Lecture 4
Biasing Enumerative Search

Nadia Polikarpova

Plan for this week

Today:

- Search space prioritization/biasing

Thursday:

- Discuss the Euphony paper
- Synthesis frameworks + suggested projects

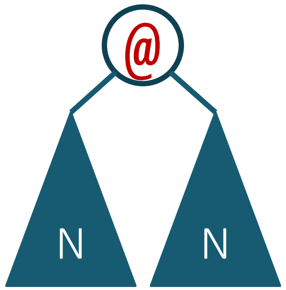
Project:

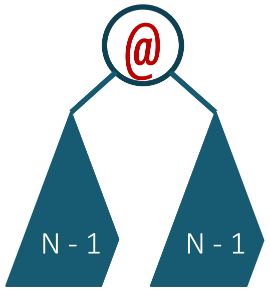
- Proposals due in ten days
- Talk to me about the topic

Scaling enumerative search

Prune

Discard useless subprograms





`m * N²`   `m * (N - 1)²`

Prioritize

Explore more promising candidates first

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

dequeue this first

Order of search

Enumerative search explores programs by depth / size

- Good default bias: small solution is likely to generalize
- But far from perfect

Result:

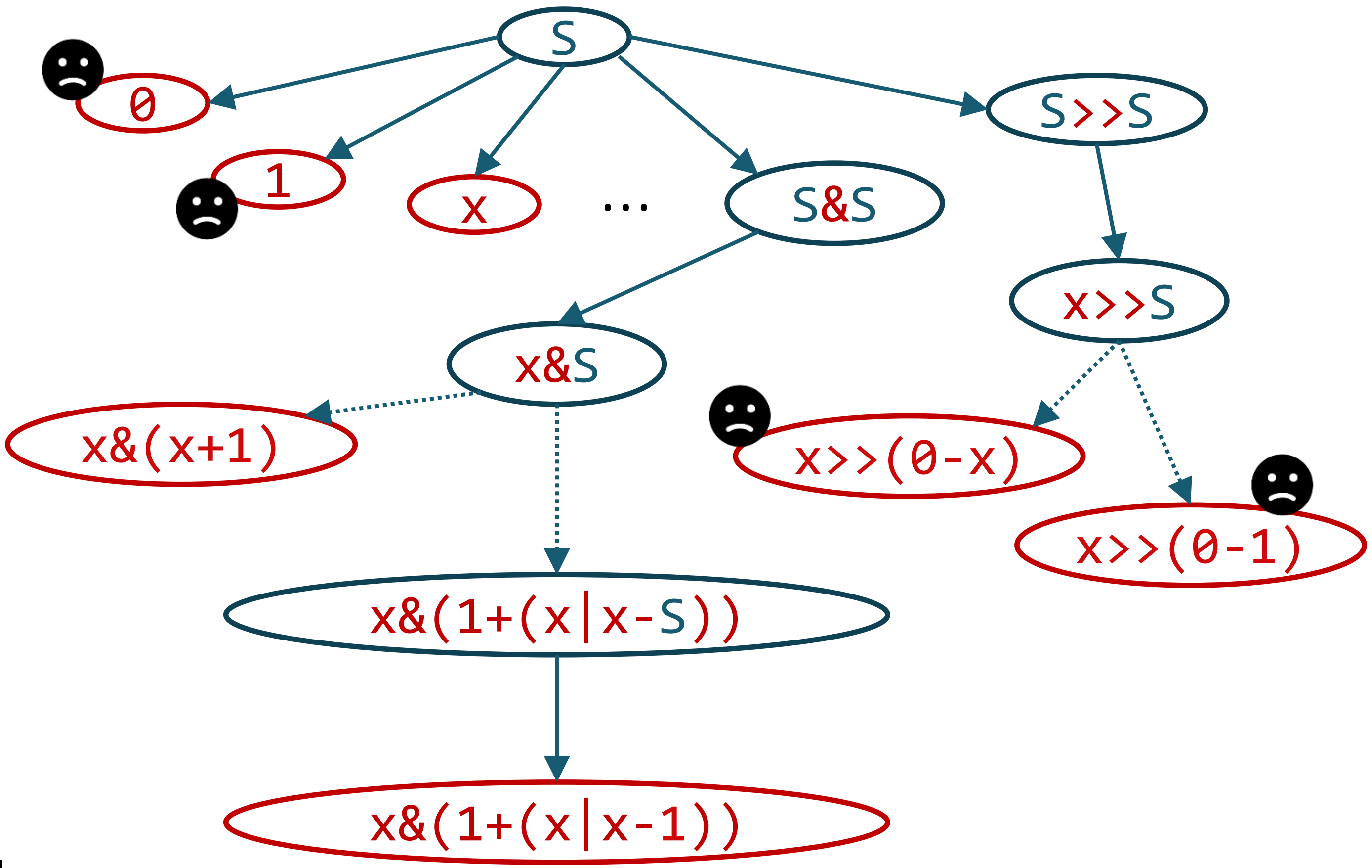
- Scales poorly with the size of the smallest solution to a given spec

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
```



Explores many unlikely programs!

Biasing the search

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

Q1: how do we know which programs are likely?

- hard-code domain knowledge
- learn from a corpus of programs

Q2: how do we use this information to guide search?

- our focus today!

Weighted enumerative search

- DeepCoder
  - Balog et al. DeepCoder: Learning to Write Programs. ICLR’17
- Weighted top-down search
  - Lee, et al: Accelerating Search-Based Program Synthesis using Learned Probabilistic Models. PLDI'18
- Weighted bottom-up search
  - Barke, Peleg, Polikarpova. Just-in-Time Learning for Bottom-Up Enumerative Synthesis. OOPSLA’20
  - Shi, Bieber, Singh. TF-Coder: Program Synthesis for Tensor Manipulations. arXiv

DeepCoder

Input: IO-examples

