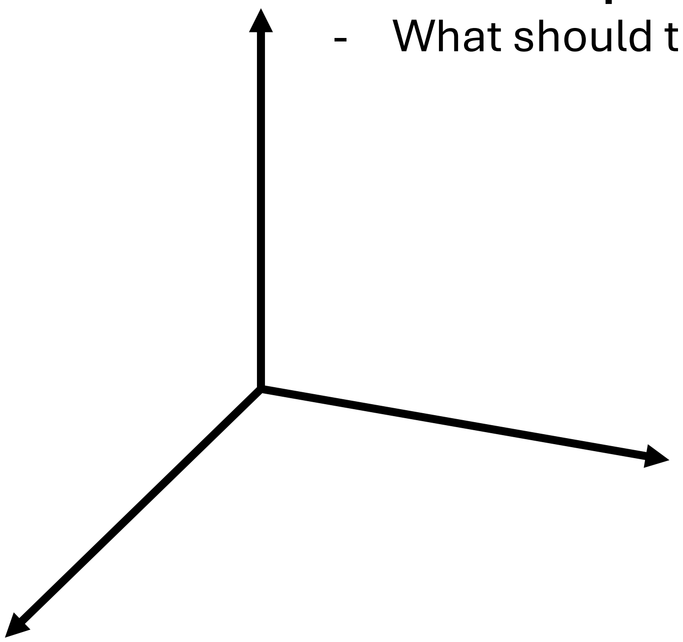
Machine Programming

Lecture 2 – Syntax, Semantics, and Inductive Synthesis

Ziyang Li

Dimensions in Program Synthesis



Behavioral Specification
- What should the program do?

Structural Specification
- What is the space of the programs?

Synthesis Strategy
- How do we find such a program?

Today

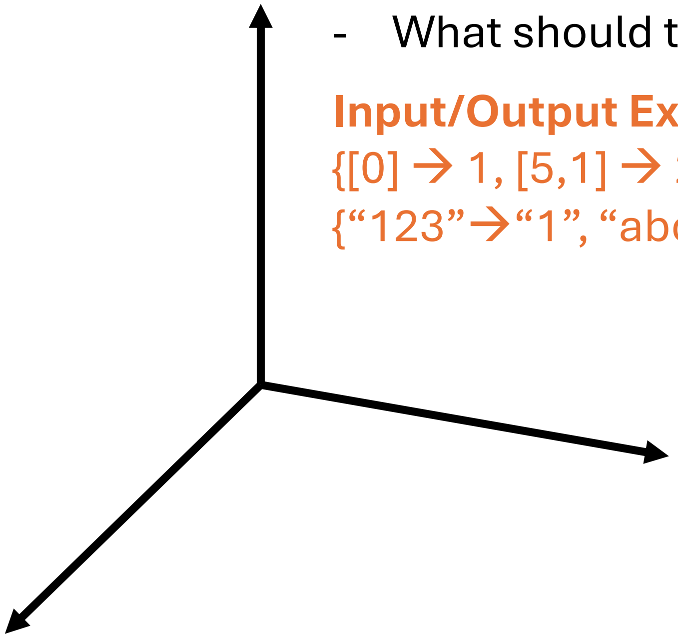
Behavioral Specification

- What should the program do?

Input/Output Examples

{[0] → 1, [5,1] → 2}

{“123”→“1”, “abc”→“a”}



Synthesis Strategy

- How do we find such a program?

Enumeration

- Enumerating all programs with a grammar
- Bottom-up vs top-down

Structural Specification

- What is the space of the programs?

Context-Free / Regular Tree Grammar

Expr e ::= c | e + e | e * e

Inductive Synthesis

=

Programming by Example

=

Inductive Program Synthesis

=

Inductive Programming

=

Inductive Learning

{“123”→“1”, “abc”→“a”} → ???

{“123”→“1”, “abc”→“a”} ➔ input[0]

{“123”→“1”, “abc”→“a”} → input[0]

{[0] → 1, [5,1] → 2} → ???

{“123”→“1”, “abc”→“a”}	→	input[0]
{[0] → 1, [5,1] → 2}	→	len(input)

{“123”→“1”, “abc”→“a”} ➔ input[0], input[0:1], ...

{[0] → 1, [5,1] → 2} ➔ len(input) , min(input) + 1, ...

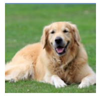
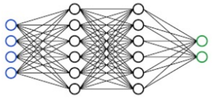
High-level Picture

- Learning abstraction / generalization from a set of observations

Program Synthesis

{[0] → 1, [5,1] → 2} ➔ len(input)
{“123”→“1”, “abc”→“a”} ➔ input[0]

Deep Learning

{  → dog,  → cat } ➔ 

MIT/LCS/TR-76

LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

Patrick H. Winston

September 1970

MIT/LCS/TR-76

LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

Patrick H. Winston

September 1970

Abstract

The research here described centers on how a machine can recognize concepts and learn concepts to be recognized. Explanations are found in computer programs that build and manipulate abstract descriptions of scenes such as those children construct from toy blocks. One program uses sample scenes to create models of simple configurations like the three-brick arch. Another uses the resulting models in making identifications. Throughout emphasis is given to the importance of using good descriptions when exploring how machines can come to perceive and understand the visual environment.

High-level Picture

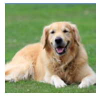
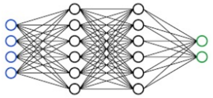
- Learning abstraction / generalization from a set of observations

Program Synthesis

{[0] → 1, [5,1] → 2} → min(input) + 1

{“123”→“1”, “abc”→“a”} → chars(input)[0]

Deep Learning

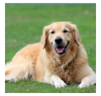
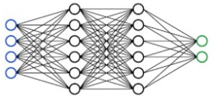
{  → dog,  → cat } → 

Space of Programs

Program Synthesis

{[0] → 1, [5,1] → 2} ➔ len(input)
{“123”→“1”, “abc”→“a”} ➔ input[0]

Deep Learning

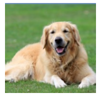
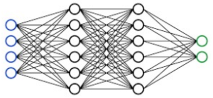
{  → dog,  → cat } ➔ 

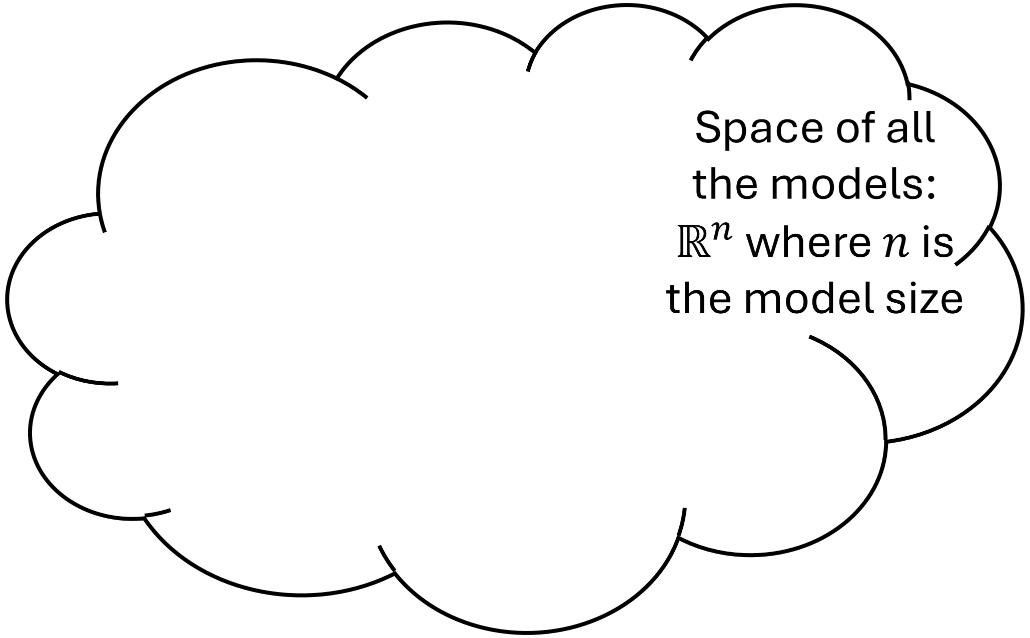
Space of Programs

Program Synthesis

{[0] → 1, [5,1] → 2} ➔ len(input)
{“123”→“1”, “abc”→“a”} ➔ input[0]

Deep Learning

{  → dog,  → cat } ➔ 

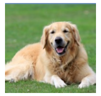
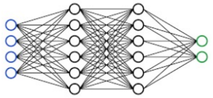


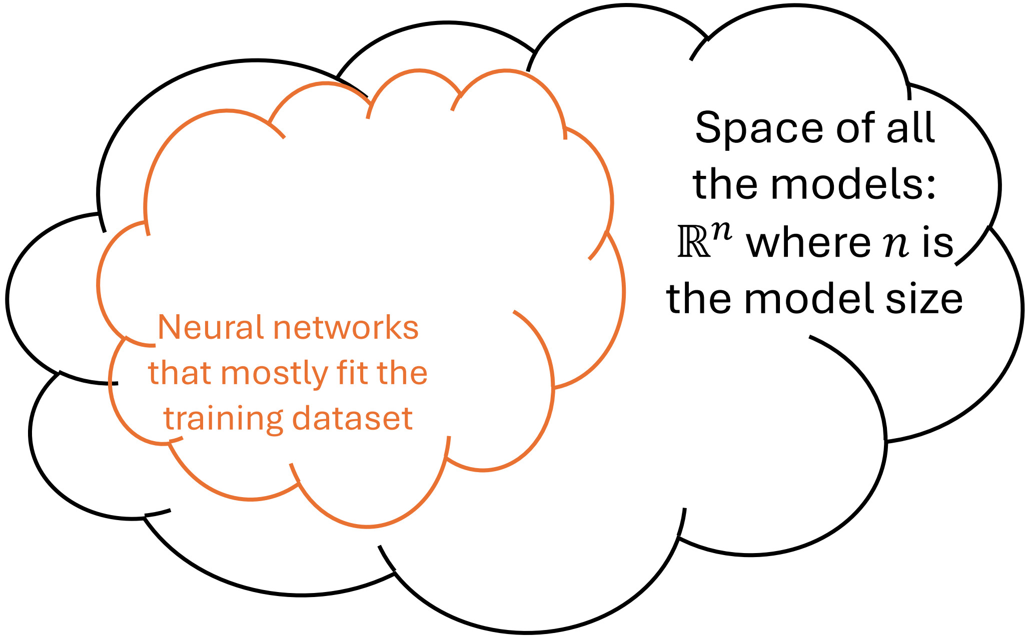
Space of Programs

Program Synthesis

{[0] → 1, [5,1] → 2} → len(input)
{“123”→“1”, “abc”→“a”} → input[0]

Deep Learning

{  → dog,  → cat } → 

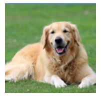
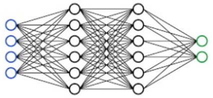


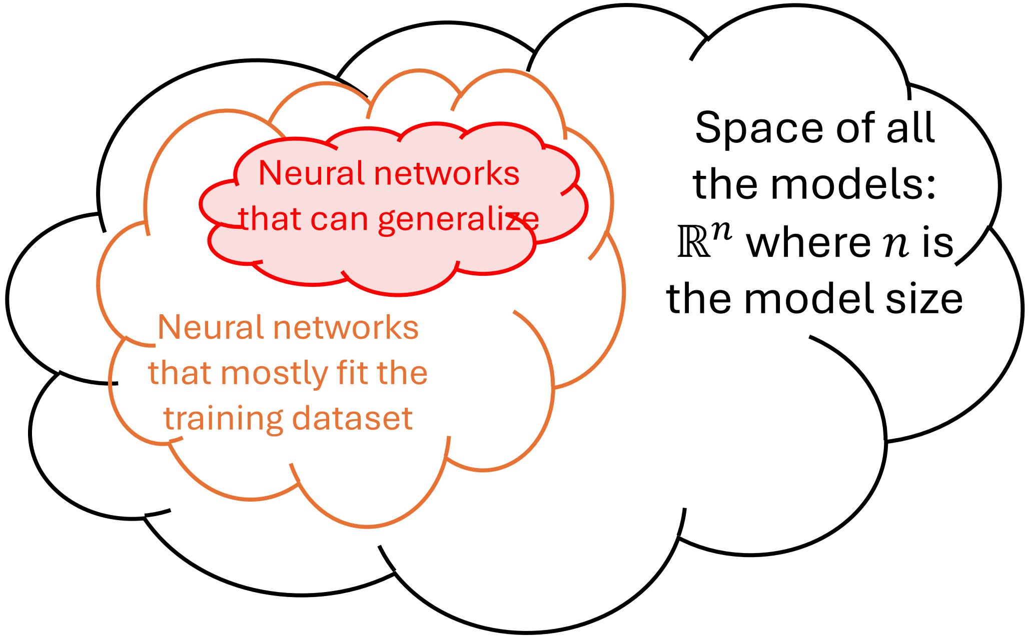
Space of Programs

Program Synthesis

{[0] → 1, [5,1] → 2} → len(input)
{“123”→“1”, “abc”→“a”} → input[0]

Deep Learning

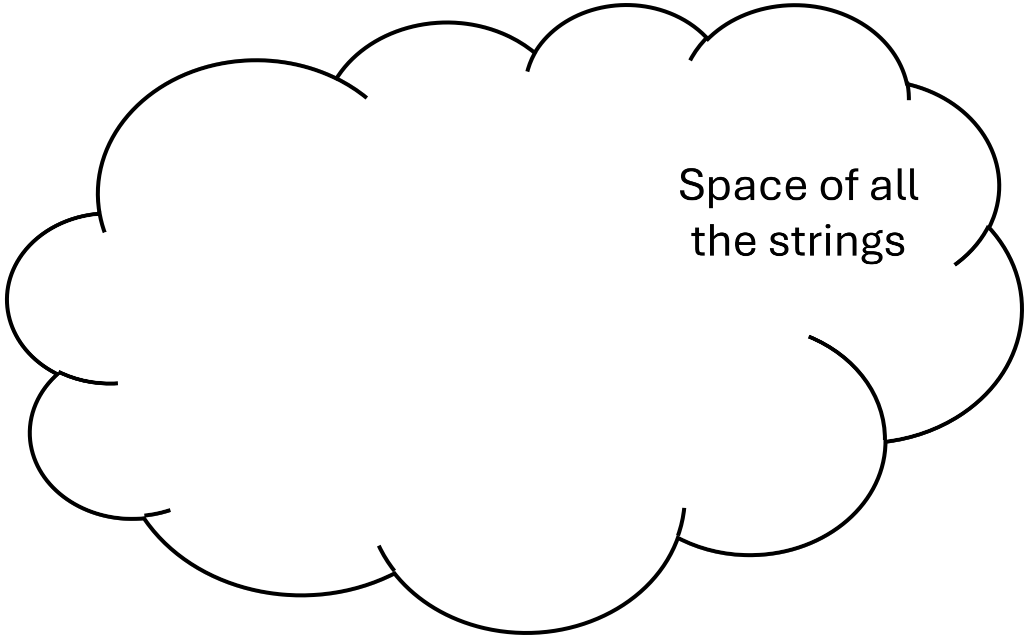
{  → dog,  → cat } → 



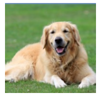
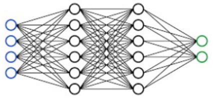
Space of Programs

