

#3: Enumerative Search

Sankha Narayan Guria

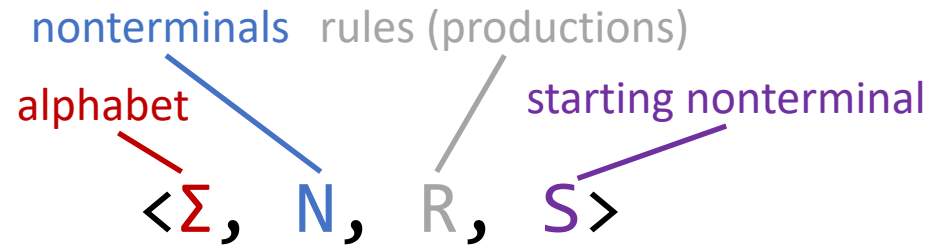
EECS 700: Introduction to Program Synthesis



Logistics

- **Discord**
 - Has everyone signed up?
 - #seeking-teammates channel
 - #paper-discussions channel
- **Paper reading #1 is out!**
- **Other questions?**

Regular tree grammars (RTGs)



- Trees: $\tau \in T_{\Sigma}(N)$ = all trees made from $N \cup \Sigma$
- Rules are of the form: $A \rightarrow \sigma(A_1, \dots, A_n)$
- Derives in one step: $\mathcal{C}[A] \rightarrow \mathcal{C}[t]$ if $(A \rightarrow t) \in R$
 A is the leftmost non-terminal in $\mathcal{C}[A]$
- Incomplete terms/programs: $\{\tau \in T_{\Sigma}(N) \mid A \rightarrow^* \tau\}$
- Complete terms/programs: $\{t \in T_{\Sigma} \mid A \rightarrow^* t\}$
 = programs without holes
- Whole programs: $\{t \in T_{\Sigma} \mid S \rightarrow^* t\}$
 = roughly, programs of the right type

$\text{concat}(L, \emptyset)$

$L \rightarrow \text{concat}(L, L)$

$\text{concat}(L, L) \rightarrow \text{concat}(x, L)$

$\text{find}(\text{concat}(L, L), N)$

$\text{find}(\text{concat}(x, x), \emptyset)$

$\text{sort}(\text{concat}(L, L))$

SyGuS problems

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

A first-order logic formula over
the theory

Examples:

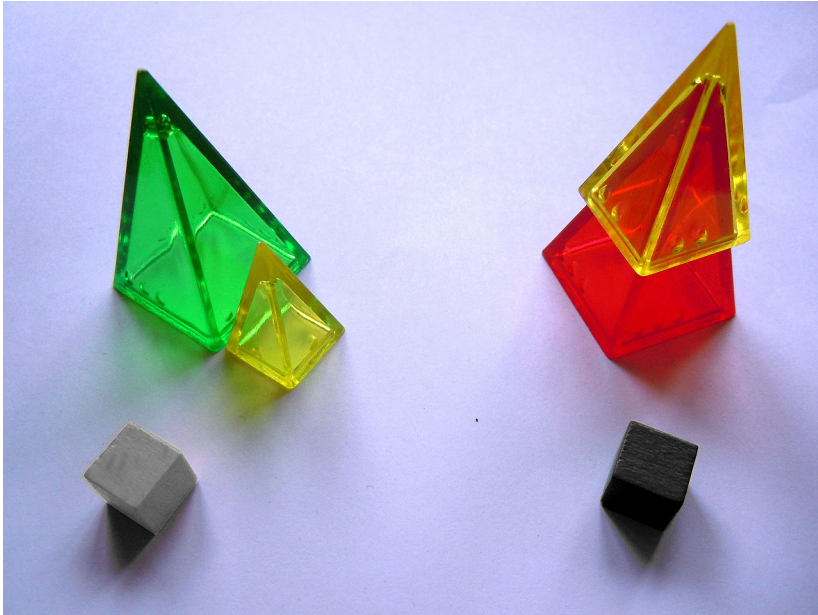
$$\begin{aligned} f(0, 1) &= 1 \wedge \\ f(1, 0) &= 1 \wedge \\ f(1, 1) &= 1 \wedge \\ f(2, 0) &= 2 \end{aligned}$$

