Recent progress in Neural Code Generation

Aman @ Yiming Yang's Lab Seminar, 2/22/22

Code generation

```
def get_dfdx(func, x) -> float:
    ### derivative of func at x
    d = 1e-6
    return (func(x + d) - func(x - d)) / (2 * d)
```

Completing code

```
"""
write a program to search google for a given question
"""
def query_google(query):
    import requests
    from bs4 import BeautifulSoup
    url = "https://www.google.com/search?q=" + query
    r = requests.get(url)
    soup = BeautifulSoup(r.text, "html.parser")
    links = [a.attrs.get("href") for a in soup.select("h3.r a")]
    return links
```

Generating code from
natural language description

What makes code generation interesting?

- Programming ***language*** is language
    - Sequence of tokens sampled from an underlying grammar (so does English)
    - From a modeling perspective $p(code \mid input)$ can be factorized auto regressively, no difference on the surface
- Yet
    - Syntax is much more restrictive (indentation, types)
    - Long-form generations needed
    - Specialized knowledge
    - A single token has higher odds of disrupting everything
- Potential to be the first real-world, pervasive application of language generation models in the next few years

FORTUNE RANKINGS MAGAZINE NEWSLETTERS PODCASTS COVID-19 MORE SEARCH SIGN IN Subscribe Now

Most Popular

Elon Musk laughed at the idea that Tesla's German Gigafactory would use too much water. Now it's a main reason why the plant isn't open

PAID CONTENT
This is how A.I. shapes the future of data automation
FROM BASWARE

Why Putin's Russia is so interested in Ukraine's Donetsk and Luhansk regions

Goldman Sachs lays out a worst-case scenario for markets if Russia-Ukraine conflict escalates

NEWSLETTERS • EYE ON A.I.

Learning to code will not save your kids

BY JEREMY KAHN
February 8, 2022 11:19 AM EST

Watch out Developers: DeepMind AI Can Now Write Code as well as the Average Programmer

Programming jobs may be on the decline in the not-so-distant future.

Trending Technology Security Business Finance Education Home & Office More

MUST READ: Here's how to secure your home network

Bad news for developers? This AI is getting very good at writing code

DeepMind says its research could eventually help programmers code more efficiently and open up the field to people who don't code.

Outline

- **Part 1**
  - State of the art language models are impressive to the point that telling them apart from human text is difficult. **Does this progress translate to code-generation?**
  - Alphacode: SOTA in solving competitive programming problems (“better” than 54% humans)
  - Treat code like other language
- **Part 2**
  - **Can we exploit key properties of code to improve over using language models alone?**
  - Using familiar techniques from NLP toolbox and PL
    - Masked language objective when dealing with code
    - Retrieval-augmented generation
    - Backtranslation
    - Syntax-guided generation

DeepMind

2022-2-2

Competition-Level Code Generation with AlphaCode

Yujia Li*, David Choi*, Junyoung Chung*, Nate Kushman*, Julian Schrittwieser*, Rémi Leblond*, Tom Eccles*, James Keeling*, Felix Gimeno*, Agustin Dal Lago*, Thomas Hubert*, Peter Choy*, Cyprien de Masson d'Autume*, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu and Oriol Vinyals

*Joint first authors

Competitive programming

- Given
  - A new problem
  - Few (~5) input - output test cases
- Generate 1 million programs, test them on the 5 test cases
  - Only 10k programs pass these 5 test cases
- Check solution on a larger number of hidden test cases
- Widely used in hiring for CS

A. Team

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

One day three best friends Petya, Vasya and Tonya decided to form a team and take part in programming contests. Participants are usually offered several problems during programming contests. Long before the start the friends decided that they will implement a problem if at least two of them are sure about the solution. Otherwise, the friends won't write the problem's solution.

This contest offers n problems to the participants. For each problem we know, which friend is sure about the solution. Help the friends find the number of problems for which they will write a solution.

Input
The first input line contains a single integer n $(1 \le n \le 1000)$ — the number of problems in the contest. Then n lines contain three integers each, each integer is either 0 or 1. If the first number in the line equals 1, then Petya is sure about the problem's solution, otherwise he isn't sure. The second number shows Vasya's view on the solution, the third number shows Tonya's view. The numbers on the lines are separated by spaces.

Output
Print a single integer — the number of problems the friends will implement on the contest.

Examples

input Copy
```
3
1 1 0
1 1 1
1 0 0
```
output Copy
```
2
```

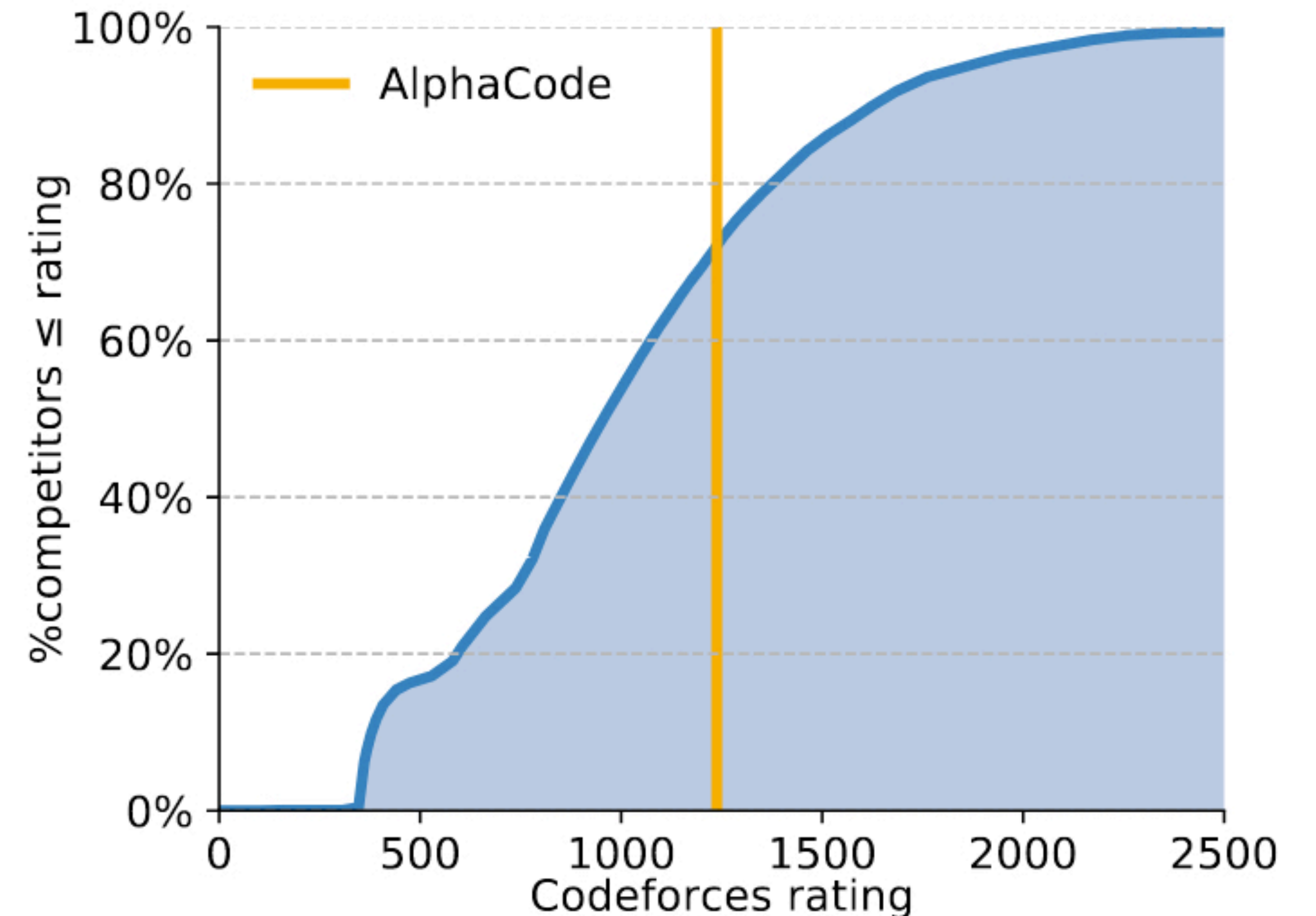
input Copy
```
2
1 0 0
0 1 1
```
output Copy
```
1
```

Taken from https://codeforces.com/

Alphacode

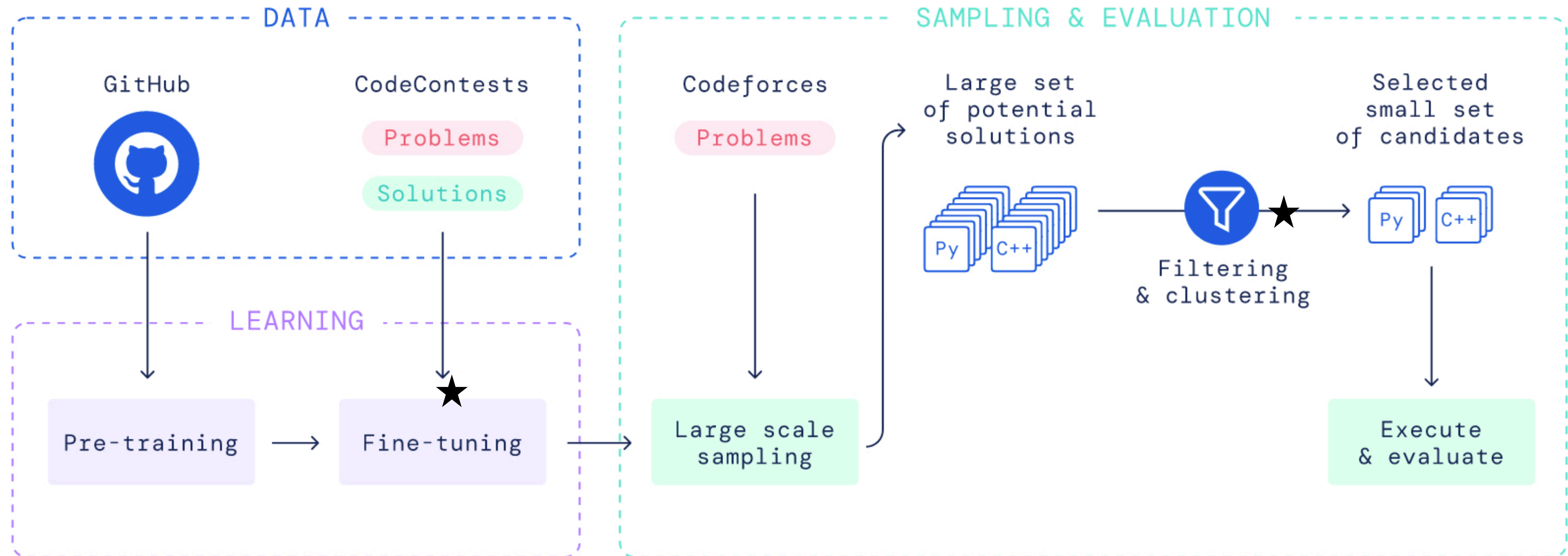
Introduction

- Competitive programming contests: write program for a complex requirement, test on O(10^4) test cases.
- Example
  - Given two strings **A** and **B,** find if **A** and **B** are palindromes
- Many competitive programming websites



Alphacode is better than 72% of human participants at codeforces.com

Overview



Datasets

- Pre-training: essentially all publicly available code on Github, filter out files > 1000 characters ~ 715.1 GB
- Fine-tuning:
  - Scraped codeforces.com
  - Need dataset of (problem, correct solutions)
  - Generated additional test cases by mutating, and checked for correctness
  - Do not care about time complexity
- Architecture: encoder-decoder
  - Number of parameters vary from 1B to 41B
  - Shallow (8 layers) + wide (1500 tokens) encoder —> deep (56 layers) + narrow (768 tokens) decoder
  - Code is shorter than program description and test cases

Fine-tuning

Tags (meta-data about the problem)

