When do I need an invariant?

CS 5010 Program Design Paradigms

Lesson 7.3

Learning Objectives

- At the end of this lesson, the student should be able to
  - decide whether a purpose statement needs an invariant or not.

When do I need an invariant?

- It all depends on the purpose statement.
- If your code fulfills the purpose statement for any arguments of the types listed in the contract, you don't need an invariant.
- If the function fulfills its purpose statement only for certain values or combinations of values of the arguments, then you must document that restriction with a WHERE-clause.

What kind of things belong in an invariant?

- If the function needs additional information that is not in the arguments, then you need an invariant to document the needed information
- What kind of information might you want?
  - context information (e.g. we are position **n** in the list)
  - other knowledge that isn't expressed in the contract (e.g. we've figured out the ball isn't going to bounce).

Whose responsibility is it?

- The invariant, along with the contract, sets down the assumptions that each function makes about the arguments that it processes
- It is up to each caller of the function to make sure that the invariant is true at every call.
- The function gets to assume that the invariant is true.

Example:

```
;; ball-normal-motion : Ball -> Ball
;; GIVEN: a Ball


;; RETURNS: the state of the ball after a
;; tick.
(define (ball-normal-motion b)
  (make-ball
    (+ (ball-x-pos b) BALLSPEED)))
```

Doesn't work for every Ball!..
Needs more information

Invariant provides the necessary information

Example

```
;; number-list-from : ListOfX Number -> NumberedListOfX
;; RETURNS: a list with same elements as lst, but numbered
;;  starting at n.
;; EXAMPLE: (number-list-from (list 88 77) 2)
;;          = (list (list 2 88) (list 3 77))
;; STRATEGY: Use template for ListOfX on lst
(define (number-list-from lst n)
  (cond
    [(empty? lst) empty]
    [else
      (cons
        (list n (first lst))
        (number-list-from (rest lst) (+ n 1)))]))
```

Satisfies its purpose statement for any lst and n, so no invariant necessary.

Function can't fulfill its purpose unless it knows where slst is in lst0

ame Code, different ose statement

```
;; number-sublist :
;;     ListOfX Number -> NumberedListOfX
;; GIVEN: a sublist slst of some list lst0

;; RETURNS: a copy of slst numbered according to its
;;  position in lst0.
;; STRATEGY: Use template for ListOfX on slst
(define (number-sublist slst n)
  (cond
    [(empty? slst) empty]
    [else
      (cons
        (list n (first slst))
        (number-sublist (rest slst) (+ n 1)))]))
```

Invariant supplies the extra information

Wait, weren't those functions very similar?

- Yes. In fact they were identical (except for their names).
- The moral of the story is that it is the purpose statement that determines whether you need an invariant.

Once more: When do I need an invariant?

- If your code fulfills the purpose statement for any arguments of the types listed in the contract, you don't need an invariant.
- If the function only works for certain values or combinations of values of the arguments, then you must document the assumptions that it needs with a WHERE-clause (i.e. an invariant).

What needs to be in my purpose statement?

- The purpose statement must account for all the parameters.
  - if it doesn't then either you are passing more parameters than you need, or there's something going on that you haven't described.
- The RETURNS clause must describe the value returned by the function for all possible values of the parameters.
- If the RETURNS clause describes the value returned by the function only for some values of the arguments or some combination of arguments, then that restriction must be stated in a WHERE clause.
- It becomes the responsibility of the caller to guarantee that the restriction is satisfied.

Another example

```
;; add-remaining-length : LoN -> LoN
;; RETURNS: a list like the original, but with each
;; element increased by the length of the sublist
;; starting at that element.
;; (100 300 500) => (103 302 501)
;; Strategy: Use template for LoN on lst
(define (add-remaining-length lst)
  (cond
    [(empty? lst) empty]
    [else (cons
            (+ (first lst) (length lst))
            (add-remaining-length
              (rest lst)))]))
```

Yuck! You have to recalculate the length of list every time through (repeated computation might be slow!)

Let's help the function along by giving it the length of the list as an argument

```
;; add-remaining-length-1 : LoN Number-> LoN
;; GIVEN: a Lon lst and a number n
;; WHERE: n = (length lst)
;; RETURNS: a list like the original, but with each
;; element increased by the length of the sublist
;; starting at that element.
;; (100 300 500) 3 => (103 302 501)
;; Strategy: Use template for LoN on lst
(define (add-remaining-length-1 lst n)
  (cond [(empty? lst) empty]
        [else (cons
               (+ (first lst) n)
               (add-remaining-length-1 (rest lst)
                                       (- n 1)))]))
```

Doesn't give the right answer unless invariant is satisfied

Recapture the original function by initializing the invariant

```
;; add-remaining-length-version-2 : LoN -> LoN
;; GIVEN: a Lon lst
;; RETURNS: a list like the original,
;;  but with each element increased by the
;;  length of the sublist starting at that
;;  element.
;; (100 300 500) => (103 302 501)
;; STRATEGY: Initialize the invariant
;;           of add-remaining-length-1
(define (add-remaining-length-version-2 lst)
  (add-remaining-length-1 lst (length lst)))
```

Summary: When do I need an invariant?

- It all depends on your purpose statement!
- If the function needs additional information to fulfill its stated purpose, and that information is not in the arguments, then you need an invariant to document the needed information.
- It is up to each caller of the function to make sure that the invariant is true at every call.

Summary

- The student should now be able to
  - decide whether a purpose statement needs an invariant or not.

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board
- Look at 07-3-invariants.rkt in the Exam
- Go on to the next lesson