```
[-17 -3 4 11 0 -5 -9 13 6 6 -8 11]
→ [-12 -20 -32 -36 -68]
```

DeepCoder

Output: Program in a list DSL

```
a <- [int]
b <- Filter (<0) a
c <- Map (*4) b
d <- Sort c
e <- Reverse d
```

DeepCoder

Input: IO-examples

```
[-17 -3 4 11 0 -5 -9 13 6 6 -8 11]
→ [-12 -20 -32 -36 -68]
```

neural network

	(+1)	(-1)	(*2)	(/2)	(*-1)	(**2)	(*3)	(/3)	(*4)	(/4)	(>0)	(>0)	(%2==1)	(%2==0)	HEAD	LAST	MAP
component likelihoods	.0	.0	.1	.0	.0	.0	.0	.0	1.0	.0	.0	1.0	.0	.2	.0	.0	1.0

	FILTER	SORT	REVERSE	TAKE	DROP	ACCESS	ZIPWITH	SCANL1	+	-	*	MIN	MAX	COUNT	MINIMUM	MAXIMUM	SUM
component likelihoods	1.0	1.0	.7	.0	.1	.0	.4	.0	.0	.1	.0	.2	.1	.0	.0	.0	.0

weighted search

Output: Program in
a list DSL

DeepCoder: search strategies

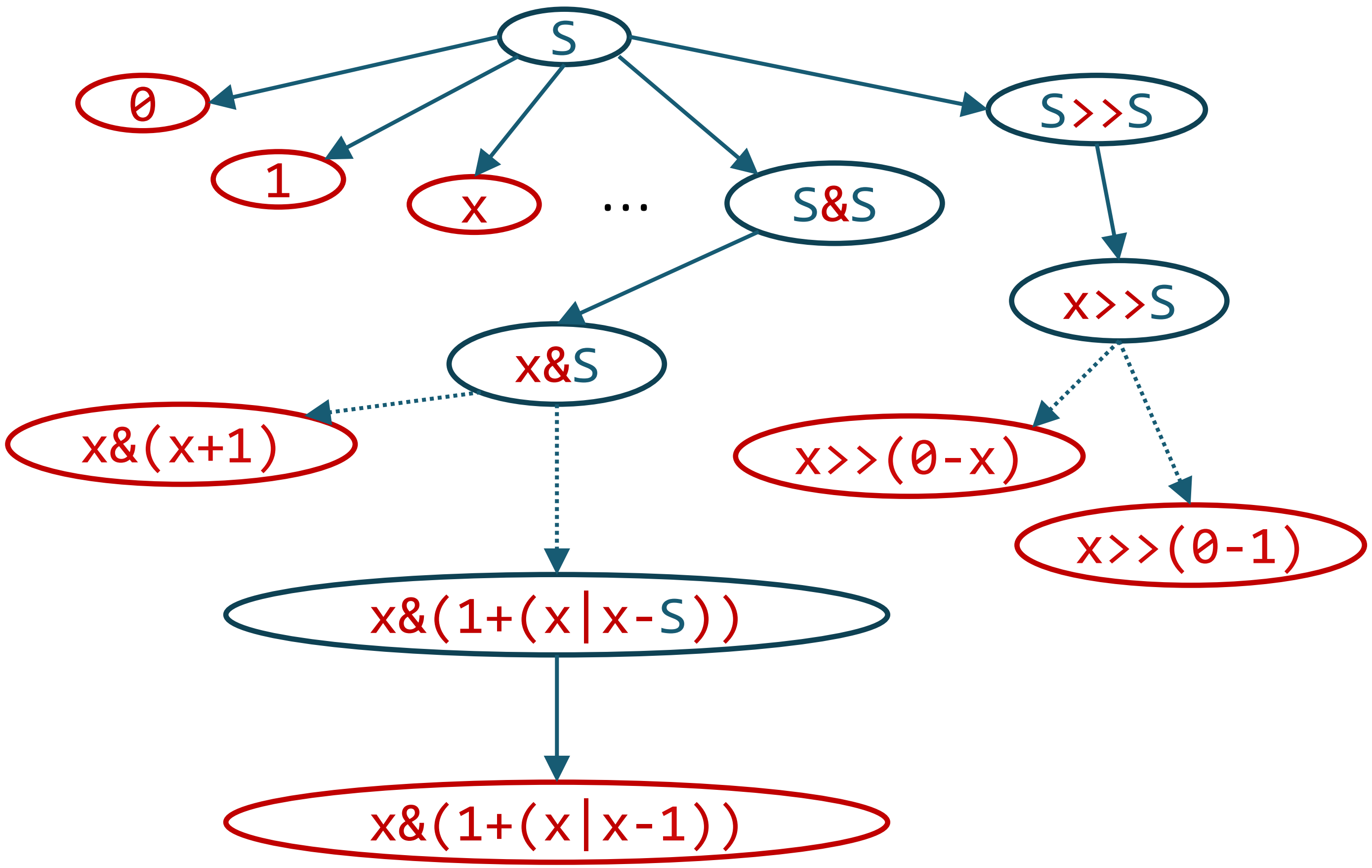
Top-down DFS

- Picks expansions for the current non-terminal in the order of probability

Sort-and-add

- start with N most probable functions
- when search fails, add next N functions

Pros and cons?



Recall: we want to explore programs in the order of likelihood!

Probabilistic Language Models

Originated in Natural Language Processing

