Grammar-based Pattern Matching and Type Checking for Difference Data Structures

Naoki Yamamoto, Kazunori Ueda

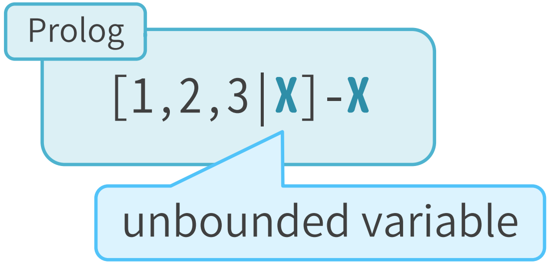
Waseda University, Tokyo, Japan

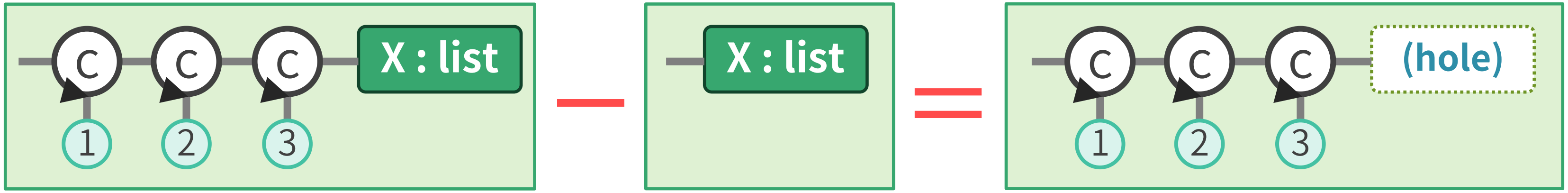
PPDP 2024 @ Milan, Italy
(co-located with FM 2024)

September 10th, 2024

Background: Difference Lists

- **Difference List**: List that supports constant-time append
  - Common in Prolog
  - a.k.a. **list segment** in Separation Logic
  - Can be regarded as **a list with a hole**





Q. Can we generalize this idea to **structures with holes**?
→ **Difference Data Structures (DDSs)**

Overview

Background: Difference Data Structures (DDSs)

Useful for uniformly discussing important concepts

- e.g., (linear) functions, continuations, evaluation contexts

Problem:

The formulation of **types for DDSs** is not obvious

Contributions:

We propose **LMNtalGG** and **Difference Types**,
a typing framework for DDSs based on graph grammars

- Implemented on a graph rewriting language **LMNtal**
- Applications: **Pattern matching** and **Static type checking of rules**

Outline

1. Target Language: LMNtal
2. LMNtalGG and Difference Types
3. Classifying LMNtalGGs: Disjoint & Indexed
4. Applications: Pattern Matching & Static Type Checking
5. Type Checking of Functional Atoms

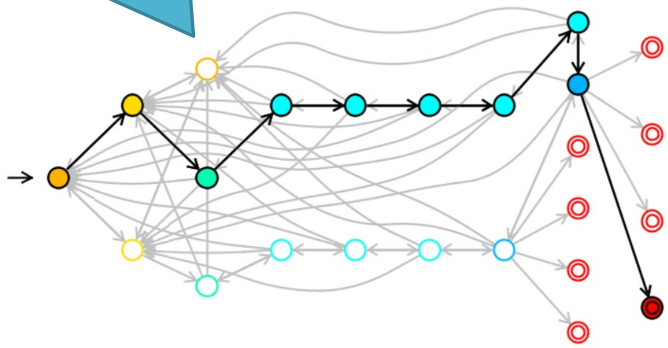
Outline

1. Target Language: LMNtal
2. LMNtalGG and Difference Types
3. Classifying LMNtalGGs: Disjoint & Indexed
4. Applications: Pattern Matching & Static Type Checking
5. Type Checking of Functional Atoms

LMNtal[†1]: Graph Rewriting Language

- LMNtal comes with two aspects:
  - **Programming language** & **Modeling language**
- Its implementation **SLIM**[†2] provides:
  - Ordinary **program execution** & Parallel **model checking** features

Tools are available from GitHub
https://github.com/lmntal

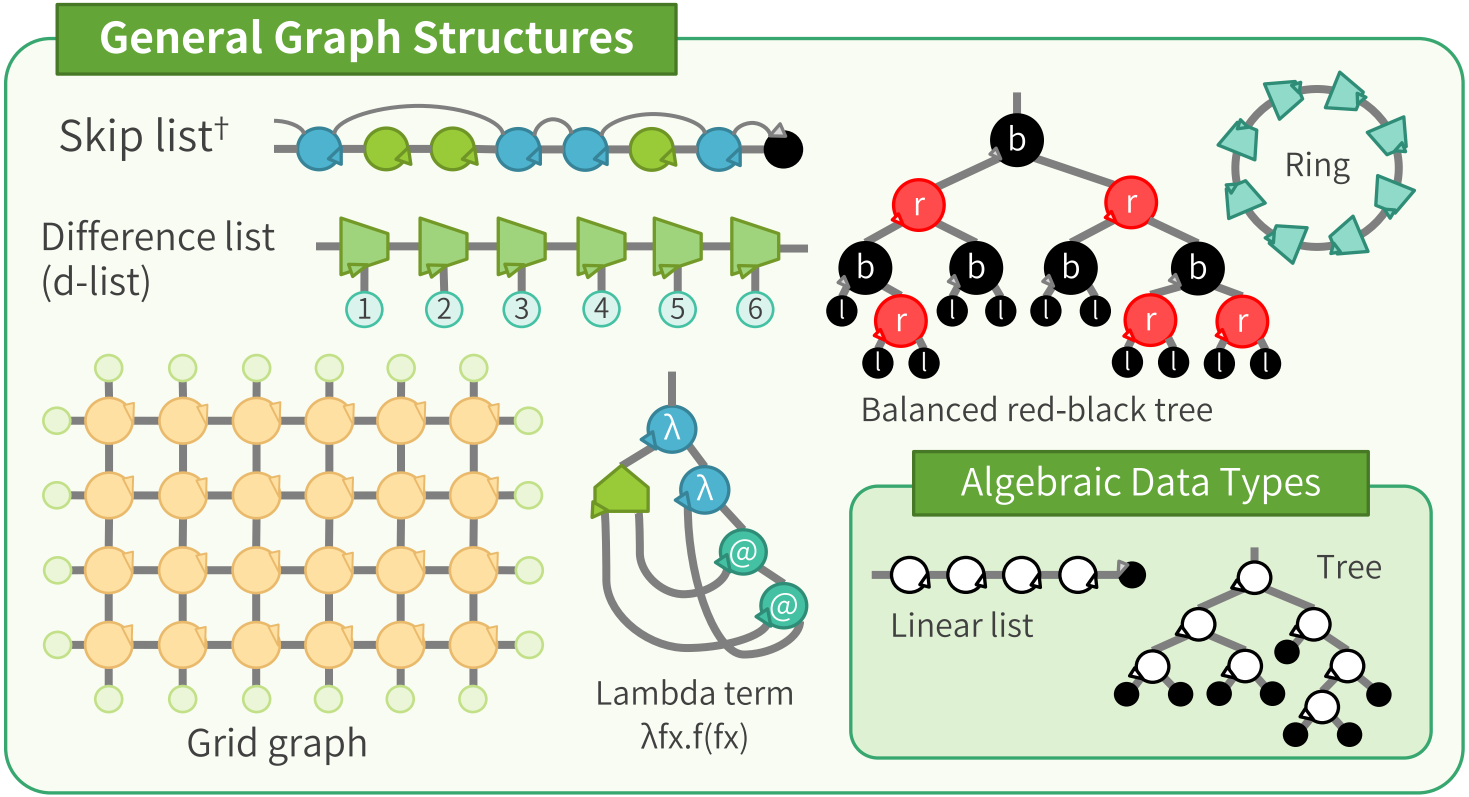


†1 K. Ueda: LMNtal as a hierarchical logic programming language. Theoretical Computer Science 410(46), 2009.

†2 M. Gocho et al.: Evolution of the LMNtal Runtime to a Parallel Model Checker. Computer Software 28(4), 2011.

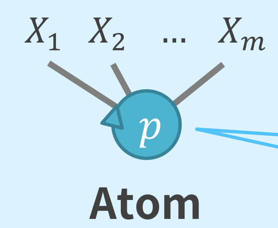
LMNtal: Expressive Data Structures

We can handle non-algebraic data types without dangling pointers



† W. Pugh: Skip lists: A probabilistic alternative to balanced trees, C. ACM, 33(6), 1990.

LMNtal: Syntax



Links are totally ordered (= **port graph**)

Functor p/m (m-ary p atom)

$$
\begin{aligned}
\mathit{Process} &::= \mathit{Graph}\,,\mathit{Ruleset} \\
\mathit{Graph} &::= \mathbf{0} \mid p(X_1,\ldots,X_m) \mid \mathit{Graph}\,,\mathit{Graph} \\
\mathit{Ruleset} &::= \mathbf{0} \mid \mathit{Graph}\ \texttt{:-}\ \mathit{Graph} \mid \mathit{Ruleset}\,,\mathit{Ruleset}
\end{aligned}
$$

Composition

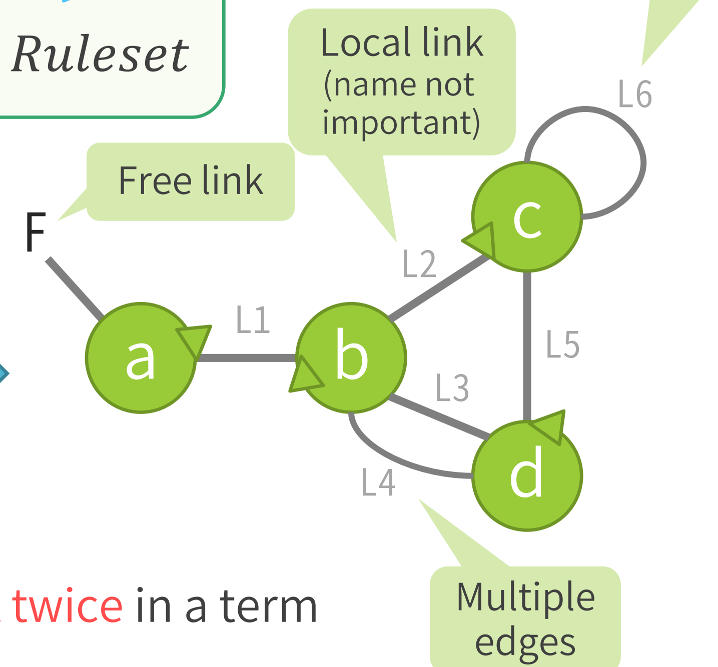
Null

Rewrite Rule

Example

```
a(L1,F),b(L1,L2,L3,L4),
c(L2,L5,L6,L6),d(L5,L3,L4)
```

Self loop



Link Condition: Each link name must occur at most twice in a term

LMNtal: Semantics (1) Structural Congruence

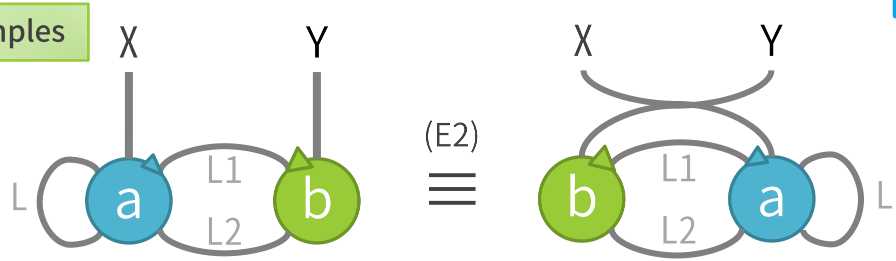
- Gives the interpretation of **LMNtal terms** as **graphs**
  - Plays the role of **isomorphism** between graphs

$$
\begin{array}{llcl}
\text{(E1)} & \mathbf{0}\,,P & \equiv & P \\
\text{(E2)} & P\,,Q & \equiv & Q\,,P \\
\text{(E3)} & P\,,(Q\,,R) & \equiv & (P\,,Q)\,,R \\
\text{(E4)} & P & \equiv & P[Y/X] \\
 & & & \text{(if } X \text{ is a local link of } P\text{)} \\
\text{(E5)} & P \equiv P' & \Rightarrow & P\,,Q \equiv P'\,,Q
\end{array}
$$

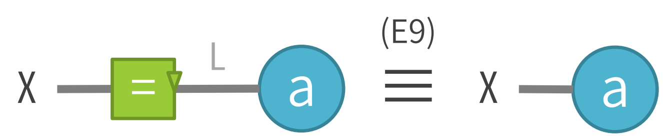
Connector: A binary infix atom $X = Y$ fuses two links

$$
\begin{array}{llcl}
\text{(E7)} & X{=}X & \equiv & \mathbf{0} \\
\text{(E8)} & X{=}Y & \equiv & Y{=}X \\
\text{(E9)} & X{=}Y\,,P & \equiv & P[Y/X] \\
 & \multicolumn{3}{l}{\text{(if } P \text{ is an atom and } X \text{ is a free link of } P\text{)}}
\end{array}
$$

Examples



`a(L1,L2,L,L,X),b(L1,L2,Y)`   `b(L1,L2,Y),a(L1,L2,L,L,X)`



`L=X,a(L)`   `a(X)`

* (E6) & (E10) are omitted because these are rules for membranes

LMNtal: Semantics (2) Reduction Relation

Describes the **small-step semantics** of the language (1-step rule application)

Structural Rules

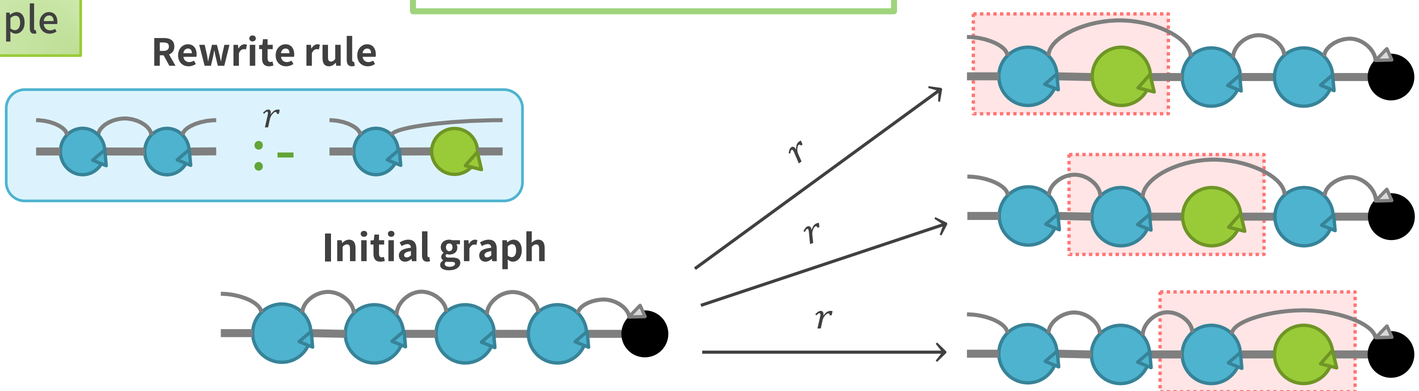
$$\text{(R1)}\ \frac{G_1 \xrightarrow{T\,:\!-\,U} G_1'}{G_1, G_2 \xrightarrow{T\,:\!-\,U} G_1', G_2} \qquad \text{(R3)}\ \frac{G_2 \equiv G_1 \quad G_1 \xrightarrow{T\,:\!-\,U} G_1' \quad G_1' \equiv G_2'}{G_2 \xrightarrow{T\,:\!-\,U} G_2'}$$

Main reduction rule

$$\text{(R6)}\ T \xrightarrow{T\,:\!-\,U} U$$

Non-deterministic rule application

Example



Outline

1. Target Language: LMNtal
2. LMNtalGG and Difference Types
3. Classifying LMNtalGGs: Disjoint & Indexed
4. Applications: Pattern Matching & Static Type Checking
5. Type Checking of Functional Atoms

LMNtalGG: Graph Grammar on LMNtal

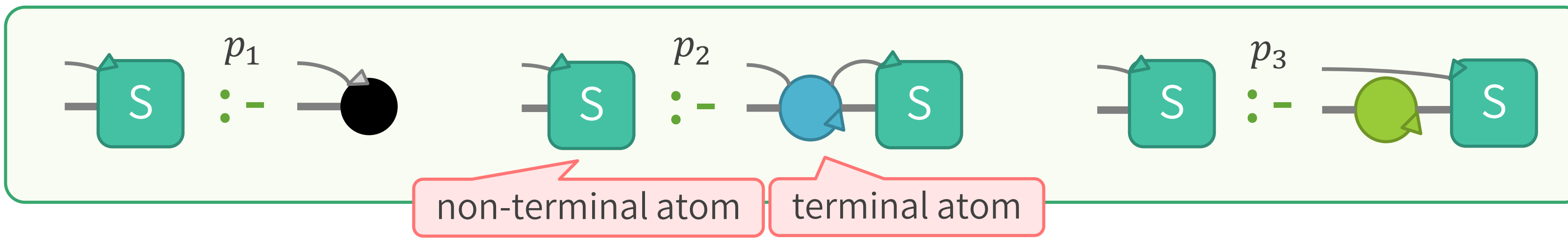
- Inductively defines a set of graphs
  by a context-free graph grammar

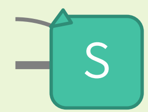
Formal Language Theory	LMNtal
Production rules	**Rewrite rules**
Symbols	Functors*

* Pairs of name and arity of atoms

Example: Production rules of skip lists

written as LMNtal rules



When we repeatedly apply the rules above on 

and get a graph without 

, then the resulting graph is a skip list.

LMNtalGG: Context-freeness Assumption

- We assume all production rules are **context-free**
  - i.e., the LHS must consist of a single (non-terminal) atom
- and refer to **a set of production rules** as **a grammar**
  - Every non-terminal atom can be the **start symbol**
  - The sets of **non-terminal/terminal symbols** are automatically determined by the grammar (Def. 3.2)

$$\mathrm{N}(P) \triangleq \bigcup_{(\alpha\,:-\,\beta)\in P} \mathrm{Funct}(\alpha), \quad \mathrm{T}(P) \triangleq \mathrm{Funct}(P) \setminus \mathrm{N}(P).$$

Non-terminal symbols

Terminal symbols

LMNtalGG: Difference Types (Def. 3.3)

- Types with the concept of **difference** based on LMNtalGG
  - Generalizes the idea of difference lists to general data structures

The graph G has the type $\alpha - \beta$ with the grammar P

$$G :_P \alpha - \beta \overset{\text{def}}{\iff} \alpha \xrightarrow{P}{}^{*} (G, \beta)$$

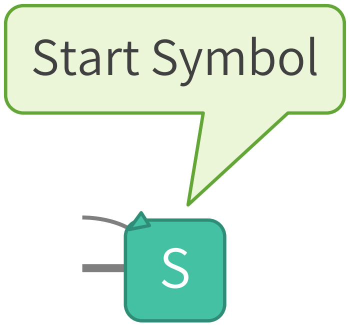
where
α consists of a single non-terminal atom
β consists only of non-terminal atoms
G doesn't include non-terminal atoms

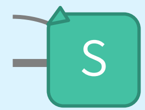
* We may omit P if it's clear from the context

LMNtalGG: Example

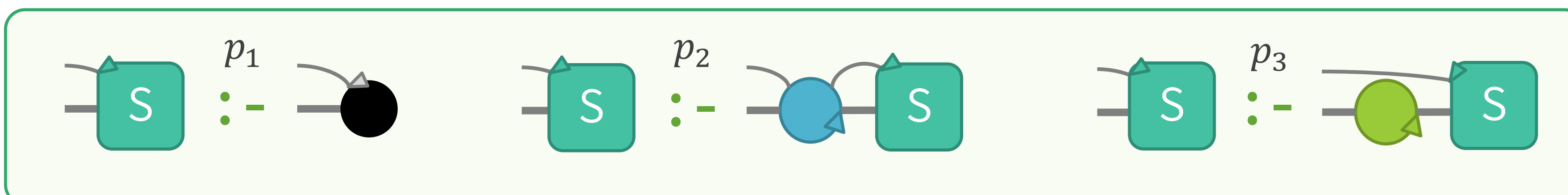
$$G :_P \alpha - \beta \overset{\text{def}}{\iff} \alpha \xrightarrow{P}{}^{*} (G, \beta)$$

Applying the production rules on the start symbol



In this case,  is the only non-terminal symbol and all the other atoms are terminal

