# Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions

Woosuk Lee, Hangyeol Cho

Jaehoon Jang

2024.05.09

IS661

# Background

Program Synthesis?

- Creates a program that users want automatically
- If you give a requirement, create a program
    - satisfies this requirement

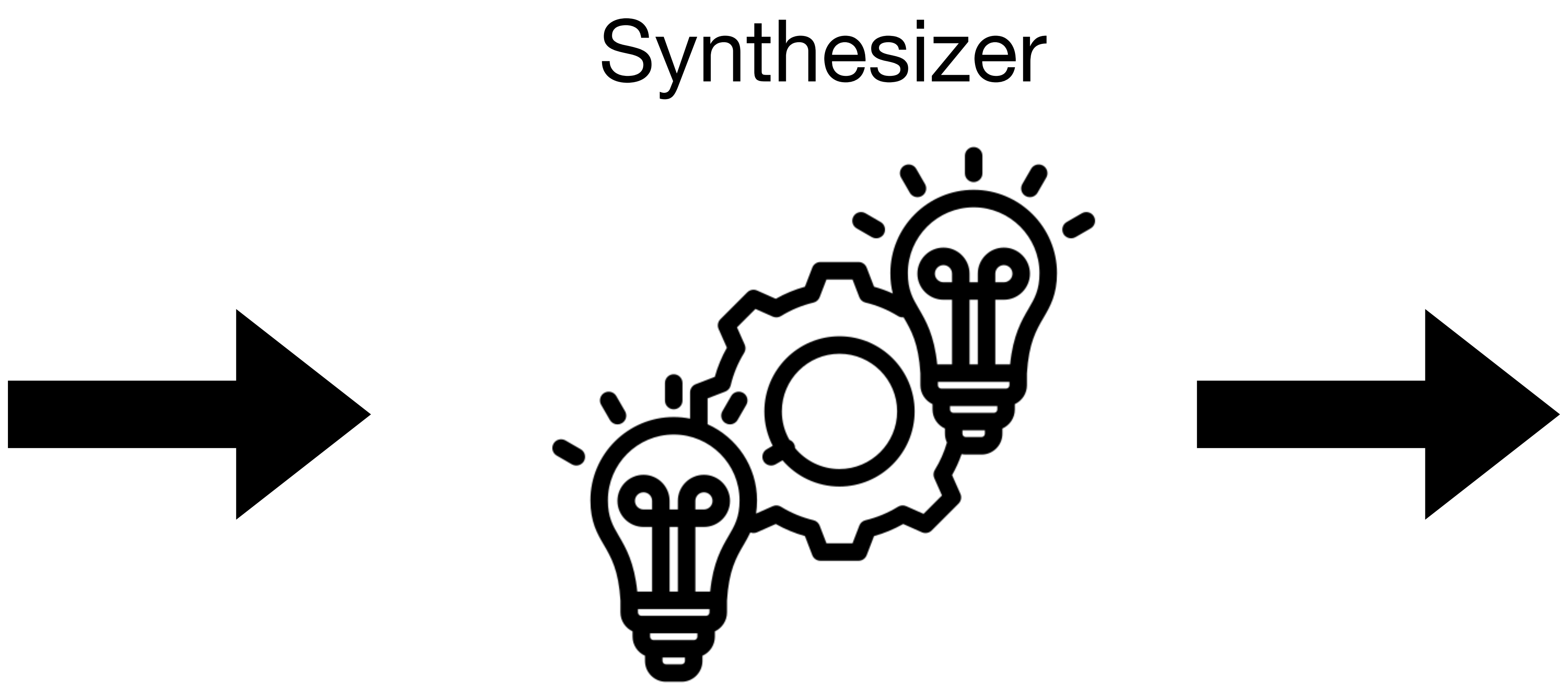
# Motivation

Recursive Functional Program Synthesis?

```
type nat = Z
         | S -> nat
```

```
f : nat -> nat

Z → Z
S(Z) → S(S(Z))
```



```
let rec f x =
  match x with
  | Z -> Z
  | S n -> (f n) + S(S(Z))
```

```
add:
nat * nat -> nat
```

# Motivation

Recursive Functional Program Synthesis?

Input 1) custom data type

```
type nat = Z
         | S -> nat
```

```
f : nat -> nat

Z → Z
S(Z) → S(S(Z))
```

Synthesizer

```
let rec f x =
  match x with
  | Z -> Z
  | S n -> (f n) + S(S(Z))
```

```
add:
nat * nat -> nat
```

# Motivation

Recursive Functional Program Synthesis?

```
type nat = Z
         | S -> nat
```

Input 1) custom data type

```
f : nat -> nat

Z → Z
S(Z) → S(S(Z))
```

Input 2) target function type, I/O examples

```
add:
nat * nat -> nat
```

Synthesizer

```
let rec f x =
  match x with
  | Z -> Z
  | S n -> (f n) + S(S(Z))
```

# Motivation

Recursive Functional Program Synthesis?

```
type nat = Z
         | S -> nat
```

Input 1) custom data type

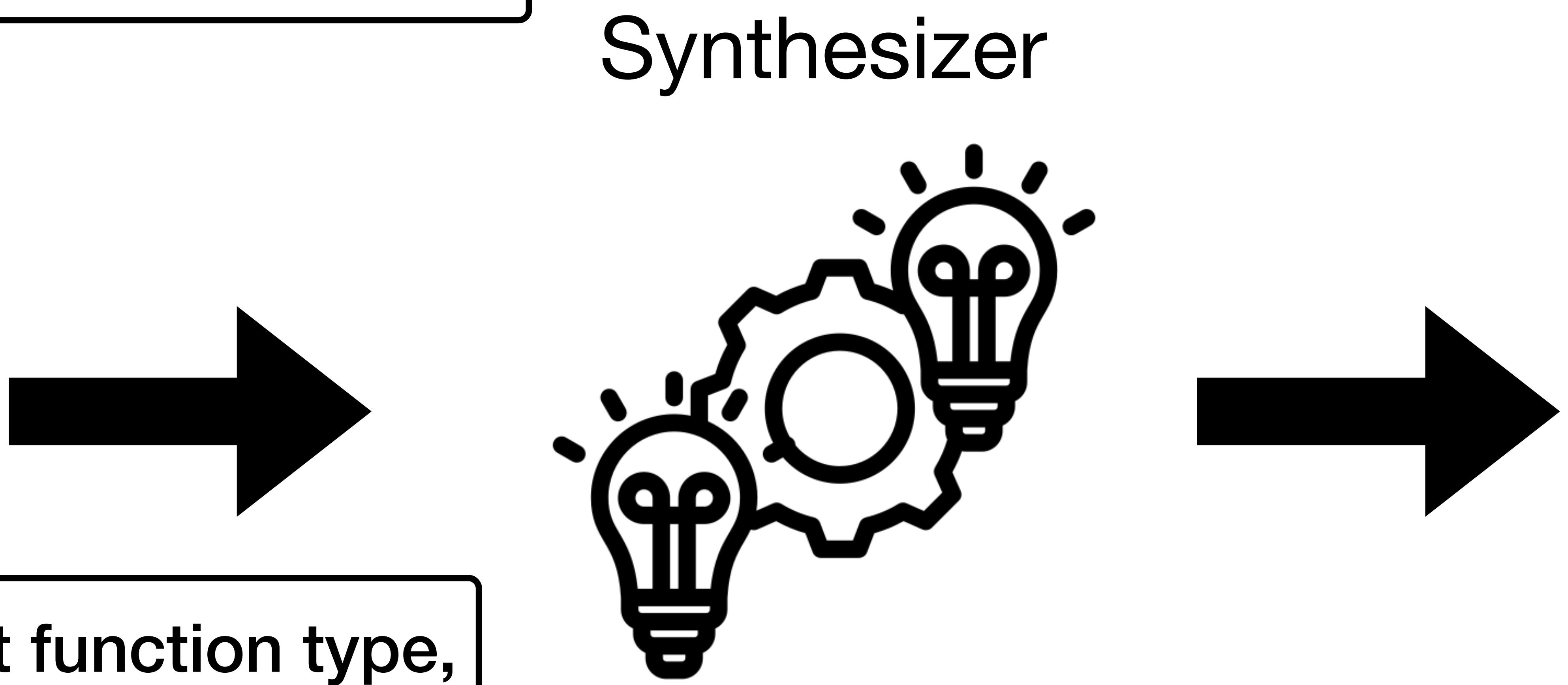
```
f : nat -> nat

Z → Z
S(Z) → S(S(Z))
```

Input 2) target function type, I/O examples

```
add:
nat * nat -> nat
```

Input 3) library of external operators


Synthesizer

```
let rec f x =
  match x with
  | Z -> Z
  | S n -> (f n) + S(S(Z))
```

# Motivation

## Recursive Functional Program Synthesis?

```
type nat = Z
         | S -> nat
```

Input 1) custom data type

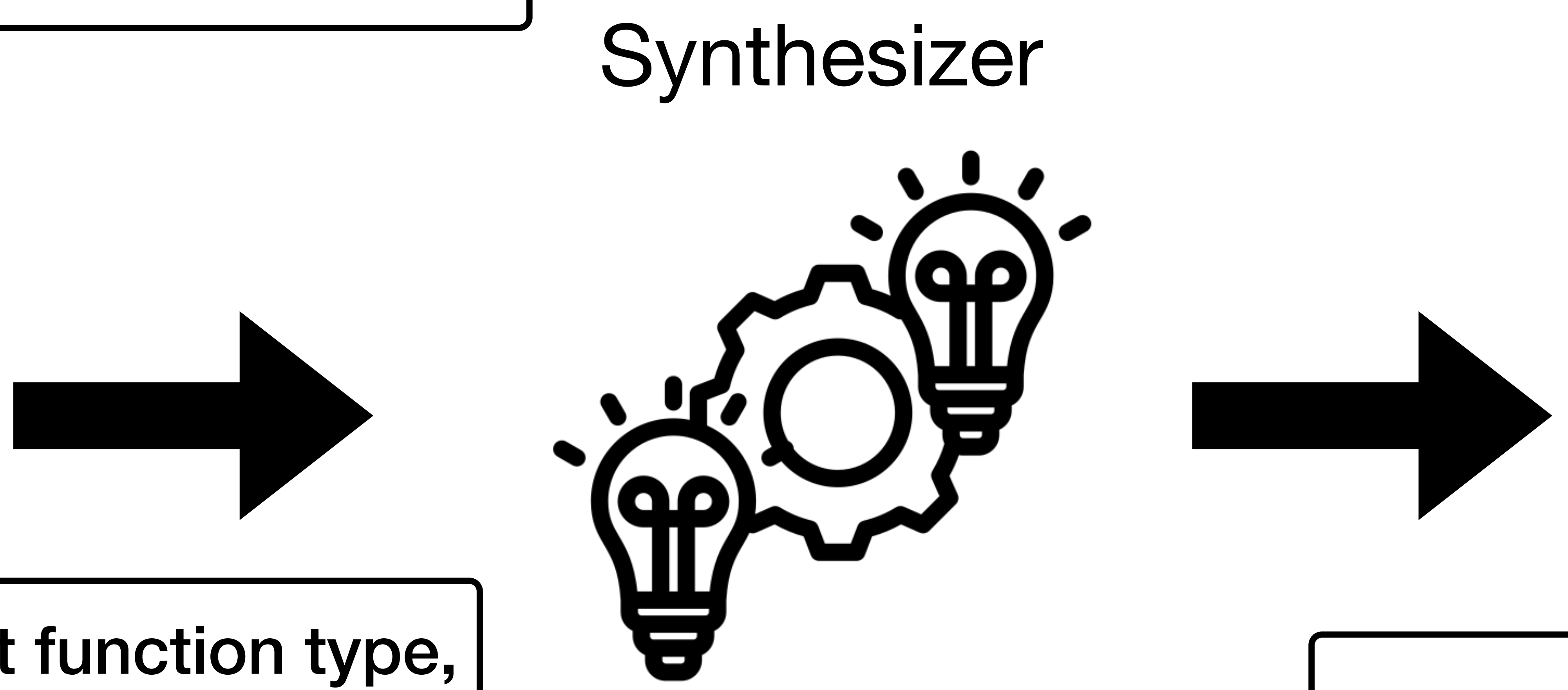
```
f : nat -> nat

Z → Z
S(Z) → S(S(Z))
```

Input 2) target function type, I/O examples

```
add:
nat * nat -> nat
```

Input 3) library of external operators



```
let rec f x =
  match x with
  | Z -> Z
  | S n -> (f n) + S(S(Z))
```

Output)  Recursive functional program

# Problem

- Synthesizing recursive functional program is hard

# Problem

- Synthesizing recursive functional program is hard
- One of the main reasons is...
    - Because it has to be evaluated for an **undefined function**

# Problem

- Synthesizing recursive functional program is hard
- One of the main reasons is...
    - Because it has to be evaluated for an **<u>undefined function</u>**
- There is two methods to synthesizing program
    - Top-down
    - Bottom-up