In general: a probability distribution over sentences in a language

- $P(s)$ for $s \in L$

In practice:

- must be in a form that can be used to guide search
- for enumerative search: grammar-based (PCFG, PHOG)

Probabilistic CFG (PCFG)

Rule	$\wp(R)$
`S -> 0`	0.13
`S -> 1`	0.13
`S -> x`	0.18
`S -> S + S`	0.11
`S -> S - S`	0.11
`S -> S & S`	0.12
`S -> S | S`	0.12
`S -> S << S`	0.05
`S -> S >> S`	0.05

Encodes the popularity of each operation (terminal)

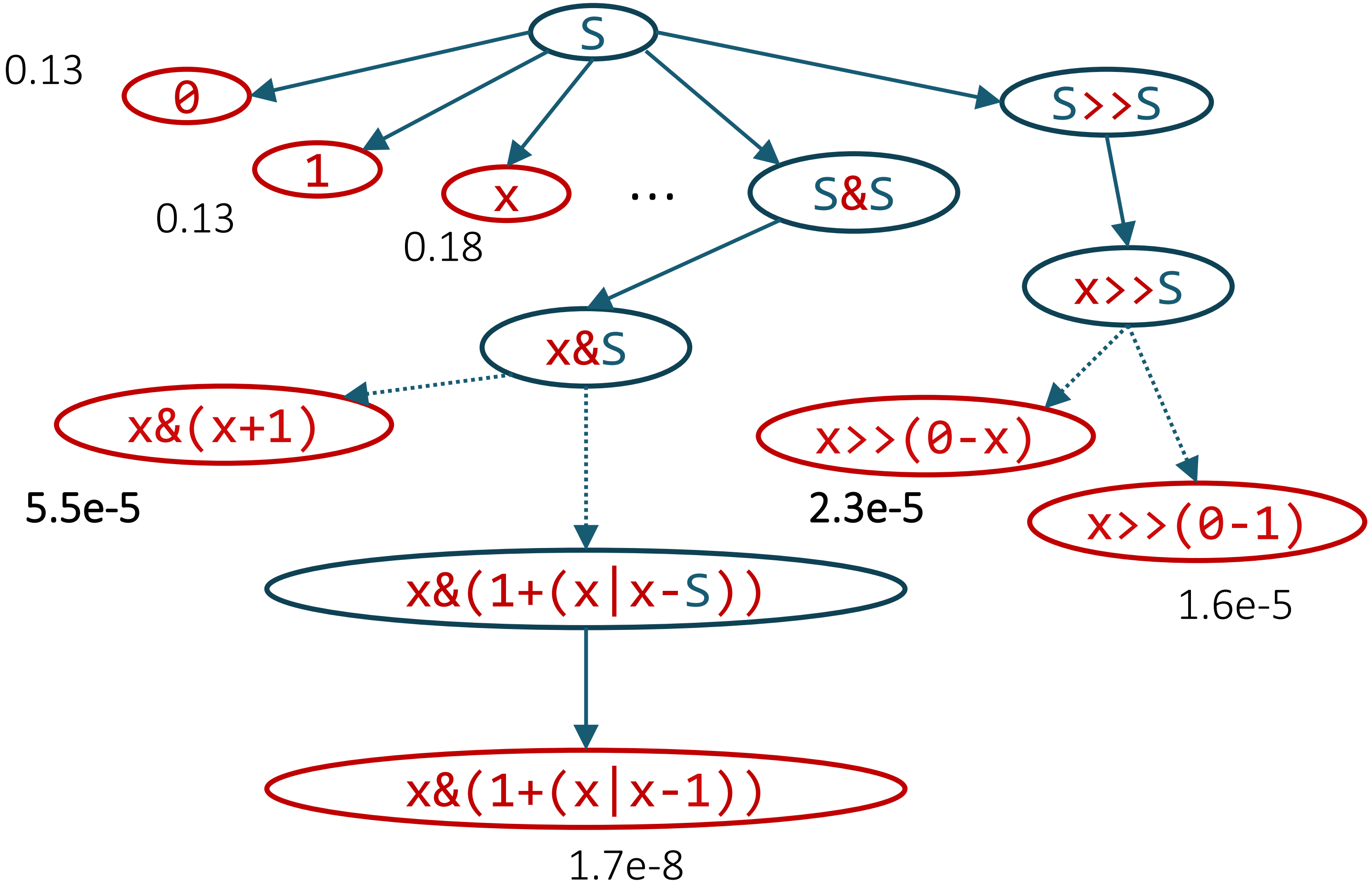
- here: variable more likely than constant, plus more likely than shift

More useful if specific to a spec

Probabilistic CFG (PCFG)

<table>
<tr><th></th><th>$\wp(R)$</th></tr>
<tr><td>S -> 0</td><td>0.13</td></tr>
<tr><td>S -> 1</td><td>0.13</td></tr>
<tr><td>S -> x</td><td>0.18</td></tr>
<tr><td>S -> S + S</td><td>0.11</td></tr>
<tr><td>S -> S - S</td><td>0.11</td></tr>
<tr><td>S -> S & S</td><td>0.12</td></tr>
<tr><td>S -> S | S</td><td>0.12</td></tr>
<tr><td>S -> S << S</td><td>0.05</td></tr>
<tr><td>S -> S >> S</td><td>0.05</td></tr>
</table>

$$\wp(p) = \prod_{R \in S \to^* p} \wp(R)$$



Probabilistic Higher-Order Grammar (PHOG)

[Bielik, Raychev, Vechev '16]

