#7: Weighted Enumerative Search

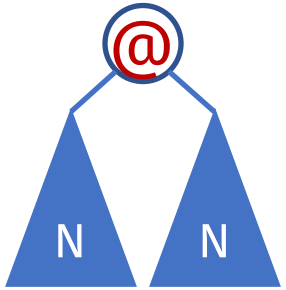
Sankha Narayan Guria

EECS 700: Introduction to Program Synthesis

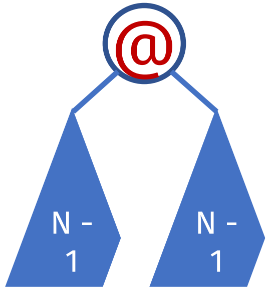
Scaling enumerative search

Prune

- Discard useless subprograms



$m * N^2$



$m * (N - 1)^2$

Prioritize

- Explore more promising candidates first

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

dequeue this first

Order of search

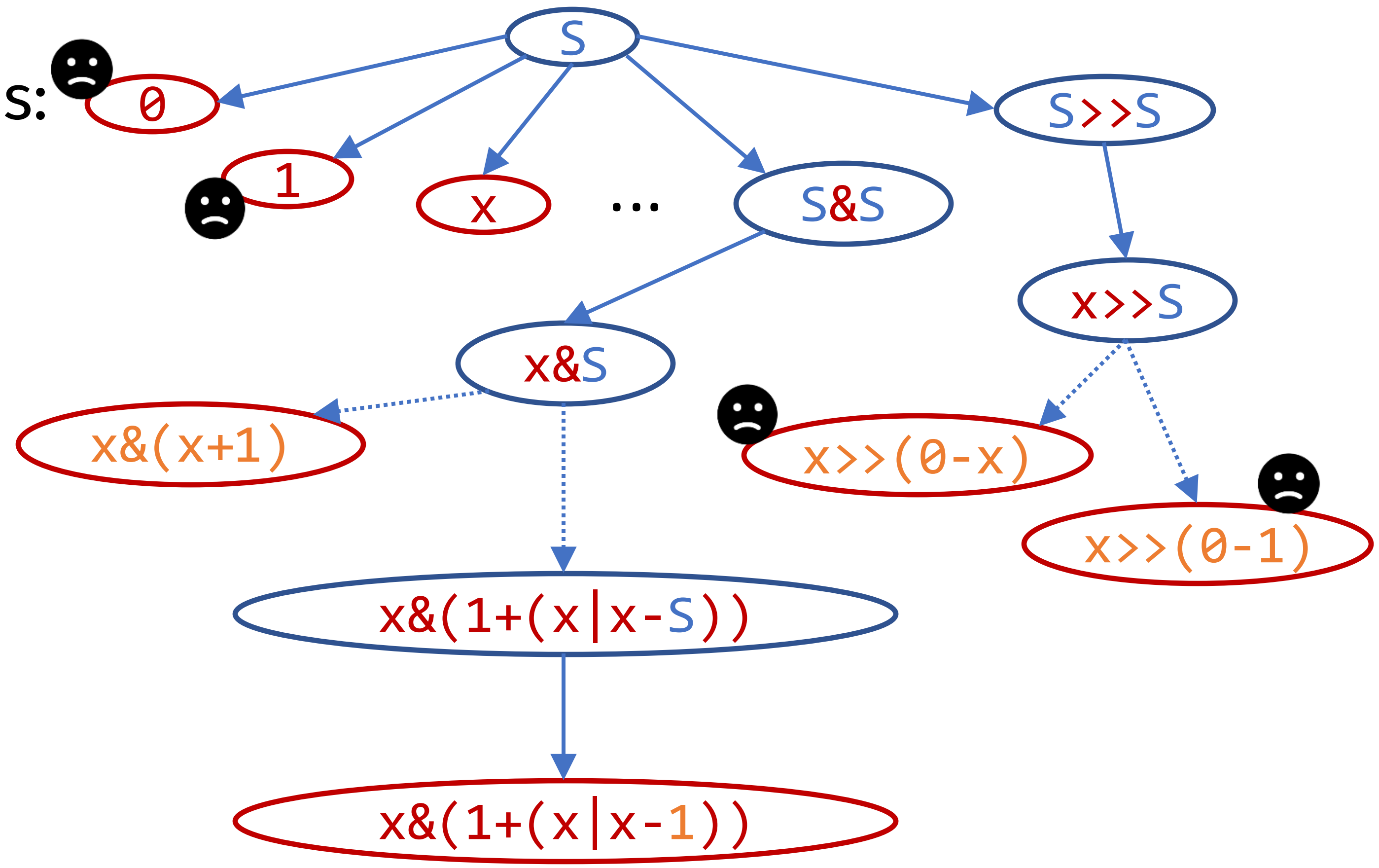
- Enumerative search explores programs by depth / size
  - Good default bias: small solution is likely to generalize
  - But far from perfect
- Result:
  - Scales poorly with the size of the smallest solution to a given spec