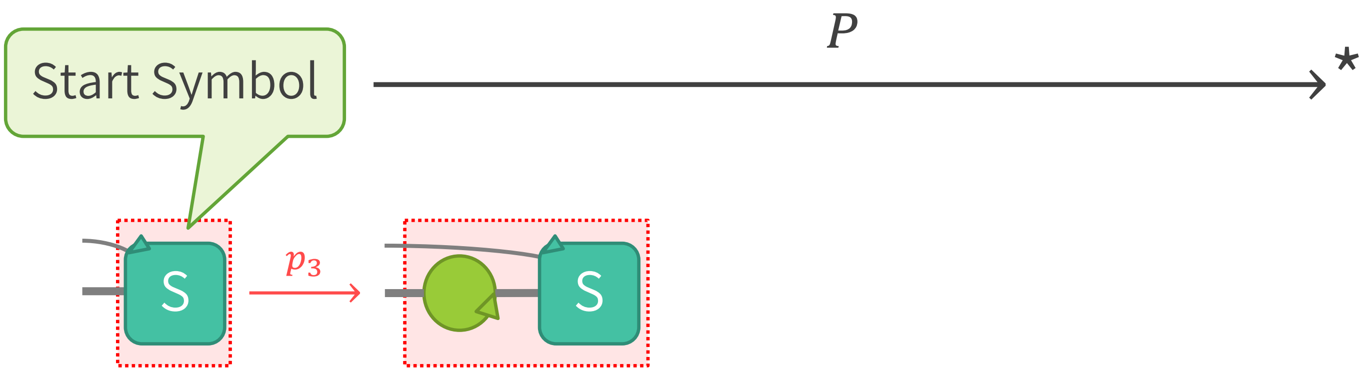
Production rules of skip lists



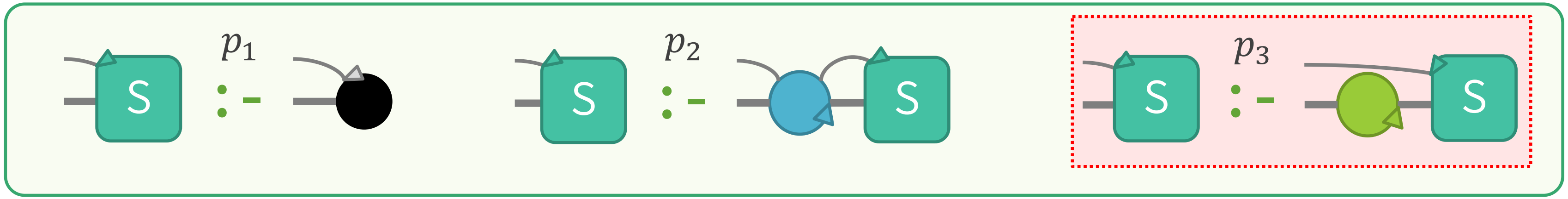
LMNtalGG: Example

$$G :_P \alpha - \beta \overset{\text{def}}{\iff} \alpha \xrightarrow{P}{}^{*} (G, \beta)$$

Applying the production rules on the start symbol



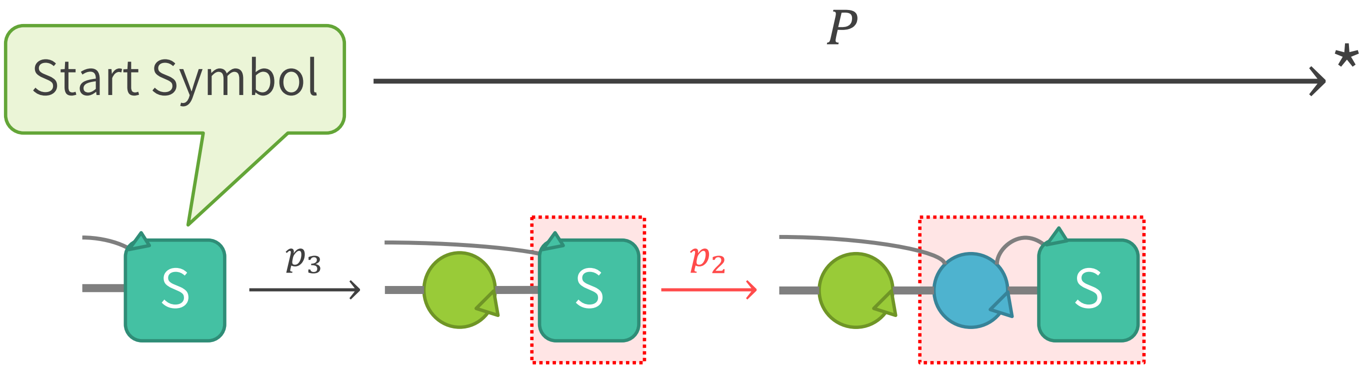
Production rules of skip lists



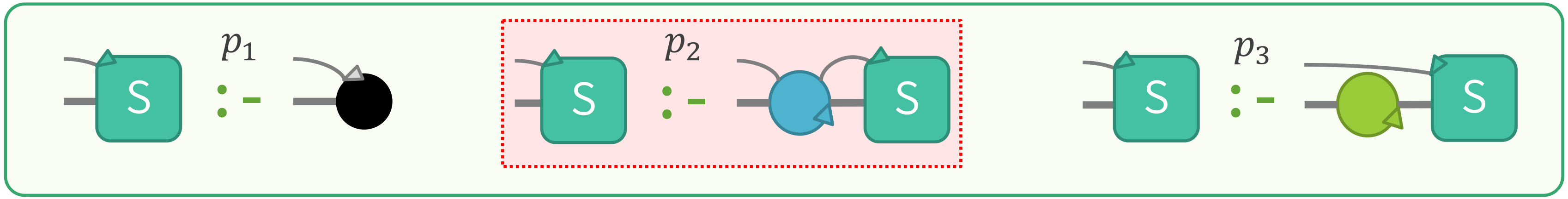
LMNtalGG: Example

$$G :_P \alpha - \beta \overset{\text{def}}{\iff} \alpha \xrightarrow{P}{}^{*} (G, \beta)$$

Applying the production rules on the start symbol



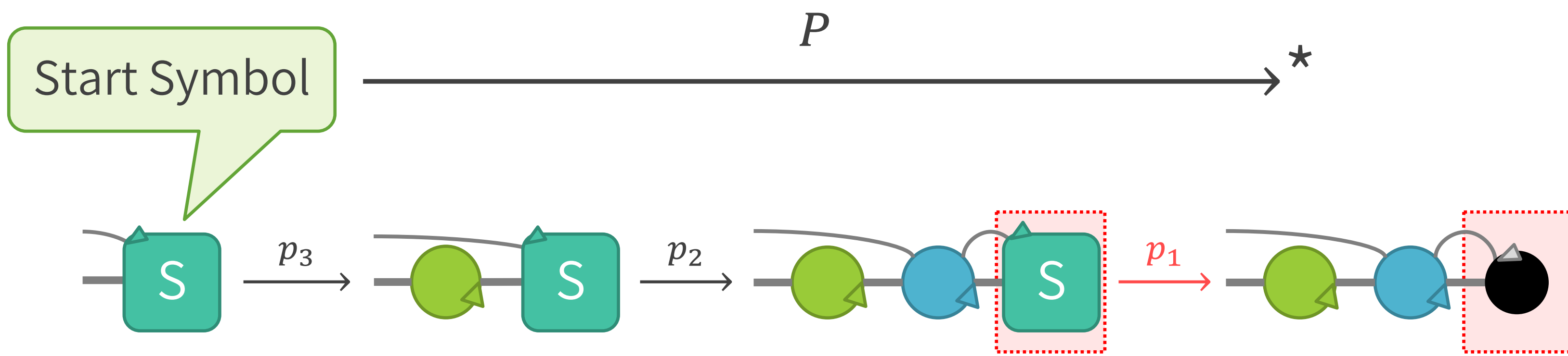
Production rules of skip lists



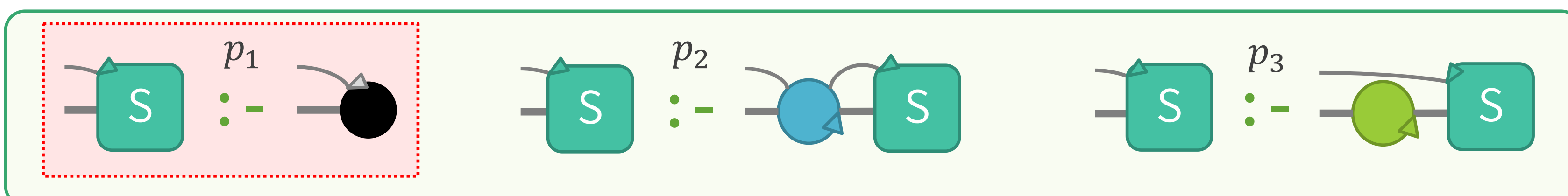
LMNtalGG: Example

$$G :_P \alpha - \beta \overset{\text{def}}{\iff} \alpha \xrightarrow{P}{}^{*} (G, \beta)$$

Applying the production rules on the start symbol



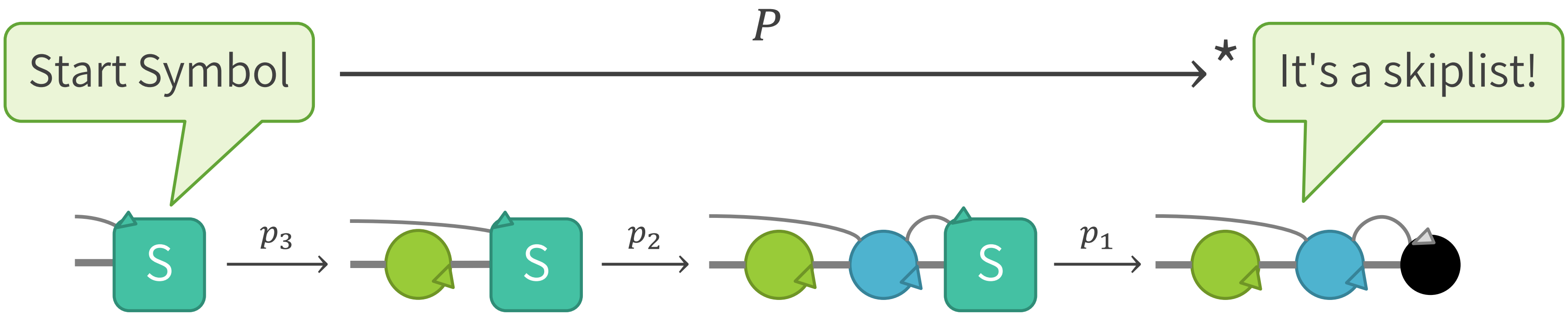
Production rules of skip lists

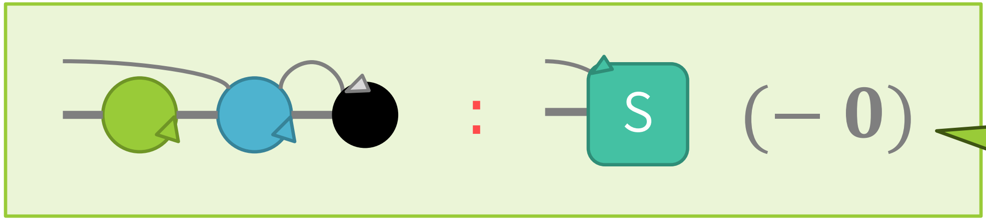


LMNtalGG: Example

$$G :_P \alpha - \beta \overset{\text{def}}{\iff} \alpha \xrightarrow{P}{}^{*} (G, \beta)$$

Resulting in a graph without non-terminal symbols



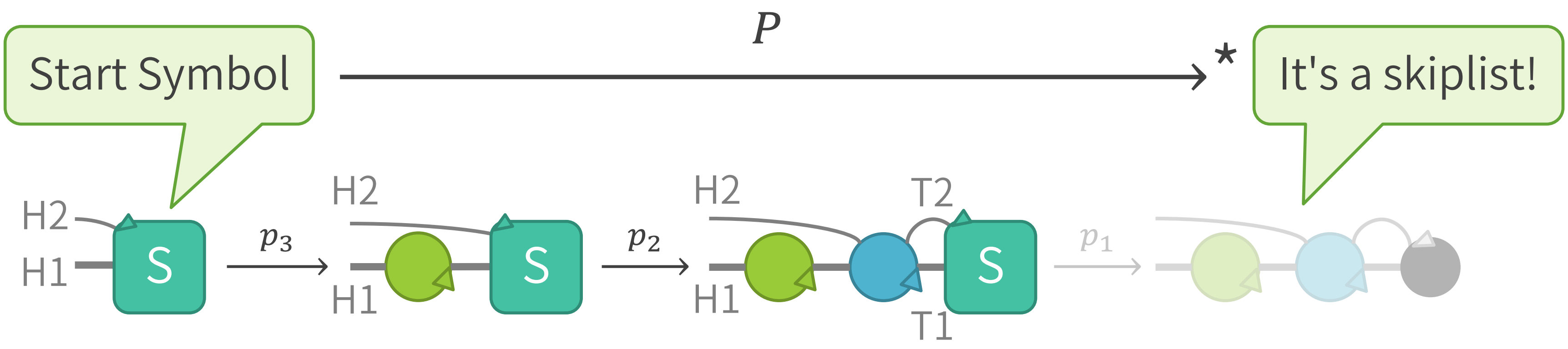


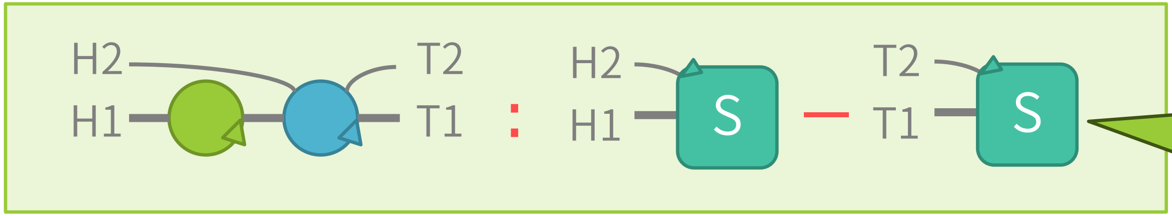
The graph on the left is a skip list

Difference Types

$$G :_P \alpha - \beta \overset{\text{def}}{\iff} \alpha \xrightarrow{P}{}^{*} (G, \beta)$$

Difference data structures can also be typed





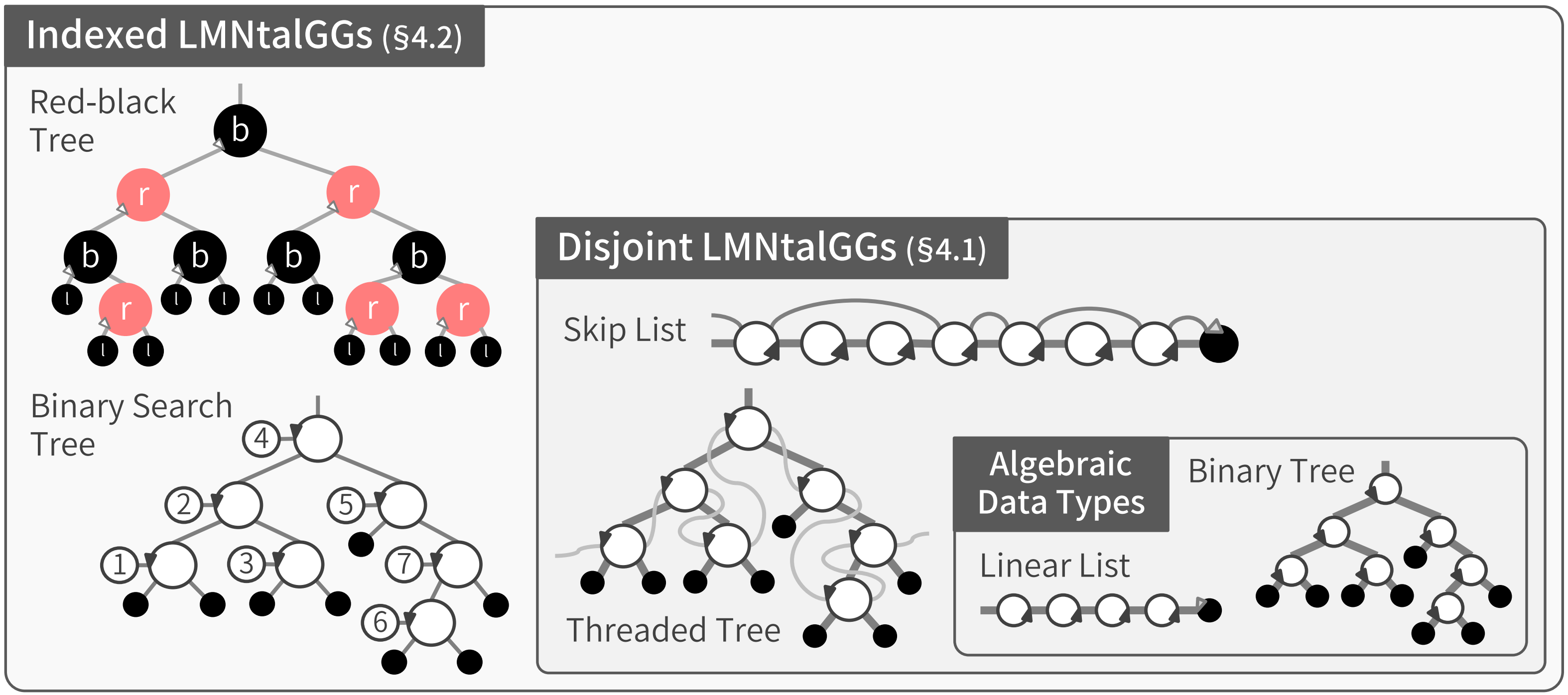
The graph on the left is a **difference** skip list

Outline

1. Target Language: LMNtal
2. LMNtalGG and Difference Types
3. Classifying LMNtalGGs: Disjoint & Indexed
4. Applications: Pattern Matching & Static Type Checking
5. Type Checking of Functional Atoms

Classifying LMNtalGGs

We provide two useful classes of LMNtalGGs

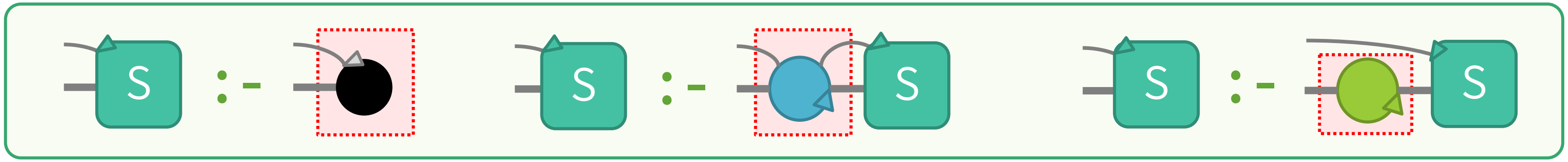


Basic Class: Disjoint LMNtalGG

A grammar P is **disjoint**

$\overset{\text{def}}{\Longleftrightarrow}$ the RHS of each rule contains **exactly one** terminal symbol that never appears in the RHSs of other production rules

- Called **inversion property** in standard type theory
  - The types of subterms can be inferred from the top-level constructor
- Example: The grammar of skip lists is disjoint

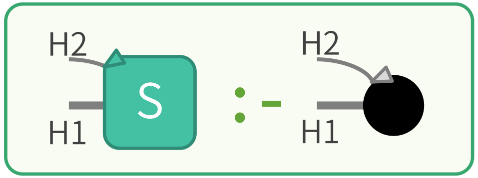


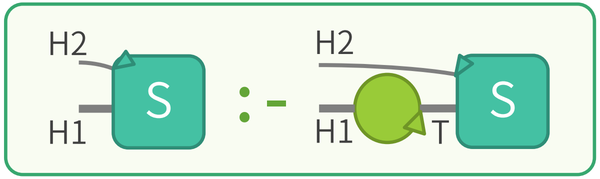
For disjoint LMNtalGGs, we can derive types of graphs uniquely

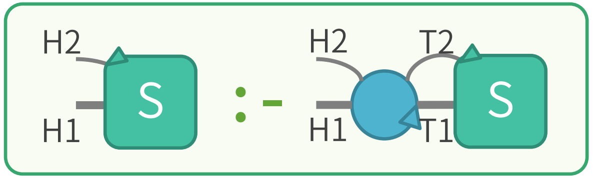
Type Derivation from Graphs (Section 3.2)

With disjoint LMNtalGG, types of graphs can be constructed:

1. Obtain typings of all terminal symbols from production rules (Lemma 3.6)



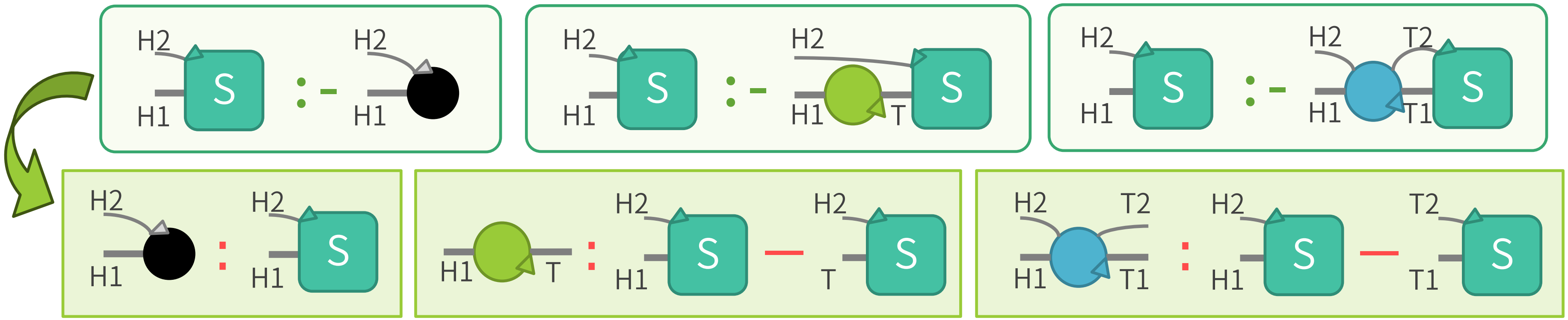




Type Derivation from Graphs (Section 3.2)

With disjoint LMNtalGG, types of graphs can be constructed:

