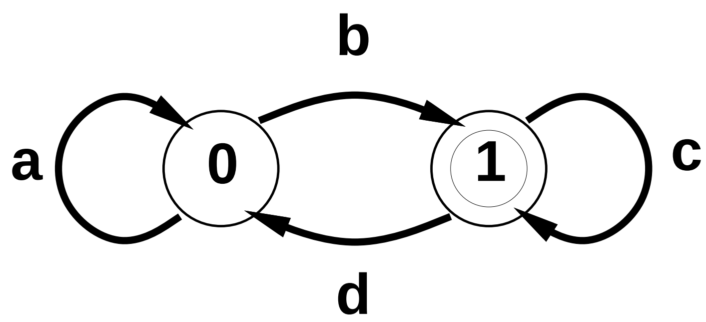
Cycle Therapy

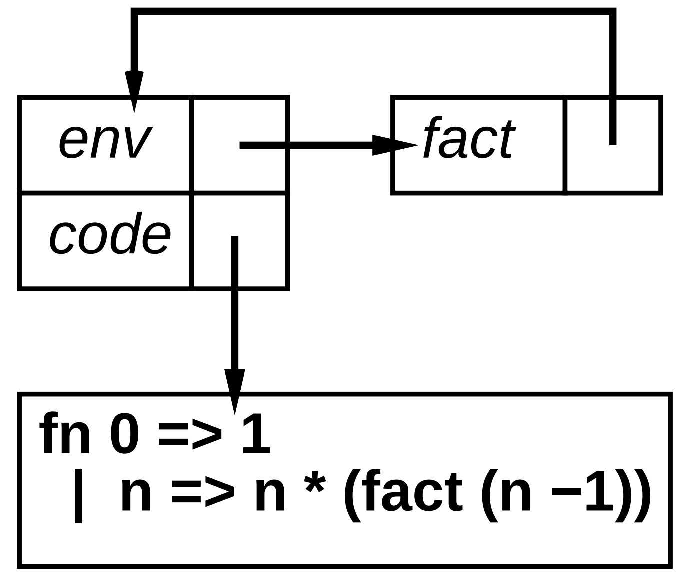
A Prescription for Fold and Unfold on Regular Trees

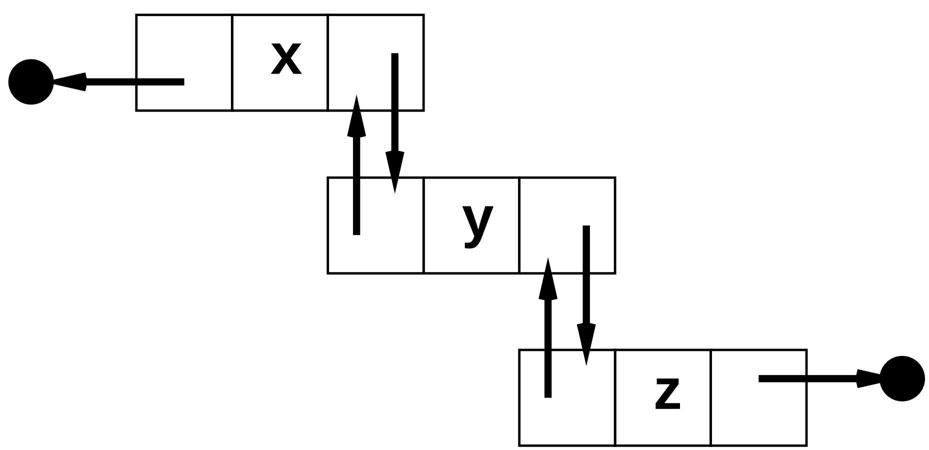
Franklyn Turbak — *Wellesley College*

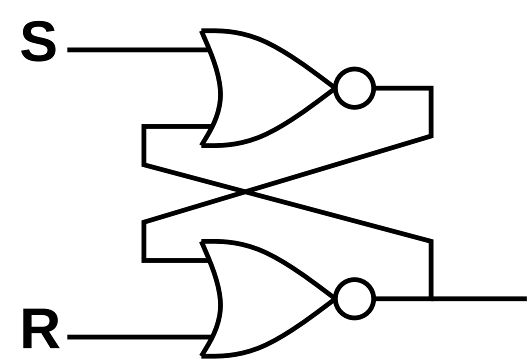
J. B. Wells — *Heriot-Watt University*

Cyclic Structures Are Ubiquitous









Digression: Strictness

Let $\bot$ (pronounced "bottom") stand for a computation which diverges (e.g., loops infinitely) or signals an error.

A mathematical function is *strict* in a parameter if the function returns $\bot$ whenever that parameter is $\bot$.

Examples:

- The + operator is strict in both arguments.
- The function `f(x,y) = x` is strict in the x parameter but non-strict in the y parameter.

Digression: Eagerness vs. Laziness

- An *eager* language models all programming language functions as mathematical functions that are strict in all parameter positions. E.g., the previous `f` would be treated as if it were written:

  ```
  f(x,y) = (if y == ⊥ then ⊥ else x)
  ```

  Most programming languages are eager. E.g.: Java, C, C++, Pascal, Fortran, Scheme, ML, ...

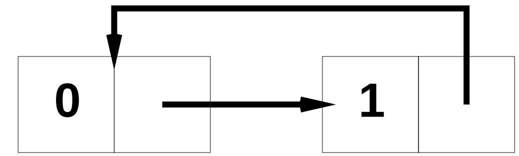
- A *lazy* language models programming language functions with their “natural” strictness. In particular, all data constructors are non-strict in all arguments. E.g.:

  ```
  f(3,(loop)) = 3
  length((loop):(loop):[]) = 2
  ```

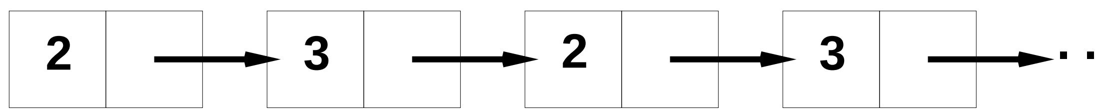
  Haskell is an example of a lazy language.

Cycles Are Tricky To Manipulate

Consider Haskell's `alts = 0:1:alts`



- Naïve generation ⇒ unbounded structures:
  - **let** `inf x y = x:(inf y x)` **in** `inf 2 3`



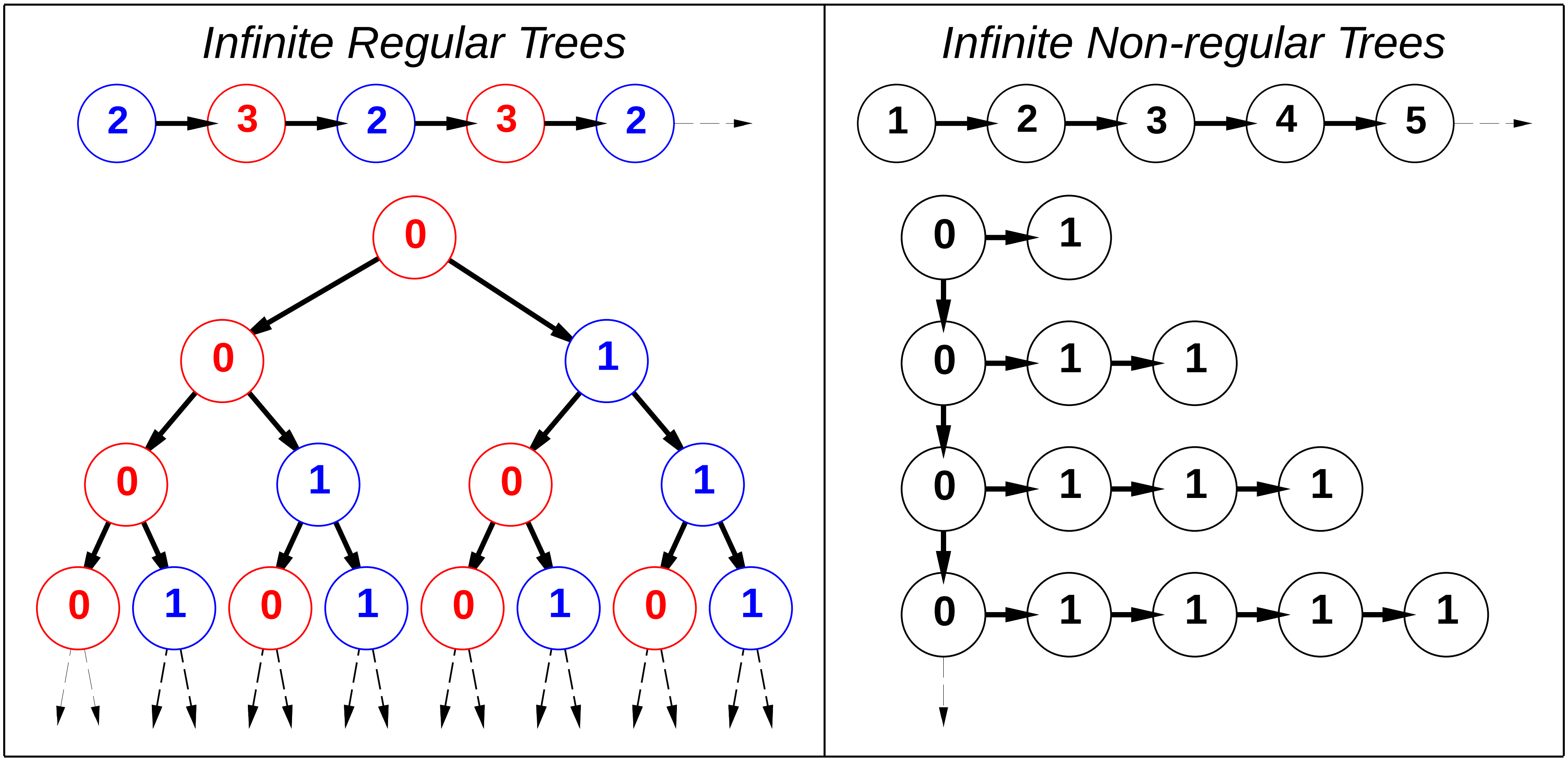
  - `map (\ x -> x + 2) alts`
- Naïve accumulation ⇒ divergence:
  - `foldr (+) 0 alts`
  - `foldr Set.insert Set.empty alts`
- Dependency on language features: laziness, side effects, node equality, recursive binding constructs, etc.

Road Map

- Viewing cyclic structures as infinite regular trees.
- Adapting the tree-generating unfold function to generate cyclic structures for infinite regular trees.
- Adapting the tree-accumulating fold function to return non-trivial results for strict combining functions and infinite regular trees.
- Cycamores: an abstraction for manipulating regular trees that we have implemented in ML and Haskell.

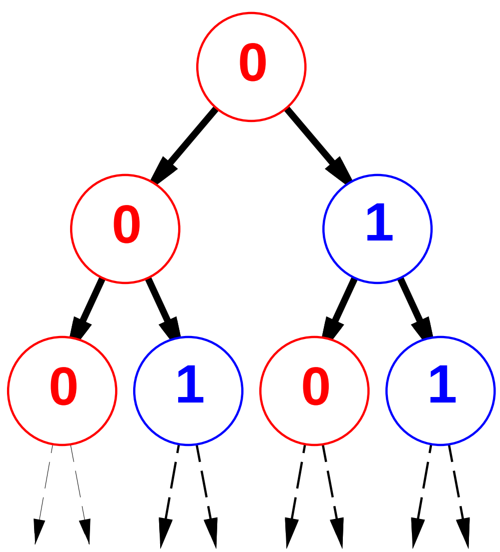
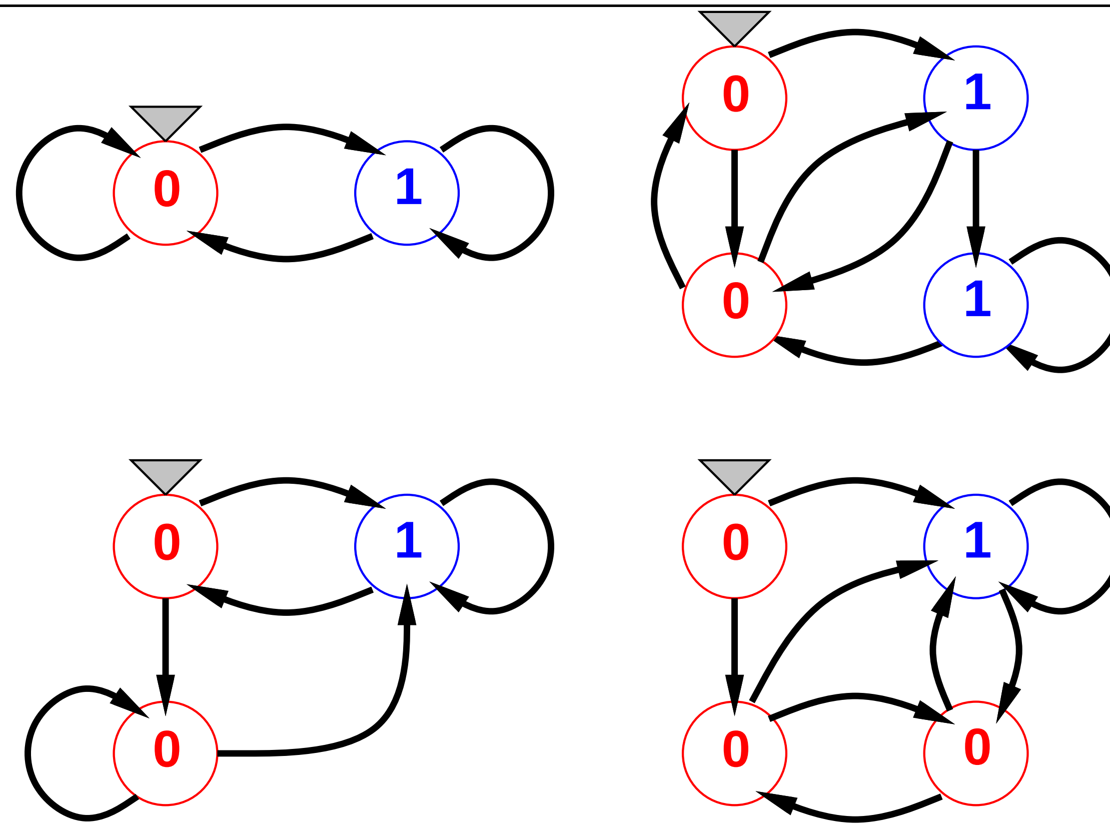
Regular Trees

A tree is *regular* if it has a finite number of distinct subtrees.



Cyclic Representatives

Finite cyclic graphs denote infinite regular trees.
The same tree may be represented by many finite graphs.

Infinite Regular Tree	*Some Finite Cyclic Representatives*
	

Goals

Develop high-level abstractions for creating and manipulating regular trees that:

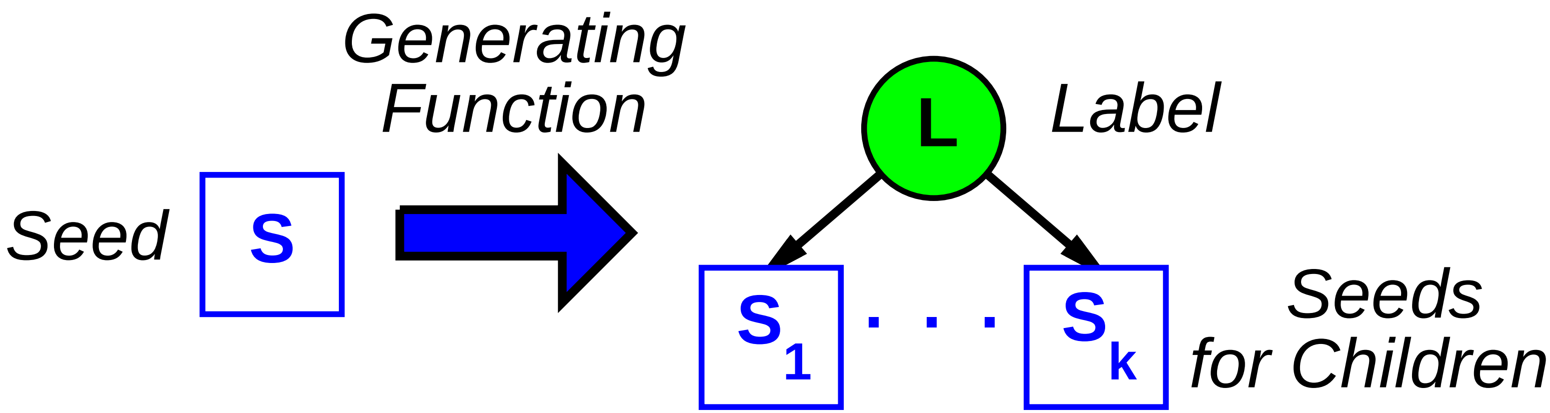
- efficiently represent regular trees using cyclic graphs;
- do not expose the finite representative denoting an infinite regular tree;
- are relatively insensitive to the features of the programming language in which they are embedded.

Road Map

- Viewing cyclic structures as infinite regular trees.
- Adapting the tree-generating unfold function to generate cyclic structures for infinite regular trees.
- Adapting the tree-accumulating fold function to return non-trivial results for strict combining functions and infinite regular trees.
- Cycamores: an abstraction for manipulating regular trees that we have implemented in ML and Haskell.

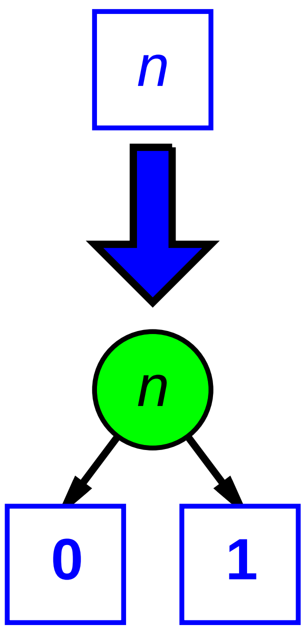
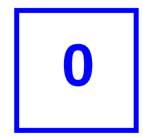
Tree Generation via Unfold

The `unfold` operator generates a tree from a generating function and a seed.

