Compact Lists for Tabled Evaluation

João Raimundo and Ricardo Rocha
CRACS & INESC-Porto LA
Faculty of Sciences, University of Porto, Portugal
jraimundo@dcc.fc.up.pt *ricroc@dcc.fc.up.pt*

PADL 2010, Madrid, Spain, January 2010

Tabling in Logic Programming

- **Tabling** is an implementation technique where answers for subcomputations are stored and then reused when a repeated computation appears.
  - Tabled calls are evaluated by storing their answers in an appropriate data space, called the **table space**.
  - Variant tabled calls are resolved by **consuming** the answers already stored in the table space instead of being re-evaluated against the program clauses.
- Tabling has proven to be particularly effective in logic (**Prolog**) programs:
  - **Avoids recomputation**, thus reducing the search space.
  - **Avoids infinite loops**, thus ensuring termination for a wider class of programs.

Motivation

- A critical component in the implementation of an efficient tabling system is the **table space**. Arguably, the most successful data structure for tabling is **tries**.

Motivation

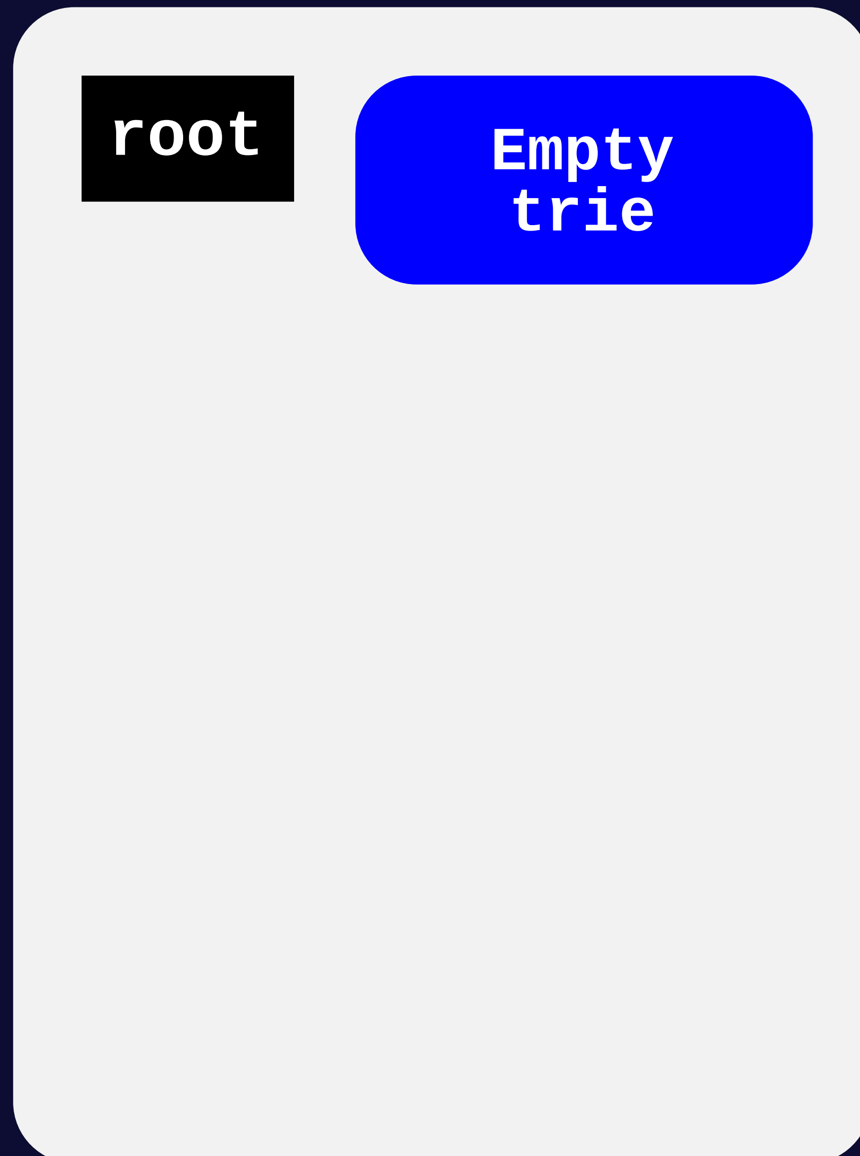
- A critical component in the implementation of an efficient tabling system is the **table space**. Arguably, the most successful data structure for tabling is **tries**.
- When representing terms in tries, most tabling engines, try to mimic the **WAM representation** of these terms in the Prolog stacks in order to avoid unnecessary transformations when storing/loading these terms to/from the tries.
- This idea seems straightforward for almost all type of terms but for **list terms** we found that we can design even more compact and efficient representations by **eliminating the recursive nature** of the WAM representation of list terms.

Motivation

- A critical component in the implementation of an efficient tabling system is the **table space**. Arguably, the most successful data structure for tabling is **tries**.
- When representing terms in tries, most tabling engines, try to mimic the **WAM representation** of these terms in the Prolog stacks in order to avoid unnecessary transformations when storing/loading these terms to/from the tries.
- This idea seems straightforward for almost all type of terms but for **list terms** we found that we can design even more compact and efficient representations by **eliminating the recursive nature** of the WAM representation of list terms.
- We will focus our discussion on a concrete implementation, the **YapTab system**, but our proposals can be easy generalized and applied to other tabling systems.

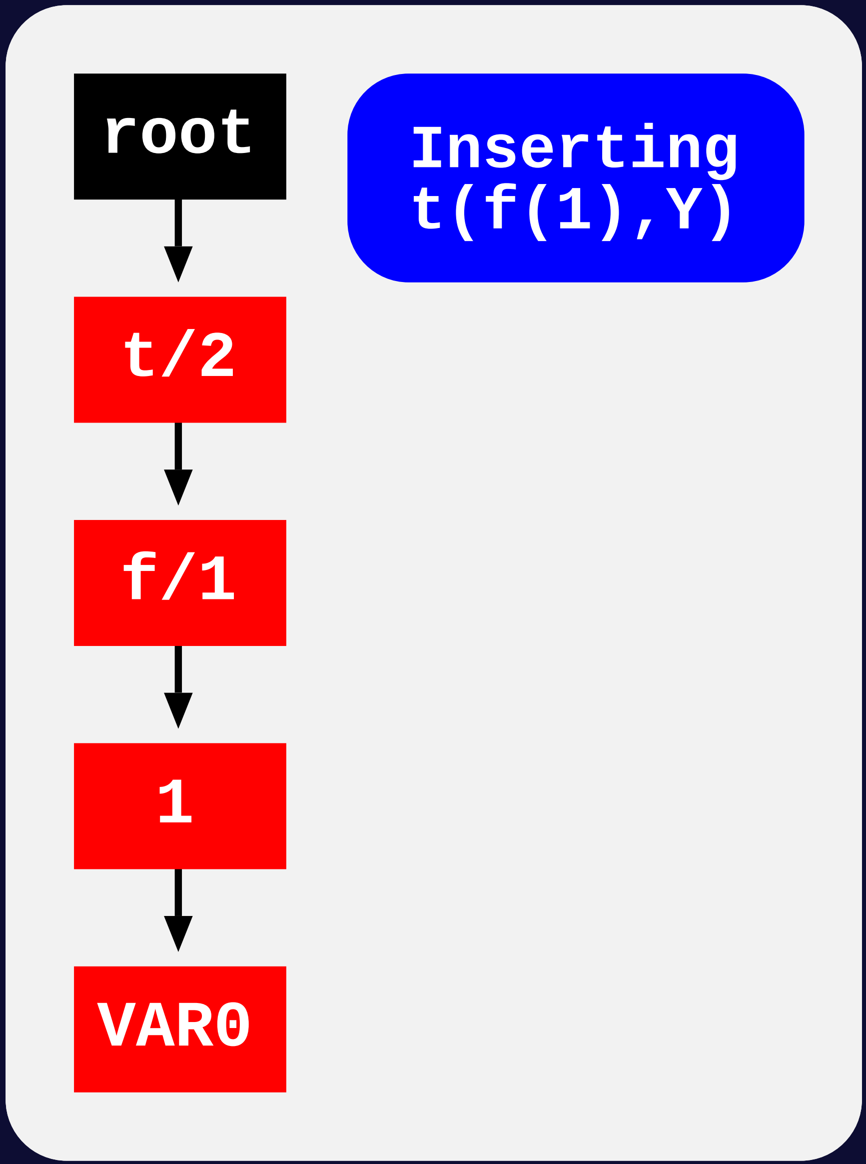
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
- Each different path through the nodes in the trie corresponds to a term.
- Terms with common prefixes branch off from each other at the first distinguishing token.