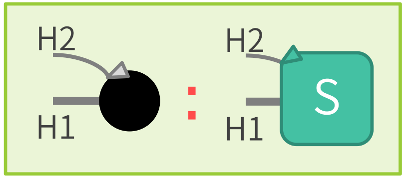
1. Obtain typings of all terminal symbols from production rules (Lemma 3.6)

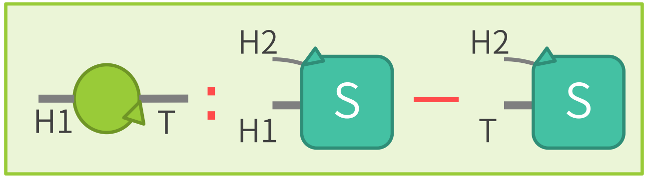


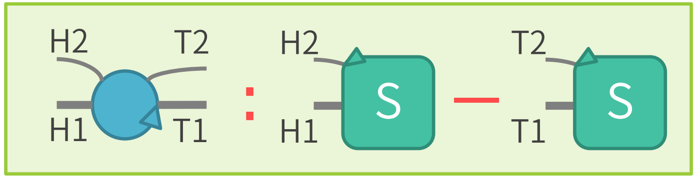
Type Derivation from Graphs (Section 3.2)

With disjoint LMNtalGG, types of graphs can be constructed:

1. Obtain typings of all terminal symbols from production rules (Lemma 3.6)





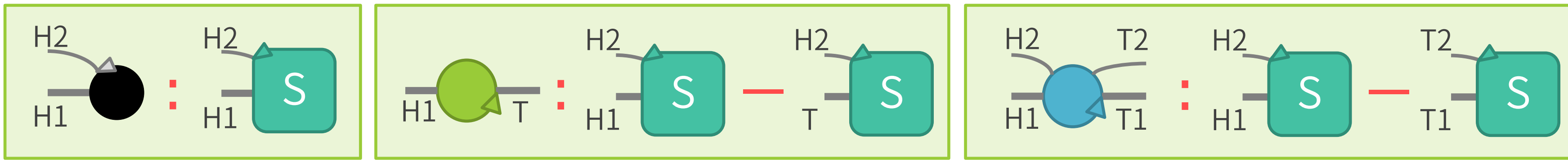


We use these typings as **axioms** of typing

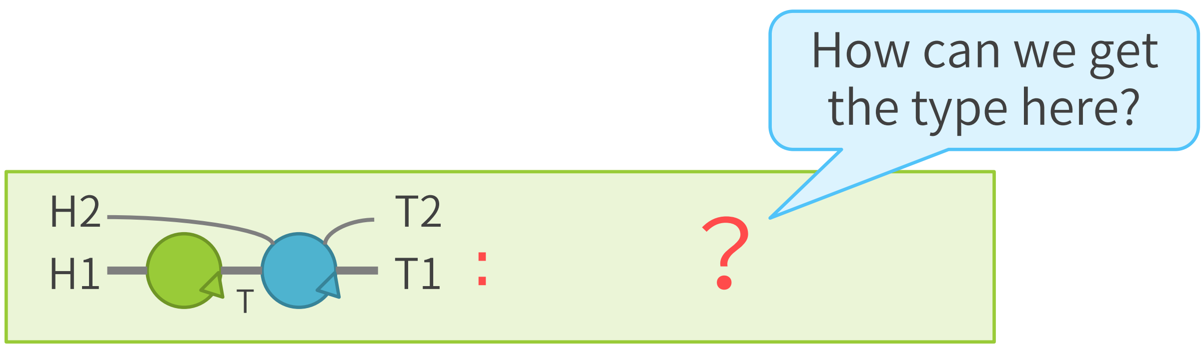
Type Derivation from Graphs (Section 3.2)

With disjoint LMNtalGG, types of graphs can be constructed:

1. Obtain typings of all terminal symbols from production rules (Lemma 3.6)



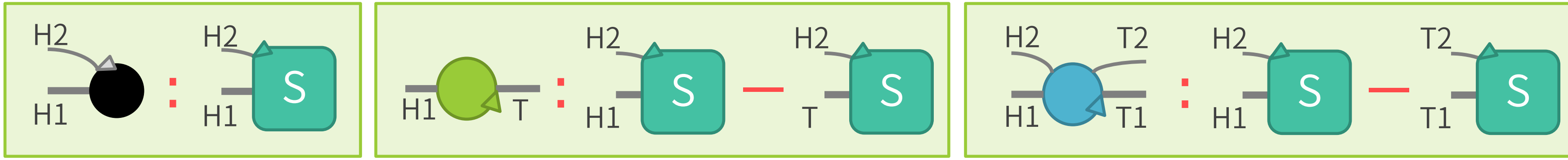
2. Construct the type of graph from subgraphs' typings (Theorem 3.7)



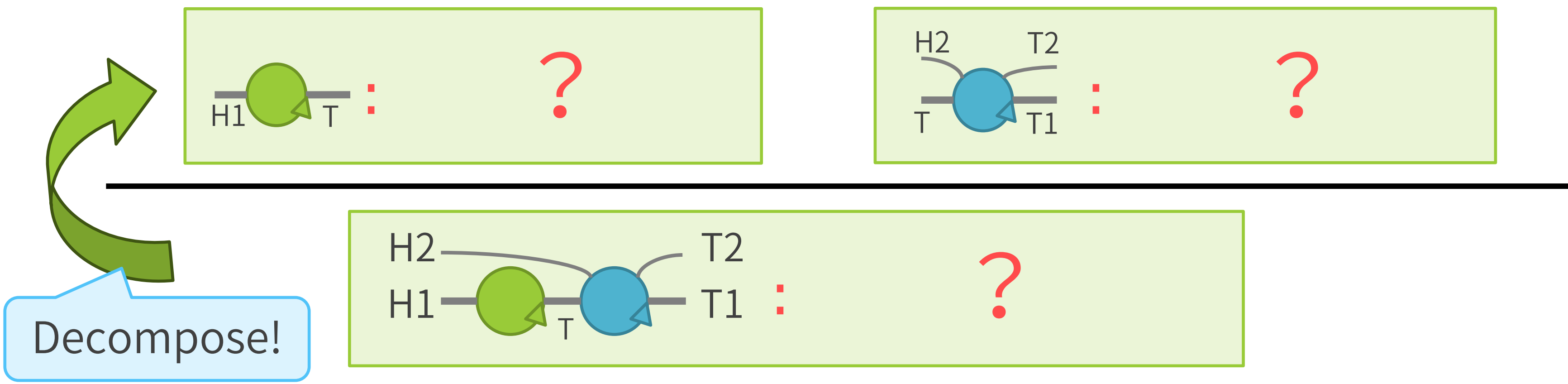
Type Derivation from Graphs (Section 3.2)

With disjoint LMNtalGG, types of graphs can be constructed:

1. Obtain typings of all terminal symbols from production rules (Lemma 3.6)



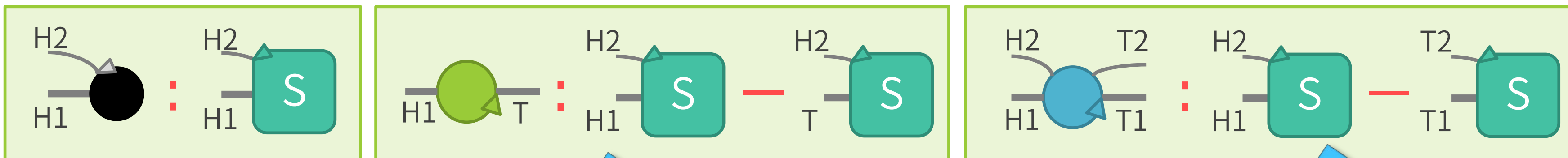
2. Construct the type of graph from subgraphs' typings (Theorem 3.7)



Type Derivation from Graphs (Section 3.2)

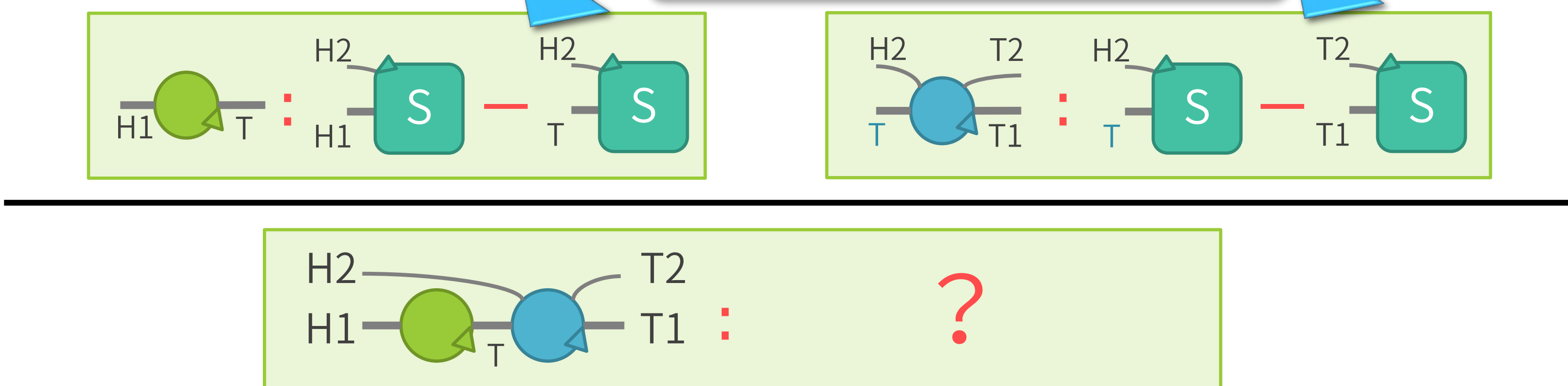
With disjoint LMNtalGG, types of graphs can be constructed:

1. Obtain typings of all terminal symbols from production rules (Lemma 3.6)



2. Construct the type of gr… n … m 3.7)

We know these types!

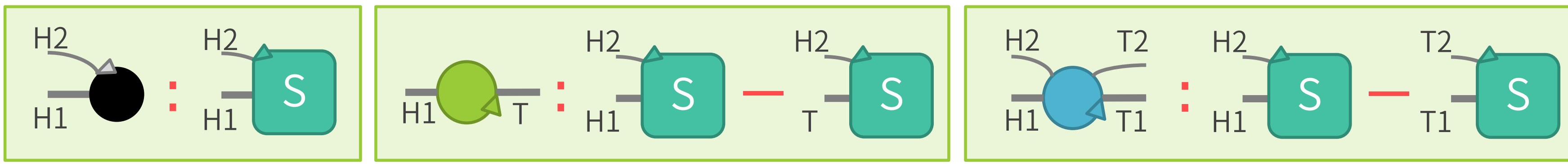


* We rename link names as needed (α-conversion, Prop. 3.5)

Type Derivation from Graphs (Section 3.2)

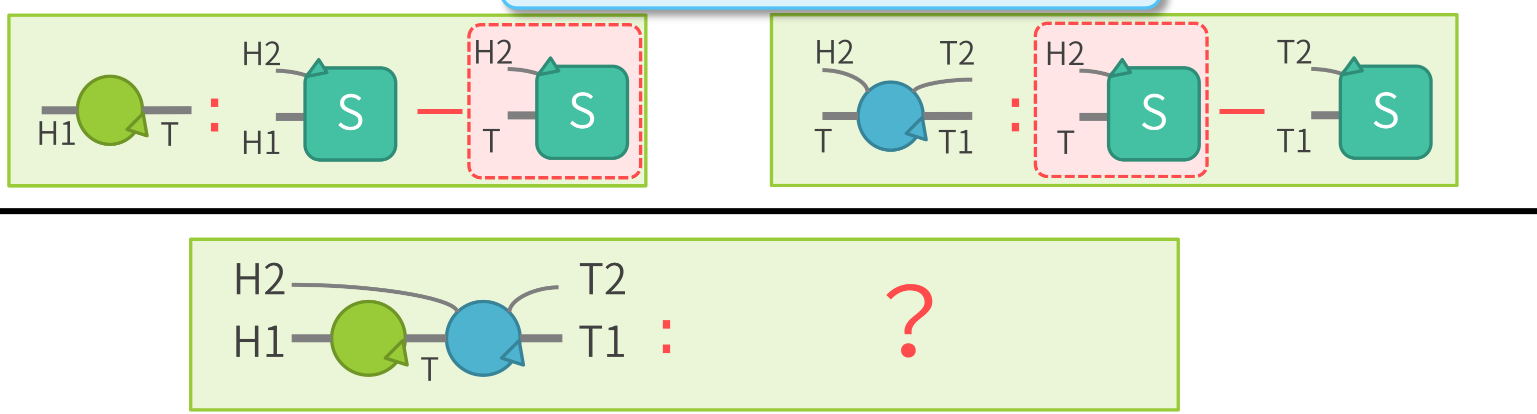
With disjoint LMNtalGG, types of graphs can be constructed:

1. Obtain typings of all terminal symbols from production rules (Lemma 3.6)



2. Construct the type of gra[…]s (Theorem 3.7)

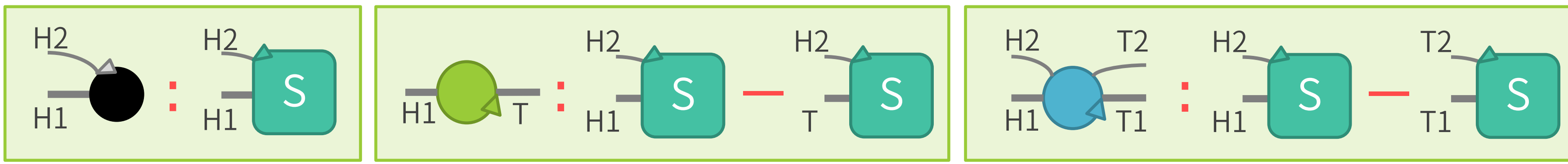
These are the same!



Type Derivation from Graphs (Section 3.2)

With disjoint LMNtalGG, types of graphs can be constructed:

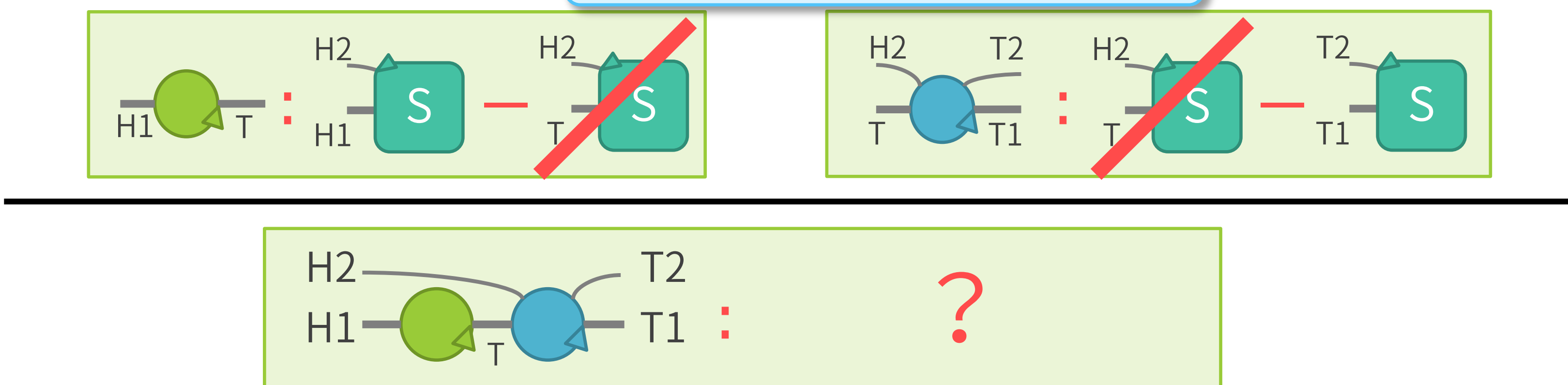
1. Obtain typings of all terminal symbols from production rules (Lemma 3.6)



2. Construct the type of gra

Cancel them

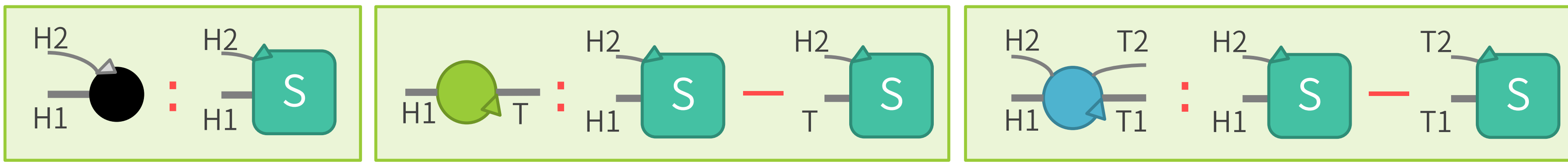
s (Theorem 3.7)



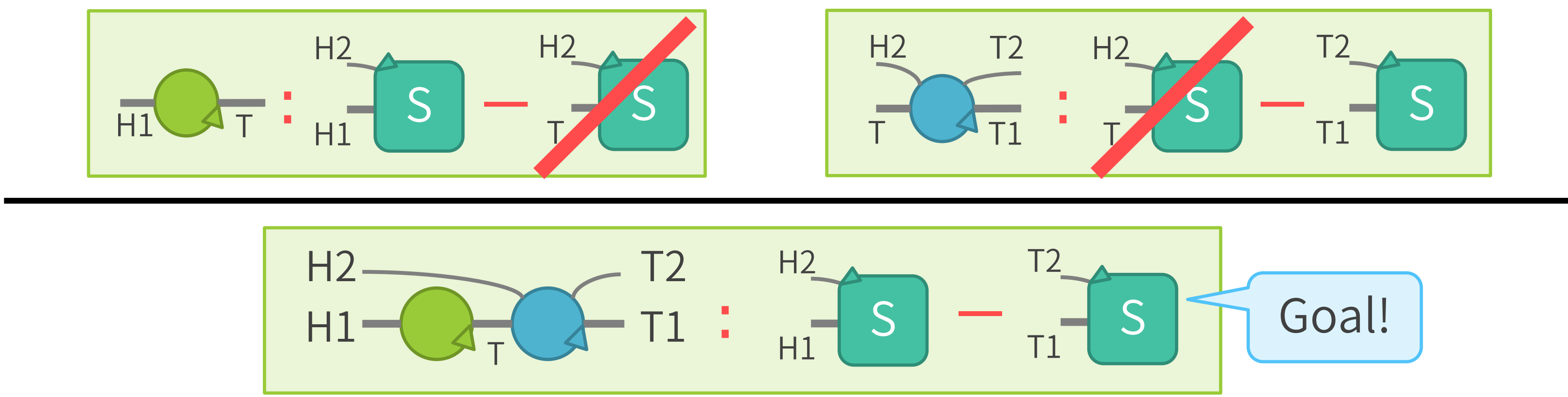
Type Derivation from Graphs (Section 3.2)

With disjoint LMNtalGG, types of graphs can be constructed:

1. Obtain typings of all terminal symbols from production rules (Lemma 3.6)

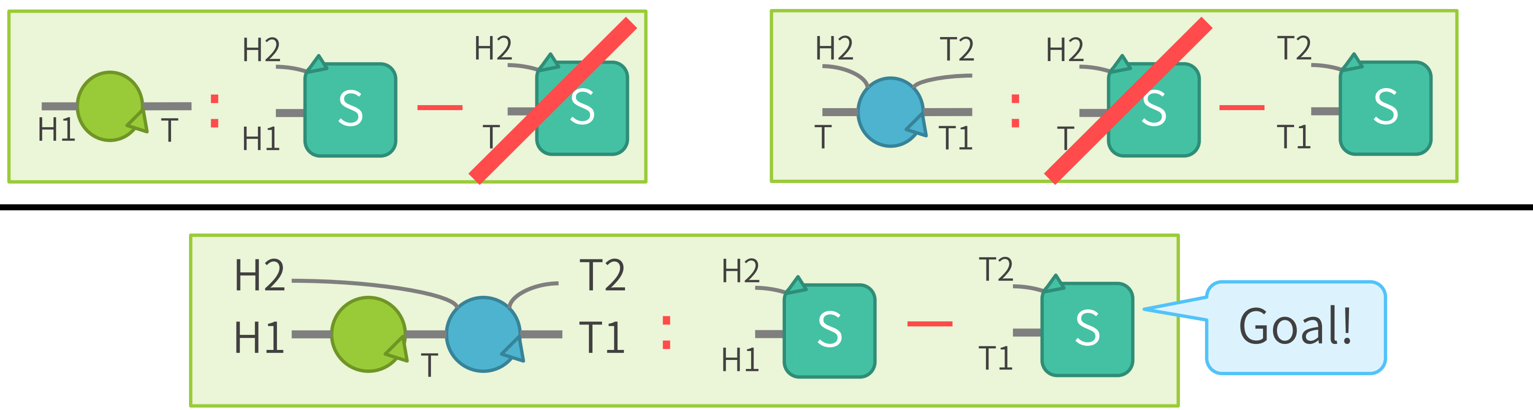


2. Construct the type of graph from subgraphs' typings (Theorem 3.7)



Remarks: Type Derivation from Graphs

- Costs linear time w.r.t. # of atoms
- Similar to the **cut rule** in the **Sequent Calculus**
  - Analogy: $\begin{cases} \textbf{difference types} \leftrightarrow \textbf{sequents}, \\ \textbf{type composition} \leftrightarrow \textbf{cut rule} \end{cases}$

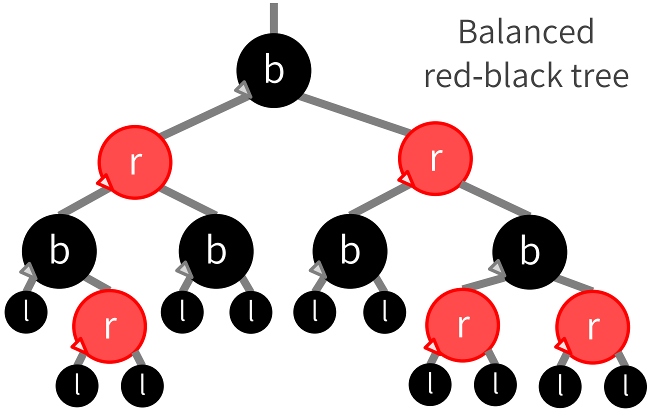


Broader Class of LMNtalGG

- **Indexed LMNtalGGs**: inspired by indexed grammars†

Non-terminal symbols can be equipped with integers as indices

✔ Shapes with numeric constraints (e.g., balanced red-black trees)



Balanced red-black tree

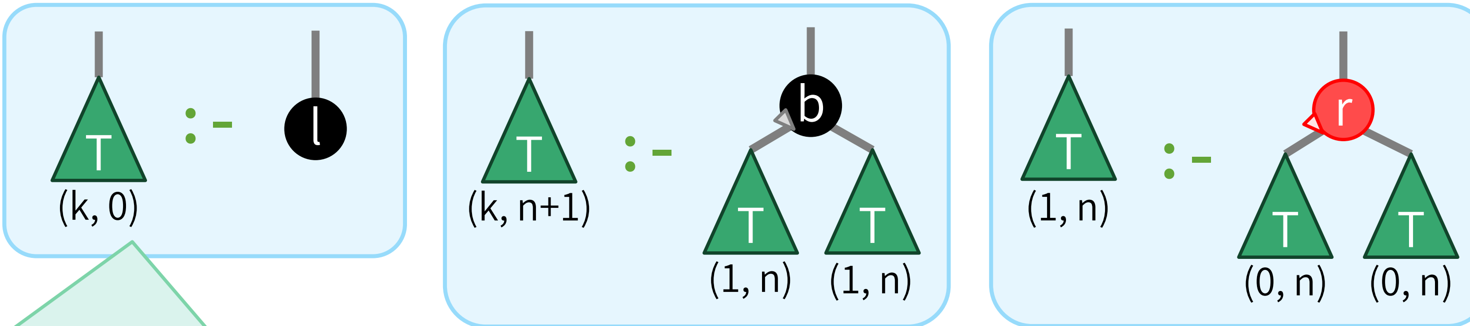
Requirements for red-black trees

1. The root and leaves are b
2. r 's children are b
3. # of b on the path from the root to a leaf (black height) is a constant

numeric constraint

† A. V. Aho: Indexed Grammars—An Extension of Context-Free Grammars, J. ACM, 15(4), 1968.