Formula with free variables:

$$\begin{aligned} x &\leq f(x, y) \wedge \\ y &\leq f(x, y) \wedge \\ (f(x, y) &= x \vee f(x, y) = y) \end{aligned}$$

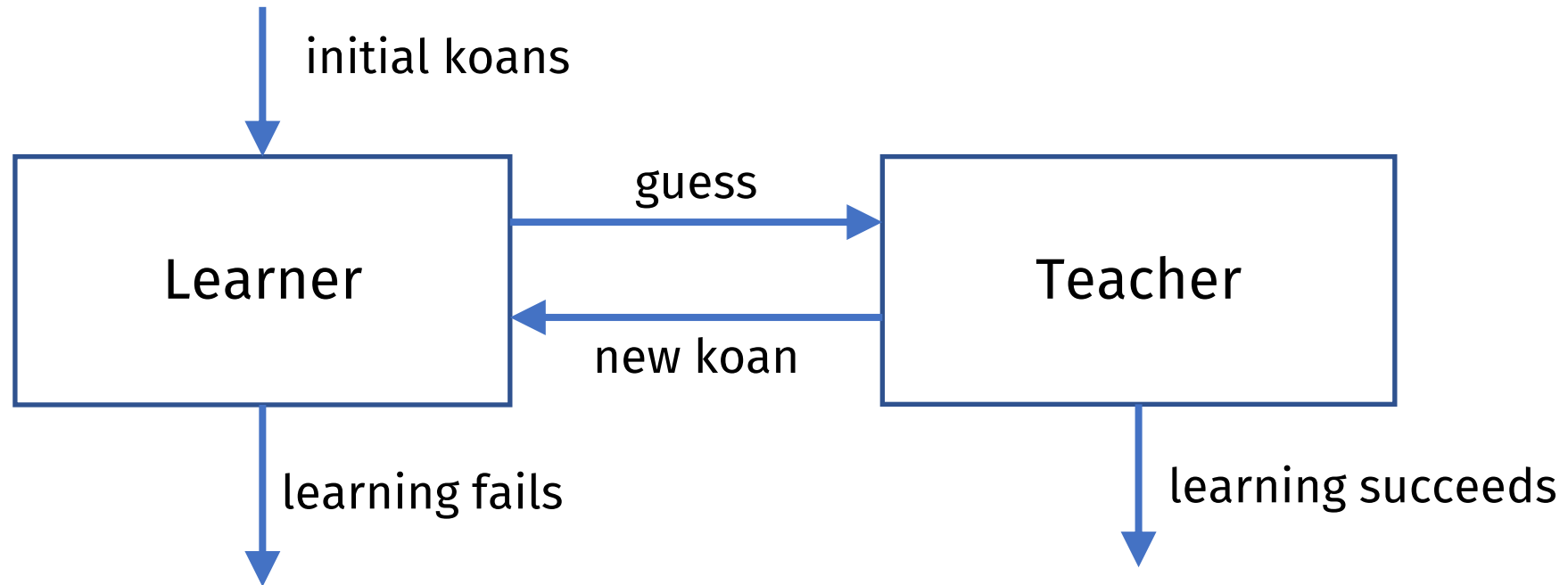
can inductive synthesis
handle these?

