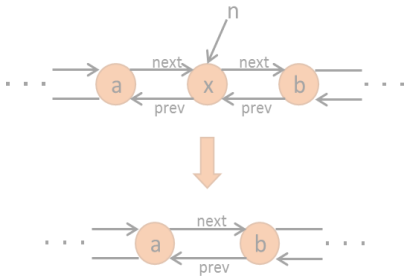
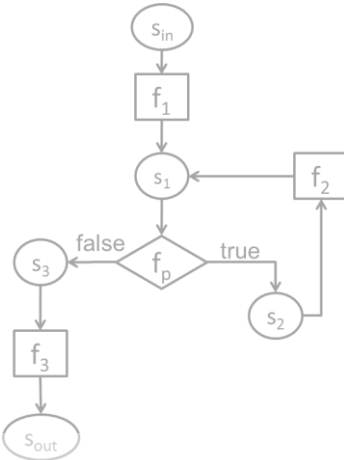


$$\exists c \forall in \ Q(c, in)$$

```

/* Average of x and y without using x+y (avoid overflow)*/
int avg(int x, int y){
    int t = expr({x/2, y/2, x%2, y%2, 2 }, {PLUS, DIV});
    assert t == (x+y)/2;
    return t;
}

```

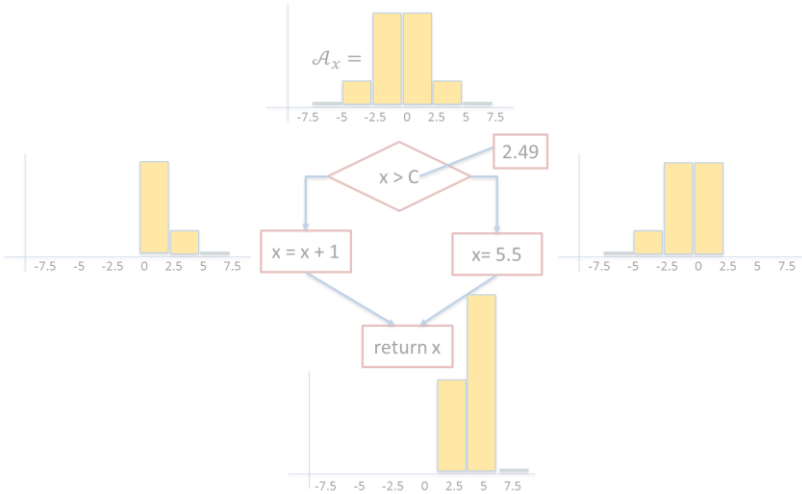
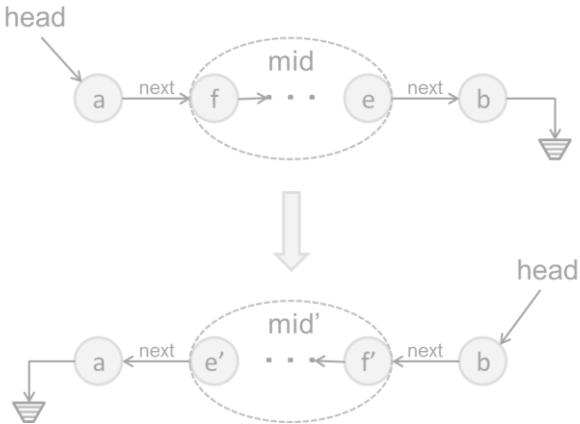


```

{
    s = n.succ;
    p = n.pred;
    p.succ = s;
    s.pred = p;
}

