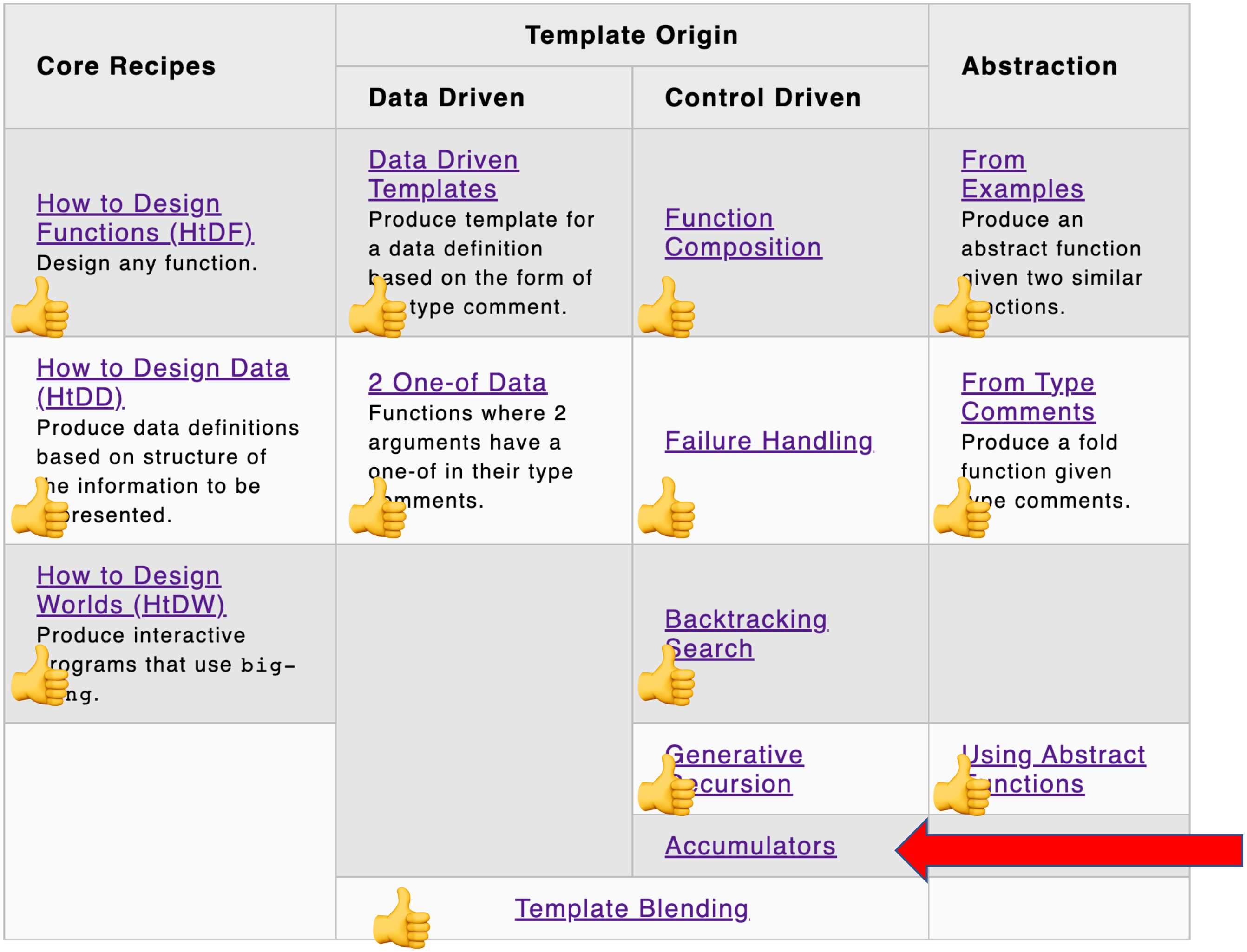
👍 atomic
👍 compound
👍 lists
👍 trees
graphs

search
👍 trees
graphs



Core Recipes	Template Origin		Abstraction
	Data Driven	Control Driven	
How to Design Functions (HtDF) Design any function. 👍	Data Driven Templates Produce template for a data definition based on the form of the type comment. 👍	Function Composition 👍	From Examples Produce an abstract function given two similar functions. 👍
How to Design Data (HtDD) Produce data definitions based on structure of the information to be represented. 👍	2 One-of Data Functions where 2 arguments have a one-of in their type comments. 👍	Failure Handling 👍	From Type Comments Produce a fold function given type comments. 👍
How to Design Worlds (HtDW) Produce interactive programs that use big-bang. 👍		Backtracking Search 👍	
		Generative Recursion 👍	Using Abstract Functions 👍
		Accumulators	
	👍 Template Blending		