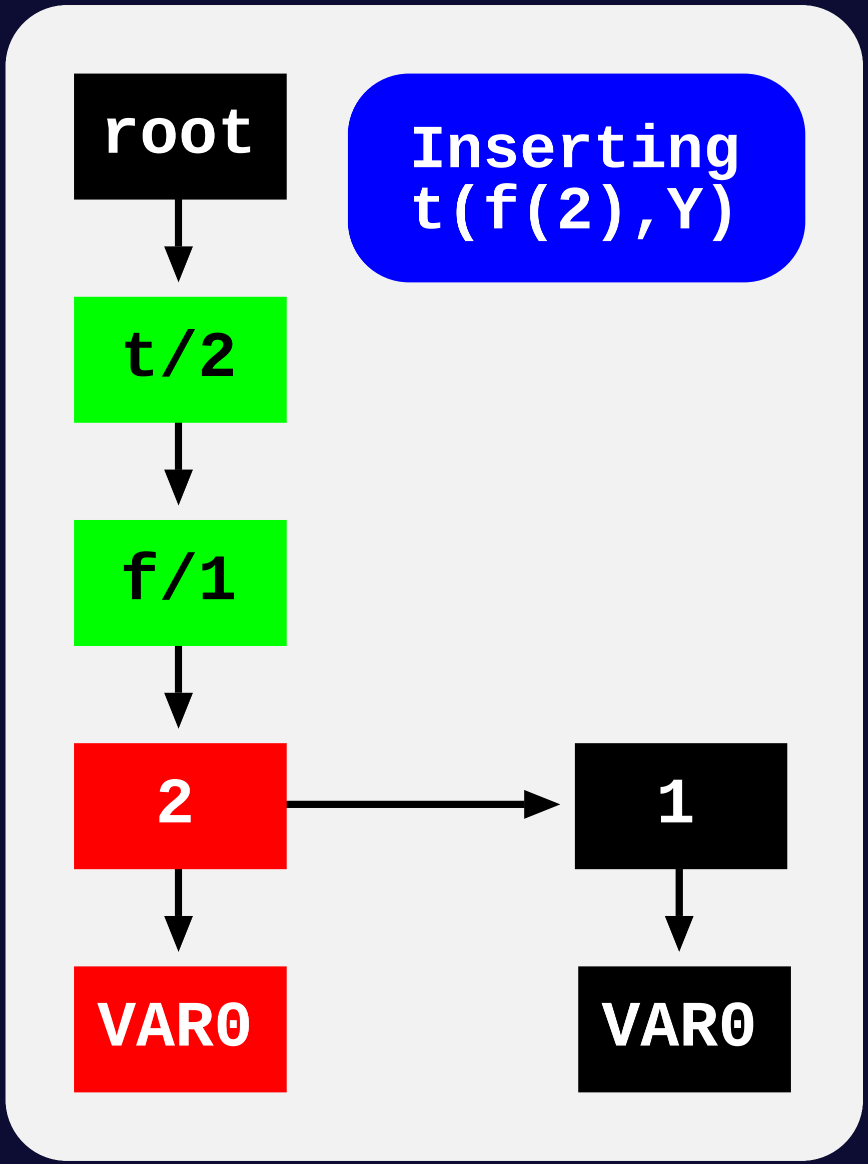
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
- Each different path through the nodes in the trie corresponds to a term.
- Terms with common prefixes branch off from each other at the first distinguishing token.



Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
- Each different path through the nodes in the trie corresponds to a term.
- Terms with common prefixes branch off from each other at the first distinguishing token.



Using Tries to Represent the Table Space

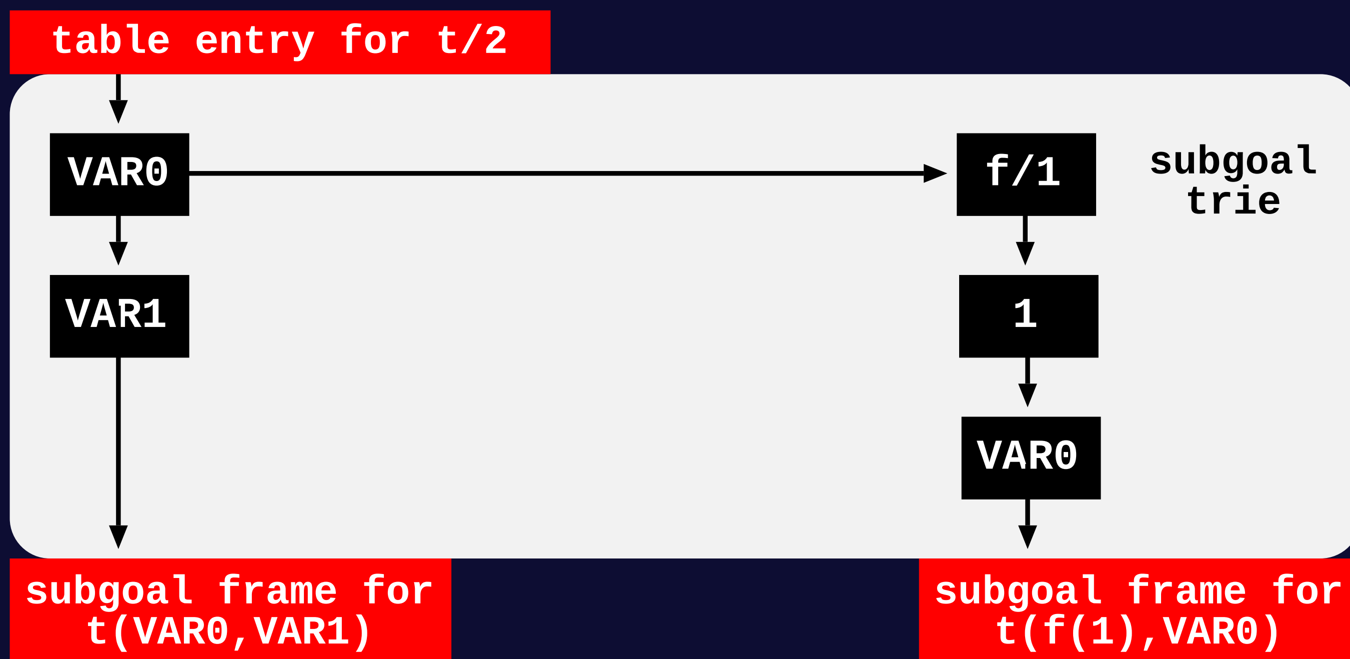
➤ Subgoal Trie

- Stores the tabled subgoal calls.
- Starts at a table entry and ends with subgoal frames.
- A subgoal frame is the entry point for the subgoal answers.

```
:- table t/2.

t(X,Y):- term(X),
         term(Y).

term(f(1)).
term(f(2)).
```



Using Tries to Represent the Table Space

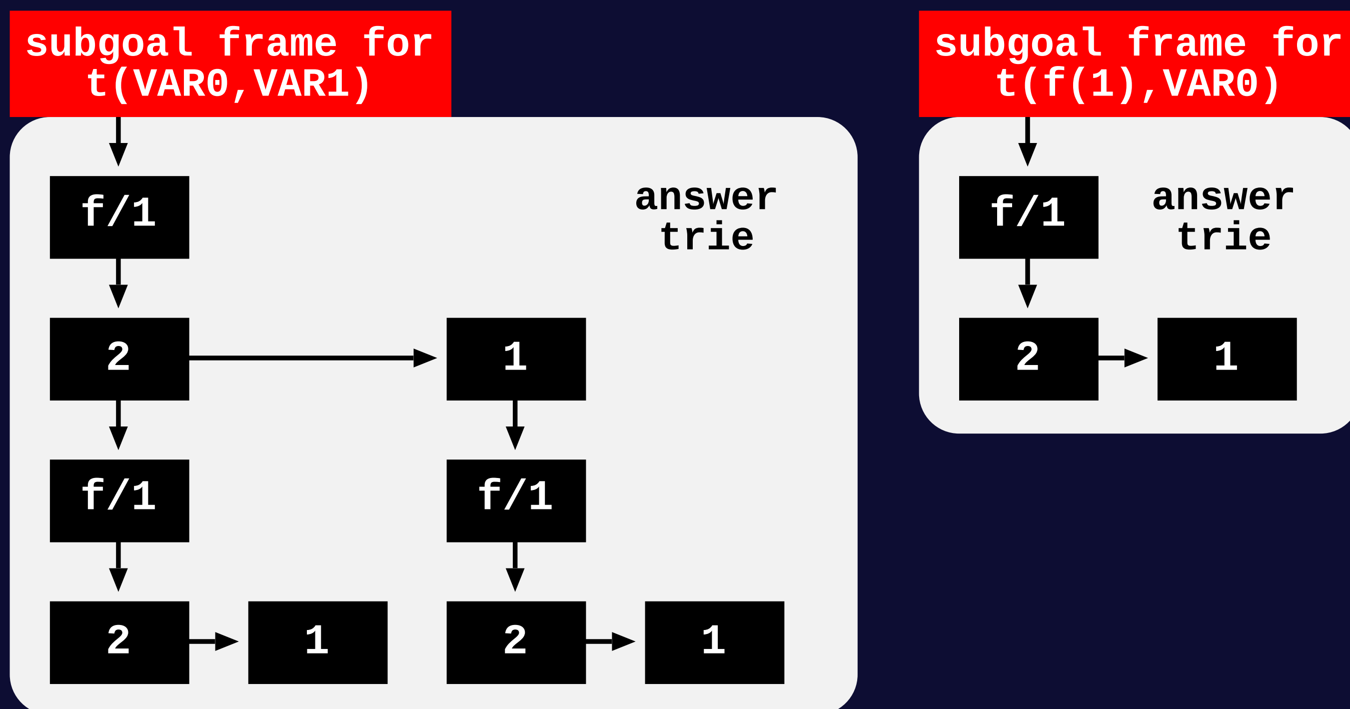
Answer Trie

- Stores the subgoal answers.
- Answer tries hold just the substitution terms for the free variables which exist in the corresponding subgoal call.

