Lecture 4
Search Bias

Today

EUSolver discussion

Search space prioritization

- statistical models of code
- how to learn them
- how to use them during search

EUSover

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

- First-order formula
- Conditional expression grammar
- Bottom-up enumerative + pruning

Why do they need the specification to be pointwise?

- Example of a non-pointwise spec?
- How would it break the enumerative solver?

EUSover

Q2: What are pruning/decomposition techniques EUSolver uses to speed up the search?

- Condition abduction + special form of equivalence reduction

Why does EUSolver keep generating additional terms when all inputs are covered?

Branch-wise verification: are more counter-examples always better?

EUSolver

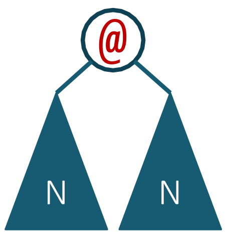
Q3: What would be a naive alternative to decision tree learning for synthesizing branch conditions?

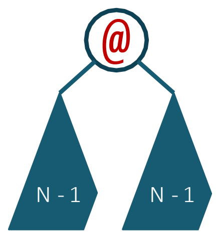
- Learn atomic predicates that precisely classify points
  - why is this worse?
- Next best thing is decision tree learning w/o heuristics
  - why is this worse?

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 in the order of depth

- 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
- If spec is insufficient: plays monkey's paw

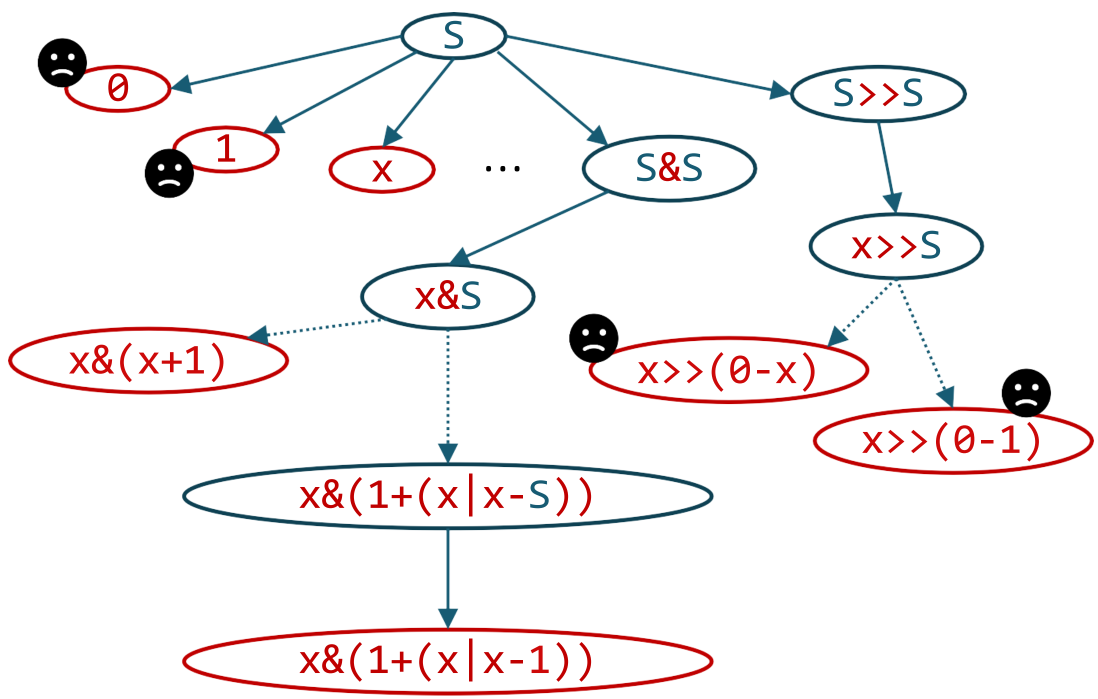
Top-down search (revisited)

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?

- learn a statistical (probabilistic) model from a corpus of programs!

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

Statistical 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
- and also that can be learn from the data we have

Statistical Models in Synthesis

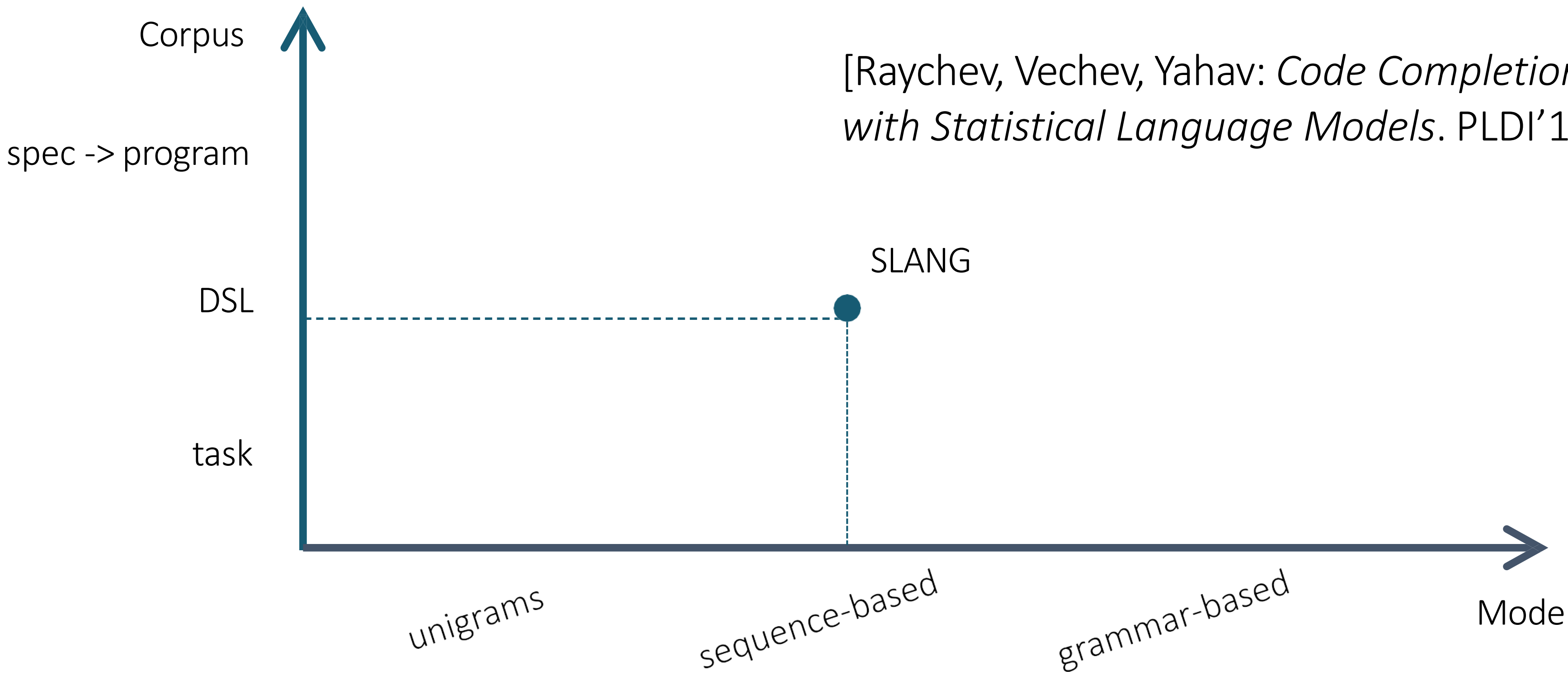
Kinds of corpora:

