CS5733 Program Synthesis

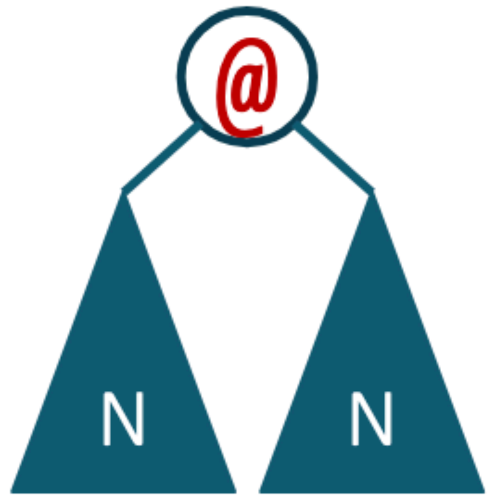
#9.Weighted Enumerative Search & Representation Based Search

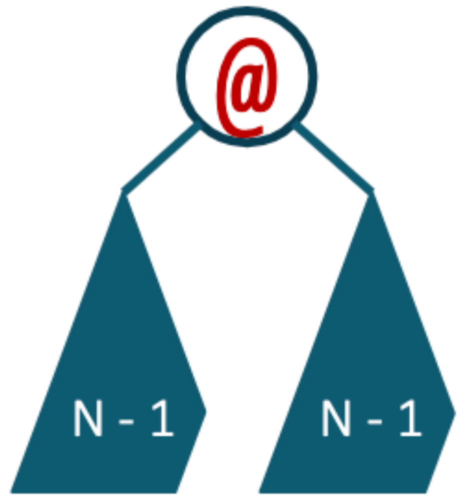
Ashish Mishra, August 30, 2024

Recap: Scaling enumerative search

Prune

Discard useless subprograms





`m * N`^{2} `m * (N - 1)`^{2}

Prioritize

Explore more promising candidates first

```
P = { [0][N..N] ,
      x[N..N]   ,
      ... }
```

← dequeue this first

Order of search

Enumerative search explores programs by depth / size

- Good default bias: small solution is likely to generalize
- But far from perfect

Result:

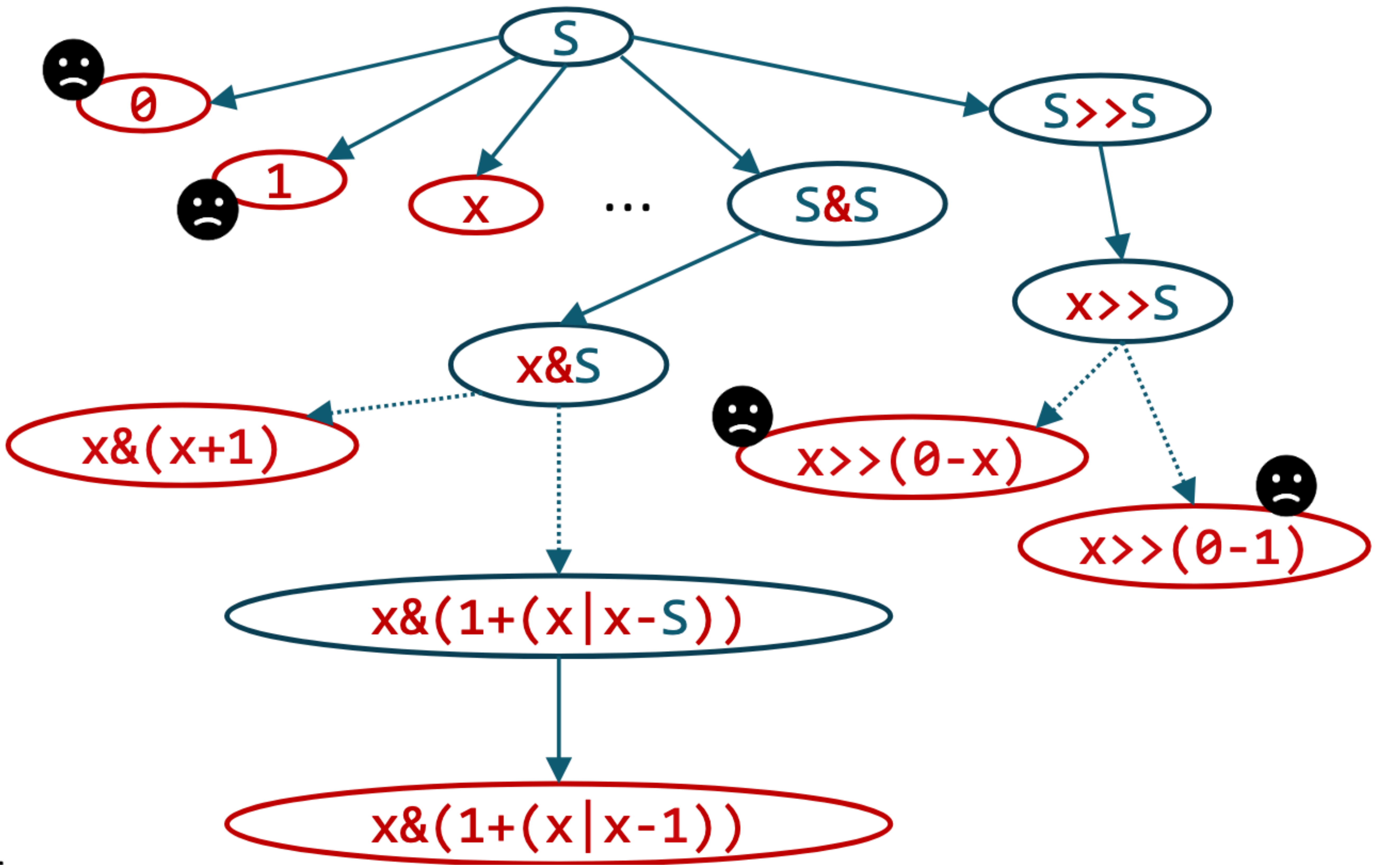
- Scales poorly with the size of the smallest solution to a given spec

Top-down search (revisited)

Turn off the rightmost sequence of **1**s:

```
00101 → 00100
01010 → 01000
10110 → 10000
```

```
S ->  0 | 1 | x |
      S + S     |
      S - S     |
      S & S     |
      S | S     |
      S << S    |
      S >> S
```



Explores many unlikely programs

Biasing the search

Idea: explore programs in the order of lieklihood, not size

Q1: how do we know which programs are likely?

- hard-code domain knowledge
- learn from a corpus of programs
- learn on the fly

Q2: how do we use this information to guide search?

- our focus today!

Weighted enumerative search

Example: DeepCoder

Balog et al. DeepCoder: Learning to Write Programs. ICLR'17

Probabilistic Grammars

Weighted top-down search

Weighted bottom-up search

DeepCoder

Input: IO-examples

[-17 -3 4 11 0 -5 -9 13 6 6 -8 11]
→ [-12 -20 -32 -36 -68]

DeepCoder

Output: Program in a list DSL

```
a <- [int]
b <- Filter (<0) a
c <- Map (*4) b
d <- Sort c
e <- Reverse d
```

A SQL inspired DSL

DeepCoder

Input: IO-examples

```
[-17 -3 4 11 0 -5 -9 13 6 6 -8 11]
→ [-12 -20 -32 -36 -68]
```

↓ neural network

component weights

(+1)	(-1)	(*2)	(/2)	(*-1)	(**2)	(*3)	(/3)	(*4)	(/4)	(>0)	(>0)	(%2==1)	(%2==0)	HEAD	LAST	MAP
.0	.0	.1	.0	.0	.0	.0	.0	1.0	.0	.0	1.0	.0	.2	.0	.0	1.0

FILTER	SORT	REVERSE	TAKE	DROP	ACCESS	ZIPWITH	SCANL1	+	-	*	MIN	MAX	COUNT	MINIMUM	MAXIMUM	SUM
1.0	1.0	.7	.0	.1	.0	.4	.0	.0	.1	.0	.2	.1	.0	.0	.0	.0

↓ weighted search

Output: Program in a list DSL
Goal: Minimize sum of component weights

DeepCoder: search strategies

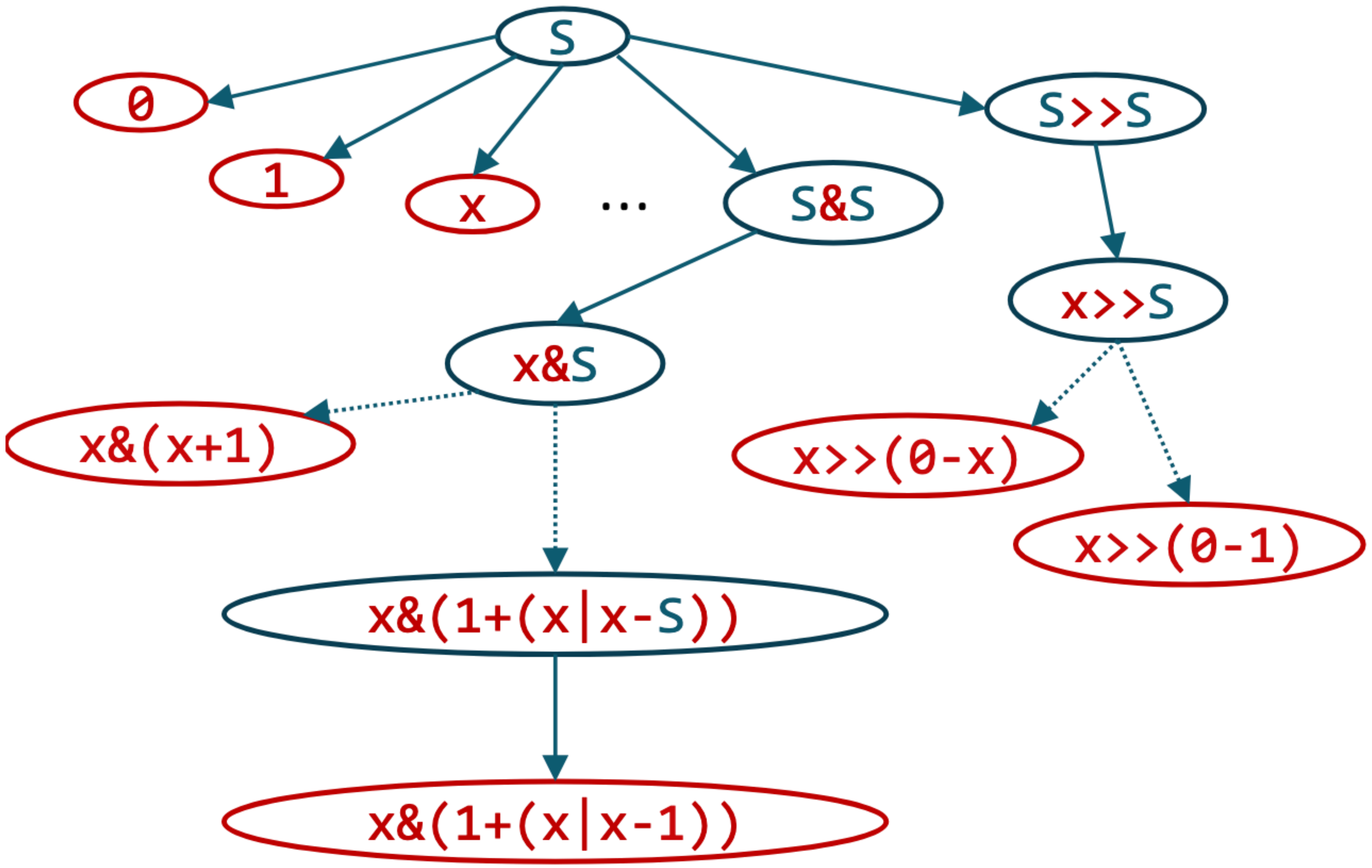
Top-down DFS

- Picks expansions for the current non-terminal in the order of probability

Sort-and-add

- start with N most probable functions
- when search fails, add next N functions

Pros and cons?



Recall: goal is to explore programs in the order of total weight!

Weighted enumerative search

DeepCoder

Probabilistic Grammars

Weighted top-down search

Weighted bottom-up search

Probabilistic Language Models

Originated in Natural Language Processing

In general: a probability distribution over sentences in a language

- $P(s)$ for $s \in L$

In practice:

- must be in a form that can be used to guide search
- for enumerative search: probabilistic (or weighted) grammars

Probabilistic (Tree) Grammar

regular tree grammar

production probability (given context)

$$\langle G, \wp \rangle$$

Production probability: $\wp : R \times T_\Sigma(N) \to [0,1]$

- for example: $\wp(S \to x \mid S) = 0.3$ $\quad \wp(S \to x \mid x - S) = 0.0001$
- only defined for contexts where rule's LHS is the leftmost non-terminal
- probabilities of all productions in the same context add up to 1:

$$\forall \tau. S \to^* \tau \wedge \tau \notin T_\Sigma \Rightarrow \sum_{r \in dom(P(.|\tau))} P(r \mid \tau) = 1$$

Term probability:

- let $S = \tau_0 \to^{r_1} \tau_1 \to^{r_2} \ldots \to^{r_n} \tau_n = \tau$ be the unique derivation of partial program τ

$$\wp(\tau) = \prod_{i=1}^{n} \wp(r_i \mid \tau_i)$$

Types of context

$$\wp : \mathrm{R} \times T_{\Sigma}(N) \rightarrow [0,1]$$

In general, can depend on any part of the context term!