The Zendo game



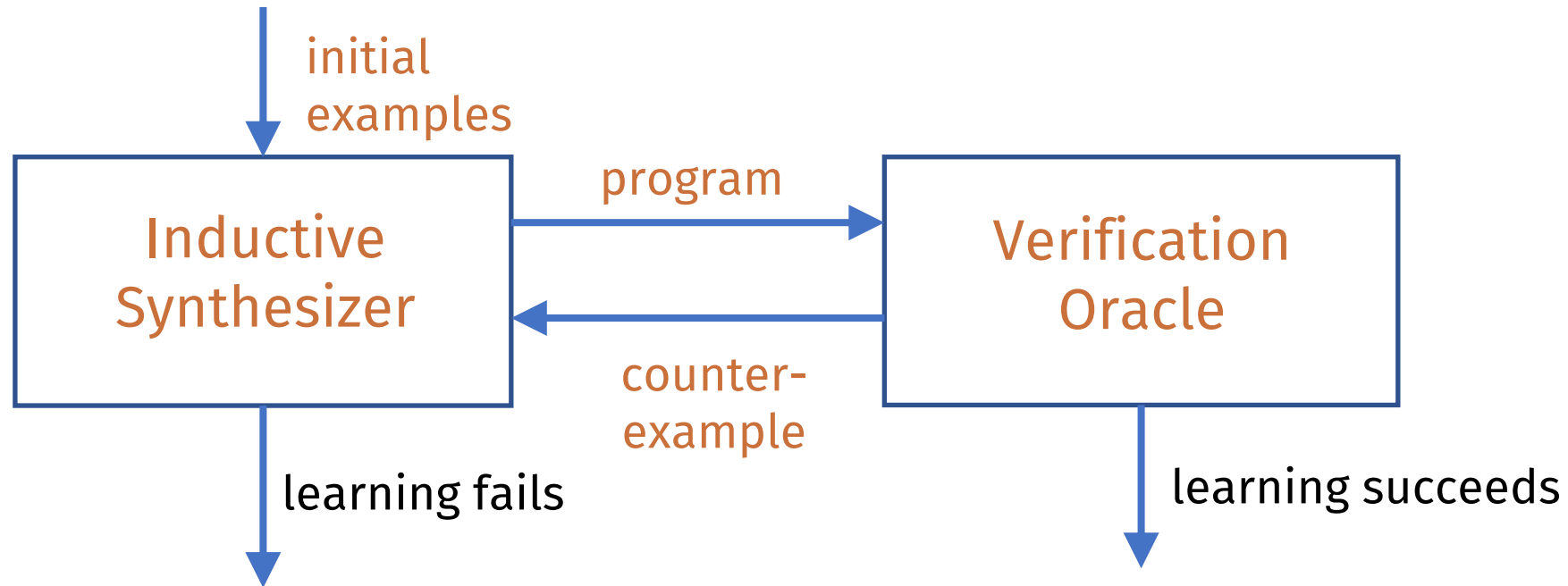
- The **teacher** makes up a secret rule
 - e.g. all pieces must be grounded
- The teacher builds two **koans** (a positive and a negative)
- A **student** can try to guess the rule
 - if they are right, they win
 - otherwise, the teacher builds a koan on which the two rules disagree