- All programs from DSL: what are natural programs in this DSL?
- Solutions to specific task (e.g. for MOOCs)
- Spec-program pairs: what are likely programs for this spec?

Kinds of models:

- Likely components (aka unigrams)
- Sequence-based: n-grams, RNN (LSTM)
- Grammar-based: PCFG, PHOG

Statistical Models in Synthesis



[Raychev, Vechev, Yahav: *Code Completion with Statistical Language Models*. PLDI'14]

SLANG

Input: code snippet with holes

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
  ? {smsMgr, msgList}  // (H1)
} else {
  ? {smsMgr, message}  // (H2)
}
```

SLANG

Output: holes completed with (sequences) of method calls

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
  smsMgr.sendMultipartTextMessage(...msgList...);
} else {
  smsMgr.sendTextMessage(...message...);
}
```

SLANG: inference phase

code snippet with holes

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
       smsMgr.divideMsg(message);
  ? {smsMgr, msgList}  // (H1)
} else {
  ? {smsMgr, message}  // (H2)
}
```

static analysis →

abstract histories of objects

$\texttt{smsMgr} \mapsto \{\langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \textbf{H2}\rangle\ ,\ \langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \texttt{divideMsg}, 0\rangle \cdot \langle \textbf{H1}\rangle\}$

$\texttt{message} \mapsto \{\langle \texttt{length}, 0\rangle \cdot \langle \textbf{H1}\rangle \langle \texttt{length}, 0\rangle \cdot \langle \textbf{H2}\rangle\}$

$\texttt{msgList} \mapsto \{\langle \texttt{divideMsg}, \texttt{ret}\rangle \cdot \langle \textbf{H1}\rangle\}$

learned generative model:

- bigrams suggest candidates
- n-grams / RNNs rank them

Partial History	Id	Candidate Completions	Pr
$\langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \textbf{H2}, \textbf{smsMgr}\rangle$	11	$\langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \texttt{sendTextMessage}, 0\rangle$	0.0073
	12	$\langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \texttt{sendMultipartTextMessage}, 0\rangle$	0.0010
$\langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \texttt{divideMsg}, 0\rangle \cdot \langle \textbf{H1}, \textbf{smsMgr}\rangle$	21	$\langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \texttt{divideMsg}, 0\rangle \cdot \langle \texttt{sendMultipartTextMessage}, 0\rangle$	0.0033
