1 Tree automata

What is a Tree Automaton?
Decision Problems

2 Logic

Logic for Words
Logic for Trees
Transitive Closure Logic

3 Temporal Logics

Temporal Logic for Words
Temporal Logic for Trees
XPath

4 Tree-Walking Automata, 1

Tree-Walking Automata
Expressive Power
Pebble Automata

5 Tree-Walking Automata, 2

Tree-Walking Automata Cannot Be Determinized

Some logics that describe tree properties

monadic second-order logic

"There is a set of nodes that is closed under parents, has an a label, and has no c label"

$$\exists X \ \wedge \begin{cases} \exists x \in X \ \ a(x) \\ \forall x \in X \ \forall y \ \ \mathrm{parent}(x,y) \Rightarrow y \in X \\ \forall x \in X \ \ \neg c(x) \end{cases}$$

temporal logics

"On some path, b holds until a holds"

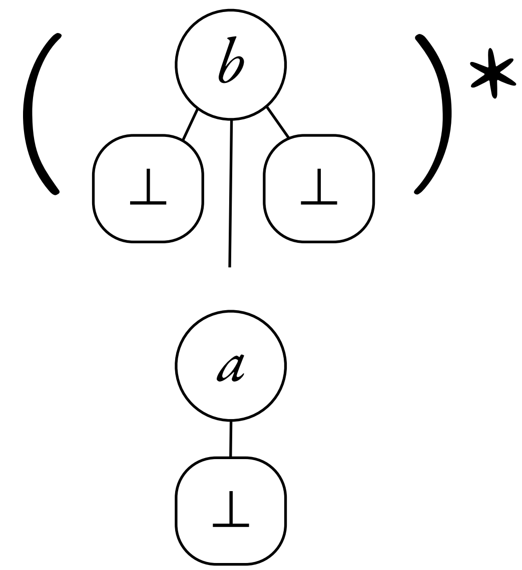
$\mathbf{E}\, b\, \mathbf{U}\, a$

first-order logic

"There is a node with label a that has only b-labeled ancestors"

$$\exists x \ \ a(x) \ \ \wedge \ \ (\forall y < x \ \ b(y))$$

regular expressions



first-order logic with transitive closure

Instead of < we can write

$(\mathrm{parent}(x,y))^*$

Temporal Logic for Words

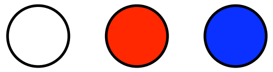
definition
the virtuous cycle
MSO=regular

Temporal Logic for Trees

definition
CTL, PDL, CTL*
expressivity

XPath

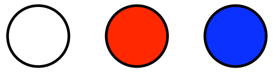
definition
two-variable logic
regular XPath

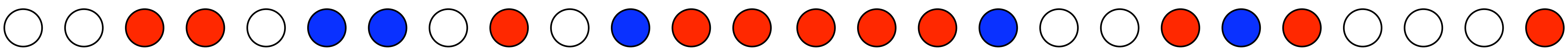
alphabet: 

LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

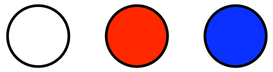
alphabet: 

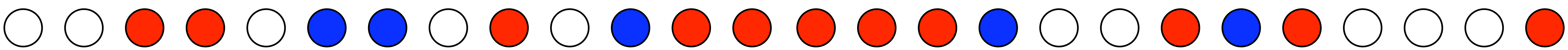


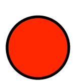
LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

alphabet: 

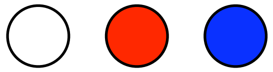


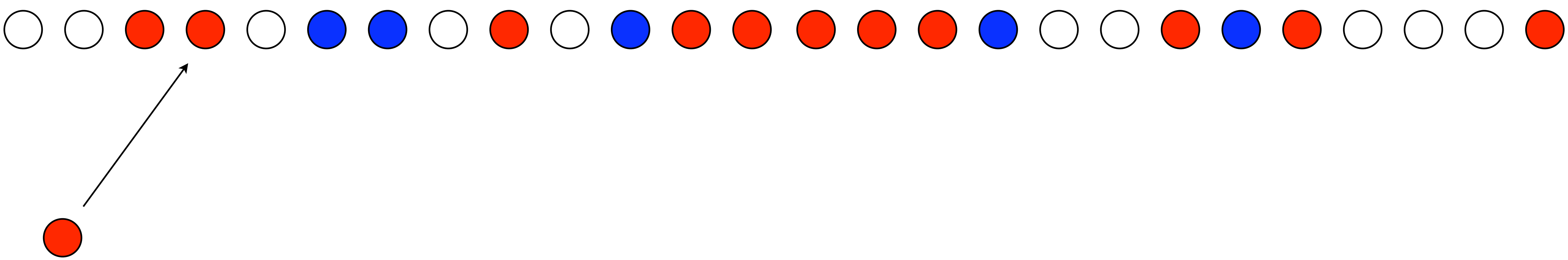


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

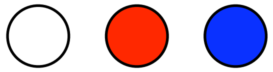
alphabet: 

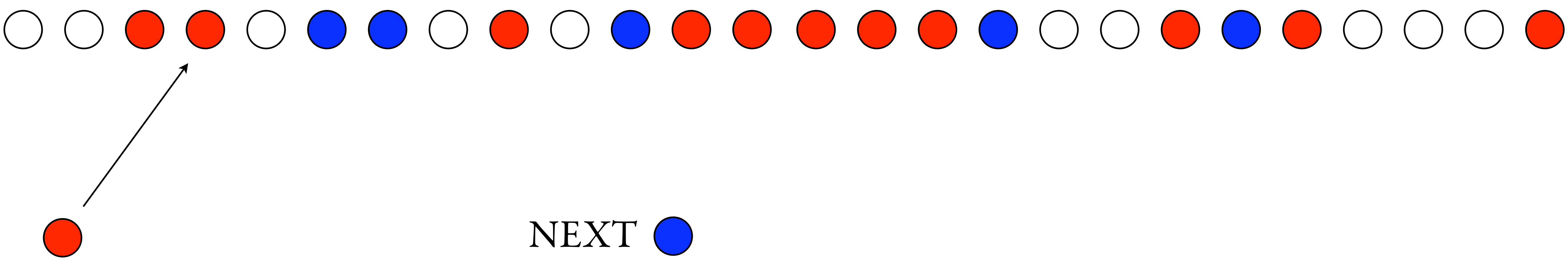


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

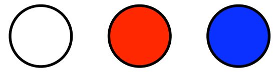
alphabet: 

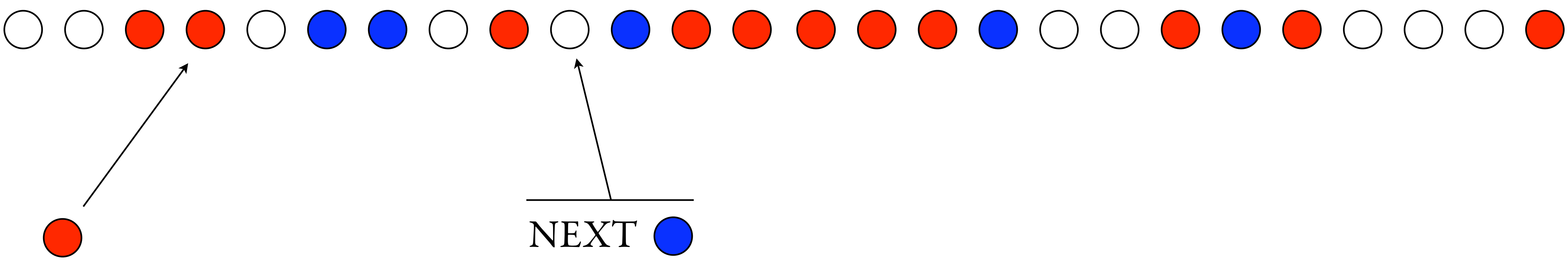


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

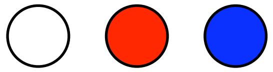
alphabet: 

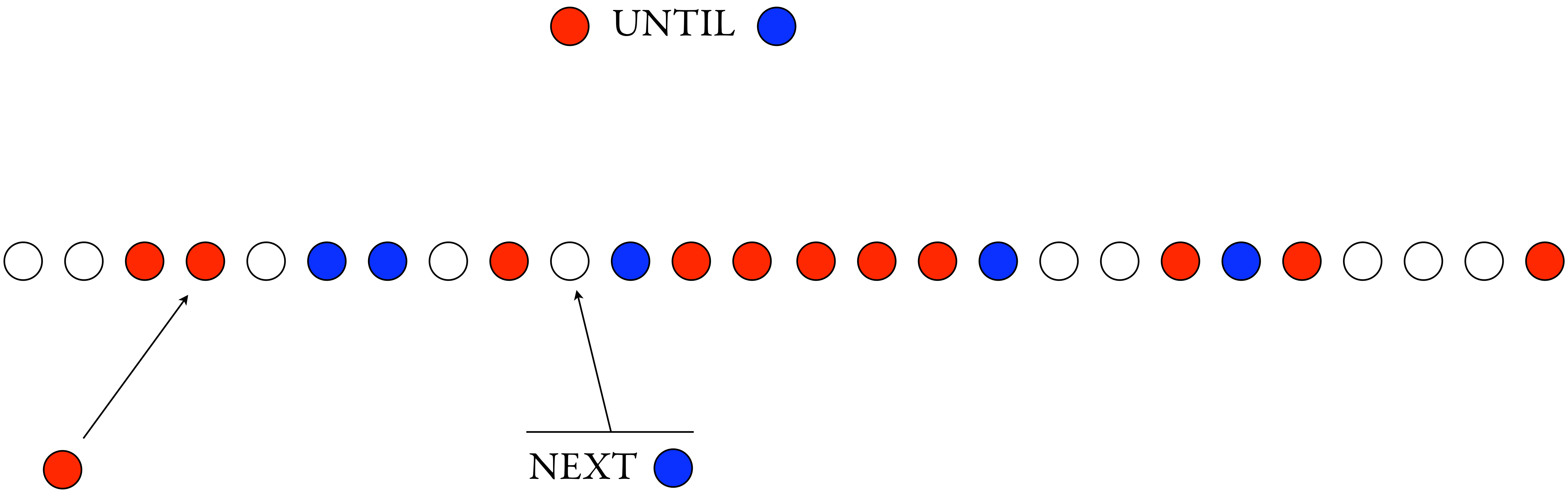


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

alphabet: 

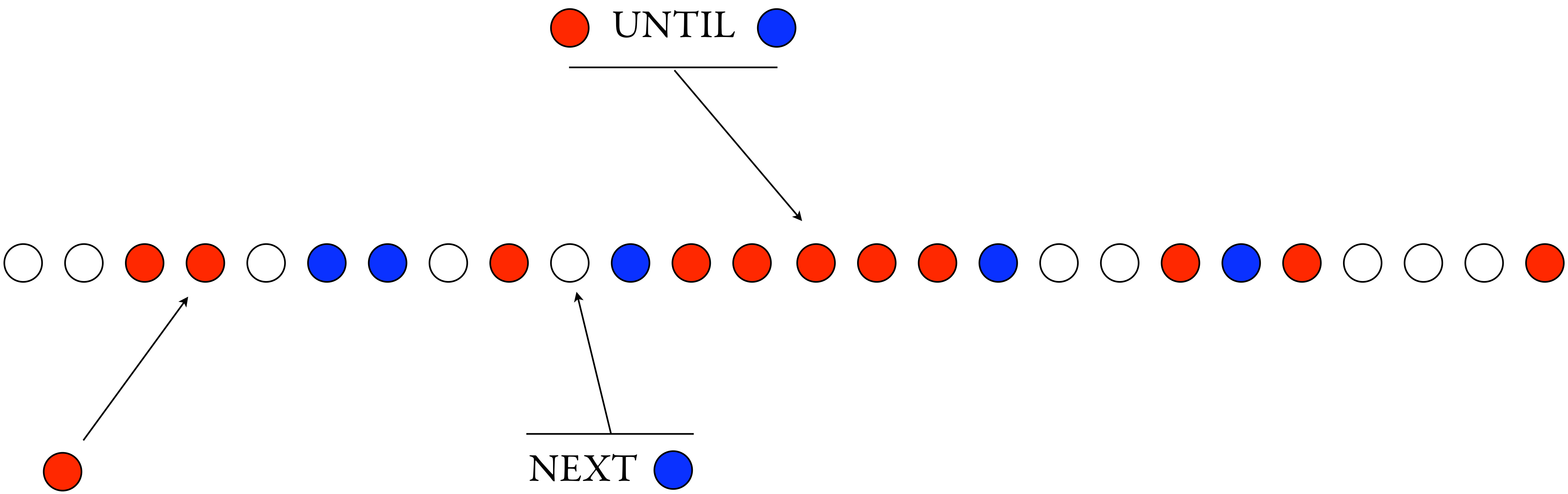


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

alphabet:

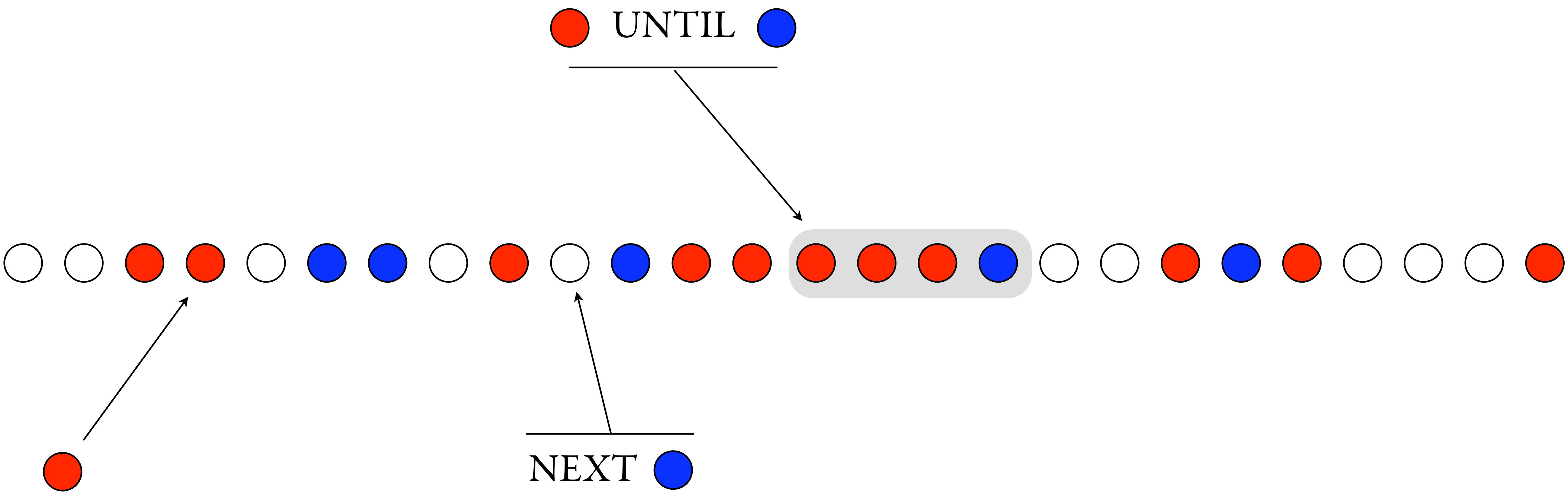


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

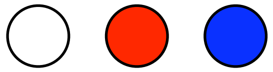
alphabet:

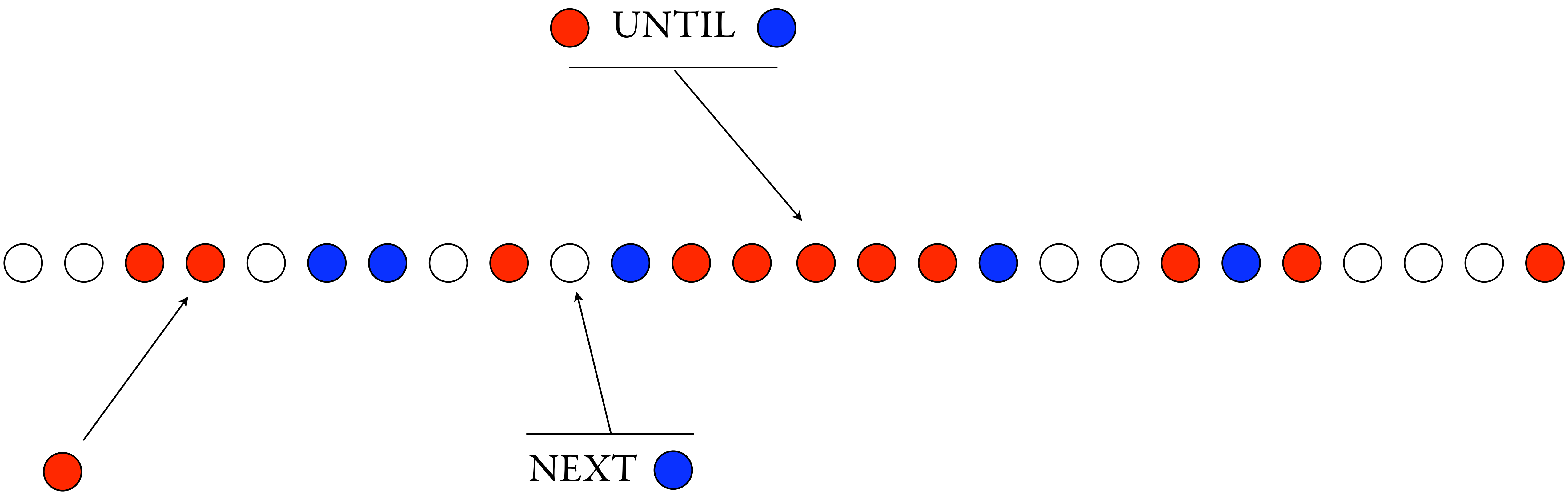


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

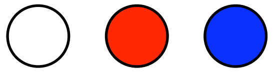
alphabet: 

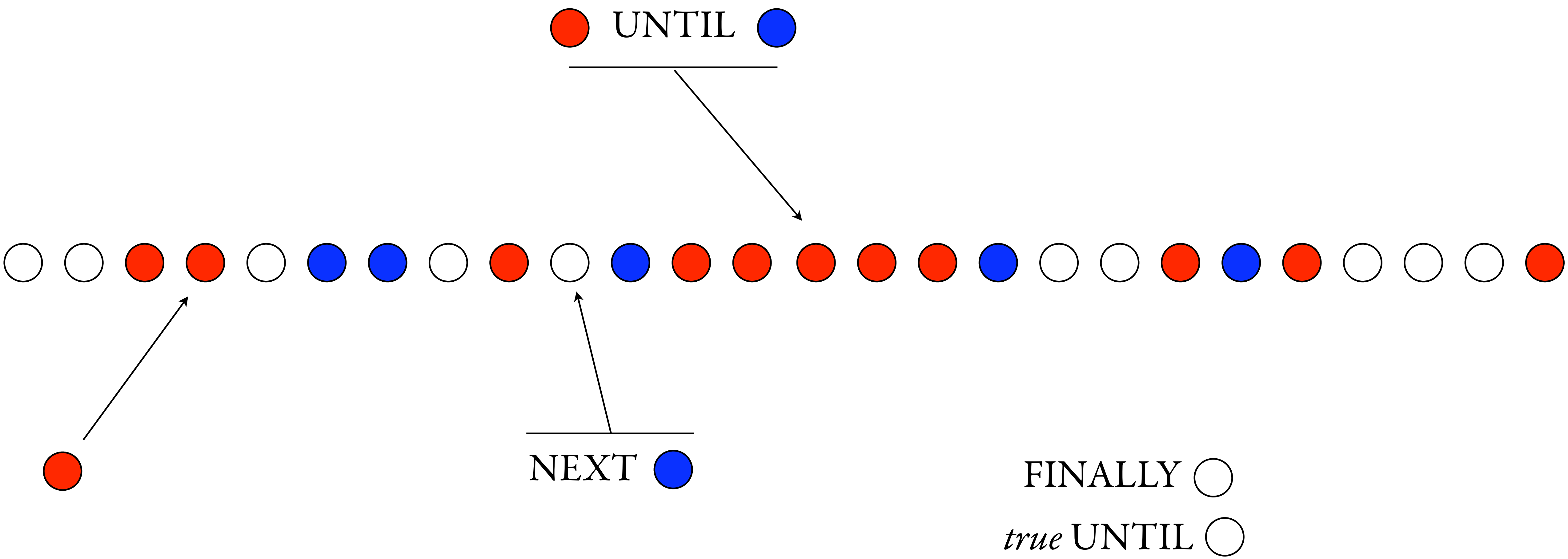


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

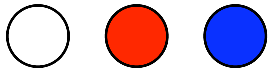
alphabet: 

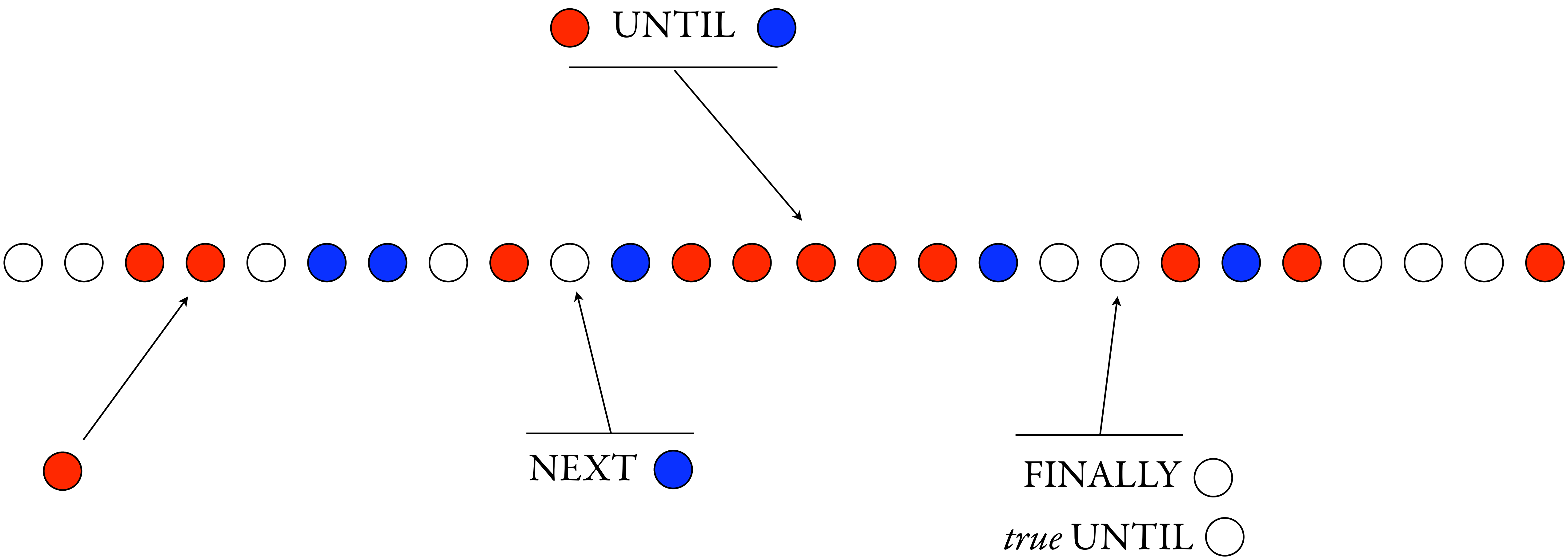


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

alphabet: 

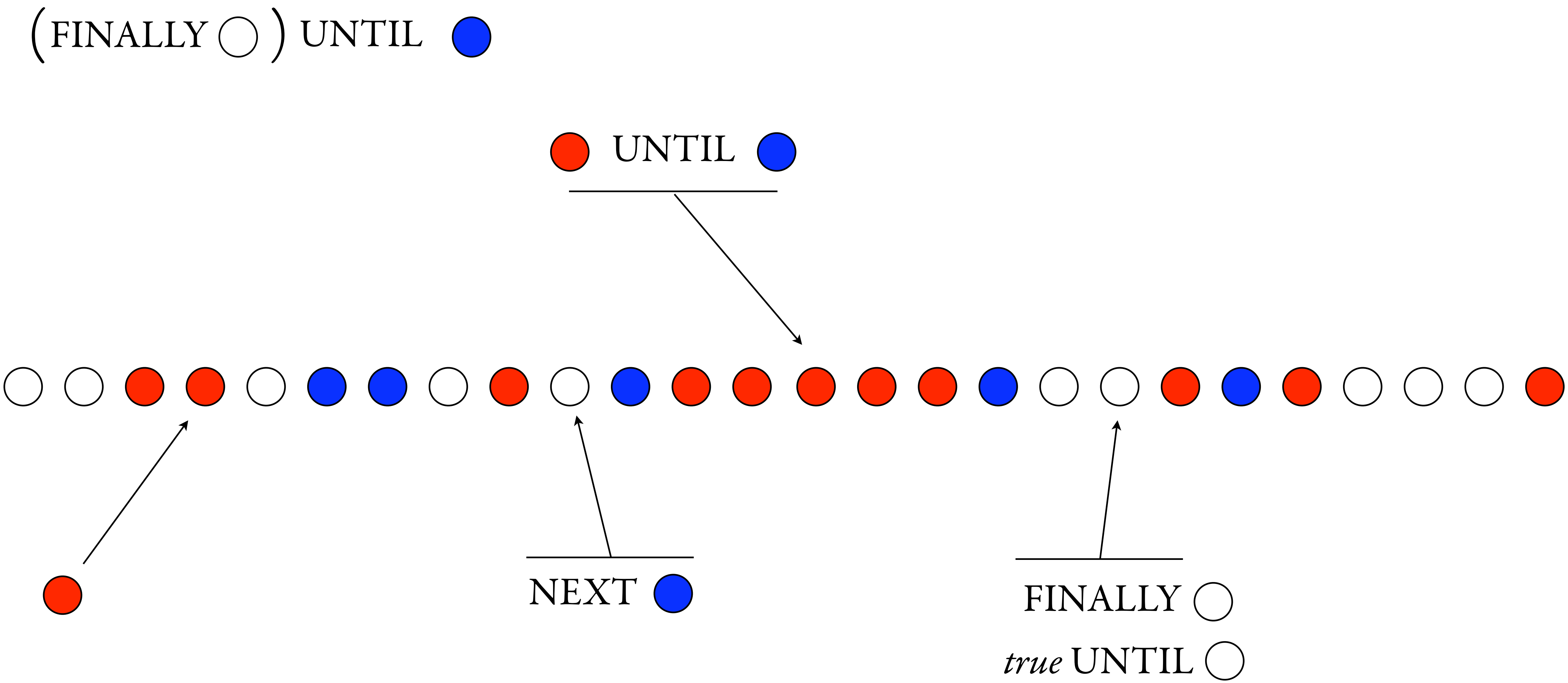


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

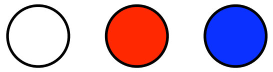
alphabet:

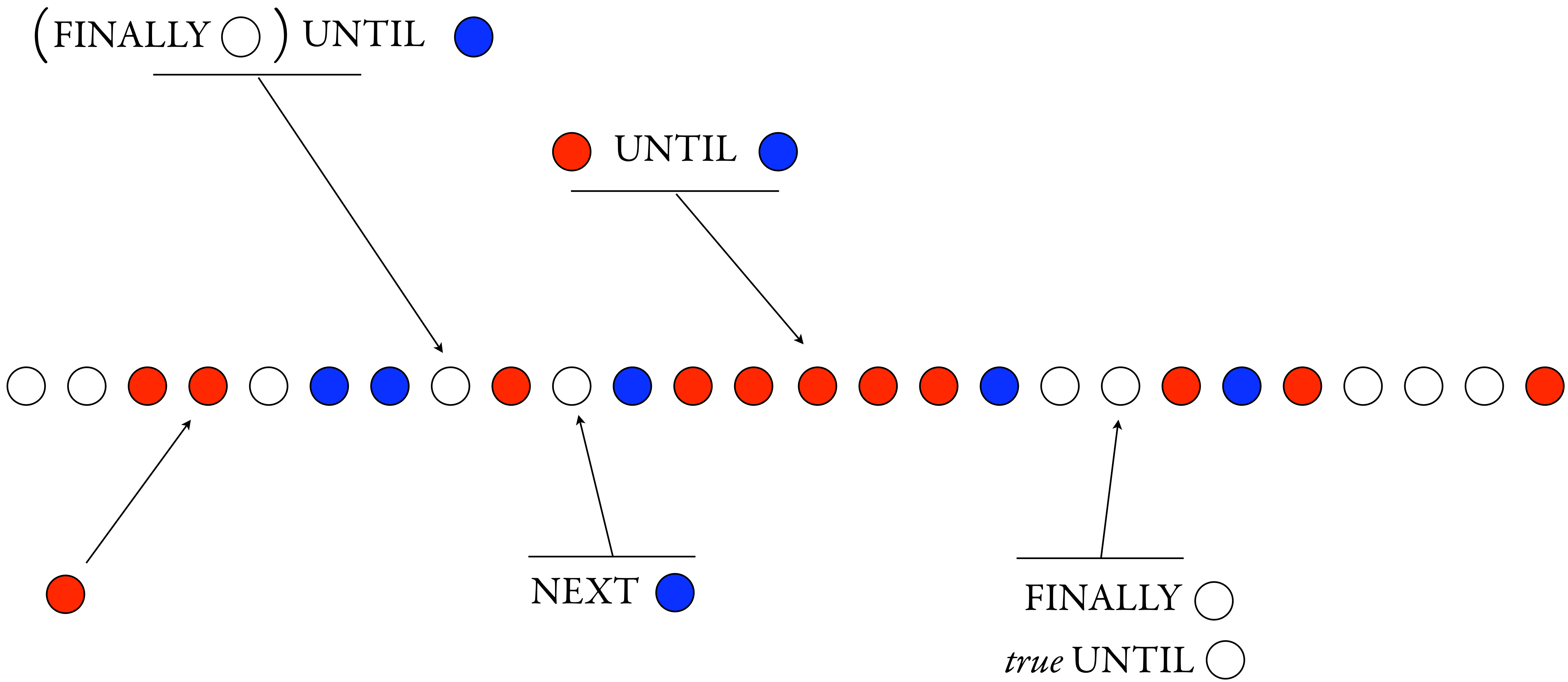


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

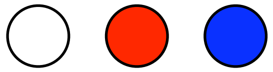
alphabet: 