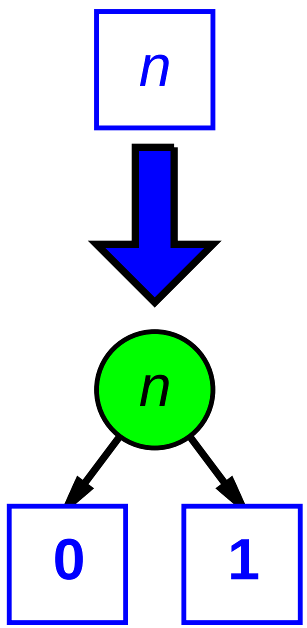
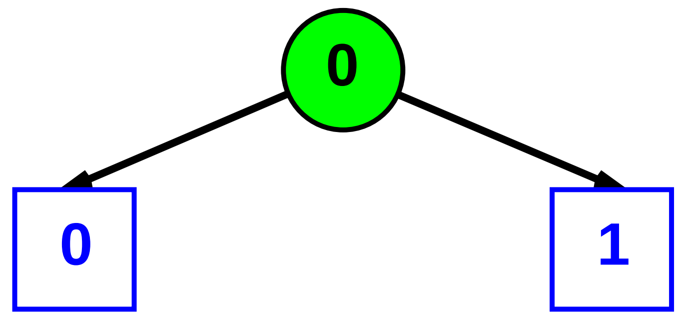


$$\text{unfold} : \underbrace{(S \rightarrow (L \times (S^{\omega})))}_{\text{generating function } \psi} \rightarrow \underbrace{\underbrace{S}_{\text{seed}} \rightarrow \underbrace{\text{Tree}(L)}_{\text{trees over L}}}_{\psi\text{-anamorphism}}$$

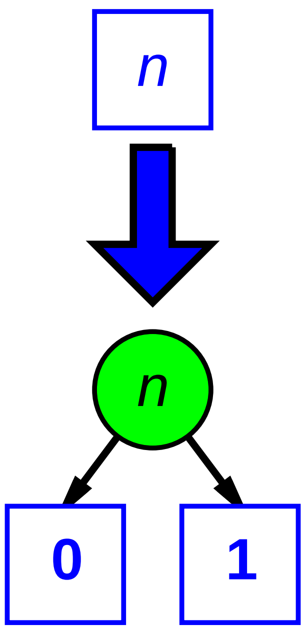
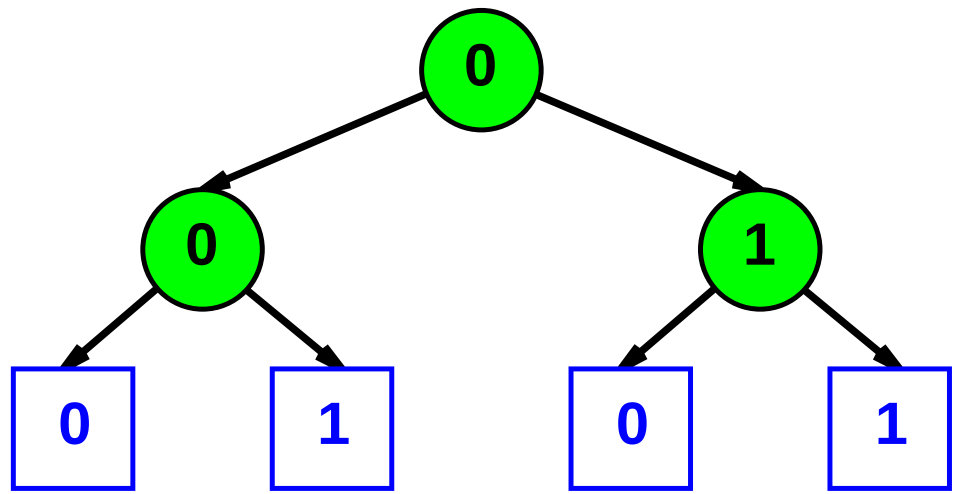
Unfold Example 1: Regular Tree

Generating Function ψ	*Tree*
	

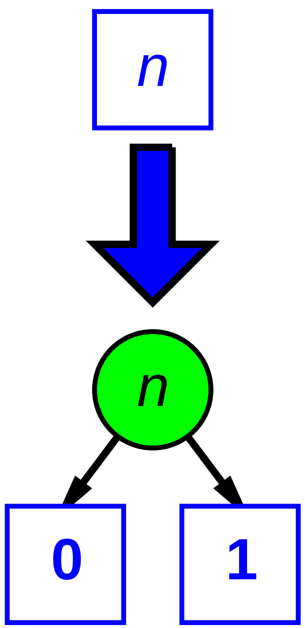
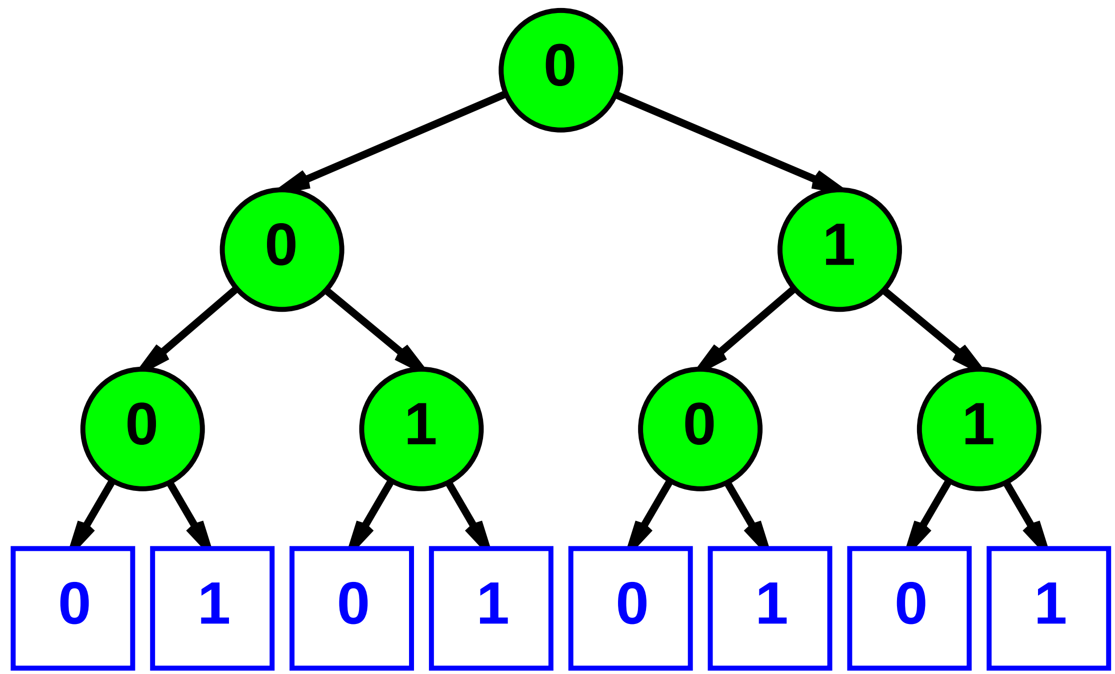
Unfold Example 1: Regular Tree

<table>
<tr><th>Generating Function ψ</th><th>Tree</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

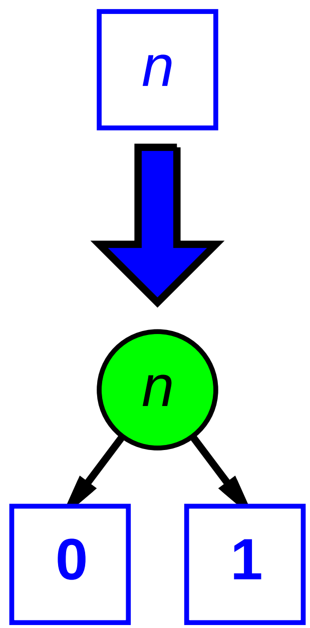
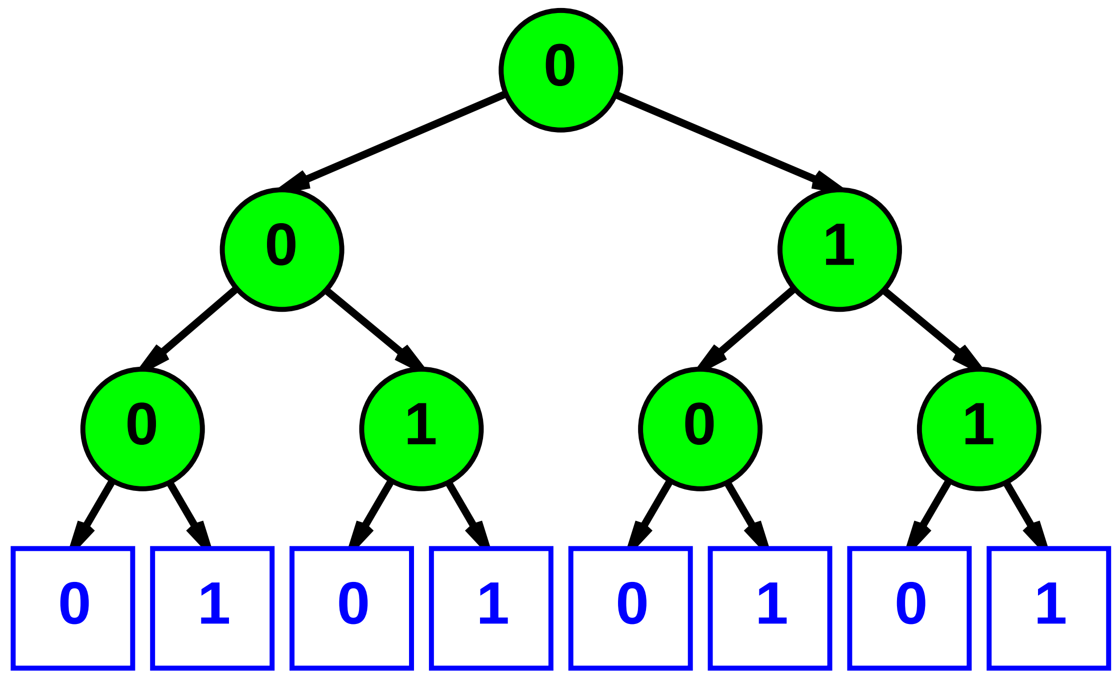
Unfold Example 1: Regular Tree

Generating Function ψ	*Tree*
	

Unfold Example 1: Regular Tree

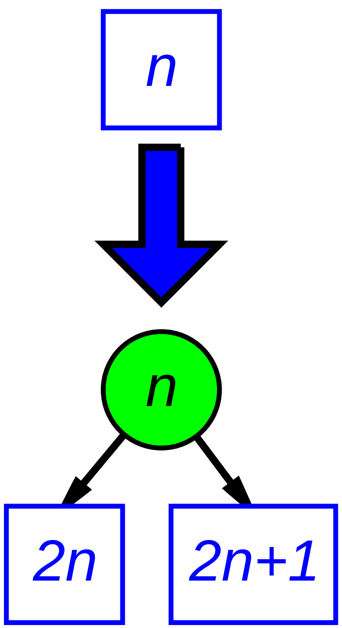
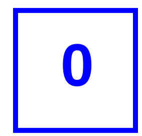
Generating Function ψ	*Tree*
	

Unfold Example 1: Regular Tree

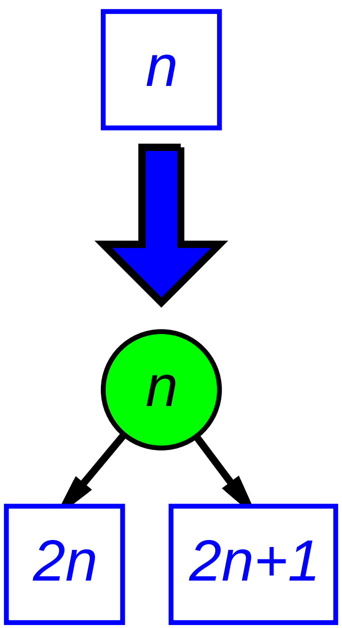
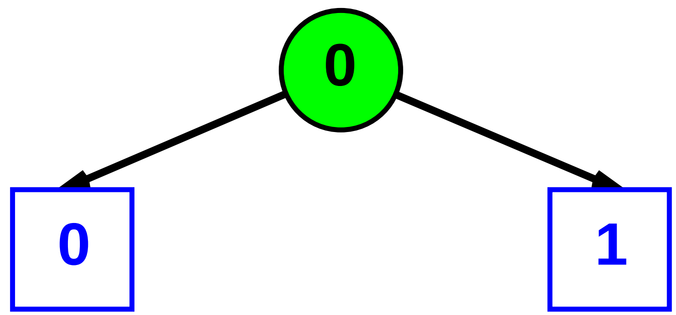
Generating Function ψ	*Tree*
	

$$\mathsf{deps}(0, \psi) = \{0, 1\}$$

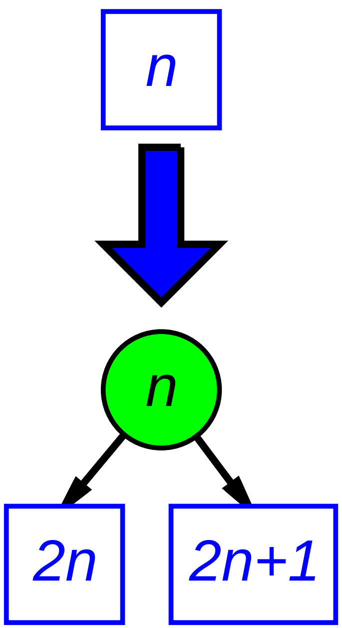
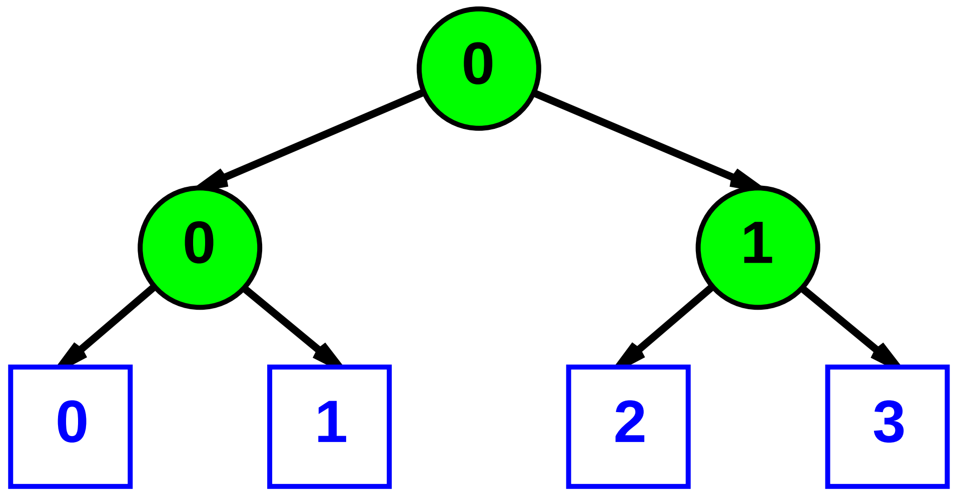
Unfold Example 2: Non-regular Tree

Generating Function ψ	*Tree*
	

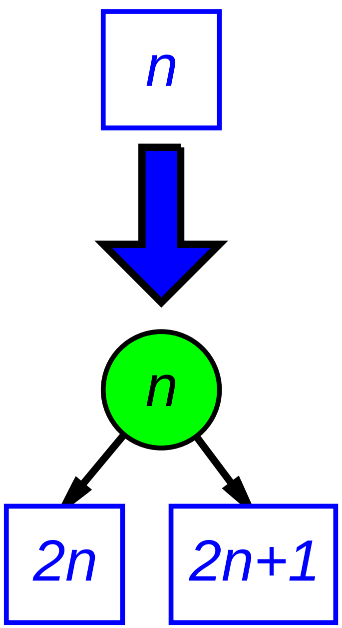
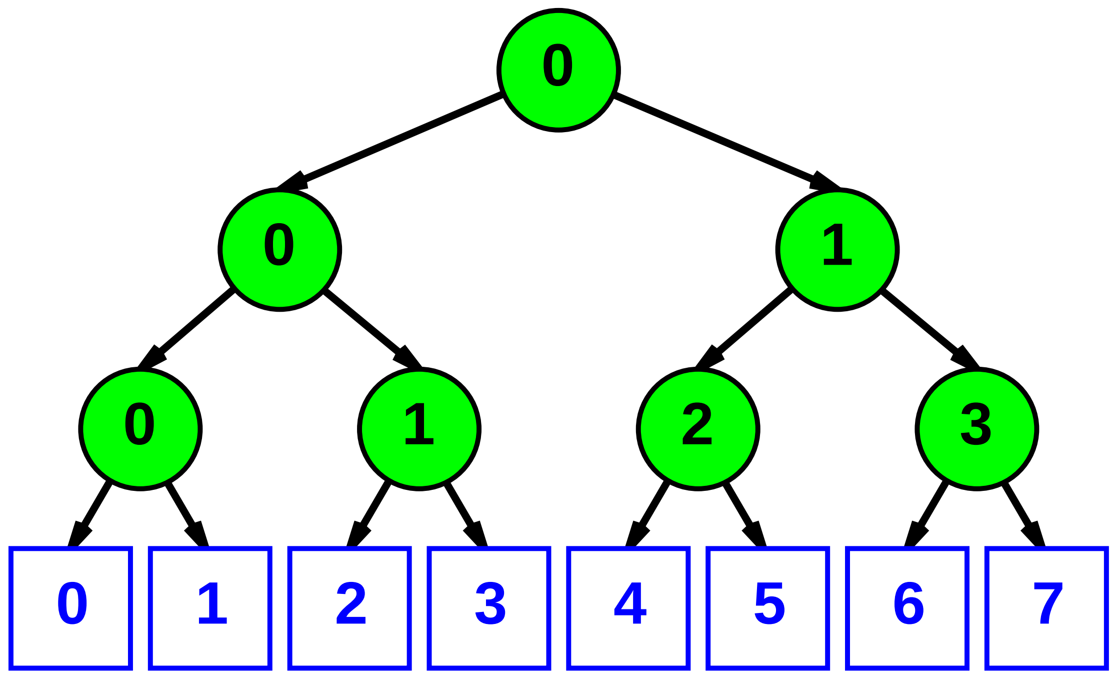
Unfold Example 2: Non-regular Tree

<table>
<tr><th>Generating Function ψ</th><th>Tree</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

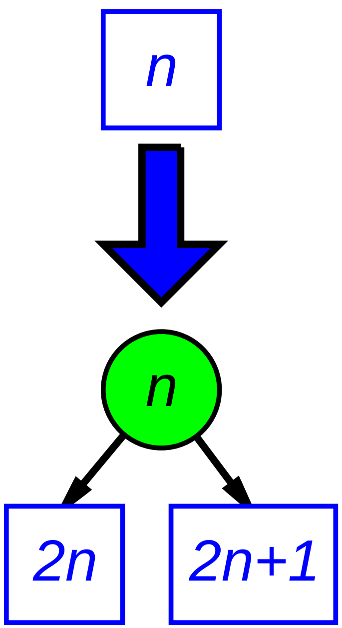
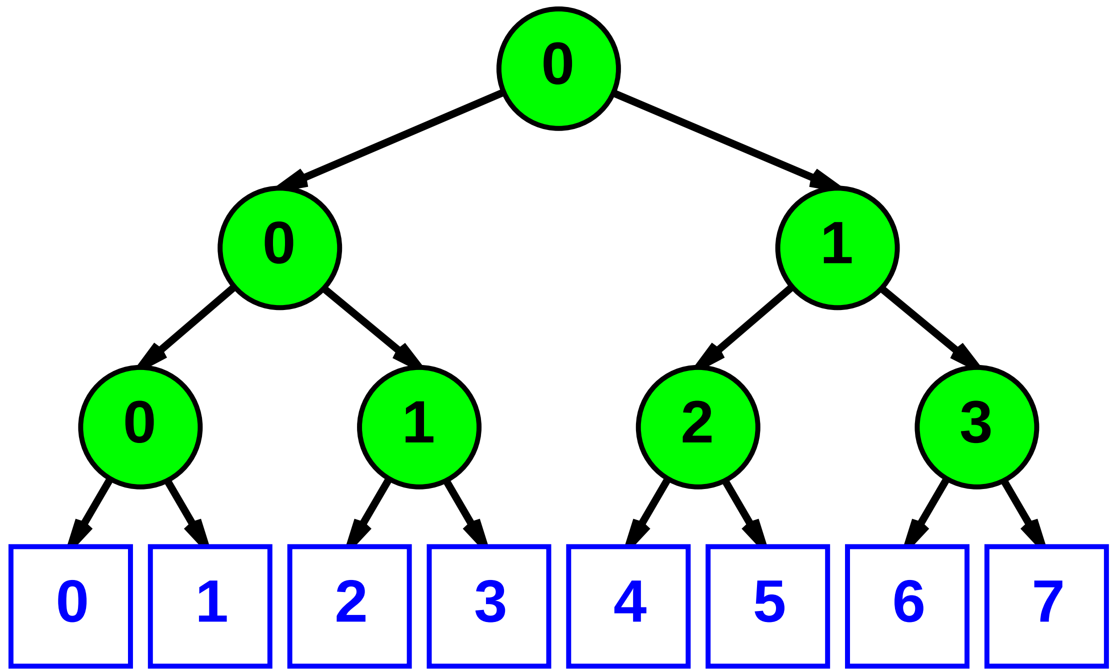
Unfold Example 2: Non-regular Tree

Generating Function ψ	*Tree*
	

Unfold Example 2: Non-regular Tree

Generating Function ψ	*Tree*
	

Unfold Example 2: Non-regular Tree

Generating Function ψ	*Tree*
	

$$\mathsf{deps}(0, \psi) = \{0, 1, 2, 3, 4, 5, 6, 7, \ldots\}$$

Unfold Lemma

If $\mathsf{deps}(x, \psi)$ is finite, then $\mathsf{unfold}(\psi)(x)$ is a regular tree.

- Converse of this lemma does not hold.
- Basis for implementation of unfold that “ties cyclic knots” for (some) regular trees via memoization on seeds (a la Hughes’s *Lazy Memo Functions*, FPCA’85).