	22	$\langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \texttt{divideMsg}, 0\rangle \cdot \langle \texttt{sendTextMessage}, 0\rangle$	0.0016
$\langle \texttt{length}, 0\rangle \cdot \langle \textbf{H2}, \textbf{message}\rangle$	31	$\langle \texttt{length}, 0\rangle \cdot \langle \texttt{length}, 0\rangle$	0.0132
	32	$\langle \texttt{length}, 0\rangle \cdot \langle \texttt{split}, 0\rangle$	0.0080
	33	$\langle \texttt{length}, 0\rangle \cdot \langle \texttt{sendTextMessage}, 3\rangle$	0.0017
	34	$\langle \texttt{length}, 0\rangle \cdot \langle \texttt{sendMultipartTextMessage}, 1\rangle$	0.0001
$\langle \texttt{divideMsg}, \texttt{ret}\rangle \cdot \langle \textbf{H1}, \textbf{msgList}\rangle$	41	$\langle \texttt{divideMsg}, \texttt{ret}\rangle \cdot \langle \texttt{sendMultipartTextMessage}, 3\rangle$	0.0821

SLANG

Predicts completions for sequences of API calls

Treats programs as (sets of) abstract histories

- Performs static analysis to abstract programs into finite histories

Training: learns bigrams, n-grams, RNNs on histories

Inference: given a history with holes

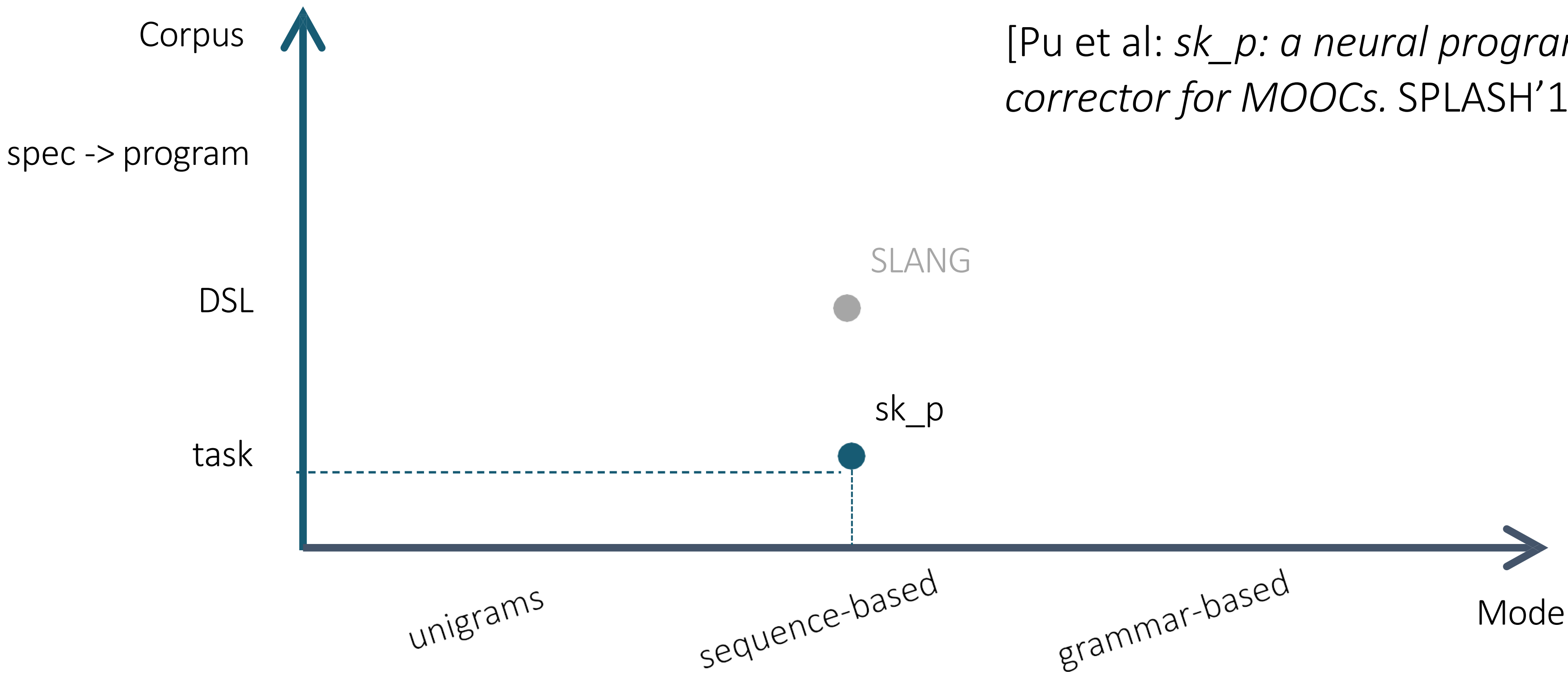
- Uses bigrams to get possible completions
- Uses n-grams / RNN to rank them
- Combines history completions into a coherent program

Features: fast (very little search)

Limitations: all invocation pairs must appear in training set

Statistical Models in Synthesis

[Pu et al: *sk_p: a neural program corrector for MOOCs*. SPLASH'16]



sk_p

Program corrections for MOOCs

Treats programs as a sequence of tokens

- Abstracts away variables names

Uses the skipgram model to predict which statement is most likely to occur between the two

Features

- Can repair syntax errors

Limitations

- Needs all algorithmically distinct solutions to appear in the training set

Skipgram

"I like to write computer programs with an editor"

"I like to write computer with an editor"

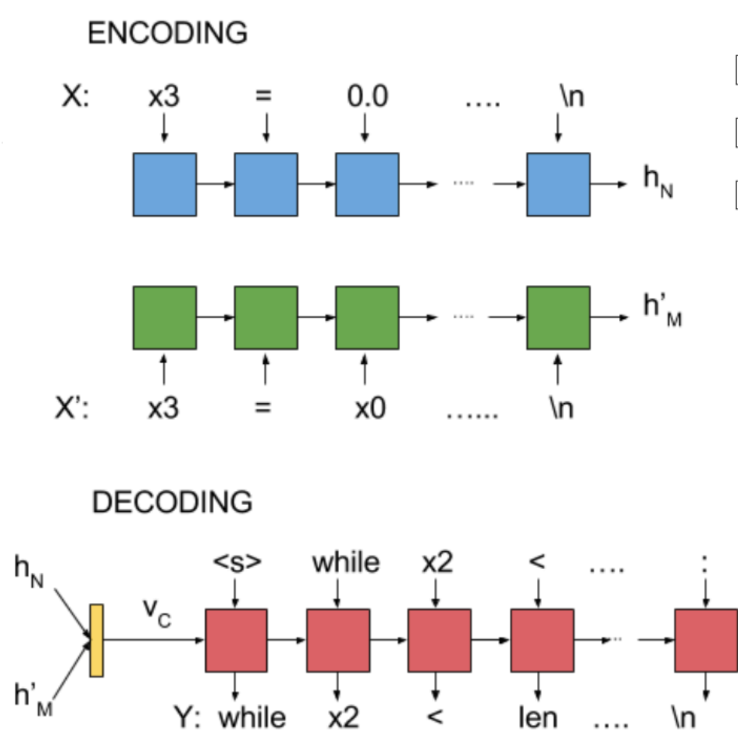
Skipgram for synthesis

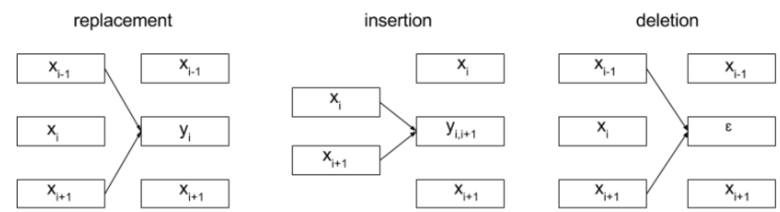
Example Training Input:

```
else:
[          ]
  x2 += x0[x3] * (x1 ** x
```

Example Training Output:

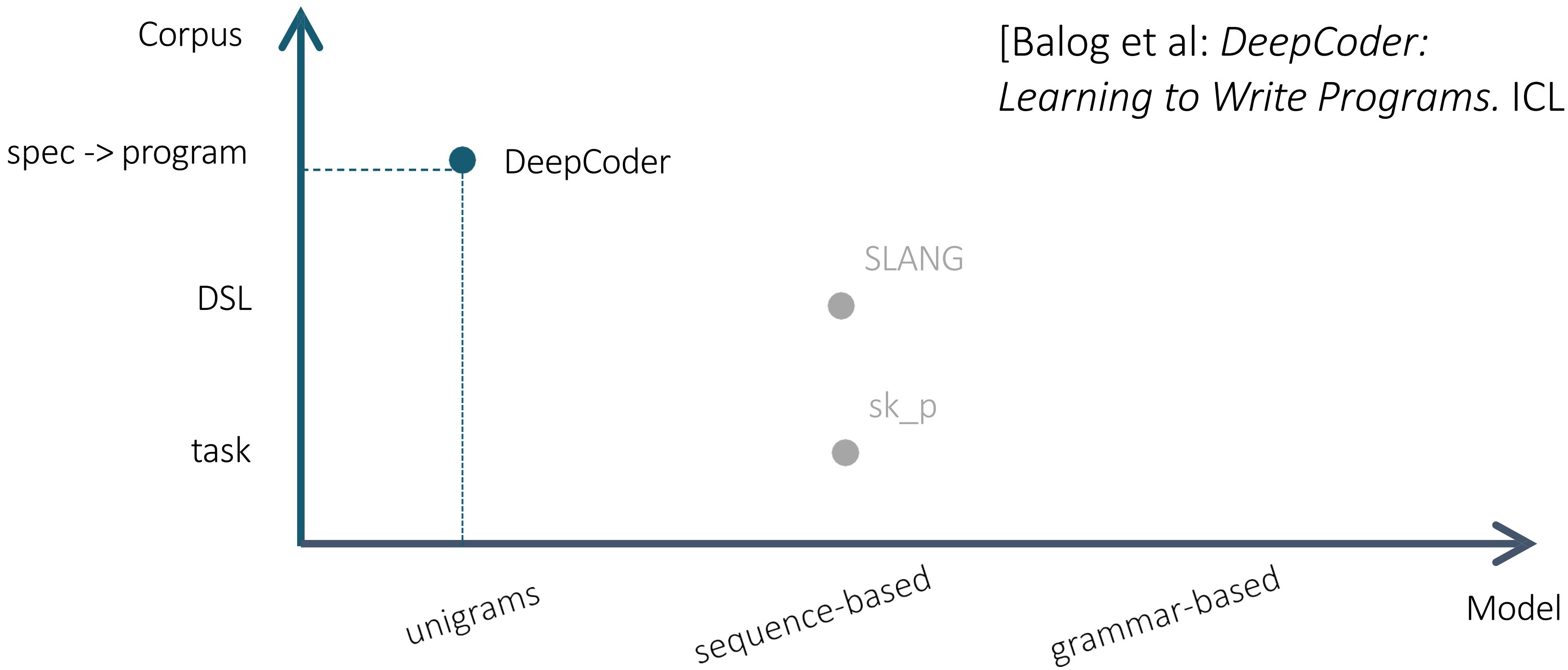
```
while x3 < len ( x0 ) :
```





Statistical Models in Synthesis

[Balog et al: *DeepCoder: Learning to Write Programs.* ICLR’17]



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

↓ existing search technique + sort-and-add

Output: Program in a list DSL

DeepCoder

Predicts likely components from IO examples

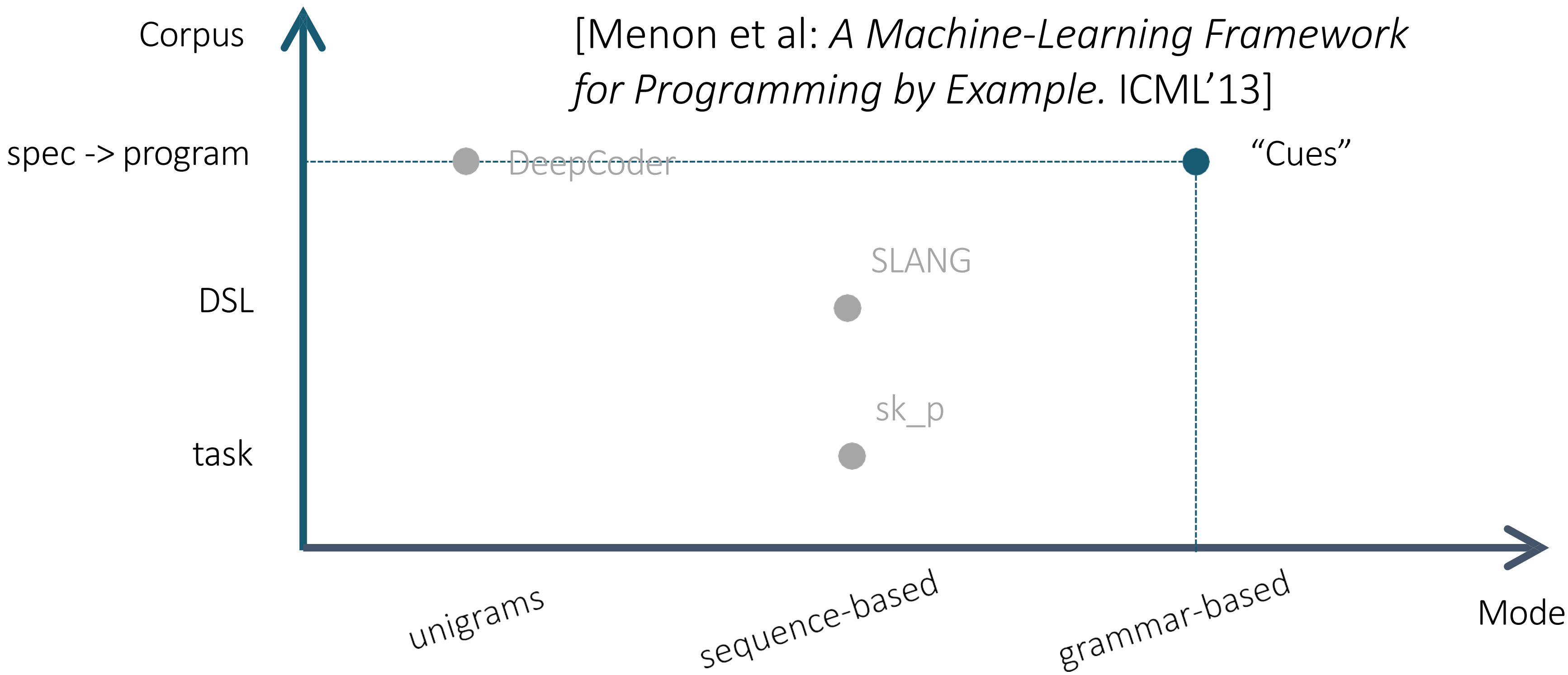
Features

- Can be easily combined with any enumerative search
