

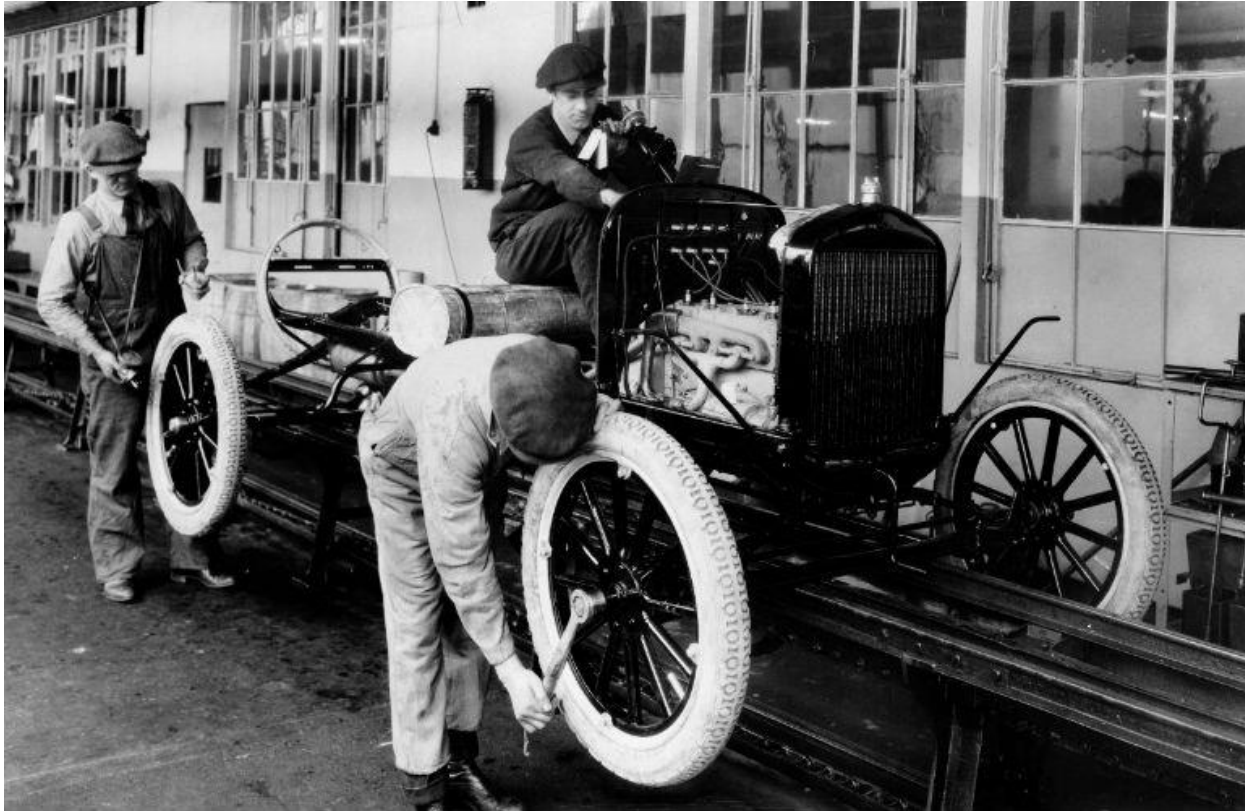
program synthesis

Nadia Polikarpova



UCSDCSE
Computer Science and Engineering

goal: automate programming

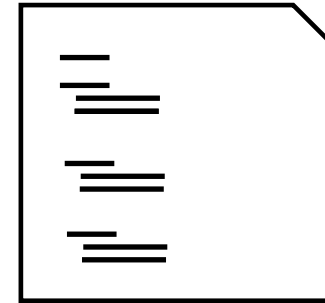


program synthesis

specification



program



program synthesis

specification



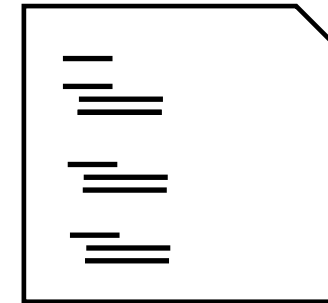
search



program
space



program



program synthesis

specification

?

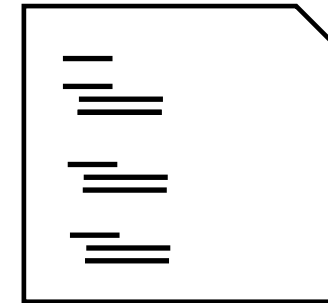


search

?



program



program
space

?

program synthesis

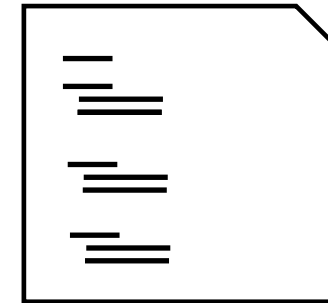
specification



search

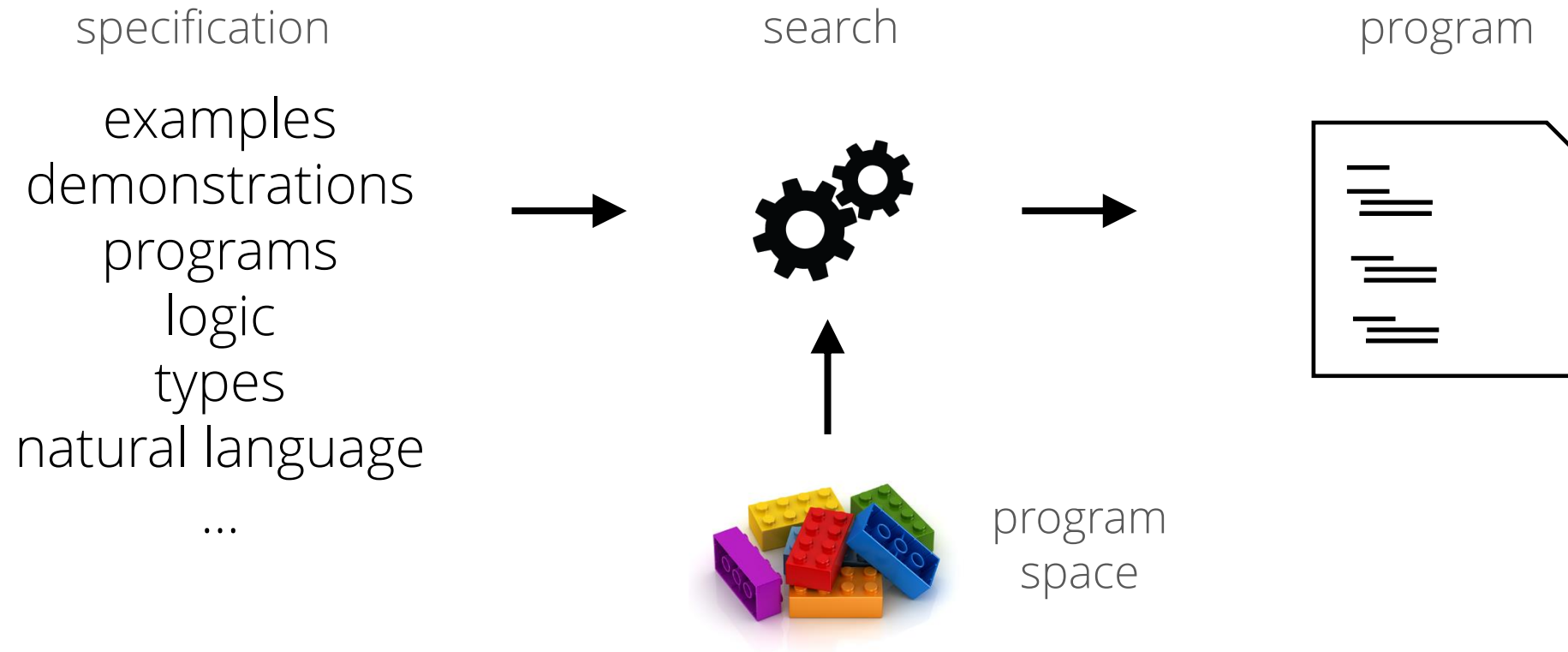


program



program
space

program synthesis

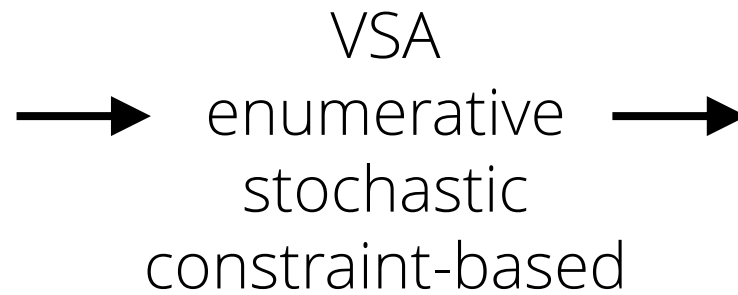


program synthesis

specification

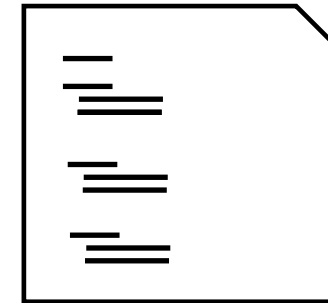


search



...

program

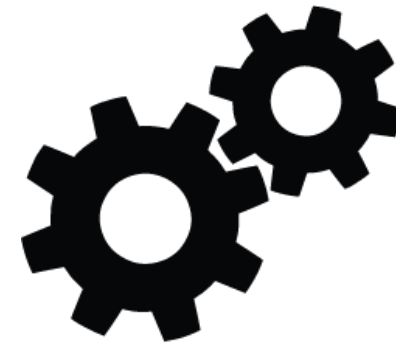


this talk

specifications



search strategies



this talk

specifications



search strategies



what makes a good spec?

specification



1. human-friendly

2. informative

3. synthesizer-friendly

what makes a good spec?

specification



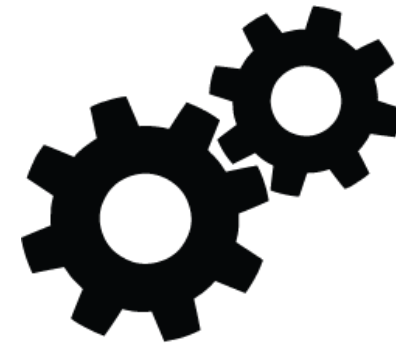
1. **human-friendly**
easier to write than the program
2. **informative**
minimal ambiguity
3. **synthesizer-friendly**
easy to check, guides the search

this talk

specifications

1. examples (PBE)
2. programs
3. types
4. natural language

search strategies



this talk

specifications

1. examples (PBE)
2. programs
3. types
4. natural language

search strategies



this talk

specifications

1. examples (PBE)
2. programs
3. types
4. natural language

FlashFill

[Gulwani 2011]

CNNMoney TECH

Excel 2013's coolest new feature that should have been available years ago

lifehacker AUST

TECHWORLD

WIRED SUBSCRIBE >> SECTIONS >> BLOGS >> REVIEWS >> VIDEO >> Sign In | RSS

THE TIMES OF INDIA Tech

Home City India World Business Tech Sports Entertainment Life & Style

PCWorld News Reviews How-To

TRENDING: Phones Tablets Laptops Windows

engadget COMPUTEX 2012

ZDNet White Paper

US Edition Laptops Data Center

The Seattle Times Winner of a 2012 Pulitzer Prize

Excel

demo!

	A	B	C	D	E	F	G	H
1	Malcolm Turnbull	Malcolm						
2	Bernie Ripoll	Bernie						
3	Steven Clark							

pbe / pbd: discussion

Domains:

- text transformations: [FlashFill](#), [FlashExtract](#)
- web scraping: WebRelate, Rousillon
- data science: Morpheus, Wrex
- programmer's assistant: FrAngel, Snippy

pbe / pbd: discussion

Domains:

- text transformations: [FlashFill](#), [FlashExtract](#)
- web scraping: [WebRelate](#), [Rousillon](#)
- data science: [Morpheus](#), [Wrex](#)
- programmer's assistant: [FrAngel](#), [Snippy](#)

+ beginner-friendly
+ easy to check
correctness

- ambiguous
- hard to write for complex
programs / data structures
- cannot express non-
functional properties

this talk

specifications

1. examples (PBE)
2. programs
3. types
4. natural language

Sketch

[Solar-Lezama et al, 2005]

Problem: isolate the least significant zero bit in a word

0010 0101 → 0000 0010

Sketch

Problem: isolate the least significant zero bit in a word
Easy to implement with a loop

```
bit[32] isolate0 (bit[32] x) {  
    bit[32] ret = 0;  
    for (int i = 0; i < 32; i++)  
        if (!x[i]) { ret[i] = 1; return ret; }  
}
```

Sketch

Problem: isolate the least significant zero bit in a word
Easy to implement with a loop

```
bit[32] isolate0 (bit[32] x) {  
    bit[32] ret = 0;  
    for (int i = 0; i < 32; i++)  
        if (!x[i]) { ret[i] = 1; return ret; }  
}
```

Can this be done more efficiently with bit manipulation?

Sketch

Problem: isolate the least significant zero bit in a word
Easy to implement with a loop

```
bit[32] isolate0 (bit[32] x) {  
    bit[32] ret = 0;  
    for (int i = 0; i < 32; i++)  
        if (!x[i]) { ret[i] = 1; return ret; }  
}
```

Can this be done more efficiently with bit manipulation?

Trick: adding 1 to a string of ones turns the next zero to a 1
i.e. $000111 + 1 = 001000$

Sketch: synthesis goal

```
bit[32] isolate0fast (bit[32] x) implements isolate0 {  
    return expr(x, 3);  
}
```

Sketch: synthesis goal

```
bit[32] isolate0fast (bit[32] x) implements isolate0 {
    return expr(x, 3);
}

// Sketch for bit-vector expressions with
// +, &, xor and bitwise negation (~)
generator bit[32] expr(bit[32] x, int depth){
    assert depth > 0;
    if(??) return x;
    if(??) return ??;
    if(??) return ~gen(x, depth-1);
    if(??){
        return { | expr(x, depth-1) (+ | & | ^) expr(x, depth-1) | };
    }
}
```

Sketch: synthesis goal

```
bit[32] isolate0fast (bit[32] x) implements isolate0 {  
    return expr(x, 3);  
}  
  
// Sketch for bit-vector expressions with  
// +, &, xor and bitwise negation (~)  
generator bit[32] expr(bit[32] x, int depth){  
    assert depth > 0;  
    if(??) return x;  
    if(??) return ??;  
    if(??) return ~gen(x, depth-1);  
    if(??){  
        return { | expr(x, depth-1) (+ | & | ^) expr(x, depth-1) | };  
    }  
}
```



Sketch: output

```
bit[W] isolate0fast (bit[W] x) {  
    return (~x) & (x + 1);  
}
```

reference programs: discussion

Domains:

- superoptimization: [Stoke](#), [Lens](#)
- verified lifting: [QBS](#), [STNG](#), other Alvin Cheung's work

reference programs: discussion

Domains:

- superoptimization: [Stoke](#), [Lens](#)
- verified lifting: [QBS](#), [STNG](#), other Alvin Cheung's work

+ programmer-friendly
+ precise

- simple program does not always exist
- hard to check correctness

this talk

specifications

1. examples (PBE)
2. programs
3. types
4. natural language

Hoogle+

[[Guo et al, 2020](#)]

Problem: remove adjacent duplicates from a list
using library functions

`[1, 1, 1, 2, 2, 1] → [1, 2, 1]`

inspiration: Hoogle

Hoogλe

inspiration: Hoogle

Hoogle

```
map :: (a -> b) -> [a] -> [b]
```

`Data.List`

`map f xs` is the list obtained by applying `f` to each element of `xs`, i.e.,

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
```

Hoogle needs synthesis!

Hoog λ e

[a] $\rightarrow$ [a]

Search

Hoogle needs synthesis!

Hoog λ e

Hoogle needs synthesis!

Hoogle

```
nub :: Eq a => [a] -> [a]
```

Data.List

The nub function removes duplicate elements from a list.

```
tail :: [a] -> [a]
```

Prelude

Extract elements after the head of the list.

Hoogle needs synthesis!

Hoogle

```
nub :: Eq a => [a] -> [a]
```

Data.List

The nub function removes duplicate elements from a list.

```
tail :: [a] -> [a]
```

Prelude

Extract elements after the head of the list.

no relevant results!

Hoogle+

specification

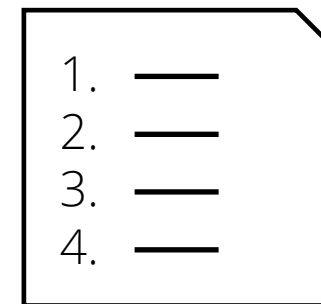
$\text{Eq } a \Rightarrow [a] \rightarrow [a]$



H+



programs

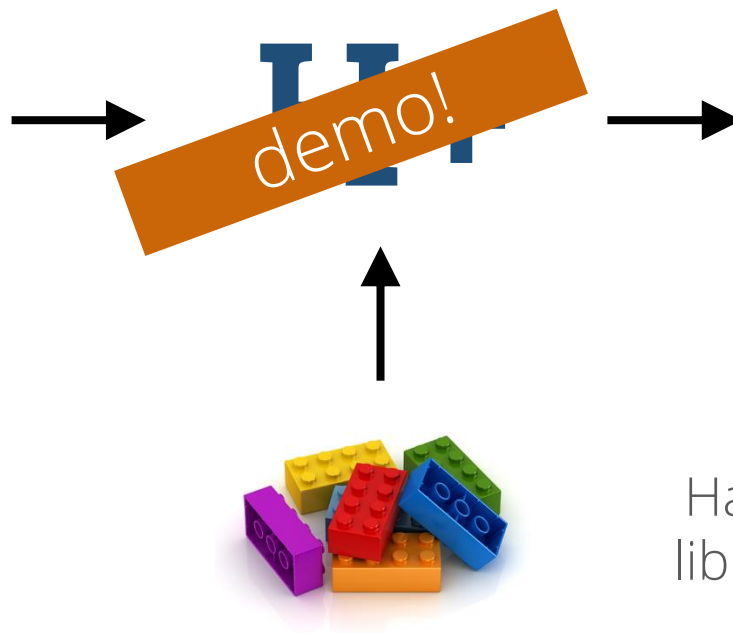


Haskell
libraries

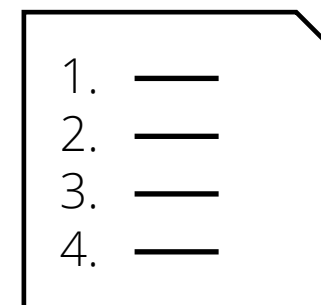
Hoogle+

specification

`Eq a => [a] -> [a]`



programs



Haskell
libraries

Synquid

[[Polikarpova et al, 2016](#)]

Problem: duplicate every element in a list
using recursion

[1, 2, 3] → [1, 1, 2, 2, 3, 3]

type-driven synthesis

specification

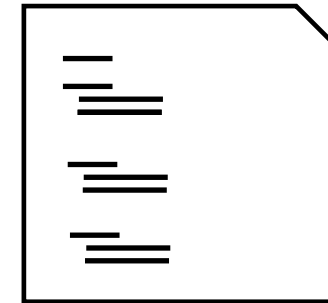
$[a] \rightarrow [a]$



search

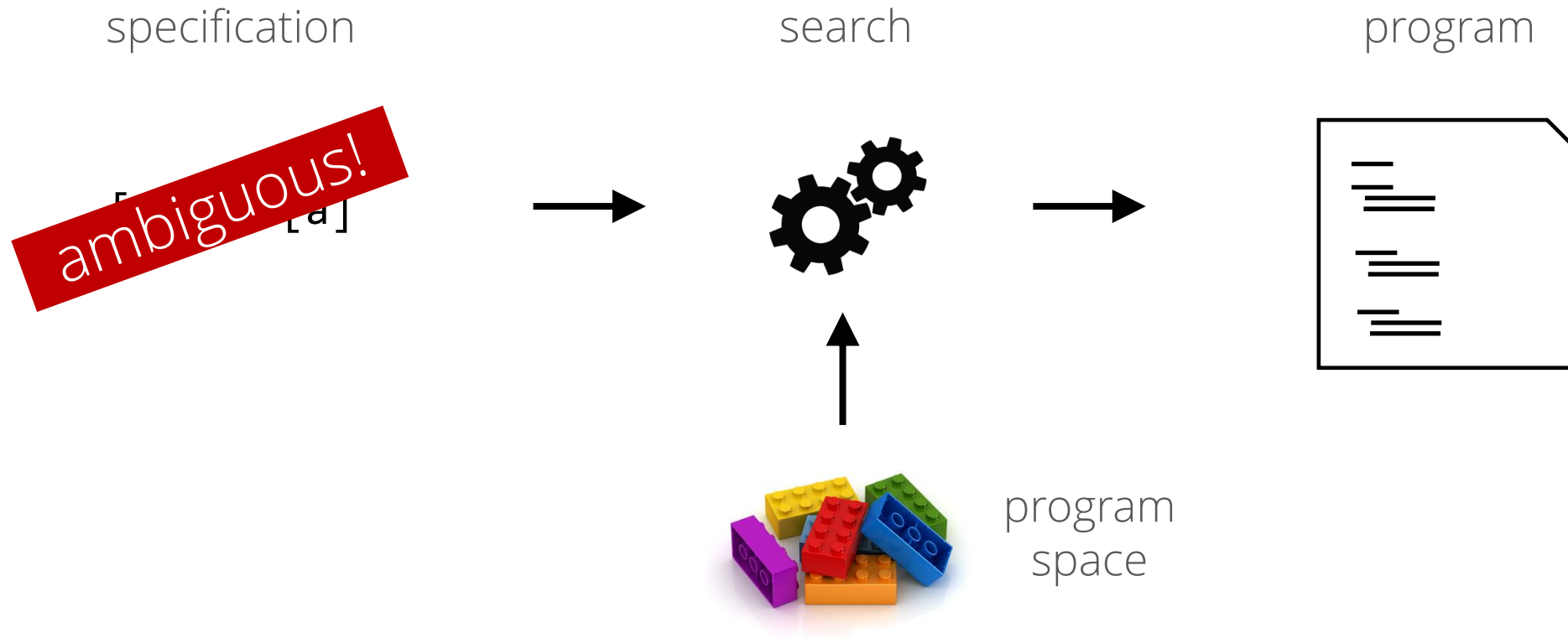


program

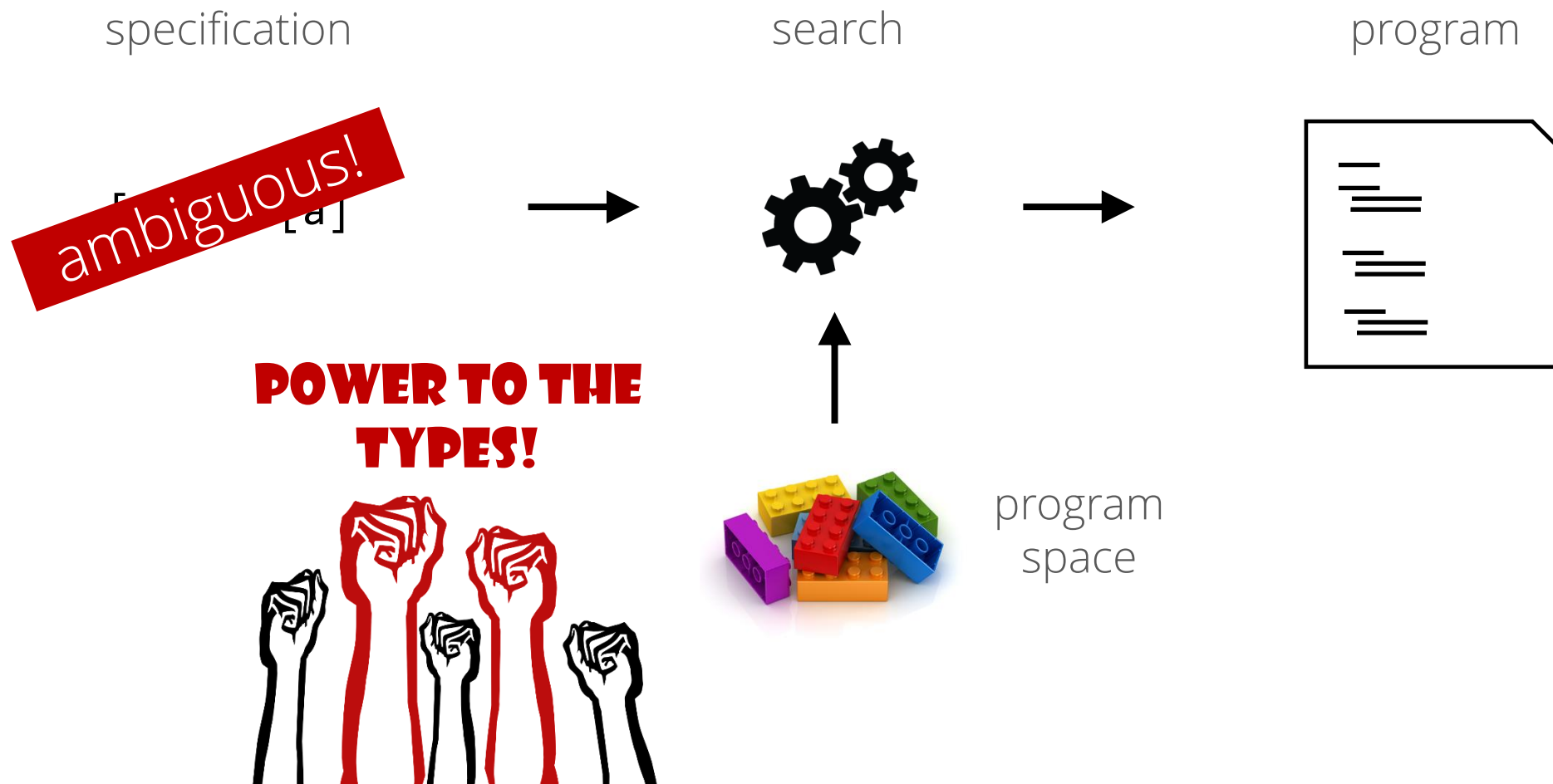


program
space

type-driven synthesis



type-driven synthesis



refinement types

Int

refinement types

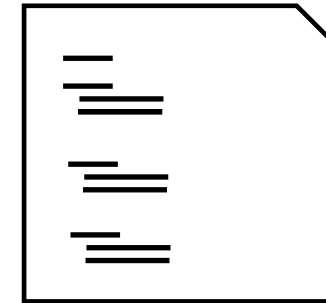
$$\{ \textcolor{red}{v}:\text{Int} \mid \textcolor{red}{0} \leq \textcolor{red}{v} \}$$

Synquid

specification

program

refinement
types



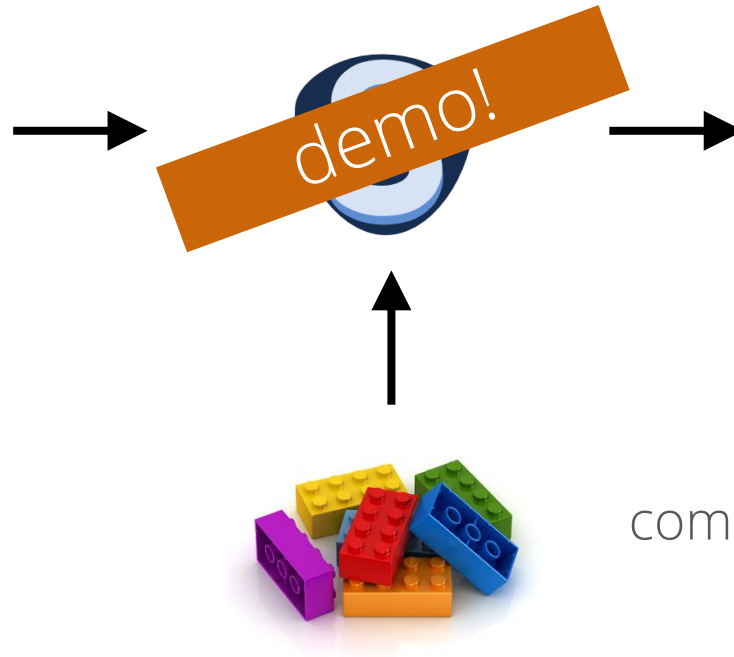
components

Synquid

specification

program

refinement
types



components

types: discussion

Domains:

- API discovery: [Sypet](#), [Hoogle+](#)
- verified functional programs: [Synquid](#)

types: discussion

Domains:

- API discovery: [Sypet](#), [Hoogle+](#)
- verified functional programs: [Synquid](#)

- + programmer friendly
 - + concise
- + automatic checking
 - + can express non-functional properties
 - + guide search

- simple types are ambiguous
- advanced types require expertise
- some things are hard to express / check

this talk

specifications

1. examples (PBE)
2. programs
3. types
4. natural language

Regel

[[Chen et al, 2020](#)]

“I need a regular expression that validates **Decimal(18, 3)**, which means the max number of digits before comma is 15 then accept at max 3 numbers after the comma.”

Regel

[[Chen et al, 2020](#)]

"I need a regular expression that validates **Decimal(18, 3)**, which means the max number of digits before comma is 15 then accept at max 3 numbers after the comma."

Positive examples:

123456789.123

12345.1

Negative examples:

1234567891234567

123.1234

Regel

[[Chen et al, 2020](#)]

"I need a regular expression that validates **Decimal(18, 3)**, which means the max number of digits before comma is 15 then accept at max 3 numbers after the comma."