```
N[context] -> rhs
```

		$\wp$
`S[x,-] ->`	`1`	0.72
`S[x,-] ->`	`x`	0.02
`S[x,-] ->`	`S + S`	0.12
`S[x,-] ->`	`S - S`	0.12
`...`		
`S[1,+] ->`	`1`	0.26
`S[1,+] ->`	`x`	0.25
`S[1,+] ->`	`S + S`	0.19
`S[1,+] ->`	`S - S`	0.08

Encodes context-specific likelihood

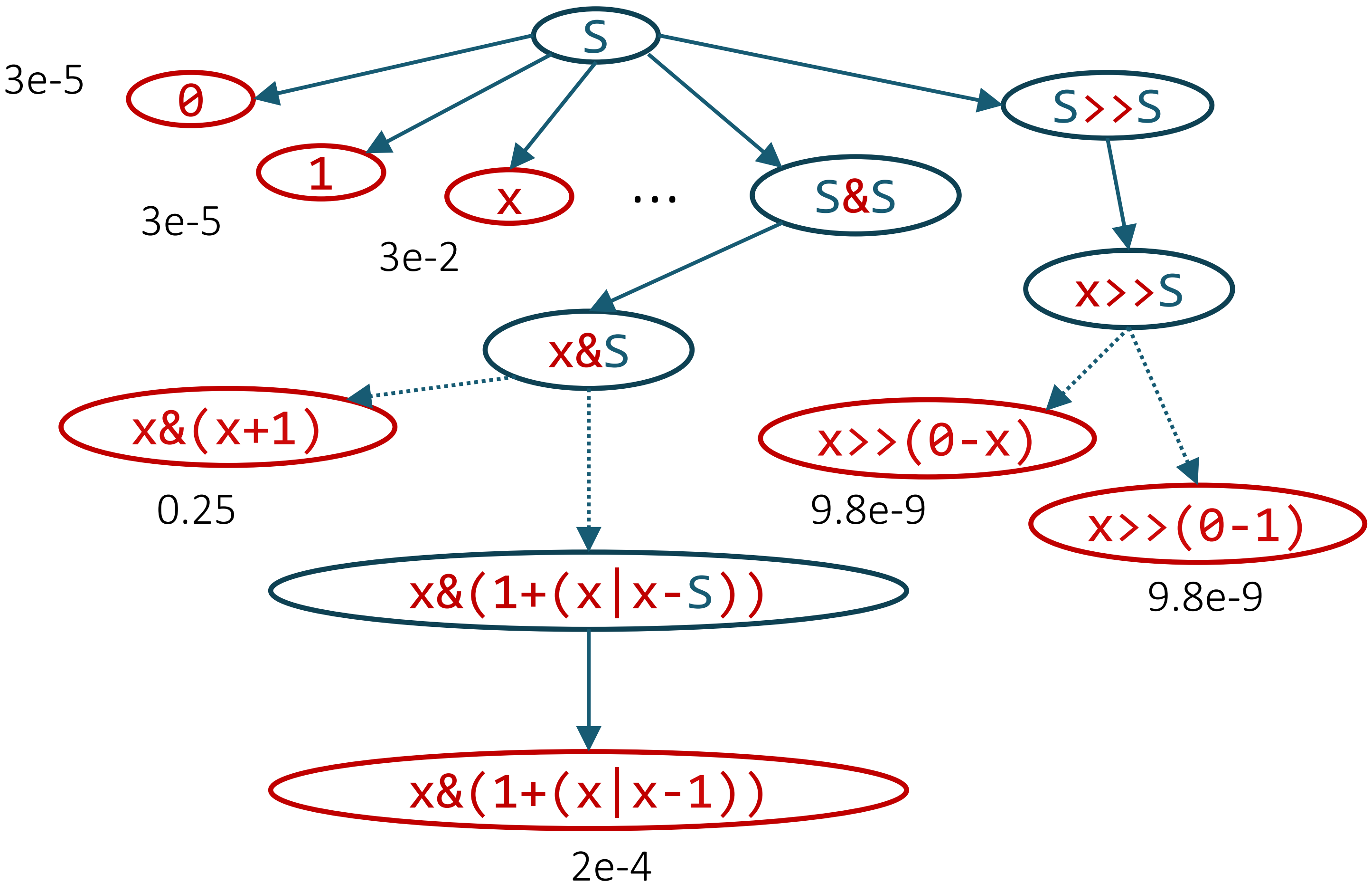
- here: `x` is not likely in `x - ?` but likely in `1 + ?`

Probabilistic Higher-Order Grammar (PHOG)

[Bielik, Raychev, Vechev '16]