Top-down search (revisited)

Turn off the rightmost sequence of 1s:

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
  - learn on the fly
- **Q2:** how do we use this information to guide search?
  - our focus today!

Weighted enumerative search

Example: DeepCoder

Balog et al. DeepCoder: Learning to Write Programs. ICLR'17

Probabilistic Grammars

Weighted top-down search

Weighted bottom-up search

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

component weights

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

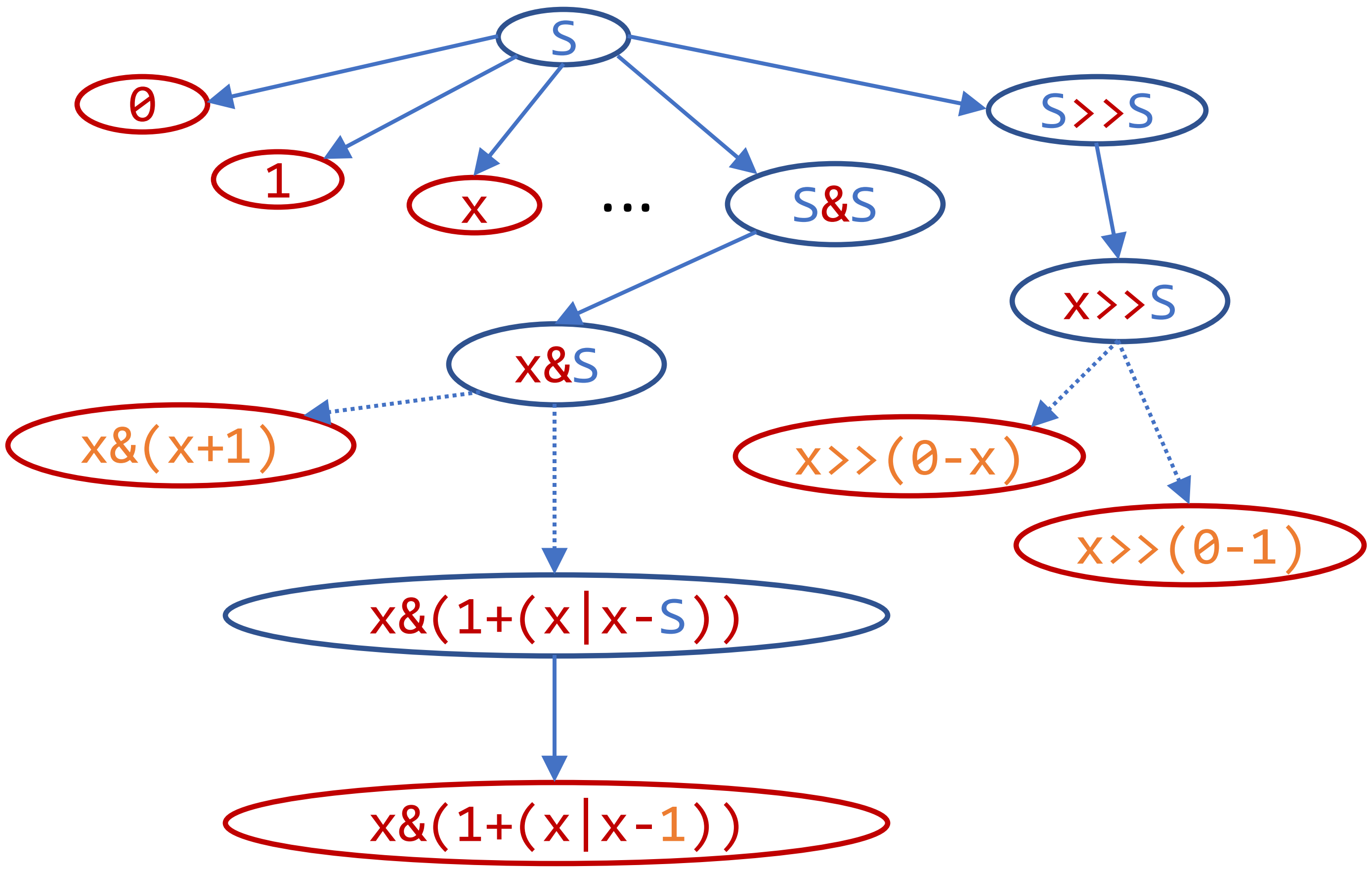
FILTER	SORT	REVERSE	TAKE	DROP	ACCESS	ZIPWITH	SCANL1	+	-	*	MIN	MAX	COUNT	MINIMUM	MAXIMUM	SUM
1.0	1.0	.7	.0	.1	.0	.4	.0	.0	.1	.0	.2	.1	.0	.0	.0	.0