```
Concat(  
  RepeatRange(<num>,1,15),  
  Optional(Concat(  
    <.>,  
    RepeatRange(<num>,1,3))
```

Positive examples:

123456789.123

12345.1

Negative examples:

1234567891234567

123.1234

natural language: discussion

Domains:

- queries (SQL, regex): [SQLizer](#), [Regel](#)
- programmer's assistant: [SWIM](#), GPT-3?

natural language: discussion

Domains:

- queries (SQL, regex): [SQLizer](#), [Regel](#)
- programmer's assistant: [SWIM](#), GPT-3?

+ beginner-friendly
+ expressive

- ambiguous
- verbose

specifications: conclusions

synthesis is more than just PBE/PBD!

specifications: conclusions

synthesis is more than just PBE/PBD!

depends on target domain and audience

targeting programmers? can use programs or types!

specifications: conclusions

synthesis is more than just PBE/PBD!

depends on target domain and audience

targeting programmers? can use programs or types!

best results when combined

types + examples, NL + examples

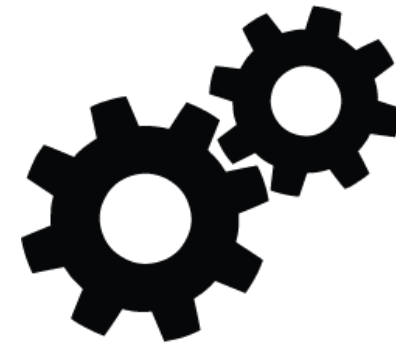
help eliminate ambiguity

this talk

specifications



search strategies



this talk

specifications



search strategies



program synthesis

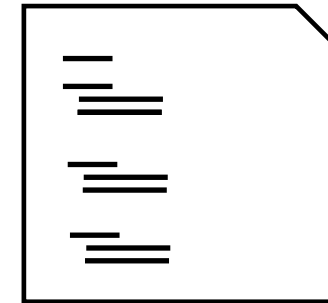
specification



search



program



program
space

program synthesis

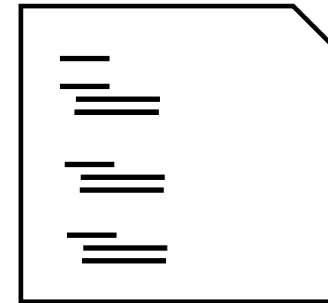
specification



search



program



DSL

program
space



context-free grammars (CFGs)

a grammar of list expressions

```
L ::= sort(L)      |  
      L[N..N]      |  
      L + L        |  
      [N]           |  
      X  
N ::= find(L,N)    |  
      Ø
```

context-free grammars (CFGs)

a grammar of list expressions

terminals

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

context-free grammars (CFGs)

a grammar of list expressions

	$L ::= \text{sort}(L)$	
	$L[N..N]$	
terminals	$L + L$	
	$[N]$	
	x	
nonterminals	$N ::= \text{find}(L, N)$	
	$\emptyset$	

context-free grammars (CFGs)

a grammar of list expressions

terminals

nonterminals

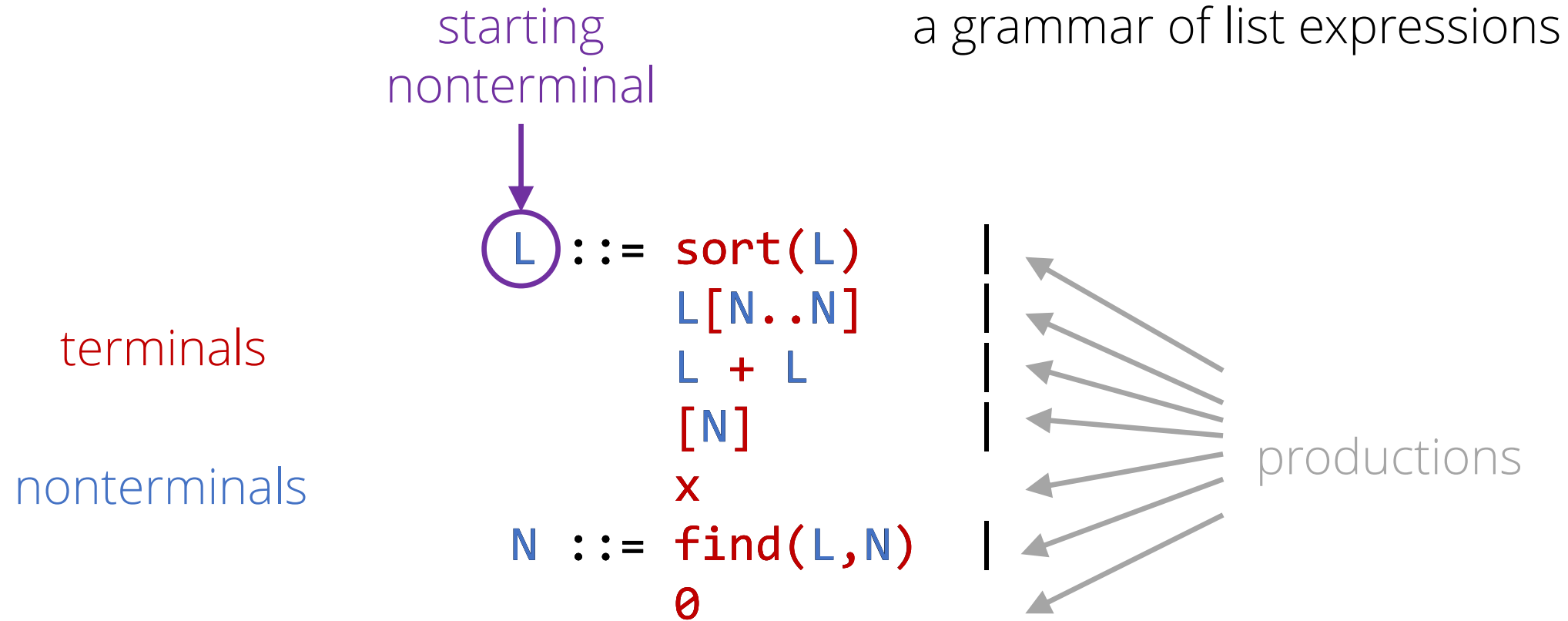
```

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

```



context-free grammars (CFGs)



this talk

specifications



search strategies

1. enumerative
2. VSA & co
3. stochastic
4. constraint-based

this talk

search strategies

1. enumerative
2. VSA & co
3. stochastic
4. constraint-based

enumerative search

idea: sample programs from the grammar one by one
and check if they satisfy the spec

challenge: how do we systematically enumerate all programs?

enumerative search

idea: sample programs from the grammar one by one
and check if they satisfy the spec

challenge: how do we systematically enumerate all programs?

bottom-up **vs** top-down

bottom-up enumeration

start from variables / literals
combine sub-programs 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: example

Program bank

height 0:

```
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

height 0:

```
L ::= sort(L)      |  
    L[N..N]        |  
    L + L          |  
    [N]            |  
    x               |  
N ::= find(L,N)    |  
    0               |  
[[1,4,0,6] → [1,4]]
```



bottom-up: example

Program bank

height 0: x

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



[[1,4,0,6] → [1,4]

bottom-up: example

Program bank

height 0: x

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

[[1,4,0,6]] → [1,4]



bottom-up: example

Program bank

height 0: x 0

```
L ::= sort(L)      |  
    L[N..N]        |  
    L + L          |  
    [N]            |  
    x              |  
N ::= find(L,N)    |  
    0              |  
[[1,4,0,6] → [1,4]] ←
```

bottom-up: example

Program bank

height 0: x 0

height 1:

```
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

height 0: x 0

height 1:

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

[[1,4,0,6] → [1,4]

bottom-up: example

Program bank

height 0: x 0

height 1: sort(x)

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

[[1,4,0,6] → [1,4]

bottom-up: example

Program bank

height 0: `x` `0`

height 1: `sort(x)`

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

`[[1,4,0,6]] → [1,4]`

bottom-up: example

Program bank

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]`

```
L ::= sort(L)      |  
      L[N..N]      | ←  
      L + L        |  
      [N]          |  
      x            |  
N ::= find(L,N)    |  
      0            |  
[[1,4,0,6] → [1,4]]
```

bottom-up: example

Program bank

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]`

```
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

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x`

```
L ::= sort(L)      |  
      L[N..N]      |  
      L + L        | ←  
      [N]           |  
      x             |  
N ::= find(L,N)    |  
      0             |  
[[1,4,0,6] → [1,4]]
```

bottom-up: example

Program bank

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x`

```
L ::= sort(L)      |  
      L[N..N]      |  
      L + L        |  
      [N]           | ←  
      x  
N ::= find(L,N)    |  
      0  
[[1,4,0,6] → [1,4]]
```

bottom-up: example

Program bank

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`

```
L ::= sort(L)      |
      L[N..N]      |
      L + L        | ←
      [N]          |
      x            |
N ::= find(L,N)    |
      0            |
[[1,4,0,6] → [1,4]]
```

bottom-up: example

Program bank

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`

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

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

bottom-up: example

Program bank

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

```
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

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

height 2:

```
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

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

height 2:

```
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

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

height 2: `sort(sort(x))` `sort(x[0..0])` `sort(x + x)`
`sort([0])`

```
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

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

height 2: `sort(sort(x))` `sort(x[0..0])` `sort(x + x)`
`sort([0])`

```
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

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

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

```
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

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

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

```
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

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

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

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

`[[1,4,0,6]] → [1,4]`



bottom-up: example

Program bank

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

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

...

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

`[[1,4,0,6]] → [1,4]`



bottom-up: example

Program bank

height 0: `x` `0`

height 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

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

...

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

`[[1,4,0,6]] → [1,4]`



top-down enumeration

start from the start non-terminal
expand remaining non-terminals
using productions

$$\begin{array}{lcl} L & ::= & L[N..N] \quad | \\ & & x \\ N & ::= & \text{find}(L, N) \quad | \\ & & \emptyset \end{array}$$
$$[[1, 4, \emptyset, 6] \rightarrow [1, 4]]$$

top-down: example

Worklist

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

```
[[1,4,0,6]] → [[1,4]]
```

top-down: example

Worklist

L

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

x

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

$\emptyset$

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

top-down: example

Worklist

L

iter 1:

L ::= L[N..N] |

x

N ::= find(L,N) |

0

[[1,4,0,6] → [1,4]]

top-down: example

Worklist

L

iter 1:

```
L ::= L[N..N] |  
      x  
N ::= find(L,N) | ←  
      0  
[[1,4,0,6] → [1,4]]
```

top-down: example

Worklist

L

iter 1: x

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



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

top-down: example

Worklist

L

iter 1: x

L ::= L[N..N] | ←

N ::= ^xfind(L,N) |
 ₀

[[1,4,0,6] → [1,4]]

top-down: example

Worklist

iter 1:

L
x L[N..N]

L ::= L[N..N] | ←

N ::= ^xfind(L,N) |
 ⁰

[[1,4,0,6] → [1,4]]

top-down: example

Worklist

L

iter 1:

x

L[N..N]

iter 2:

L ::= L[N..N] |

x

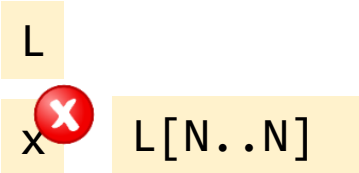
N ::= find(L,N) |

0

[[1,4,0,6] → [1,4]]

top-down: example

Worklist

iter 1: 

iter 2:

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

x

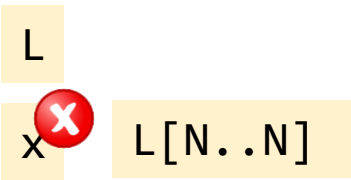
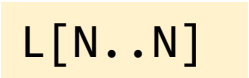
$N ::= \text{find}(L, N) \quad |$

$\emptyset$

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

top-down: example

Worklist

iter 1: 
iter 2: 

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



top-down: example

Worklist

L

iter 1: x L[N..N]

iter 2: L[N..N]

iter 3:

L ::= L[N..N] |

x

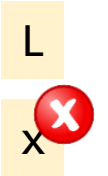
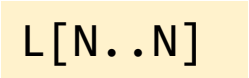
N ::= find(L,N) |

0

[[1,4,0,6] → [1,4]]

top-down: example

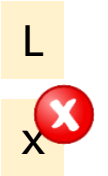
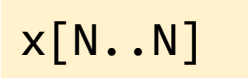
Worklist

iter 1:  
iter 2: 
iter 3:

```
L ::= L[N..N] |  
      x  
N ::= find(L,N) | ←  
      0  
  
[[1,4,0,6] → [1,4]]
```

top-down: example

Worklist

iter 1:  
iter 2: 
iter 3: 

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

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

top-down: example

Worklist

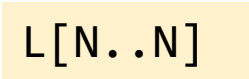
L
iter 1: x  L[N..N]
iter 2: L[N..N]
iter 3: x[N..N]

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

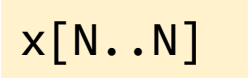
[[1, 4, 0, 6] → [1, 4]]

top-down: example

Worklist

iter 1:   

iter 2: 

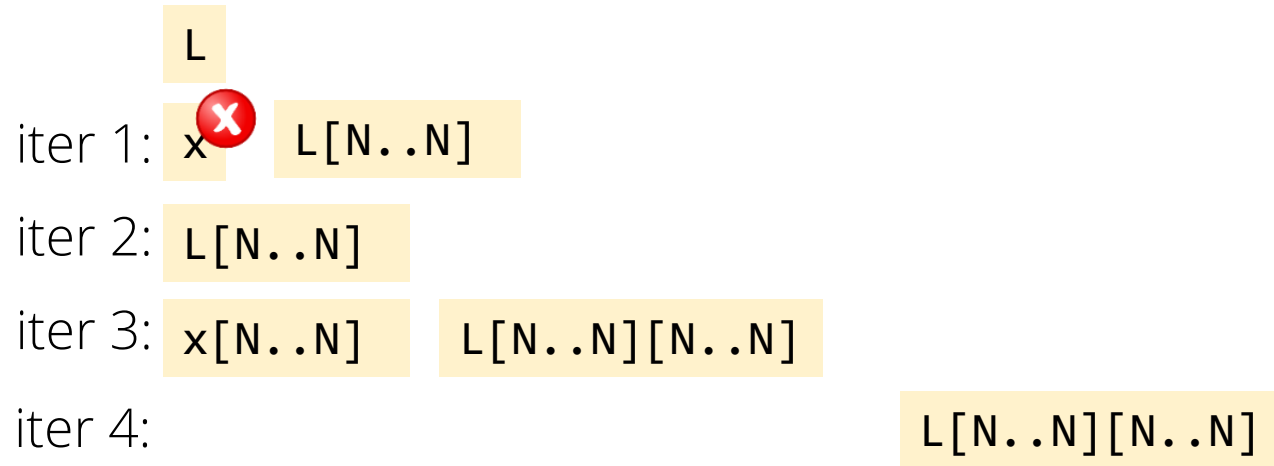
iter 3:  

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

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

top-down: example

Worklist

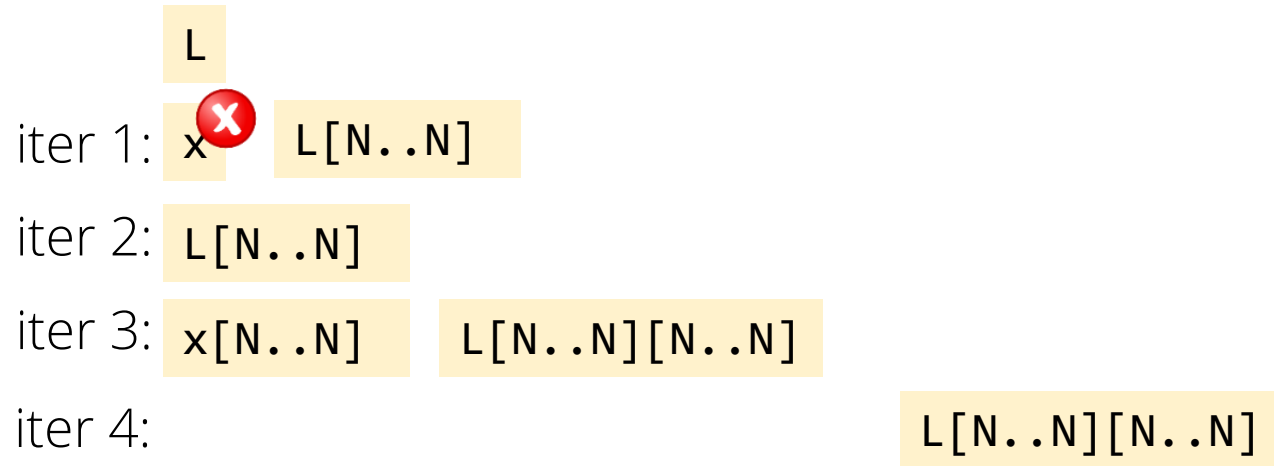


```
L ::= L[N..N] |  
      x  
N ::= find(L,N) |  
      0
```



top-down: example

Worklist



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





top-down: example

Worklist

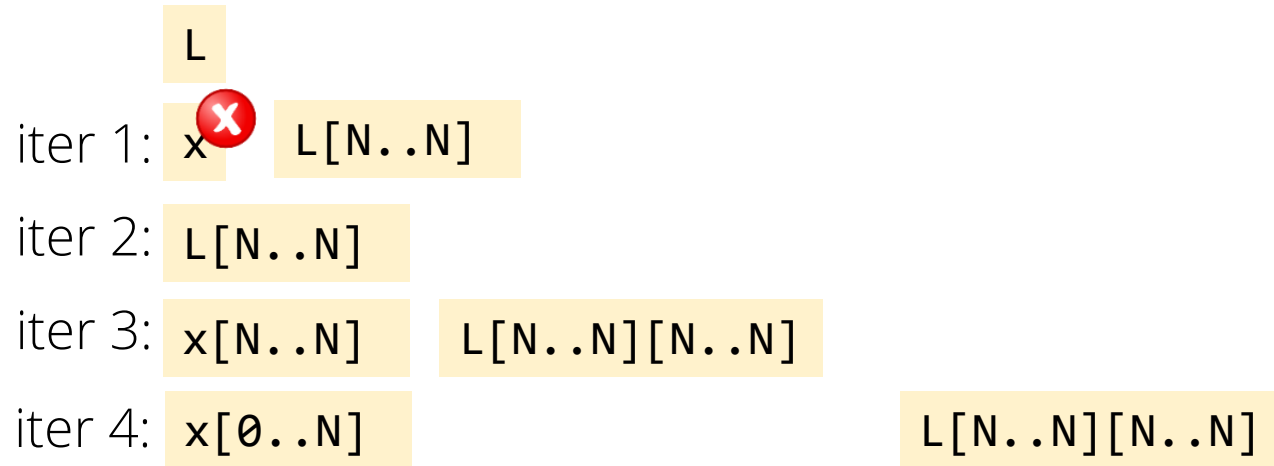


Diagram illustrating the top-down process with a worklist:

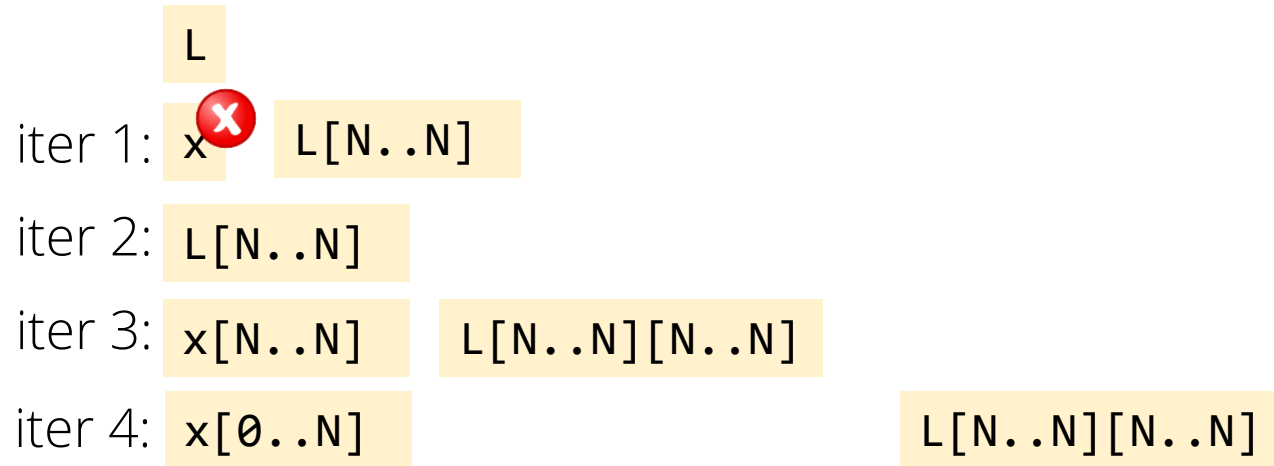
```
L ::= L[N..N] |  
      x  
N ::= find(L, N) |  
      0
```

←

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

top-down: example

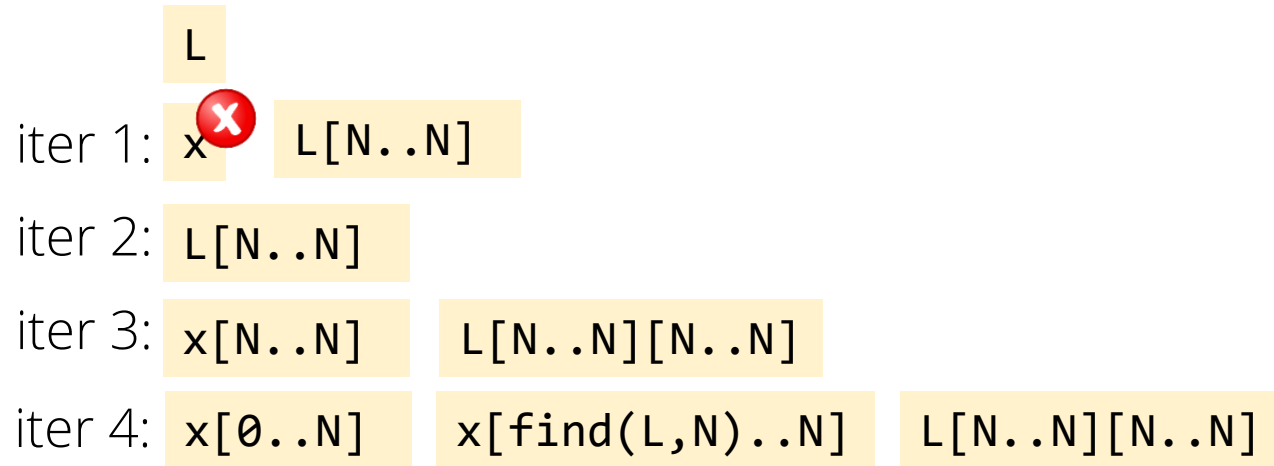
Worklist