```
N[context] -> rhs
```

		℘
`S[x,-] ->`	`1`	0.72
`S[x,-] ->`	`x`	0.02
`S[x,-] ->`	`S + S`	0.12
`S[x,-] ->`	`S - S`	0.12
`...`		
`S[1,+] ->`	`1`	0.26
`S[1,+] ->`	`x`	0.25
`S[1,+] ->`	`S + S`	0.19
`S[1,+] ->`	`S - S`	0.08



Weighted enumerative search

- DeepCoder
  - Balog et al. DeepCoder: Learning to Write Programs. ICLR’17
- Weighted top-down search
  - Lee, et al: Accelerating Search-Based Program Synthesis using Learned Probabilistic Models. PLDI'18
- Weighted bottom-up search
  - Barke, Peleg, Polikarpova. Just-in-Time Learning for Bottom-Up Enumerative Synthesis. OOPSLA’20
  - Shi, Bieber, Singh. TF-Coder: Program Synthesis for Tensor Manipulations. arXiv

Weighted top-down search

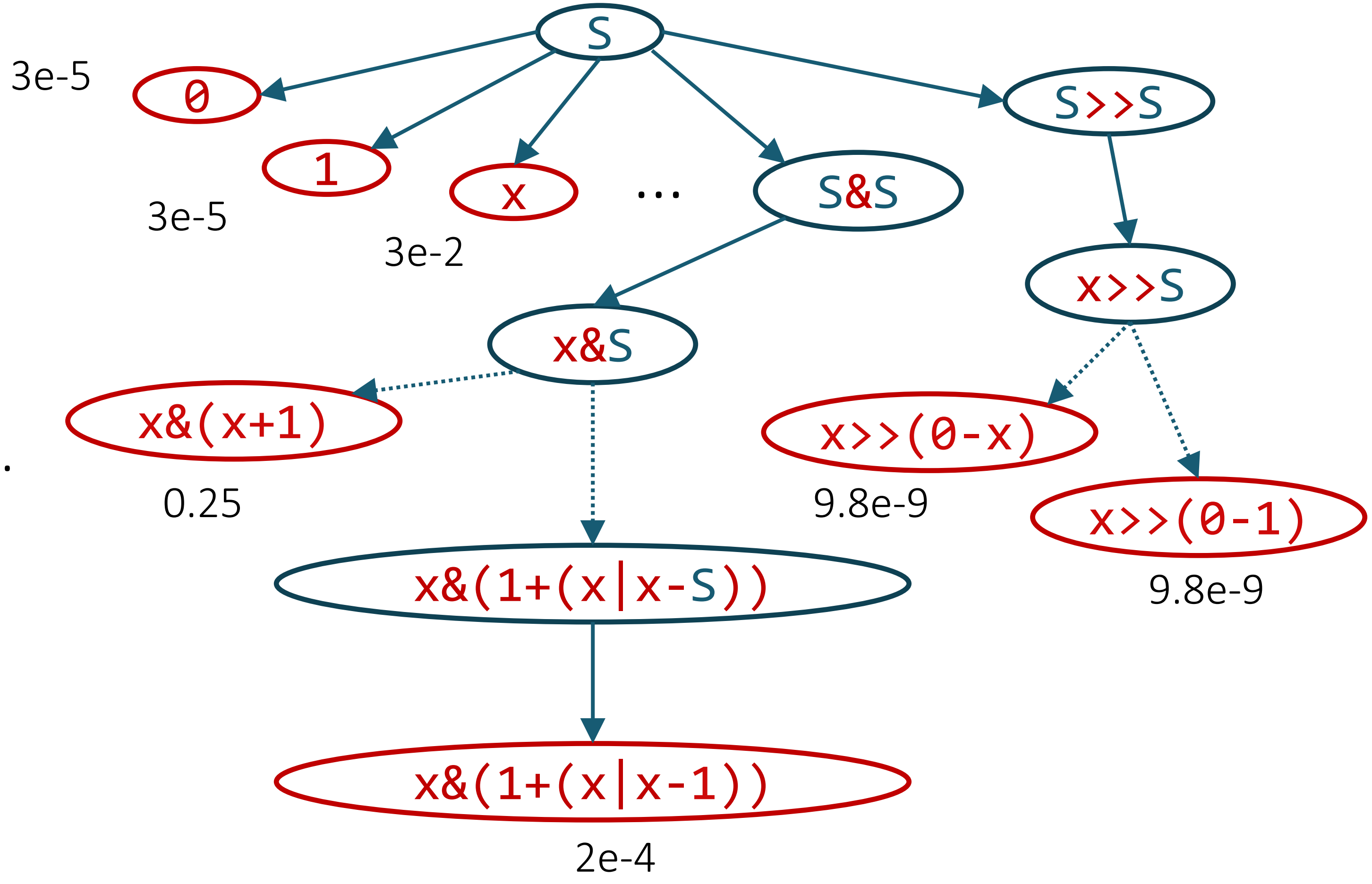
Wanted: explore programs in the order of likelihood

$$\wp(p) = \prod_{R \in S \to^* p} \wp(R)$$

Hard to maximize multiplicative cost…
but easy to minimize additive cost!
= shortest path

$$cost(p) = \sum_{R \in S \to^* p} cost(R)$$

$$-\log_2 \wp(p) = \sum_{R \in S \to^* p} -\log_2 \wp(R)$$



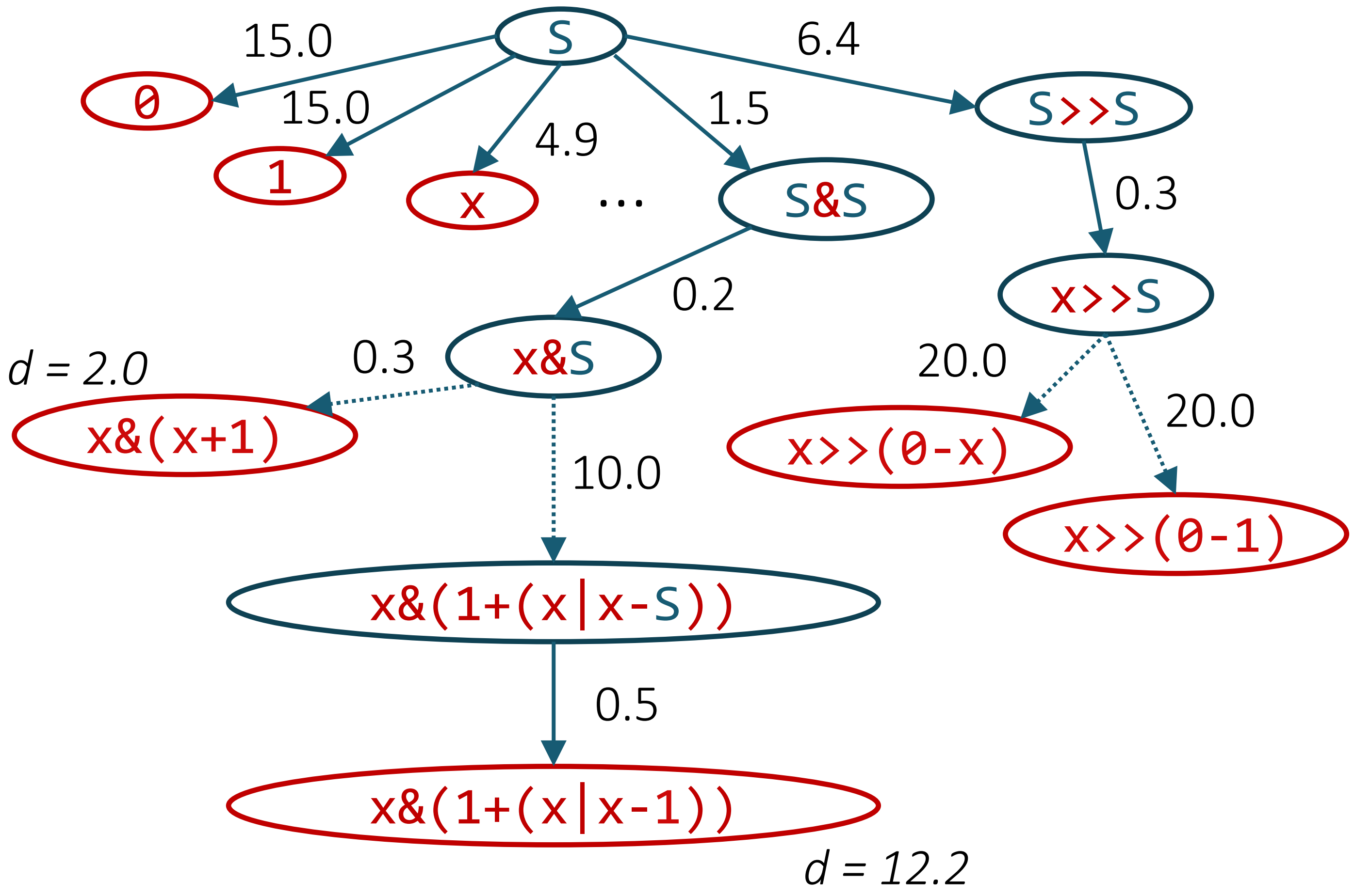
Weighted top-down search

Assigns costs to edges:

$$cost(R) = -\log_2 \wp(R)$$

Now *cost(p) < cost(p')*
iff *p* is more likely than *p'*!

We can use shortest path algo
(e.g. Dijkstra) to search by cost!



Weighted top-down search (Dijkstra)