But this is unwieldy

- bad for learning
- bad for (some) search algorithms

In practice we want to restrict the context

- PCFG   Philip Resnik ACL '92
- n-grams
- PHOG

Probabilistic Context-Free Grammars (PCFG)

$\wp: R \to [0,1]$

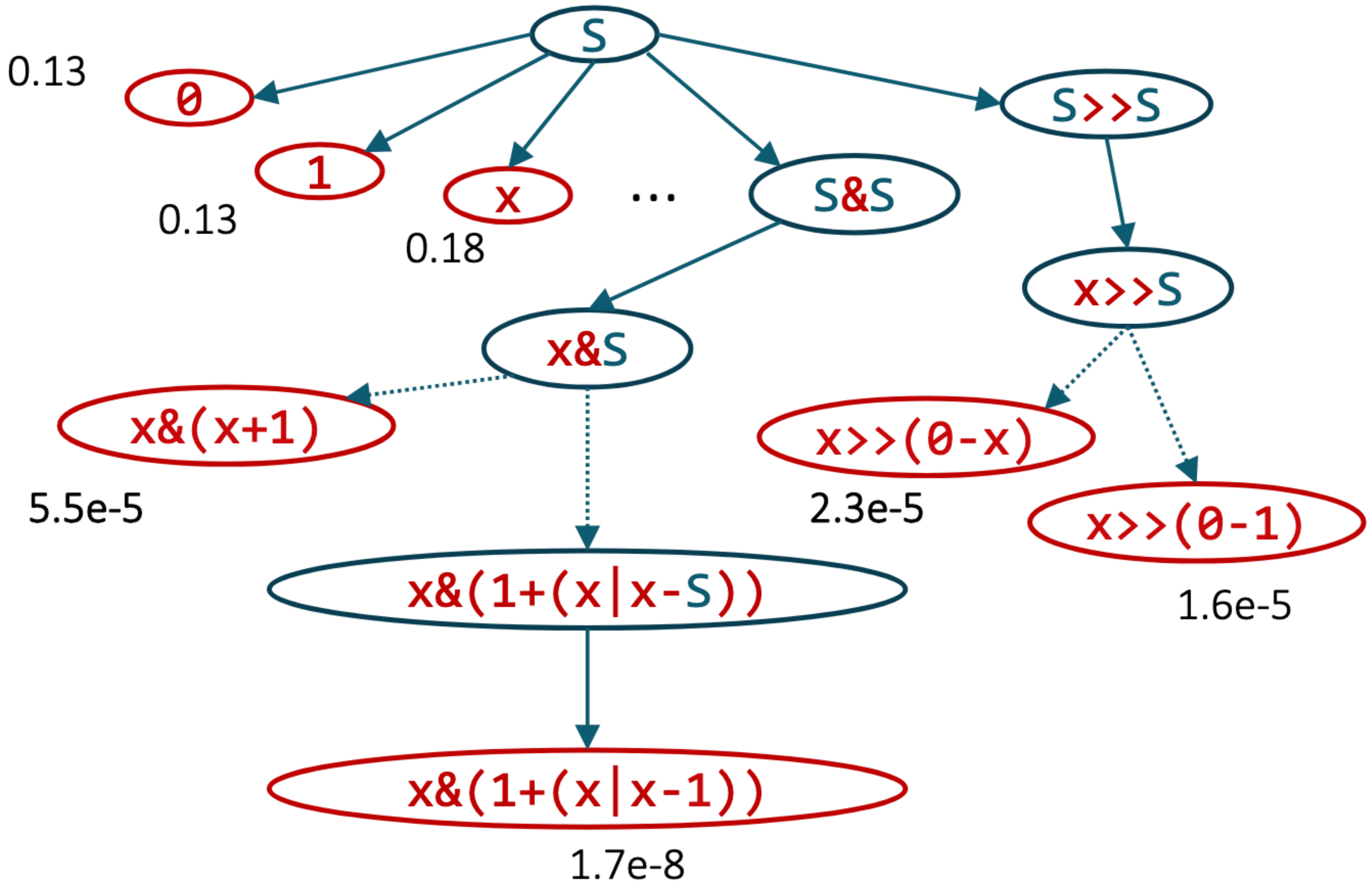
Encodes the popularity of each production (operation)

- here: variable more likely than constant, plus more likely than shift

	$\wp(R)$
`S -> 0`	0.13
`S -> 1`	0.13
`S -> x`	0.18
`S -> S + S`	0.11
`S -> S - S`	0.11
`S -> S & S`	0.12
`S -> S | S`	0.12
`S -> S << S`	0.05
`S -> S >> S`	0.05

Probabilistic Context-Free Grammars (PCFG)

Rule	$\wp(R)$
S -> 0	0.13
S -> 1	0.13
S -> x	0.18
S -> S + S	0.11
S -> S - S	0.11
S -> S & S	0.12
S -> S | S	0.12
S -> S << S	0.05
S -> S >> S	0.05



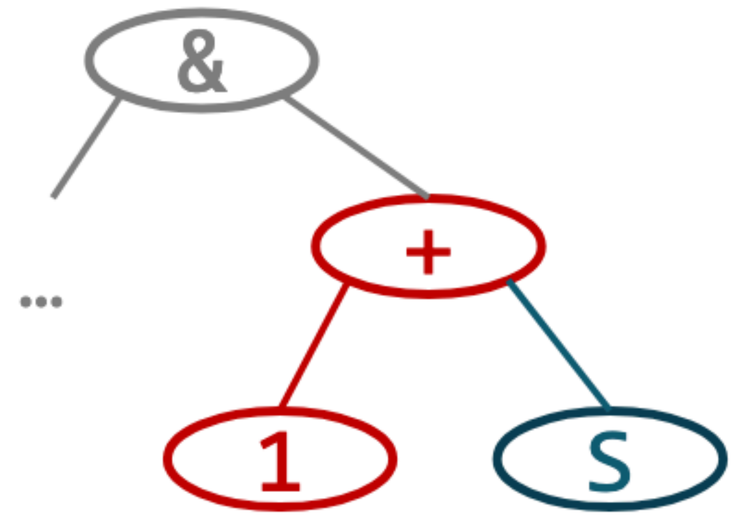
N-grams

```
N[left sibling, parent] -> rhs
```

		$\wp$
`S[x,-] ->`	`1`	0.72
`S[x,-] ->`	`x`	0.02
`S[x,-] ->`	`S + S`	0.12
`S[x,-] ->`	`S - S`	0.12
`...`		
`S[1,+] ->`	`1`	0.26
`S[1,+] ->`	`x`	0.25
`S[1,+] ->`	`S + S`	0.19
`S[1,+] ->`	`S - S`	0.08

Encodes likelihood of a production in a fixed context

- fixed set of AST nodes determined relative to the focus nonterminal
- e.g. left sibling and parent



- here: **`x`** is not likely in **`x - S`** but likely in **`1 + S`**

Probabilistic Higher-Order Grammar (PHOG)

The same fixed context might not work for every problem

Idea:

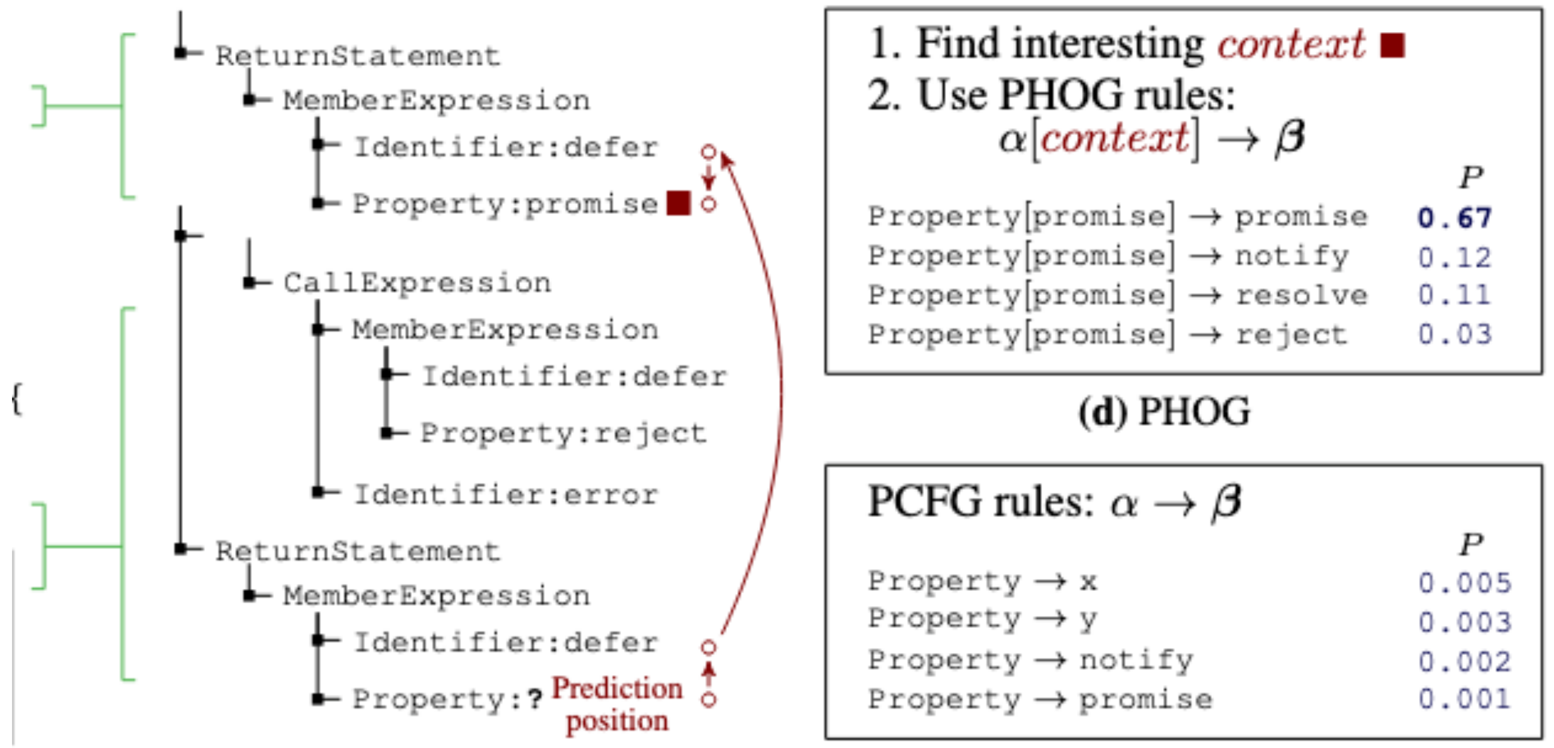
1. define context as a program that traverses the AST
2. learn the best context together with probabilities

Bielik, Raychev, Vechev. PHOG: Probabilistic Model for Code. ICML’16

PHOG Example

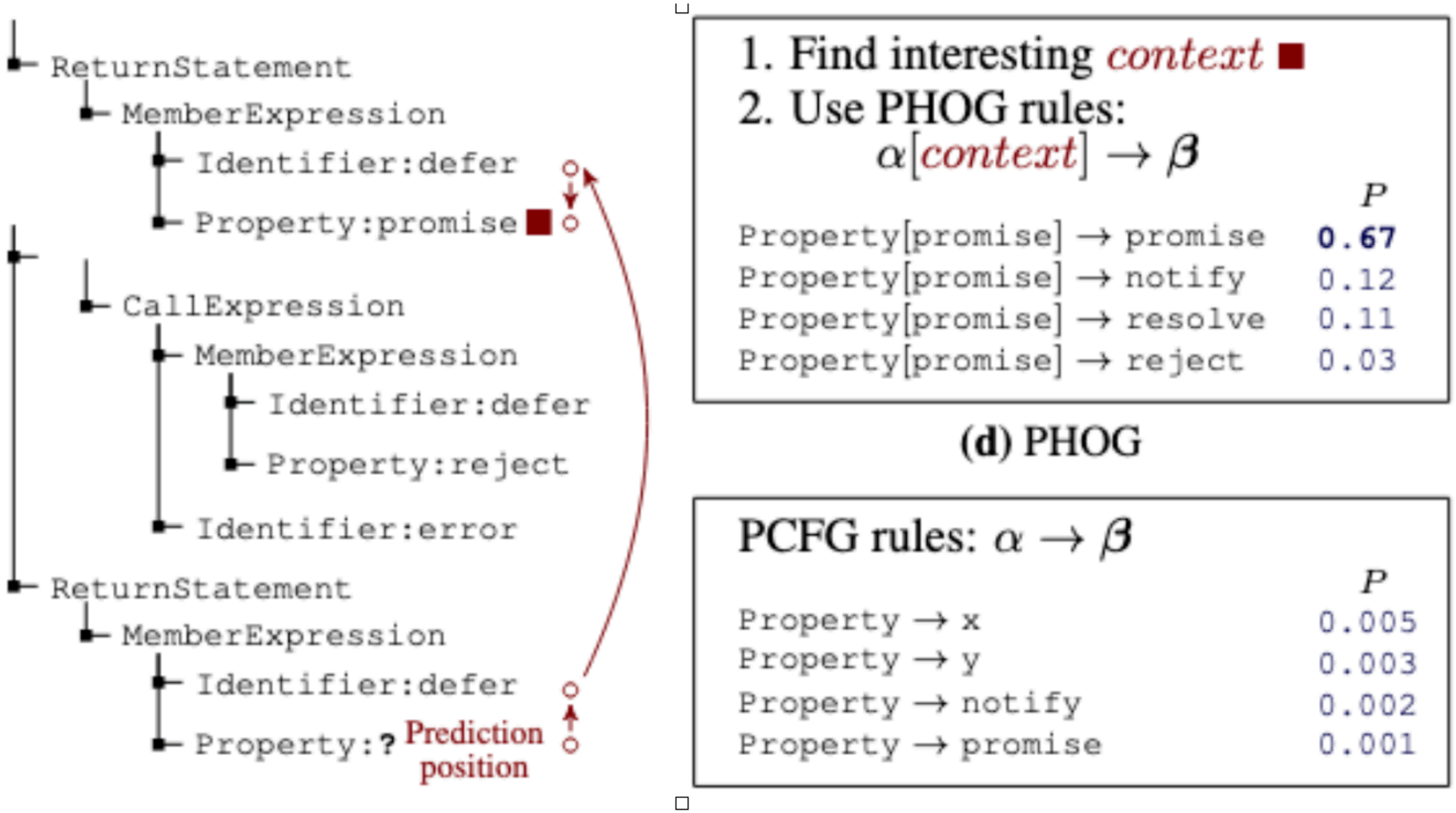
```
awaitReset = function() {
  ...
  return defer.promise;
}

...

awaitRemoved = function() {
  ...
  fail(function(error) {
    if (error.status === 401) {
      ...
    }
    defer.reject(error);
  });
  return defer.
}
```

	P
promise	**0.67**
notify	0.12
resolve	0.11
reject	0.03



(d) PHOG

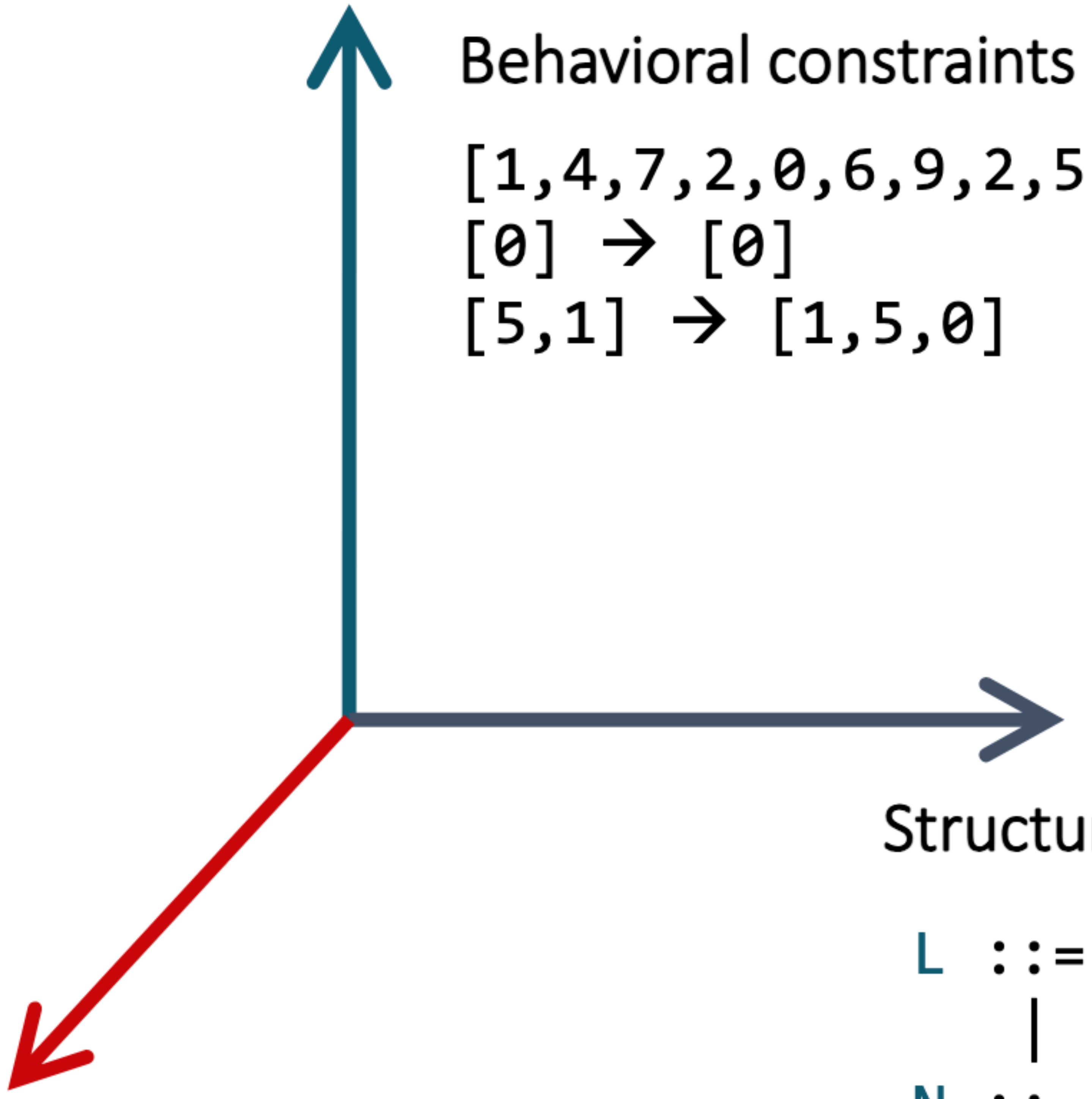
PHOG Example



How to get the context

Representation-based Search for Synthesis

The problem statement



Behavioral constraints = examples

```
[1,4,7,2,0,6,9,2,5]  →  [1,2,4,7,0]
[0] → [0]
[5,1] → [1,5,0]
```

Search strategy?