↓ weighted search

Output: Program in a list DSL
Goal: Minimize sum of component weights

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: goal is to explore programs in the order of total weight!

Weighted enumerative search

DeepCoder

Probabilistic Grammars

Weighted top-down search

Weighted bottom-up search

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: probabilistic (or weighted) grammars

Probabilistic (Tree) Grammar

regular tree grammar

production probability (given context)

$$\langle G, \wp \rangle$$

- Production probability: $\wp$: $R \times T_\Sigma(N) \to [0,1]$
  - for example: $\wp(S \to x \mid S) = 0.3$ $\wp(S \to x \mid x - S) = 0.0001$
  - only defined for contexts where rule's LHS is the leftmost non-terminal
  - probabilities of all productions in the same context add up to 1:

$$\forall \tau. S \to^* \tau \wedge \tau \notin T_\Sigma \Rightarrow \sum_{r \in dom(P(.|\tau))} P(r \mid \tau) = 1$$

- Term probability:
  - let $S = \tau_0 \to^{r_1} \tau_1 \to^{r_2} \dots \to^{r_n} \tau_n = \tau$ be the unique derivation of partial program τ

$$\wp(\tau) = \prod_{i=1}^{n} \wp(r_i \mid \tau_i)$$

Types of context

$$\wp\colon \mathrm{R} \times T_{\Sigma}(N) \to [0,1]$$

- In general, can depend on any part of the context term!
- But this is unwieldy
  - bad for learning
  - bad for (some) search algorithms
- In practice we want to restrict the context
  - PCFG
  - n-grams
  - PHOG

Probabilistic Context-Free Grammars (PCFG)

