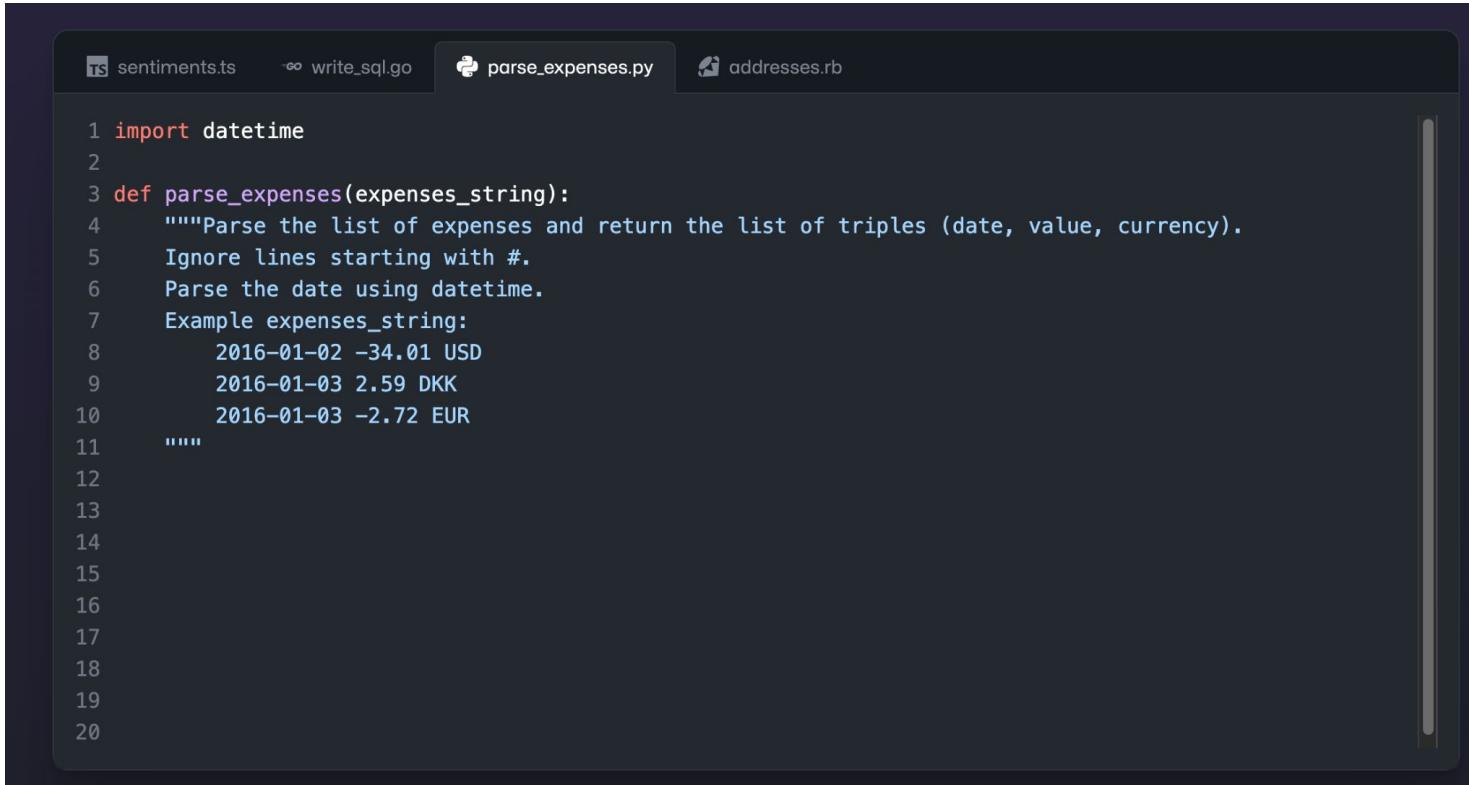




CodeLMs

Ge Qu & Jinyang Li

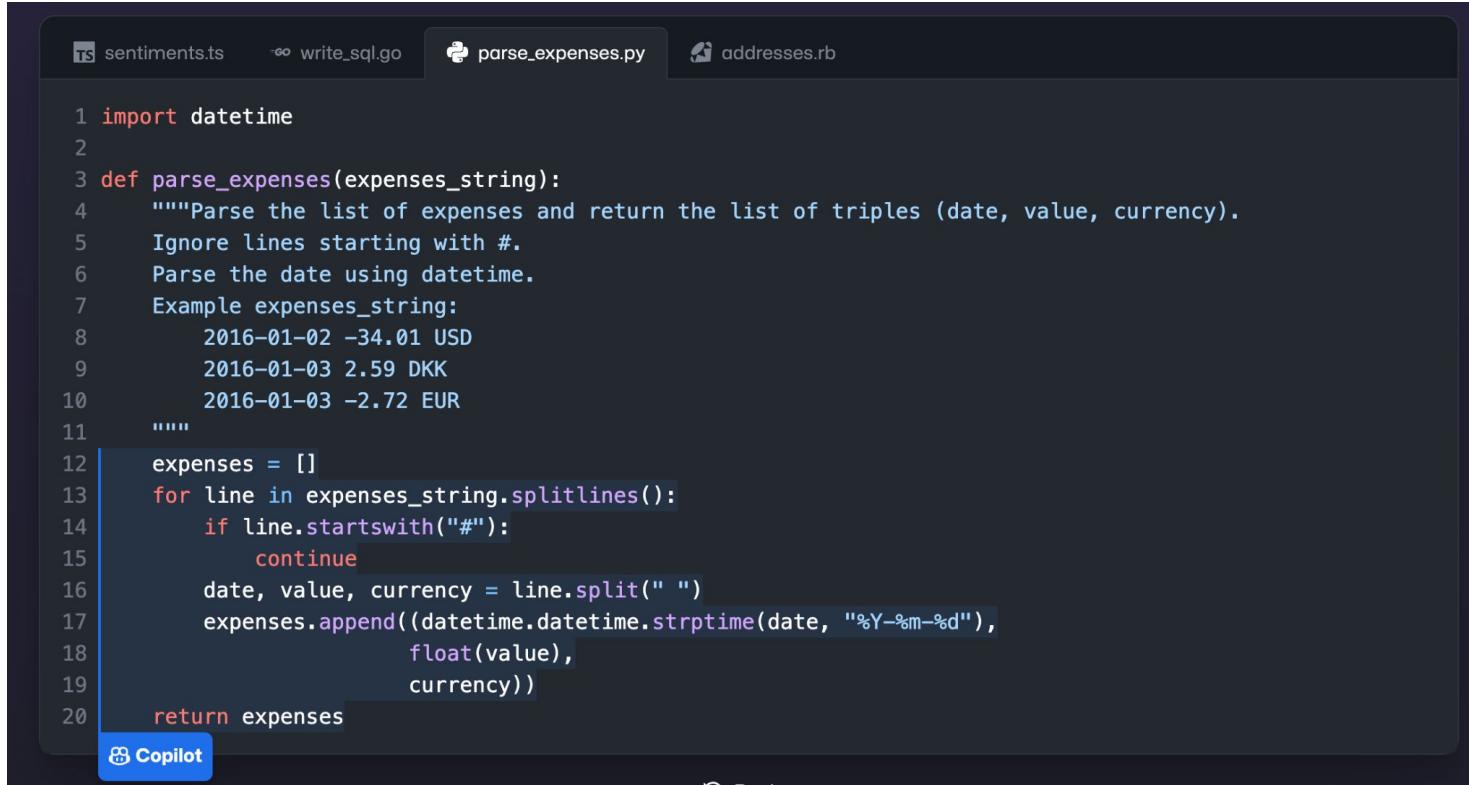
Copilot: An AI pair Programmer



A screenshot of a dark-themed code editor interface. At the top, there are four tabs: 'sentiments.ts' (TypeScript icon), 'write_sql.go' (Go icon), 'parse_expenses.py' (Python icon, currently active), and 'addresses.rb' (Ruby icon). The main area displays a Python script named 'parse_expenses.py'. The code is as follows:

```
1 import datetime
2
3 def parse_expenses(expenses_string):
4     """Parse the list of expenses and return the list of triples (date, value, currency).
5     Ignore lines starting with #.
6     Parse the date using datetime.
7     Example expenses_string:
8         2016-01-02 -34.01 USD
9         2016-01-03 2.59 DKK
10        2016-01-03 -2.72 EUR
11    """
12
13
14
15
16
17
18
19
20
```

Copilot: An AI pair Programmer



A screenshot of a code editor interface. At the top, there are four tabs: 'sentiments.ts', 'write_sql.go', 'parse_expenses.py' (which is the active tab), and 'addresses.rb'. The main area displays Python code for parsing expense strings. The code uses the `datetime` module to parse dates. It iterates over lines in the input string, ignoring lines starting with '#'. For each valid line, it splits the input into date, value, and currency, then appends a tuple to a list named 'expenses'. Finally, it returns the list of expenses. The code editor has a dark theme with syntax highlighting.

```
1 import datetime
2
3 def parse_expenses(expenses_string):
4     """Parse the list of expenses and return the list of triples (date, value, currency).
5     Ignore lines starting with #.
6     Parse the date using datetime.
7     Example expenses_string:
8         2016-01-02 -34.01 USD
9         2016-01-03 2.59 DKK