Encoder Input X:

```
# RATING: 3100
# TAGS: binary search,math
# LANGUAGE IS python3
# CORRECT SOLUTION
# n towns are arranged in a circle sequentially. The towns are numbered from 1
# to n in clockwise order. In the i-th town, there lives a singer with a
# repertoire of a_i minutes for each i ∈ [1, n].
#
# Each singer visited all n towns in clockwise order, starting with the town he
# lives in, and gave exactly one concert in each town. In addition, in each
# town, the i-th singer got inspired and came up with a song that lasts a_i
# minutes. The song was added to his repertoire so that he could perform it in
# the rest of the cities.
#
# Hence, for the i-th singer, the concert in the i-th town will last a_i
# minutes, in the (i + 1)-th town the concert will last 2 · a_i minutes, ..., in
# the ((i + k) mod n + 1)-th town the duration of the concert will be (k + 2) ·
# a_i, ..., in the town ((i + n - 2) mod n + 1) - n · a_i minutes.
#
# You are given an array of b integer numbers, where b_i is the total duration
# of concerts in the i-th town. Reconstruct any correct sequence of positive
# integers a or say that it is impossible.
#
# Input
#
# The first line contains one integer t (1 ≤ t ≤ 10^3) - the number of test
# cases. Then the test cases follow.
#
# Each test case consists of two lines. The first line contains a single integer
# n (1 ≤ n ≤ 4 · 10^4) - the number of cities. The second line contains n
# integers b_1, b_2, ..., b_n (1 ≤ b_i ≤ 10^{9}) - the total duration of
# concerts in i-th city.
#
# The sum of n over all test cases does not exceed 2 · 10^5.
#
# Output
#
# For each test case, print the answer as follows:
#
# If there is no suitable sequence a, print NO. Otherwise, on the first line
# print YES, on the next line print the sequence a_1, a_2, ..., a_n of n
# integers, where a_i (1 ≤ a_i ≤ 10^{9}) is the initial duration of repertoire
# of the i-th singer. If there are multiple answers, print any of them.
#
# Example
#
# Input
#
#
# 4
# 3
# 12 16 14
# 1
# 1
# 3
```

Decoder Output Y:

```python
import sys
input = sys.stdin.readline

t=int(input())
for tests in range(t):
    n=int(input())
    B=list(map(int,input().split()))

    A=[-1<<100]*n
    MIN=float("inf")
    MAX=-1

    for i,b in enumerate(B):
        MIN=min(MIN,b)
        MAX=max(MAX,b)

    SUM=sum(B)

    if SUM%(n*(n+1)//2)!=0:
        print("NO")
        continue

    SUM//=(n*(n+1)//2)

    for i in range(n):
        j=i-1
        if j==-1:
            j=n-1
        a=(SUM-B[i]+B[j])//n
        if a>MIN or a>MAX or a<1:
            print("NO")
            break
        A[i]=a
    else:
        print("YES")
        print(*A)
```

Fine-tuning

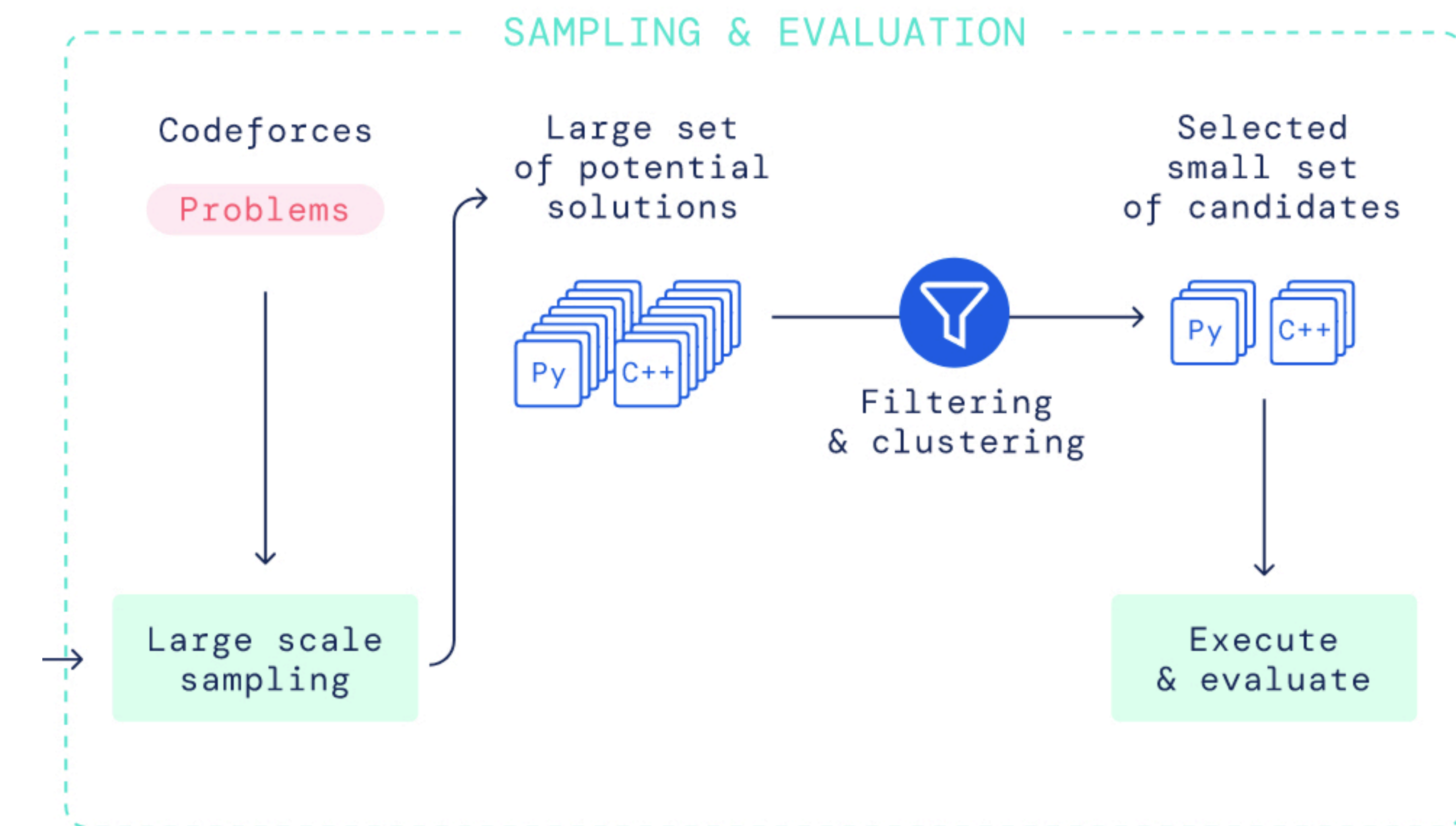
Tricks

- Tempering to avoid overfitting:
    - During training, divide the logits by T = 0.2 (<1)
    - Causes sharper logits during training, but smoother logits during inference
    - No explanation provided, but intuitively could be working as backprop penalizes overly confident predictions
- GOLD training objective to improve precision (added to the standard MLE gradient)
    - The gradient for a token is up-weighted if it is already being predicted with a high confidence

$$\nabla \mathcal{L}_{\text{GOLD}}(\theta) = - \sum_{s \in \text{Solution tokens}} P_\theta(s) \nabla \log P_\theta(s)$$

Sampling and Filtering

- Sample **1M** programs for a given problem, select **10** programs from this pool
  - Done by randomly varying tags in the input + changing the input temperature
- If one of the 10 solves the problem, consider it solved
- **Step 1:** Filter obviously wrong programs by using unit tests
  (~99% discarded)
  - Left with 10k
- **Step 2:** Cluster programs using artificially generated test cases (key insight)



Filtering by clustering

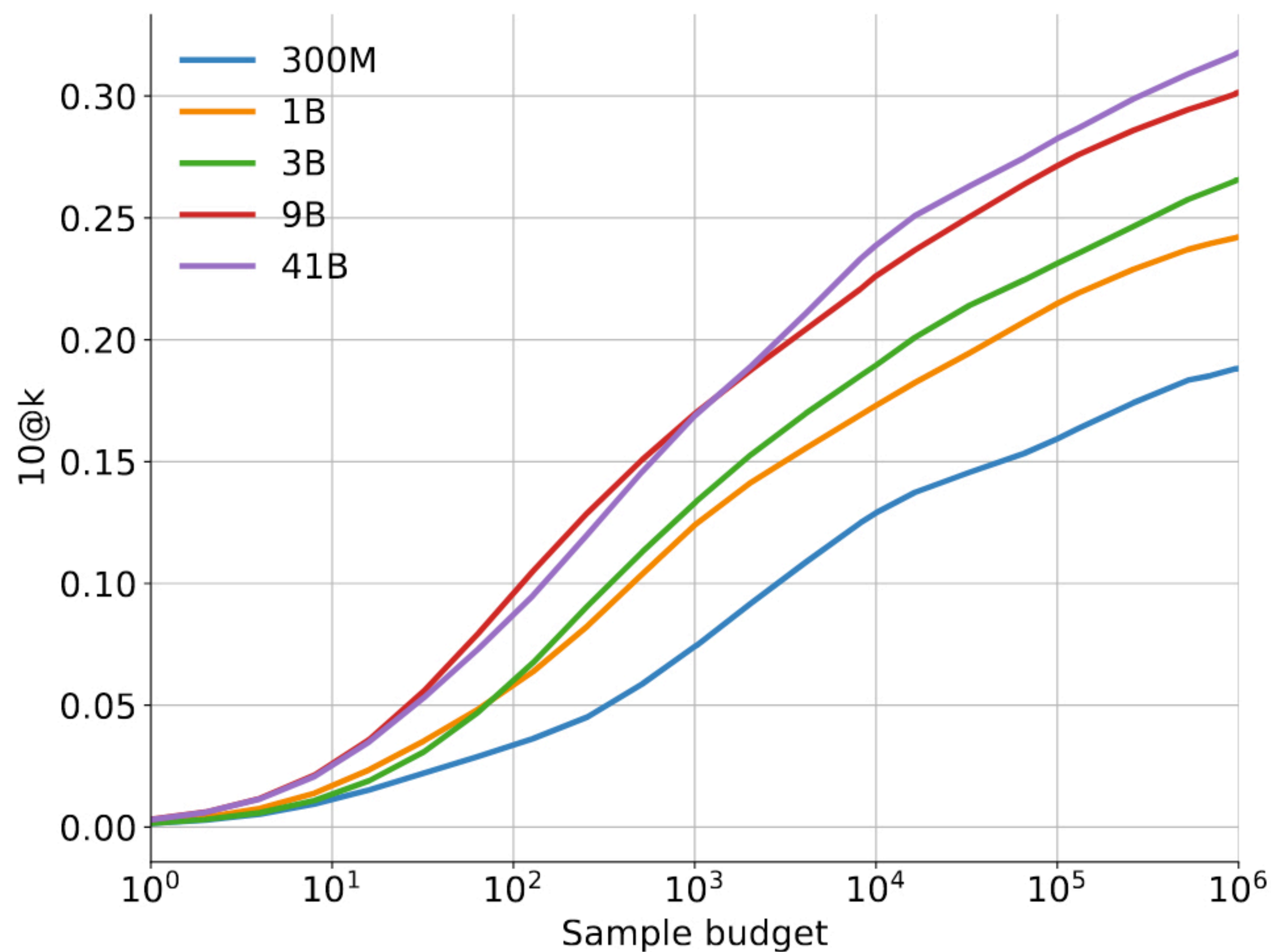
- Let $\{p_1, p_2, \ldots, p_n\}$ be a set of programs, $\{x_1, x_2, \ldots, x_m\}$ be a set of inputs
- If programs p_i and p_j have the same output for the given input set, they belong to the same cluster
  - We don't have to care about correctness
  - Programs that generate same output for the same input must be semantically the same
- Sample one program per cluster