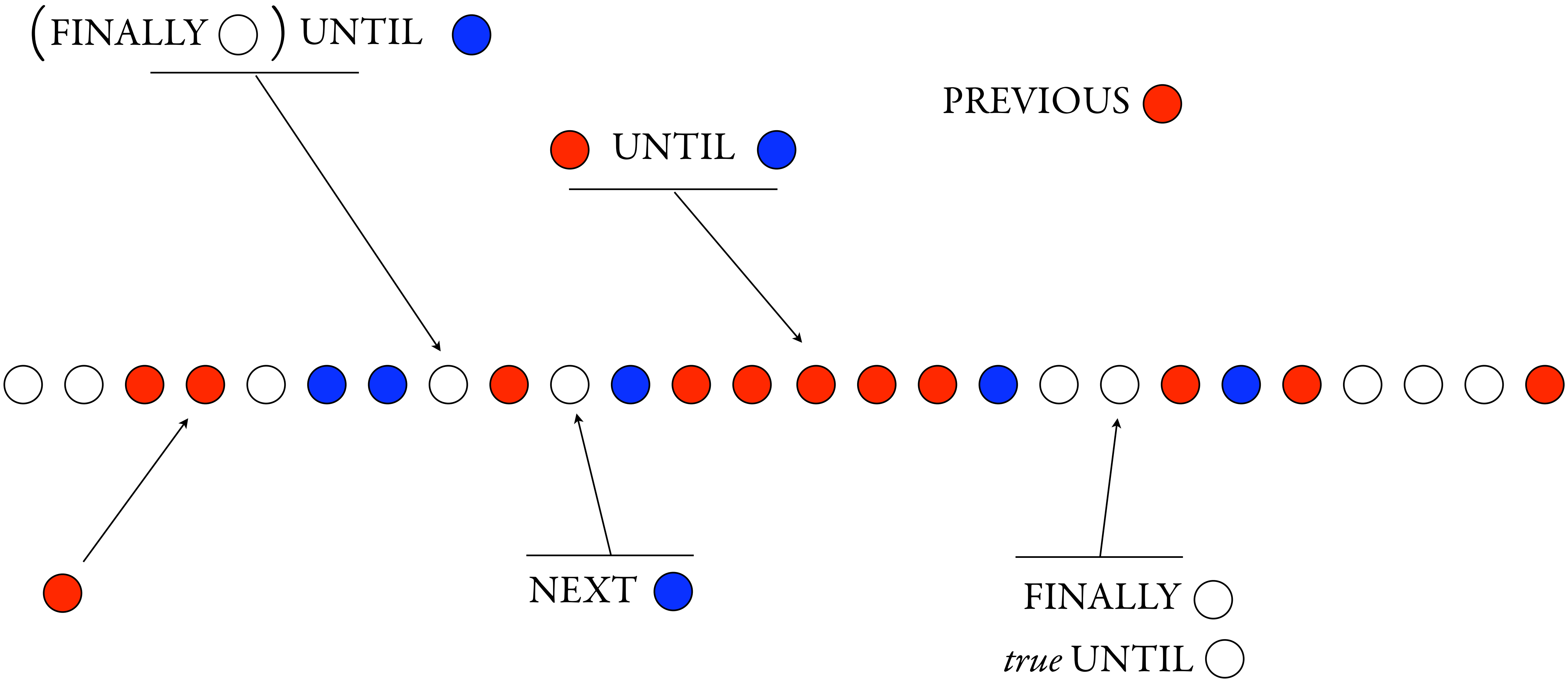


LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

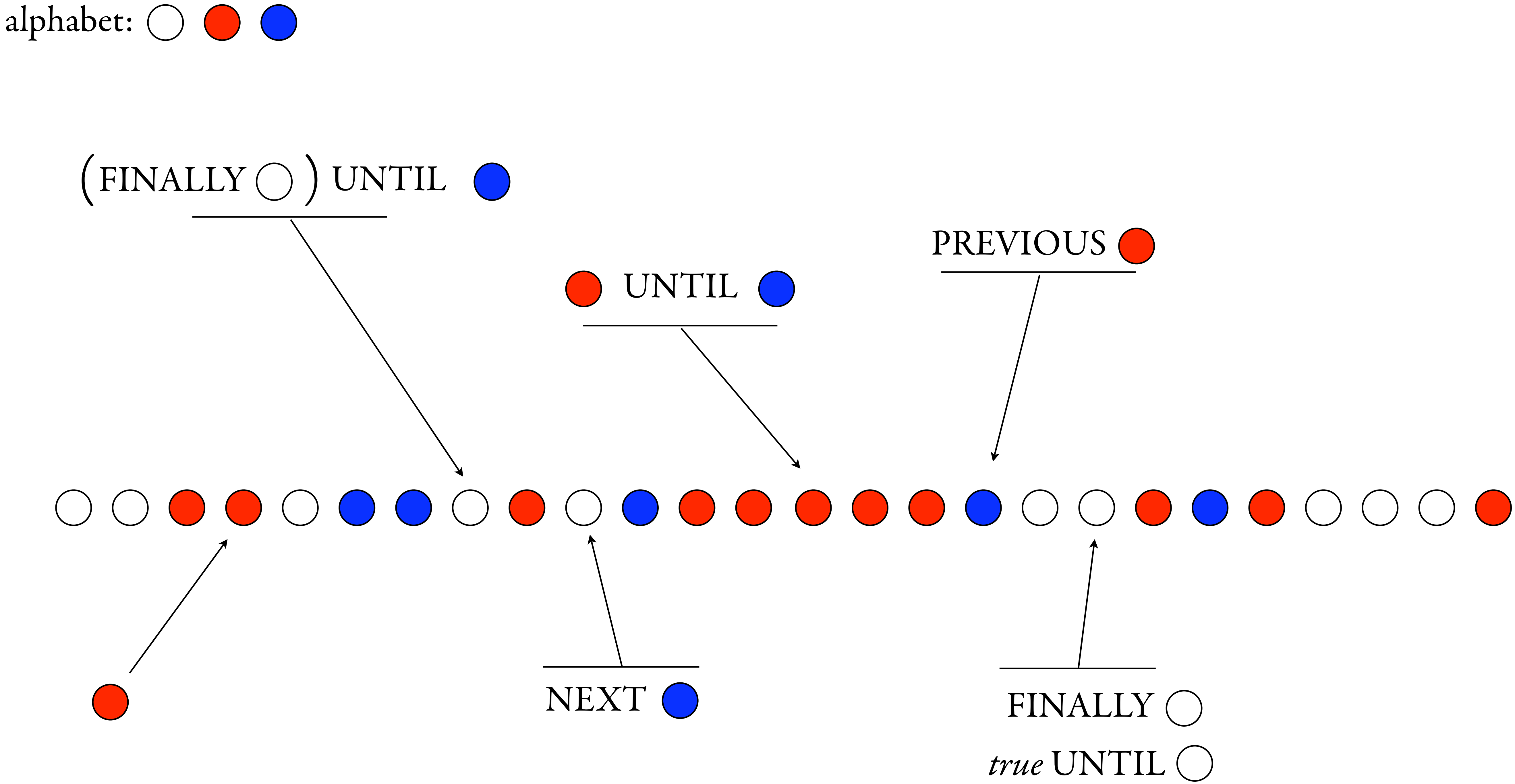
alphabet: 



LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

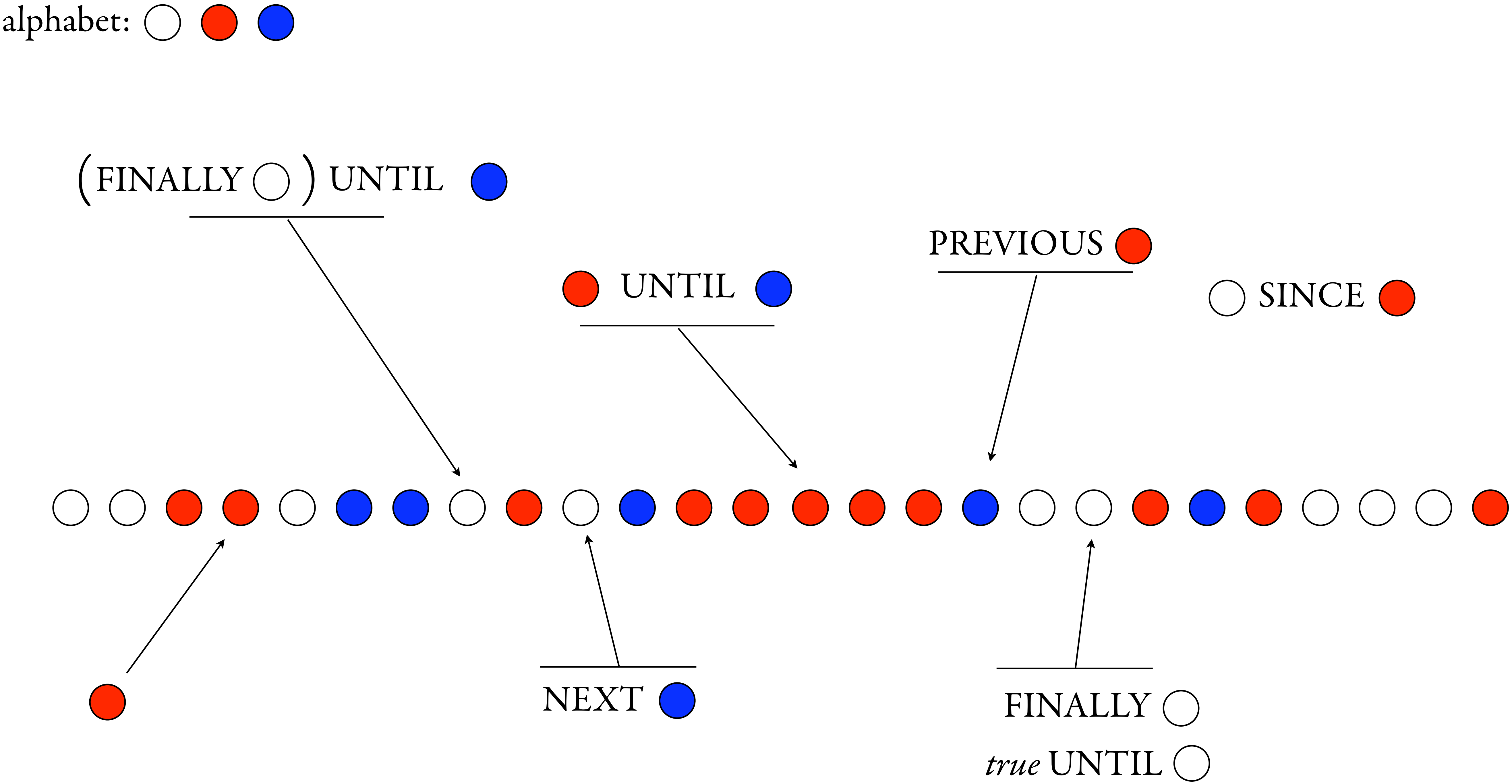
Thm (Kamp)
LTL = FO(<)



LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

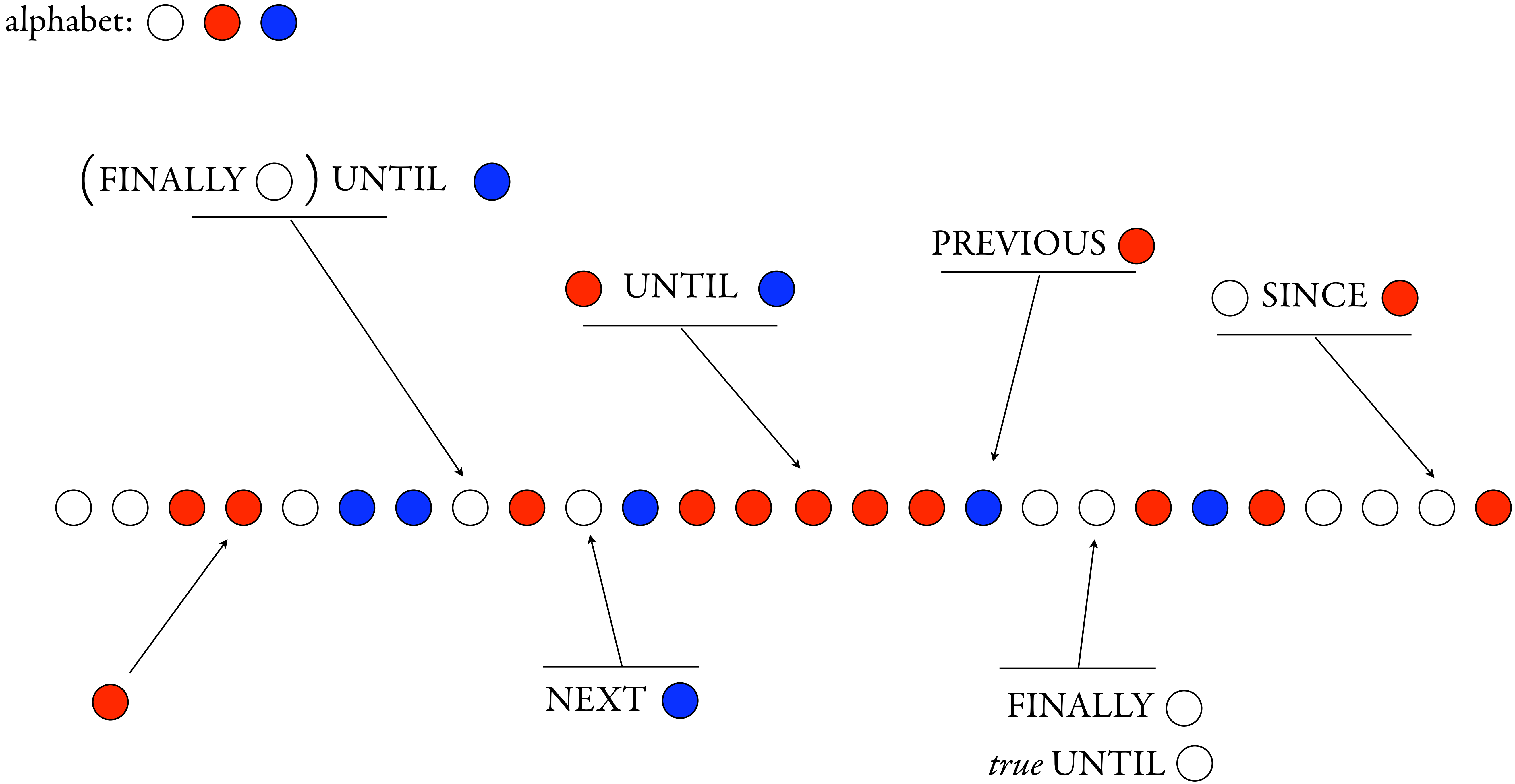
Thm (Kamp)
LTL = FO(<)



LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)



LTL (Linear Time Logic):
UNTIL and NEXT

2LTL
UNTIL, NEXT, SINCE, PREVIOUS

Thm (Kamp)
LTL = FO(<)

Virtuous Cycle

For word languages, the following have the same expressive power:

FO(<)

aperiodic semigroups

LTL

Virtuous Cycle

For word languages, the following have the same expressive power:

FO(<)

easy

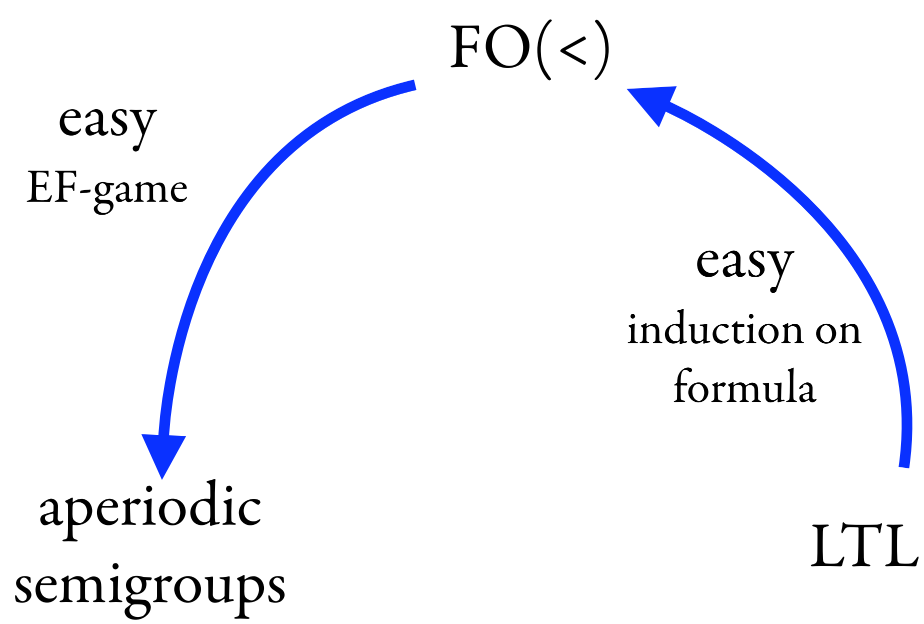
EF-game

aperiodic semigroups

LTL

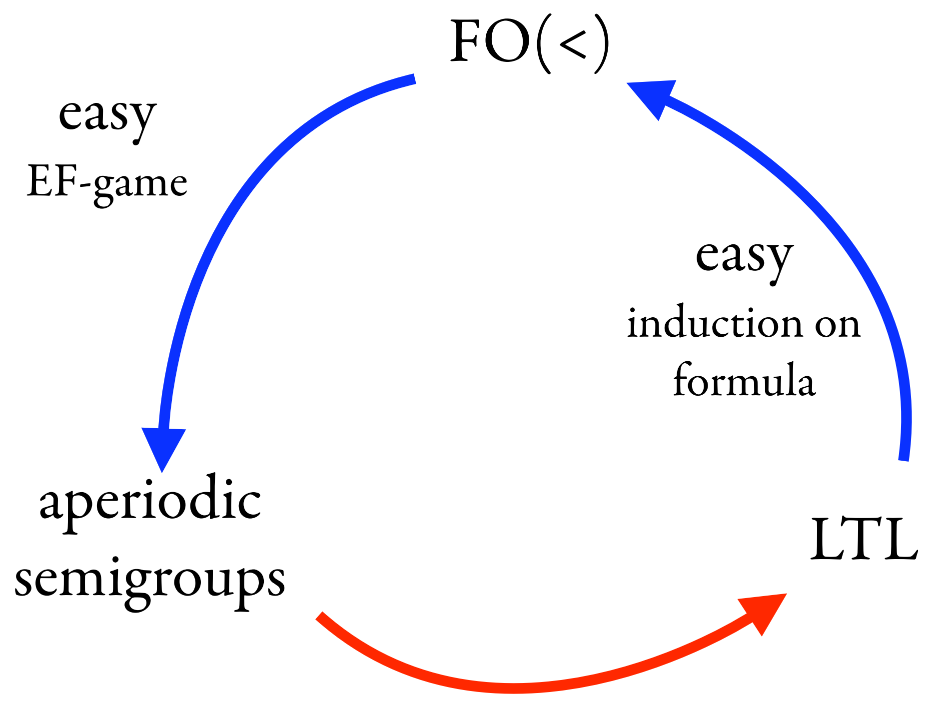
Virtuous Cycle

For word languages, the following have the same expressive power:



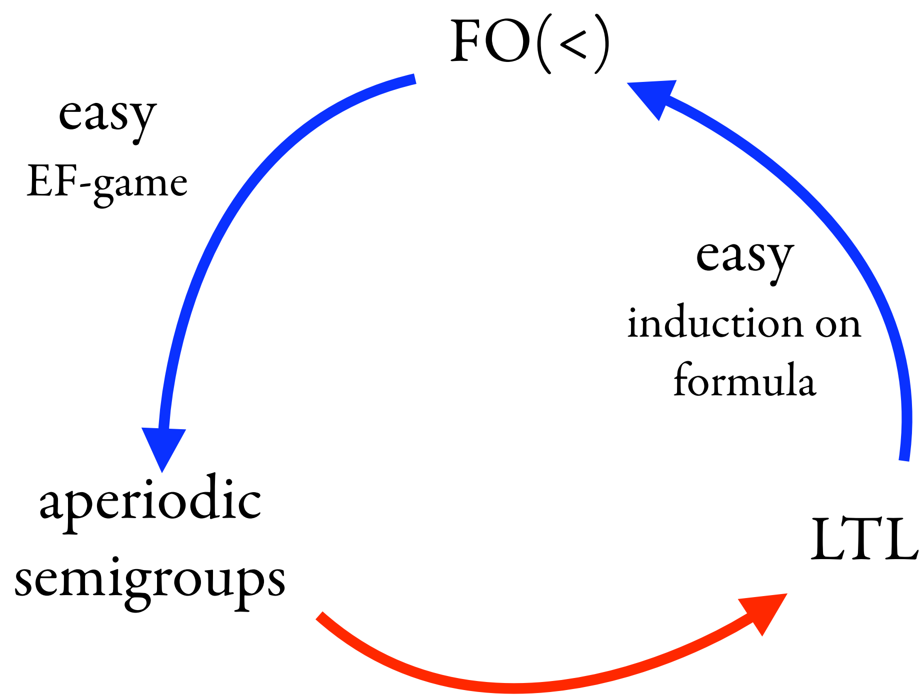
Virtuous Cycle

For word languages, the following have the same expressive power:



Virtuous Cycle

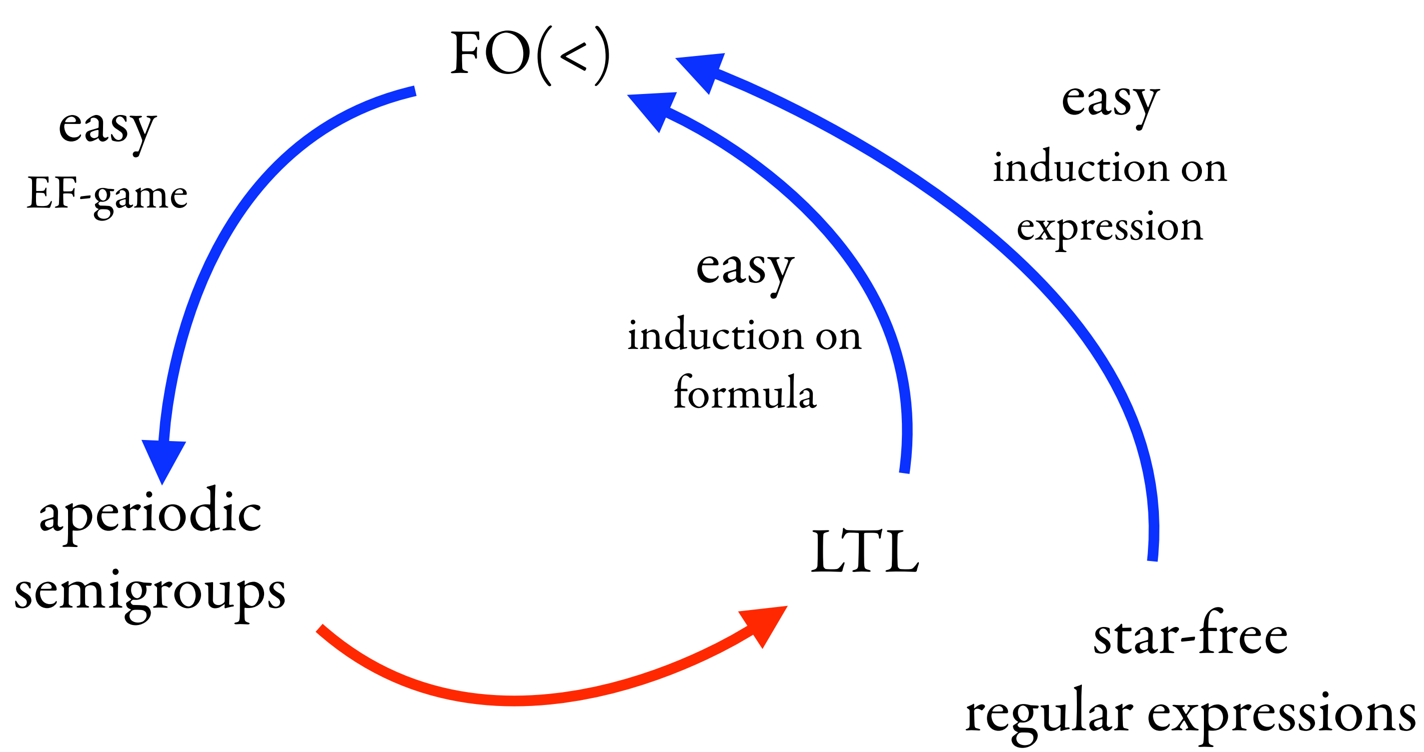
For word languages, the following have
the same expressive power:



star-free
regular expressions

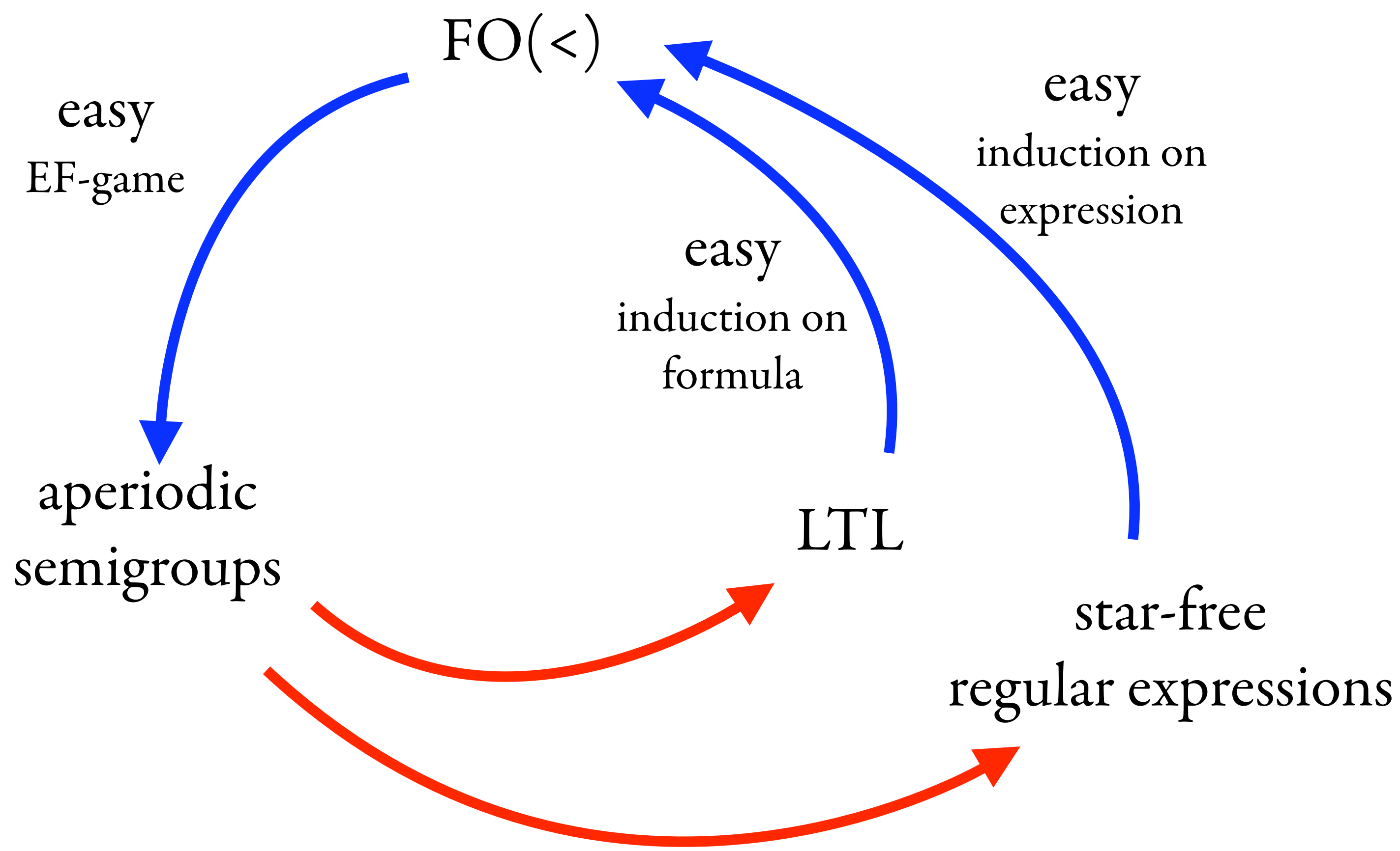
Virtuous Cycle

For word languages, the following have the same expressive power:



Virtuous Cycle

For word languages, the following have the same expressive power:

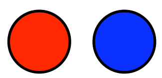


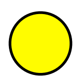
An LTL-relabeling is a function $f \colon A^* \longrightarrow B^*$

the relabeling is given by formulas $\{\varphi_b\}_{b \in B}$ and $\varphi_\perp$ over alphabet A.

An LTL-relabeling is a function $f: A^* \longrightarrow B^*$

the relabeling is given by formulas $\{\varphi_b\}_{b \in B}$ and $\varphi_\perp$ over alphabet A.

$A=$ 

$B=$ 

An LTL-relabeling is a function $f : A^* \longrightarrow B^*$

the relabeling is given by formulas $\{\varphi_b\}_{b \in B}$ and $\varphi_\perp$ over alphabet A.

$A=$ 🔴 🔵

$B=$ 🟡

$\varphi_\perp$ = (🔴 and NEXT 🔴) or (🔵 and NEXT 🔵)

$\varphi_{🟡}$ = $\neg\varphi_\perp$ = last in same-colored block

An LTL-relabeling is a function $f: A^* \longrightarrow B^*$

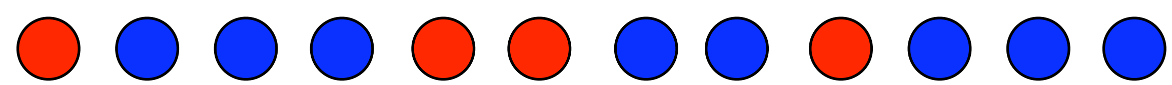
the relabeling is given by formulas $\{\varphi_b\}_{b \in B}$ and $\varphi_\perp$ over alphabet A.

$A=$

$B=$

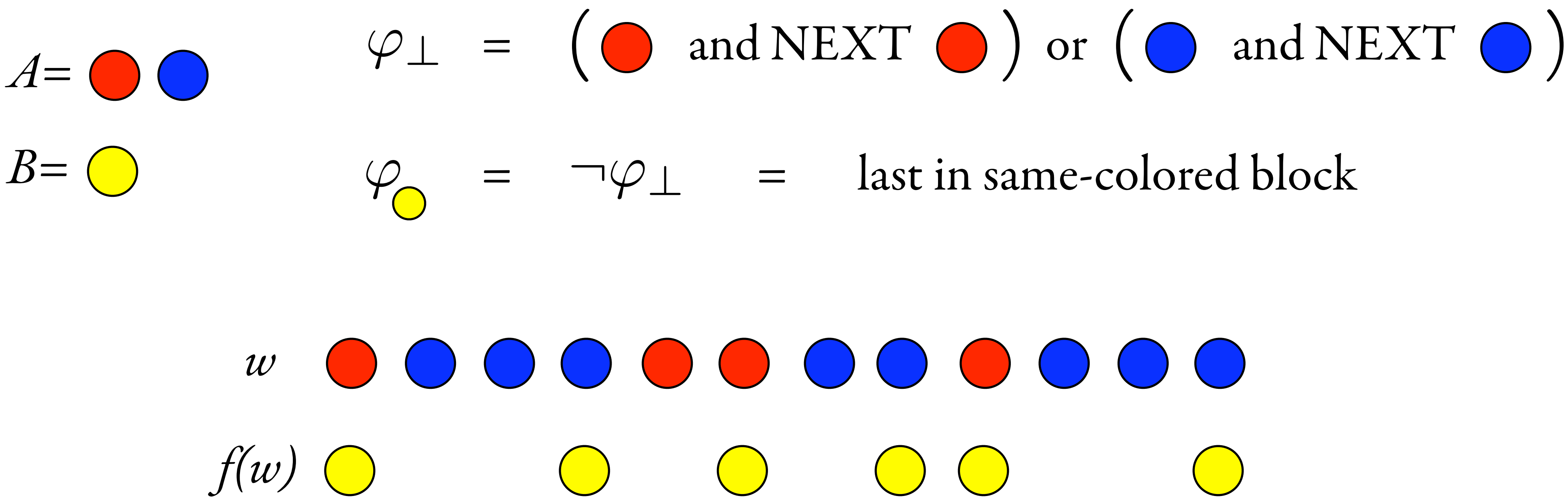
$\varphi_\perp = ($ and NEXT $)$ or $($ and NEXT $)$

$\varphi = \neg\varphi_\perp =$ last in same-colored block

w 

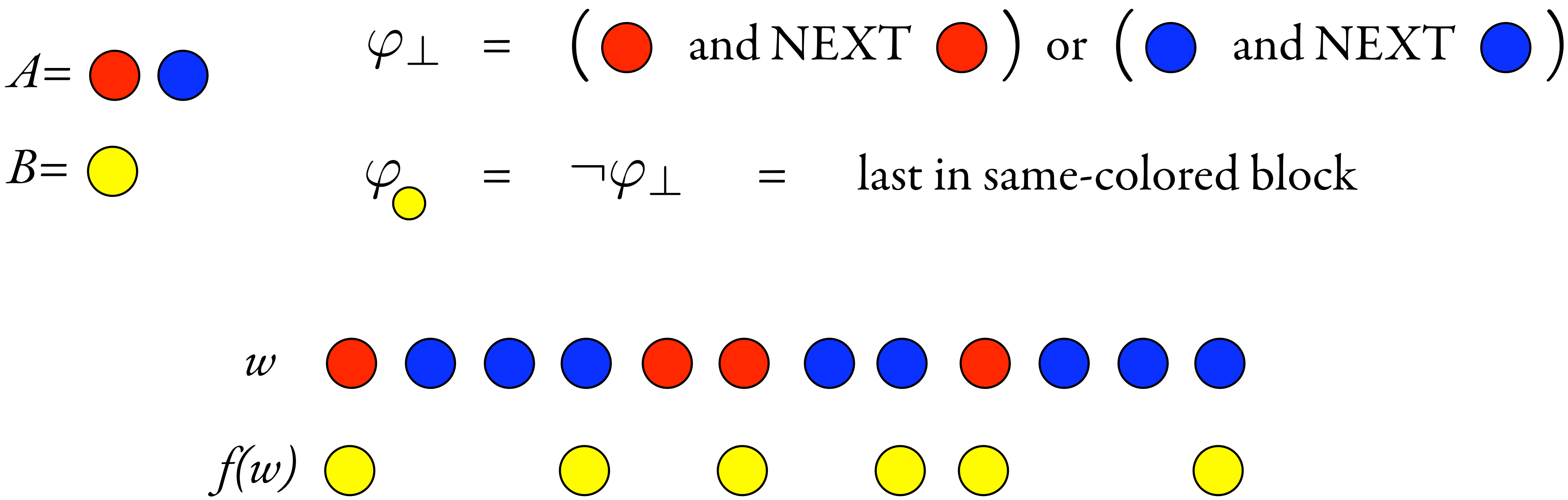
An LTL-relabeling is a function $f: A^* \longrightarrow B^*$

the relabeling is given by formulas $\{\varphi_b\}_{b\in B}$ and $\varphi_\perp$ over alphabet A.