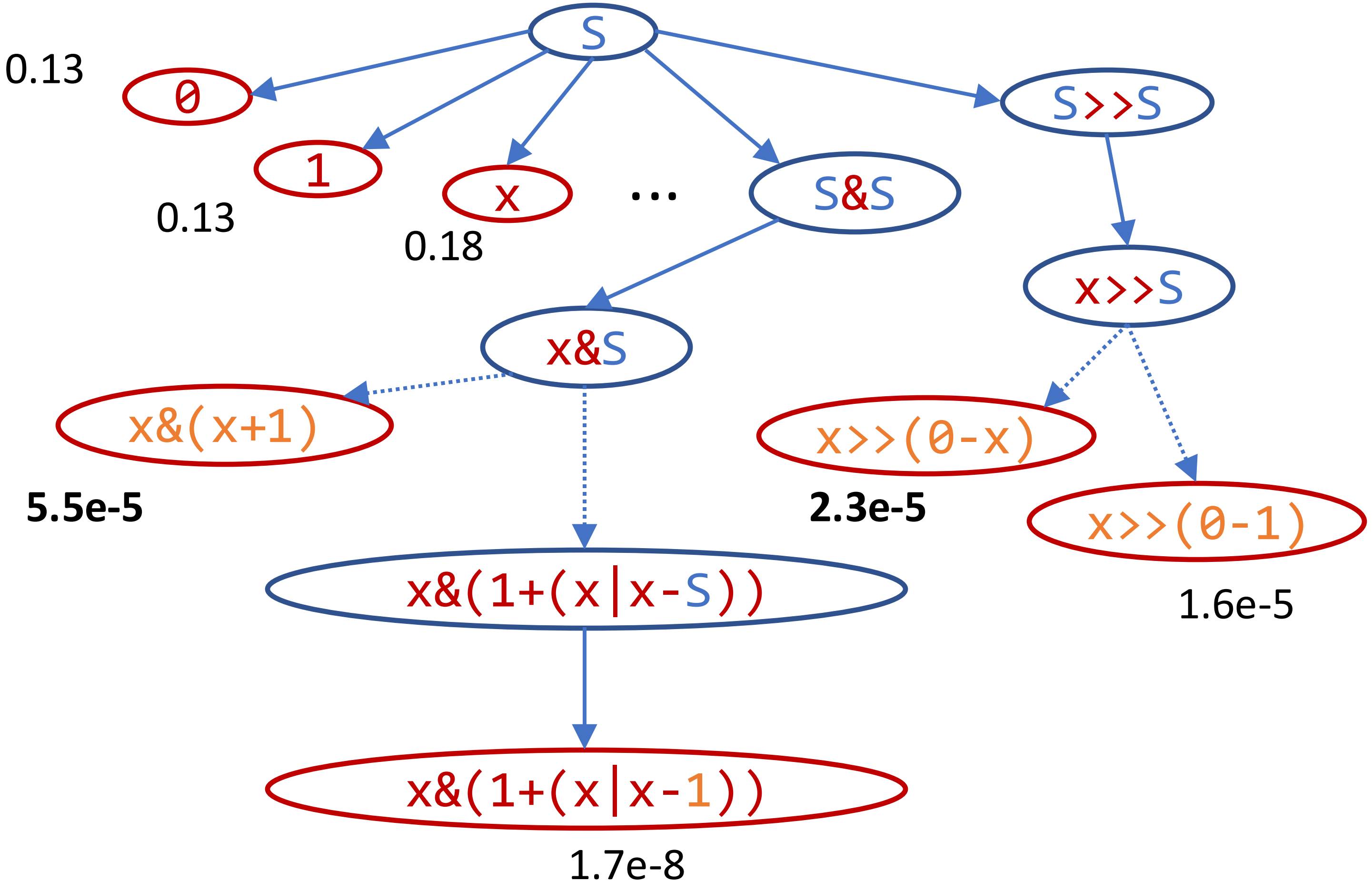
	$\wp(R)$
`S -> 0`	0.13
`S -> 1`	0.13
`S -> x`	0.18
`S -> S + S`	0.11
`S -> S - S`	0.11
`S -> S & S`	0.12
`S -> S | S`	0.12
`S -> S << S`	0.05
`S -> S >> S`	0.05

$$\wp\colon \mathrm{R} \to [0,1]$$

- Encodes the popularity of each production (operation)
  - here: variable more likely than constant, plus more likely than shift

Probabilistic Context-Free Grammars (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

