

From Linear to Incremental

Christophe Grand, Clojure Conj 2011
@cgrand

**In the beginning was a
frustration**

Why do I need...

- ... to reparse a whole file when I make a minor edit in it.
- Seriously, editing one symbol doesn't change the overall structure!

Consequences

- Text editors
 - Can't rely on the parse-tree being always available
 - Ad hoc code (eg regexes) for:
 - structure highlighting
 - parens matching
 - (semi) structural editing

Incremental parsing

- Simple edits should be incremental!
- $O(\log n)$: rebuilding the parent nodes

Incremental parsing

- My definition:
 - Recompute the whole parse-tree at a fraction of the cost after an edit.
- Overloaded term:
 - Restartable
 - Lazy restartable (Yi, haskell editor)

What's a parser?

- Consumes a string, returns a tree
- A reduction
(get-tree (reduce parse-step init input))
- Many programs can be coerced in such a form

It can't be that hard

- Just another reduce
 - The problem isn't specific to parsing
- Let's solve the generic case!
 - Famous last words
 - I've never said *which* Christmas!

What I'm interested into

(get-tree (reduce parse-step init input'))

knowing

(get-tree (reduce parse-step init input))

and the edit between input and input'

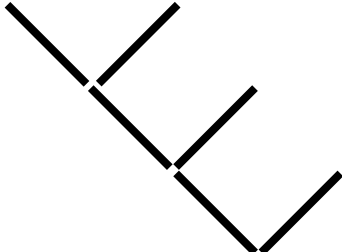
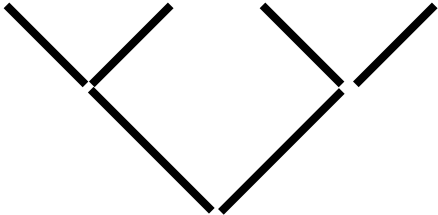
“reduce considered slightly harmful”

- unconstrained function = sequential computation
- detrimental to:
 - parallelism
 - incrementalism too

**Incrementalism is
parallelism with your
former self!**

I ♥ associativity

- $(= (f (f a b) c) (f a (f b c)))$

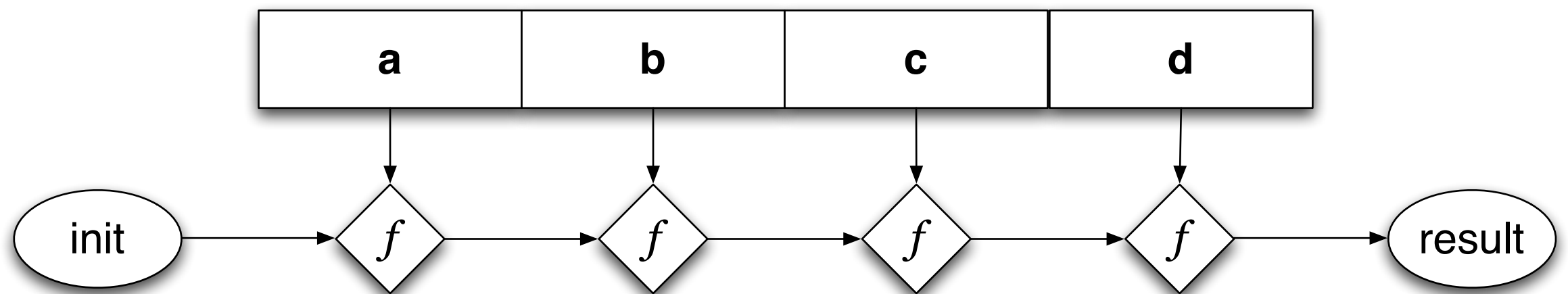
- $(=$   $)$

Sad fact

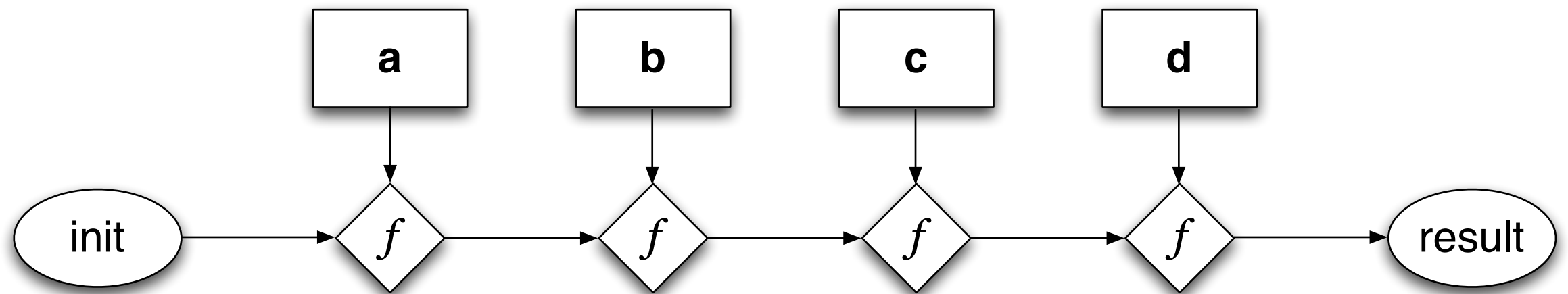
- Most functions aren't associative
- parse-step is not associative
 - $\text{state} * \text{token} \rightarrow \text{state}$
- Associative fns are $A * A \rightarrow A$

Squeezing associativity out of reduce

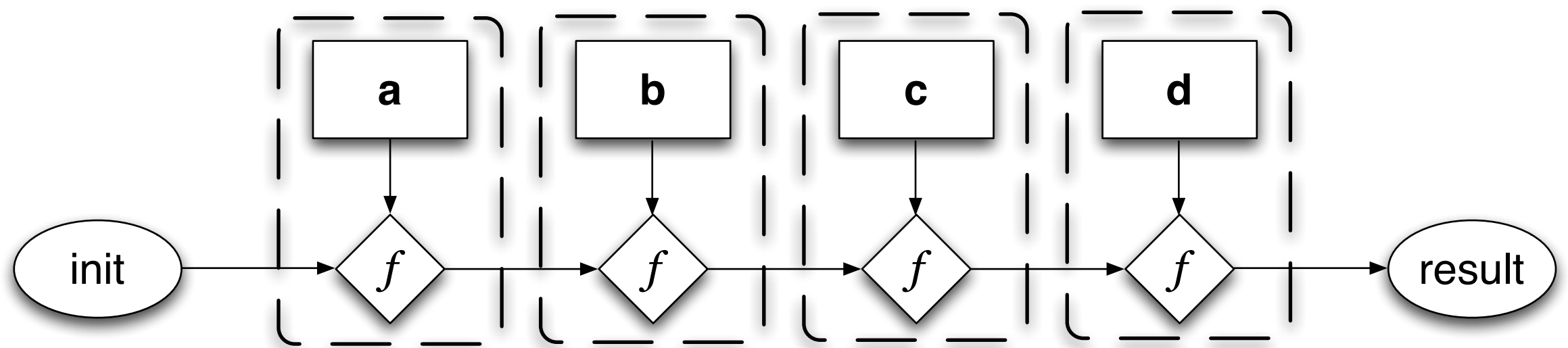
(reduce f init [a b c d])