Enumerative
Representation-based
Stochastic
Constraint-based

Structural constraints = grammar

```
L ::= sort(L)  |  L[N..N]
    |  L + L  |  [N]  |  x
N ::= find(L,N)  |  0
```

Representation-based search

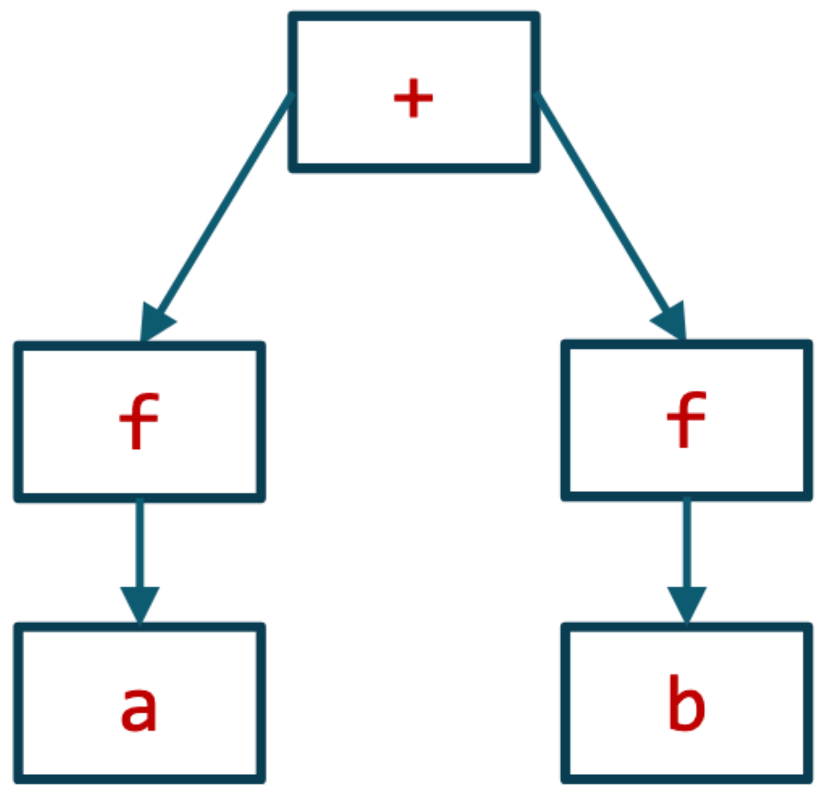
Idea:

1. build a data structure that compactly represents good parts of the program space
2. extract solution from that data structure

Compact term representation

Consider the space of 9 programs:

```
f(a) + f(a)    f(a) + f(b)    f(a) + f(c)
f(b) + f(a)    f(b) + f(b)    f(b) + f(c)
f(c) + f(a)    f(c) + f(b)    f(c) + f(c)
```



Can we represent this compactly?

- observation 1: same top level structure, independent subterms

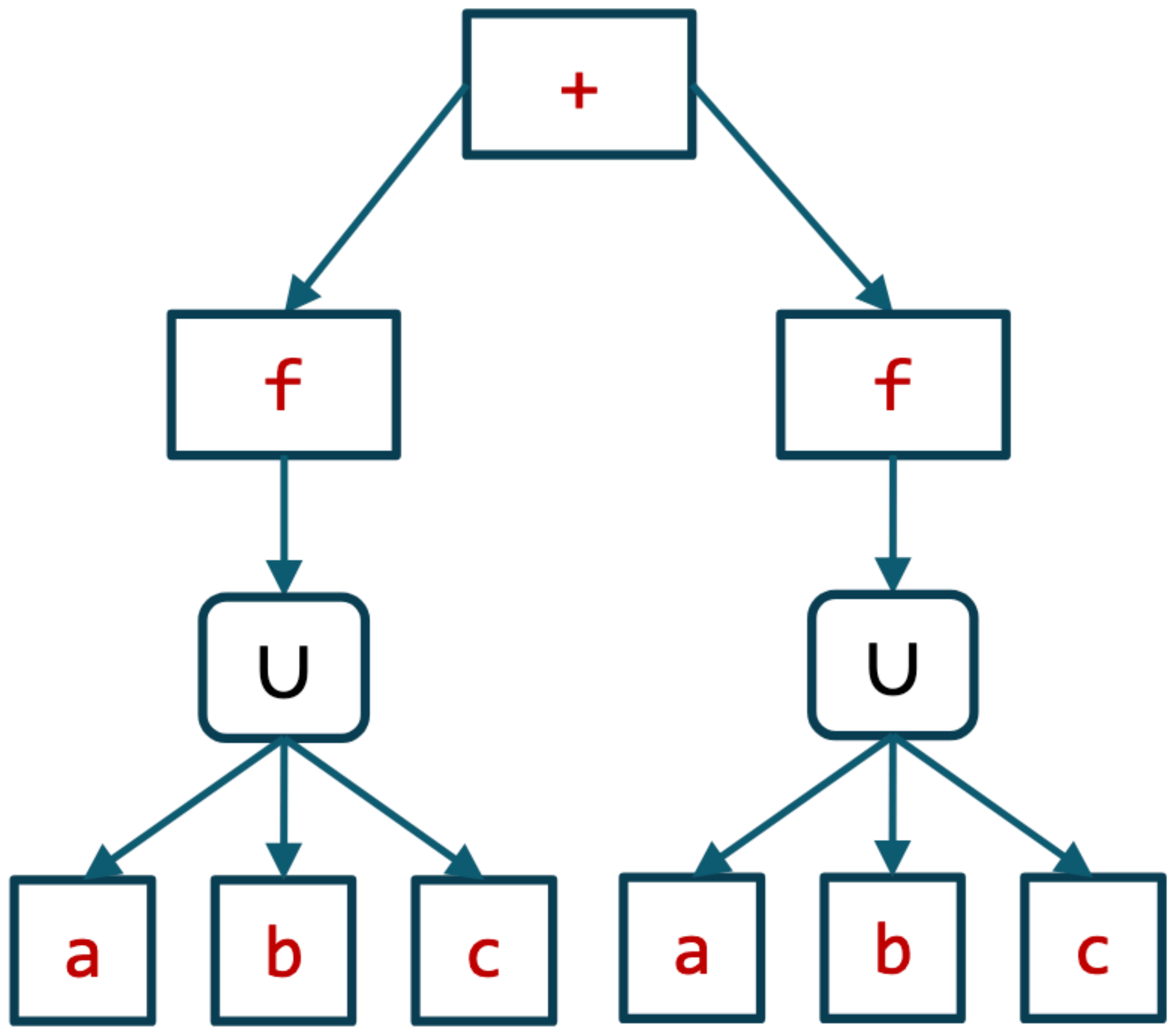
Compact term representation

Consider the space of 9 programs:

```
f(a) + f(a)   f(a) + f(b)   f(a) + f(c)
f(b) + f(a)   f(b) + f(b)   f(b) + f(c)
f(c) + f(a)   f(c) + f(b)   f(c) + f(c)
```

Can we represent this compactly?

- observation 1: same top level structure, independent subterms
- observation 2: shared sub-spaces



Compact term representation

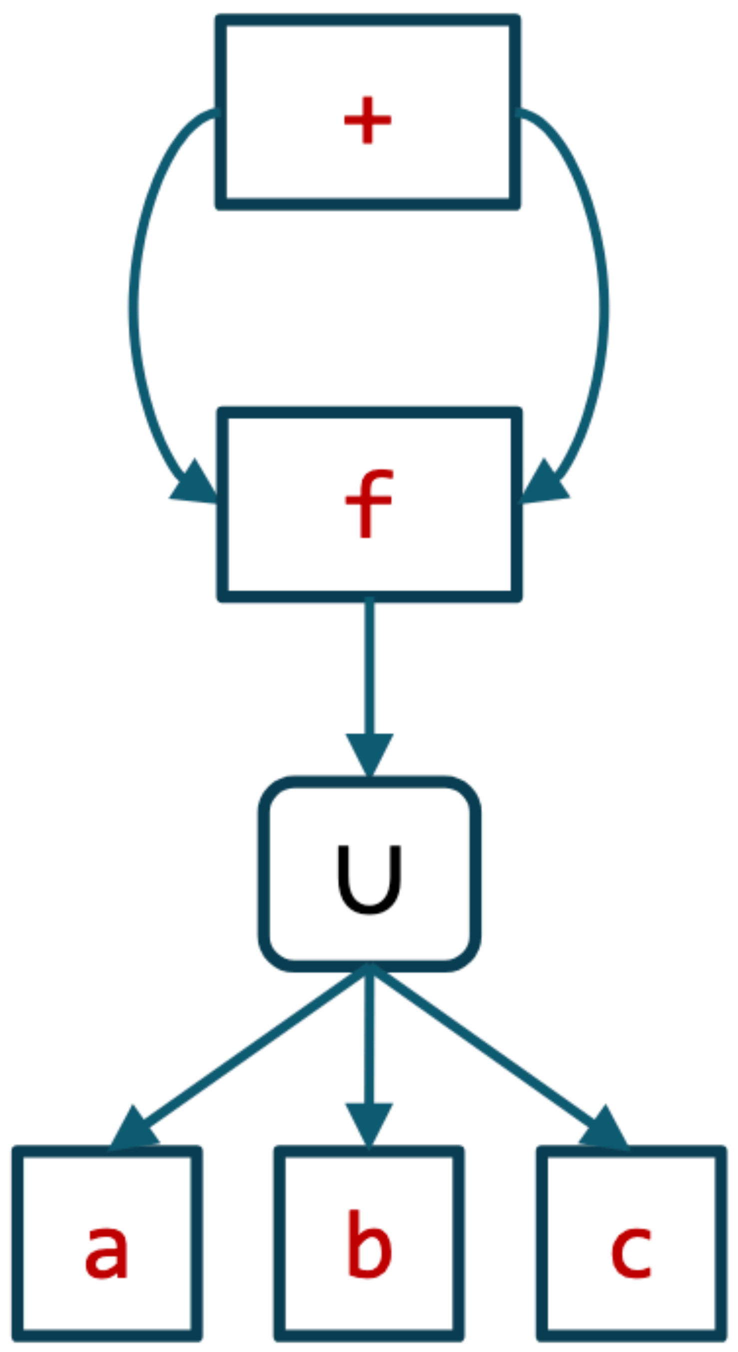
Consider the space of 9 programs:

```
f(a) + f(a)   f(a) + f(b)   f(a) + f(c)
f(b) + f(a)   f(b) + f(b)   f(b) + f(c)
f(c) + f(a)   f(c) + f(b)   f(c) + f(c)
```

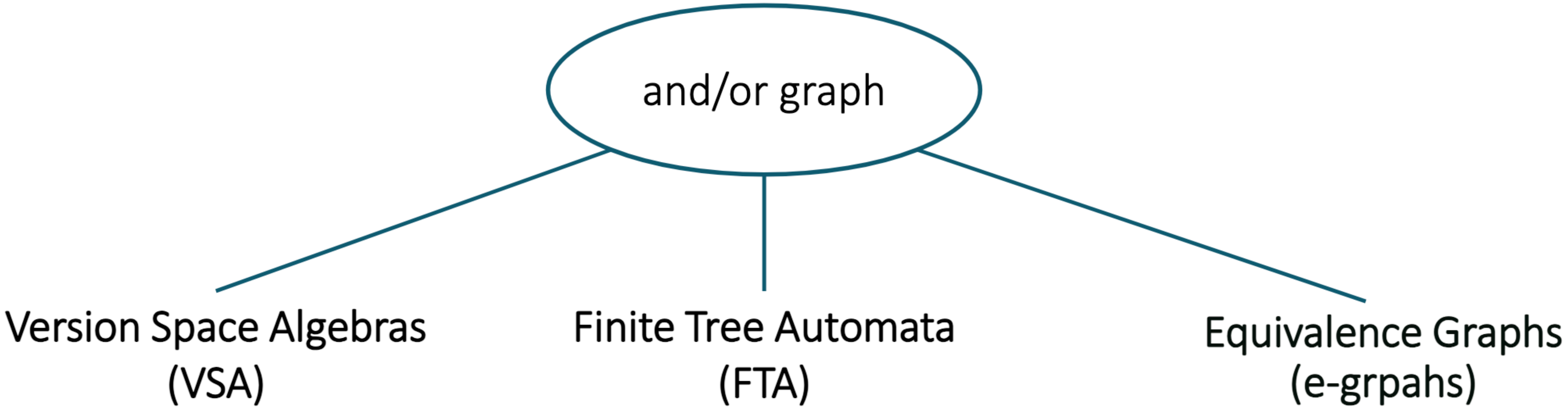
Can we represent this compactly?

- observation 1: same top level structure, independent subterms
- observation 2: shared sub-spaces

Key idea: use an **and-or** graph!