Unfold Implementation: Standard ML

generating fcn.: **fun** `P n = (n,[(n+1) mod 2])`

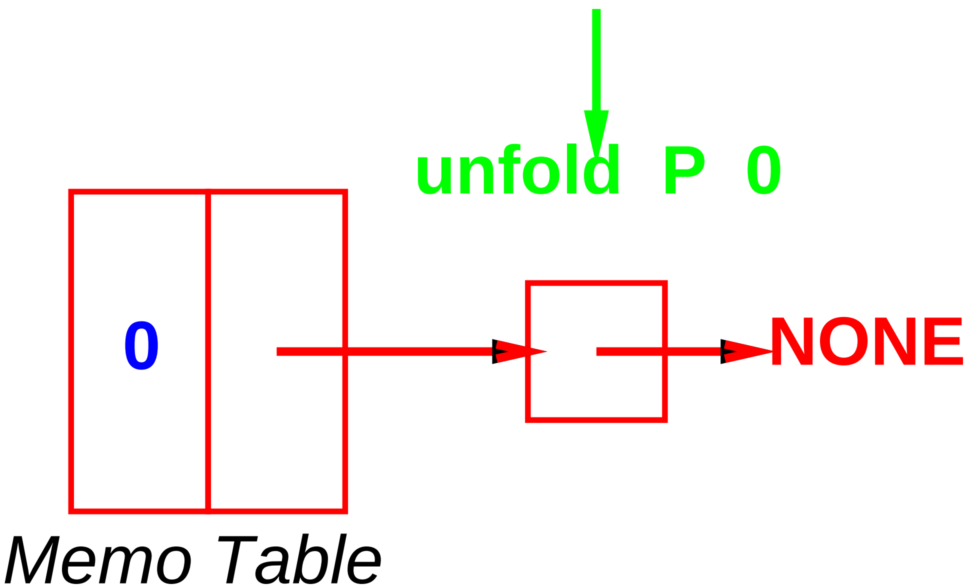
initial seed : `0`

unfold P 0

Unfold Implementation: Standard ML

generating fcn.: **fun** `P n = (n,[(n+1) mod 2])`

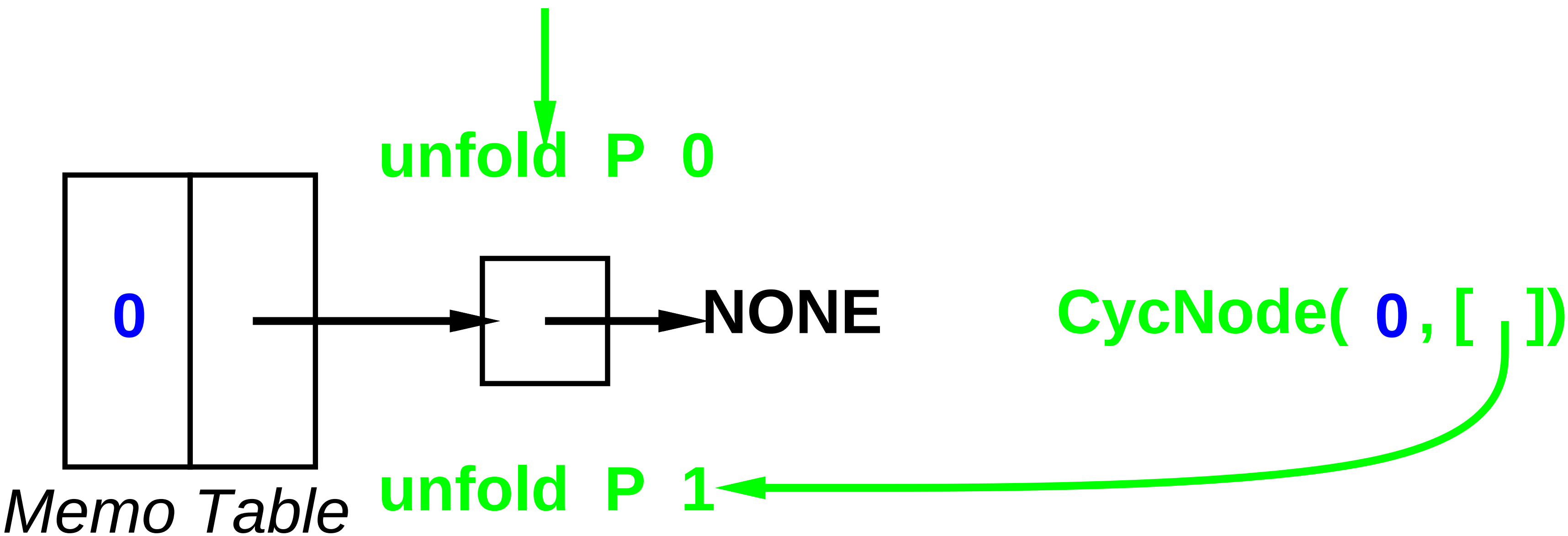
initial seed : `0`



Unfold Implementation: Standard ML

generating fcn.: **fun** `P n = (n,[(n+1) mod 2])`

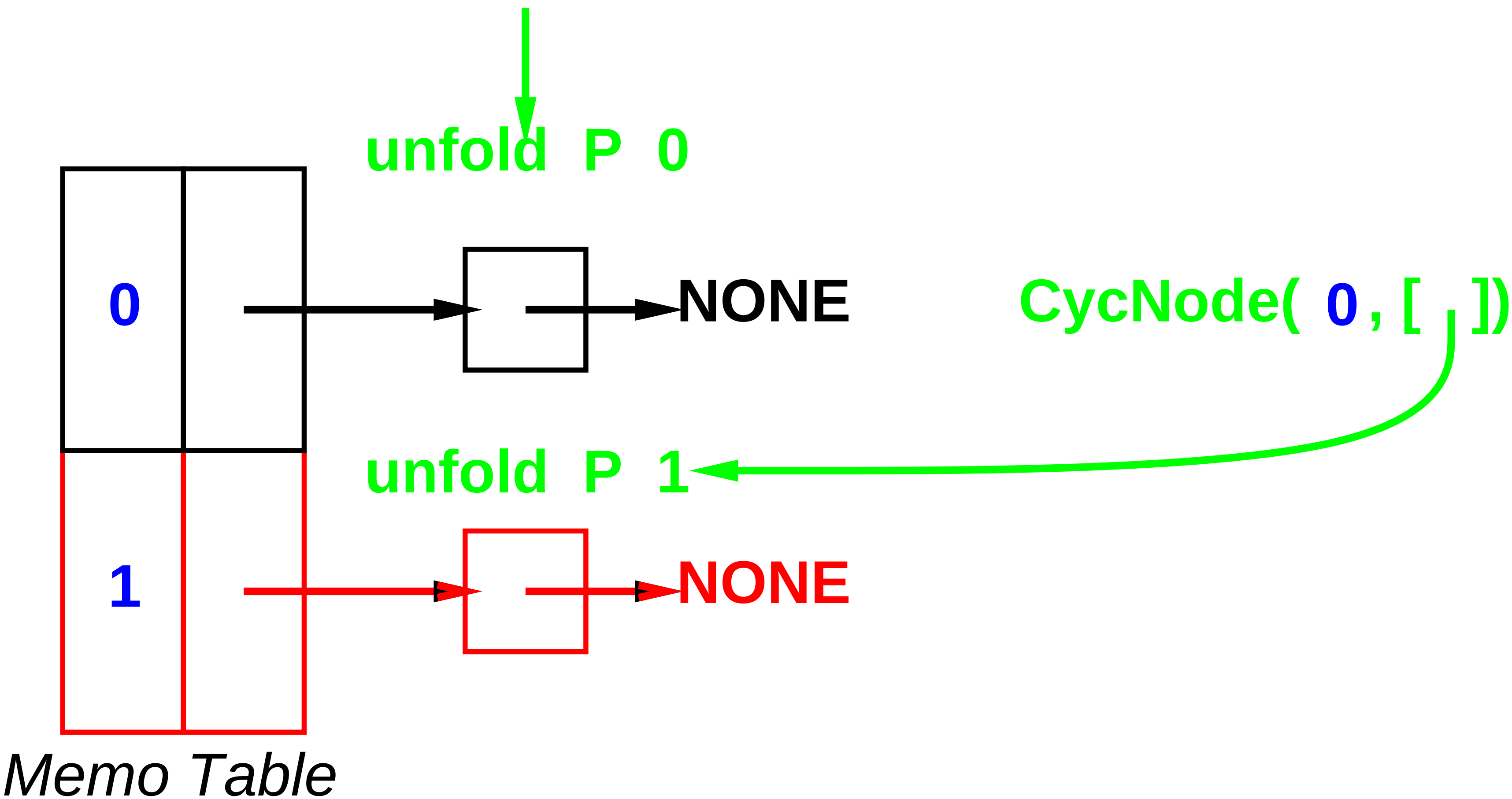
initial seed : `0`



Unfold Implementation: Standard ML

generating fcn.: **fun** `P n = (n,[(n+1) mod 2])`

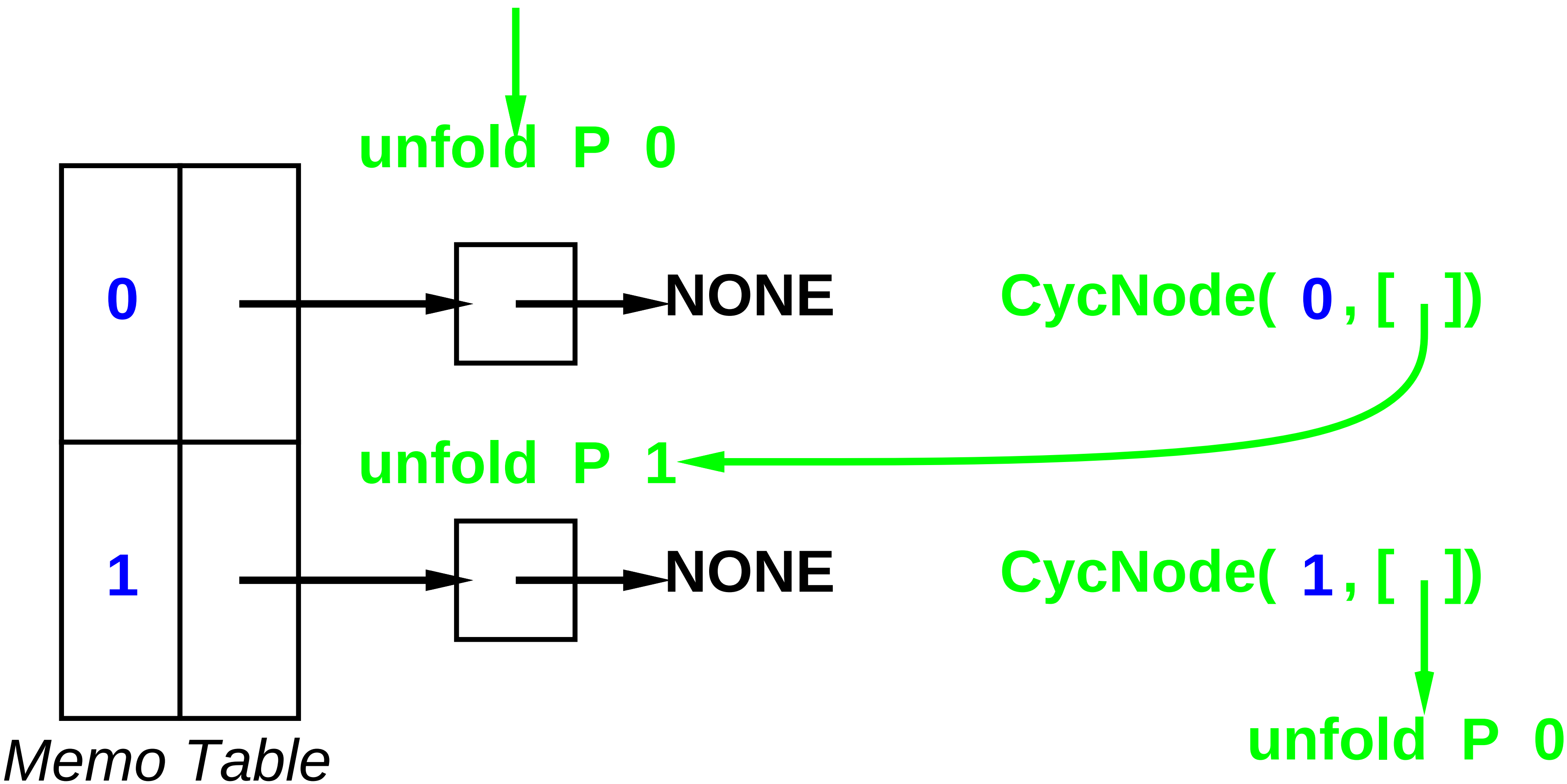
initial seed : `0`



Unfold Implementation: Standard ML

generating fcn.: **fun** `P n = (n,[(n+1) mod 2])`

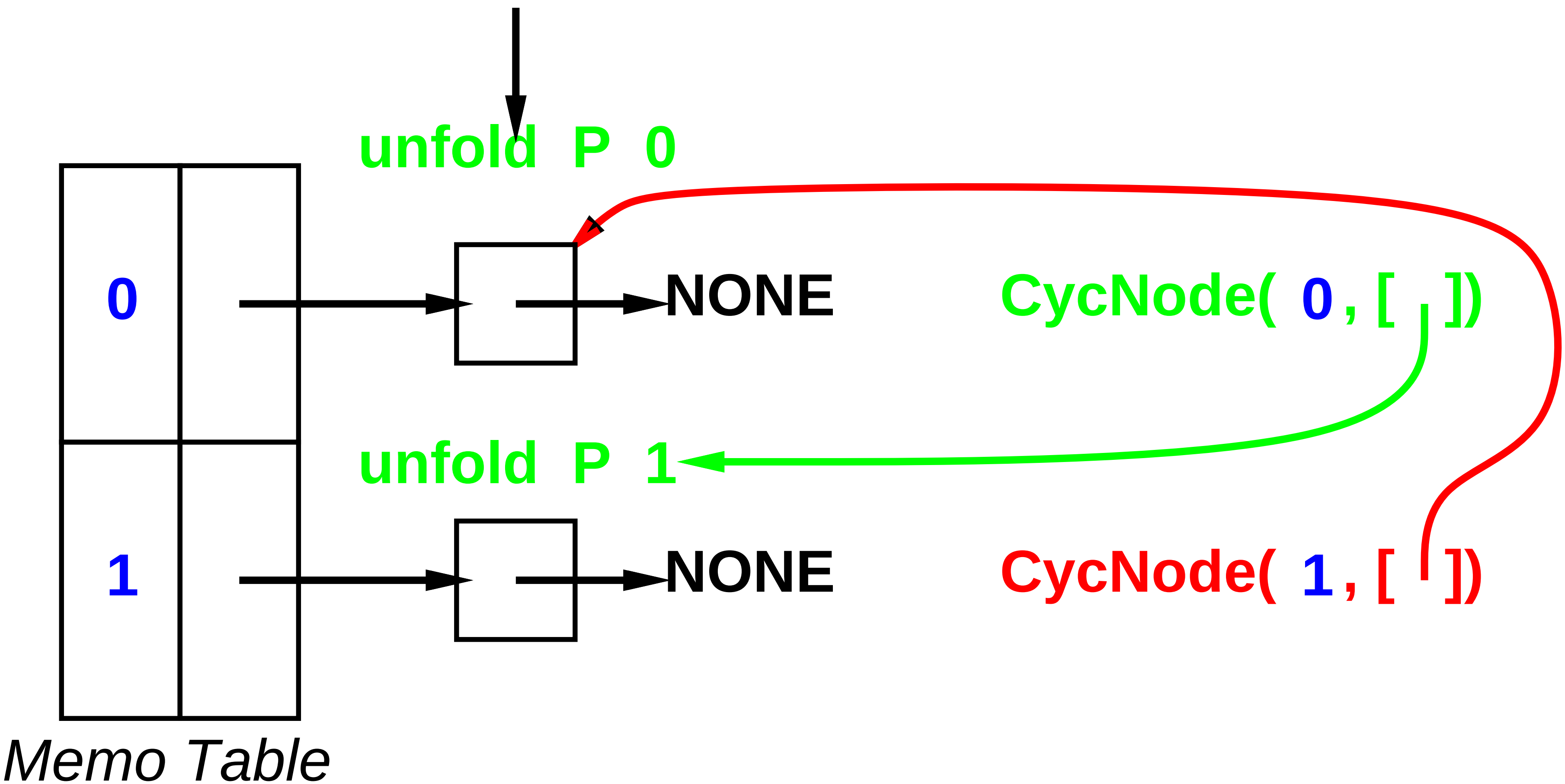
initial seed : `0`



Unfold Implementation: Standard ML

generating fcn.: **fun** `P n = (n,[(n+1) mod 2])`

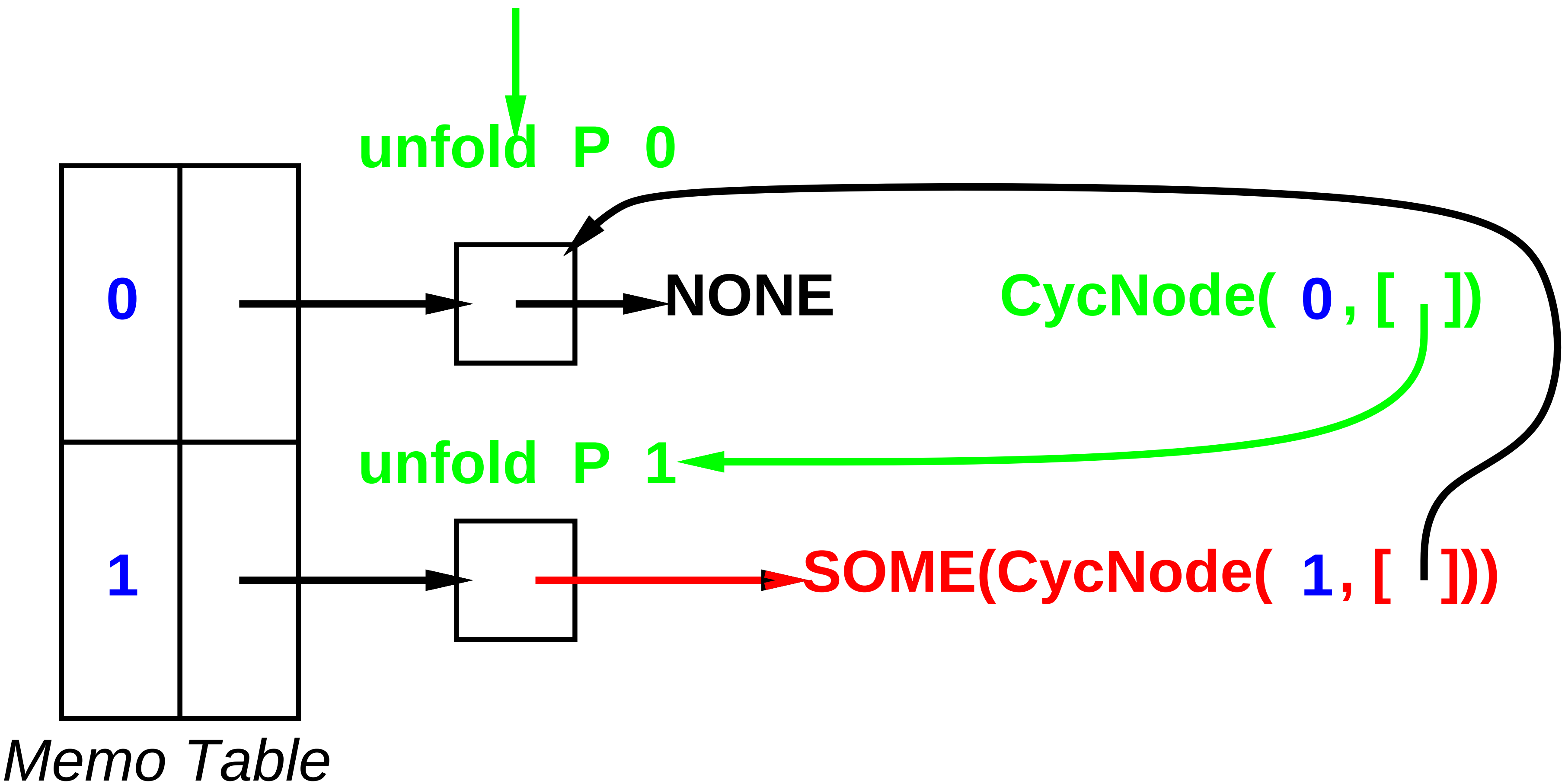
initial seed : `0`



Unfold Implementation: Standard ML

generating fcn.: `fun P n = (n,[(n+1) mod 2])`

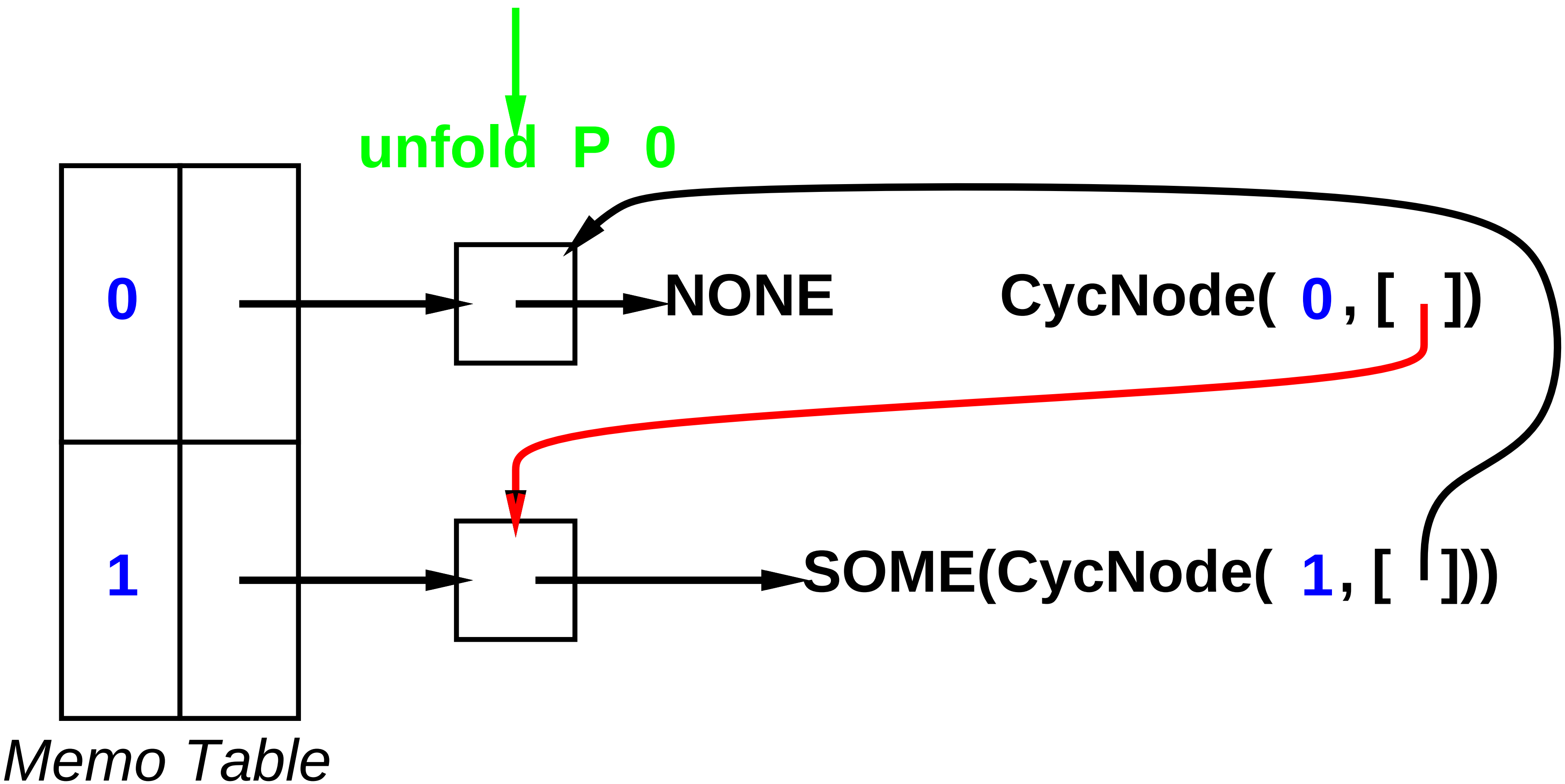
initial seed : `0`



Unfold Implementation: Standard ML

generating fcn.: `fun P n = (n,[(n+1) mod 2])`

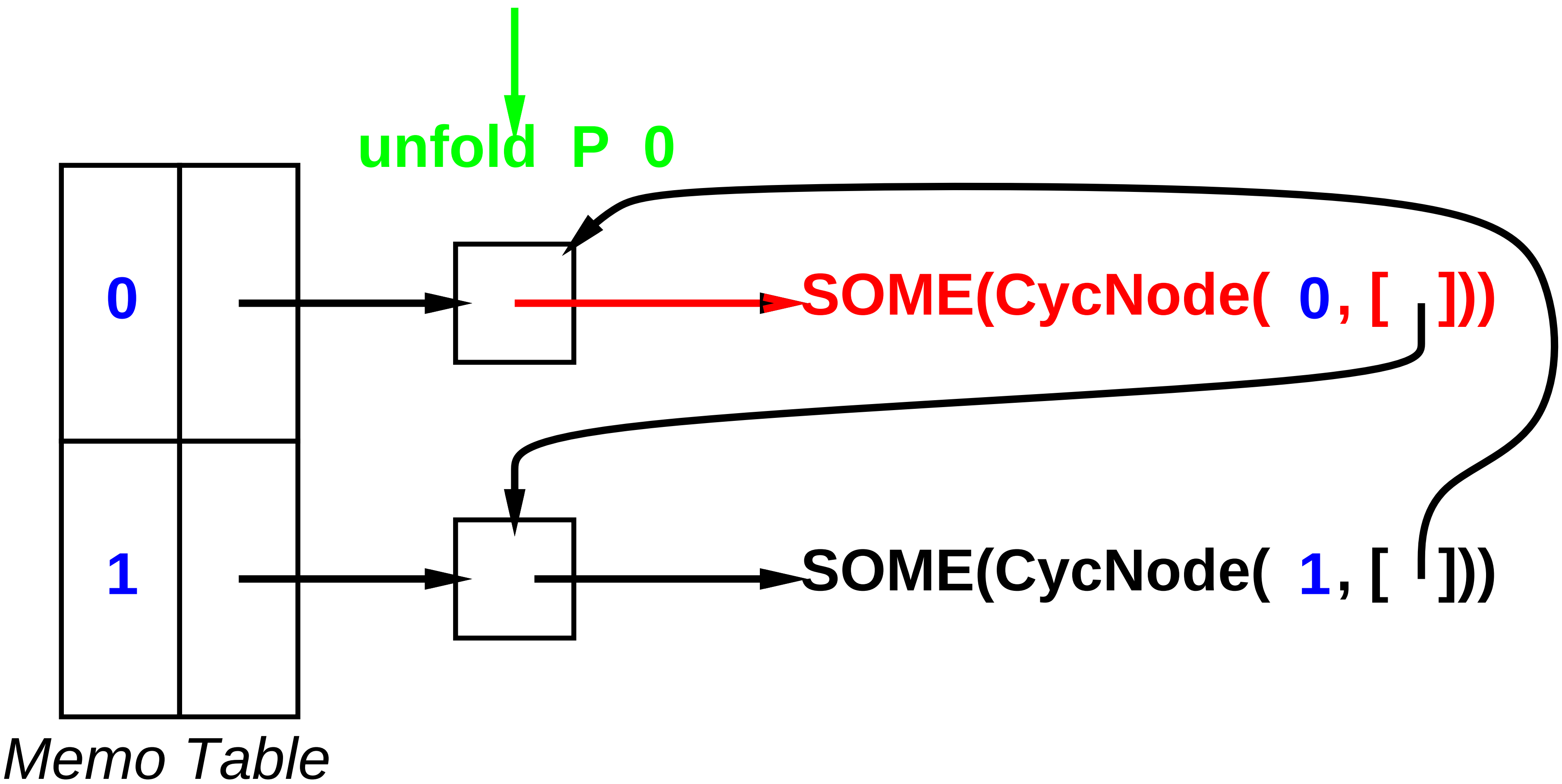
initial seed : `0`



Unfold Implementation: Standard ML

generating fcn.: **fun** `P n = (n,[(n+1) mod 2])`

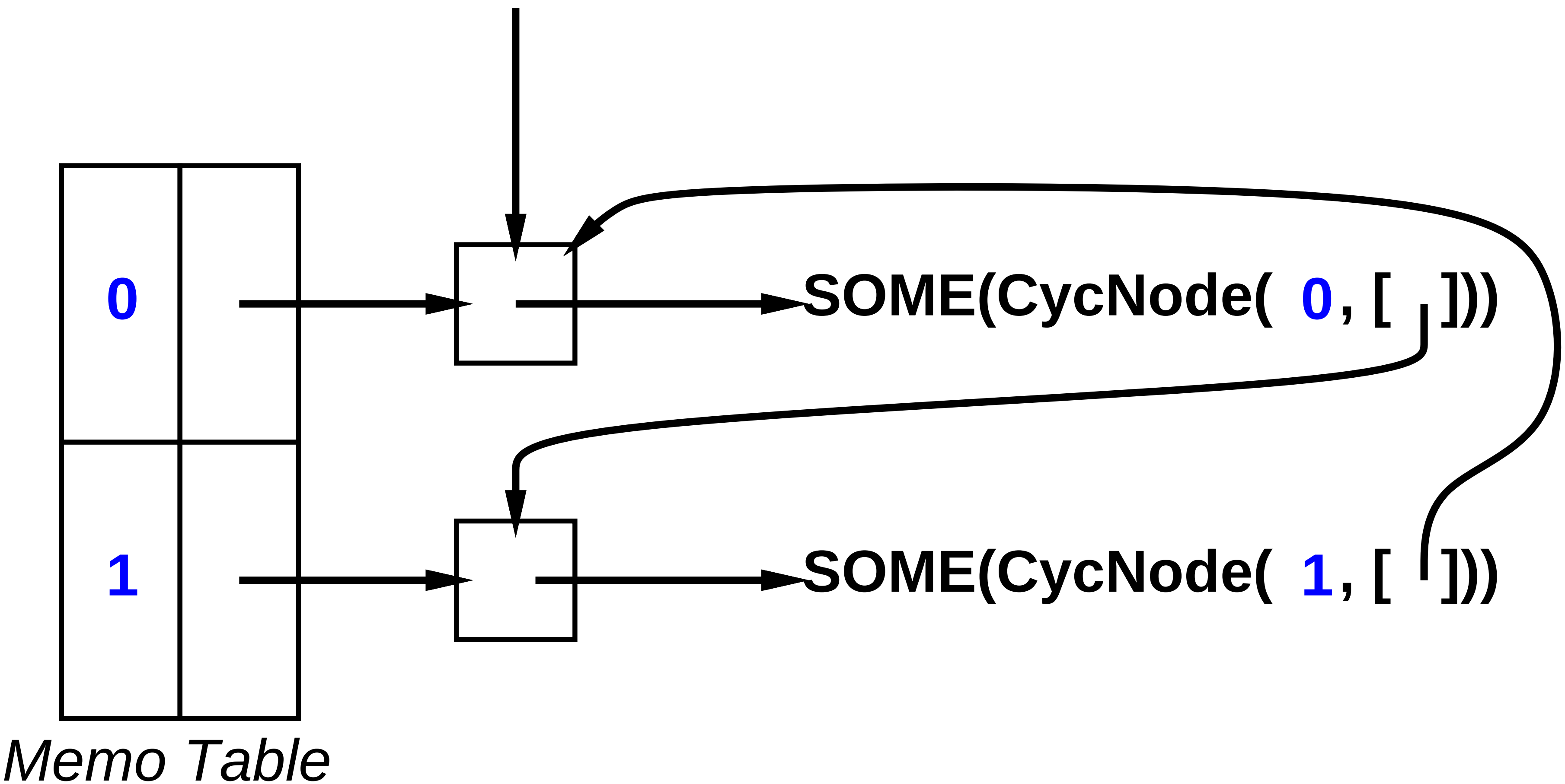
initial seed : `0`



Unfold Implementation: Standard ML