```

# Module I: Synthesizing Simple Programs



$$\varphi(p)$$

$$Sk[c](in)$$

# Lecture 2

## Syntax-Guided Synthesis and Enumerative Search

*Nadia Polikarpova*

# Logistics

---

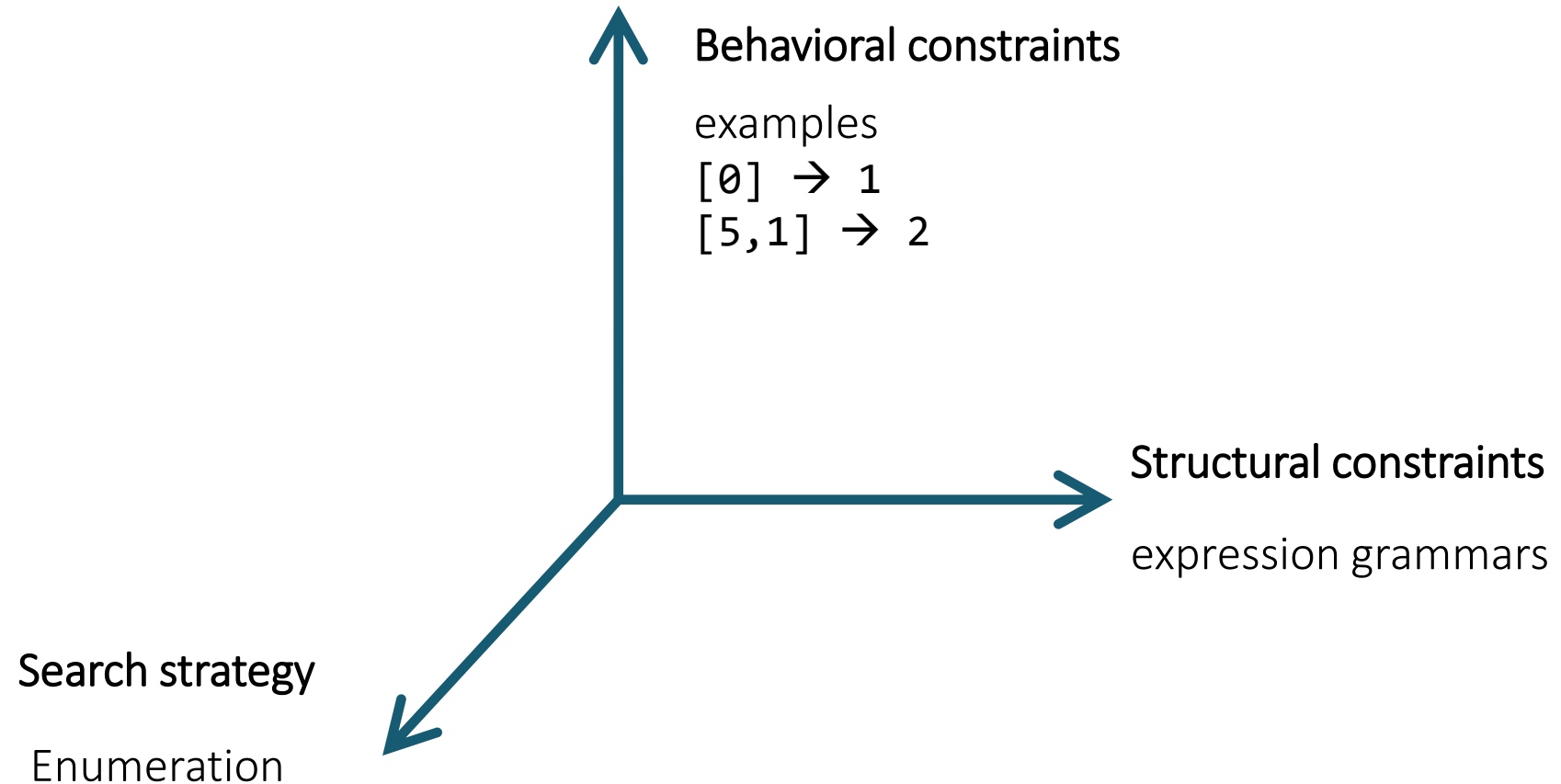
Shared Google folder

- Does everyone have access?
- Register your team by next Friday

Other questions?

# Week 1-2

---



# Today

---

Synthesis from examples

Syntax-guided synthesis

- expression grammars as structural constraints
- the SyGuS project

Enumerative search

- enumerating all programs generated by a grammar
- bottom-up vs top-down

**Synthesis from examples**

# Synthesis from Examples

---

=

Programming by Example

=

Inductive Programming

Inductive Learning

# A little bit of history: inductive learning

---

MIT/LCS/TR-76

LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

Patrick H. Winston

September 1970



Patrick  
Winston

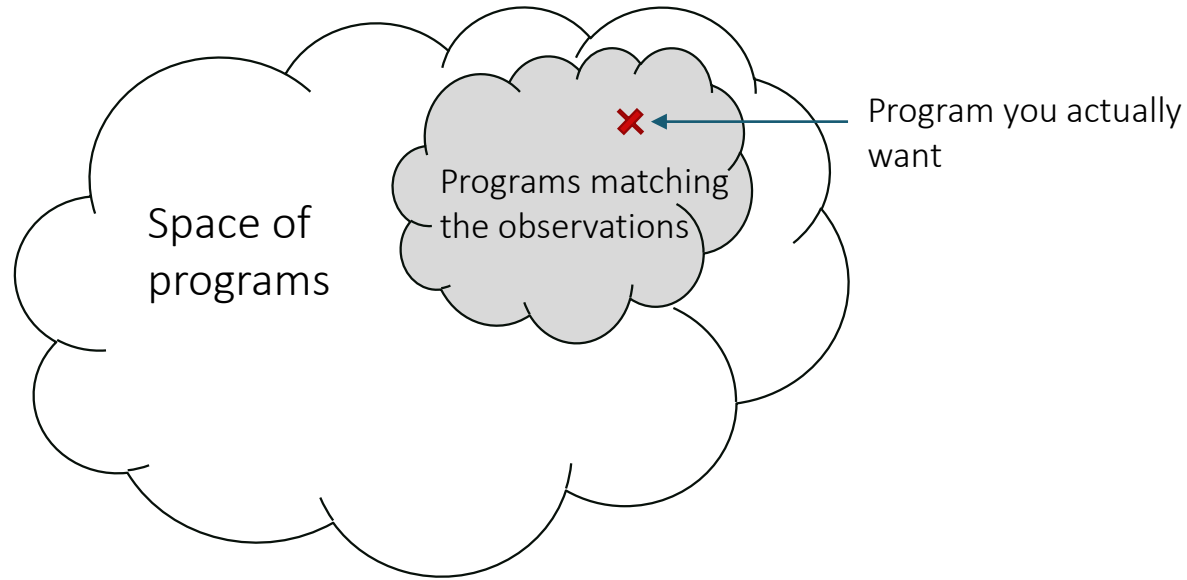
Explored the question of generalizing from a set of observations

Became the foundation of machine learning



# Key issues in inductive learning

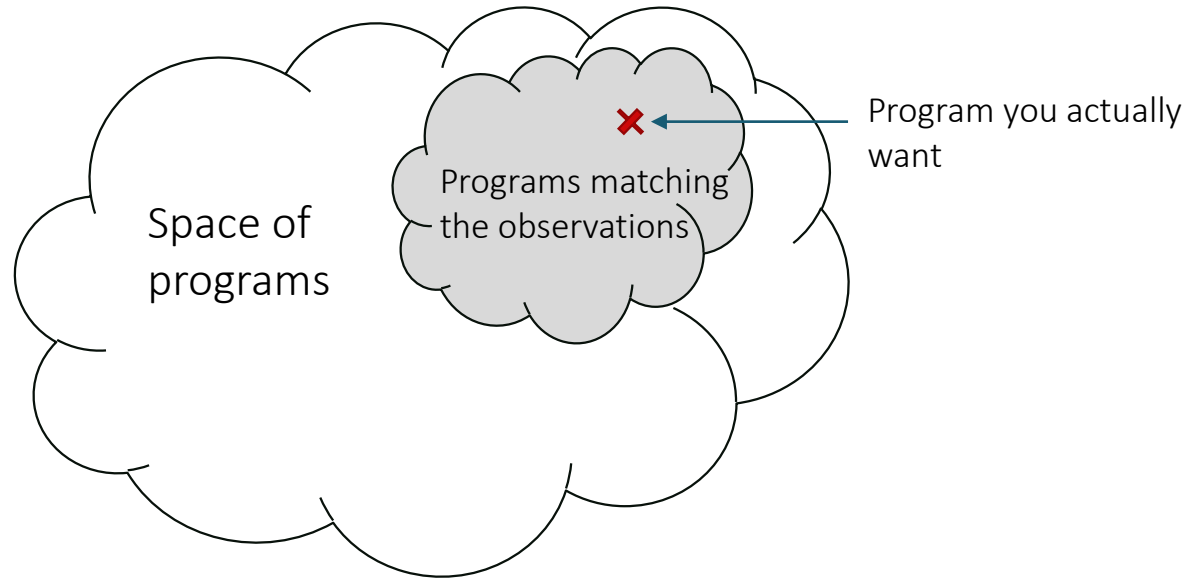
---



- (1) How do you find a program that matches the observations?
- (2) How do you know it is the program you are looking for?