Main results

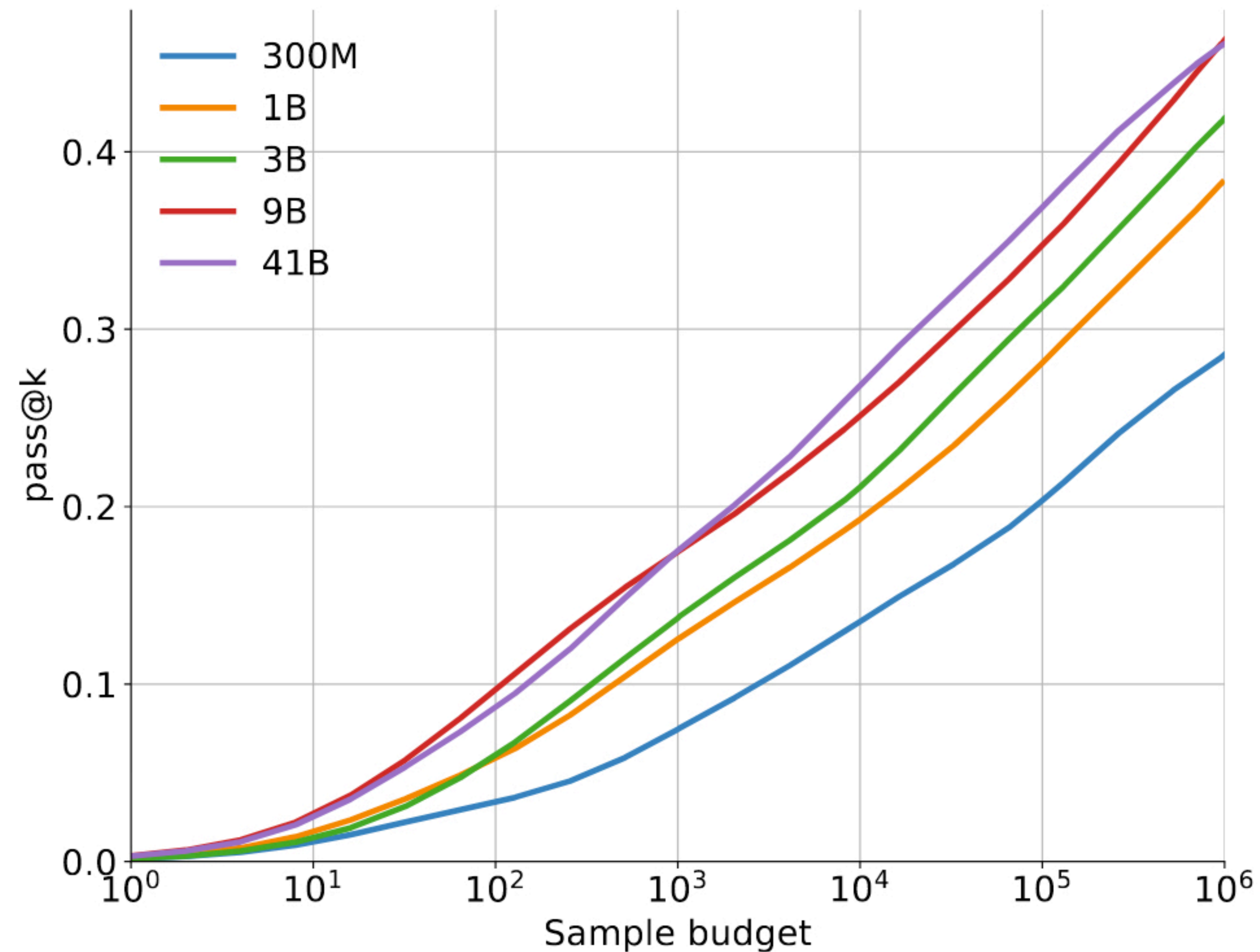
https://alphacode.deepmind.com/

Approach	Validation Set				Test Set		
	10@1k	10@10k	10@100k	10@1M	10@1k	10@10k	10@100k
9B	16.9%	22.6%	27.1%	30.1%	14.3%	21.5%	25.8%
41B	16.9%	23.9%	28.2%	31.8%	15.6%	23.2%	27.7%
41B + clustering	21.0%	26.2%	31.8%	34.2%	16.4%	25.4%	29.6%

Contest ID	1591	1608	1613	1615	1617	1618	1619	1620	1622	1623	Average
Best	43.5%	43.6%	59.8%	60.5%	65.1%	32.2%	47.1%	54.0%	57.5%	20.6%	**48.4%**
Estimated	44.3%	46.3%	66.1%	62.4%	73.9%	52.2%	47.3%	63.3%	66.2%	20.9%	**54.3%**
Worst	74.5%	95.7%	75.0%	90.4%	82.3%	53.5%	88.1%	75.1%	81.6%	55.3%	**77.2%**



(a) 10 attempts per problem



(b) Unlimited attempts per problem

Figure 6 | **Solve rate scaling vs. number of samples**. The solve rate scales approximately log-linearly with the number of samples, although this tapers off slightly in the **10@k** setting. The better, larger scale models have higher scaling slopes in this log-linear plot.

Similar works

Evaluating Large Language Models Trained on Code

Mark Chen [*1] Jerry Tworek [*1] Heewoo Jun [*1] Qiming Yuan [*1] Henrique Ponde de Oliveira Pinto [*1]
Jared Kaplan [*2] Harri Edwards [1] Yuri Burda [1] Nicholas Joseph [2] Greg Brockman [1] Alex Ray [1] Raul Puri [1]
Gretchen Krueger [1] Michael Petrov [1] Heidy Khlaaf [3] Girish Sastry [1] Pamela Mishkin [1] Brooke Chan [1]
Scott Gray [1] Nick Ryder [1] Mikhail Pavlov [1] Alethea Power [1] Lukasz Kaiser [1] Mohammad Bavarian [1]
Clemens Winter [1] Philippe Tillet [1] Felipe Petroski Such [1] Dave Cummings [1] Matthias Plappert [1]
Fotios Chantzis [1] Elizabeth Barnes [1] Ariel Herbert-Voss [1] William Hebgen Guss [1] Alex Nichol [1] Alex Paino [1]
Nikolas Tezak [1] Jie Tang [1] Igor Babuschkin [1] Suchir Balaji [1] Shantanu Jain [1] William Saunders [1]
Christopher Hesse [1] Andrew N. Carr [1] Jan Leike [1] Josh Achiam [1] Vedant Misra [1] Evan Morikawa [1]
Alec Radford [1] Matthew Knight [1] Miles Brundage [1] Mira Murati [1] Katie Mayer [1] Peter Welinder [1]
Bob McGrew [1] Dario Amodei [2] Sam McCandlish [2] Ilya Sutskever [1] Wojciech Zaremba [1]

Model behind the outputs we saw on the first page

Program Synthesis with Large Language Models

Jacob Austin[*] Augustus Odena[*]

Maxwell Nye[†] Maarten Bosma Henryk Michalewski David Dohan Ellen Jiang Carrie Cai

Michael Terry Quoc Le Charles Sutton

Google Research

Outline

- **Part 1** ✓
  - State of the art language models are impressive to the point that telling them apart from human text is difficult. Does this progress translate to code-generation?
  - Alphacode: SOTA in solving competitive programming problems (“better” than 54% humans)
  - Treat code like other language
- **Part 2**
  - **Can we exploit key properties of code to improve over using language models alone?**
  - Using familiar techniques from NLP toolbox and PL
    - Masked language objective when dealing with code
    - Retrieval-augmented generation
    - Backtranslation
    - Syntax-guided generation

DOBF: A Deobfuscation Pre-Training Objective for Programming Languages

Baptiste Roziere*
Facebook AI Research
Paris-Dauphine University
broz@fb.com

Marie-Anne Lachaux*
Facebook AI Research
malachaux@fb.com

Marc Szafraniec
Facebook AI Research
szafraniec@fb.com

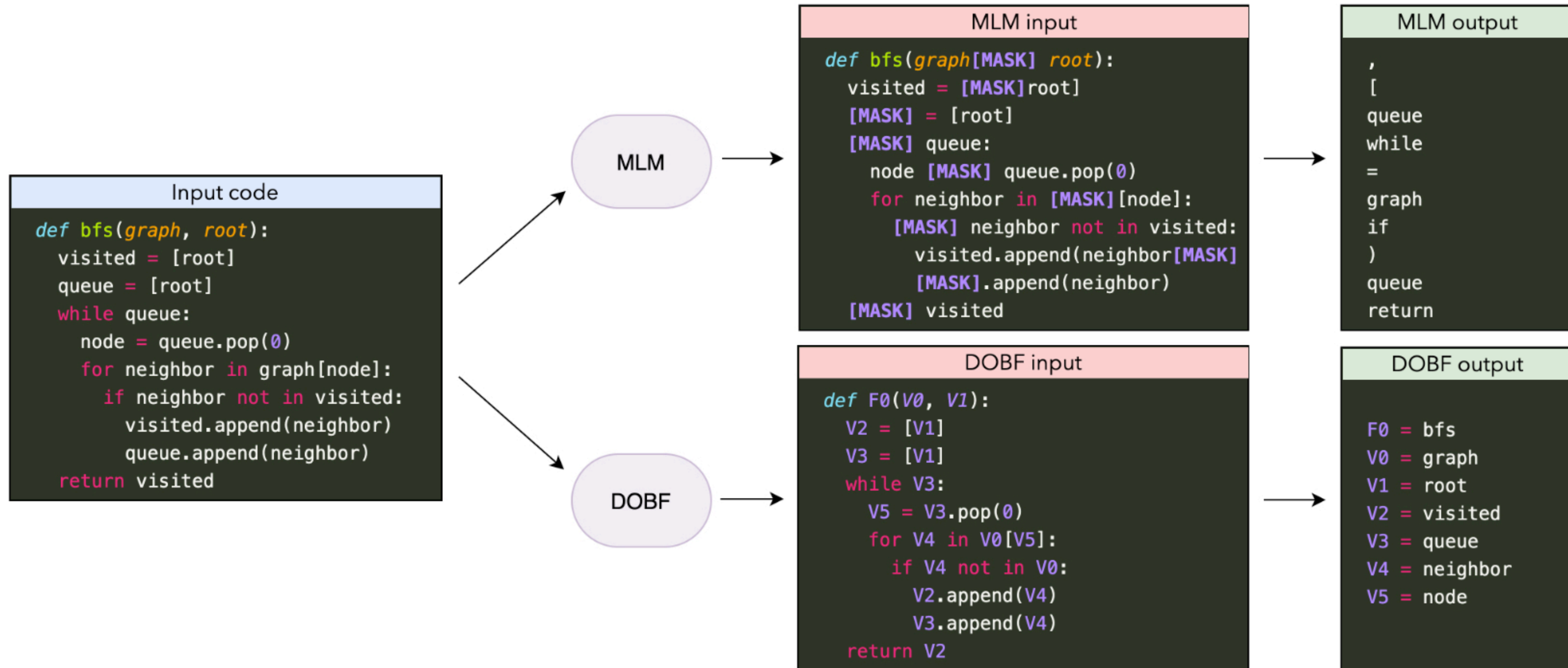
Guillaume Lample
Facebook AI Research
glample@fb.com

Neurips 2021

Key idea

- Masked language modeling (MLM) for language drops tokens randomly
- Too simple for programming languages, not very informative
  - Syntax errors (e.g., missing “;”) might be easily corrected by the code
  - Masking a variable once, but not everywhere, allows copying names
- Instead of MLM, they propose DOBF (de-obfuscation objective) that leverages structure of programming languages

De-obfuscation



- Replace class, function, and variable names with special tokens, and train a model to recover them.
- Syntax related tokens are not masked out
- Model has to come up with meaningful variable and function names, which requires deep understanding of code semantics.

Implementation

- Encoder-decoder transformer based Seq2seq model with the new DOBF objective (12 layers, 12 heads)
- Dataset: collect all publicly available code on Github, retain files with < 2000 tokens

	Eval $p_{\text{obf}} = 0$		Eval $p_{\text{obf}} = 1$	
	Acc	F1	Acc	F1
DOBF_0	56.3	68.0	0.4	0.9
$\text{DOBF}_{0.5}$	61.1	71.2	41.8	54.8
DOBF_1	18.1	27.0	45.6	58.1
$\text{DOBF}_{0.5}$ init MLM	**67.6**	**76.3**	45.7	58.0
DOBF_1 init MLM	20.0	28.3	**49.7**	**61.1**

Initializing with MLM helps further

Results

	Clone Det (F1 score)	Code Sum Java (BLEU)	Code Sum Python (BLEU)	NLCS (MRR)	Python→Java (CA@1)		Java→Python (CA@1)	
					k=1	k=10	k=1	k=10
Transformer	88.14	16.58	16.43	0.025	24.0	28.4	29.0	29.7
MLM	91.89	18.59	17.95	0.308	44.8	45.4	34.5	35.6
DAE	96.30	19.19	18.28	0.380	**48.3**	**49.2**	32.1	32.8
CodeBERT	96.50	18.25	18.22	0.315	40.8	45.6	36.5	36.7
GraphCodeBERT	96.38	18.78	18.51	0.377	44.3	44.1	35.6	37.8
DOBF init scratch	**96.52**	18.19	17.51	0.272	43.9	44.1	35.2	34.7
DOBF	95.87	19.05	18.24	0.383	43.5	44.1	38.7	40.0
DOBF+DAE	95.82	**19.36**	**18.58**	**0.397**	46.6	47.3	**40.6**	**42.4**

Qualitative Results

Completing matrix operations code

Input Code	Function Name Proposals	

`def FUNC_0 (m1, m2):`
`  assert m1.shape == m2.shape`
`  n, m = m1.shape`
`  res = [[0 for _ in range(m)] for _ in range(n)]`
`  for i in range(n):`
`    for j in range(m):`
`      res[i][j] = m1[i][j] + m2[i][j]`
`  return res`
	matrix_add
	matrixAdd
	matrixadd
	matrix_sum
	matrix_addition
		25.9%
		22.5%
		18.8%
		16.7%
		16.1%

