
        [else
```

```
          (local [(define children (generate-children b))]
```

```
            (cons b (find-solution-dfs (best-child children))))]))
```

Finding a Solution



; generate-children: board --> non-empty-list-of-boards

; Purpose: To generate a list of the children of the given board

(define (generate-children b)

(local [(define blank-pos (get-blank-sq-num b))]

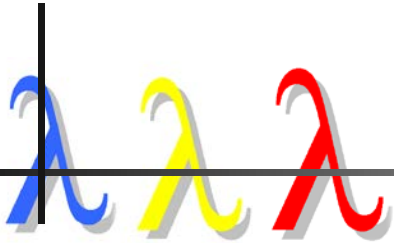
(map (lambda (p)

(swap-tiles b blank-pos p))

(blank-neighs blank-pos))))

Reinforces lessons on abstraction over lists

Finding a Solution



```
; best-child: non-empty-list-of-boards --> board
; Purpose: To find the board with the board with the smallest
;   Manhattan distance in the given non-empty list of boards
(define (best-child lob)
  (cond [(empty? (rest lob)) (car lob)]
        [else
         (local [(define best-of-rest (best-child (rest lob)))]
           (cond [(< (manhattan-distance (car lob))
                    (manhattan-distance best-of-rest))
                  (car lob)]
                 [else best-of-rest]))]))]
```

Example 1

Example 2 (little man works forever)



Finding a Solution



- What have we accomplished?
 - Reinforces lessons on structural recursion
 - Introduced students to generative recursion and DFS
 - Introduced students to heuristic-based programming
 - Got students interested in all of the above via a functional video game

There is more!

Finding a Solution

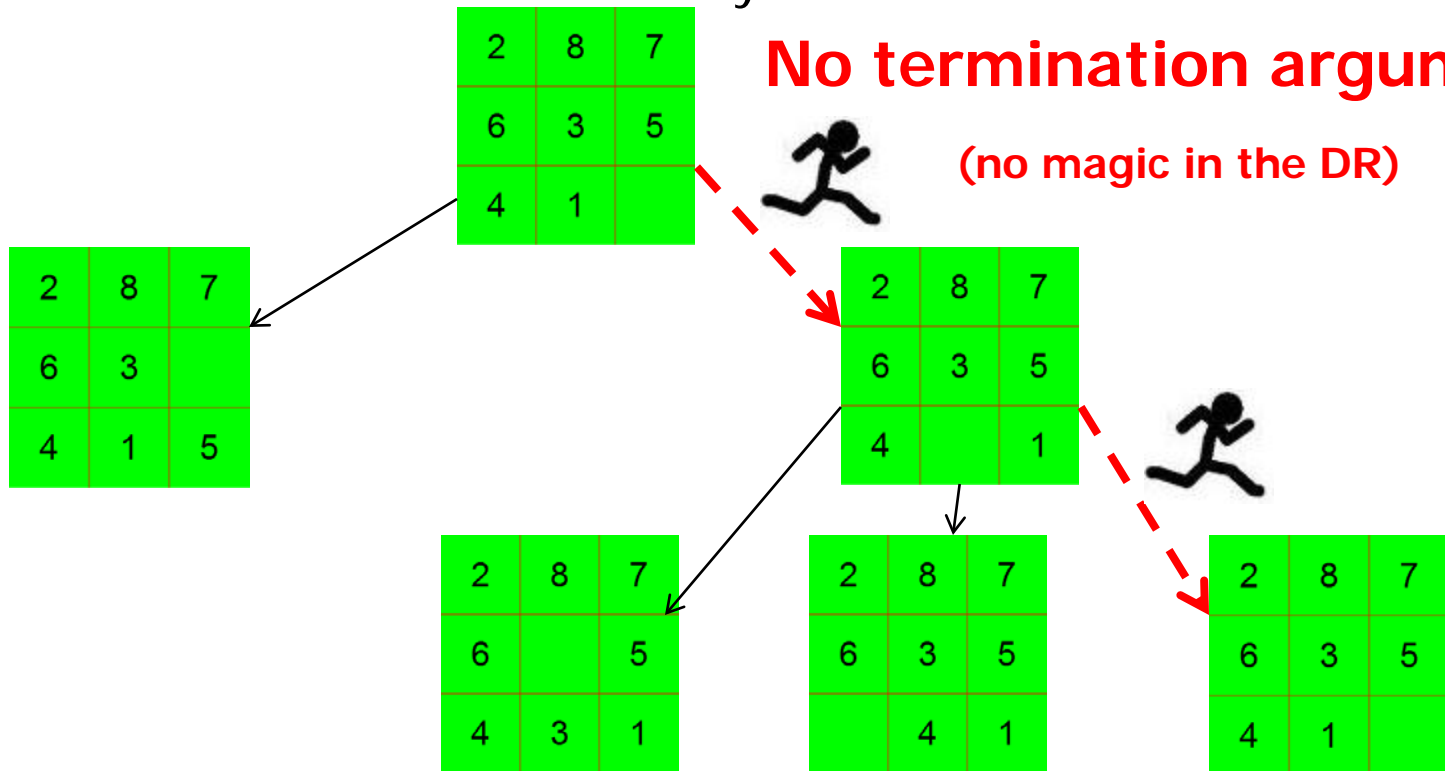


- Reinforced the value of testing and iterative refinement
 - Testing reveals that solution is not found for all legal boards!
 - CS1 students can understand why!

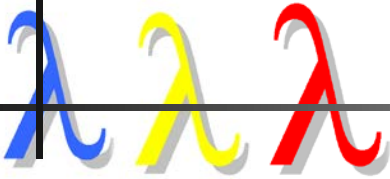
Finding a Solution



- Reinforced the value of testing and iterative refinement
 - Testing reveals that solution is not found for all legal boards!
 - CS1 students can understand why!

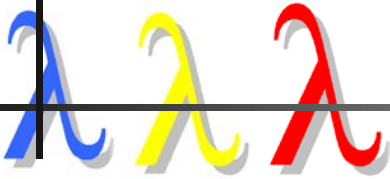


Refining the Solution



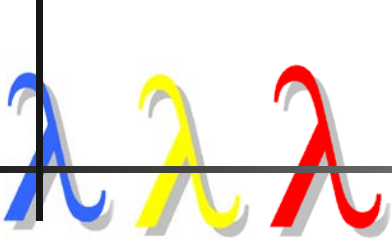
- Lead students to believe all sequences must be explored
 - Requires *remembering* all paths generated
 - Welcome to **accumulative recursion!**

Refining the Solution



- Lead students to believe all sequences must be explored
 - Requires *remembering* all paths generated
 - Welcome to **accumulative recursion!**
- Introduce students to BFS
 - Keep all sequences in order by length (introduce Qs????)
 - Build using work done for DFS solver
- The basic idea
 - if the first board in the first sequence is WIN, return the first sequence
 - Otherwise, generate new sequences using the successors of the first board in the first sequence

Refining the Solution