The Zendo game

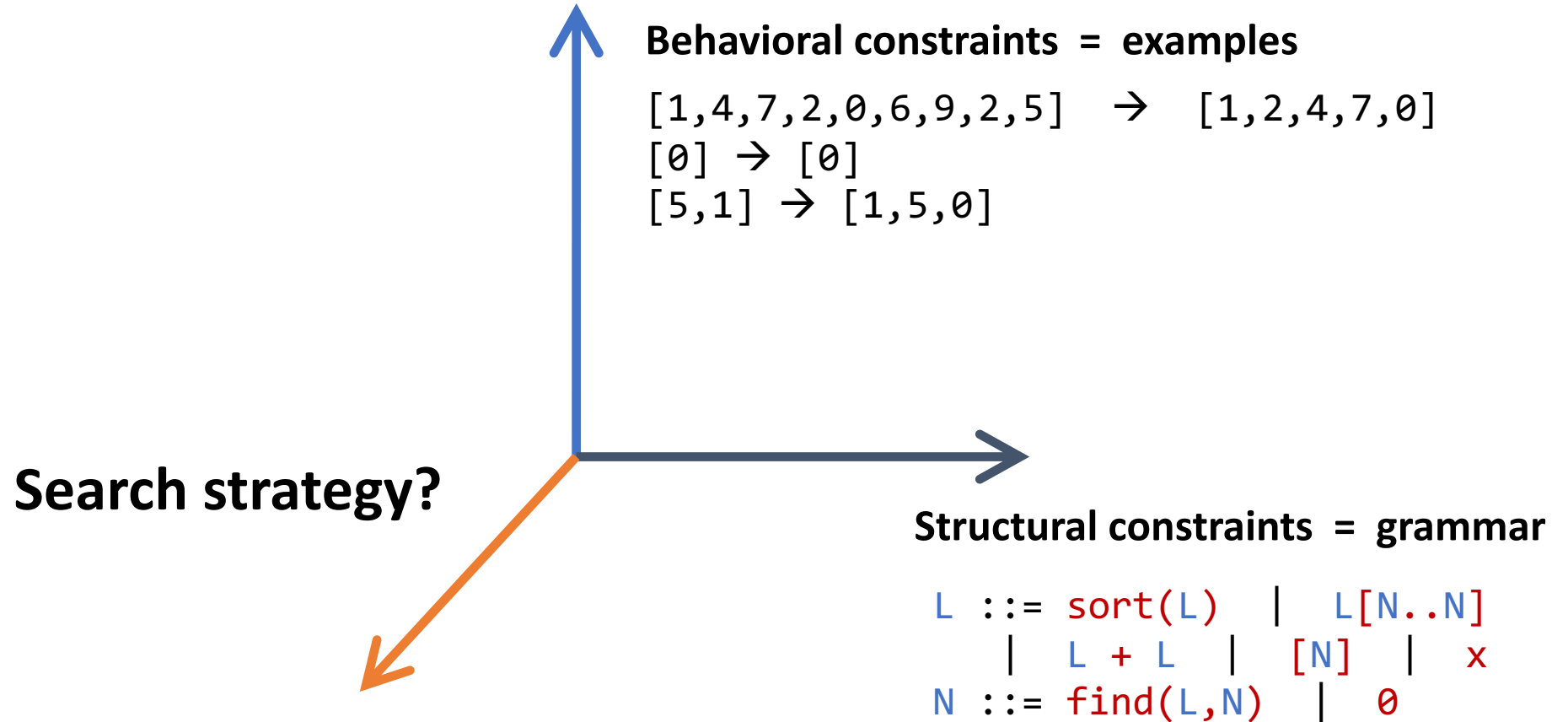


Counter-example guided inductive synthesis (CEGIS)

The Zendo of program synthesis



The problem statement



Enumerative search

Enumerative search

=

Explicit / Exhaustive Search

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

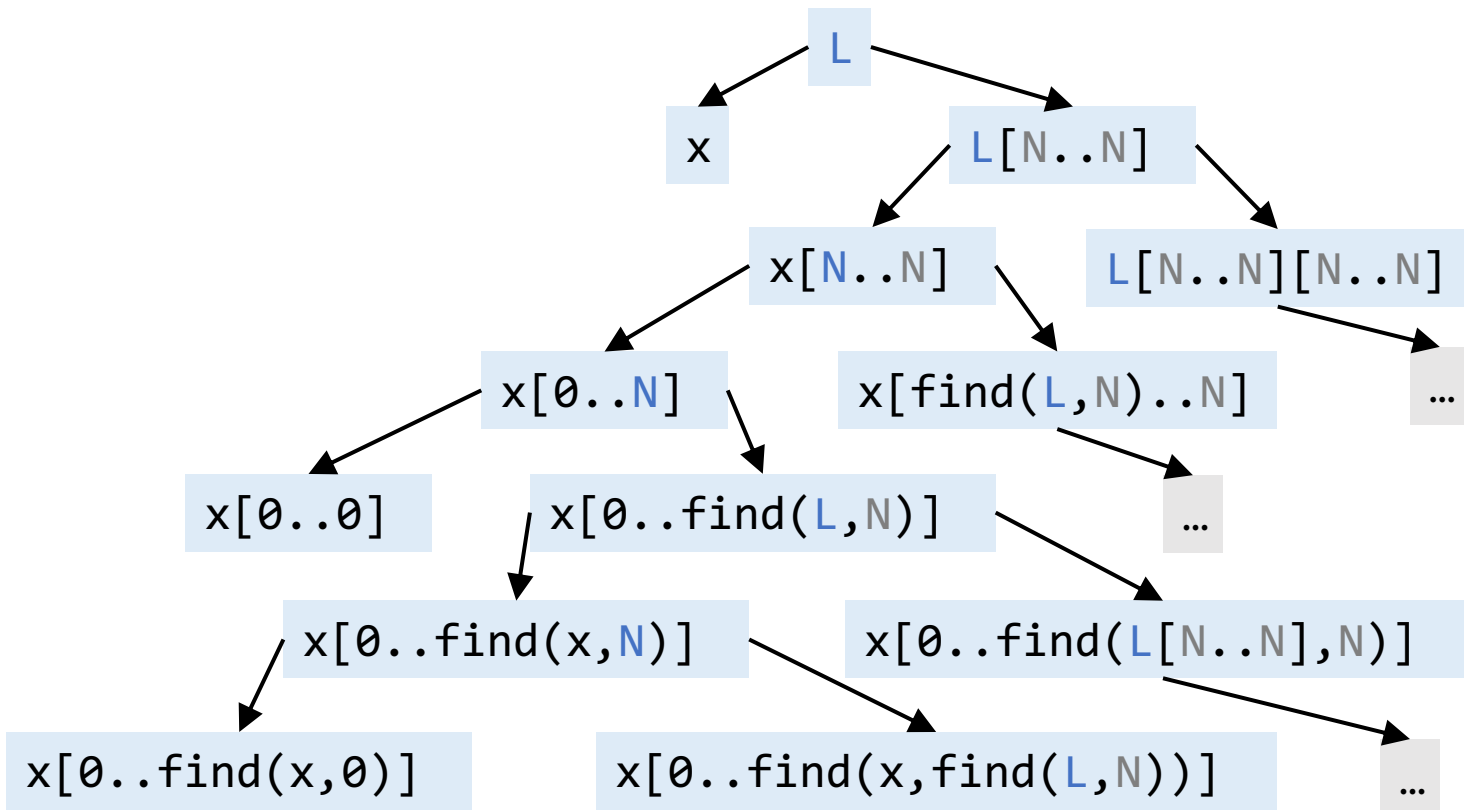
Challenge: How do we systematically enumerate all programs?