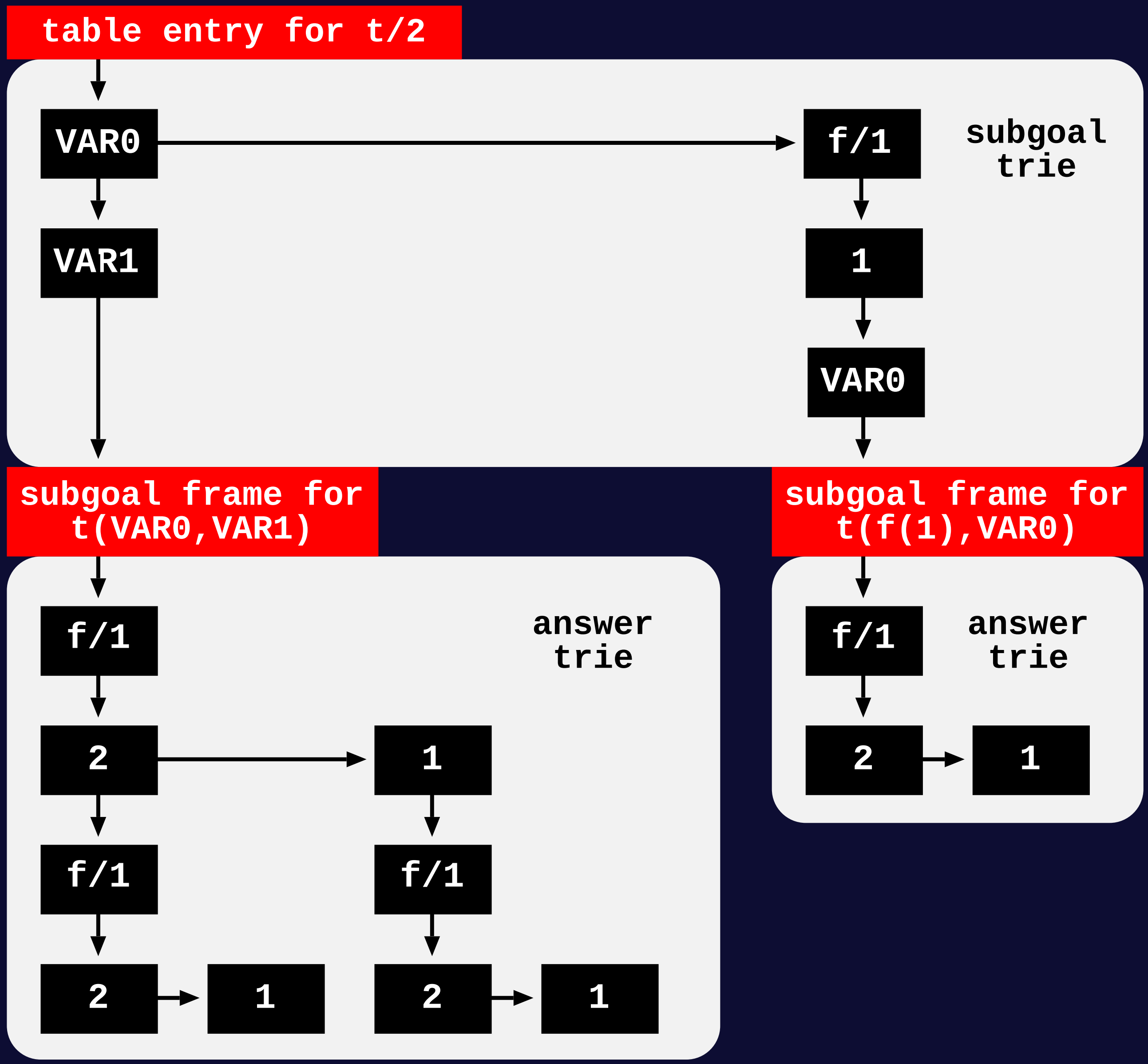
```
:- table t/2.

t(X,Y):- term(X),
         term(Y).

term(f(1)).
term(f(2)).
```