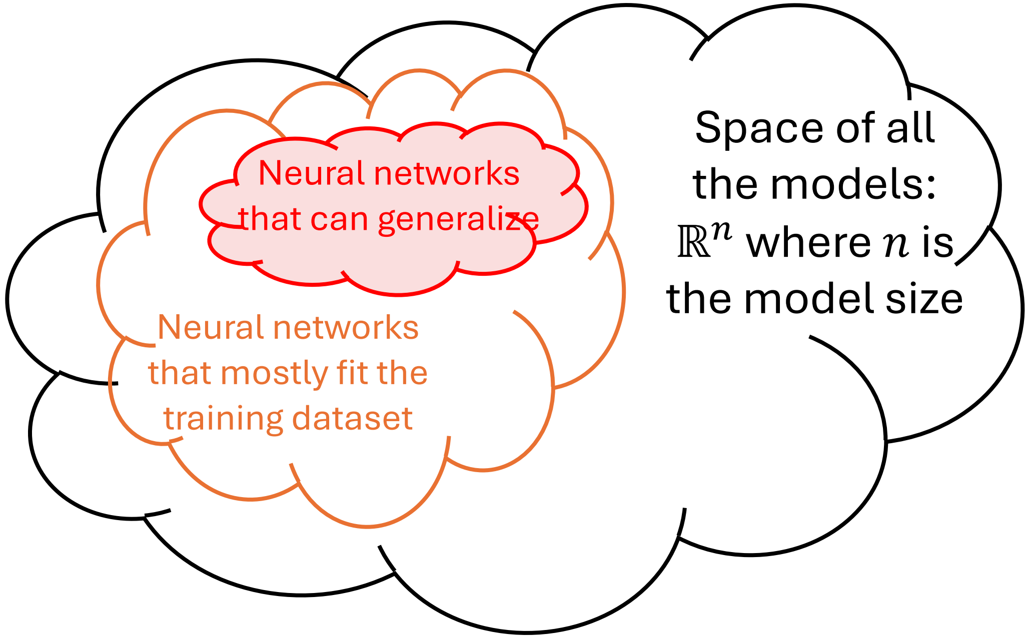
Program Synthesis

{[0] → 1, [5,1] → 2} ➔ len(input)
{“123”→“1”, “abc”→“a”} ➔ input[0]



Deep Learning

{  → dog,  → cat } ➔ 

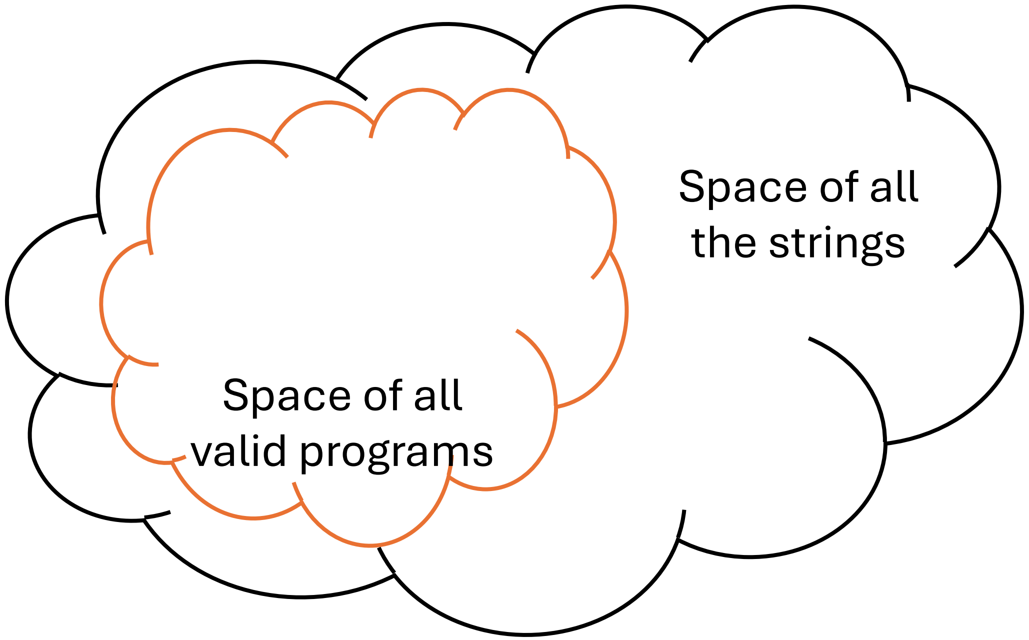


[Program Synthesis, Nadia Polikarpova, UCSD CSE291]

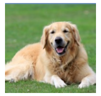
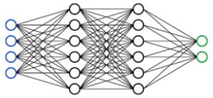
Space of Programs

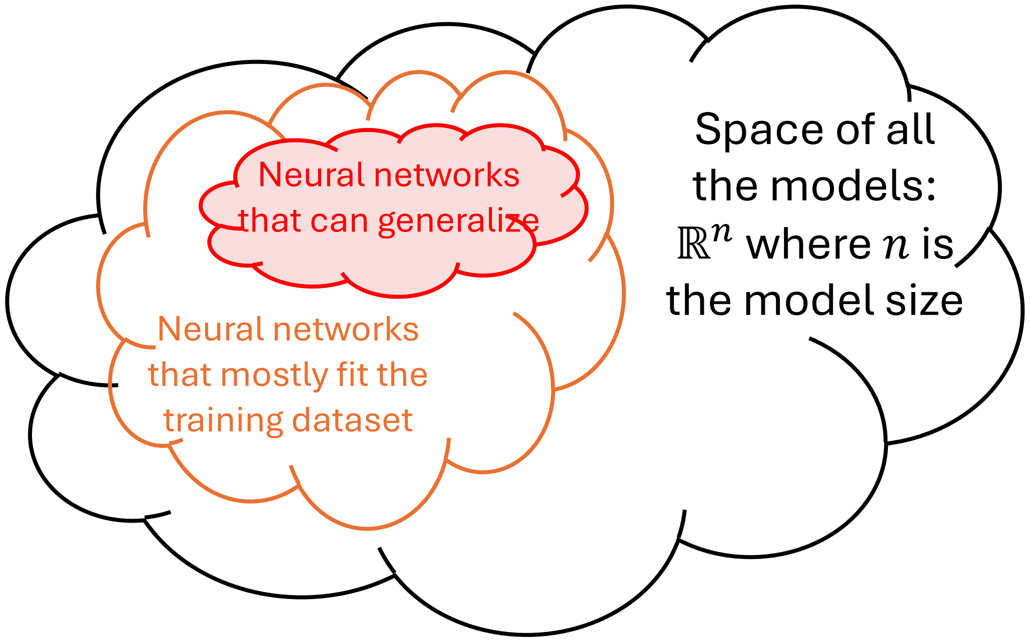
Program Synthesis

{[0] → 1, [5,1] → 2} → len(input)
{"123"→"1", "abc"→"a"} → input[0]



Deep Learning

{  → dog,  → cat } → 



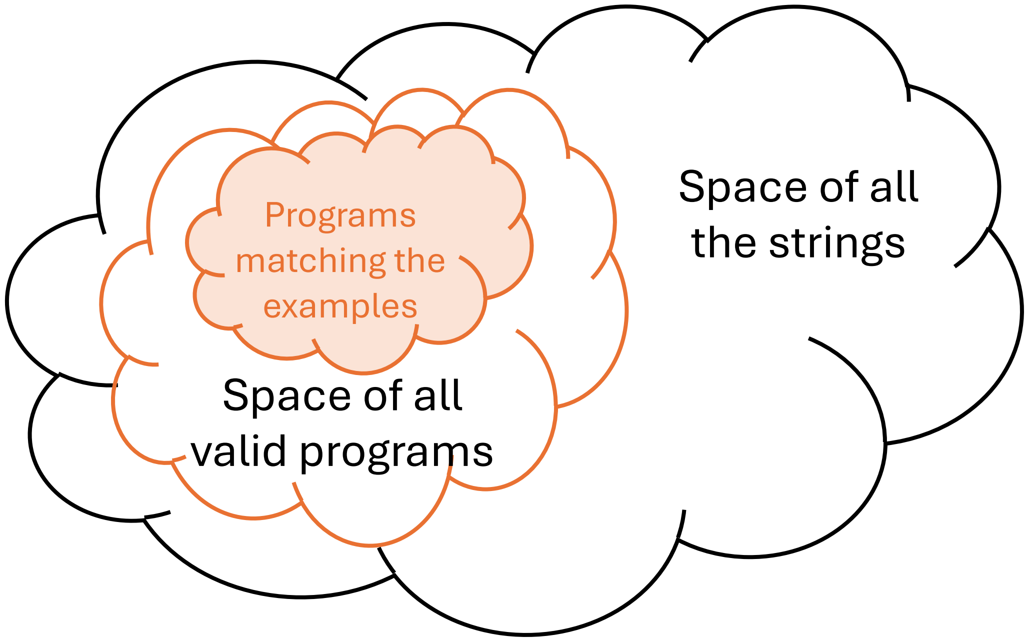
[Program Synthesis, Nadia Polikarpova, UCSD CSE291]

Space of Programs

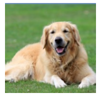
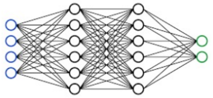
Program Synthesis

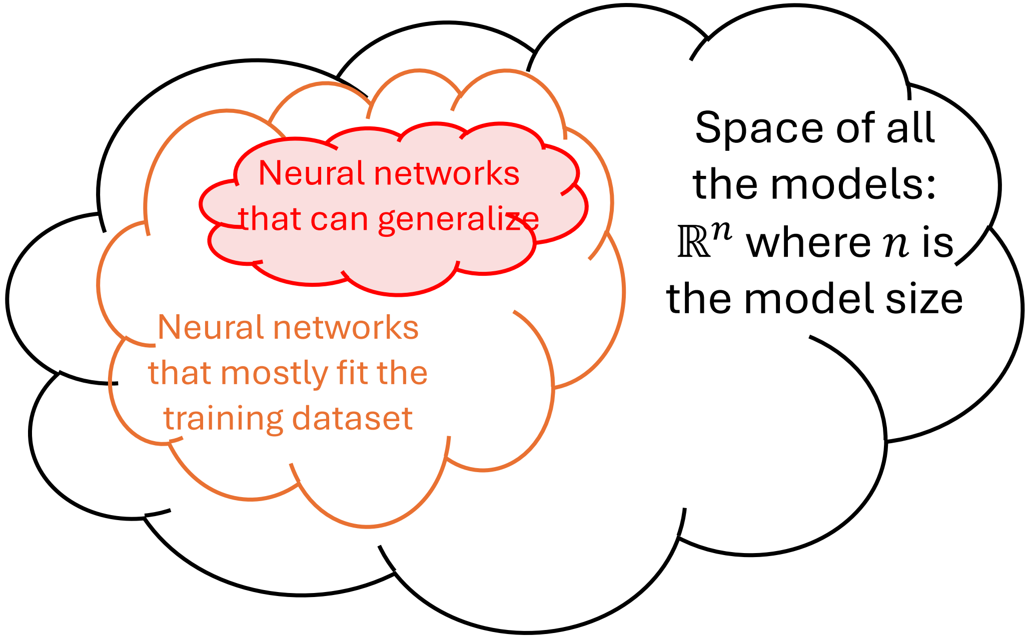
{[0] → 1, [5,1] → 2} → len(input)

{“123”→“1”, “abc”→“a”} → input[0]



Deep Learning

{  → dog,  → cat } → 

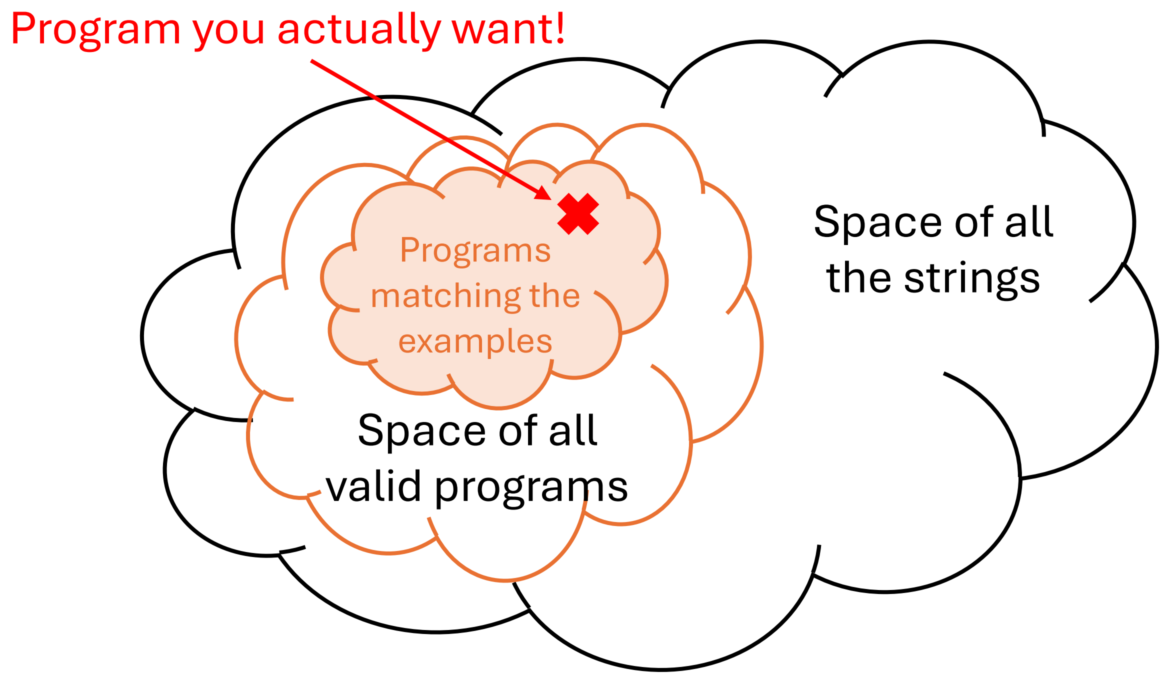


[Program Synthesis, Nadia Polikarpova, UCSD CSE291]

