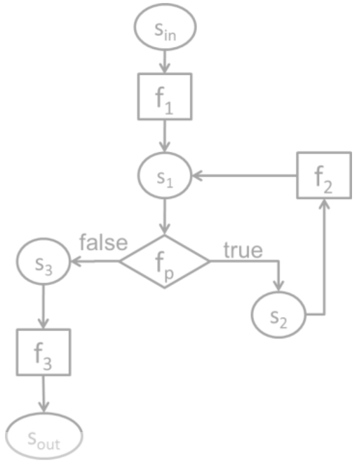
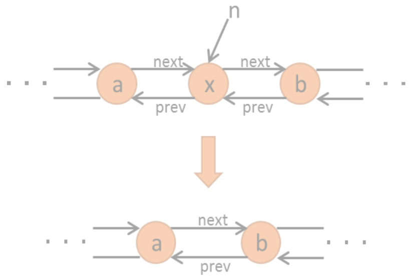
$\exists c \forall in\, Q(c, in)$

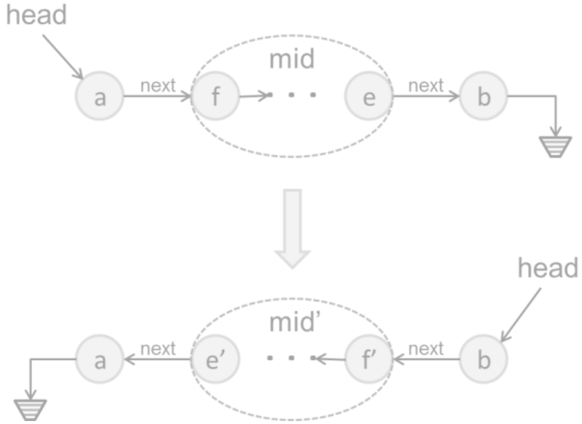
```
/* Average of x and y without using x+y (avoid overflow)*/
int avg(int x, int y){
  int t = expr({x/2, y/2, x%2, y%2, 2 }, {PLUS, DIV});
  assert t == (x+y)/2;
  return t;
}
```

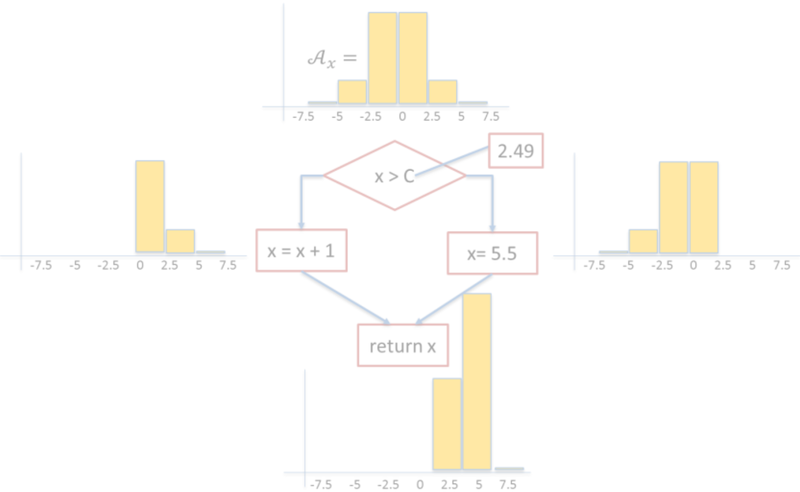





```
{
   s  = n.succ;
   p = n.pred;
   p.succ = s;
   s.pred = p;
}
```

Unit I: Synthesis from Examples





$\varphi(p)$

$Sk[c](in)$

Lecture 2
Syntax-Guided Synthesis and Enumerative Search

Nadia Polikarpova

Logistics

Shared Google folder

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

EasyChair

- Does everyone have PC access?
- Submit review by Wednesday
- Paper discussion on Thursday

Other questions?

This week

Synthesis from examples: motivation and history

Syntax-guided synthesis

- grammars as structural constraints

Enumerative search

- enumerating all programs generated by a grammar

How to make it scale

- search space pruning and prioritization

Synthesis from Examples

=

Programming by Example

=

Inductive Synthesis
(Inductive Learning)

Programming by Example: Motivation

(logics, automata, etc.)