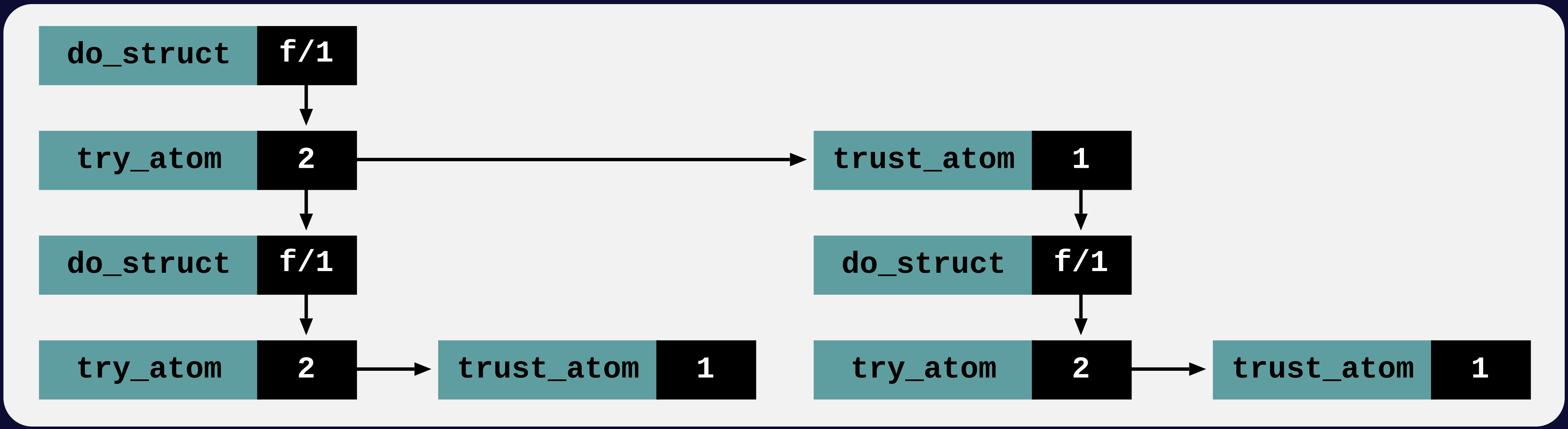


Using Tries to Represent the Table Space

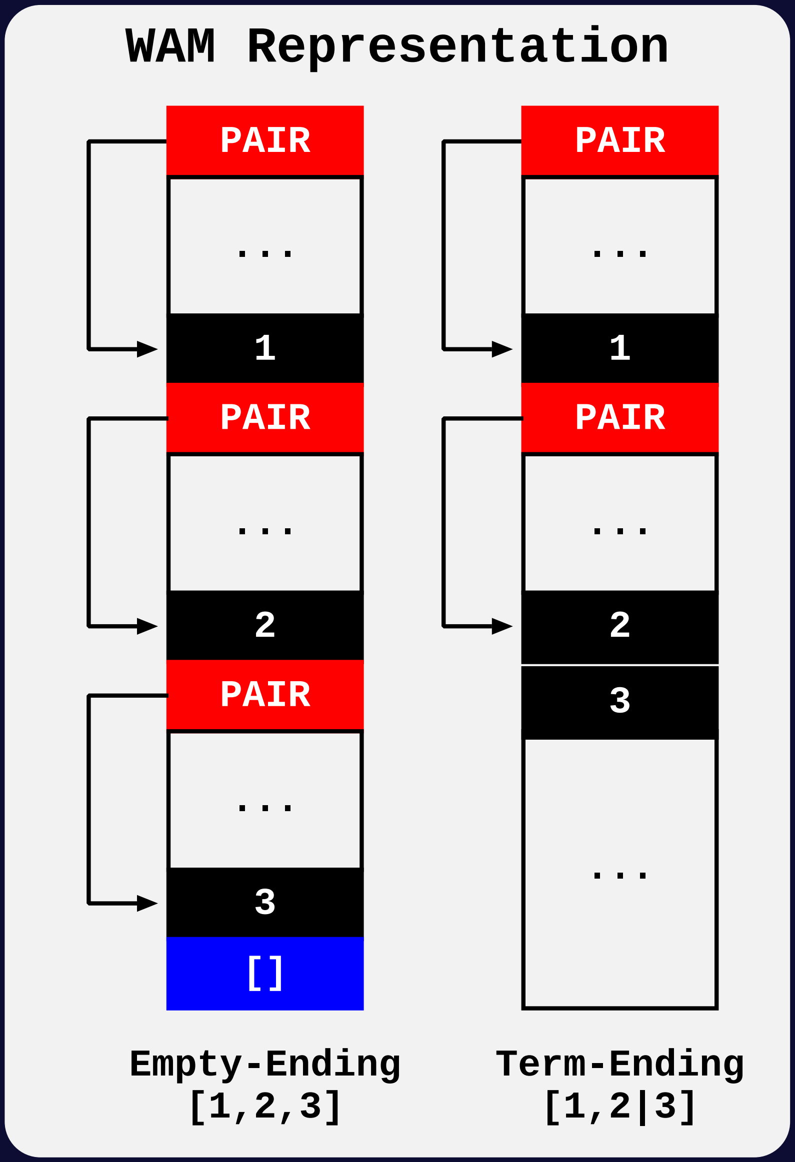


Compiled Tries

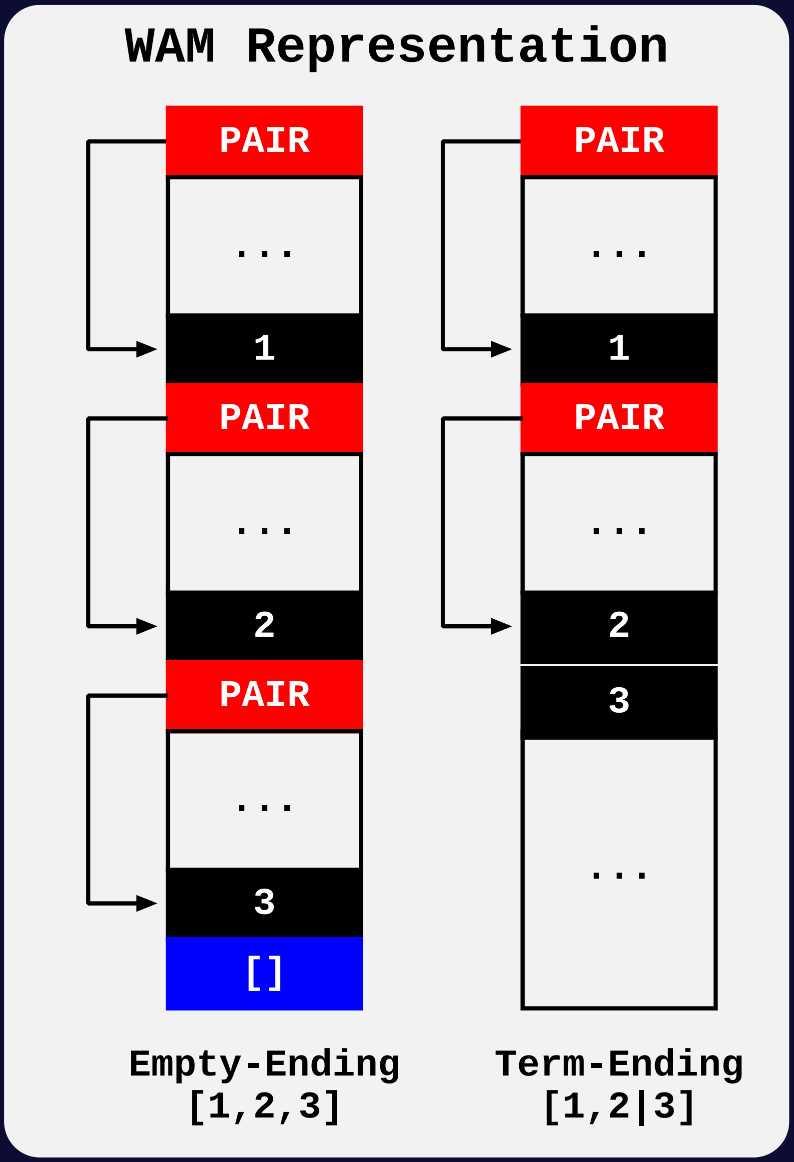
- When a tabled call is **completely evaluated** we can recover answers by traversing **top-down** the completed answer trie and by executing **dynamically compiled WAM-like code** from the answer trie nodes.