Red-black Trees with Indexed LMNtalGGs



Indices:
1. Color of the root (0: black, 1: red or black)
2. Black height

This grammar can be considered disjoint
(when the indices are ignored)

Requirements for red-black trees

1. The root and leaves are b
2. r's children are b
3. # of b on a path from the root to a leaf (black height) is a constant

Outline

1. Target Language: LMNtal
2. LMNtalGG and Difference Types
3. Classifying LMNtalGGs: Disjoint & Indexed
4. Applications: Pattern Matching & Static Type Checking
5. Type Ch… to…

Dynamic Extension

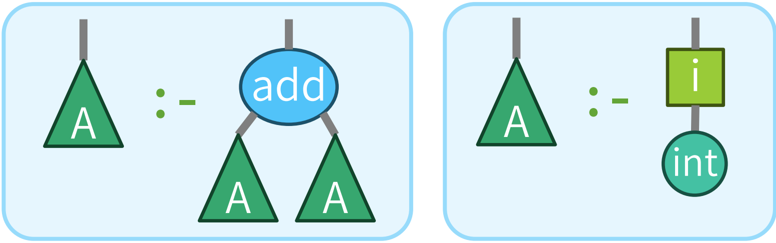
Static Extension

Application (1) Pattern Matching

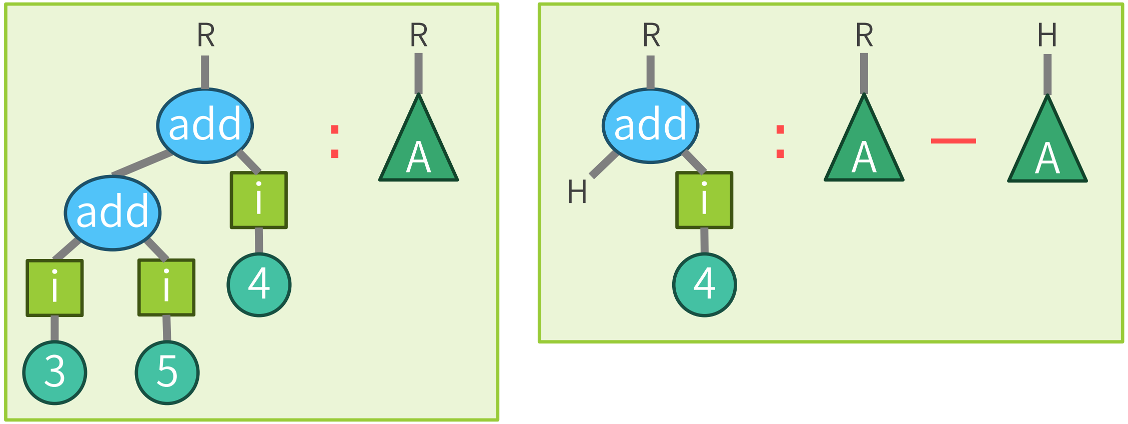
Disjoint LMNtalGG supports tree-shaped (difference) structures

Example: binary trees consisting of add nodes (addtree)

Grammar

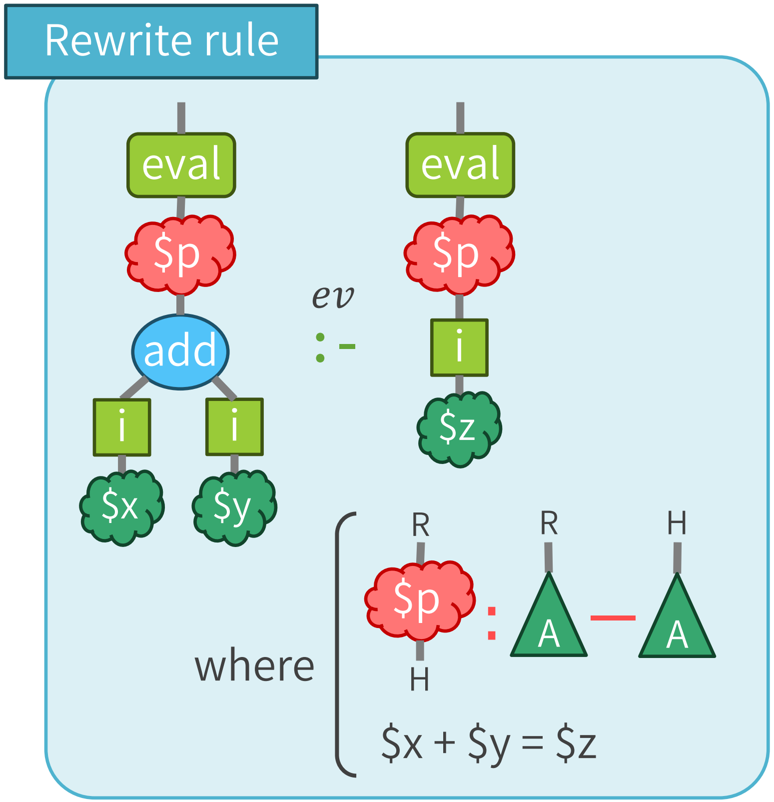


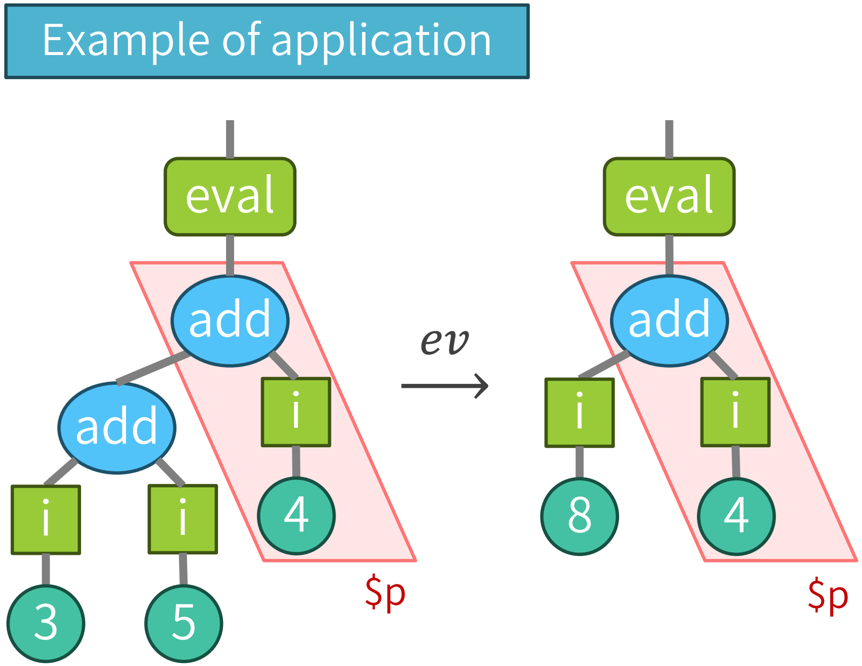
Example Typings



Application (1) Pattern Matching

We can describe pattern matching on DDSs with LMNtalGGs



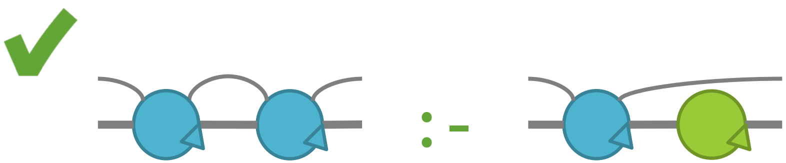


This simulates evaluation contexts

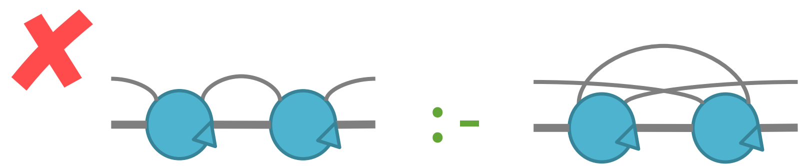
Application (2) Type Checking of Rules

Checks if the application of a rule **preserves** types of graphs

$$\forall G, G'. \quad G : \tau \ \land \ G \xrightarrow{R} G' \implies G' : \tau$$



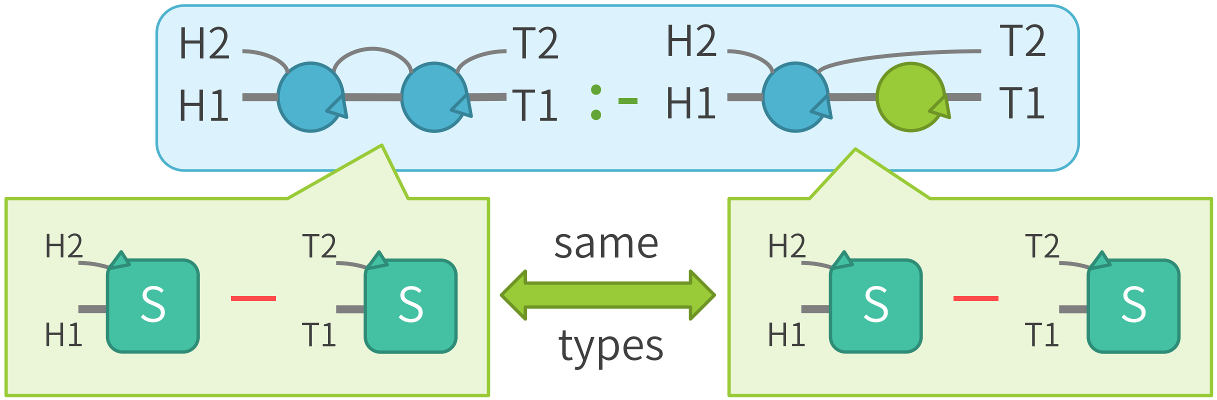
Applying this rule on a skip list always results in a skip list



Applying this rule on a skip list may result in not a skip list

Application (2) Type Checking of Rules

To confirm that a given rule **preserves** types of graphs,



check the types of both sides are the same (Theorem 6.2)

- i.e., simply perform type derivation for both sides (linear time w.r.t. # of atoms)
- Intuition: The type of the whole graph will not change
  because it just rewrites a difference skip list to a difference skip list

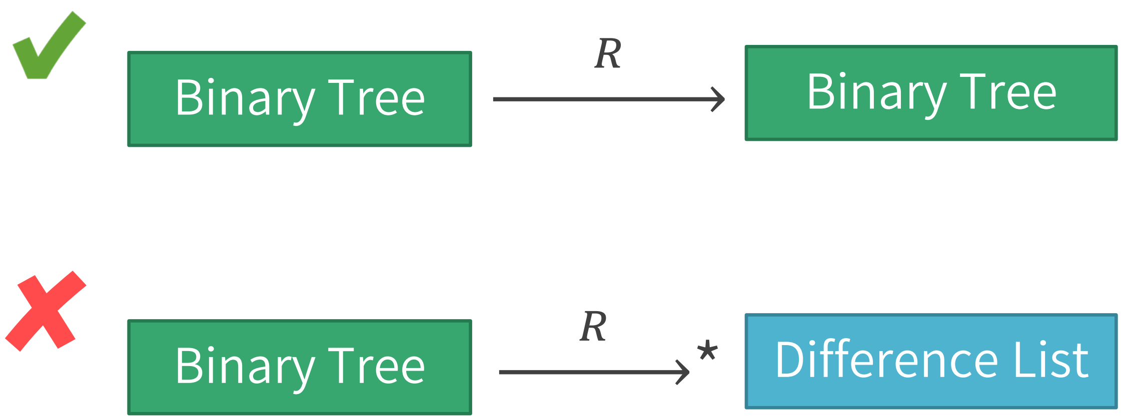
Outline

1. Target Language: LMNtal
2. LMNtalGG and Difference Types
3. Classifying LMNtalGGs: Disjoint & Indexed
4. Applications: Pattern Matching & Static Type Checking
5. Type Checking of Functional Atoms

Multi-step/shape-changing operations

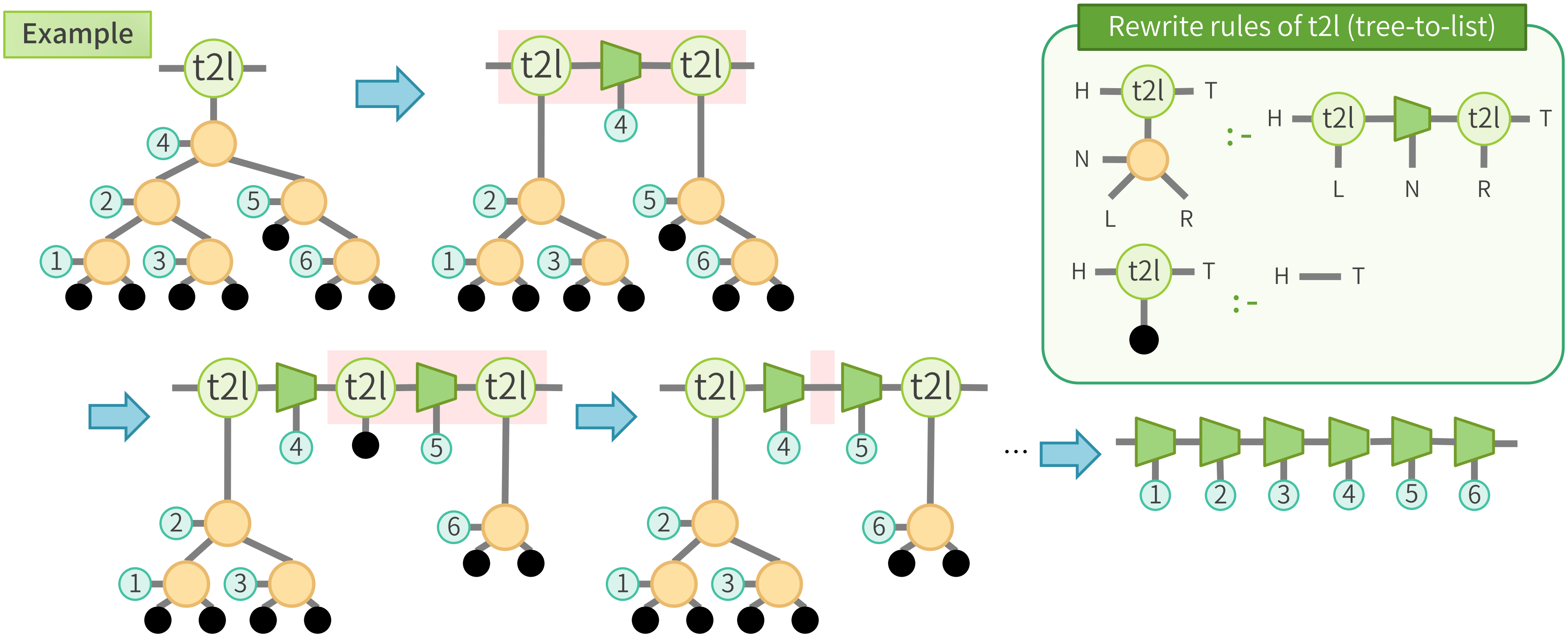
- In most of the existing typing frameworks for graphs,
  - Type Safety: “Rewrite rules will **never destroy** the shape of graphs”

    → Operations that may result in other types were out of scope



Functional Atoms: Example

- Graph nodes that behave like functions (in functional languages)
- Common design pattern (LMNtal has **no functions a-priori**)



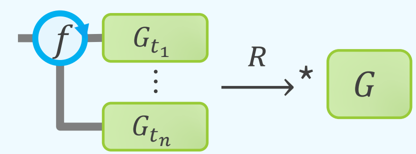
Expected Property of Functional Atoms

- We expect t2l satisfies ... Binary Tree $\xrightarrow{R}{}^{*}$ Difference List
  - If it receives a binary tree, it eventually returns a difference list
- In general, this property can be formalized as:

f is **a functional atom** that takes types $t_1, \ldots, t_n$ and returns type T

$\overset{\text{def}}{\iff}$

For any graphs $G_{t_1}, \ldots, G_{t_n}$ having types $t_1, \ldots, t_n$ (resp.),
if $(f, G_{t_1}, \ldots, G_{t_n})$ can be reduced to G
that includes no f atoms, then G has the type T.



We write this property as $F: t_1; \cdots; t_n \rightarrowtail T$

e.g., `t2l(P,T,H)` : `tree(P)` $\rightarrowtail$ `list(H)-list(T)`

Checking Functional Atoms with LMNtalGG

- If the input types include **no** differences (i.e., of the form $\tau - \mathbf{0}$),

  To check that the atom F has the functional property,

  $$F\colon \boxed{\tau_1; \ldots; \tau_n} \rightarrowtail \alpha - \beta$$

  $\tau_1, \ldots, \tau_n$ are **inputs**

  we assume the following typing

  $$F\colon \ \alpha - (\beta, \boxed{\tau_1, \ldots, \tau_n})$$

  $\tau_1, \ldots, \tau_n$ are **holes**

  and confirm that the rules preserve types

- For details (esp., correctness), see our previous work†

† N. Yamamoto et al.: Engineering Grammar-based Type Checking for Graph Rewriting Languages. IEEE ACCESS, 10, 2022.

Related Work

- **Graph Types**[†1]: Based on regular expressions
- **Structured Gamma**[†2] and **Shape Types**[†3]
  - Based on context-free graph grammars

a subset that satisfies completeness

- **Refinement Types**[†4]: Types with numeric constraints
  - Implemented on Liquid Haskell with type inference[†5]
- **Typed Prolog**[†6]: Difference lists are typable
  - Types (e.g., list, int) with Modes (direction)

†1 P. Fradet et al.: Structured Gamma, Science of Computer Programming, 31(2), 1998.
†2 N. Klarlund et al.: Graph Types, Proc. POPL'93.
†3 P. Fradet et al.: Shape types, Proc. POPL'97.
†4 N. Vazou et al.: Refinement types for Haskell, SIGPLAN Not., 49(9), 2014.
†5 P.M. Rondon et al.: Liquid types, SIGPLAN Not., 43(6), 2008.
†6 T.K. Lakshman et al.: Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System, Proc. ICLP'90.