generating fcn.: **fun** `P n = (n,[(n+1) mod 2])`

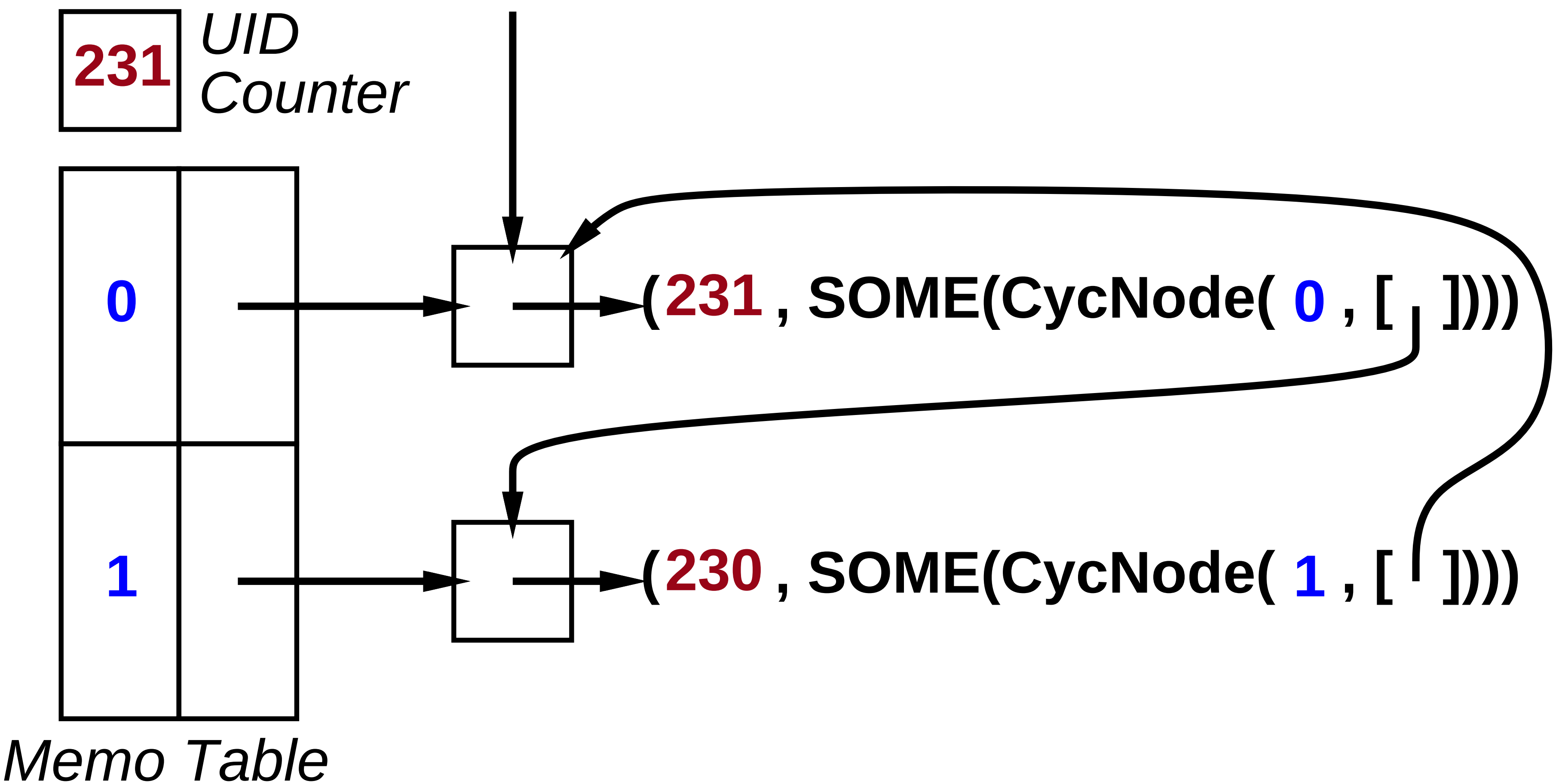
initial seed : `0`



Unfold Implementation: Standard ML

generating fcn.: **fun** `P n = (n,[(n+1) mod 2])`

initial seed : `0`



Unfold Implementation: Discussion

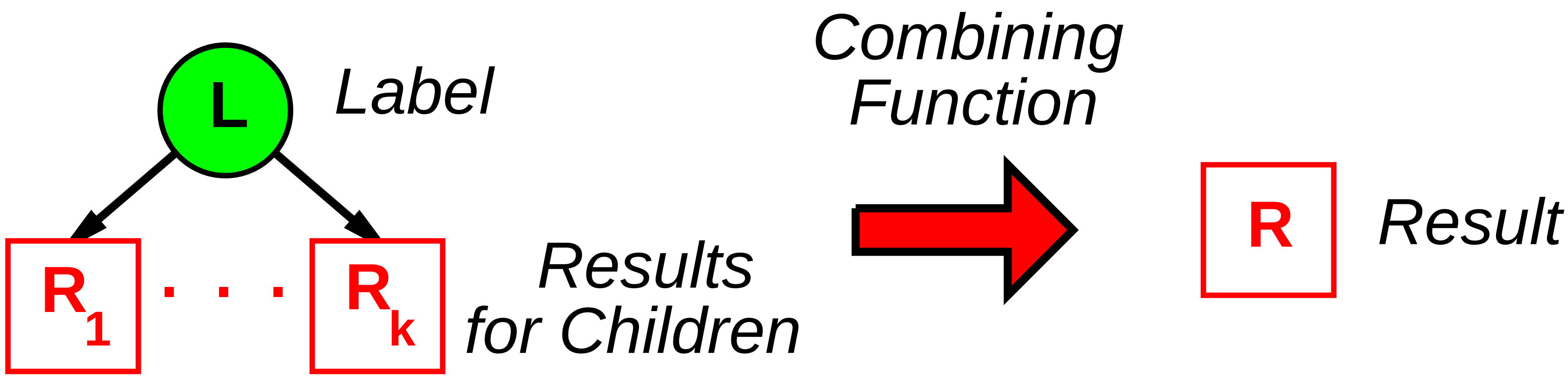
- Can use fewer reference cells in SML implementation.
- Cyclic hash-consing yields minimal graphs (Mauborgne, ESOP 2000; Considine & Wells, unpublished).
- Haskell implementation:
  - Uses laziness to tie cyclic knots.
  - Uses a `Cycle` monad to thread UID counter and memoization tables through computation.
  - Tricky to tie cyclic knots in presence of monad; use techniques of Erkok and Launchbury (ICFP '00).
- In practice, a memofix function is more flexible than unfold (see paper).

Road Map

- Viewing cyclic structures as infinite regular trees.
- Adapting the tree-generating unfold function to generate cyclic structures for infinite regular trees.
- Adapting the tree-accumulating fold function to return non-trivial results for strict combining functions and infinite regular trees.
- Cycamores: an abstraction for manipulating regular trees that we have implemented in ML and Haskell.

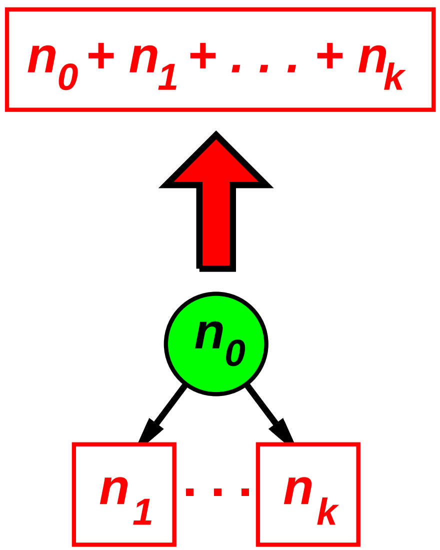
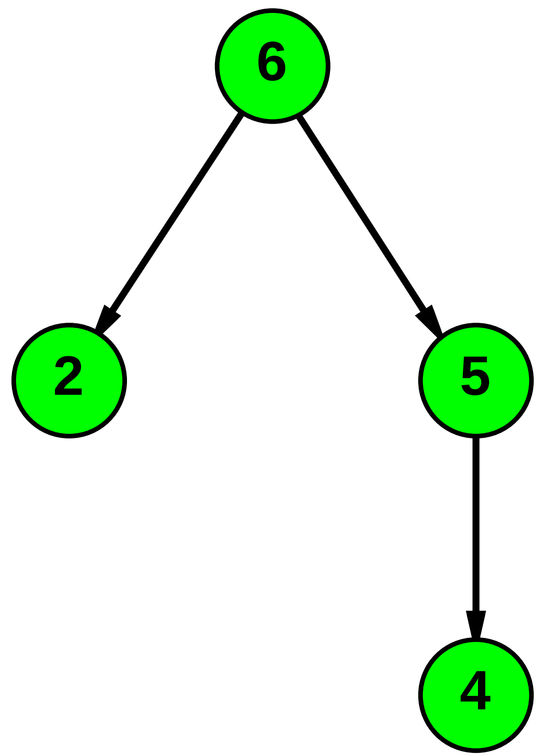
Tree Accumulation via Fold

The fold operator accumulates a result from a tree using a combining function.