# Problem

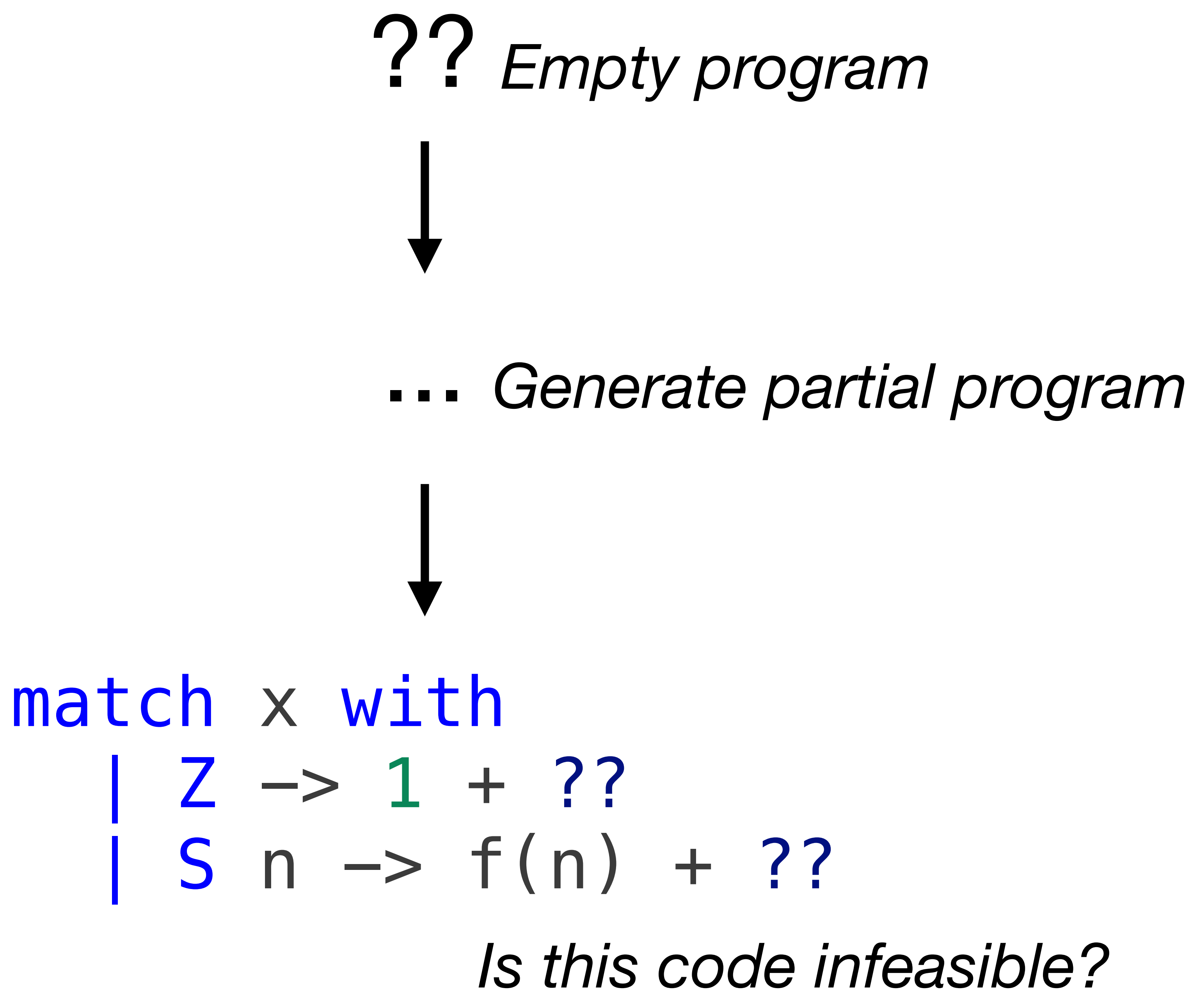
- Top-down
    - Start from empty program, generate partial program
    - Prune infeasible partial programs

# Problem

- Top-down
  - Start from empty program, generate partial program
  - Prune infeasible partial programs

```
let rec f x = ??
```

I/O examples: (0 → 1, 1 → 2)



?? *Empty program*

... *Generate partial program*

```
match x with
  | Z -> 1 + ??
  | S n -> f(n) + ??
```

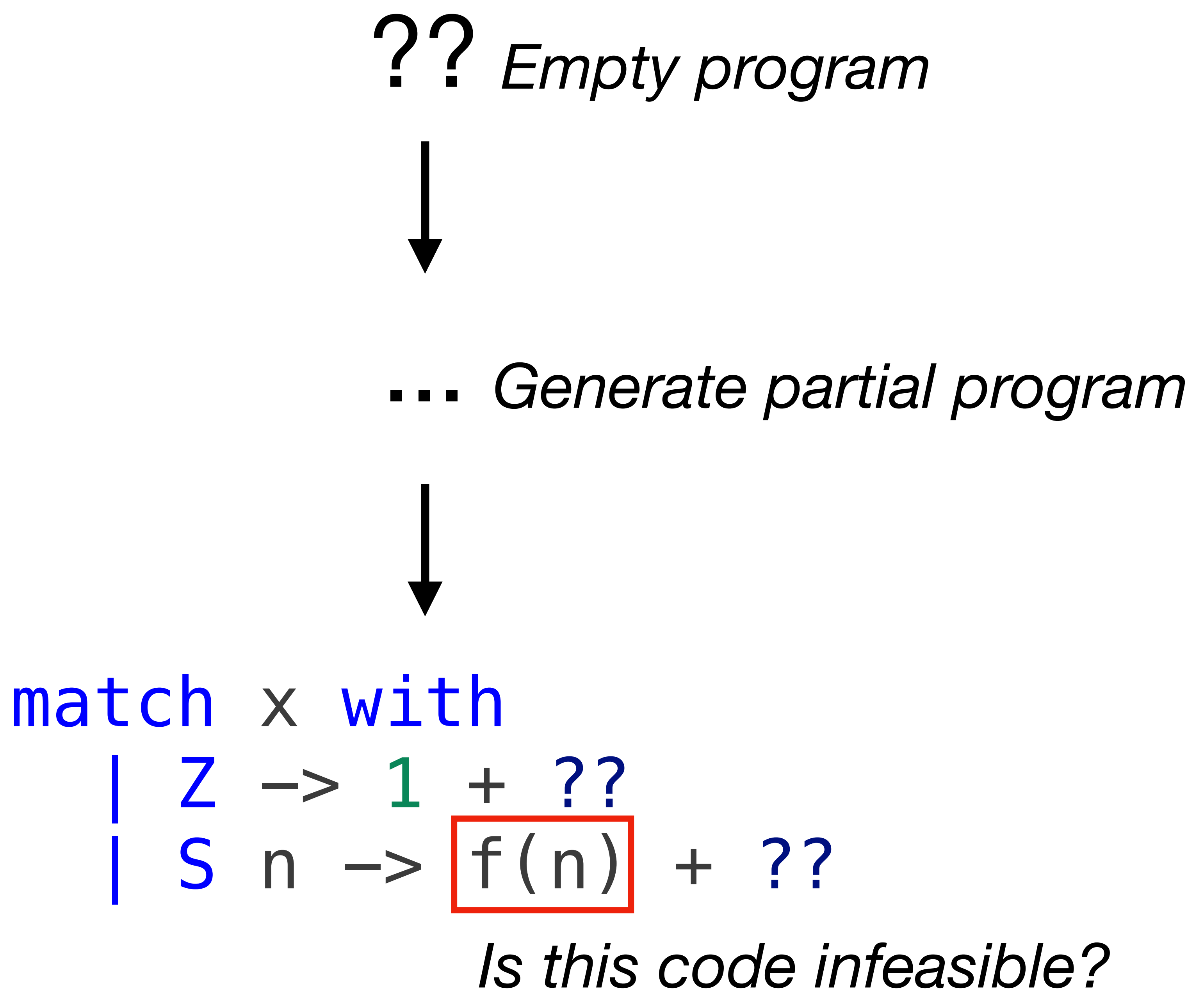
*Is this code infeasible?*

# Problem

- Top-down
  - Start from empty program, generate partial program
  - Prune infeasible partial programs

```
let rec f x = ??
```

I/O examples: (0 → 1, 1 → 2)



?? *Empty program*

... *Generate partial program*

```
match x with
  | Z -> 1 + ??
  | S n -> f(n) + ??
```

*Is this code infeasible?*

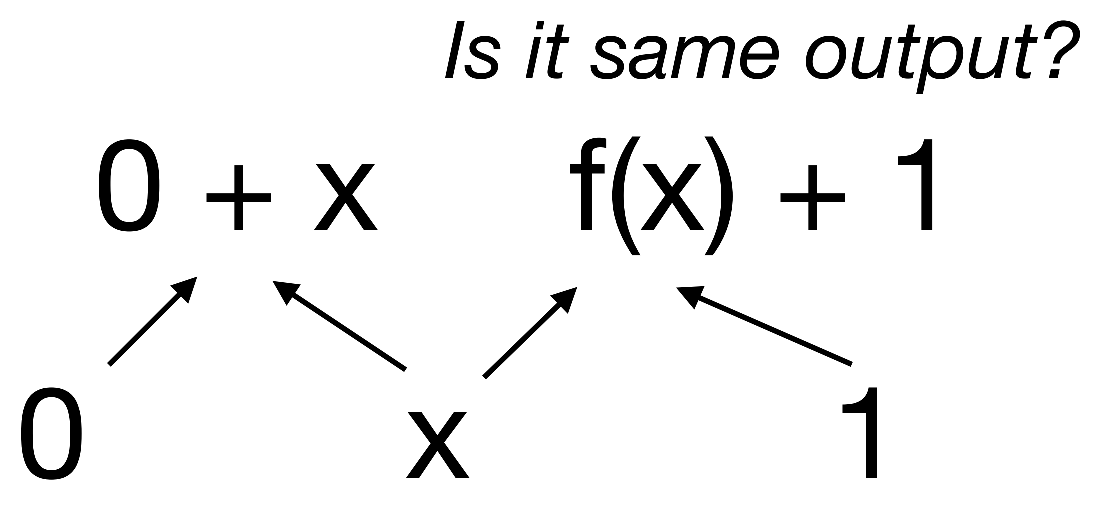
- Need to know if this candidate is infeasible
- The candidate is calling an undefined f
- To prune this candidate, we should approximate its possible behavior
- But it is **difficult** problem due to **recursion**

# Problem

- Bottom-up
  - Builds larger programs from smaller one
  - Prune redundant sub-expressions by evaluation

# Problem

- Bottom-up
  - Builds larger programs from smaller one
  - Prune redundant sub-expressions by evaluation

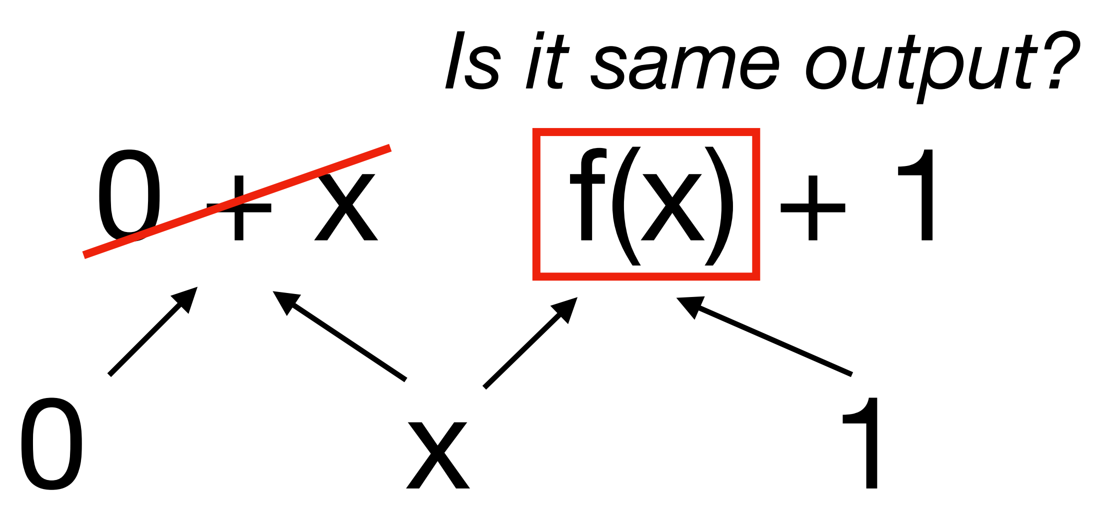


```
let rec f x = ??
```

I/O examples: (0 → 1, 1 → 2)

# Problem

- Bottom-up
  - Builds larger programs from smaller one
  - Prune redundant sub-expressions by evaluation



*Is it same output?*

~~0 + x~~ f(x) + 1

0 x 1

```
let rec f x = ??
```

I/O examples: (0 → 1, 1 → 2)

- Build a program by completing from terminal nodes such as 0, x, and 1
- By evaluating, sub-expression with the same result is pruned
- Bottom-up requires evaluation to check if f(x) + 1 is redundant
- However, **f is not defined**, so it is **difficult** to check

# Idea

1. Synthesize all possible recursion- and conditional-free expressions satisfying each I/O example

# Idea

1. Synthesize **all possible recursion-** and **conditional-free expressions satisfying each I/O example**

We call these **blocks**

# Idea

1. Synthesize **all possible recursion-** and **conditional-free expressions satisfying each I/O example**

We call these **blocks**

2. **Prune candidates** inconsistent with the blocks obtained during **top-down** search for a recursive solution

# TRIO

- To solve the current problem,
  - they provide a recursive functional program synthesizer called **TRIO**
- Implementation of the idea in three steps
- Released TRIO tools as open source[1]

[1] TRIO, https://github.com/pslhy/trio

# Result overview

- Total 60 benchmarks
- Evaluate with IO spec, Ref spec
- Synthesize more programs than conventional SOTA

| | BURST[1] | SMYTH[2] | **TRIO** |
|---|---|---|---|
| # Solved (IO spec.) | 50/60 | 50/60 | **59/60** |
| # Solved (Ref spec.) | 39/60 | 54/60 | **57/60** |

[1] Anders Miltner et al, Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. Proc. ACM Program. Lang. 6, POPL (2022)

[2] Justin Lubin et al, Program sketching with live bidirectional evaluation. Proceedings of the ACM on Programming Languages 4, ICFP (2020)