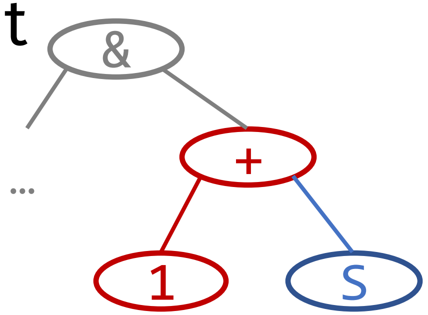


N-grams

```
N[left sibling, parent] -> 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 likelihood of a production in a fixed context
  - fixed set of AST nodes determined relative to the focus nonterminal
  - e.g. left sibling and parent

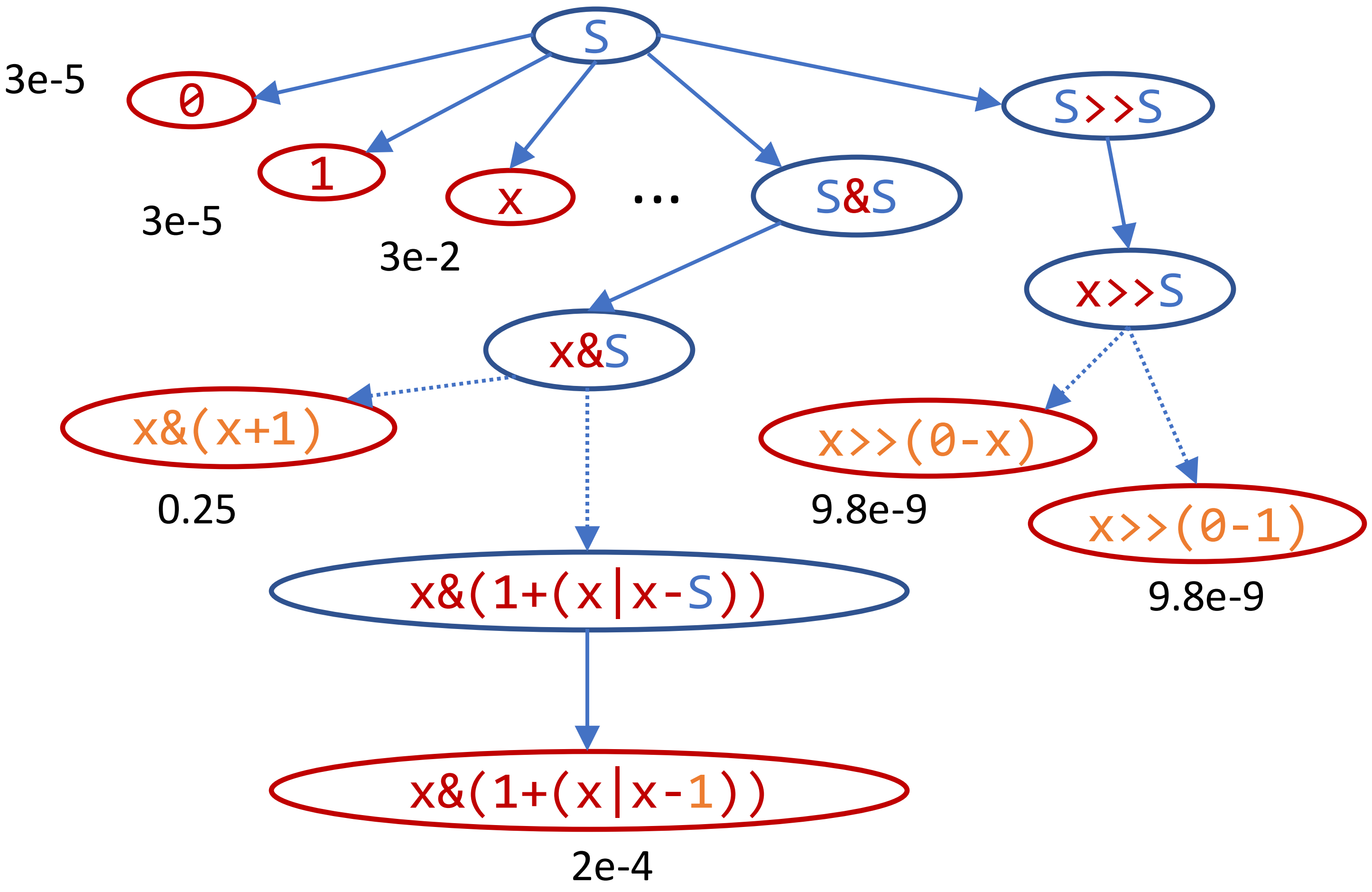


  - here: x is not likely in x - S? but likely in 1 + S?

N-grams

`N[left sibling, parent] -> 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



Probabilistic Higher-Order Grammar (PHOG)

- The same fixed context might not work for every problem
- Idea:

1. 1. define context as a program that traverses the AST
2. 2. learn the best context together with probabilities

Bielik, Raychev, Vechev. PHOG: Probabilistic Model for Code. ICML'16

Conditional models

- **Unconditional model**
- Which programs are more natural in this DSL?

+ easier to get data / learn