```
top-down(<Σ, N, R, S>, [i → o]) {
  wl := [<S,0>]
  while (wl != [])
    <p,c> := wl.dequeue_min(c);
    if (ground(p) && p([i]) = [o])
      return p;
    wl.enqueue(unroll(p,c));
}

unroll(p,c) {
  wl' := []
  A := left-most nonterminal in p
  forall (A → rhs) in R:
    wl' += <p[A -> rhs], c + w(A → rhs)>
  return wl';
}
```

wl now stores candidates (nodes) together with their costs

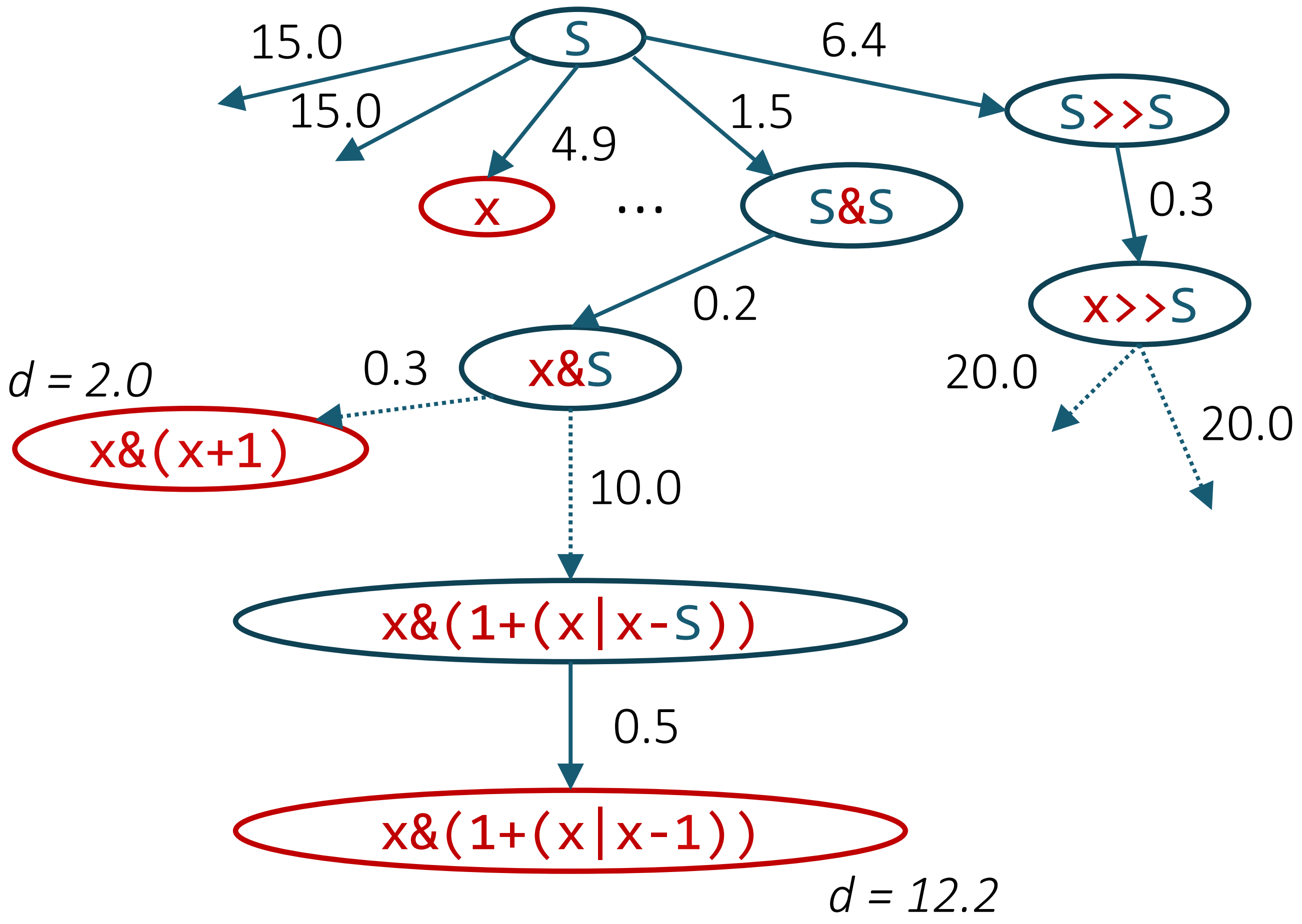
Dequeue the node with minimal cost

Distance to a new node: add the *w(R)*

Can we do better?

Dijkstra: explores a lot of intermediate nodes that don't lead to any cheap leaves

A*: introduce heuristic function *h(p)* that estimates how close we are to the closest leaf



Weighted top-down search (A*)

```
top-down(<Σ, N, R, S>, [i → o]) {
  wl := [<S,0,h(S)>]
  while (wl != [])
    <p,c,h> := wl.dequeue_min(c + h);
    if (ground(p) && p([i]) = [o])
      return p;
    wl.enqueue(unroll(p,c));
}

unroll(p,c) {
  wl' := []
  A := leftmost nonterminal in p
  forall (A → rhs) in R:
    wl' += <p[A -> rhs], c + w(A → rhs),
                                h(p[A -> rhs])>

  return wl';
}
```

Roughly how close is this program to the closest leaf

Weighted enumerative search

- DeepCoder
  - Balog et al. DeepCoder: Learning to Write Programs. ICLR’17
- Weighted top-down search
  - Lee, et al: Accelerating Search-Based Program Synthesis using Learned Probabilistic Models. PLDI'18
- Weighted bottom-up search
  - Barke, Peleg, Polikarpova. Just-in-Time Learning for Bottom-Up Enumerative Synthesis. OOPSLA’20
  - Shi, Bieber, Singh. TF-Coder: Program Synthesis for Tensor Manipulations. arXiv

Bottom-up search (revisited)

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank[A,d] := {} forall A, d
  for d in [0..]:
    forall (A → rhs) in R:
      forall p in new-terms(A→rhs, d, bank):
        if (A = S ∧ p([i]) = [o]):
          return p
        bank[A,d] += p;

new-terms(A → σ(A₁…Aₙ), d, bank):
 if (d = 0 ∧ n = 0) yield σ
 else forall <d₁,…,dₙ> in [0..d-1]ⁿ s.t. max(d₁,…,dₙ) = d-1:
        forall <p₁,…,pₙ> in bank[A₁,d₁] × … × bank[Aₙ,dₙ]:
          yield σ(p₁,…,pₙ)
```

Search by depth

Bottom-up variations

```
new-terms(A → σ(A₁…Aₙ), d, bank):
 if (d = 0 ∧ n = 0) yield σ
 else forall <d₁,…,dₙ> in [0..d-1]ⁿ s.t. max(d₁,…,dₙ) = d-1:
        forall <p₁,…,pₙ> in bank[A₁,d₁] × … × bank[Aₙ,dₙ]:
          yield σ(p₁,…,pₙ)
```

by depth

```
new-terms(A → σ(A₁…Aₙ), s, bank):
 if (s = 1 ∧ n = 0) yield σ
 else forall <s₁,…,sₙ> in [0..s-1]ⁿ s.t. sum(s₁,…,sₙ) = s-1:
        forall <p₁,…,pₙ> in bank[A₁,s₁] × … × bank[Aₙ,sₙ]:
          yield σ(p₁,…,pₙ)
```

by size

```
new-terms(A → σ(A₁…Aₙ), c, bank):
 budget = c - w(A → σ(A₁…Aₙ))
 if (budget = 0 ∧ n = 0) yield σ
 else forall <c₁,…,cₙ> in [0.. budget]ⁿ s.t. sum(c₁,…,cₙ) = budget:
        forall <p₁,…,pₙ> in bank[A₁,c₁] × … × bank[Aₙ,cₙ]:
          yield σ(p₁,…,pₙ)
```

by cost!

