Programming with State

and

Genetic Algorithms
One-Slide Summary

- The substitution model for evaluating Scheme does not allow us to reason about mutation. In the environment model:
  - A name is a place for storing a value. define, mcons, cons and function application create places. set! changes the value in a place.
  - Places live in frames. An environment is a frame and a pointer to a parent frame. The global environment has no parent.
  - To evaluate a name, walk up the frames until you find a definition.
  - Functional and imperative procedures may have different asymptotic costs.
Outline

• Functional vs. Imperative
• Names and Places
• Environment Model practice
• Cost computation practice
• Undergrad research
Functional vs. Imperative

Functional Solution: A procedure that takes a procedure of one argument and a list, and returns a list of the results produced by applying the procedure to each element in the list.

```
(define (list-map proc lst)
  (if (null? lst) null
      (cons (proc (car lst))
            (list-map proc (cdr lst))))
)
Imperative Solution

A procedure that takes a procedure and list as arguments, and *replaces* each element in the list with the value of the procedure applied to that element.

```
(define (mlist-map! f lst)
  (if (null? lst) (void)
   (begin
    (set-mcar! lst (f (mcar lst)))
    (mlist-map! f (mcdr lst))))
```

```
(define (list-map proc lst)
  (if (null? lst) null
   (cons (proc (car lst))
        (list-map proc (cdr lst)))))
```
Programming with Mutation

Imperative

> (mlist-map! square (intsto 4))
> (define i4 (intsto 4))
> (mlist-map! square i4)
> i4
(1 4 9 16)

Functional

> (define i4 (intsto 4))
> (list-map square i4)
(1 4 9 16)
> i4
(1 2 3 4)
Names and Places

• A name is a **place** for storing a value.
• **define** creates a new place
• **cons** and **mcons** create two new places, the **car** and the **cdr**
• **(set! name expr)** changes the value in the place **name** to the value of **expr**
• **(set-mcar! pair expr)** changes the value in the **car** place of **pair** to the value of **expr**
New Application Rule:

1. **Construct a new environment**, whose parent is the environment to which the environment pointer of the applied procedure points.

2. **Create places** in that frame for each parameter containing the value of the corresponding operand expression.

3. **Evaluate the body in the new environment**. Result is the value of the application.
1. Construct a new environment, parent is procedure’s environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment

> (define x 3)
1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment

> (define x 3)
> (define (adder x)
  (lambda (y) (+ x y))))
1. Construct a new environment, parent is procedure’s environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment

> (define x 3)
> (define (adder x)
   (lambda (y) (+ x y)))
> (define add17 (adder 17))
1. Construct a new environment, parent is procedure’s environment pointer

2. Make places in that frame with the names of each parameter, and operand values

3. Evaluate the body in the new environment

> (define x 3)
> (define (adder x)
  (lambda (y) (+ x y))))
> (define add17 (adder 17))
1. Construct a new environment, parent is procedure’s environment pointer

2. Make places in that frame with the names of each parameter, and operand values

3. Evaluate the body in the new environment

> (define x 3)
> (define (adder x)
    (lambda (y) (+ x y)))
> (define add17 (adder 17))
1. Construct a new environment, parent is procedure’s environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment

> (define x 3)
> (define (adder x)
  (lambda (y) (+ x y)))
> (define add17 (adder 17))
> (add17 3)
1. Construct a new environment, parent is procedure’s environment pointer

2. Make places in that frame with the names of each parameter, and operand values

3. Evaluate the body in the new environment

> (define x 3)
> (define (adder x)
   (lambda (y) (+ x y)))
> (define add17 (adder 17))
> (add17 3)
1. Construct a new environment, parent is procedure’s environment pointer.

2. Make places in that frame with the names of each parameter, and operand values.

3. Evaluate the body in the new environment.

> (define x 3)
> (define (adder x)
  (lambda (y) (+ x y)))
> (define add17 (adder 17))
> (add17 3)

This answers the thought question on Slide #22 of Last Lecture!
Functional vs. Imperative Costs

(define (l-map f p)
  (if (null? p) null
      (cons (f (car p))
           (l-map f (cdr p)))))

• Running Time: O(N)
  - where N is (length p)
• Memory Use: O(N)
  - N new cons cells

(define (ml-map! f p)
  (if (null? p) (void)
      (begin
       (set-mcar! p (f mcar p))