An LTL-relabeling is a function $f: A^* \longrightarrow B^*$

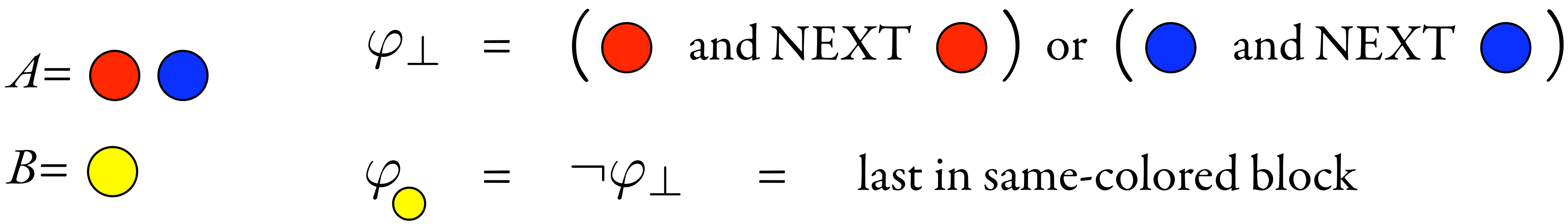
the relabeling is given by formulas $\{\varphi_b\}_{b \in B}$ and $\varphi_\perp$ over alphabet A.

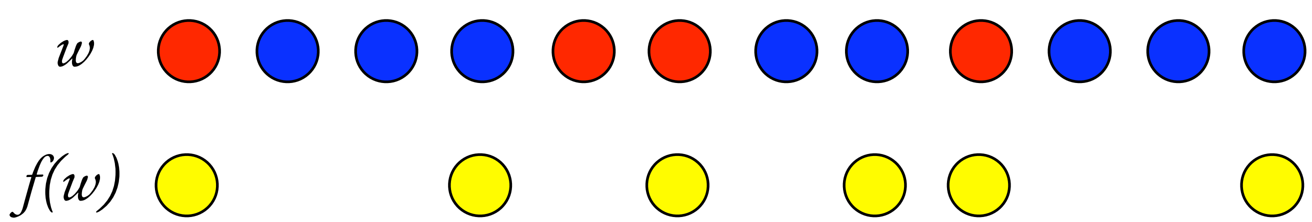


Lemma.
If $L \subseteq B^*$ is LTL-definable, and f is an LTL relabeling, then $f^{-1}(L)$ is also LTL-definable.

An LTL-relabeling is a function $f: A^* \longrightarrow B^*$

the relabeling is given by formulas $\{\varphi_b\}_{b \in B}$ and $\varphi_\perp$ over alphabet A.



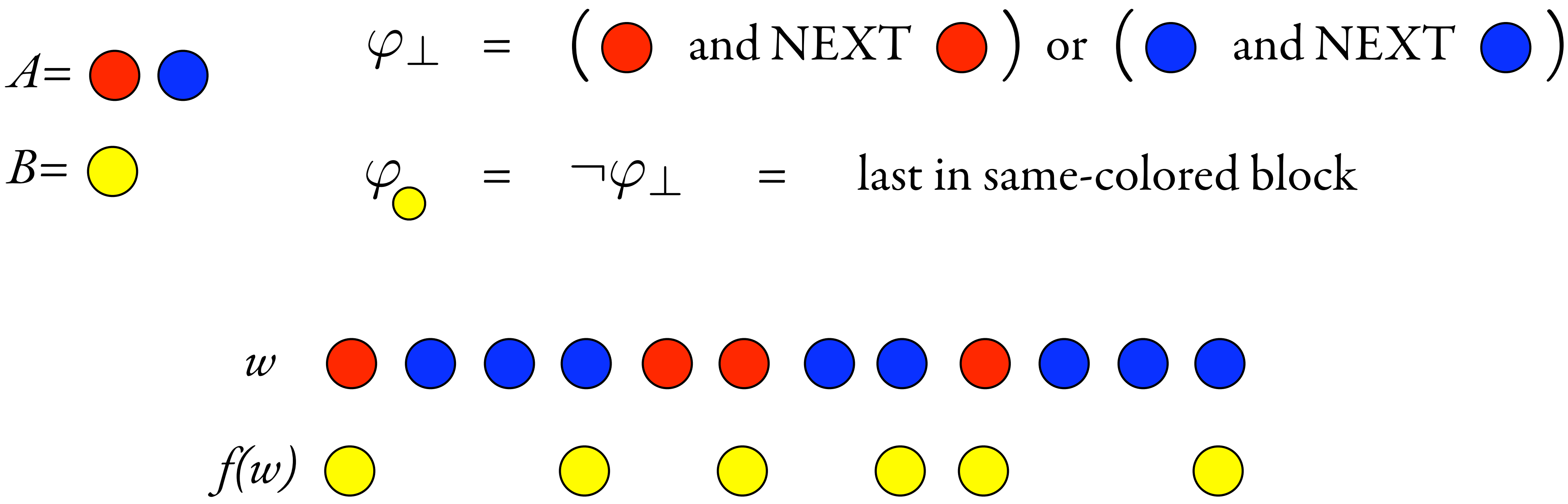


Lemma.
If $L \subseteq B^*$ is LTL-definable, and f is an LTL relabeling, then $f^{-1}(L)$ is also LTL-definable.

preimages: yes, images: no

An LTL-relabeling is a function $f: A^* \longrightarrow B^*$

the relabeling is given by formulas $\{\varphi_b\}_{b \in B}$ and $\varphi_\perp$ over alphabet A.



Lemma.
If $L \subseteq B^*$ is LTL-definable, and f is an LTL relabeling, then $f^{-1}(L)$ is also LTL-definable.

preimages: yes, images: no

$f($ first letter is and last letter is $) = ( )^* \notin$ LTL

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some ● $\in A$ such that $s = \alpha($●$)$ satisfies $sS \not\subset S$ or $Ss \not\subset S$

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

$$sS = \{ st \ : \ t \in S \}$$

$$sS \cdot sS \subseteq sS$$

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some ● $\in A$ such that $s = \alpha($●$)$ satisfies $sS \not\subset S$ or $Ss \not\subset S$

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

$$sS = \{\, st \;:\; t \in S \,\}$$

$$sS \cdot sS \subseteq sS$$

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some $\bullet \in A$ such that $s=\alpha(\bullet)$ satisfies $sS \not\subset S$ or $Ss \not\subset S$

Proof of Claim.
If $Ss = S$ then $t \longmapsto ts$ is a bijection. For aperiodic semigroups, such a bijection has to be the identity.

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

$sS = \{ st : t \in S \}$

$sS \cdot sS \subseteq sS$

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some ● $\in A$ such that $s = \alpha($●$)$ satisfies $sS \not\subset S$ or $Ss \not\subset S$

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some $\in A$ such that $s=\alpha()$ satisfies $sS \not\subset S$ or $Ss \not\subset S$

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

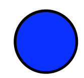
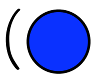
Induction base, when S has one element: the inverse image is A^+

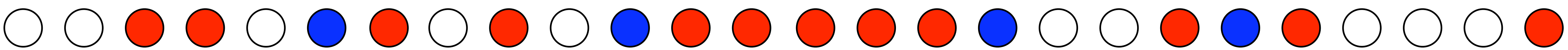
Claim.
Since S is aperiodic, there must be some $\in A$ such that $s=\alpha()$ satisfies $sS \not\subset S$ or $Ss \not\subset S$

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some  $\in A$ such that $s=\alpha($$)$ satisfies $sS \not\subset S$ or $Ss \not\subset S$

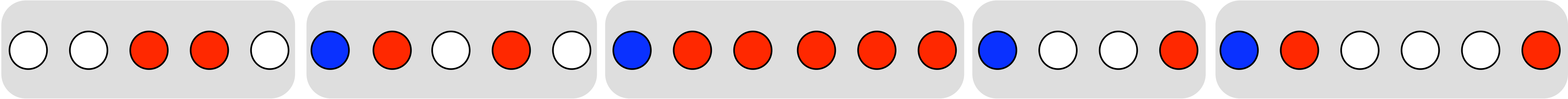


Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some $\in A$ such that $s=\alpha()$ satisfies $\boxed{sS \not\subset S}$ or $Ss \not\subset S$

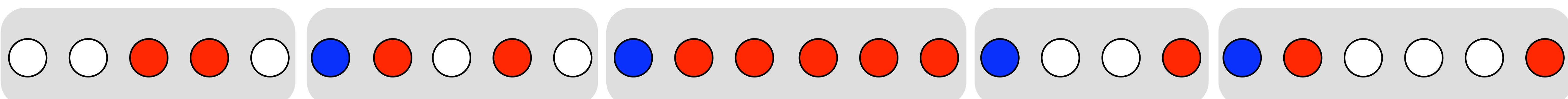


Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some $\in A$ such that $s = \alpha( )$ satisfies $sS \not\subset S$ or $Ss \not\subset S$



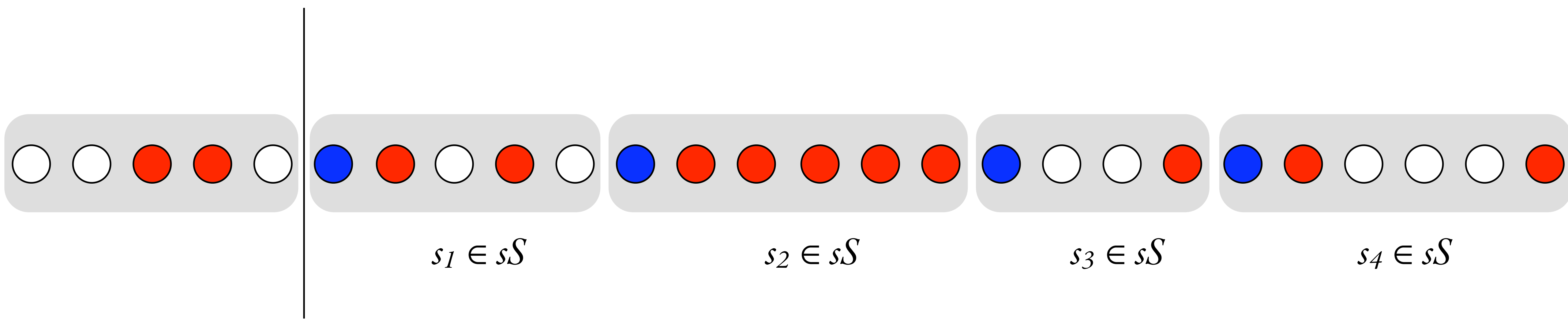
$s_1 \in sS$ $s_2 \in sS$ $s_3 \in sS$ $s_4 \in sS$

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some $\in A$ such that $s=\alpha(\;)$ satisfies $sS \not\subset S$ or $Ss \not\subset S$

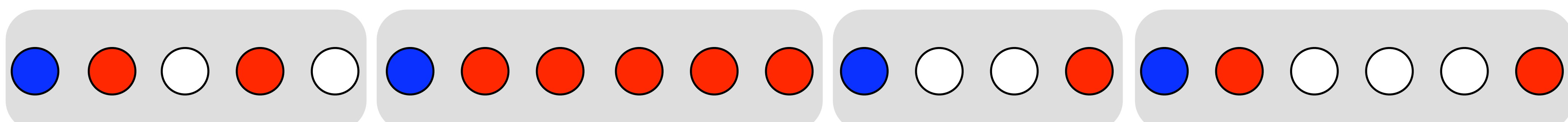


Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some $\in A$ such that $s = \alpha()$ satisfies $\boxed{sS \not\subset S}$ or $Ss \not\subset S$

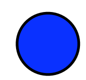
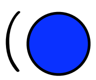


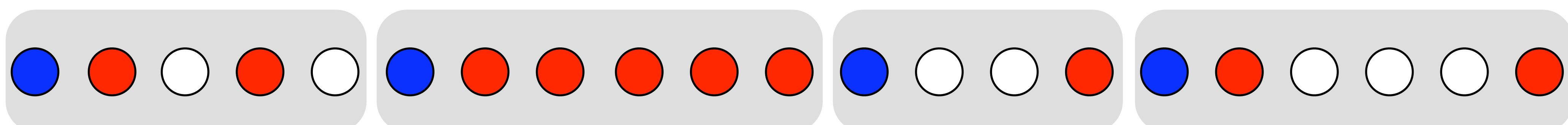
$s_1 \in sS$ $\quad$ $s_2 \in sS$ $\quad$ $s_3 \in sS$ $\quad$ $s_4 \in sS$

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some  $\in A$ such that $s=\alpha($$)$ satisfies $sS \not\subset S$ or $Ss \not\subset S$



$s_1 \in sS$ $s_2 \in sS$ $s_3 \in sS$ $s_4 \in sS$

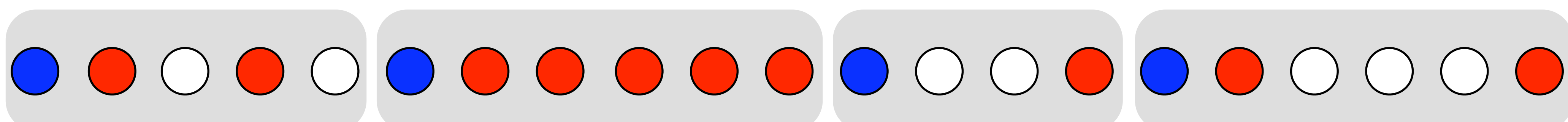
This is an LTL-relabeling (thanks to the induction assumption on a smaller alphabet).

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some ● $\in A$ such that $s=\alpha($●$)$ satisfies $sS \not\subset S$ or $Ss \not\subset S$



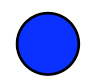
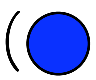
$s_1 \in sS$ $s_2 \in sS$ $s_3 \in sS$ $s_4 \in sS$

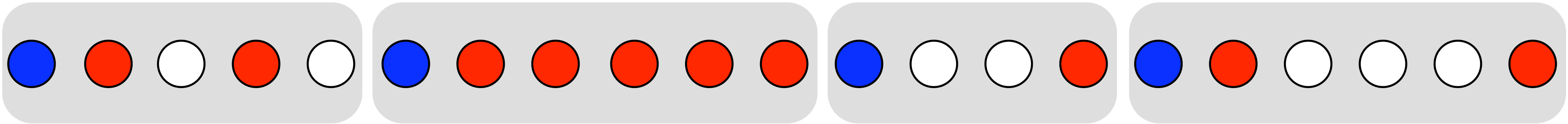
This is an LTL-relabeling (thanks to the induction assumption on a smaller alphabet).
A word has the same value as its relabeling.

Prop. Let $\alpha : A^+ \to S$ be a semigroup morphism, with S aperiodic.
For every $s \in S$, the language $\alpha^{-1}(s)$ is LTL definable.

Proof. Induction on size of S, then size of A.

Induction base, when S has one element: the inverse image is A^+

Claim.
Since S is aperiodic, there must be some  $\in A$ such that $s=\alpha($$)$ satisfies $sS \not\subset S$ or $Ss \not\subset S$



$s_1 \in sS$ $s_2 \in sS$ $s_3 \in sS$ $s_4 \in sS$

This is an LTL-relabeling (thanks to the induction assumption on a smaller alphabet).
A word has the same value as its relabeling.
After the relabeling, we can use the smaller semigroup sS.

Thm. Emptiness for LTL is PSPACE-complete.

Thm. Emptiness for LTL is PSPACE-complete.

Upper bound.
Compile into an alternating automaton, determinize, check for emptiness.

Thm. Emptiness for LTL is PSPACE-complete.

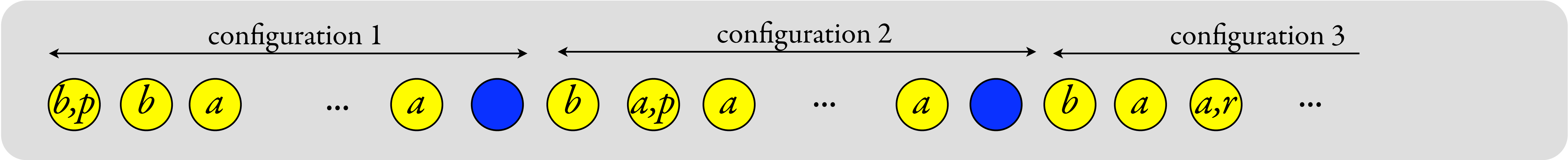
Upper bound.
Compile into an alternating automaton, determinize, check for emptiness.

Lower bound.
LTL has negation, so emptiness is same problem as universality, which is PSPACE-hard.

Thm. Emptiness for LTL is PSPACE-complete.

Upper bound.
Compile into an alternating automaton, determinize, check for emptiness.

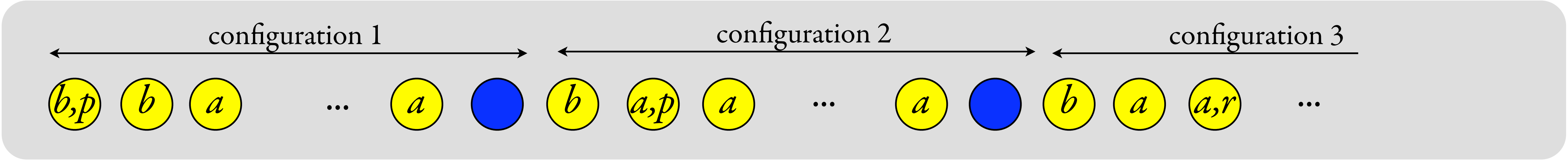
Lower bound.
LTL has negation, so emptiness is same problem as universality, which is PSPACE-hard.



Thm. Emptiness for LTL is PSPACE-complete.

Upper bound.
Compile into an alternating automaton, determinize, check for emptiness.

Lower bound.
LTL has negation, so emptiness is same problem as universality, which is PSPACE-hard.



"some position has a different label than its n-fold successor."

Temporal Logic for Words

definition
the virtuous cycle
MSO=regular

Temporal Logic for Trees

definition
CTL, PDL, CTL*
expressivity

XPath

definition
two-variable logic
regular XPath

Temporal Logic for Words

- definition
- the virtuous cycle
- MSO=regular

Temporal Logic for Trees

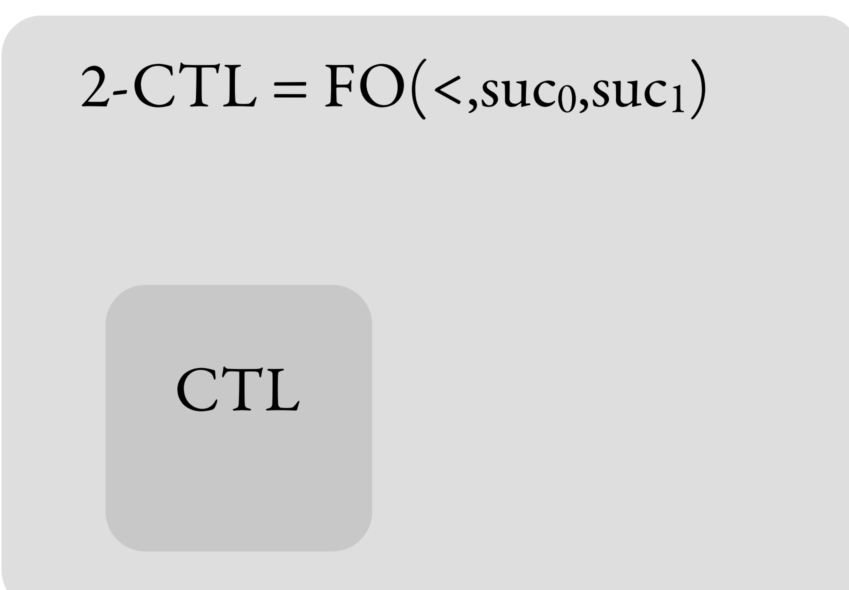
- definition
- CTL, PDL, CTL*
- expressivity

XPath

- definition
- two-variable logic
- regular XPath

Temporal Logic for Trees

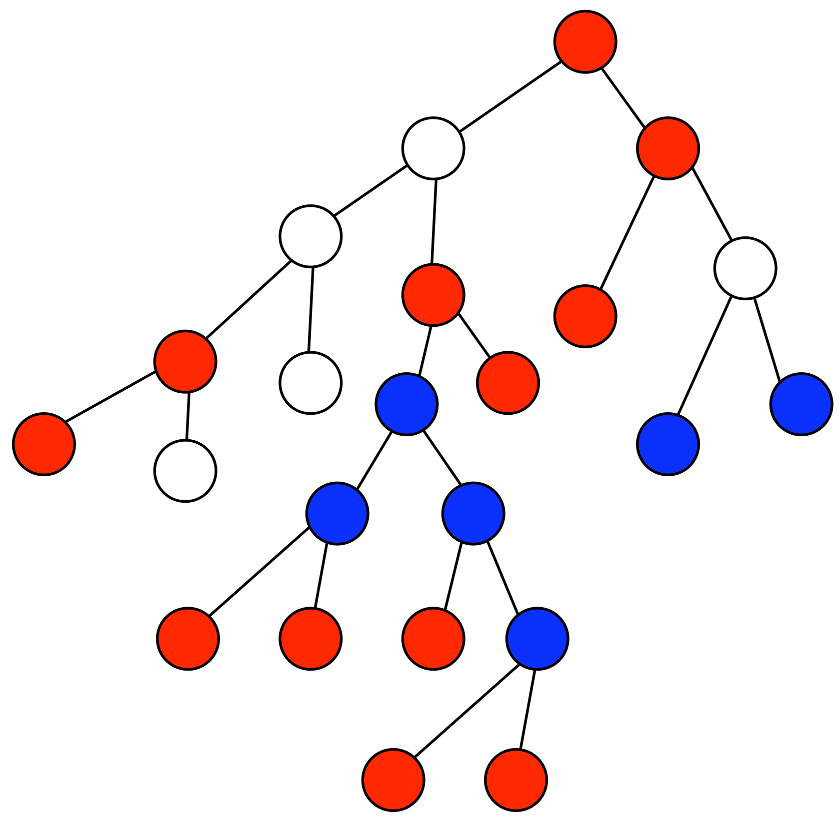
first approach: **CTL**



Emptiness is EXPTIME-complete for both CTL and 2CTL, but model checking is linear time (formula times tree).

Temporal Logic for Trees

first approach: **CTL**



2-CTL = FO($<$,suc_0,suc_1)

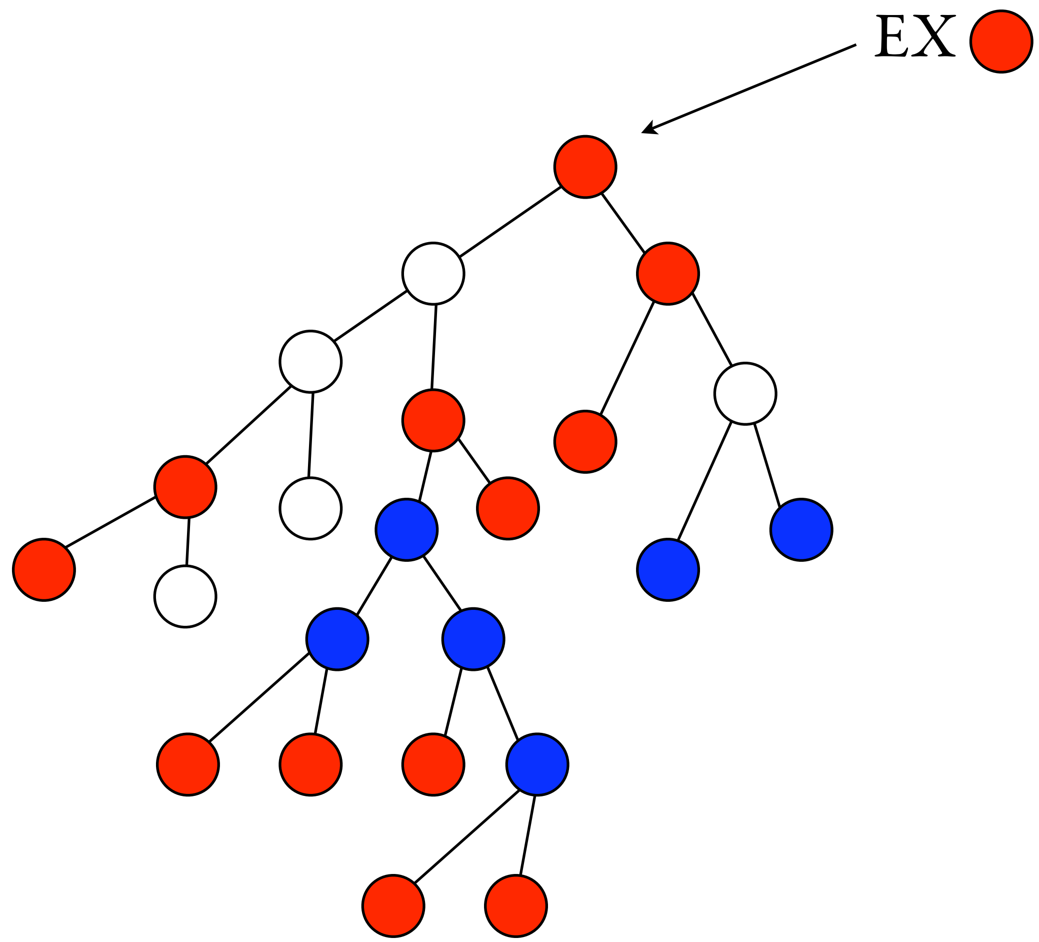
CTL

Emptiness is EXPTIME-complete for both CTL and 2CTL,
but model checking is linear time (formula times tree).

Temporal Logic for Trees

EXISTS NEXT

first approach: **CTL**



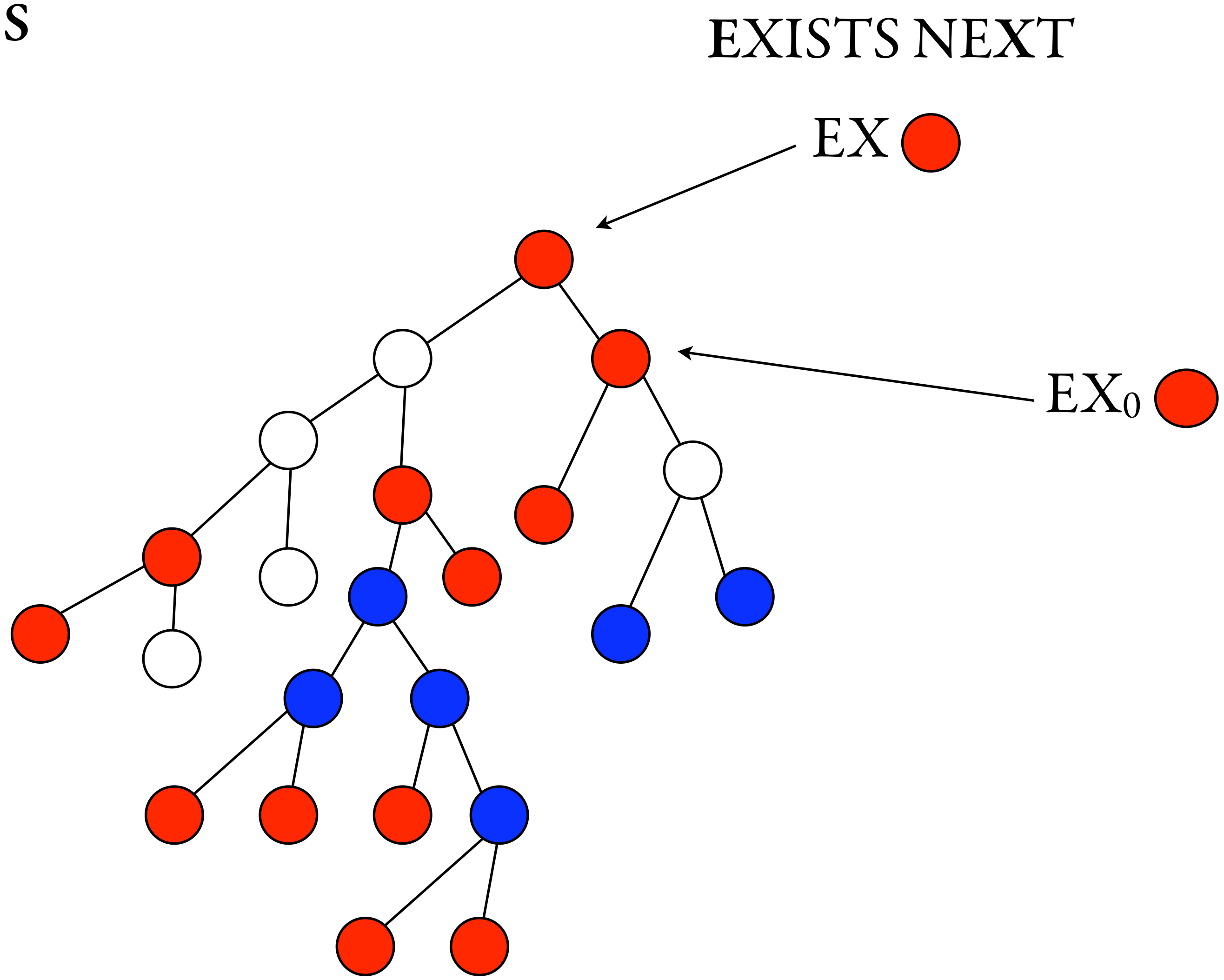
2-CTL = FO($<$,suc_0,suc_1)

CTL

Emptiness is EXPTIME-complete for both CTL and 2CTL, but model checking is linear time (formula times tree).

Temporal Logic for Trees

first approach: **CTL**



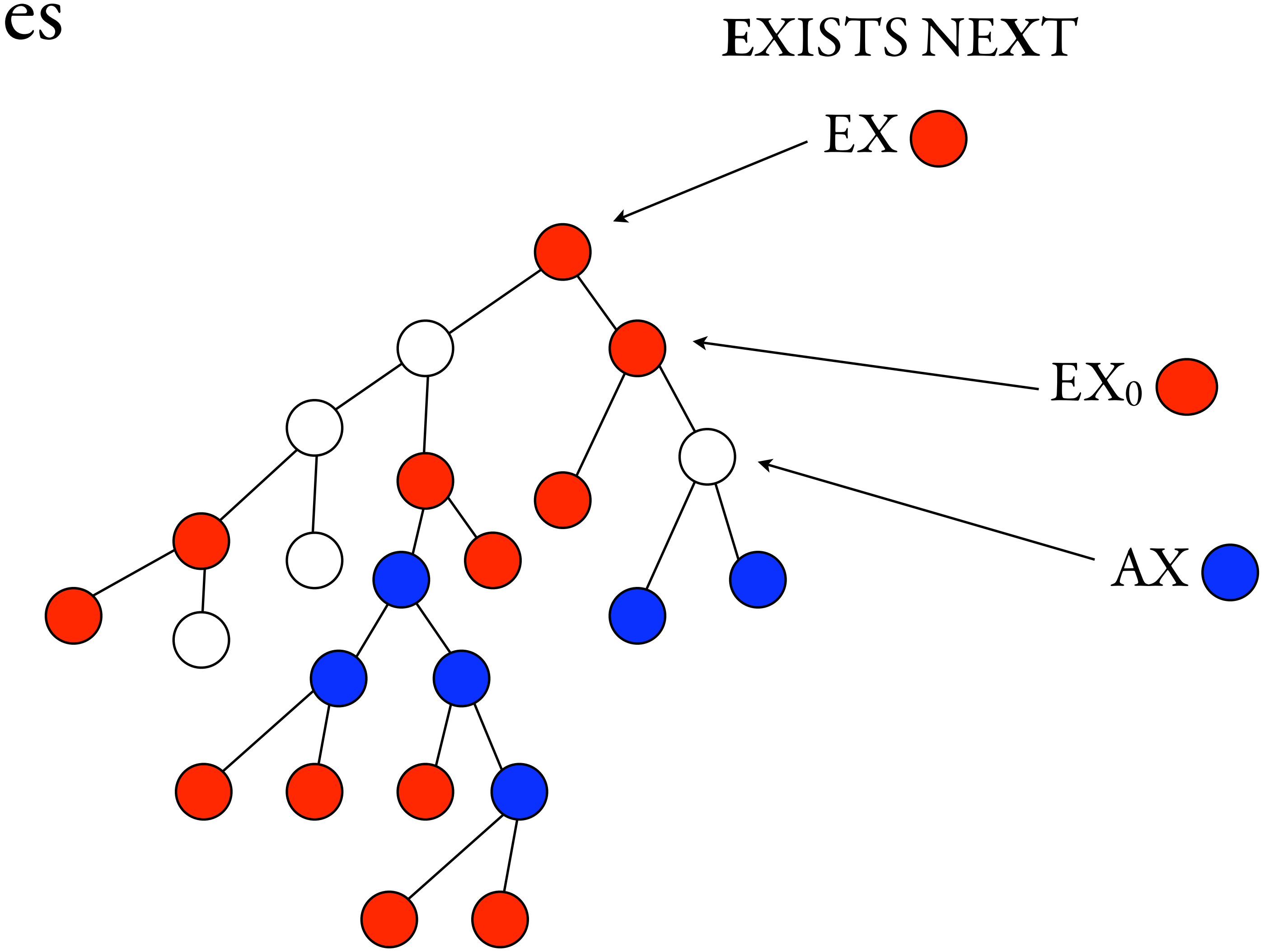
2-CTL = FO(<,suc$_0$,suc$_1$)

CTL

Emptiness is EXPTIME-complete for both CTL and 2CTL, but model checking is linear time (formula times tree).