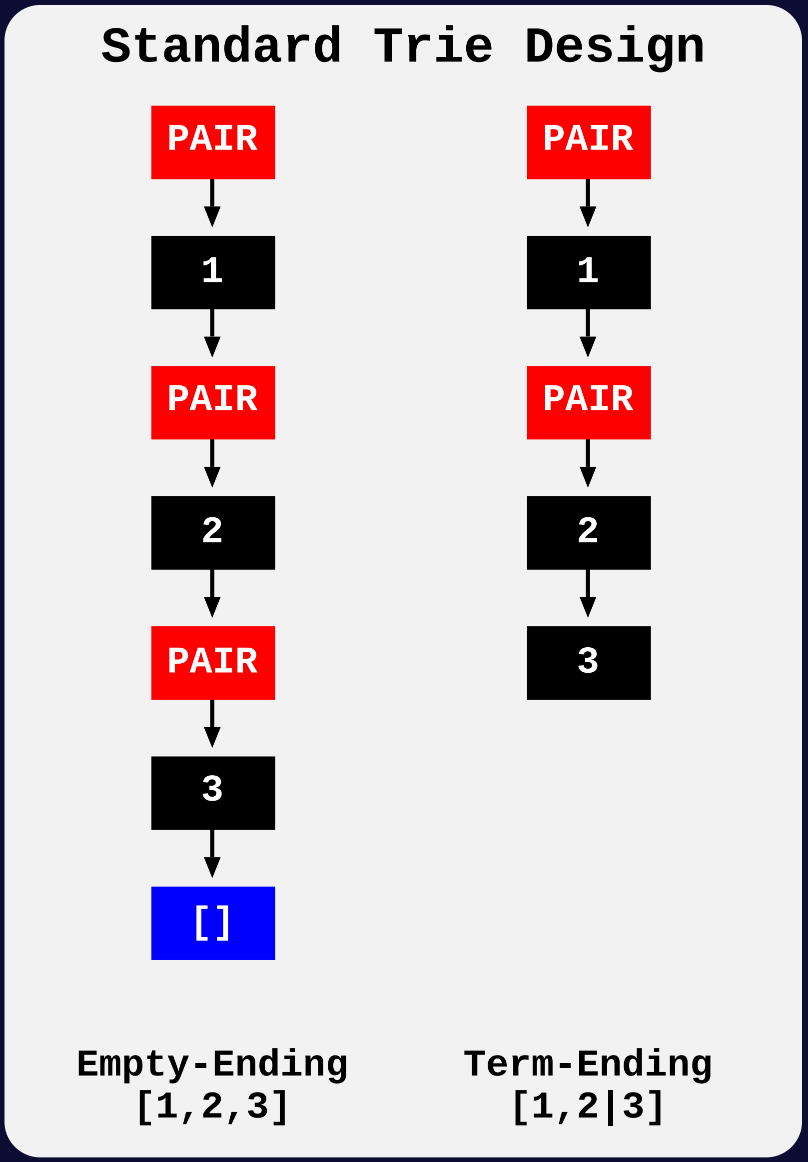


Standard Lists

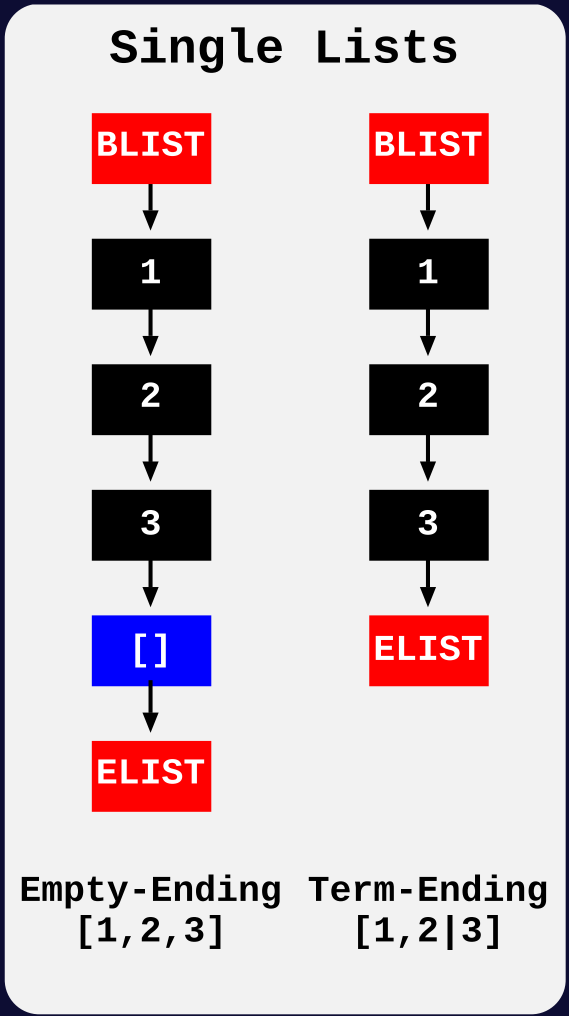


Standard Lists



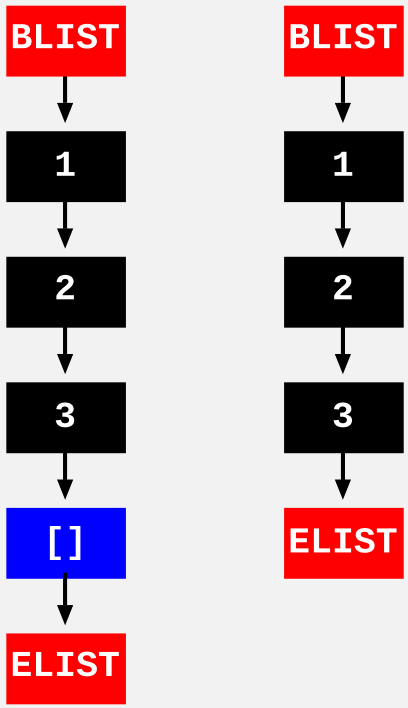


Compact Lists: Initial Approach



Compact Lists: Initial Approach

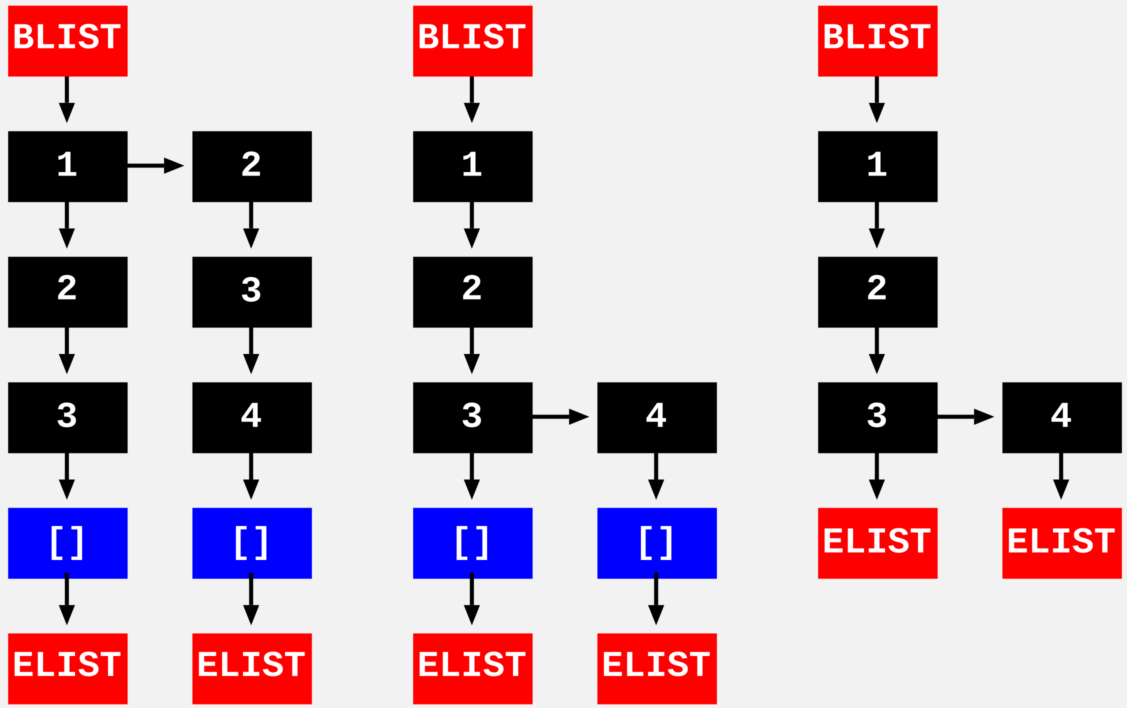
Single Lists



Empty-Ending
[1,2,3]

Term-Ending
[1,2|3]

Multiple Lists

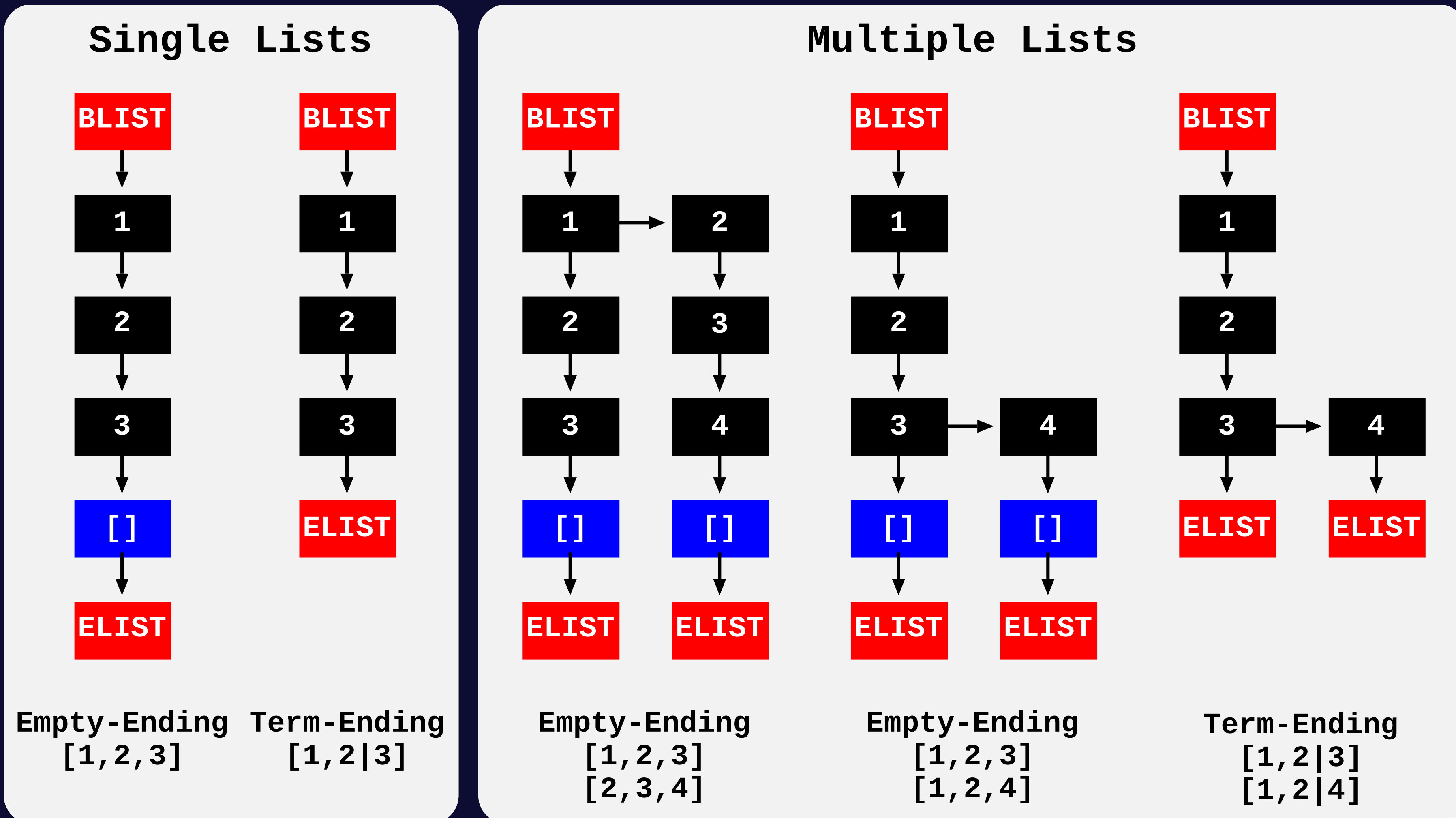


Empty-Ending
[1,2,3]
[2,3,4]

Empty-Ending
[1,2,3]
[1,2,4]

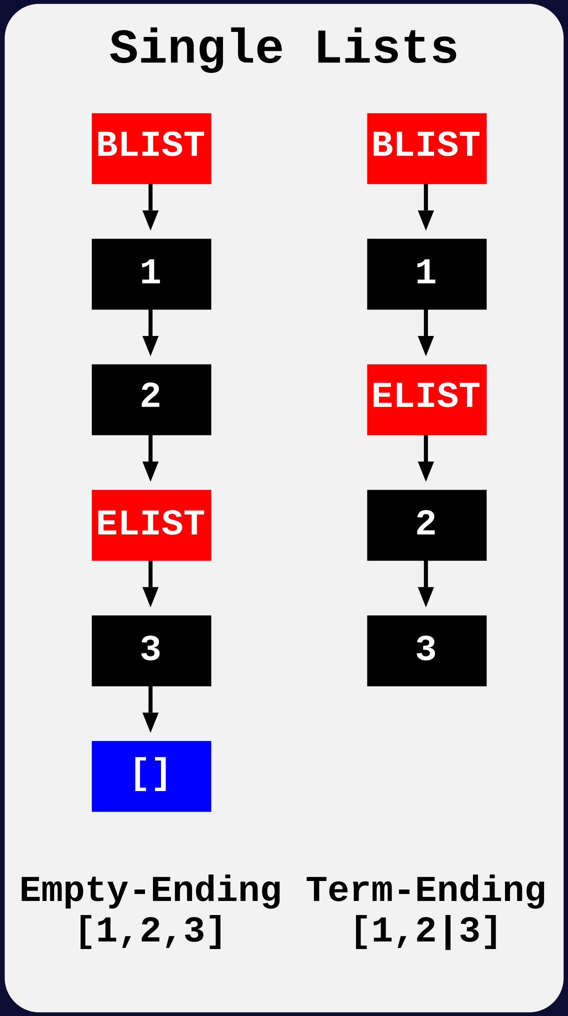
Term-Ending
[1,2|3]
[1,2|4]