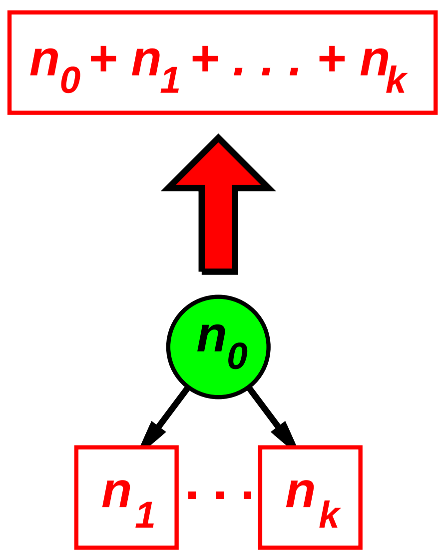
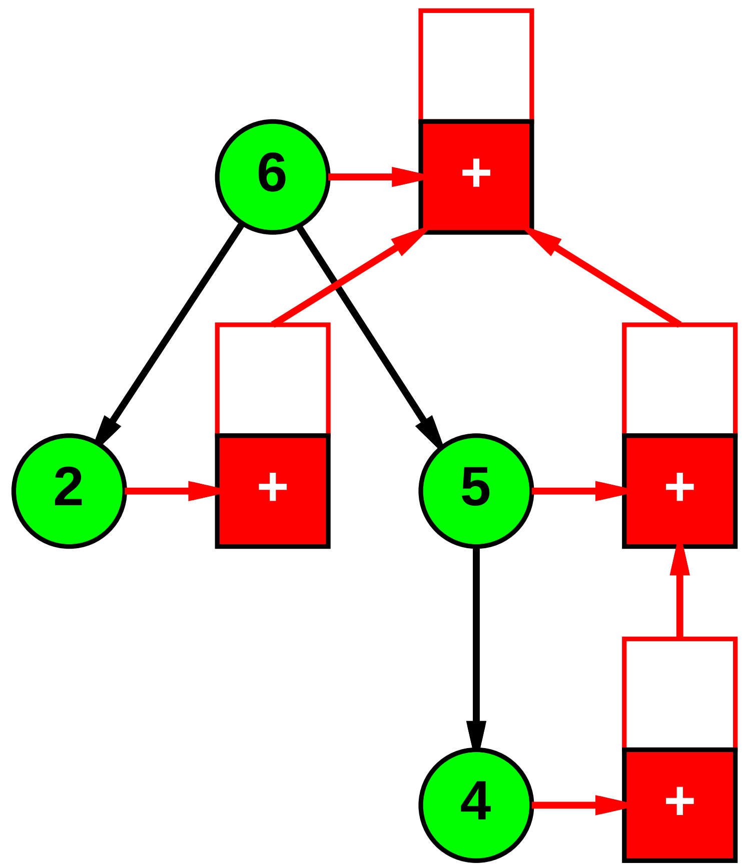


$$\text{fold} : \underbrace{((L \times (\mathcal{C}_{\text{res}}{}^{\omega})) \xrightarrow{\text{cont}} \mathcal{C}_{\text{res}})}_{\text{accumulating function } \phi} \longrightarrow \underbrace{\text{Tree}(L) \xrightarrow{\text{cont}} \mathcal{C}_{\text{res}}}_{\substack{\text{tree valuation } \theta \\ (\phi\text{-catamorphism})}}$$

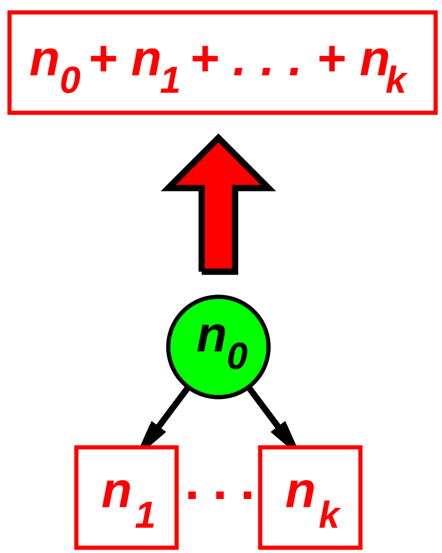
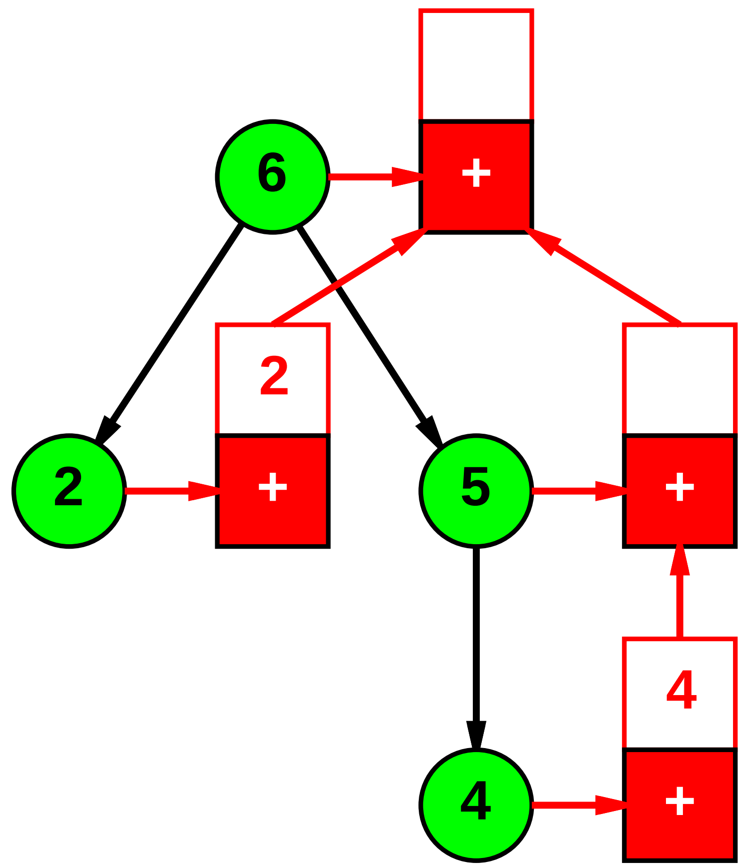
Folding over a Finite Tree

Accumulating Function ϕ	*Tree*
	

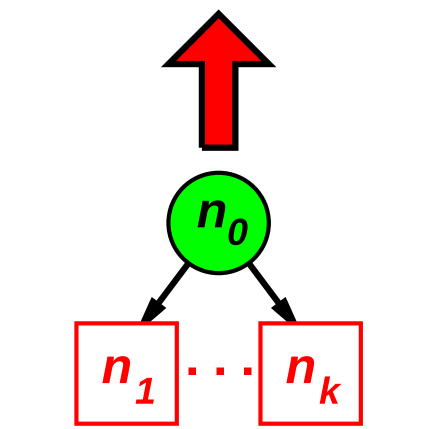
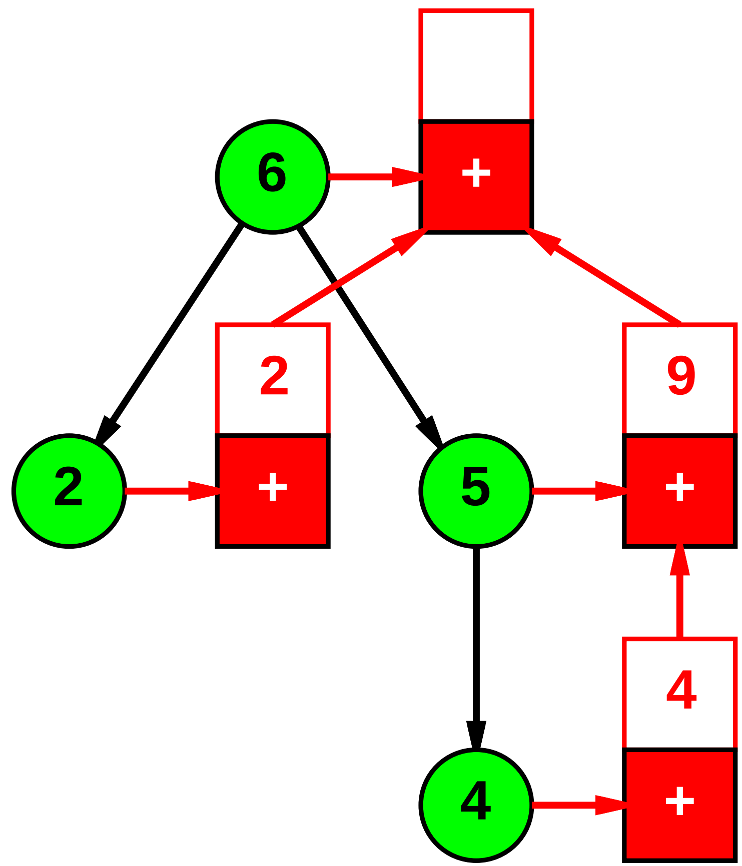
Folding over a Finite Tree

Accumulating Function ϕ	*Tree*
	

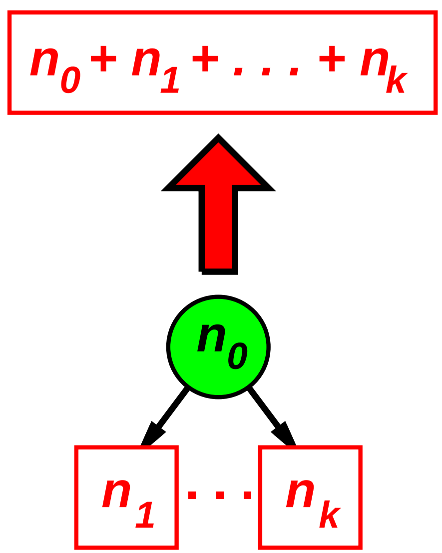
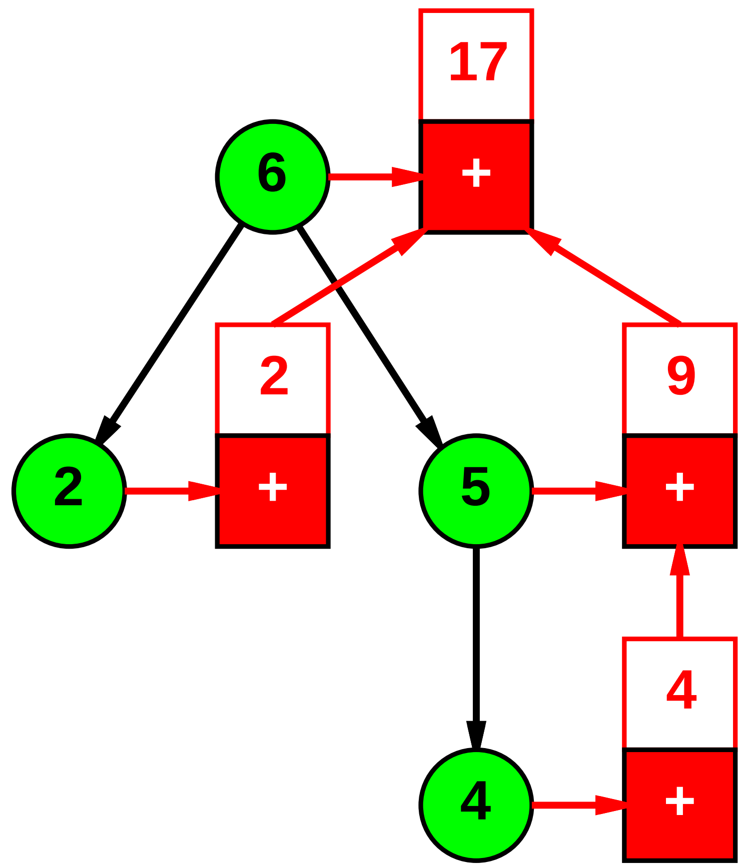
Folding over a Finite Tree