Temporal Logic for Trees

first approach: **CTL**



2-CTL = FO(<,suc_0,suc_1)

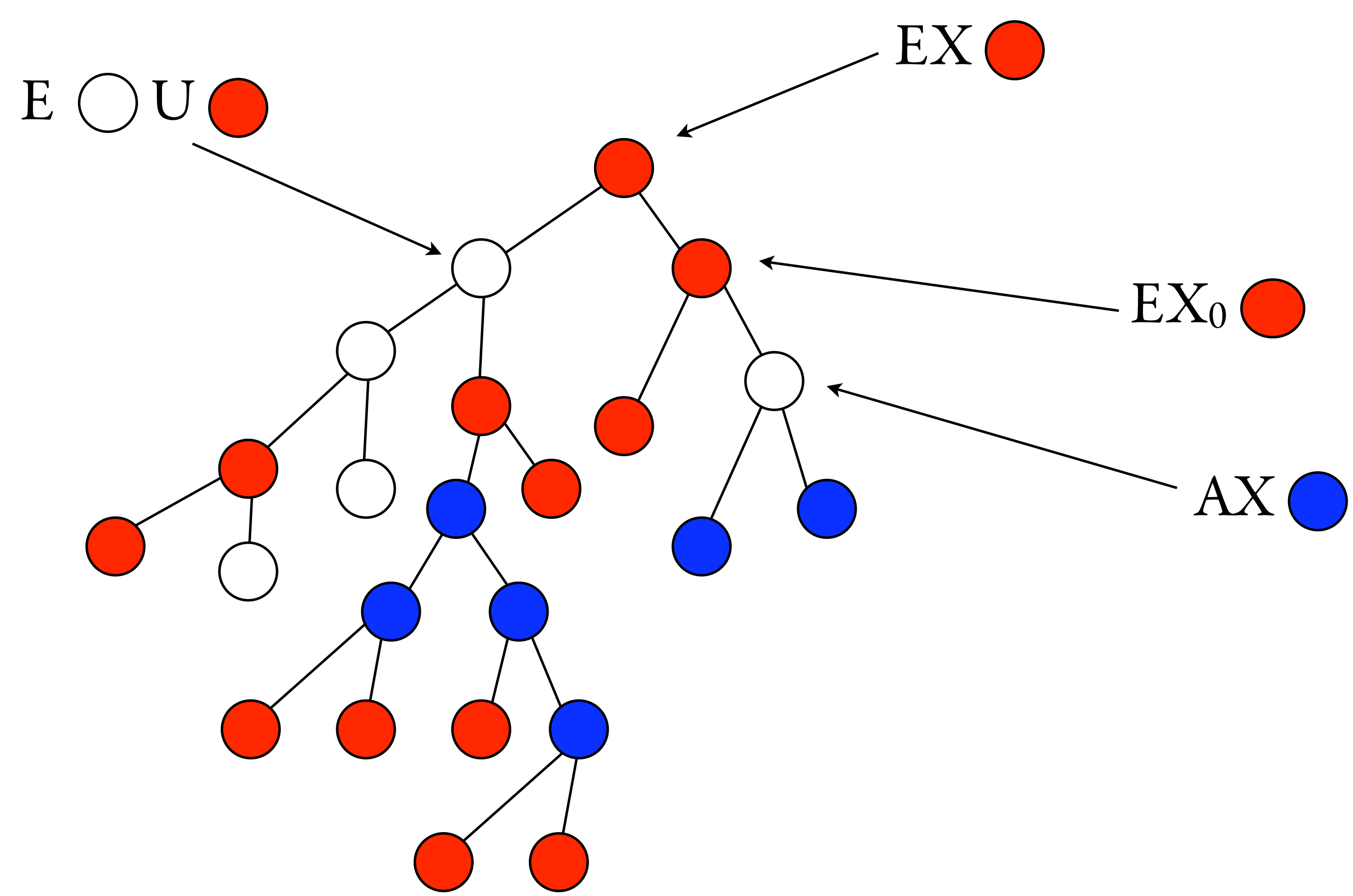
CTL

Emptiness is EXPTIME-complete for both CTL and 2CTL, but model checking is linear time (formula times tree).

Temporal Logic for Trees

EXISTS NEXT

first approach: **CTL**



2-CTL = FO(<,suc_0,suc_1)

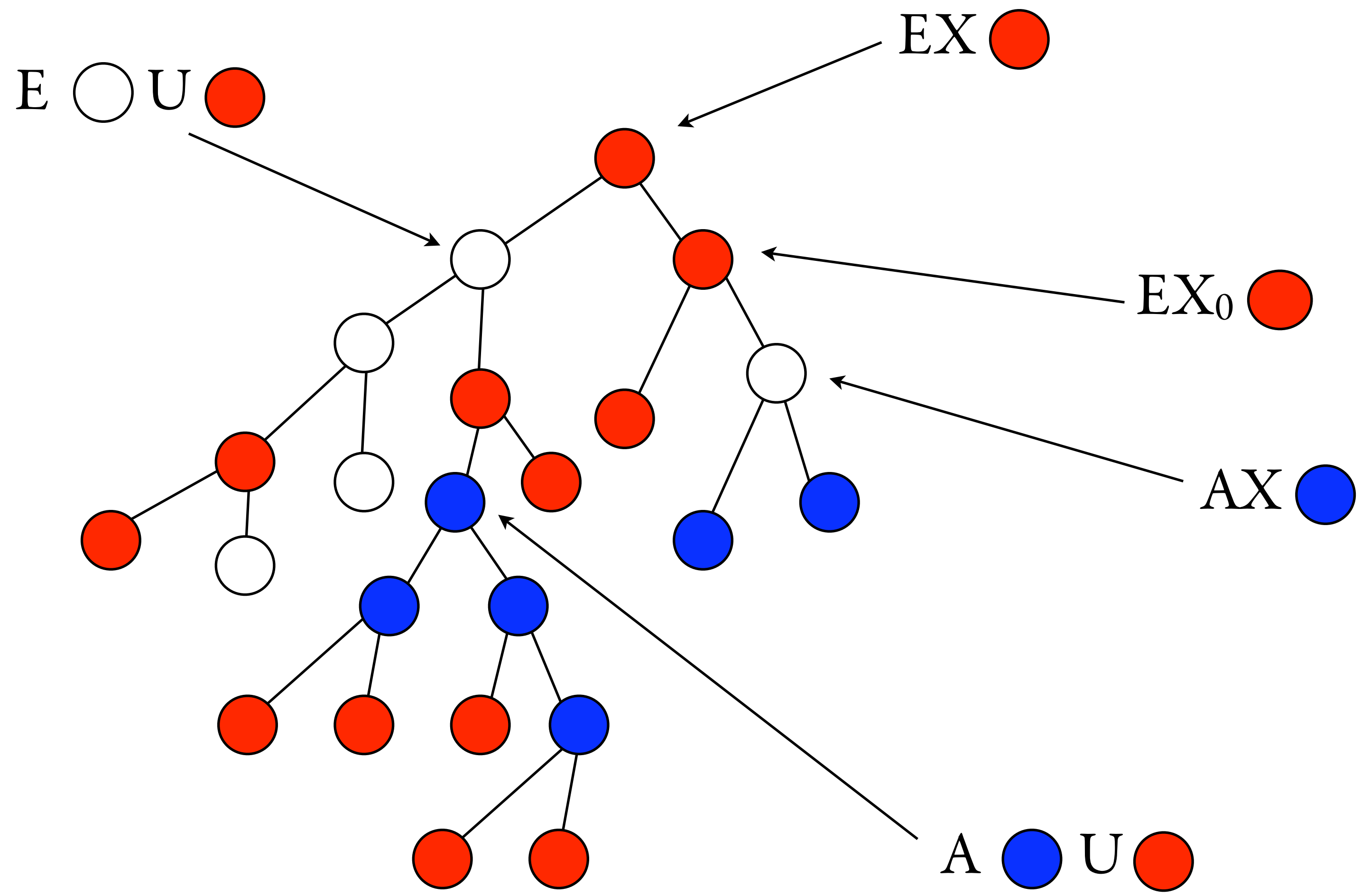
CTL

Emptiness is EXPTIME-complete for both CTL and 2CTL, but model checking is linear time (formula times tree).

Temporal Logic for Trees

EXISTS NEXT

first approach: **CTL**



2-CTL = FO(<,suc_0,suc_1)

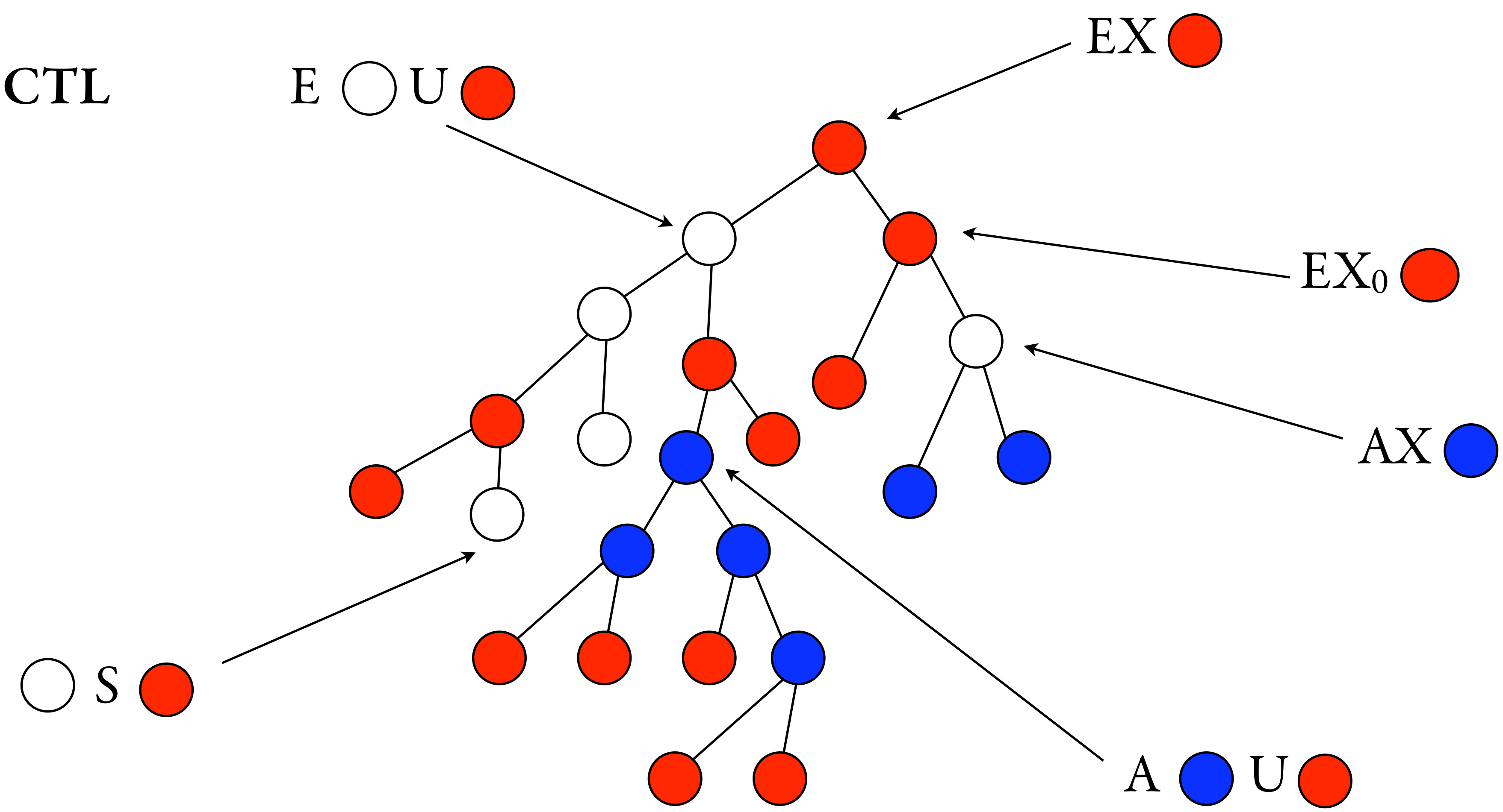
CTL

Emptiness is EXPTIME-complete for both CTL and 2CTL, but model checking is linear time (formula times tree).

Temporal Logic for Trees

EXISTS NEXT

first approach: **CTL**

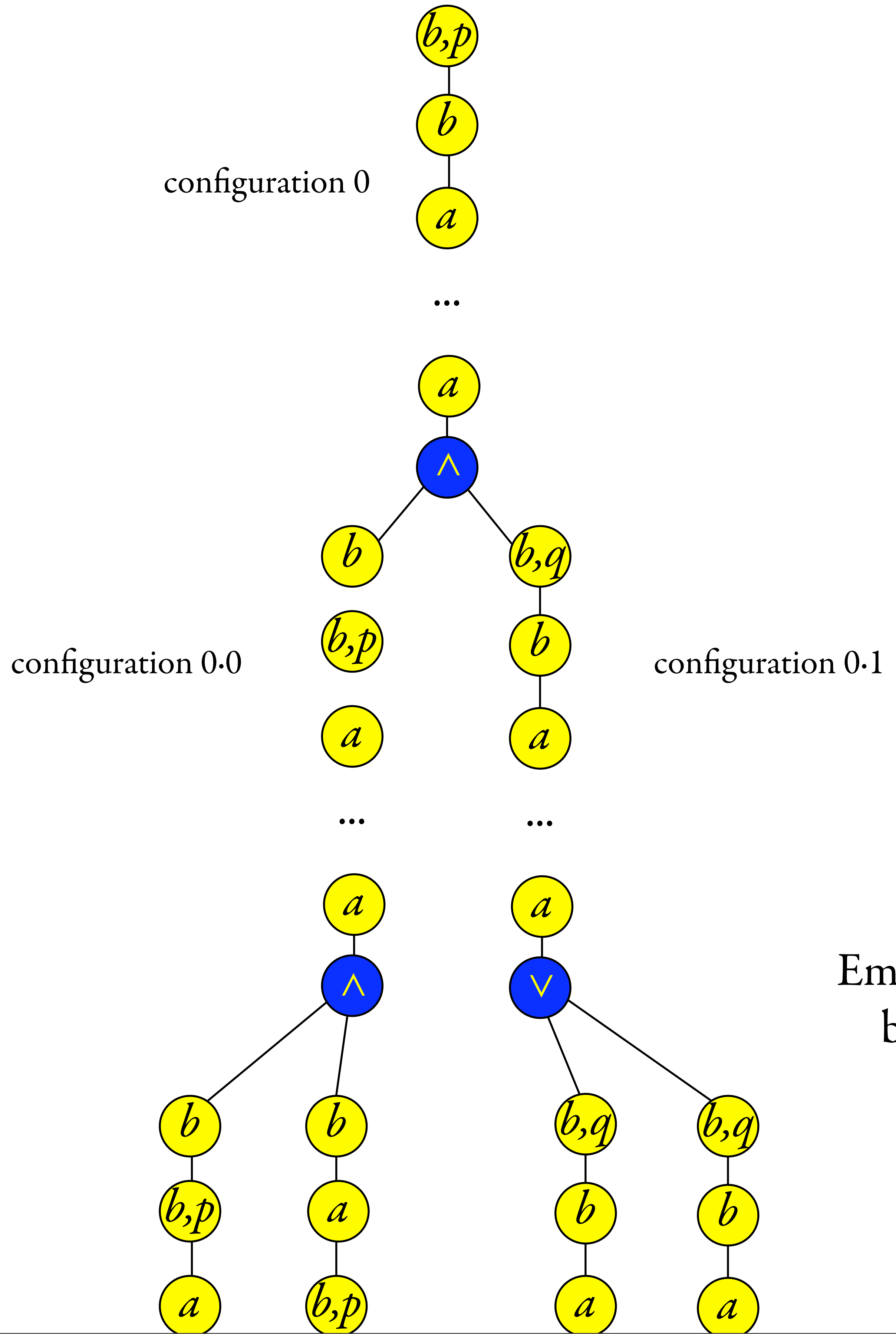


2-CTL = FO(<,suc_0,suc_1)

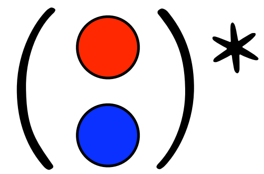
CTL

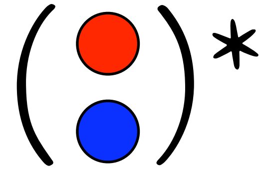
Emptiness is EXPTIME-complete for both CTL and 2CTL, but model checking is linear time (formula times tree).

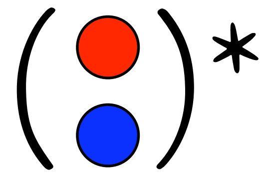
Emptiness is EXPTIME-complete for both CTL and 2CTL,
but model checking is linear time (formula times tree).

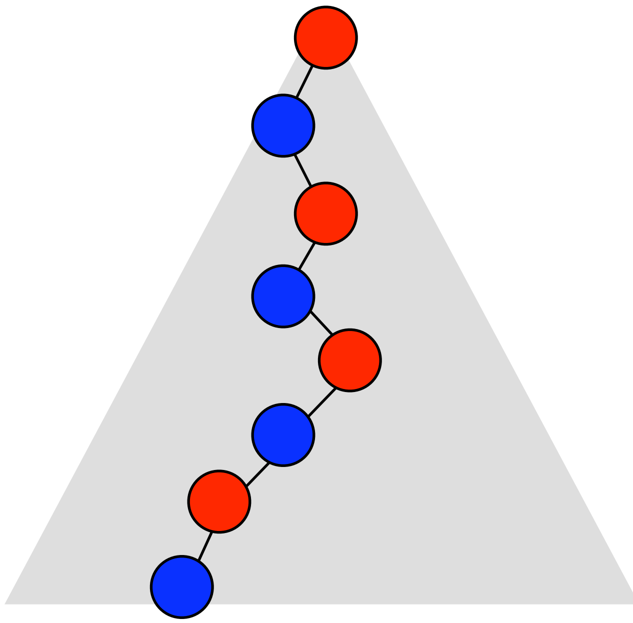


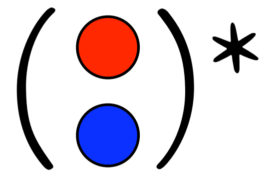
Emptiness is EXPTIME-complete for both CTL and 2CTL, but model checking is linear time (formula times tree).

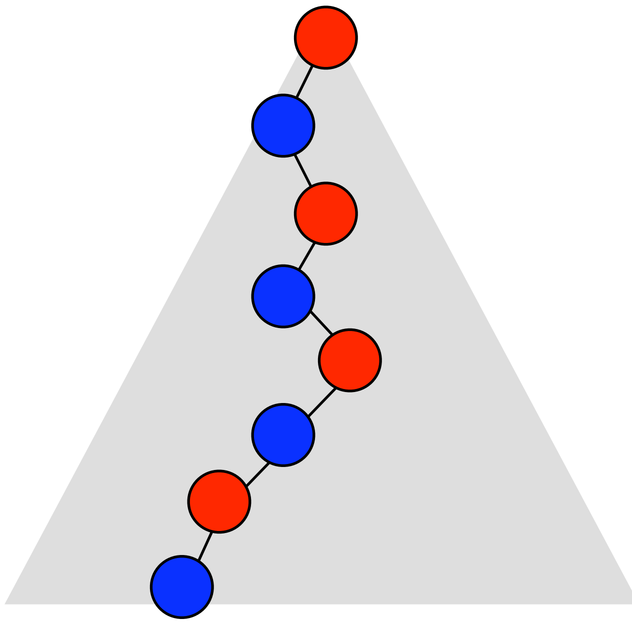
L = some (maximal) path in 

L = some (maximal) path in 

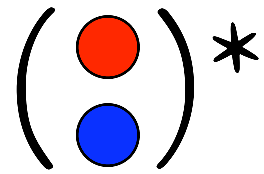
L = some (maximal) path in 

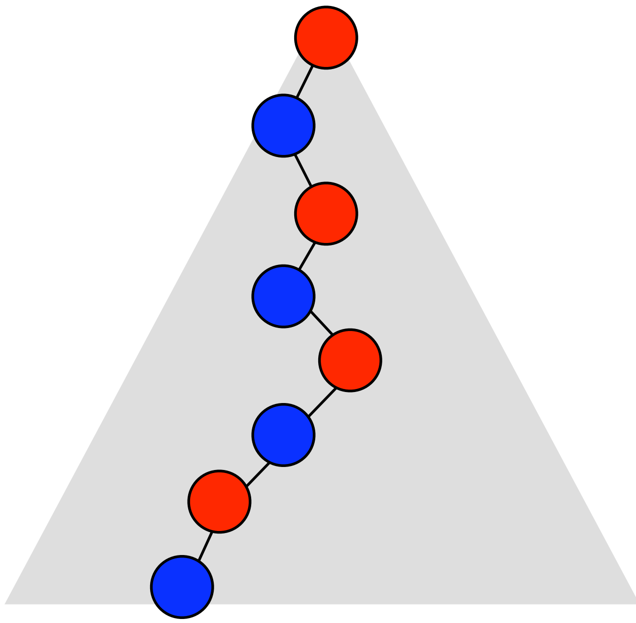


L = some (maximal) path in 



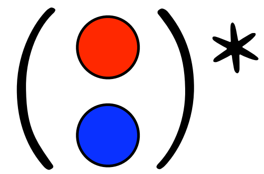
Claim.
$L \in$ 2CTL but $L \notin$ CTL

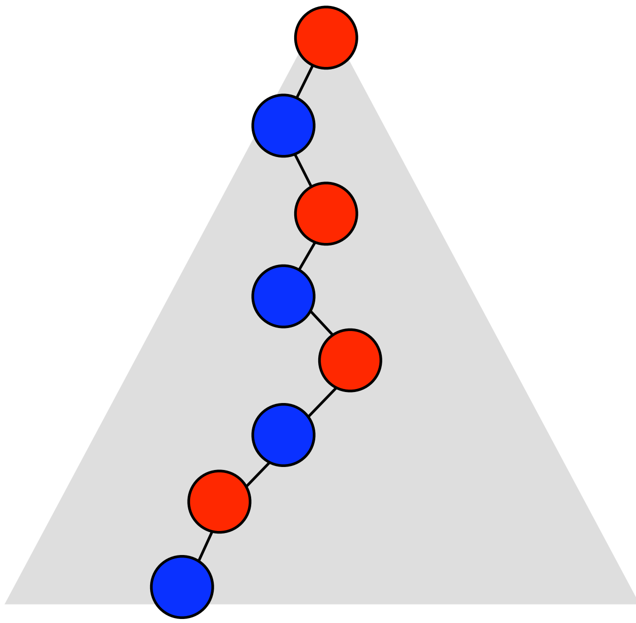
L = some (maximal) path in 



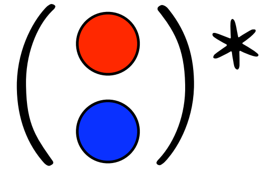
Claim.
$L \in$ 2CTL but $L \notin$ CTL

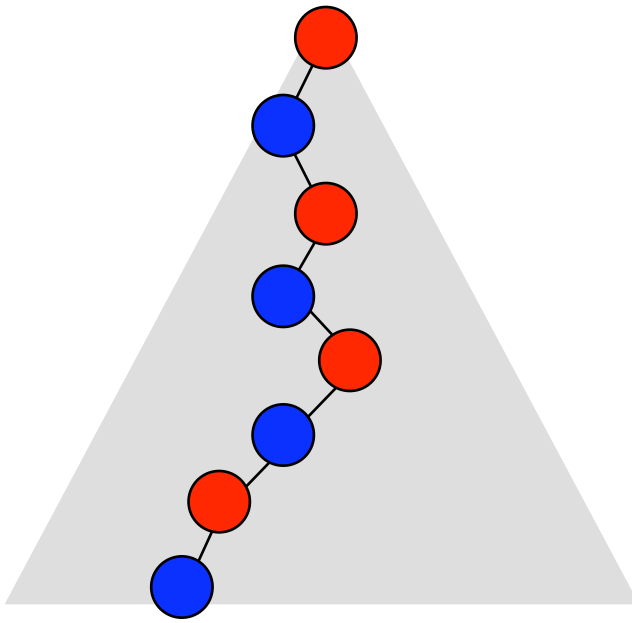
there is a leaf
such that every ancestor has a parent, and vice versa.

L = some (maximal) path in 



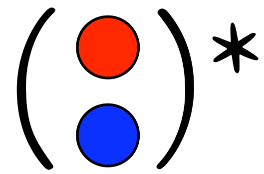
Claim.
$L \in$ 2CTL but $L \notin$ CTL

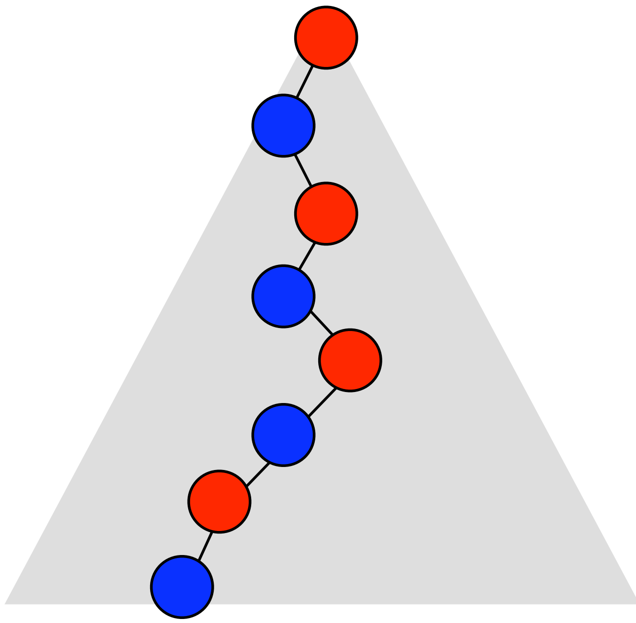
L = some (maximal) path in 



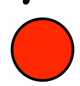
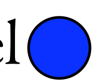
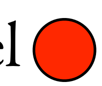
Claim.
$L \in 2\mathrm{CTL}$ but $L \notin \mathrm{CTL}$

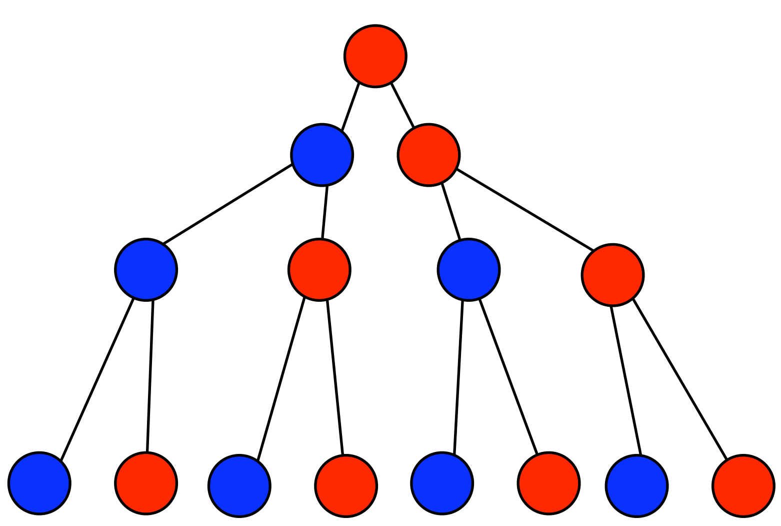
$t_n =$ Complete binary tree of depth n,
with root label ,
where left children have label and
right children have label .

L = some (maximal) path in 

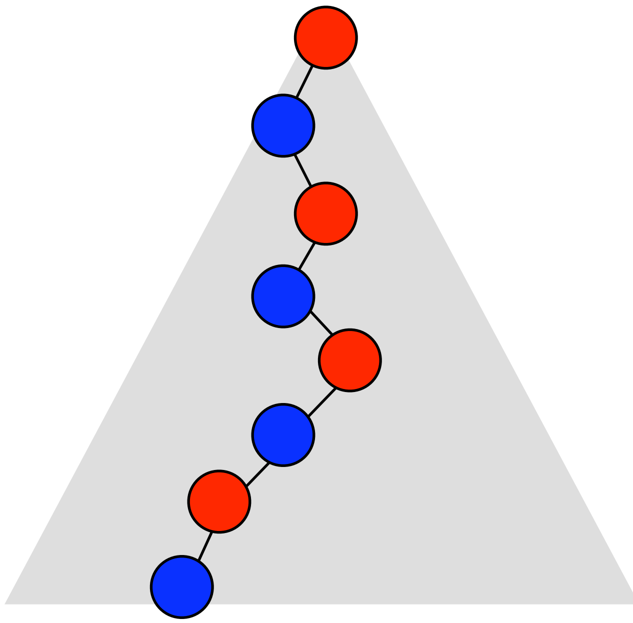


Claim.
$L \in$ 2CTL but $L \notin$ CTL

$t_n =$ Complete binary tree of depth n,
with root label  ,
where left children have label  and
right children have label  .

$t_4 =$ 

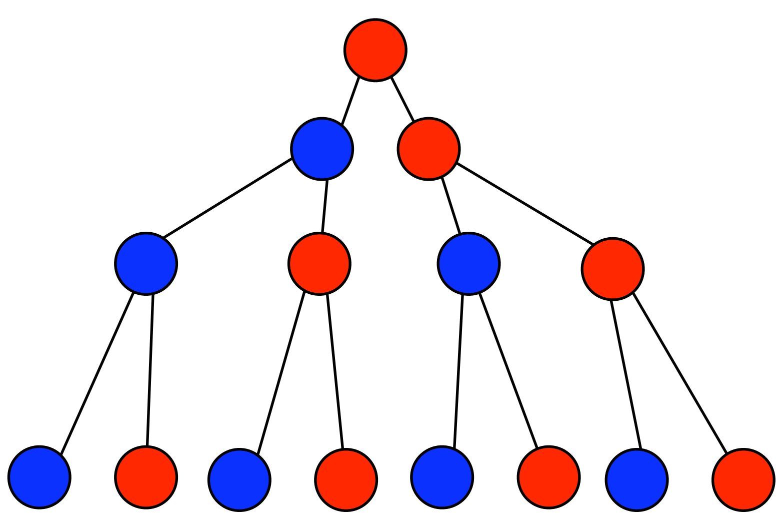
L = some (maximal) path in $(\quad)^*$



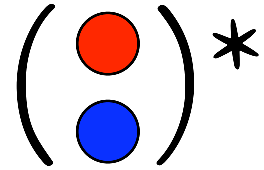
Claim.
$L \in$ 2CTL   but $L \notin$ CTL

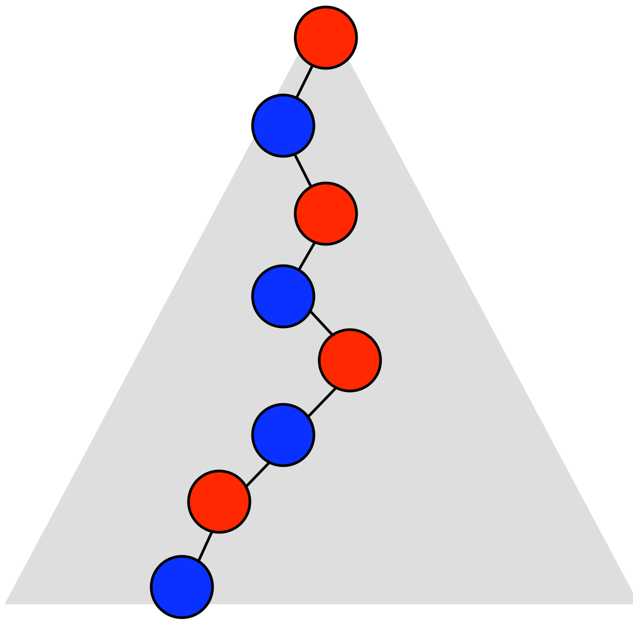
$t_n =$ Complete binary tree of depth n,
with root label ,
where left children have label and
right children have label .