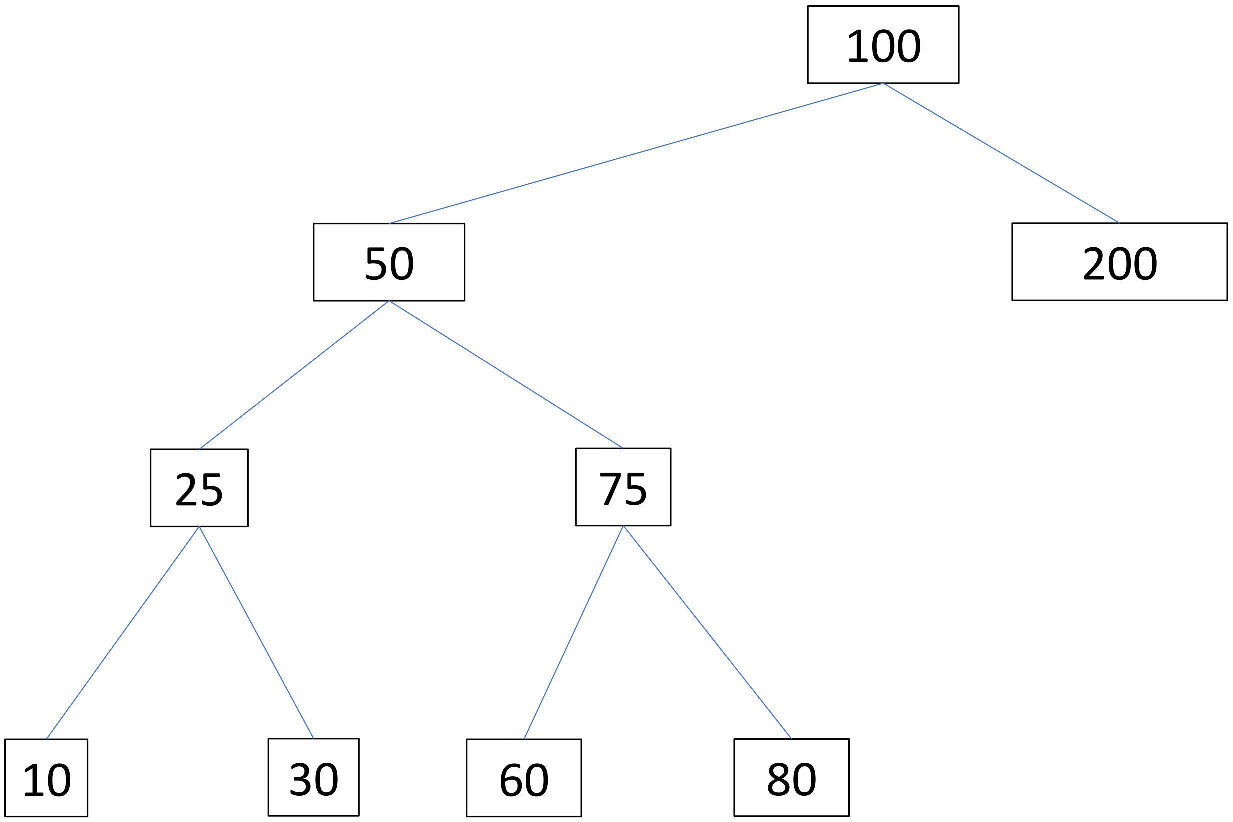
Accumulators – 2 goals, 3 kinds

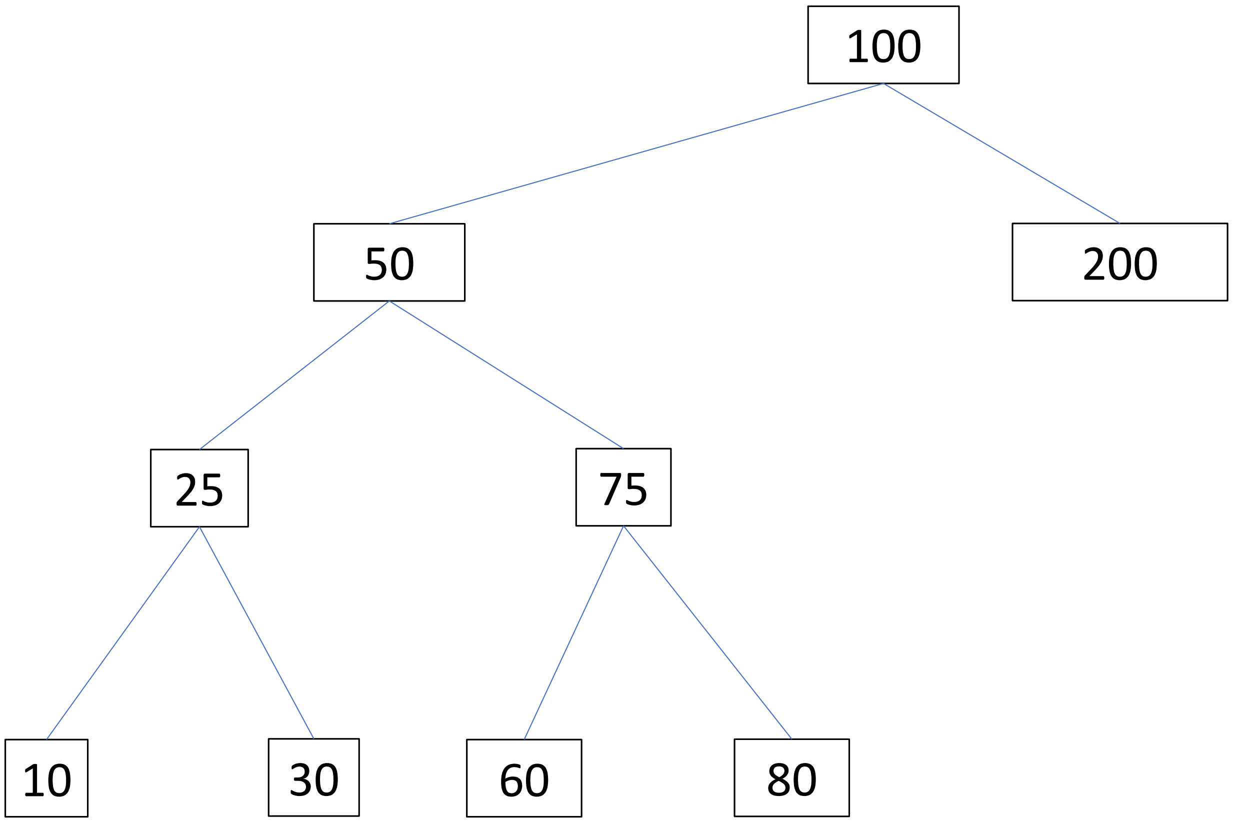
Goal	Kind of invariant
Preserve context from prior recursive calls	Context preserving parent house in same house… Result so far rsf in sum, product Work list right fringe of tree
Achieve tail recursion	

accumulator design recipe (htdf + this)