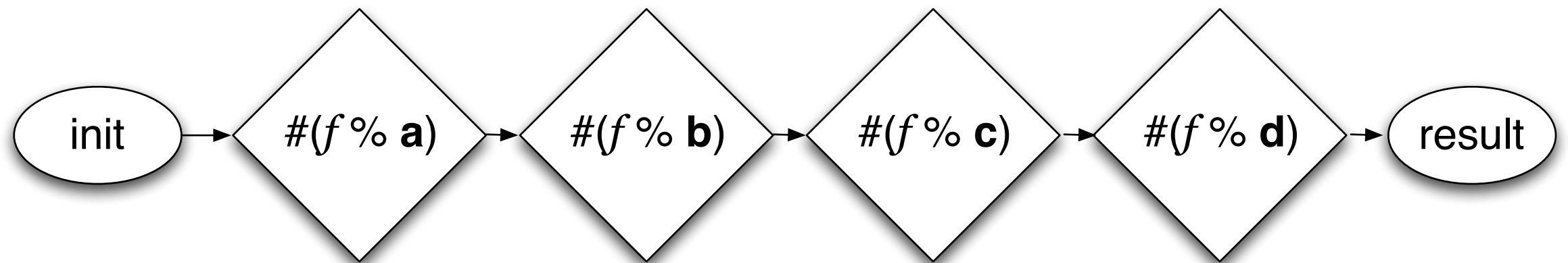
(reduce f init [a b c d])



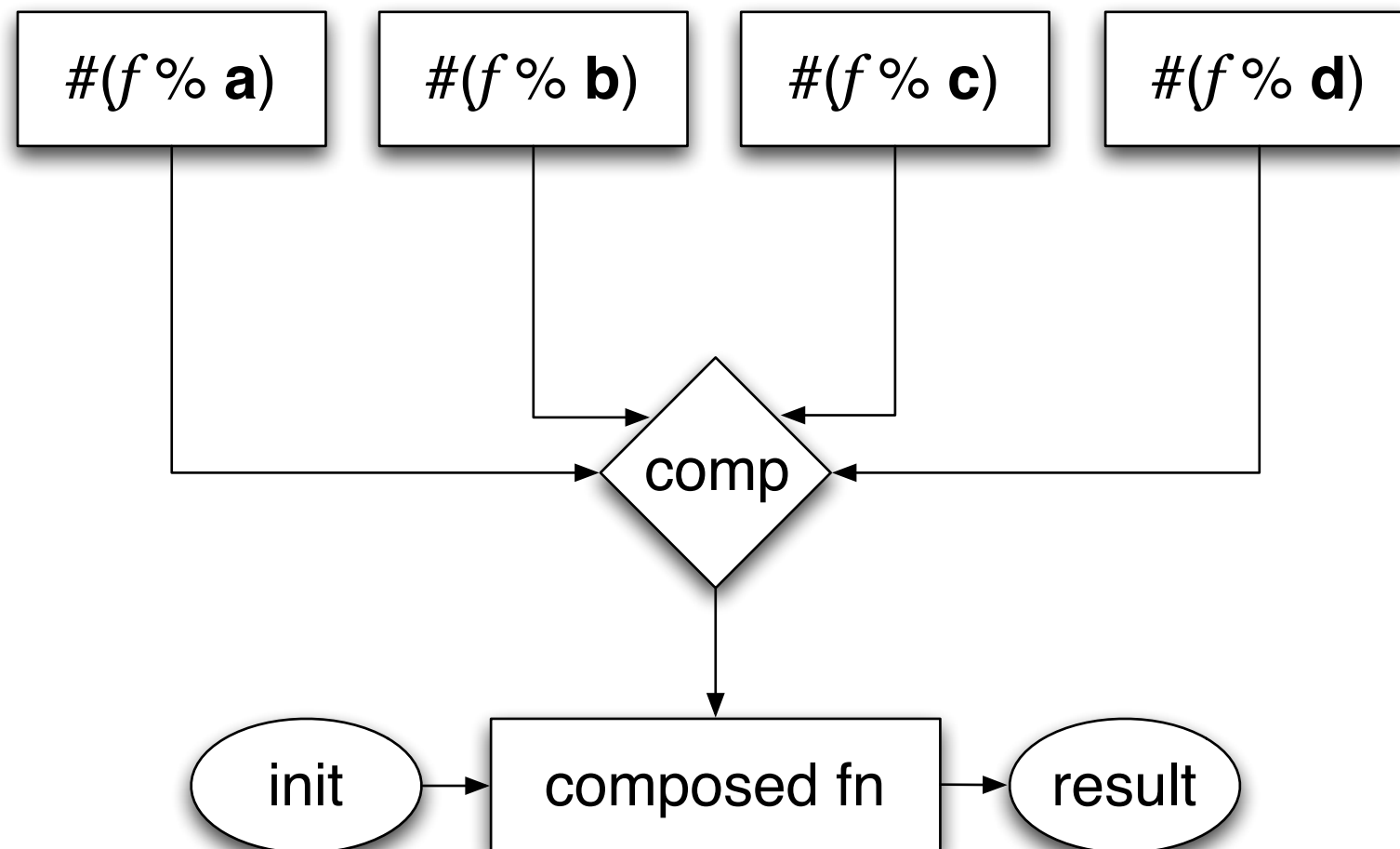
(reduce f init [a b c d])



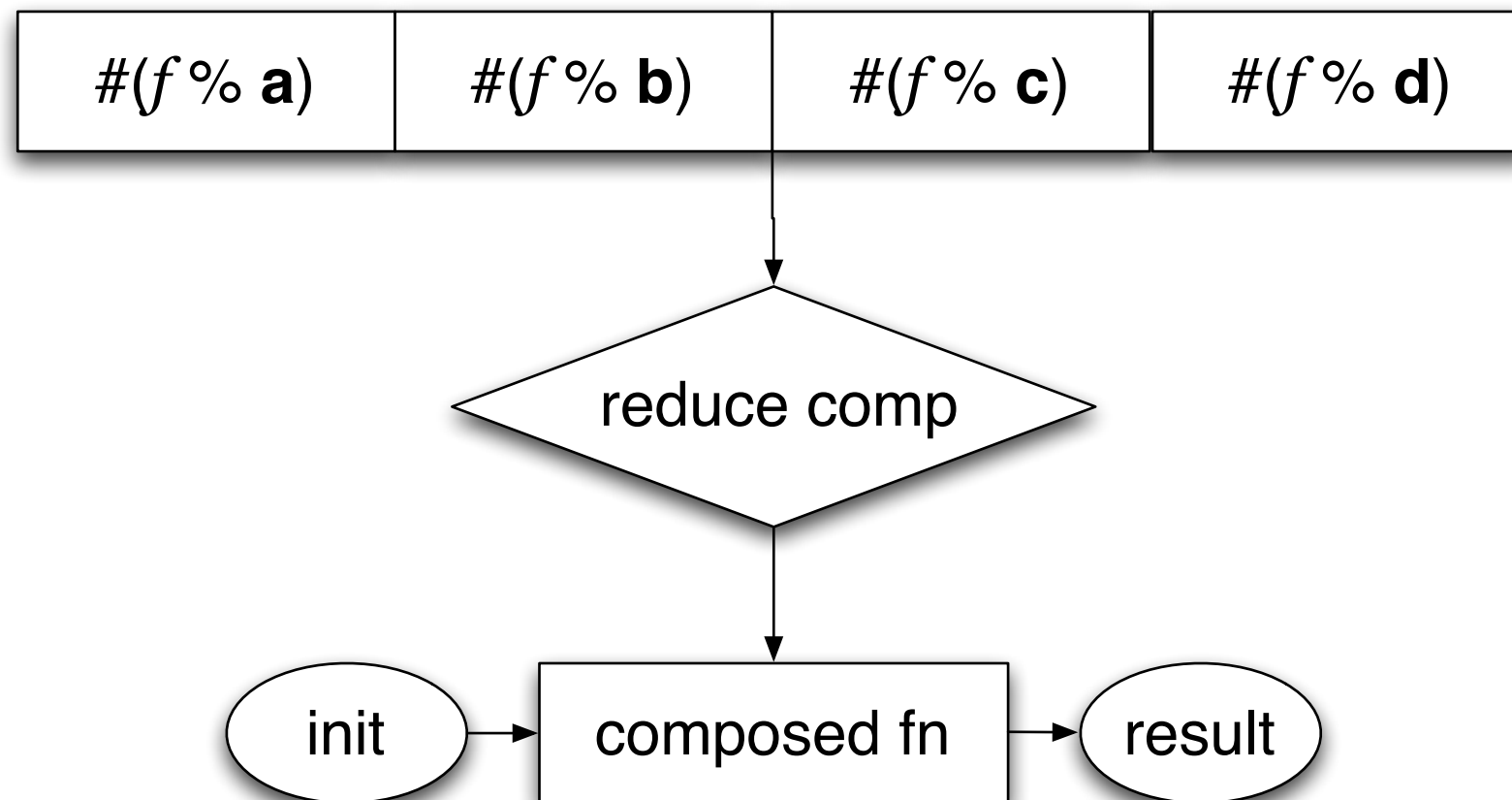
(reduce f init [a b c d])