# Key issues in inductive learning

---



Traditional ML:

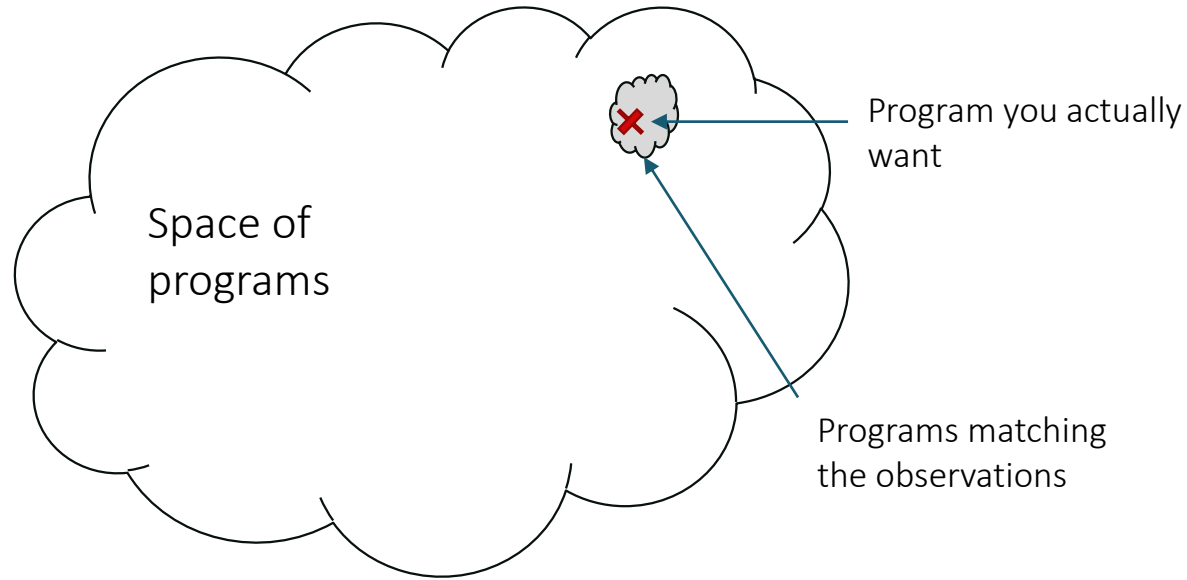
- Fix the space so that (1) is easy
- (2) becomes the main challenge

(1) How do you find a program that matches the observations?

(2) How do you know it is the program you are looking for?

# The synthesis approach

---



Program synthesis:

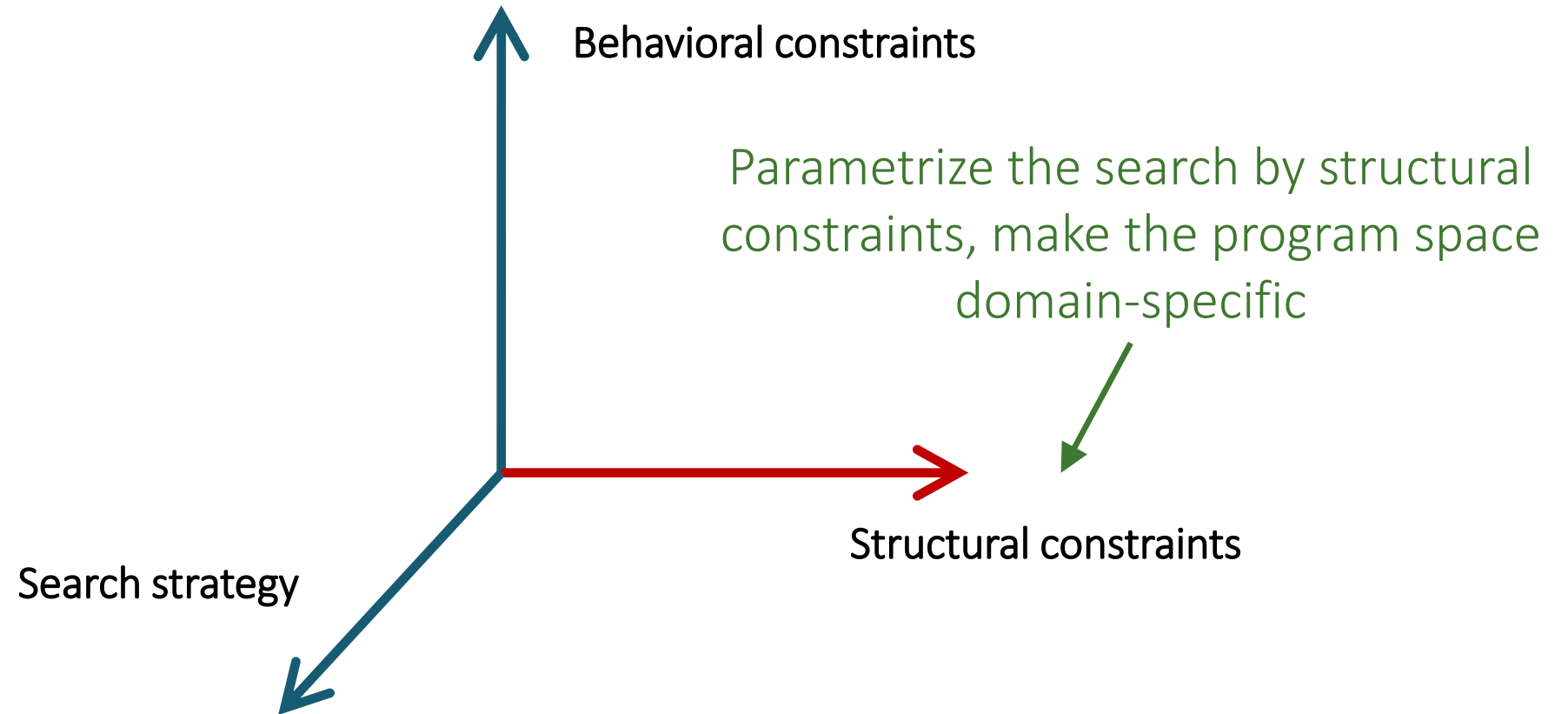
- Customize the space so that (2) becomes easier
- (1) is now the main challenge

(1) How do you find a program that matches the observations?