```
L ::= L[N..N] |  
      x  
N ::= find(L, N) | ←  
      0  
[[1, 4, 0, 6] → [1, 4]]
```

top-down: example

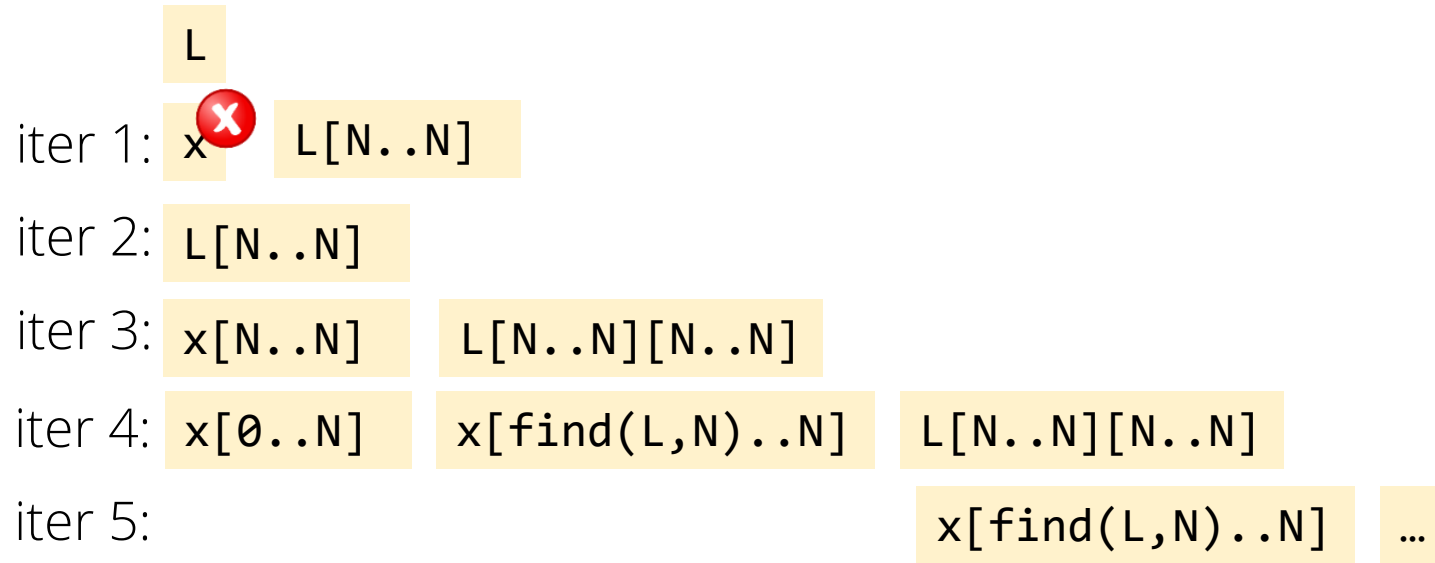
Worklist



```
L ::= L[N..N] |  
      x  
N ::= find(L,N) | ←  
      0  
  
[[1,4,0,6] → [1,4]]
```

top-down: example

Worklist

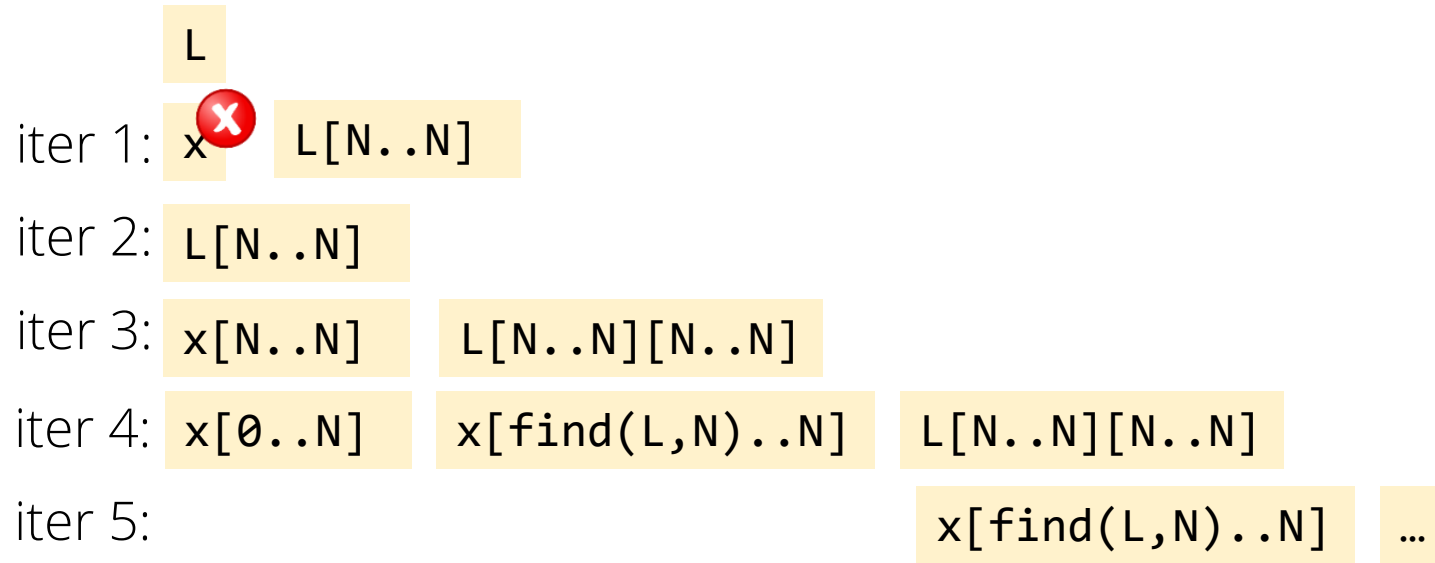


```
L ::= L[N..N] |  
      x  
N ::= find(L,N) |  
      0
```

```
[[1,4,0,6] → [1,4]]
```

top-down: example

Worklist



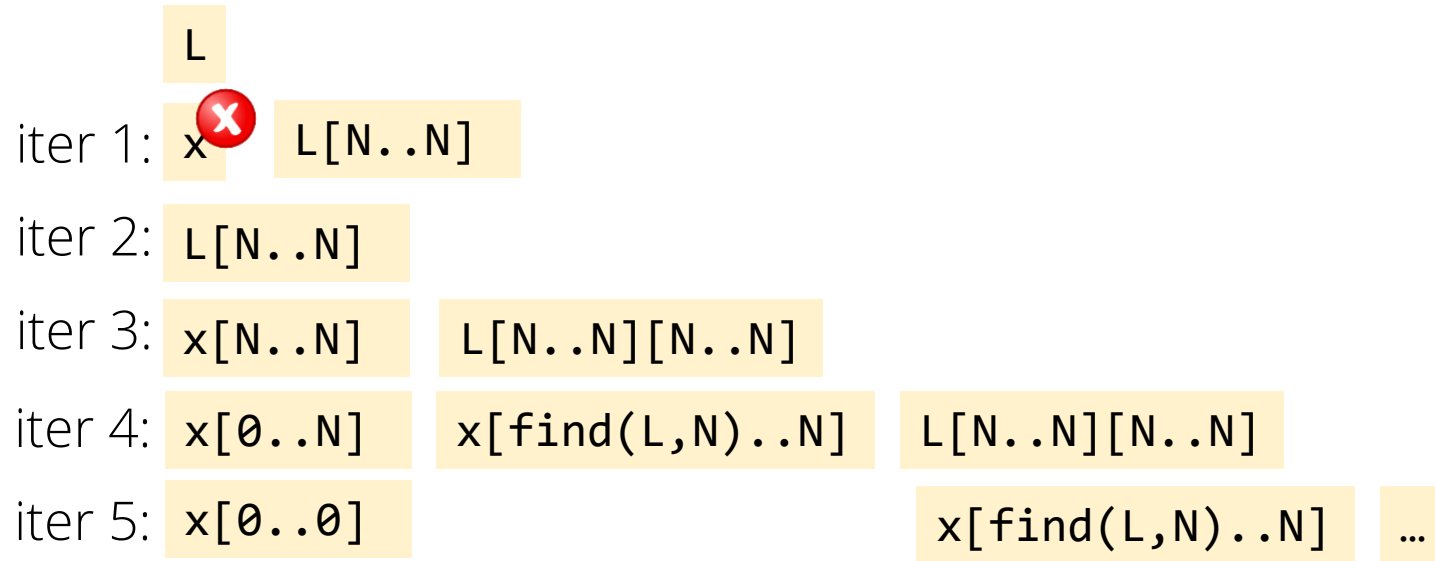
```
L ::= L[N..N] |
      x
N ::= find(L,N) |
      0
```

←

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

top-down: example

Worklist



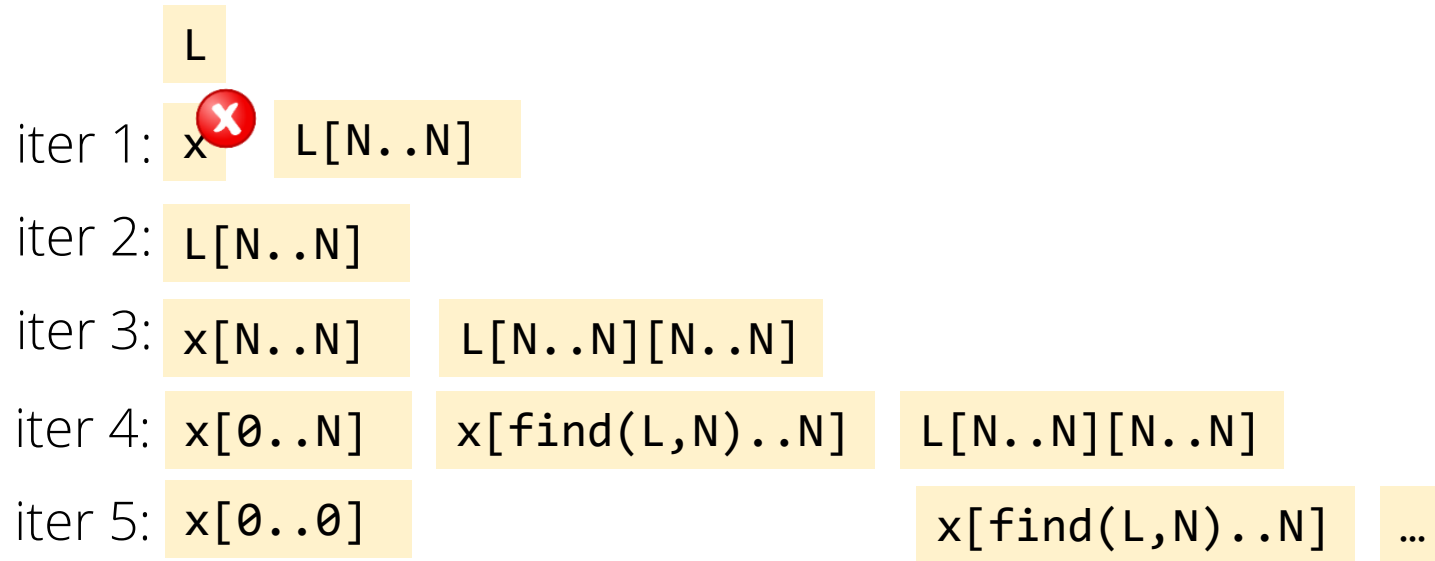
```
L ::= L[N..N] |  
      x  
N ::= find(L, N) |  
      0
```

←

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

top-down: example

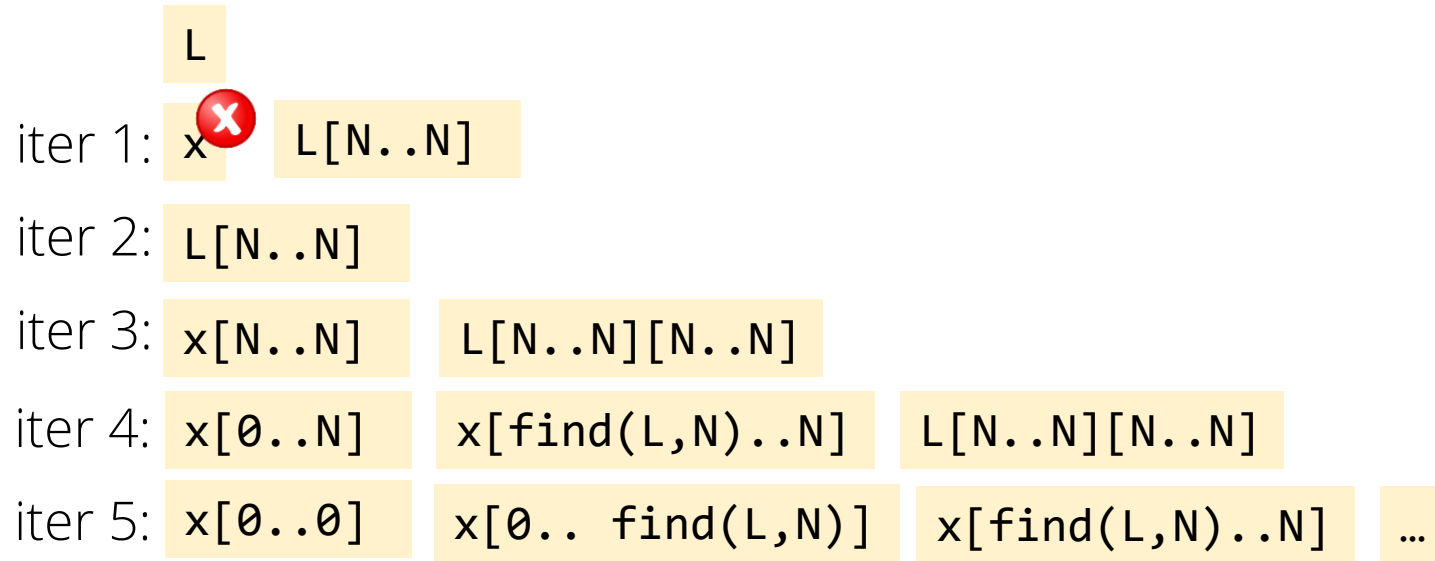
Worklist



```
L ::= L[N..N] |  
      x  
N ::= find(L,N) | ←  
      0  
  
[[1,4,0,6] → [1,4]]
```

top-down: example

Worklist



```
L ::= L[N..N] |  
      x  
N ::= find(L,N) | ←  
      0  
  
[[1,4,0,6] → [1,4]]
```

top-down: example

Worklist

	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:				

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

`[[1,4,0,6]] → [1,4]`

top-down: example

Worklist

L

iter 1: x^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 x[0.. find(L,N)] x[find(L,N)..N] ...

iter 6:

L ::= L[N..N] |

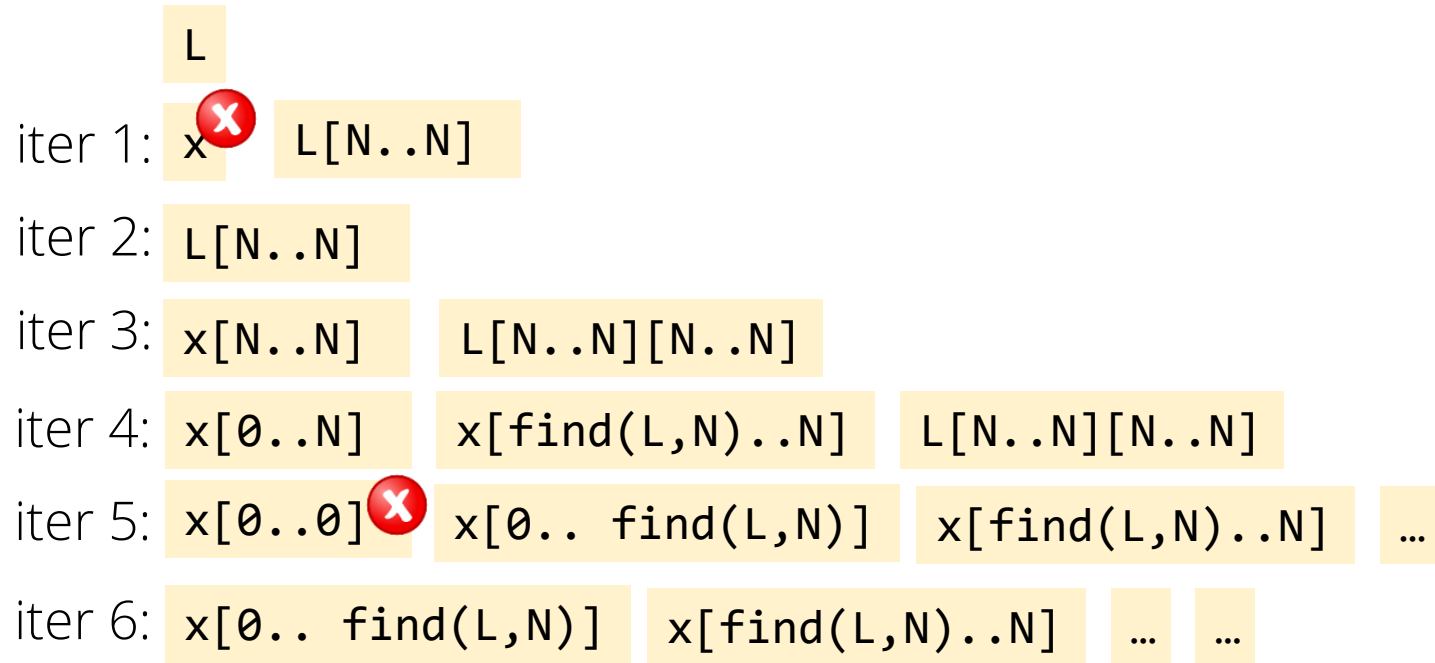
^x
N ::= find(L,N) |

⁰

[[1,4,0,6] → [1,4]]

top-down: example

Worklist



L ::= L[N..N] |

x

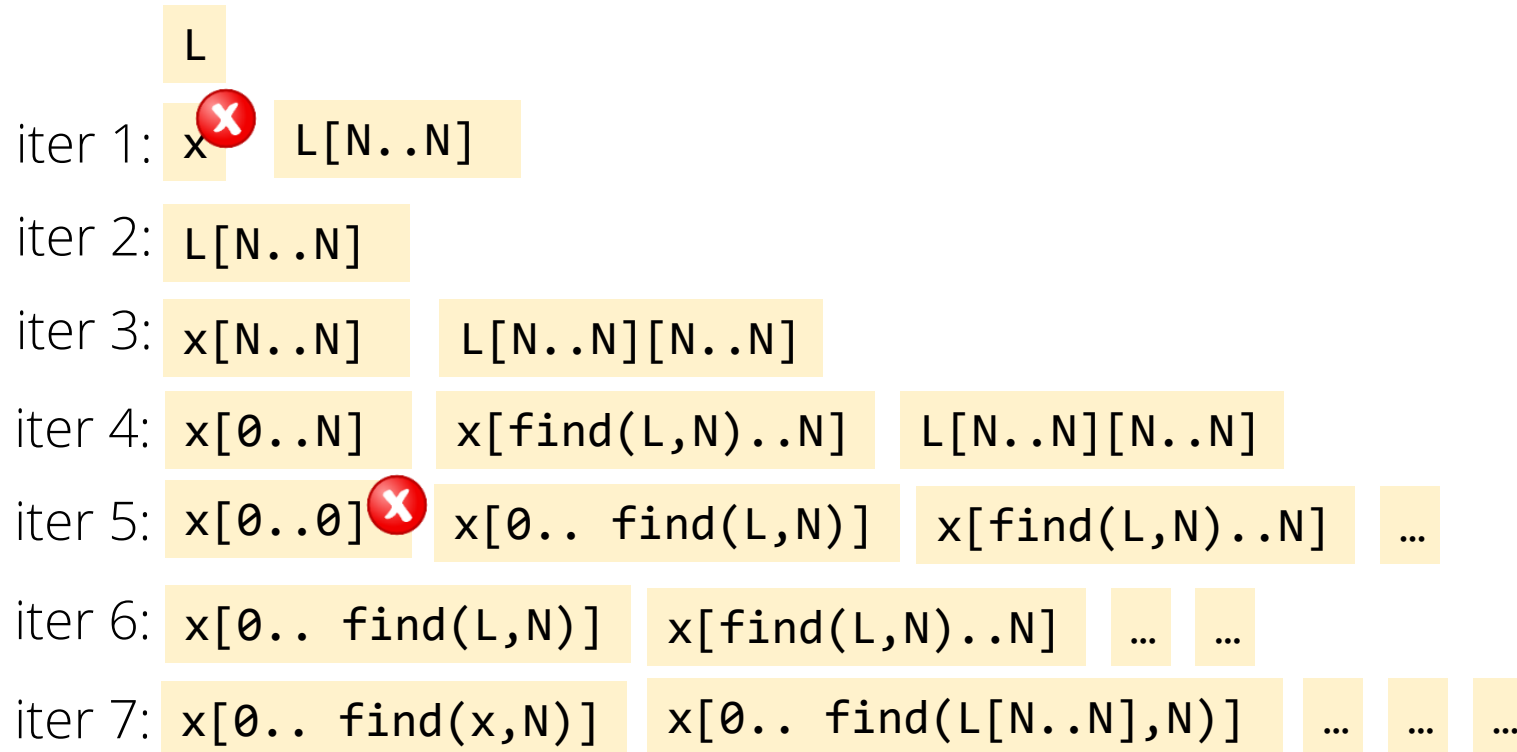
N ::= find(L,N) |

0

[[1,4,0,6] → [1,4]]

top-down: example

Worklist



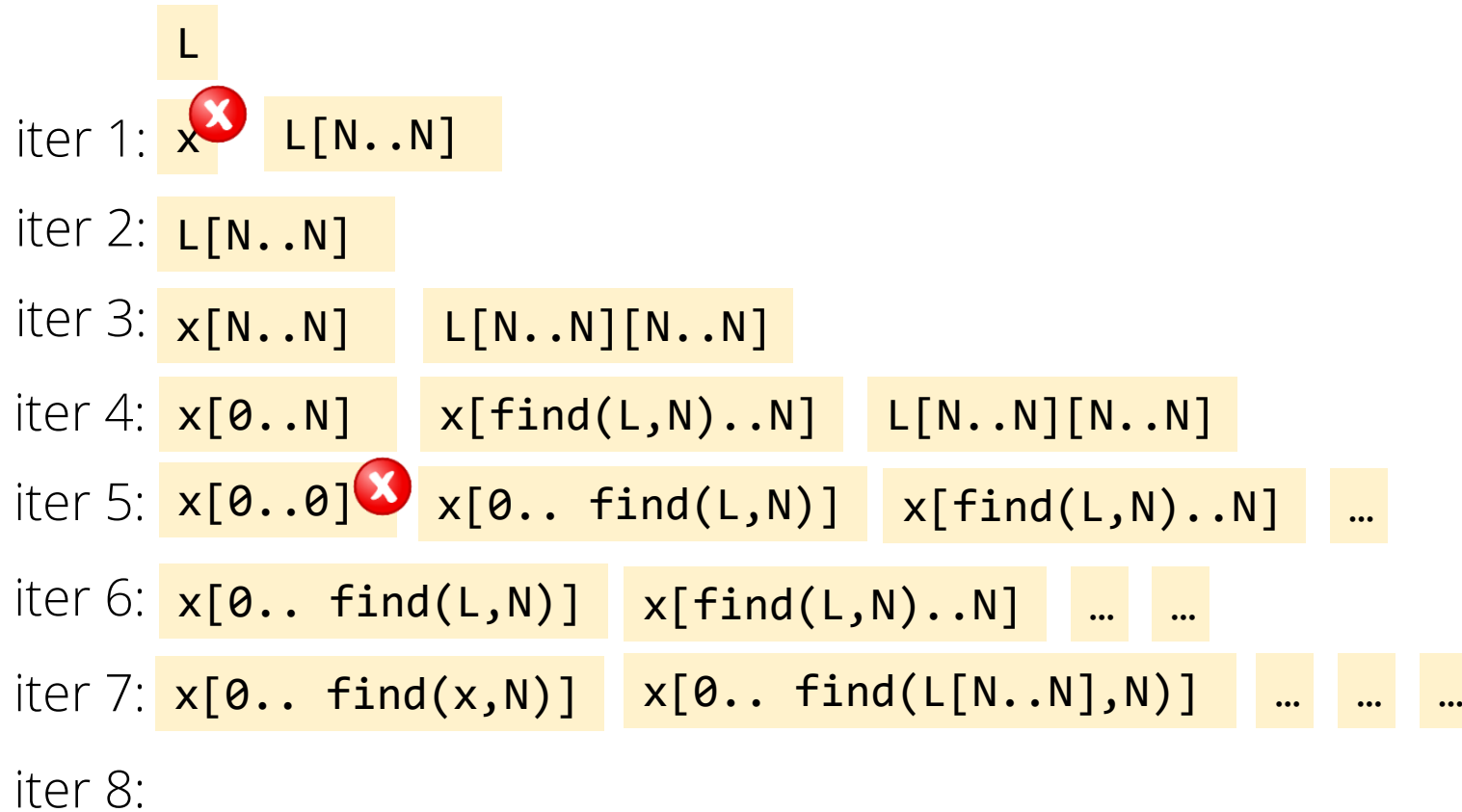
$L ::= L[N..N]$ |

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

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

top-down: example

Worklist

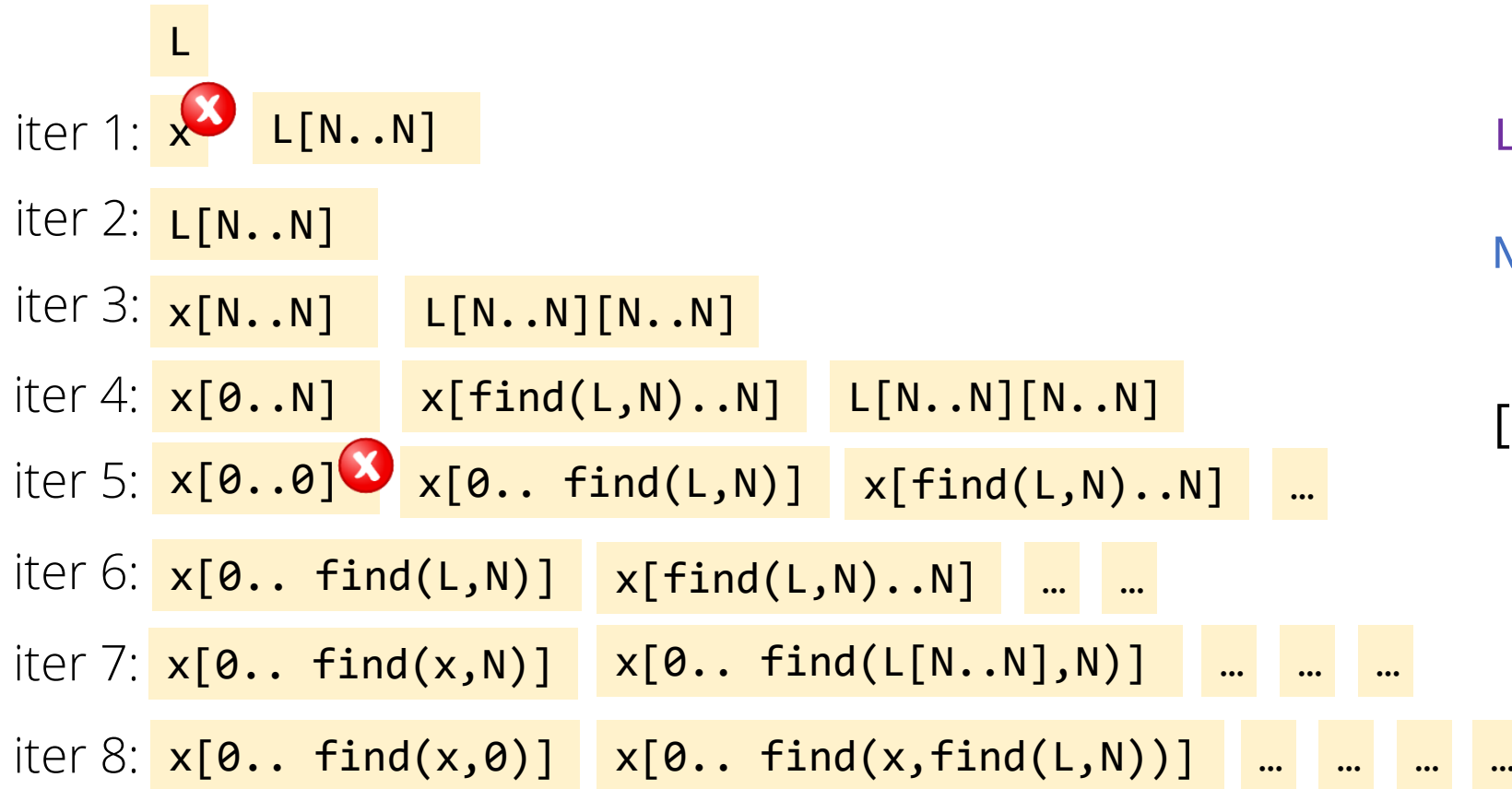