Bottom-up by cost: discussion

What kind of cost functions are supported?

- positive
- integer
- compositional:
  $\texttt{cost}(\sigma(p_1,\dots,p_n)) = C + \texttt{cost}(p_1) + \dots + \texttt{cost}(p_n)$

Bottom-up: example

```
L ::= sort(L)    | 10
      L + L      | 3
      x            1
```

cost

by depth

d= 0: `x`

d =1: `sort(x)`
`x + x`

d = 2: `sort(sort(x))`
`sort(x + x)`
`x + sort(x)`
`sort(x) + x`
`x + (x + x)`
`(x + x) + x`

d = 3: …

by size

s= 1: `x`

s =2: `sort(x)`

s = 3: `x + x`
`sort(sort(x))`

s = 4: `sort(x + x)`
`sort(sort(sort(x)))`
`x + sort(x)`
`sort(x) + x`

s = 5: …

by cost

c= 1: `x`

c =2,3,4:

c = 5: `x + x`

c =6,7,8:

c = 9: `x + (x + x)`
`(x + x) + x`

c = 10:

c = 11: `sort(x)`

c = 12:

c = 13: `x + (x + (x + x))`
`(x + x) + (x + x)`
`(x + (x + x)) + x`

Weighted search

Top-down

+ Supports real-valued weights: optimal enumeration order

+ Supports context-dependent weights

Bottom-up

+ Inherits benefits of bottom up: fast, supports OE

Euphony

Q1: What does Euphony use as behavioral constraints? Structural constraint? Search strategy?

- IO Examples (or first-order formula via CEGIS)
- PHOG
- Weighted enumerative search via A*

Euphony

`Rep(x, “-”, S)`

Q2: What would these productions look like if we replaced the PHOG with a PCGF? With 3-grams?

PHOG:

```
S[“-”,Rep] ->  “.”     0.72
S[“-”,Rep] ->  “-”     0.001
S[“-”,Rep] ->  x       0.12
S[“-”,Rep] ->  S + S   0.02
…
```

PCFG:

```
S ->  “.”     0.2
S ->  “-”     0.2
S ->  x       0.3
S ->  S + S   0.2
…
```

3-grams:

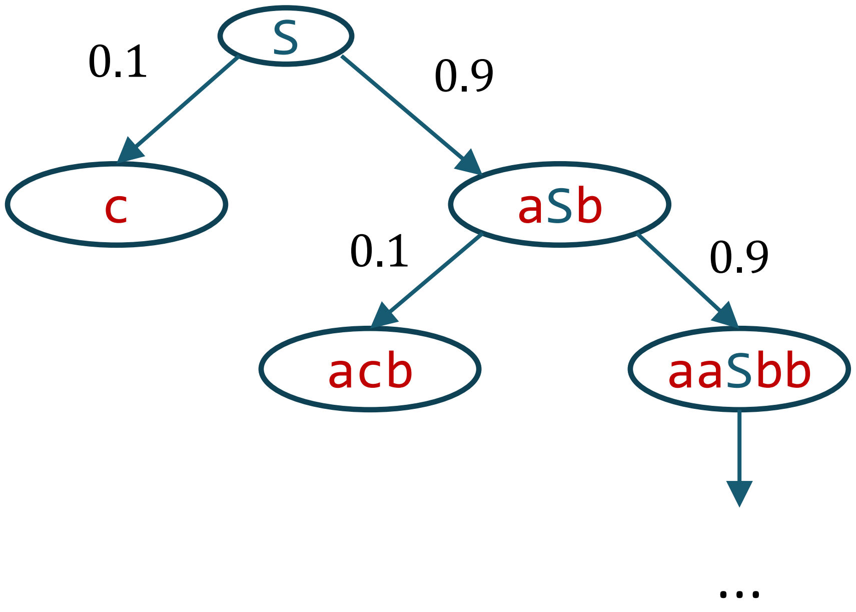
```
S[x,“-”] ->  “.”     0.72
S[x,“-”] ->  “-”     0.001
S[x,“-”] ->  x       0.12
S[x,“-”] ->  S + S   0.02
…
```

Do you think these other probabilistic models would work as well as a PHOG?

Euphony

Q3: What does h(S) = 0.1 mean? Why is it the case?

```
S ->  a S b  0.9
S ->  c      0.1
```



Euphony

Q4: Give an example of sentential forms n_i, n_j and set of points *pts* such that n_i and n_j are equivalent on *pts* but not weakly equivalent

```
S ->  S + S
S ->  x
```

```
n1 = S + S
n2 = x
```

```
pts = [(“” -> “”)]
```

Euphony: strengths

Efficient way to guide search by a probabilistic grammar

- Much better than DeepCoder's sort-and-add
- First to use A* and propose a sound heuristic

Transfer learning for PHOGs

- Remember: abstraction is key to learning models of code!

Extend observational equivalence to top-down search

Euphony: weaknesses

Requires high-quality training data

- for each problem domain!

Transfer learning requires manually designed features

Next week

Topics:

- Representation-based search
- Stochastic search

Paper: Rishabh Singh: BlinkFill: Semisupervised Programming By Example for Syntactic String Transformations. VLDB'16

Projects:

- Proposals due Friday
- Should demonstrate that you started working on the project or at least researched the area
- Once you have decided on the topic, put it on the Google sheet next to any of the team members
- If you haven't decided, talk to me after class or in OH