(2) How do you know it is the program you are looking for?

# Key idea

---



# Syntax-Guided Synthesis

# Example

---

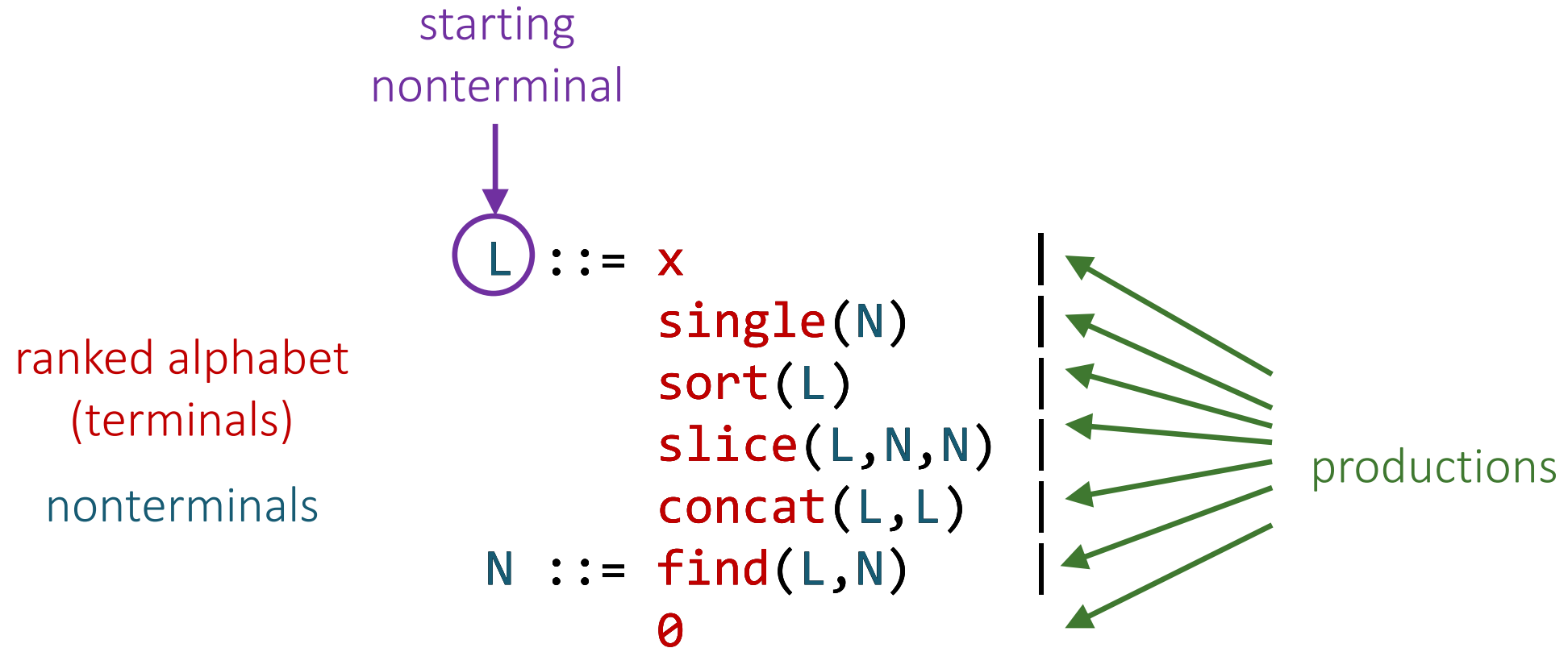
$[1,4,7,2,0,6,9,2,5,0] \rightarrow [1,2,4,7,0]$

L ::= x		the input
single(N)		single(1) = [1]
sort(L)		sort([6,9,2,5]) = [2,5,6,9]
slice(L,N,N)		slice([6,9,2,5],0,2) = [6,9]
concat(L,L)		concat([6,9],[2,5]) = [6,9,2,5]
N ::= find(L,N)		find([6,9],9) = 1
0		0

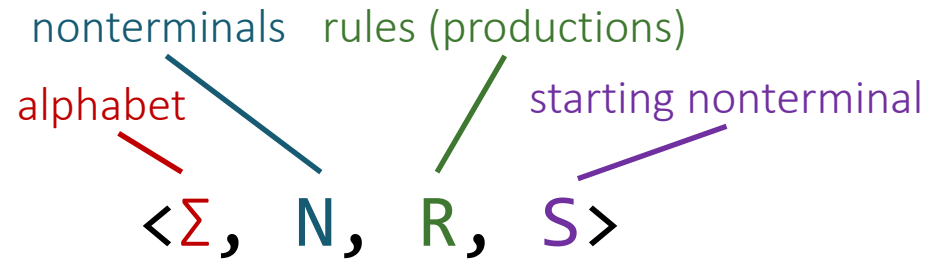
$f(x) := \text{concat}(\text{sort}(\text{slice}(x,0,\text{find}(x,0))), \text{single}(0))$

# Regular tree grammars (RTGs)

---



# Regular tree grammars (CFGs)



Terms:  $t \in T_{\Sigma}(N)$  = all terms made from  $N \cup \Sigma$

Rules are of the form:  $A \rightarrow \sigma(A_1, \dots, A_n)$

Rewrites to:  $\mathcal{C}[A] \rightarrow \mathcal{C}[t]$  if  $(A \rightarrow t) \in R$

(Incomplete) programs:  $\{t \in T_{\Sigma}(N) \mid A \rightarrow^* t\}$

Ground programs:  $\{t \in T_{\Sigma} \mid A \rightarrow^* t\}$   
= programs without holes, complete programs

Whole programs:  $\{t \in T_{\Sigma} \mid S \rightarrow^* t\}$   
= roughly, programs of the right type

`concat(L,  $\emptyset$ )`

`L  $\rightarrow$  concat(L, L)`

`concat(L, L)  $\rightarrow$  concat(x, L)`

`find(concat(L, L), N)`

`find(concat(x, x),  $\emptyset$ )`

`sort(concat(L, L))`



# RTGs as structural constraints

---

Space of programs  
= the *language* of an RTG  
= all **ground**, **whole** programs

$\textcircled{L} ::= x$   
 $\text{single}(N)$   
 $\text{sort}(L)$   
 $\text{slice}(L, N, N)$   
 $\text{concat}(L, L)$   
 $N ::= \text{find}(L, N)$   
 $\emptyset$



$x$     $\text{sort}(x)$     $\text{concat}(x, x)$     $\text{slice}(x, \emptyset, \emptyset)$   
...  
 $\text{slice}(x, \emptyset, \text{find}(x, \emptyset))$   
...  
 $\text{concat}(\text{sort}(\text{slice}(x, \emptyset, \text{find}(x, \emptyset))), \text{single}(\emptyset))$   
...