(code)

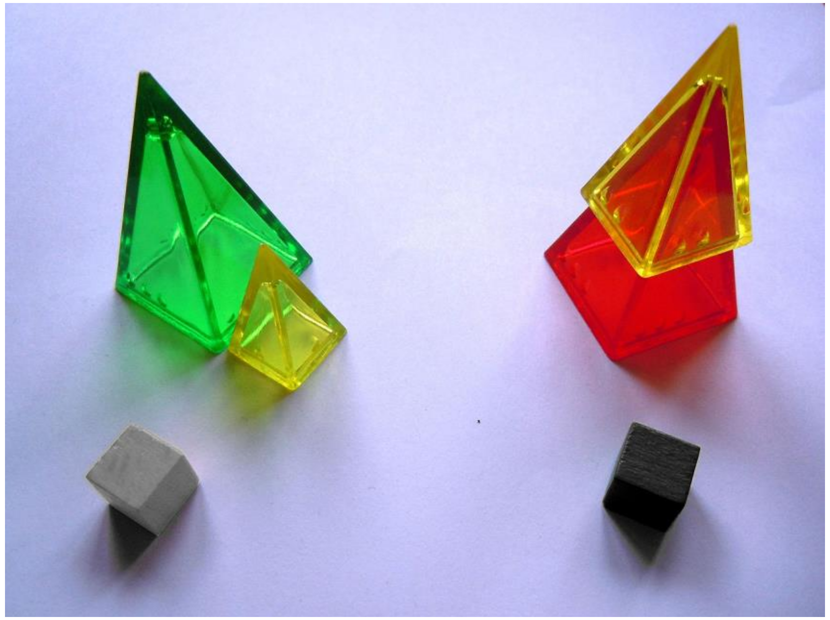
Algorithm Designers

Software Developers

End-Users

(examples!)

The Zendo game



This is called inductive learning!