$t_4 =$

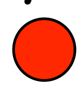
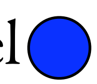
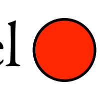


A CTL formula of depth n cannot distinguish t_{n+1} and t_{n+2}.

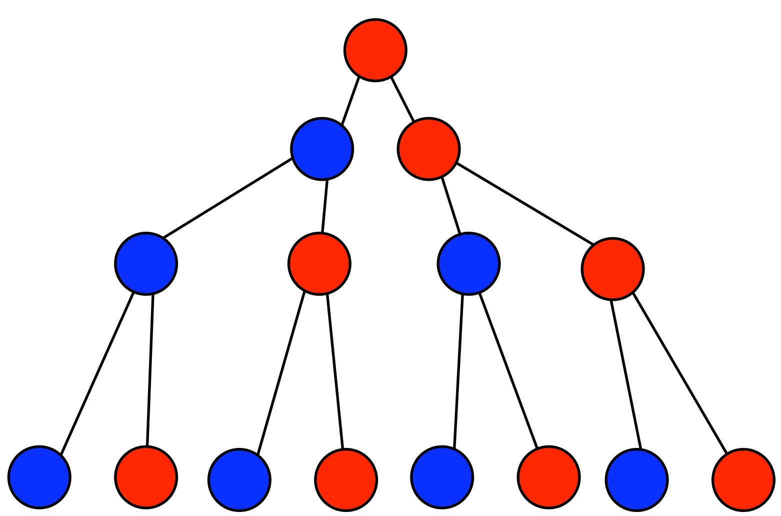
L = some (maximal) path in 



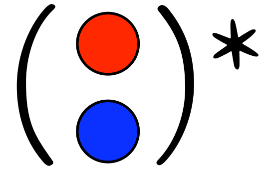
Claim.
$L \in$ 2CTL   but $L \notin$ CTL

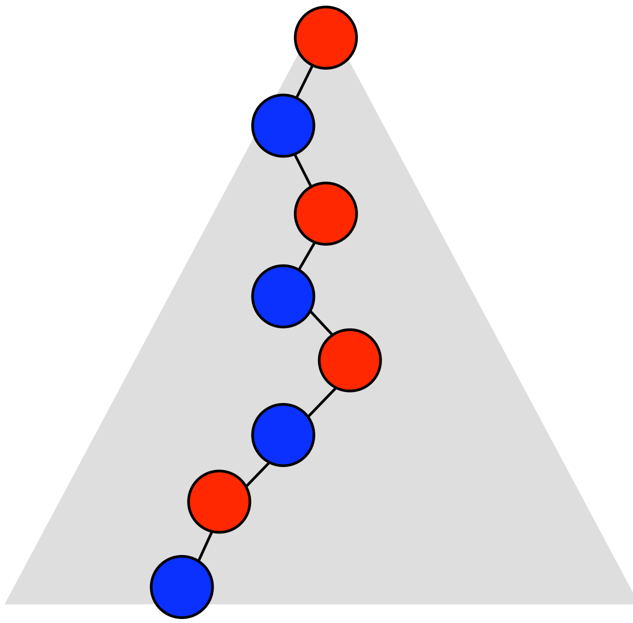
$t_n =$ Complete binary tree of depth n,
with root label  ,
where left children have label  and
right children have label  .

$t_n \in L$ iff n is even.

$t_4 =$ 

A CTL formula of depth n cannot distinguish t_{n+1} and t_{n+2}.

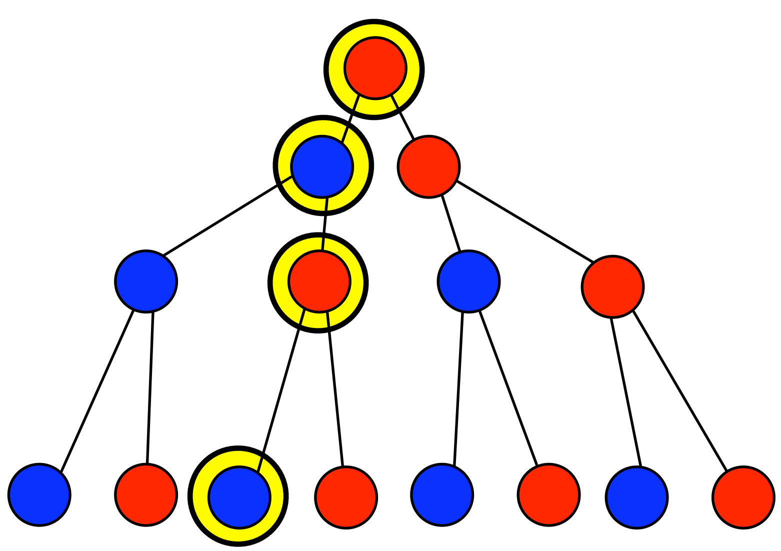
L = some (maximal) path in 



Claim.
$L \in 2\mathrm{CTL}$ but $L \notin \mathrm{CTL}$

$t_n =$ Complete binary tree of depth n,
with root label ,
where left children have label and
right children have label .

$t_n \in L$ iff n is even.

$t_4 =$ 

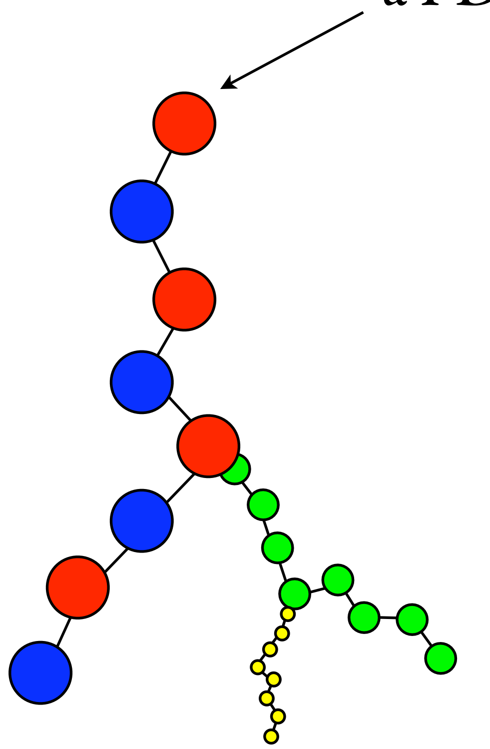
A CTL formula of depth n cannot distinguish t_{n+1} and t_{n+2}.

PDL

label tests, boolean combinations

If $\Phi_1,\ldots,\Phi_n$ are formulas of PDL, and $L \subseteq \{\Phi_1,\ldots,\Phi_n\}^*$ is a regular word language, then "exists a path in L", written $\mathbf{E}L$, is a formula of PDL

a PDL formula



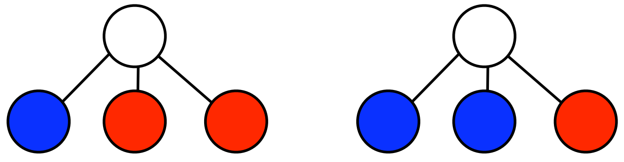
CTL*

fragment of PDL where the word language L must be first-order definable.
Usually L is written in LTL.

How to tell a right child from a left child?
add a formula: "I am a left child"

Thm. (Hafer, Thomas `87)
Over binary trees, CTL* (with left/right child) has the same expressive power as FO($<$,suc_0,suc_1).

Without left/right child, CTL* has the same expressive power as FO($<$), but only for binary trees (and not ternary ones).

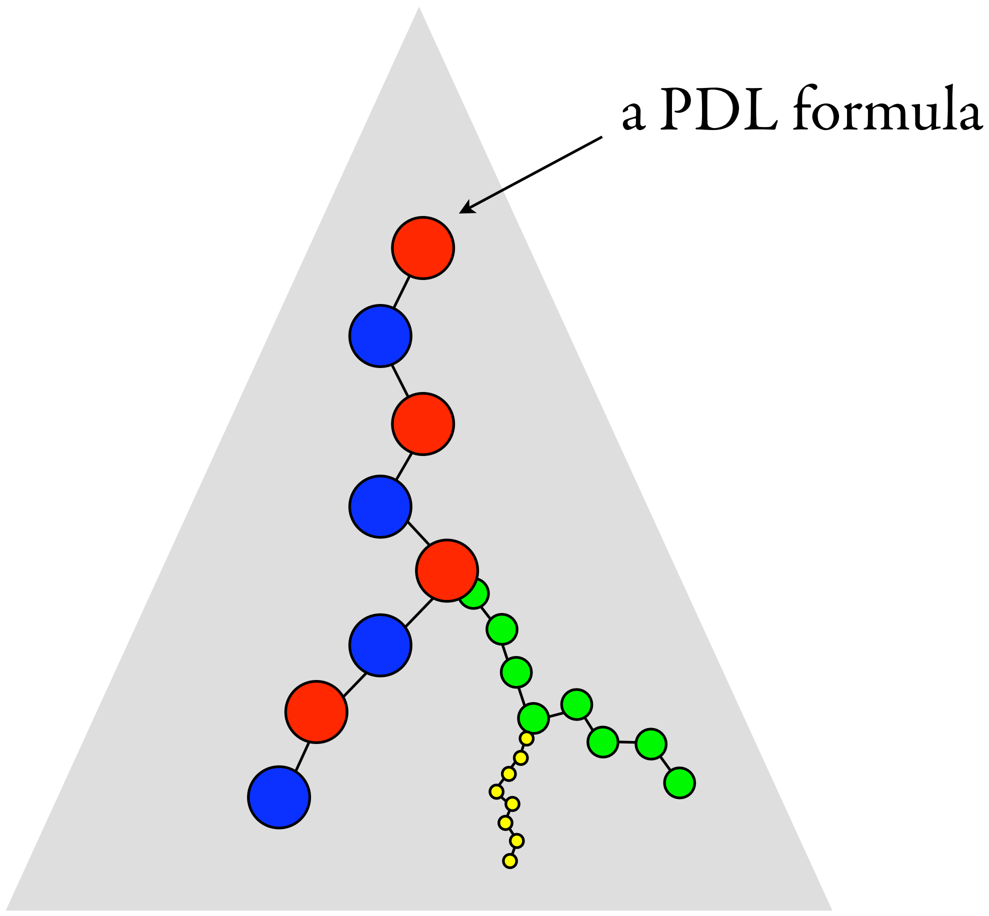


look the same

PDL

label tests, boolean combinations

If $\Phi_1,\ldots,\Phi_n$ are formulas of PDL, and $L \subseteq \{\Phi_1,\ldots,\Phi_n\}^*$ is a regular word language, then "exists a path in L", written **E**L, is a formula of PDL

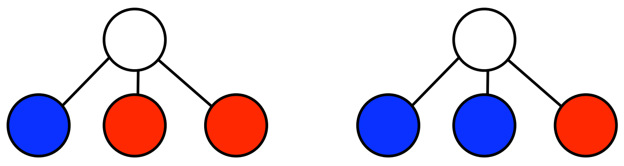


CTL*

fragment of PDL where the word language L must be first-order definable.
Usually L is written in LTL.

How to tell a right child from a left child?
add a formula: "I am a left child"

Thm. (Hafer, Thomas `87)
Over binary trees, CTL* (with left/right child) has the same expressive power as FO($<$,suc_0,suc_1).



look the same

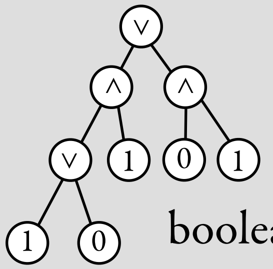
Without left/right child, CTL* has the same expressive power as FO($<$), but only for binary trees (and not ternary ones).

Regular = MSO

Regular = MSO

PDL

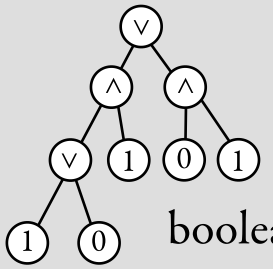
Regular = MSO



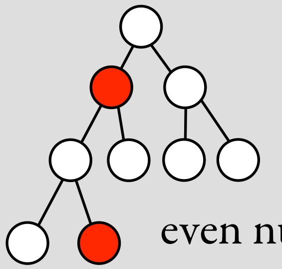
boolean expressions

PDL

Regular = MSO



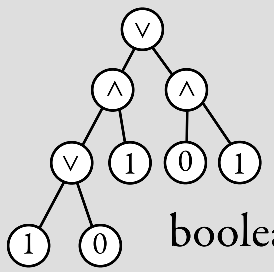
boolean expressions



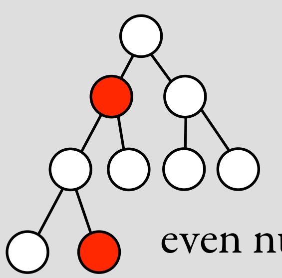
even number of 

PDL

Regular = MSO



boolean expressions

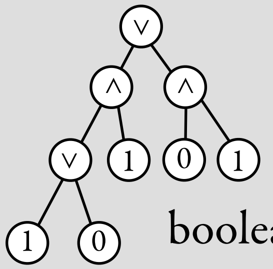


even number of ●

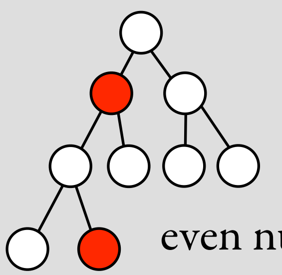
PDL

CTL* = FO(<,suc_0,suc_1) = 2CTL

Regular = MSO

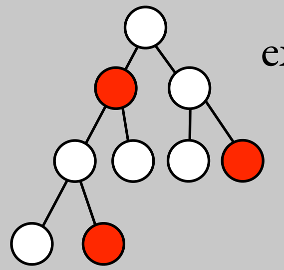


boolean expressions



even number of

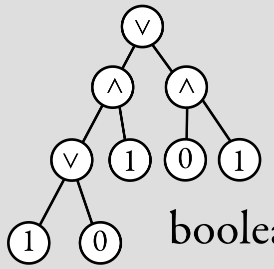
PDL



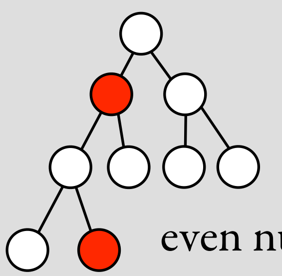
exists a path with even number of

CTL* = FO(<, suc_0, suc_1) = 2CTL

Regular = MSO

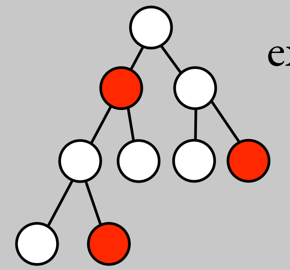


boolean expressions



even number of

PDL

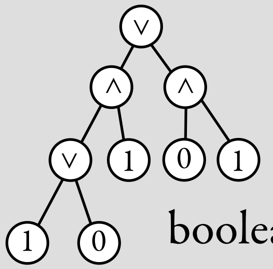


exists a path with even number of

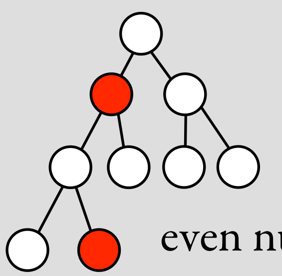
CTL* = FO(<,suc_0,suc_1) = 2CTL

CTL

Regular = MSO

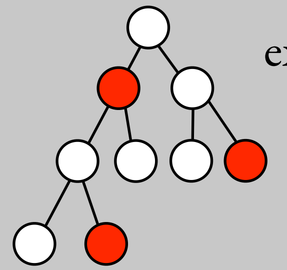


boolean expressions



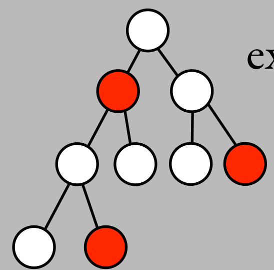
even number of 

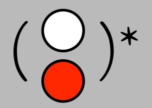
PDL



exists a path with even number of 

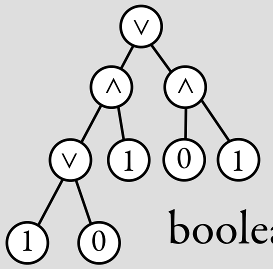
CTL* = FO(<,suc_0,suc_1) = 2CTL



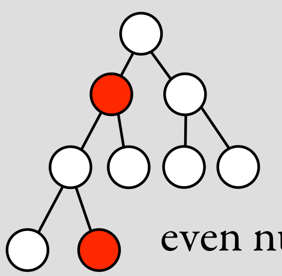
exists a path with 

CTL

Regular = MSO

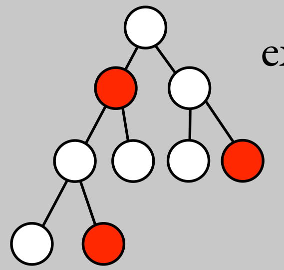


boolean expressions



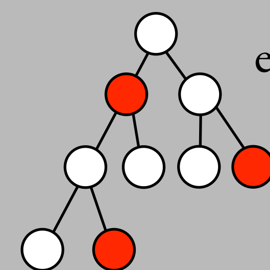
even number of

PDL



exists a path with even number of

CTL* = FO($<$,suc_0,suc_1) = 2CTL

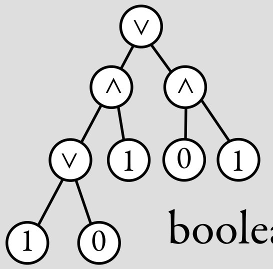


exists a path with ( )*

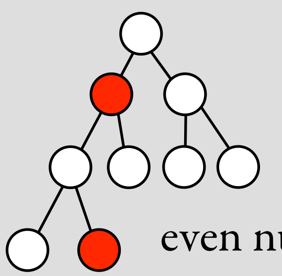
CTL

FO(suc_0,suc_1)

Regular = MSO

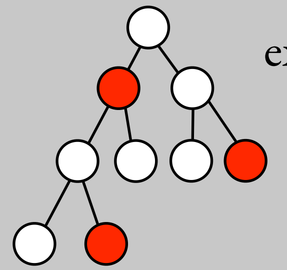


boolean expressions



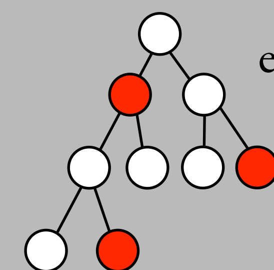
even number of

PDL



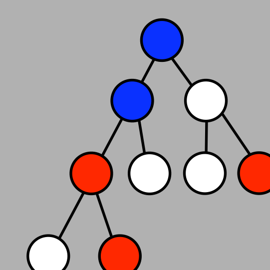
exists a path with even number of

CTL* = FO($<$,suc_0,suc_1) = 2CTL



exists a path with ( )*

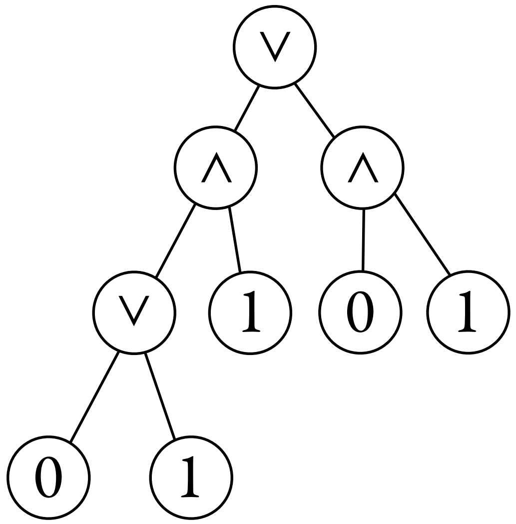
CTL



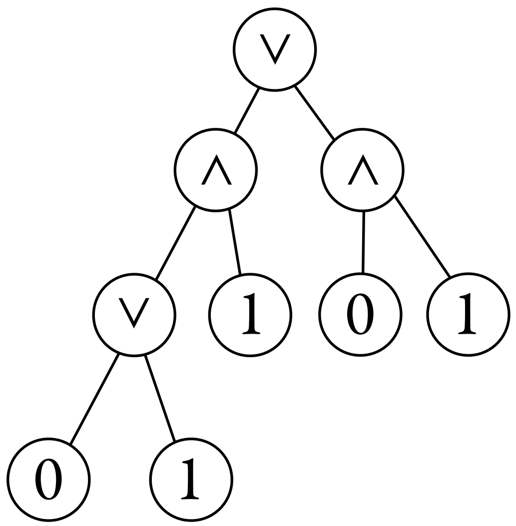
E **U**

FO(suc_0,suc_1)

Why can't you do Boolean expressions in PDL?

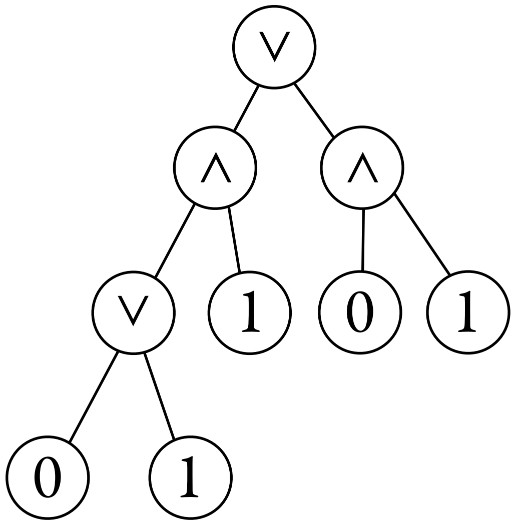


Why can't you do Boolean expressions in PDL?



Induction on nesting depth in formula.
We only do the case of nesting depth 1.

Why can't you do Boolean expressions in PDL?

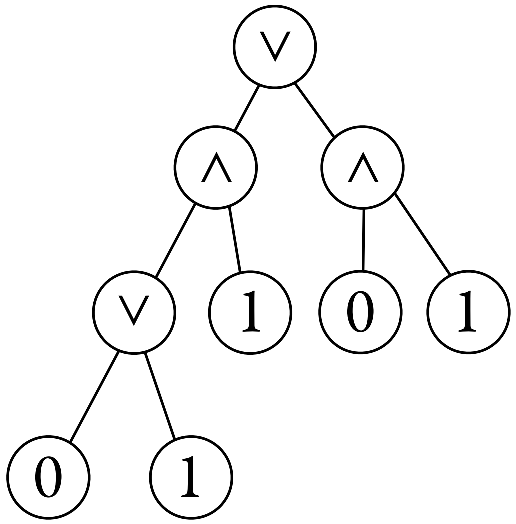


Induction on nesting depth in formula.
We only do the case of nesting depth 1.

Let $L_1,\ldots, L_n$ be regular word languages over $\{\vee, \wedge , 0 , 1\} \times \{\text{left},\text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

Why can't you do Boolean expressions in PDL?

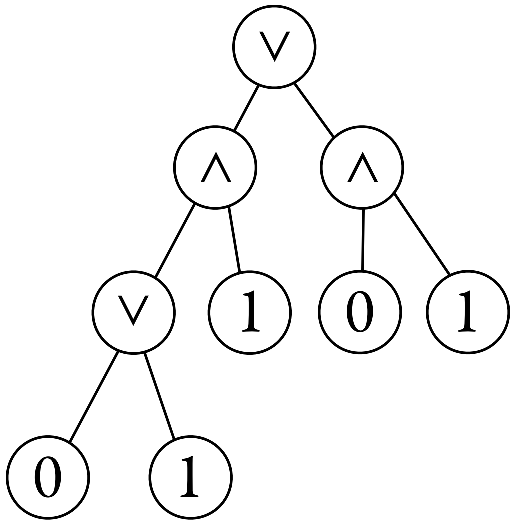
Induction on nesting depth in formula.
We only do the case of nesting depth 1.



Let $L_1, \ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left}, \text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

Let Q be the state space of the automaton recognizing all L_i.

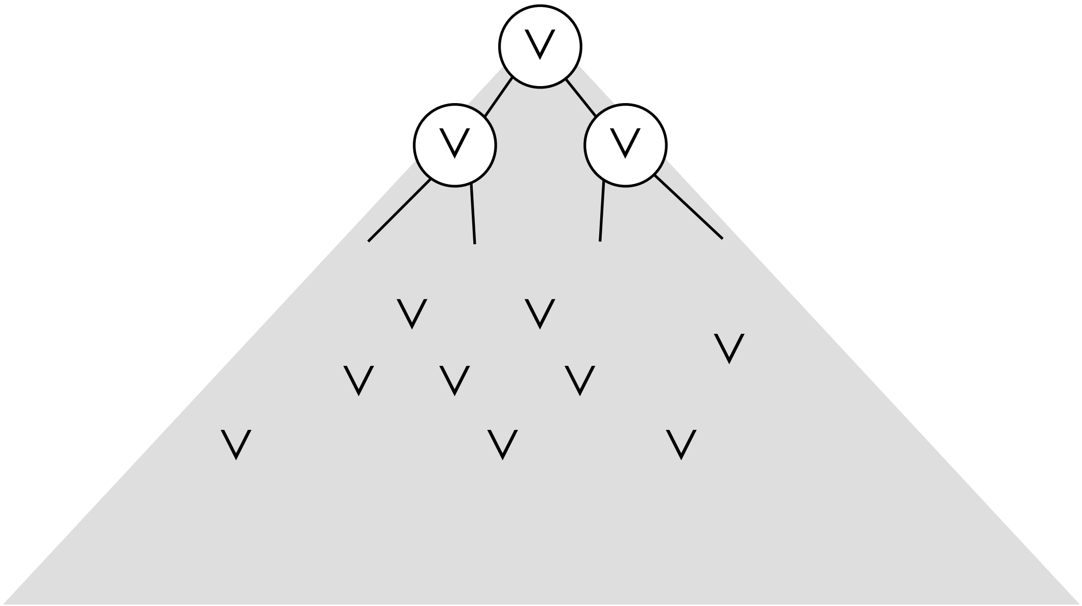
Why can't you do Boolean expressions in PDL?



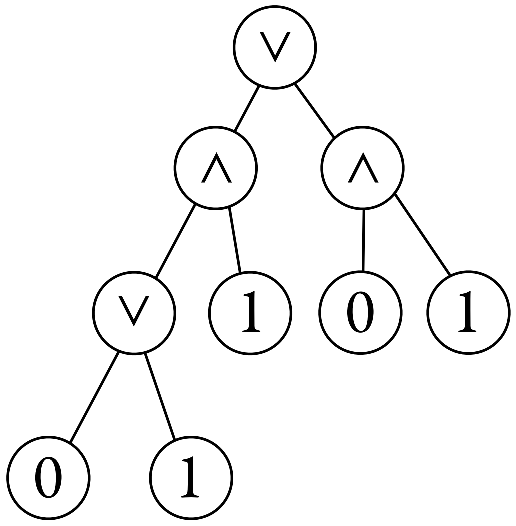
Induction on nesting depth in formula.
We only do the case of nesting depth 1.

Let $L_1, \ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left}, \text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

Let Q be the state space of the automaton recognizing all L_i.



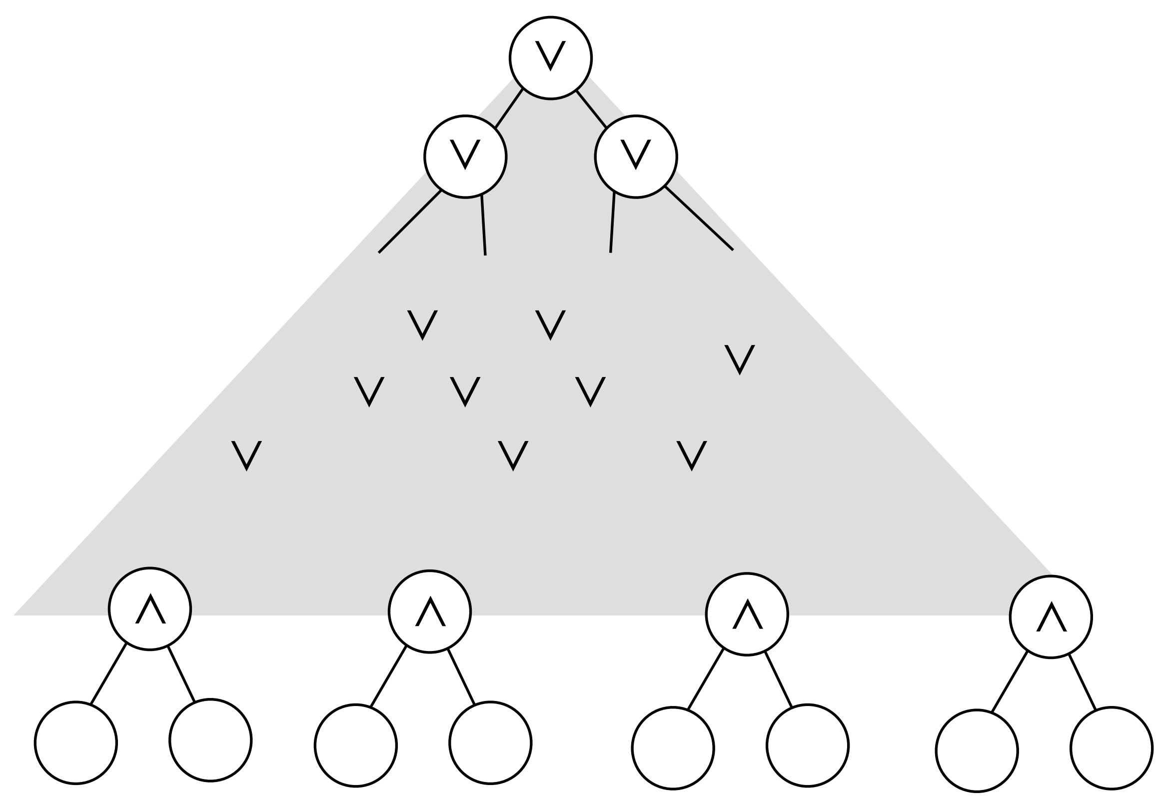
Why can't you do Boolean expressions in PDL?



Induction on nesting depth in formula.
We only do the case of nesting depth 1.

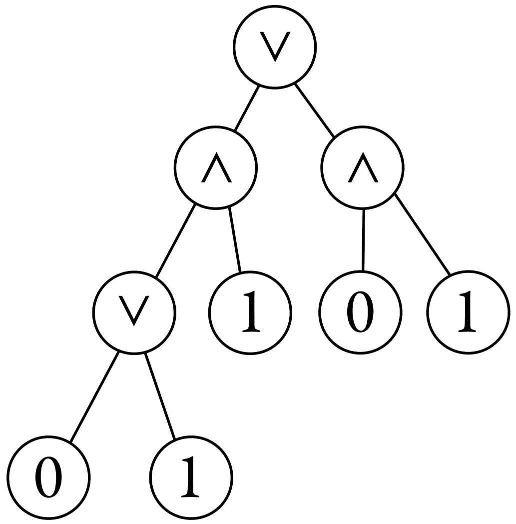
Let $L_1, \ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left}, \text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

Let Q be the state space of the automaton recognizing all L_i.