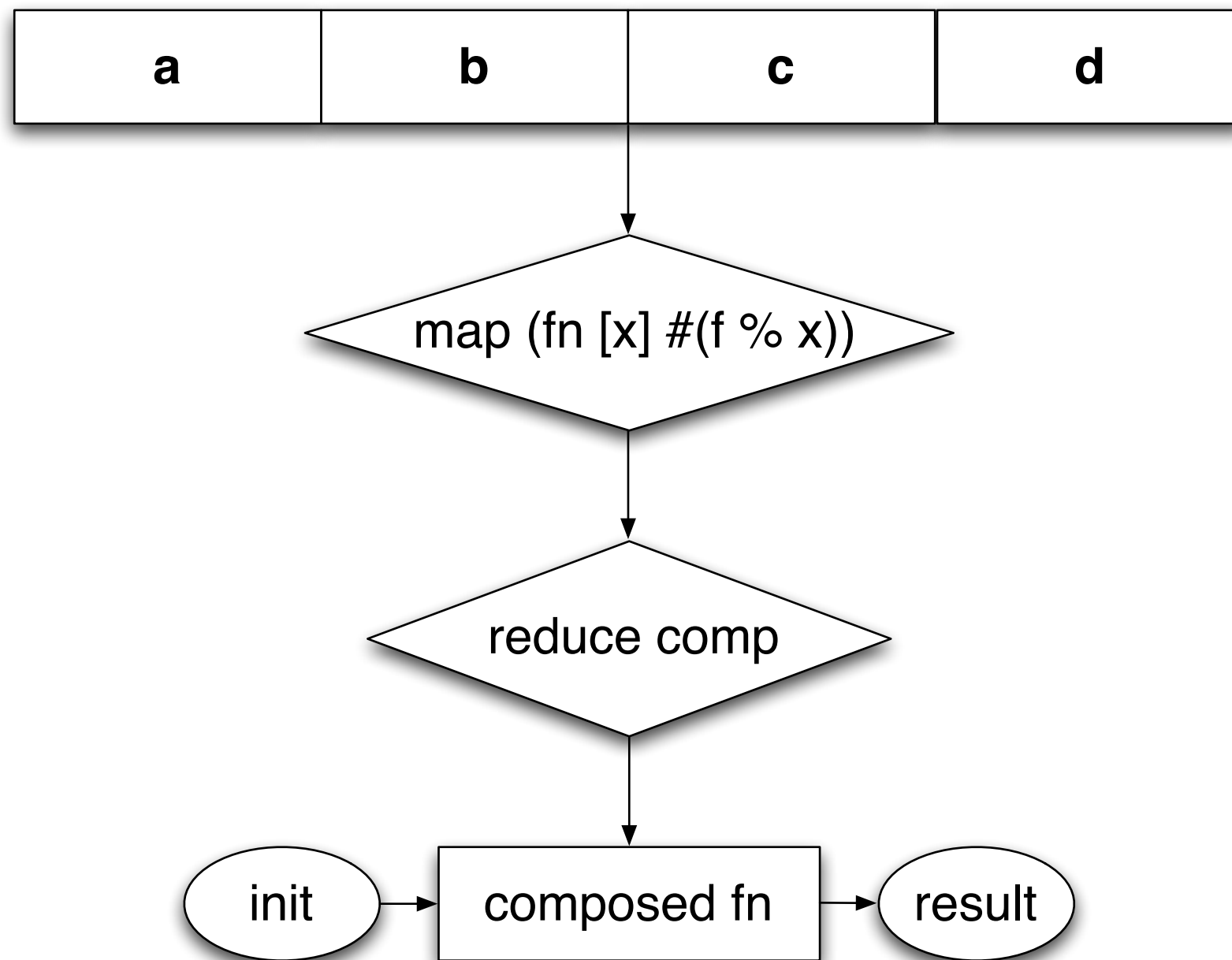
(reduce f init [a b c d])



(reduce f init [a b c d])



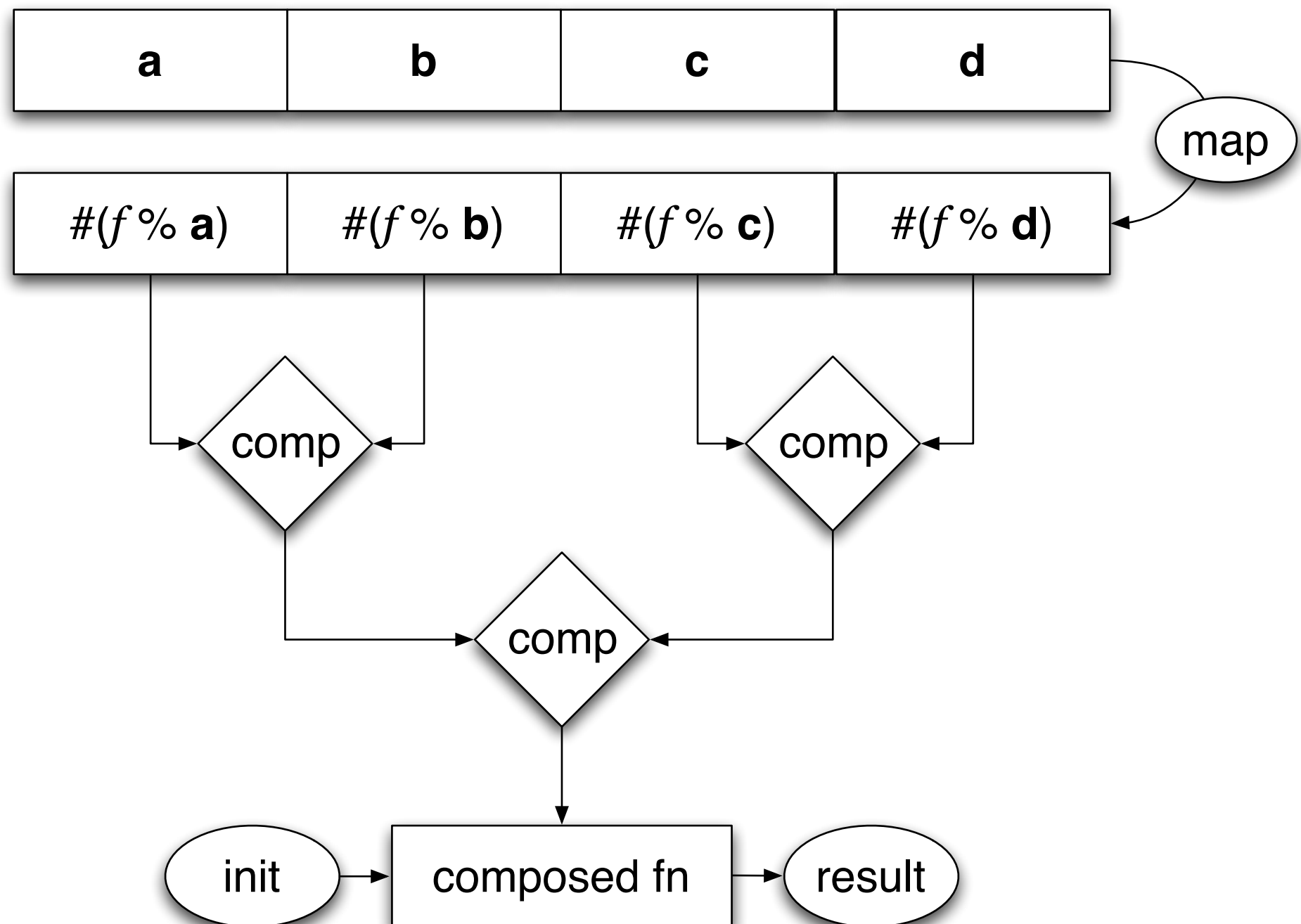
(reduce f init [a b c d])