# How big is the space?

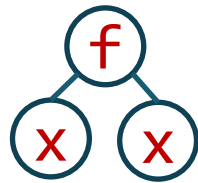
$$E ::= x \mid f(E, E)$$

depth  $\leq 0$



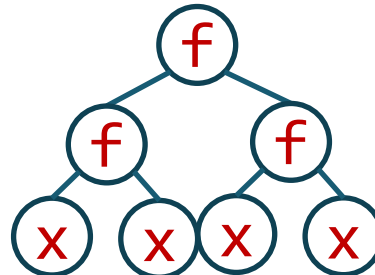
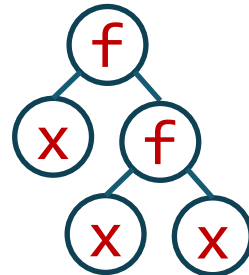
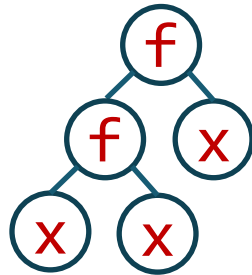
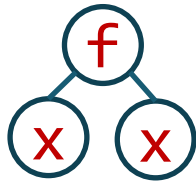
$$N(0) = 1$$

depth  $\leq 1$



$$N(1) = 2$$

depth  $\leq 2$



$$N(2) = 5$$

$$N(d) = 1 + N(d - 1)^2$$

# How big is the space?

---

$E ::= x \mid f(E, E)$

$$N(d) = 1 + N(d - 1)^2$$

$$N(d) \sim c^{2^d} \quad (c > 1)$$

$$N(1) = 1$$

$$N(2) = 2$$

$$N(3) = 5$$

$$N(4) = 26$$

$$N(5) = 677$$

$$N(6) = 458330$$

$$N(7) = 210066388901$$

$$N(8) = 44127887745906175987802$$

$$N(9) = 1947270476915296449559703445493848930452791205$$

$$N(10) = 3791862310265926082868235028027893277370233152247388584761734150717768254410341175325352026$$

# How big is the space?

---

$$E ::= \begin{array}{c} x_1 \mid \dots \mid x_k \\ f_1(E, E) \mid \dots \mid f_m(E, E) \end{array}$$

$$N(\emptyset) = k$$

$$N(d) = k + m * N(d - 1)^2$$

$$N(1) = 3$$

$$N(2) = 30$$

$$N(3) = 2703$$

$$N(4) = 21918630$$

$$N(5) = 1441279023230703$$

$$N(6) = 6231855668414547953818685622630$$

$$N(7) = 116508075215851596766492219468227024724121520304443212304350703$$

$$k = m = 3$$

# Syntactic sugar

---

Instead of this:

```
L ::= x |
      single(N) |
      sort(L) |
      slice(L,N,N) |
      concat(L,L) |
N ::= find(L,N) |
      0
```

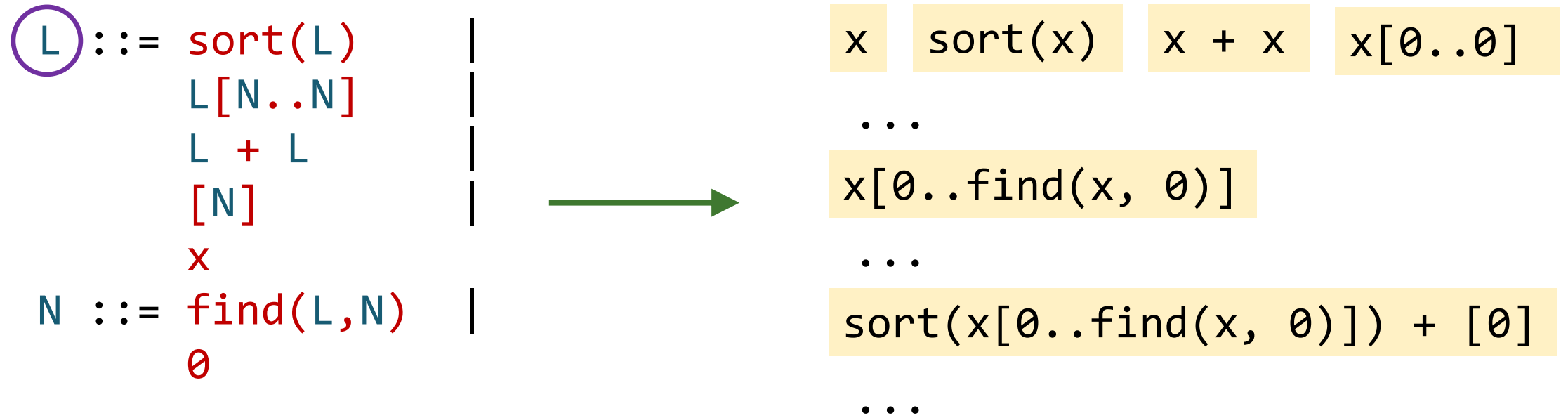
We will often write this:

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

- allow custom syntax for terminal symbols
- not an RTG strictly speaking, but you know what we mean...

# Syntactic sugar

---



# The SyGuS project

---

[Alur et al. 2013]

<https://sygus.org/>

Goal: Unify different syntax-guided approaches

Collection of synthesis benchmarks + yearly competition

- 6 competitions since 2013
- consider writing a SyGuS solver for your project!

Common input format + supporting tools

- parser, baseline synthesizers

# SyGuS problems

---

SyGuS problem =  $\langle$  theory, spec, grammar  $\rangle$

A “library” of types and function symbols

**Example:** Linear Integer Arithmetic (LIA)

True, False

0, 1, 2, ...