# Example

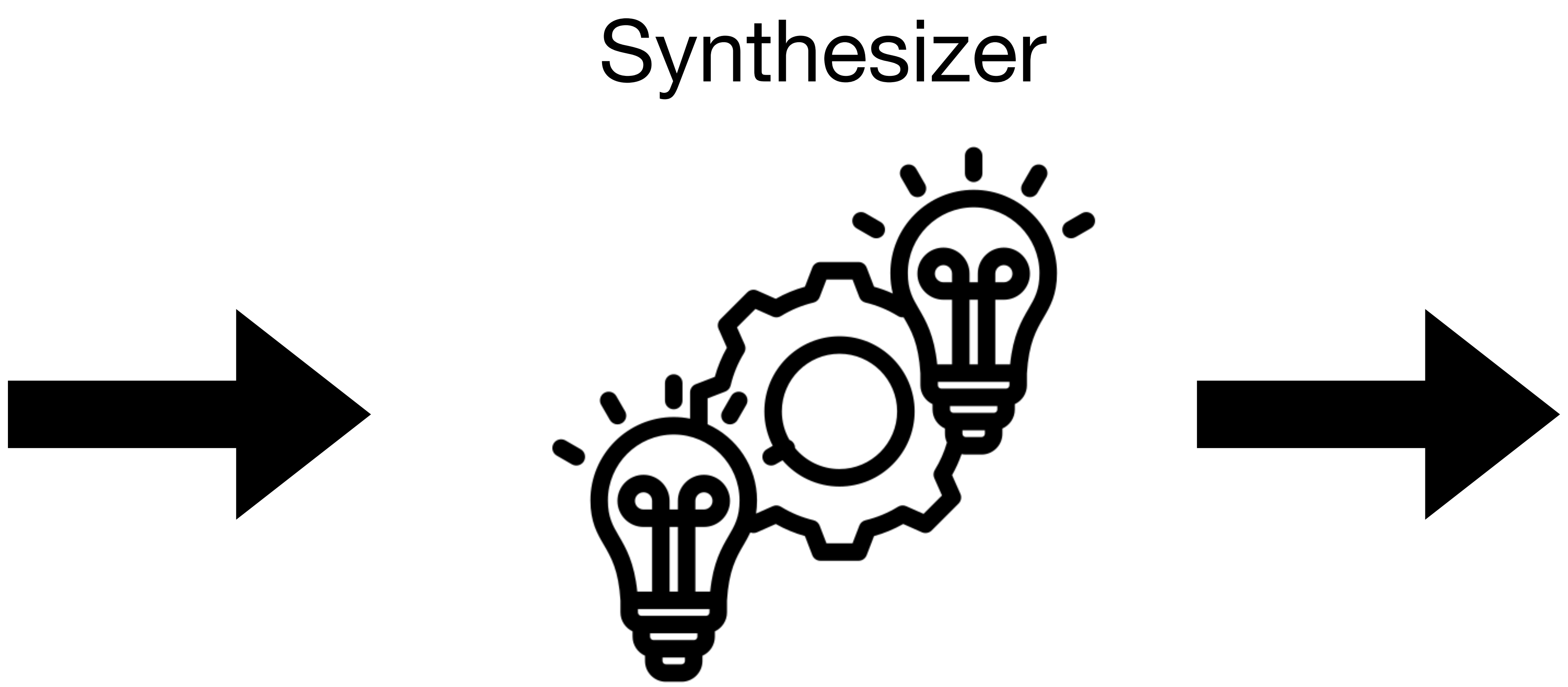
Assume synthesize the double function

```
type nat = Z
         | S -> nat
```

```
 f : nat -> nat

 Z → Z
 S(Z) → S(S(Z))
 S(S(Z)) → S(S(S(S(Z))))
```

```
add:
nat * nat -> nat
```



```
let rec f x =
  match x with
  | Z -> Z
  | S n -> (f n) + S(S(Z))
```

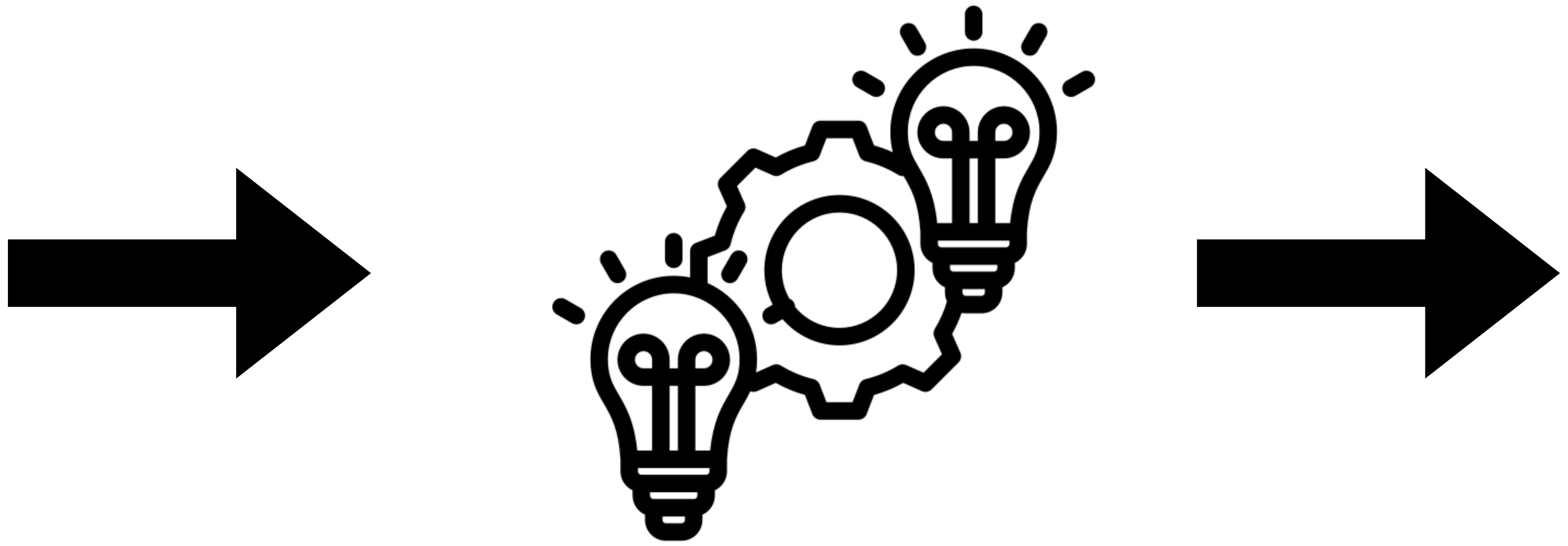
# Example

Assume synthesize the double function

```
type nat = Z
         | S -> nat
```

```
f : nat -> nat
```

```
0 → 0
1 → 2
2 → 4
```



Synthesizer

```
let rec f x =
  match x with
  | Z -> Z
  | S n -> (f n) + S(S(Z))
```

```
add:
nat * nat -> nat
```

# 1) Synthesized blocks

| I/O Example | Synthesized Blocks |
|---|---|
| 0 → 0 | |
| 1 → 2 | |
| 2 → 4 | |

# 1) Synthesized blocks

| I/O Example | Synthesized Blocks |
|---|---|
| 0 → 0 | 0, x, 0+0, 0+x, x+0, x+x, ... |
| 1 → 2 | |
| 2 → 4 | |

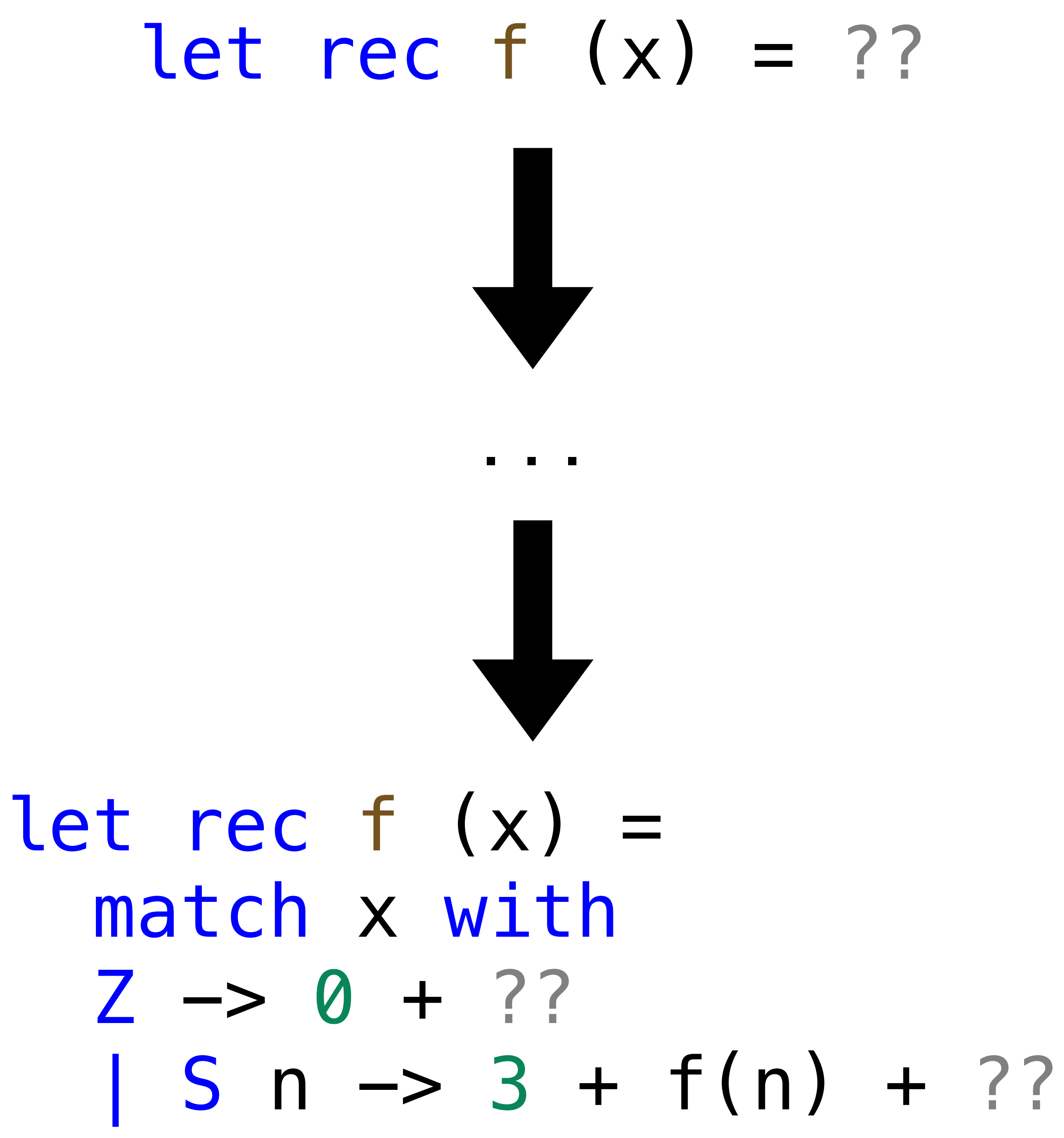
# 1) Synthesized blocks

| I/O Example | Synthesized Blocks |
|---|---|
| 0 → 0 | 0, x, 0+0, 0+x, x+0, x+x, ... |
| 1 → 2 | 2, 1+1, 0+2, 2+0, x+1, 1+x, x+x, ... |
| 2 → 4 | |

# 1) Synthesized blocks

| I/O Example | Synthesized Blocks |
|---|---|
| `0 → 0` | 0, x, 0+0, 0+x, x+0, x+x, ... |
| `1 → 2` | 2, 1+1, 0+2, 2+0, x+1, 1+x, x+x, ... |
| `2 → 4` | 4, 1+3, 2+2, 3+1, x+2, 2+x, x+x, ... |

# 2) Prune candidates



```
let rec f (x) = ??
```

...

```
let rec f (x) =
  match x with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

Assume we evaluated this during top-down search

# 2) Prune candidates

| I/O Example | Synthesized Blocks |
|---|---|
| 0 → 0 | 0, x, 0+0, 0+x, x+0, x+x, ... |

```
let rec f (x) =
  match x with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

# 2) Prune candidates

| I/O Example | Synthesized Blocks |
|---|---|
| 0 → 0 | 0, x, 0+0, 0+x, x+0, x+x, ... |

```
let rec f (x) =
  match x with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

➡

```
let rec f (x) =
  match Z with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

# 2) Prune candidates

| I/O Example | Synthesized Blocks |
|---|---|
| 0 → 0 | 0, x, 0+0, 0+x, x+0, x+x, ... |

```
let rec f (x) =
  match x with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

➡

```
let rec f (x) =
  match Z with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

➡

```
0 + ??
```

# 2) Prune candidates

| I/O Example | Synthesized Blocks |
|---|---|
| 0 → 0 | 0, x, 0+0, **0+x**, x+0, x+x, ... |

```
let rec f (x) =
  match x with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

➡

```
let rec f (x) =
  match Z with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

➡

```
0 + ??
```

there exists a completion of the partial program that satisfies I/O example 1

# 2) Prune candidates

| I/O Example | Synthesized Blocks |
|---|---|
| 1 → 2 | 2, 1+1, 0+2, 2+0, x+1, 1+x, x+x, ... |

```
let rec f (x) =
  match x with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

➡

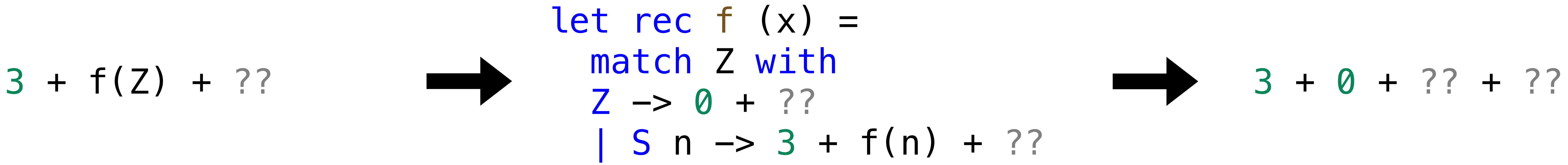
```
let rec f (x) =
  match S(Z) with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

➡

```
3 + f(Z) + ??
```

# 2) Prune candidates

| I/O Example | Synthesized Blocks |
|---|---|
| `1 → 2` | 2, 1+1, 0+2, 2+0, x+1, 1+x, x+x, ... |



```
3 + f(Z) + ??
```

```
let rec f (x) =
  match Z with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

```
3 + 0 + ?? + ??
```

# 2) Prune candidates

| I/O Example | Synthesized Blocks |
|---|---|
| `1 → 2` | 2, 1+1, 0+2, 2+0, x+1, 1+x, x+x, ... |

```
3 + f(Z) + ??
```

➡

```
let rec f (x) =
  match Z with
  Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

➡

```
3 + 0 + ?? + ??
```

there is no completion of the partial program that satisfies I/O example 2

# TRIO

- To solve the current problem,
  - they provide a recursive functional program synthesizer called **TRIO**
- Implementation of the idea in **three steps**
- Released TRIO tools as open source[1]

[1] TRIO, https://github.com/pslhy/trio

# TRIO

- To solve the current problem,
  - they provide a recursive functional program synthesizer called **TRIO**
- Implementation of the idea in **three steps**
  1. Bottom-up enumerator
  2. Block Generation
  3. Candidate Generation

# Bottom-up enumerator

- Initially put the input with the component size **n** and the **inputs**
- Input is **I/O examples**, **library function**

# Bottom-up enumerator

- Initially put the input with the component size **n** and the **inputs**
- Input is **I/O examples**, **library function**

```
f : nat -> nat

Z → Z
S(Z) → S(S(Z))
```

Input 2) target function type, I/O examples

```
add:
nat * nat -> nat
```

Input 3) library of external operators

# Component Generation

- Generate as many components as the number of n inputs
- The component is sub-expressions to be used in the solution
    - For example, it may be x, 0, 1, x+1, or the like

If n=5, then C = {x, 0, 1, 2, x+1}

# Library Sampling

- Proceed with the processing for the library function

# Library Sampling

- Proceed with the processing for the library function
- One of the existing problems was the processing problem for library function