`def FUNC_0 (matrix):`
`  n, _ = matrix.shape`
`  for i in range(n):`
`    for j in range(i,n):`
`      matrix[i][j], matrix[j][i] = \`
`        matrix[j][i], matrix[i][j]`
	transpose
	rotate
	rotate_matrix
	symmetric
	rotate_matrix_by_row
		36.7%
		29.5%
		17.1%
		8.9%
		7.7%

`def FUNC_0 (m1, m2):`
`  n1, m1 = m1.shape`
`  n2, m2 = m2.shape`
`  assert n2 == m1`
`  res = [[0 for _ in range(m2)] for _ in range(n1)]`
`  for i in range(n1):`
`    for j in range(m2):`
`      res[i][j] = sum([m1[i][k] * m2[k][j]`
`                       for k in range(n2)])`
`  return res`
	matrix_product
	mat_mult
	matmul_mat
	matprod
	matrixProduct
		28.8%
		23.8%
		17.0%
		16.0%
		14.4%

Neural Program Generation Modulo Static Analysis

Rohan Mukherjee
Rice University

Yeming Wen
UT Austin

Dipak Chaudhari
UT Austin

Thomas W. Reps
University of Wisconsin

Swarat Chaudhuri
UT Austin

Chris Jermaine
Rice University

Neurips 2021 (Spotlight)

Introduction

- Task: **complete** a given piece of code
- $p_\Theta(Y \mid X)$ where
  - X is the input specification: class name, type of the variables, other complete methods
  - Y is a completion of X
- Key idea:
  - Neurosymbolic attribute grammars: **Use static analysis and grammar to guide code generation**

(a)

```
public class FileUtil{
  String err;
  public int read(File f){...}

  /* write lines to file */
  public void write(
    File f, String str){??}}
```

(b)

```
void write(File f, String str){
  try {
    FileWriter var_0;
    var_0 = new FileWriter(f);
    var_0.write(str);
  } catch(IOException var_0) {
    var_0.printStackTrace();
    System.out.println( ARG ); }
  return; }
```

Model

- Let Z be the (latent) user intent behind the ambiguous incomplete code X
- $p(Y \mid X) = \int_{Z} p(Z \mid X) p(Y \mid Z) dZ$
- $p(Z \mid X)$: context encoder
- $p(Y \mid Z)$: program synthesizer

Context encoder

- X: input or *evidence*
    - Consists of method names, formal parameters, comments (7 types of “evidence”)
- Assume Z is Normal, X is sampled from Z

$$P(\mathsf{X}|\mathsf{Z},\theta) = \left(\prod_j \text{Normal}(f(\mathsf{X}_{Calls,j})|\mathsf{Z}, \mathbf{I}\sigma^2_{Calls})\right)\left(\prod_j \text{Normal}(f(\mathsf{X}_{Types,j})|\mathsf{Z}, \mathbf{I}\sigma^2_{Types})\right)$$
$$\left(\prod_j \text{Normal}(f(\mathsf{X}_{Keys,j})|\mathsf{Z}, \mathbf{I}\sigma^2_{Keys})\right).$$

- And thus:

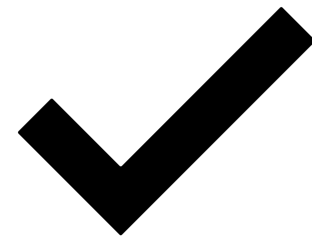
$$\mathcal{P}(\mathsf{Z}|\mathsf{X}) = \mathcal{N}\left(\mathsf{Z} \mid \frac{\sum_{j,k}\sigma_j^{-2} f_j(\mathsf{X}_{j,k})}{1+\sum_j |\mathsf{X}_j|\sigma_j^{-2}}, \frac{1}{1+\sum_j |\mathsf{X}_j|\sigma_j^{-2}}\mathbf{I}\right)$$

    -

- In summary, X (input) is used to generate parameters of normal from which Z is sampled. Each part of the input contributes individually to sampling the latent variable

Program Synthesizer

- $p(Y \mid X) = \int_Z p(Z \mid X) p(Y \mid Z) dZ$
- $p(Z \mid X)$: context encoder
- $p(Y \mid X)$: program synthesizer



Aside

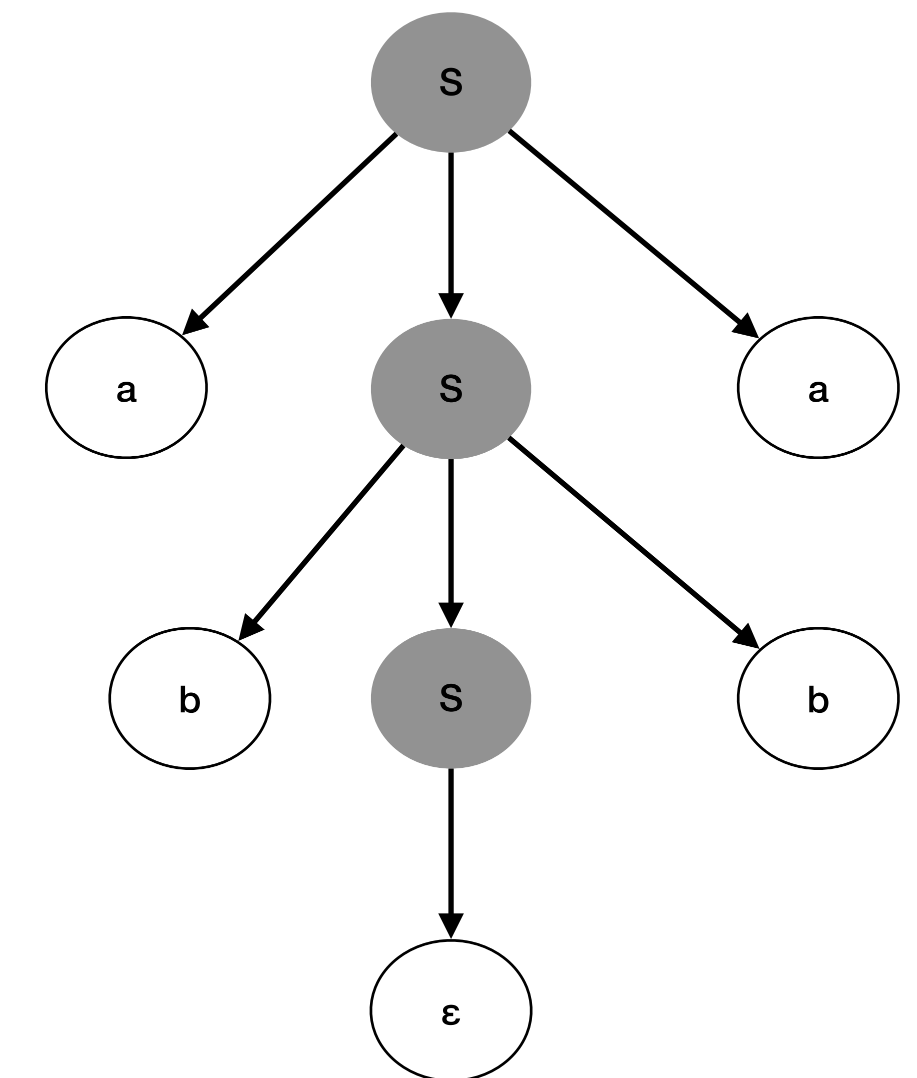
Context-free grammar

- Set of production rules, where left-hand side is a non-terminals, and right-hand side is a combination of terminals and non-terminals

$$S \rightarrow aSa$$
$$S \rightarrow bSb$$
$$S \rightarrow \varepsilon$$

- Generative process:
  - Start with a production, recursively expand non-terminals
    - $S \rightarrow aSa \rightarrow abSba \rightarrow abba$
  - Parse trees
  - If each rule is picked probabilistically, it's called a P-CFG
- **Aside:** if the LHS contains a terminal the grammar is context sensitive

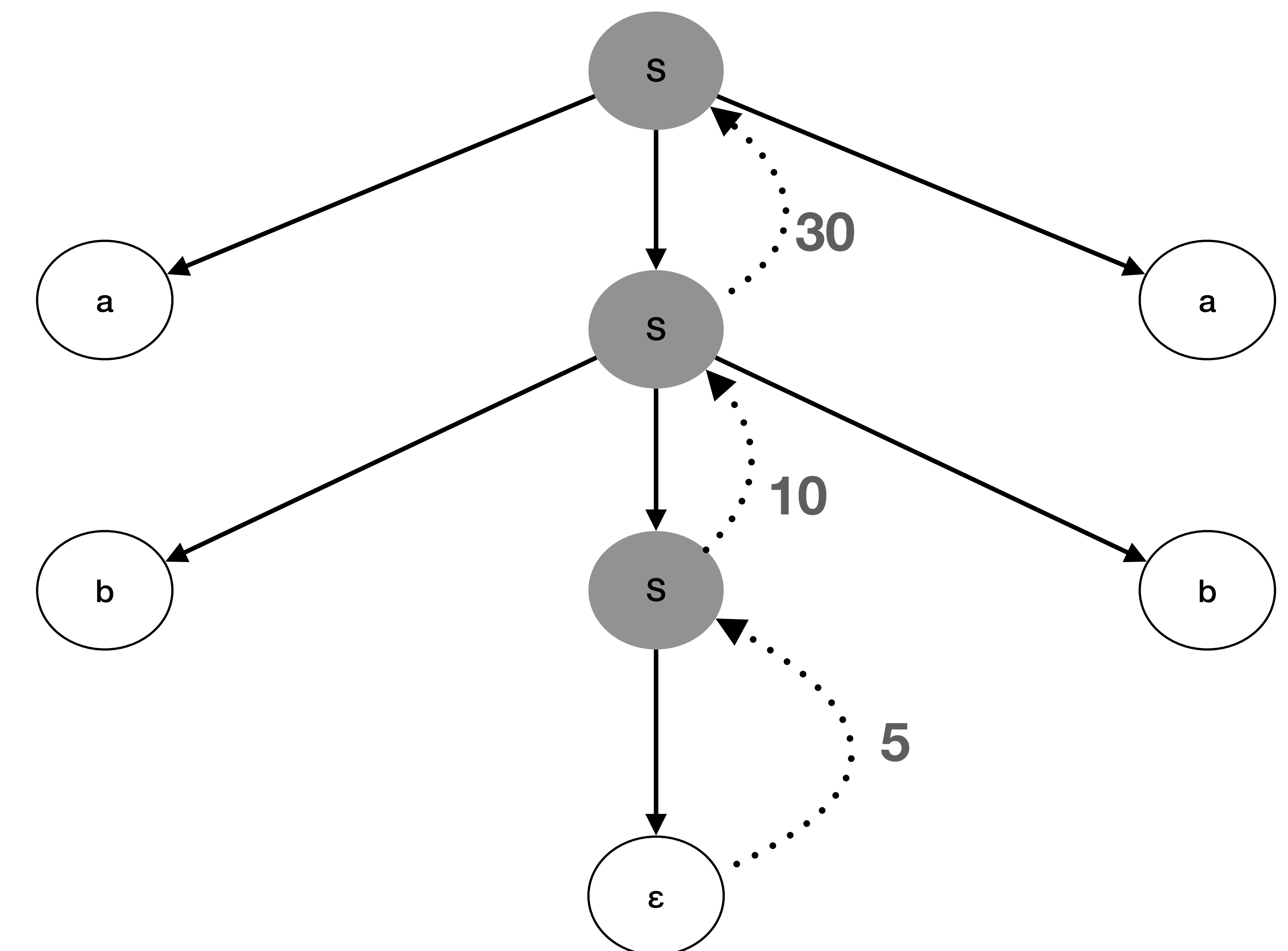
$$S \rightarrow aSa$$
$$S \rightarrow bSb$$
$$S \rightarrow \varepsilon$$
$$\mathrm{ab}Sba \rightarrow abaaba$$



Attribute grammars

- Context-free grammars + attributes associated with each symbol
- Attribute grammar
  - Attaches some auxiliary information with each symbol, and defines how the information flows as parsing proceeds
  - Each symbol gets inherited attributes and synthesized attributes
- Example

$$S1 \rightarrow aS2a\ [S1.value = S2.value * 3]$$
$$S1 \rightarrow bS2b\ [S1.value = S2.value * 2]$$
$$S \rightarrow \varepsilon\ [\ S.value = 5\ ]$$