10        2016-01-03 -2.72 EUR
11    """
12    expenses = []
13    for line in expenses_string.splitlines():
14        if line.startswith("#"):
15            continue
16        date, value, currency = line.split(" ")
17        expenses.append((datetime.datetime.strptime(date, "%Y-%m-%d"),
18                         float(value),
19                         currency))
20    return expenses
```

 Copilot

Definition

Semantic Parsing:

- 💡 Convert natural language into a **logical form** that is **machine-understandable**.

Generate a python function
for web scrabing



```
import requests
from bs4 import BeautifulSoup
# Create a session object of the website you want to scrape
url = "https://www.example-news-website.com/"
# Create a GET request to the website
response = requests.get(url)
# Check if the request was successful (status code 200)
if response.status_code == 200:
    # Parse the response text using the html parser
    soup = BeautifulSoup(response.text, "html.parser")
    # Find all the article titles using their HTML tag and class
    articles = soup.findAll("div", {"class": "article-title"})
    # Print the titles of the articles
    for article in articles:
        print(article.text)
else:
    print(f"Failed to fetch the website with status code {response.status_code}")
```

Program Synthesis:

- 💡 Program that takes a specification and outputs a program that satisfies it

- 💡 Logical Formula
- 💡 Other Equivalent Program
- 💡 Input/Output Example
- 💡 Natural Language Description

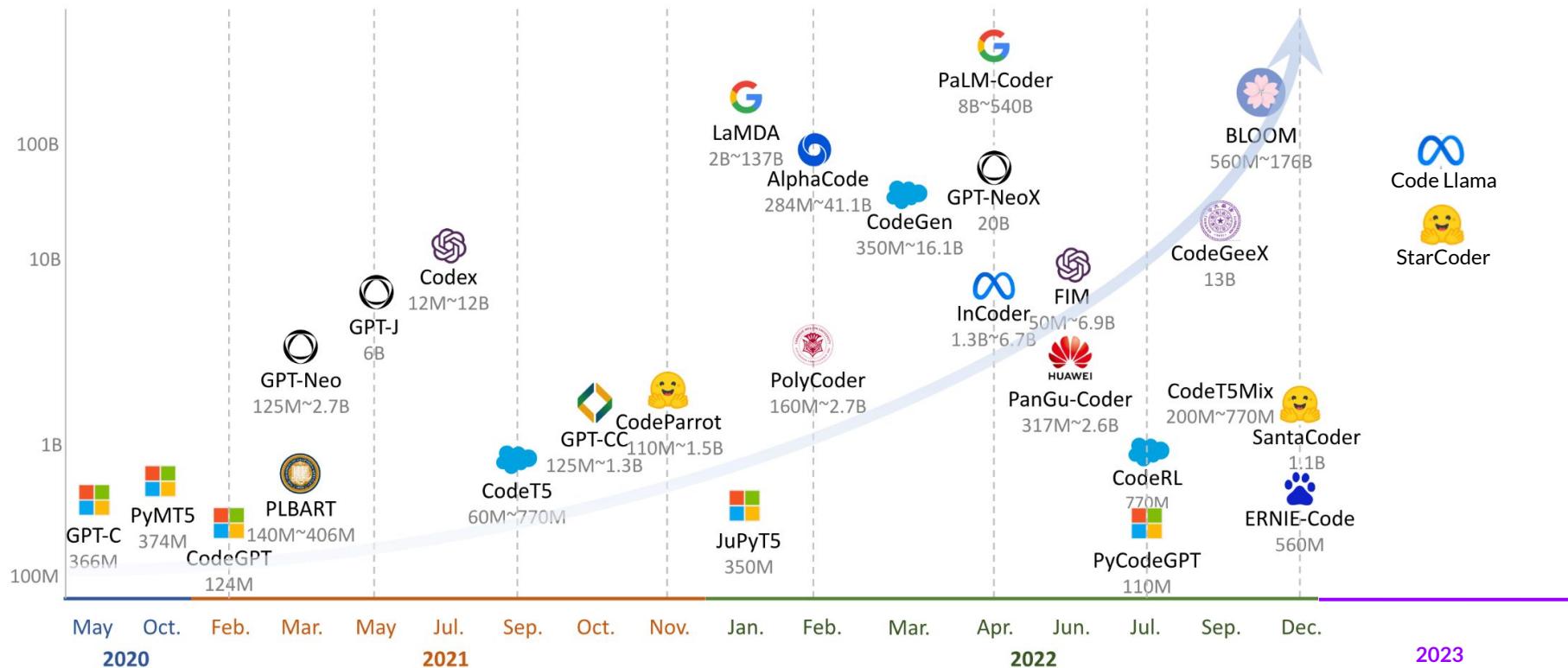


Programs

CodeLM

Language Models (LMs) that are specifically designed and trained to understand and generate code

Large Language Models Meet NL2Code: A Survey
(Zan et al., ACL 2023)



Evaluating Large Language Models Trained on Code

OpenAI

Keypoints in Codex

Motivation:

- 💡 GPT-3 could generate simple programs from Python docstrings
- 💡 Code is massively available from open source projects, like projects on Github
- 💡 Idea in OpenAI Codex: Train large language model on majority code data
- 💡 Codex (v1): Same Structure as GPT3, but with 12B parameters (vs 175B)

Synthesis Challenge:

- 💡 Python docstring → Function Implementation
- 💡 Problem: Potential data leakage

HumanEval

164 original programming problems with unit tests

Functional Correctness:
A program is correct if it passes all the hidden tests

Repeated sampling from the model is a surprisingly effective strategy

pass@k:
at least one sample is correct out of k samples

```
def incr_list(l: list):
    """Return list with elements incremented by 1.
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    """
    return [i + 1 for i in l]
```

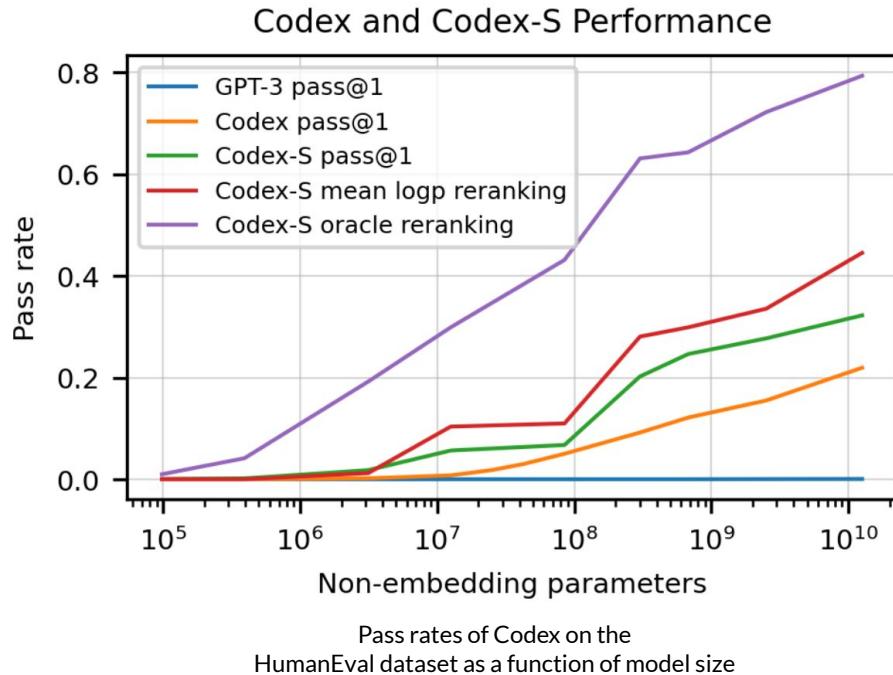
```
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements that are in even positions.

    Examples
    solution([5, 8, 7, 1]) =>12
    solution([3, 3, 3, 3, 3]) =>9
    solution([30, 13, 24, 321]) =>0
    """
    return sum(lst[i] for i in range(0, len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

```
def encode_cyclic(s: str):
    """
    returns encoded string by cycling groups of three characters.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group. Unless group has fewer elements than 3.
    groups = [(group[-1:] + group[:-1]) if len(group) == 3 else group for group in groups]
    return ''.join(groups)

def decode_cyclic(s: str):
    """
    takes as input string encoded with encode_cyclic function. Returns decoded string.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group.
    groups = [(group[-1:] + group[:-1]) if len(group) == 3 else group for group in groups]
    return ''.join(groups)
```

Evaluating LLMs Trained on Code



- 💡 GPT3-12B fails on all problems
- 💡 Codex (fine-tuned on code) solves **28.8%** of the problems
- 💡 Codex-S (further fine-tuned on correctly implemented standalone functions) solves **37.7%** of the problems
- 💡 Further gains can be realized by generating more samples and reranking them

Evaluating LLMs Trained on Code

```
def is_prime(n):
    """Return true if a given number is prime, and
       false otherwise.
    >>> is_prime(6)
    False
    >>> is_prime(101)
    True
    >>> is_prime(11)
    True
    >>> is_prime(13441)
    True
    >>> is_prime(61)
    True
    >>> is_prime(4)
    False
    >>> is_prime(1)
    False
    """
    """
```

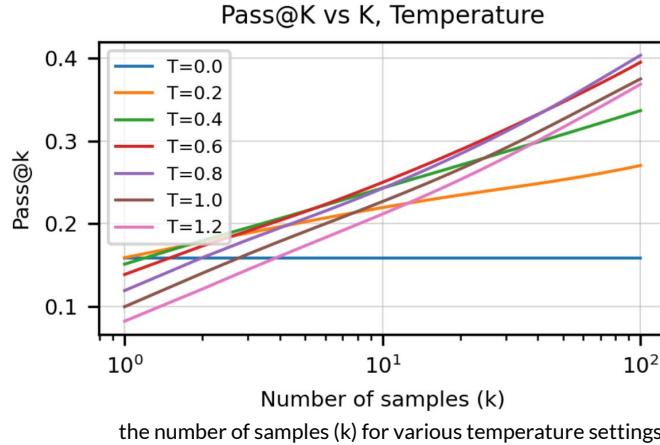
```
### COMPLETION 1 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True
```