comp

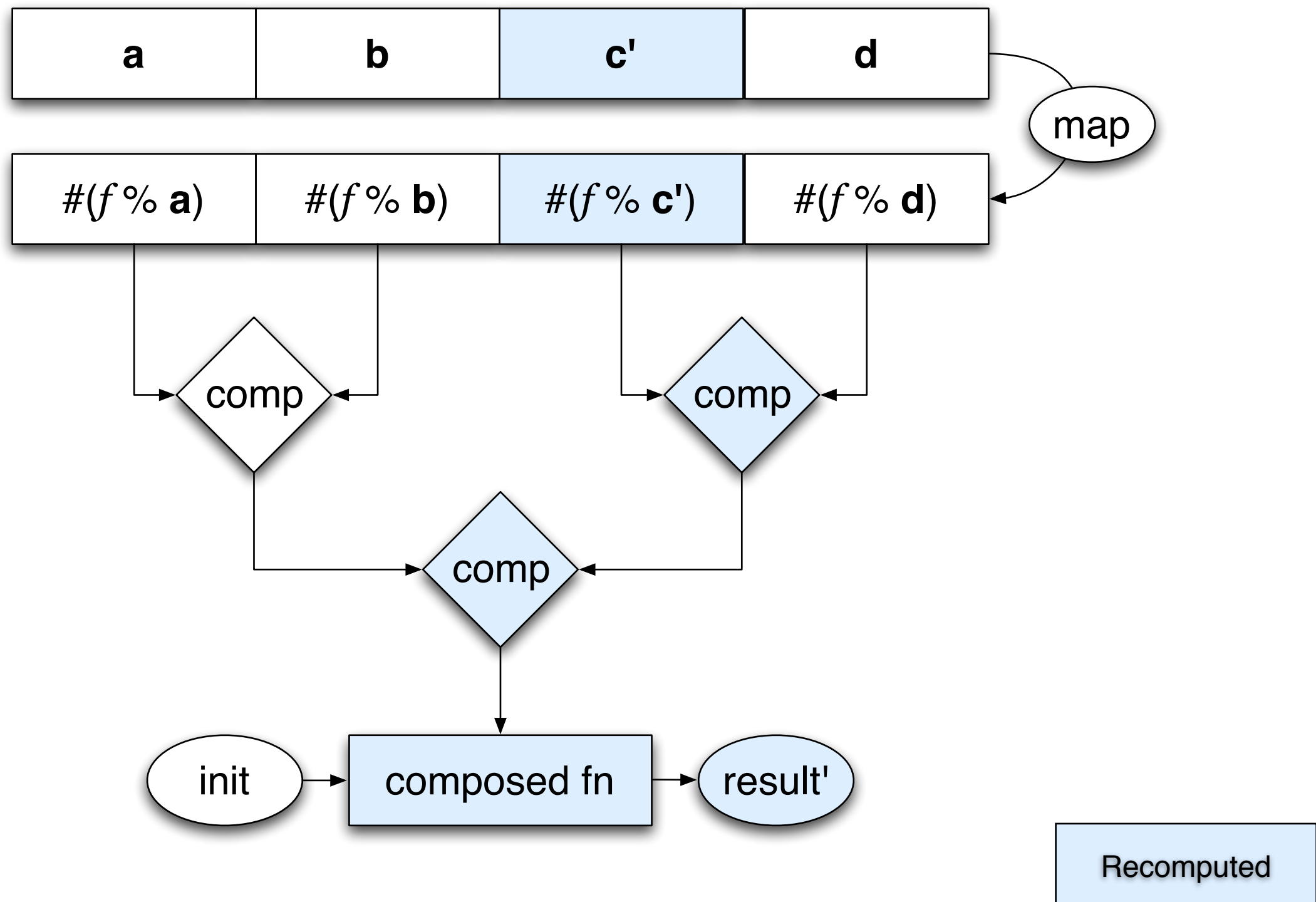
- Associative
 - $fn * fn \rightarrow fn$
 - $(= (\text{comp} (\text{comp } f \ g) \ h) (\text{comp } f (\text{comp } g \ h)))$

(reduce f init [a b c d])



Is this incremental yet?

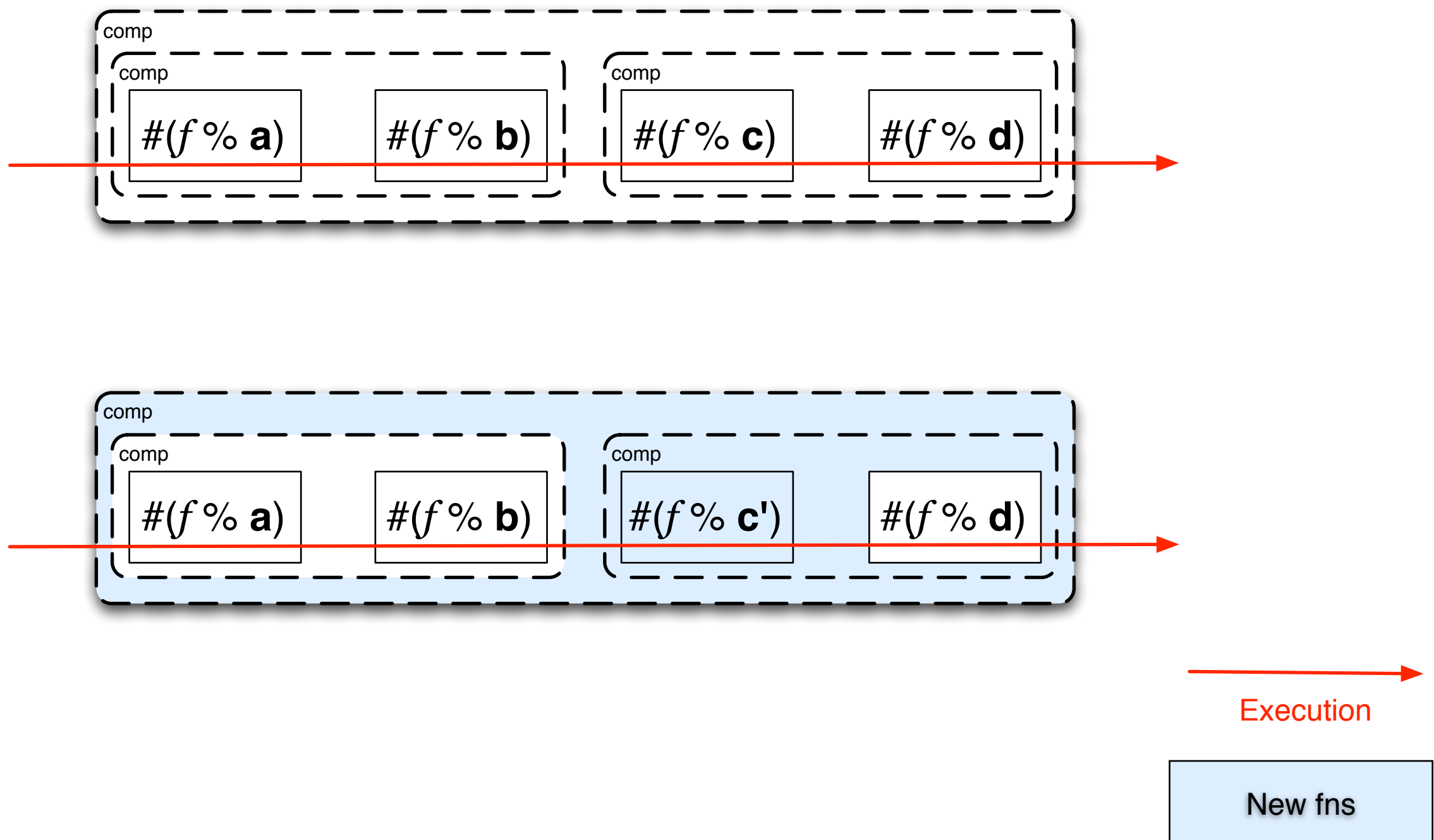
Yes but...