# Library Sampling

- Proceed with the processing for the library function
- One of the existing problems was the processing problem for library function
- We create an inverse map with output → input for the library function

# Library Sampling

- Proceed with the processing for the library function
- One of the existing problems was the processing problem for library function
- We create an inverse map with output → input for the library function
- Library inverse map for the add(+) operation is created as follow

# Library Sampling

- Proceed with the processing for the library function
- One of the existing problems was the processing problem for library function
- We create an inverse map with output → input for the library function
- Library inverse map for the add(+) operation is created as follow

```
add:
nat * nat -> nat
```

→

$+^{-1}(0) = \{(0,0)\}$, $+^{-1}(1) = \{(0,1), (1,0)\}$, $+^{-1}(2) = \{(0,2), (1,1), (2,0)\}$

$+^{-1}(3) = \{(1,2), (2,1)\}$, $+^{-1}(4) = \{(2,2)\}$

# Block Generation

- Generating blocks that satisfy each I/O sample as our idea
- Each block is expressed in a data structure called a version space
    - The reason is to effectively generate a lot of blocks

# Version Space

{ 1+1, 1+x, 1+2,

x+1, x+x, x+2,

2+1, 2+x, 2+2 }

Blocks

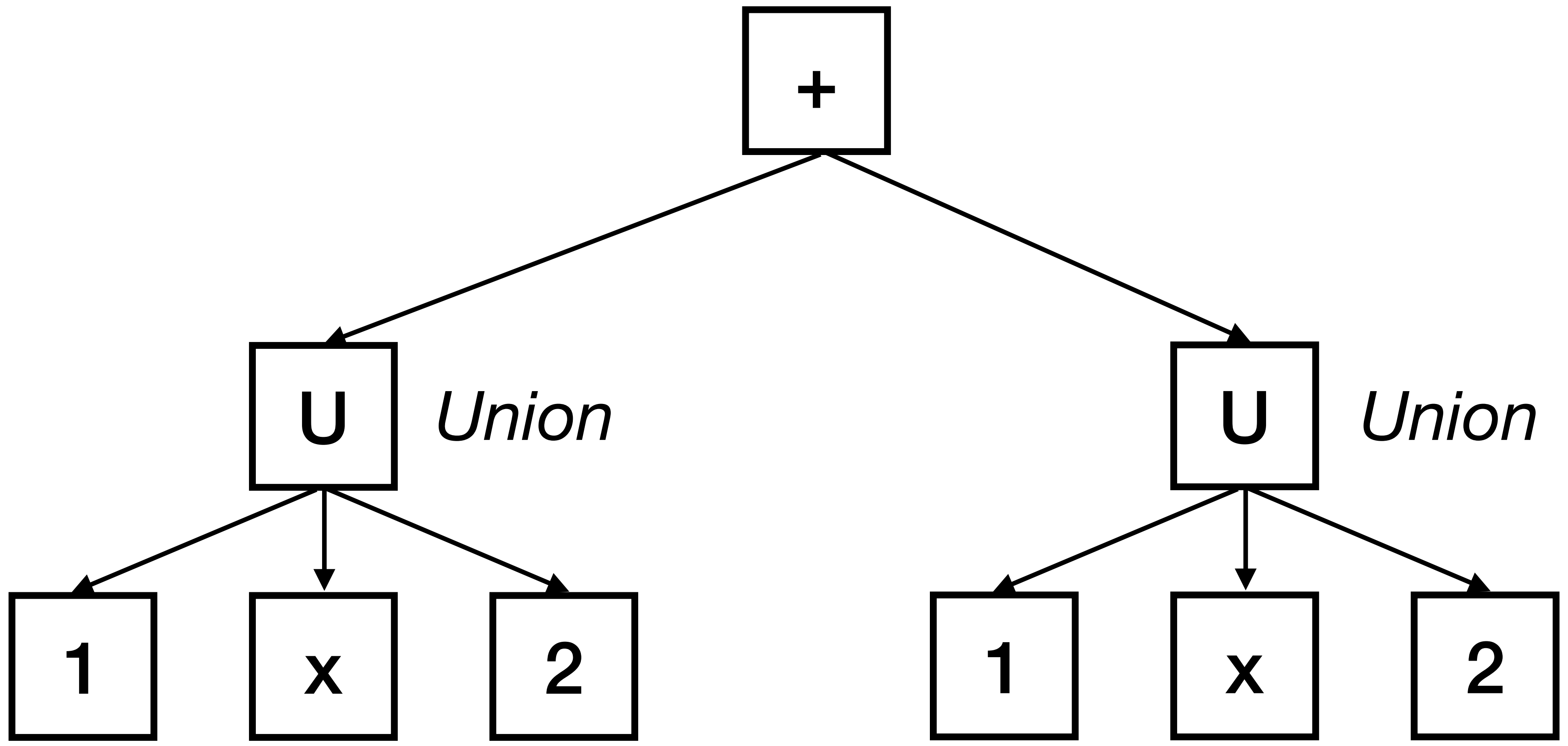
# Version Space

{ 1+1, 1+x, 1+2,

x+1, x+x, x+2,

2+1, 2+x, 2+2 }

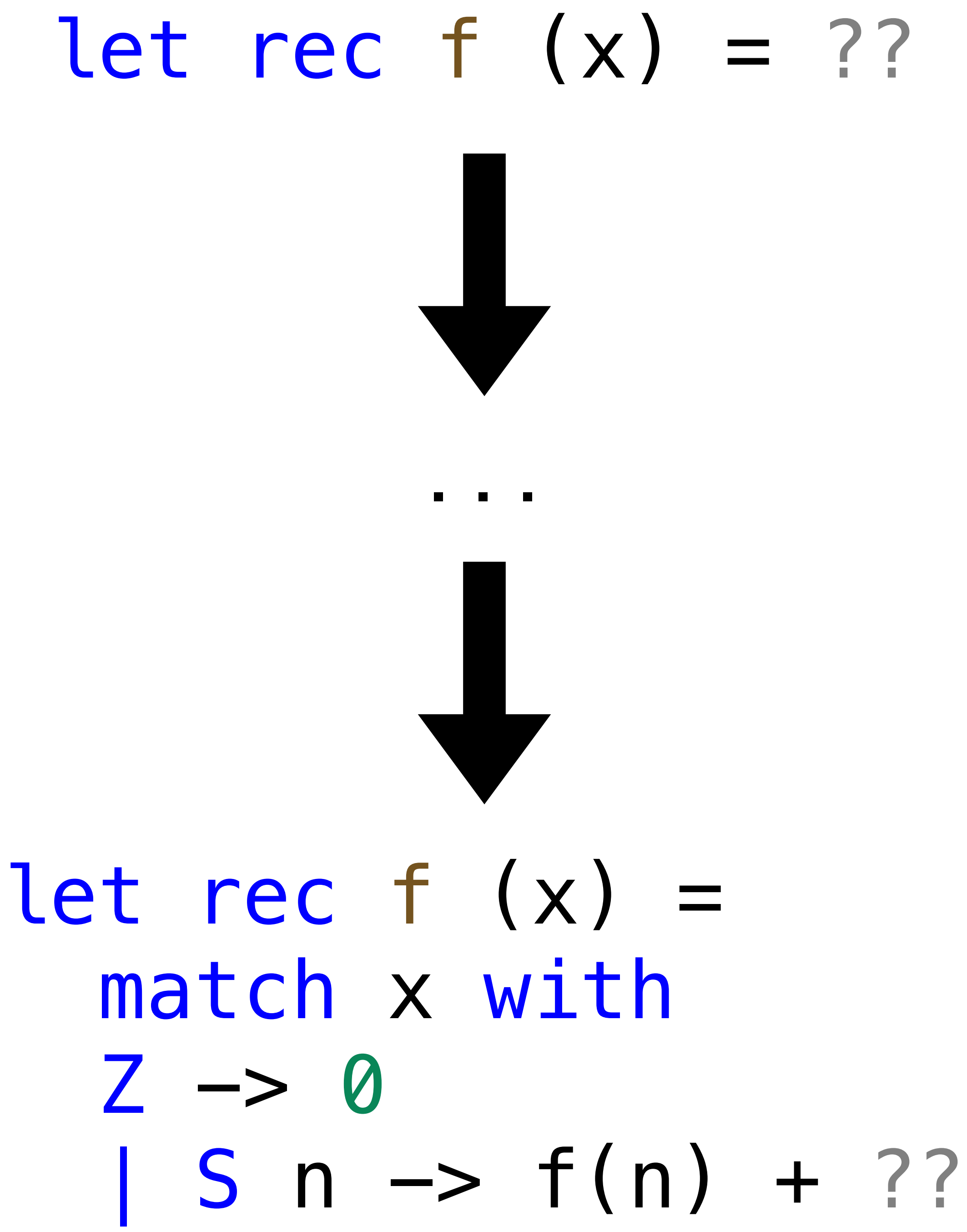
Blocks



Blocks with Version Space

# Candidate Generation

- C = {x, 0, 1, 2, x+1}



```
let rec f (x) = ??
        ↓
       ...
        ↓
let rec f (x) =
  match x with
  Z -> 0
  | S n -> f(n) + ??
```

# Candidate Generation

- 1 → 2 I/O example
- C = {x, 0, 1, 2, x+1}

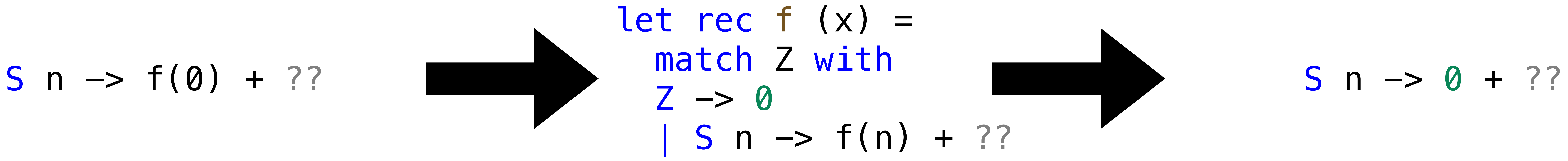
```
let rec f (x) =
  match x with
  Z -> 0
  | S n -> f(n) + ??
```

```
let rec f (x) =
  match n with
  Z -> 0
  | S n -> f(n) + ??
```

```
S n -> f(0) + ??
```

# Candidate Generation

- 1 → 2 I/O example
- C = {x, 0, 1, 2, x+1}



```
S n -> f(0) + ??
```

```
let rec f (x) =
  match Z with
  Z -> 0
  | S n -> f(n) + ??
```

```
S n -> 0 + ??
```

# Candidate Generation

- 1 → 2 I/O example
- C = {x, 0, 1, 2, x+1}

```
S n -> 0 + ??
```

| I/O Example | Synthesized Blocks |
|---|---|
| 1 → 2 | 2, 1+1, 0+2, 2+0, x+1, 1+x, x+x, ... |

# Candidate Generation

- 1 → 2 I/O example
- C = {x, 0, 1, 2, x+1}

```
S n -> 0 + ??
```

| I/O Example | Synthesized Blocks |
|---|---|
| 1 → 2 | 2, 1+1, **<u>0+2</u>**, 2+0, x+1, 1+x, x+x, ... |

# Candidate Generation

- 1 → 2 I/O example
- C = {x, 0, 1, 2, x+1}

```
S n -> 0 + ??
```

| I/O Example | Synthesized Blocks |
|---|---|
| 1 → 2 | 2, 1+1, **0+2**, 2+0, x+1, 1+x, x+x, ... |

there exists a completion of the partial program that satisfies I/O example

# Candidate Generation

- 1 → 2 I/O example
- C = {x, 0, 1, 2, x+1}

```
S n -> 0 + ??
```

| I/O Example | Synthesized Blocks |
|---|---|
| 1 → 2 | 2, 1+1, **<u>0+2</u>**, 2+0, x+1, 1+x, x+x, ... |

there exists a completion of the partial program that satisfies I/O example

Only different is using version space in TRIO

# Evaluation

- Benchmark
  - 60 programs
  - 45 from SMyth + 15 from OCaml tutorial
- Baseline
  - SMyth[1], Burst[2]
- 2 min timeout

[1] Anders Miltner et al, Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. Proc. ACM Program. Lang. 6, POPL (2022)
[2] Justin Lubin et al, Program sketching with live bidirectional evaluation. Proceedings of the ACM on Programming Languages 4, ICFP (2020)

# Evaluation

- Specifications (*bool_xor*)

```
type bool =
| False
| True

synth bool -> bool -> bool satisfying

[True,True] -> False,
[True,False] -> True,
[False,True] -> True,
[False,False] -> False
```

1) I/O examples

```
type bool =
| False
| True

synth bool -> bool -> bool satisfying

equiv

fix (f : bool -> bool -> bool) =
  fun (b1:bool) ->
    fun (b2:bool) ->
    match b1 with
      | False _ -> b2
      | True _ -> (match b2 with
                   | False _ -> True
                   | True _ -> False)
```

2) Reference implmentation

# Evaluation

- Specifications (*bool_xor*)

```
type bool =
| False
| True

synth bool -> bool -> bool satisfying

[True,True] -> False,
[True,False] -> True,
[False,True] -> True,
[False,False] -> False
```

1) I/O examples

```
type bool =
| False
| True

synth bool -> bool -> bool satisfying

equiv

fix (f : bool -> bool -> bool) =
  fun (b1:bool) ->
    fun (b2:bool) ->
    match b1 with
      | False _ -> b2
      | True _ -> (match b2 with
                     | False _ -> True
                     | True _ -> False)
```

2) Reference implmentation

candidate is semantically equivalent?

# Evaluation

| | BURST | SMYTH | **TRIO** |
|---|---|---|---|
| # Solved (IO spec.) | 50/60 | 50/60 | **59/60** |
| # Solved (Ref spec.) | 39/60 | 54/60 | **57/60** |

# Review

- Propose for general synthesis of recursive functional programs
- Release TRIO as open-source
  - Compared to previous review paper
- Wondering where this program synthesis can be used
  - ↔ FlashFill, SQLizer
  - Guide how to learn recursive functional programming for beginners?