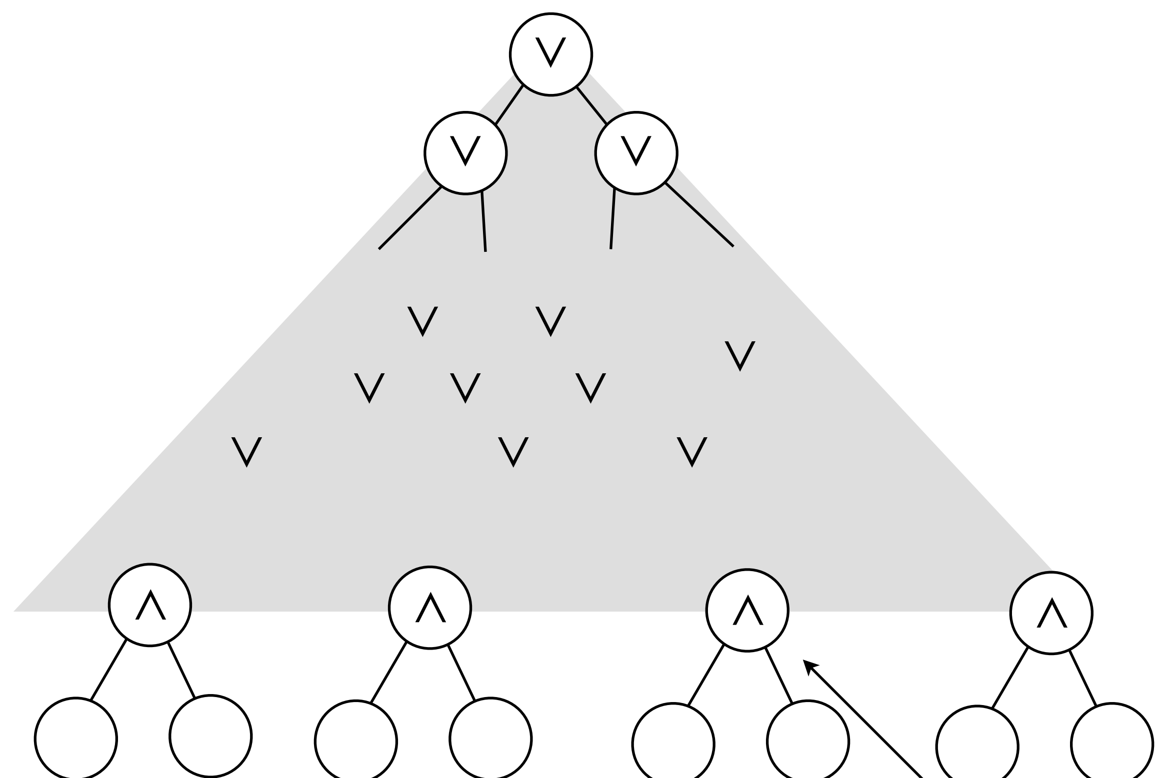
Why can't you do Boolean expressions in PDL?

Induction on nesting depth in formula.
We only do the case of nesting depth 1.



Let $L_1, \ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left}, \text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

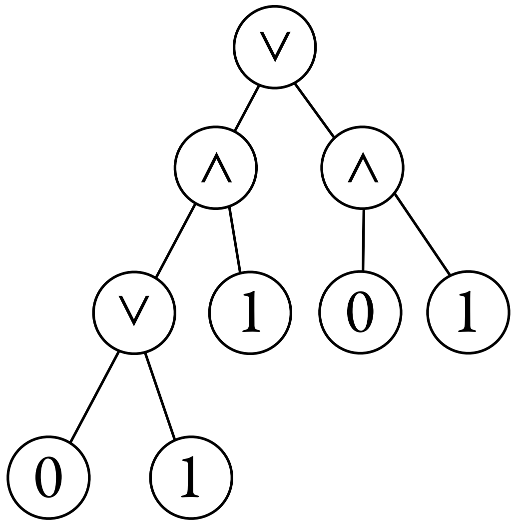
Let Q be the state space of the automaton recognizing all L_i.



with each ∧ node, we associate the state transformation of the path that leads to the root.

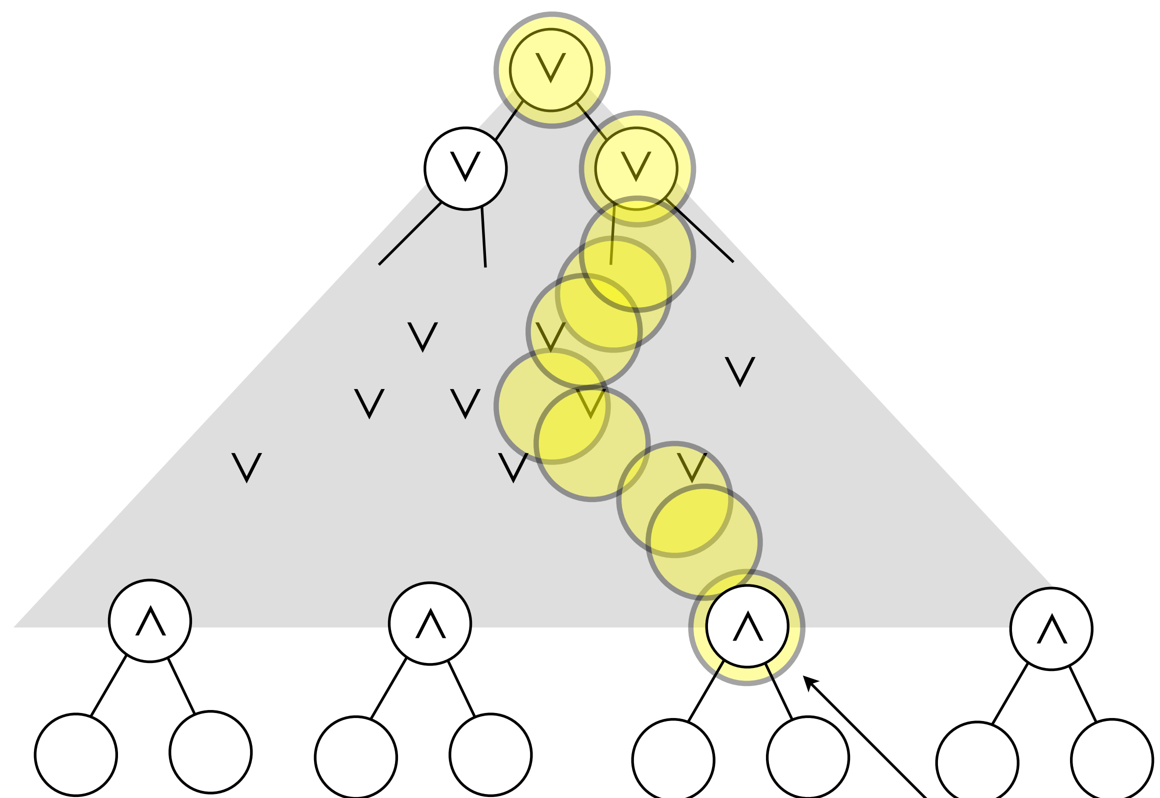
Why can't you do Boolean expressions in PDL?

Induction on nesting depth in formula.
We only do the case of nesting depth 1.



Let $L_1, \ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left}, \text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

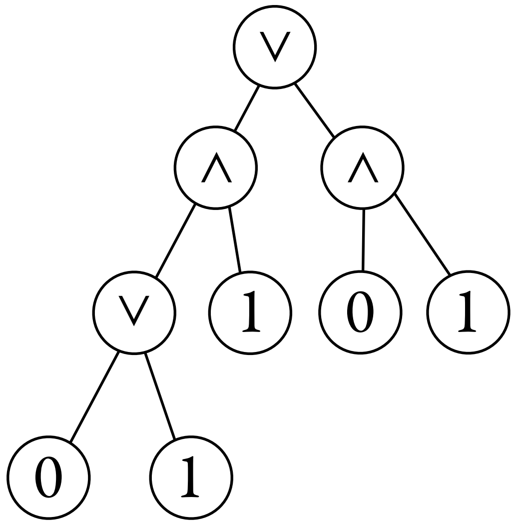
Let Q be the state space of the automaton recognizing all L_i.



with each $\wedge$ node, we associate the state transformation of the path that leads to the root.

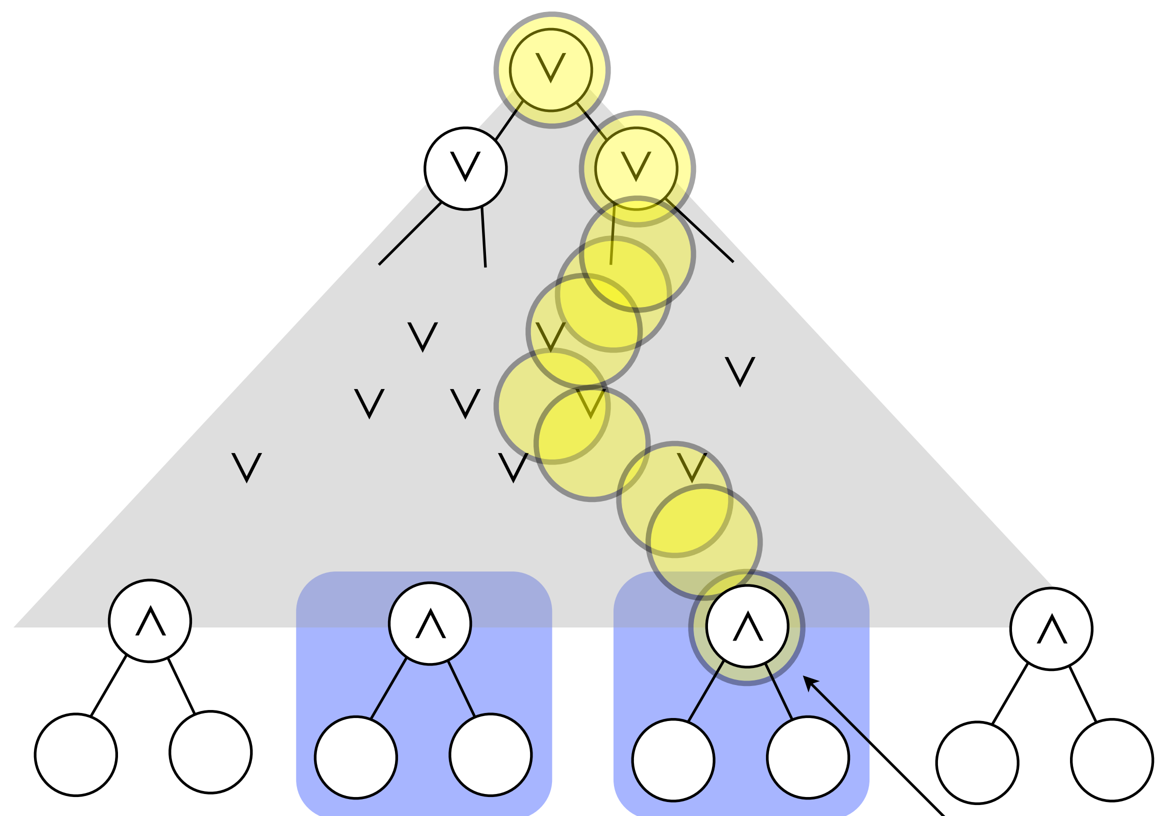
Why can't you do Boolean expressions in PDL?

Induction on nesting depth in formula.
We only do the case of nesting depth 1.



Let $L_1, \ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left},\text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

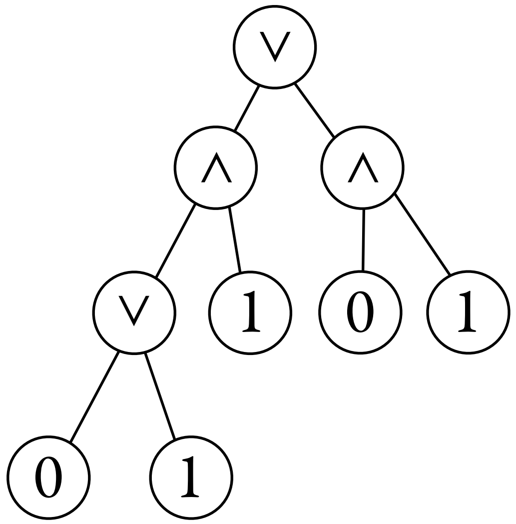
Let Q be the state space of the automaton recognizing all L_i.



with each $\wedge$ node, we associate the state transformation of the path that leads to the root.

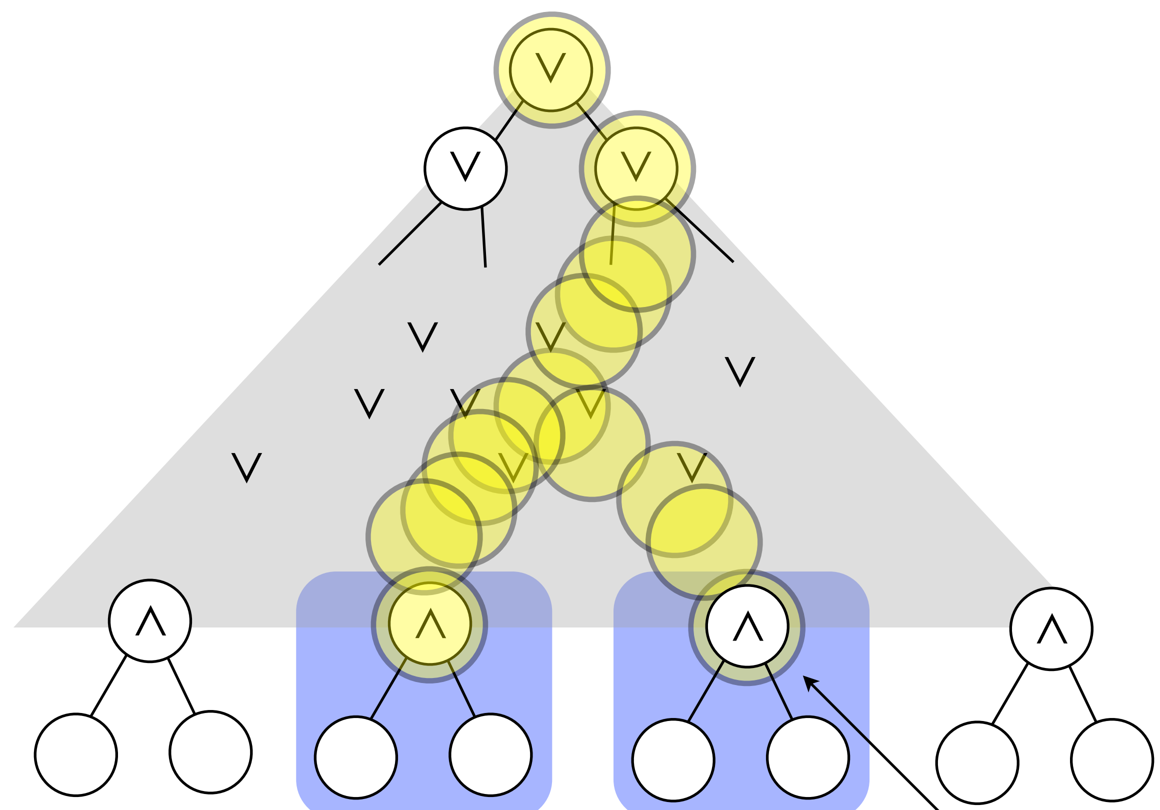
Why can't you do Boolean expressions in PDL?

Induction on nesting depth in formula.
We only do the case of nesting depth 1.



Let $L_1,\ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left},\text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

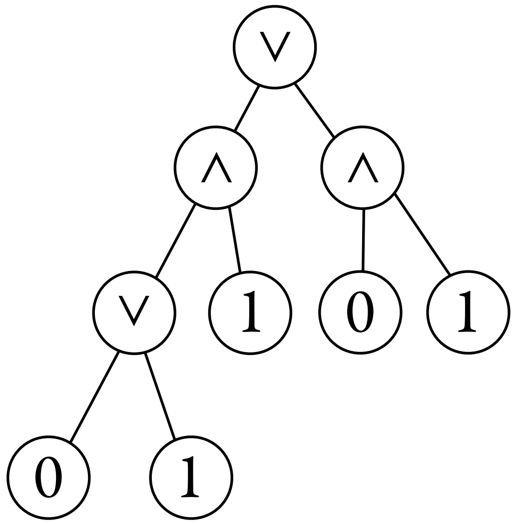
Let Q be the state space of the automaton recognizing all L_i.



with each ∧ node, we associate the state transformation of the path that leads to the root.

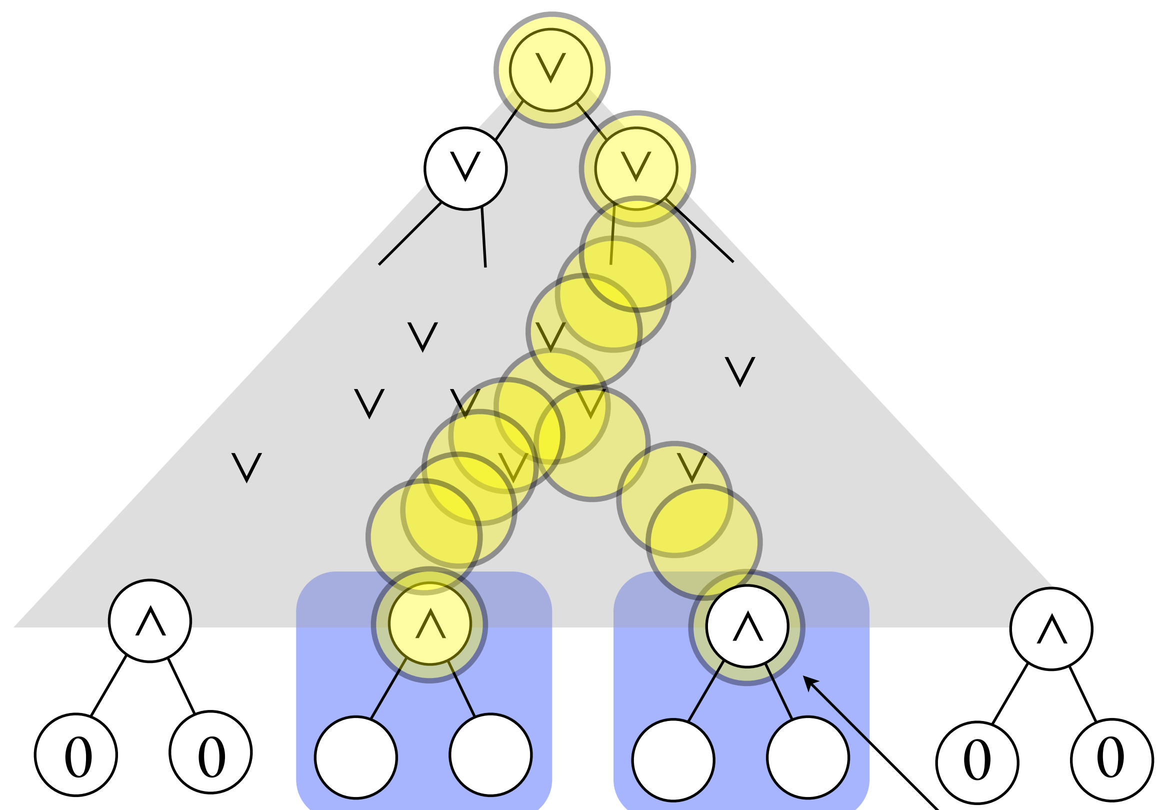
Why can't you do Boolean expressions in PDL?

Induction on nesting depth in formula.
We only do the case of nesting depth 1.



Let $L_1, \ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left}, \text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

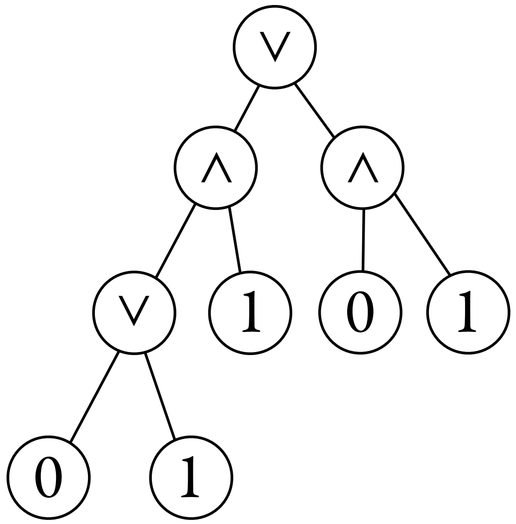
Let Q be the state space of the automaton recognizing all L_i.



we put 0 in all other leaves

with each $\wedge$ node, we associate the state transformation of the path that leads to the root.

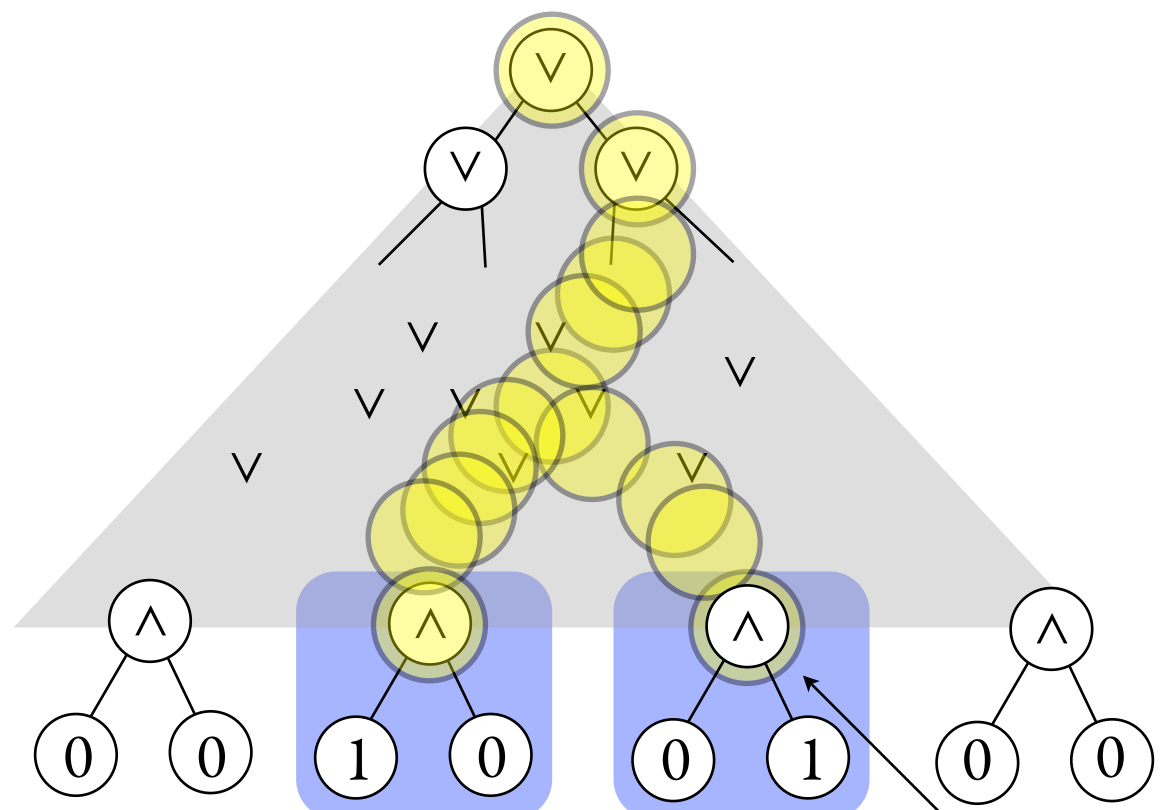
Why can't you do Boolean expressions in PDL?



Induction on nesting depth in formula.
We only do the case of nesting depth 1.

Let $L_1, \ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left}, \text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

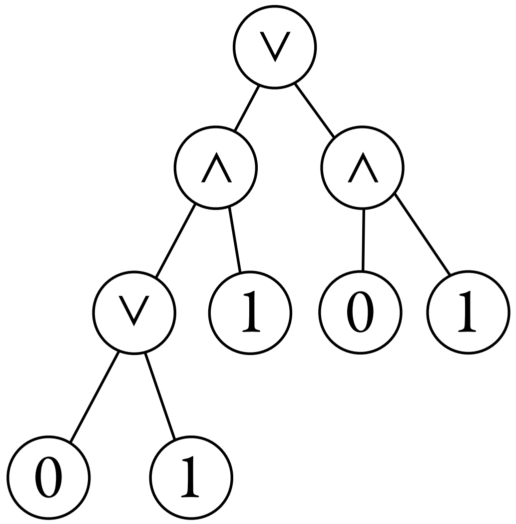
Let Q be the state space of the automaton recognizing all L_i.



we put 0 in all other leaves

with each $\wedge$ node, we associate the state transformation of the path that leads to the root.

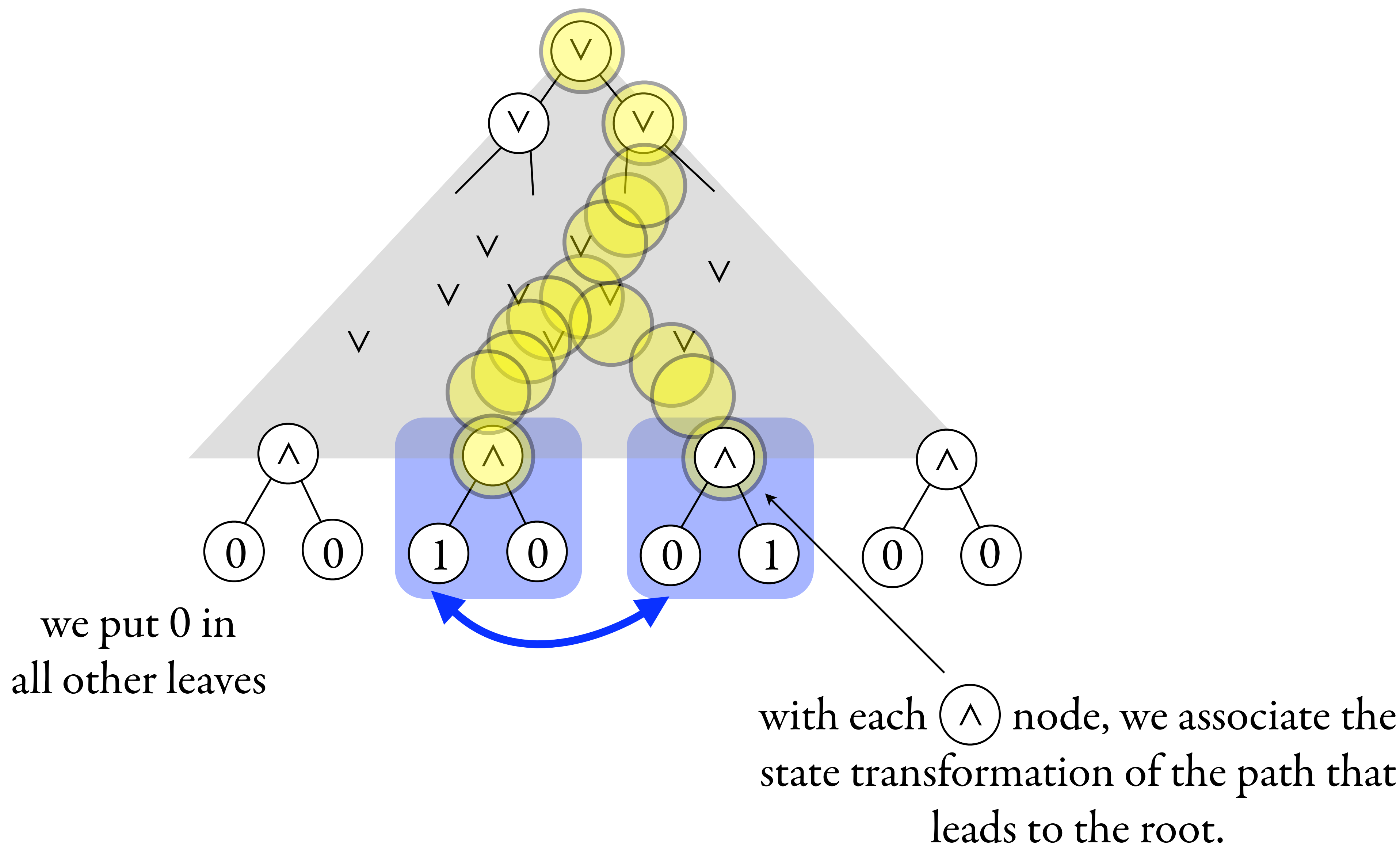
Why can't you do Boolean expressions in PDL?



Induction on nesting depth in formula.
We only do the case of nesting depth 1.

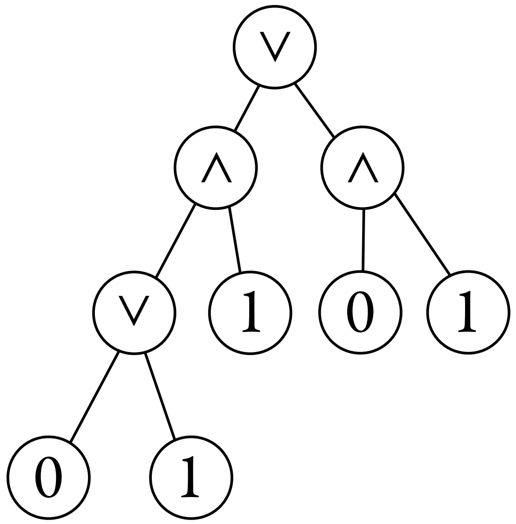
Let $L_1, \ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left}, \text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

Let Q be the state space of the automaton recognizing all L_i.



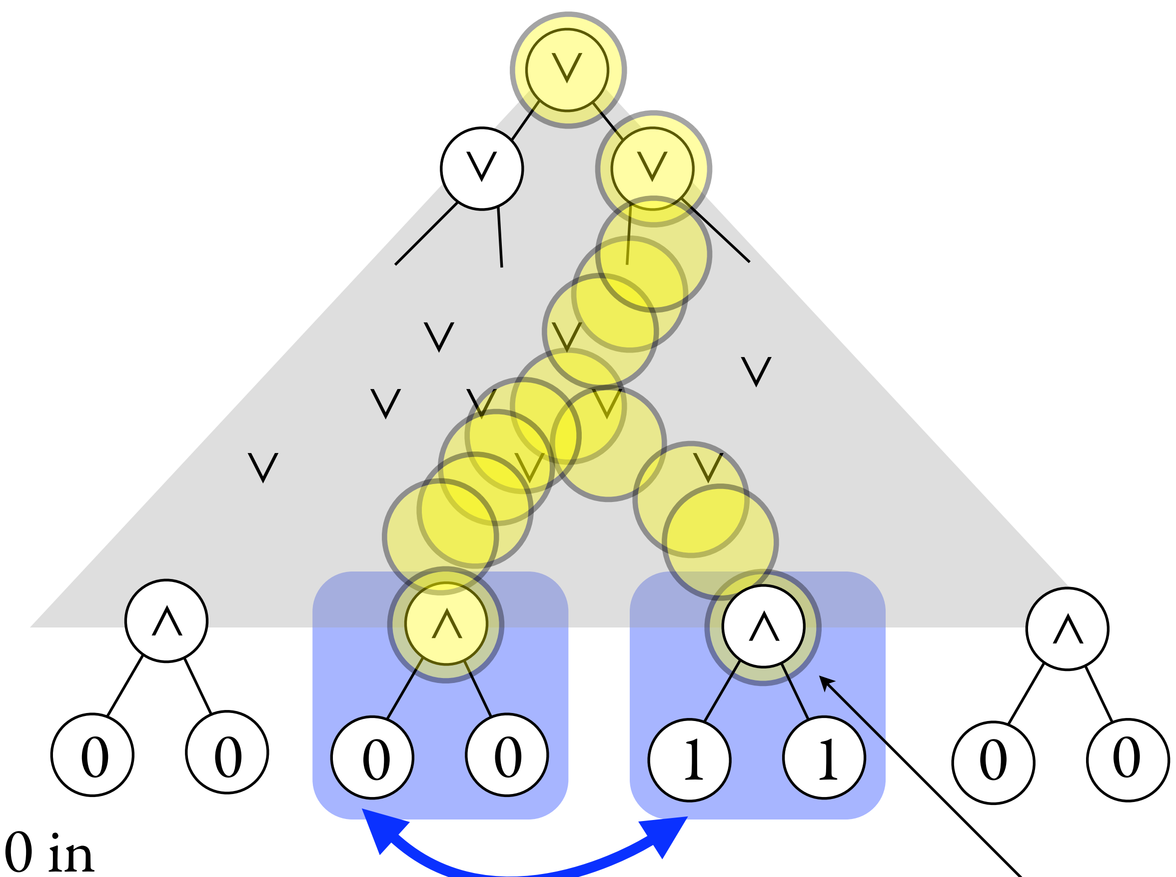
Why can't you do Boolean expressions in PDL?

Induction on nesting depth in formula.
We only do the case of nesting depth 1.



Let $L_1,\ldots, L_n$ be regular word languages over $\{\vee, \wedge, 0, 1\} \times \{\text{left},\text{right}\}$.
No boolean combination of languages $\mathbf{E}L_i$ defines the set of true boolean expressions.

Let Q be the state space of the automaton recognizing all L_i.



we put 0 in all other leaves

with each $\wedge$ node, we associate the state transformation of the path that leads to the root.