```
### COMPLETION 3 (WRONG): ###
# TODO: implement this function
pass
```

```
### COMPLETION 4 (CORRECT): ###
if n < 2: return False
if n == 2: return True
if n%2 == 0: return False
return not any(n%k == 0 for k in range(3,int(n
    **0.5)+1,2))
```

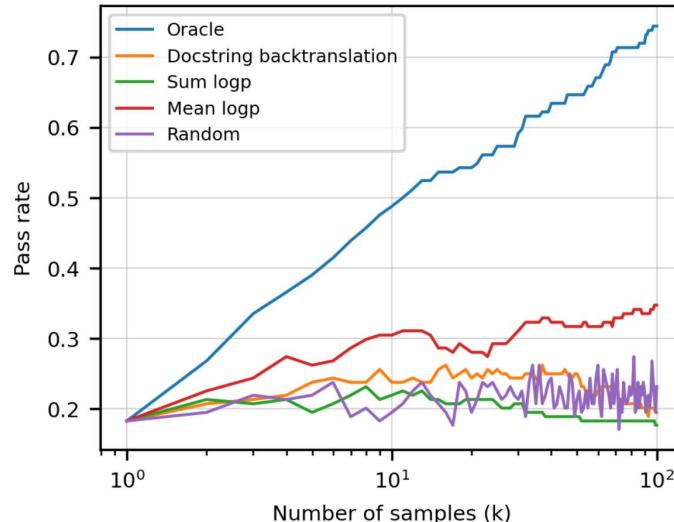
```
### COMPLETION 2 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True
```

Evaluating LLMs Trained on Code



Higher temperatures are optimal for larger k

- 💡 Reranking strategy is also important
- 💡 Oracle: Run all hidden unit tests and pick the sample that passes all
- 💡 Use model's mean log-probabilities to rerank if we can't access to the oracle



Model performance in the setting where we can generate multiple samples, but only evaluate one

Some Interesting Explorations

💡 $P(\text{code} \mid \text{docstring}) \rightarrow P(\text{docstring} \mid \text{code})$

Codex-D

Low performance in this direction

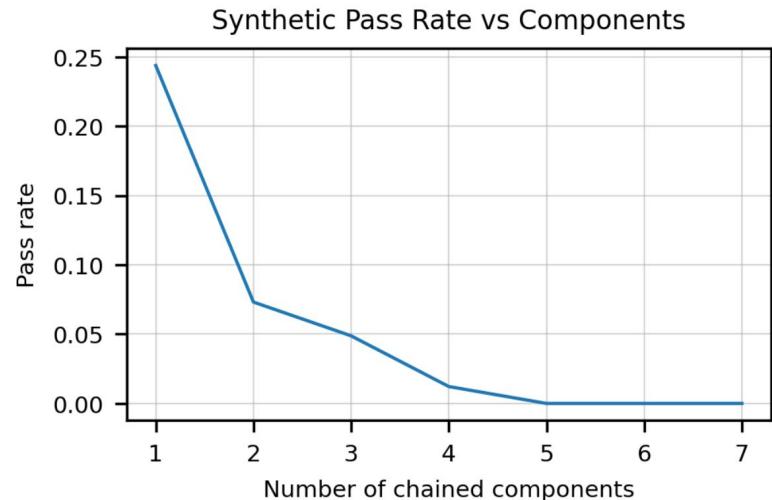
Even fine-tuning with teacher forcing didn't really help

💡 Struggle with long and higher level specifications

Even just chain simple docstrings together

MODEL	PASS @ 1	PASS @ 10
CODEX-S-12B	32.2%	59.5%
CODEX-D-12B	20.3%	46.5%

Pass rates for the docstring generating model Codex-D



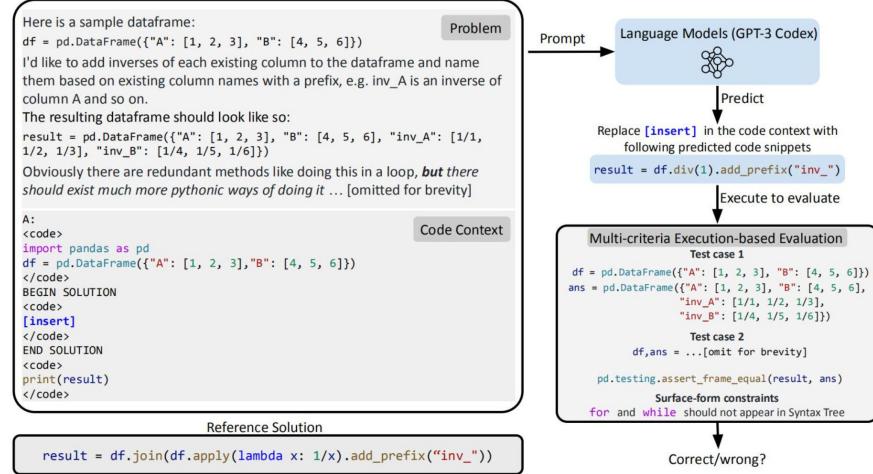
Pass rates of Codex-12B samples against the number of chained components in the synthetically generated docstring

DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation

HKU

DS-1000

- 💡 a data science code generation benchmark with **1K questions** about **7 python libraries**
- 💡 contains realistic problems with diverse contexts
- 💡 implements reliable multi-criteria execution-based evaluation metrics
- 💡 proactively defends against memorization



	Pandas	NumPy	Matplotlib	Scikit-learn	SciPy	TensorFlow	PyTorch	Total/Avg.
Problem	291	220	155	115	106	45	68	1000
Origin	100	97	111	46	58	17	22	451
Surface Perturbation	24	22	0	57	11	11	27	152
Semantic Perturbation	88	51	44	9	20	12	11	235
Difficult Rewrite	79	50	0	3	17	5	8	162
% Surface-Form Constraints	12.0	36.4	0	27.8	17.9	20.0	27.9	19.4
Avg. Test Cases	1.7	2.0	1.0	1.5	1.6	1.6	1.7	1.6
Avg. Problem Words	184.8	137.5	21.1	147.3	192.4	133.3	133.4	140.0
Avg. Lines of Code Context	9.0	8.3	6.9	11.0	10.2	9.2	9.0	8.9
Avg. Lines of Code Solution	5.4	2.5	3.0	3.3	3.1	4.1	2.1	3.6

Detailed statistics of DS-1000.

Benchmark Construction Pipeline

① Manually Selecting and Modifying StackOverflow Problems

```
6 Here is a sample dataframe:  
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})  
7 I'd like to add inverses of each existing column to the dataframe  
8 and ... [omitted for brevity]  
9 try:  
10 inv_df = df.join(df.apply(lambda x: 1/x).add_prefix("inv_"))
```

- ✓ High vote
- ✓ Testable
- ✓ Useful
- ✓ Representative



② Adding Code Context

```
import pandas as pd  
df = pd.DataFrame({"A": [1, 2, 3],  
                   "B": [4, 5, 6]})  
### BEGIN SOLUTION  
[insert]  
### END SOLUTION  
print(result)
```

rewrote the problem and the reference solution

③ Red Teaming

```
df = pd.DataFrame({"A": [1, 2, 3],  
                   "B": [4, 5, 6]})  
### BEGIN SOLUTION  
# A known WRONG SOLUTION  
result = df.join(df.apply(lambda  
x:math.e**x).add_prefix('exp_'))  
### END SOLUTION  
print(result)
```



reject all predictions known to be incorrect

④ Perturbing Original Problem

... I'd like to apply the *exponential* function to each existing column ... The resulting dataframe should look like so:

```
result = pd.DataFrame({"A": [1, 2, 3],  
                      "B": [4, 5, 6],  
                      "exp_A": [e^1, e^2, e^3],  
                      "exp_B": [e^4, e^5, e^6]})  
... [omitted for brevity]
```



memorize the pre-training corpus?

⑤ Implementing Automatic Tests

Test cases
...[omit for brevity]
pd.testing.assert_frame_equal(result,
 ans)
Surface-form constraints
for and while should not appear in Syntax Tree



Functional Correctness Test



Surface-form Constraints

Multi-Criteria Evaluations

Functional Correctness

- constructed test cases
- compared executed results with gt

Surface-Form Constraints

- functional correctness alone is insufficient
- require the presence/absence of specific APIs for keywords

- Sample Level False Discovery Rate: among all predicted samples that **pass** our automatic evaluation, 1.8% of them are **incorrect** according to our annotator.
- Sample Level False Omission Rate: among all predicted samples that **do not pass** our automatic evaluation, 0.5% of them are **correct** according to our annotator.
- Problem Level False Positive Percentage: among all problems, 5.7% of the problems contain at least one incorrect sample prediction that pass our automatic metric.
- Problem Level False Negative Percentage: among all problems, 5.7% (it happens to be the same as the above) problems contain at least one correct sample prediction that fails to pass our automatic metric.

Problem:
I could not find a built-in function in Python to generate a log uniform distribution given a min and max value (the R equivalent is here), something like: `logunif(n, min, max, base)` that returns n log uniformly distributed in the range min and max.
The closest I found though was `numpy.random.uniform`.
That is, given range of x, I want to get samples of given size (n) that suit log-uniform distribution.
Any help would be appreciated!