**The composition is
incremental**

Not the *actual*
computation!

Actual computation



Known facts

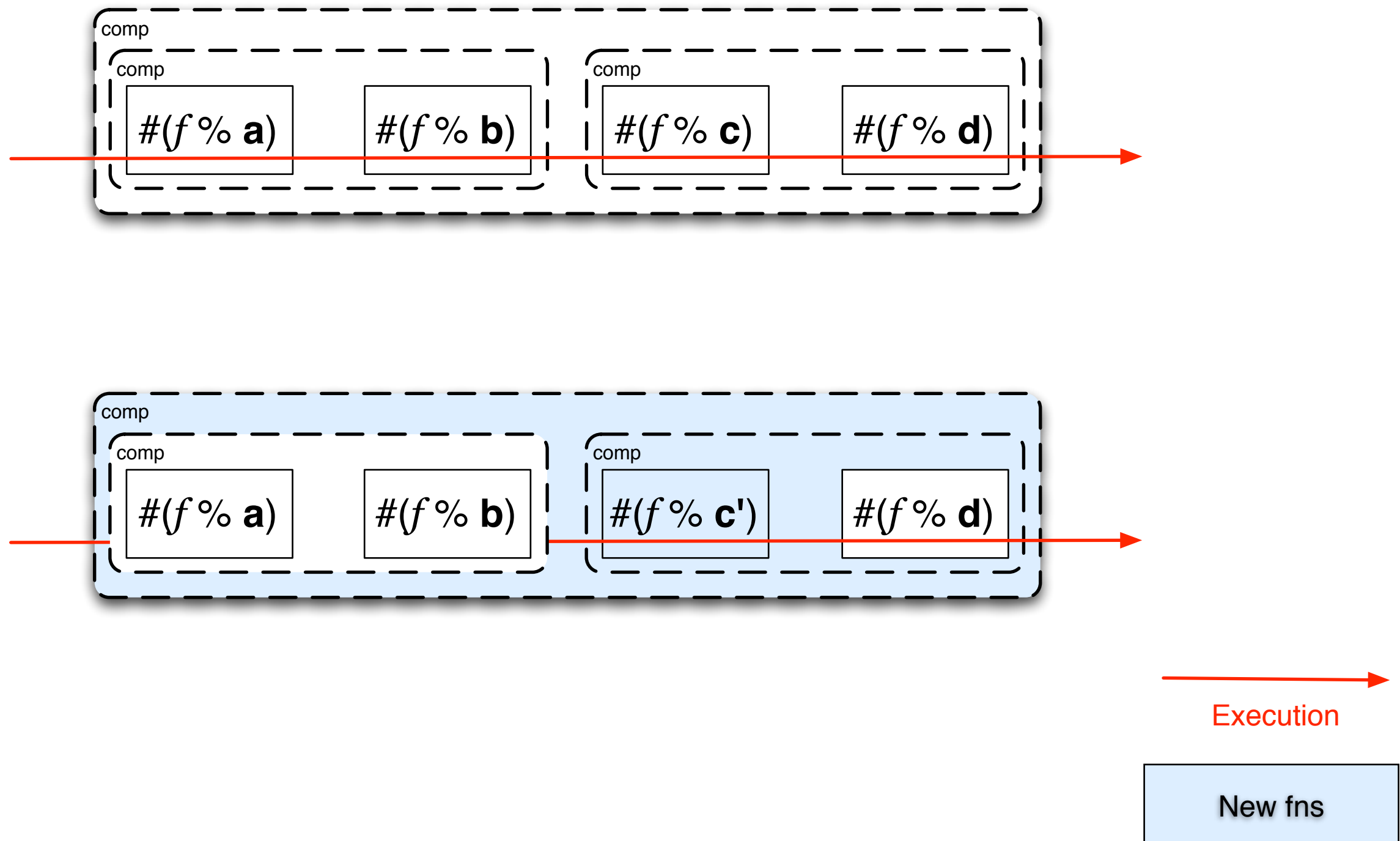
- **init** is constant
- incrementalism presupposes things don't change that much

Let's memoize!

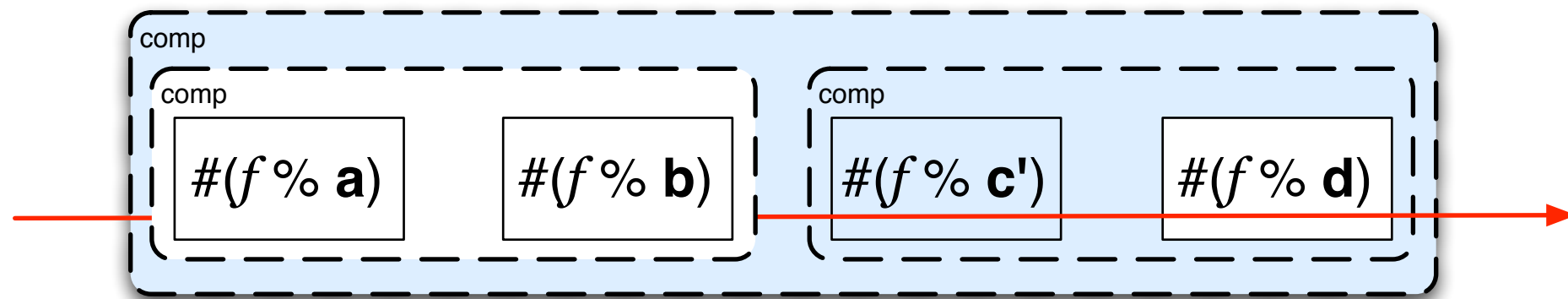
Where?