Conclusion

1. We proposed:
   i. **LMNtalGG** as graph grammar on LMNtal
   ii. **Difference Types** on LMNtalGG to deal with DDSs
2. Introduced two applications of LMNtalGG:
   i. **Pattern Matching** on DDSs
   ii. **Static Type Checking** of Rules
3. Introduced Functional Atoms
   to handle **multi-step and/or shape-changing operations**

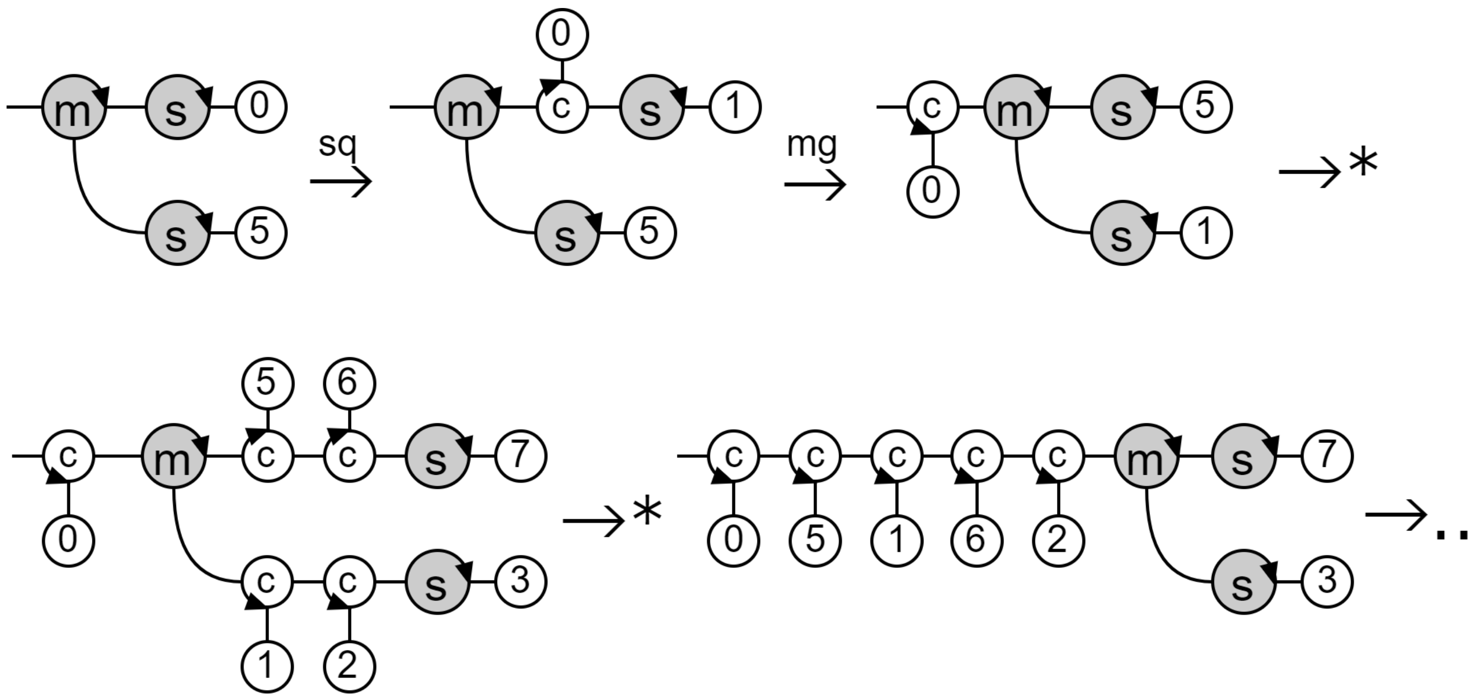
Future Work

1. **Full implementation for type checking method**
   - Adding indices and functional atoms to our PoC implementation
2. **Expanding the target language**
   - Adding some useful constructs (e.g., membranes, hyperedges)
3. **Exploring connections with other frameworks**
   - Our framework naturally supports non-terminating computation
     - History: LMNtal was made to model concurrency and non-determinism
     - We have non-terminating functional atoms in the paper

Spare Slides

Grammar-based Pattern Matching and Type Checking
for Difference Data Structures

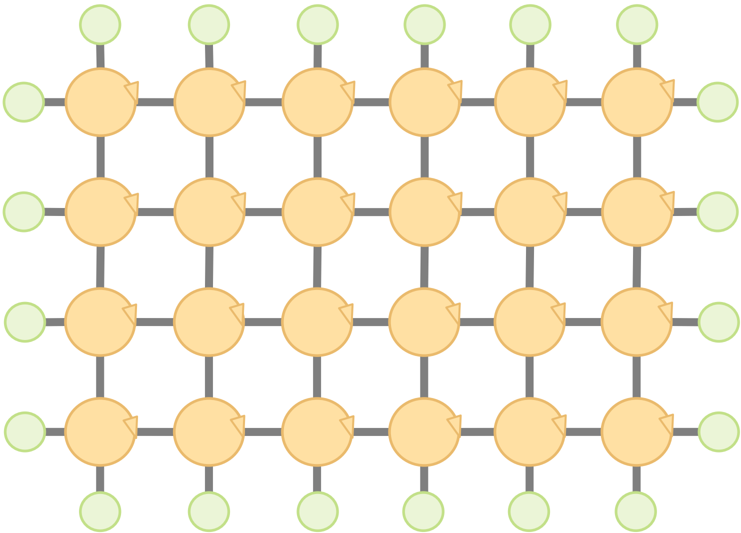
Functional Atom: Not Terminating



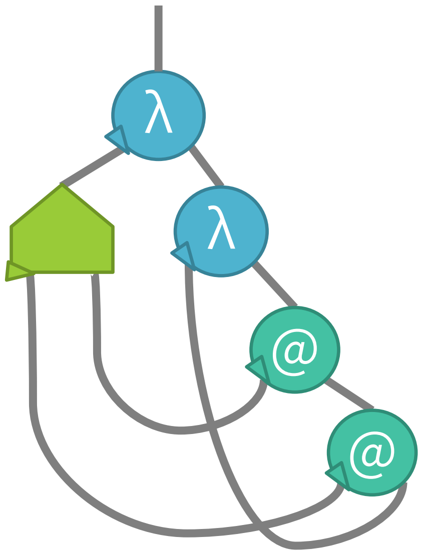
- Atom s : Outputs a counting-up stream [0, 1, 2, ...]
- Atom m : Receives two streams & Merges them

Deque of Links as Index: Examples

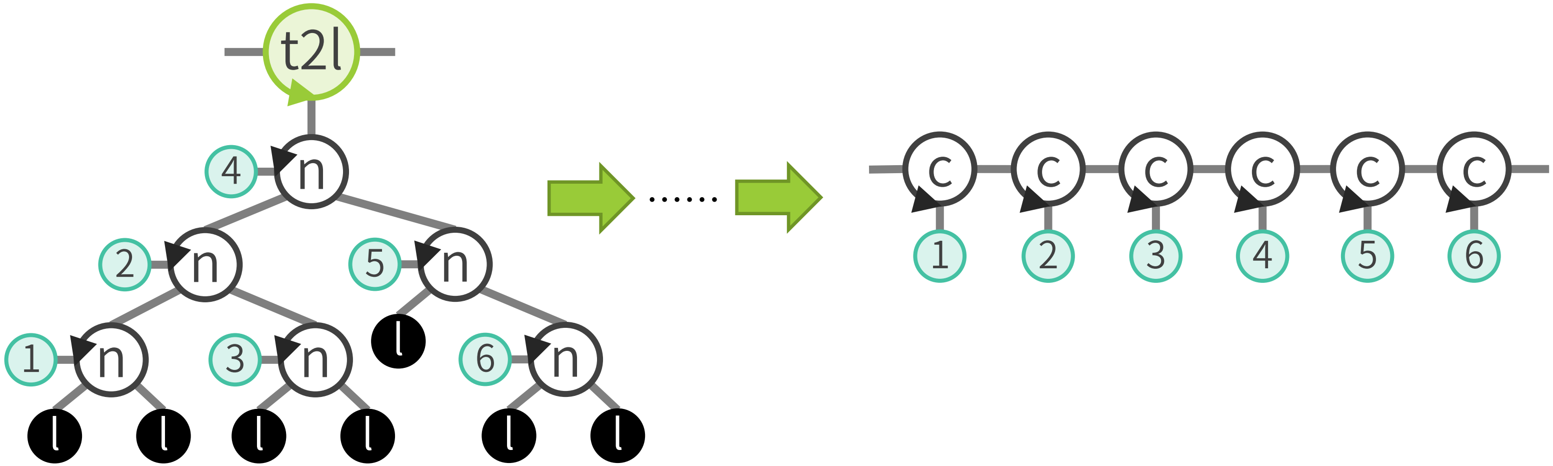
Grid graph

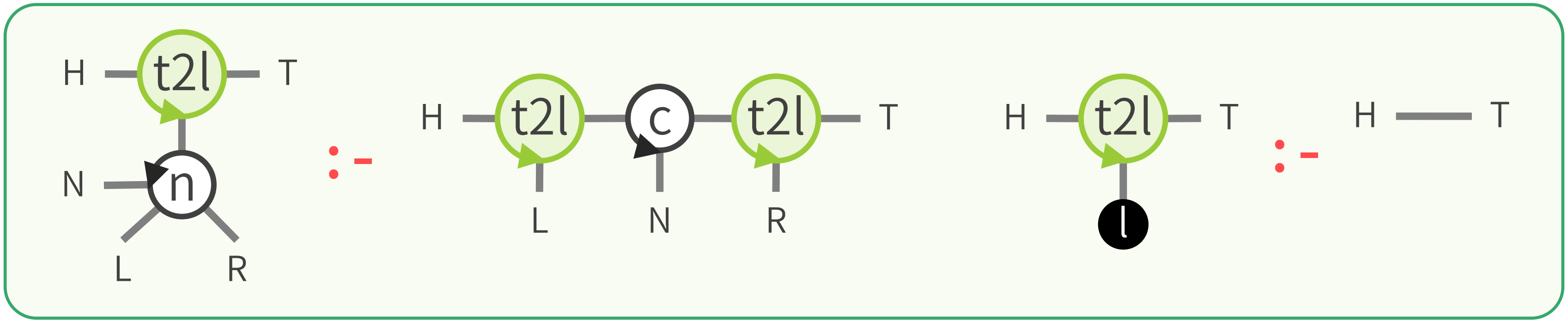


Lambda term
λfx.f(fx)

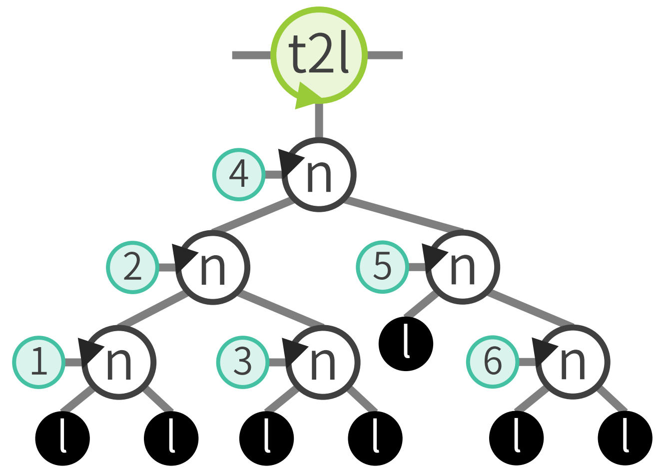


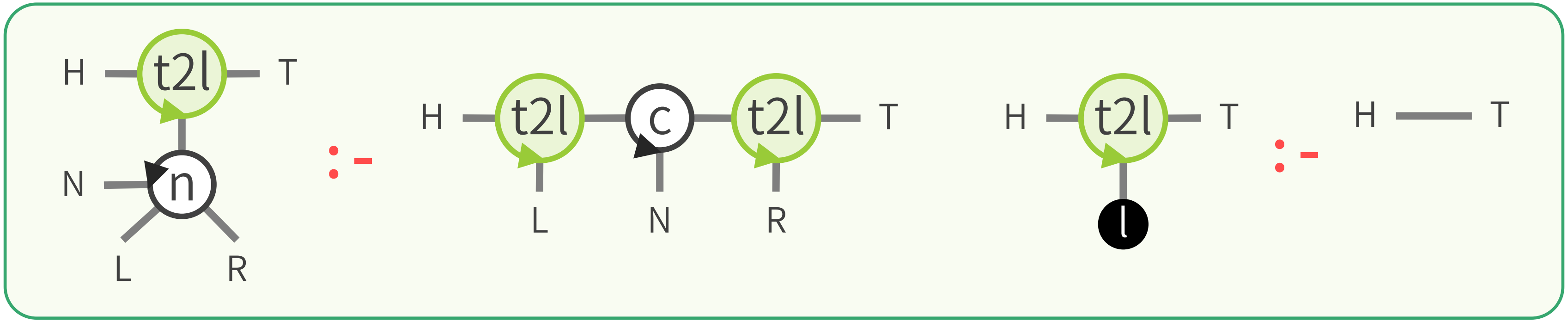
Functional Atom: t2l (tree-to-list)



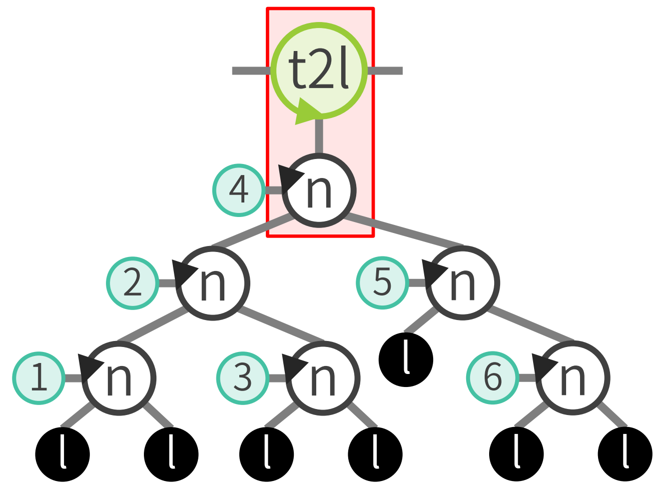


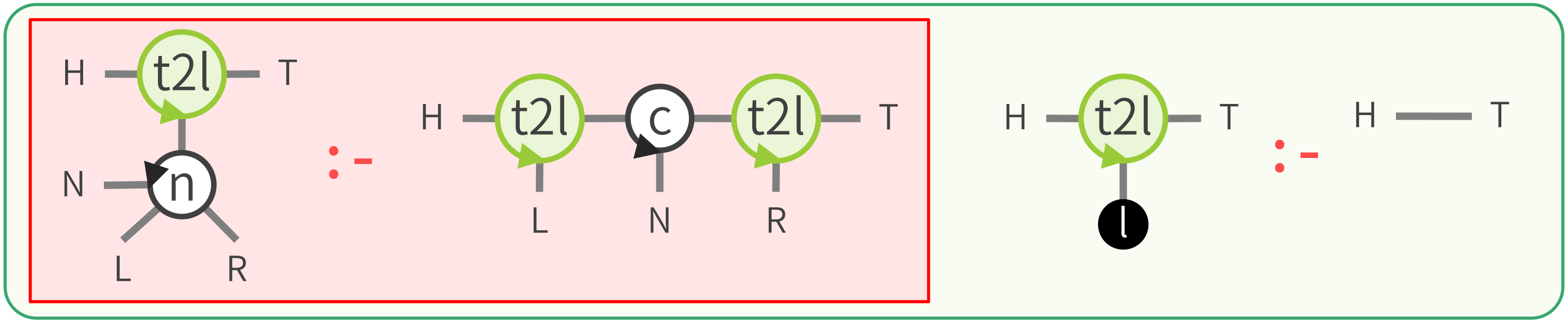
Functional Atom: t2l (tree-to-list)



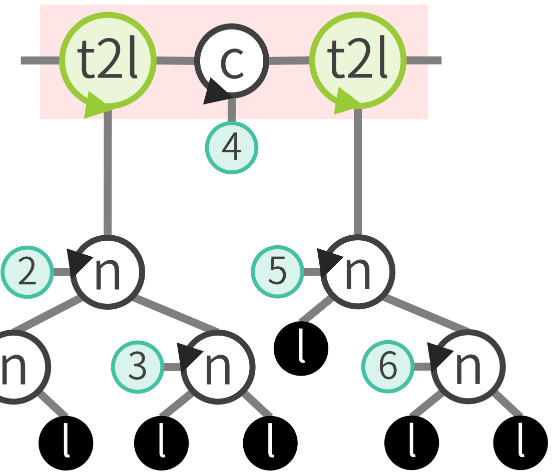


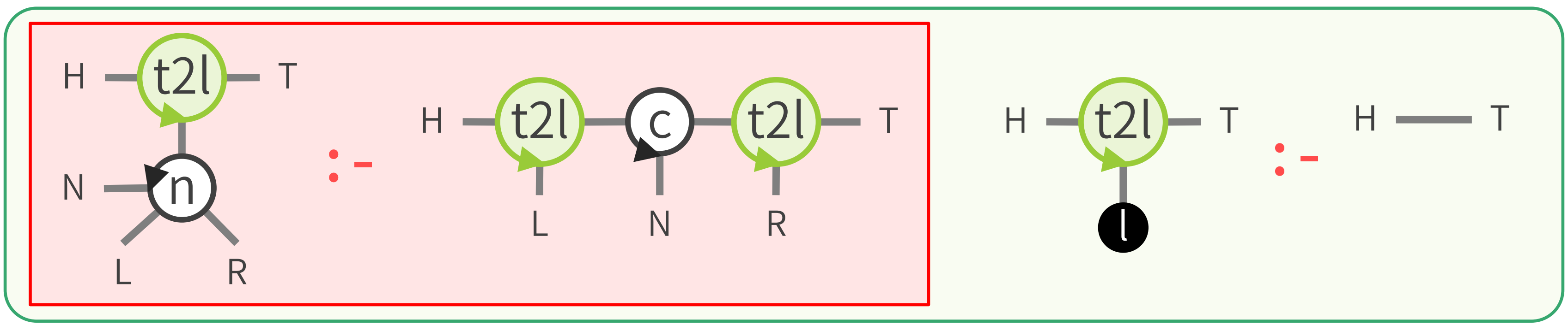
Functional Atom: t2l (tree-to-list)



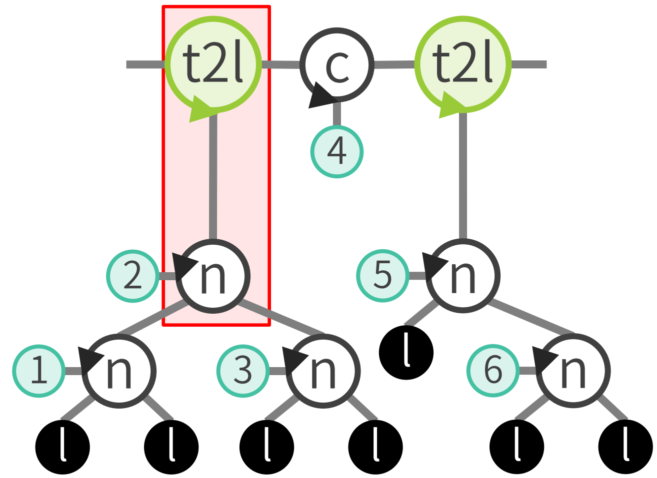


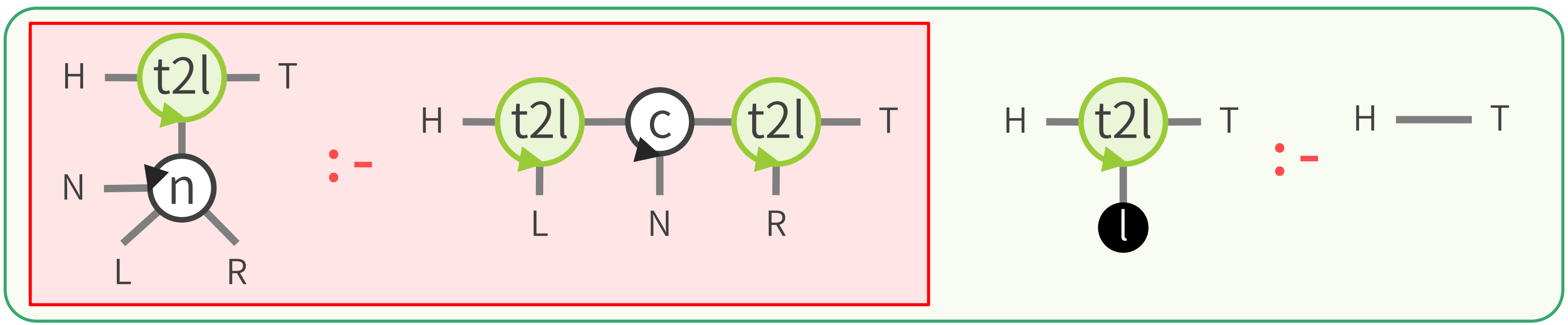
Functional Atom: t2l (tree-to-list)