- Significant speedups for a small list DSL

Limitations

- Unclear whether it scales to larger DSLs or more complex data structures

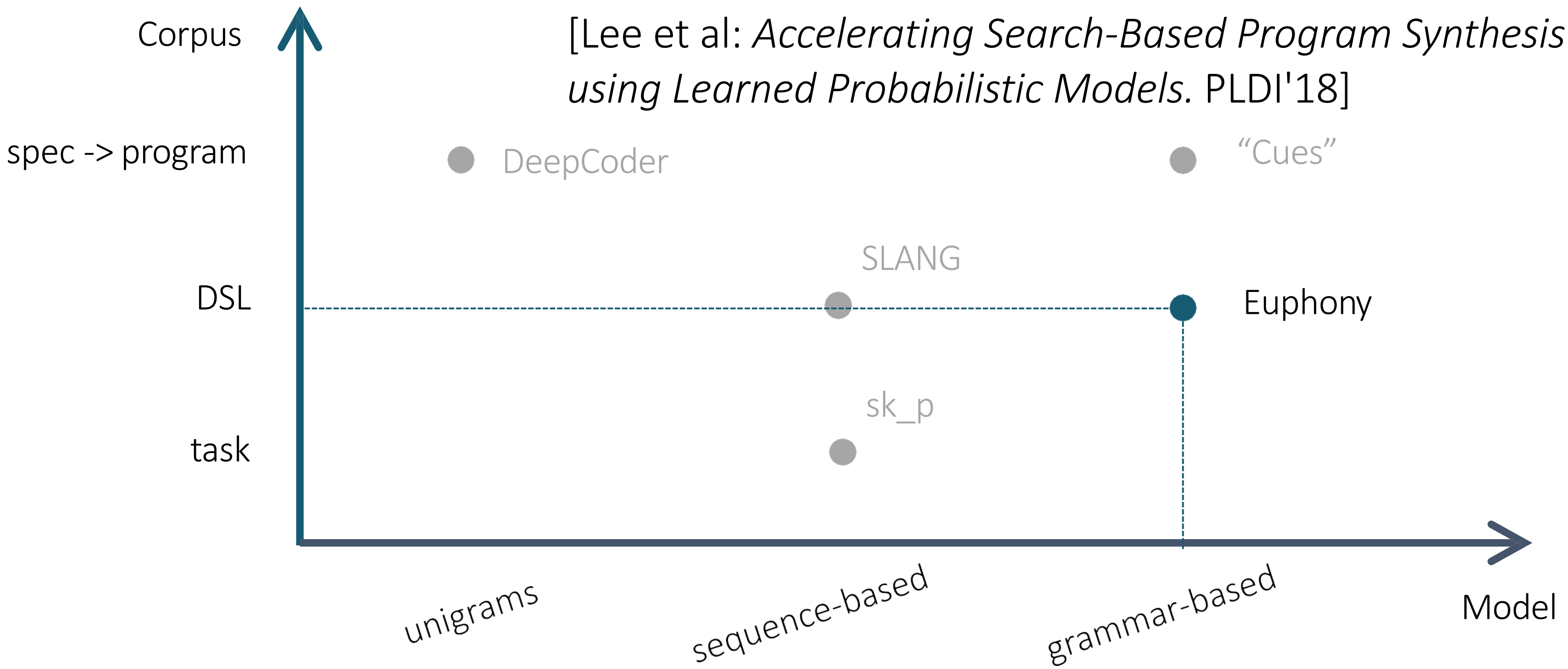
Statistical Models in Synthesis



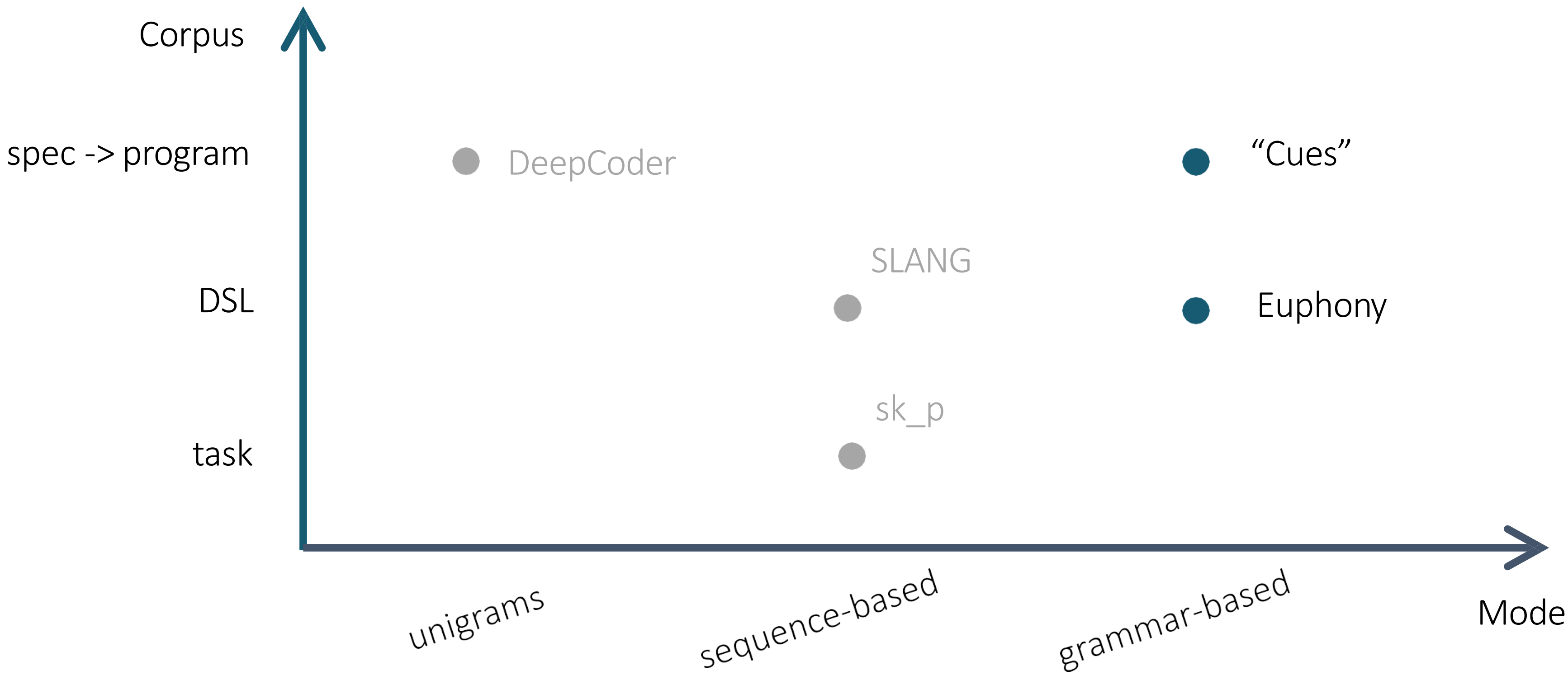
[Menon et al: *A Machine-Learning Framework for Programming by Example.* ICML'13]

Statistical Models in Synthesis

[Lee et al: *Accelerating Search-Based Program Synthesis using Learned Probabilistic Models.* PLDI'18]



Statistical Models in Synthesis



Grammar-based models

Weighted top-down search

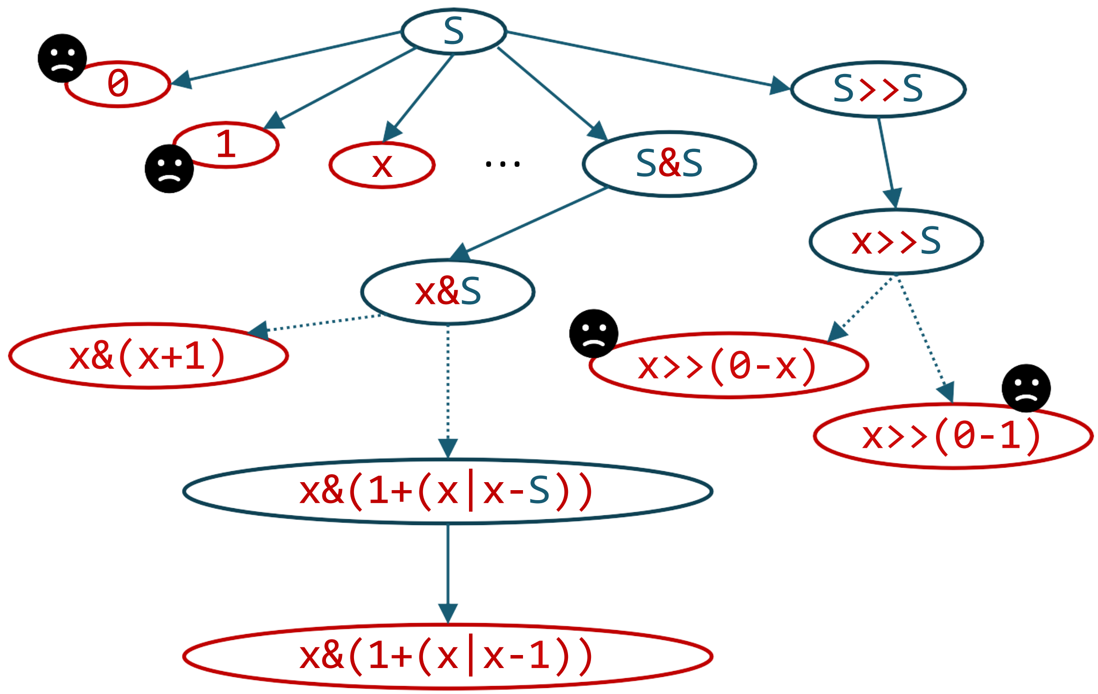