The teacher makes up a secret rule

- e.g. all pieces must be grounded

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

Students take turns to build koans and ask the teacher to label them

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

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

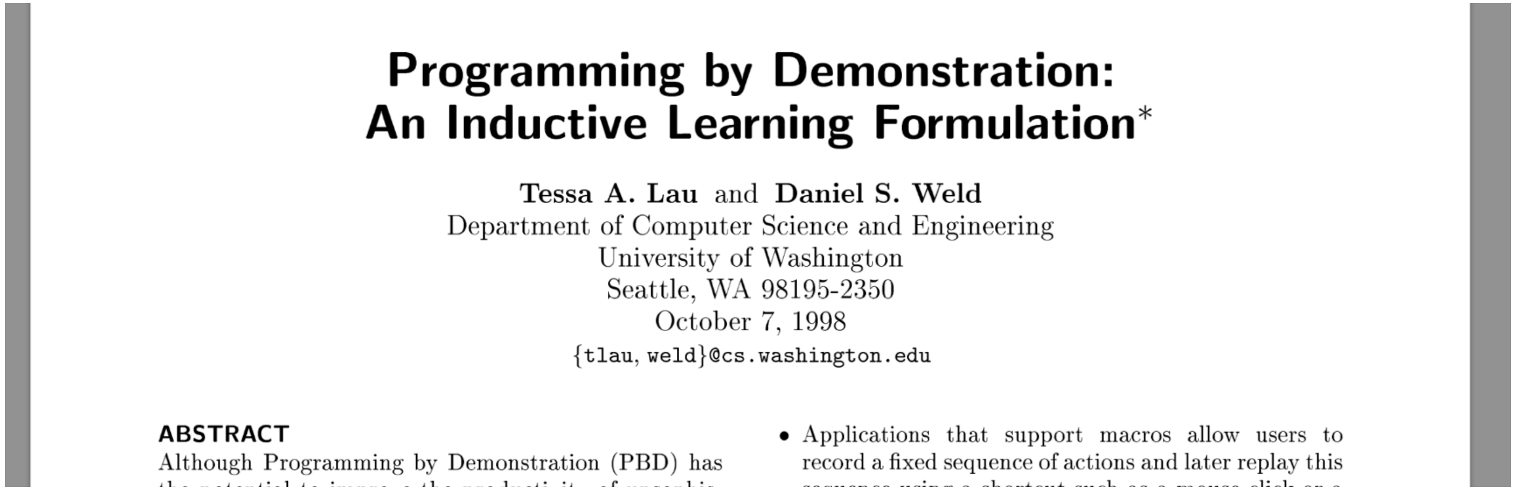
- Similar to Zendo

Became the foundation of machine learning

A little bit of history: PBE/PBD

Early systems searched a predefined list of programs

Tessa Lau: bring inductive learning techniques into PBE


Programming by Demonstration:
An Inductive Learning Formulation*

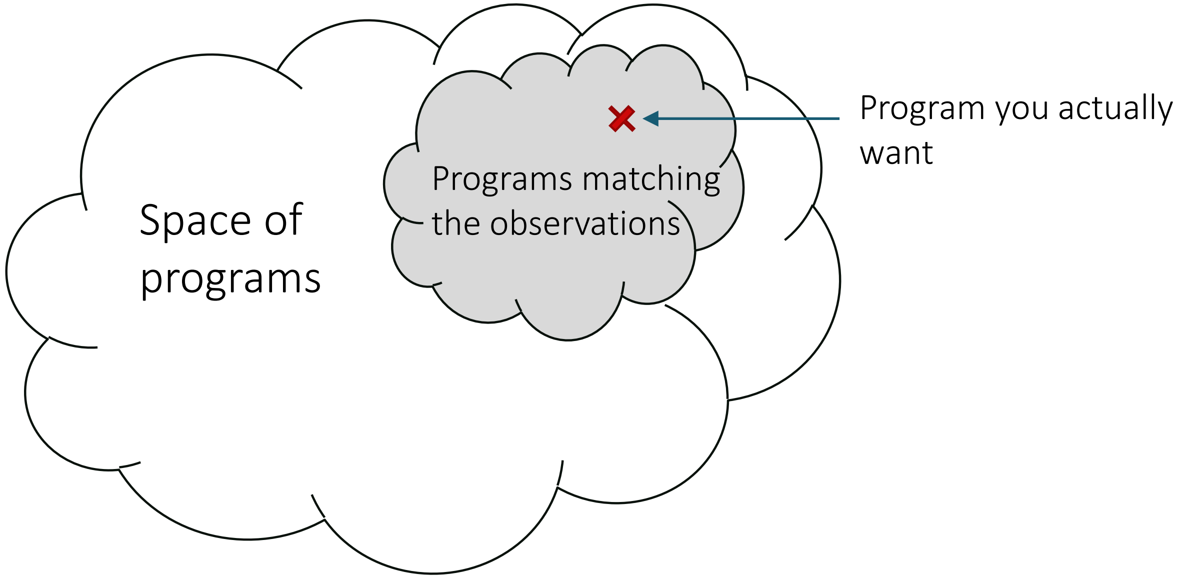
Tessa A. Lau and Daniel S. Weld
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
October 7, 1998
{tlau, weld}@cs.washington.edu

ABSTRACT
Although Programming by Demonstration (PBD) has

- Applications that support macros allow users to record a fixed sequence of actions and later replay this

