How to Design *while* Loops

Marco T Morazán
Seton Hall University

The Students…

- Students struggle with while loops
  - maybe not toy programs
- Frustration
  - Inexplainable infinite loops
  - *My loops runs, but it's not giving me the right result*
    - The sequencing mutations problem

Are students incompetent?

- No!
- Textbooks
  - syntax, examples, and warnings
  - Lip service to correctness & termination arguments

Are students incompetent?

- Textbooks
  - Operational descriptions
    - test driver, if true execute body, if false exit loop
  - Handle operations that are inherently repetitive
    - Yeah, recursion too!

Are students incompetent?

- What else is wrong with textbooks?
  - No mention of state variables
  - No mention of accumulators
  - No mention of how to design *while* loops
    - Invariants don't just spring up out of thin air!
  - Mutation sequencing
    - What's that?

Are students incompetent?

- What else is wrong with textbooks?

Programming is a human activity!

Ignore teaching students to <u>communicate</u>
how a problem is solved

A Design-Based Approach

- HtDP
  - Generative recursion → Termination arguments
  - Accumulative recursion → Accumulator Invariants
  - State-Based computations → State-var Invariants
- Denotational Semantics
  - Hoare Logic

Student Background

- First two semesters HtDP-based
  - First semester
    - Structures, structural recursion, abstraction, distributed computing
  - Second semester
    - Generative recursion, accumulative recursion, vectors, **state-based computing**

Student Background

- Resources
  - 15 weeks of 2 75-minute lectures
  - 20-25 students per classroom
  - Office hours and email
  - 20-30 hours of tutoring available

Lessons from Accumulative Recursion

- Accumulators
  - Loss of knowledge
  - Eliminate delayed operations
  - Invariants

Lessons from Accumulative Recursion

```
fact: natnum → natnum
; Purpose: To compute n!
(define (fact n)
  (cond   [(= n 0) 1]
          [else (* n (fact (- n 1)))]))

(check-expect (fact 0) 1)
(check-expect (fact 3) 6)
```

```
; fact: natnum → natnum
; Purpose: To compute n!
(define (fact n)
(local [; fact-accum: natnum natnum → natnum
        ; Purpose: To compute n!
        ; Accum Inv: accum = Π^n_{i=k+1} i
        (define (fact-accum k accum)
          (cond [(= k 0) accum]
                [else (fact-accum (sub1 k)
                                  (* accum k))]))]
  (fact-accum n 1)))

(check-expect (fact 0) 1)
(check-expect (fact 3) 6)
```

Lessons from Accumulative Recursion

Correctness

$k=0 \rightarrow accum = \Pi^n_{i=1}\, i = n!$

Invariant holds

$k=n$ AND $accum=1$

$accum = \Pi^n_{i=k+1}\, i$

$1 = \Pi^n_{i=n+1}\, i$

$1 = 1$

$accum = \Pi^n_{i=k+1}\, i$

$\Pi^n_{i=k+1}\, i * k = \Pi^n_{i=(k-1)+1}\, i$

$\Pi^n_{i=k}\, i = \Pi^n_{i=k}\, i$

```
; fact: natnum → natnum
; Purpose: To compute n!
(define (fact n)
(local [; fact-accum: natnum natnum → natnum
         ; Purpose: To compute n!
         ; Accum Inv: accum = Πⁿ_{i=k+1} i
         (define (fact-accum k accum)
           (cond [(= k 0) accum]
                 [else (fact-accum (sub1 k)
                                   (* accum k))]))]
  (fact-accum n 1)))

(check-expect (fact 0) 1)
(check-expect (fact 3) 6)
```

Lessons from State-Based Design

		k	accum
(fact 4)	= (fact-accum	4	1)
	= (fact-accum	3	4)
	= (fact-accum	2	12)
	= (fact-accum	1	24)
	= (fact-accum	0	24)
	= 24		

k = ~~4 3 2 1~~ 0

accum = ~~1 4 12 24~~ 24

Lessons from State-Based Design

```
; fact: natnum → natnum
; Purpose: To compute n!
(define (fact n)
 (local [; natnum, Inv: k>=0
        (define k (void))
        ; natnum, accum = Π^n_{i=k+1} i
       (define accum (void))
  (define (fact-state)
   (cond   [(= k 0) accum]
           [else
            (begin
             (set! k (sub1 k))
             (set! accum (* k accum))
             (fact-state))]))]
  (begin
    (set! k n)
    (set! accum 1)
    (fact-state))))
```

```
; fact: natnum → natnum
; Purpose: To compute n!
(define (fact n)
 (local [; natnum, Inv: k>=0
        (define k (void))
        ; natnum, accum = Π^n_{i=k+1} i
       (define accum (void))
  (define (fact-state)
   (cond   [(= k 0) accum]
           [else
            (begin
              (set! accum (* k accum))
              (set! k (sub1 k))
             (fact-state))]))]
  (begin
    (set! k n)
    (set! accum 1)
    (fact-state))))
```

Which one is correct?
How do you know?