- need more context to capture interesting properties

- **Conditional model**
- Which programs are more likely to solve a given spec?

- harder to get data / learn

- can get away with less context

Weighted enumerative search

DeepCoder

Probabilistic Grammars

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 probability

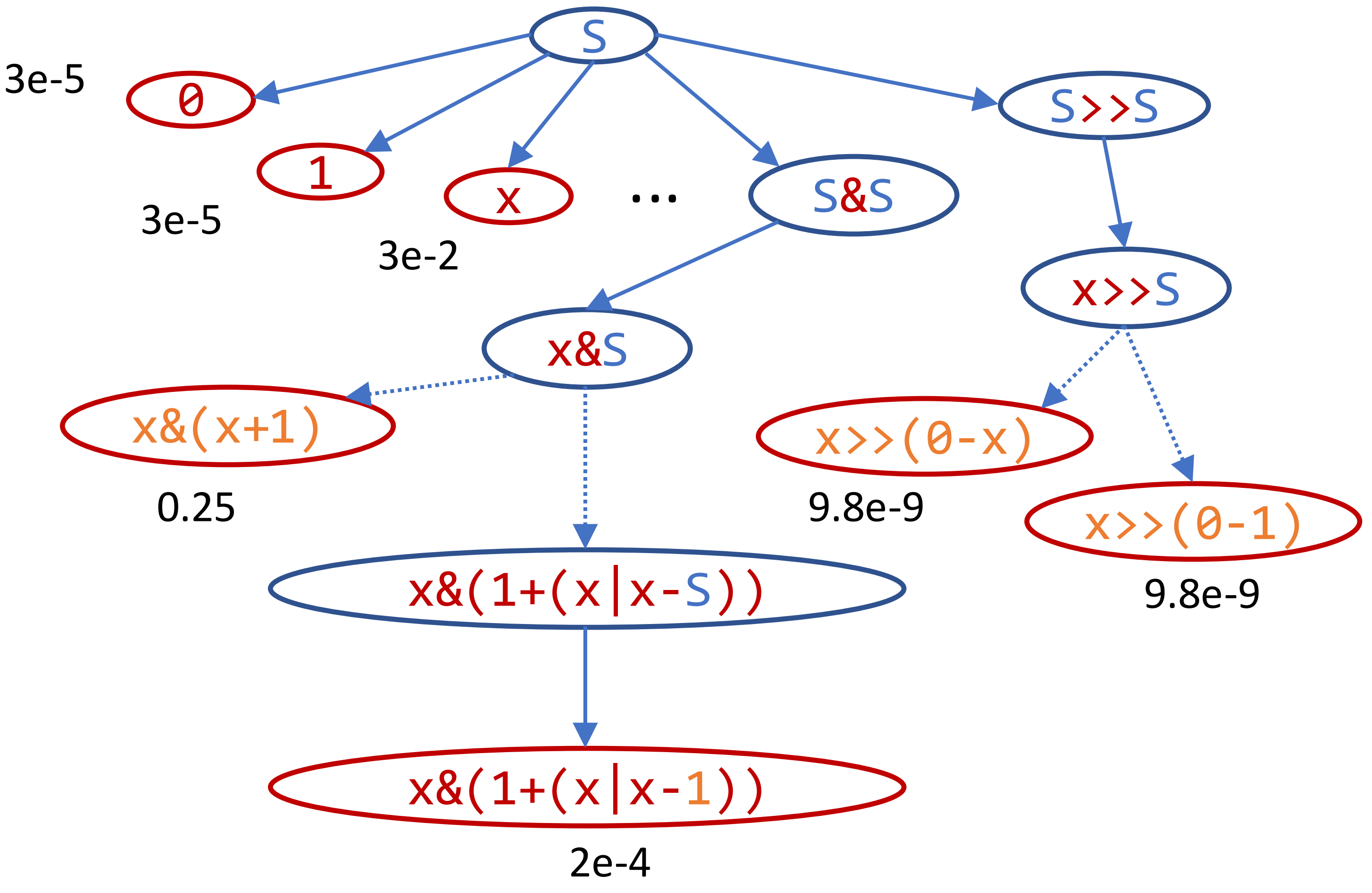
$$\wp(t) = \prod_{(r_i,\tau_i)\in S\to^* t} \wp(r_i \mid \tau_i)$$

Hard to maximize multiplicative cost... but easy to minimize additive cost!

= shortest path

$$cost(t) = \sum_{(r_i,\tau_i)\in S\to^* t} weight(r_i \mid \tau_i)$$

$$-\log_2 \wp(t) = \sum_{(r_i,\tau_i)\in S\to^* t} -\log_2 \wp(r_i \mid \tau_i)$$



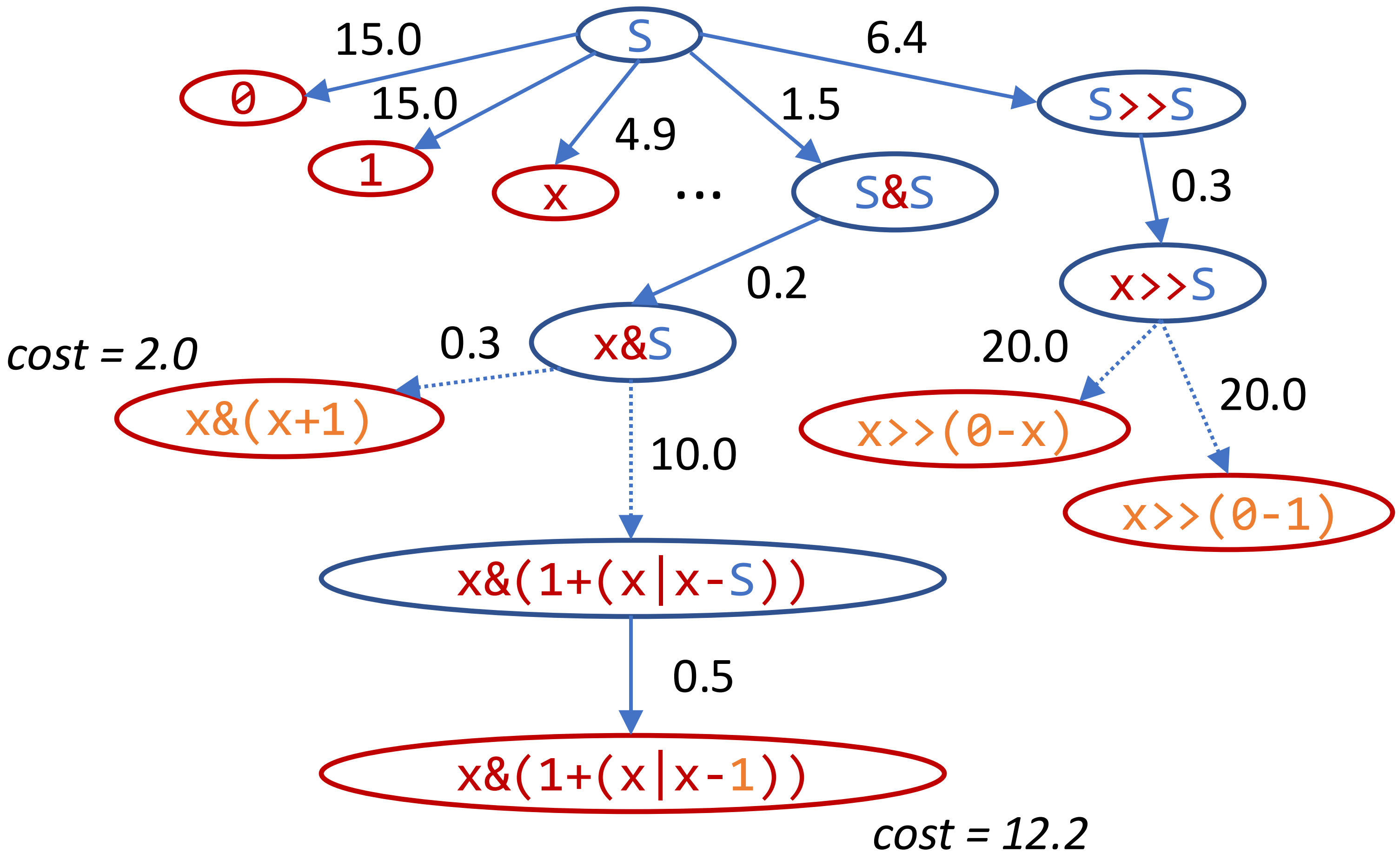
Weighted top-down search

Assigns weights to edges:

$$weight(r_i \mid \tau_i) = -\log_2 \wp(r_i \mid \tau_i)$$

Now *cost(t)* < *cost(t')*
iff *t* is more likely than *t'*!

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 != [])