Temporal Logic for Words

definition
the virtuous cycle
MSO=regular

Temporal Logic for Trees

definition
CTL, PDL, CTL*
expressivity

XPath

definition
two-variable logic
regular XPath

Temporal Logic for Words

definition
the virtuous cycle
MSO=regular

Temporal Logic for Trees

definition
CTL, PDL, CTL*
expressivity

XPath

definition
two-variable logic
regular XPath

XPath

(navigational XPath)

XPath

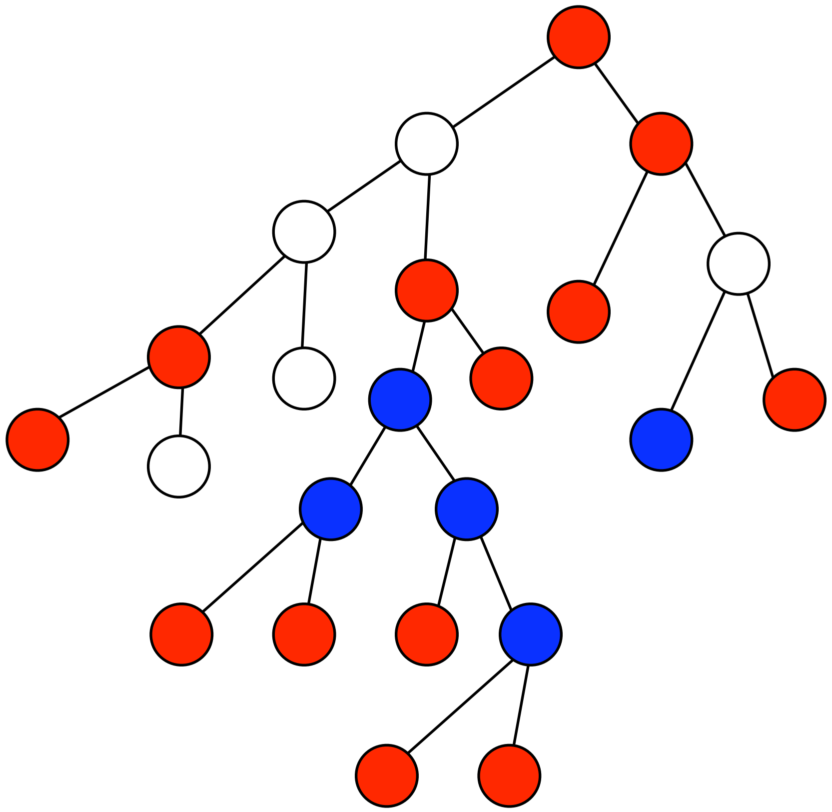
(navigational XPath)

unary query: selects a node

binary query: selects a pair of nodes

XPath

(navigational XPath)

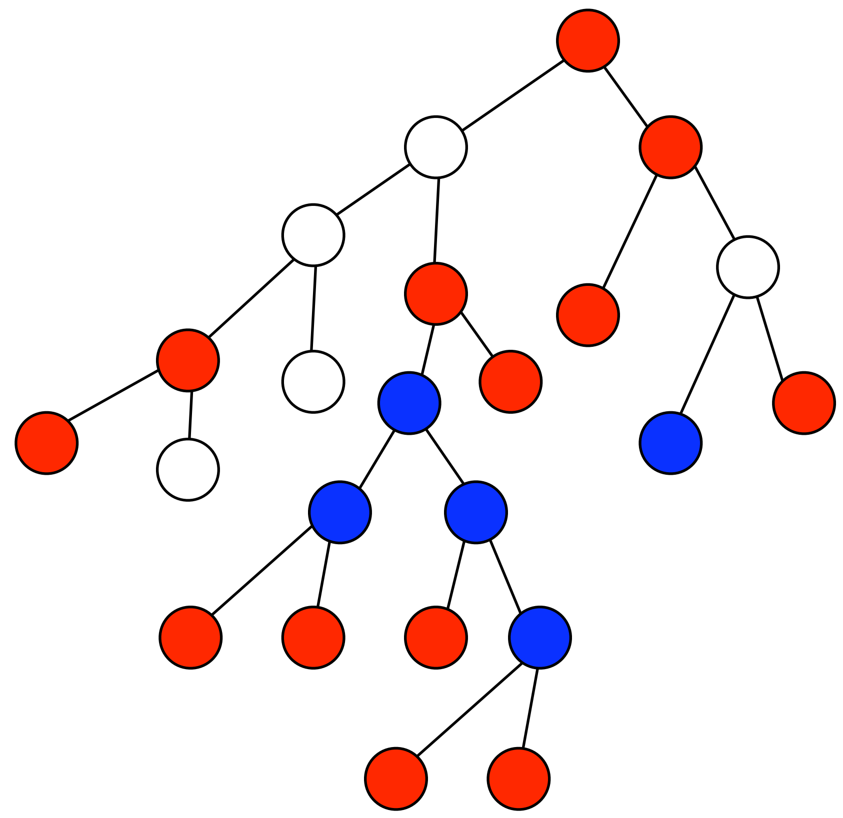


unary query: selects a node

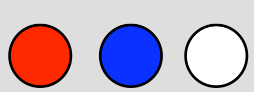
binary query: selects a pair of nodes

XPath

(navigational XPath)



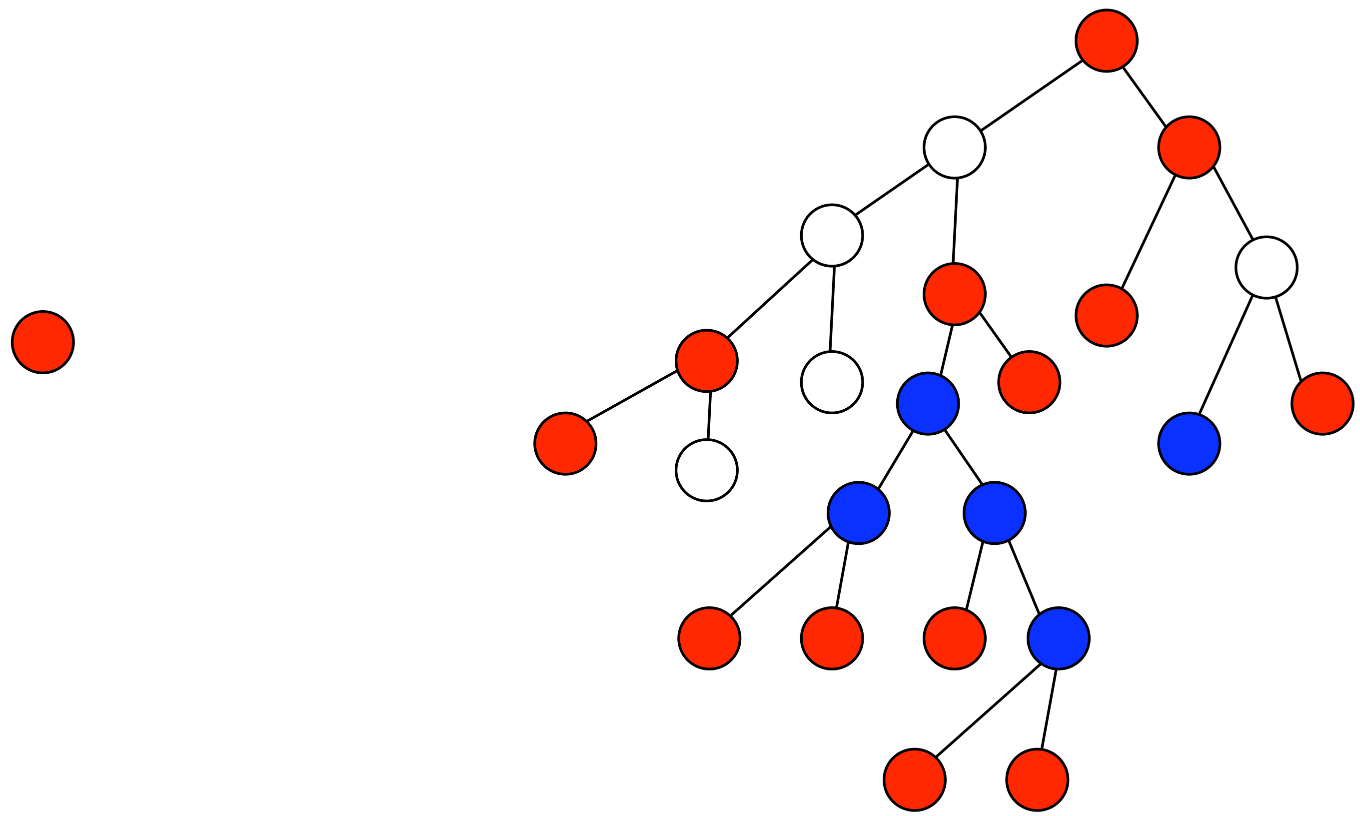
unary query: selects a node



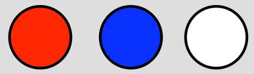
binary query: selects a pair of nodes

XPath

(navigational XPath)



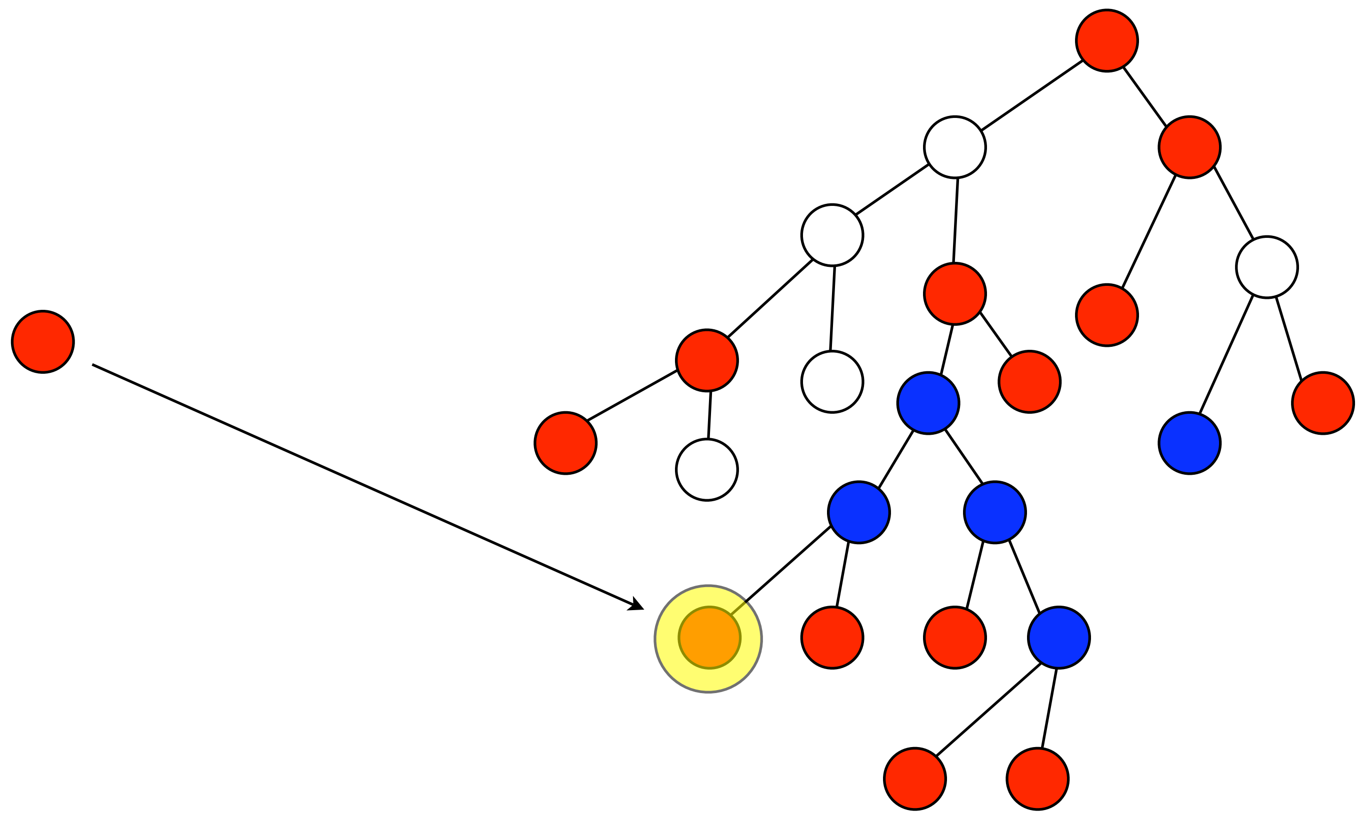
unary query: selects a node



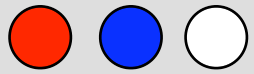
binary query: selects a pair of nodes

XPath

(navigational XPath)



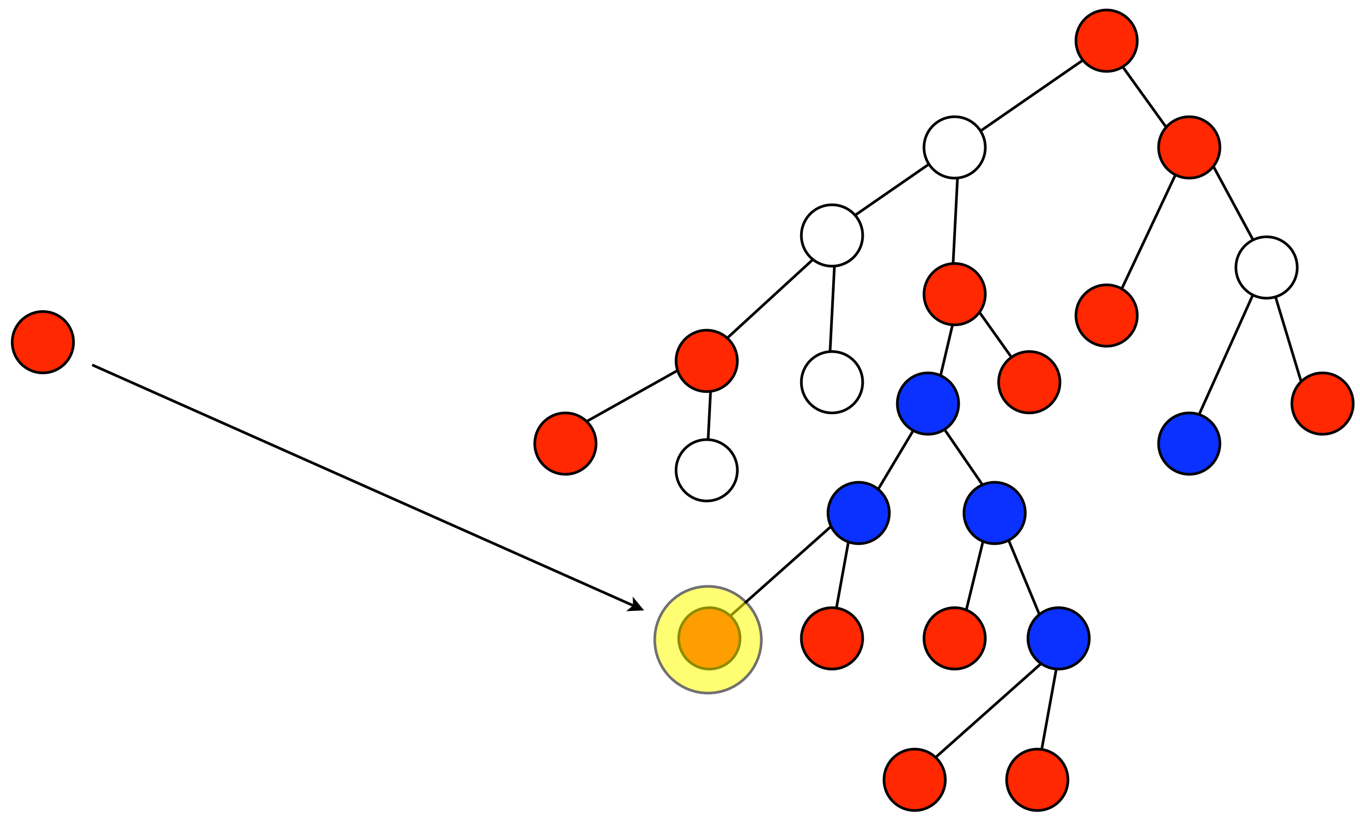
unary query: selects a node



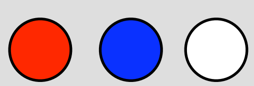
binary query: selects a pair of nodes

XPath

(navigational XPath)



unary query: selects a node

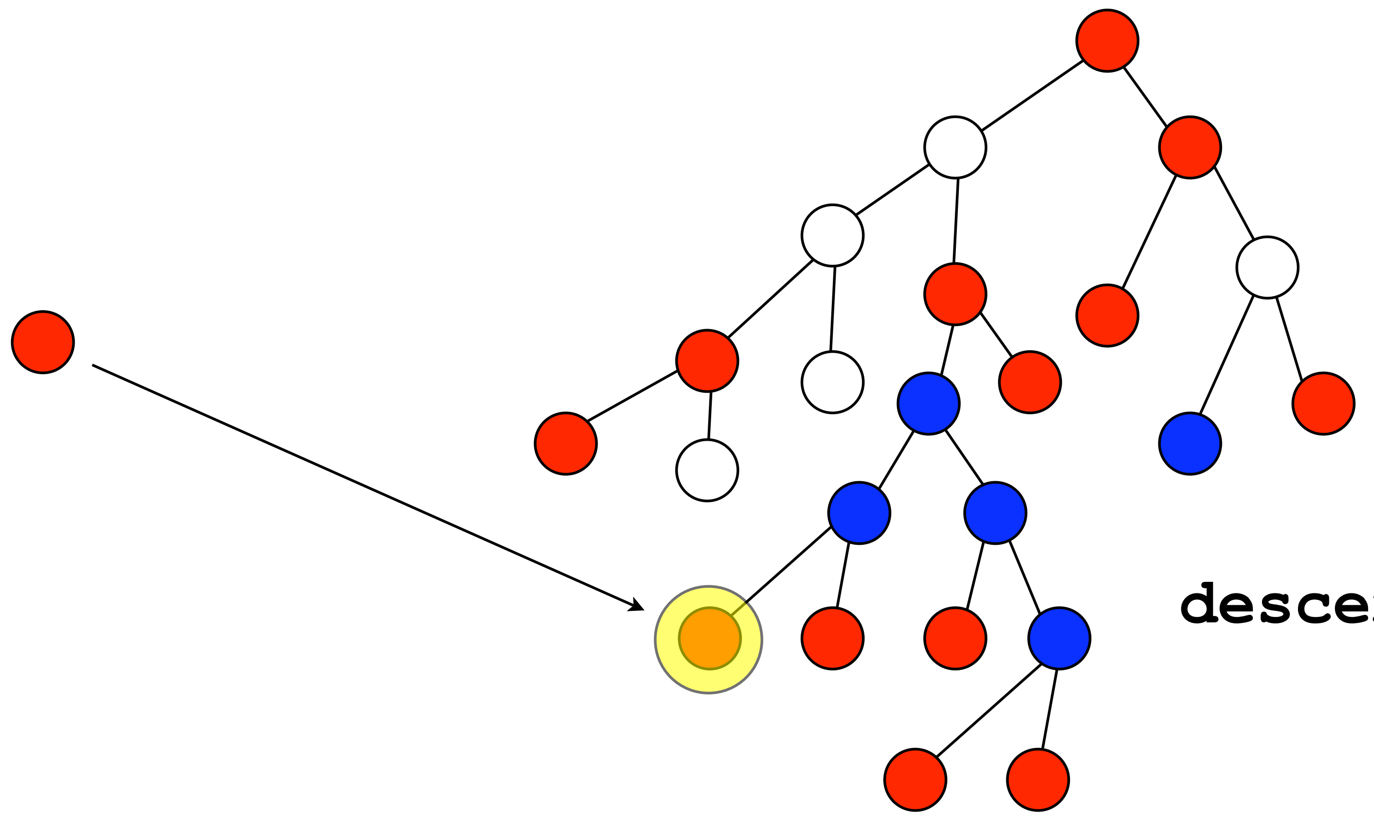


binary query: selects a pair of nodes

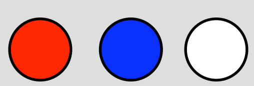
`descendant, child, right`
and their converses

XPath

(navigational XPath)



unary query: selects a node



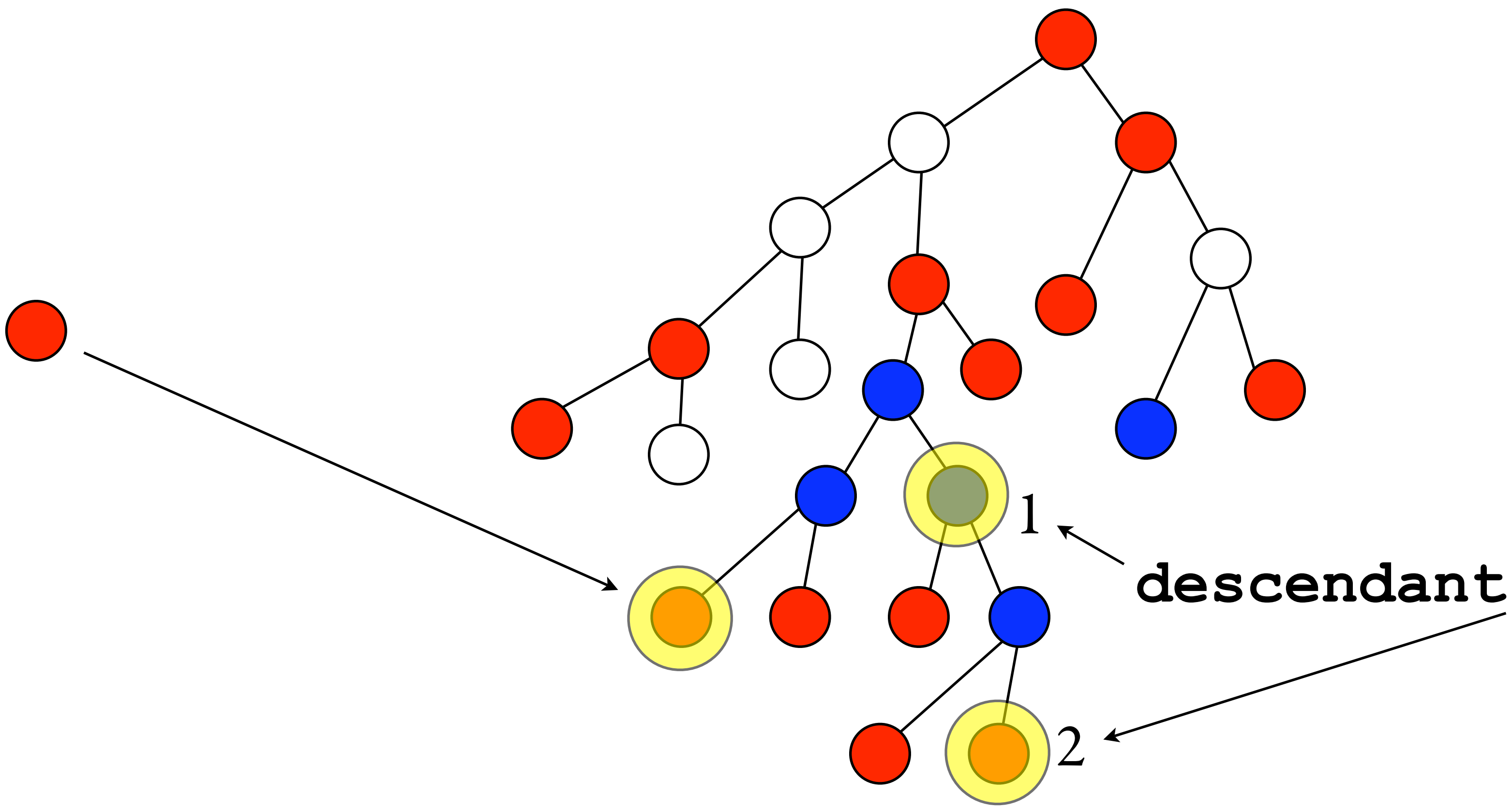
binary query: selects a pair of nodes

`descendant, child, right`

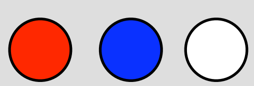
and their converses

XPath

(navigational XPath)



unary query: selects a node

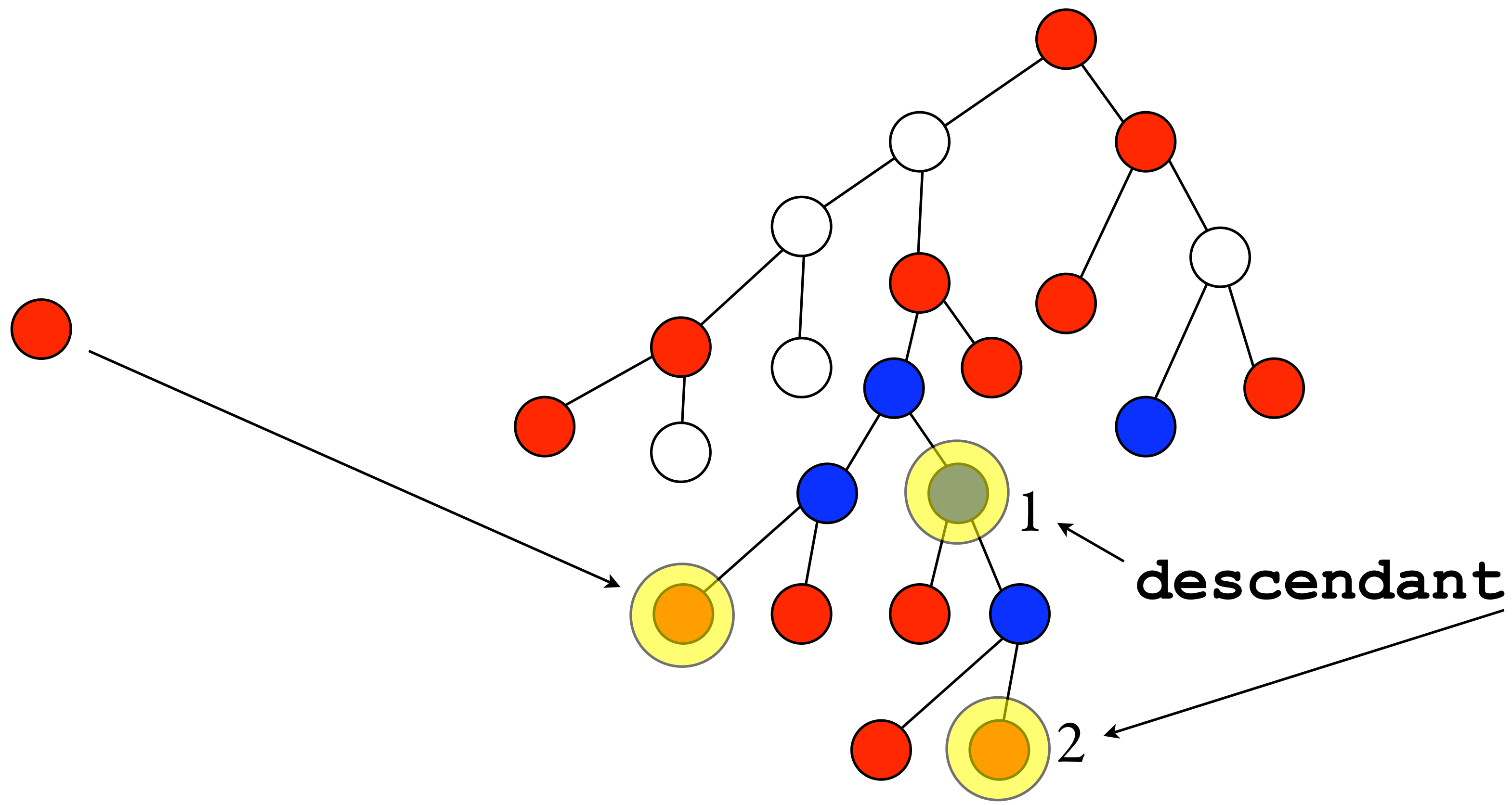


binary query: selects a pair of nodes

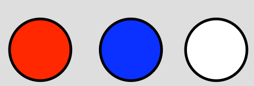
`descendant, child, right`
and their converses

XPath

(navigational XPath)



unary query: selects a node



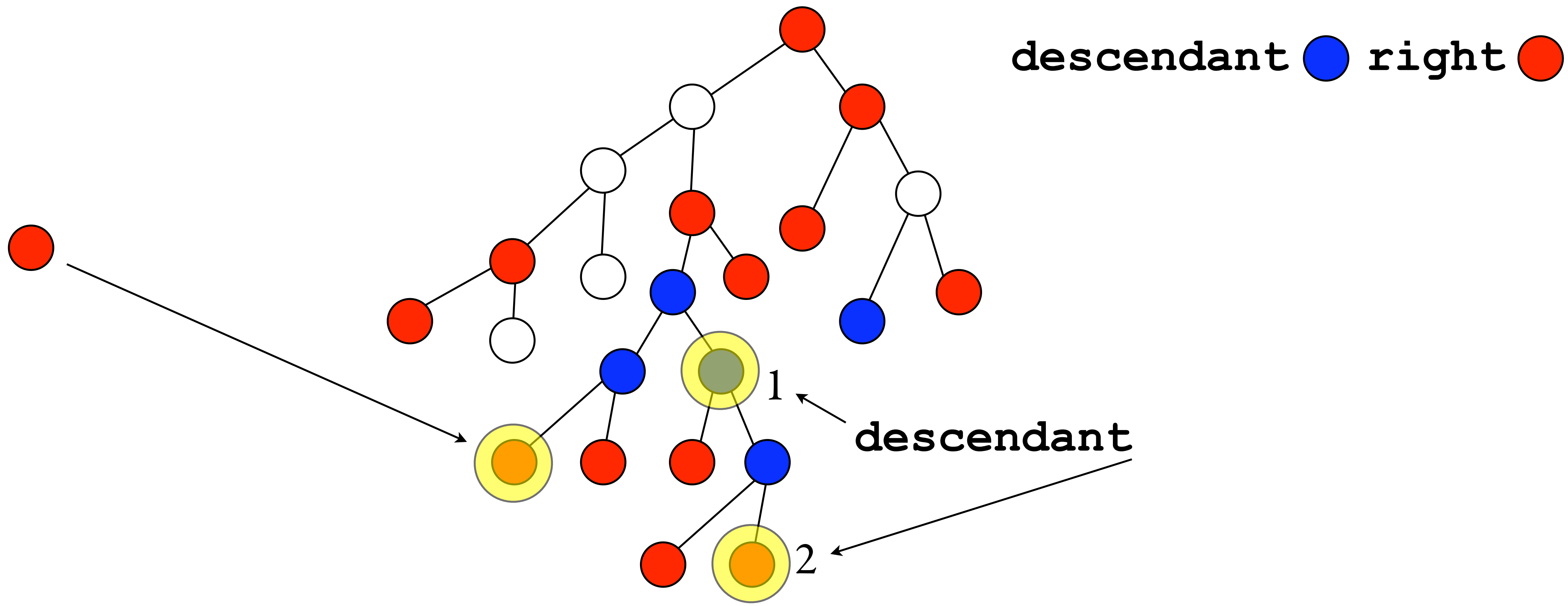
binary query: selects a pair of nodes

`descendant, child, right`
and their converses

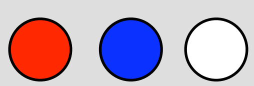
composition

XPath

(navigational XPath)



unary query: selects a node



binary query: selects a pair of nodes

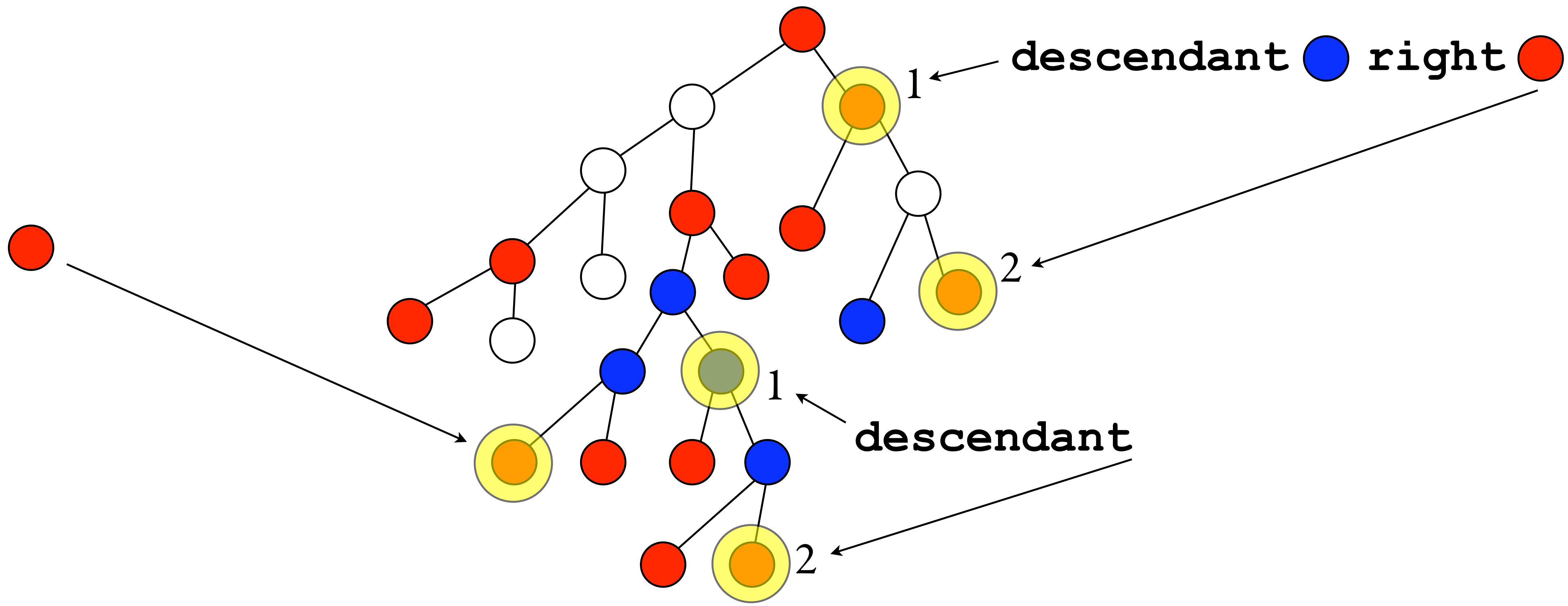
`descendant, child, right`

and their converses

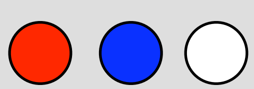
composition

XPath

(navigational XPath)