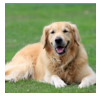
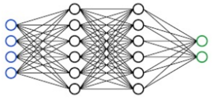
Space of Programs

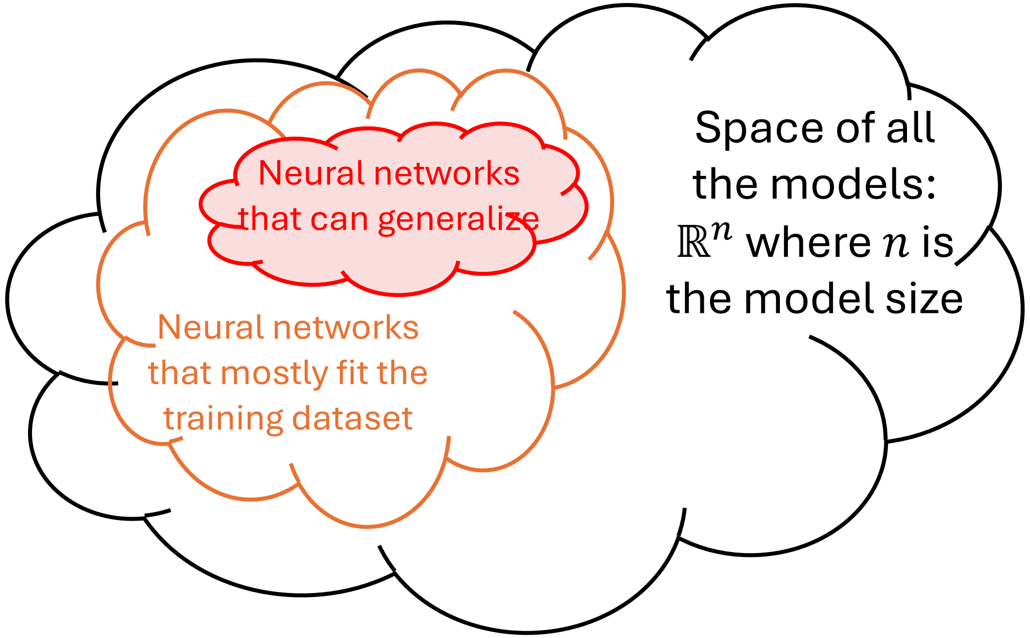
Program Synthesis

{[0] → 1, [5,1] → 2} → len(input)
{"123"→"1", "abc"→"a"} → input[0]



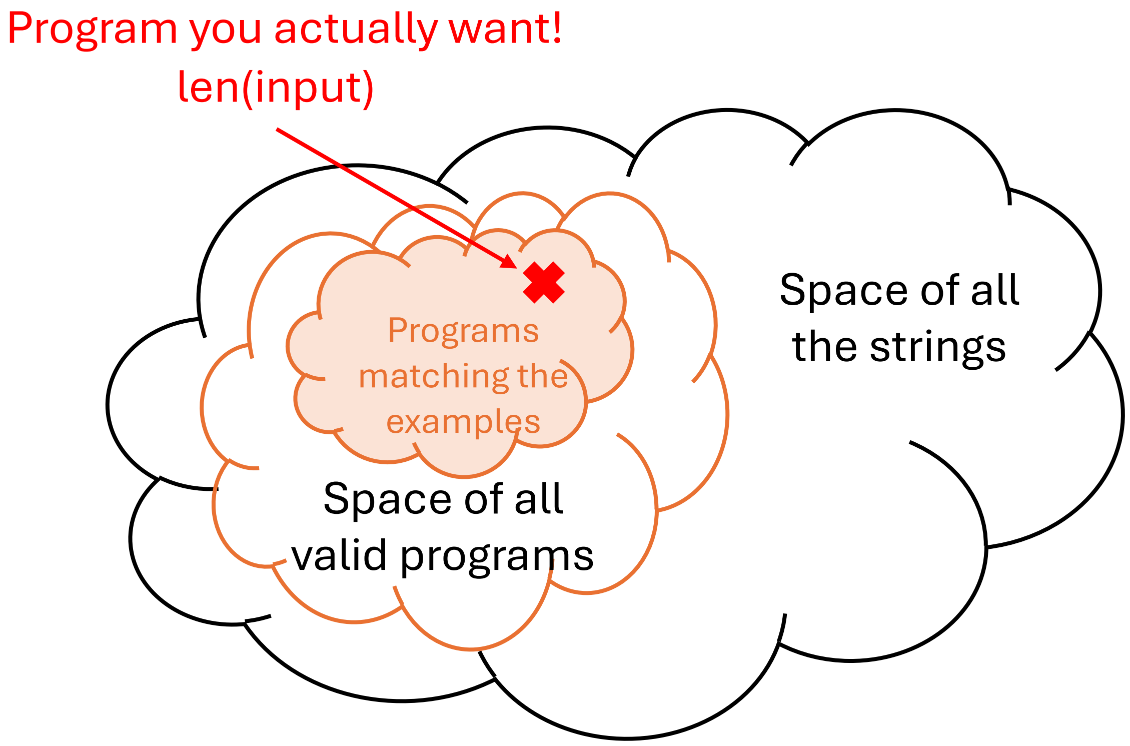
Deep Learning

{  → dog,  → cat } → 

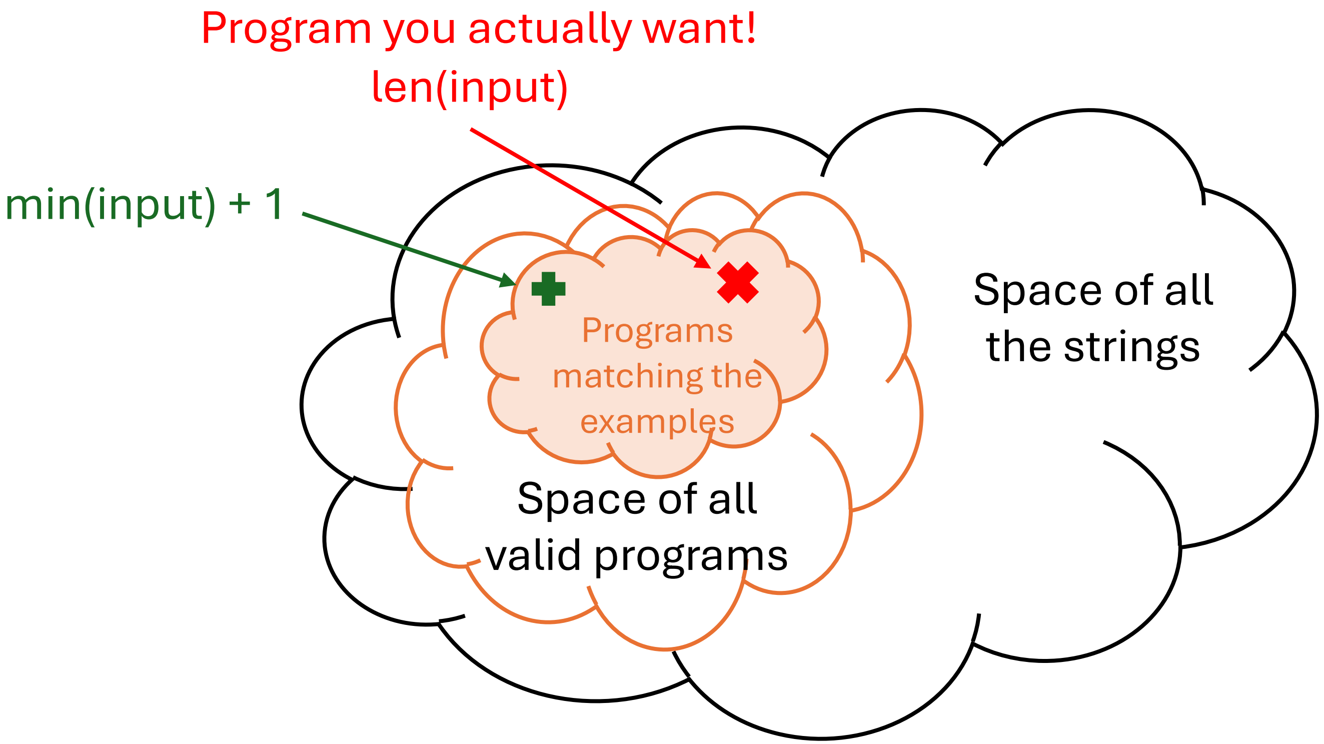


[Program Synthesis, Nadia Polikarpova, UCSD CSE291]

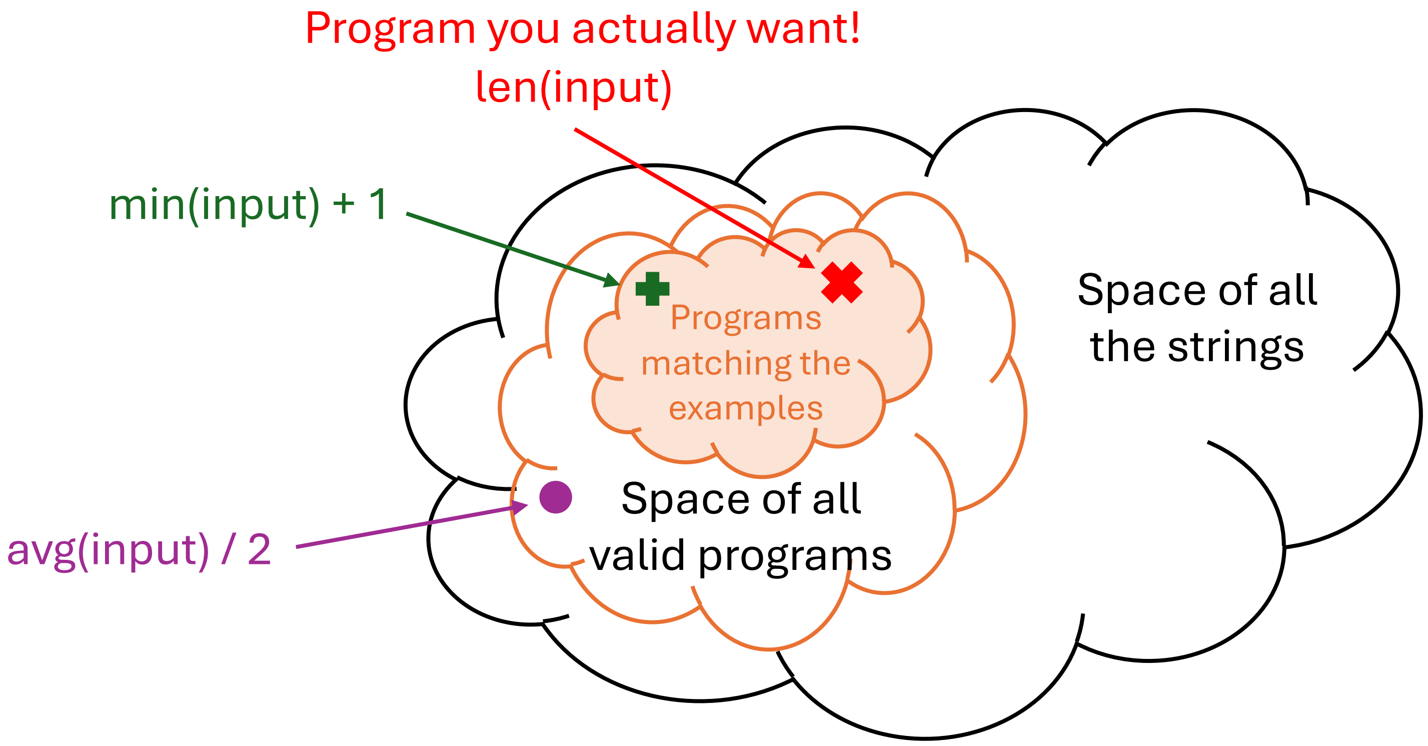
{[0] → 1, [5,1] → 2} ➔ len(input), min(input) + 1, …



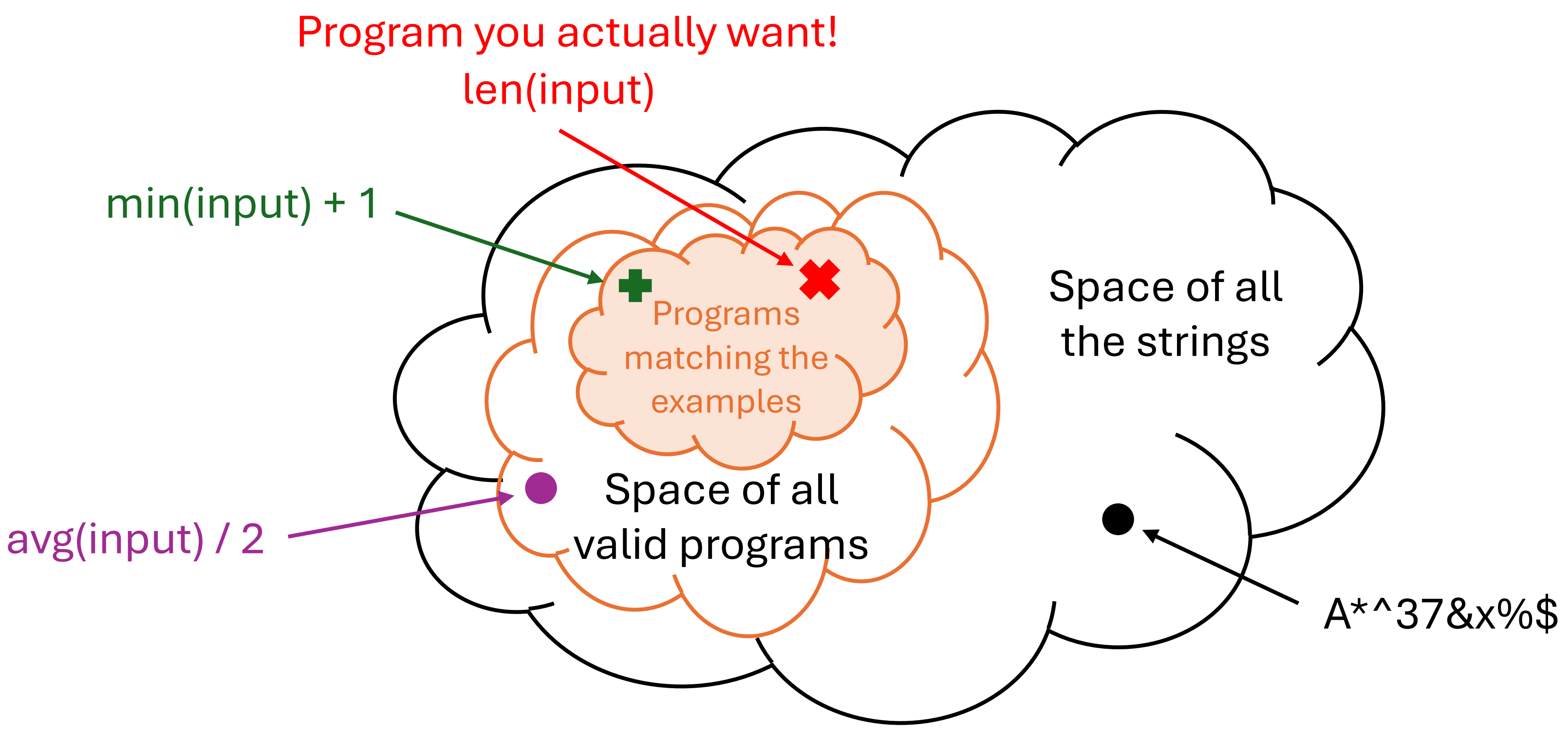
{[0] → 1, [5,1] → 2} ➔ len(input), min(input) + 1, …