    <τ,c> := wl.dequeue_min(c);
    if (complete(τ) && τ([i]) = [o])
      return τ;
    wl.enqueue(unroll(τ,c));
}

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

wl now stores candidates (nodes) together with their costs

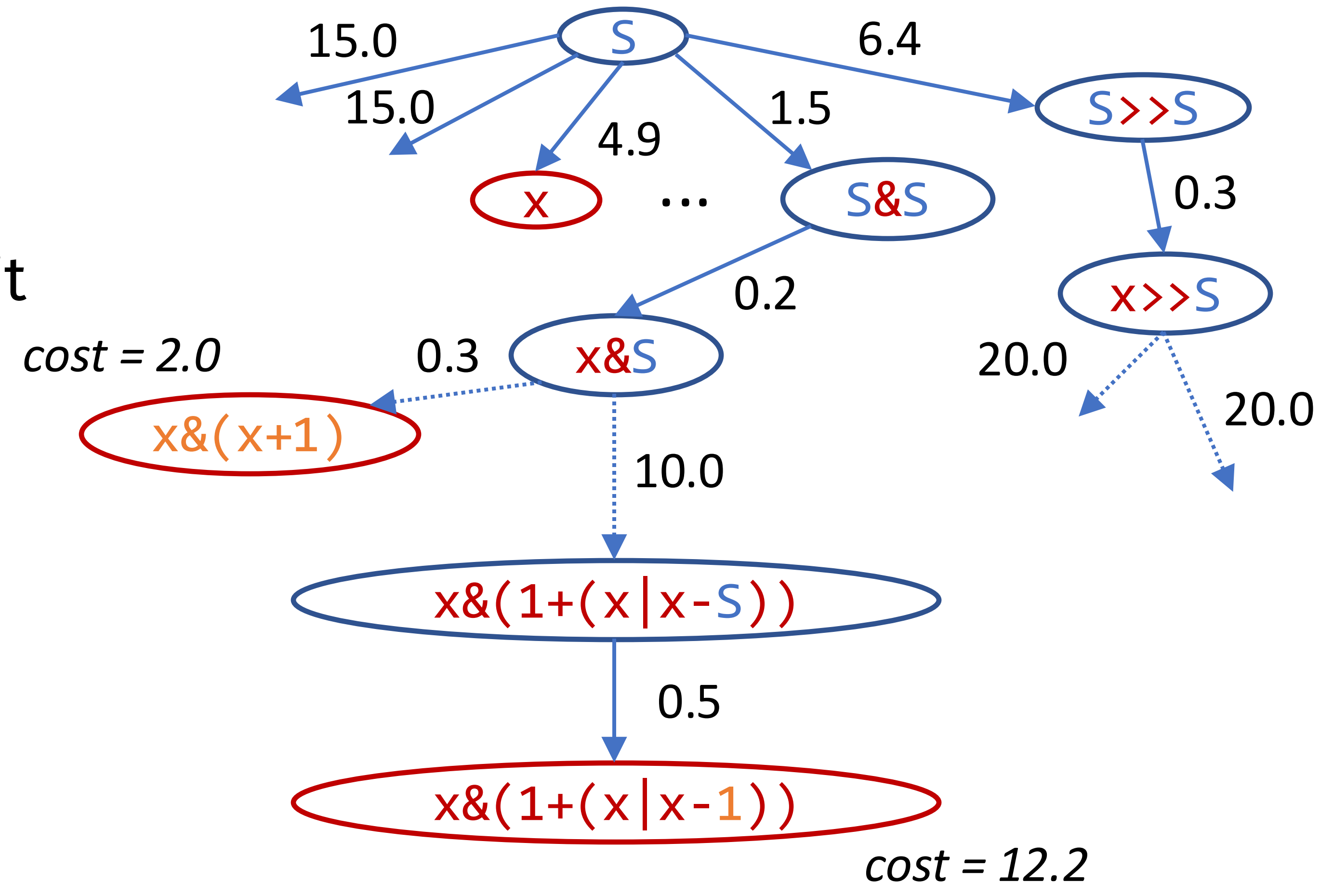
Dequeue the node with minimal cost

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 != [])
    <τ,c,h> := wl.dequeue_min(c + h);
    if (complete(τ) && τ([i]) = [o])
      return τ;
    wl.enqueue(unroll(τ,c));
}

unroll(τ,c) {
  wl’ := []
  A := leftmost nonterminal in τ
  forall (A → rhs) in R:
    wl’ += <τ[A -> rhs], c + w(A → rhs|τ),
                                h(τ[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. TOPLAS’22

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(A_1…A_n), d, bank):
 if (d = 0 ∧ n = 0) yield σ
 else forall <d_1,…,d_n> in [0..d-1]^n s.t. max(d_1,…,d_n) = d-1:
       forall <p_1,…,p_n> in bank[A_1,d_1] × … × bank[A_n,d_n]:
         yield σ(p_1,…,p_n)
```

by depth

```
new-terms(A → σ(A_1…A_n), s, bank):
 if (s = 1 ∧ n = 0) yield σ
 else forall <s_1,…,s_n> in [0..s-1]^n s.t. sum(s_1,…,s_n) = s-1:
       forall <p_1,…,p_n> in bank[A_1,s_1] × … × bank[A_n,s_n]:
         yield σ(p_1,…,p_n)
```

by size

```
new-terms(A → σ(A_1…A_n), c, bank):
 budget = c - w(A → σ(A_1…A_n))
 if (budget = 0 ∧ n = 0) yield σ
 else forall <c_1,…,c_n> in [0.. budget]^n s.t. sum(c_1,…,c_n) = budget:
       forall <p_1,…,p_n> in bank[A_1,c_1] × … × bank[A_n,c_n]:
         yield σ(p_1,…,p_n)
```

by cost!

Bottom-up by cost: discussion

- What kind of cost functions are supported?
  - positive
  - integer
  - context-free

Bottom-up: example

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

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: dynamic programming, observational equivalence