unary query: selects a node



binary query: selects a pair of nodes

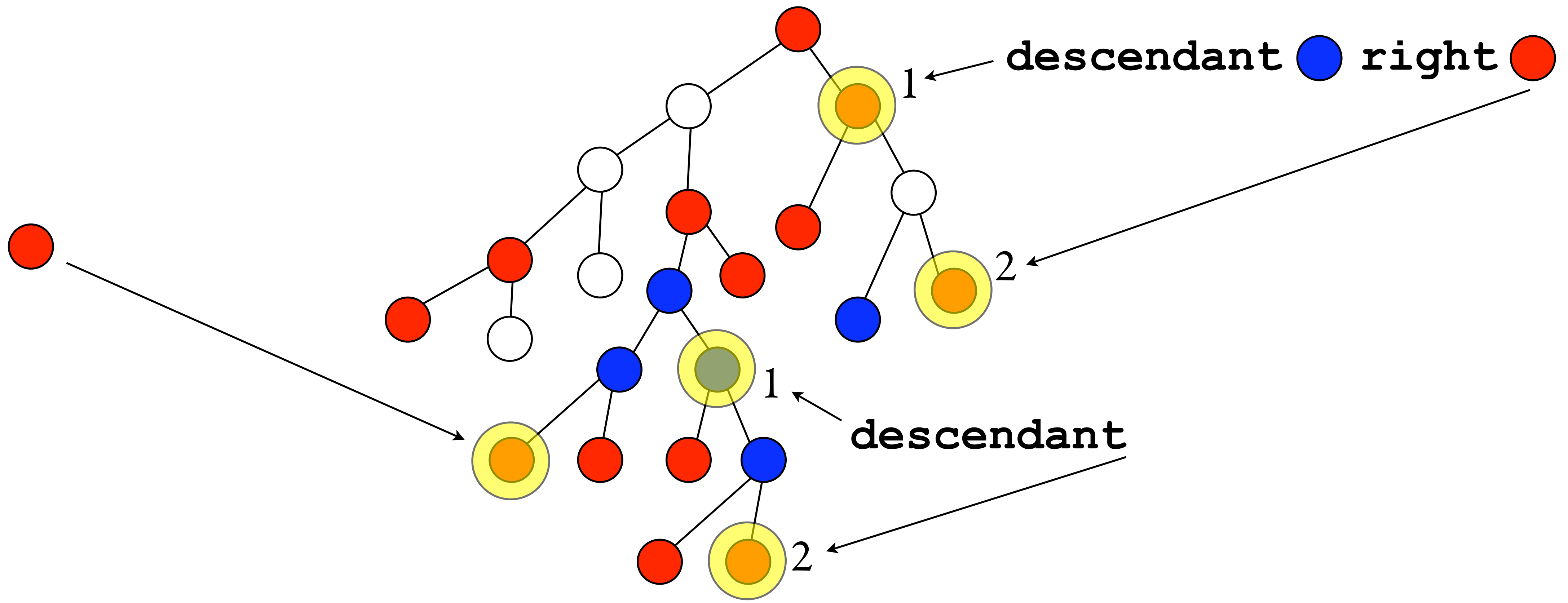
`descendant, child, right`

and their converses

composition

XPath

(navigational XPath)



unary query: selects a node

binary query: selects a pair of nodes

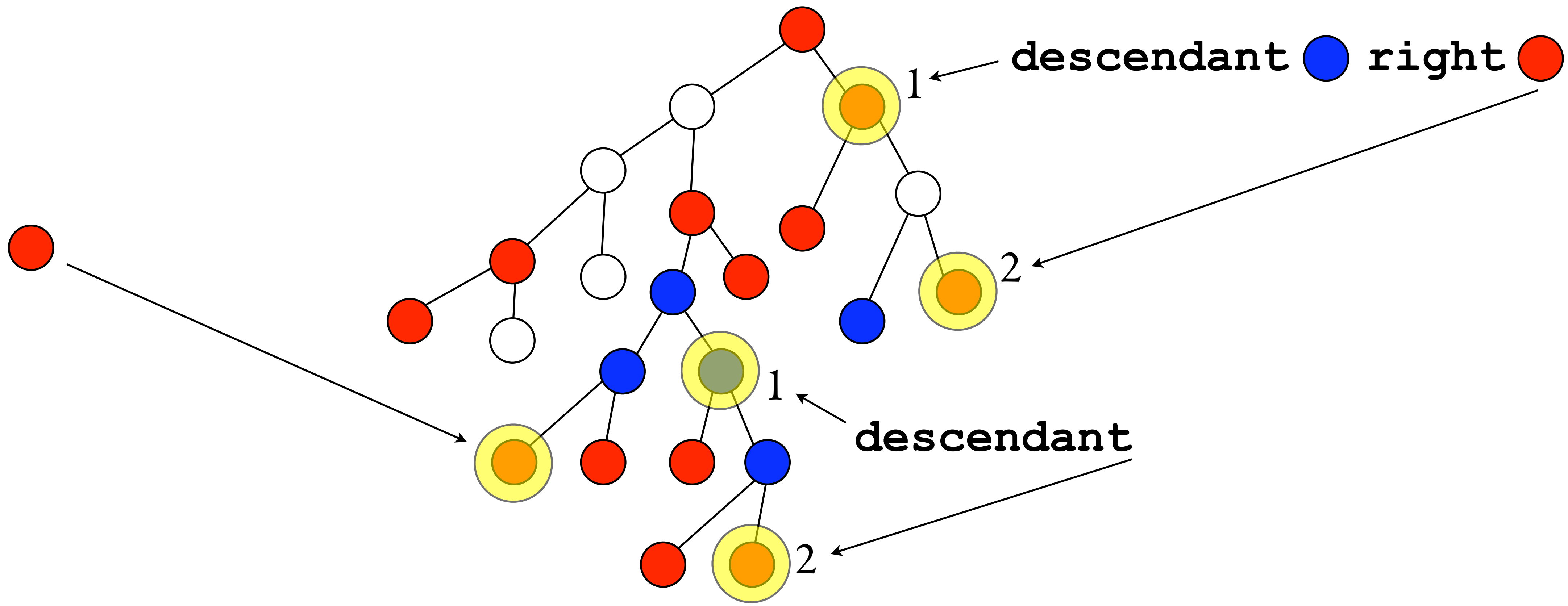
`descendant, child, right`
and their converses

composition

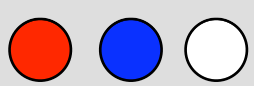
a unary query can be seen as a binary query that selects a subset of pairs (x,x)

XPath

(navigational XPath)



unary query: selects a node



exists (binary query)

binary query: selects a pair of nodes

`descendant, child, right`
and their converses

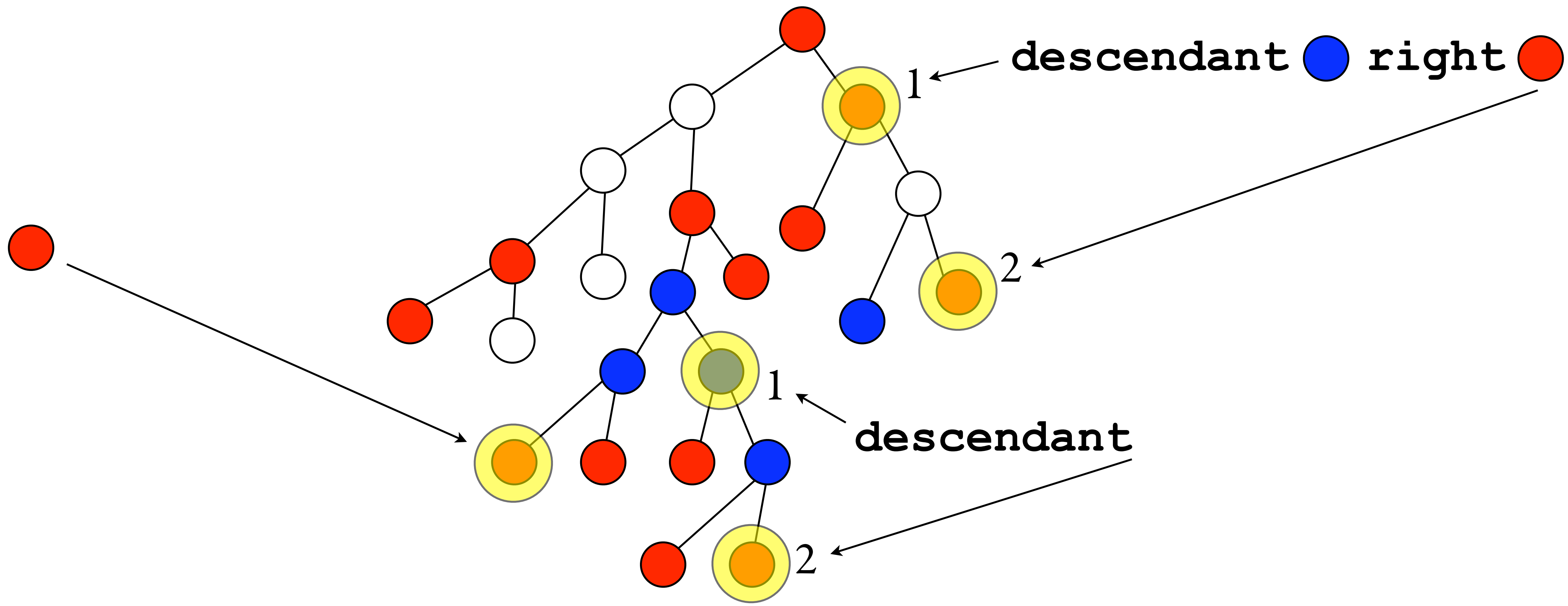
composition

a unary query can be seen as a binary query that selects a subset of pairs (x,x)

XPath

(navigational XPath)

exists(`descendant` ● `right` ○)



unary query: selects a node

● ● ○

exists (binary query)

binary query: selects a pair of nodes

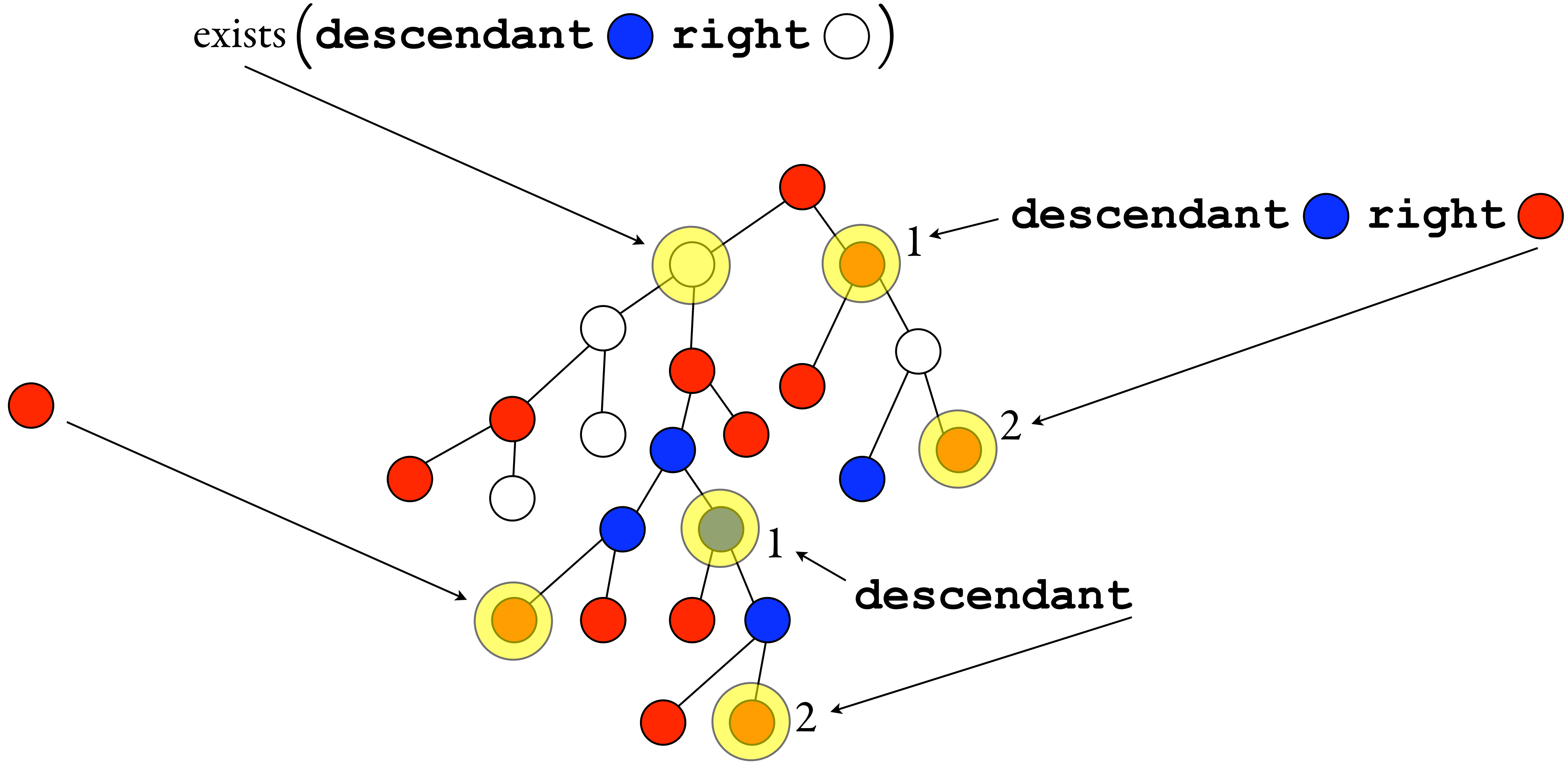
`descendant, child, right`
and their converses

composition

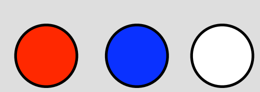
a unary query can be seen as a binary query that selects a subset of pairs (x,x)

XPath

(navigational XPath)



unary query: selects a node



exists (binary query)

binary query: selects a pair of nodes

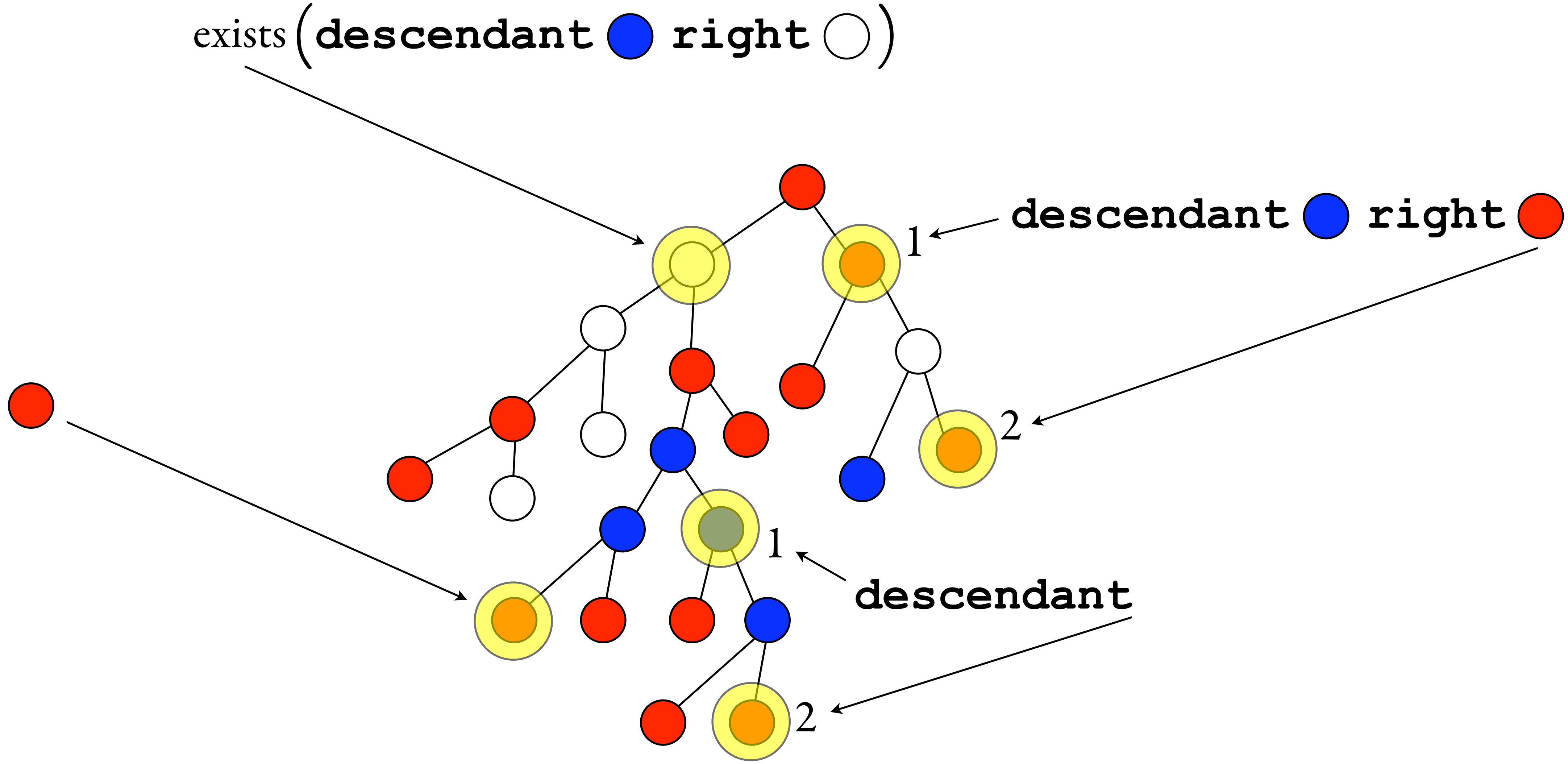
`descendant, child, right`
and their converses

composition

a unary query can be seen as a binary query that selects a subset of pairs (x,x)

XPath

(navigational XPath)



unary query: selects a node

boolean operations

exists (binary query)

binary query: selects a pair of nodes

`descendant, child, right`
and their converses

composition

a unary query can be seen as a binary query that selects a subset of pairs (x,x)

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad \bullet(x)$$

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad (x)$$

becomes

exists$\Big($**`ancestor`**$\Big)$

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath	**from XPath to logic**
$\varphi(x) = \exists y < x \quad \bullet(x)$	
becomes	
exists$\Big($**`ancestor`** $\bullet\Big)$	

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad (x)$$

becomes

exists(**`ancestor`**)

from XPath to logic

exists(**`ancestor`** **`ancestor`**)

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad (x)$$

becomes

exists (**`ancestor`**)

from XPath to logic

exists (**`ancestor`** **`ancestor`**)

becomes

exists (**`ancestor`** exists (**`ancestor`** ( )))

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}(x)$$

becomes

exists(**`ancestor`** $\textcolor{red}{\bullet}$)

from XPath to logic

exists(**`ancestor`** $\textcolor{red}{\bullet}$ **`ancestor`** $\textcolor{blue}{\bullet}$)

becomes

exists(**`ancestor`** $\textcolor{red}{\bullet}$ exists(**`ancestor`** ($\textcolor{blue}{\bullet}$)))

becomes

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}(x) \wedge \left(\exists x < y \quad \textcolor{blue}{\bullet}(y)\right)$$

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}(x)$$

becomes

exists$\big($**`ancestor`** $\textcolor{red}{\bullet}\big)$

from XPath to logic

exists$\big($**`ancestor`** $\textcolor{red}{\bullet}$ **`ancestor`** $\textcolor{blue}{\bullet}\big)$

becomes

exists$\Big($**`ancestor`** $\textcolor{red}{\bullet}$ exists$\big($**`ancestor`** $(\textcolor{blue}{\bullet})\big)\Big)$

becomes

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}(x) \wedge \big(\exists x < y \quad \textcolor{blue}{\bullet}(y)\big)$$

`ancestor` $\textcolor{red}{\bullet}$ **`ancestor`**

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}(x)$$

becomes

exists$\Big($**ancestor** $\textcolor{red}{\bullet}\Big)$

from XPath to logic

exists$\Big($**ancestor** $\textcolor{red}{\bullet}$ **ancestor** $\textcolor{blue}{\bullet}\Big)$

becomes

exists$\Big($**ancestor** $\textcolor{red}{\bullet}$ exists$\big($**ancestor** $(\textcolor{blue}{\bullet})\big)\Big)$

becomes

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}(x) \wedge \left(\exists x < y \quad \textcolor{blue}{\bullet}(y)\right)$$

$\neg\Big($**ancestor** $\textcolor{red}{\bullet}$ **ancestor** $\Big)$

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}(x)$$

becomes

$$\text{exists}\left(\texttt{ancestor}\,\textcolor{red}{\bullet}\right)$$

from XPath to logic

$$\text{exists}\left(\texttt{ancestor}\,\textcolor{red}{\bullet}\ \texttt{ancestor}\ \textcolor{blue}{\bullet}\right)$$

becomes

$$\text{exists}\left(\texttt{ancestor}\ \textcolor{red}{\bullet}\ \text{exists}\left(\texttt{ancestor}\ (\textcolor{blue}{\bullet})\right)\right)$$

becomes

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}(x) \wedge \left(\exists x < y \quad \textcolor{blue}{\bullet}(y)\right)$$

$$\left(\neg\left(\texttt{ancestor} \quad \textcolor{red}{\bullet} \quad \texttt{ancestor}\right) \cap \ \texttt{ancestor}\right)$$

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad \bullet(x)$$

becomes

exists(**`ancestor`** $\bullet$)

from XPath to logic

exists(**`ancestor`** $\bullet$ **`ancestor`** $\bullet$)

becomes

exists(**`ancestor`** $\bullet$ exists(**`ancestor`** ($\bullet$)))

becomes

$$\varphi(x) = \exists y < x \quad \bullet(x) \wedge (\exists x < y \quad \bullet(y))$$

$(\neg($ **`ancestor`** $\neg\bullet$ **`ancestor`** $)\cap$ **`ancestor`** $)$

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}(x)$$

becomes

$$\text{exists}\Big(\texttt{ancestor}\,\textcolor{red}{\bullet}\Big)$$

from XPath to logic

$$\text{exists}\Big(\texttt{ancestor}\,\textcolor{red}{\bullet}\ \texttt{ancestor}\ \textcolor{blue}{\bullet}\Big)$$

becomes

$$\text{exists}\Big(\texttt{ancestor}\ \textcolor{red}{\bullet}\ \text{exists}\big(\texttt{ancestor}\ (\textcolor{blue}{\bullet})\big)\Big)$$

becomes

$$\varphi(x) = \exists y < x \quad \textcolor{red}{\bullet}\,(x) \wedge \big(\exists x < y \quad \textcolor{blue}{\bullet}\,(y)\big)$$

$$\Big(\neg\big(\texttt{ancestor}\quad \neg\textcolor{red}{\bullet}\quad \texttt{ancestor}\ \big) \cap \ \texttt{ancestor}\Big)(x,y)$$

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$\varphi(x) = \exists y < x \quad$ 🔴(x)

becomes

exists(**`ancestor`** 🔴)

from XPath to logic

exists(**`ancestor`** 🔴 **`ancestor`** 🔵)

becomes

exists(**`ancestor`** 🔴 exists(**`ancestor`** (🔵)))

becomes

$\varphi(x) = \exists y < x \quad$ 🔴$(x) \wedge (\exists x < y \quad$ 🔵$(y))$

$\Big(\neg\big($ **`ancestor`** $\neg$🔴 **`ancestor`** $\big) \cap$ **`ancestor`** $\Big)(x,y)$ 🔴 holds between x and y

Thm. (Marx, de Rijke)
XPath has the same expressive power as two-variable first-order logic.
(the predicates are the same as in XPath: descendant, left, etc.)

(for boolean and unary queries only, not for binary queries)

from logic to XPath

$$\varphi(x) = \exists y < x \quad (x)$$

becomes

exists(**ancestor**)

from XPath to logic

exists(**ancestor** **ancestor**)

becomes

exists(**ancestor** exists(**ancestor** ( )))

becomes

$$\varphi(x) = \exists y < x \quad (x) \wedge (\exists x < y \quad (y))$$

$\Big(\neg\big($**ancestor** $\neg$ **ancestor** $\big) \cap$ **ancestor**$\Big)(x,y)$ holds between x and y

exists until

Regular XPath = XPath with Kleene star for binary queries

Regular XPath

= XPath with Kleene star for binary queries

path of even length $\left(\texttt{child child}\right)^*$

Regular XPath = XPath with Kleene star for binary queries

path of even length $\left(\texttt{\textbf{child child}}\right)^*$

Regular XPath captures all first-order logic...

Regular XPath

= XPath with Kleene star for binary queries

path of even length $\left(\texttt{child child}\right)^*$

Regular XPath captures all first-order logic... exists until =

Regular XPath = XPath with Kleene star for binary queries

path of even length $\left(\texttt{child child}\right)^*$

Regular XPath captures all first-order logic... exists until = $\left(\texttt{child}\right)^*$

Regular XPath = XPath with Kleene star for binary queries

path of even length $\left(\texttt{child child}\right)^*$

Regular XPath captures all first-order logic... exists until = $\left(\texttt{child}\right)^*$

...and all of PDL...

Regular XPath

= XPath with Kleene star for binary queries

path of even length $(\texttt{child child})^*$

Regular XPath captures all first-order logic...

exists ● until ● $=$ $(● \ \texttt{child})^*$ ●

...and all of PDL...

...and more

Regular XPath

= XPath with Kleene star for binary queries

path of even length $(\texttt{child child})^*$

Regular XPath captures all first-order logic...

exists until = $(\texttt{child})^*$

...and all of PDL...

...and more

`next` boolean query that connects a node with the next one in DFS traversal

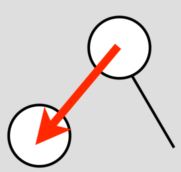
Regular XPath = XPath with Kleene star for binary queries

path of even length $\left(\texttt{child child}\right)^*$

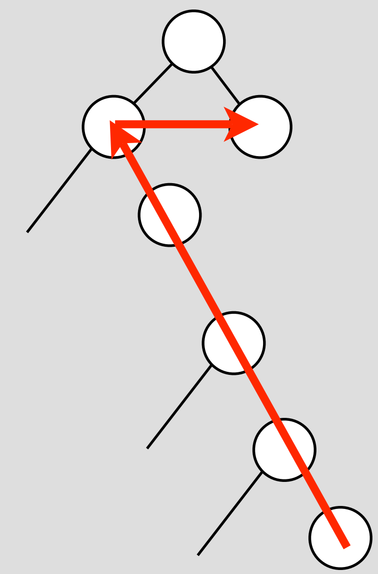
Regular XPath captures all first-order logic... exists ● until ● = $\left(● \; \texttt{child}\right)^*$ ●

...and all of PDL...

...and more



`next` boolean query that connects a node with the next one in DFS traversal



Regular XPath = XPath with Kleene star for binary queries

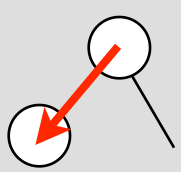
path of even length $(\texttt{child child})^*$

Regular XPath captures all first-order logic...

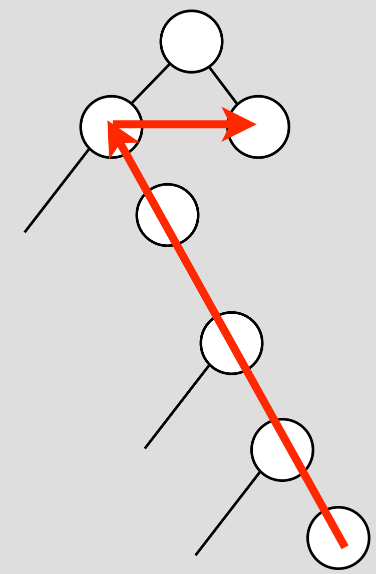
exists ● until ● $=$ $(● \texttt{child})^*$ ●

...and all of PDL...

...and more



next boolean query that connects a node with the next one in DFS traversal



$\texttt{child}\ (\text{exists } \texttt{right})$ or $\left(\texttt{parent}\ (\neg \text{ exists } \texttt{right})\right)^* \texttt{right}$

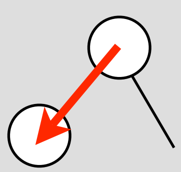
Regular XPath = XPath with Kleene star for binary queries

path of even length (**child child**)*

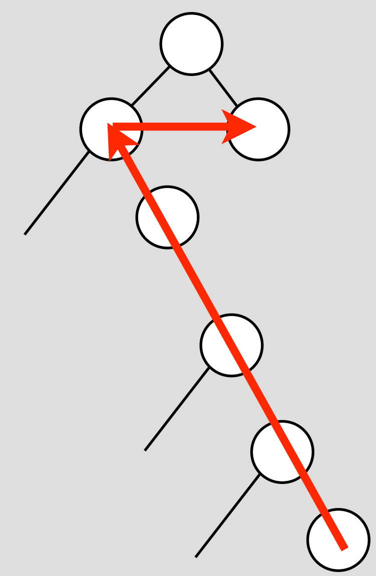
Regular XPath captures all first-order logic... exists 🔴 until 🔵 = (🔴 **child**)* 🔵

...and all of PDL...

...and more



next boolean query that connects a node with the next one in DFS traversal



child (exists **right**) or (**parent** (¬ exists **right**))* **right**

((**next** 🔵)* **next** 🔴 (**next** 🔵)* **next** 🔴)* (**next** 🔵)* ¬ exists **next**

Regular XPath

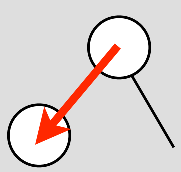
= XPath with Kleene star for binary queries

path of even length $\left(\texttt{child child}\right)^*$

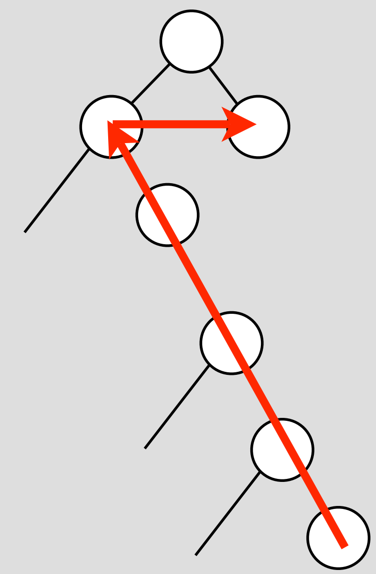
Regular XPath captures all first-order logic... exists $\color{red}\bullet$ until $\color{blue}\bullet$ $=$ $\left(\color{red}\bullet\ \color{black}\texttt{child}\right)^* \color{blue}\bullet$

...and all of PDL...

...and more



`next` boolean query that connects a node with the next one in DFS traversal



$\texttt{child}\ (\text{exists } \texttt{right})$ or $\left(\texttt{parent}\ (\neg \text{ exists } \texttt{right})\right)^* \texttt{right}$

$$\left(\left(\texttt{next}\ \color{blue}\bullet\color{black}\right)^* \texttt{next}\ \color{red}\bullet\color{black}\ \left(\texttt{next}\ \color{blue}\bullet\color{black}\right)^* \texttt{next}\ \color{red}\bullet\color{black}\right)^* \left(\texttt{next}\ \color{blue}\bullet\color{black}\right)^* \neg \text{ exists } \texttt{next}$$

selects the root iff the tree contains an even number of $\color{red}\bullet$ nodes.