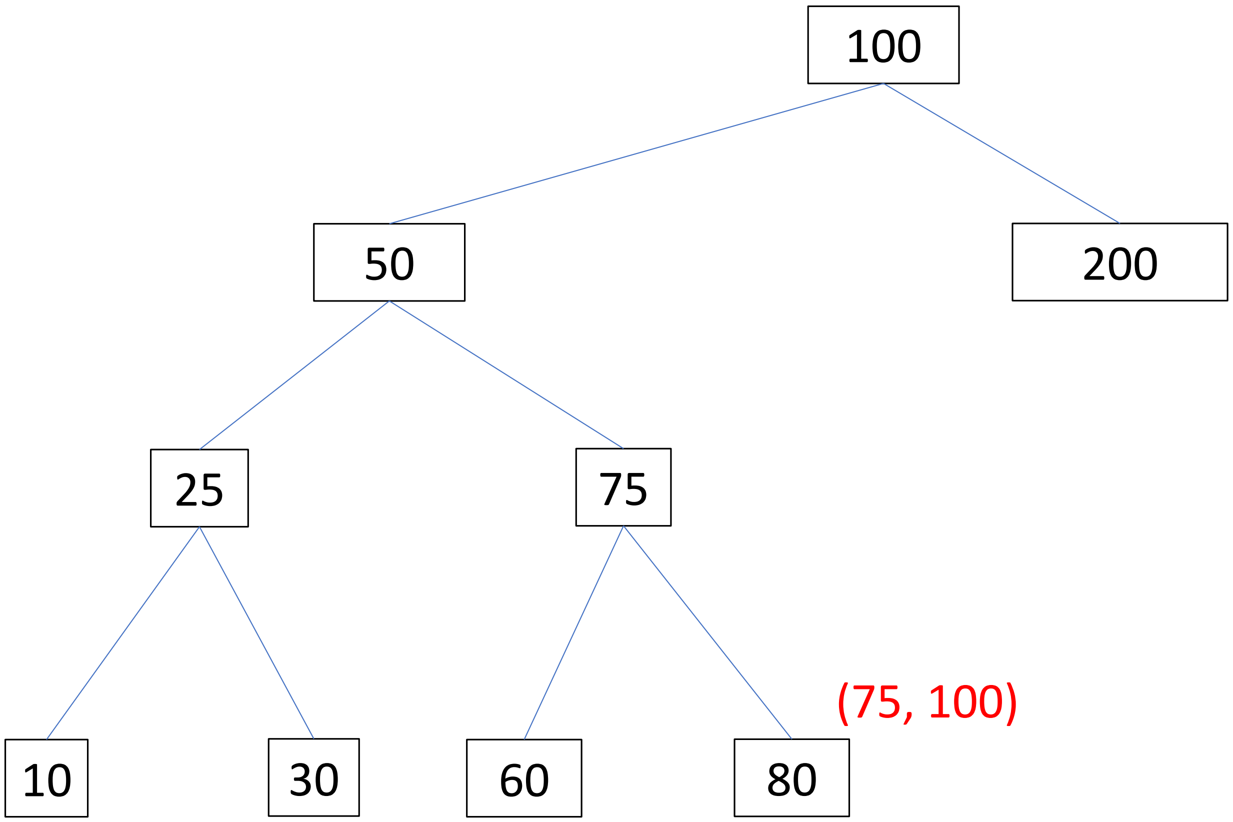
- templating:
  - recursive template wrapped in local and top-level function
  - add acc(umulator) parameter to inner function
    - add acc after all …
    - add (… acc) in recursive call
    - add … in trampoline
- work out example progression of recursive calls
- wish for what the accumulator should be at the end
- work backward through progression to get accumulator at each step
- design type and invariant (may need a new data definition)
- initialize invariant, preserve invariant, exploit invariant
- test and debug

```
(@template-origin (listof Natural) accumulator)

(define (sequence? lon0)
  ;; acc is  Natural
  ;; invariant: the element of lon0 immediately before (first lon)
  ;; (sequence? (list 2 3 4 5 6))

  ;; (sequence? (list   3 4 5 6) 2)
  ;; (sequence? (list     4 5 6) 3)
  ;; (sequence? (list       5 6) 4)
  (local [(define (sequence? lon acc)
            (cond [(empty? lon) true]
                  [else
                   (if (= (first lon) (+ 1 acc)) ;exploit
                       (sequence? (rest lon)
                                  (first lon))   ;preserve
                       false)]))]

    (sequence? (rest lon0)
               (first lon0))))                   ;initialize
```


100
50
200
25
75
10
30
60
80



Why is 80 OK here?
What range of numbers is OK here?



(75, 100)

Why is 80 OK here?
What range of numbers is OK here?

```
(@template-origin BinaryTree accumulator)

(define (bst? bt0)
  ;; lower is Integer, lower bound of key at this node (based on parents)
  ;; upper is Integer, upper bound of key at this node (based on parents)
  (local [(define (bst? bt lower upper)
            (cond [(false? bt) true]
                  [else
                   (and (< lower (node-k bt) upper)                  ;exploit
                        (bst? (node-l bt)  lower        (node-k bt)) ;preserve
                        (bst? (node-r bt)  (node-k bt)  upper))]))]  ;preserve

    (bst? bt0 -inf.0 +inf.0)));                                      ;initialize
;                             ;NOTE that we would never expect you to have
;                             ;     already known about these two constants!
```