{[0] → 1, [5,1] → 2} ➔ len(input), min(input) + 1, …



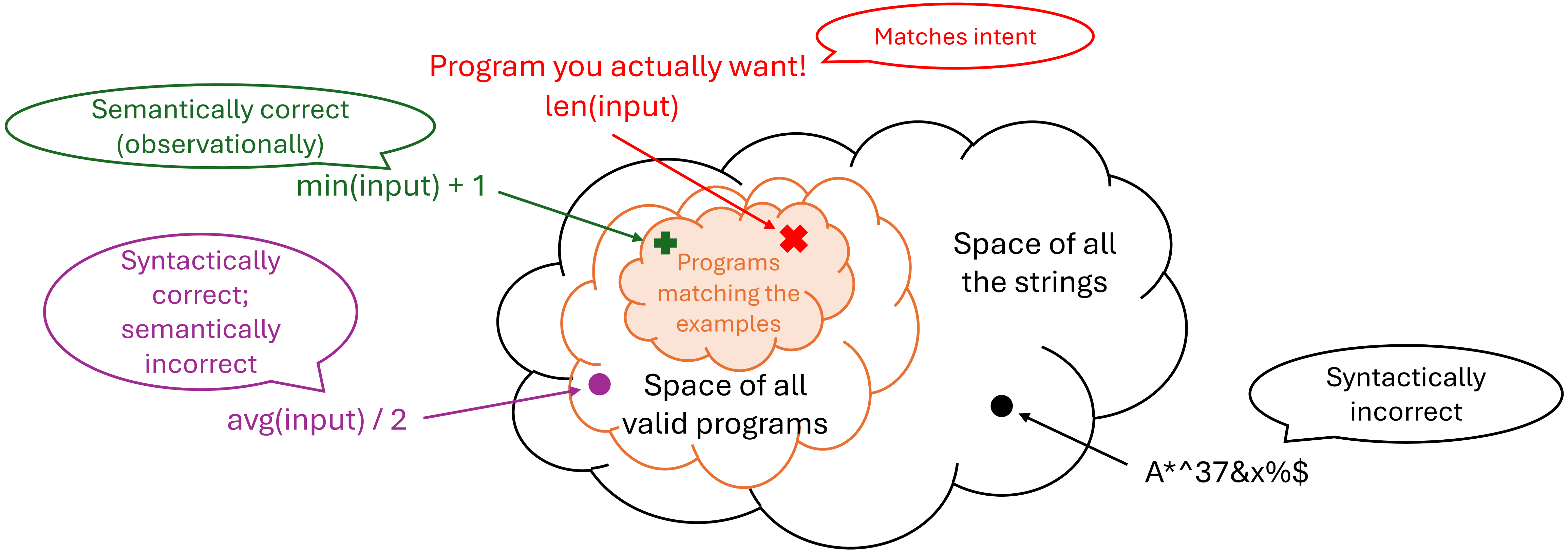
{[0] → 1, [5,1] → 2} ➔ len(input), min(input) + 1, ...

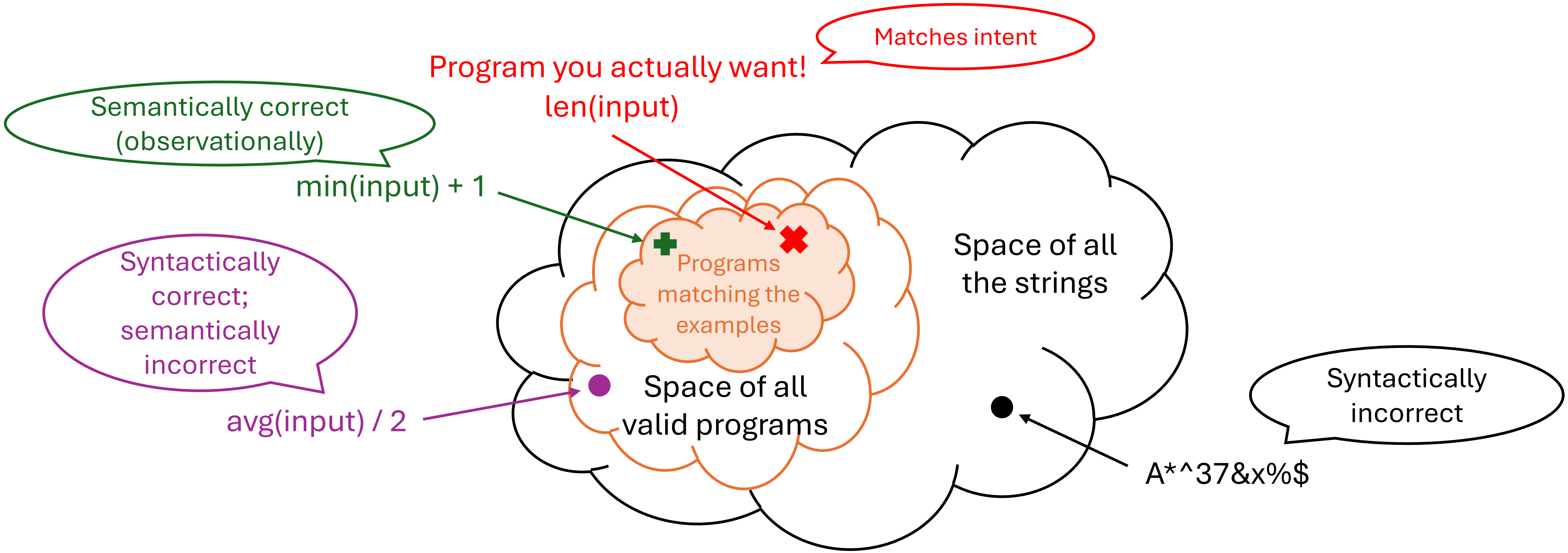


Behavioral Specification:

Examples

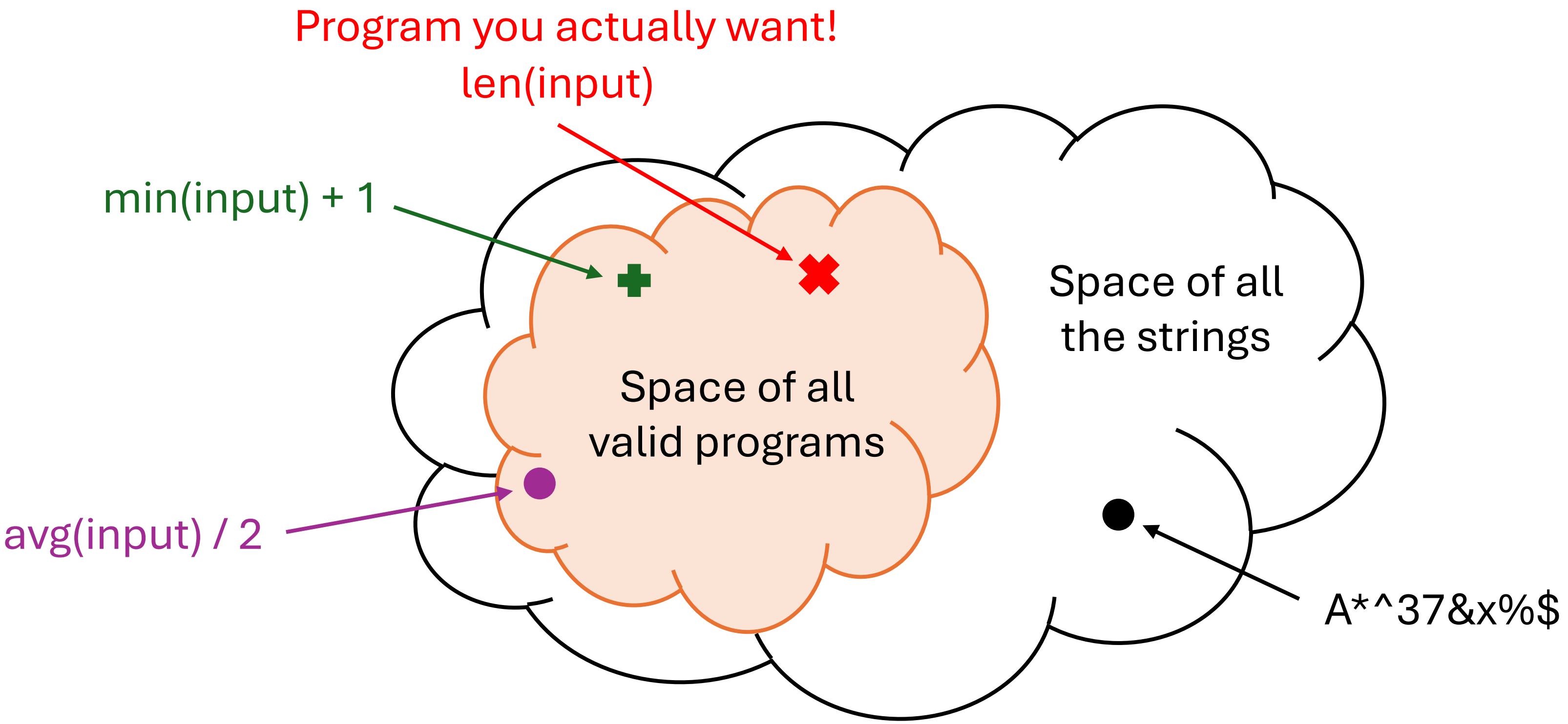
{[0] → 1, [5,1] → 2} ➔ len(input), min(input) + 1, …





Syntax + Semantics

Syntax


Program you actually want!
len(input)
min(input) + 1
Space of all
the strings
Space of all
valid programs
avg(input) / 2
A*^37&x%$

Syntax: Example

{[0] → 1, [5,1] → 2} ➔ len(input), min(input) + 1, …

```
(Program) P ::= L | N                 // either a list expr or number expr
   (List) L ::= input                 // the input list
              | empty                 // []
              | single(N)             // [N]
              | concat(L, L)          // concat([1],[2,3]) = [1,2,3]
 (Number) N ::= len(L)                // len([0]) = 1
              | min(L)                // min([1,2]) = 1
              | add(N, N)             // add(2,1) = 3
              | 0 | 1 | 2 | …         // the constant numerical literals
```

Syntax: Example

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Syntax: Regular tree grammars (RTGs)

Name of nonterminals

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Syntax: Regular tree grammars (RTGs)

non-terminals

Name of nonterminals

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Syntax: Regular tree grammars (RTGs)