# Summary

- Synthesizing recursive functional programs is hard problem
- To solve this problem, they propose TRIO with
    - Bottom-up enumerator
    - Block generator
    - Candidate generator
- As a result, TRIO outperforms the existing tools

# Backup

# Failure Analysis

- expr_div

```
[INT(1)] -> 1,
[ADD(INT(3), INT(4))] -> 7,
[MUL(INT(3), INT(3))] -> 9,
[MUL(INT(2), INT(3))] -> 6,
[SUB(INT(4), INT(3))] -> 1,
[SUB(INT(5), INT(1))] -> 4,
[DIV(INT(4), INT(2))] -> 2,
[DIV(INT(5), INT(3))] -> 1

type nat = Z | S of nat
type expr = NAT of nat | ADD of expr and expr
| SUB of expr and expr | MUL of expr and expr
| DIV of expr and expr

rec add (x : nat, y : nat) : nat = ...
rec sub (x : nat, y : nat) : nat = ...
rec mul (x : nat, y : nat) : nat = ...
rec div (x : nat, y : nat) : nat = ...
```

⬇

```
rec eval (x : expr) : nat = ??
```

- **<u>extremely many possible combinations</u>** of
  - recursive calls,
  - external operators
  - case matching

- If the specification is restricted
  - add, sub, mul,
- Only **<u>TRIO</u>** can find the solution

# Library Sampling

- To guarantee termination,
  - Generate only cases
  - that are smaller than the maximum value of input examples
- If not, there are so many possible library inverse map
- For example,
  - (0,0), (0,1), ..., (2,2) when spec is { 0↦0, 2↦4 }

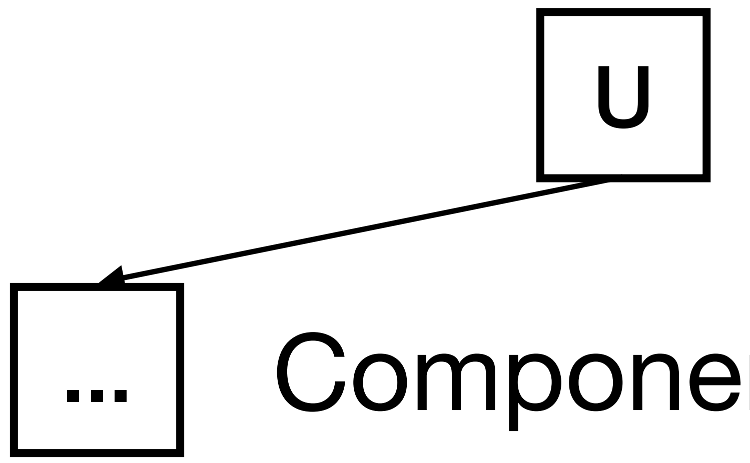
# Evaluation Ref Imp

- They integrated Burst and SMyth into a **CEGIS** loop and, for each candidate program proposed by each tool
- they use the verifier to determine whether the candidate is **semantically equivalent** to the reference implementation
- If not, a new input-output example comprising a **counter-example input** generated by the verifier and its **corresponding output is added**
- This process is repeated until the **desired program is found**