Accumulating Function ϕ	*Tree*
	

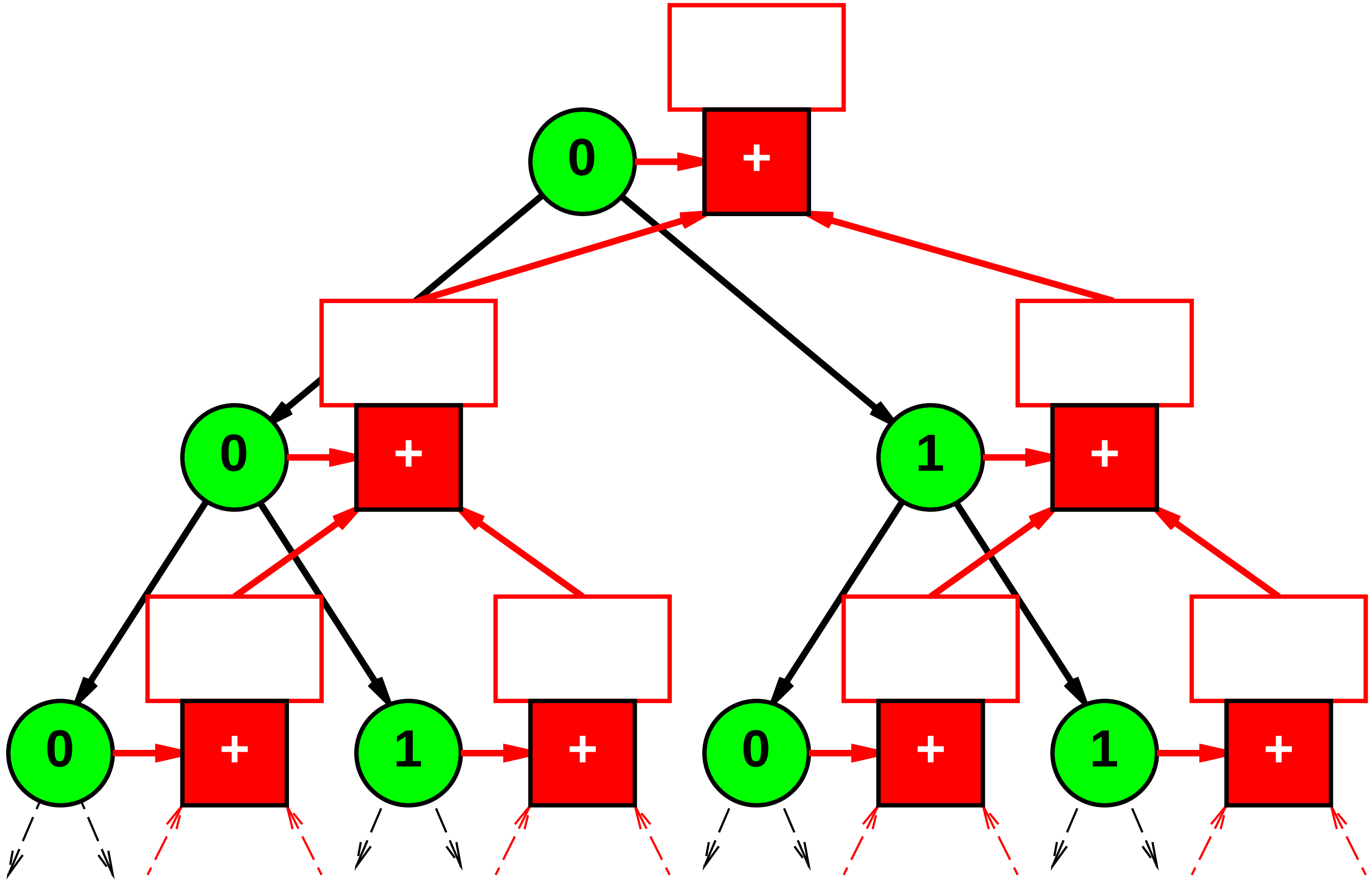
Folding over a Finite Tree

Accumulating Function ϕ	*Tree*
	

Folding over a Finite Tree

Accumulating Function ϕ	*Tree*
	

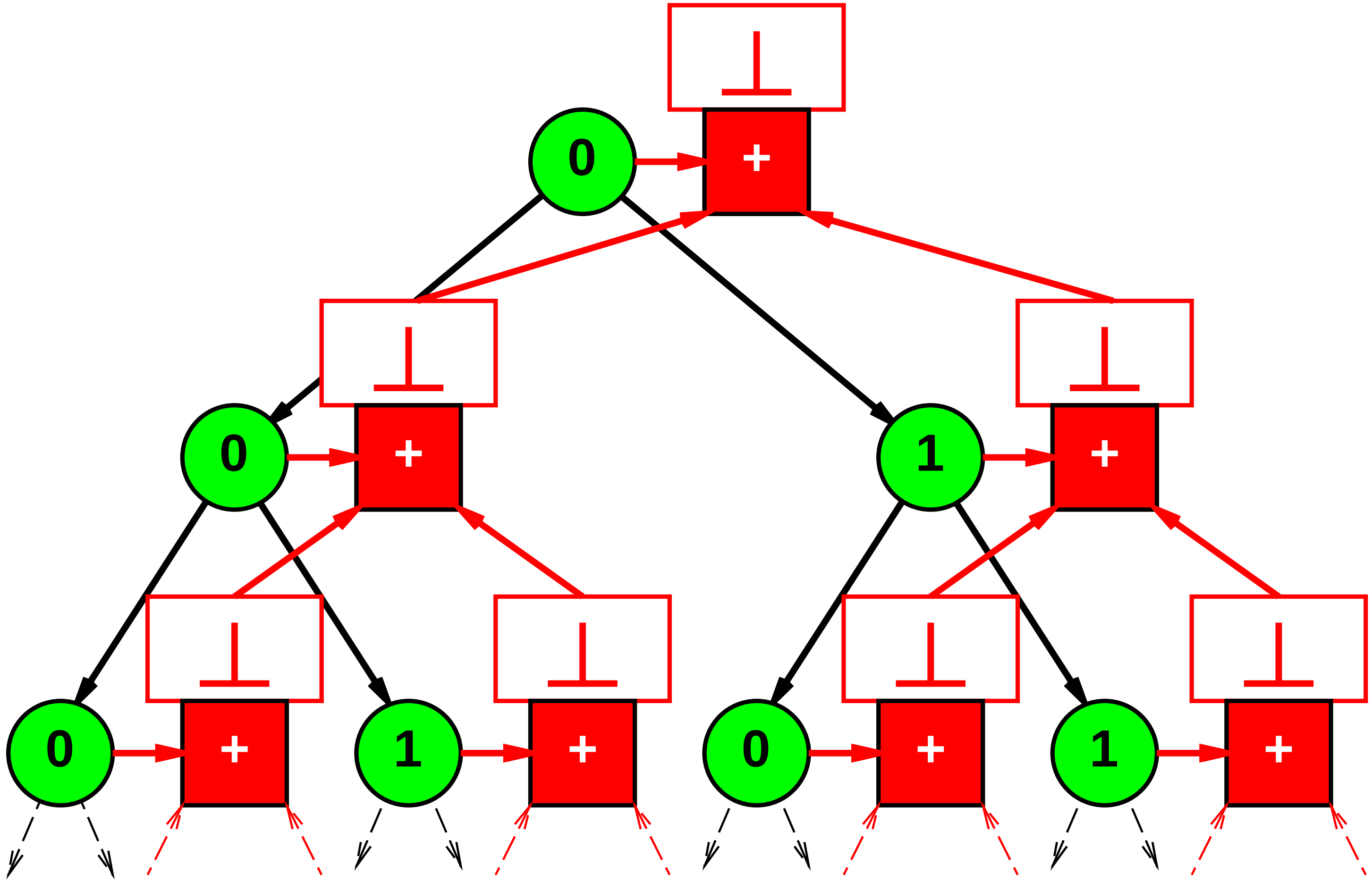
Folding over an Infinite Regular Tree



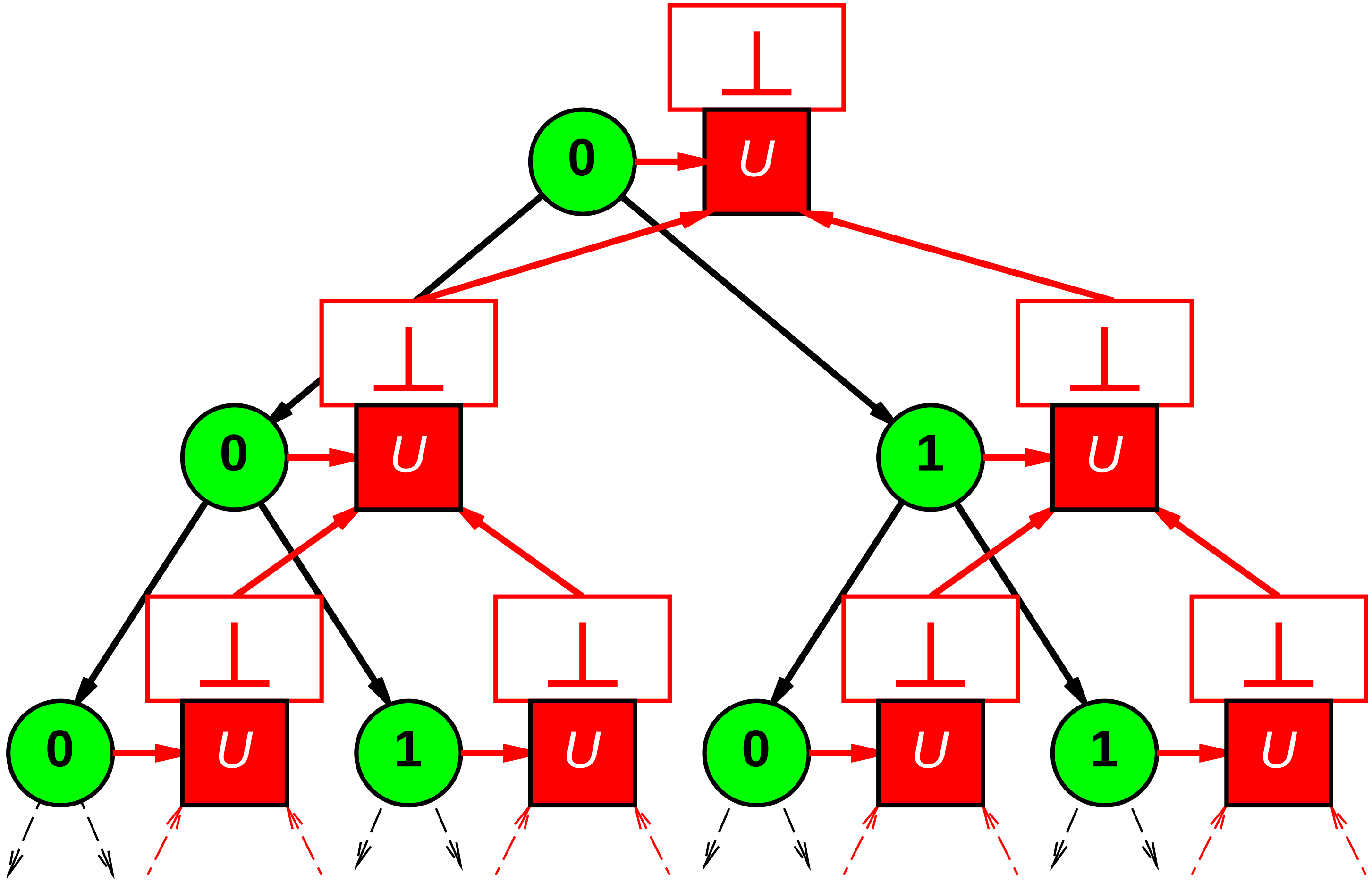
Expect θ to be ϕ-*consistent*: for each subtree t of a given tree,

$$\theta(t) = \phi(\mathsf{label}(t), \mathsf{map}(\theta)(\mathsf{children}(t))).$$

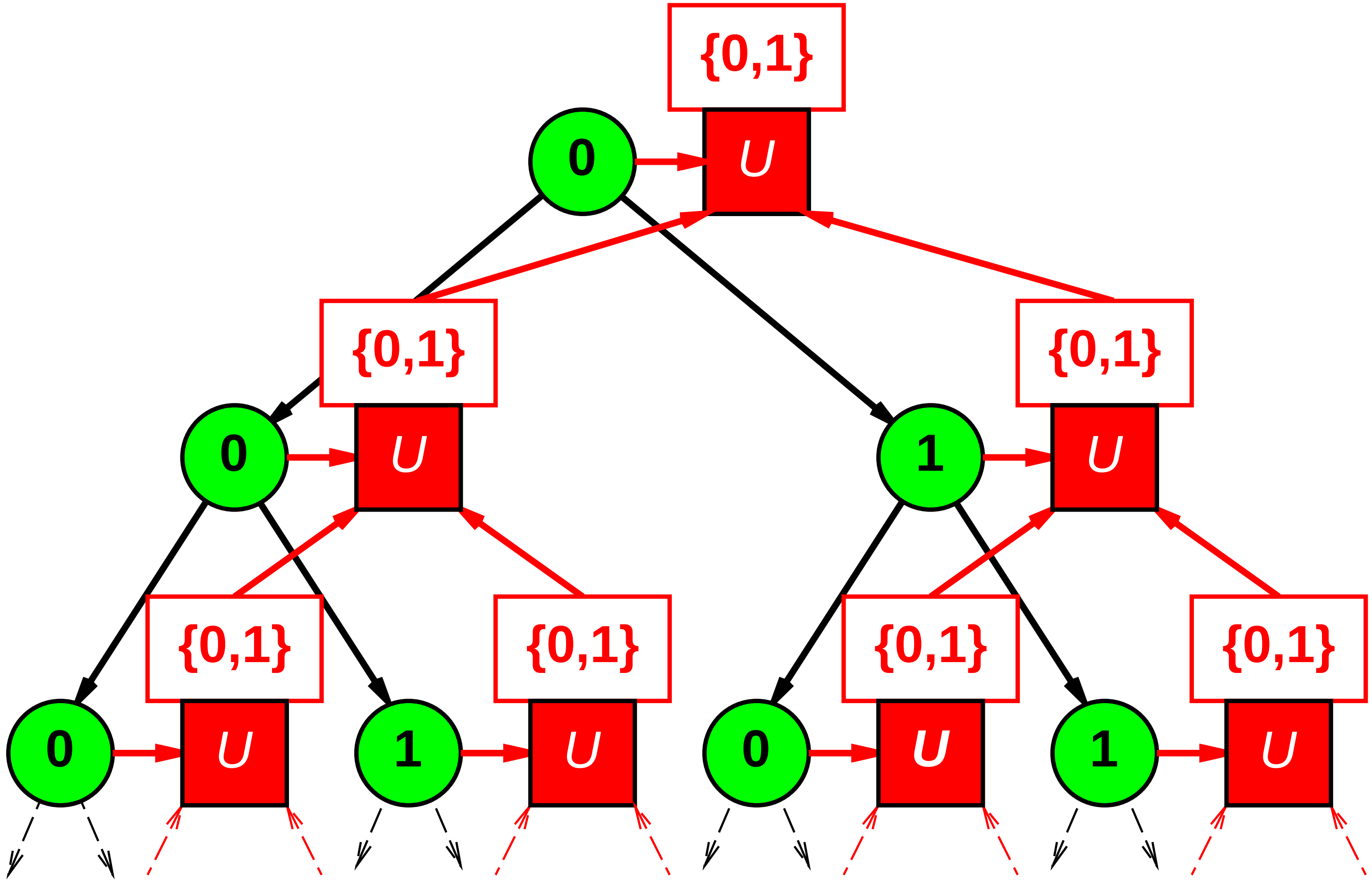
Folding over an Infinite Regular Tree



Folding over an Infinite Regular Tree

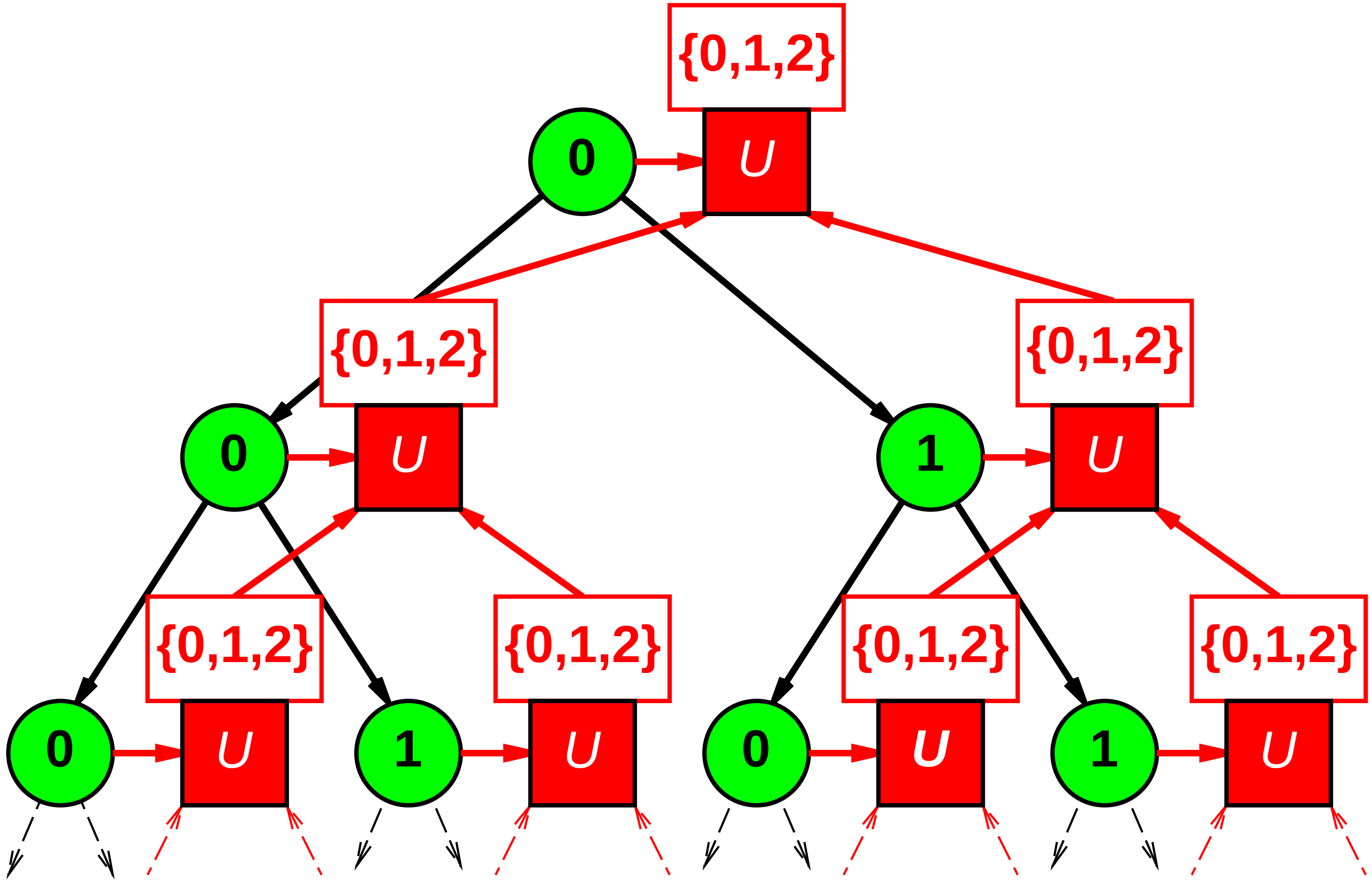


Folding over an Infinite Regular Tree



This fold may be desirable, but it is not the computed one.

Folding over an Infinite Regular Tree



This fold is not computed either.

Digression: Fixed Points

A value x is a *fixed point* of a function f if $f(x) = x$.

What are the fixed points of the following:

f_i :: Int -> Int
f_1(x) = x/2 + 3
f_2(x) = x^2
f_3(x) = x
f_4(x) = x -1

Can also have fixed points over functions manipulating data structures and other functions:

```
g :: [Int] -> [Int]
g(x) = 0:1:x

h :: (Int -> Int) -> (Int -> Int)
h(k) = \n -> if n == 0 then 1 else n*(k(n-1))
```

Digression: Least Fixed Points

Under certain conditions, functions over data structures and functions have a so-called *least fixed point*. In particular, the function must be a *continuous function* between two *pointed complete partial orders*.

- Intuitively, a pointed complete partial order is a lattice rooted at $\bot$ where elements are arranged by information content and every chain has a limit.
- Intuitively, the least fixed point of a function f is found by starting at $\bot$ and applying f until a limit is reached.
- For strict f, the least fixed point will always be $\bot$.

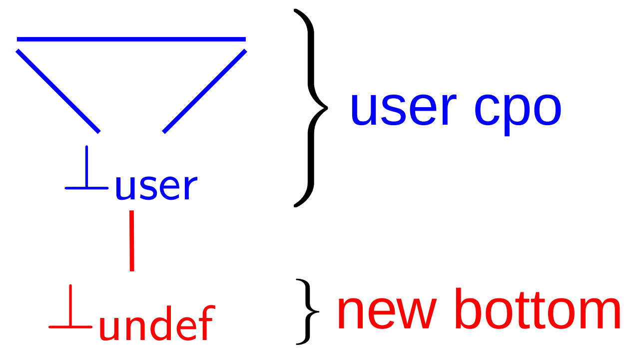
Cycfold: Goals

Given a strict combining function ϕ, want $\mathsf{cycfold}(\phi)$ that:

- Coincides with $\mathsf{fold}(\phi)$ on finite trees;
- Can return a non-trivial result for regular trees;
- Diverges on non-regular trees.

Cycfold: The Idea

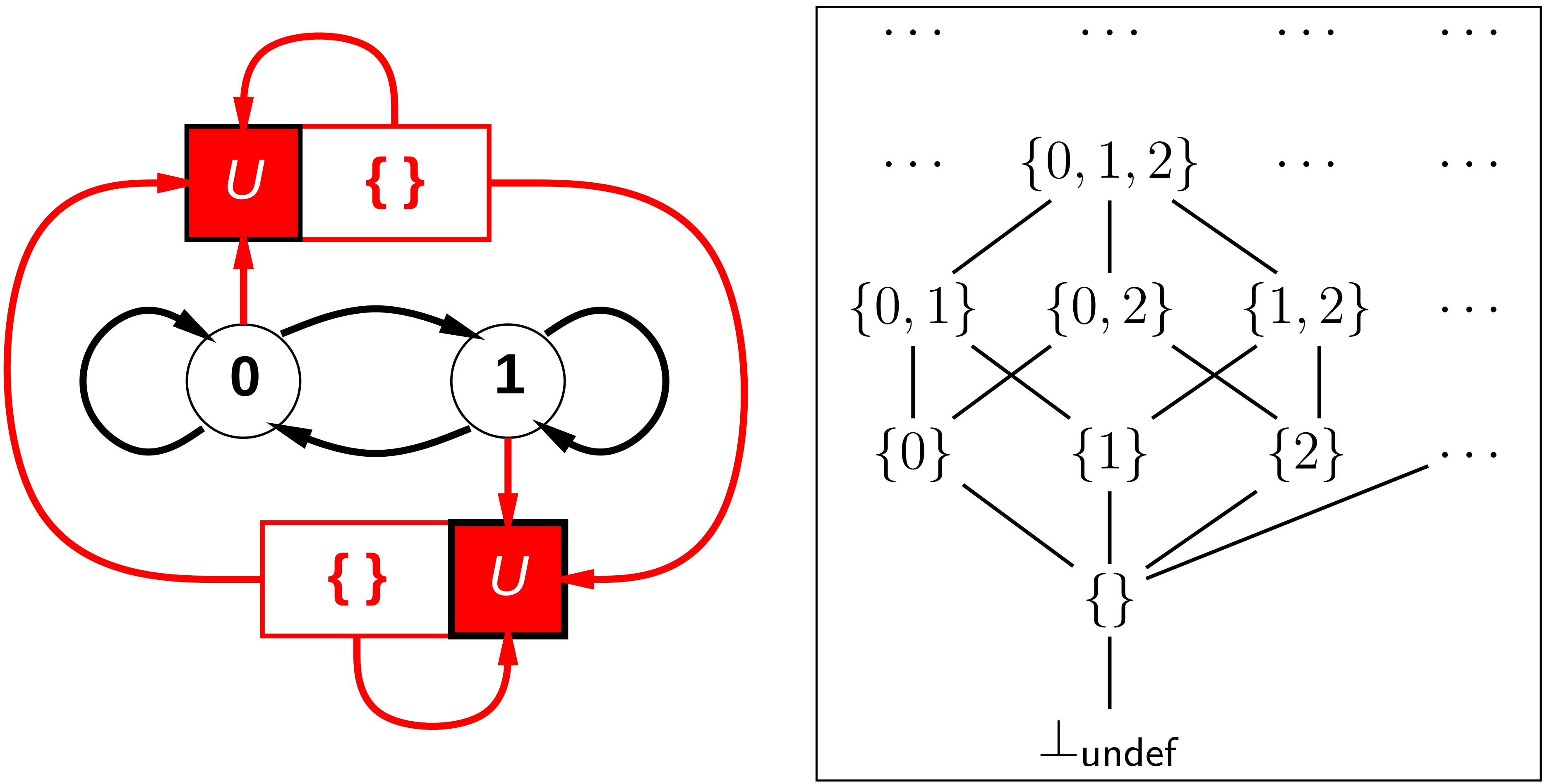
Use a result domain $\mathcal{C}_{\text{res}}$ that is a *lifted* pointed cpo (i.e., doubly pointed) and require the combining function ϕ to be strict and monotone.