Representation-based search



Version Space Formulation

Hypothesis space H

- Space of possible functions $In \rightarrow Out$

Version Space $VS_{H,D} \subseteq H$

- H is the original hypothesis space
- D is a set of examples i_j, o_j
- $h \in VS_{H,D} \Leftrightarrow \forall\ i, o \in D\ \ h(i) = \mathrm{o}$

Hypothesis space provides *restriction bias*

- Defines what functions one is allowed to consider
- *Preference bias* needs to be provided independently

Version Space Algebra

Set of Operations to manipulate and compose VSs

Idea: build a graph that succinctly represents the space of *all* programs consistent with examples

- called a version space

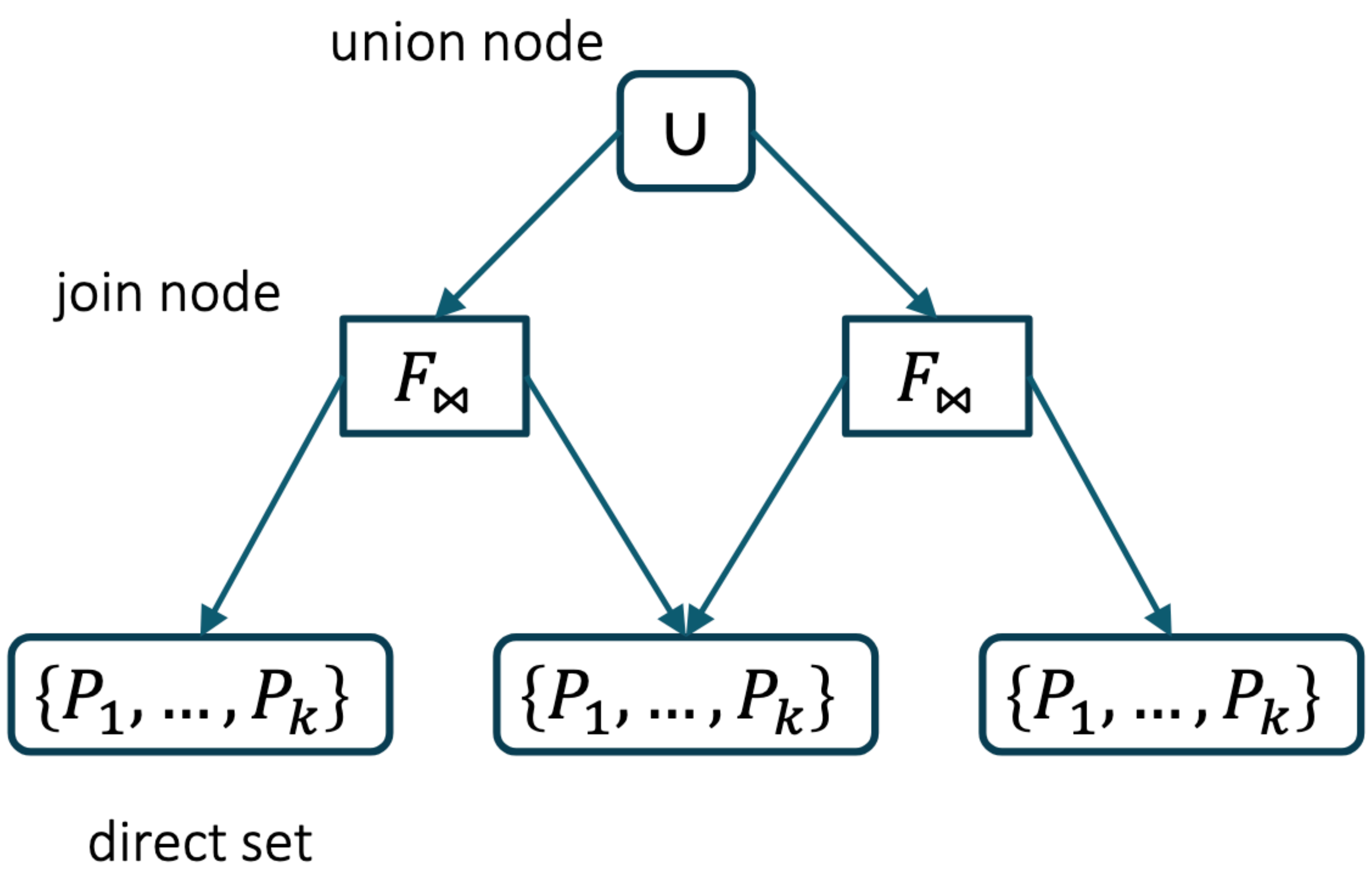
Operations on version spaces:

- `learn <i, o> → VS`
- $VS_1 \cap VS_2 \rightarrow VS$
- `extract VS → program`

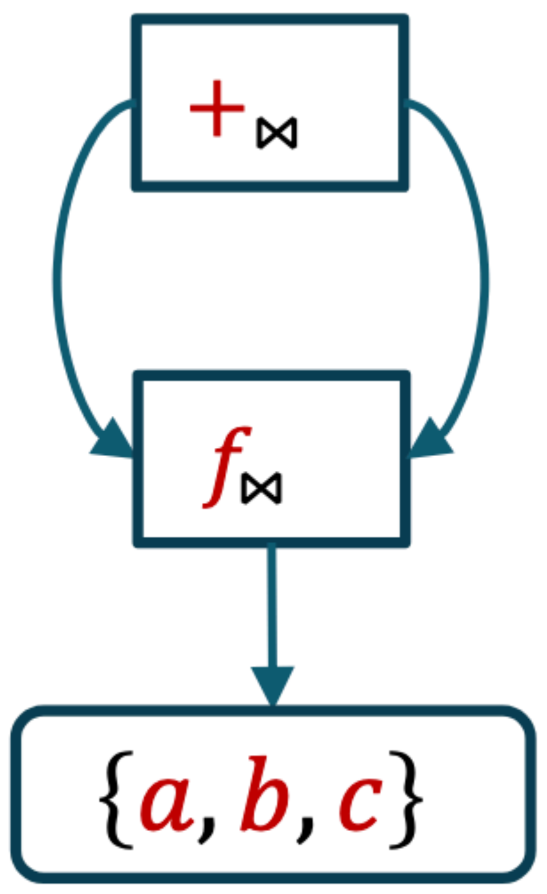
Algorithm:

1. learn a VS for each example
2. intersect them all
3. extract any (or best) program

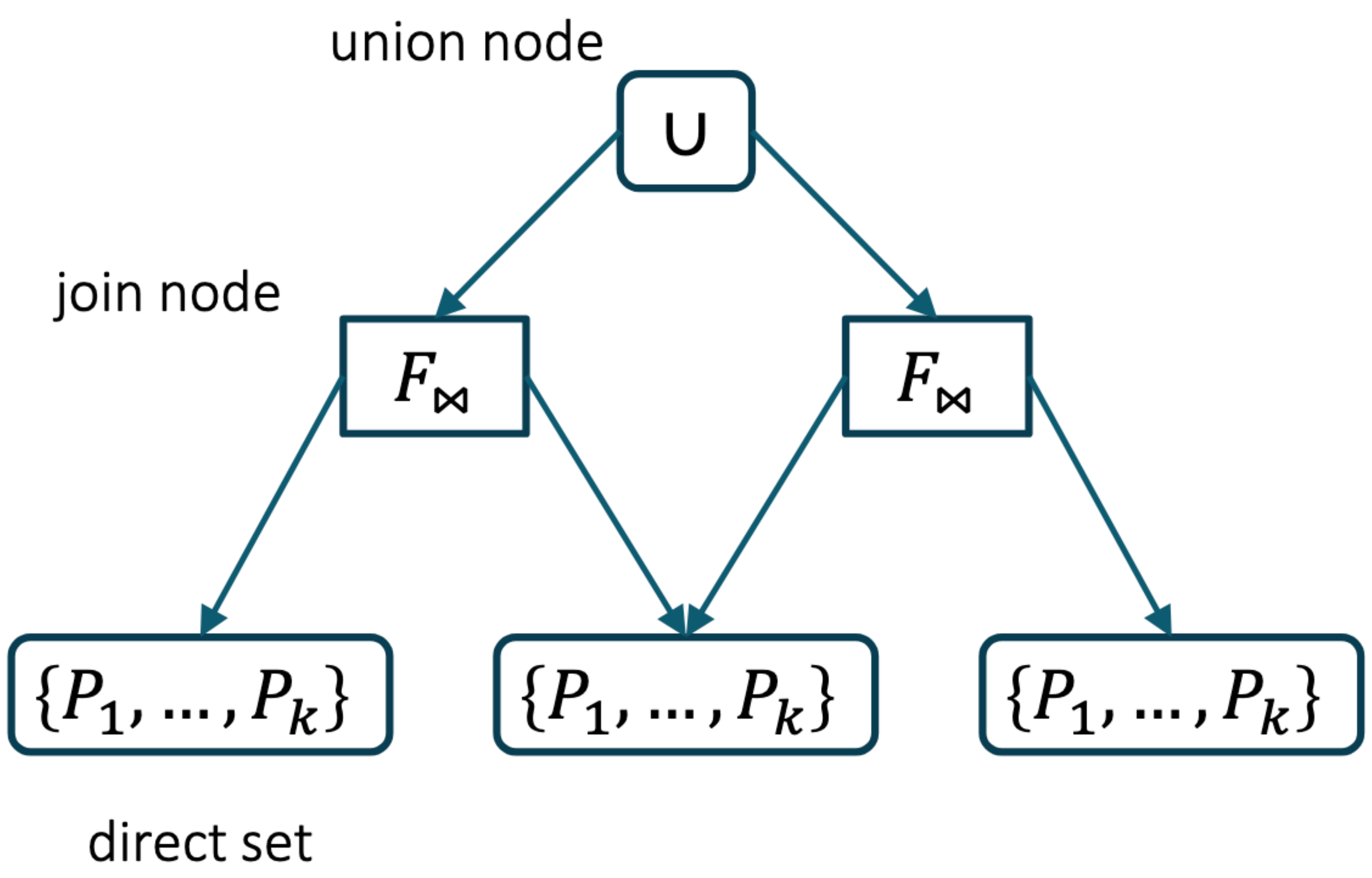
Version Space Algebra



example:



Version Space Algebra



Volume of a VSA (the number of nodes) $V(VSA)$

Size of a VSA (the number of programs) $|VSA|$

$$V(VSA) = O(\log|VSA|)$$

VSA-based search

Mitchell: *Generalization as search*. AI 1982

Lau, Domingos, Weld. *Version space algebra and its application to programming by example*. ICML 2000

Gulwani: *Automating string processing in spreadsheets using input-output examples.* POPL 2011.

- Follow-up work: BlinkFill, FlashExtract, FlashRelate, …
- generalized in the PROSE framework

FlashFill: Automating String Processing in Spreadsheets Using Input-Output Examples

[Gulwani '11]

A language for text manipulation:

Simplified grammar:

``E ::= F | concat(F, E)``		"Trace" expression
``F ::= cstr(str) | sub(P, P)``		Atomic expression
``P ::= cpos(num) | pos(R, R)``		Position expression
$R ::= \texttt{tokens}(T_1, \ldots, T_n)$		Regular expression
``T ::= C | C+``		Token expression
``C ::= ws | digit | alpha | Alpha | $ | ^ | …``		

FlashFill Example

```
  0123456789…
"Hello POPL 2024" → "POPL'2024"
"Goodbye PLDI 2021" → "PLDI'2021"
```

```
concat(
  sub(pos(ws, Alpha), pos(Alpha, ws)),
  concat(
    cstr("'"),
    sub(pos(ws, digit), pos(digit, $))))
```

$$
\begin{aligned}
E &::= F \mid \mathtt{concat}(F,\ E) \\
F &::= \mathtt{cstr}(\mathit{str}) \mid \mathtt{sub}(P,\ P) \\
P &::= \mathtt{cpos}(\mathit{num}) \mid \mathtt{pos}(R,\ R) \\
R &::= \mathtt{tokens}(T_1,\ \ldots,\ T_n) \\
T &::= C \mid C+
\end{aligned}
$$

VSAs for Flashfill

Recall operations on version spaces:

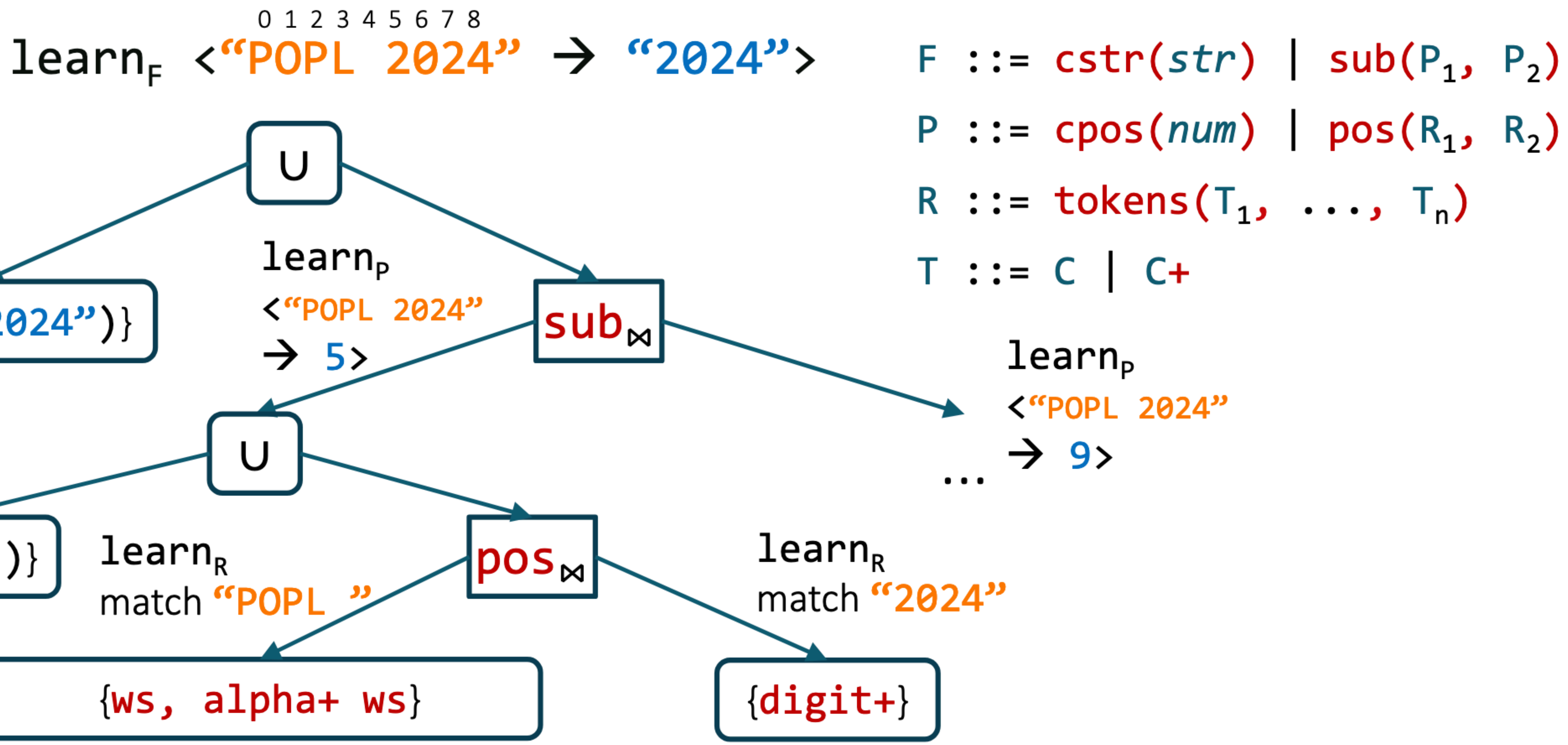
- learn <i, o> → VS
- $VS_1 \cap VS_2 \rightarrow VS$
- extract VS → program

How do we implement learn?