$\wedge$ ,  $\vee$ ,  $\neg$ ,  $+$ ,  $\leq$ , ite

RTG with terminals in the theory  
(+ input variables)

**Example:** Conditional LIA  
expressions w/o sums

$E ::= x \mid \text{ite } C \ E \ E$

$C ::= E \leq E \mid C \wedge C \mid \neg C$



# SyGuS problems

---

SyGuS problem =  $\langle \text{theory, spec, grammar} \rangle$

A first-order logic formula over  
the theory

Examples:

$$f(0, 1) = 1 \wedge$$

$$f(1, 0) = 1 \wedge$$

$$f(1, 1) = 1 \wedge$$

$$f(2, 0) = 2$$

# SyGuS demo

---

# SyGuS problems

---

SyGuS problem =  $\langle \text{theory, spec, grammar} \rangle$

A first-order logic formula over  
the theory

Examples:

$f(0, 1) = 1 \wedge$   
 $f(1, 0) = 1 \wedge$   
 $f(1, 1) = 1 \wedge$   
 $f(2, 0) = 2$

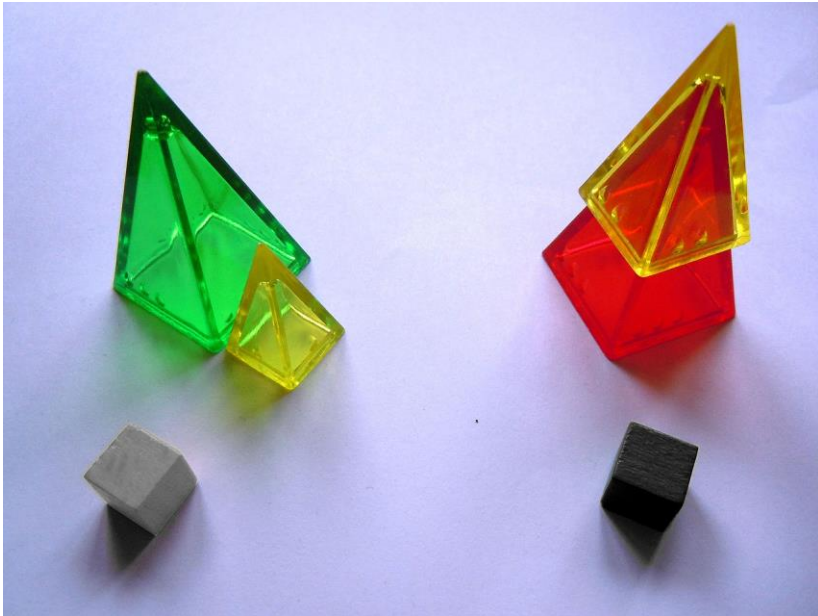
Formula with free variables:

$x \leq f(x, y) \wedge$   
 $y \leq f(x, y) \wedge$   
 $(f(x, y) = x \vee f(x, y) = y)$

can inductive synthesis  
handle these?

# The Zendo game

---



The **teacher** makes up a secret rule

- e.g. all pieces must be grounded

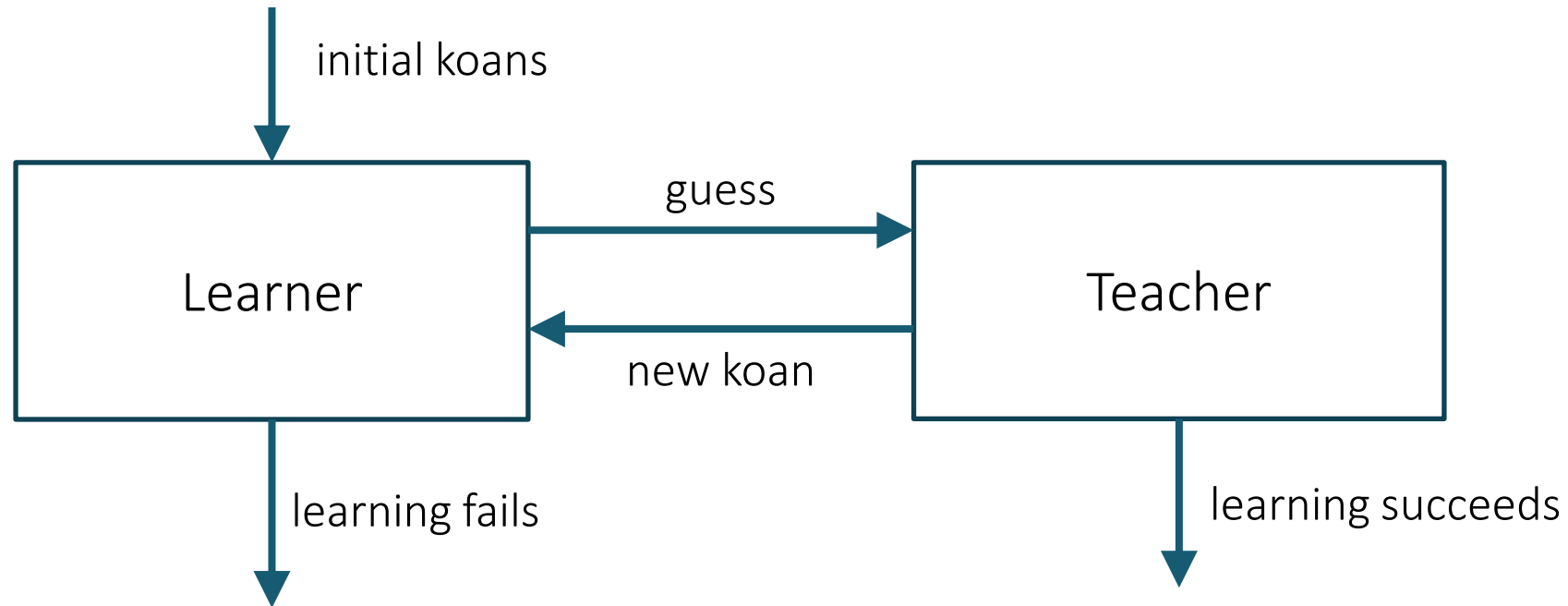
The teacher builds two **koans** (a positive and a negative)