- each fn returned by the **map** stage
- each fn returned by **comp**
- comp/map themselves are *not* memoized

With memoized nodes



It's better

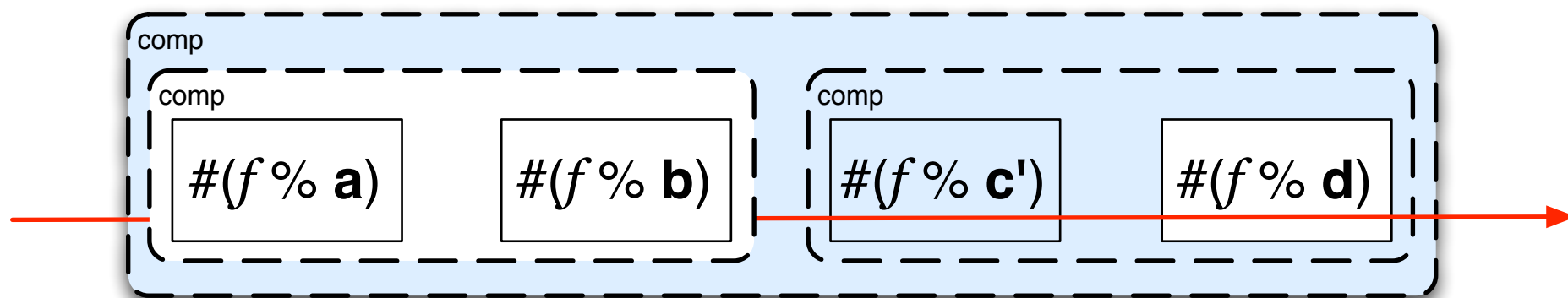


- Some computation is skipped but...
- Everything is still recomputed after the first modification.

→
Execution

New fns

It's better



composition tree + memoized fns

=

restartable

→
Execution

New fns

Much ado about
nothing

Refining memoization

- Once an item differs, the output differs
 - Subsequent memoized results aren't used
- How to leverage past computations?
 - Need to look at the state passed around
 - Function-specific

What's the function?

- A parser step fn!
- Minimally fancy algorithm:
 - LR + contextual tokenizer

LR

- A pushdown automaton
 - One finite state machine
 - Two stacks
 - States stack
 - Nodes stack

a LR step

- current state change:
 - push/peek/pop on stacks
- deep stack items are ignored
 - not even read
 - stack lengths are ignored too

a LR step

- Depends only on:
 - N top items of each stack

What's in a stack?

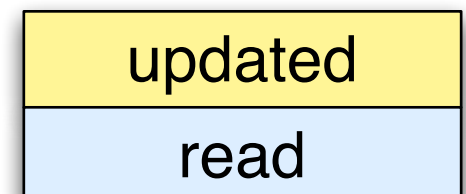
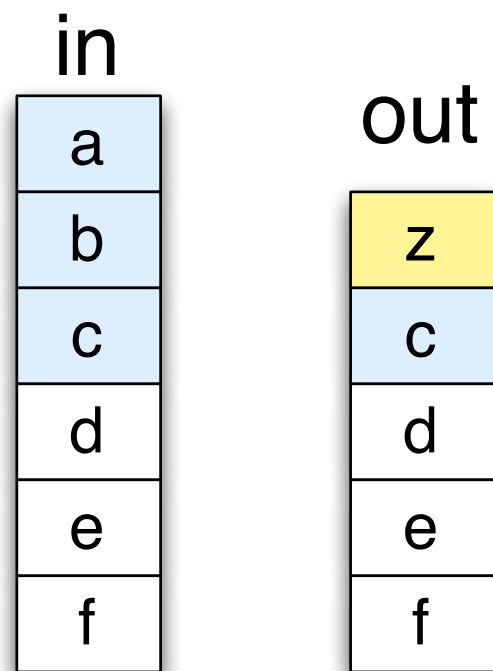
- Stratified by construction:
 - local importance near top
 - distant/global importance near bottom

LR step conclusion

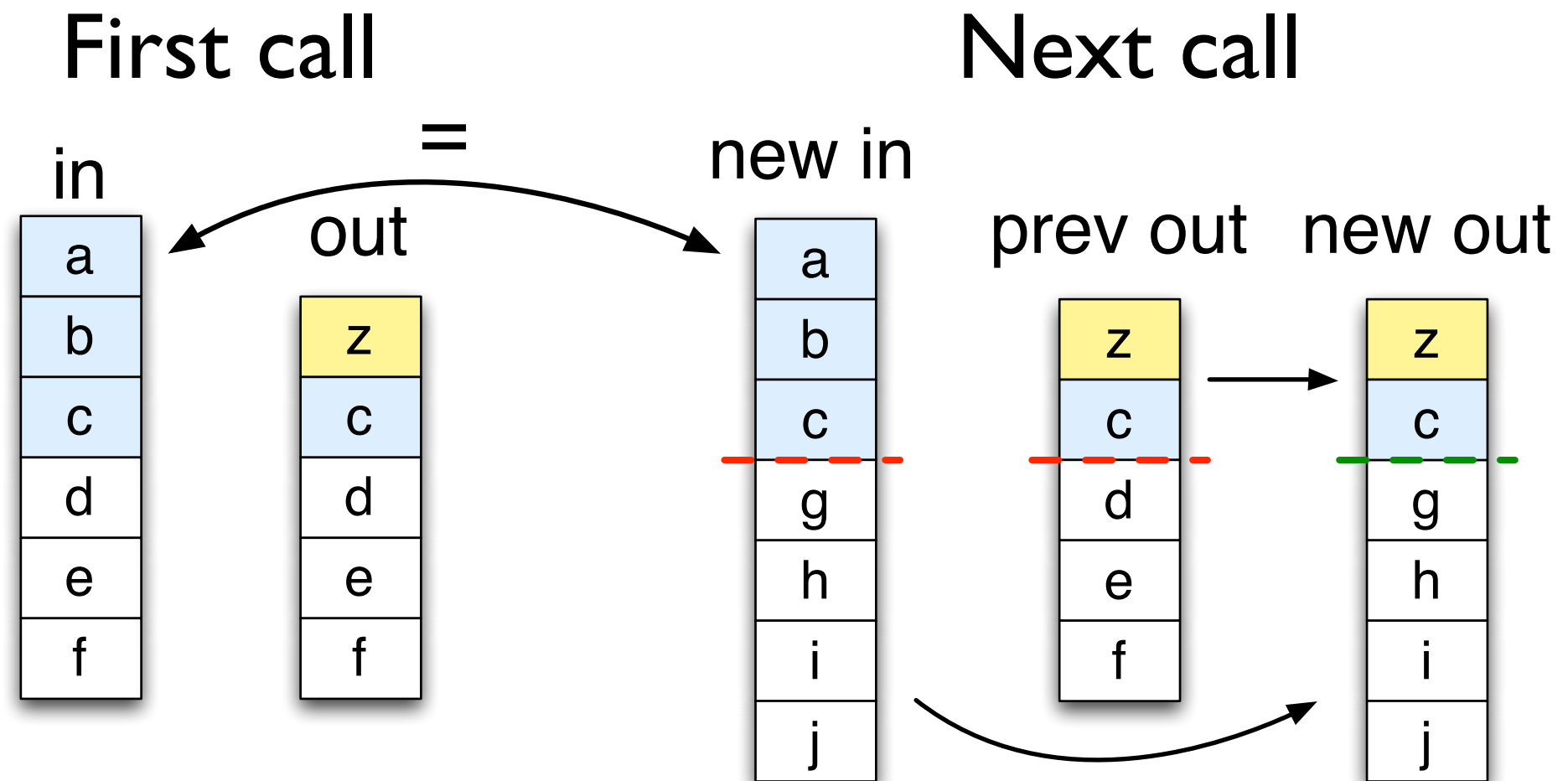
- Steps are not influenced that much by distant changes
 - As long as current state and enough stack top items don't change
- True for compositions too
 - Skipping several steps at once!