Compact Lists: Initial Approach

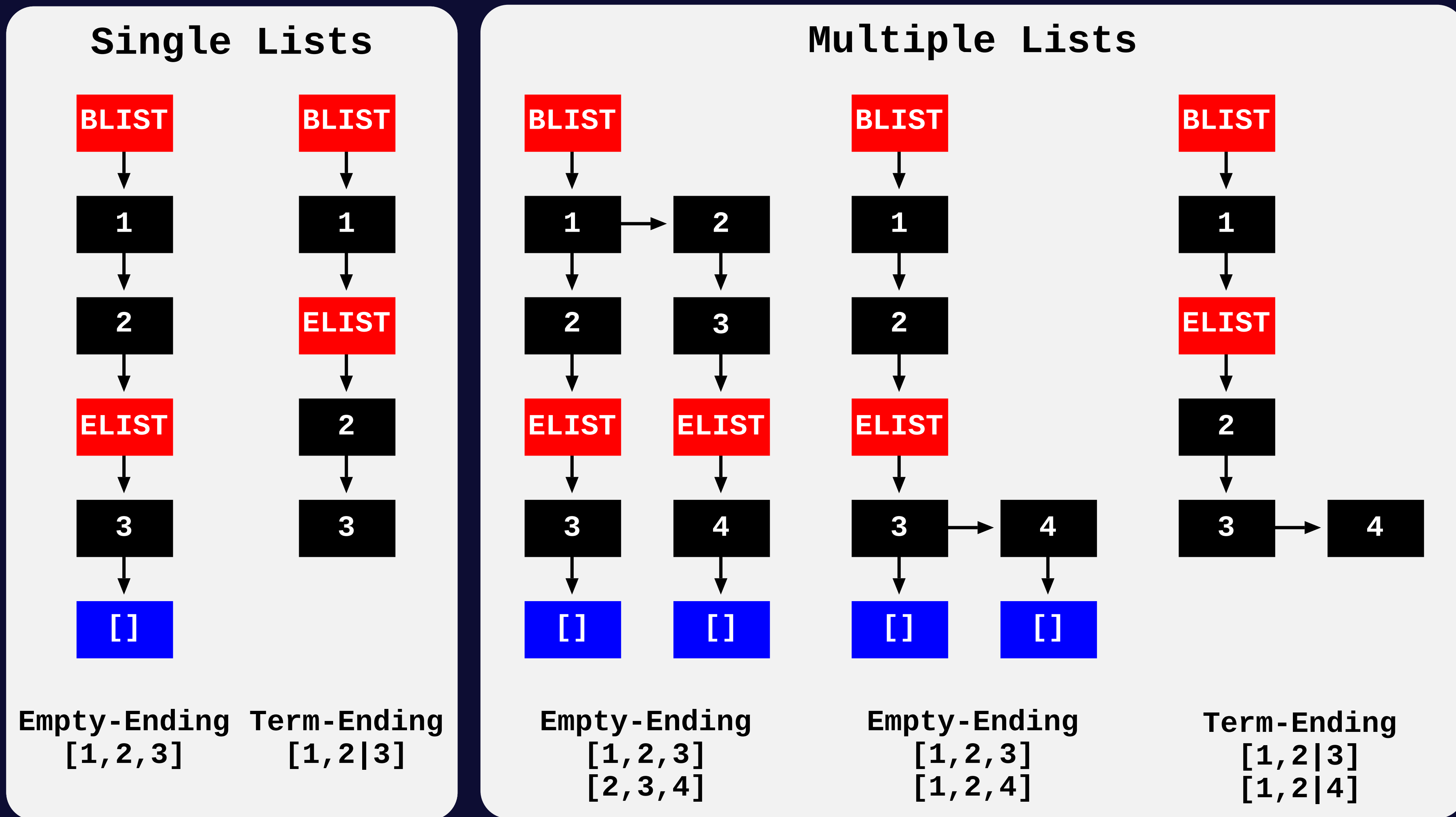


- $N * [E_1, ..., E_S]$: $NS + 2N + 1$ (1st different) $3N + S$ (last different)
- $N * [E_1, ...|E_S]$: $NS + N + 1$ (1st different) $2N + S$ (last different)