# Block Generation

- Generate Blocks satisfying 1 → 2 I/O example

C = {x, 0, 1, 2, x+1}

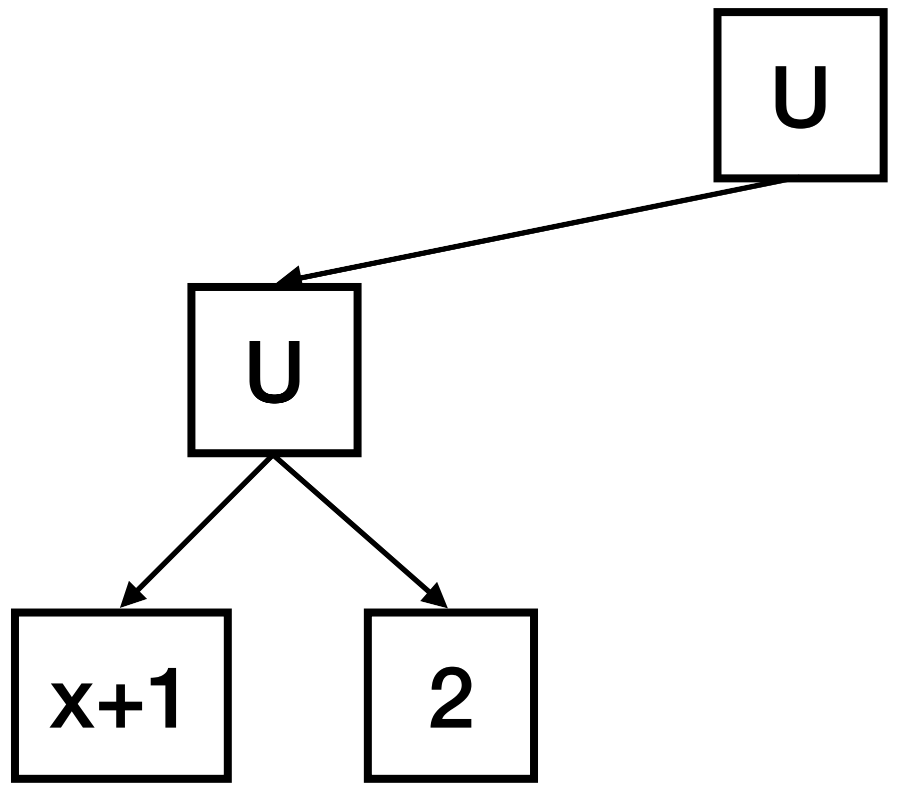


Components satisfying 1 → 2

# Block Generation

- Generate Blocks satisfying 1 → 2 I/O example

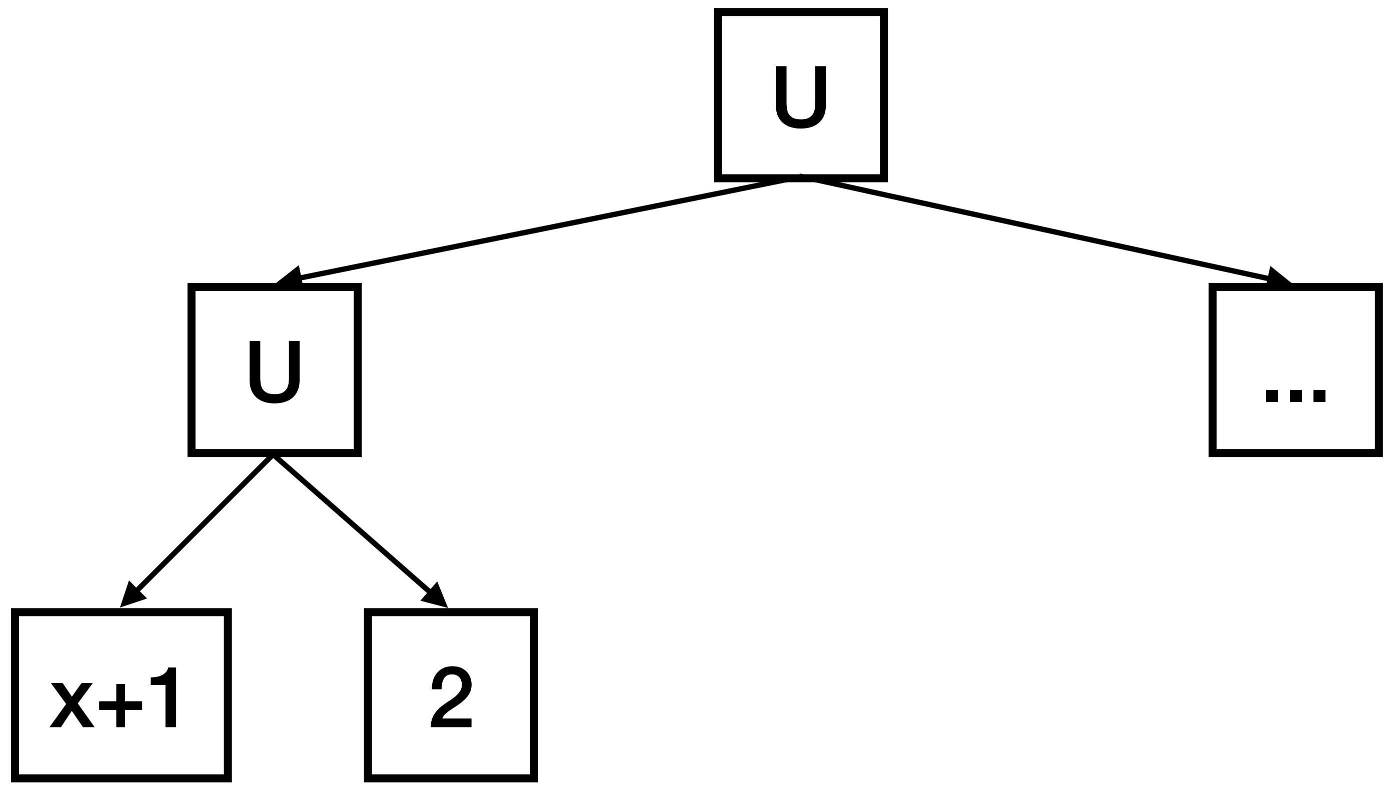
C = {x, 0, 1, 2, x+1}



# Block Generation

- Generate Blocks satisfying 1 → 2 I/O example

C = {x, 0, 1, 2, x+1}

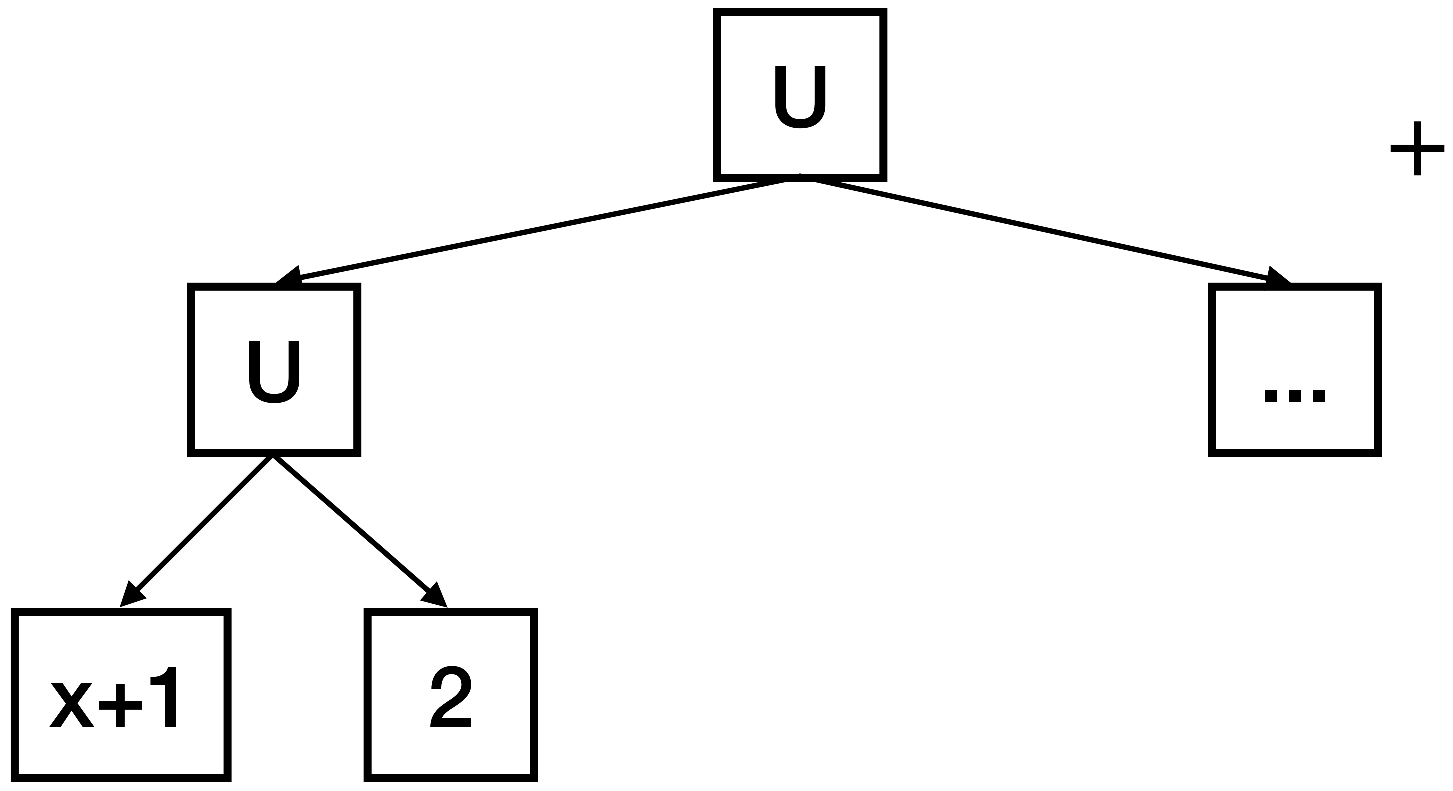


Blocks of form **e1 + e2** satisfying 1 → 2

# Block Generation

- Generate Blocks satisfying 1 → 2 I/O example

C = {x, 0, 1, 2, x+1}

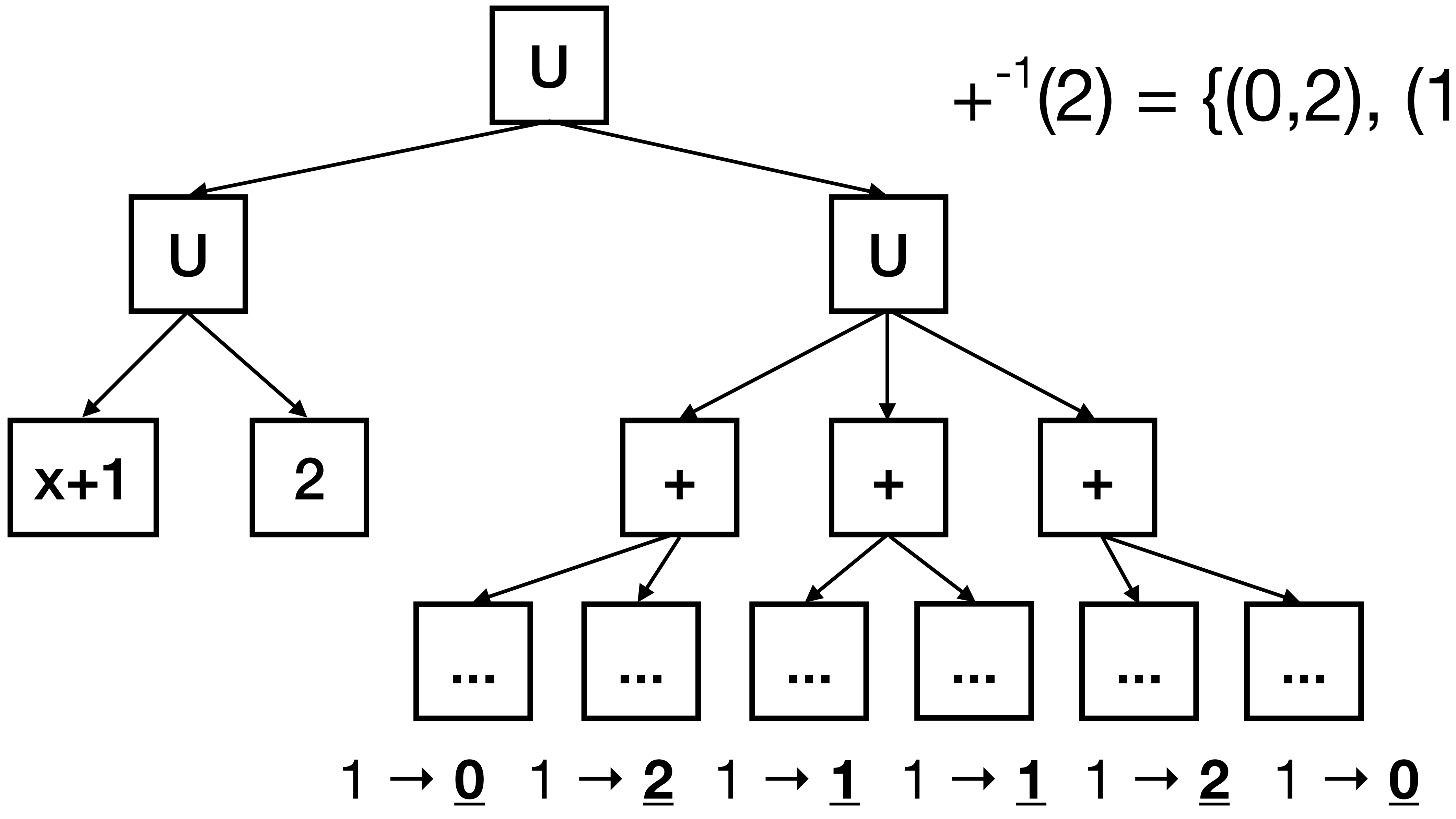


$+^{-1}(2) = \{(0,2), (1,1), (2,0)\}$

# Block Generation

- Generate Blocks satisfying 1 → 2 I/O example

C = {x, 0, 1, 2, x+1}

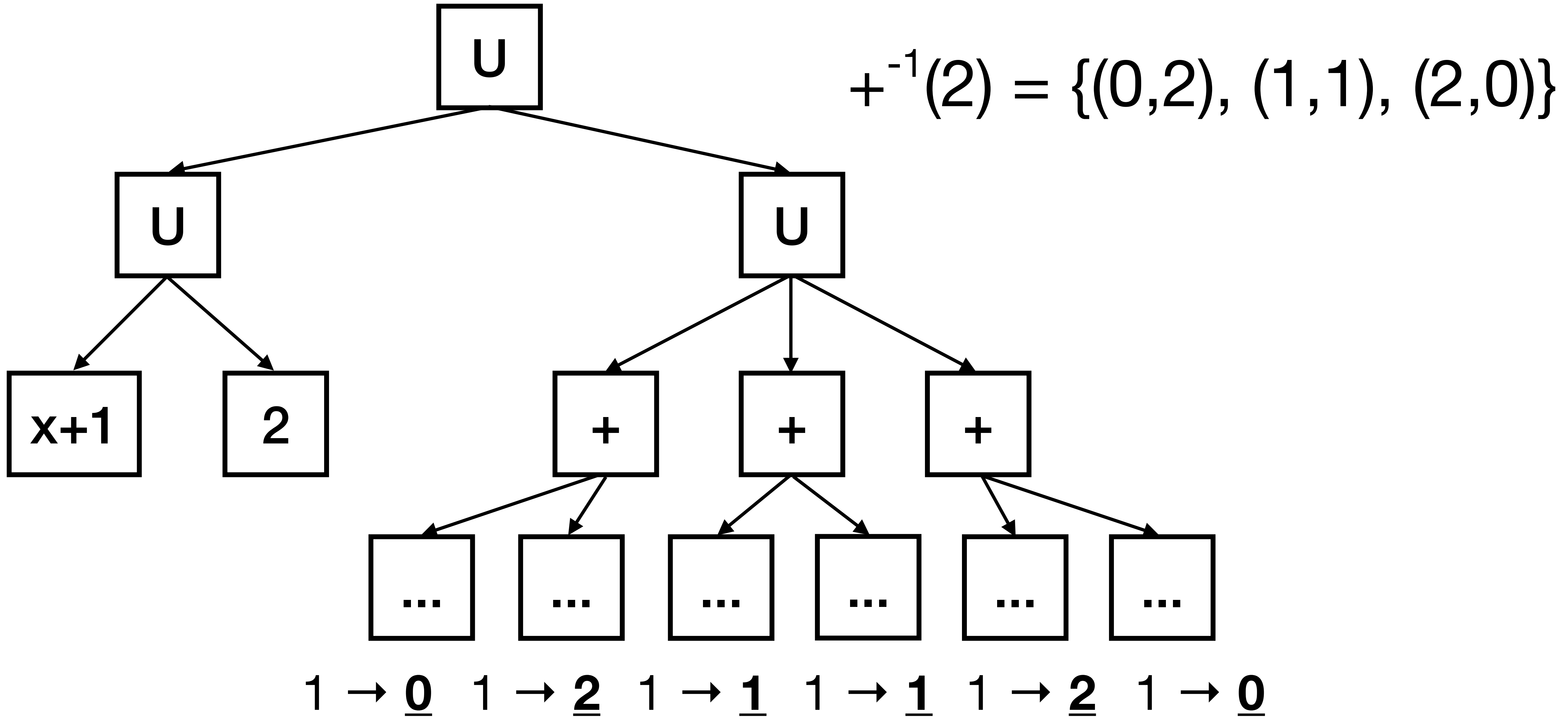


$+^{-1}(2) = \{(0,2), (1,1), (2,0)\}$

# Block Generation

- Generate Blocks satisfying 1 → 2 I/O example

C = {x, 0, 1, 2, x+1}