Tessa Lau

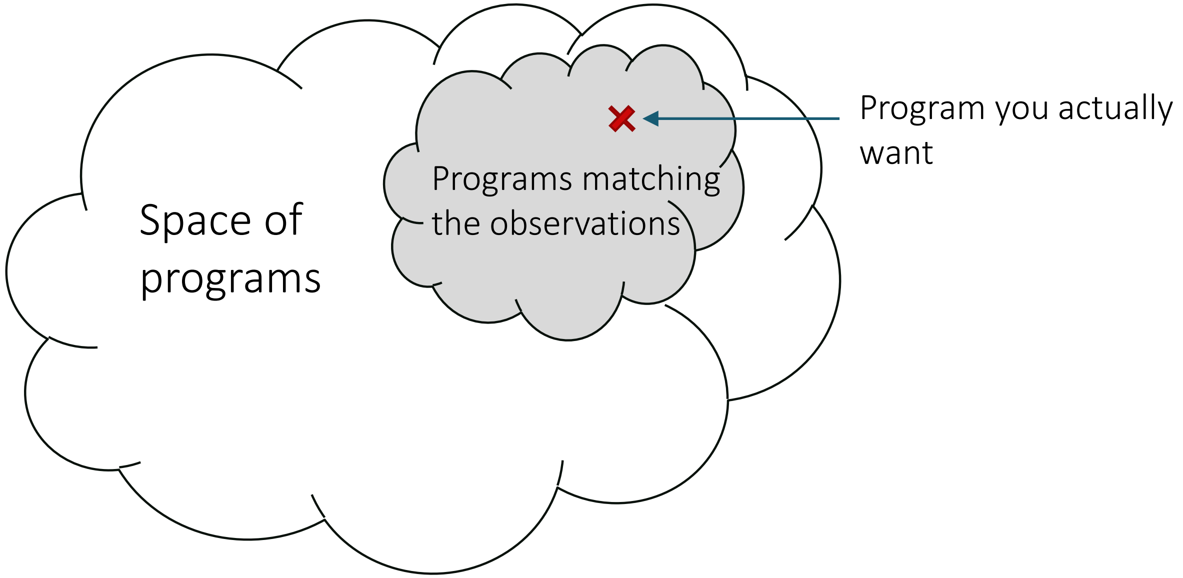
Key issues in inductive learning



(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 emphasizes (2)