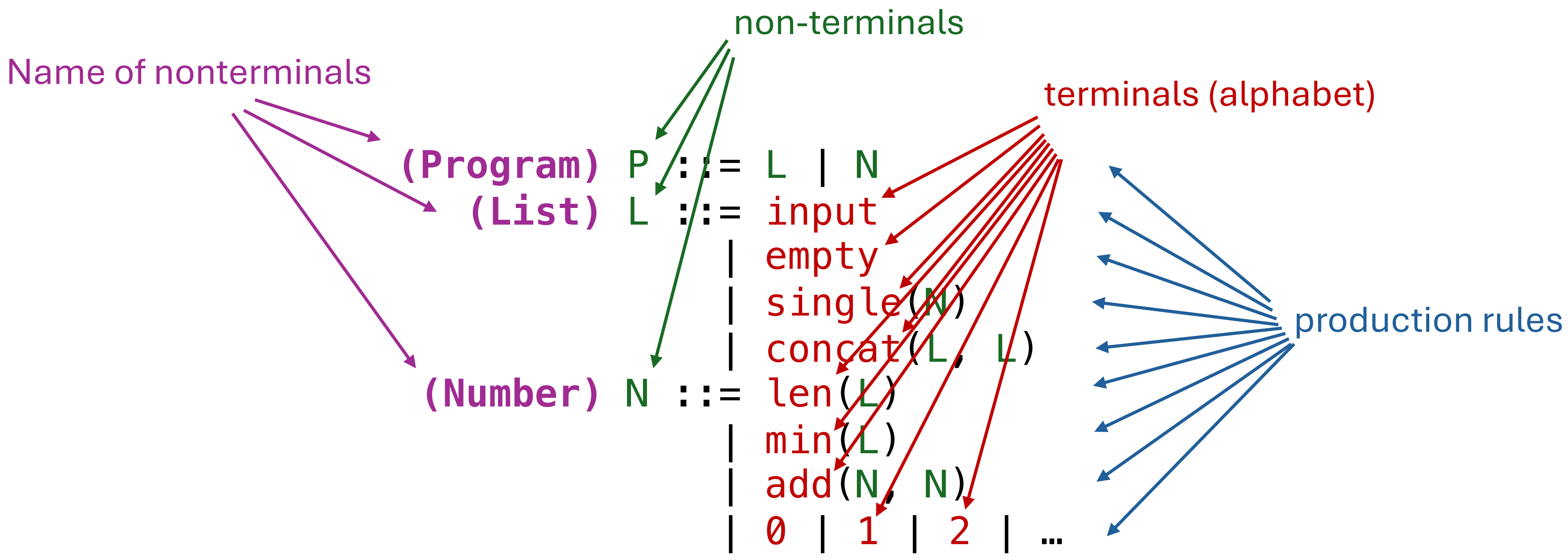
non-terminals

Name of nonterminals

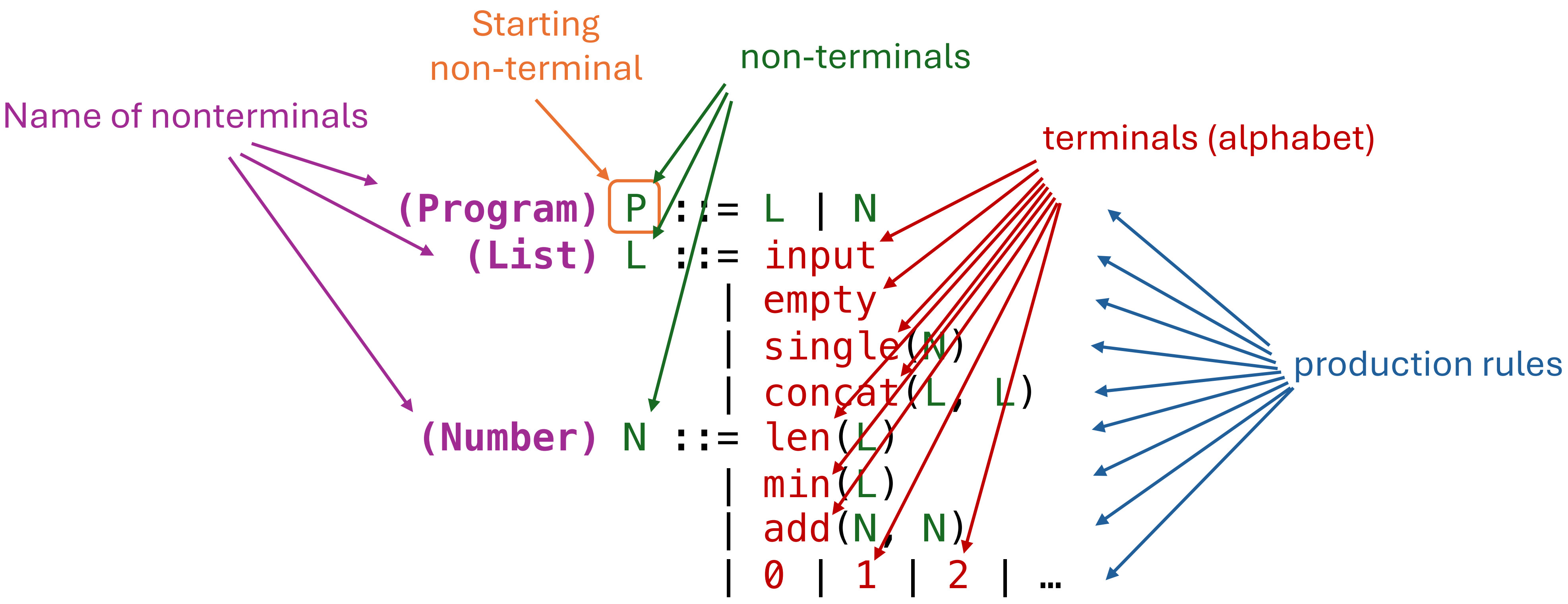
terminals (alphabet)

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

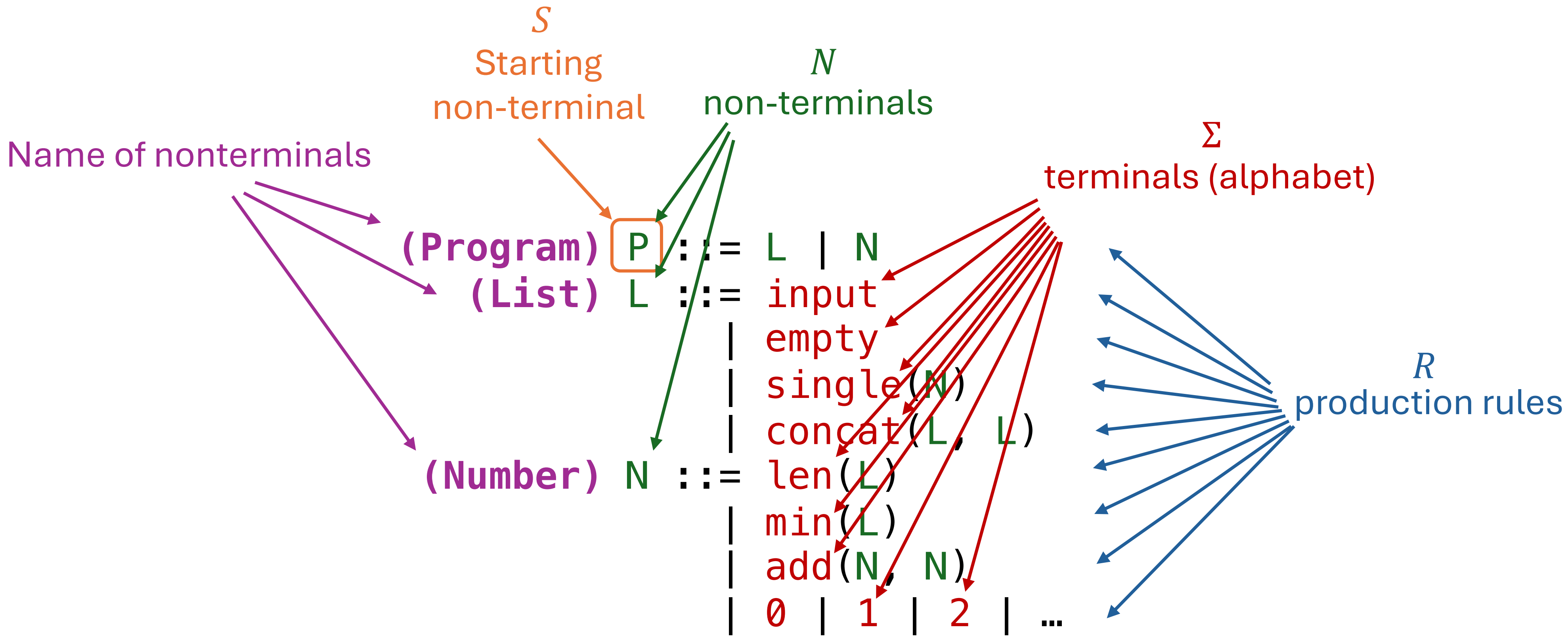
Syntax: Regular tree grammars (RTGs)



Syntax: Regular tree grammars (RTGs)



Syntax: Regular tree grammars (RTGs)



Syntax: Regular tree grammars (RTGs)

alphabet, non-terminals, production rules, Starting non-terminal

$< \Sigma , N , R , S >$

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Trees: $\tau \in T_\Sigma(N)$ = all trees made from $\Sigma \cup N$

Rules in R: $A \to \sigma(A_1, \dots, A_n)$ where $A \in N, A_i \in \Sigma \cup N$

Derivation in one step: $\to$

Derivations in multiple steps: $\to^*$

Incomplete Programs: a tree τ with non-terminals

- $\tau \in T_\Sigma(N)$ where $A \to^* \tau$

Complete Programs: a tree t without non-terminals

- $t \in T_\Sigma$ where $A \to^* t$

Whole Programs: a complete program t derivable by S

- $t \in T_\Sigma$ where $S \to^* t$

Syntax: Regular tree grammars (RTGs)

alphabet → Σ, non-terminals → N, production rules → R, Starting non-terminal → S

$$< \Sigma\ ,\ N\ ,\ R\ ,\ S >$$

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Trees: $\tau \in T_\Sigma(N)$ = all trees made from $\Sigma \cup N$

Rules in R: $A \to \sigma(A_1, \ldots, A_n)$ where $A \in N, A_i \in \Sigma \cup N$

Derivation in one step: $\to$

Derivations in multiple steps: $\to^*$

Incomplete Programs: a tree τ with non-terminals

- $\tau \in T_\Sigma(N)$ where $A \to^* \tau$

Complete Programs: a tree t without non-terminals

- $t \in T_\Sigma$ where $A \to^* t$

Whole Programs: a complete program t derivable by S

- $t \in T_\Sigma$ where $S \to^* t$

```
concat(L, 0)
L → concat(L, L)
concat(L,L) → concat(input, L)
L →* single(len(L))
len(concat(L, L))

len(concat(input,single(1)))

len(input)
```

Syntax: Regular tree grammars (RTGs)

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Space of programs

= the language of a Regular tree grammar

= all complete & whole programs

Syntax: How big is the space of programs?

```
E ::= x | f(E, E)
```

Syntax: How big is the space of programs?

```
E ::= x | f(E, E)
```

Depth <= 0   (x)   `Size(0) = 1`

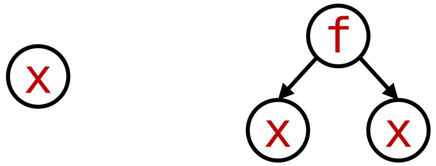
Syntax: How big is the space of programs?

`E ::= x | f(E, E)`

Depth <= 0

`Size(0) = 1`

Depth <= 1



`Size(1) = 2`

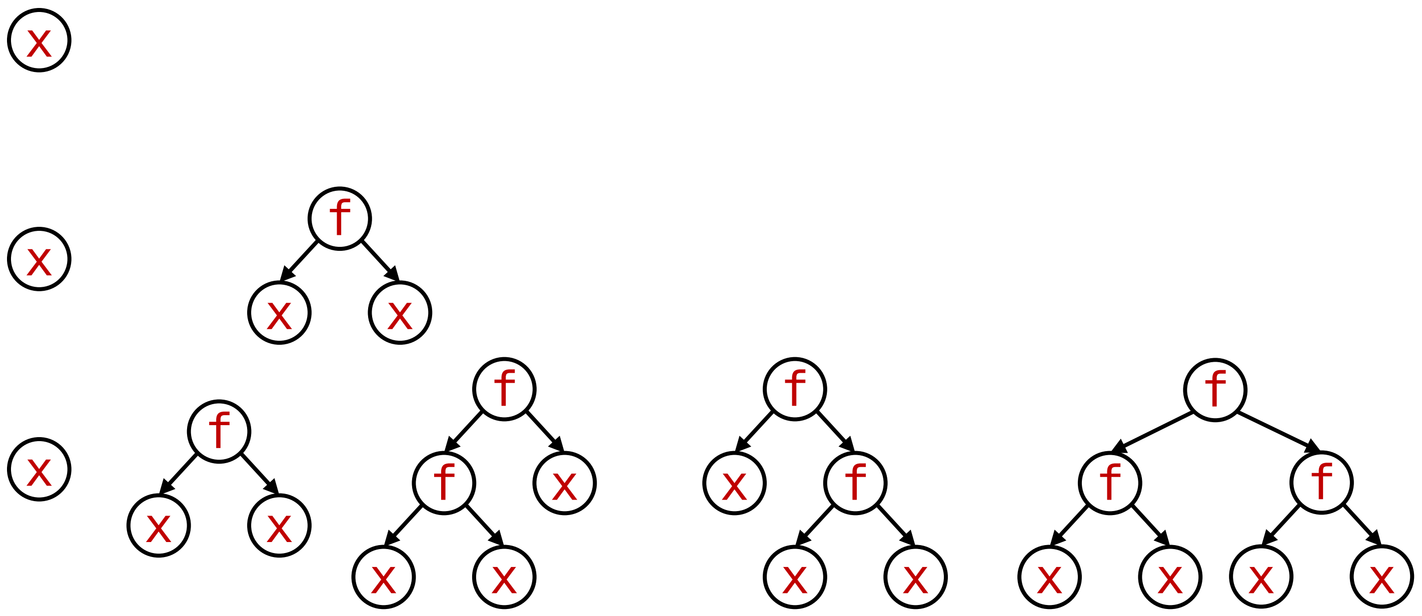
Syntax: How big is the space of programs?

```
E ::= x | f(E, E)
```

Depth <= 0 — Size(0) = 1

Depth <= 1 — Size(1) = 2

Depth <= 2 — Size(2) = 5



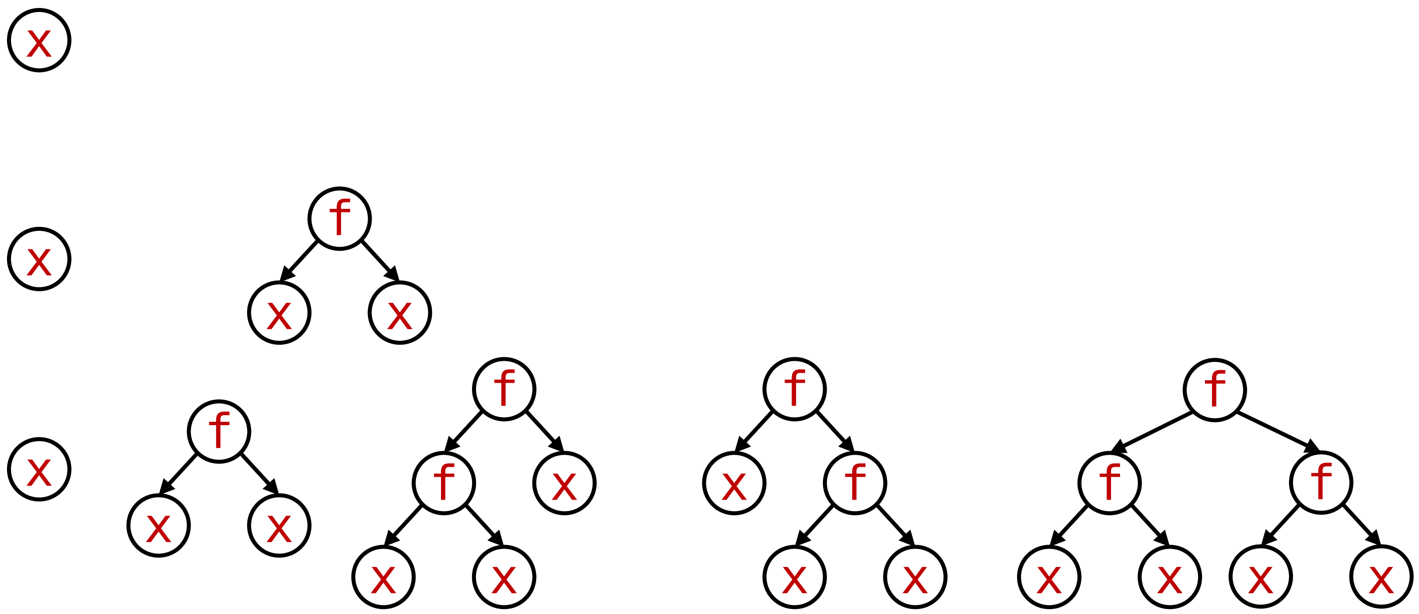
Syntax: How big is the space of programs?

E ::= x | f(E, E)

Depth <= 0 — Size(0) = 1

Depth <= 1 — Size(1) = 2

Depth <= 2 — Size(2) = 5



Size(depth) = ???