A **student** can try to guess the rule

- if they are right, they win
- otherwise, the teacher builds a koan on which the two rules disagree

# The Zendo game

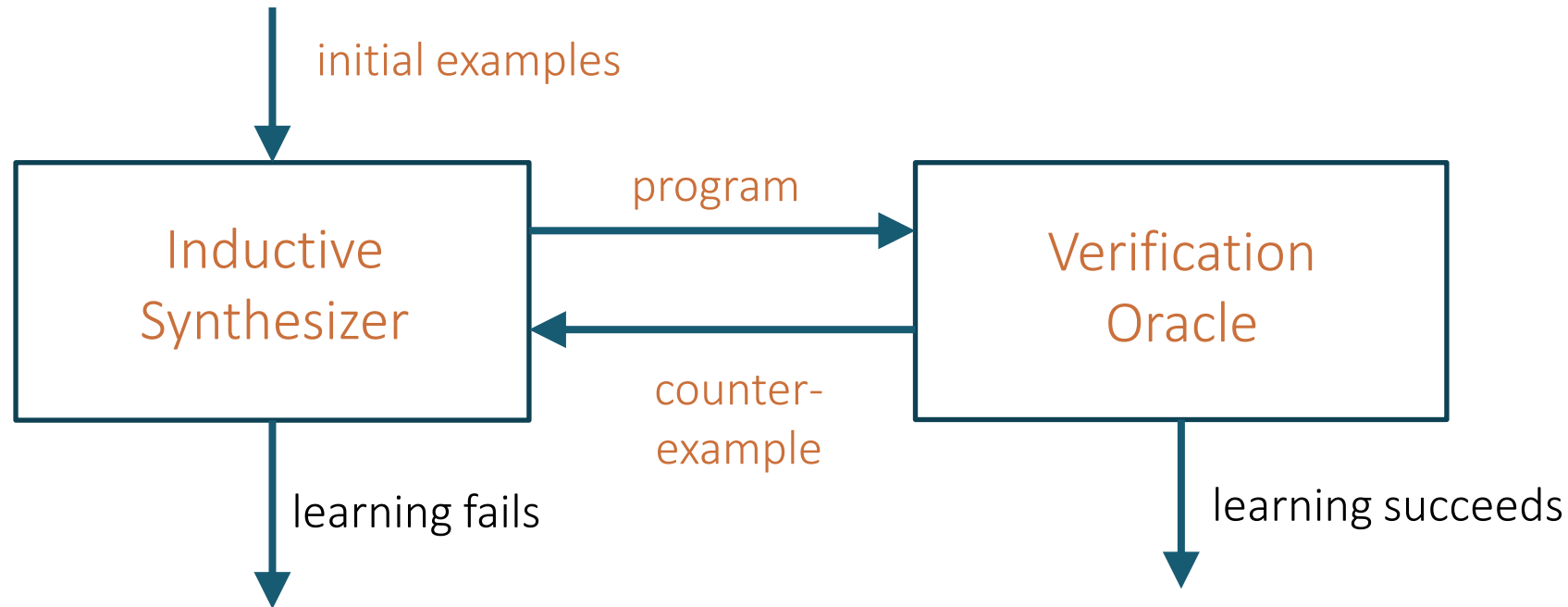
---



# Counter-example guided inductive synthesis (CEGIS)

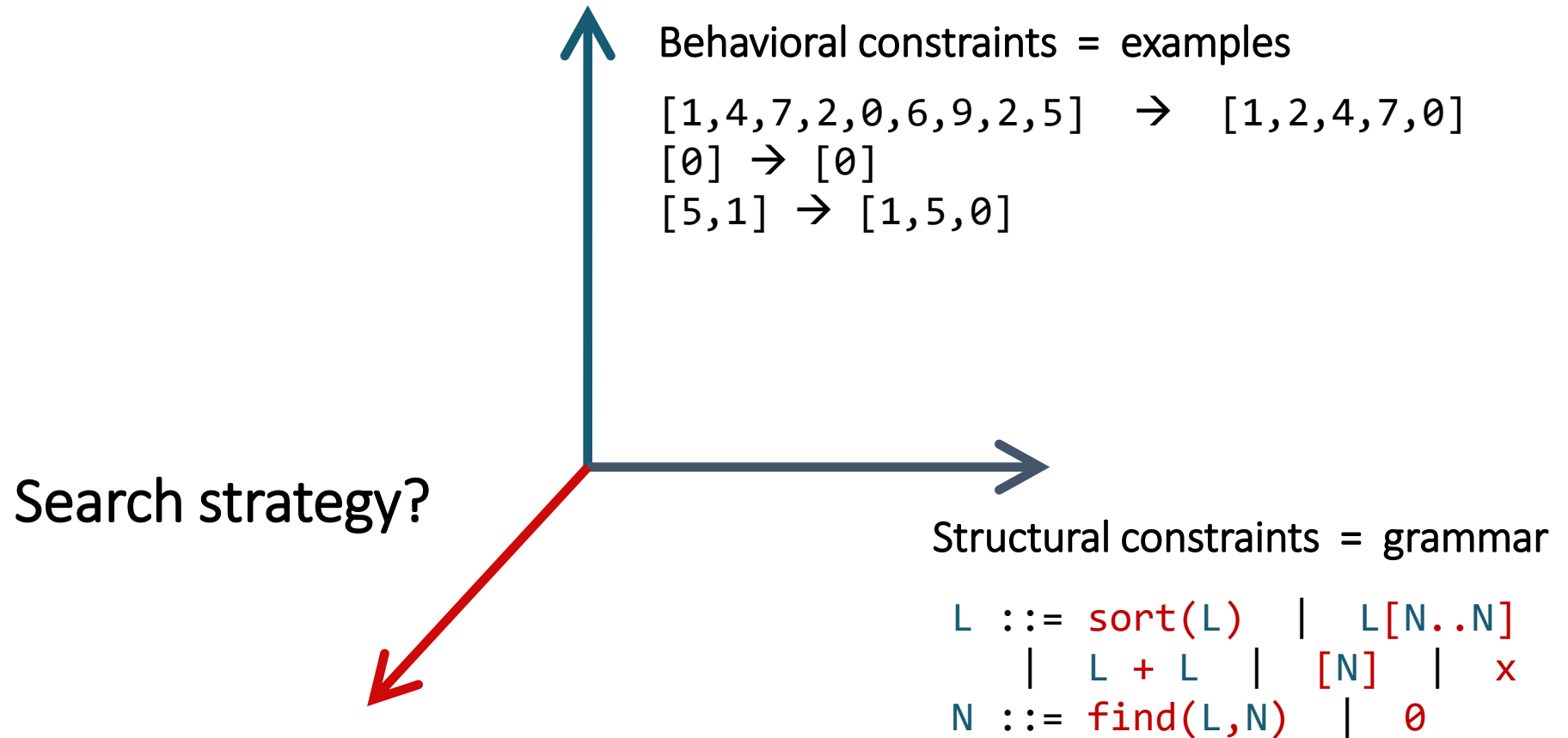
---

The Zendo of program synthesis



# The problem statement

---



# Enumerative search



# Enumerative search

---

=

Explicit / Exhaustive Search

**Idea:** Sample programs from the grammar one by one and test them on the examples

**Challenge:** How do we systematically enumerate all programs?

top-down vs bottom-up

# Top-down enumeration

---

Maintain a **worklist** of whole programs

Initialize with the start non-terminal

Expand remaining non-terminals using productions

$L ::= L[N..N] \quad |$

$N ::= \overset{x}{\text{find}}(L, N) \quad |$   
 $\quad \quad \quad \emptyset$

$[[1, 4, \emptyset, 6] \rightarrow [1, 4]]$

# Top-down enumeration

nonterminals rules (productions)  
alphabet starting nonterminal

```
top-down(<  $\Sigma$ , N, R, S >, [i  $\rightarrow$  o]):  
  w1 := [S]  
  while (w1 != []):  
    p := w1.dequeue()  
    if (ground(p)  $\wedge$  p([i]) = [o]):  
      return p  
    w1.enqueue(unroll(p))  
  
unroll(p):  
  w1' := []  
  A := left-most non-term in p  
  forall (A  $\rightarrow$  rhs) in R:  
    w1' += p[A  $\rightarrow$  rhs]  
  return w1';
```

```
L ::= L[N..N] |  
      x  
N ::= find(L,N) |  
      0  
  
[[1,4,0,6]  $\rightarrow$  [1,4]]
```

# Top-down: example

Worklist w1

iter 0: L

iter 1: x<sup>✗</sup> L[N..N]

iter 2: L[N..N]

iter 3: x[N..N] L[N..N][N..N]

iter 4: x[0..N] x[find(L,N)..N] L[N..N][N..N]

iter 5: x[0..0]<sup>✗</sup> x[0.. find(L,N)] x[find(L,N)..N] ...

iter 6: x[0.. find(L,N)] x[find(L,N)..N] ...

iter 7: x[0.. find(x,N)] x[0.. find(L[N..N],N)] ...

iter 8: x[0.. find(x,0)]<sup>✓</sup> x[0.. find(x,find(L,N))] ...

iter 9:

L ::= L[N..N] | ←

x

N ::= find(L,N) | ←

0 ←

[ [1,4,0,6] → [1,4] ]

# Bottom-up enumeration

---

Maintain a **bank** of ground programs

- indexed by non-terminal and depth

Combine programs in the bank into larger programs using productions