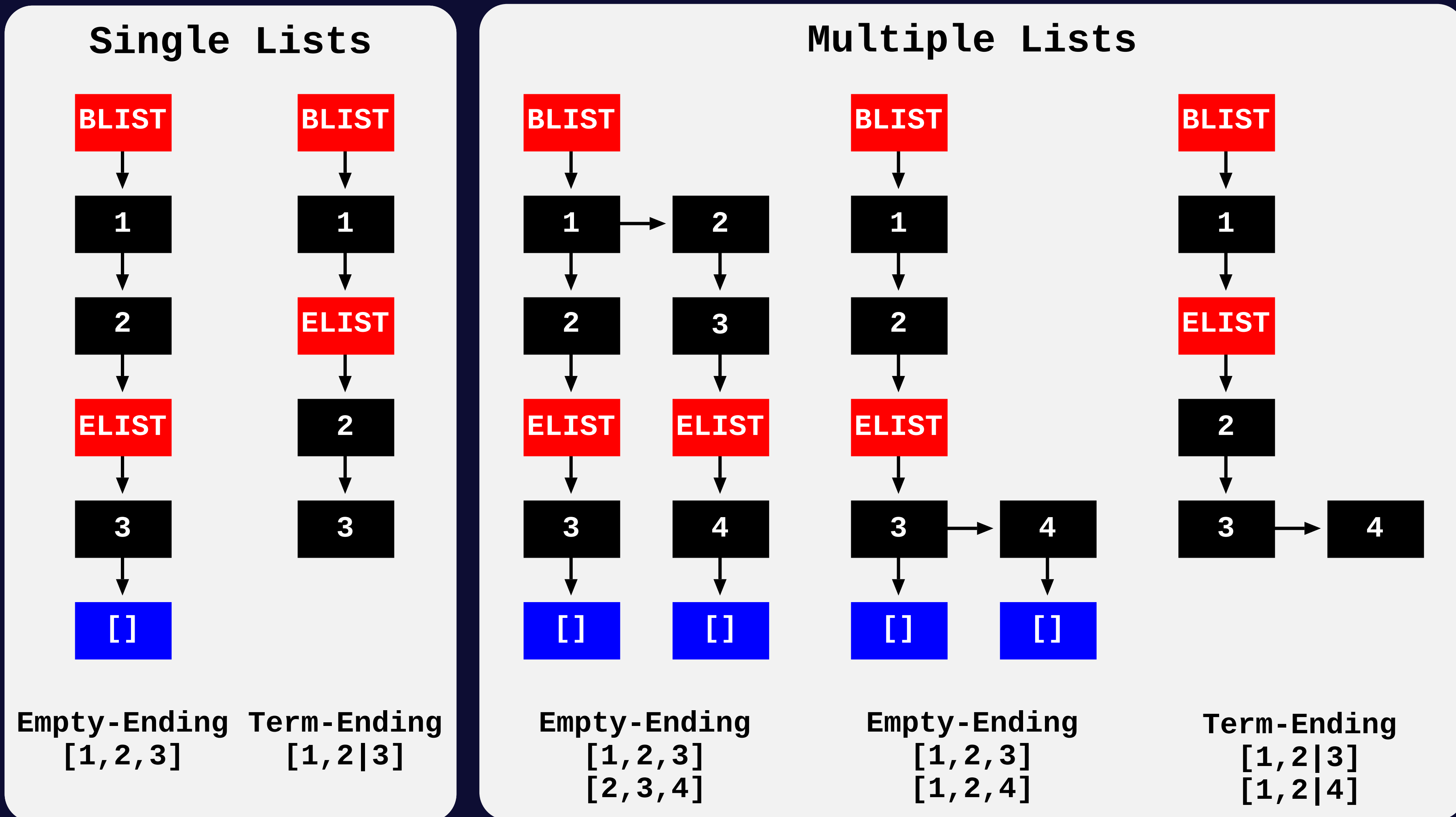
Compact Lists: Second Approach



Compact Lists: Second Approach

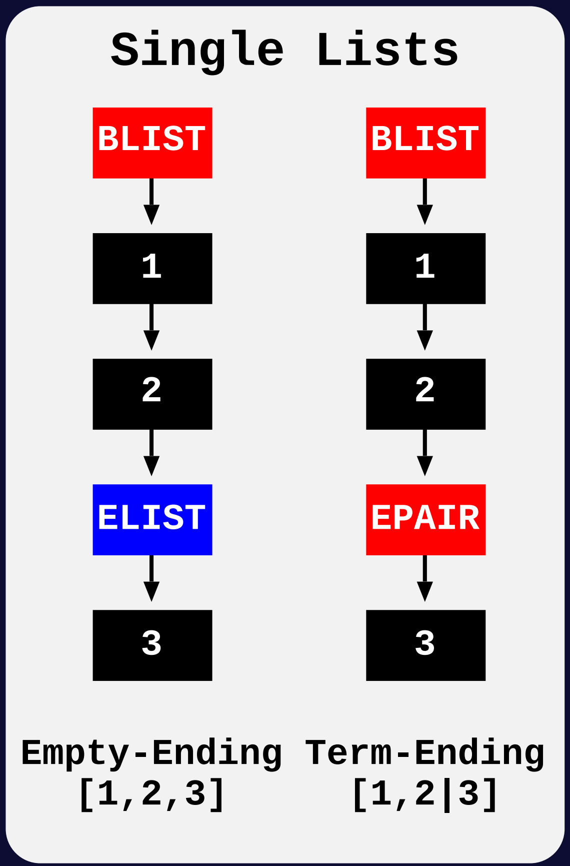


Compact Lists: Second Approach

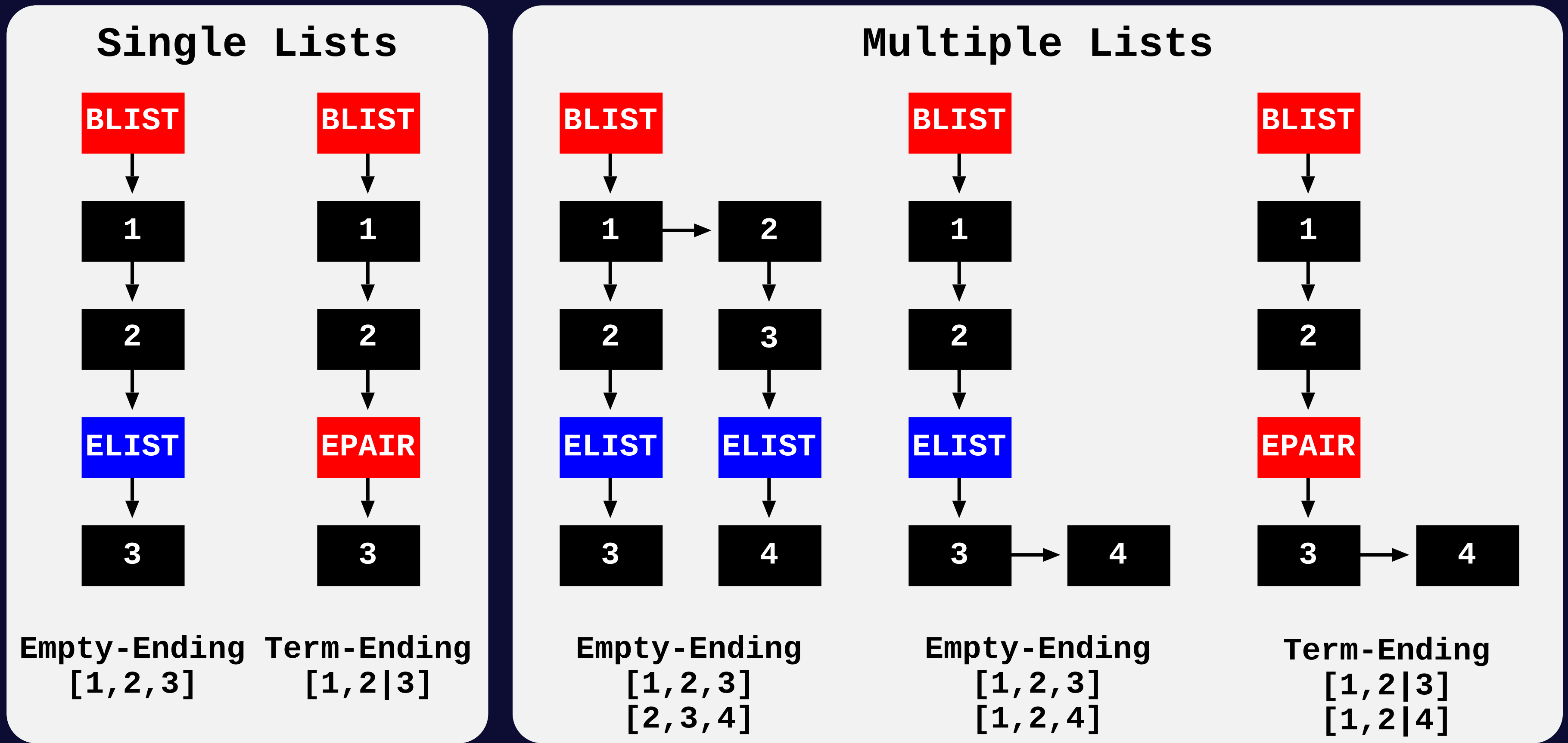


- $N * [E_1, ..., E_S]$: $NS + 2N + 1$ (1st different) $2N + S + 1$ (last different)
- $N * [E_1, ...|E_S]$: $NS + N + 1$ (1st different) $N + S + 1$ (last different)