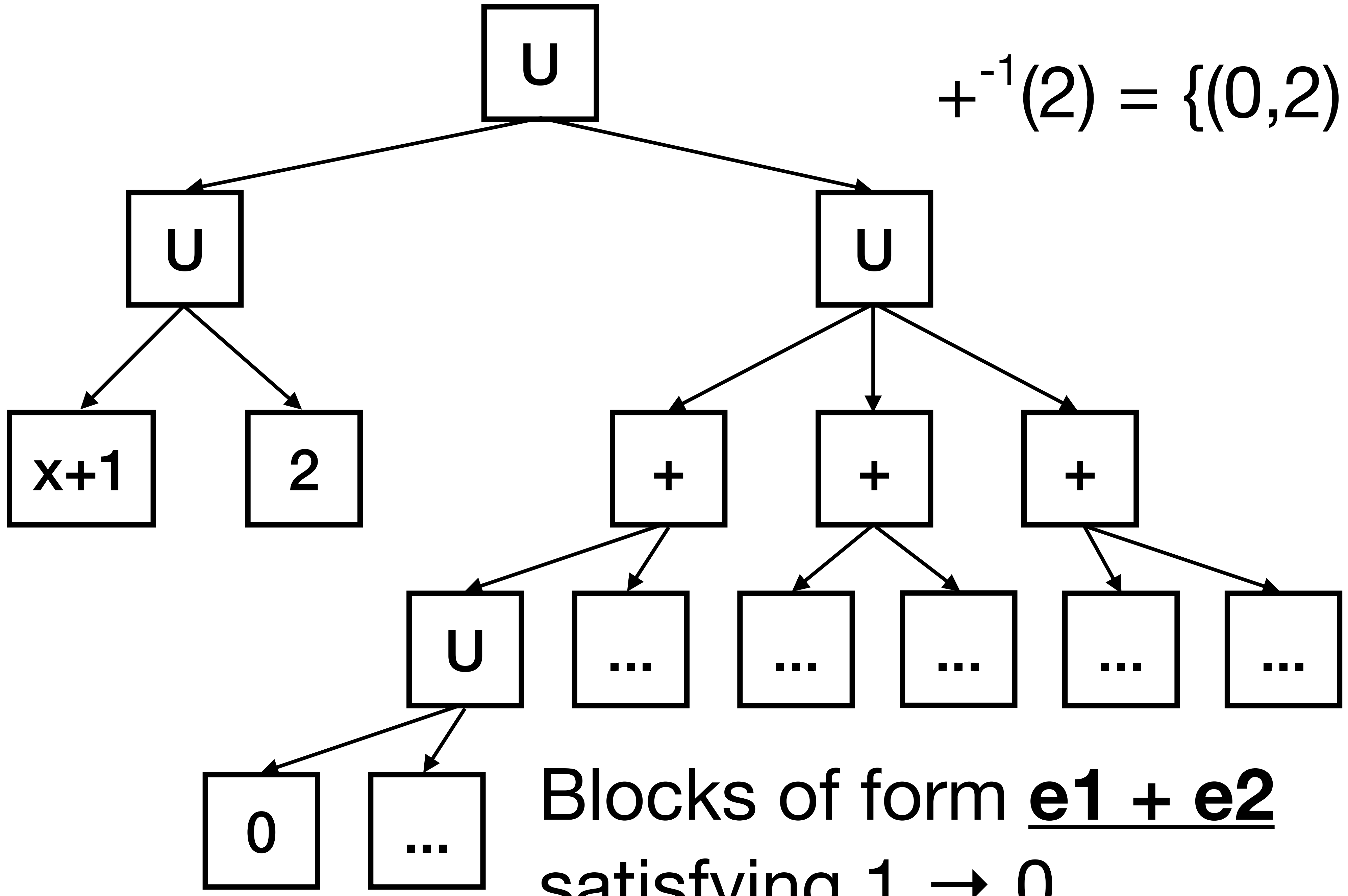


Components satisfying 1 → 0

# Block Generation

- Generate Blocks satisfying 1 → 2 I/O example

C = {x, 0, 1, 2, x+1}



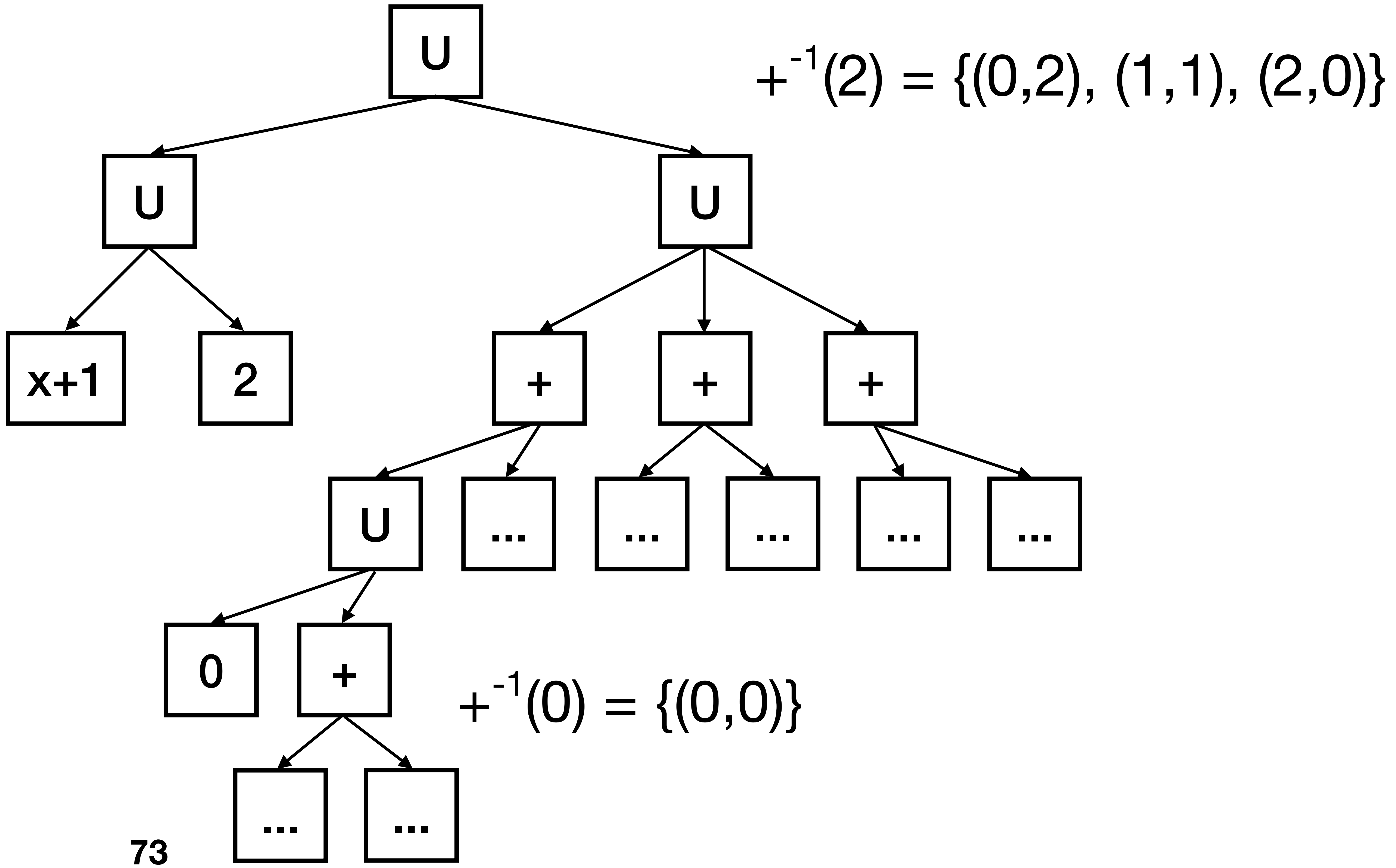
$+^{-1}(2) = \{(0,2), (1,1), (2,0)\}$

Blocks of form **e1 + e2**
satisfying 1 → 0

# Block Generation

- Generate Blocks satisfying 1 → 2 I/O example

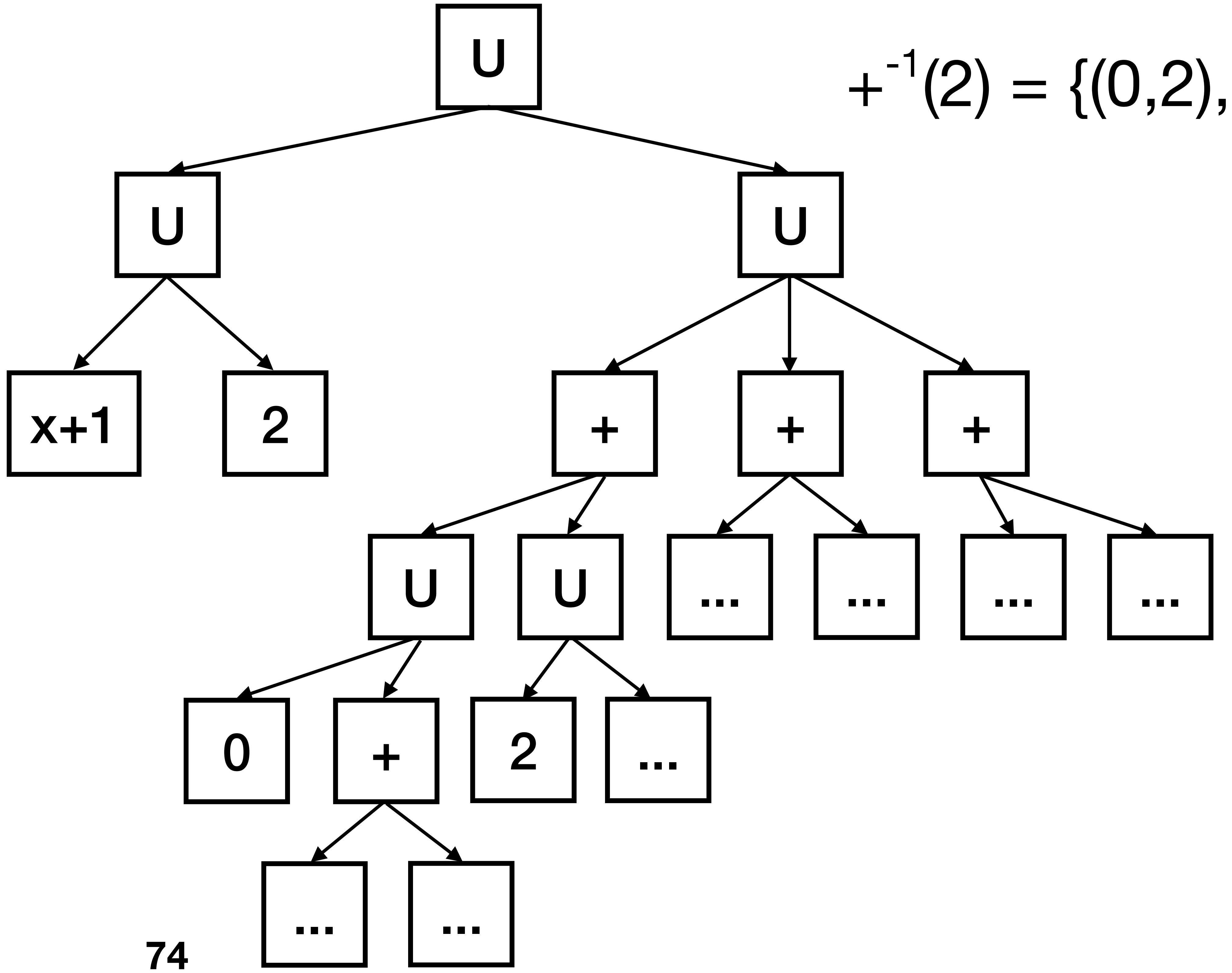
C = {x, 0, 1, 2, x+1}



# Block Generation

- Generate Blocks satisfying 1 → 2 I/O example

C = {x, 0, 1, 2, x+1}

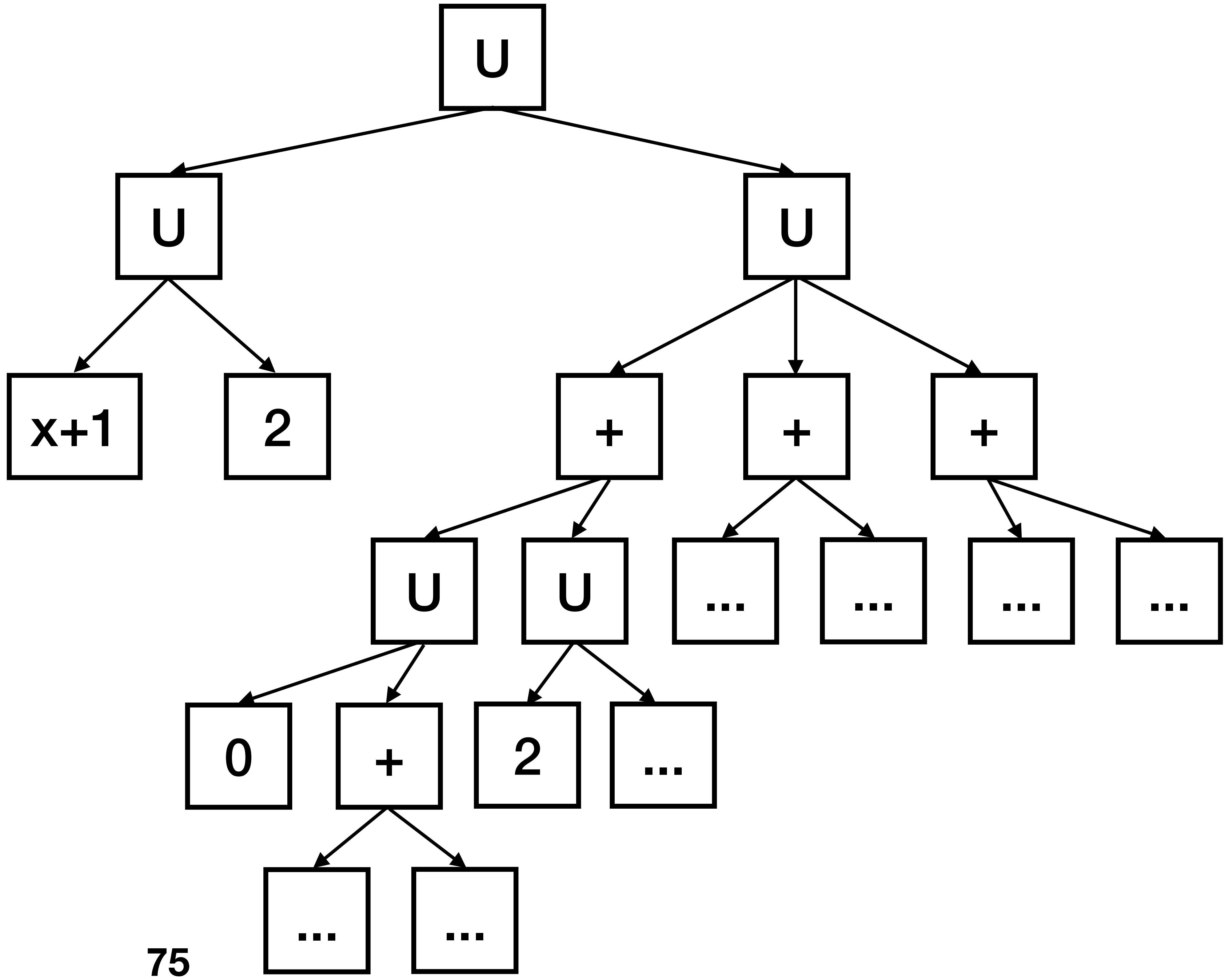


$+^{-1}(2) = \{(0,2), (1,1), (2,0)\}$

# Candidate Generation

- Prune candidate program (1 → 2 I/O example)

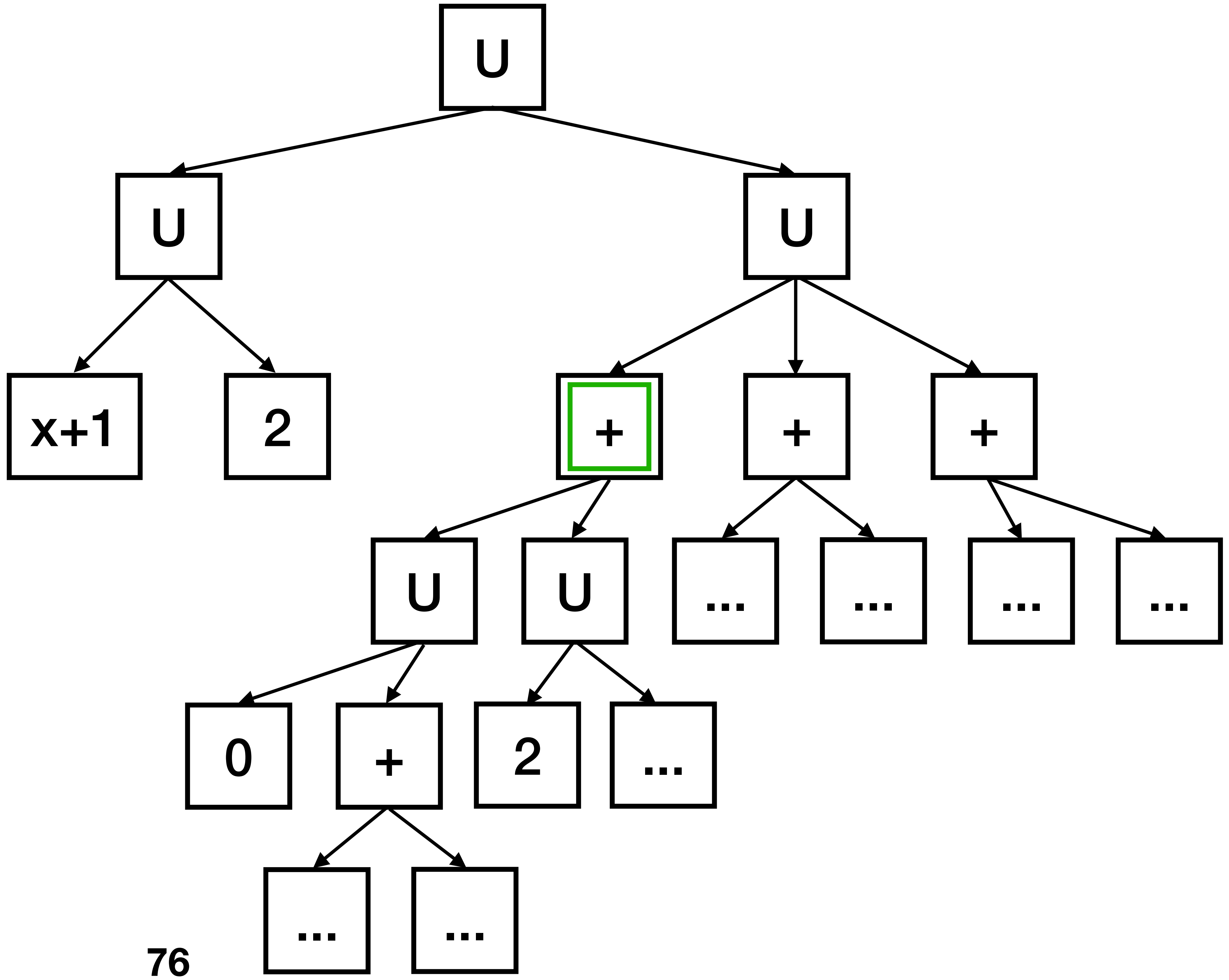
```
0 + ??
```



# Candidate Generation

- Prune candidate program (1 → 2 I/O example)

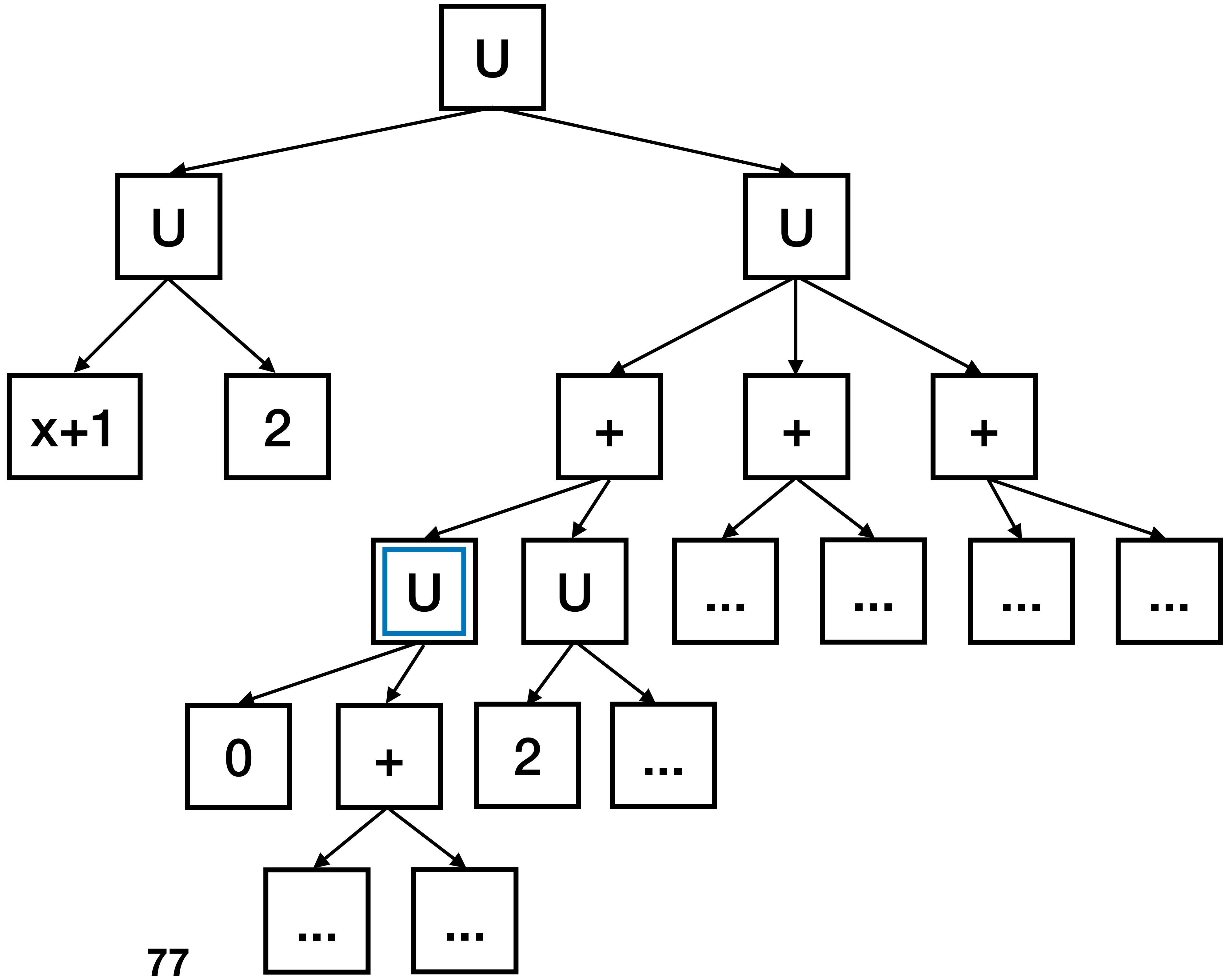
```
0 + ??
```



# Candidate Generation

- Prune candidate program (1 → 2 I/O example)