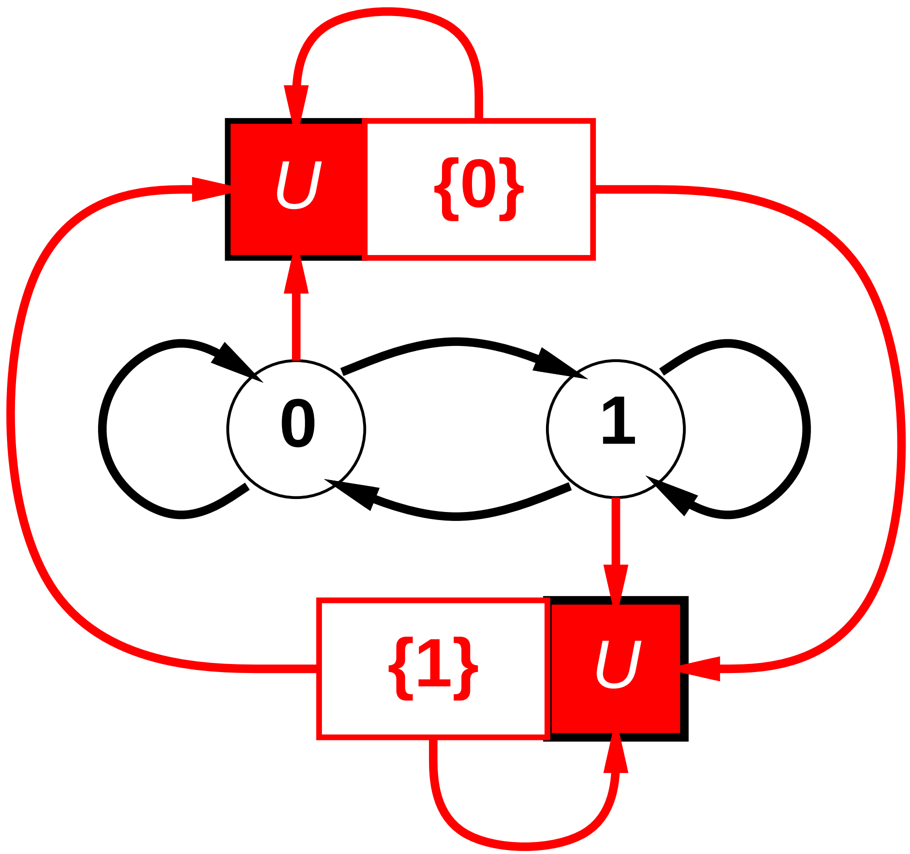
For a given tree t, calculate $\text{cycfold}(\phi)(t)$ as follows:

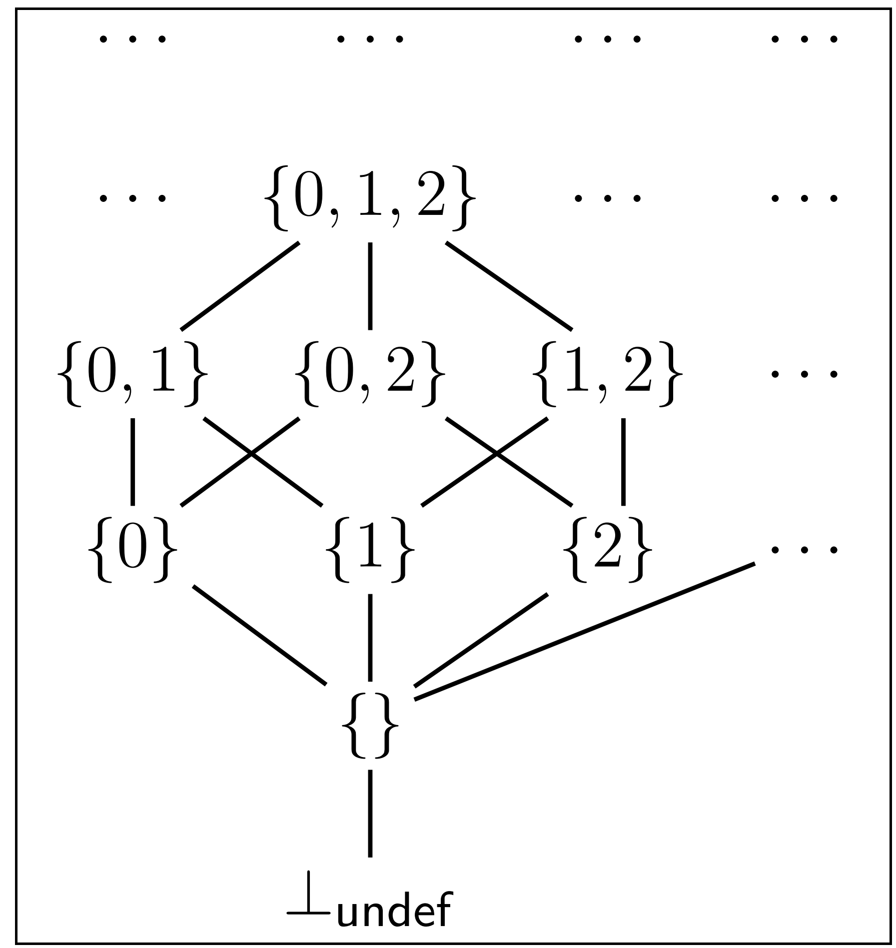
- If t not regular, return $\perp_{\text{undef}}$.
- Otherwise:
  - Let tree valuation θ_0 map all subtrees of t to $\perp_{\text{user}}$.
  - Iteratively calculate θ_{i+1} from θ_i using ϕ.
  - If $\theta_{k+1} = \theta_k$ then return $\theta_k(t)$ else return $\perp_{\text{undef}}$.

Cycfold Example 1: Node Labels

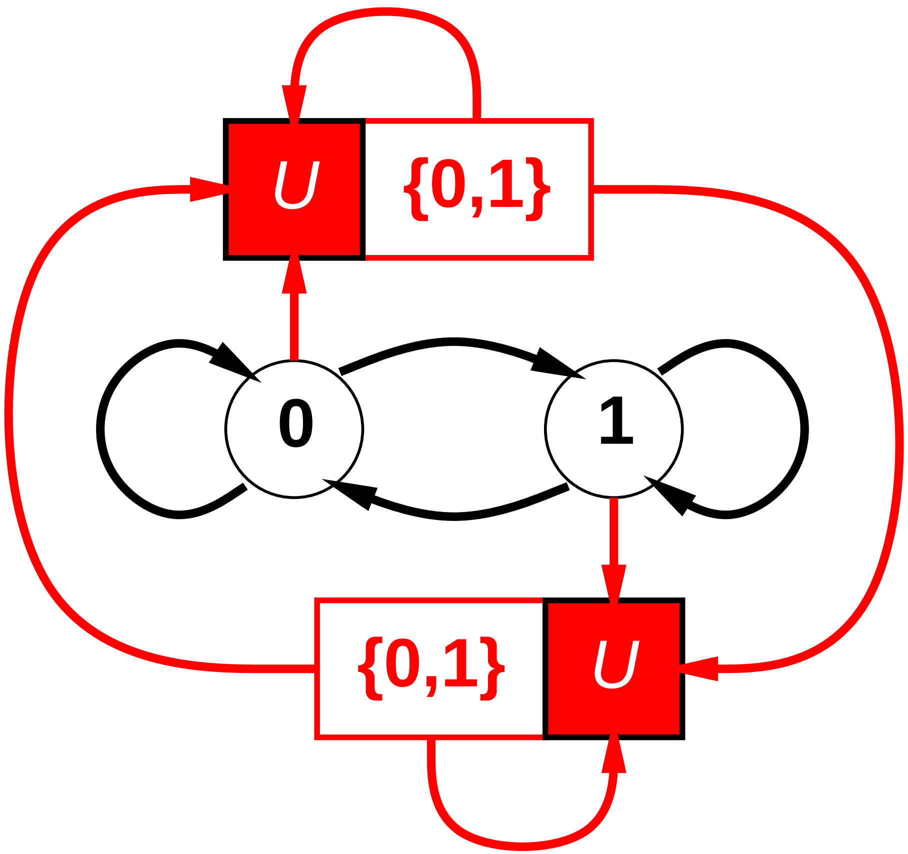


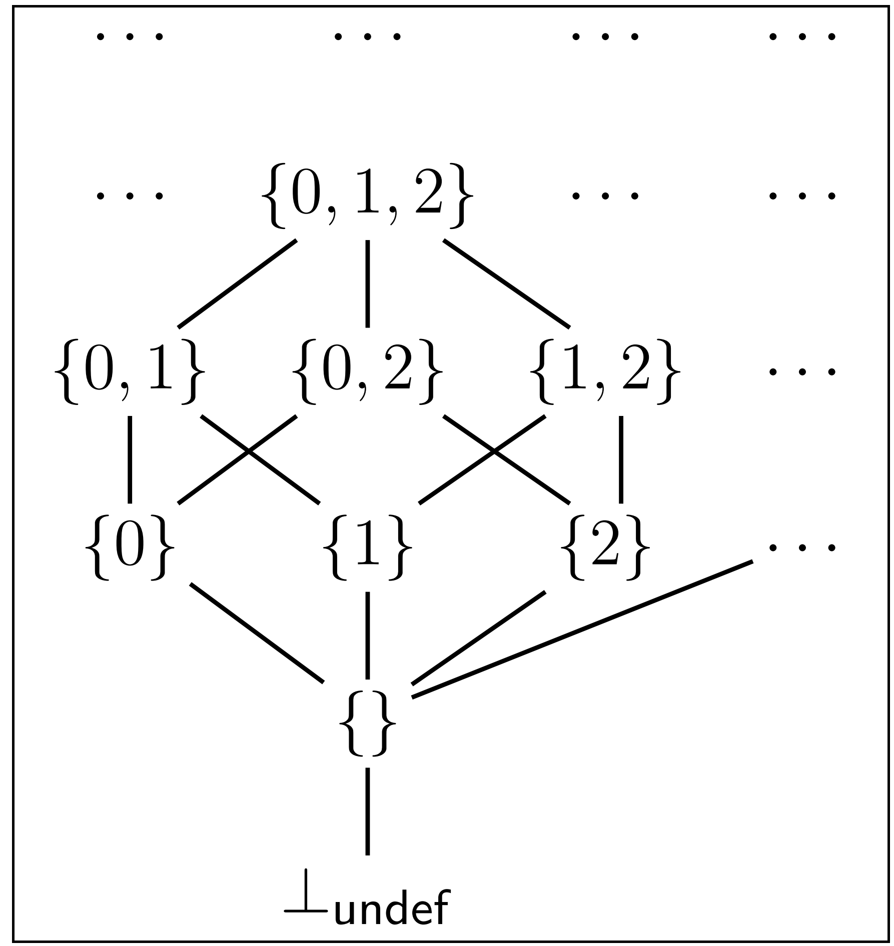
Cycfold Example 1: Node Labels



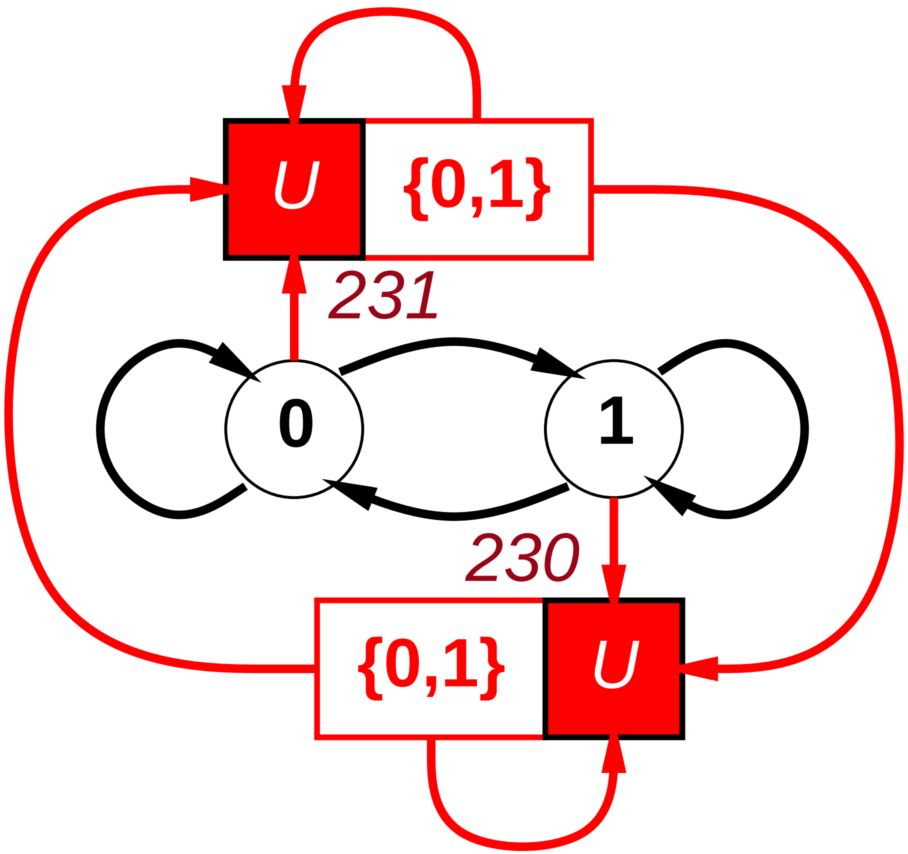


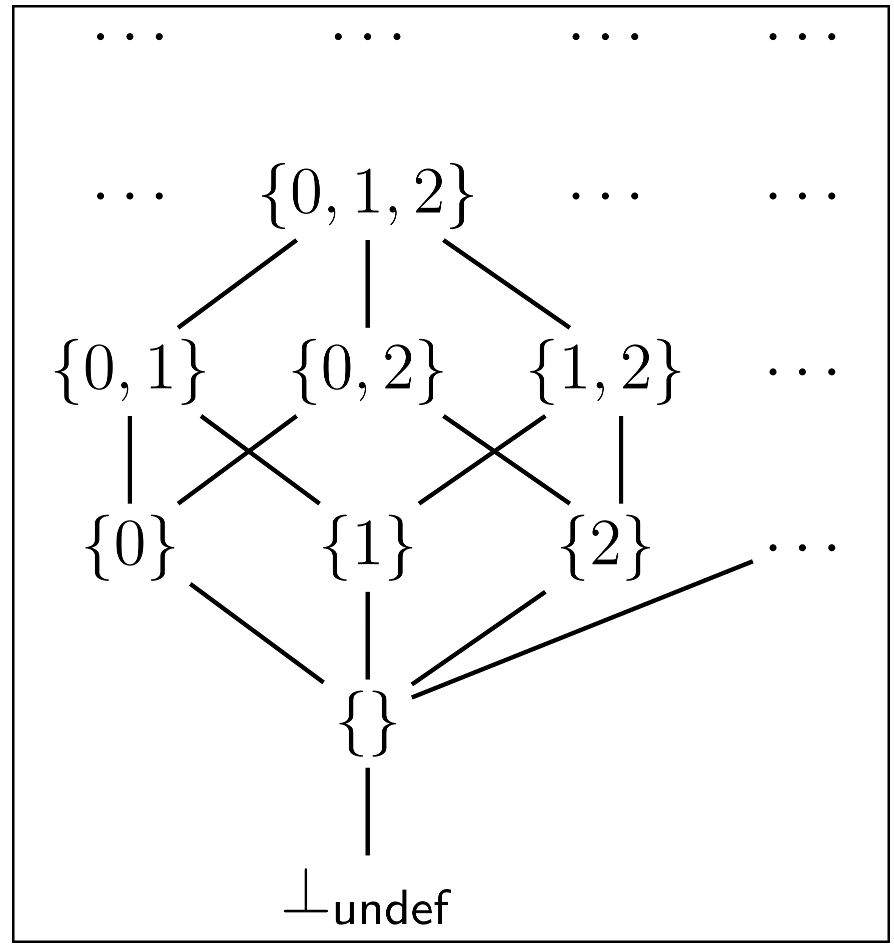
Cycfold Example 1: Node Labels





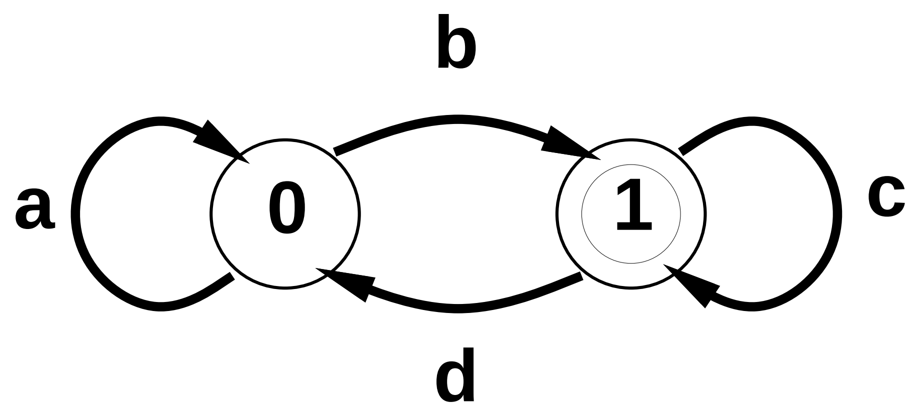
Cycfold Example 1: Node Labels

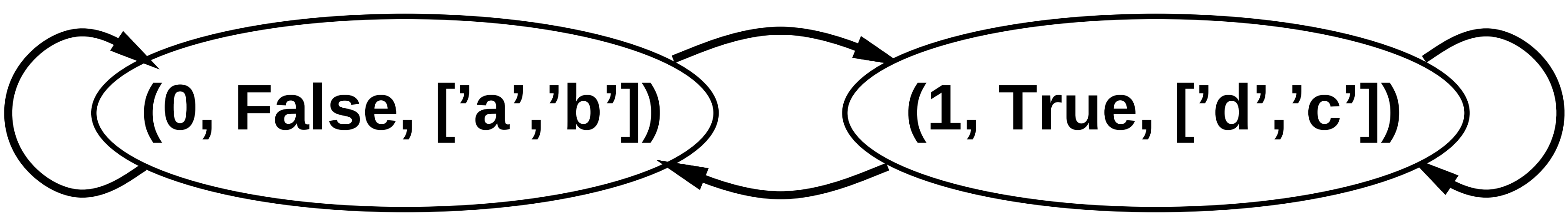




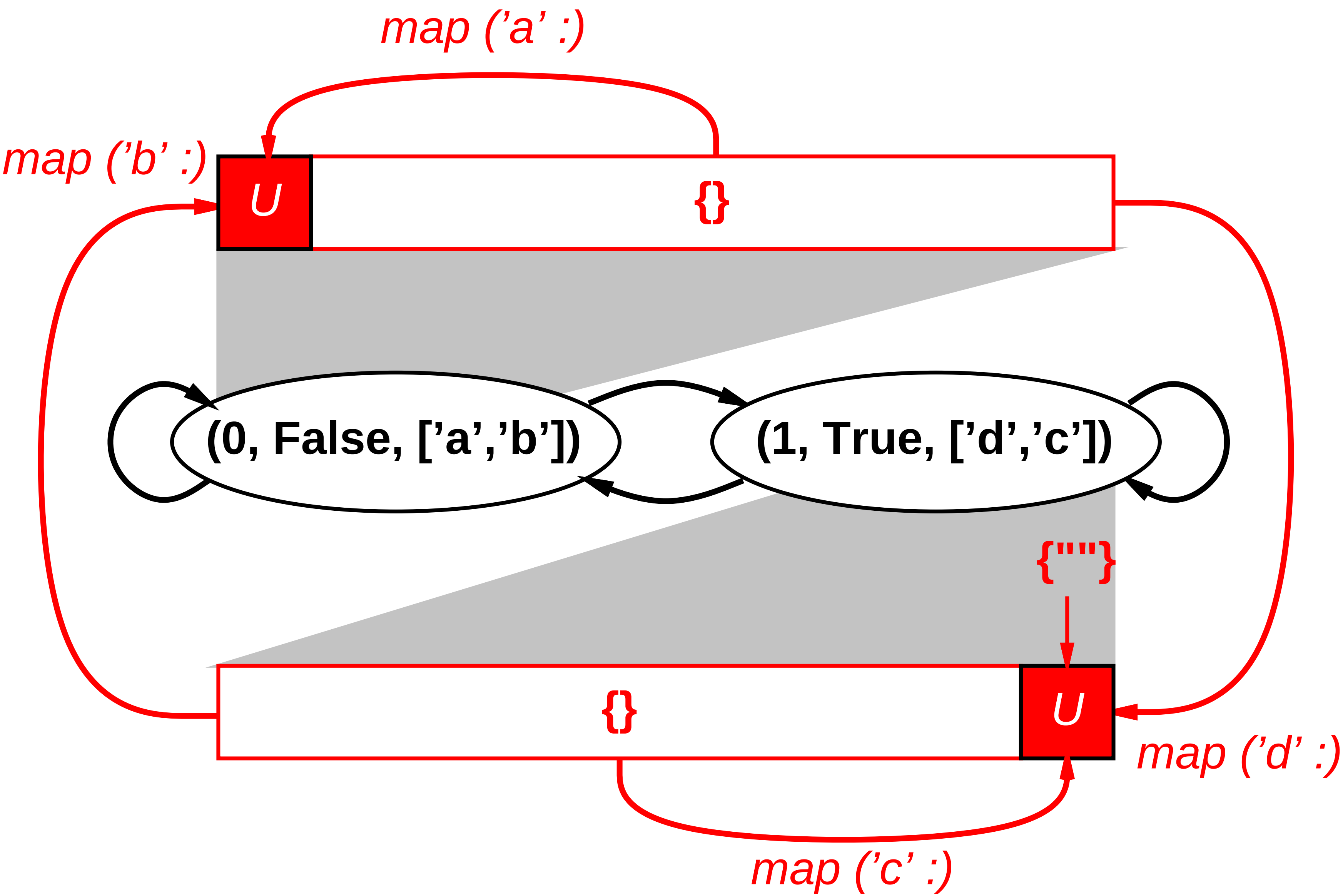
Cycfold Example 2: DFA Strings

Node labels can encode other aspects of cyclic data.