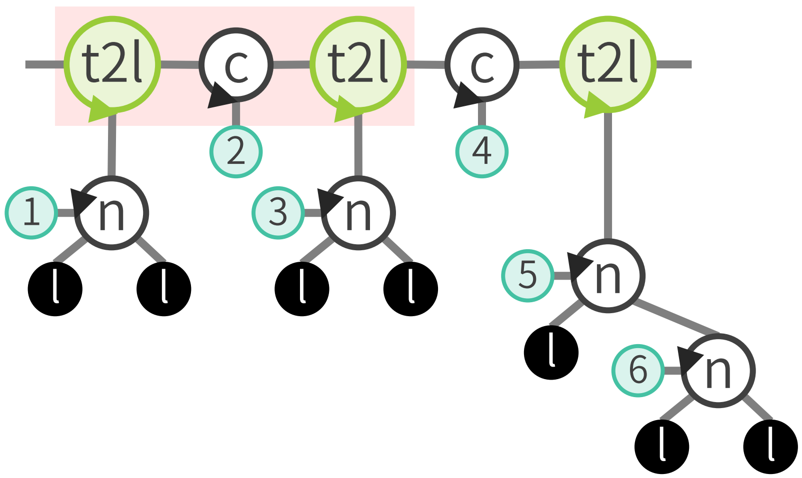


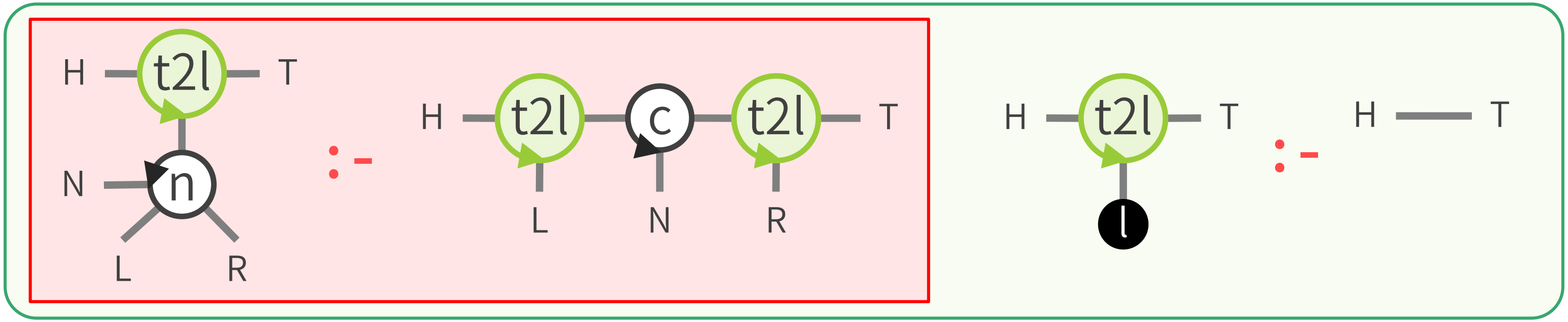
Functional Atom: t2l (tree-to-list)



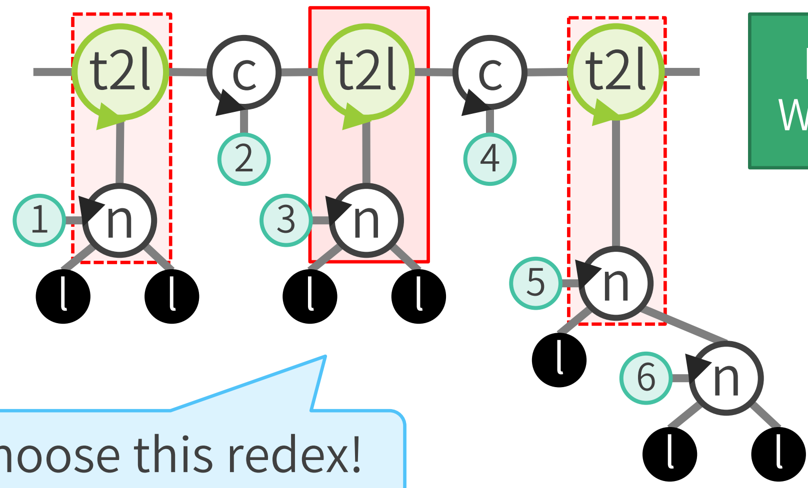


Functional Atom: t2l (tree-to-list)



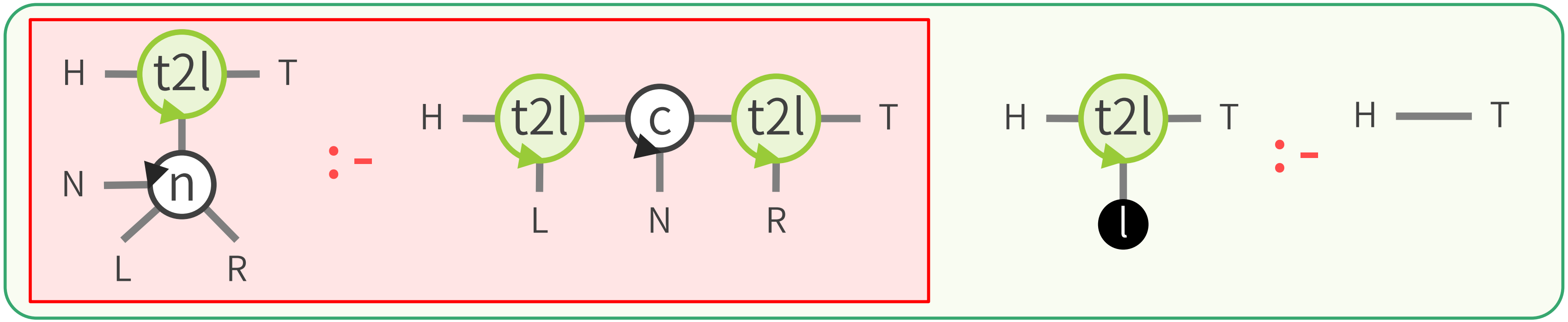


Functional Atom: t2l (tree-to-list)

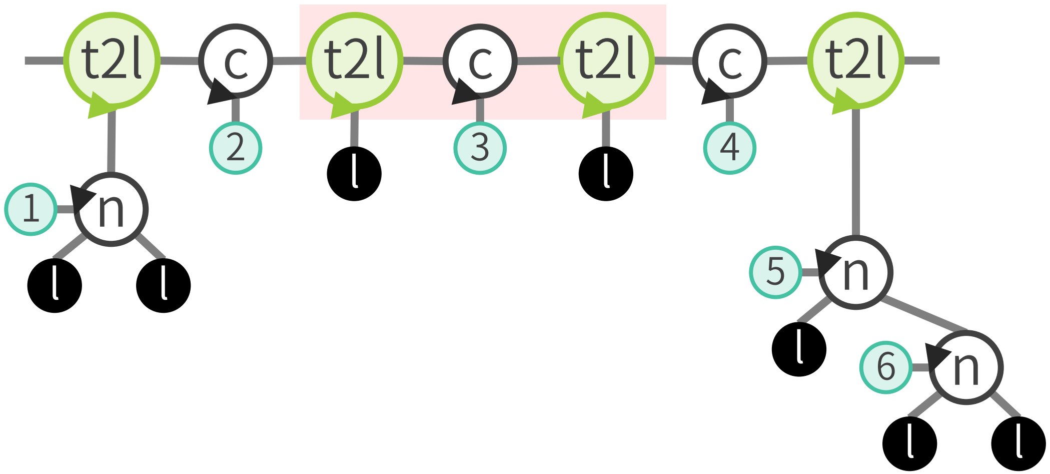


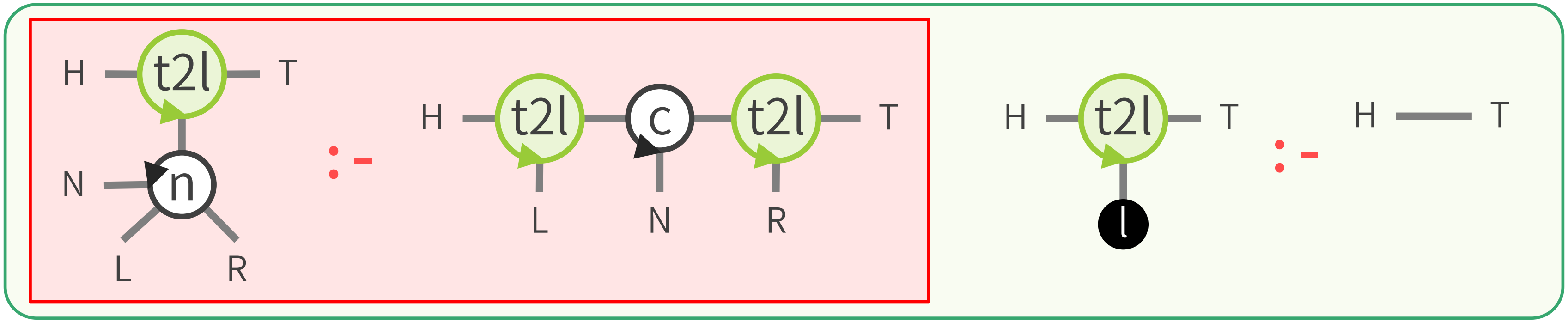
LMNtal is non-deteministic =
We can rewrite from either way

We choose this redex!

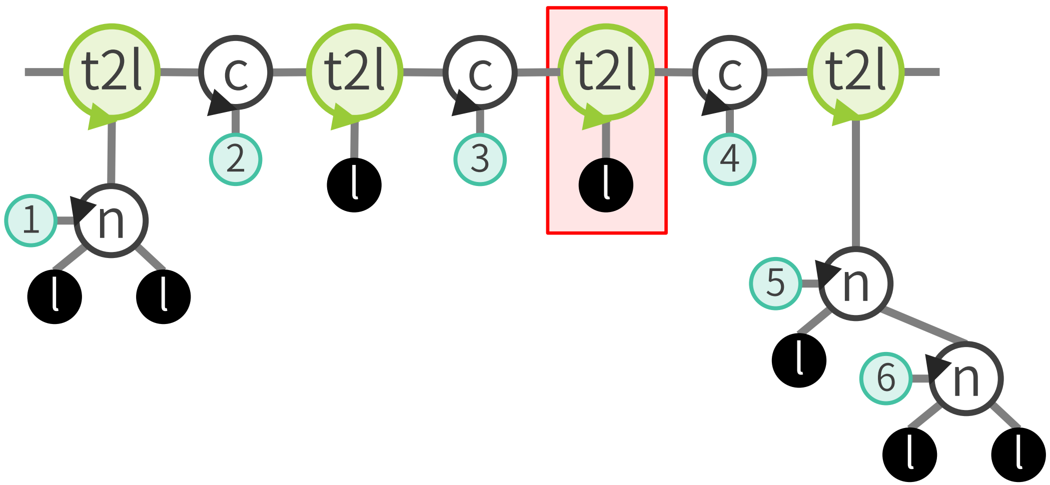


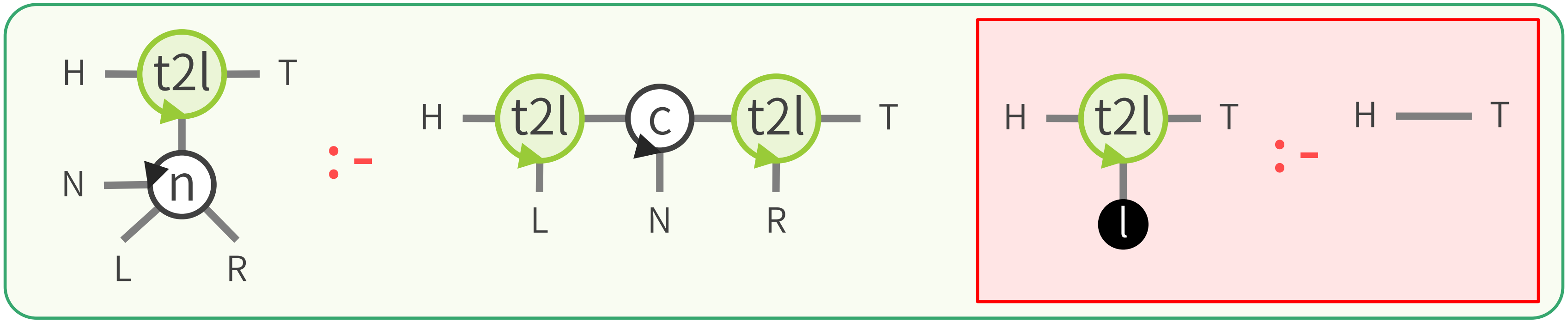
Functional Atom: t2l (tree-to-list)



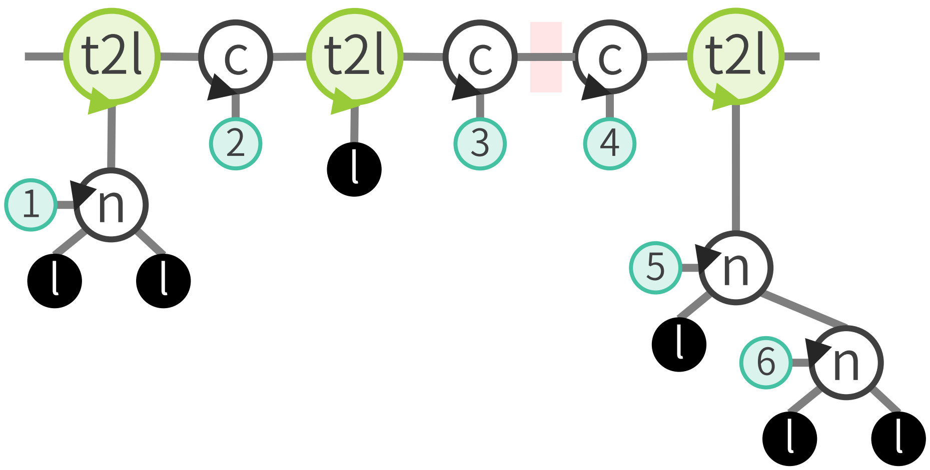


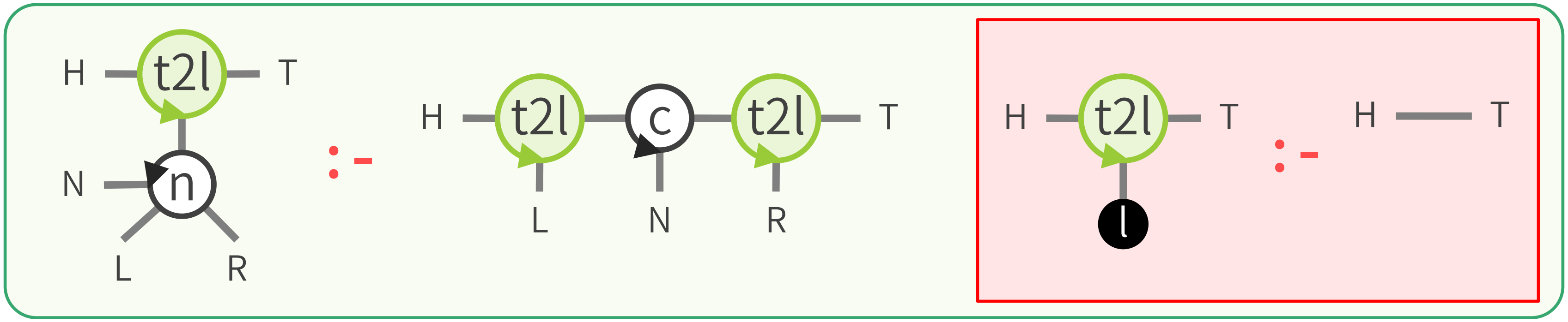
Functional Atom: t2l (tree-to-list)



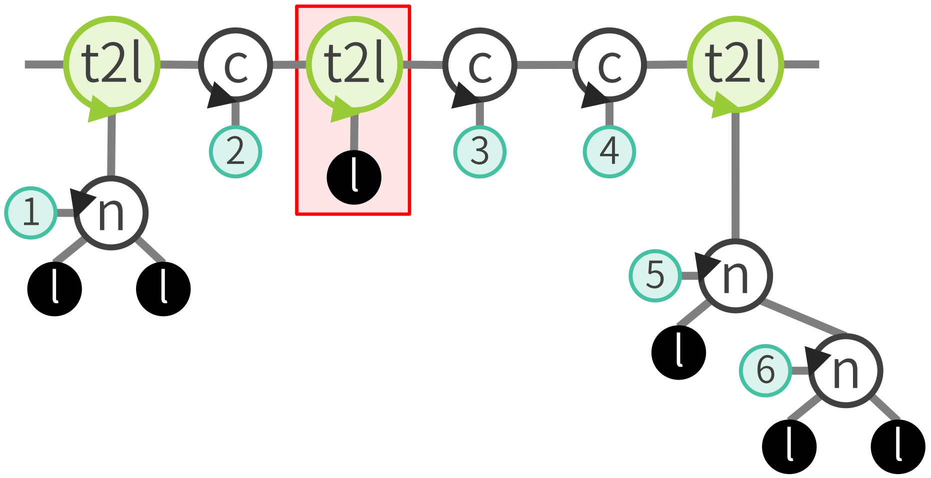


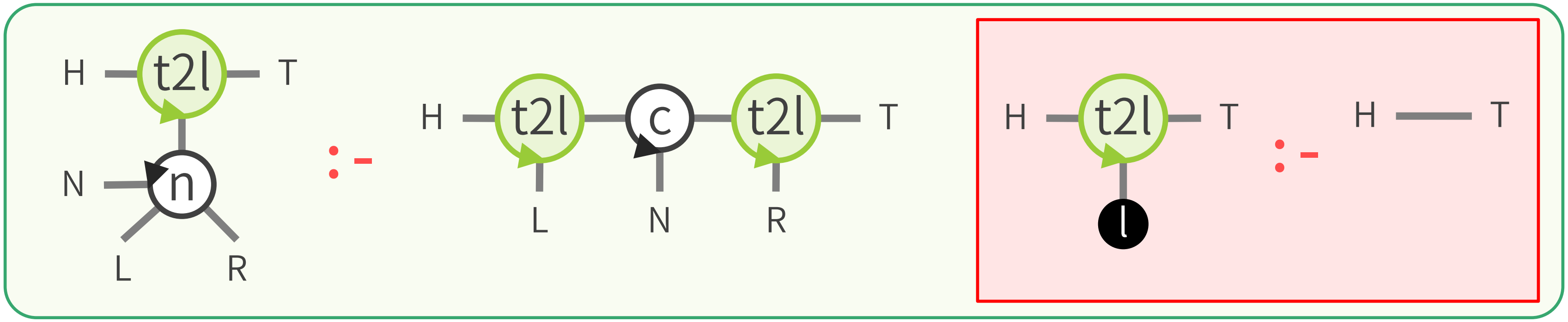
Functional Atom: t2l (tree-to-list)



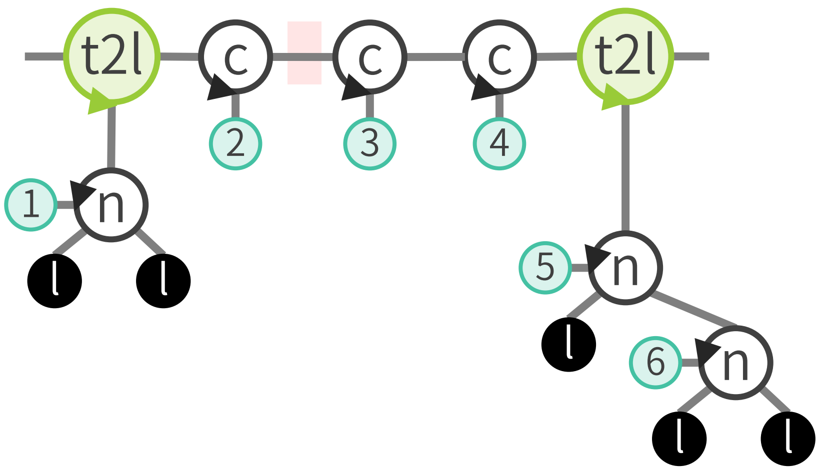


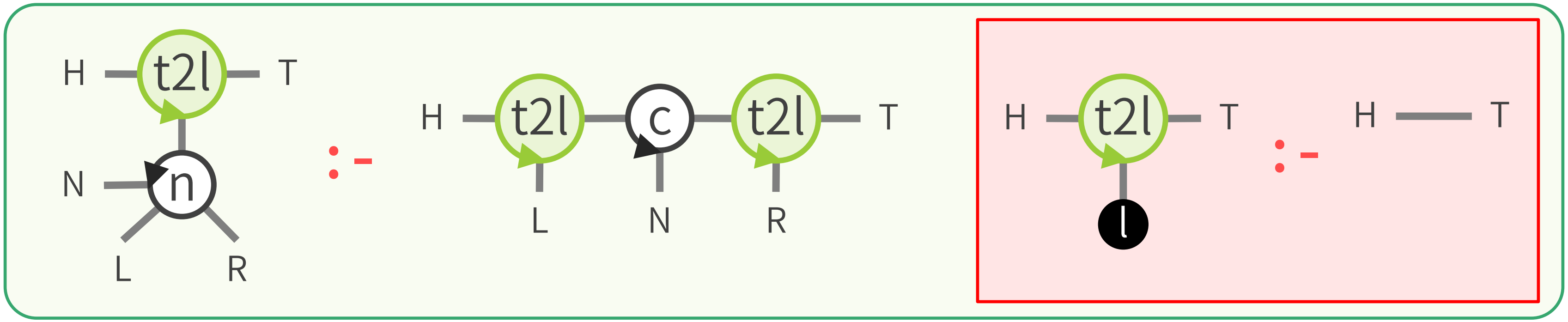
Functional Atom: t2l (tree-to-list)