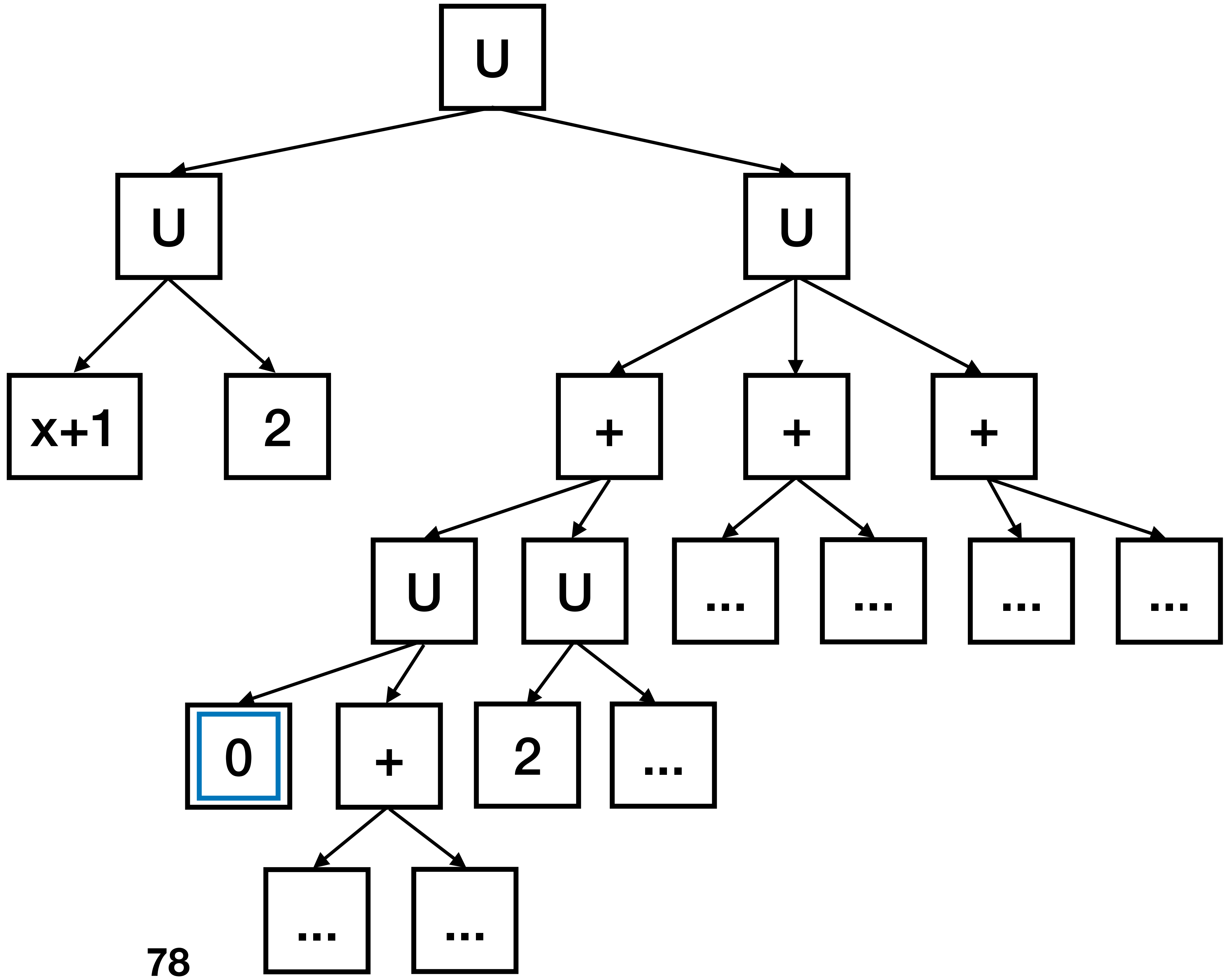
```
0 + ??
```



# Candidate Generation

- Prune candidate program (1 → 2 I/O example)

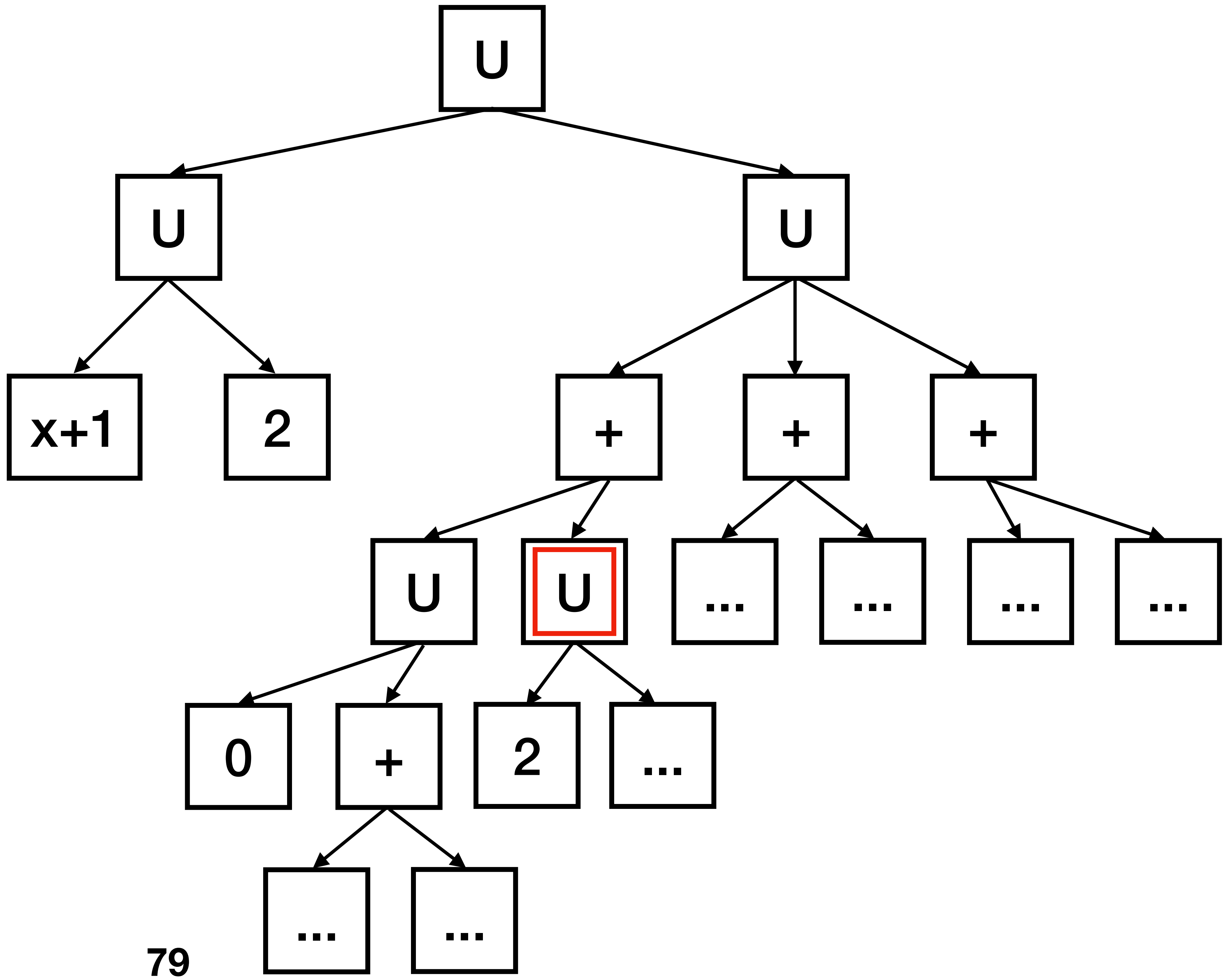
```
0 + ??
```



# Candidate Generation

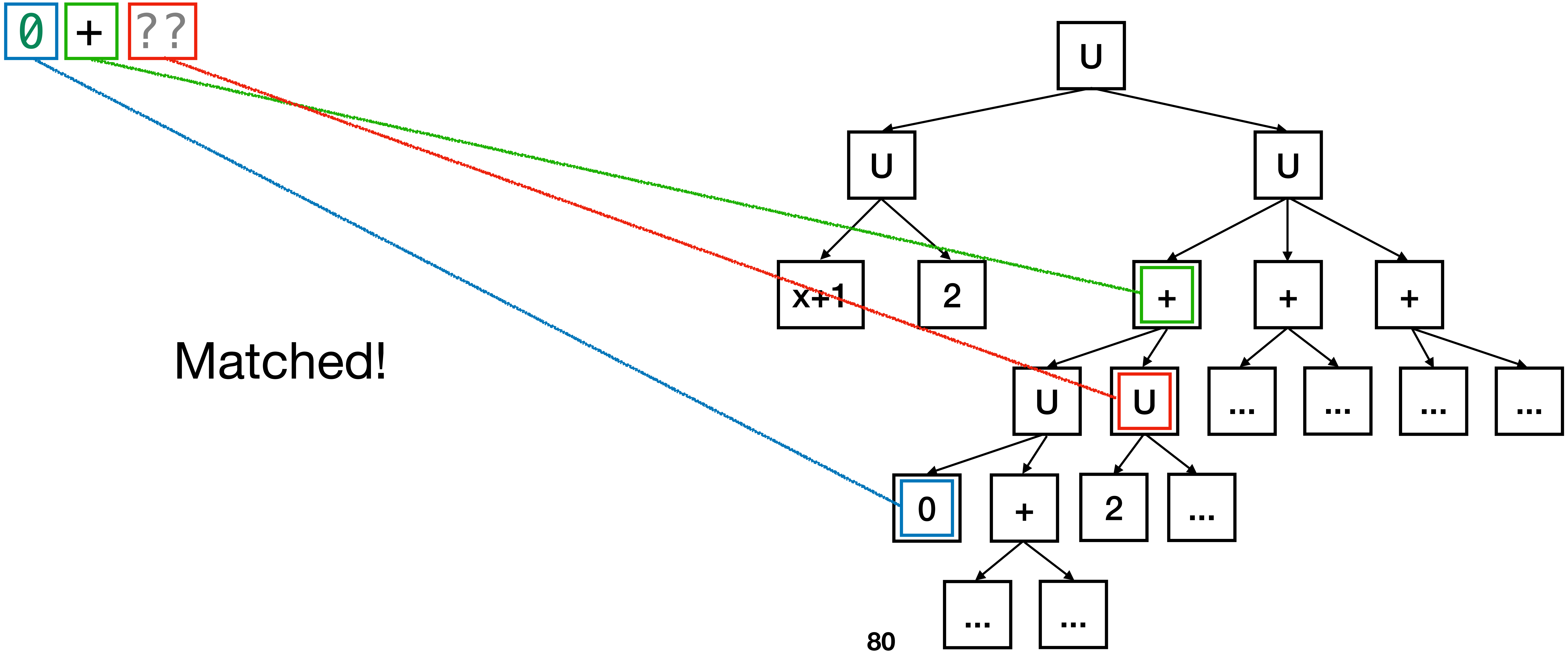
- Prune candidate program (1 → 2 I/O example)

```
0 + ??
```



# Candidate Generation

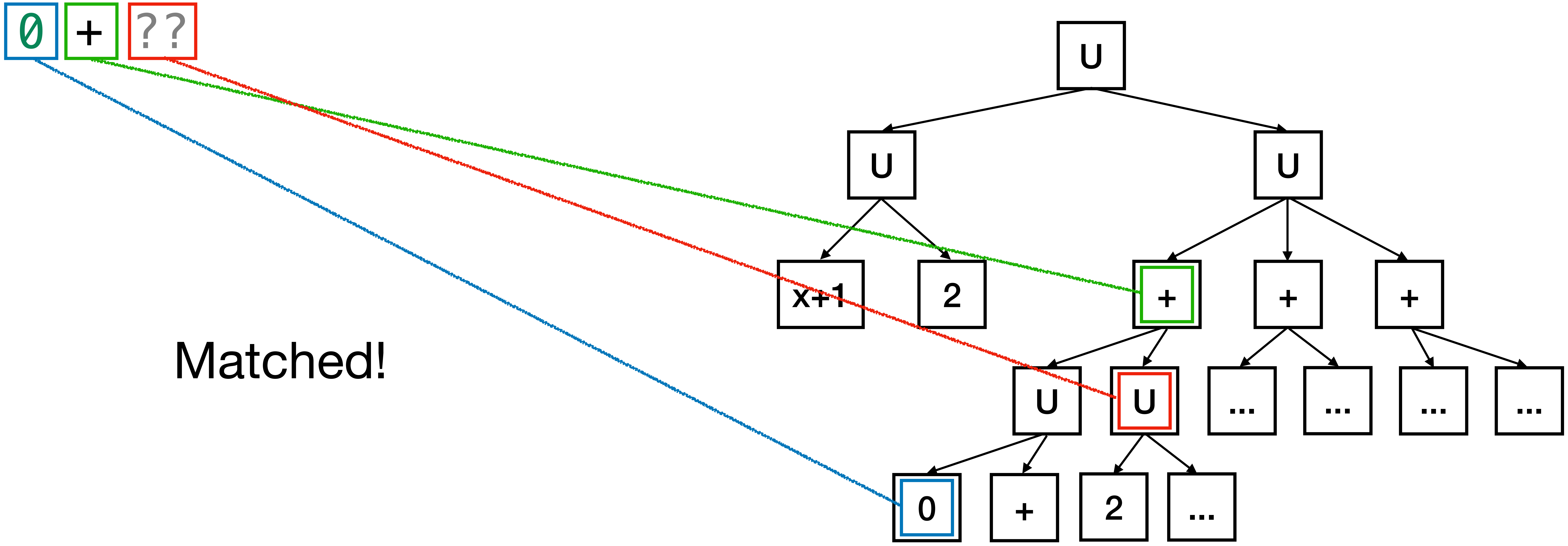
- Prune candidate program (1 → 2 I/O example)



Matched!

# Candidate Generation

- Prune candidate program (1 → 2 I/O example)



there exists a completion of the partial program that satisfies I/O example