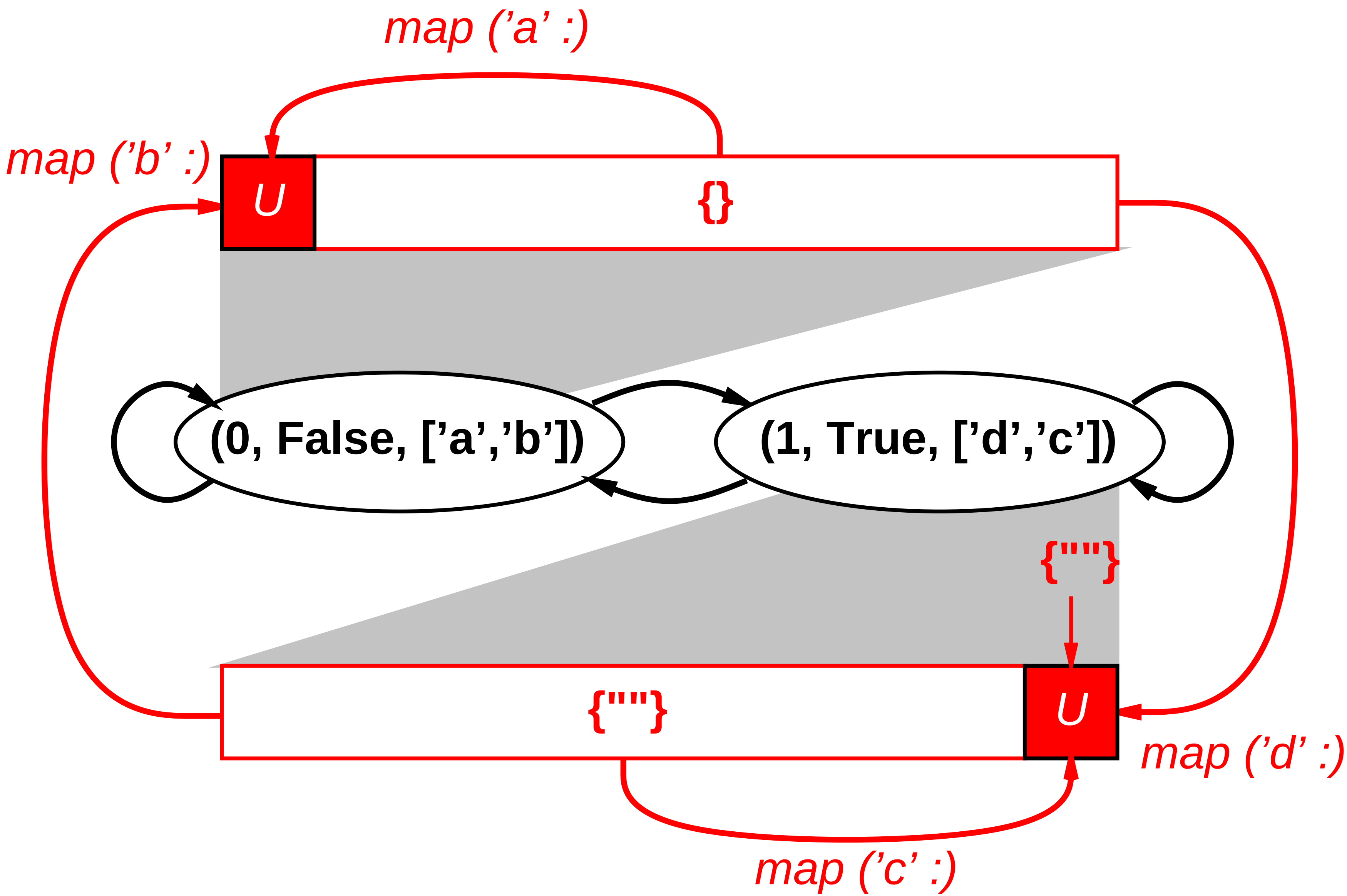




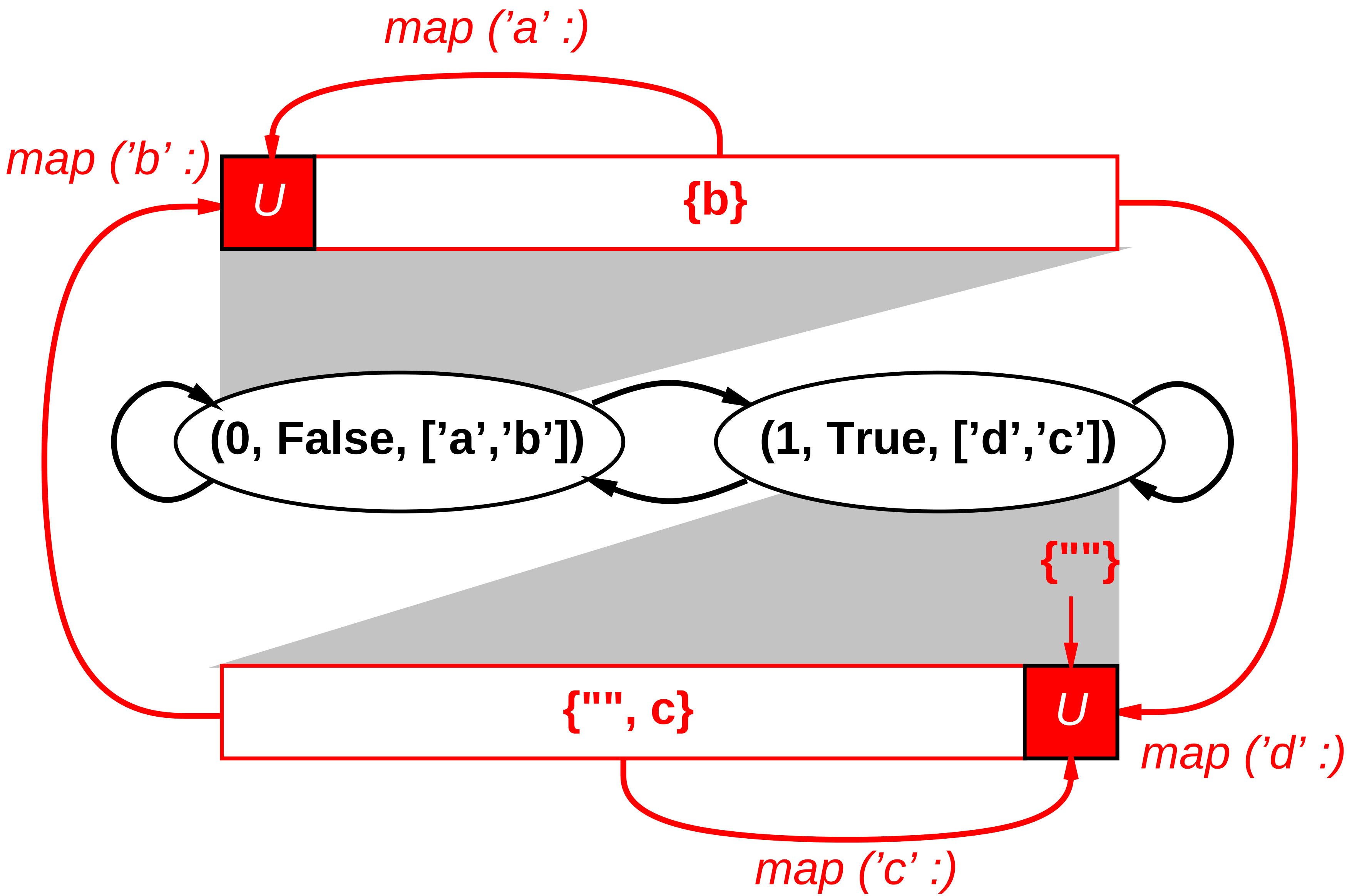
Cycfold Example 2: DFA Strings



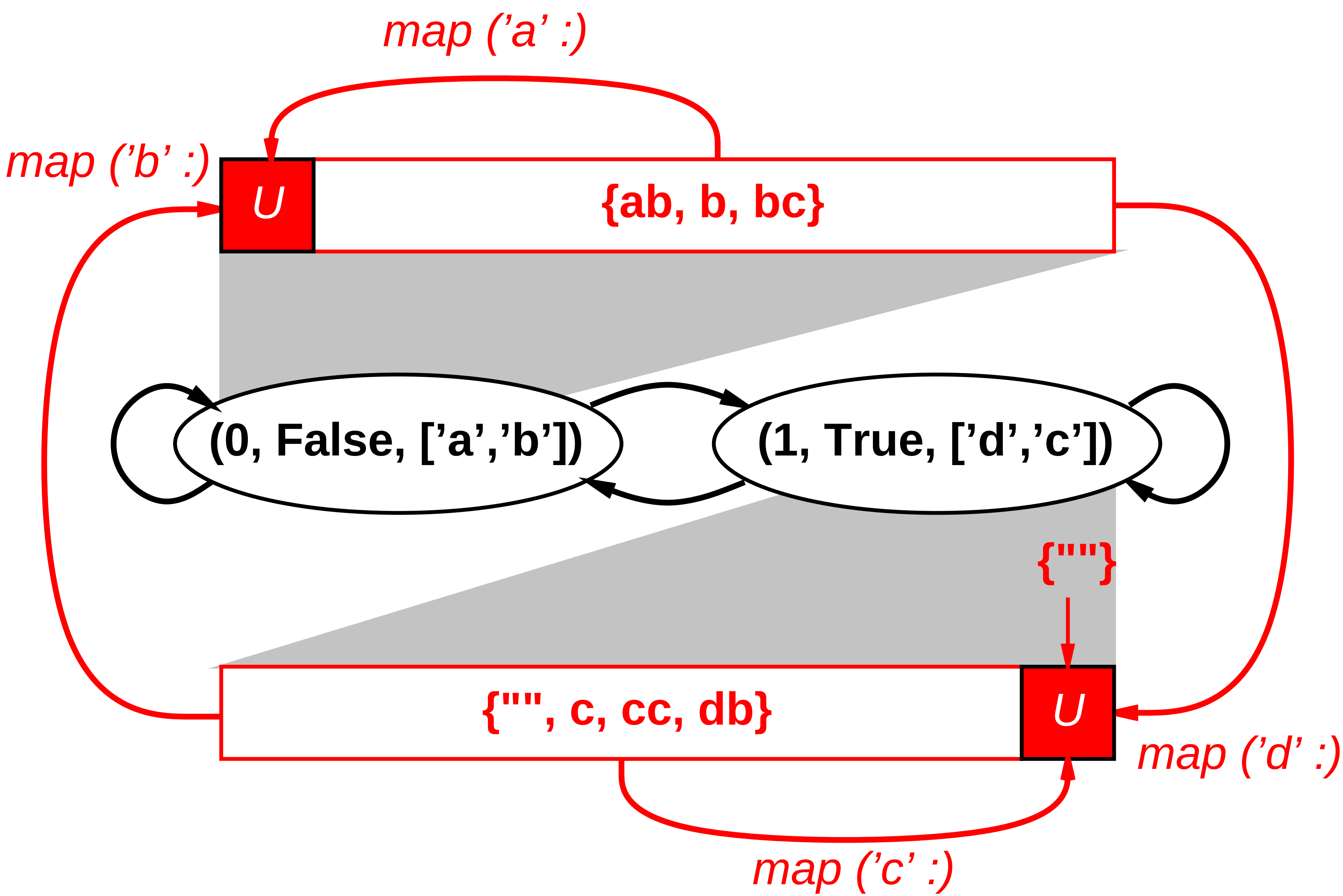
Cycfold Example 2: DFA Strings



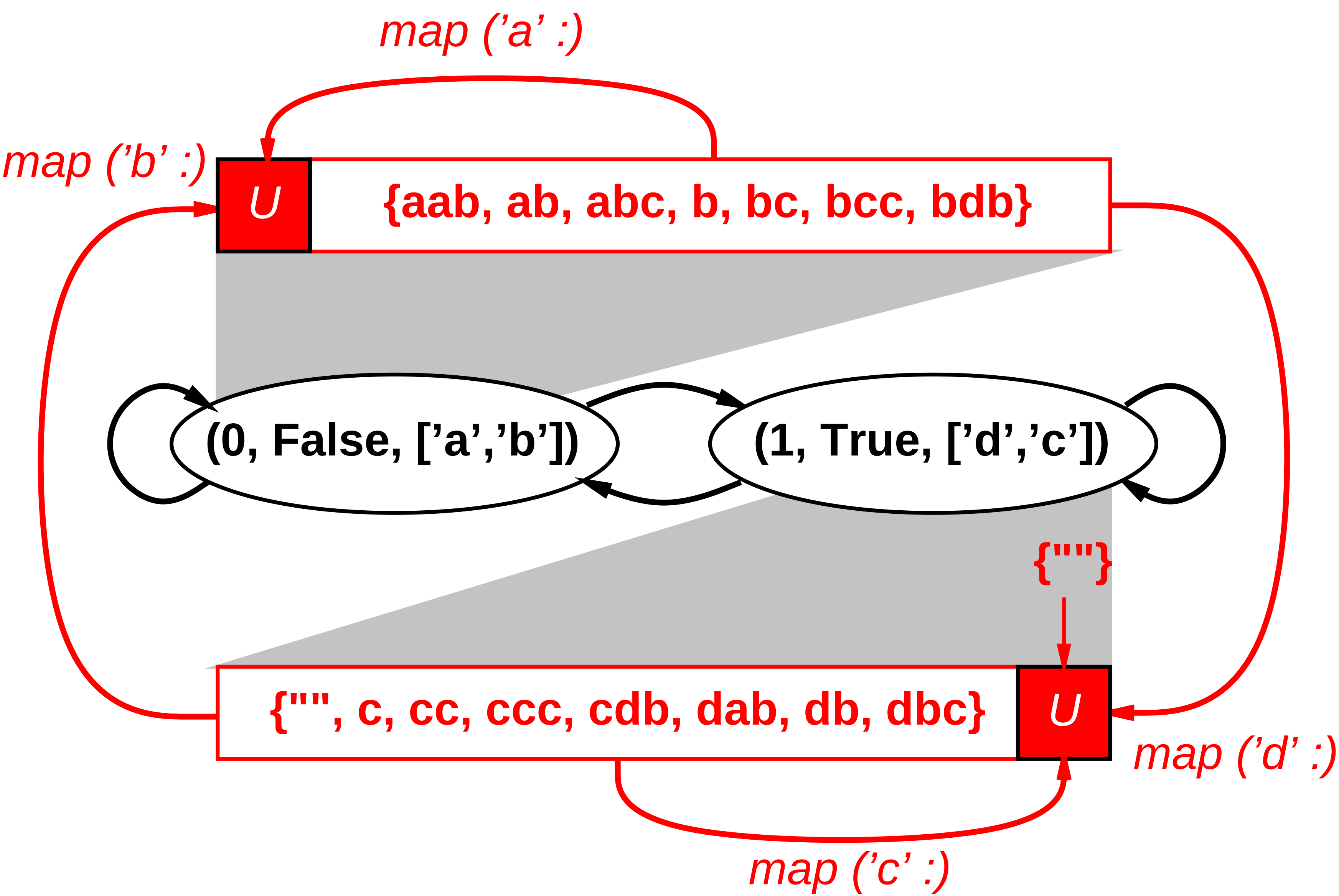
Cycfold Example 2: DFA Strings



Cycfold Example 2: DFA Strings



Cycfold Example 2: DFA Strings



Cycfold: Related Work

- Iterative fixed points common in compiler data flow.
- Graph folds (Gibbons, unpublished):
  - `ifold` = `foldtree` ∘ `untie`, analagous to fold.
  - `efold` analagous to cycfold.
- Catamorphisms over datatypes with embedded functions (Fegaras & Sheard, POPL'96):
  - Express cycles via embedded functions. E.g.,
    `val alts = Rec(fn x => Cons(0,(Cons 1 x)))`
  - Can express catamorphisms over such cycles (e.g., `map`), but these can expose the structure of the representative.

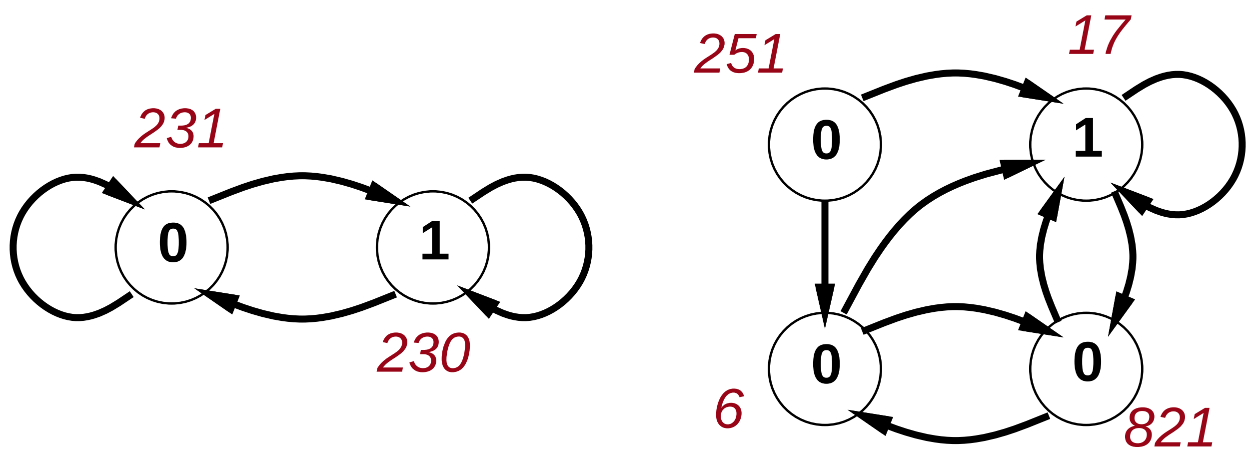
Road Map

- Viewing cyclic structures as infinite regular trees.
- Adapting the tree-generating unfold function to generate cyclic structures for infinite regular trees.
- Adapting the tree-accumulating fold function to return non-trivial results for strict combining functions and infinite regular trees.
- Cycamores: an abstraction for manipulating regular trees that we have implemented in ML and Haskell.

Cycamores

Cycamore(L) is the type of potentially cyclic graphs, with a hidden UID for each node, parameterized over label type.

- Examples:



- Key operations: make, view, unfold, fold, cycfold.
- Other operations: cycfix, memofix (see paper).
- Implementations in Standard ML and Haskell.

Cycamore Signatures 1

Standard ML		Haskell
val make :		make ::
('a * 'a Cycamore list)	*label/kids pair*	(a, [Cycamore a])
-> 'a Cycamore	*result cycamore*	-> Cycle s (Cycamore a)
val view :		view ::
'a Cycamore	*given cycamore*	Cycamore a
-> ('a * 'a Cycamore list)	*label/kids pair*	-> (a, [Cycamore a])
val unfold :		unfold ::
	order class	Ord a =>
'a MemKey	*memo key fcn.*	
-> ('a -> ('b * 'a list))	*generating fcn.*	(a -> (b, [a]))
-> 'a	*seed*	-> a
-> 'b Cycamore	*result cycamore*	-> Cycle s (Cycamore b)

Cycamore Signatures 2

Standard ML		Haskell
val fold :		fold ::
('b -> ('a list) -> 'a)	*combining fcn.*	(b -> [a] -> a)
-> ('b Cycamore)	*cycamore*	-> (Cycamore b)
-> 'a	*result*	a
val cycfold :		cycfold ::
	partial order class	(POrd a) =>
'a	*user bottom*	a
-> (('a * 'a) -> bool)	*geq*	
-> ('b -> ('a list) -> 'a)	*combining fcn.*	-> (b -> [a] -> a)
-> ('b Cycamore)	*cycamore*	-> (Cycamore b)
-> 'a	*result*	-> a

Future Work

- Theory:
  - Non-strict combining functions with cycfold.
  - Can cycfold return a cycamore?
  - Version of fold based on greatest fixed points.
- Practice:
  - Avoiding single-threaded UID generation.
  - Memoization strategies.
  - cycfold implementation heuristics.
  - Cyclic hash-consing experimentation.
- Extending ML/Haskell with general cyclic data types.