```
L ::= L[N..N] |  
      x  
N ::= find(L,N) |  
      0
```

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

top-down: example

Worklist



L ::= L[N..N] |


N ::= find(L,N) |


 → 

top-down: example

Worklist

L

iter 1: x^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 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:

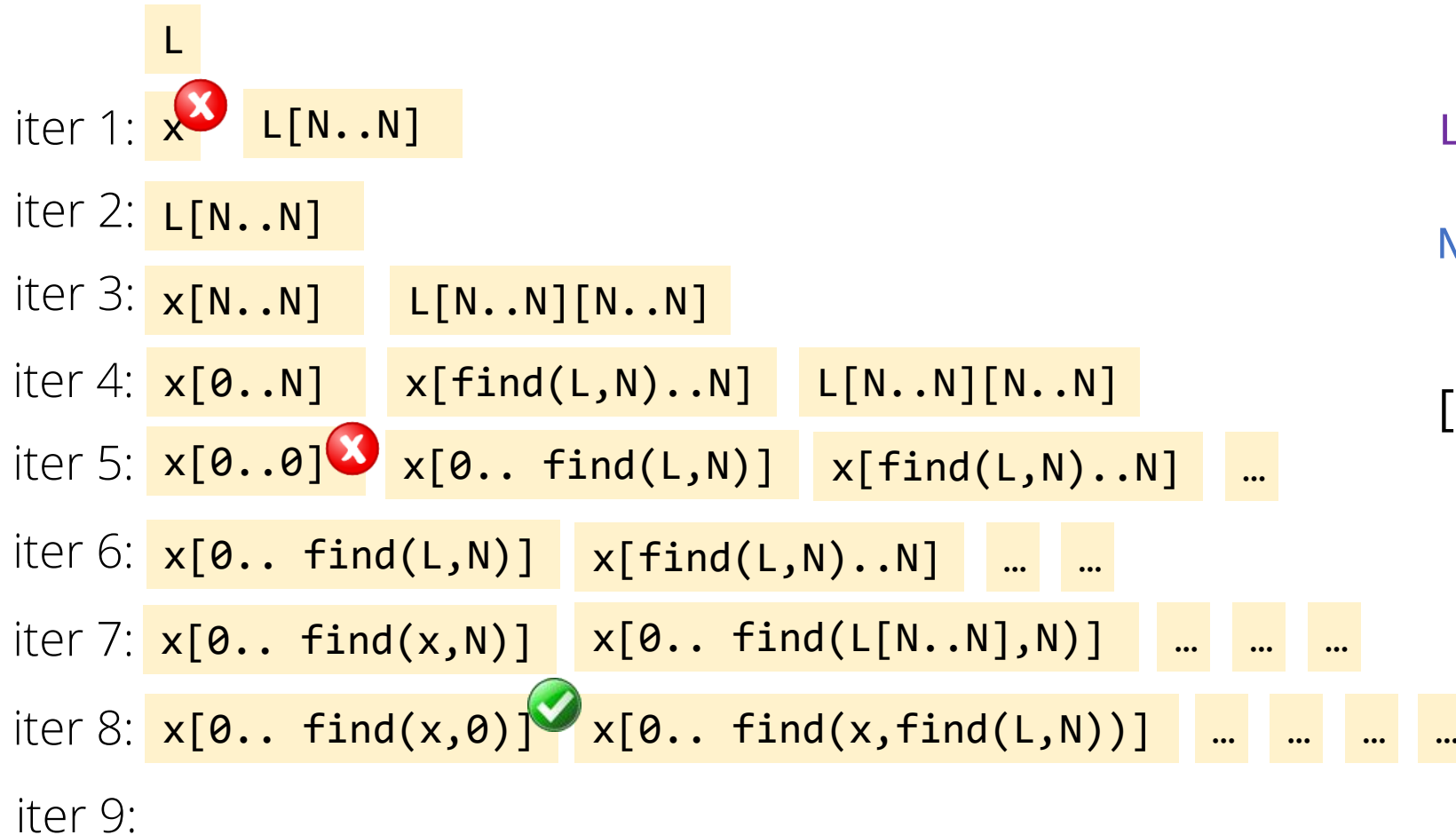
L ::= L[N..N] |

^x
N ::= find(L,N) |
⁰

[[1,4,0,6] → [1,4]]

top-down: example

Worklist



L ::= L[N..N] |


N ::= find(L,N) |


 → 

how to make it scale

Prune

Discard useless subprograms

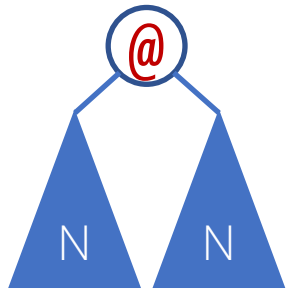
Prioritize

Explore more promising candidates first

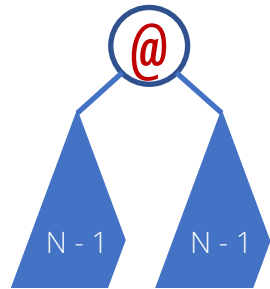
how to make it scale

Prune

Discard useless subprograms



$$m * N^2$$



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

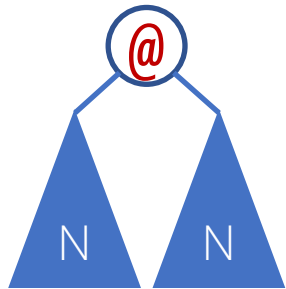
Prioritize

Explore more promising candidates first

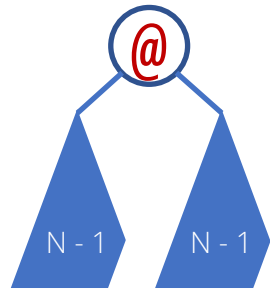
how to make it scale

Prune

Discard useless subprograms



$$m * N^2$$



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

Prioritize

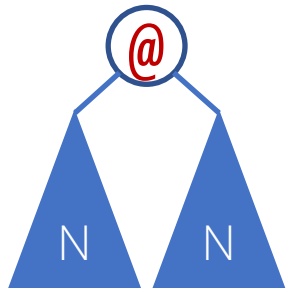
Explore more promising candidates first

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

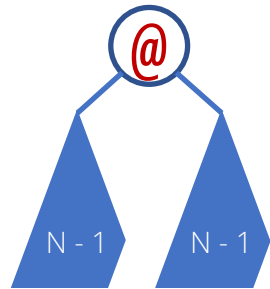
how to make it scale

Prune

Discard useless subprograms



$$m * N^2$$



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

Prioritize

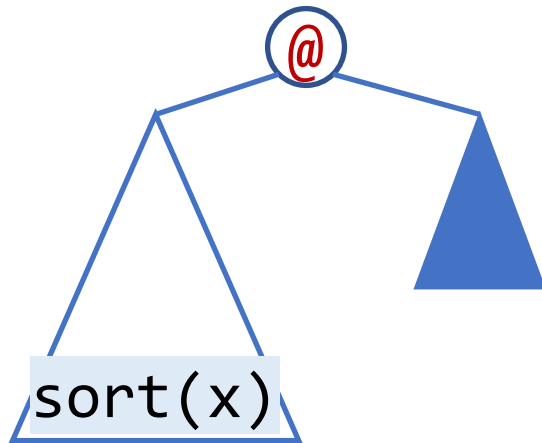
Explore more promising candidates first

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

when can we discard a subprogram?

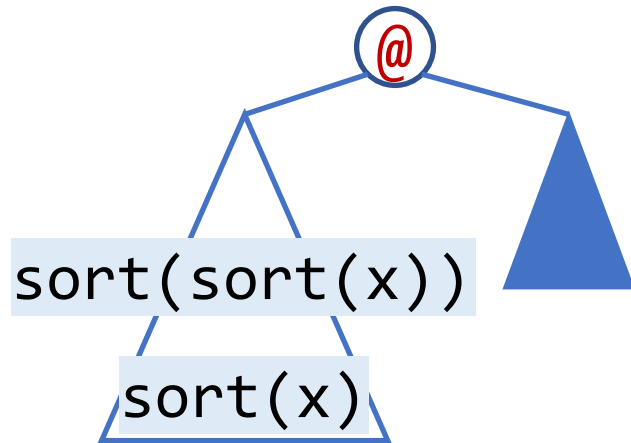
when can we discard a subprogram?

It's equivalent to something we have already explored



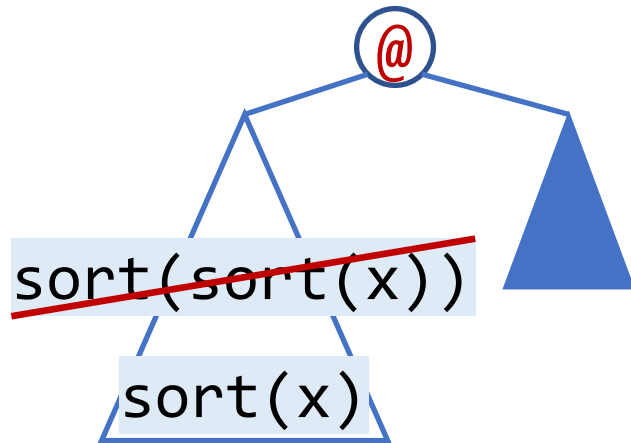
when can we discard a subprogram?

It's equivalent to something we have already explored



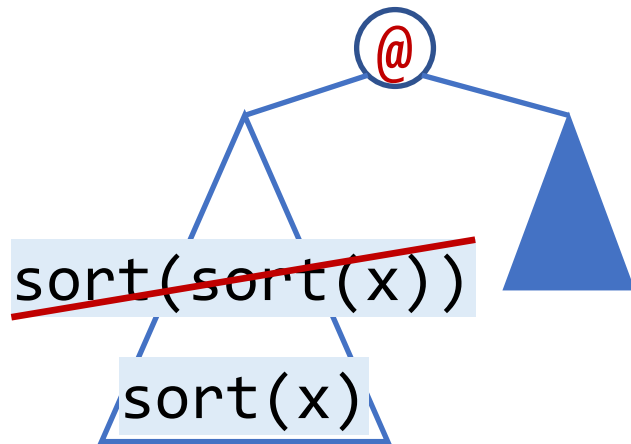
when can we discard a subprogram?

It's equivalent to something we have already explored



when can we discard a subprogram?

It's equivalent to something we have already explored

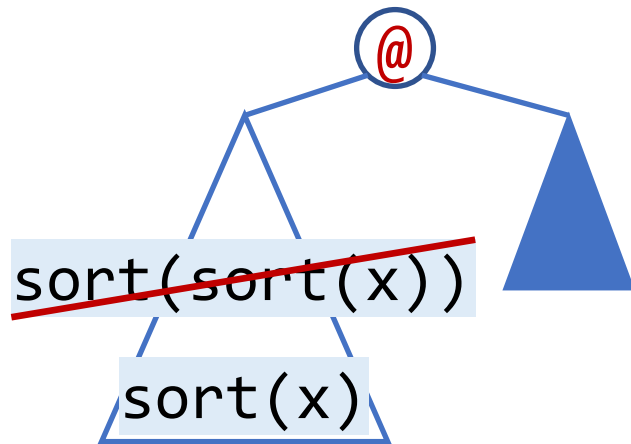


Equivalence reduction

when can we discard a subprogram?

It's equivalent to something we have already explored

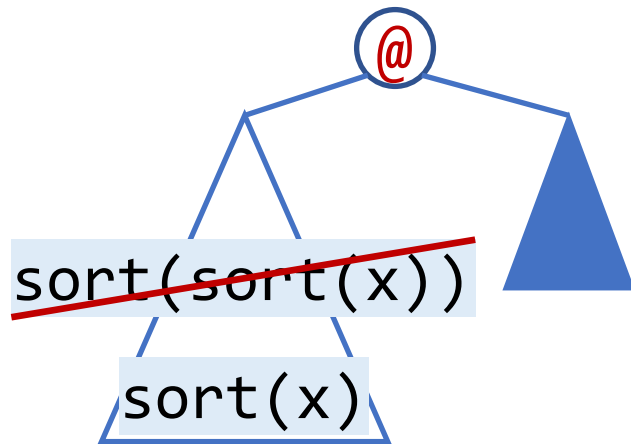
No matter what we combine it with, it cannot satisfy the spec



Equivalence reduction

when can we discard a subprogram?

It's equivalent to something we have already explored



Equivalence reduction

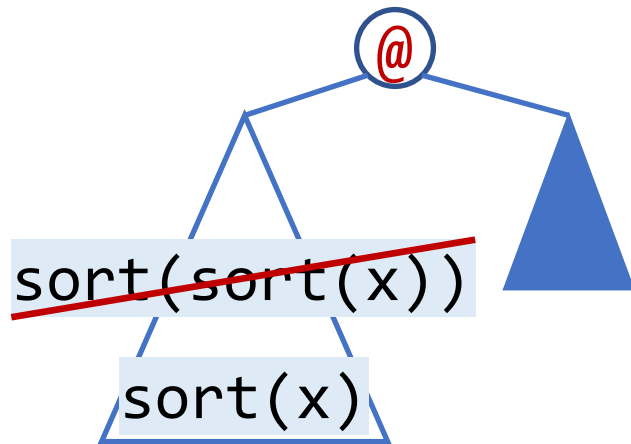
No matter what we combine it with, it cannot satisfy the spec

L

[] → []
...

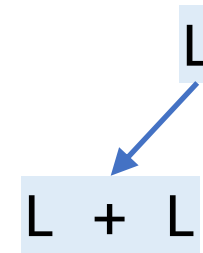
when can we discard a subprogram?

It's equivalent to something we have already explored



Equivalence reduction

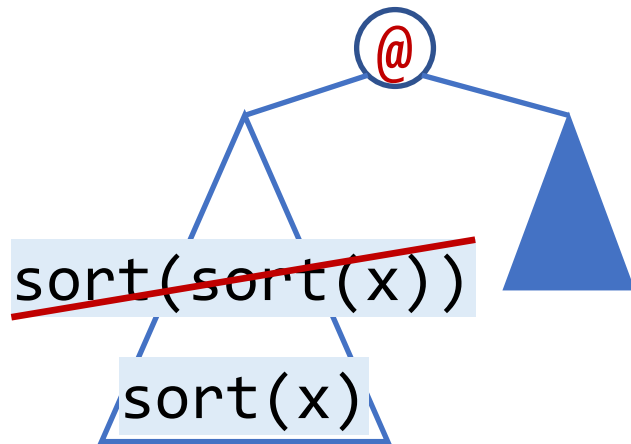
No matter what we combine it with, it cannot satisfy the spec



[] → []
...

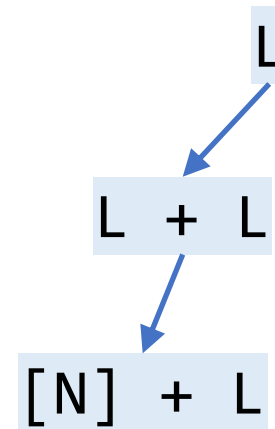
when can we discard a subprogram?

It's equivalent to something we have already explored



Equivalence reduction

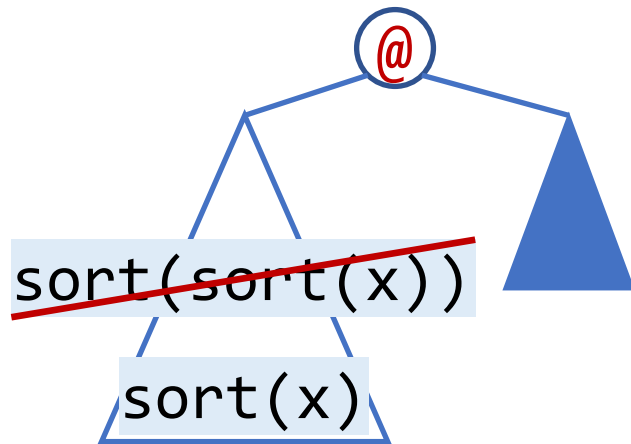
No matter what we combine it with, it cannot satisfy the spec



[] → []
...

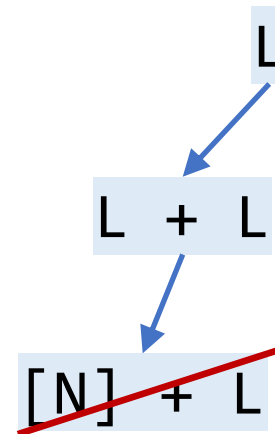
when can we discard a subprogram?

It's equivalent to something we have already explored



Equivalence reduction

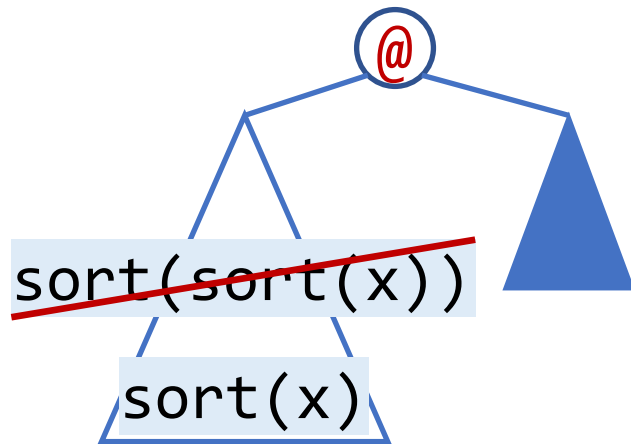
No matter what we combine it with, it cannot satisfy the spec



[] → []
...

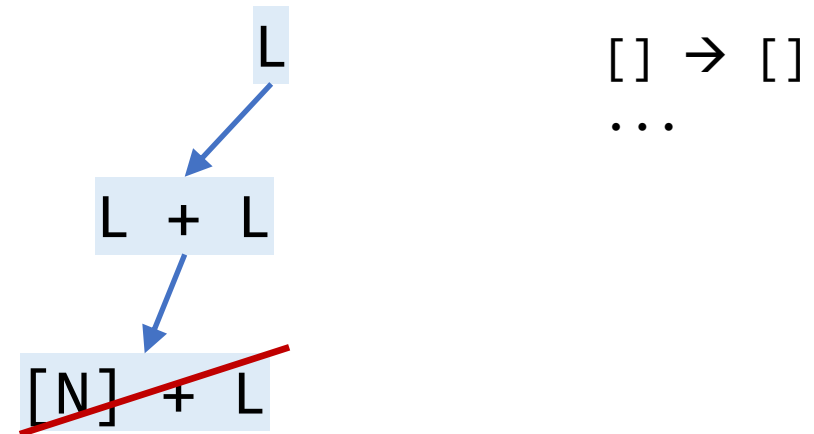
when can we discard a subprogram?

It's equivalent to something we have already explored



Equivalence reduction

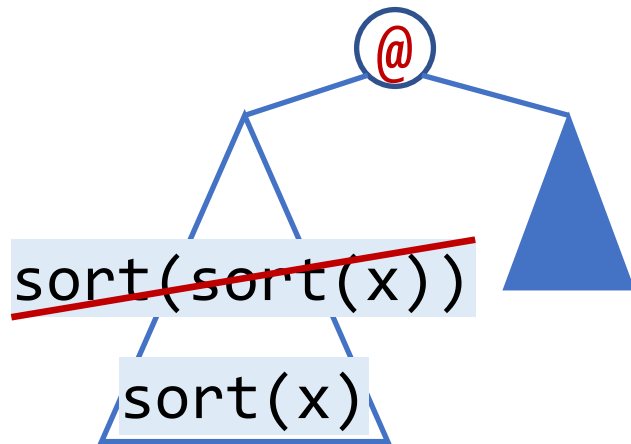
No matter what we combine it with, it cannot satisfy the spec



Top-down propagation

when can we discard a subprogram?

It's equivalent to something we have already explored



Equivalence reduction

equivalent programs

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

equivalent programs

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

equivalent programs

x 0

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

bottom_up
→

equivalent programs

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

bottom_up
→

x 0

sort(x)

x[0..0]

x + x

[0]

find(x,0)

equivalent programs

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

bottom_up



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

equivalent programs

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

bottom_up

x 0

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

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

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

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

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

...

equivalent programs

```

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

bottom_up
→

```

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

equivalent programs

```

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

bottom_up

How do we check program
equivalence?

x 0

sort(x)

x[0..0]

x + x

[0]

find(x,0)

x + (x + x)

x + [0]

sort(x) + x

[0] + x

x + sort(x)

...

observational equivalence

idea: In PBE we only care about
equivalence on the given inputs!
easy to check efficiently
even more programs are equivalent

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

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

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

observational equivalence

idea: In PBE we only care about
equivalence on the given inputs!
easy to check efficiently
even more programs are equivalent

$[[\textcolor{brown}{0}] \rightarrow [\textcolor{blue}{0}]]$

x θ
 $\text{sort}(x)$ $x[\theta..\theta]$ $x + x$ $[\theta]$ $\text{find}(x, \theta)$