Lessons from State-Based Design

```
(define (fact-state)
  (cond [(= k 0) accum]
        [else
         (begin
           ; k>0 AND accum=Π^n_{i=k+1} i
           (set! k (sub1 k))
           ; k>=0 AND accum=Π^n_{i=k+2} i
           (set! accum (* k accum))
           ; k>=0 AND accum=k * Π^n_{i=k+2} i
           (fact-state))]))
(begin
    (set! k n)
    (set! accum 1)
     ; k>=0 AND accum=Π^n_{i=k+1} i
    (fact-state))))
```

```
(define (fact-state)
  (cond   [(= k 0) accum]
          [else
           (begin
             ; k>0 AND accum=Π^n_{i=k+1} i
               (set! accum (* k accum))
             ; k>0 AND accum=Π^n_{i=k} i
               (set! k (sub1 k))
              ; k>=0 AND accum=Π^n_{i=k+1} i
               (fact-state))]))
  (begin
     (set! k n)
     (set! accum 1)
      ; k>=0 AND accum=Π^n_{i=k+1} i
     (fact-state))))
```

New Syntax

- Common to package repeated mutations with no explicit recursive call
- Our focus in on *while* loops
- Transformation of state-based accumulative recursive function
  - Initialize state vars to achieve the invariant = code before 1st call to acc rec funct
  - Negation of conjunction of non-recursive conditions is *the driver*
  - Loop body = recursive cases code
  - After loop code = non-recursive cases code

New Syntax

```
; fact: natnum → natnum
; Purpose: To compute n!
(define (fact n)
 (local [; natnum, Inv: k>=0
         (define k (void))
         ; natnum, accum = Πⁿ_{i=k+1} i
        (define accum (void))
        (define (fact-state)
        (cond [(= k 0) accum]
                [else
                 (begin
                   (set! accum (* k accum))
                   (set! k (sub1 k))
                   (fact-state))]))]
  (begin
    (set! k n)
    (set! accum 1)
    (fact-state))))
```

```
(define (fact n)
  (local
    [(define k (void))
     (define accum (void))
     (define (fact-while)
       (begin
         (set! k n)   (set! accum 1)
         ;; Invariant: k >= 0 AND accum = Πⁿ_{i=k+1} i
         (while (not (= k 0))
           ;; k>0 AND accum = Πⁿ_{i=k+1} i
           (set! accum (* k accum))
           ;; k>0 AND accum = Πⁿ_{i=k} i
           (set! k (sub1 k)
           ;; k>=0 AND accum = Πⁿ_{i=k+1} i     )
         ;; k>=0 AND accum = Πⁿ_{i=k+1} i AND k = 0
         ;; → accum = n!
         accum))]
   (fact-while)))
```

New Design Recipe

1. Problem Analysis
    - (a) Outline how the problem is solved
    - (b) Pick a mutable data representation
2. Write signature, purpose and effect statements, and function header
3. Write Tests
4. Develop the Loop Invariant
5. Define a function with a local expression as its body
    - (a) Locally declare the state variables as (void)
    - (b) Define the type and purpose for each state variable
    - (c) Define headers for helper functions
6. Write the body of the local using a begin expression
    - (a) Initialize the state variables to achieve the invariant
    - (b) Define the while loop
        - i. Define the driver and write the loop header
        - ii. Use the invariant to correctly sequence mutations
        - iii. Make progress towards termination
    - (c) Use the negation of the driver and the invariant to determine the value to return
7. Develop a Termination Argument
8. Run Tests

New Design Recipe

```
; signature:         Purpose: Effect:
(define (f-while ...)
(local [   ; <type>                          ; <type>
           ; Purpose:                        ; Purpose:
           (define state-var1 (void)) … (define state-varN (void))
           <helper functions>]
  (begin
    (set! state-var1 ...) … (set! state-varN ...)
    ; <Invariant>
    (while <driver>
       <while-body>)
    ; <Invariant> and (not <driver>)
    <return value code>))
    ; <Termination argument>  )
(check-expect (f-while …) …) … (check-expect (f-while : : :) : : :)
```

Insertion Sorting in Place

- Problem Analysis
  - Sort a vector, V, by mutating it
  - Sort entire vector → sort vector interval [0..(sub1 (vector-length V))]
  - Halt when vector interval is empty
  - Process VI from high to low
  - Vector is split in two: sorted and unsorted portions
  - Insert high element, h, of unsorted portion into sorted portion
    - h is a state variable