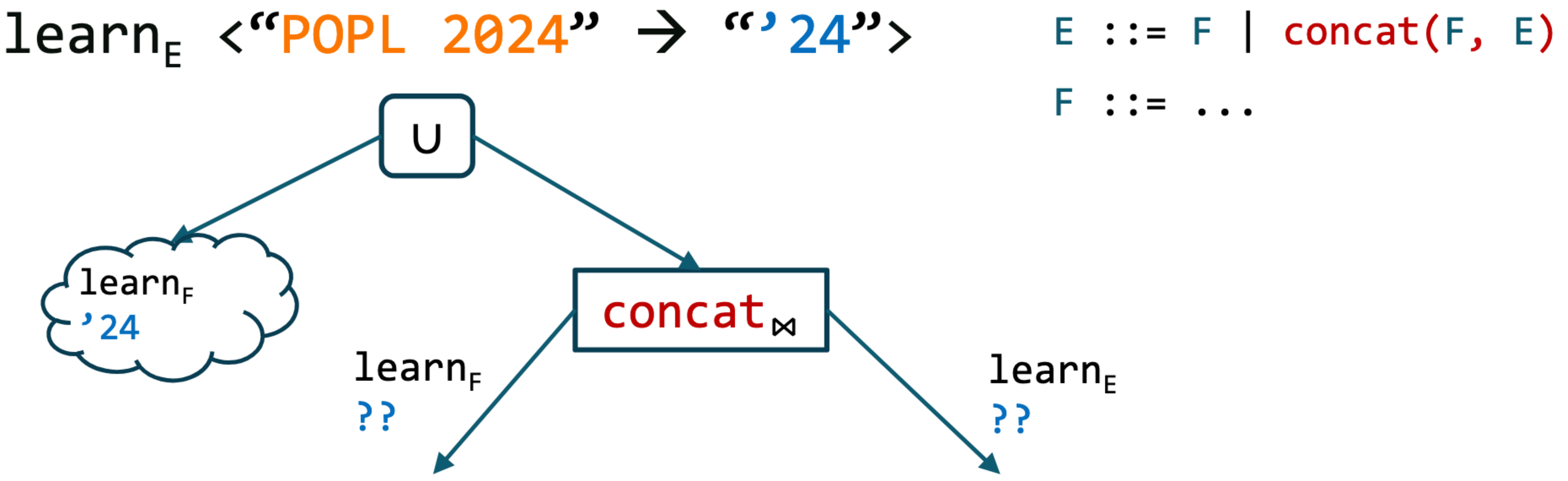
- define learn_N <i, o> for every non-terminal N
- build VS top-down, propagating <i, o> the example

```
E ::= F | concat(F, E)
F ::= cstr(str) | sub(P, P)
P ::= cpos(num) | pos(R, R)
R ::= tokens(T_1, ..., T_n)
T ::= C | C+
```

Learning atomic expressions



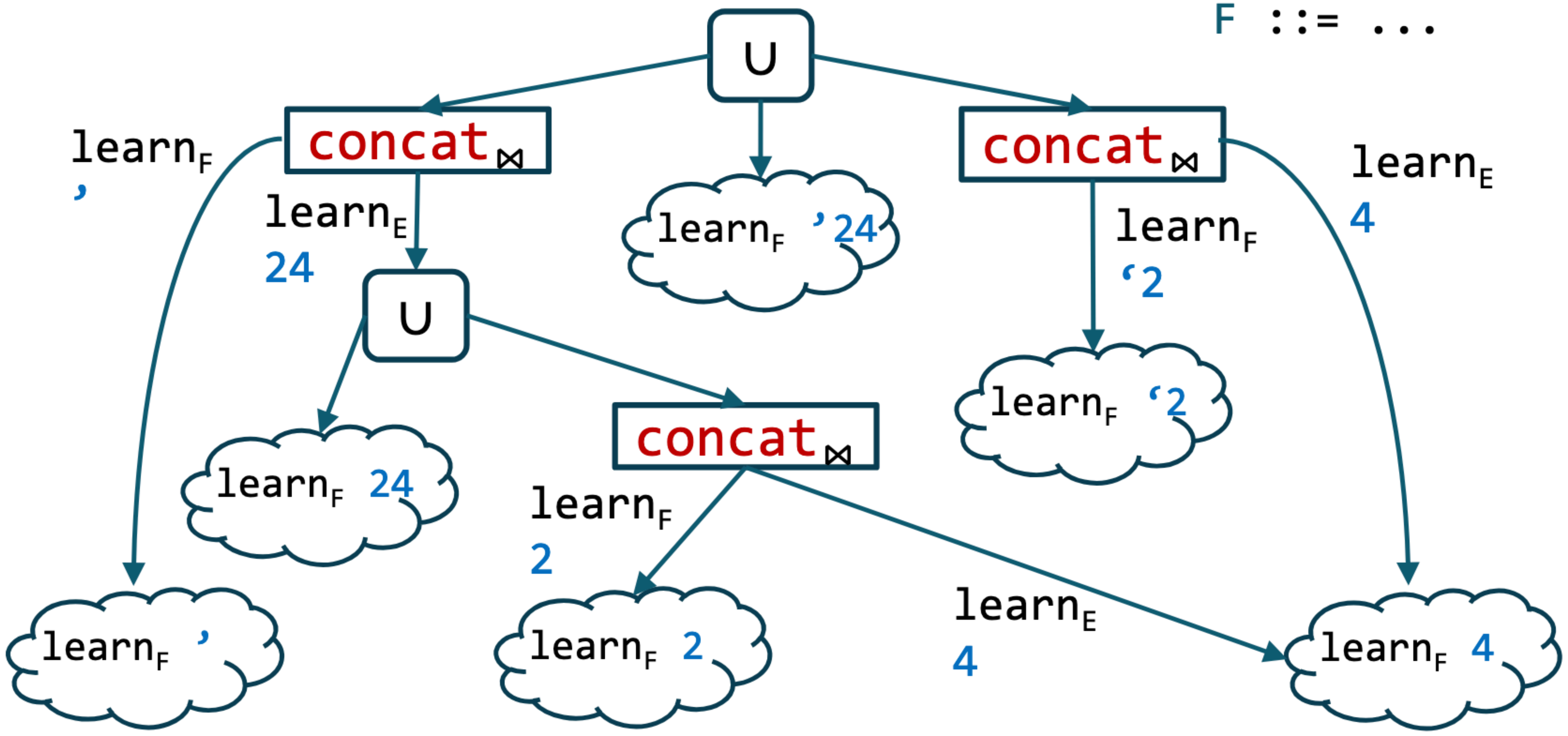
Learning trace expressions



Learning trace expressions

$\texttt{learn}_E$ <"POPL 2024" → "'24">

```
E ::= F | concat(F, E)
F ::= ...
```



VSAs for Flashfill

Recall operations on version spaces:

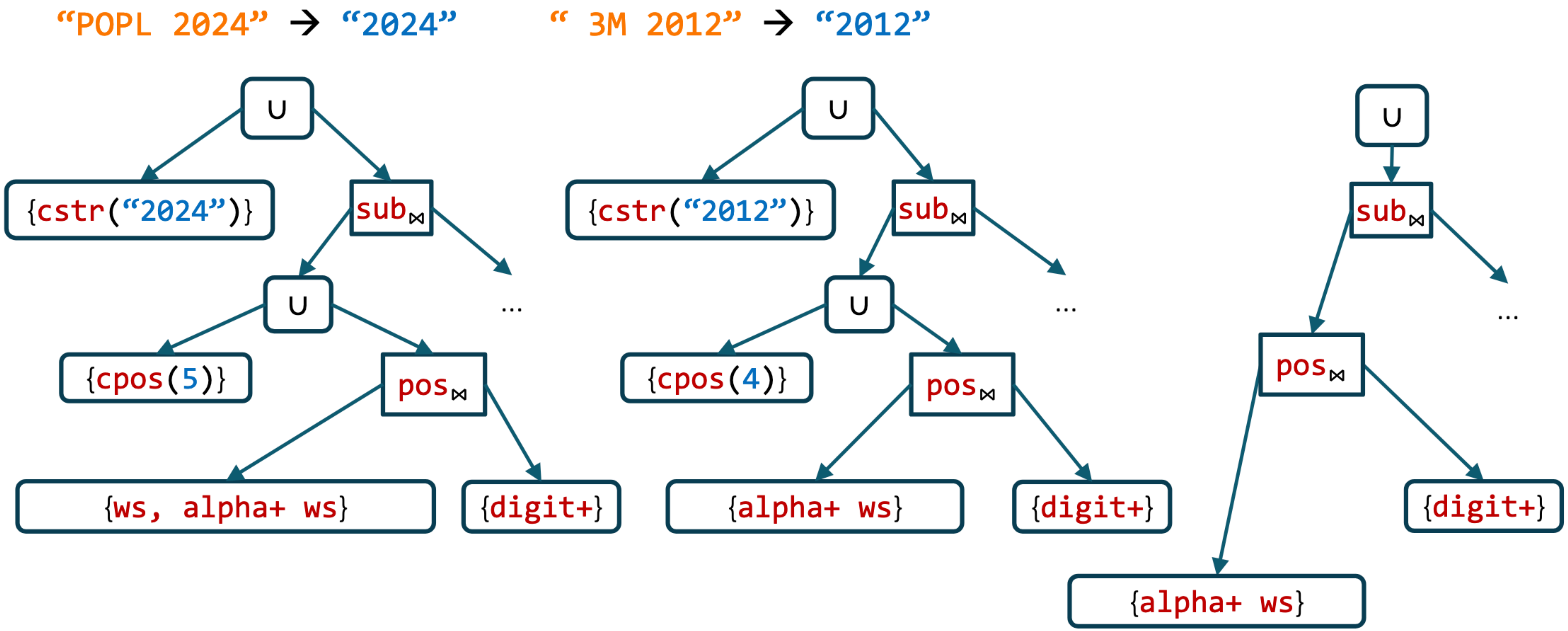
- `learn <i, o> → VS`
- $VS_1 \cap VS_2 \rightarrow VS$
- `extract VS → program`

How do we implement intersection?

- top-down
- union: intersect all pairs of children
- join: intersect children pairwise

```
E ::= F | concat(F, E)
F ::= cstr(str) | sub(P, P)
P ::= cpos(num) | pos(R, R)
R ::= tokens(T1, ..., Tn)
T ::= C | C+
```

Intersection



VSAs for Flashfill

Recall operations on version spaces:

- `learn <i, o> → VS`
- $VS_1 \cap VS_2 \rightarrow VS$
- `extract VS → program`

How do we implement extract?

- any program: just pick one child from every union
- best program: shortest path in a DAG

```
E ::= F | concat(F, E)
F ::= cstr(str) | sub(P, P)
P ::= cpos(num) | pos(R, R)
R ::= tokens(T1, ..., Tn)
T ::= C | C+
```

Discussion

Why could we build a finite representation of all solutions?

- Could we do it for this language?

$$E ::= F + F$$
$$F ::= k \mid x$$

$k \in \mathbb{Z}$ $\quad$ $+$ is integer addition

- What about this language?

$$E ::= E + 1 \mid x$$

DSL restrictions: efficiently invertible

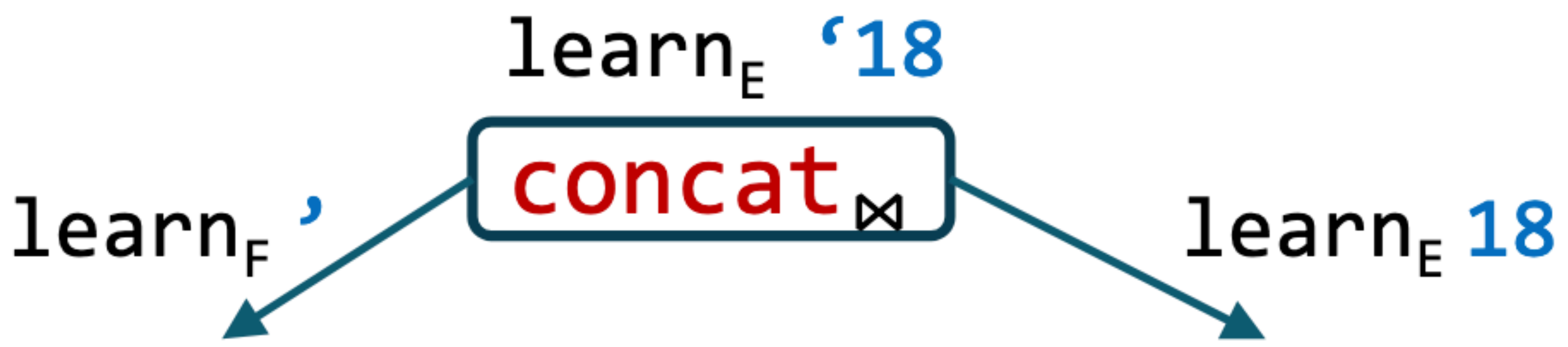
Every operator has a small, easily computable inverse

- Example when an inverse is small but hard to compute?