top-down **vs** bottom-up

Top-down enumeration: search space

Search space is a tree where

- nodes are whole incomplete programs
- edges are “derives in one step”



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

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

Top-down enumeration = tree traversal

Search tree can be traversed:

- depth-first (for fixed max depth)
- breadth-first
- later in class: best-first

General algorithm:

- Maintain a **worklist** of incomplete programs
- Initialize with the start non-terminal
- Expand left-most non-terminal using all productions

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

```
[[1,4,0,6]] → [[1,4]]
```

Top-down: algorithm

nonterminals rules (productions)
alphabet starting nonterminal
top-down($\langle \Sigma, N, R, S \rangle, [i \rightarrow o]$):
 $wl := [S]$
 while ($wl \neq []$):
 $\tau := wl.dequeue()$
 if ($complete(\tau) \wedge \tau([i]) = [o]$):
 return τ
 $wl.enqueue(unroll(\tau))$

unroll(τ):
 $wl' := []$
 $A :=$ left-most non-term in τ
 forall ($A \rightarrow rhs$) **in** R :
 $\tau' = \tau[A \rightarrow rhs]$
 if !exceeds_bound(τ'): $wl' += \tau'$
 return wl'

depth- or breadth-first
depending on where you enqueue

$L ::= L[N..N] \quad |$
 x
 $N ::= find(L, N) \quad |$
 \emptyset

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

can impose bounds on depth/size

Top-down: example (depth-first)

Worklist w1

iter 0: L

iter 1: x[✗] 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][✗] 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)][✓] 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

- The dynamic programming approach
- Maintain a **bank** of complete programs
- 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]] → [[1,4]]`

Bottom-up: algorithm (take 1)


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

new-terms($A \rightarrow \sigma(A_1 \dots A_k)$, d, bank):
 if ($d = 0 \wedge k = 0$) yield σ
 else forall $\langle t_1, \dots, t_k \rangle$ in bank^k:
 if $A_i \rightarrow^* t_i$: yield $\sigma(t_1, \dots, t_k)$

```

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

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


 inefficient, better index bank by non-terminal!

Bottom-up: algorithm (take 2)

```
bottom-up (< $\Sigma$ , N, R, S>, [ $i \rightarrow o$ ]):
```

```
  bank[A] := {} forall A
```

```
  for d in [0..]:
```

```
    forall (A  $\rightarrow$  rhs) in R:
```

```
      forall t in new-terms(A $\rightarrow$ rhs, d, bank):
```

```
        if (A = S  $\wedge$  t([i]) = [o]):
```

```
          return t
```

```
        bank[A] += t;
```

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