Program Synthesizer

- Generative process:
  - $Y = (S_1, S_2, \ldots, S_n)$ where each of the S_i is a symbol and its expansion
- **Standard conditional distribution:** PCFG conditioned by the input latent representation Z, and the expansion so far
  - $Y = \prod_i P(S_i \mid S_{<i}, Z)$
- **Their work:**
  - $Y = \prod_i P(S_i \mid S_{<i}, Z, A(S_i))$ where the extra $A(S_i)$ term is supplied by the auxiliary grammar
- **Key idea:** inform the generation process with Auxiliary attributes of the program generated so far using a static analyzer:
  - Static analyzer: can infer the types of the method generated so far

Generation without attribute grammar

```
2   import java.io.*;
3
4   public class FileUtils{
5
6     public int read(File a){
7       Reader a = new FileReader("file.txt");
8       ...
9     }
10
11    /**
12    write lines to file
13    */
14    @synthesize public void write(File fp_0,
15                                  String fp_1){
16
17    }
18  }
```

Input

```
void write(java.io.File fp_0, java.lang.String fp_1){
    try {
            java.io.FileWriter var_0;
            var_0 = new java.io.FileWriter.FileWriter(
                    (java.io.File) fp_0, (boolean) BOOL_LITERAL);
            java.io.BufferedWriter var_1;
            var_1 = new java.io.BufferedWriter.BufferedWriter(
                            (java.io.Writer) var_0);
            var_1.write(java.lang.String: fp_1);
            var_1.newLine();
    }
    catch(java.io.IOException var_0){
        var_0.printStackTrace();
    }
    return;
}
```

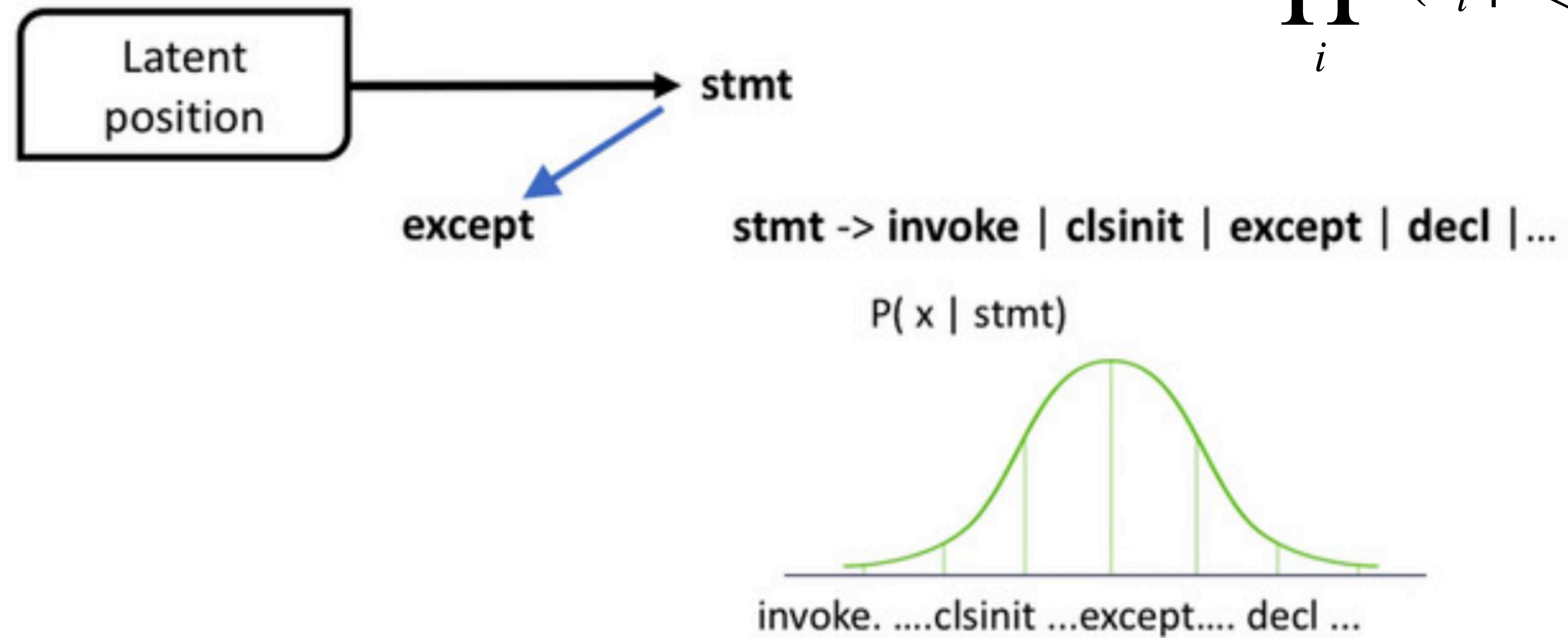
Output

```
root     -> stmt;
stmt     -> stmt; stmt
stmt     -> decl;
stmt     -> invoke
…
invoke   -> ret_var = expr_var . api_call (formal_params)
api_call -> readln | writeln | …
```