Stack transplantation

First call



Stack transplantation



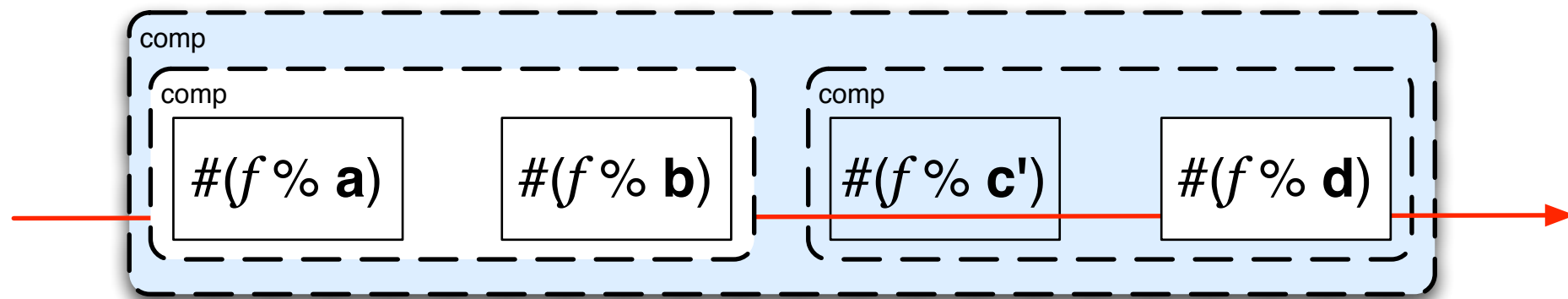
updated

read

Stack transplantation

- «differential» memoization
 - like memoization but works for inputs resembling known inputs
 - tweaks memoized output to match new inputs
 - when input too different, call the actual fn

Transplantation



- Best case above: everything is reused
- Runs in logarithmic time
- Incremental, at last!
- Worst case: restartable

New fns

Transplantation

- Two fns are needed:
 - One to approximate dependencies between output and input
 - One to *quickly* compute a new output based on a new input matching dependencies of a known [in out] pair.
- Totally dependent on the algorithm

Transplantation

- So dependent on the algorithm that:
 - The exposed transplantation works only for the state stack
 - Too coarse for nodes stack
 - Another strategy is used

Dependencies

- I use side-effects
 - tracing ops on stacks
 - easy, fast, expedient
 - scoped to the step fn
- Step fn now returns:
 - Actual result + dependencies info

Inside the step fn

- Stacks are made mutable
 - Reduces object churn
- Processes a chunk of text
 - Reduces memory overhead
 - Reduces «context» switches (persistent/mutable)

The buffer

- Parsley API
 - (parser options & grammar)
 - (incremental-buffer parser)
 - (edit buffer offset length s) ; composition
 - (parse-tree buffer) ; actual computation

The buffer

- Is the composition tree
 - edits on the buffer compute a new composition
 - Input is chunked
 - One fn to perform all updates

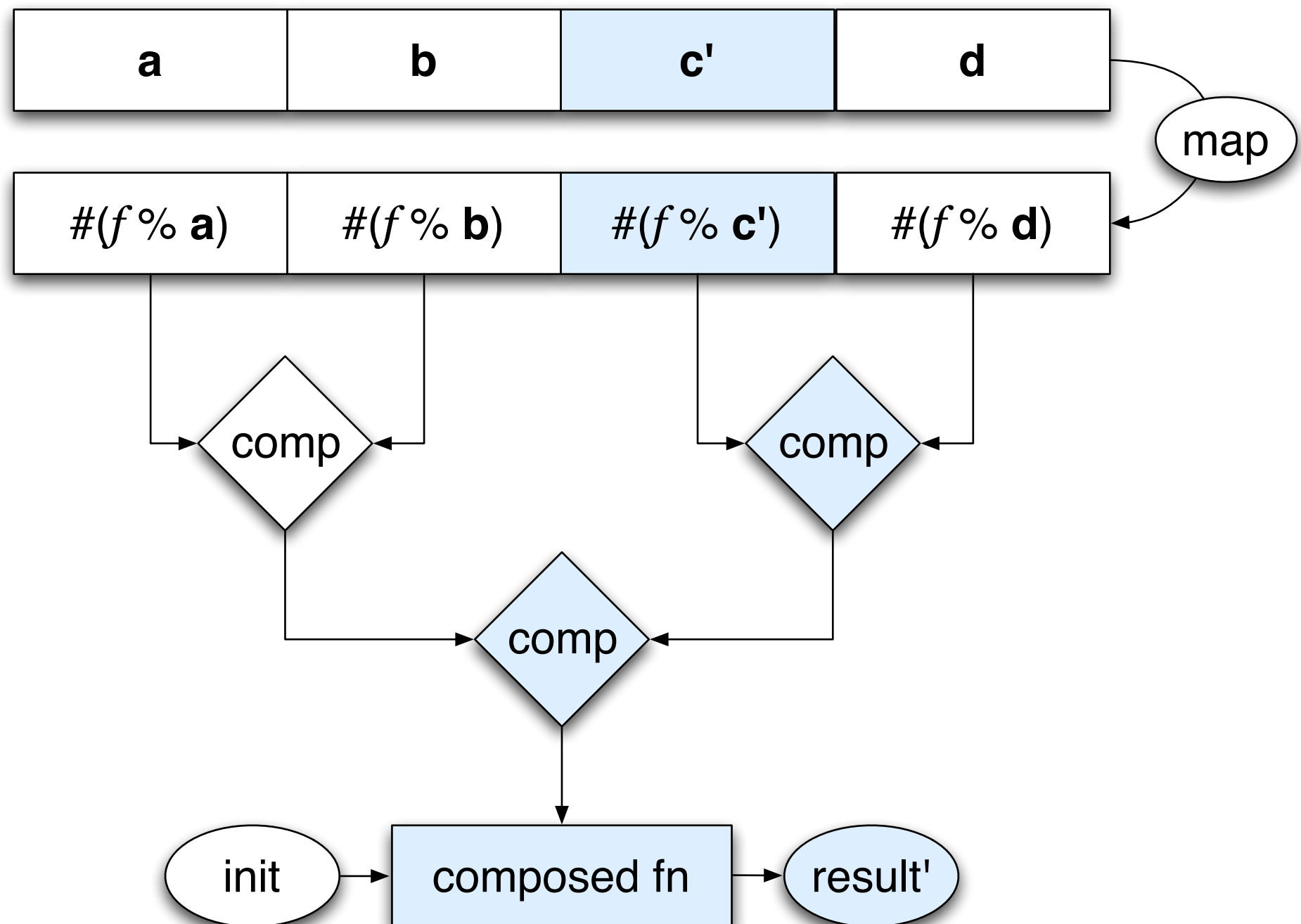
Composition tree

- Reasonably balanced: 2-3 tree
 - Leaves are partial fns
 - Internal nodes are compositions
 - All nodes know their length
 - To find where to perform edits