The space of sub-specs is finite

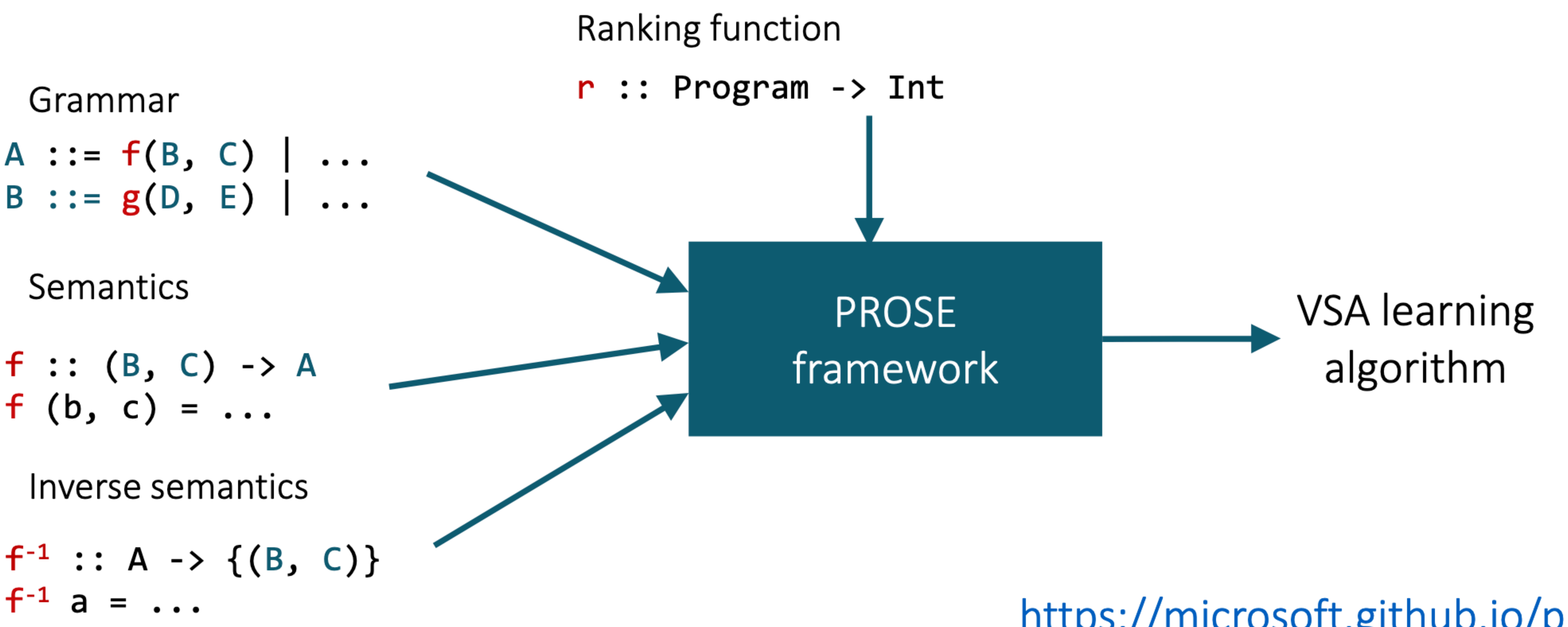
- either non-recursive grammar
- or finite space of values for the recursive non-terminal (e.g. bit-vectors)
- or every recursive production generates a strictly smaller spec

```
E ::= F | concat(F, E)
```



PROSE

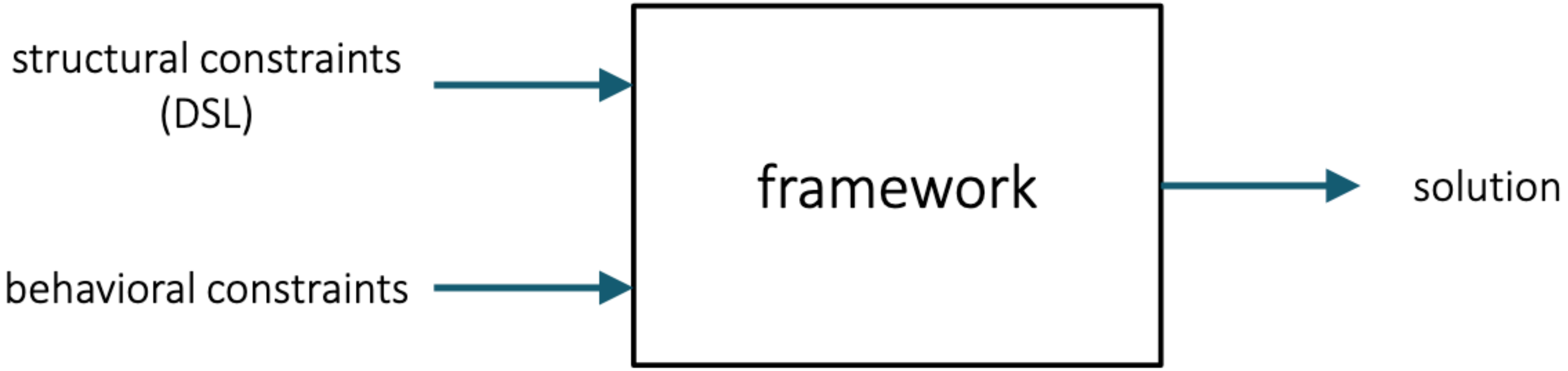
[Polozov, Gulwani ‘15]



https://microsoft.github.io/prose/

Synthesis frameworks

synthesis framework = a highly-configurable synthesizer



Synthesis frameworks

- Sketch (https://people.csail.mit.edu/asolar/)
- Rosette (https://emina.github.io/rosette/)
    - see also: https://www.cs.utexas.edu/~bornholt/post/building-synthesizer.html
- PROSE (https://www.microsoft.com/en-us/research/project/prose-framework/)

VSAs Again

Version Space Formulation

Hypothesis space H

- Space of possible functions $In \rightarrow Out$

Version Space $VS_{H,D} \subseteq H$

- H is the original hypothesis space
- D is a set of examples i_j, o_j
- $h \in VS_{H,D} \Leftrightarrow \forall\ i, o \in D\ \ h(i) = \mathrm{o}$

Hypothesis space provides *restriction bias*

- Defines what functions one is allowed to consider
- *Preference bias* needs to be provided independently

Partial Ordering of hypothesis

Partial order $h_1 \sqsubseteq h_2$

- h_2 is "better" than h_1

Ex: For boolean hypothesis

- "better" == more general
- $h_1 \sqsubseteq h_2 \Leftrightarrow (h_1 \Rightarrow h_2)$

For booleans, VS forms a lattice

Partial Orders

Set P

Partial order ≤ such that ∀x,y,z∈P

- x ≤ x (reflexive)
- x ≤ y and y ≤ x implies x = y (asymmetric)
- x ≤ y and y ≤ z implies x ≤ z (transitive)

Can use partial order to define

- Upper and lower bounds
- Least upper bound
- Greatest lower bound

Upper Bounds

If $S \subseteq P$ then

- $x \in P$ is an upper bound of S if $\forall y \in S.\ y \leq x$
- $x \in P$ is the least upper bound of S if
  - x is an upper bound of S, and
  - $x \leq y$ for all upper bounds y of S
- $\vee$ - join, least upper bound, lub, supremum, sup
  - $\vee S$ is the least upper bound of S
  - $x \vee y$ is the least upper bound of $\{x,y\}$
- Often written as $\sqcup$ as well

Lower Bounds

If $S \subseteq P$ then

- $x \in P$ is a lower bound of S if $\forall y \in S.\ x \leq y$
- $x \in P$ is the greatest lower bound of S if
  - x is a lower bound of S, and
  - $y \leq x$ for all lower bounds y of S
- $\wedge$ - meet, greatest lower bound, glb, infimum, inf
  - $\wedge S$ is the greatest lower bound of S
  - $x \wedge y$ is the greatest lower bound of $\{x,y\}$

- Often written as $\sqcap$ as well

Lattices

If $x \wedge y$ and $x \vee y$ exist for all $x,y \in P$

then P is a lattice

If $\wedge S$ and $\vee S$ exist for all $S \subseteq P$

then P is a complete lattice

All finite lattices are complete

Example of a lattice that is not complete

- Integers I
- For any $x, y \in I$, $x \vee y = \max(x,y)$, $x \wedge y = \min(x,y)$
- But $\vee$ I and $\wedge$ I do not exist
- $I \cup \{+\infty, -\infty\}$ is a complete lattice

Partial Ordering of hypothesis

Partial order $h_1 \sqsubseteq h_2$

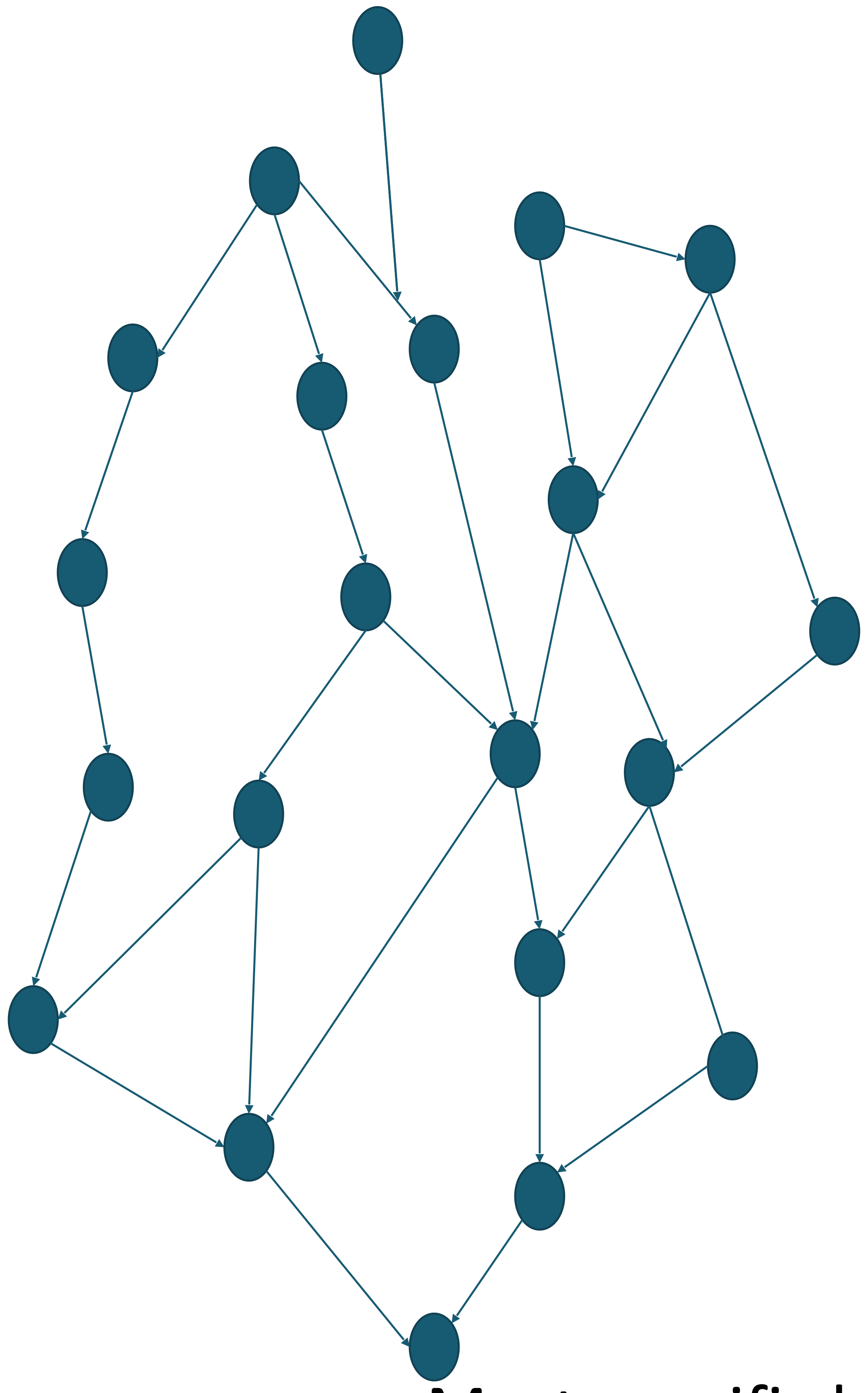
- h_2 is "better" than h_1

Ex: For boolean hypothesis

- "better" == more general
- $h_1 \sqsubseteq h_2 \Leftrightarrow (h_1 \Rightarrow h_2)$

For booleans, VS forms a lattice

- $h_1, h_2 \in VS \Rightarrow h_1 \sqcap h_2 = h_1 \wedge h_2 \in VS$



Most specific hypothesis that satisfies the observations

Boundary set representable

You can represent a VS by the pair (G,S) where

- G is most general hypothesis (i.e. $\top$)
- S is the most specific (i.e. $\bot$)

Applies in general when hypothesis space is partially ordered and version space is a lattice

Update

$$U(VS,\ d) = \{p \in VS \mid p(i) = o \text{ where } d = (i, o)\}$$

- Subset of a version space satisfying a new example d

Ex: For boolean HS

- VS=(G,S)
- If $d = (i, true)$

$$U(VS, d) = (G, S \vee \lambda x.\ if\ x = i\ then\ true\ else\ false)$$

- If $d = (i, false)$

$$U(VS, d) = (G \wedge \lambda x.\ if\ x = i\ then\ false\ else\ true, S)$$

Example: FindSuffix

FS_T: move to the position right before the next occurrence of T.

We shall go on to the end. We |shall fight in France, we |shall fight on the seas and oceans, we shall fight with growing confidence and growing strength in the air,…

$FS_{""}$

$FS_{"s"}$

$FS_{"sha"}$

$FS_{"shall"}$

$FS_{"shall\ fight"}$

$FS_{"shall\ fight\ on"}$

$FS_{"shall\ fight\ on\ the\ seas\ and\ oceans,\ we\ shall\ fight\ldots"}$

Example: FindSuffix

FS_T: move to the position right before the next occurrence of T.

$FS_{""}$

$FS_{"s"}$

$FS_{"sha"}$

$FS_{"shall"}$

$FS_{"shall\ fight"}$

$FS_{"shall\ fight\ on"}$

$FS_{"shall\ fight\ on\ the\ seas\ and\ oceans,\ we\ shall\ fight\ldots"}$

We shall go on to the end. We |shall fight in France, we |shall fight on the seas and oceans, we |shall fight with growing confidence and growing strength in the air,…

Example: FindSuffix

FS_T: move to the position right before the next occurrence of T.

We shall go on to the end. We |shall fight in France, we |shall fight on the seas and oceans, we |shall fight with growing confidence and growing strength in the air,…

$FS_{""}$

$FS_{"s"}$

$FS_{"sha"}$

$FS_{"shall"}$

$FS_{"shall\ fight"}$

$FS_{"shall\ fight\ on"}$

$FS_{"shall\ fight\ on\ the\ seas\ and\ oceans,\ we\ shall\ fight\ldots"}$

Example: FindSuffix

FS_T: move to the position right before the next occurrence of T.

We shall go on to the end. We |shall fight in France, we |shall fight on the seas and oceans, we |shall fight with growing confidence and growing strength in the air,…

$FS_{""}$

$FS_{"s"}$

$FS_{"sha"}$

$FS_{"shall"}$

$FS_{"shall\ fight"}$

$FS_{"shall\ fight\ on"}$

$FS_{"shall\ fight\ on\ the\ seas\ and\ oceans,\ we\ shall\ fight\dots"}$

Idea

If your hypothesis space is partially ordered and your VS are boundary set representable, you can represent and search very efficiently

If they are not?

Break them down into simpler hypothesis spaces!

Union

$$VS_{H_1 D} \cup VS_{H_2 D} = VS_{H_1 \cup H_2\ D}$$

FindSuffix U FindPrefix

We shall go on to the end. We |shall fight in France, we |shall fight on the seas and oceans, we |shall fight with growing confidence and growing strength in the air,…

FS(“sh”-”shall fight ”)

U

FP(“we “ – “, we”)

FindSuffix U FindPrefix

We shall go on to the end. We | shall fight in France, we | shall fight on the seas and oceans, we | shall fight with growing confidence and growing strength in the air,…

FS("sh"-"shall fight ")

U

∅

Join

$$VS_{H_1D_1} \bowtie VS_{H_2D_2} = \{\langle h_1, h_2\rangle \mid h_1 \in VS_{H_1D_1},\ h_2 \in VS_{H_2D_2},\ C(\langle h_1, h_2\rangle,\ D)\}$$

- Where $D_1 = \{d_1^i\}_{i=0..n}$ and $D_2 = \{d_2^i\}_{i=0..n}$ and $D = \{\langle d_1^i, d_2^i\rangle\}_{i=0..n}$
- $C(\langle h_1, h_2\rangle,\ D)$ means that $\langle h_1, h_2\rangle$ is consistent with the input output pairs in D

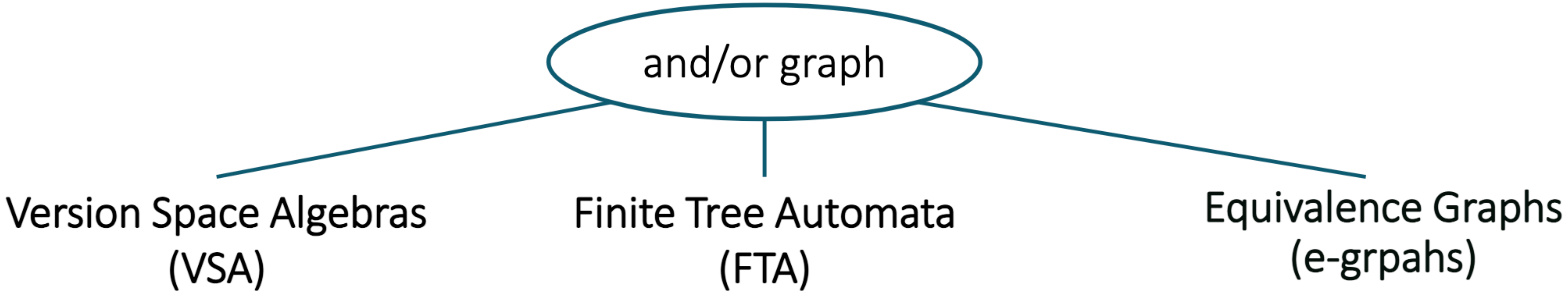
What does $\langle h_1, h_2\rangle$ mean? What about $\langle d_1, d_2\rangle$?