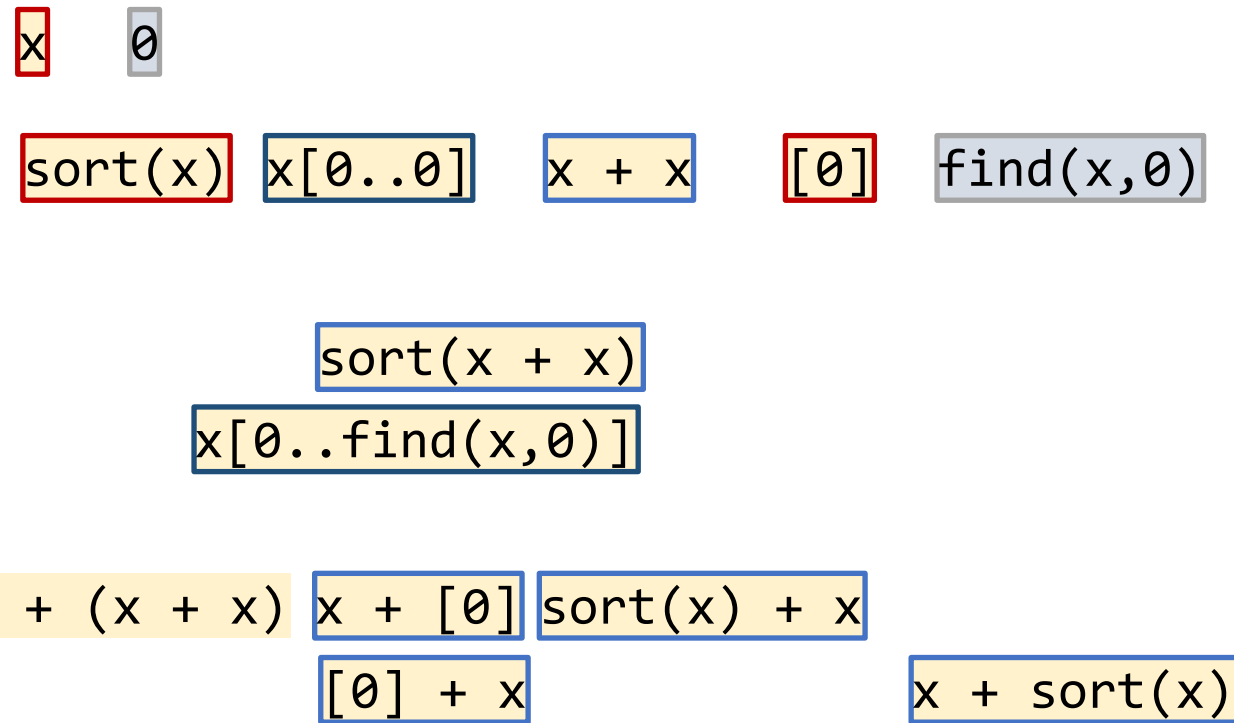
$\text{sort}(x + x)$
 $x[\theta..\text{find}(x, \theta)]$

$x + (x + x)$ $x + [\theta]$ $\text{sort}(x) + x$
 $[\theta] + x$ $x + \text{sort}(x)$

observational equivalence

idea: In PBE we only care about
equivalence on the given inputs!
easy to check efficiently
even more programs are equivalent

$[[\textcolor{brown}{0}]] \rightarrow [\textcolor{blue}{0}]$



observational equivalence

idea: In PBE we only care about
equivalence on the given inputs!
easy to check efficiently
even more programs are equivalent

$[[\textcolor{brown}{0}]] \rightarrow [[\textcolor{blue}{0}]]$

x

$\emptyset$

$x[\emptyset..\emptyset]$

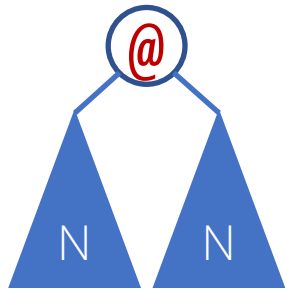
$x + x$

$x + (x + x)$

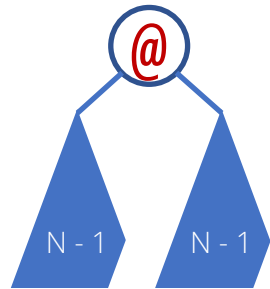
how to make it scale

Prune

Discard useless subprograms



$$m * N^2$$



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

Prioritize

Explore more promising candidates first

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

how to make it scale

Prioritize

Explore more promising candidates first

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

top-down search (revisited)

Turn off the rightmost sequence of **1**s:

00101 → 00100

01010 → 01000

10110 → 10000

top-down search (revisited)

Turn off the rightmost sequence of **1**s:

00101 → 00100

01010 → 01000

10110 → 10000

S	->	0		1		x	
S	+	S					
S	-	S					
S	&	S					
S		S					
S	<<	S					
S	>>	S					

top-down search (revisited)

Turn off the rightmost sequence of **1**s:

00101 → 00100

01010 → 01000

10110 → 10000

S	->	0		1		x	
S	+	S					
S	-	S					
S	&	S					
S		S					
S	<<	S					
S	>>	S					

$x \& (1 + (x | x - 1))$

top-down search (revisited)

Turn off the rightmost sequence of **1**s:

00101 → 00100

01010 → 01000

10110 → 10000

$S \rightarrow$

0		1		x	
S	+	S			
S	-	S			
S	&	S			
S		S			
S	<<	S			
S	>>	S			

S

$x \& (1 + (x | x - 1))$

top-down search (revisited)

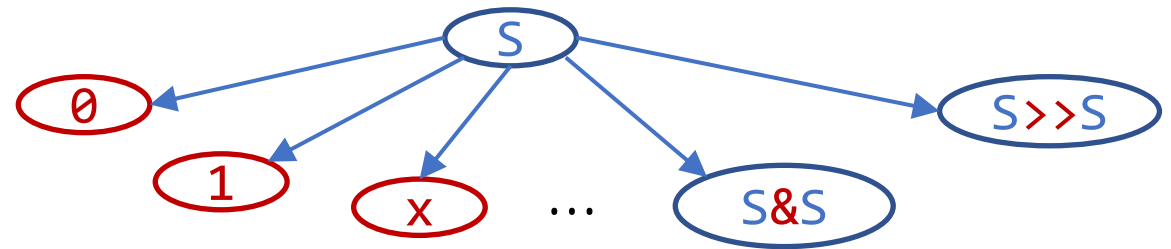
Turn off the rightmost sequence of 1s:

00101 → 00100

01010 → 01000

10110 → 10000

S	->	0		1		x	
S		+		S			
S		-		S			
S		&		S			
S				S			
S		<<		S			
S		>>		S			



$x \& (1 + (x | x - 1))$

top-down search (revisited)

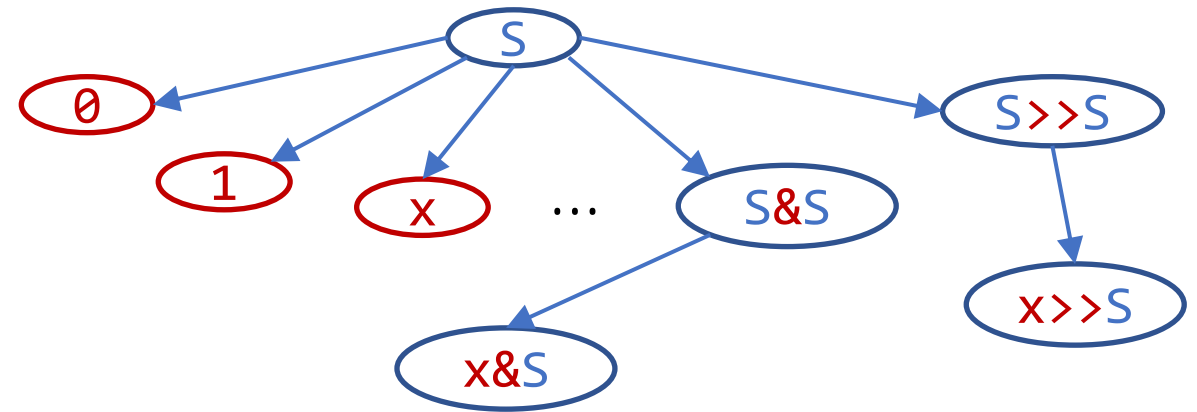
Turn off the rightmost sequence of 1s:

00101 $\rightarrow$ 00100

01010 $\rightarrow$ 01000

10110 $\rightarrow$ 10000

$S \rightarrow$	0		1		x	
	S	+	S			
	S	-	S			
	S	&	S			
	S		S			
	S	<<	S			
	S	>>	S			



$x \& (1 + (x | x - 1))$

top-down search (revisited)

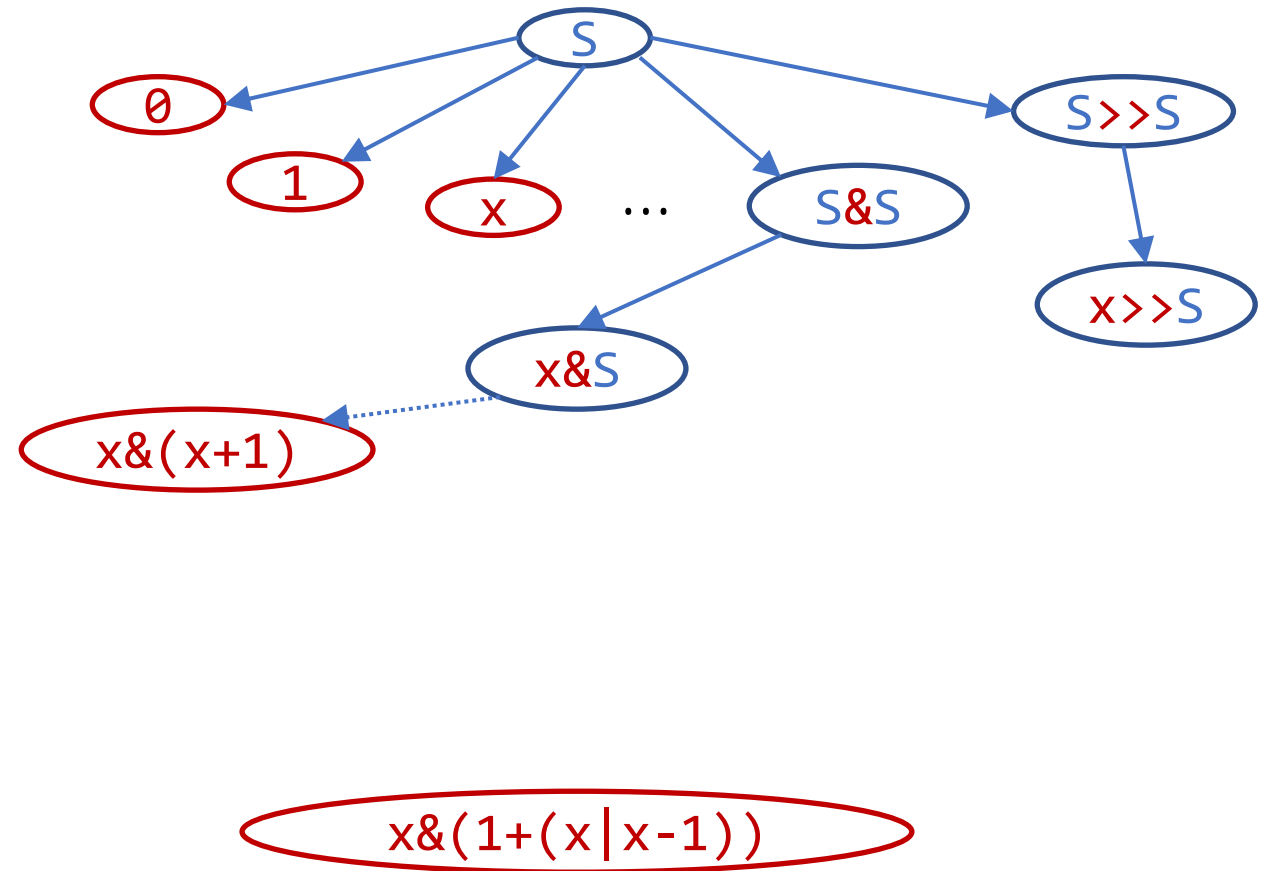
Turn off the rightmost sequence of 1s:

00101 $\rightarrow$ 00100

01010 $\rightarrow$ 01000

10110 $\rightarrow$ 10000

$S \rightarrow$	0		1		x	
	S	+	S			
	S	-	S			
	S	&	S			
	S		S			
	S	<<	S			
	S	>>	S			



top-down search (revisited)

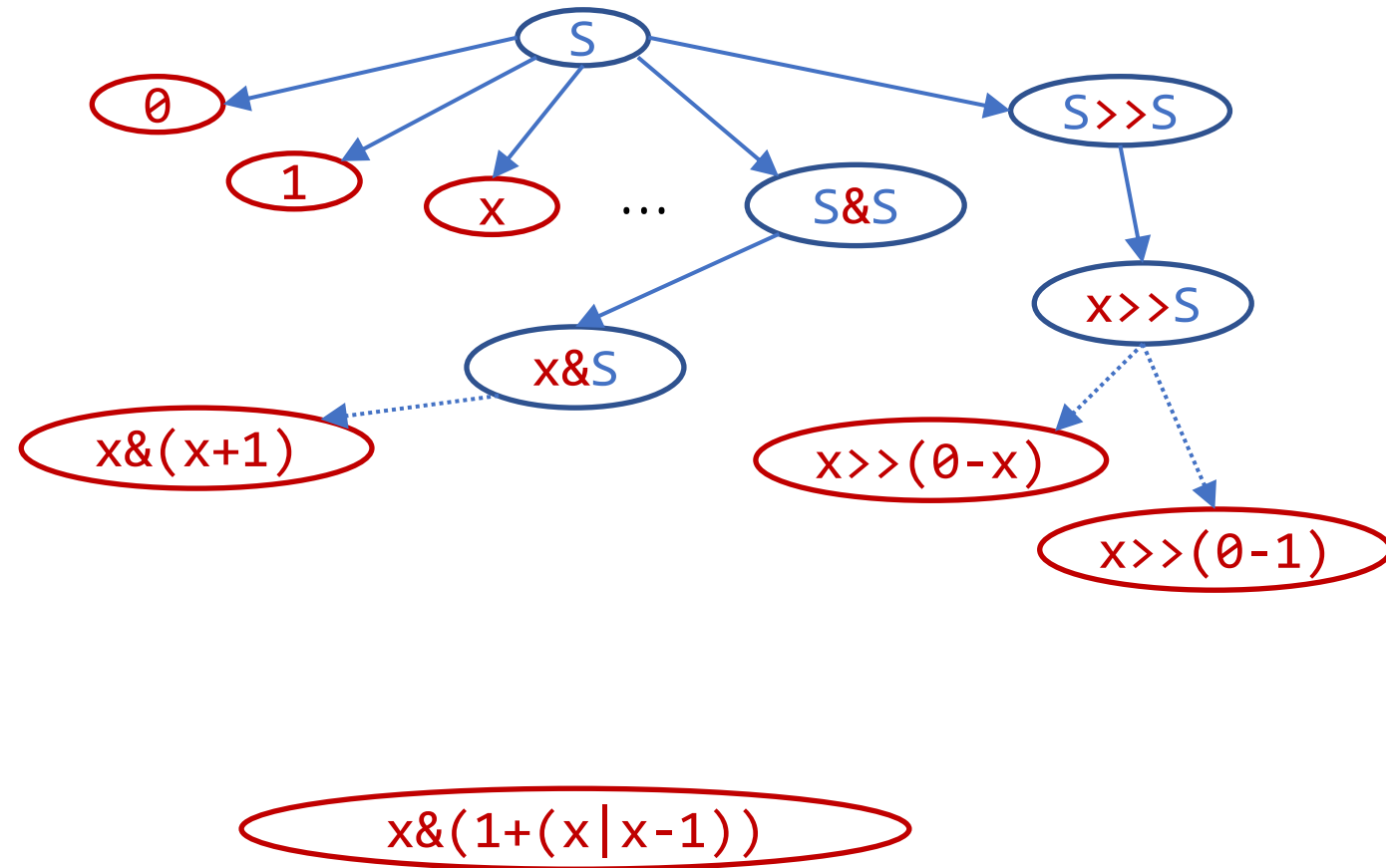
Turn off the rightmost sequence of 1s:

00101 $\rightarrow$ 00100

01010 $\rightarrow$ 01000

10110 $\rightarrow$ 10000

$S \rightarrow$	θ		1		x	
	S	+	S			
	S	-	S			
	S	&	S			
	S		S			
	S	<<	S			
	S	>>	S			



top-down search (revisited)

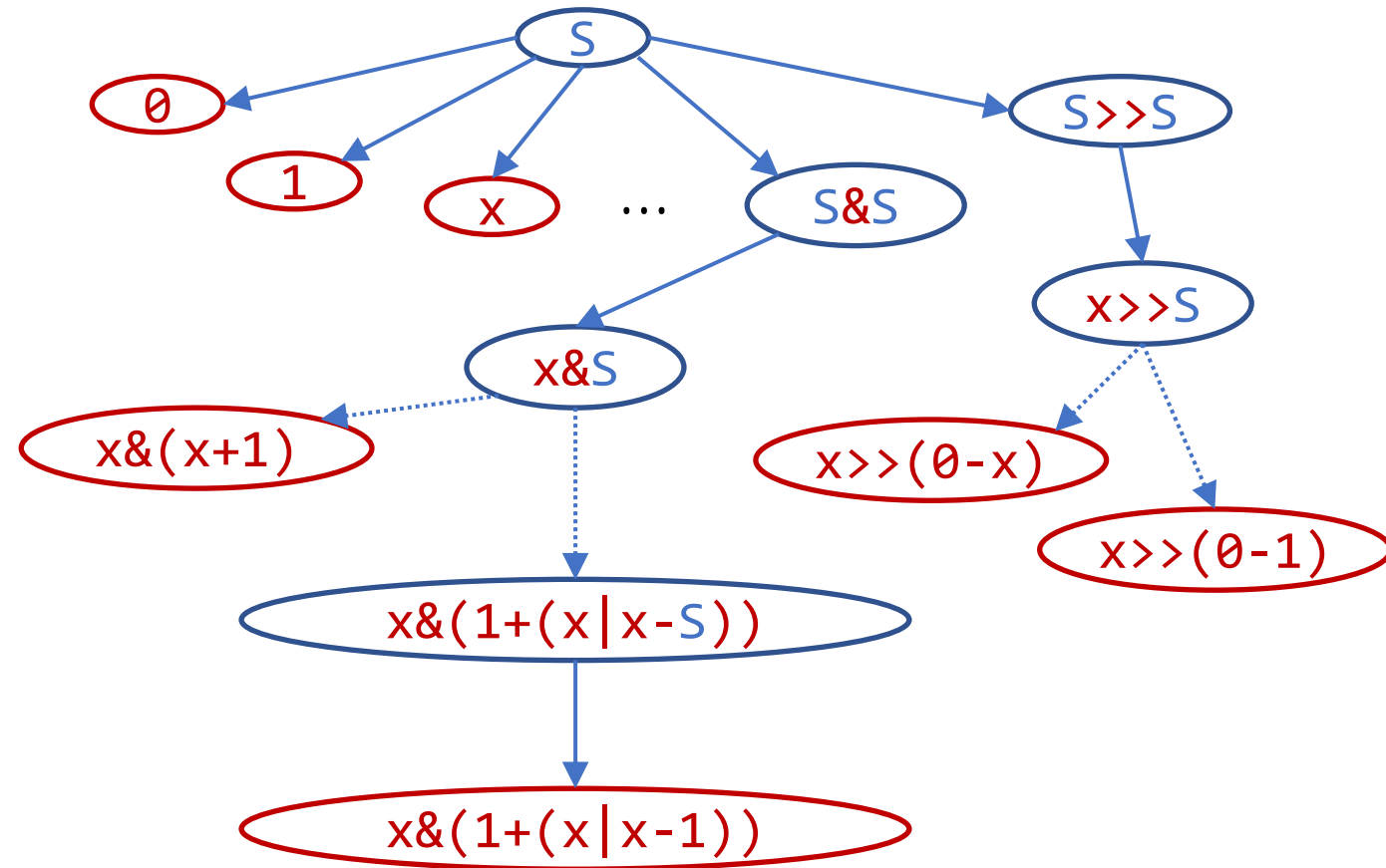
Turn off the rightmost sequence of 1s:

00101 $\rightarrow$ 00100

01010 $\rightarrow$ 01000

10110 $\rightarrow$ 10000

$S \rightarrow$	θ		1		x	
	S	+	S			
	S	-	S			
	S	&	S			
	S		S			
	S	<<	S			
	S	>>	S			



top-down search (revisited)

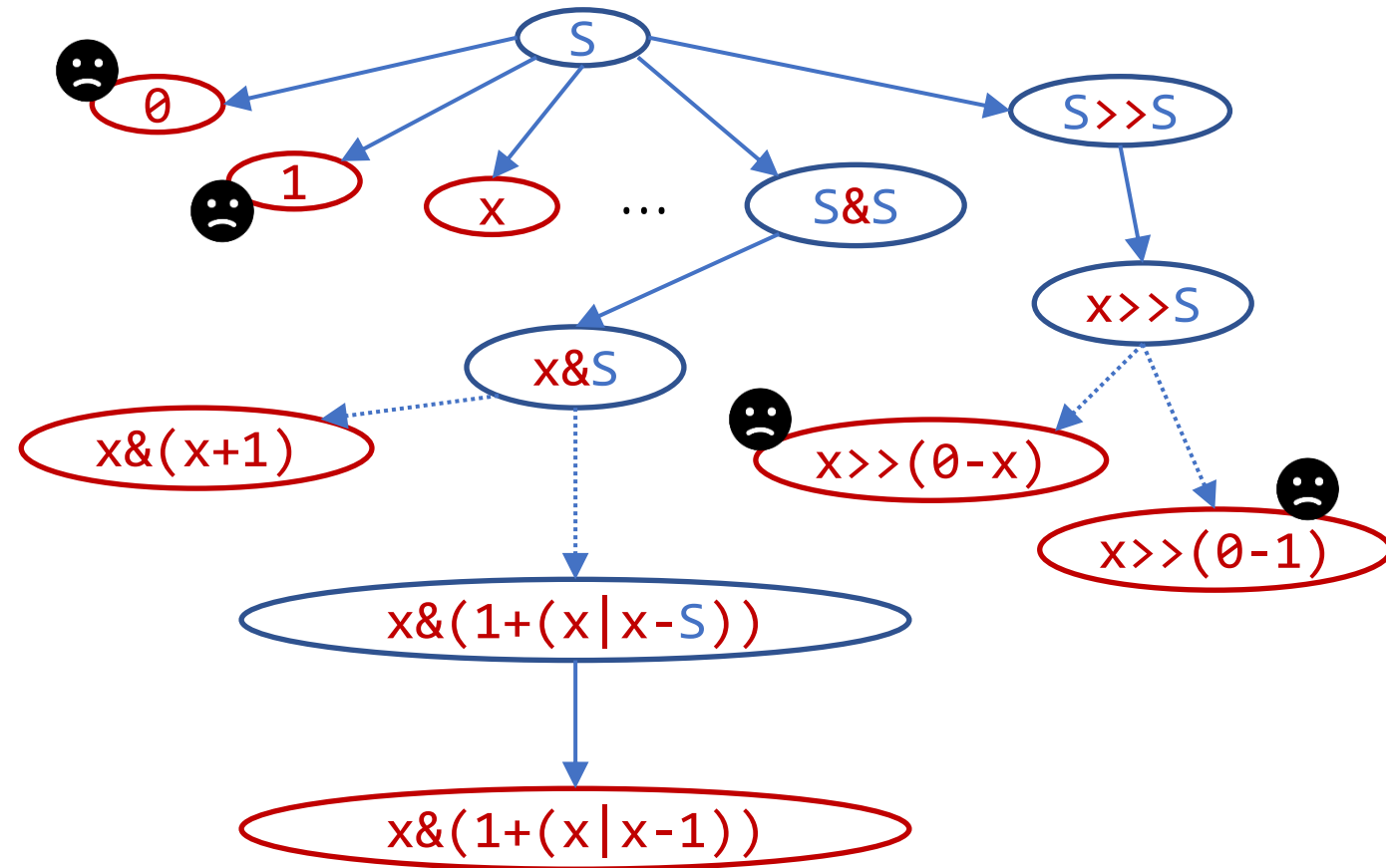
Turn off the rightmost sequence of 1s:

00101 $\rightarrow$ 00100

01010 $\rightarrow$ 01000

10110 $\rightarrow$ 10000

$S \rightarrow$	θ		1		x	
S	+		S			
S	-		S			
S	&		S			
S			S			
S	<<		S			
S	>>		S			



top-down search (revisited)

Turn off the rightmost sequence of 1s:

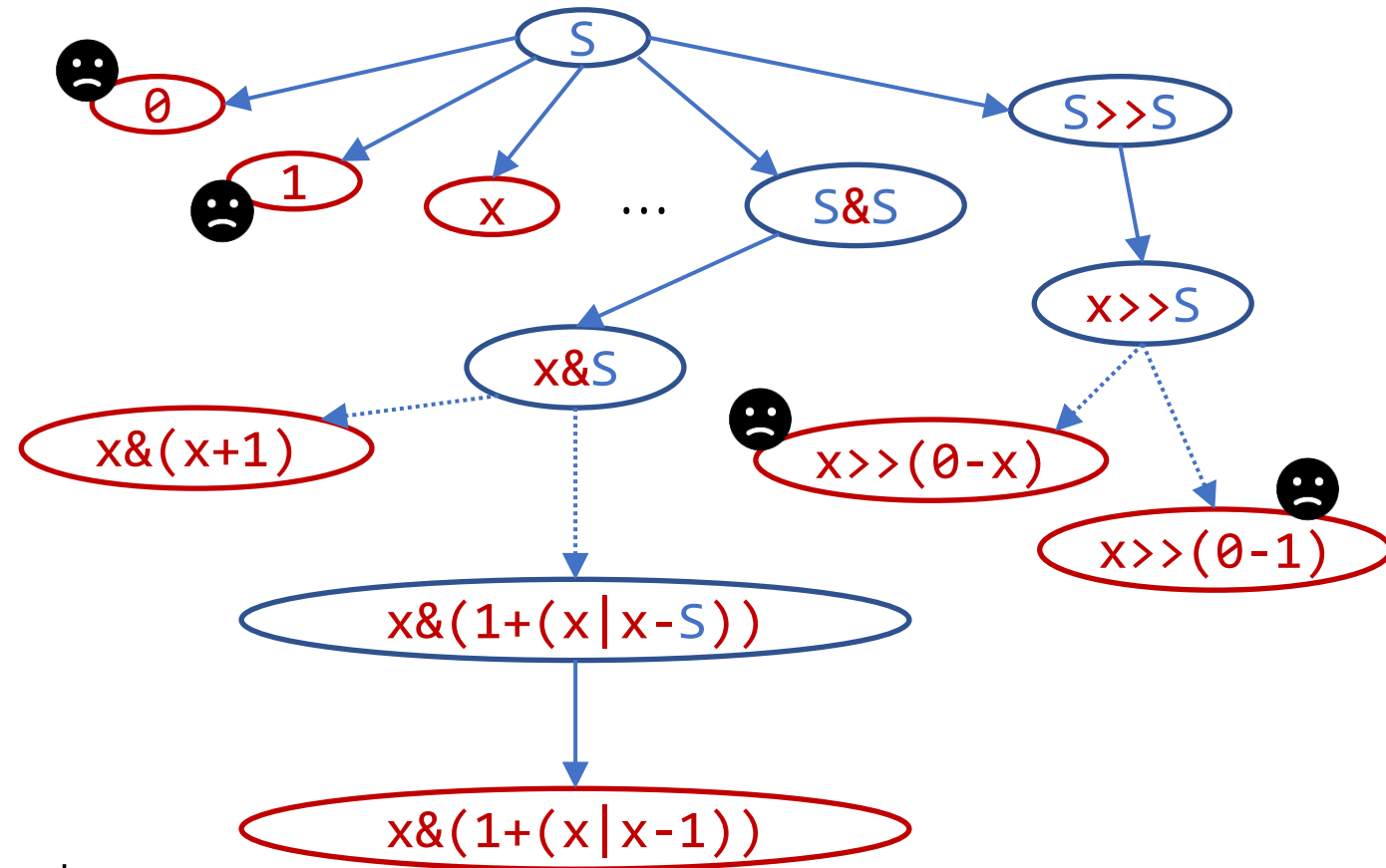
00101 $\rightarrow$ 00100

01010 $\rightarrow$ 01000

10110 $\rightarrow$ 10000

$S \rightarrow$	θ		1		x	
S	+		S			
S	-		S			
S	&		S			
S			S			
S	<<		S			
S	>>		S			

Explores many unlikely programs!



weighted top-down search

[[Euphony. Lee et al'15](#)]

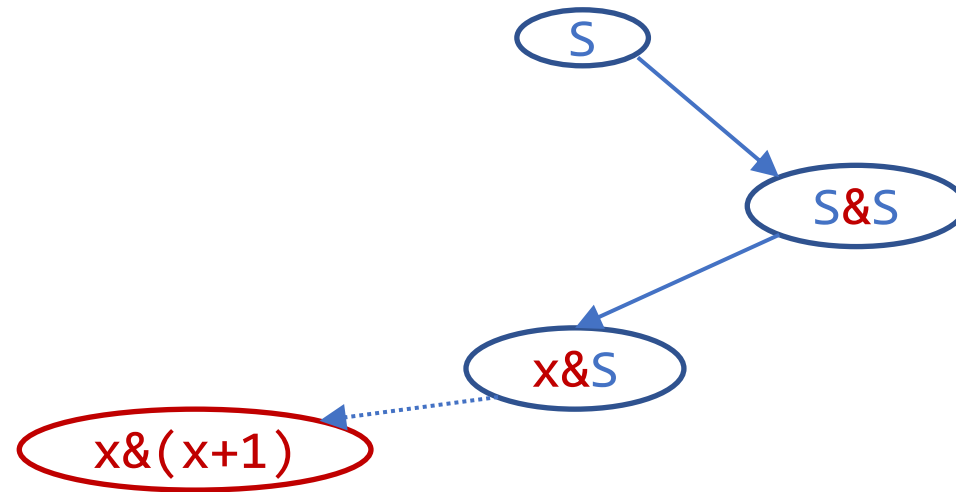
Idea: explore programs in the order of **likelihood**, not **size**



weighted top-down search

[[Euphony. Lee et al'15](#)]

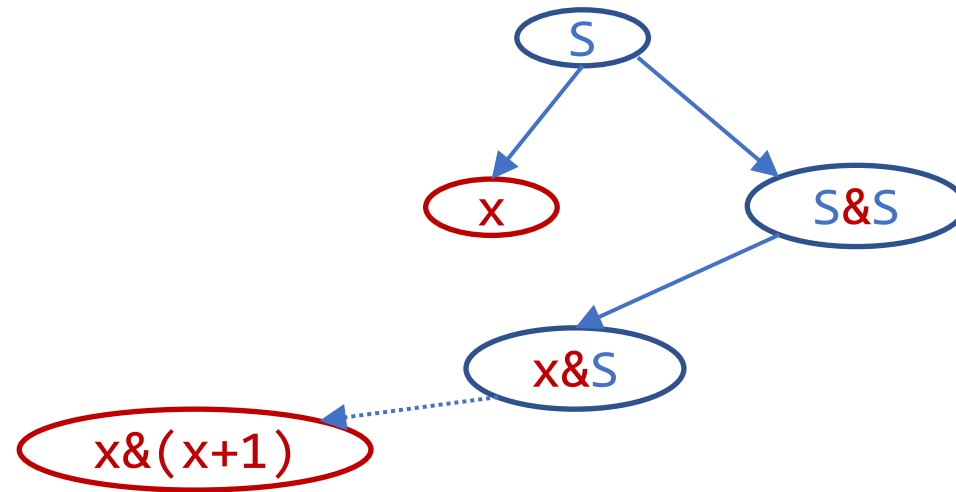
Idea: explore programs in the order of **likelihood**, not **size**



weighted top-down search

[[Euphony. Lee et al'15](#)]

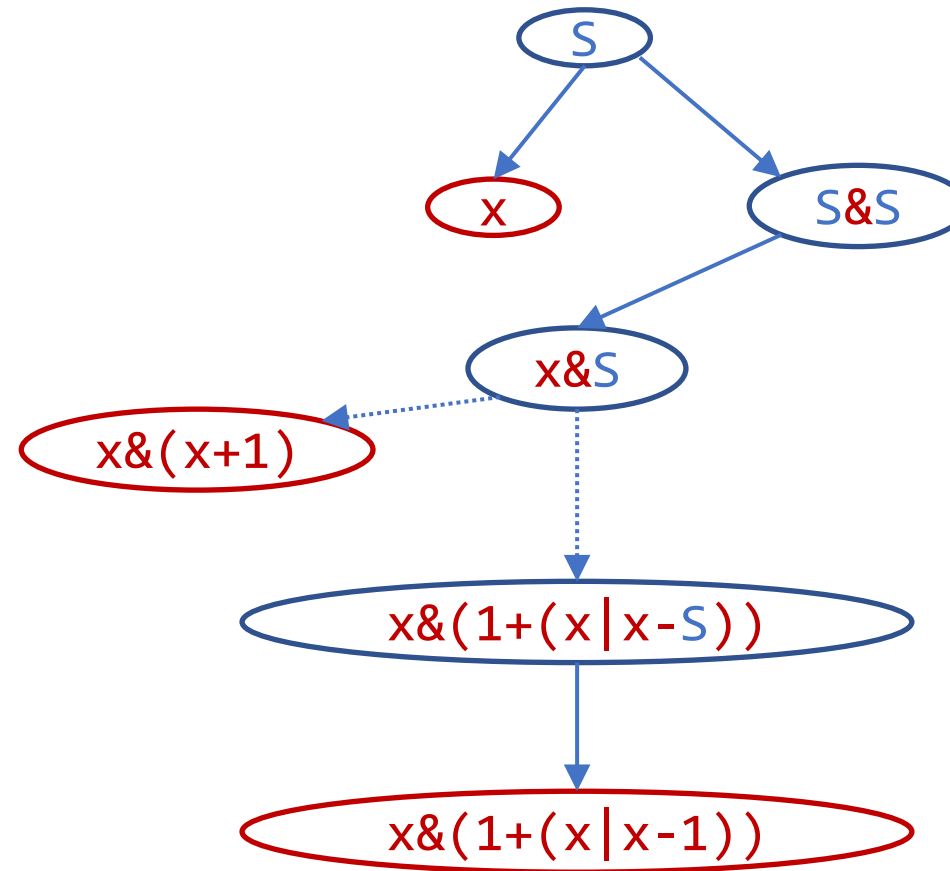
Idea: explore programs in the order of **likelihood**, not **size**



weighted top-down search

[[Euphony. Lee et al'15](#)]

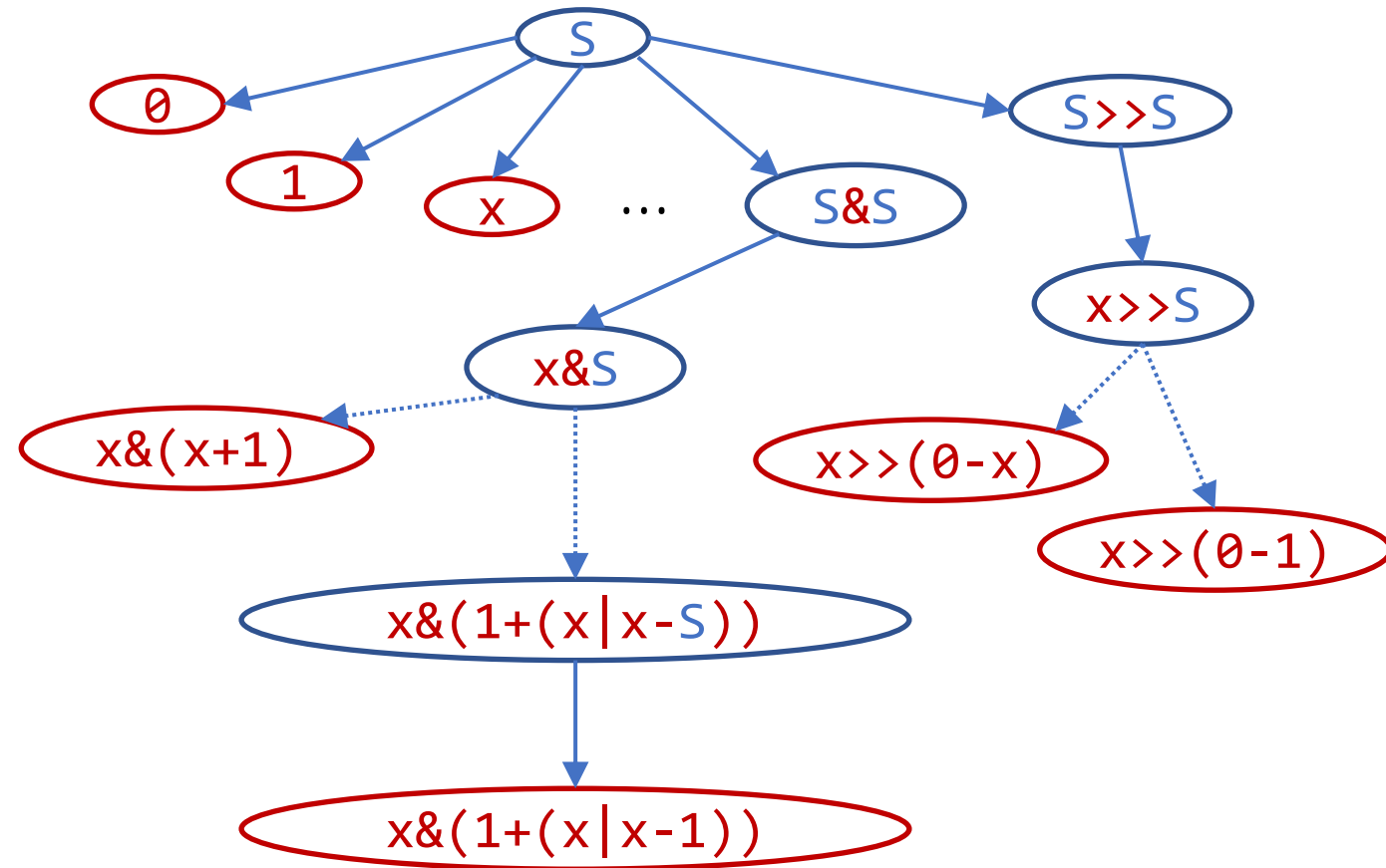
Idea: explore programs in the order of **likelihood**, not **size**



weighted top-down search

[[Euphony. Lee et al'15](#)]

Idea: explore programs in the order of **likelihood**, not **size**

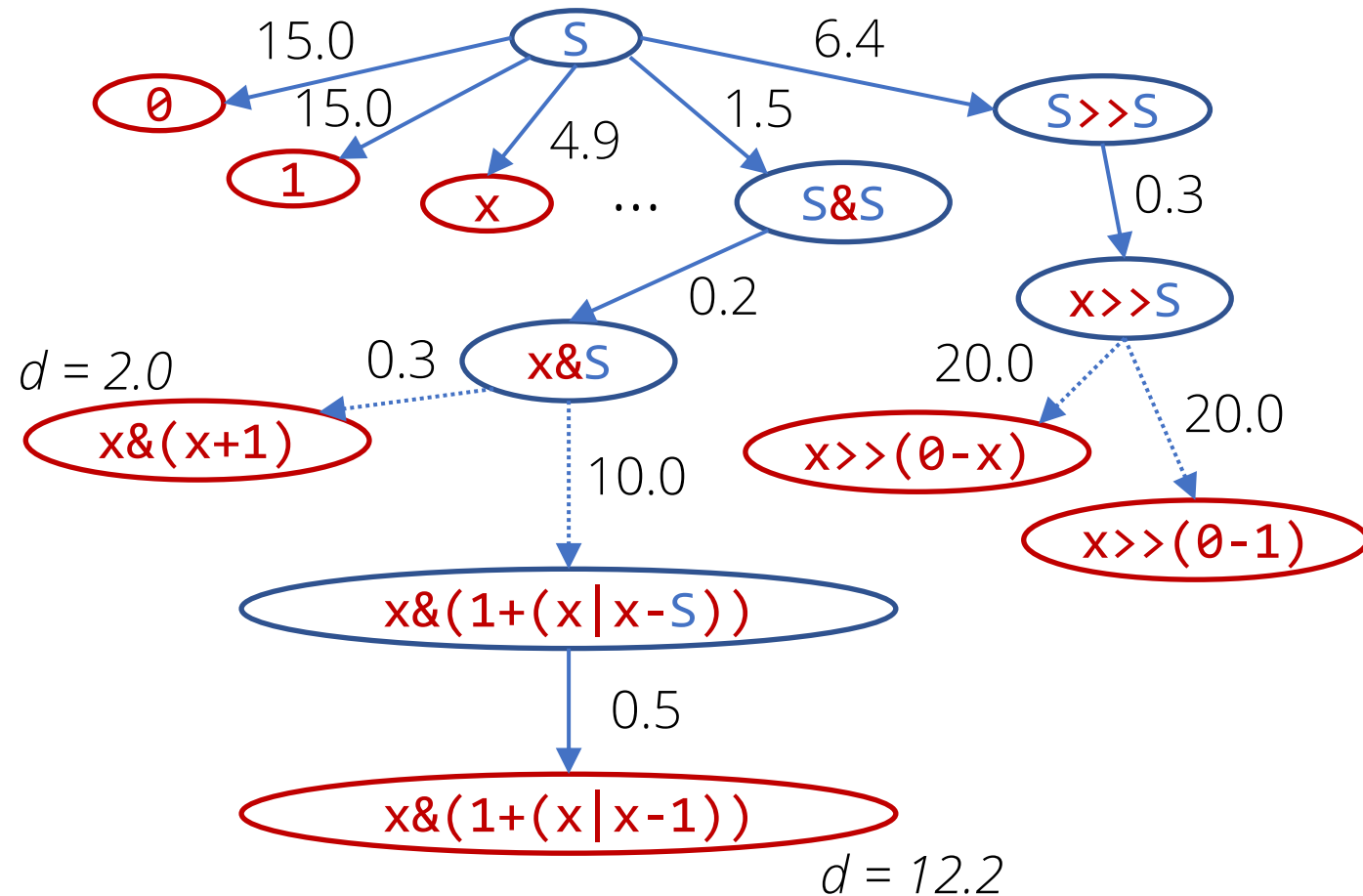


weighted top-down search

[[Euphony. Lee at al'15](#)]

Idea: explore programs in the order of **likelihood**, not **size**

1. Assign weights $w(e)$ to edges such that $d(p) < d(p')$ iff p is more likely than p'



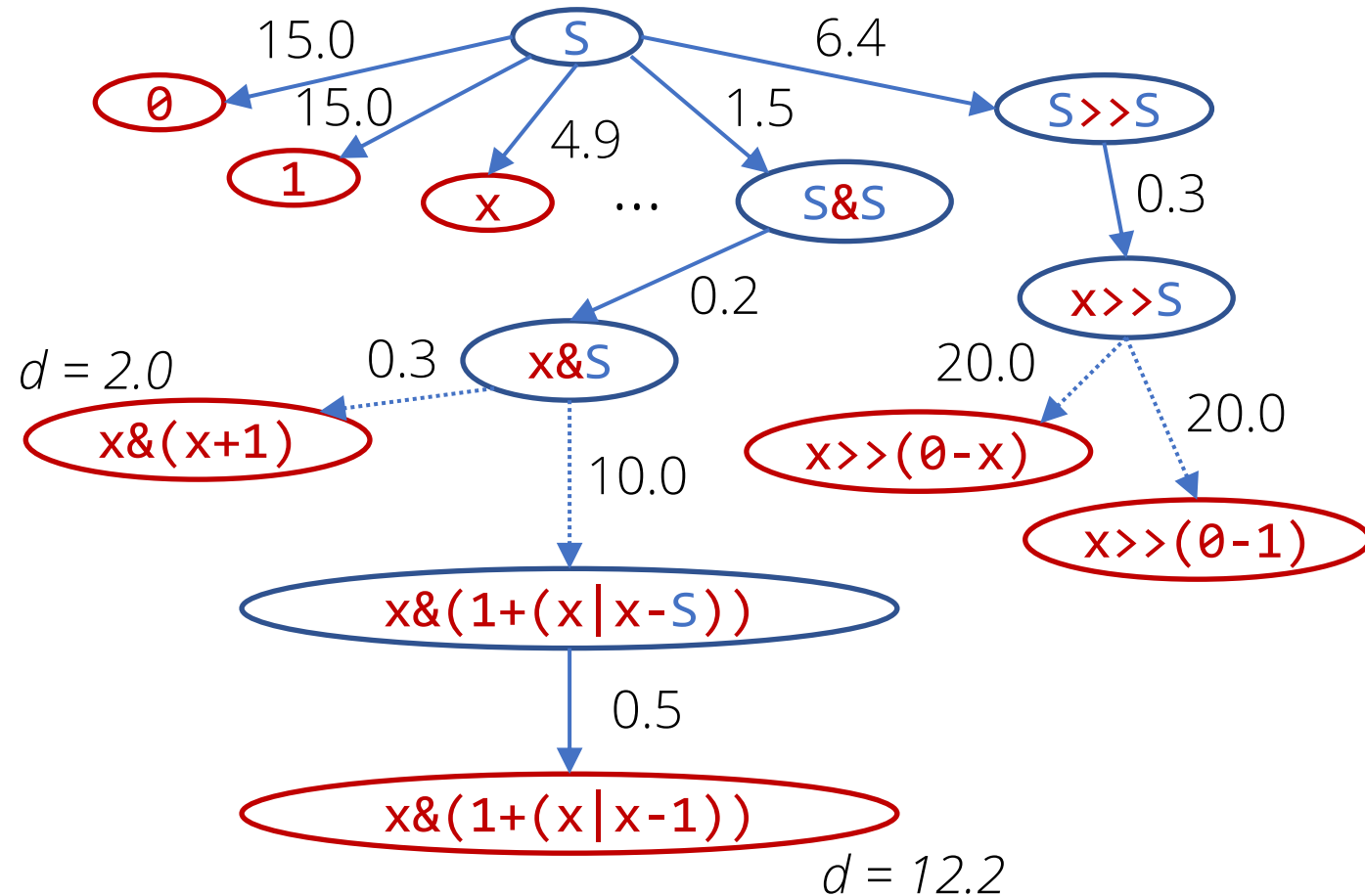
weighted top-down search

[[Euphony. Lee at al'15](#)]

Idea: explore programs in the order of **likelihood**, not **size**

1. Assign weights $w(e)$ to edges such that $d(p) < d(p')$ iff p is more likely than p'

2. Use A* algorithm to find closest leaves



weighted top-down search

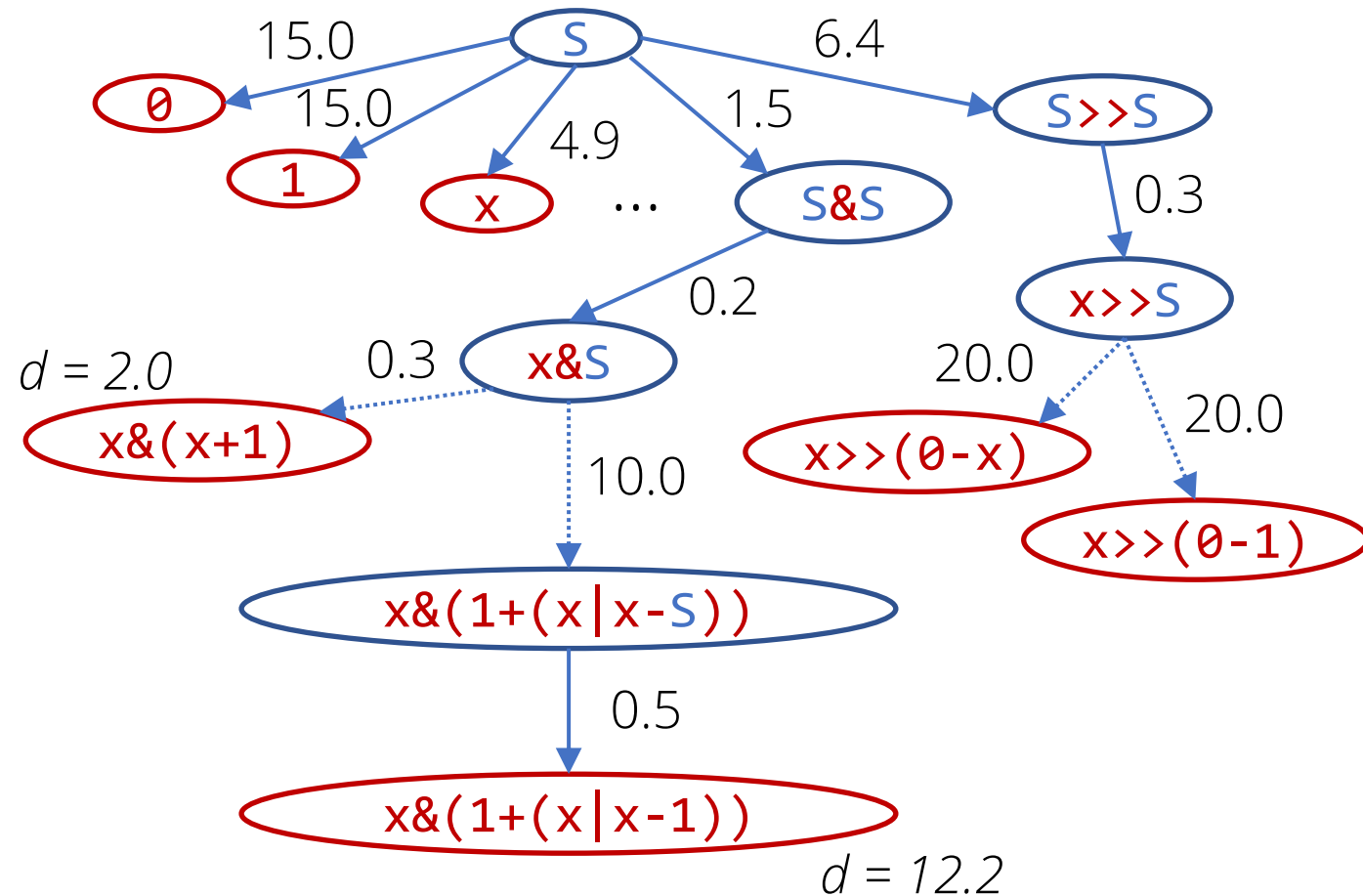
[[Euphony. Lee at al'15](#)]