```
A:  
<code>  
import numpy as np  
min = 1  
max = np.e  
n = 10000  
</code>  
BEGIN SOLUTION  
<code>  
[insert]  
</code>  
END SOLUTION  
<code>  
print(result)  
</code>
```

Reference Solution

```
import scipy.stats  
result = scipy.stats.loguniform.rvs(a = min, b = max, size = n)
```

Automatic Evaluation

Test case 1

```
min = 1  
max = np.e  
n = 10000  
ans = ... # generated by Reference solution
```

Test code

```
np.testing.assert_array_equal(result.shape, ans.shape)  
from scipy.stats import ks_2samp  
# Kolmogorov-Smirnov Test judges whether the two sampled  
# from similar distribution  
assert ks_2samp(result, ans)[0] <= 0.1
```

Surface-form constraints
`for` and `while` should not appear in Syntax Tree

Figure 9: NumPy example problem involving randomness, requiring the use of a specialist knowledge test.

Here is a sample dataframe:

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

I'd like to add inverses of each existing column to the dataframe and name them based on existing column names with a prefix, e.g. `inv_A` is an inverse of column A and so on.

The resulting dataframe should look like so:

```
result = pd.DataFrame([{"A": [1, 2, 3], "B": [4, 5, 6], "inv_A": [1/1, 1/2, 1/3], "inv_B": [1/4, 1/5, 1/6]}])
```

Obviously there are redundant methods like doing this in a loop, but there should exist much more pythonic ways of doing it ... [omitted for brevity]

A:
<code>
import pandas as pd
df = pd.DataFrame([{"A": [1, 2, 3], "B": [4, 5, 6]})
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
print(result)
</code>

Code Context

Problem
Prompt → Language Models (GPT-3 Codex)

Predict

Replace `[insert]` in the code context with following predicted code snippets

```
result = df.div(1).add_prefix("inv_")
```

Execute to evaluate

Multi-criteria Execution-based Evaluation
Test case 1

```
df = pd.DataFrame([{"A": [1, 2, 3], "B": [4, 5, 6]})  
ans = pd.DataFrame([{"A": [1, 2, 3], "B": [4, 5, 6],  
"inv_A": [1/1, 1/2, 1/3],  
"inv_B": [1/4, 1/5, 1/6]})
```

Test case 2