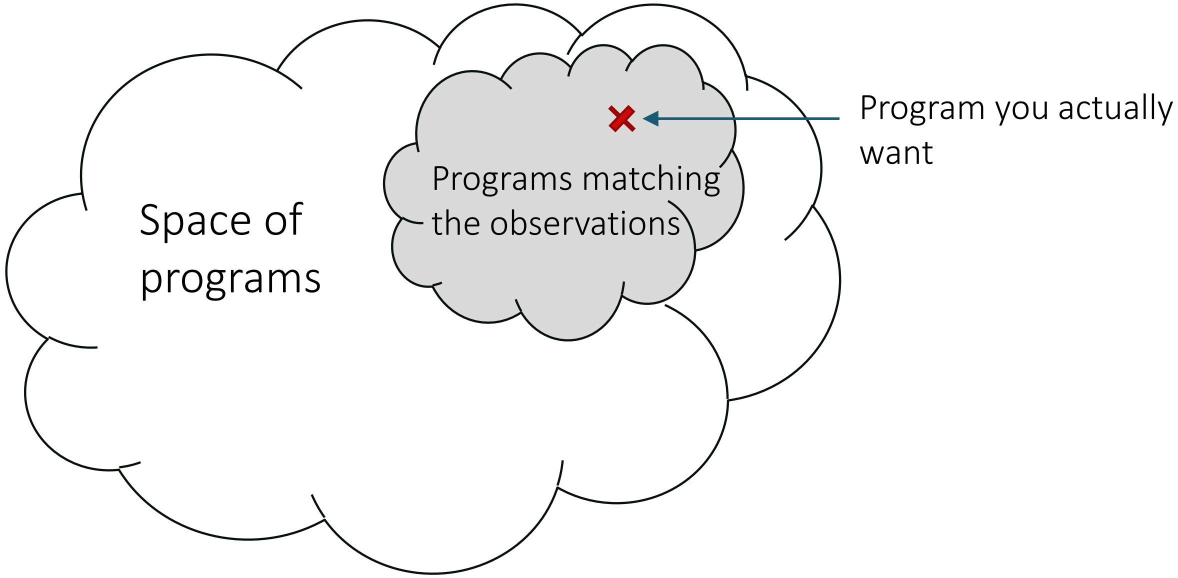
- Fix the space so that (1) is easy

So did a lot of PBD work

(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

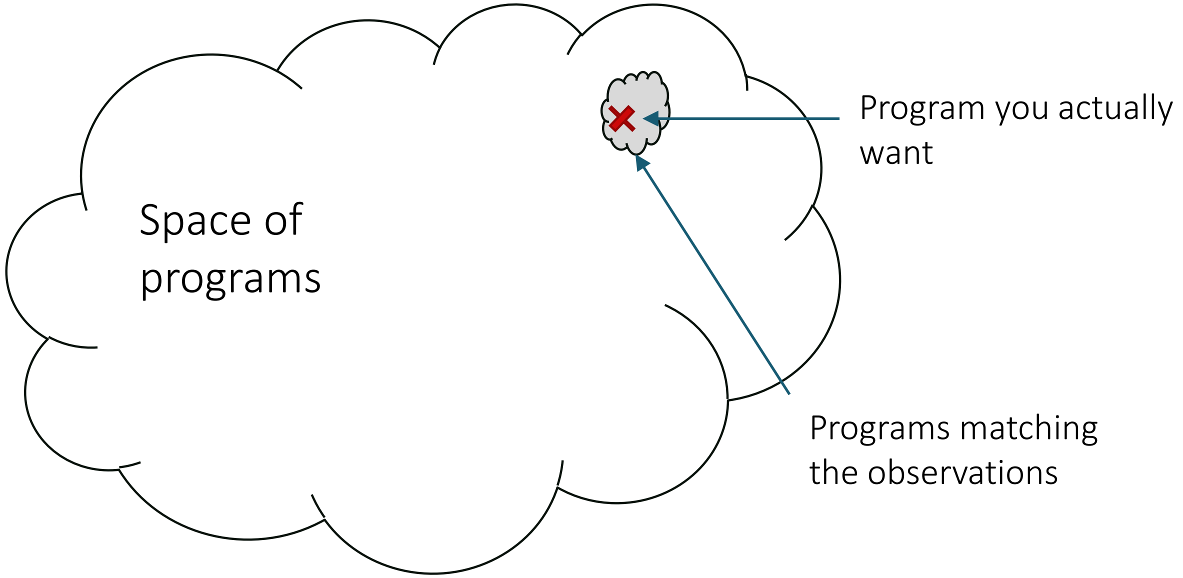


Modern emphasis

(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



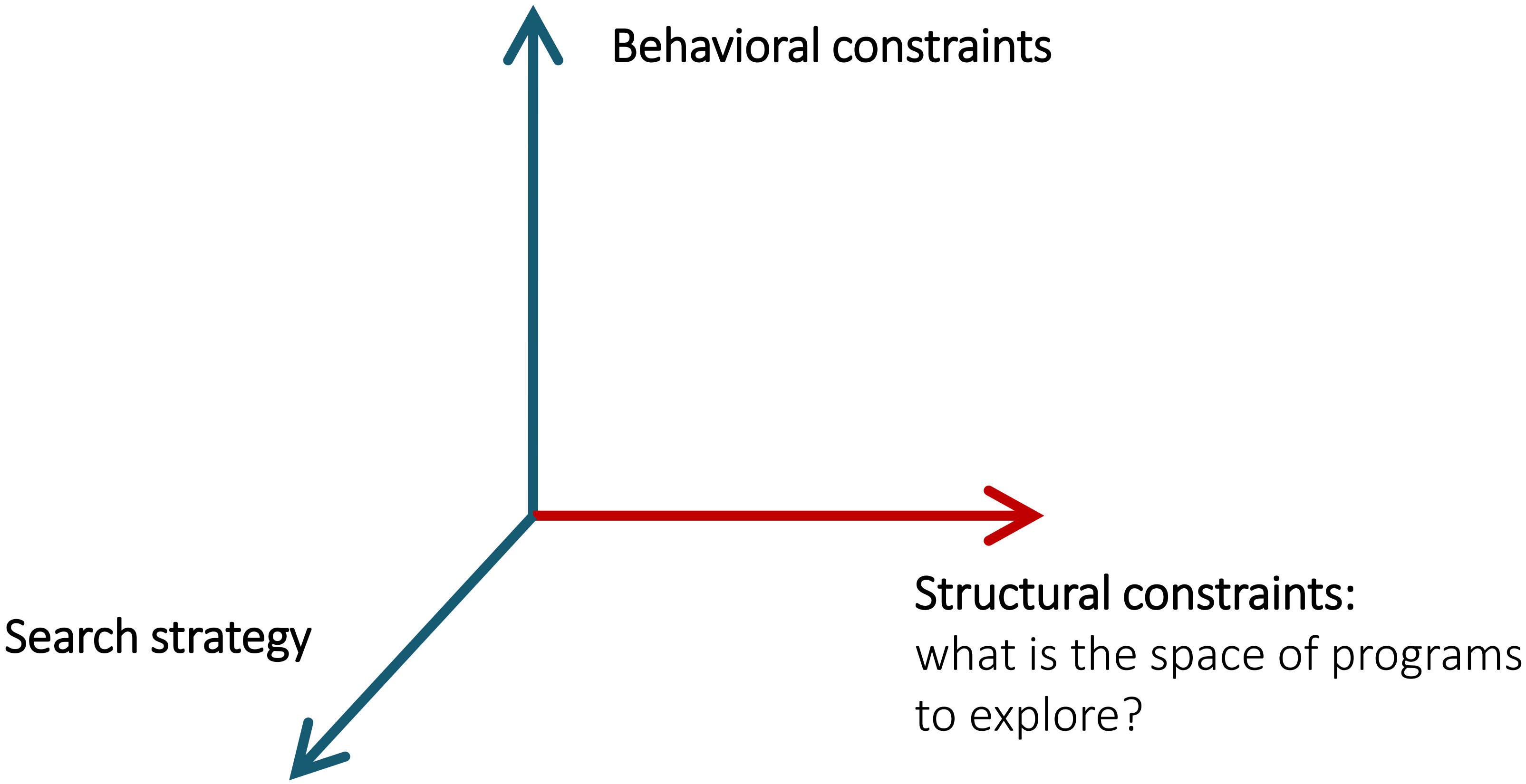
Modern emphasis

- If you can do really well with (1) you can win
- (2) is still important

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

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

Dimensions in program synthesis



Syntax-Guided Synthesis

Example

```
[1,4,7,2,0,6,9,2,5,0,3,2,4,7]  →  [1,2,4,7,0]

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