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

$[[1,4,0,6] \rightarrow [1,4]]$

# Bottom-up enumeration

nonterminals   rules (productions)  
alphabet   starting nonterminal  
 bottom-up ( $\langle \Sigma, N, R, S \rangle, [i \rightarrow o]$ ):  
   bank[A,d] := {} forall A, d  
   for d in [0..]:  
     forall (A  $\rightarrow$  rhs) in R:  
       forall p in new-terms(A $\rightarrow$ rhs, d, bank):  
         if (A = S  $\wedge$  p([i]) = [o]):  
           return p  
         bank[A,d] += p;

new-terms(A  $\rightarrow \sigma(A_1 \dots A_n)$ , d, bank):

if (d = 0  $\wedge$  n = 0) yield  $\sigma$   
 else forall  $\langle d_1, \dots, d_n \rangle$  in  $[0..d-1]^n$  s.t.  $\max(d_1, \dots, d_n) = d-1$ :  
   forall  $\langle p_1, \dots, p_n \rangle$  in  $\text{bank}[A_1, d_1] \times \dots \times \text{bank}[A_n, d_n]$ :  
     yield  $\sigma(p_1, \dots, p_n)$

```

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

$[1, 4, 0, 6] \rightarrow [1, 4]$

# Bottom-up: example

Program bank

d = 0:    x    0

d = 1:    sort(x) x + x x[0..0] [0]  
          find(x,0)

d = 2:    sort(sort(x)) sort(x[0..0]) sort(x + x)  
          sort([0]) x + (x + x) x + [0] sort(x) + x  
          x[0..0] + x (x + x) + x [0] + x x + x[0..0]  
          x + sort(x) x[0..find(x,0)] ✓

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

[1,4,0,6] → [1,4]

# Bottom-up vs top-down

---

## Top-down

## Bottom-up

Smaller to larger depth

- Has to explore between  $3 \cdot 10^9$  and  $10^{23}$  programs to find `sort(x[0..find(x, 0)]) + [0]` (depth 6)

Candidates are **whole** but might not be **ground**

- Cannot always run on inputs
- Can always relate to outputs (?)

Candidates are **ground** but might not be **whole**

- Can always run on inputs
- Cannot always relate to outputs

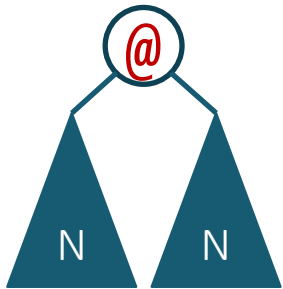


# How to make it scale

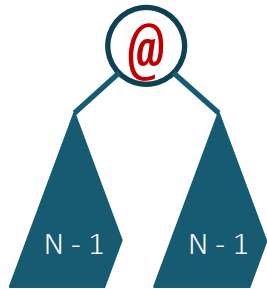
---

## Prune

Discard useless subprograms



$$m * N^2$$



$$m * (N - 1)^2$$

## Prioritize

Explore more promising candidates first

$P = \{ \begin{array}{l} [0][N..N] \\ x[N..N] \\ \dots \end{array} , \dots \}$  ← dequeue this first