```
df,ans = ...[omit for brevity]  
np.testing.assert_frame_equal(result, ans)
```

Surface-form constraints
`for` and `while` should not appear in Syntax Tree

Correct/wrong?

Experiential Results

Format	Model	Pandas	NumPy	Matplotlib	Scikit-learn	SciPy	TensorFlow	PyTorch	Overall
Left-to-right Completion	Codex-002	26.5	43.1	57.0	44.8	31.8	39.3	41.8	39.2
	Codex-001	9.4	26.6	41.8	18.5	15.0	17.2	9.7	20.2
	Codex-Cushman	7.9	21.8	40.7	18.0	11.3	12.2	12.4	18.1
	CodeGen-6B	1.9	12.1	18.6	5.8	7.4	12.8	3.4	8.4
	InCoder-6B	3.1	4.4	28.3	2.8	2.8	3.8	4.4	7.4
Insertion	Codex-002	30.1	46.5	57.0*	53.7	34.8	53.4	47.7	43.3
	InCoder-6B	2.9	4.6	28.3*	3.1	3.1	7.8	3.2	7.5

pass@1 accuracy with 40 samples generated for each problem

	Pandas	NumPy	Scikit-learn	SciPy	TensorFlow	PyTorch	Overall
Origin _{surface}	37.3	61.2	52.6	33.0	64.9	64.8	53.2
	31.9 <small>-5.4</small>	58.4 <small>-2.8</small>	55.7 <small>+3.1</small>	32.1 <small>-0.9</small>	58.0 <small>-8.9</small>	50.0 <small>-14.8</small>	49.8 <small>-3.4</small>
Origin _{semantic}	36.8	56.7	60.6*	40.3	71.3	65.1	47.2
	33.2 <small>-3.6</small>	49.0 <small>-7.7</small>	38.9* <small>-21.7</small>	34.3 <small>-6.0</small>	42.5 <small>-25.8</small>	30.5 <small>-34.6</small>	38.2 <small>-9.0</small>
Origin _{difficult}	39.9	52.7	5.0*	58.1	73.0*	53.8*	46.8
	17.7 <small>-22.2</small>	27.1 <small>-25.6</small>	0.0* <small>-5.0</small>	13.8 <small>-44.3</small>	38.0* <small>-35.0</small>	28.8* <small>-25.0</small>	21.0 <small>-25.8</small>

The performance of Codex-002 drops after perturbation

Surface Perturbations

- (pharaphrase problems/ code context)
- keep the reference solution same

Semantic Perturbations

- change the semantics of the reference solution
- keep the difficulties same

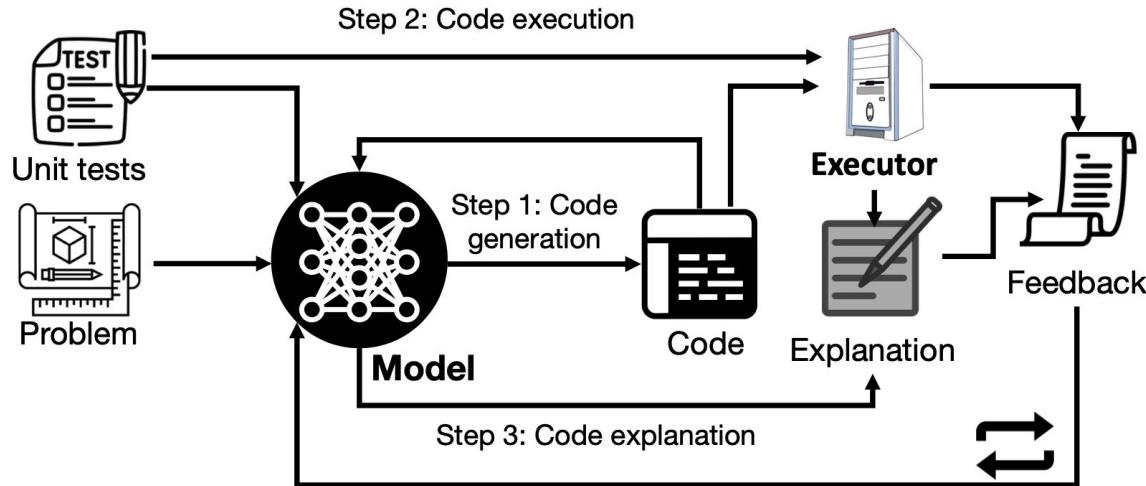
Teaching Large Language Models to Self-Debug

Google DeepMind

Teaching LLMs to Self-Debug

Motivation:

- ✗ Semantic Parsing
- ✓ Find themself Bugs



Teaching LLMs to Self-Debug

Motivation:

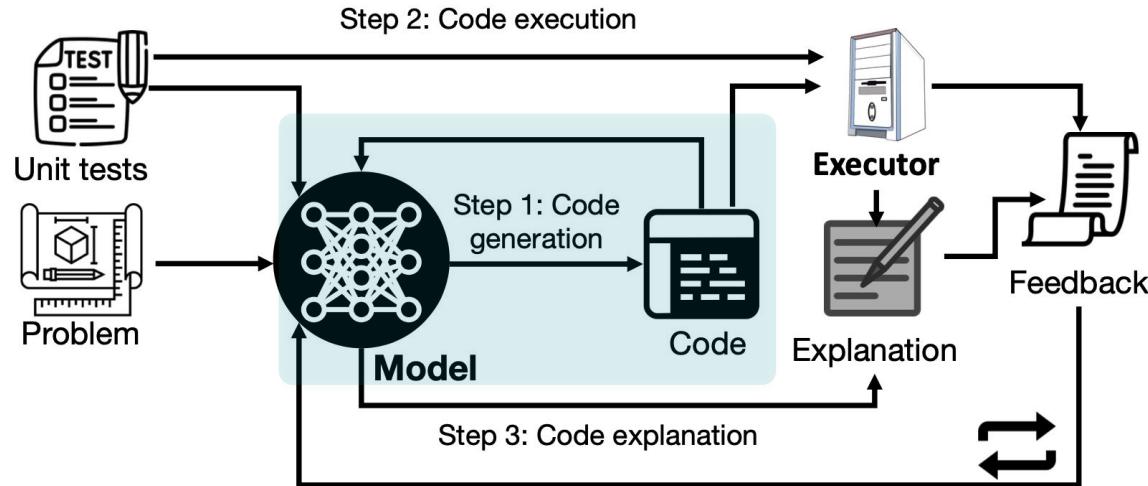
- ✗ Semantic Parsing
- ✓ Find themself Bugs

Framework:



- 🍋 Generation
- 🍋 Explanation
- 🍋 feedback

🍋 Generate the candidate programs



Teaching LLMs to Self-Debug

Motivation:

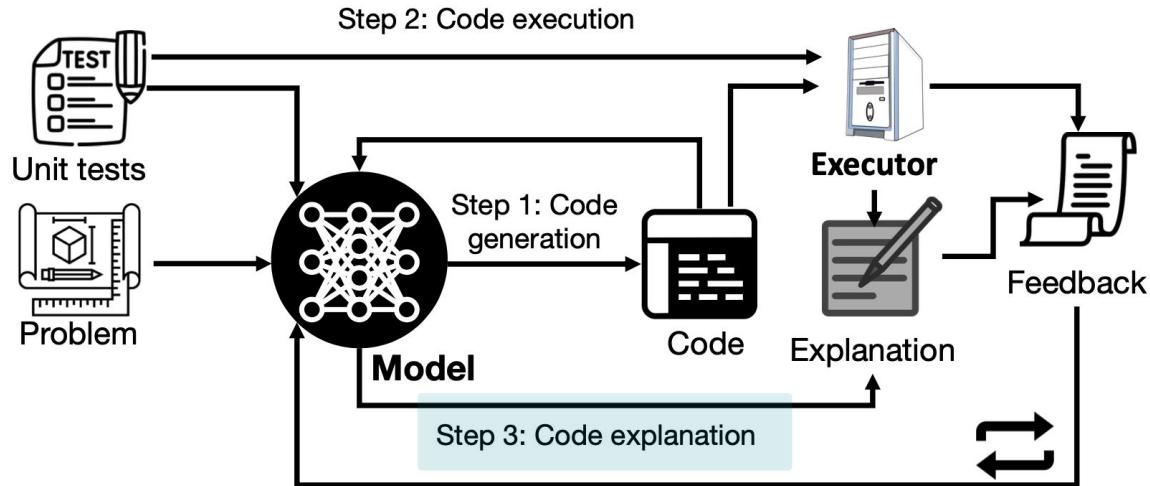
- ✗ Semantic Parsing
- ✓ Find themself Bugs

Framework:

- 💡 Generation
- 💡 Explanation
- 💡 Feedback

💡 Code Explanation:
rubber duck debugging

💡 Execution Trace:
Explain the middle execution steps line by line.



Teaching LLMs to Self-Debug

Motivation:

- ✗ Semantic Parsing
- ✓ Find themself Bugs

Framework:

- 🟡 Generation
- 🟡 Explanation
- 🟡 Feedback

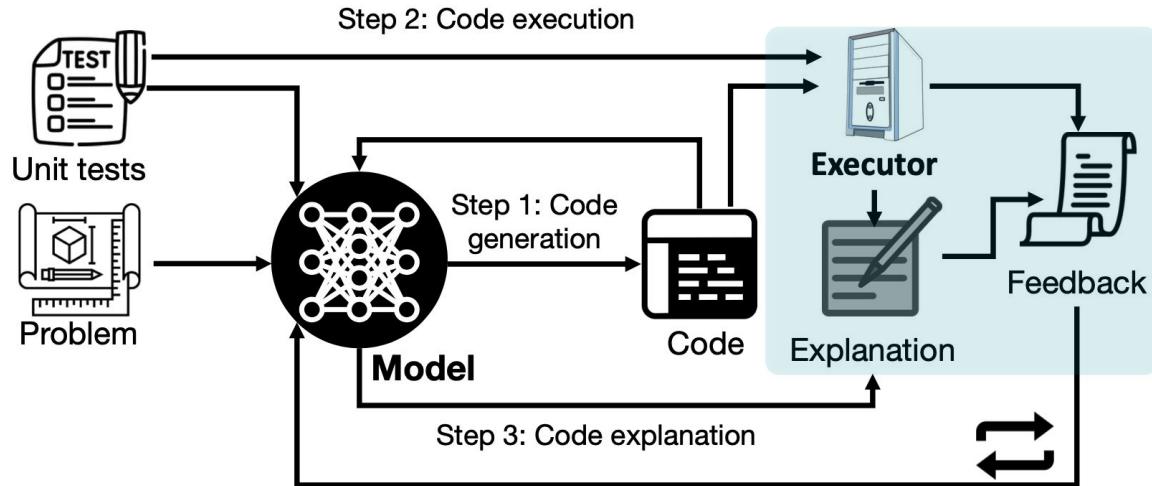


🟡 Simple feedback:

"The SQL is correct!"

🟡 Unit test feedback:

"def test_case1 (val1, val2): ..."



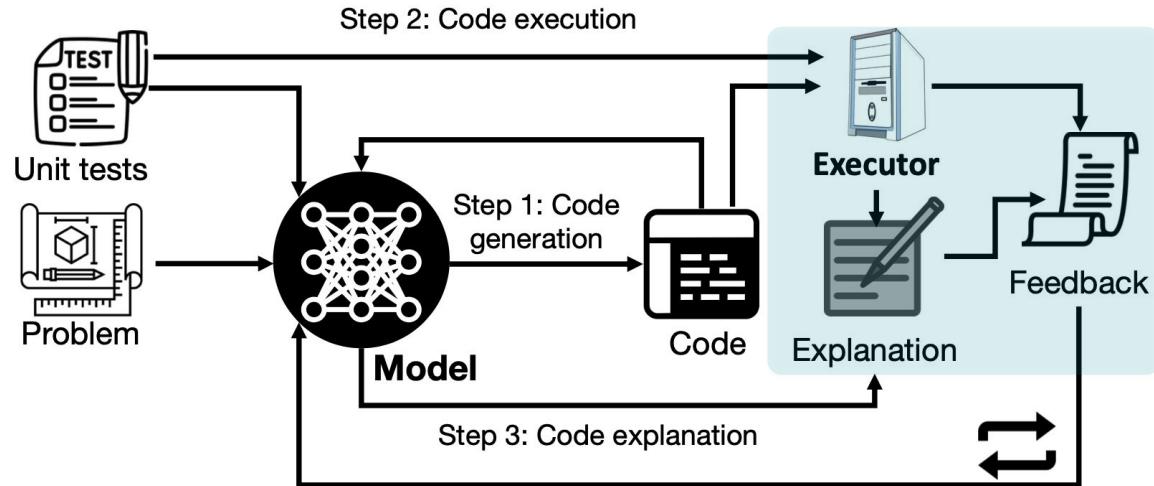
Teaching LLMs to Self-Debug

Motivation:

- ✗ Semantic Parsing
- ✓ Find themself Bugs

Framework:

- 🍋 Generation
- 🍋 Explanation
- 🍋 Feedback



Feedback:

- 🍋 Simple feedback:
"The SQL is correct!"
- 🍋 Unit test feedback:
"def test_case1 (val1, val2): ..."
- 🍋 Code Explanation Feedback:
rubber duck debugging

Execution Trace Feedback:

Explain the middle execution steps line by line.
(simulating)

Note:

In their paper: they state SELF-DEBUG doesn't involve all steps and consider **Explanation** as one form of **Feedbacks**

Teaching LLMs to Self-Debug

Example (Text-to-SQL):

Step 1: Question explanation

Infer the return type of the question.

[Question]

Which customers have both "On Road" and "Shipped" as order status? List the customer names.

[Question Explanation]

"List the customer names" returns 1 column. The question returns the customer names who have both "On Road" and "Shipped" as order status. So the question returns 1 column.

Step 1:

 Prompt the model to summarize the question and infer the return type required by the question

 N.O. of rows...

Teaching LLMs to Self-Debug

Example (Text-to-SQL):

Step 2: Code explanation

Summarize the return type of the execution.

[SQL]

