LEC 01/04

Structural recursion

- The structure of the program matches the structure of the data.

```
(define (fact n)
    (if (= n 0) 1
        (* n (fact (- n 1)))))
```

- The cases in the function match the caess in the data definition.
- The recursive call uses argumentsthat either stay the same or get one step closer to the base of the datatype.

Ex.

```
(define (length L)
    (cond
        [(empty? L) 0]
        [else (add1 (length rest L))]])
```

- A `(listof X)` is either `empty` or `(cons x y)` where `x` is an `x` and `y` is a `(listof x)`.
- If the recursive is structural, the structure of the program matches the structure of its correctness proof by induction.

Proof

- **Claim** `(length L)` produces the length of the list `L`.
- **Proof** structural induction on `L`
  - Case 1: `L` is `empty`
    - Then `(length L)` produces 0, which is the length of an empty list.
  - Case 2: `L` is `cons x L'`
    - Assume that (length L') produces length n, then (length L) produces `(add1 n)`, which is the length of `(cons x L')`.
- **Correctness proof is just a restatement of the program itself.**

Accumulative Recursion

- One or more parameters “grow” while the other parameter “shrink”.

Ex.

```
(define (sum-list L)
    (define (sum-list-help L acc)
        (cond
            [(empty? L) acc]
            [else (sum-list-help (rest L) (+ (first L) acc))]))
    (sum-list-help L 0))
```

Proof. ***Induction on an invariant.***

- To prove that `(sum-list L)` sums `L`, it suffices to prove `(sum-list-help L 0)` sums `L`.
- Attepmt to prove by structural induction on `L` (Gonna fail **XD**)
    - Case 1. `L` is `empty`.
        - Then `(sum-list-help L 0)` = `(sum-list-help empty 0)` = `0` (True).
    - Case 2. `L' = '(cons x L')`
        - Then `(sum-list-help L 0)` = `(sum-list-help (cons x L') 0)` = `(sum-list-help L' (+ x 0))` = `(sum-list-help L' x)`
        - Since our inductive hypothesis only deals with `x` = `0`, our proof fails.
- We need a stronger statement relate the relationship between `L` and `acc` that holds throughout the recursion —— the ***invarient***.
- Attempt to prove using the invariant.
    - $\forall L^+ \forall acc$, `(sum-list L' acc)` produces `acc + <the sum of L>` by structural induction.
    - Case 1: `L` is `empty`
    - Case 2: `L` = `(cons x L')`
        - Assume `(sum-list-help L' acc)` produces $\sum L' + acc$.
        - `(sum-list-help (cons x L') acc)` = `(sum-list-help L' (+ x acc))` $= \sum L' + (x + acc) = (x + \sum L') + acc = \sum L + acc$.

Generative recursion

- Does not follow the structure of the data.
- Proofs rerquire more creativity.

How do we reason about imperative programs?

Recall: impure Racket

```
(begin expr1 ... expr_n)
```

- Evaluates all of `expr1 ... expr_n` in left-to-right order
- Produces the value of the `expr_n`
- Only the last statement affect the outcome
- Useless in a pure functional setting
- But useful if all the expressions are evaluated for their side-effects.
- Implicit `begin` in the bodies of functions, `lambda` , `local` , answer of `cond/match` .