- Pair
- Composition $\langle h_1, h_2\rangle = h_1 \circ h_2$ and $\langle d_1, d_2\rangle = (d_1.in\ ,\ d_2.out)$

Independent join: C is unnecessary

- It's a property of $\langle .,\ .\rangle$
- True for pair, not for composition

Representation-based search



ops: learn-1, intersect, extract
DSL: efficiently invertible
similar to: top-down prop,
but can infer constants

Next Class

Logsitics

- Submission deadline, Sunday, No extension!
- Scores for the reading assignments, this Tuesday!
- We will start the Project selection.
- Next Class:
  - FTA
  - E-graphs
  - Equivalences.

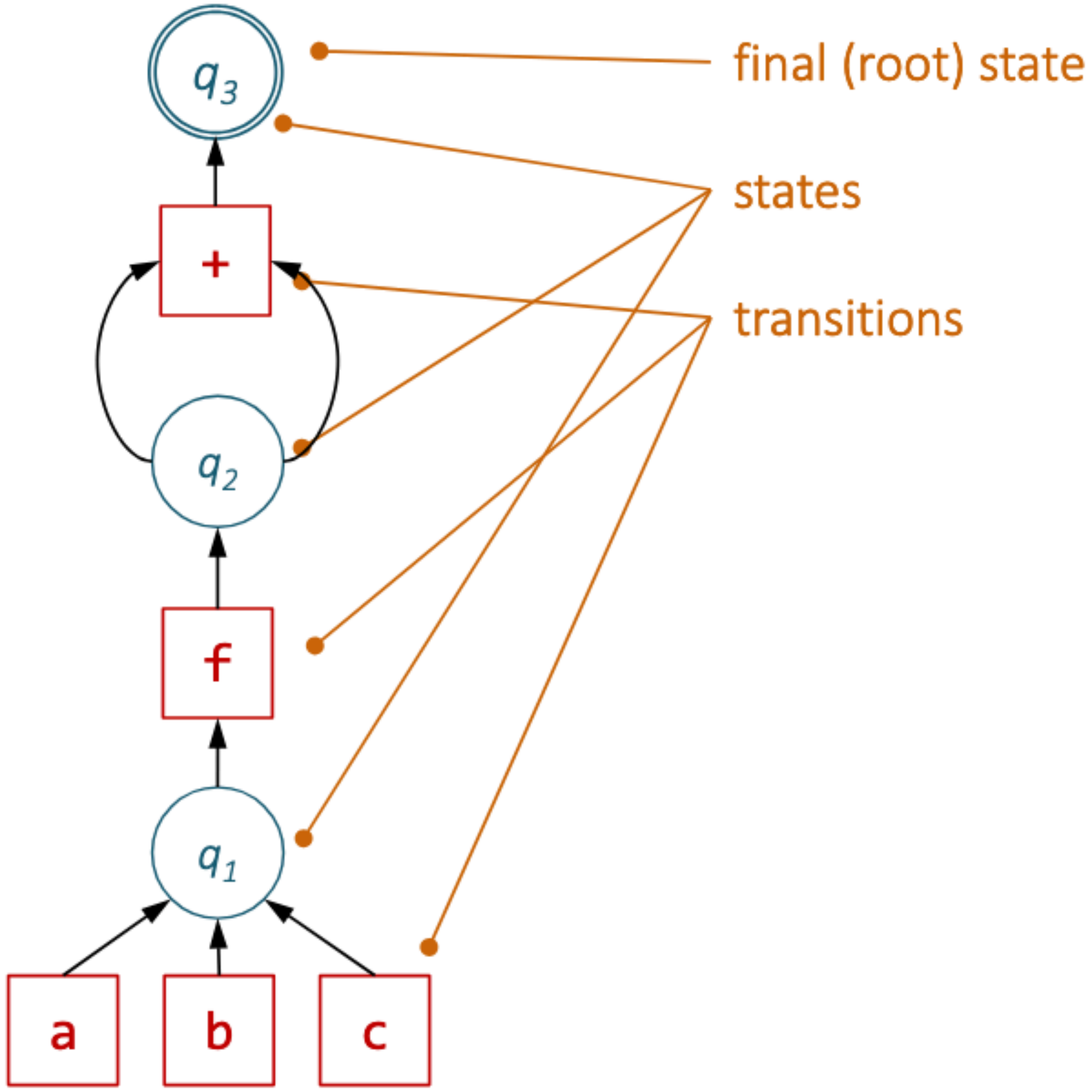
Finite Tree Automata

states — final state(s)

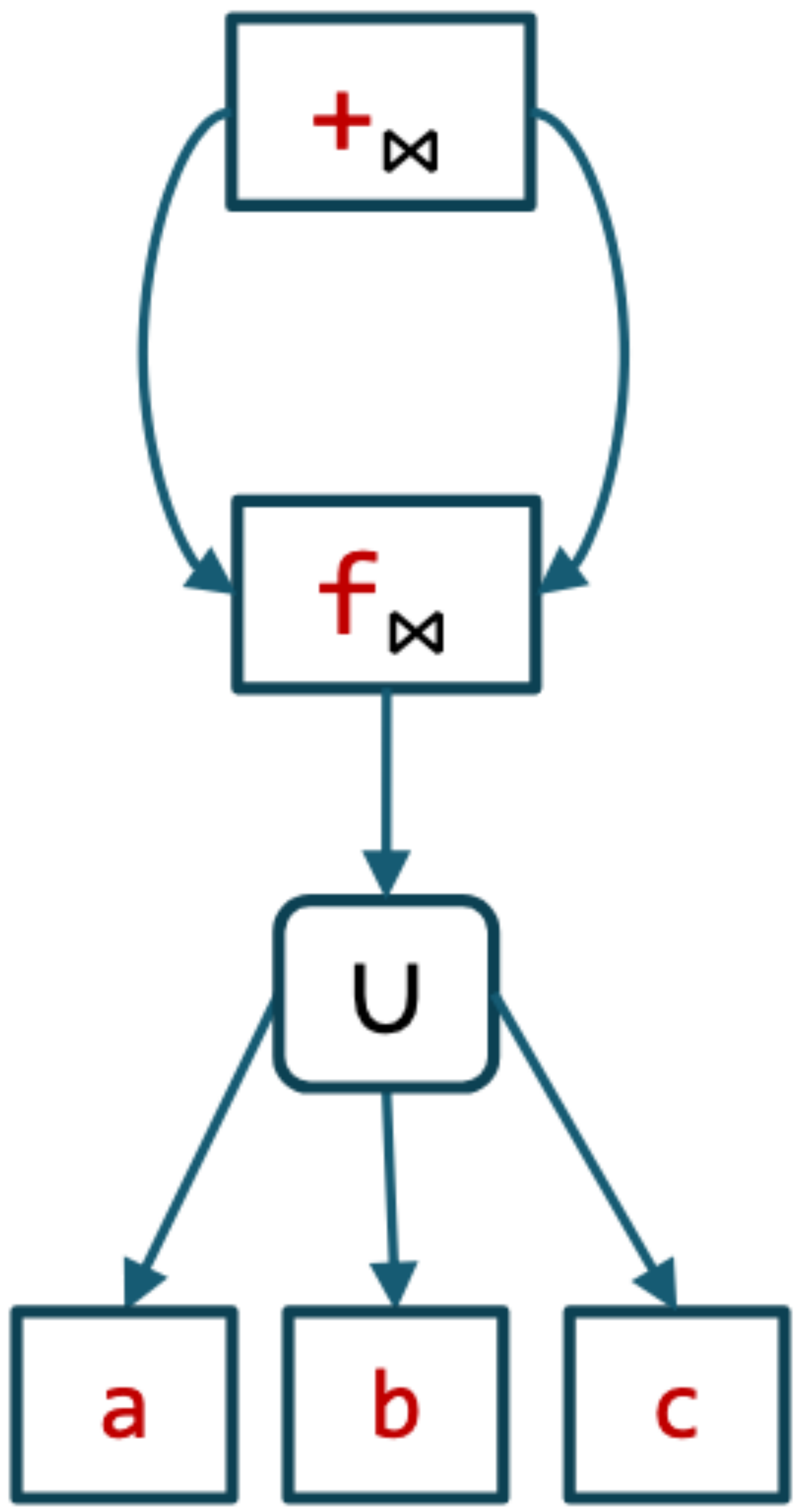
$$\mathcal{A} = \langle Q, F, Q_f, \Delta \rangle$$

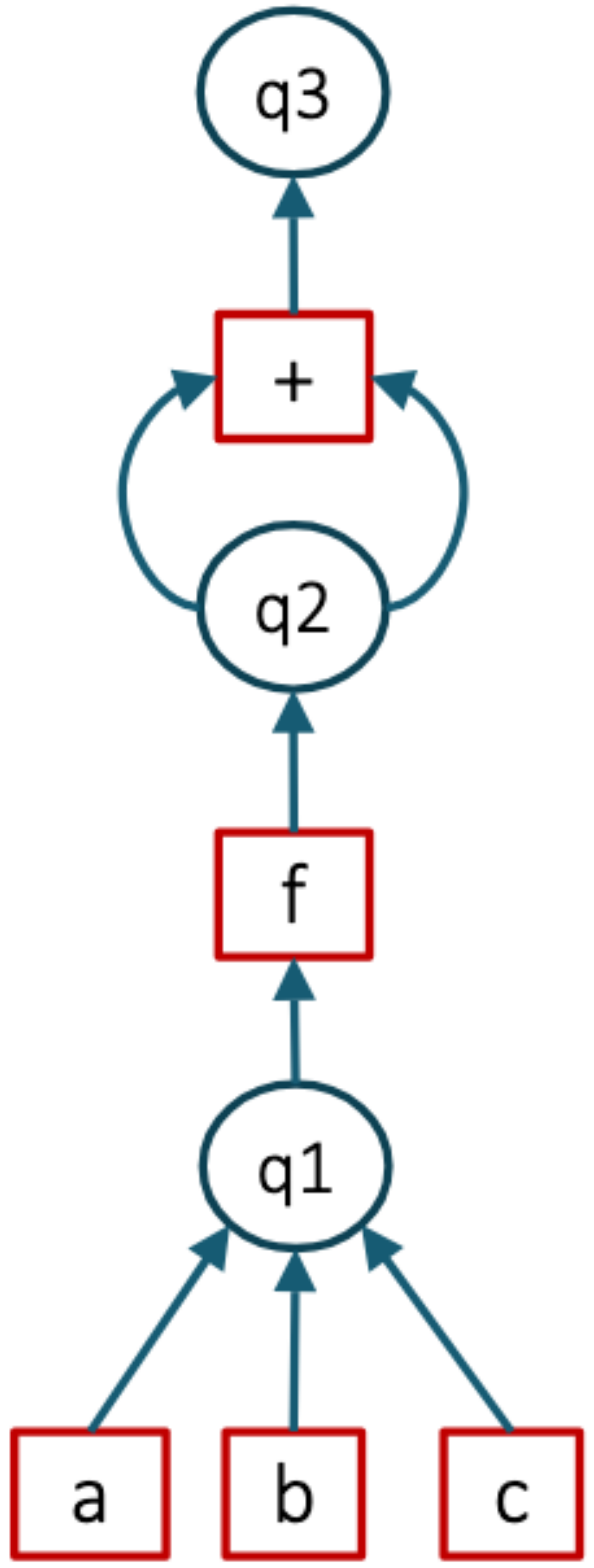
ranked alphabet — transitions

$$f(q_1, \dots, q_n) \rightarrow q$$



VSA vs FTA





Both are and-or graphs

FTA state = VSA union node

- in VSAs singleton unions are omitted

FTA transition = VSA join node

FTA-based search

Synthesis of Data Completion Scripts using Finite Tree Automata
Xinyu Wang, Isil Dillig, Rishabh Singh, *OOPSLA'17*

Program Synthesis using Abstraction Refinement
Xinyu Wang, Isil Dillig, Rishabh Singh, *POPL'18*

Searching Entangled Program Spaces
James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, Nadia Polikarpova. *ICFP'22*

FTA-based search

Synthesis of Data Completion Scripts using Finite Tree Automata
Xinyu Wang, Isil Dillig, Rishabh Singh, *OOPSLA'17*

Program Synthesis using Abstraction Refinement
Xinyu Wang, Isil Dillig, Rishabh Singh, *POPL'18*

Searching Entangled Program Spaces
James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, Nadia Polikarpova. *ICFP'22*

Example

Grammar

```
N ::= id(V) | N + T | N * T
T ::= 2 | 3
V ::= x
```

Spec

1 → 9

PBE with Finite Tree Automata

$\langle A, \mathbb{Z} \rangle$

$A \in \{N, T, X\}$

$\{\langle N, 9 \rangle\}$

states

final states

$$\mathcal{A} = \langle Q, F, Q_f, \Delta \rangle$$

alphabet

transitions

$f(q_1, \ldots, q_n) \rightarrow q$

```
id, +, *
```

```
+(<N,1>,<T,2>) → <N,3>
...
```