```
SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id  
WHERE orders.order_status = "On Road" OR orders.order_status = "Shipped"
```

Execution:

```
| George |
```

[SQL Explanation]

The execution of the SQL query above would return a table with 1 column.

The first column, "customers.customer_name" would contain the customer names.

With "customers JOIN orders", the table would contain the data about customers with orders.

With "WHERE orders.order_status = 'On Road' OR orders.order_status = 'Shipped'", the table filters the records to only include customers who have order status "On Road" or "Shipped".

So the SQL query returns a table with 1 column, the customer names who have the order status "On Road" or "Shipped".

Step 2:

- execute the SQL query and add the returned table to the model input for **code explanation**.



SQL Explanation contains meanings of SQL



only return 2 rows

Teaching LLMs to Self-Debug

Step 3: Self-Debugging with explanation

[Question]

[Explanation summary]

As in your explanation, the SQL query returns a table with 1 column, the customer names who have the order status "On Road" or "Shipped". The question returns 1 column, the customer names who have both "On Road" and "Shipped" as order status.

[Feedback]

So the SQL prediction above is wrong. Please fix the SQL.

[Revised SQL #1]

```
SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id  
WHERE orders.order_status = "On Road" AND orders.order_status = "Shipped"
```

[Explanation summary #1]

As in your explanation, the SQL query returns an empty table, because the "WHERE" clause contains mutually exclusive conditions. The question returns 1 column, the customer names who have order status of both "On Road" and "Shipped".

[Feedback #1]

So the SQL prediction above is wrong. Please fix the SQL.

[Revised SQL #2]

```
SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id WHERE  
orders.order_status = "On Road" INTERSECT SELECT customers.customer_name FROM customers JOIN orders ON  
customers.customer_id = orders.customer_id WHERE orders.order_status = "Shipped"
```

[Explanation summary #2]

As in your explanation, the SQL query returns a table with 1 column, the customer names who have the order status "On Road" and "Shipped". The question returns 1 column, the customer names who have order status of both "On Road" and "Shipped". So the question returns 1 column.

[Feedback #2]

So the SQL prediction above is correct!

Step 3:



Compare:



SQL explanation



question description



Terminate Until:

The SQL is considered correct; or maximum debugging turns.

Other Tasks:



Code Translation



Text-to-Python

Teaching LLMs to Self-Debug

Experiment:

Models:

- 🟡 Codex: code-davinci-002
- 🟡 GPT-3.5: gpt-3.5-turbo
- 🟡 GPT-4: gpt-4
- 🟡 Starcoder

Datasets:

- 🟡 Spider dev: text-to-SQL
- 🟡 TransCoder: code translation
- 🟡 MBPP: text-to-Python

Main Result:

Table 1: Comparing SELF-DEBUGGING to prior ranking techniques.

(a) Results on the Spider development set.

Spider (Dev)	
<i>w/ training</i>	
T5-3B + N-best Reranking	80.6
LEVER (Ni et al., 2023)	81.9
<i>Prompting only w/o debugging</i>	
Coder-Reviewer	74.5
MBR-Exec	75.2
SELF-DEBUGGING (this work)	
Codex	81.3
+ Expl.	84.1

(b) Results on MBPP dataset.

	<i>n</i> samples
Prior work	
MBR-Exec	63.0 (<i>n</i> = 25)
Reviewer	66.9 (<i>n</i> = 25)
LEVER	68.9 (<i>n</i> = 100)
SELF-DEBUGGING (this work)	
Codex	72.2 (<i>n</i> = 10)
Simple	73.6
UT	75.2
UT + Expl.	75.6

Teaching LLMs to Self-Debug

Experiment:

Models:

- 🟡 Codex: code-davinci-002
- 🟡 GPT-3.5: gpt-3.5-turbo
- 🟡 GPT-4: gpt-4
- 🟡 Starcoder

Datasets:

- 🟡 Spider dev: text-to-SQL
- 🟡 TransCoder: code translation
- 🟡 MBPP: text-to-Python

Main Result:

(a) Results on the Spider development set.

	Spider	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	81.3	71.1	73.2	64.7	
Simple	81.3	72.2	73.4	64.9	
+Expl.	84.1	72.2	73.6	64.9	

(b) Results on TransCoder.

	TransCoder	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	80.4	89.1	77.3	70.0	
Simple	89.3	91.6	80.9	72.9	
UT	91.6	92.7	88.8	76.4	
+ Expl.	92.5	92.7	90.4	76.6	
+ Trace.	87.9	92.3	89.5	73.6	

(c) Results on MBPP.

	MBPP	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	61.4	67.6	72.8	47.2	
Simple	68.2	70.8	78.8	50.6	
UT	69.4	72.2	80.6	52.2	
+ Expl.	69.8	74.2	80.4	52.2	
+ Trace.	70.8	72.8	80.2	53.2	

Observations:

- 🟡 It doesn't work at Spider due to lack of uni test
- 🟡 Unit test can even benefit simple feedbacks

Teaching LLMs to Self-Debug

Prev.:

Step 2: Code explanation

Summarize the return type of the execution.

[SQL]

