CS61A Week 4 | The Invariable March Towards Midterm One (v1.1)

MoonLanding

Your spaceship has just crash-landed on the moon. You were scheduled to rendezvous with a mother ship 200 miles away on the lighted surface of the moon, but the rough landing has ruined your ship and destroyed all of the equipment on board, except for the 15 items listed below.

Your crew's survival depends on reaching the mother ship, so you must choose the most critical items available for the 200-mile trip. Your task is to rank the 15 items in terms of their importance for survival. Place the number 1 by the most important item, number 2 by the second most important, and so on through number 15, the least important. Your ranking will be compared with that of the experts at NASA to determine the winner.

INSTRUCTIONS: This is an exercise in group decision making. Your group is to employ the method of group consensus in reaching its decision. This means that the prediction for each of the 15 survival items must be agreed upon by each group member before it becomes a part of the group decision. Consensus is difficult to reach. Therefore, not every ranking will meet with everyone's complete approval. Try, as a group, to make each ranking one with which all group members can at least partially agree.

Here are some guides to use in reaching consensus:

1. Avoid arguing for your own individual judgments.Approach the task on the basis of logic.
2. Avoid changing your mind only in order to reach agreement and avoid conflict. Support only solutions with which you are able to agree somewhat, at least.
3. Avoid "conflict-reducing" techniques such as majority vote, averaging, or trading in reaching your decisions.
4. View differences of opinion as helpful rather than as a hindrance to decision making.

Iterm	Your Rank	Difference	Group Rank	Difference
Box of matches				
Food concentrate				
Fifty feet of nylon rope				
Parachute silk				
Solar-powered portable heating unit				
Two 45-Caliber pistols				
One case of dehydrated milk				
Two 100 pound tanks of oxygen				
Stellar map (of moon's constellations)				
Self-inflating life raft				
Magnetic compass				
Five gallons of water				
Signal flares				
First-aid kit containing injection needles				
Solar-powered FM receiver-transmitter				
Total	-		-	

Variations on Invariants

Invariants confused you in lecture and confounded you on homework. Now it's really time to show them who's boss.

Invariants typically are used to prove the correctness of a program, and only apply to iterative processes. Very simply, an **invariant** *is a statement that holds true through all recursive calls of an iterative procedure*. This statement typically refers to a relationship between the various arguments in each recursive call.

It's a little hard to grasp because, through each recursive call, the value of each argument changes. The important thing to note here is that while the value of an argument may change, its relationship with the other arguments may stay constant! To make this concrete, let's look at an easy example, an iterative version of `count`:

```
;; returns the number of elements in sent
(define (count sent)
        (define (count-helper s count-so-far)
                (if (empty? s)
                    count-so-far
                    (count-helper (bf s) (+ 1 count-so-far))))
        (count-helper sent 0))
```

Simple procedures like the one above should no longer be a challenge to understand. The iterative procedure we're interested in is `count-helper`; `count` is merely a wrapper around `count-helper` that makes using `count-helper` easier. So what invariants does `count-helper` have?

Well, here's an easy one. *Through all recursive calls of* `count-helper`*,* `count-so-far` *will always be non-negative.* This is pretty obviously true; `count-so-far` always starts out as 0, and we only add to `count-so-far`, so of course it will be either 0 or positive. This guarantees that `count-helper` will always return a non-negative number.

Well okay, so it's a little comforting to know that I won't ever have negative number of elements in my sentences. But how do I know that `count-helper` is correct? The above invariant, while true, is not terribly useful. And this is true in general – often, there are many invariants that are true for a procedure, but most of them will not be very interesting.

The most interesting ones are ones that prove correctness. And here's one for `count-helper`: *the sum of* `count-so-far` *and the number of elements in* `s` *will always be the same.* That is, no matter which recursive call you look at, `count-so-far` plus number of elements in `s` will always be the same – in fact, this sum will be the number of elements in the original `sent`. This is also pretty obvious for this simple procedure. It starts out with `count-so-far` being 0 and `s` being the original sentence, and whenever we add one to `count-so-far`, we take one element away from `s`. Because of this invariant, in the base case, when `s` is empty (and thus has no elements), `count-so-far` must be the same number of elements in the original `sent`. And `count-so-far` is exactly the number that we return, so `count` must be correct!

Let's try a few slightly trickier ones.

QUESTIONS: What are "interesting" invariants of the following procedures (or their helpers)?

1.
```
;; returns sum of elements in sent
(define (sum-of-sent sent)
        (define (helper sum s)
                (if (empty? s)
                    sum
                    (helper (+ (first s) sum) (bf s))))
        (helper 0 sent))
```

2.
```
;; returns a*b+c
(define (ab+c a b c)
        (define (helper aa ans)
                (if (= aa 0)
                    ans
                    (helper (- aa 1) (+ ans b))))
        (helper a c))
```

3.
```
;; reverses sent
(define (reverse sent)
        (define (helper old new)
                (if (empty? old)
                    new
                    (helper (bf old) (se (first old) new))))
        (helper sent '()))
```

4.
```
(define (mystery a b)
     (if (> a b)
         a
         (mystery (+ a 1) (- b 1))))
```

Recursions

QUESTIONS:

1. I have a bag of n oranges and m apples. If I eat them one at a time, how many ways can I eat all oranges and apples?

2. Define a procedure `subsent` that takes in a sentence and an integer i. It returns a sentence with ith item in the original sentence to the last item (The first element being the 0th item). For example, `(subsent '(6 4 2 7 5 8) 3)` returns `(7 5 8)`.

3. Define a procedure `sum-of-sents` that takes in two sentences of numbers and returns a sentence with the sum of respective elements from both sentences. Note that the sentences *do not* have to be the same size! For example, `(sum-of-sents '(1 2 3) '(4 5 6))` will return `(5 7 9)`, whereas `(sum-of-sents '(1 2 3 4 5) '(6 7))` will return `(7 9 3 4 5)`.

4. Consider the `subset-sum` problem: you are given a sentence of integers and a number `k`. Is there a subset of the sentence that add up to `k`? For example,
   `(subset-sum '(2 4 7 3) 5) ==> #t`, since $2 + 3 = 5$
   `(subset-sum '(1 9 5 7 3) 2) ==> #f`