From probabilistic grammars to weights

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!

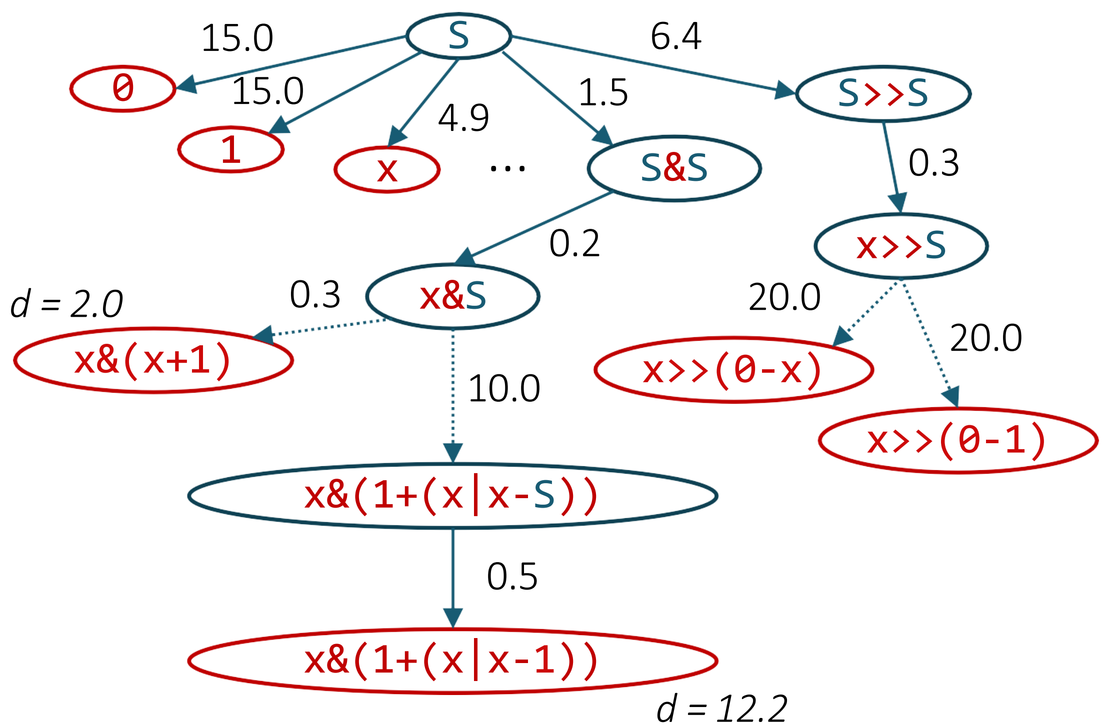
Weighted top-down search

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'*

$$d(p) = \Sigma_{e \in S \to p} w(e)$$

2. Use Dijkstra's algorithm to find closest leaves



Weighted top-down search (Dijkstra)

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

unroll(p,d) {
  P’ := []
  N := leftmost nonterminal in p
  forall (N ::= rhs in R)
    P’ += <p[N -> rhs], d + w(rhs, p)>
  return P’;
}
```