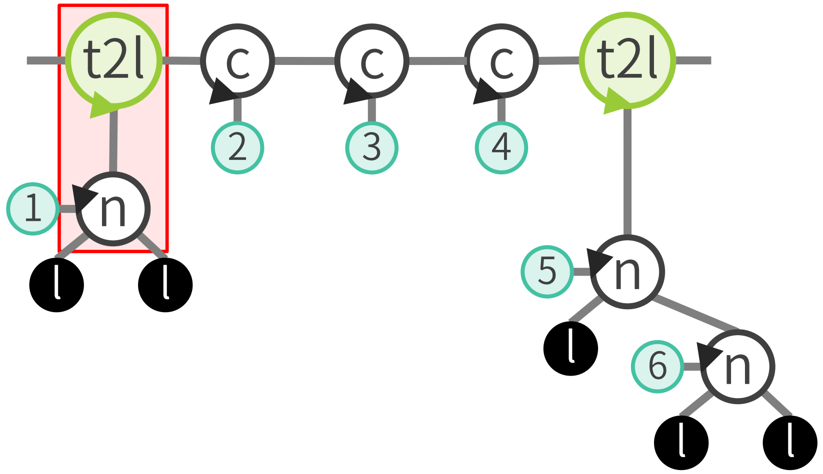


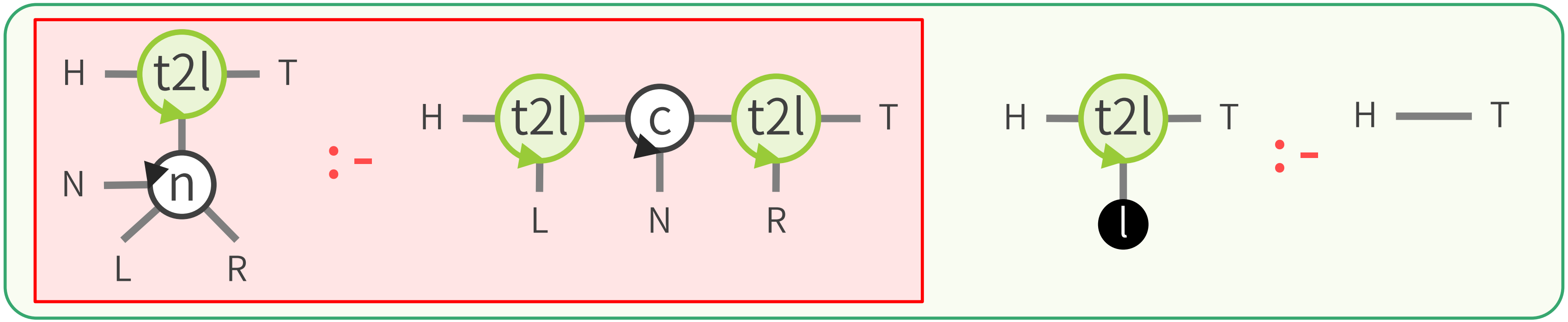
Functional Atom: t2l (tree-to-list)



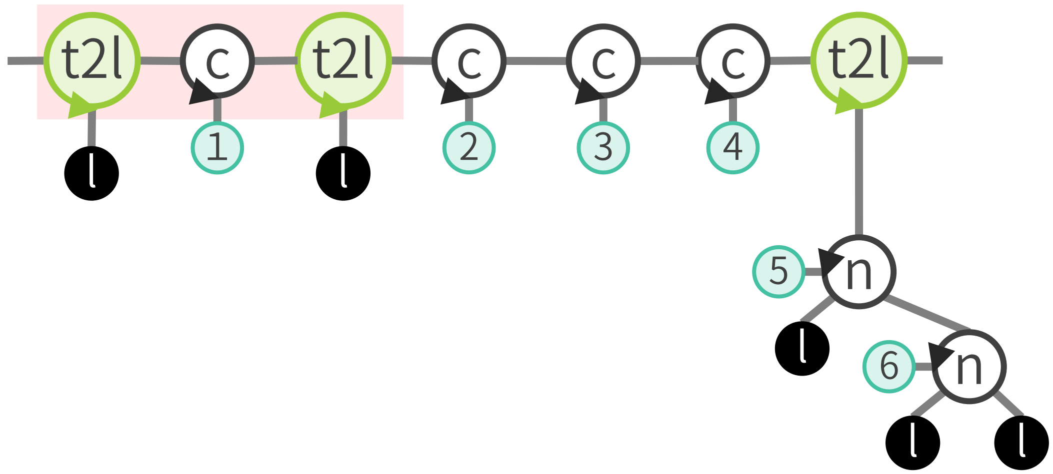


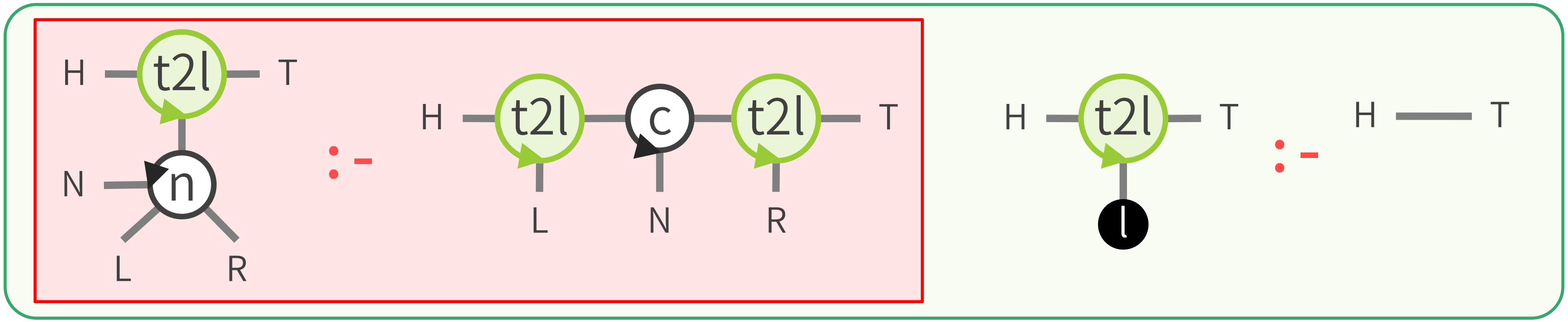
Functional Atom: t2l (tree-to-list)



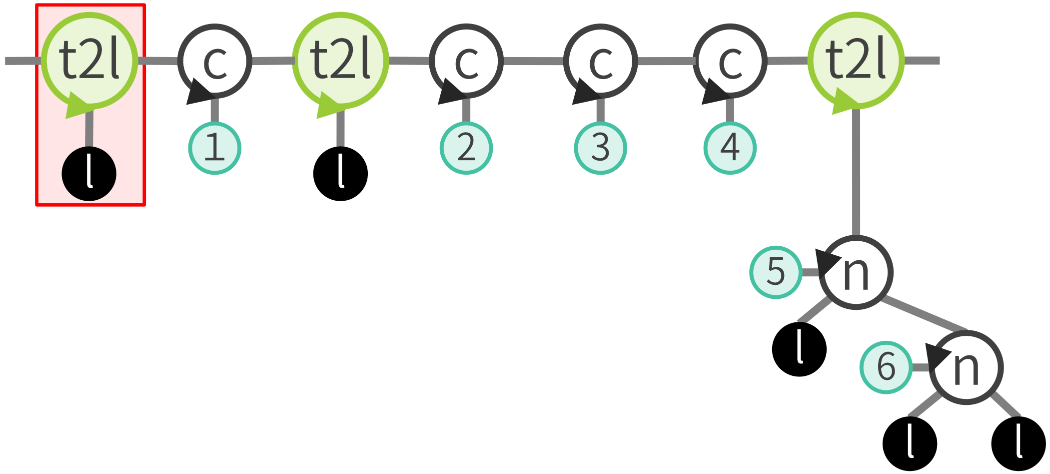


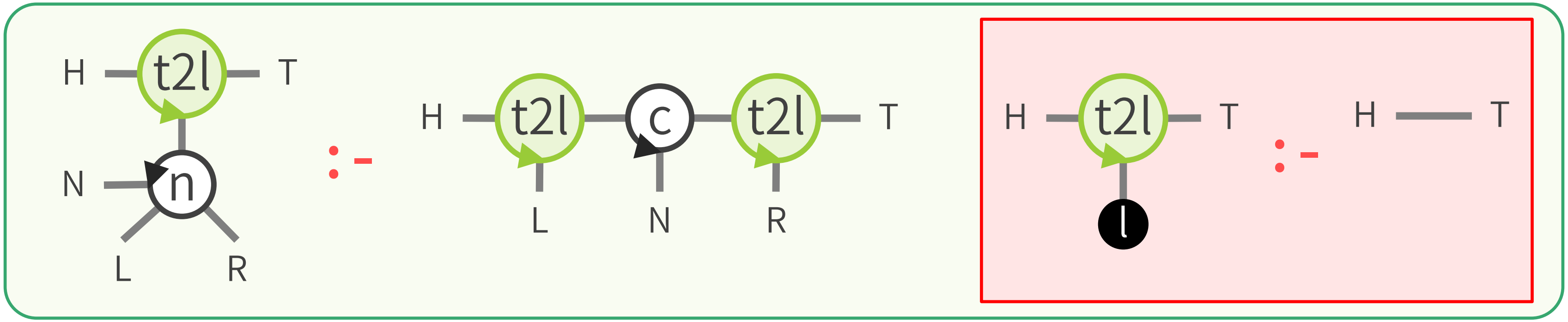
Functional Atom: t2l (tree-to-list)



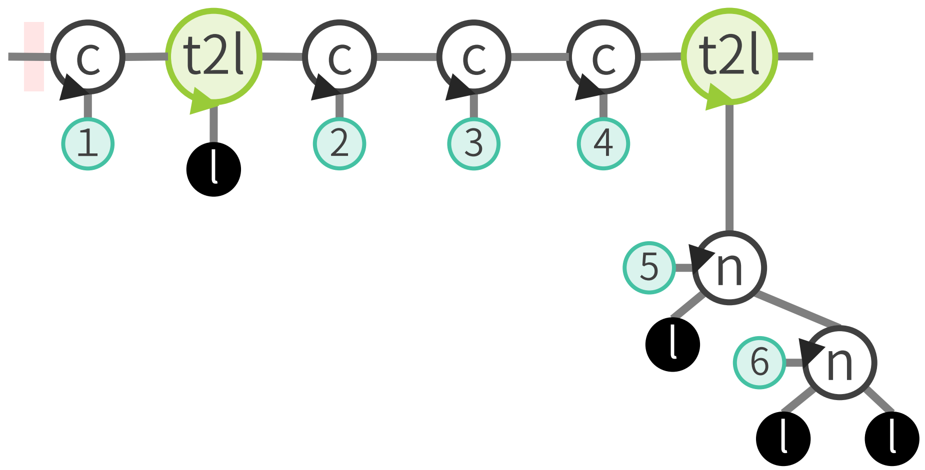


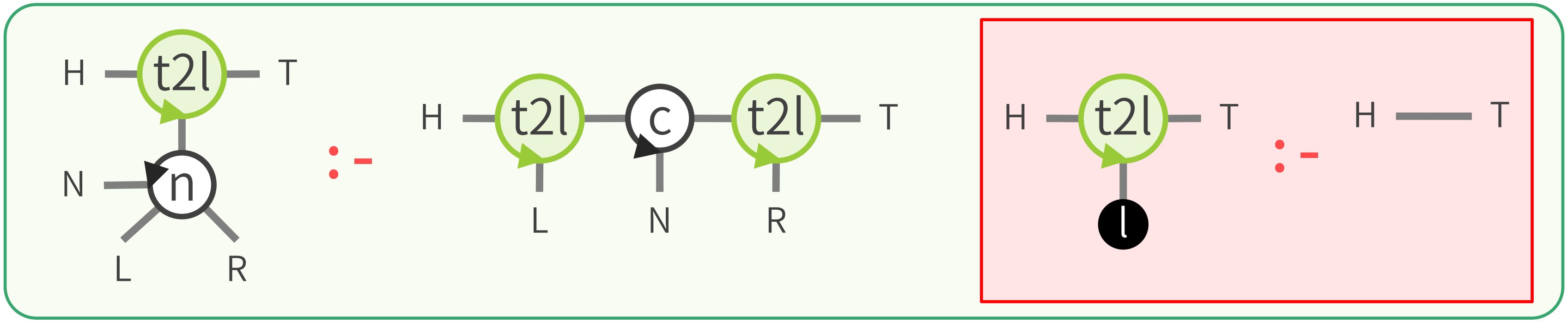
Functional Atom: t2l (tree-to-list)



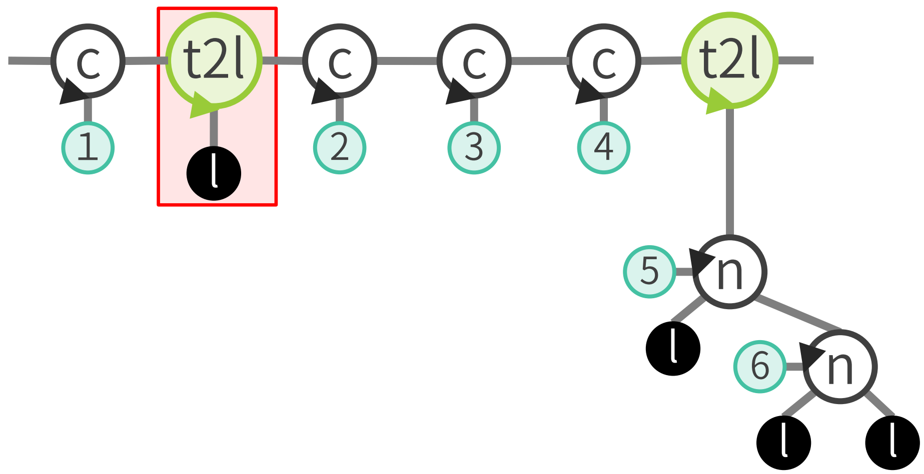


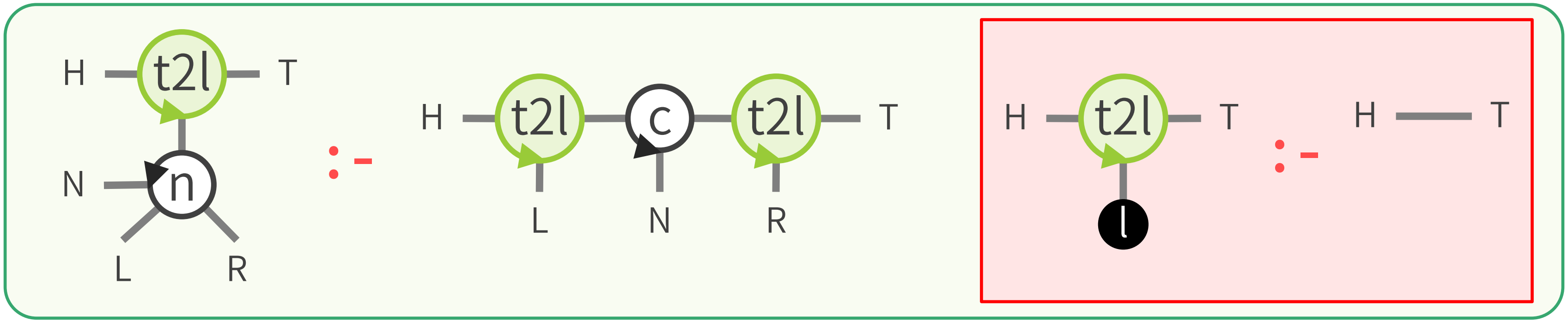
Functional Atom: t2l (tree-to-list)



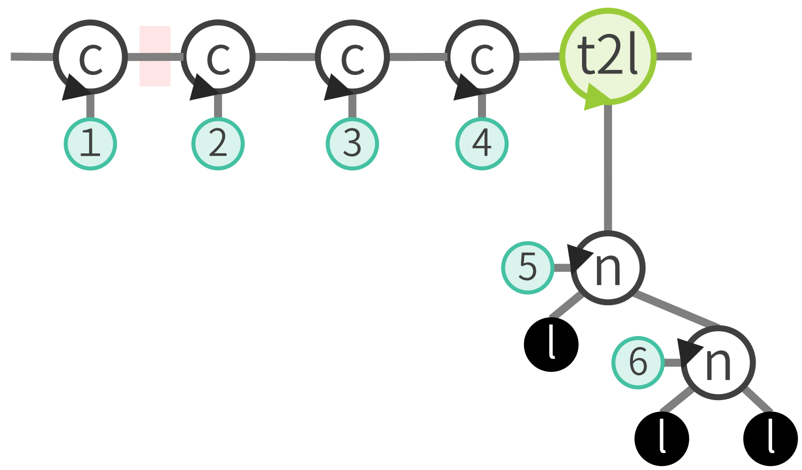


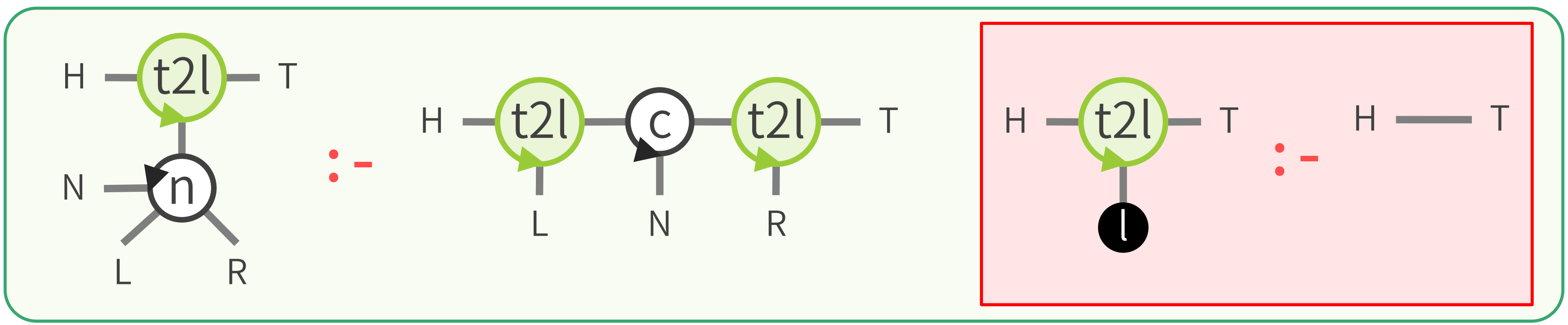
Functional Atom: t2l (tree-to-list)



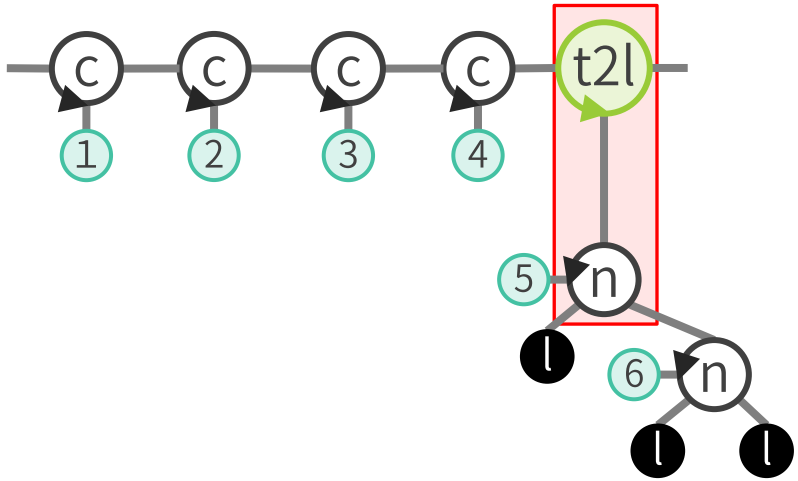


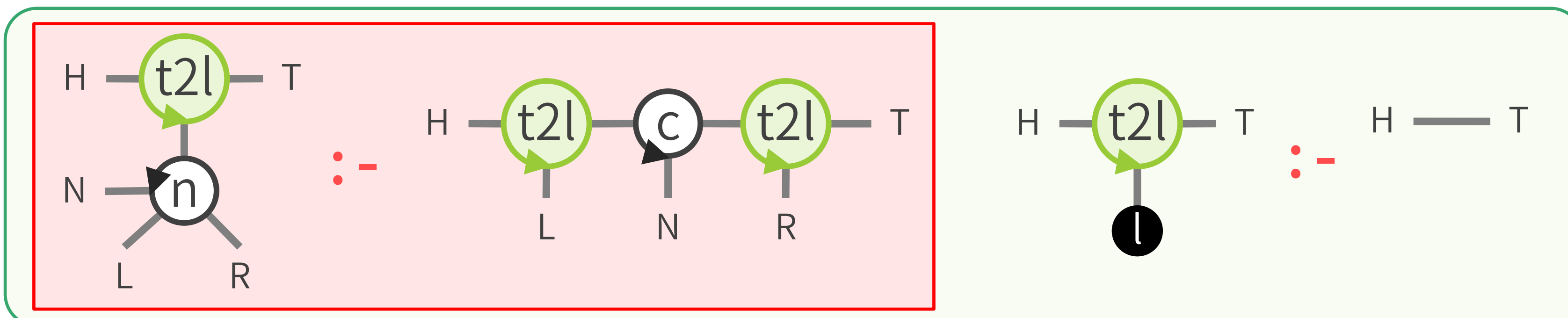
Functional Atom: t2l (tree-to-list)