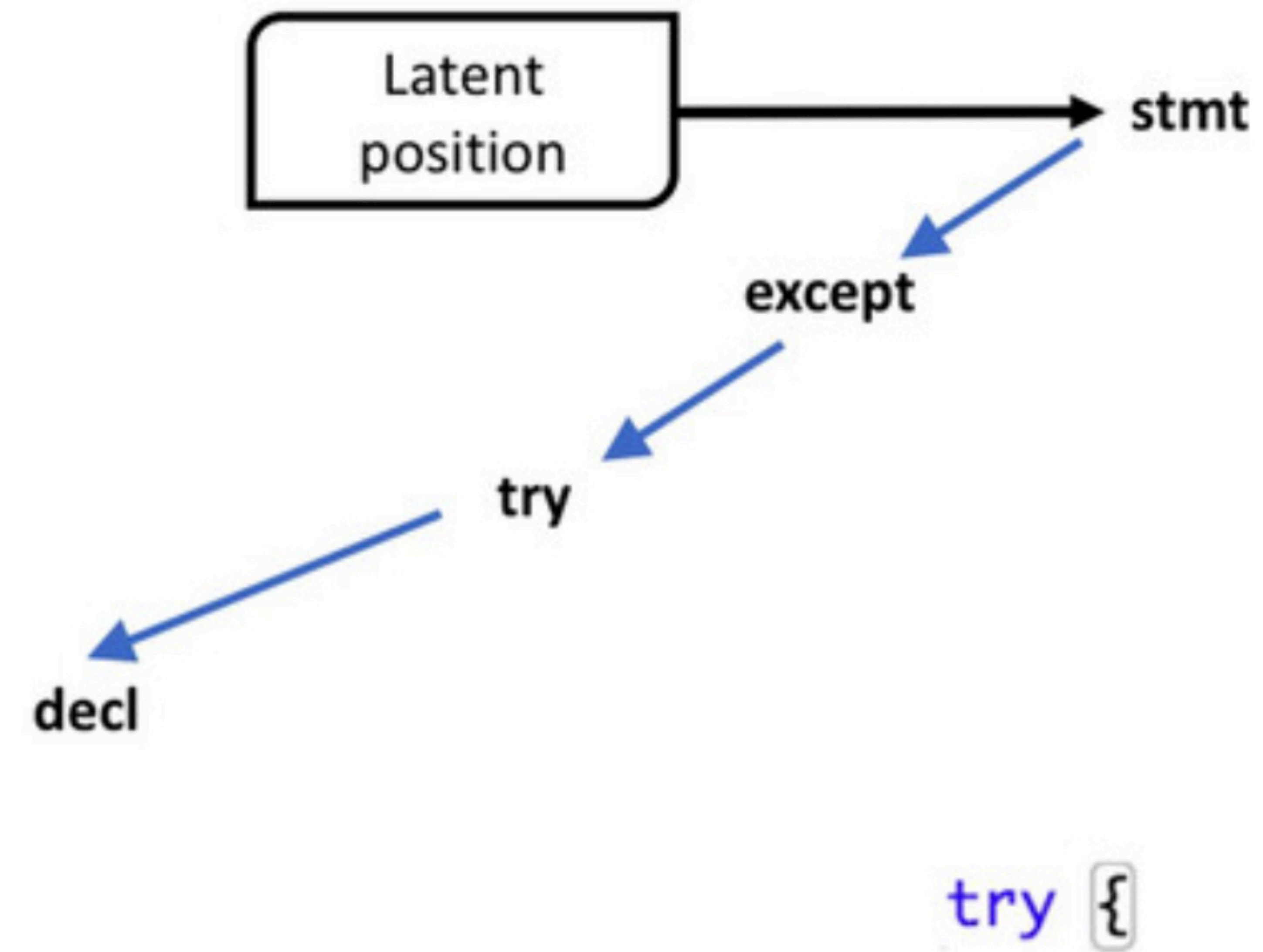
Grammar

Generation without attribute grammar

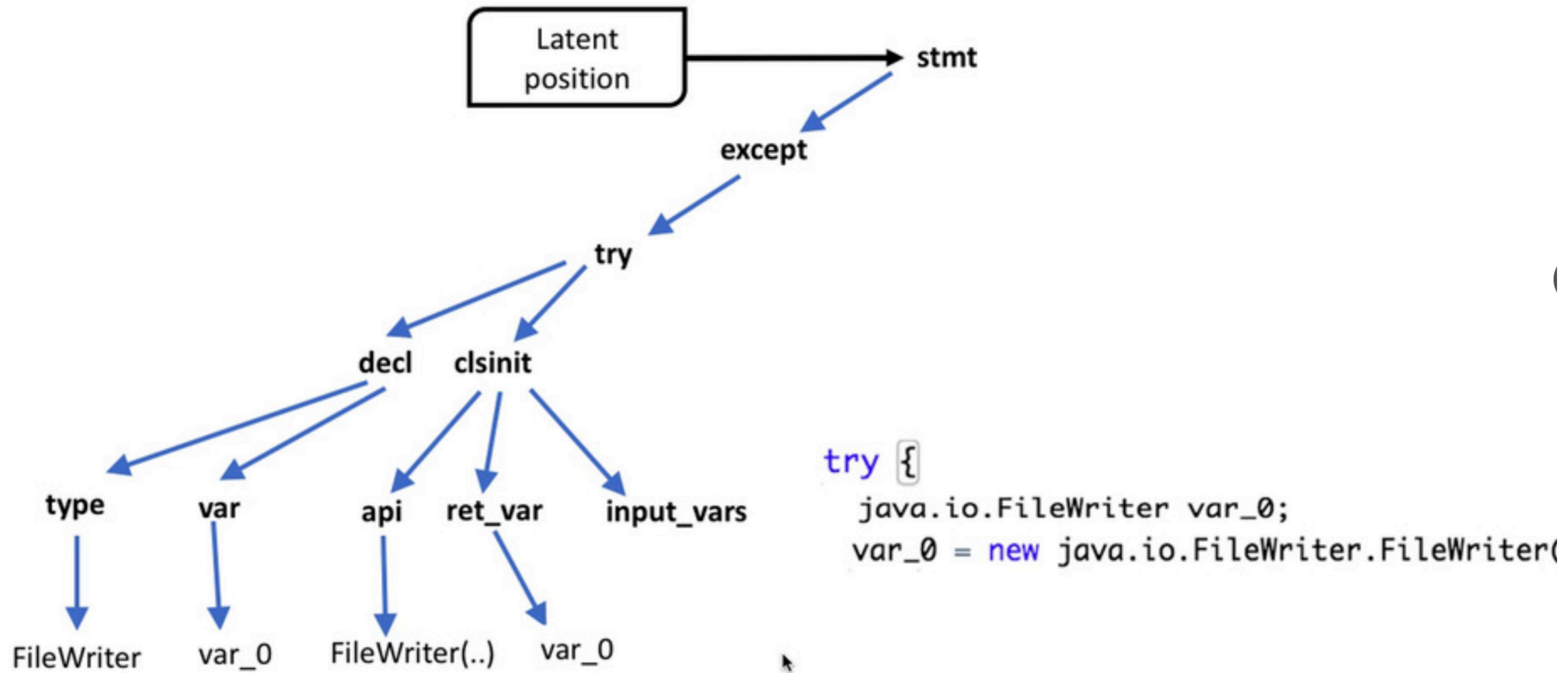
$$Y = \prod_i P(S_i \mid S_{<i}, Z)$$



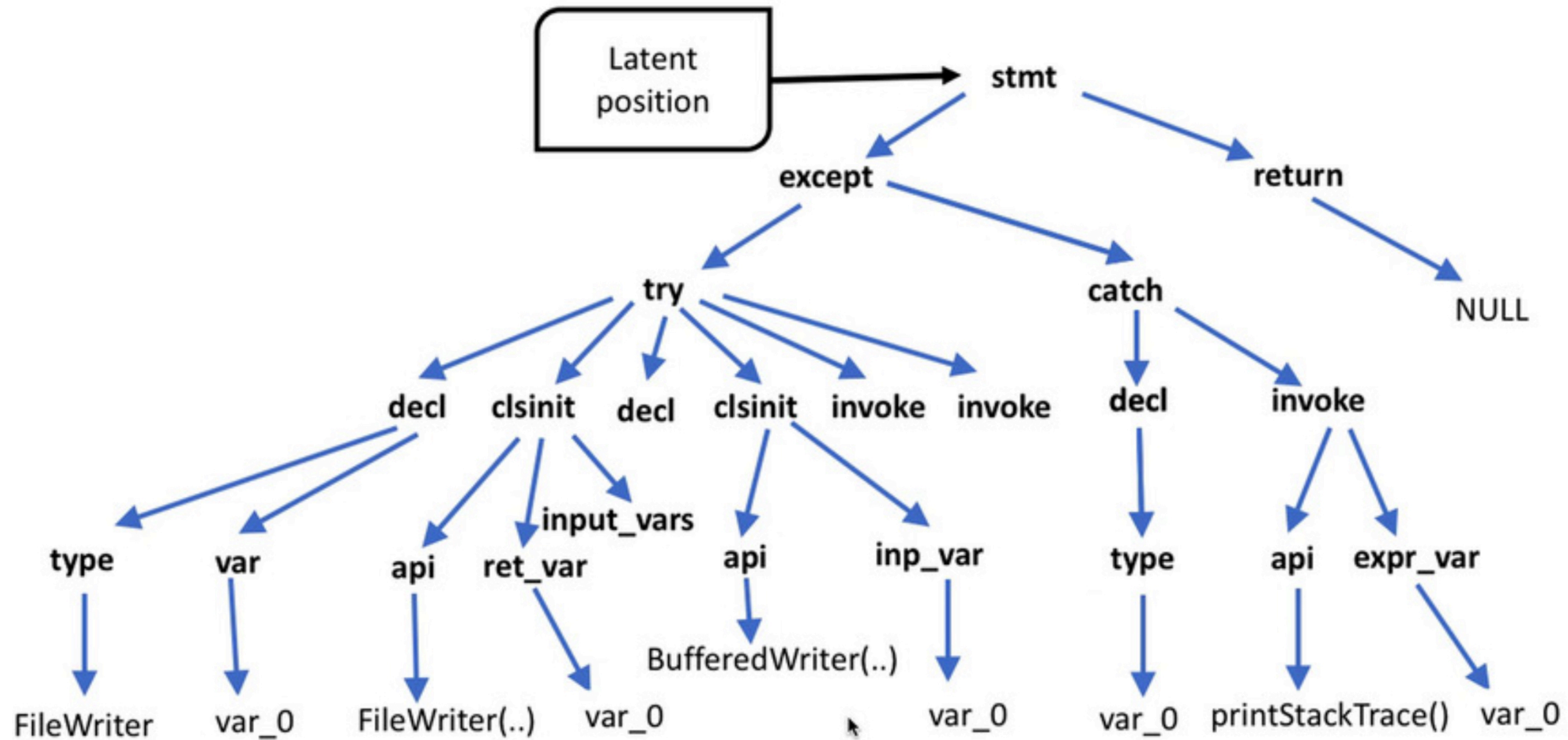
Generation without attribute grammar



Generation with vanilla CFG



Generation with vanilla CFG



What is vanilla PCFG generation losing?

- Given the data, we can train a PCFG guided generation model

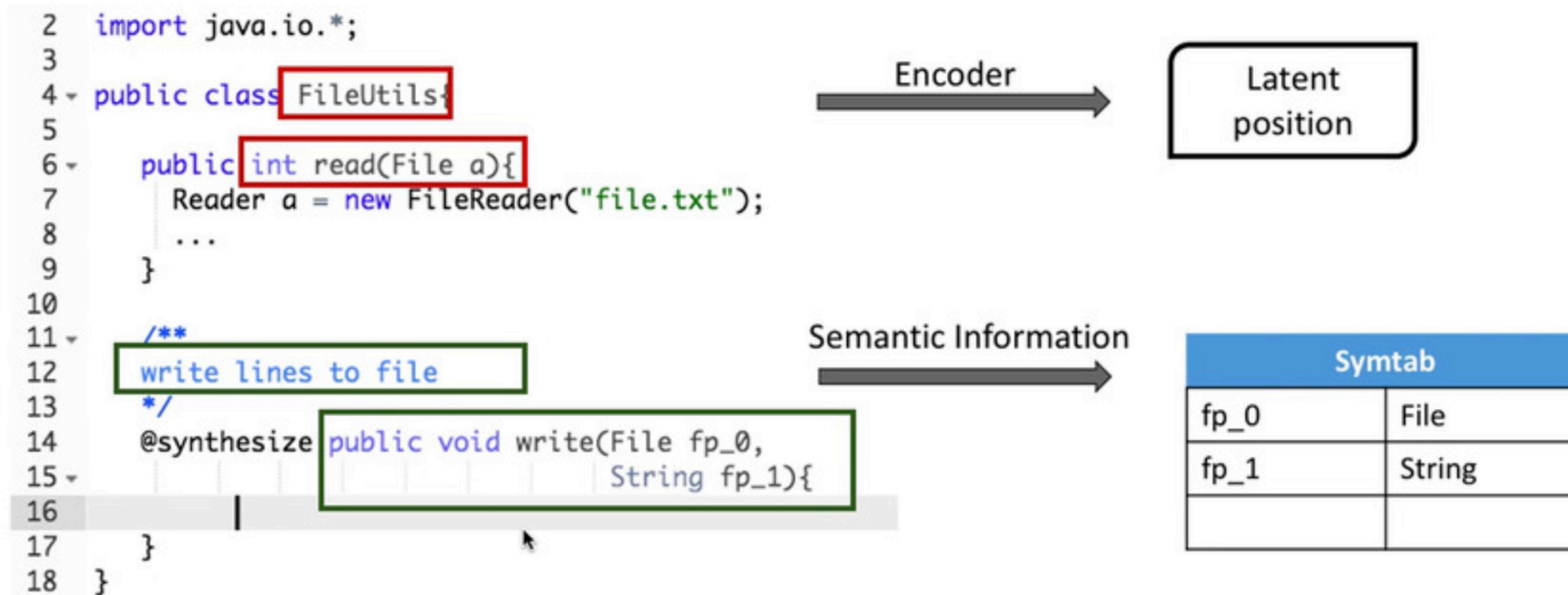
- There are some global constraints of the code that are readily available by static analysis

    - Type constraints, scope constraints etc.

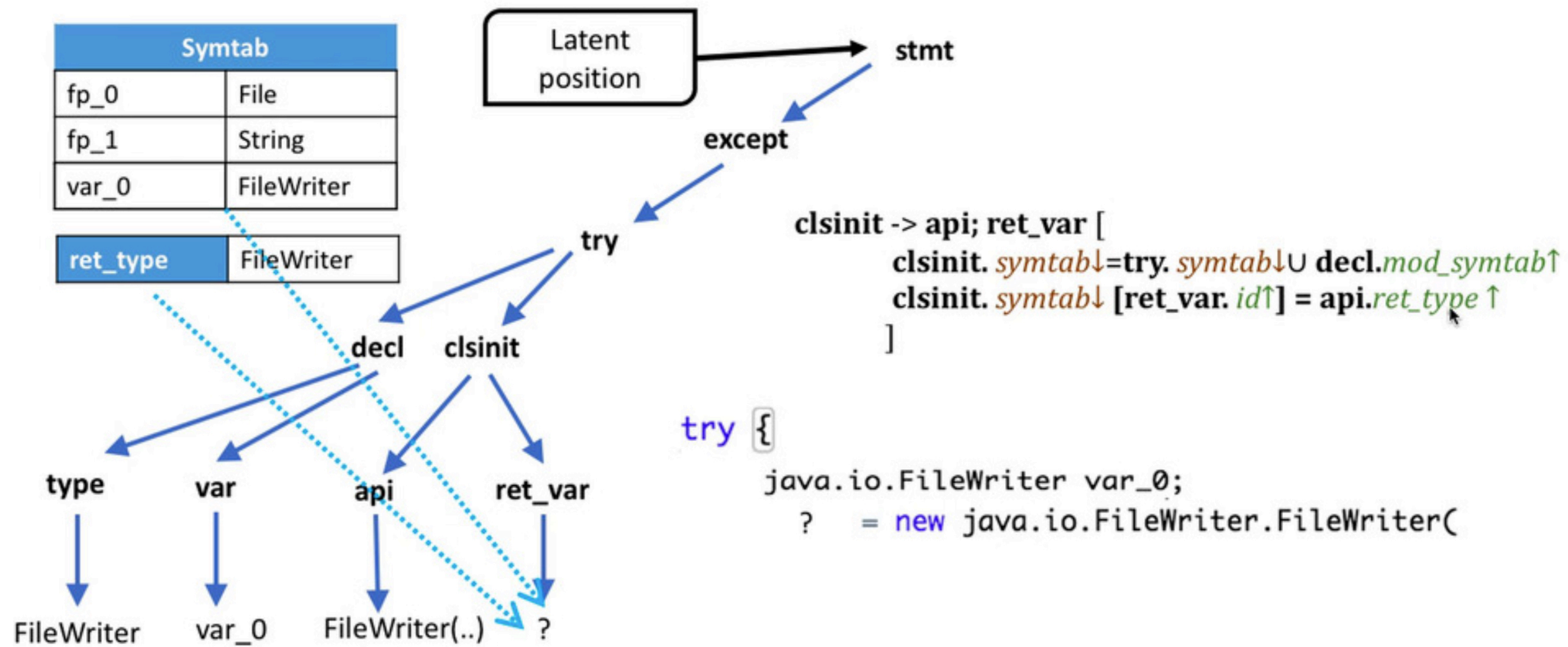
    - Can make life of the model easier

```
void write(java.io.File fp_0, java.lang.String fp_1){
    try {
            java.io.FileWriter var_0;
            var_0 = new java.io.FileWriter.FileWriter(
                    (java.io.File) fp_0, (boolean) BOOL_LITERAL);
            java.io.BufferedWriter var_1;
            var_1 = new java.io.BufferedWriter.BufferedWriter(
                        (java.io.Writer) var_0);
            var_1.write(java.lang.String: fp_1);
            var_1.newLine();
    }
    catch(java.io.IOException var_0){
        var_0.printStackTrace();
    }
    return;
}
```

Attribute grammars can maintain useful auxiliary information



Generation with Attribute Grammar



Experiments

- Architecture: **Tree-LSTM** with 63M parameters
  - However, they can only work on a subset of Java for which their grammar is defined
- Training on 1.57M method bodies of java
- Randomly remove a method body, and try to complete it with their method and baselines
- Evaluation based on key properties (compiler errors) of generated code

Results

Their method without attribute grammar (shown earlier)

Their method

	GPTNeo125M	GPTNeo1.3B	CODEX	CODEGPT	GNN2NAG	CNG	NSG
No undeclared variable access	89.87%	90.36%	88.62%	90.94%	47.44%	19.78%	**99.82%**
Valid formal parameter access	NA	NA	NA	NA	25.78%	11.03%	**99.55%**
Valid class variable access	NA	NA	NA	NA	15.40%	12.75%	**99.53%**
No uninitialized objects	93.90%	91.73%	90.82%	94.37%	21.20%	21.56%	**99.01%**
No variable access error	90.36%	90.51%	88.86%	91.32%	28.92%	17.92%	**99.69%**
Object-method compatibility	**98.36%**	98.09%	98.35%	97.84%	21.43%	12.23%	97.53%
Return type at call site	97.38%	98.01%	**98.53%**	97.83%	23.86%	16.40%	98.01%
Actual parameter type	87.03%	86.36%	92.28%	88.71%	9.27%	16.09%	**97.96%**
Return statement type	84.05%	85.09%	88.13%	85.23%	12.34%	9.51%	**90.97%**
No type errors	87.25%	88.13%	91.42%	88.10%	16.31%	13.56%	**97.08%**
Return statement exists	99.61%	99.80%	98.44%	99.57%	94.02%	**99.92%**	97.10%
No unused variables	96.42%	96.46%	96.82%	**97.64%**	20.95%	24.29%	93.84%
Percentage of parsing	98.18%	98.13%	96.41%	97.08%	**100.0%**	**100.0%**	**100.0%**
Pass all checks	65.26%	64.88%	47.49%	67.73%	17.34%	12.87%	**86.41%**

Results

- BLEU score is useless for comparing code
- Compare Jaccard score of API calls

	GPTNeo125M	GPTNeo1.3B	CODEX	CODEGPT	GNN2NAG	CNG	NSG
Set of API Calls	32%	37%	36%	36%	3%	22%	**53%**
Sequences of API Calls	17%	20%	16%	19%	0.3%	18%	**42%**
Sequences of Program Paths	12%	15%	10%	14%	0%	17%	**39%**
AST Exact Match	12%	15%	10%	14%	0%	6%	**26%**

SYNCHROMESH: RELIABLE CODE GENERATION FROM PRE-TRAINED LANGUAGE MODELS

Gabriel Poesia[*†]
Stanford University
poesia@stanford.edu

Oleksandr Polozov[*‡]
X, the moonshot factory
polozov@google.com

Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, Sumit Gulwani
Microsoft Research, Redmond
{levu,astiwar,gustavo.soares,meek,sumitg}@microsoft.com

ICLR 2022

Introduction

- **Task:** generating program p from natural language description u

Which city has the highest number of departing flights? →

```
SELECT City
FROM Flights AS T1
JOIN Airports AS T2
ON T1.SourceAirport =
    T2.AirportCode
GROUP BY City
ORDER BY COUNT(*)
DESC LIMIT 1
```

- **Setup:** few-shot prompting - create a prompt from k examples $\{(u_i, p_i)\}_{i=1}^{k}$
- **Questions**
  - 1. How to select relevant examples for the prompt?
  - 2. How to enforce additional syntactic constraints while decoding the output?