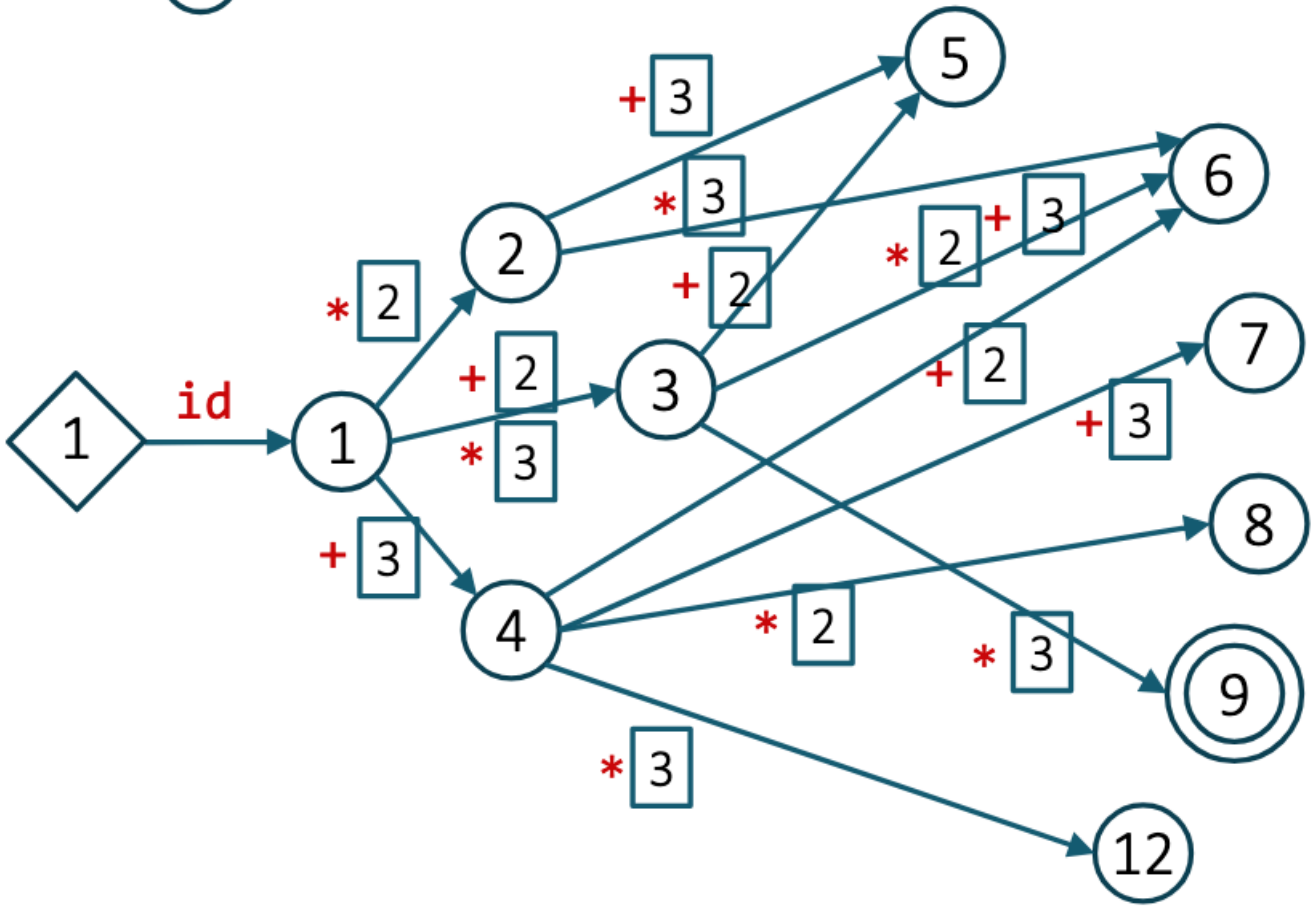
PBE with Finite Tree Automata

N ::= id(V) | N + T | N * T ○

T ::= 2 | 3 □

V ::= x ◇

1 → 9



Discussion

What do FTAs remind you of in the enumerative world?

- FTA ~ bottom-up search with OE

How are they different?

- More size-efficient: sub-terms in the bank are replicated, while in the FTA they are shared
- Hence, can store all terms, not just one representative per class
- Can construct one FTA per example and intersect
- More incremental in the CEGIS context!

FTA-based search

Synthesis of Data Completion Scripts using Finite Tree Automata
Xinyu Wang, Isil Dillig, Rishabh Singh, *OOPSLA'17*

Program Synthesis using Abstraction Refinement
Xinyu Wang, Isil Dillig, Rishabh Singh, *POPL'18*

Searching Entangled Program Spaces
James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, Nadia Polikarpova. *ICFP'22*

Abstract FTA

Challenge: FTA still has too many states

Idea:

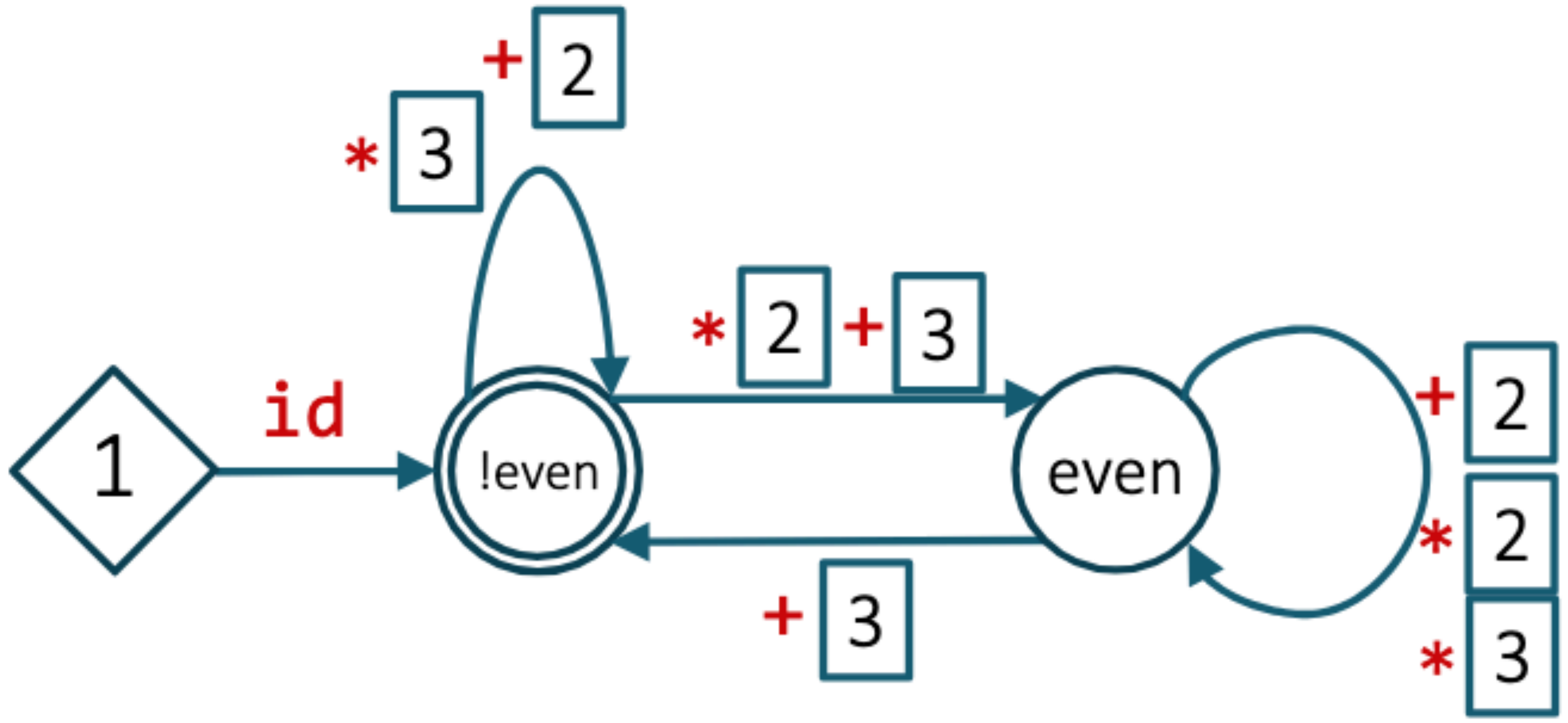
- instead of one state = one value
- we can do one state = set of values (= abstract value)

Abstract FTA

[Wang, Dillig, Singh POPL'18]

```
N ::= id(V) | N + T | N * T  ○
T ::= 2 | 3  □
V ::= x  ◇
```

1 → 9

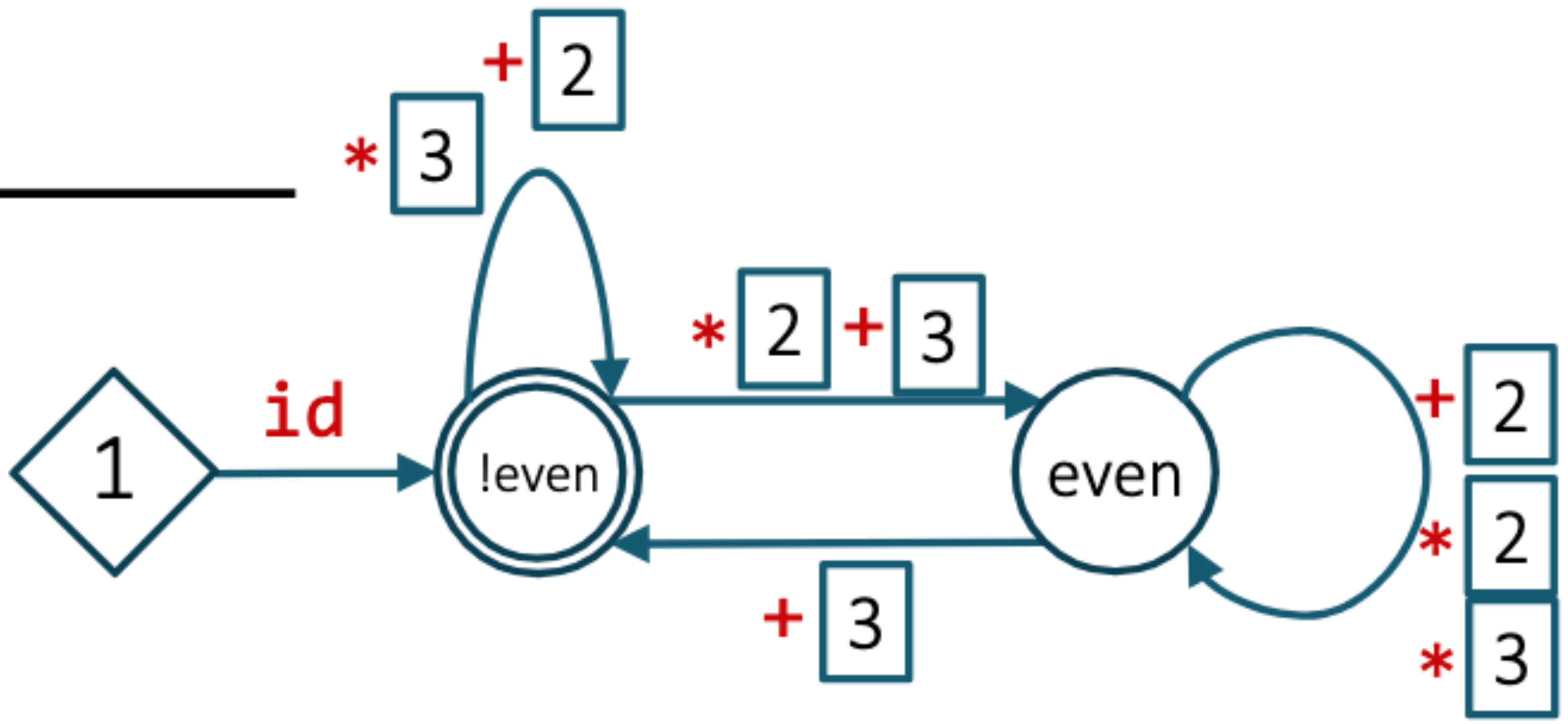


What now?

- idea 1: enumerate from reduced space
- idea 2: refine abstraction!

Abstract FTA

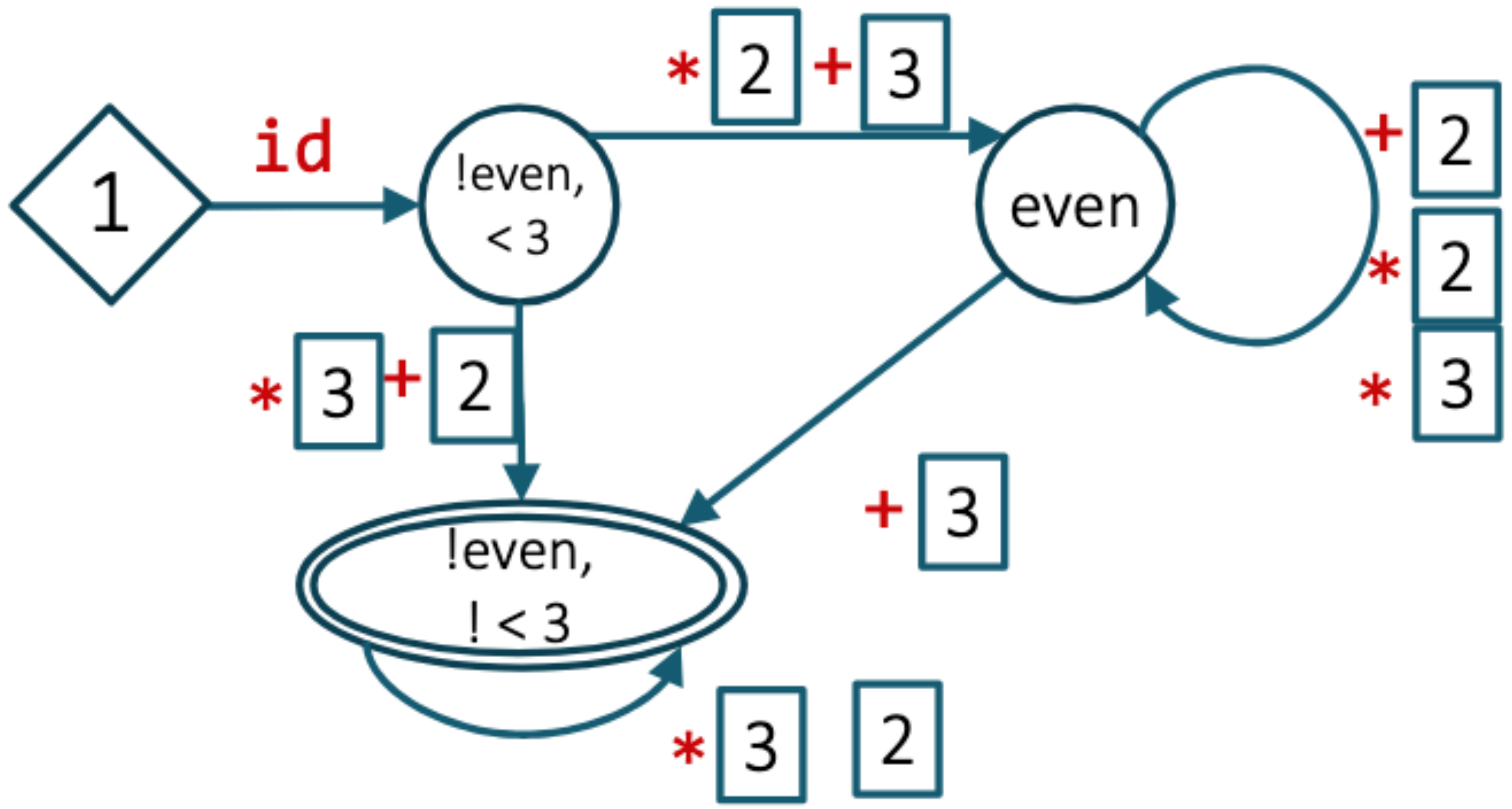
```
N ::= id(V) | N + T | N * T
T ::= 2 | 3
V ::= x
```



solution: `id(x)`

1 → 9

Predicates: {even, < 3, ...}



solution: `id(x)*3`