Idea: explore programs in the order of **likelihood**, not **size**

1. Assign weights $w(e)$ to edges such that $d(p) < d(p')$ iff p is more likely than p'

2. Use A* algorithm to find closest leaves

3. Weights can be learned offline or online



bottom-up vs top-down

bottom-up

top-down

bottom-up vs top-down

bottom-up

top-down

- + dynamic programming
(result caching)
- + can evaluate candidates
(good for equivalence reduction)

bottom-up vs top-down

bottom-up

- + dynamic programming
(result caching)
- + can evaluate candidates
(good for equivalence reduction)

top-down

- + goal-directed
- + good for spec propagation
- + can be guided with context-
dependent weights

enumerative search: discussion

examples: [Synquid](#), [Morpheus](#), [Snippy](#)

enumerative search: discussion

examples: [Synquid](#), [Morpheus](#), [Snippy](#)

- + works for any language you can evaluate / check

- might not fully leverage spec / DSL

this talk

search strategies

1. enumerative
2. VSA & co
3. stochastic
4. constraint-based

representation-based search

idea: build a graph that compactly represents all programs that satisfy the spec

examples: [FlashhFill](#), [FlashExtract](#), [WebRelate](#)

- + efficient
- + returns a ranked list of solutions

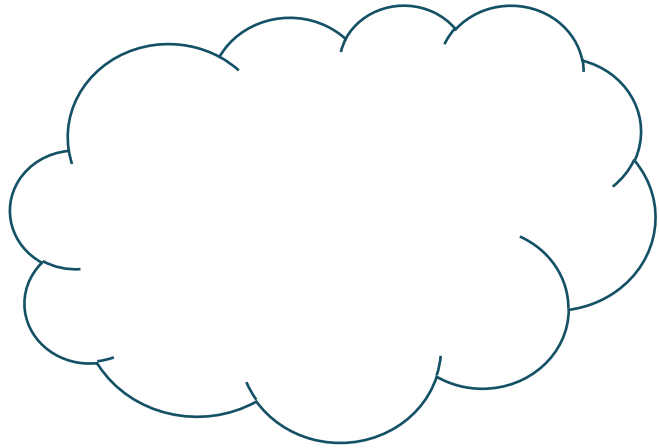
- restricted class of DSLs

this talk

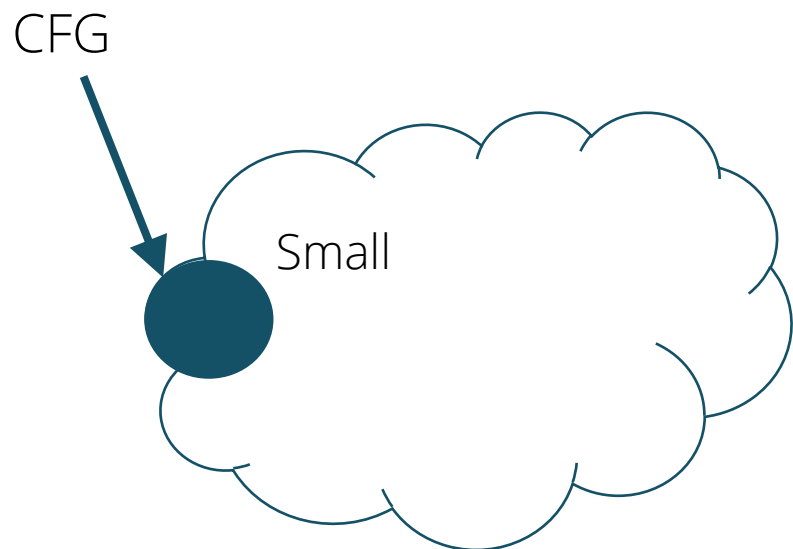
search strategies

1. enumerative
2. VSA & co
3. stochastic
4. constraint-based

search space

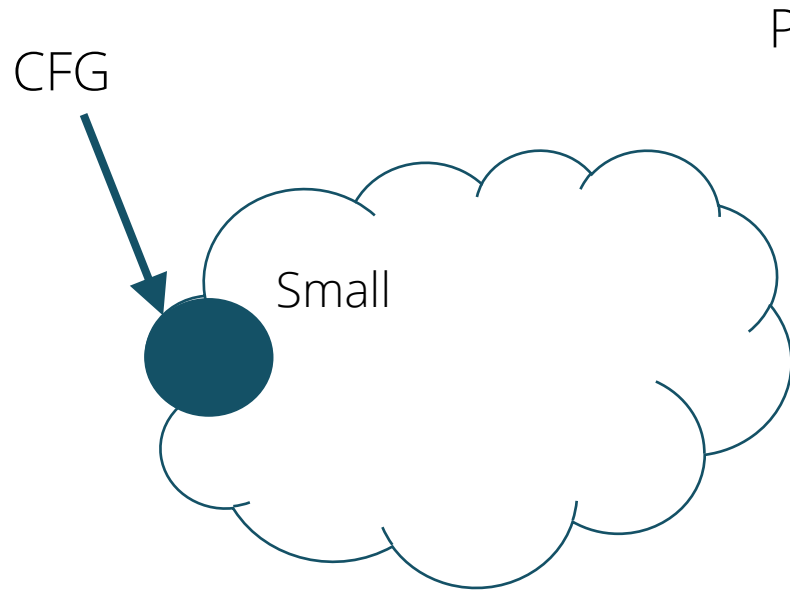


search space

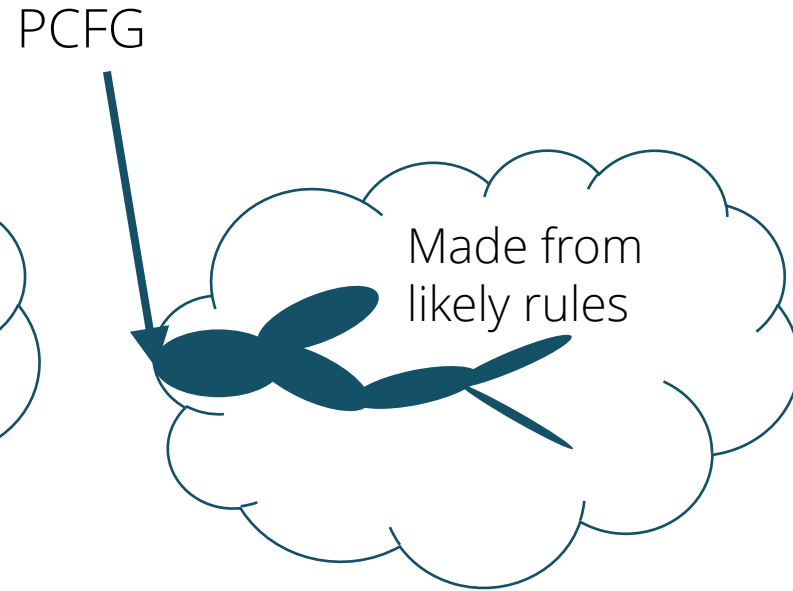


Enumerative search

search space

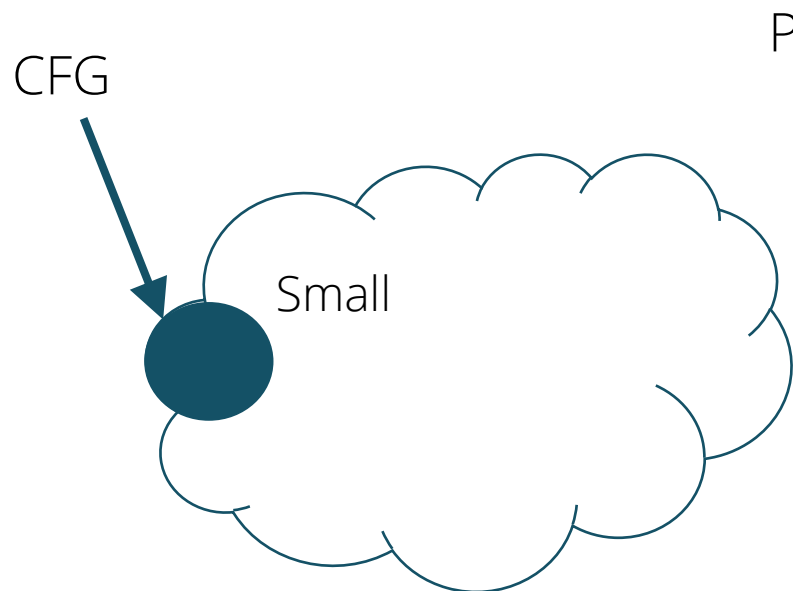


Enumerative search

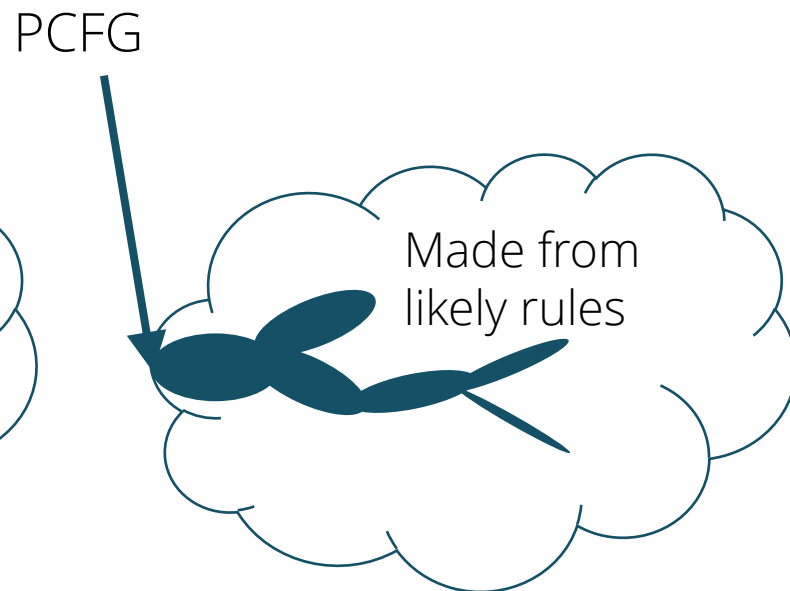


Weighted
enumerative search

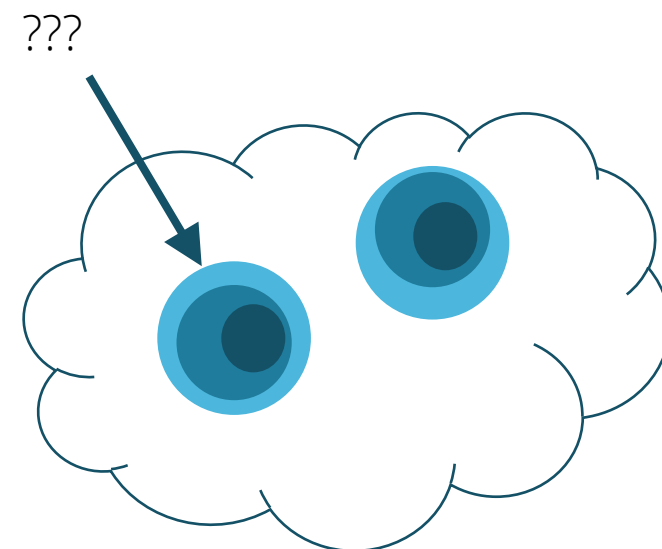
search space



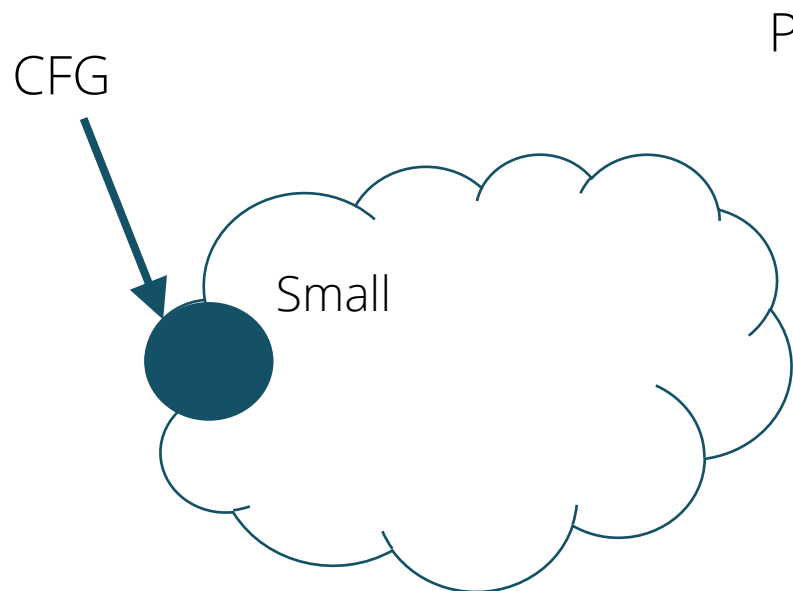
Enumerative search



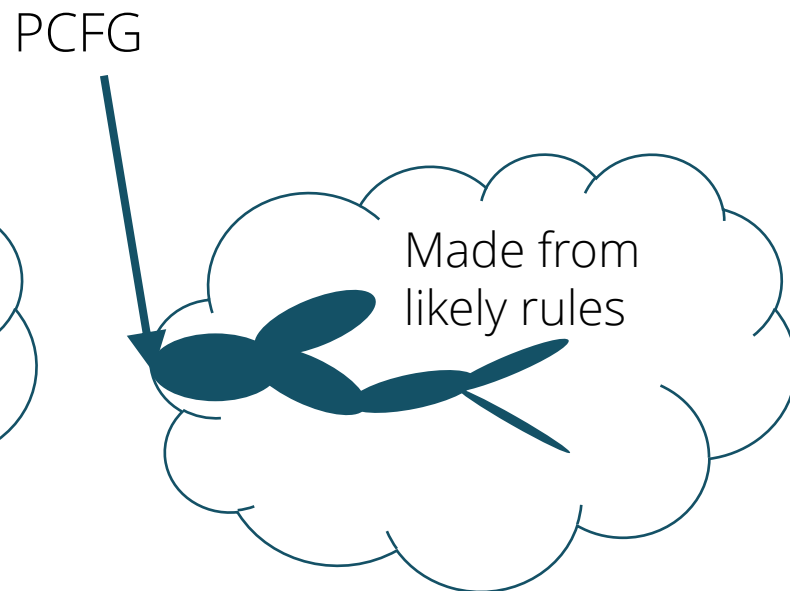
Weighted
enumerative search



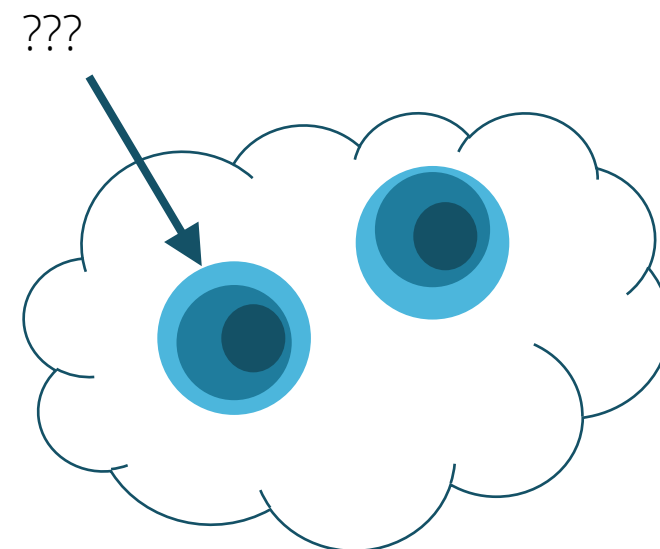
search space



Enumerative search

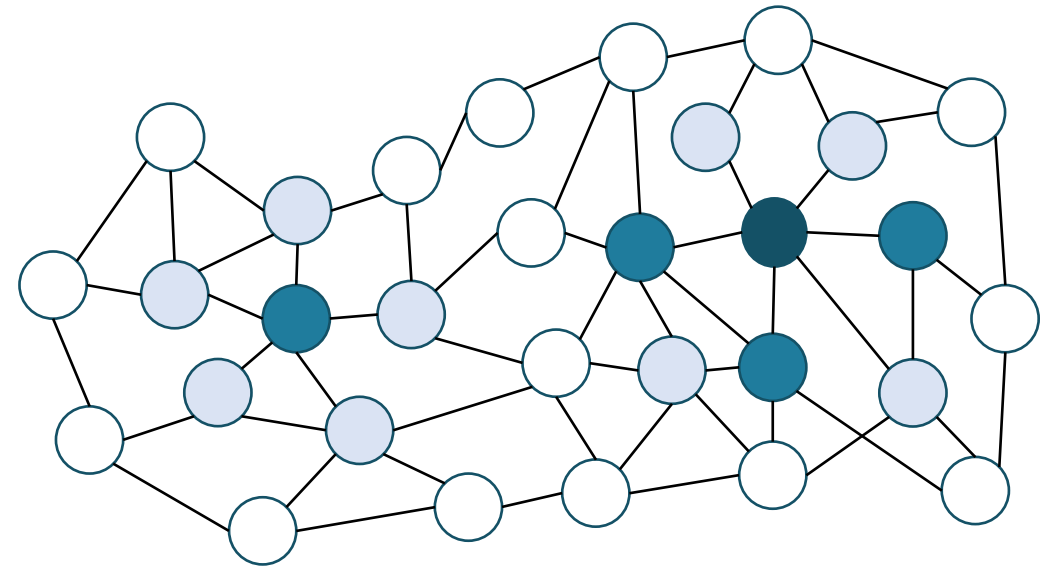


Weighted
enumerative search



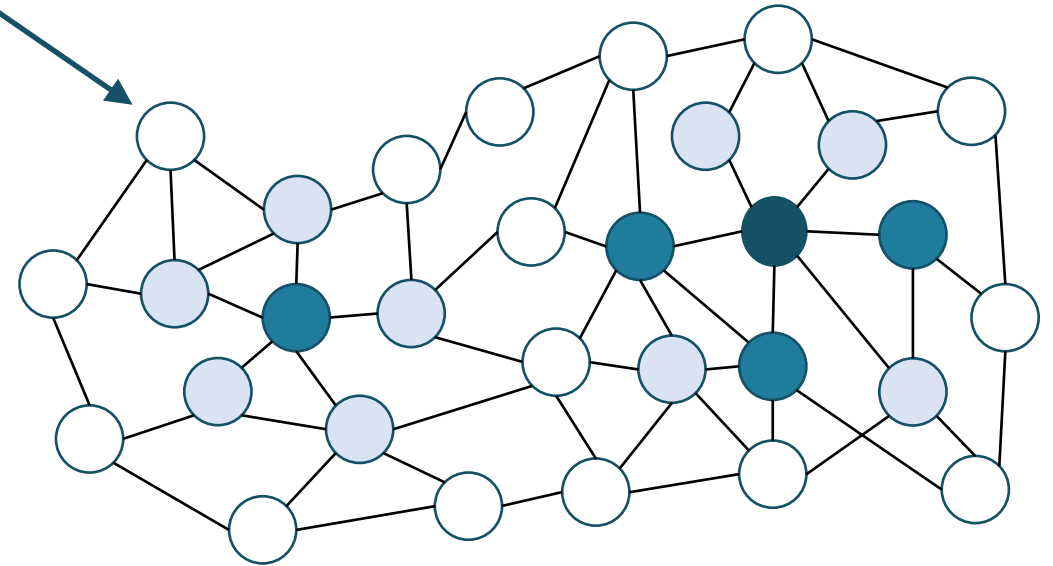
Local search

naïve local search

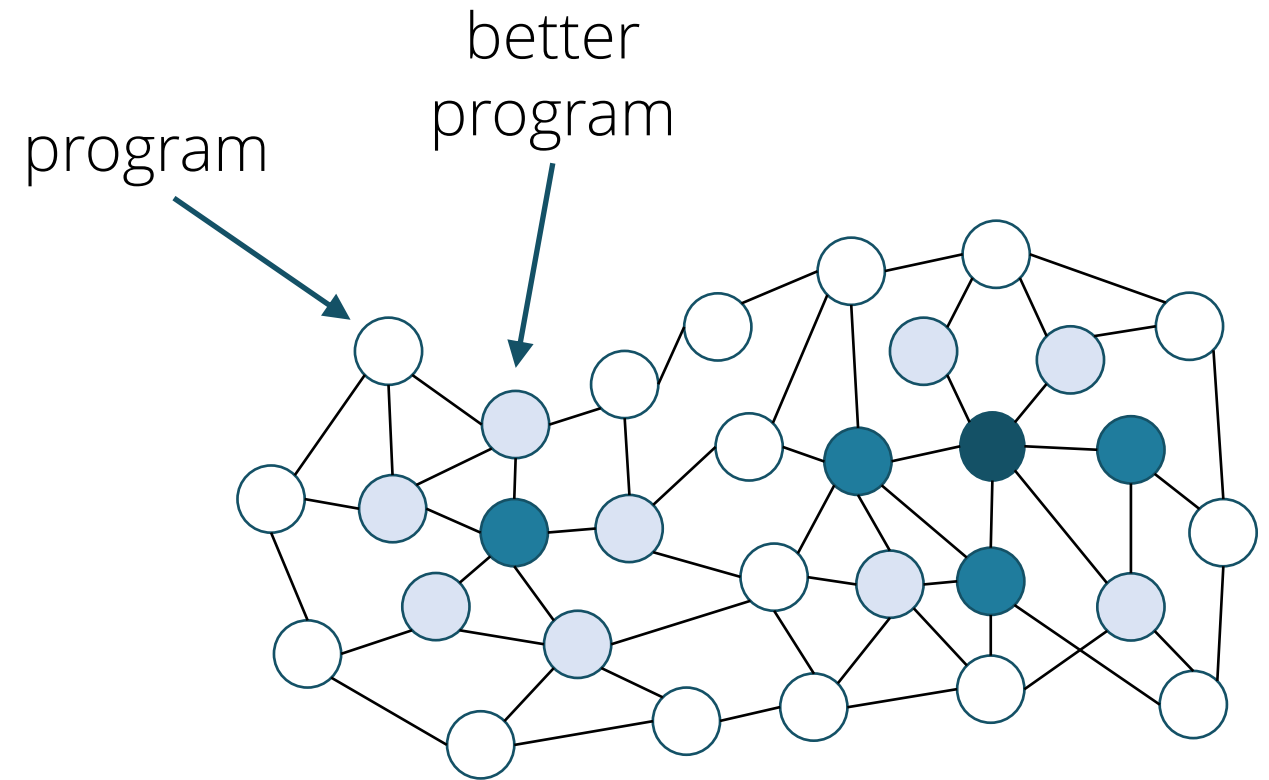


naïve local search

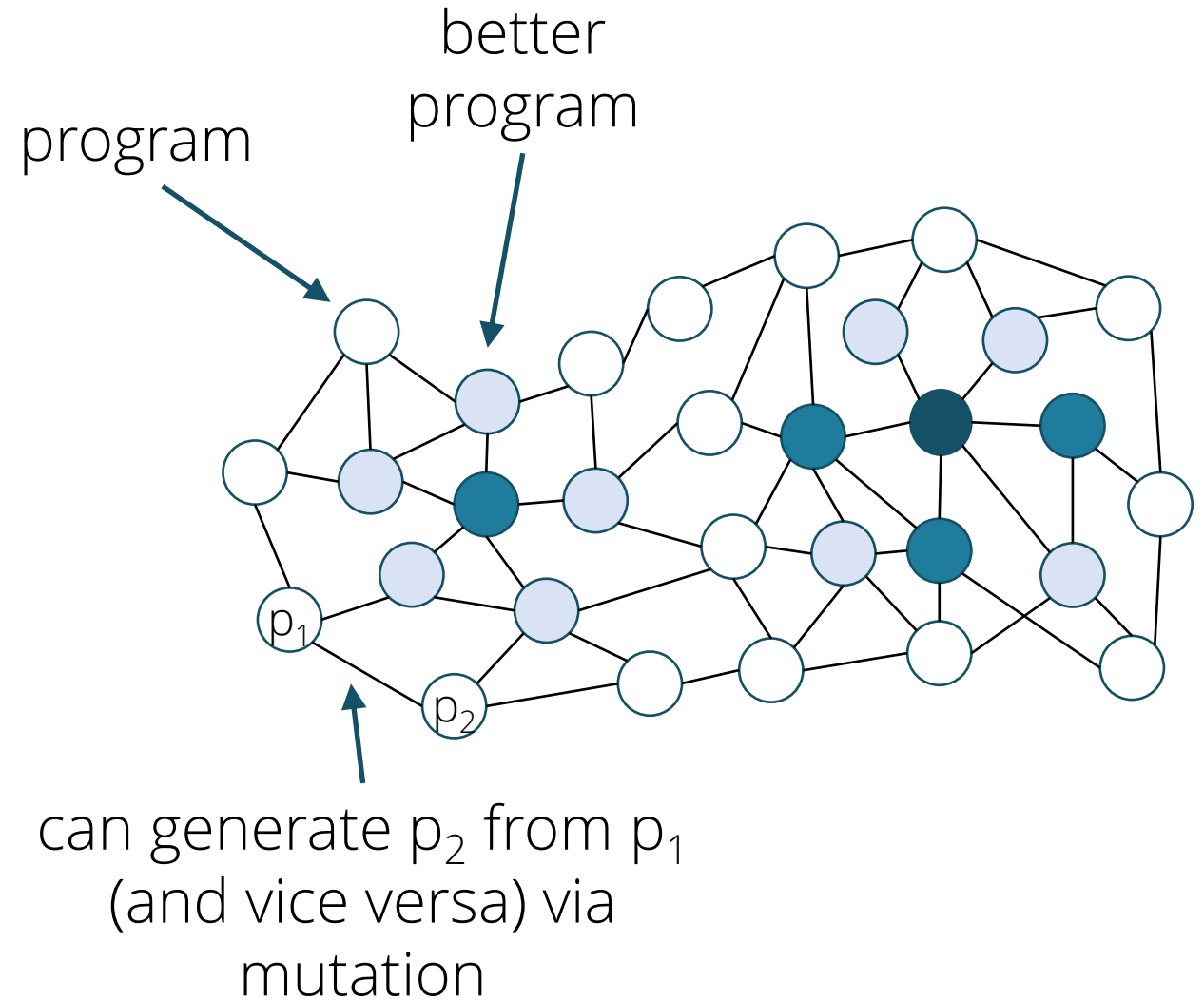
program



naïve local search



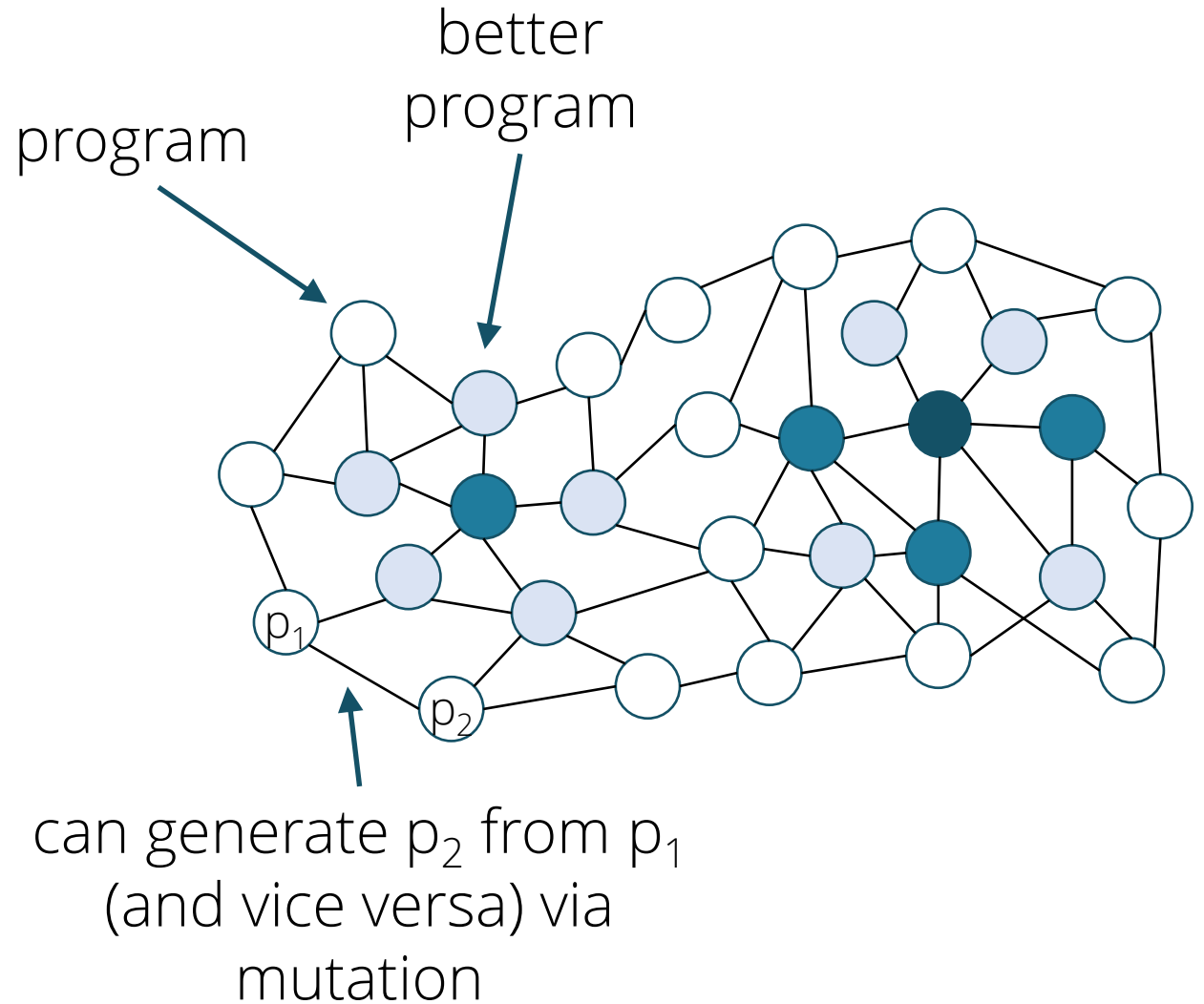
naïve local search



naïve local search

To find the best program:

```
p := random()
while (true) {
  p' := mutate(p);
  if (cost(p') < cost(p))
    p := p';
}
```

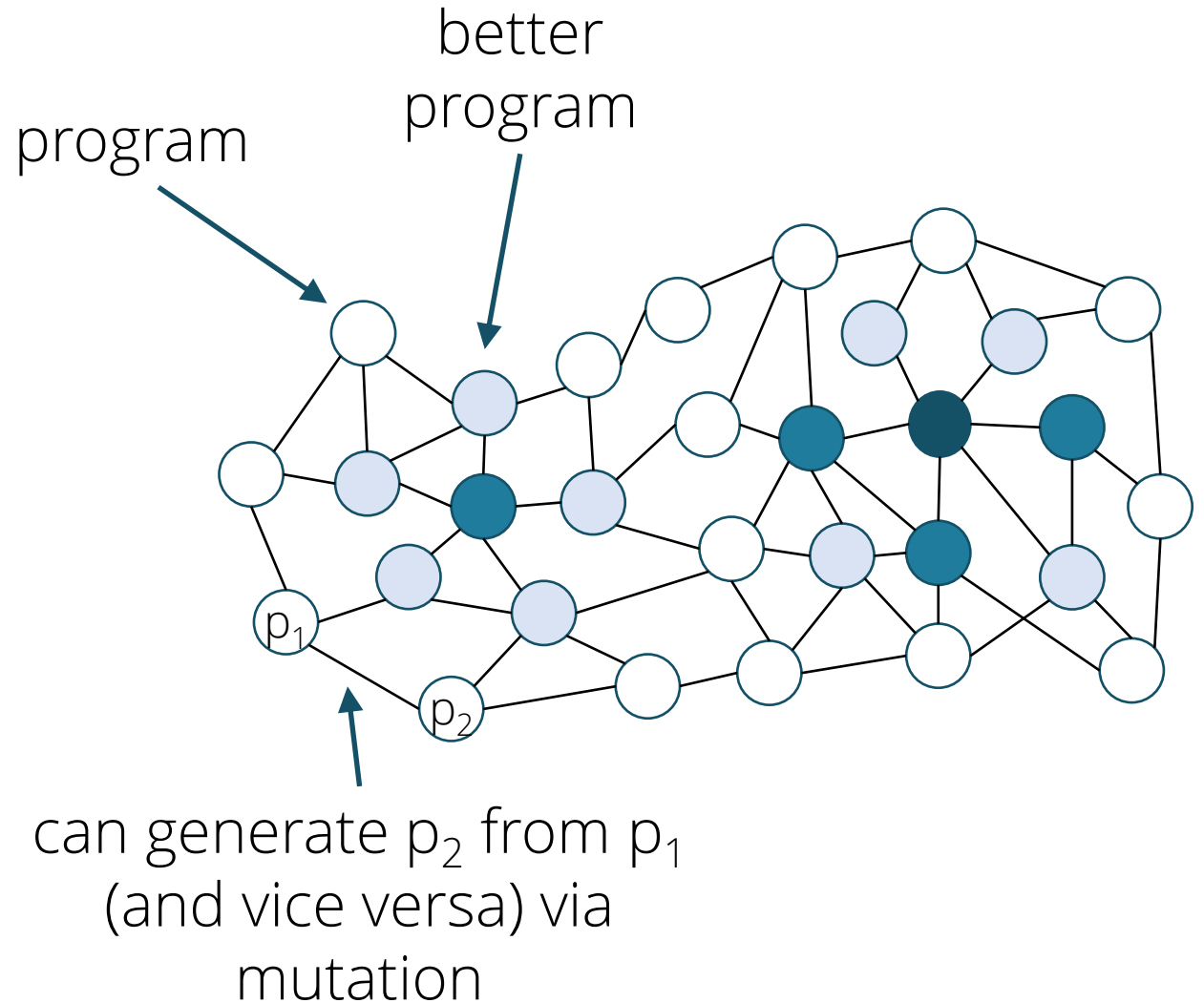


naïve local search

To find the best program:

```
p := random()
while (true) {
  p' := mutate(p);
  if (cost(p') < cost(p))
    p := p';
}
```

Will never get to ● from p_1 !



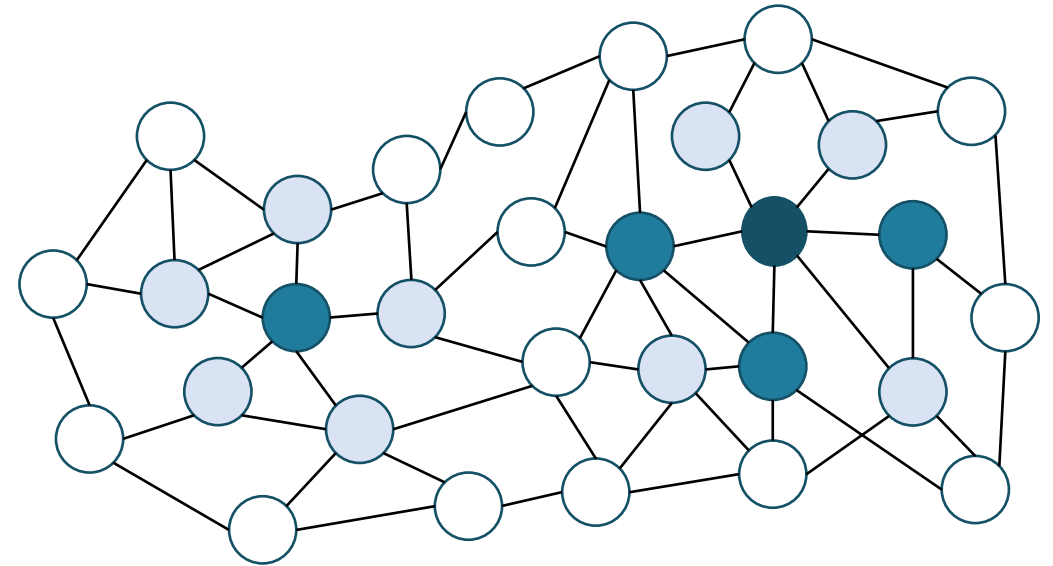
MCMC sampling

Avoid getting stuck in local minima:

```
p := random()
while (true) {
  p' := mutate(p);
  if (random( $A(p \rightarrow p')$ ))
    p := p';
}
```

where

if p' is better than p : $A(p \rightarrow p') = 1$
otherwise: $A(p \rightarrow p')$ decreases with
difference in cost between p' and p

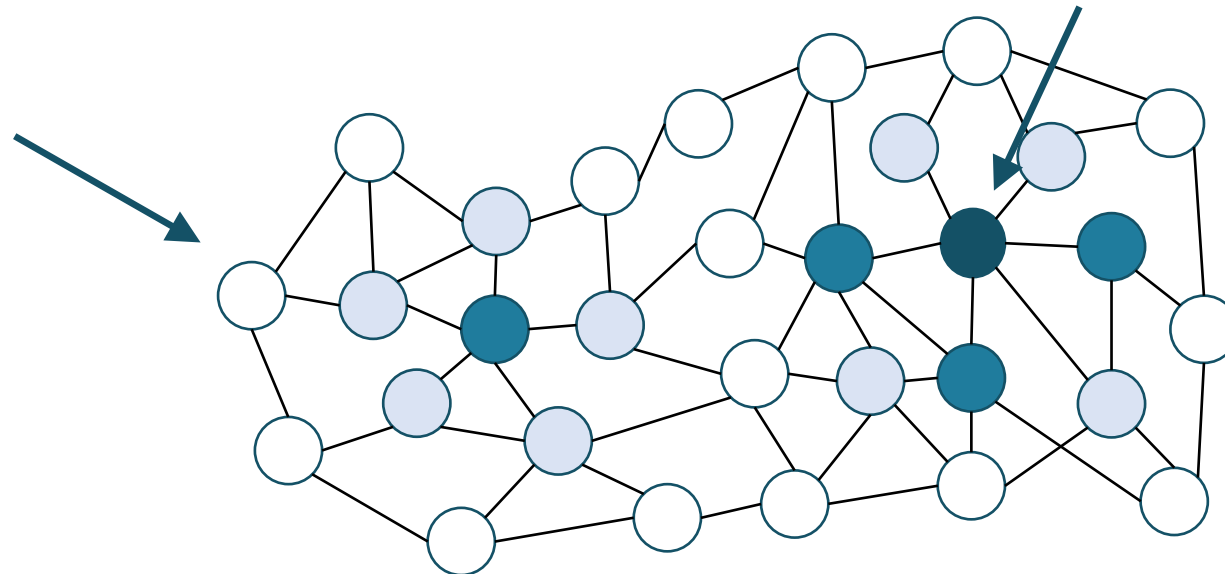


MCMC for superoptimization

[[STOKE. Schkufza et al '13](#)]

- .L0:
- movq rsi, r9
- movl ecx, ecx
- shrq 32, rsi
- andl 0xffffffff, r9d
- movq rcx, rax
- movl edx, edx
- imulq r9, rax
- imulq rdx, r9
- imulq rsi, rdx