Selecting relevant examples for the prompt

Target similarity tuning

- Typical approach: Given a **u** (problem description), k examples $\{(u_i, p_i)\}_{i=1}^{k}$ find closest examples based on $f_\theta(u_i, u_j)$ for some distance metric $f_\theta \in [0,1]$
  - $L_{sim} = E_{i,j \sim \mathcal{D}_{diff}} f_\theta(u_i, u_j) - E_{i,j \sim \mathcal{D}_{sim}} f_\theta(u_i, u_j)$
- Problem: we want to retrieve **similar code**, not similar **descriptions**
  - Similarity in descriptions may not translate to similarity in the output code
- Their approach:
  - $L_{sim} = E_{i,j \sim \mathcal{D}}[f_\theta(u_i, u_j) - S(p_i, p_j)]^2$
  - Where $S(p_i, p_j) \in [0,1]$ is similarity between the syntax trees
  - Similarity function now needs to pay attention to the difference in the syntax trees

Selecting relevant examples for the prompt

Target similarity tuning

Question:
"Which city has the highest number of airports?"

Training set

S-BERT

S-BERT + TST

Prompt a.

```
Example #1:
"Which city has the highest elevation?"
SELECT City FROM Airports
ORDER BY Elevation DESC LIMIT 1
Example #2: ...
```

GPT-3

b.

```
SELECT City FROM Airports
ORDER BY NumberOfAirports DESC LIMIT 1
```

No column "NumberOfAirports" in table "Airports"

Prompt

```
Example #1:
"Return the team with the most technicians."
SELECT Team FROM Technician
GROUP BY Team
ORDER BY COUNT(*) DESC LIMIT 1
Example #2: ...
```

GPT-3

c.

```
SELECT City FROM Airports
GROUP BY City
ORDER BY COUNT(*) DESC LIMIT 1
```

Constrained semantic decoding

- Unconstrained language models may produce wrong output when completing complex expressions
  - Key idea: code is structured, can help using grammar
- Two layers: context-sensitive and context-free
- Context-free layer:
  - Uses parser to restrict the set of next tokens
- Context-sensitive layer:

Constraint	Example of partial program	Valid/Invalid Examples
A valid identifier must follow after `AS`.	`SELECT Name, Role FROM User AS` ^	U ✓ T1 ✓ 2 ×
Column names must come from schema, even behind aliases.	`SELECT U.Name FROM User AS U WHERE U.` ^	Name ✓ DoB ✓ Birthday ×

Completion Engine

- Instead of sampling from an unconstrained set, sample tokens that would keep the program generated so far valid
  - Let p_t be the program decoded till current time step t
  - Use a parser to find a list of production rules and possible token types that can follow p_t
- Let L^c be the set of languages that can be completed to a valid program. Given a string s, how to determine if it is in L^c?
- $V_M(s) = \{t \in \Sigma_M : st \in L^c\}$

Completion Engine

Question:
"Which city has the highest number of departing flights?"

Training set

S-BERT + TST

Prompt

Example #1:
"Return the team with the most technicians."
SELECT Team
FROM Technician
GROUP BY Team
ORDER BY COUNT(*) DESC
LIMIT 1
Example #2:
...

GPT-3

a.

```
SELECT City
FROM Flights AS T1 JOIN Airports AS T2
ON T1.AirportCode = T2.SourceAirport
GROUP BY City
ORDER BY COUNT(*) DESC LIMIT 1
```

No column "AirportCode" in table aliased as T1

CSD

SQL Completion Engine

GPT-3

b.

```
SELECT City
FROM Flights AS T1 JOIN Airports AS T2
ON T1.~~AirportCode~~SourceAirport = T2.AirportCode
GROUP BY City
ORDER BY COUNT(*) DESC LIMIT 1
```

Results

Model	SQL Exec.	SQL Valid	SQL Dist.	Vega-Lite Acc.	Vega-Lite Valid	Vega-Lite Dist.	SMCalFlow Acc.	SMCalFlow Valid	SMCalFlow Dist.
Andreas et al. (2020)	-	-	-	-	-	-	72%$^{(S)}$	-	-
Srinivasan et al. (2021)	-	-	-	64%$^{(S)}$	-	-	-	-	-
Rubin & Berant (2021)	71%$^{(S)}$	-	-	-	-	-	-	-	-
Scholak et al. (2021)	79%$^{(S)}$	98%	-	-	-	-	-	-	-
GPT-3 13B	16%	43%	0.42	14%	55%	0.51	38%	76%	0.43
” + CSD	20%	66%	0.44	17%	100%	0.48	40%	95%	0.40
” + TST	14%	48%	0.42	-	-	-	60%	88%	0.22
” + CSD + TST	19%	72%	0.43	-	-	-	63%	98%	0.17
GPT-3 175B	28%	49%	0.36	20%	67%	0.36	44%	77%	0.41
” + CSD	35%	73%	0.36	25%	100%	0.32	45%	97%	0.37
” + TST	31%	56%	0.35	-	-	-	60%	88%	0.24
” + CSD + TST	37%	76%	0.34	-	-	-	66%	97%	0.18
Codex 175B	56%	73%	0.25	39%	87%	0.24	45%	79%	0.37
” + CSD	61%	85%	0.23	40%	99%	0.23	46%	97%	0.33
” + TST	60%	81%	0.23	-	-	-	63%	90%	0.21
” + CSD + TST	64%	85%	0.23	-	-	-	63%	99%	0.19

Break-It-Fix-It: Unsupervised Learning for Program Repair

Michihiro Yasunaga [1] **Percy Liang** [1]

ICML 2021

Overview

- Fix a given piece of incorrect code

- No parallel data available

- BUT:
  - Large quantities of code online
  - Compiler can check if the code is correct or not

Initialization with synthetic errors

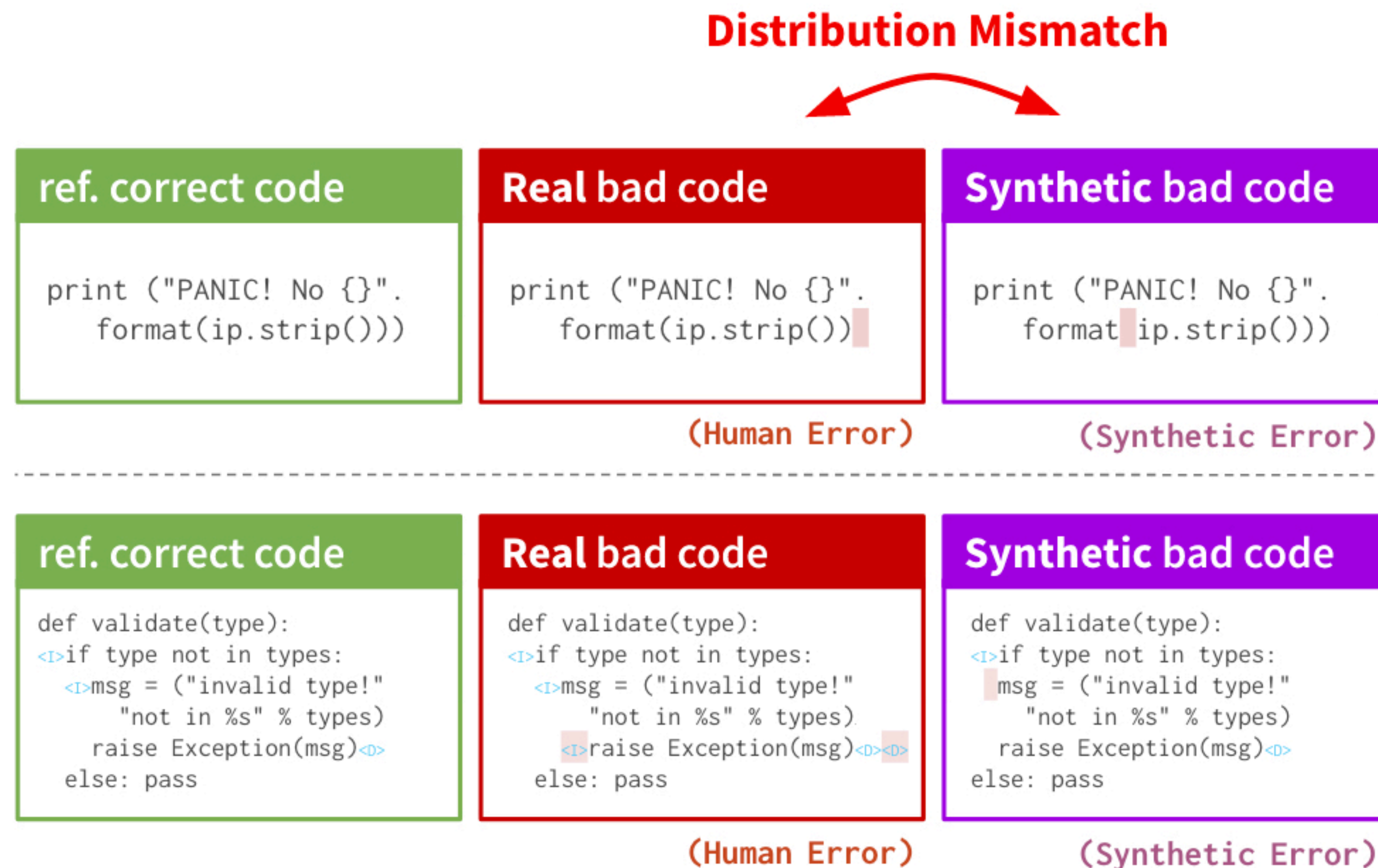
- Let D be a large corpus of code, and D_{good} and D_{bad} be the correct and incorrect splits.
- Introduce synthetic perturbations (random token drop/typos/punctuation errors) in D_{good}
- $\mathcal{P}_{\text{synthetic}} = \{(b_{\text{synthetic}}(y), y) \mid y \in \mathcal{D}_{\text{good}}\}$
- Train initial *fixer* f0 to go from bad to good, and initial breaker b0 to go from good to bad

$$b_0 = \textsc{Train}^{\text{good}\to\text{bad}}(\mathcal{P}_{\text{synthetic}})$$
$$f_0 = \textsc{Train}^{\text{bad}\to\text{good}}(\mathcal{P}_{\text{synthetic}})$$

Initialization with synthetic errors

Why is it not enough?