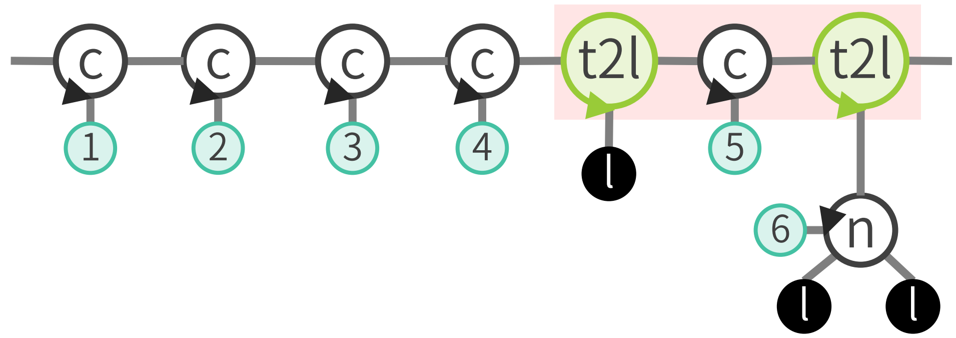


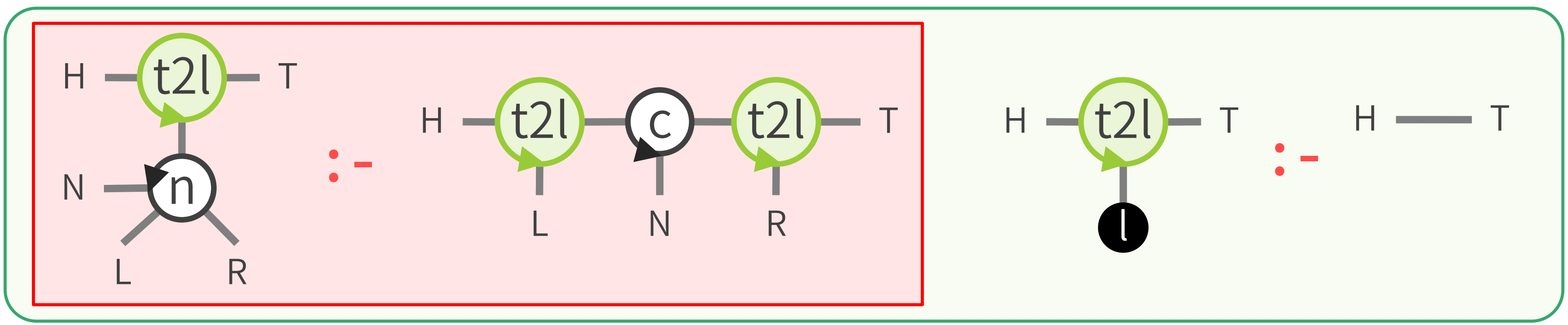
Functional Atom: t2l (tree-to-list)



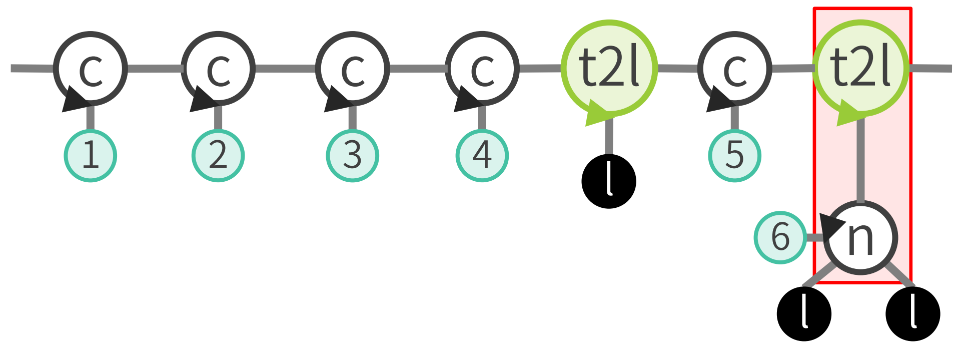


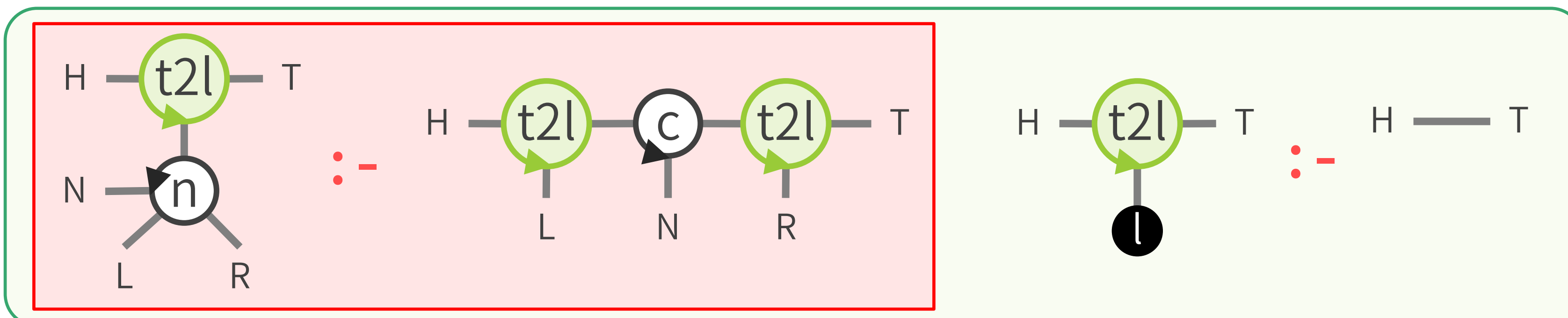
Functional Atom: t2l (tree-to-list)



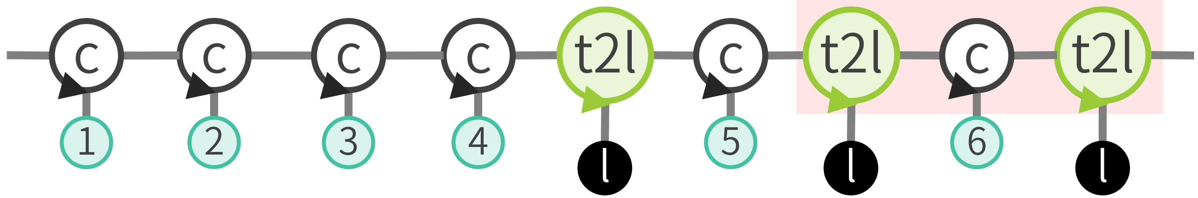


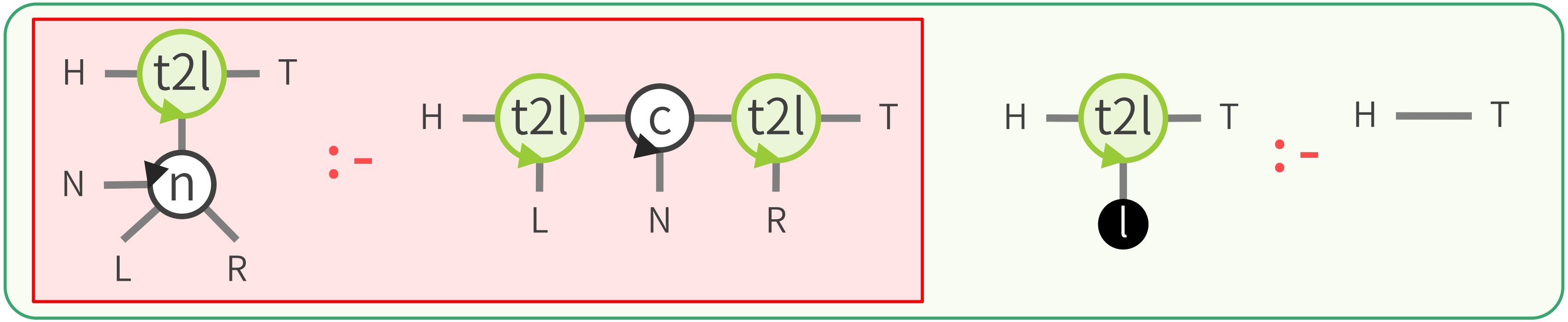
Functional Atom: t2l (tree-to-list)



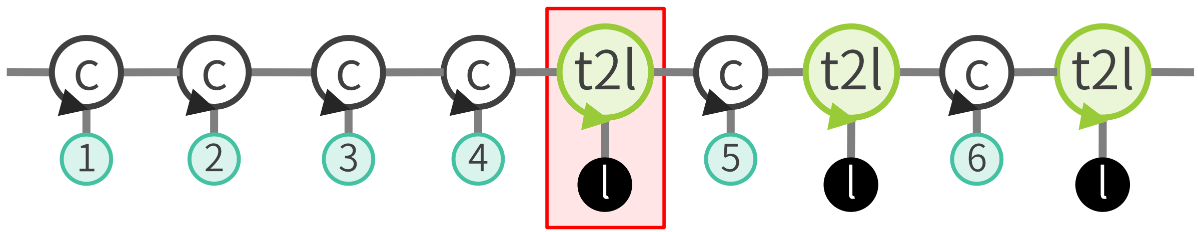


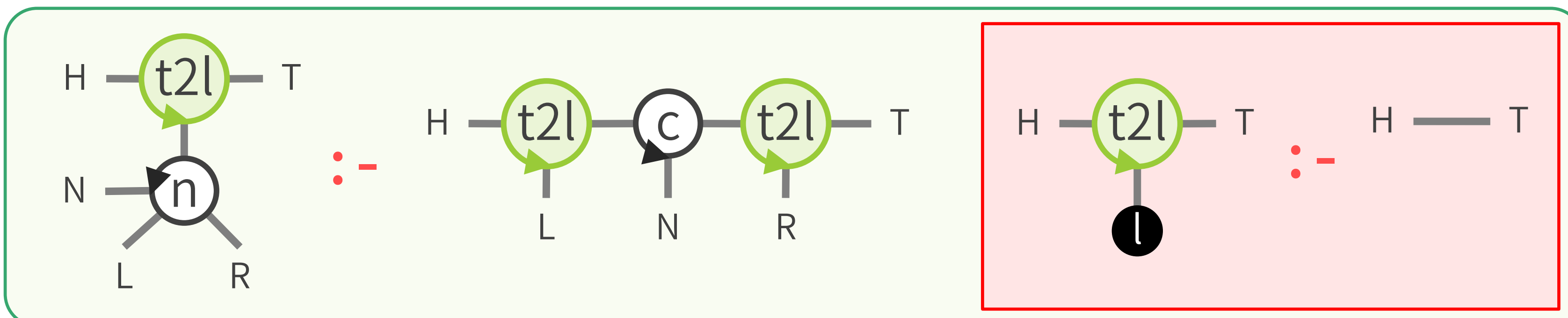
Functional Atom: t2l (tree-to-list)



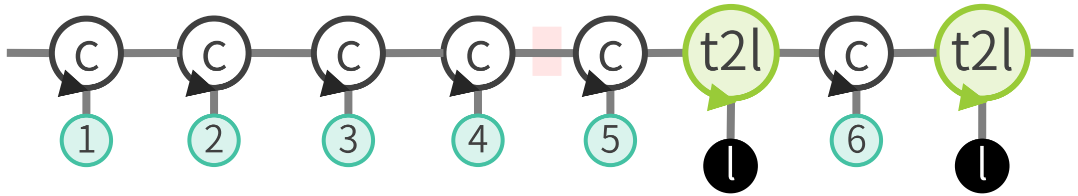


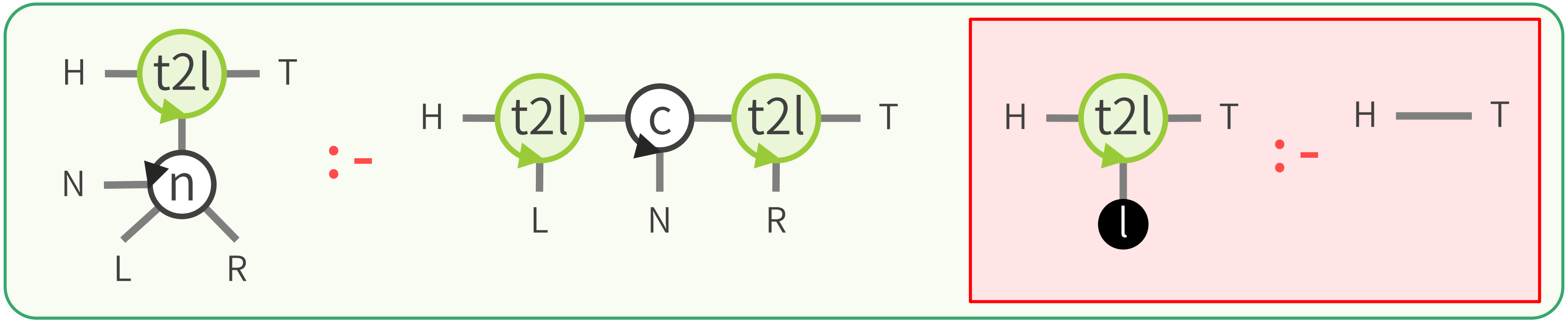
Functional Atom: t2l (tree-to-list)



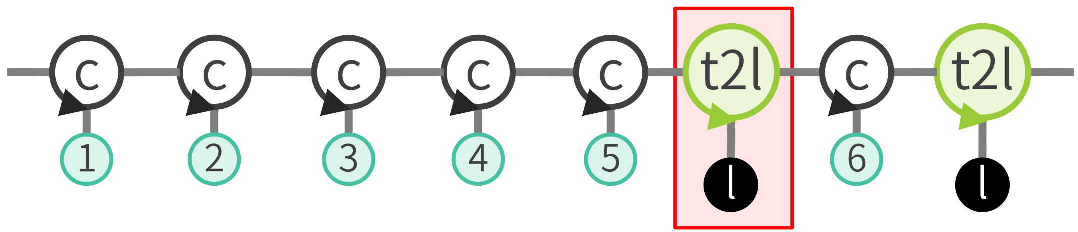


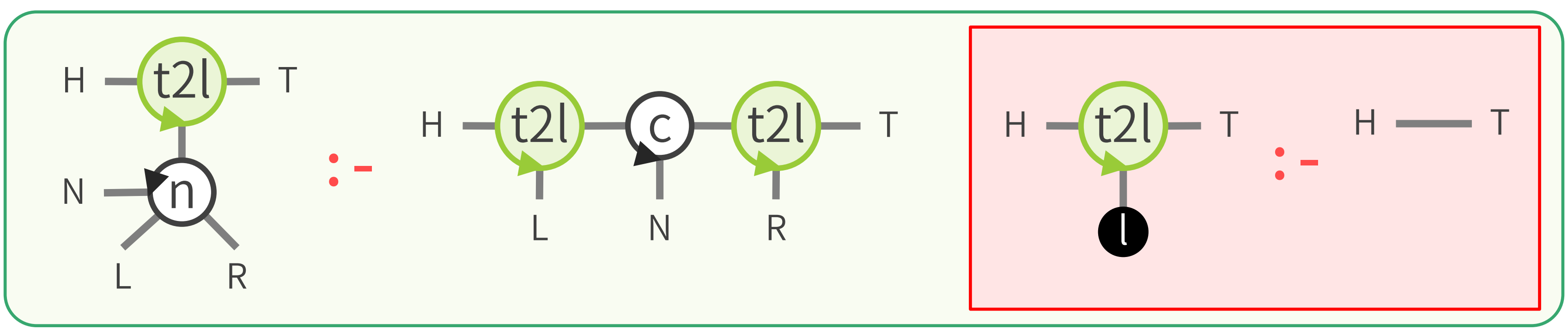
Functional Atom: t2l (tree-to-list)



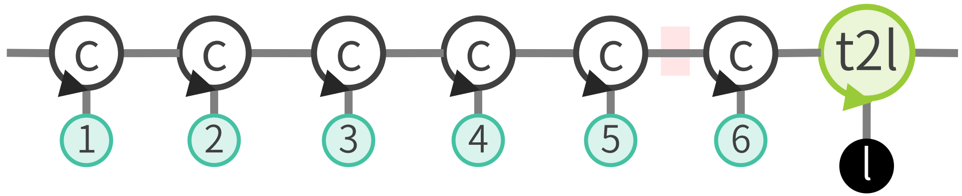


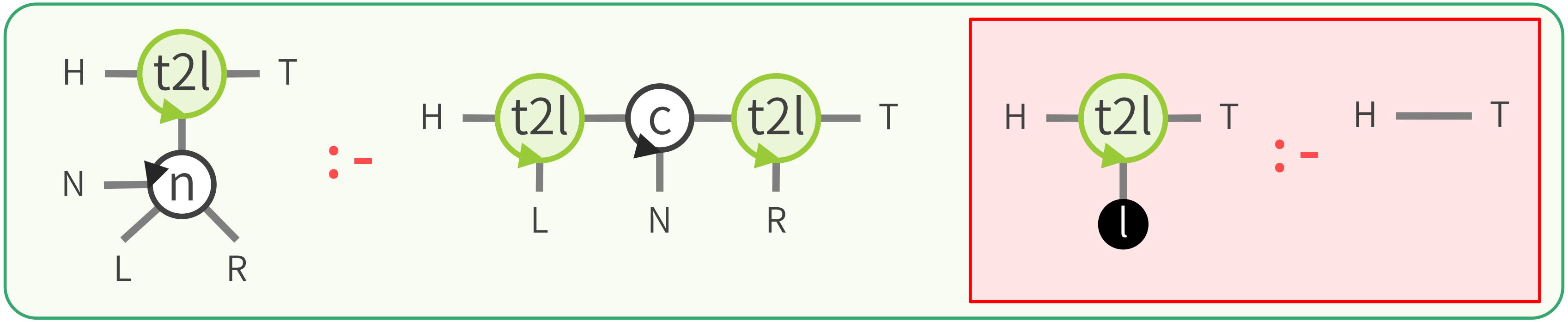
Functional Atom: t2l (tree-to-list)



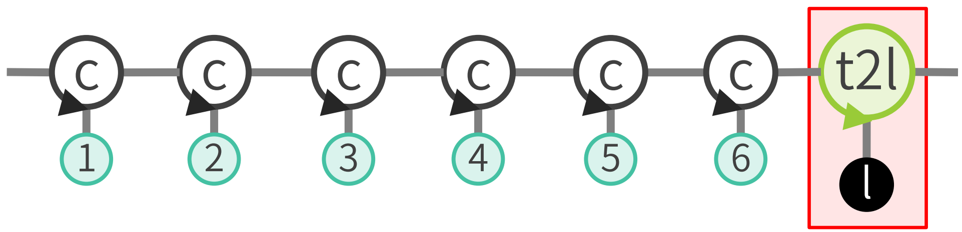


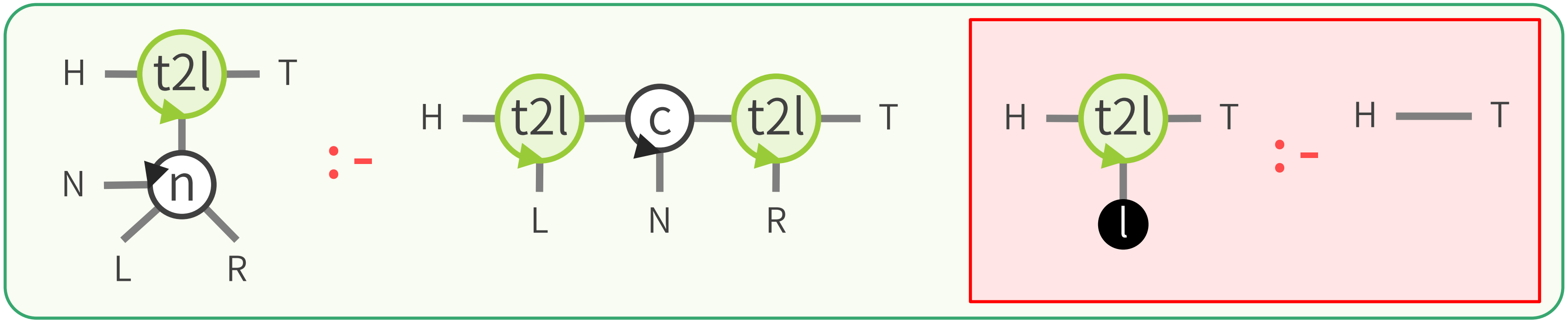
Functional Atom: t2l (tree-to-list)



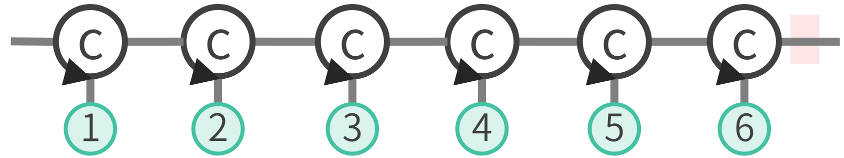


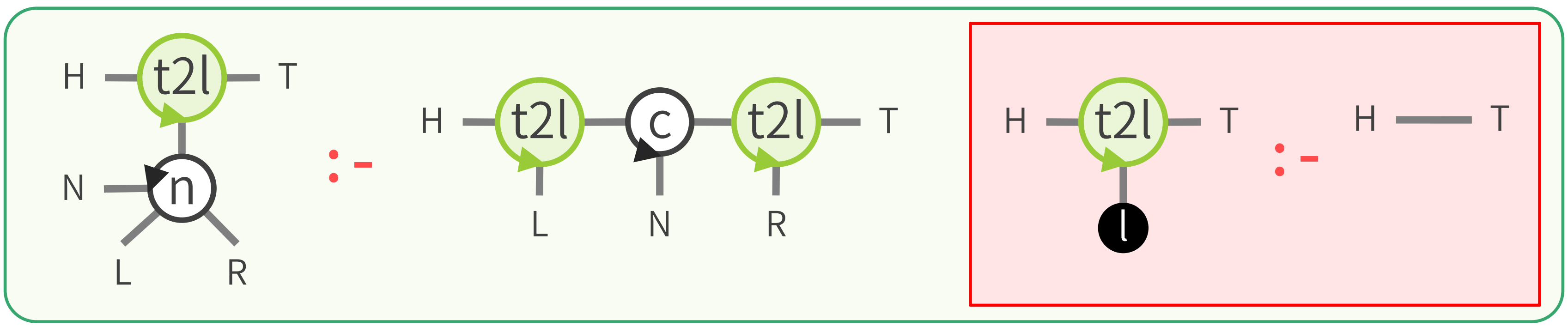
Functional Atom: t2l (tree-to-list)





Functional Atom: t2l (tree-to-list)





Functional Atom: t2l (tree-to-list)