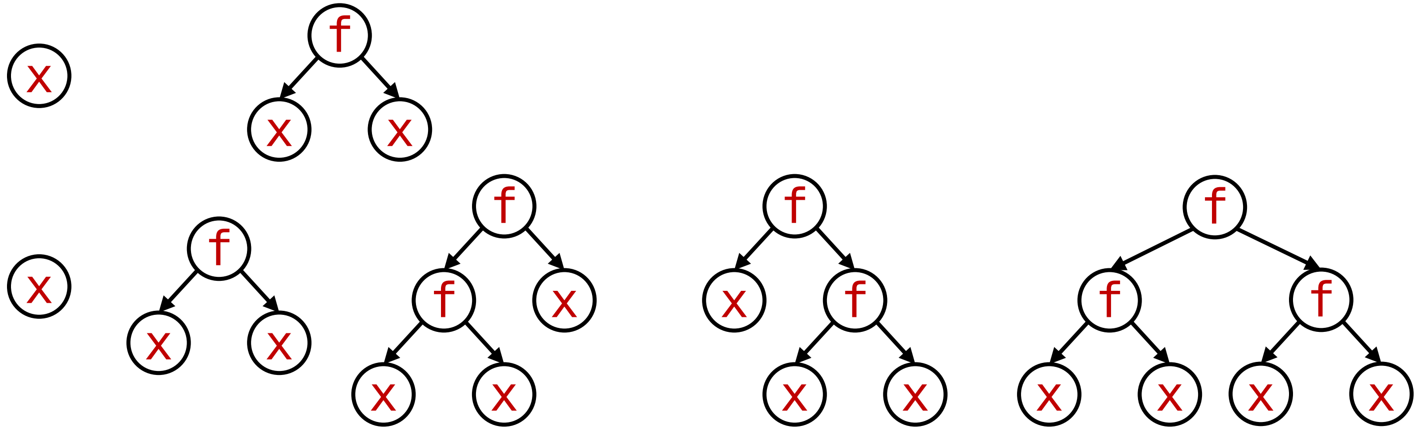
Syntax: How big is the space of programs?

`E ::= x | f(E, E)`

Depth <= 0 — `Size(0) = 1`

Depth <= 1 — `Size(1) = 2`

Depth <= 2 — `Size(2) = 5`



`Size(depth) = 1 + Size(depth – 1)`^{2}

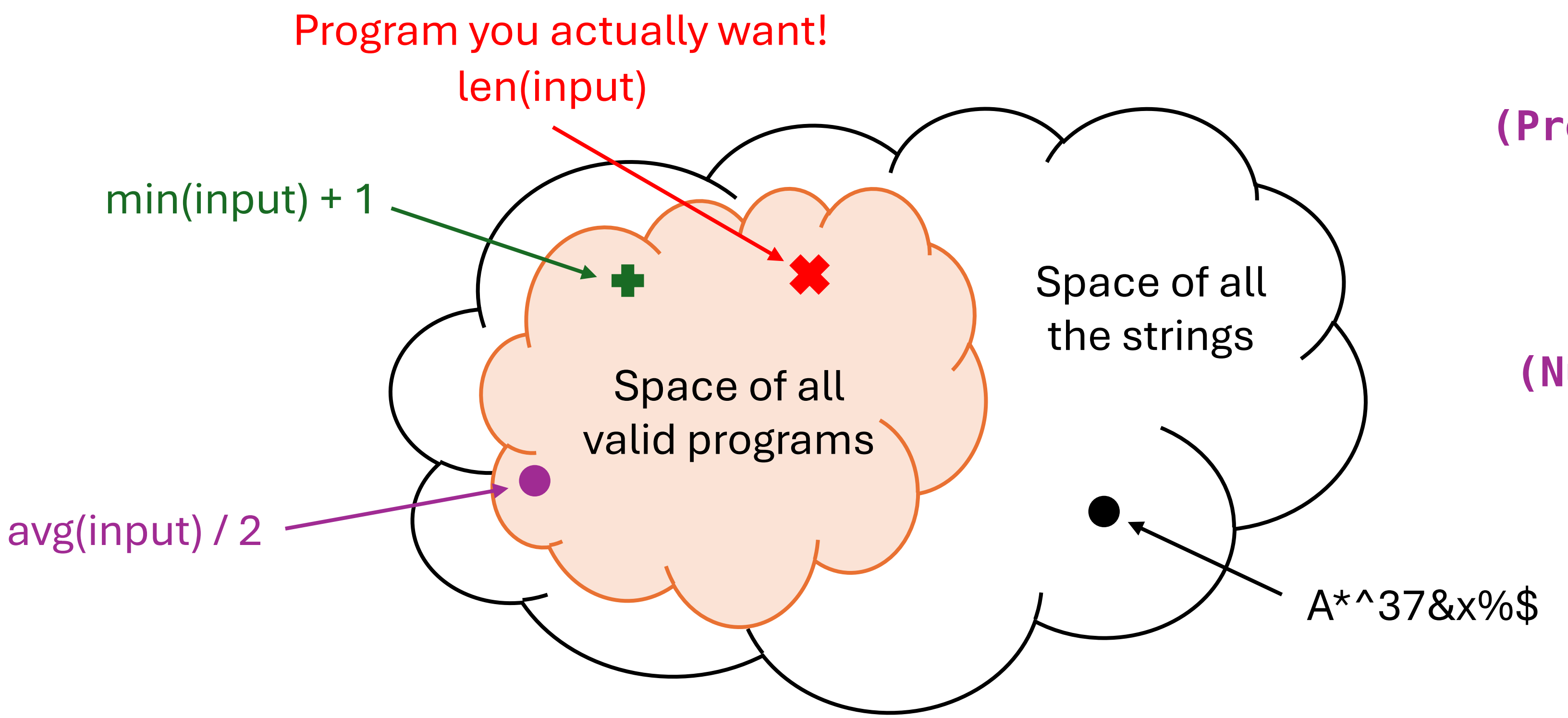
Syntax: How big is the space of programs?

```
E ::= x | f(E, E)
```

```
Size(depth) = 1 + Size(depth – 1)²
```

```
size(1) = 1
size(2) = 2
size(3) = 5
size(4) = 26
size(5) = 677
size(6) = 458330
size(7) = 210066388901
size(8) = 44127887745906175987802
size(9) = 1947270476915296449559703445493848930452791205
size(10) = 3791862310265926082868235028027893277370233152247388584761734150717768254410341175325352026
```

Syntax: Sugars




```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Syntax: Sugars

min(input) + 1

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

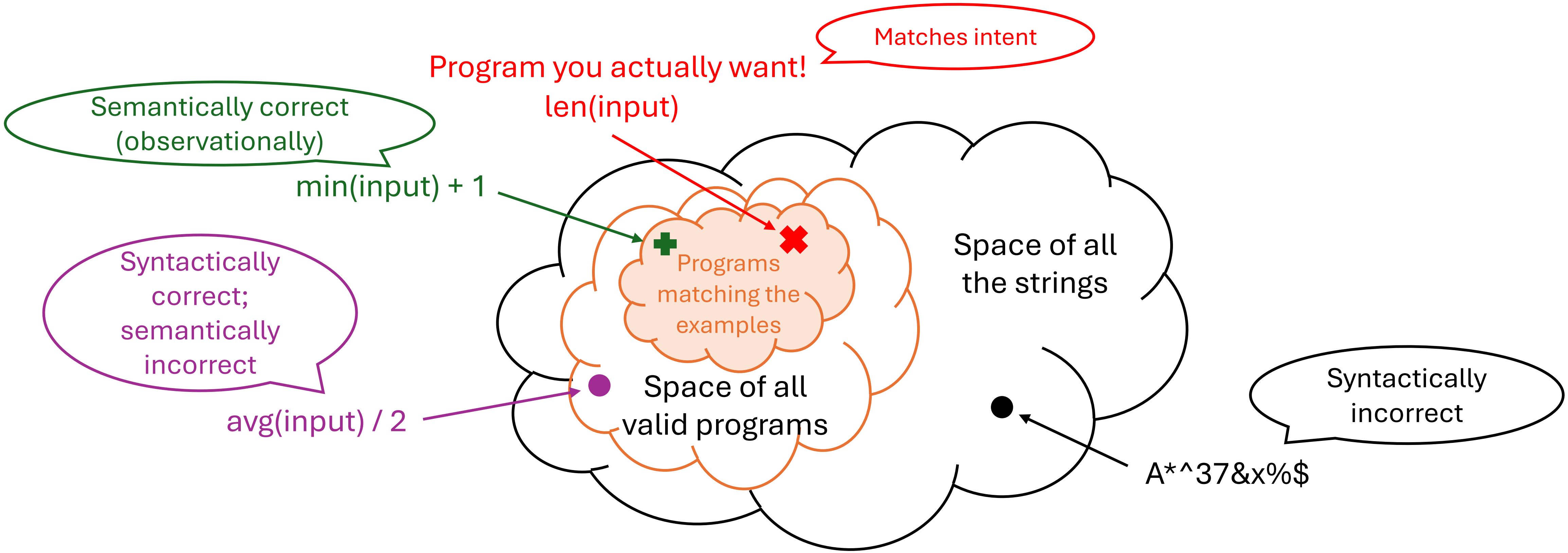
```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | [N]
              | L :: L
 (Number) N ::= len(L)
              | min(L)
              | N + N
              | 0 | 1 | 2 | …
```

Syntax: Sugars

min(input) + 1

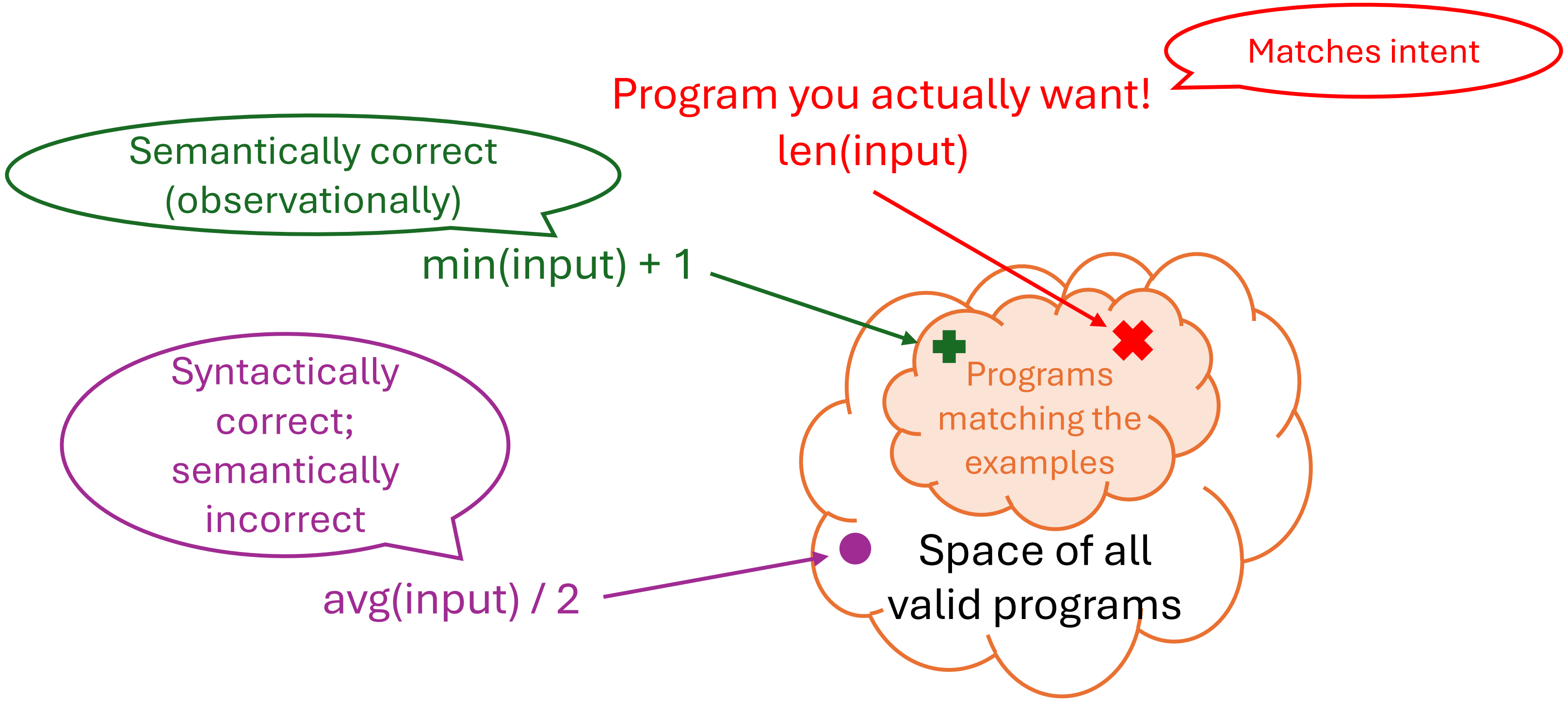
```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

```
(Program) P ::= L | N
   (List) L ::= input
              | []
              | [N]
              | L :: L
 (Number) N ::= len(L)
              | min(L)
              | N + N
              | 0 | 1 | 2 | …
```



Syntax + Semantics

Semantics


Matches intent
Program you actually want!
len(input)
Semantically correct (observationally)
min(input) + 1
Syntactically correct; semantically incorrect
avg(input) / 2
Programs matching the examples
Space of all valid programs

Semantics: Meaning of a Language

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

$$eval : T_\Sigma \times IntList \rightarrow List \mid Int$$
$$eval(\grave{}\text{input}\grave{}, x) = x$$
$$eval(\grave{}\text{empty}\grave{}, x) = []$$
$$\forall \tau \in T_\Sigma, eval(\grave{}\text{single}(\tau)\grave{}, x) = [eval(\tau, x)]$$
$$\forall \tau_1, \tau_2 \in T_\Sigma, eval(\grave{}\text{concat}(\tau_1, \tau_2)\grave{}, x) = eval(\tau_1, x) + eval(\tau_2, x)$$
$$\forall \tau \in T_\Sigma, eval(\grave{}\text{len}(\tau)\grave{}, x) = |eval(\tau, x)|$$
$$\forall \tau \in T_\Sigma, eval(\grave{}\min(\tau)\grave{}, x) = \min_v(v \in eval(\tau, x))$$
$$\forall \tau_1, \tau_2 \in T_\Sigma, eval(\grave{}\text{add}(\tau_1, \tau_2)\grave{}, x) = eval(\tau_1, x) + eval(\tau_2, x)$$
$$eval(\grave{}0\grave{}, x) = 0, \ldots$$