- Real and synthetic error distribution is different



Break-it-fix-it loop

- During initialization, D_{bad} was not used
- Run inference **f0** on D_{bad} to fix the originally bad examples, keep those that are actually fixed

$$\mathcal{P}_k^{(f)} = \{(x, f_{k-1}(x)) \mid x \in \mathcal{D}_{\text{bad}}, c(f_{k-1}(x)) = 1\}$$
$$b_k = \text{TRAIN}^{\text{good}\to\text{bad}}(\mathcal{P}_k^{(f)})$$

$c(f_k(y)) = 1$: code is correct after applying f_k

- Breaker **b1** learns to generate *actual* errors
- Create more examples, train **f1,** repeat

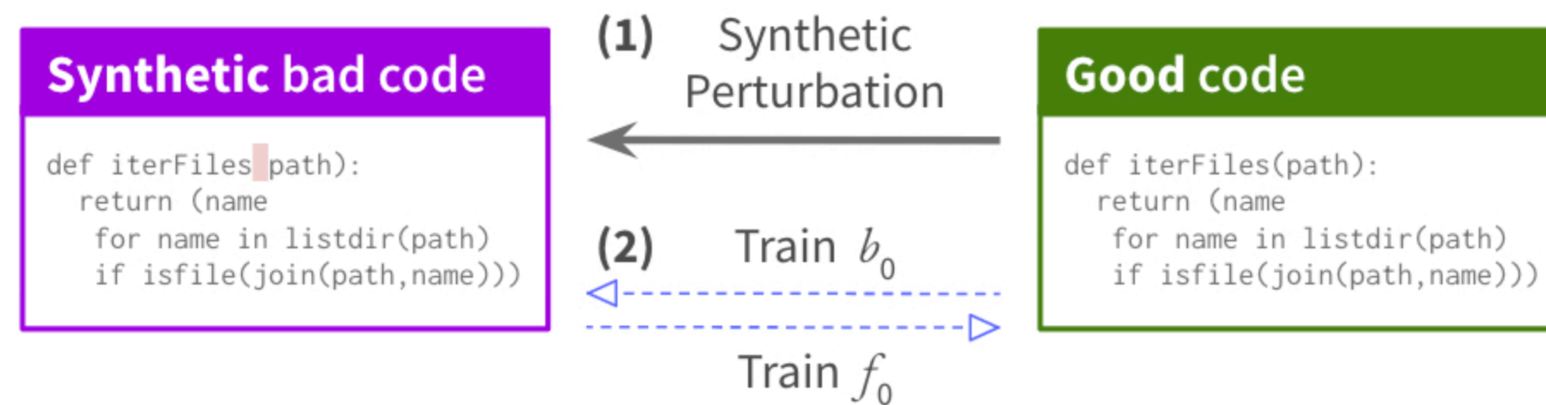
$$\mathcal{P}_k^{(b)} = \{(b_k(y), y) \mid y \in \mathcal{D}_{\text{good}}, c(b_k(y)) = 0\}$$
$$f_k = \text{TRAIN}^{\text{bad}\to\text{good}}(\mathcal{P}_k^{(f)} \cup \mathcal{P}_k^{(b)}).$$

Difference with Backtranslation

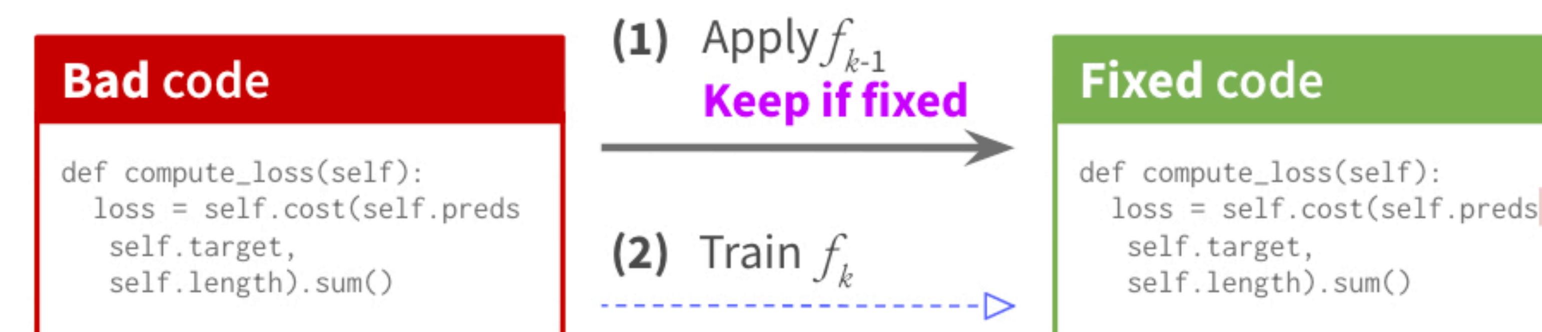
- Conceptually identical

- Main differences: no strict criteria for filtering in back translation, have a good measure here (compiler/parser)

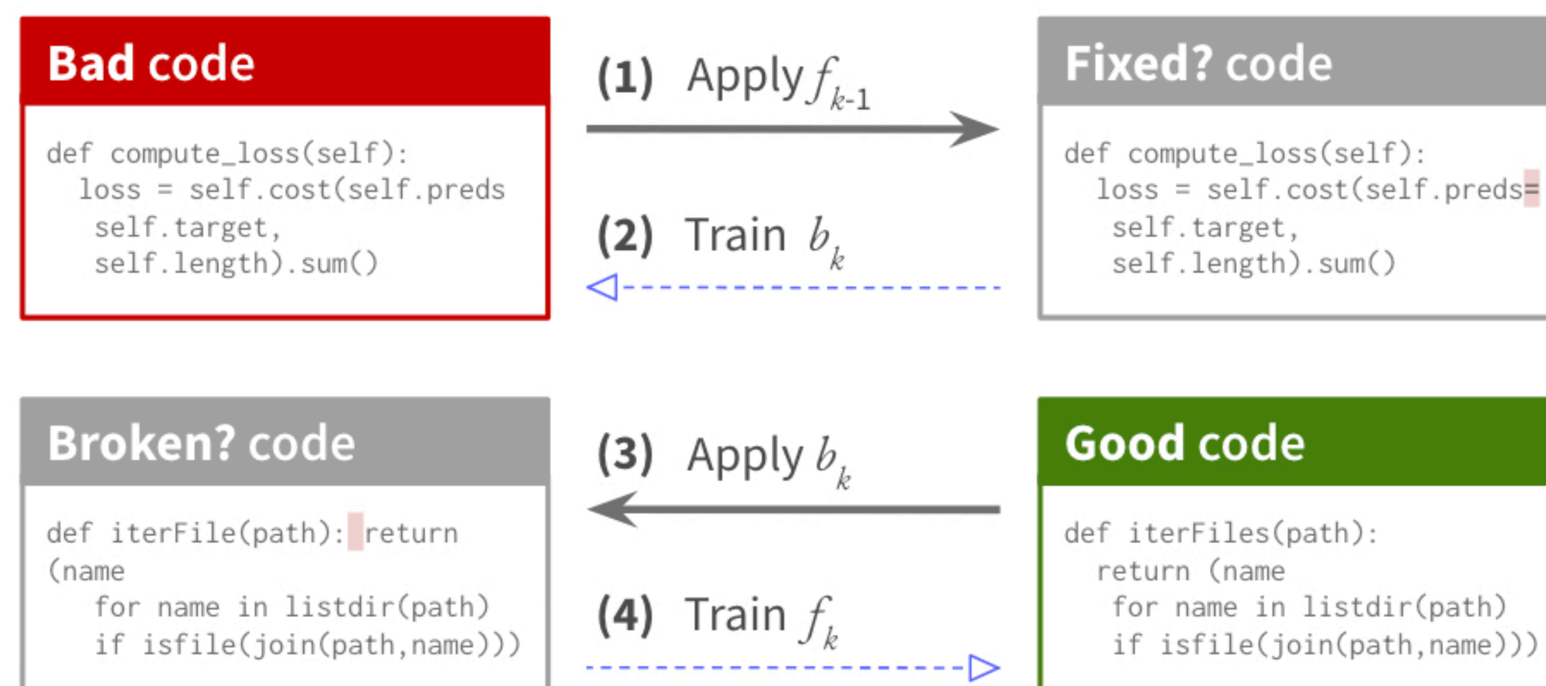
§3.1 Initialization — Round 0



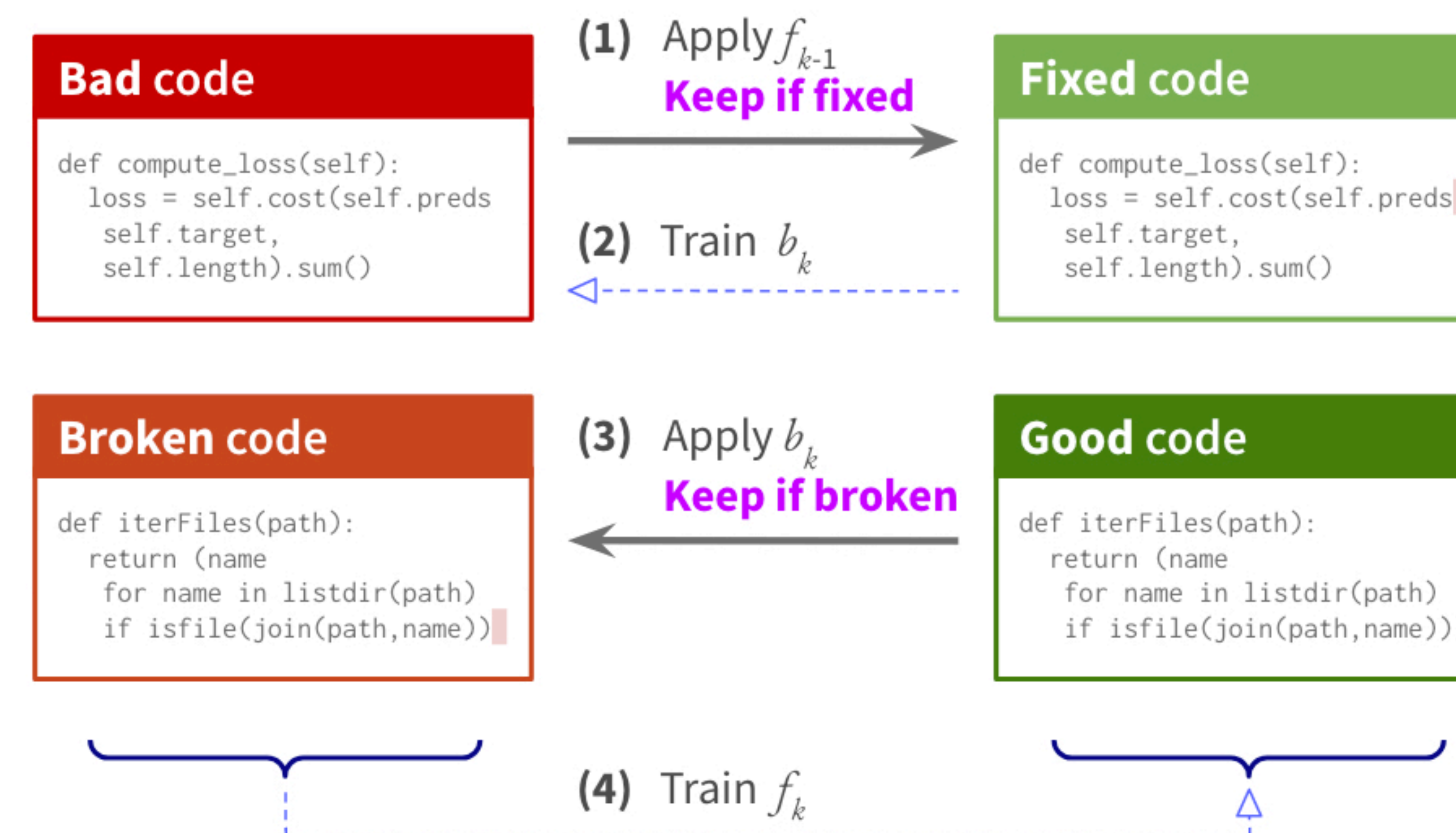
§3.4 FixerOnly — Round *k* (=1,2, ...)



§3.3 ref. Backtranslation — Round *k* (=1,2, ...)



§3.2 Break-It-Fix-It (BIFI) — Round *k* (=1,2, ...)



Implementation

- **Data:** open source Python files from Github
  - 3M code snippets
  - Check errors using Python parser
  - Code is correct if it has no parse errors AND is less than 5 tokens from the input code (correct but not change)
  - 38k wrong (from 3M)
- **Implementation:**
  - Uses encoder-decoder architecture
  - Relatively small model: 4 layers, 8 heads

Results

- Bad code helps

Method		Test accuracy			
		Bad 100%	Bad 50%	Bad 10%	ref. Synthetic bad only
Initial	Round-0	62.0%	62.0%	62.0%	62.0%
FixerOnly	Round-2	88.6%	84.7%	78.5%	62.7%
BIFI	Round-2	90.5%	89.0%	86.7%	63.3%

Method		**Test accuracy**
Initial	Round-0	62.0%
BIFI (**ours**)	Round-2	**90.5%**
– real bad	Round-2	84.6%
– critic	Round-2	84.0%
– both (backtranslation)	Round-2	80.1%

Summary

- Lots of progress made by large language models by treating code as language
- Exploiting properties of code lead to useful modifications of popular NLP techniques:
  - Backtranslation
  - Syntax-guided generation
  - Retrieval-augmented generation
  - MLM
- Future work: using some of these techniques in graph generation