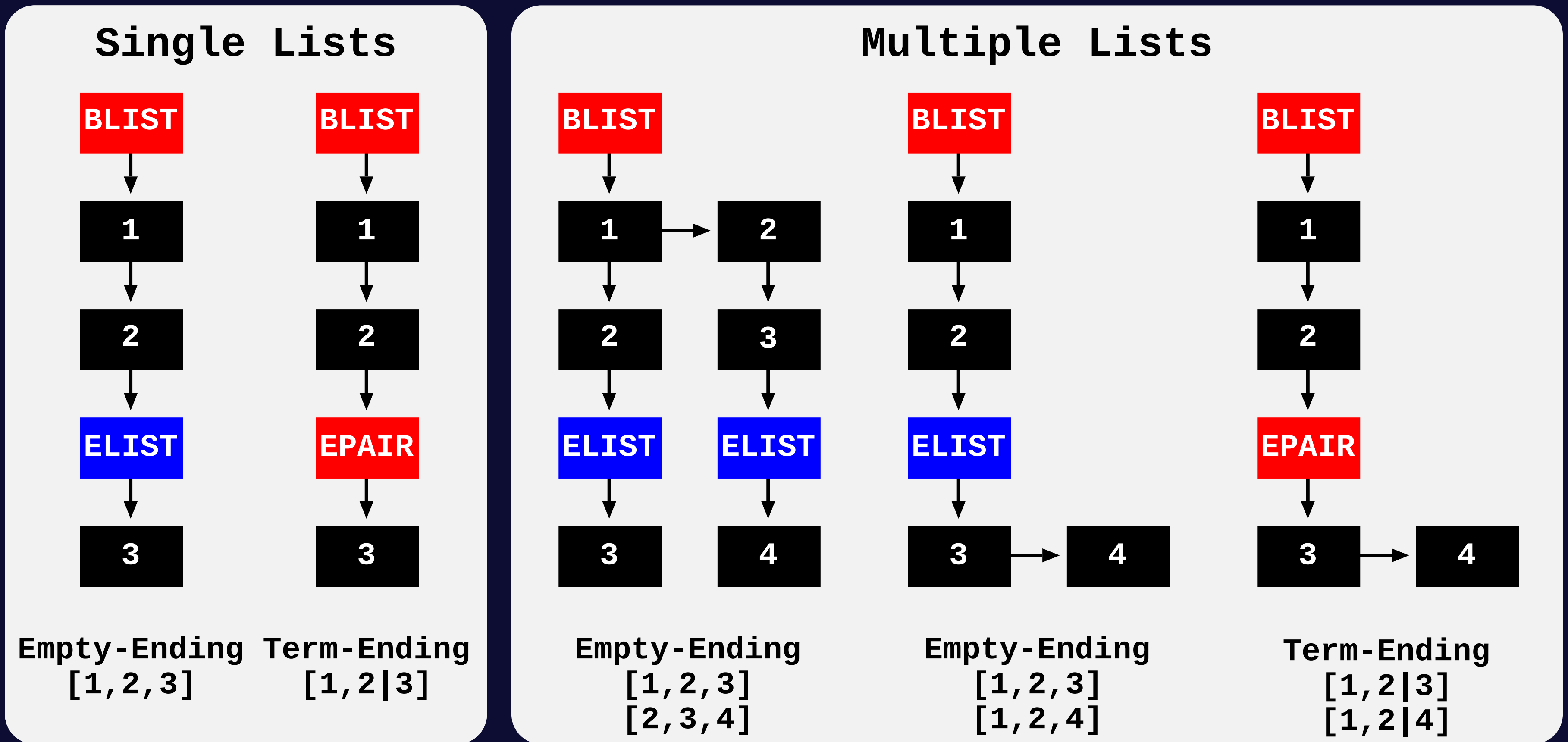
Compact Lists: Final Approach



Compact Lists: Final Approach



Compact Lists: Final Approach



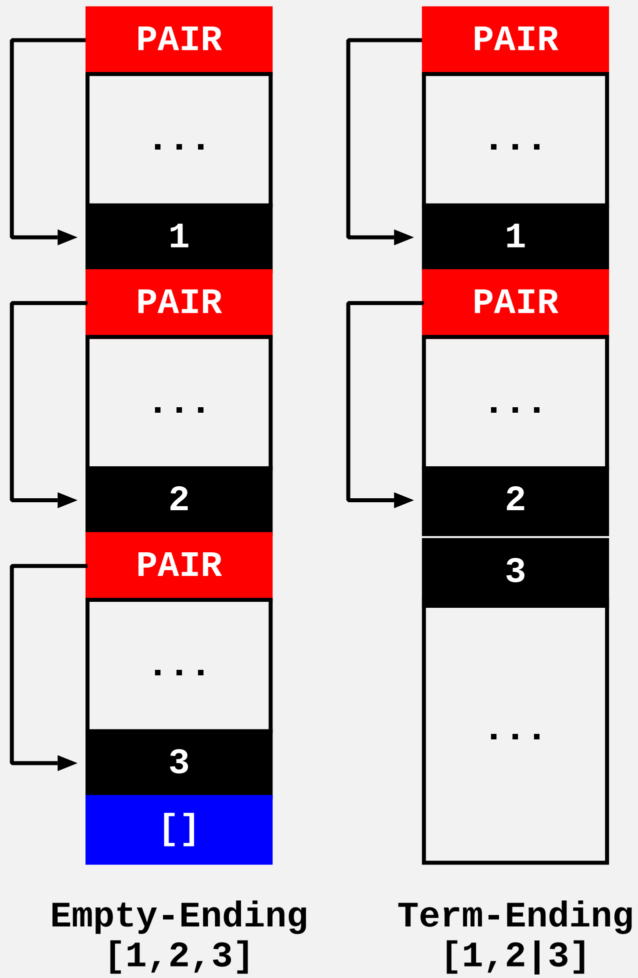
- $N * [E_1, ..., E_S]$: $NS + N + 1$ (1st different) $N + S + 1$ (last different)
- $N * [E_1, ...|E_S]$: $NS + N + 1$ (1st different) $N + S + 1$ (last different)

Compact Lists: Final Approach

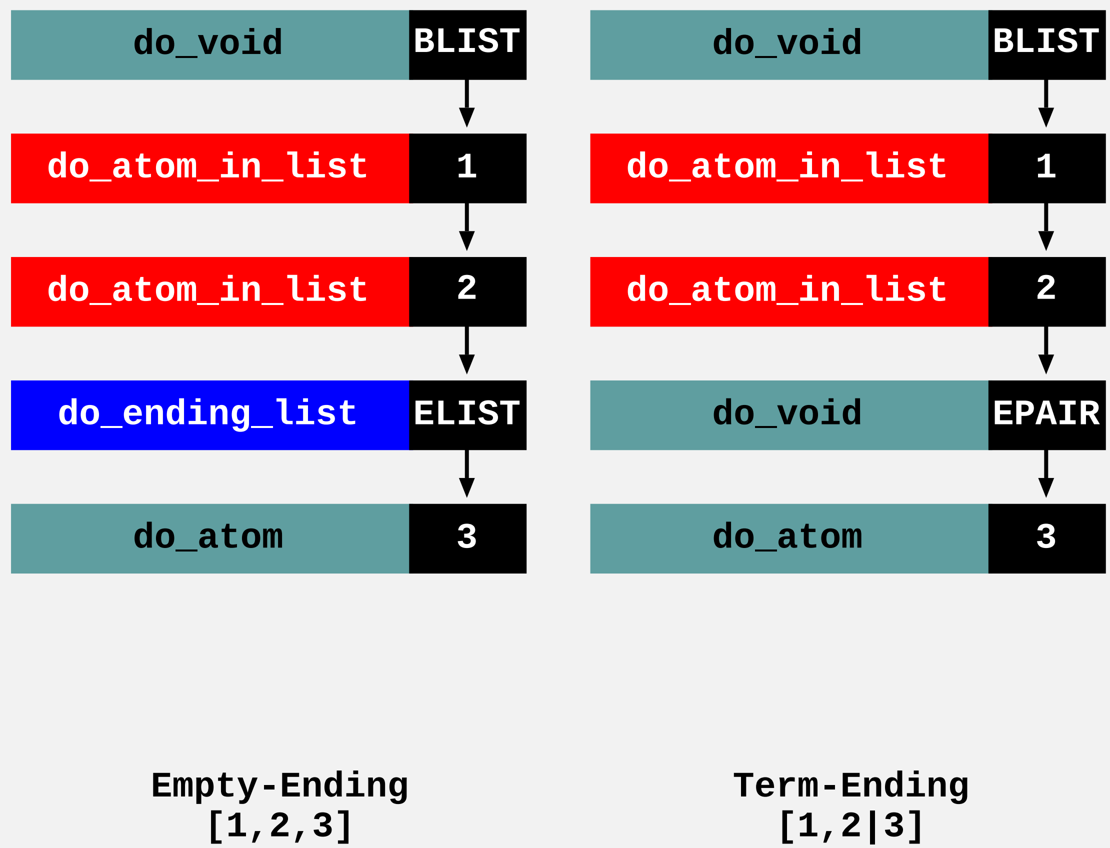
List Terms	Standard Lists	Compact Lists
First different		
$N * [E_1, ..., E_S]$	$2NS + 1$	$NS + N + 1$
$N * [E_1, ...\|E_S]$	$2NS - 2N + 1$	$NS + N + 1$
Last different		
$N * [E_1, ..., E_S]$	$2N + 2S - 1$	$N + S + 1$
$N * [E_1, ...\|E_S]$	$N + 2S - 2$	$N + S + 1$

Compact Lists: Compiled Tries

WAM Representation



Compiled Tries



Experimental Results

Empty-Ending 100,000 Lists	YapTab Mem	YapTab Store	YapTab Load	YapTab Cmp	YapTab+CL / YapTab Mem	YapTab+CL / YapTab Store	YapTab+CL / YapTab Load	YapTab+CL / YapTab Cmp
First different								
$[E_1, ..., E_{60}]$	234,375	1036	111	105	**0.51**	**0.52**	**0.71**	**0.69**
$[E_1, ..., E_{80}]$	312,500	1383	135	128	**0.51**	**0.52**	**0.73**	**0.64**
$[E_1, ..., E_{100}]$	390,625	1733	166	170	**0.51**	**0.53**	**0.67**	**0.55**
Last different								
$[E_1, ..., E_{60}]$	3,909	138	50	7	**0.50**	**0.75**	**0.64**	**0.56**
$[E_1, ..., E_{80}]$	3,909	171	71	8	**0.50**	**0.81**	**0.61**	**0.40**
$[E_1, ..., E_{100}]$	3,910	211	82	9	**0.50**	**0.76**	**0.62**	**0.44**

Table memory usage (in KBytes) and store/load times (in milliseconds) for empty-ending lists using YapTab with and without support for compact lists.

Experimental Results

Term-Ending 100,000 Lists	YapTab Mem	YapTab Store	YapTab Load	YapTab Cmp	YapTab+CL / YapTab Mem	YapTab+CL / YapTab Store	YapTab+CL / YapTab Load	YapTab+CL / YapTab Cmp
First different								
$[E_1, ...\|E_{60}]$	230,469	1028	113	97	**0.52**	**0.54**	**0.67**	**0.64**
$[E_1, ...\|E_{80}]$	308,594	1402	138	134	**0.51**	**0.53**	**0.69**	**0.63**
$[E_1, ...\|E_{100}]$	386,719	1695	162	163	**0.51**	**0.55**	**0.66**	**0.60**
Last different								
$[E_1, ...\|E_{60}]$	1,956	121	45	4	1.00	**0.86**	**0.82**	1.00
$[E_1, ...\|E_{80}]$	1,956	150	59	4	1.00	**0.88**	**0.72**	1.00
$[E_1, ...\|E_{100}]$	1,957	194	96	4	1.00	**0.88**	**0.53**	1.00

Table memory usage (in KBytes) and store/load times (in milliseconds) for term-ending lists using YapTab with and without support for compact lists.

Conclusions and Further Work

- We have presented a new and more compact representation of list terms for tabled data that avoids the recursive nature of the WAM representation by removing unnecessary intermediate pair tokens.
- Our experimental results are quite interesting, they clearly show that with compact lists, it is possible not only to reduce the memory usage overhead, but also the running time of the execution for storing and loading list terms.
- As further work we intend to explore the impact of our proposal in concrete real-world applications, such as, Inductive Logic Programming and Probabilistic Logic Learning applications, that heavily use list terms to represent, respectively, hypotheses and proofs in trie data structures.