Denotational Semantics

Mathematical meaning to each program construct

Semantics: Meaning of a Language

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

$$[[.]] : T_\Sigma \times IntList \rightarrow List \mid Int$$
$$[[\text{input}]](x) = x$$
$$[[\text{empty}]](x) = []$$
$$\forall \tau \in T_\Sigma, [[\text{single}(\tau)]](x) = [\ [[\tau]](x)]$$
$$\forall \tau_1, \tau_2 \in T_\Sigma, [[\text{concat}(\tau_1, \tau_2)]](x) = [[\tau_1]](x) + [[\tau_2]](x)$$
$$\forall \tau \in T_\Sigma, [[\text{len}(\tau)]](x) = |[[\tau]](x)|$$
$$\forall \tau \in T_\Sigma, [[\text{min}(\tau)]](x) = \min_v \left(v \in [[\tau]](x)\right)$$
$$\forall \tau_1, \tau_2 \in T_\Sigma, [[\text{add}(\tau_1, \tau_2)]](x) = [[\tau_1]](x) + [[\tau_2]](x)$$
$$[[0]](x) = 0, \ldots$$

Denotational Semantics

Mathematical meaning to each program construct

Semantics: Meaning of a Language

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

$$\frac{}{x \vdash \texttt{input} \Downarrow x} \qquad \frac{}{x \vdash \texttt{empty} \Downarrow []}$$

$$\frac{x \vdash N \Downarrow n}{x \vdash \texttt{single}(N) \Downarrow [n]} \qquad \frac{x \vdash L1 \Downarrow vL1 \quad x \vdash L2 \Downarrow vL2}{x \vdash \texttt{concat}(L1, L2) \Downarrow vL1 \mathbin{++} vL2}$$

…

Operational Semantics

Meaning in terms of computation steps

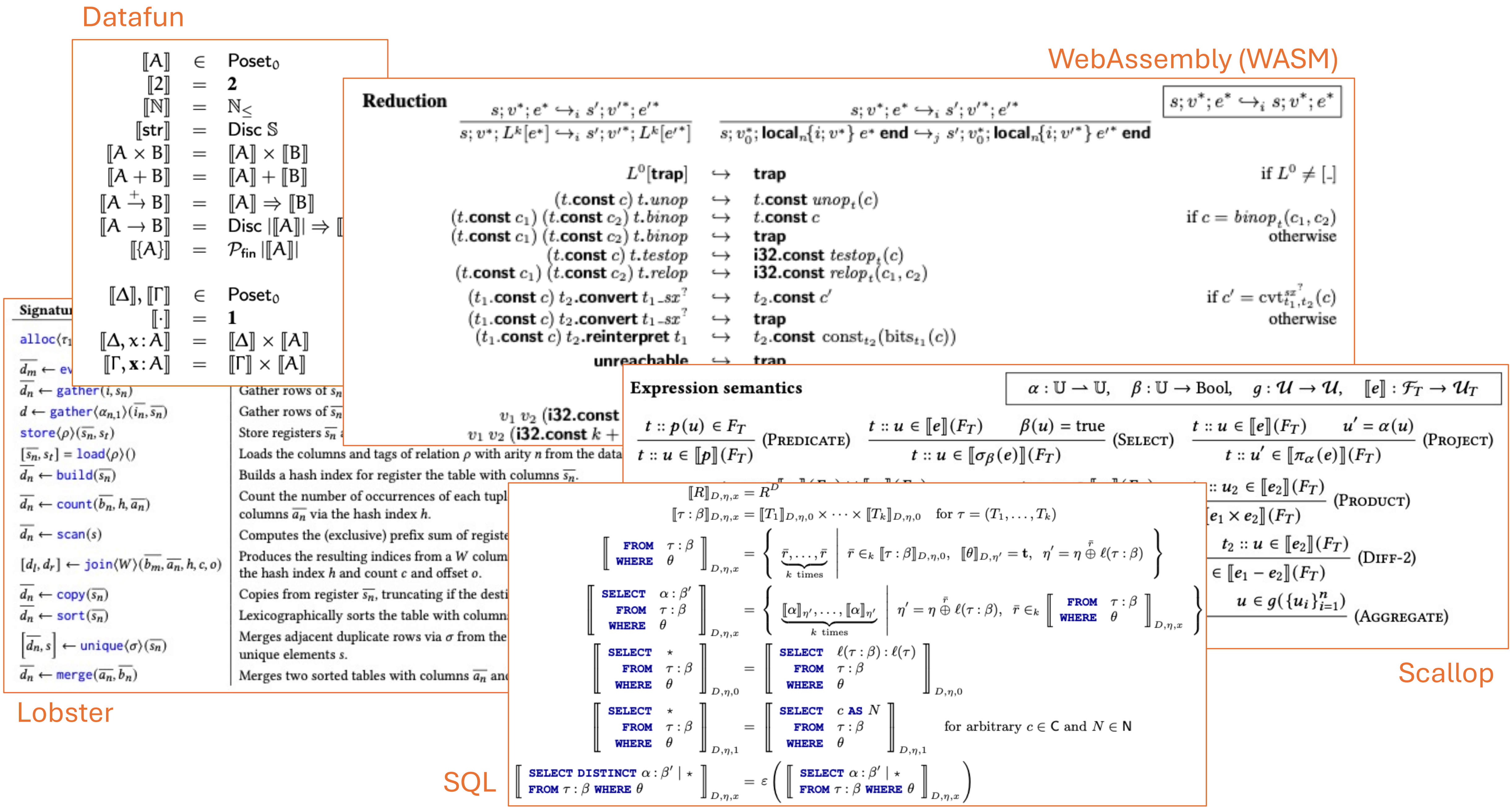
Semantics: Meaning of a Language

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

```
def evaluate(program, input):
  if instanceof(program, Empty):
    return []
  elif instanceof(program, Input):
    return input
  elif instanceof(program, Concat):
    left = evaluate(program.left, input)
    right = evaluate(program.right, input)
    return left + right
  elif ...
```

Semantics written in Python…

Meaning encoded as an evaluator / interpreter


Datafun
WebAssembly (WASM)
Reduction
Expression semantics
Lobster
Scallop
SQL

Grounded with concrete inputs...

Syntax

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

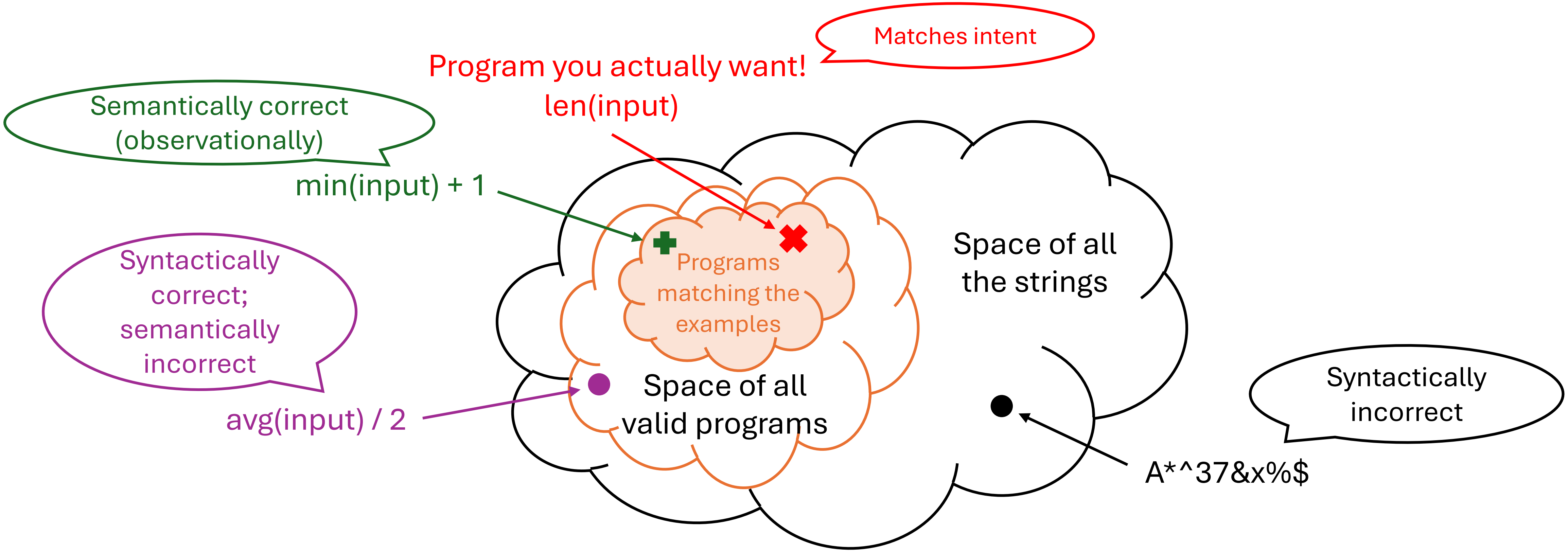
Semantics

```
def evaluate(program, input):
  if instanceof(program, Empty):
    return []
  elif instanceof(program, Input):
    return input
  elif instanceof(program, Concat):
    left = evaluate(program.left, input)
    right = evaluate(program.right, input)
    return left + right
  elif ...
```

Examples

```
[0, 1]    -> 3
[1]       -> 2
[3, 5, 4] -> 4
```

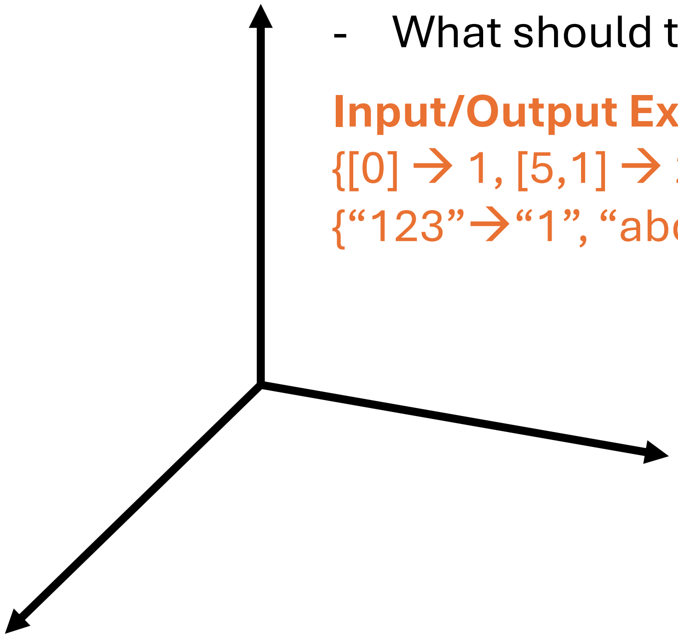
```
evaluate(`len(concat(single(3),input))`, [0, 1]) ➔ len([3, 0, 1]) = 3
└─> evaluate(`concat(single(3),input)`, [0, 1]) ➔ [3] + [0, 1] = [3,0,1]
    ├─> evaluate(`single(3)`, [0, 1]) ➔ [3]
    └─> evaluate(`input`, [0, 1]) ➔ [0, 1]
```


Matches intent
Program you actually want!
len(input)
Semantically correct (observationally)
min(input) + 1
Syntactically correct; semantically incorrect
avg(input) / 2
Programs matching the examples
Space of all valid programs
Space of all the strings
Syntactically incorrect
A*^37&x%$

Today

Behavioral Specification
- What should the program do?

Input/Output Examples
{[0] → 1, [5,1] → 2}
{“123”→“1”, “abc”→“a”}



Synthesis Strategy
- How do we find such a program?

Enumeration
- Enumerating all programs with a grammar
- Bottom-up vs top-down

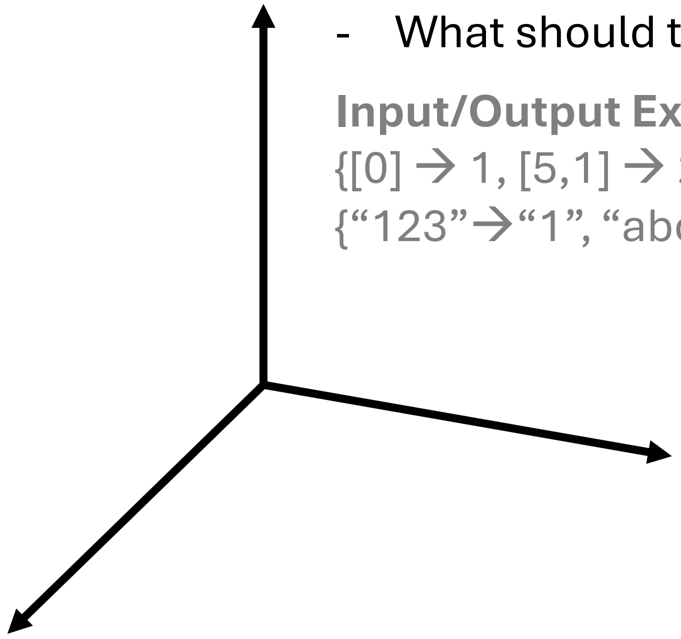
Structural Specification
- What is the space of the programs?

Context-Free / Regular Tree Grammar
Expr e ::= c | e + e | e * e

Today

Behavioral Specification
- What should the program do?

Input/Output Examples
{[0] → 1, [5,1] → 2}
{“123”→“1”, “abc”→“a”}



Synthesis Strategy
- How do we find such a program?

Enumeration
- Enumerating all programs with a grammar
- Bottom-up vs top-down

Structural Specification
- What is the space of the programs?