- imulq rsi, rcx
- addq rdx, rax
- jae .L2
- movabsq 0x100000000, rdx
- addq rdx, rcx
- .L2:
- movq rax, rsi
- movq rax, rdx
- shrq 32, rsi
- salq 32, rdx
- addq rsi, rcx
- addq r9, rdx
- adcq 0, rcx
- addq r8, rdx
- adcq 0, rcx
- addq rdi, rdx
- adcq 0, rcx
- movq rcx, r8
- movq rdx, rdi

- .L0:
- shlq 32, rcx
- movl edx, edx
- xorq rdx, rcx
- movq rcx, rax
- mulq rsi
- addq r8, rdi
- adcq 0, rdx
- addq rdi, rax
- adcq 0, rdx
- movq rdx, r8
- movq rax, rdi



stochastic search: discussion

examples: [STOKE](#), [FrAngel](#)

- + can explore program spaces with no a-priori bias
- + good for sampling from a distribution
- needs cost function that faithfully approximates correctness

this talk

search strategies

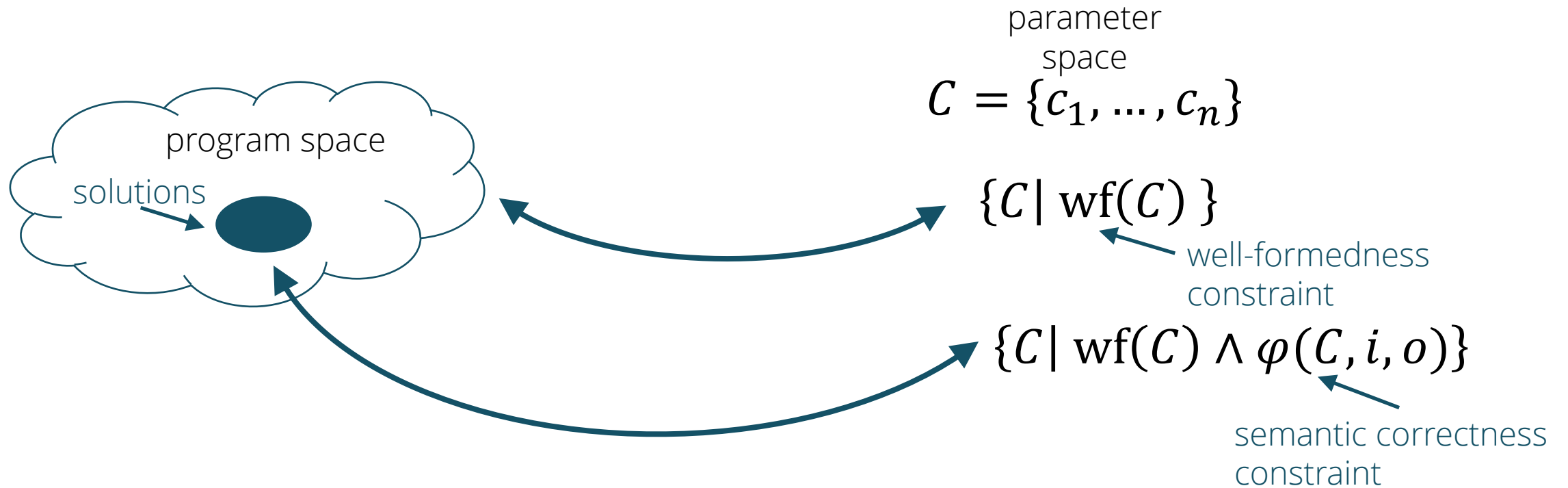
1. enumerative
2. VSA & co
3. stochastic
4. constraint-based

constraint-based search

idea: reduce synthesis to a SAT/SMT problem
and let a solver deal with it

challenge: how to encode a synthesis problem into SAT/SMT?

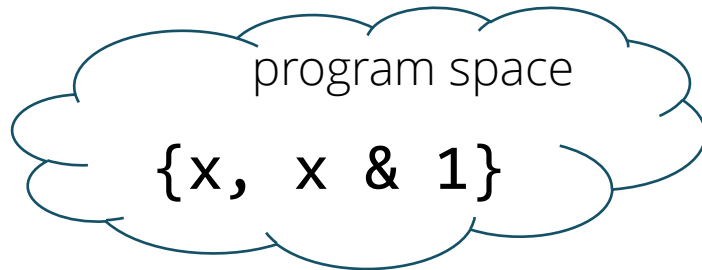
what is an encoding?



how to define an encoding

- define the parameter space $\mathcal{C} = \{c_1, \dots, c_n\}$
 - **encode** : $\text{Prog} \rightarrow \mathcal{C}$
 - **decode** : $\mathcal{C} \rightarrow \text{Prog}$ (might not be defined for all $\mathcal{C}$)
- define a formula $\text{wf}(c_1, \dots, c_n)$
 - that holds iff **decode**[c] is a “well-formed” program
- define a formula $\varphi(c_1, \dots, c_n, i, o)$
 - that holds iff (**decode**[c])(i) = o

SAT encoding: example



$$\text{wf}(c) \equiv \text{T}$$

$$\varphi(c, i_h, i_l, o_h, o_l) \equiv (\neg c \Rightarrow o_h = i_h \wedge o_l = i_l) \\ \wedge (c \Rightarrow o_h = 0 \wedge o_l = i_l)$$

$$\text{SAT}(\varphi(c, 1, 1, 0, 1))$$

$$\text{SAT}((\neg c \Rightarrow 0 = 1 \wedge 1 = 1) \wedge (c \Rightarrow 0 = 0 \wedge 1 = 1)) \xrightarrow{\text{SAT solver}} \text{Model } \{c \rightarrow 1\}$$

x is a two-bit word
($x = x_h x_l$)

$$E = [11 \rightarrow 01]$$

parameter space

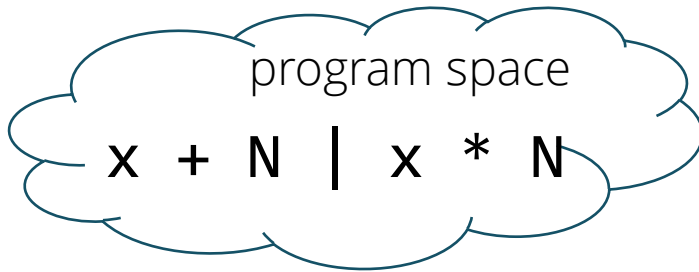
$$C = \{c: \text{Bool}\}$$

$$\text{decode}[0] \rightarrow x$$

$$\text{decode}[1] \rightarrow x \& 1$$

return decode[1] i.e. $x \& 1$

SMT encoding: example



$$\text{wf}(c_{op}, c_N) \equiv \top$$

$$\varphi(c_{op}, c_N, i, o) \equiv (\neg c_{op} \Rightarrow o = i + c_N) \wedge (c_{op} \Rightarrow o = i * c_N)$$

$$\text{SAT}(\varphi(c_{op}, c_N, 2, 9))$$

$$\text{SAT}((\neg c_{op} \Rightarrow 9 = 2 + c_N) \wedge (c_{op} \Rightarrow 9 = 2 * c_N))$$

return decode[0,7] i.e. x + 7

N is an integer literal
x is an integer input

$$E = [2 \rightarrow 9]$$

parameter space

$$\mathcal{C} = \{c_{op}: \text{Bool}, c_N: \text{Int}\}$$

$$\text{decode}[0, N] \rightarrow x + N$$

$$\text{decode}[1, N] \rightarrow x * N$$

SMT solver



Model $\{c_{op} \rightarrow 0, c_N \rightarrow 7\}$

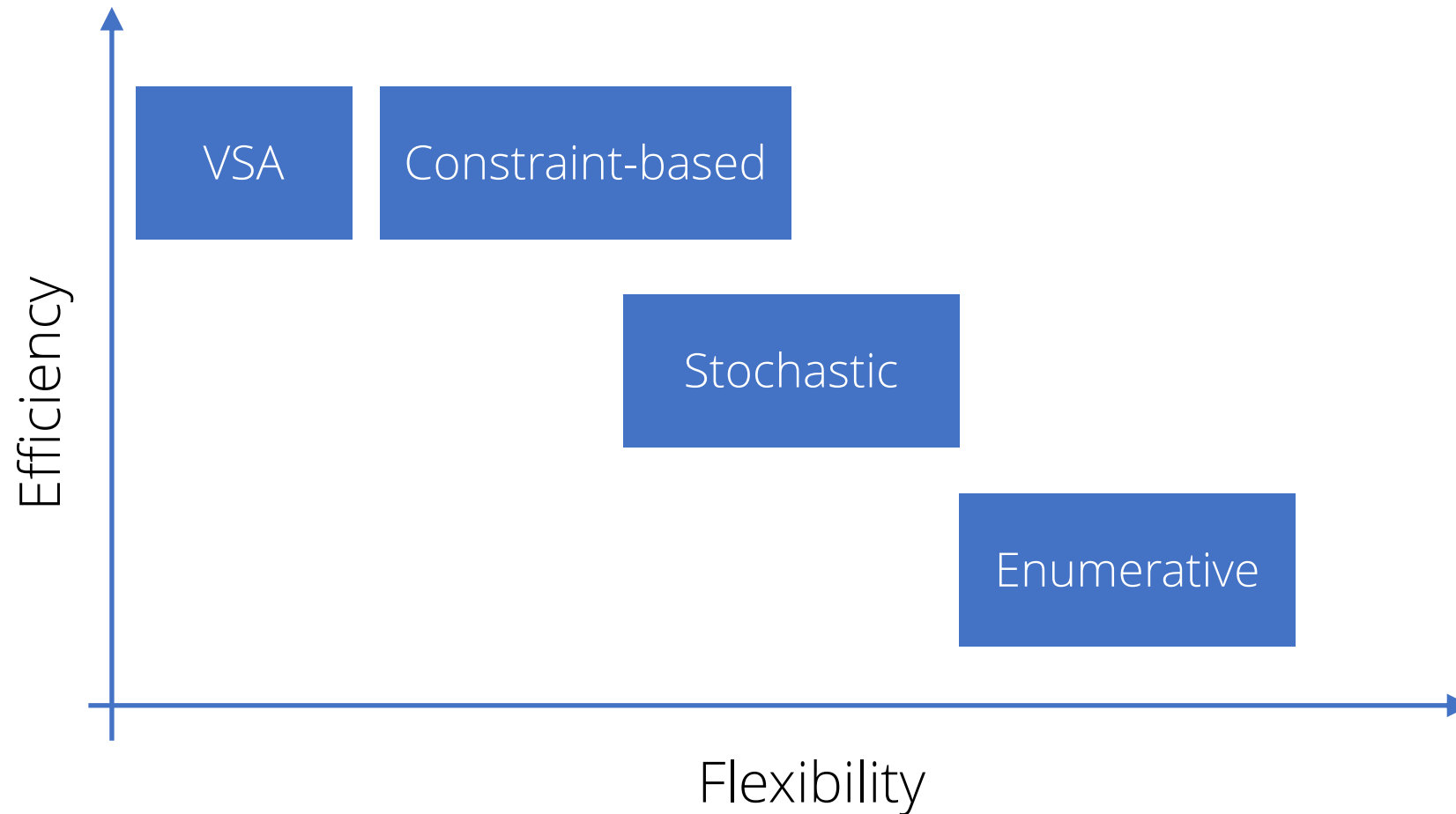
constraint-based search: discussion

examples: [Sketch](#), [Hoople+](#)

- + efficient
- + SMT solvers can discover constants

- finite number of parameters
(no recursive grammars)
- program semantics expressible
as a SAT/SMT formula

Comparison of search strategies

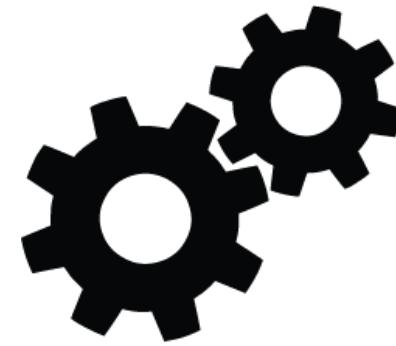


this talk

specifications



search strategies



program synthesis

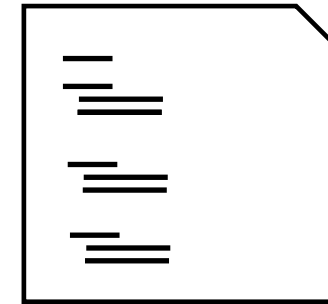
specification



search



program



program
space