P now stores candidates (nodes) together with their distances

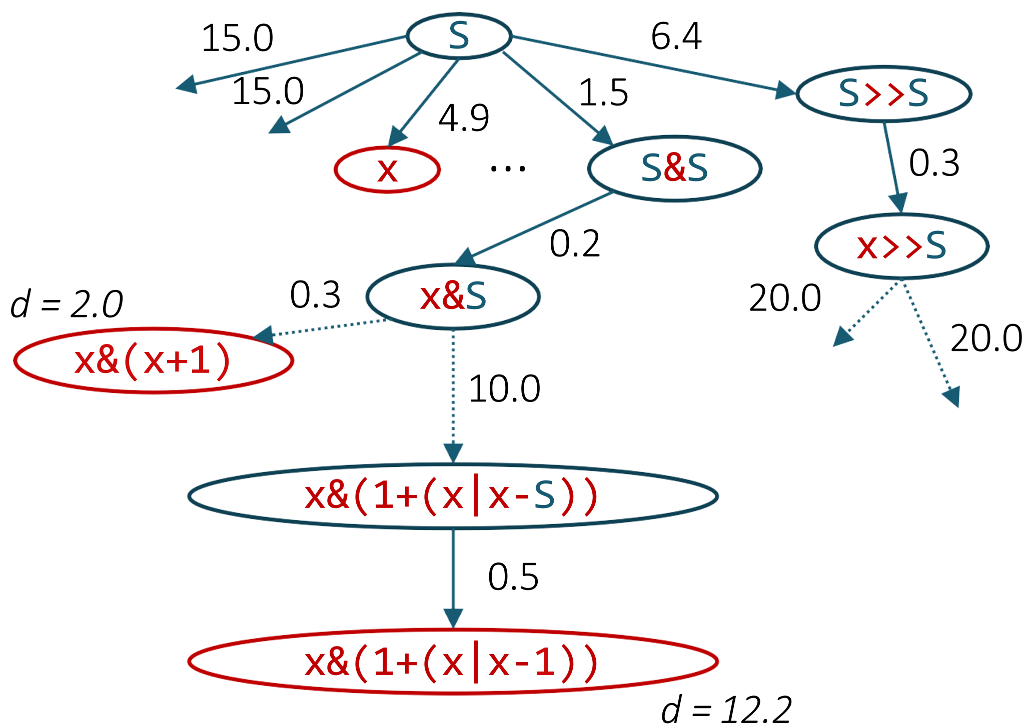
Dequeue the node with the shortest distance from the root

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

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*)

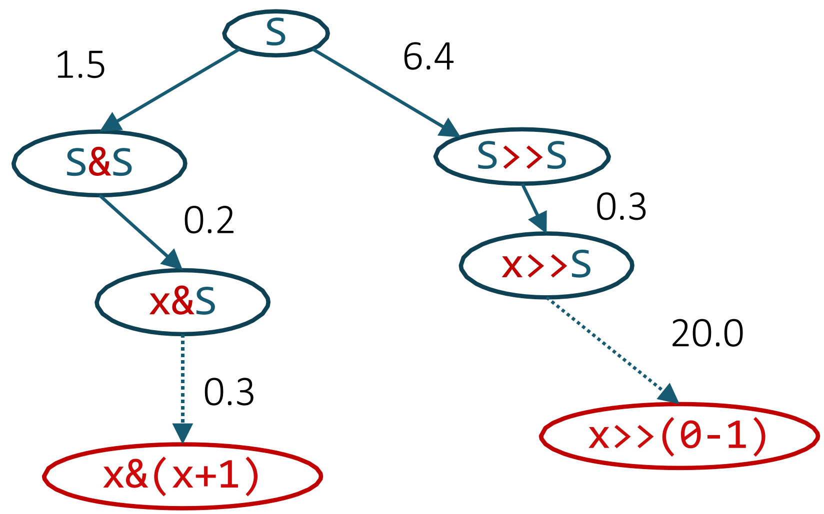
```
top-down(<T, N, R, S>, [i →
  P := [<S,0,h(S)>]          o]) {
  while (P != [])
    <p,d,h> := P.dequeue_min(d + h);
    if (ground(p) && p([i]) = [o])
      return p;
    P.enqueue(unroll(p,d));
}

unroll(p,d) {
  P’ := []
  N := leftmost nonterminal in p
  forall (N ::= rhs in R)
    P’ += <p[N -> rhs], d + w(rhs, p),
                        h(p[N -> rhs])>
  return P’;
}
```

Roughly how close is this program to the closest leaf

So, where do these come from?

Assigning weights to edges



min $d(p)$ = min $\Sigma_{e\in S\to p}\, w(e)$

max $2^{-d(p)}$ = max $\Pi_{e\in S\to p}\, 2^{-w(e)}$

set $w(e) = -\log_2 \wp(e)$ where $\wp(e)$ denotes the probability of taking each edge e.