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

Context-free grammars (CFGs)

starting nonterminal

terminals

nonterminals

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

productions

CFGs as structural constraints

Space of programs

=

all complete programs generated by rewriting the starting nonterminal according to productions

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

→

`x` `sort(x)` `x + x` `x[0..0]`

...

`x[0..find(x, 0)]`

...

`sort(x[0..find(x, 0)]) + [0]`

...

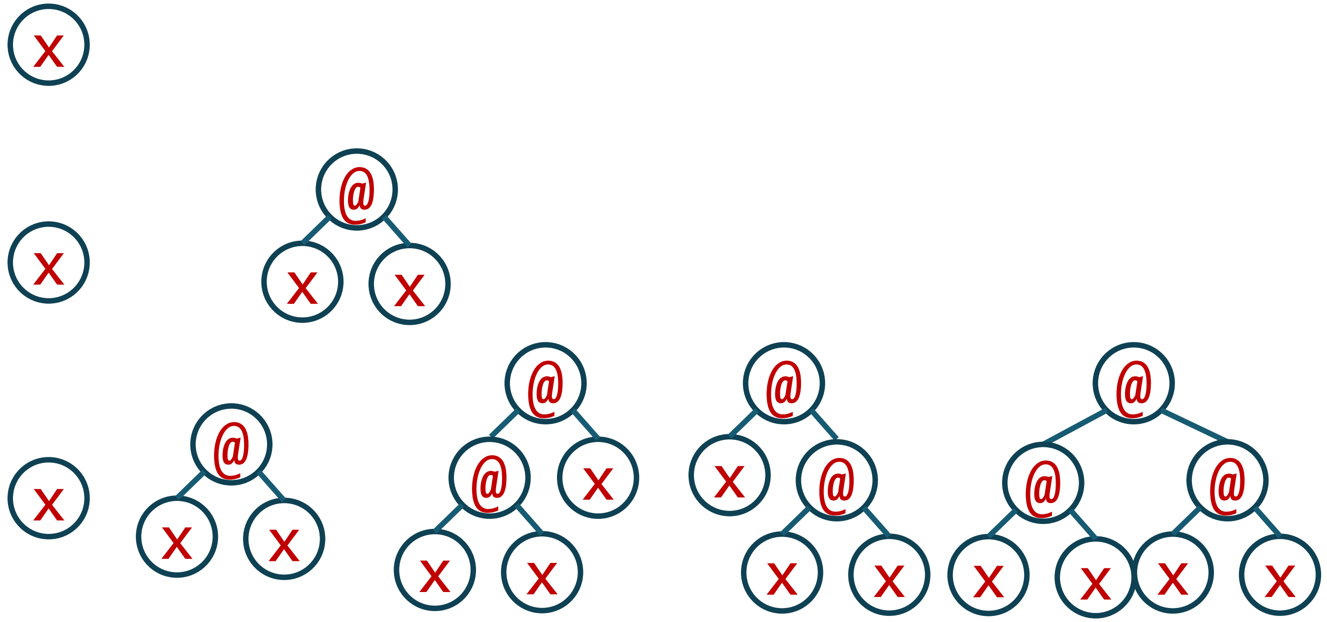
How big is the space?

`E ::= x | E @ E`

depth <= 1

depth <= 2

depth <= 3



N(1) = 1

N(2) = 2

N(3) = 5

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

How big is the space?