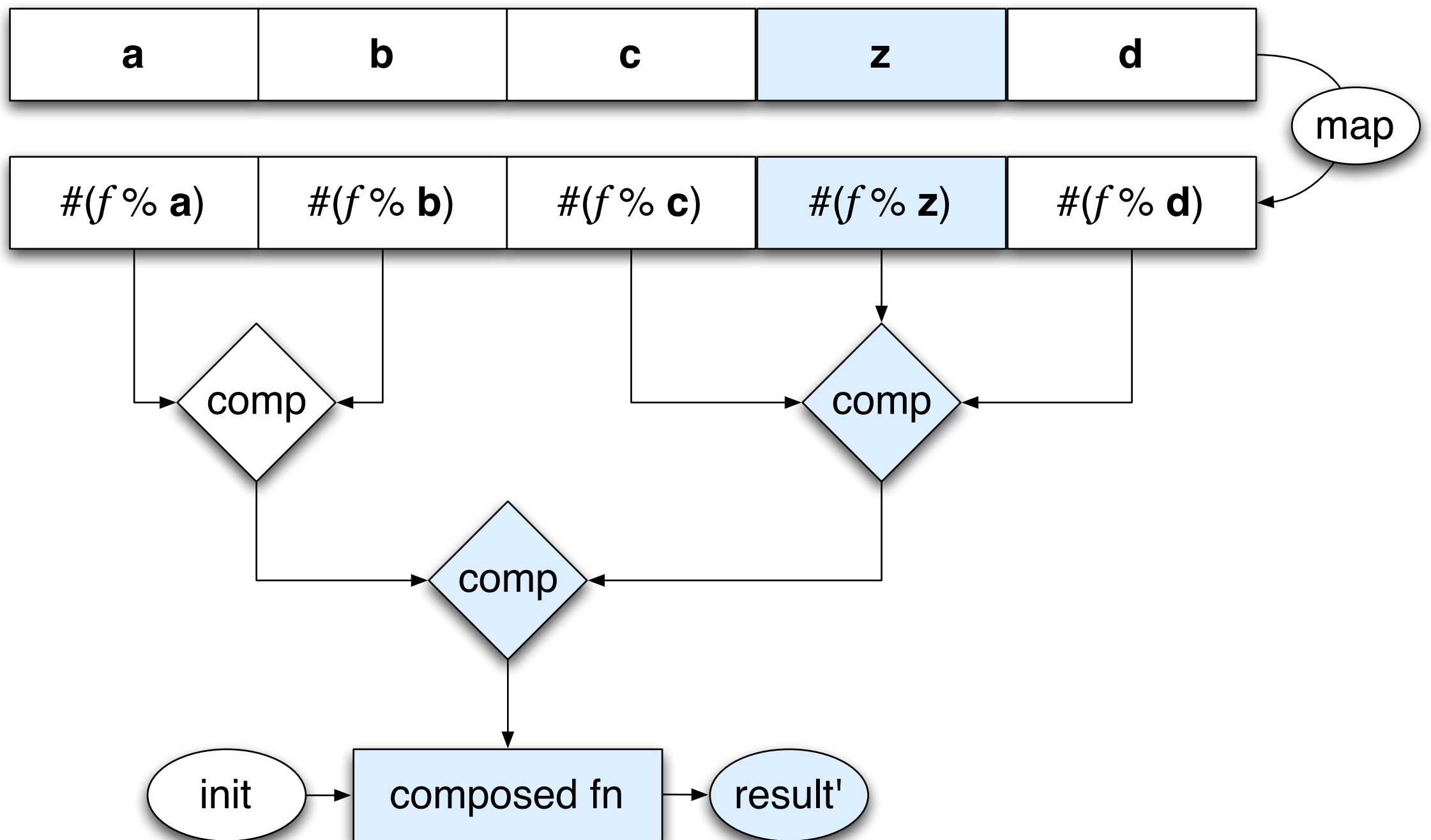
Composition tree

- A finger-tree may fit but:
 - The 2-3 tree was easier
 - No need for privileged access to ends

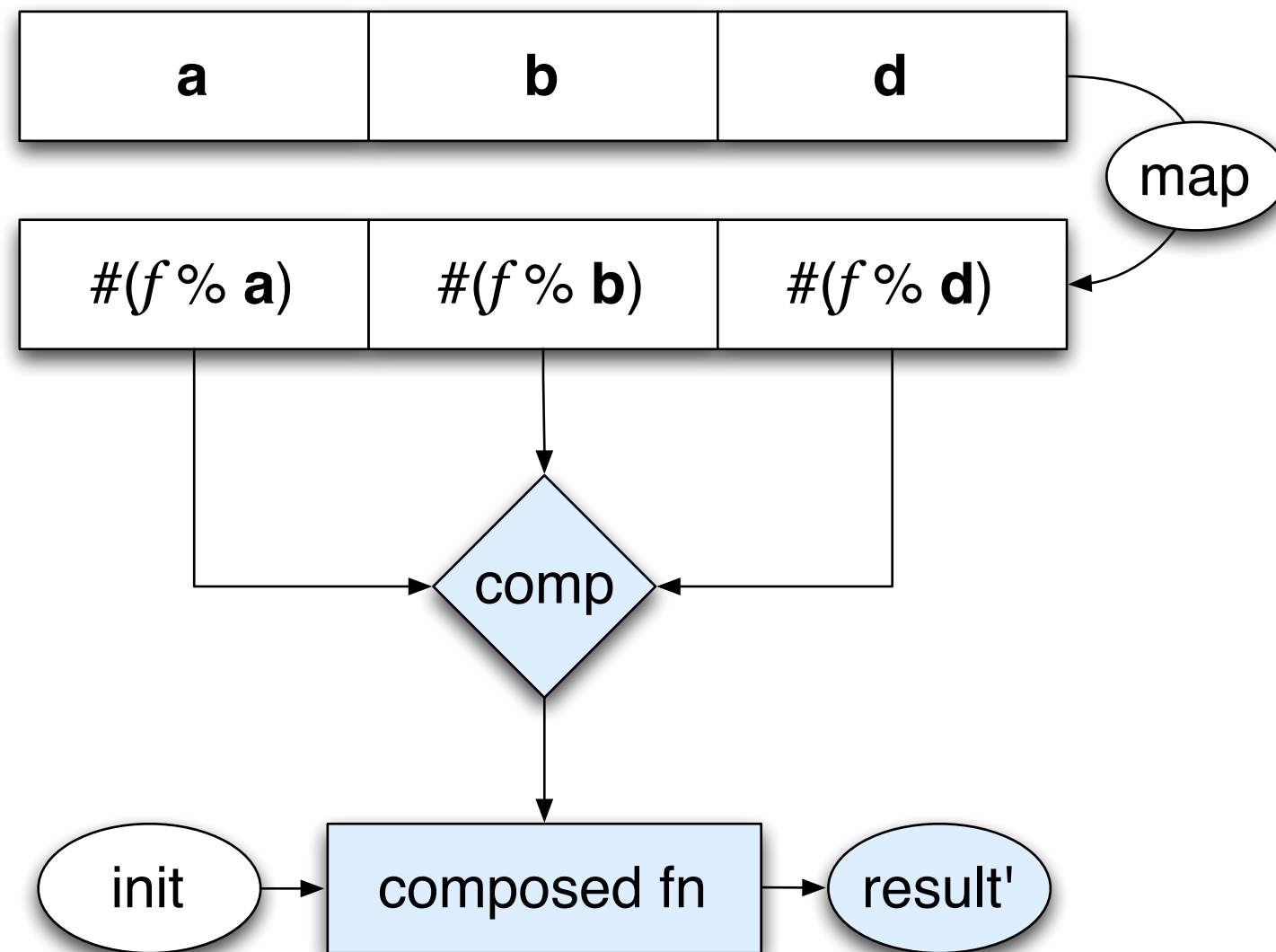
Update



Insert



Delete



Demo time!

What's next?

- Refine dependencies computation
 - Think hard how to mechanize that
- Try to incrementalize other parsing techniques (namely GLR)
- Minor but practical feature: bridge with the Regex DSL
- Make the imperative code less brutal
- More caching strategies

Thanks!