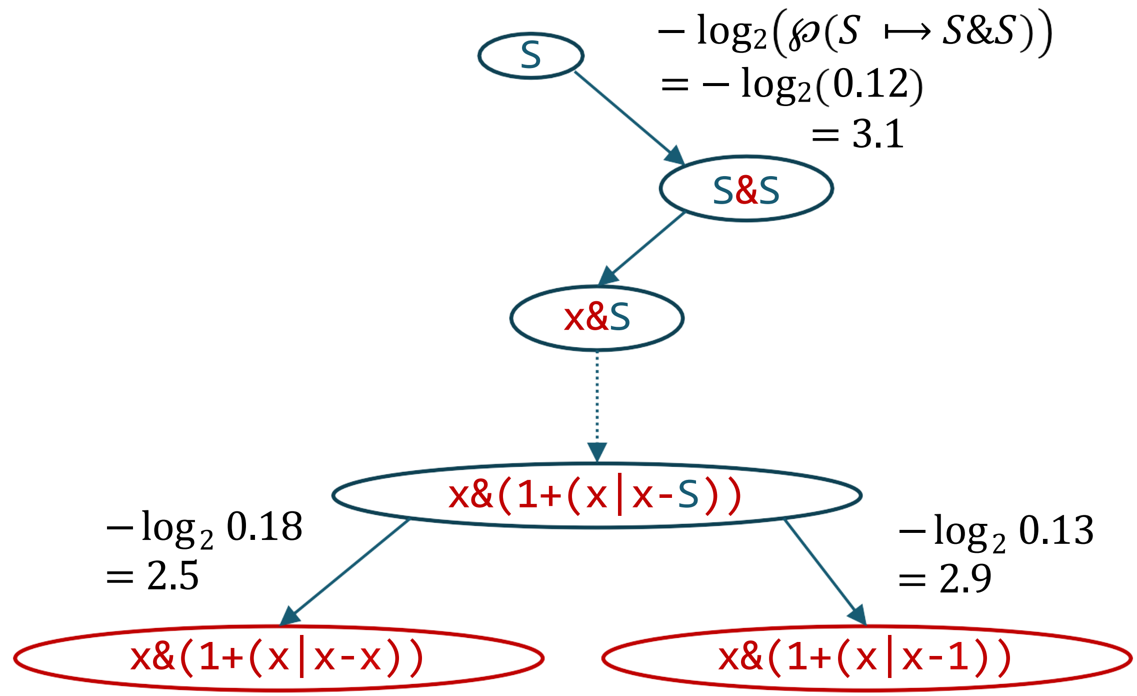
max $\Pi_{e\in S\to P}\, 2^{\log_2 \wp(e)}$

max $\Pi_{e\in S\to p}\, \wp(e)$

So, we should decide what is the probability of taking each edge $\wp(e)$ and then set $w(e)$ = − log2 $\wp(e)$

Probabilistic CFG (PCFG)

	$\wp$
`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

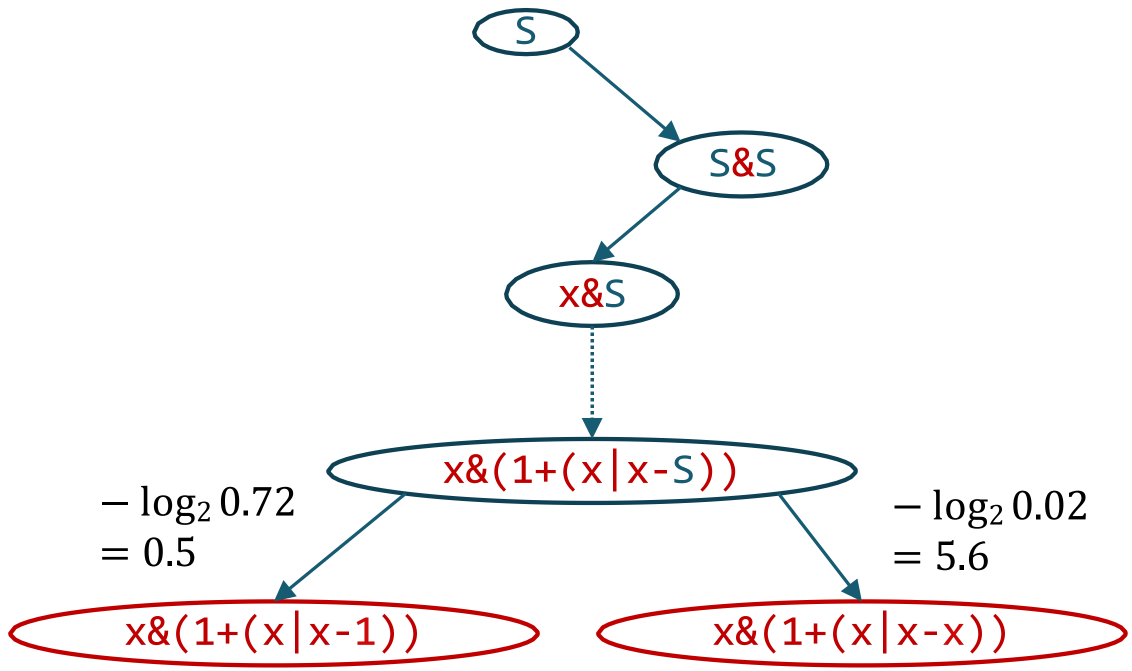


Probabilistic Higher-Order Grammar (PHOG)

[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



Learning PHOGs

[Bielik, Raychev, Vechev ‘16]

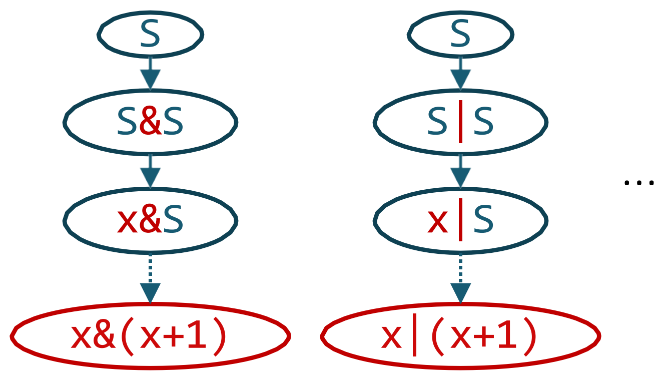
CFG +

Corpus

```
x&(x+1)
x|(x-1)
x
x&(x+x)
x&(1+(x|x-1))
...
```

parse

ASTs / Paths



learn

`context`, $\wp$

PHOGs useful for:

- code completion
- deobfuscation
- programming language translation
- statistical bug detection

How do they compare?