Context-Free / Regular Tree Grammar
Expr e ::= c | e + e | e * e

Enumerative search

- Idea: enumerate programs from the grammar one by one and test them on the examples
- Challenge: How do we systematically enumerate all programs?
  - Bottom-up
  - Top-down

Bottom-up enumeration

- Maintain a bank of complete programs
  - Starting from all the terminal symbols
- Combine programs in the bank using production rules
  - Applying all possible production rules at each iteration

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Bottom-up enumeration: algorithm

```
bottom-up(<Σ, N, R, S>, [i → o], max_depth):
  bank := {}
  for depth in [0..max_depth]:
    forall rule in R:
      forall new_prog in grow(rule, depth, bank):
        if (A = S ∧ new_prog([i]) = [o]):
          return new_program
        insert new_program to bank;

grow(A → σ(A1…Ak), d, bank):
  if (d = 0 ∧ k = 0) yield σ // terminal
  else forall <t1,…,tk> in bank^k: // cartesian product
    if Ai ->* ti:
      yield σ(t1,…,tk)
```

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

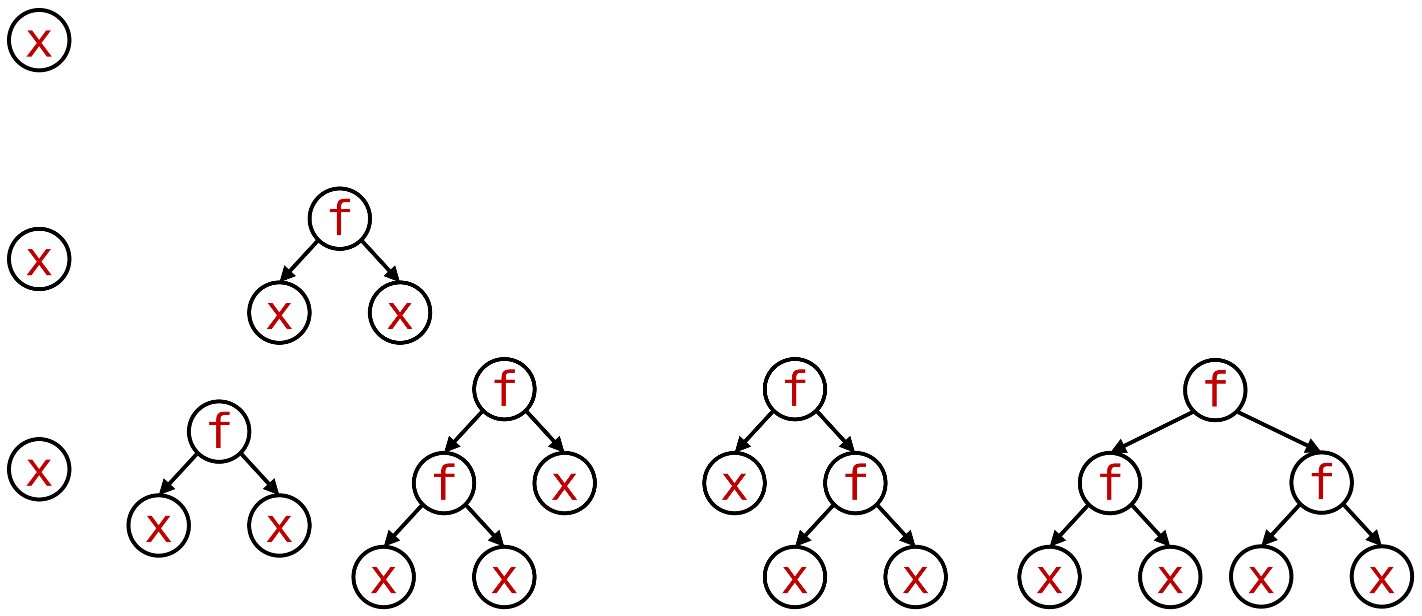
Bottom-up enumeration: example

```
E ::= x | f(E, E)
```

Depth <= 0

Depth <= 1

Depth <= 2



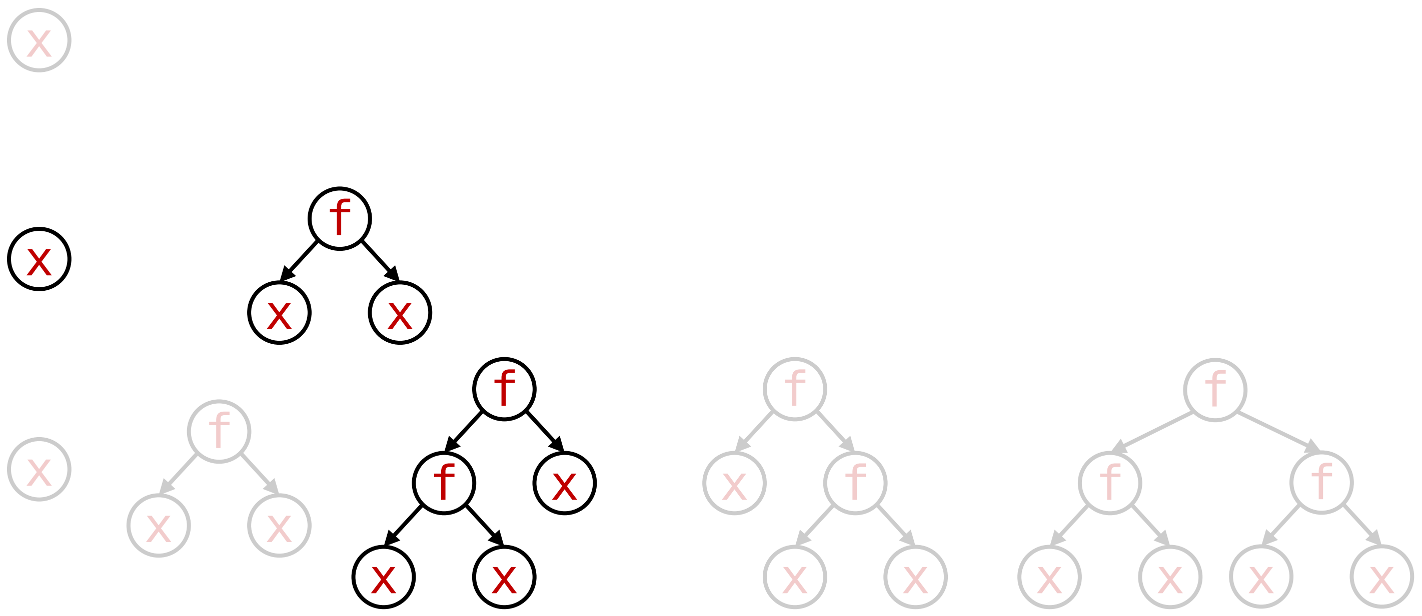
Bottom-up enumeration: example

```
E ::= x | f(E, E)
```

Depth <= 0

Depth <= 1

Depth <= 2



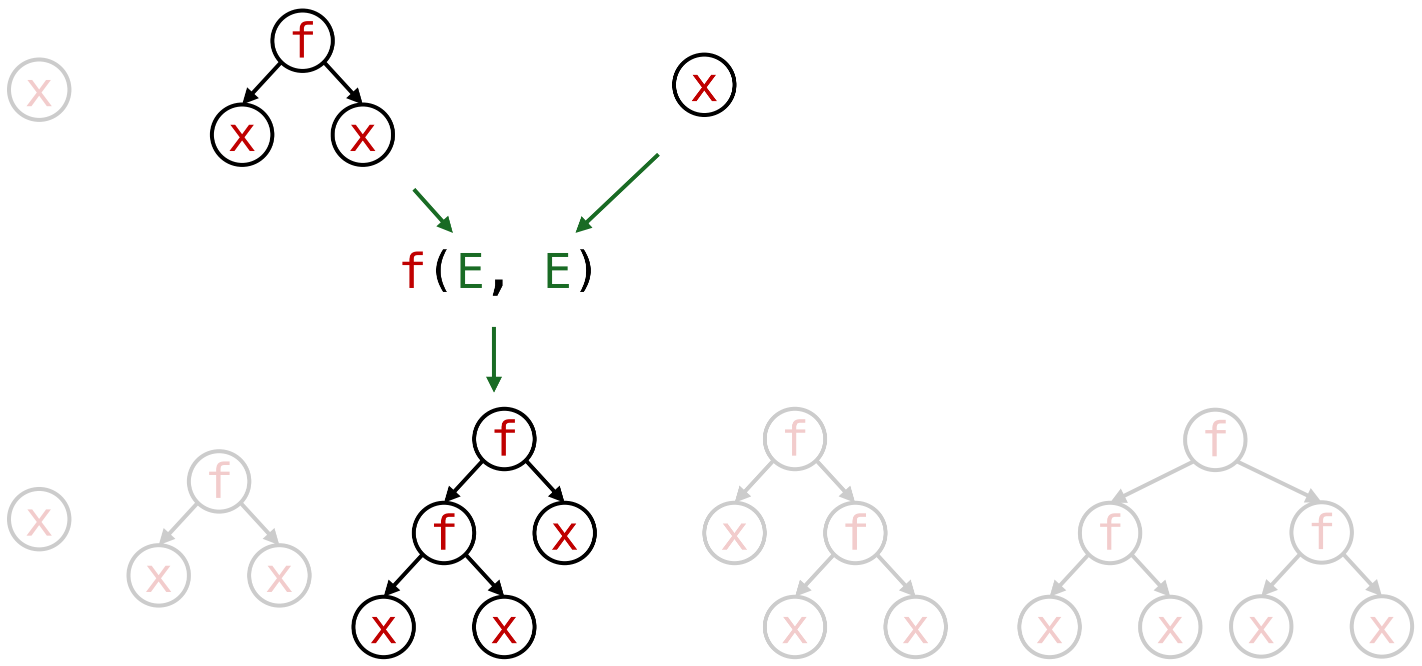
Bottom-up enumeration: example

`E ::= x | f(E, E)`

Depth <= 0

Depth <= 1

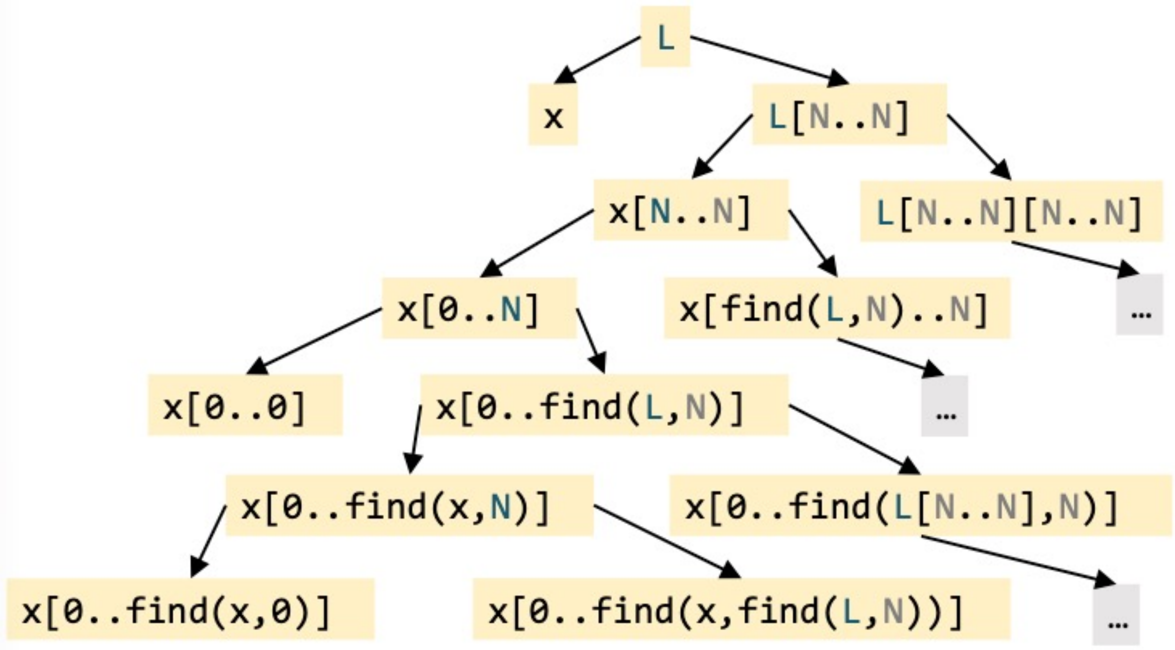
Depth <= 2



Top-down enumeration

- Search space is a tree, where:
  - Nodes are whole incomplete programs
  - Edges are derivations in a single step

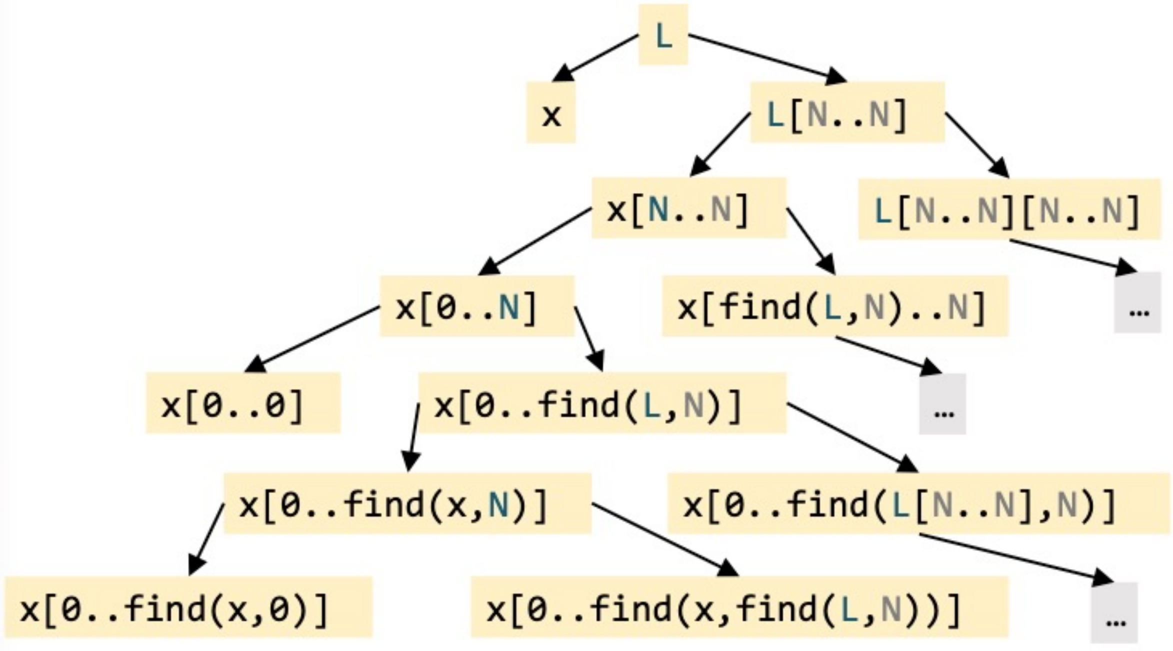
```
(List) L ::= L[N:N]
           | x // input
(Number) N ::= find(L, N)
             | 0
```



Top-down enumeration

- Search tree can be traversed
  - Depth-first
  - Breadth-first
  - …

```
(List) L ::= L[N:N]
           | x // input
(Number) N ::= find(L, N)
           | 0
```



Bottom-up vs top-down

Top-down

Program candidates are whole but might not be complete

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

Bottom-up

Program 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

Prioritize

- Explore more promising candidates

Summary

- Syntax
- Semantics
- Enumerative algorithms
  - Bottom-up
  - Top-down

Week 1

- Assignment 1
  - Released: https://github.com/machine-programming/assignment-1
  - Autograder will be on GradeScope later today
  - API keys will be sent out later today
- Waitlisted students
  - Please contact me by sending emails; will add you to Courselore, GradeScope, and give you API keys
- Any questions?