```
E ::= x | E @ E
```

$$N(d) = 1 + N(d - 1)^2 \qquad N(d) \sim c^{2^d} \qquad (c > 1)$$

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?

$$
\begin{aligned}
E ::= \quad & x_1 \mid \ldots \mid x_k \mid \\
& E\ @_1\ E \mid \ldots \mid E\ @_m\ E
\end{aligned}
$$

$$N(0) = 0$$

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

N(1) = 3
N(2) = 30
N(3) = 2703
N(4) = 21918630
N(5) = 1441279023230703
N(6) = 6231855668414547953818685622630
N(7) = 116508075215851596766492219468227024724121520304443212304350703

k = m = 3

CFGs as structural constraints

Pros:

- Clean declarative description
- Easy to sample
- Easy to explore exhaustively

Cons:

- Insufficiently expressive

What if we know the following:

- Sort can be called at most once
- Sub-list is never called on a concatenation of singletons
- In a call to sub-list, the start index is <= the end index

Grammars vs generators

Grammars

Pros:

- Clean declarative description
- Easy to sample
- Easy to explore exhaustively

Cons:

- Insufficiently expressive

Generators

- Programs that produce programs

Pros:

- Extremely general
  - easy to enforce arbitrary constraints

Cons:

- Extremely general
  - Hard to analyze and reason about
  - Hard to automatically discover structure of the space

The SyGuS project

[Alur et al. 2013]

SyGuS problem = < theory, spec, grammar >

A "library" of types and function symbols

Example: Linear Integer Arithmetic (LIA)

```
True, False
0,1,2,...
∧, ∨, ¬, +, ≤, ite
```

CFG with terminals in the theory (+ input variables)

Example: Conditional LIA expressions w/o sums

```
E ::= x | ite C E E
C ::= E ≤ E | C ∧ C | ¬C
```

The SyGuS project

SyGuS problem = < theory, spec, grammar >

A first-order logic formula over the theory

Examples:

```
f(0, 1) = 1 ∧
f(1, 0) = 1 ∧
f(1, 1) = 1 ∧
f(2, 0) = 2
```

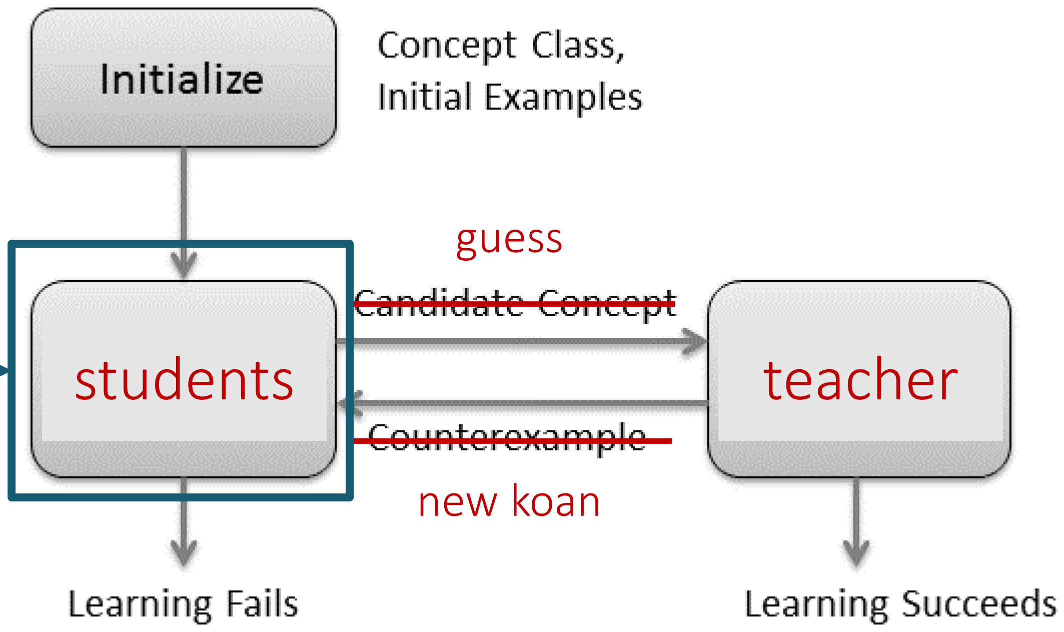
Formula with free variables:

```
x ≤ f(x, y) ∧
y ≤ f(x, y) ∧
(f(x, y) = x ∨ f(x, y) = y)
```

Counter-example guided inductive synthesis

The Zendo of program synthesis

Our focus in this unit



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]
```

Structural constraints = grammar

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

Search strategy?

Enumerative search

Enumerative search

=

Explicit / Exhaustive Search

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

Bottom-up enumeration

Start from terminals

Combine sub-programs into larger programs using productions

Q: “Run” bottom-up on the board with

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

```
[[1,4,0,6]  →  [1,4]]
```

terminals nonterminals rules (productions) starting nonterminal

```
bottom-up (<T, N, R, S>, [i → o]) {
  P := [t | t in T && t is nullary]
  while (true)
    P += grow(P);
    forall (p in P)
      if (whole(p) && p([i]) = [o])
        return p;
}

grow (P) {
  P' := []
  forall (r in R)
    P' += [r[N -> ps] | ps in P]
  return P';
}
```

Top-down enumeration

Start from the start non-terminal

Expand remaining non-terminals using productions

Q: "Run" top-down on the board with

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

[[1,4,0,6] → [1,4]]
```

```
top-down(<T, N, R, S>, [i → o]) {
  P := [S]
  while (P != [])
    p := P.dequeue();
    if (ground(p) && p([i]) = [o])
      return p;
    P.enqueue(unroll(p));
}

unroll(p) {
  P' := []
  forall (N in p)
    forall (N ::= rhs in R)
      P' += p[N -> rhs]
  return P';
}
```

Bottom-up vs top-down

Bottom-up | **Top-down**

Smaller to larger

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

Candidates are ground but might not be whole

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

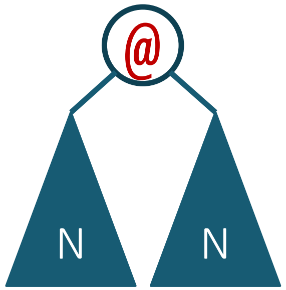
Candidates are whole but might not be ground

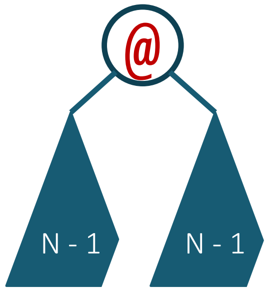
- Cannot always run on inputs
- Can always relate to outputs (?)

How to make it scale

Prune

Discard useless subprograms





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

Prioritize

Explore more promising candidates first

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

dequeue this first