```
; find-solution-bfs: board → lseq
```

```
; Purpose: To find a solution to the given board
```

```
(define (find-solution-bfs b)
```

```
  (local
```

```
    [; search-paths: lseq → seq
```

```
    ; Purpose: To find a solution to b by searching all possible paths
```

```
    ; ACCUMULATOR INVARIANT:
```

```
    ; paths is a list of all seqs generated so far starting at b from
```

```
    ; from the shortest to the longest in reversed order
```

```
  (define (search-paths paths)
```

```
    (cond [(equal? (first (first paths)) WIN) (car paths)]
```

```
          [else
```

```
            (local [(define chldrn (generate-children (first (first paths))))]
```

```
                  (define new-paths (map (lambda (c) (cons c (first paths))) chldrn))]
```

```
                  (search-paths (append (rest paths) new-paths))))]))]
```

```
  (reverse (search-paths (list (list b)))))
```

EXAMPLE 1

EXAMPLE 2



Further Refining the Solution



- Nothing is worse than a slow video game!
- The problem
 - Exponential growth
 - after 10 moves the number of sequences being searched surpasses 2^{10}
 - after 20 moves it surpasses 2^{20}

Further Refining the Solution



- Is searching all possible sequences and searching all possible sequences at the same time necessary?
 - Most students can not answer and say yes

Further Refining the Solution



- Is searching all possible sequences and searching all possible sequences at the same time necessary?
 - Most students can not answer and say yes
- Two main ideas
 - not every sequence needs to be explored
 - visited successors can be ignored
 - explore the most promising sequence first

Further Refining the Solution



```
(define (find-solution-a-star b)
```

```
(local
```

```
  [(define (find-best-seq seqs)
```

```
    (cond [(empty? (rest seqs)) (first seqs)]
```

```
          [else
```

```
            (local [(define best-of-rest (find-best-seq (rest seqs)))]
```

```
              (cond [(< (manhattan-dist (first (first seqs)))
```

```
                      (manhattan-dist (first best-of-rest))]
```

```
                (first seqs)]
```

```
              [else best-of-rest])))])
```

Further Refining the Solution



```
(define (find-solution-a-star b)
```

```
(local
```

```
[...
```

```
(define (search-paths visited paths)
```

```
(local [(define bstseq (find-best-seq paths))]
```

```
(cond [(equal? (first best-path) WIN) bstseq]
```

```
      [else (local
```

```
                [(define children (filter (lambda(c)(not(member c visited)))
```