Insertion Sorting in Place

- Write signature, purpose and effect statements, and function header

```
; (vectorof number) → (void)
; Purpose: To sort the given vector in non-decreasing order
; Effect: The given vector elements are rearranged in-place.
(define (ins-vector! V)
          (local
            [ … ]
  (sort! 0 (sub1 (vector-length V)))))
```

Insertion Sorting in Place

- Write Tests

```
(check-expect (begin
                (ins-vector! (vector))
                V) ← empty vector
              (vector))
(check-expect (begin
                (ins-vector! (vector 20 76 3 44))
                V) ← non-empty random
              (vector 3 44 20 76))
(check-expect (begin
                (ins-vector! (vector 1 2 3))
                V) ← non-empty sorted
              (vector 1 2 3))
(check-expect (begin
                (ins-vector! (vector 101 87 8))
                V) ← non-empty reversed
              (vector 8 87 101))
```

Insertion Sorting in Place

- Develop the Loop Invariant ← Hardest Step!
  - Show that vector is divided into two portions: sorted and unsorted
  - Show that V is sorted at the end
  - INV & (not driver) ➔ post condition
- Does this work?
  - V[low..h] is unsorted & V[h+1..high] in non-decreasing order

    INV & [low..h] is empty ➔? V[low..high] in non-decreasing order

    No, can't determine h.

    Observe: V[low..h] is unsorted is not useful
  - V[h+1..high] is sorted in non-decreasing order & h >= low-1

    INV & [low..h] is empty ➔? V[low..high] in non-decreasing order

    → h = low-1

    → V[low..high] in non-decreasing order

Insertion Sorting in Place

- Define a function with a local expression as its body
  - (a) Locally declare the state variables as (void)
  - (b) Define the type and purpose for each state variable
  - (c) Define headers for helper functions

; sort!: $VINTV_V$ [low..high] → (void)
; Purpose: Sort given vector interval in non-decreasing order
; Effect: Given interval elements are rearranged in-place

```
(define (sort! low high)
(local
   [; int
     ; Purpose: Next element index to move to sorted part of V
     (define h (void))]
  …)
(define (insert! lo hi) …)
```

← Make local?

Insertion Sorting in Place

- Write the body of the local using a begin expression
  - (a) Initialize the state variables to achieve the invariant
  - (b) Define the while loop
    - i. Define the driver and write the loop header
    - ii. Use the invariant to correctly sequence mutations
    - iii. Make progress towards termination
  - (c) Use the negation of the driver & invariant to determine return value

```
(begin (set! h high)
       ; INV: V[h+1..high] in non-decreasing order & h >= low-1
       (while (not (empty-VINTV? low h))
       ; h >= low & V[h+1..high] in non-decreasing order
       (insert! h  (sub1 high))
       ; h >= low & V[h..high] in non-decreasing order
       (set! h (sub1 h))
       ; h >= low-1 & V[h+1..high] in non-decreasing order  ) ; closes while
```

Insertion Sorting in Place

- Write the body of the local using a begin expression
  - (a) Initialize the state variables to achieve the invariant
  - (b) Define the while loop
    - i. Define the driver and write the loop header
    - ii. Use the invariant to correctly sequence mutations
    - iii. Make progress towards termination
  - (c) Use the negation of the driver & invariant to determine return value

```
(begin …) ; closes while
; h >= low-1 & V[h+1..high] in non-decreasing order & [low..h] is empty
; ==> h < low
; ==> h = low-1
; ==> V[low..high] in non-decreasing order
(void)))) ; closes sort!
```

Insertion Sorting in Place

- Develop a Termination Argument

```
(begin    (set! h high)
           ; INV: V[h+1..high] in non-decreasing order & h >= low-1
          (while (not (empty-VINTV? low h))
                    ; h >= low & V[h+1..high] in non-decreasing order
                    (insert! h  (sub1 high))
                    ; h >= low & V[h..high] in non-decreasing order
                    (set! h (sub1 h))
                    ; h >= low-1 & V[h+1..high] in non-decreasing order  )
```

h starts at high making [low..h] a valid vector interval. Each loop iteration decreases h by 1. Eventually, h becomes < low. This makes [low..h] empty and the loop terminates.

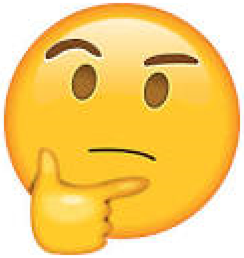
Insertion Sorting in Place

- Similar development for insert!
- Run tests

Concluding Remarks

- Beginning students can **design** while loops
  - Designing generative recursive, accumulative recursive, and state-based functions prepares them well
  - A modicum of Hoare Logic goes a long way!
  - Less frustration
    - sequencing mutations
    - infinite loops
- Prepares students for program verification
- Future work
  - Making *while* loops iterative
  - Measuring student reaction and retention
  - Vertical integration into the curriculum

Thank you!



Any questions?