```
  if (d = 0  $\wedge$  k = 0) yield  $\sigma$ 
```

```
  else forall <t1, ..., tk> in bank[A1]  $\times$  ...  $\times$  bank[Ak]:
```

```
    yield  $\sigma(t_1, \dots, t_k)$ 
```

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

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

inefficient, generating same terms again and again!
better index bank by depth

Bottom-up enumeration

```
bottom-up (< $\Sigma$ , N, R, S>, [i  $\rightarrow$  o]):
```

```
  bank[A,d] := {} forall A, d
```

```
  for d in [0..]:
```

```
    forall (A  $\rightarrow$  rhs) in R:
```

```
      forall t in new-terms(A $\rightarrow$ rhs, d, bank):
```

```
        if (A = S  $\wedge$  t([i]) = [o]):
```

```
          return t
```

```
        bank[A,d] += t;
```

```
new-terms(A  $\rightarrow$   $\sigma$ (A1...Ak), d, bank):
```

```
  if (d = 0  $\wedge$  k = 0) yield  $\sigma$ 
```

```
  else forall <d1,...,dk> in [0..d-1]k s.t. max(d1,...,dk) = d-1:
```

```
    forall <t1,...,tk> in bank[A1,d1]  $\times$  ...  $\times$  bank[Ak,dk]:
```

```
      yield  $\sigma$ (t1,...,tk)
```

```
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)`

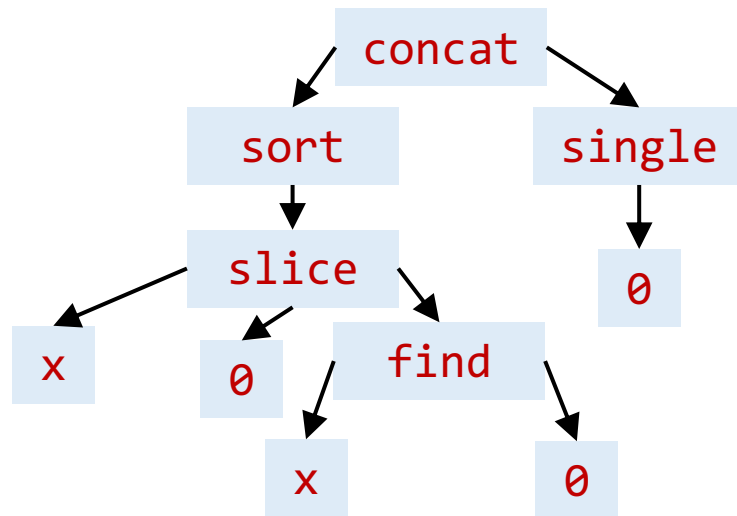
d = 2: `sort(sort(x))` `sort(x[0..0])` `sort(x + x)`
 `sort([0])` `x + (x + x)` `x + [0]` `sort(x) + x`
 `x[0..0] + x` `(x + x) + x` `[0] + x` `x + x[0..0]`
 `x + sort(x)` `x[0..find(x,0)]` 

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

`[[1,4,0,6]]` → `[1,4]`

Bottom-up: discussion

- What are some optimizations that come to mind?
- Instead of by depth, we can enumerate by size
 - Why would we want that?



depth = 4, size = 10
programs of size ≤ 10 : 8667
programs of depth ≤ 4 : >1M

- Which parts of the algo would we need to change?

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 **complete**

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

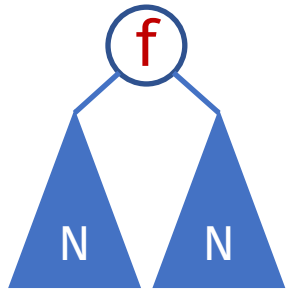
- Candidates are **complete** but might not be **whole**

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

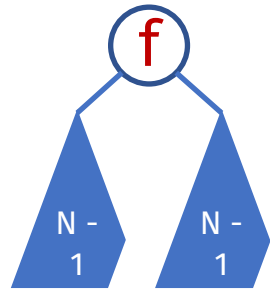
How to make it scale

Prune

- Discard useless subprograms



$$m * N^2$$



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

Prioritize

- Explore more promising candidates first

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