```
                (generate-children (first bstseq))))]
```

```
(define new-seqs (map (lambda (c) (cons c bstseq))
```

```
children))]
```

```
(search-paths
```

```
(cons (first bstseq) visited)
```

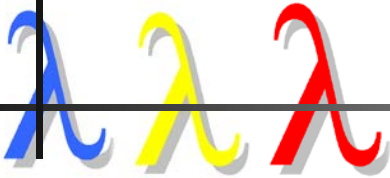
```
(append new-seqs (rem-path bstseq paths)))))))])))]
```

```
(reverse (search-paths '() (list (list b))))))
```

EXAMPLE



Related Work



- HtDP (the blue book)
 - Presents DRs for generative and accumulative recursion
 - Generative recursion
 - move away from structural recursion
 - recursion does not operate on part of the input
 - Accumulative recursion
 - Solves the problem of loss of knowledge

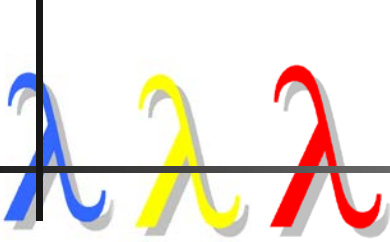
Related Work



- Soccer-Fun (Achten)
 - Play soccer
 - Used to teach FP to students sophomores
 - Not used in CS1, but used to motivate HS students

- Yampa (Courtney, Nilsson, & Peterson)
 - program reactive systems such as video games
 - Used to teach FP to students already exposed to programming (not recently)

Related Work



- N-Puzzle (Markov et al.)
 - used in AI & Machine Learning courses
 - used in Data Structures and Algorithms course

- Informed Heuristic Searching in CS1
 - Nijmegen: Exposure, but no implementation
 - Utrecht: CS majors in second year
 - Others: Intro to AI

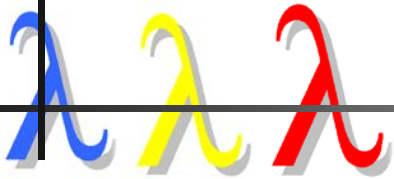
Wishlist



- Student languages & HtDP with built-in
 - Queues
 - Stacks

- HtDP introducing BNF grammars
 - Good for CS majors

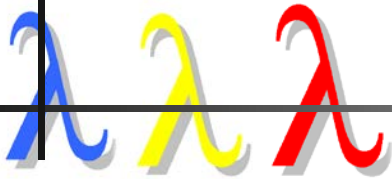
Conclusions



- In CS1

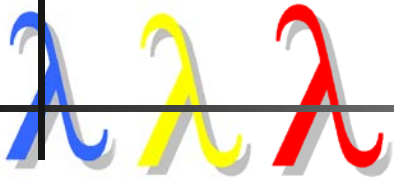
- make the transition from structural recursion to generative & accumulative recursion using a video game as motivation
- Make FP relevant early in CS development
- informed heuristic search strategies are for CS1 students
 - CS1 students can reason such an algorithm into existence
 - CS1 students can be enthusiastic about implementation

Conclusions

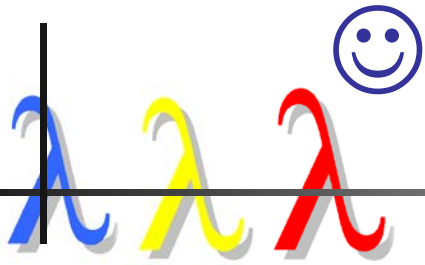


- In CS1
 - make the transition from structural recursion to generative & accumulative recursion using a video game as motivation
 - Make FP relevant early in CS development
 - informed heuristic search strategies are for CS1 students
 - CS1 students can reason such an algorithm into existence
 - CS1 students can be enthusiastic about implementation
- Future Work
 - Distributed Programming in CS1 using functional video games
 - State-based video games (???????)

FP in Education



- Workshop @ TFP 2012
 - St. Andrews University, Scotland (K. Hammond)
 - Workshop @ TFP 2013, BYU, Utah? (J. McCarthy)
- IFL 2011
 - University of Kansas (Andy Gill)



Any Questions?



1	3	7
4		6
2	5	8

HELP ME!

?

--->

been there,
done that!

1	2	3
4	5	6
7	8	

HELP ME!

Copies of two papers available @ front table