       (ml-map! f (mcdr p)))))

• Running Time: O(N)
  - Also N recursive calls with O(1) work each
• Memory Use: O(1)
  - No new cons cells
Functional vs. Imperative Costs

(define (list-append p q)
  (if (null? p) q
      (cons (car p) (list-append (cdr p) q))))

- Running Time: $O(p)$; $p = \text{(length } p\text{)}$
- New Cons Cells: $O(p)$

(define (mlist-append! p q)
  (if (null? p) (error "append to empty list!")
      (if (null? (mcdr p)) (set-mcdr! p q)
          (mlist-append! (mcdr p) q))))

- Running Time: $O(p)$, New Cons Cells: Zero
mlist-append! in action

> (define (mlist-append! p q)
  (if (null? p) (error "append to empty list!")
   (if (null? (mcdr p)) (set-mcdr! p q)
     (mlist-append! (mcdr p) q))))
> (define animals (mcons "ant" (mcons "bat" null)))
> (define colors (mcons "red" (mcons "green" null)))
> (mlist-append! animals colors)
> animals

???

> colors

???
mlist-append! in action

> (define (mlist-append! p q)
  (if (null? p) (error "append to empty list!"))
  (if (null? (mcdr p)) (set-mcdr! p q)
    (mlist-append! (mcdr p) q))))

> (define animals (mcons "ant" (mcons "bat" null)))
> (define colors (mcons "red" (mcons "green" null)))
> (mlist-append! animals colors)

> animals
{"ant" "bat" "red" "green"}
> colors
{"red" "green"}
Open-Ended Question

• You're designing cars.
• Each of your designs has various properties:
  - High or low fuel efficiency, spacious or cramped interior, manual or automatic transmission, high or low purchase price, etc.
• You can also run a few focus groups or trial sales:
  - To see how many people will buy a given design.
• **How do you make the best-selling design?**
Genetic Algorithms

• Search Strategy based on biological evolution
  - Find X that maximizes P(X)
  - Find CarDesign maximizing Sales(CarDesign)

• Idea:
  - Represent car design as genome (string)
    • “LSMH” = “low fuel efficiency, spacious, manual, …”
  - Maintain a population of variant genomes
  - Apply a random mutation operator to them
  - The fittest individuals survive and mate
  - Process repeats with a new generation
Mutation and Crossover

• Mutation:
  - “LSMH” might become “LCMH”

• Crossover:
  “LSMH”
+ “LCAL”
= “LSAL” + “LCMH”

• Fitness:
  - Turn your genotype into a phenototype (i.e., build a model car from your design) and evaluate it (i.e., hire a focus group, test sales, etc.)
Genetic Algorithm Successes

- Creation of a **soccer-playing program** that ranked in the middle of the field of 34 human-written programs in the Robo Cup 1998 competition
- Synthesis of an **electronic thermometer**
- Automated **Re-Invention of Six Patented** Optical Lens Systems using Genetic Programming
- Towards Better than Human Capability in **Diagnosing Prostate Cancer** Using Infrared Spectroscopic Imaging
- Scaffolding for Interactively **Evolving Novel Drum Tracks** for Existing Songs
Unrelated Question

- So Genetic Algorithms are a heuristic approach for making small random changes to something until, by chance, you make some external judge happy.

- Totally unrelated question. How do your friends pass Automatic Adjudication programming assignments in this class?
Automated Program Repair

- On December 31st, 2008, MS Zune 30 music players died
- Microsoft had sold 1.2 million
- Problem: infinite loop in a recursive computation converting days to years
  - Bug shows up on last day of each leap year
- Microsoft's recommendation was to drain the Zune's battery and reset it
- Our genetic programming approach fixes the source code in 42 seconds.
The Bug

(define (days-to-years day year)
  (if (<= day 365)
      year
    (if (is-leap-year? year)
        (if (> day 366)
            (days-to-years (- day 366) (+ year 1))
            (days-to-years day year))
        (days-to-years (- day 365) (+ year 1))))

Try These Tests:
(days-to-years 300 2008)
(days-to-years 700 2008)
(days-to-years 366 2008)
The Fix

(define (days-to-years day year)
  (if (<= day 365)
      year
      (if (is-leap-year? year)
          (if (> day 366)
              (days-to-years (- day 366) (+ year 1))
              (days-to-years (- day 366) year))
          (days-to-years (- day 365) (+ year 1))))
Automated Program Repair

• On 105 bugs taken systematically from 5.1 millions of lines of code with 10,000 test cases …
  - Our genetic algorithm can repair 55 of them
  - For only $7.32 each!
  - Typical human cost: $25

• Why should you care?
  - Ethan Fast, now at Stanford Grad School
  - Briana Satchell, now at UMass Amherst Grad School
Homework

• PS 5!
  - Due Friday 21 October
  - It is longer than PS4.
  - If you wait, you will probably not have enough time.

• Read Course Book 9

• Plus anything else Dave Evans assigned …