```
SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id  
WHERE orders.order_status = "On Road" OR orders.order_status = "Shipped"
```

Execution:
| George |

[SQL Explanation]

The execution of the SQL query above would return a table with 1 column.
The first column, "customers.customer_name" would contain the customer names.
With "customers JOIN orders", the table would contain the data about customers with orders.
With "WHERE orders.order_status = 'On Road' OR orders.order_status = 'Shipped'", the table filters the records to only include customers who have order status "On Road" or "Shipped".
So the SQL query returns a table with 1 column, the customer names who have the order status "On Road" or "Shipped".

Step 2:

- 💡 execute the SQL query and add the returned table to the model input for **code explanation**.

- 💡 *SQL Explanation contains meanings of SQL*
- 💡 *only return 2 rows*

Observations:

- 💡 *It doesn't work at Spider since Spider doesn't have unit test*
- 💡 Unit test can even benefit simple feedbacks



- 💡 Whether their conclusions are **conflicting**?
- 💡 If we consider comparison w/ ground truth as one **Unit Test**?

Teaching LLMs to Self-Debug

W/O unit test execution:

(a) Results on Transcoder.

TransCoder	Codex	GPT-3.5	GPT-4
Baseline	80.4	89.1	77.3
Simple	83.4	89.1	78.2
+ Expl.	83.9	89.1	78.0
+ Trace.	83.9	89.1	78.4

(b) Results on MBPP

MBPP	Codex	GPT-3.5	GPT-4
Baseline	61.4	67.6	72.8
Simple	57.6	68.2	76.0
+ Expl.	64.4	68.2	76.0
+ Trace.	66.2	69.2	76.4

Main Result:

(a) Results on the Spider development set.

Spider	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	81.3	71.1	73.2	64.7
Simple	81.3	72.2	73.4	64.9
+ Expl.	84.1	72.2	73.6	64.9

(b) Results on TransCoder.

TransCoder	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	80.4	89.1	77.3	70.0
Simple	89.3	91.6	80.9	72.9
UT	91.6	92.7	88.8	76.4
+ Expl.	92.5	92.7	90.4	76.6
+ Trace.	87.9	92.3	89.5	73.6

(c) Results on MBPP.

MBPP	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	61.4	67.6	72.8	47.2
Simple	68.2	70.8	78.8	50.6
UT	69.4	72.2	80.6	52.2
+ Expl.	69.8	74.2	80.4	52.2
+ Trace.	70.8	72.8	80.2	53.2

Observations:

- 🟡 It doesn't work at Spider due to lack of uni test
- 🟡 Unit test can even benefit simple feedbacks significantly
- 🟡 Code execution env is crucial to SELF-DEBUGGING

Teaching LLMs to Self-Debug

W/O unit test execution:

(a) Results on Transcoder.

TransCoder	Codex	GPT-3.5	GPT-4
Baseline	80.4	89.1	77.3
Simple	83.4	89.1	78.2
+ Expl.	83.9	89.1	78.0
+ Trace.	83.9	89.1	78.4

(b) Results on MBPP

MBPP	Codex	GPT-3.5	GPT-4
Baseline	61.4	67.6	72.8
Simple	57.6	68.2	76.0
+ Expl.	64.4	68.2	76.0
+ Trace.	66.2	69.2	76.4

Main Result:

(a) Results on the Spider development set.

Spider	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	81.3	71.1	73.2	64.7
Simple	81.3	72.2	73.4	64.9
+ Expl.	84.1	72.2	73.6	64.9

(b) Results on TransCoder.

TransCoder	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	80.4	89.1	77.3	70.0
Simple	89.3	91.6	80.9	72.9
UT	91.6	92.7	88.8	76.4
+ Expl.	92.5	92.7	90.4	76.6
+ Trace.	87.9	92.3	89.5	73.6

(c) Results on MBPP.

MBPP	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	61.4	67.6	72.8	47.2
Simple	68.2	70.8	78.8	50.6
UT	69.4	72.2	80.6	52.2
+ Expl.	69.8	74.2	80.4	52.2
+ Trace.	70.8	72.8	80.2	53.2

TakeAway:

- 🟡 When code execution env is available, Uni test feedback is enough
- 🟡 When code execution env is **not** available (private), explanation feedback is beneficial



Code Explanation (rubber duck)



Execution Trace

Code LM for Reasoning

Code As Intermedia Representation

Code LLM for Reasoning

Information Retrieval:

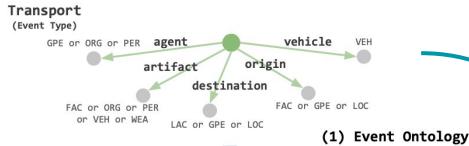


Table QA:

Shirt	Made_in	Sales	Price
Polo	U.S.	500	\$29.8
Luckyland	U.S.	900	\$19.8
Timber Bay	Mexico	400	\$25.9
Office	Turkey	600	\$31.8

Which is the best-selling shirt made in North America and with no chemicals?

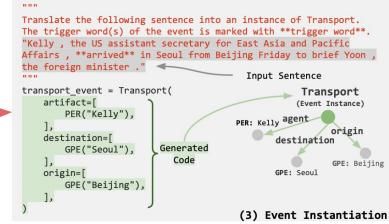
Embodied AI:



grab_and_puton('banana', 'plate')

Math:

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?



```
SELECT Shirt FROM T WHERE  
fCodex("No chemicals?";Shirt)='yes' AND  
fCodex("North America?";Made_in)='yes'  
ORDER BY Sales DESC LIMIT 1
```

Prompt

```
def __init__(self, import_grab_and_putin <obj>=obj, grab_and_putin <obj>=obj, switchoff <obj>, open <obj>, ...):  
    self.import_grab_and_putin = import_grab_and_putin  
    self.grab_and_putin = grab_and_putin  
    self.switchoff = switchoff  
    self.open = open  
    ...  
  
def throw_away_banana():  
    objects = ['banana', 'garbage can', ...]  
    # 1: put banana in garbage can  
    grab_and_putin('banana', 'garbagecan').  
    # 2: Done  
  
def put_fork_and_spoon_on_the_box():  
    objects = ['fork', 'spoon', 'box', ...]  
    ...  
  
def put_fork_on_plate_and_spoon_in_box():  
    objects = ['fork', 'spoon', 'box', ...]  
    ...  
  
def sort_fruits_on_plate_and_bottles_in_box():  
    objects = ['banana', 'bottle', 'box', 'plate', 'table', 'drill', 'strawberry']  
    ...  
  
# 1: put banana on plate  
grab_and_putin('banana', 'plate')  
# 2: put strawberry on plate  
grab_and_putin('strawberry', 'plate')  
# 3: put bottle in box  
grab_and_putin('bottle', 'box')  
# 4: Done  
->grab_and_puton('strawberry', 'plate')
```

LLM [GPT-3]

Model Output

```
A: The bakers started with 200 loaves  
loaves_baked = 200  
They sold 93 in the morning and 39 in the afternoon  
loaves_sold_morning = 93  
loaves_sold_afternoon = 39  
The grocery store returned 6 loaves.  
loaves_returned = 6  
The answer is  
answer = loaves_baked - loaves_sold_morning  
- loaves_sold_afternoon + loaves_returned  
>>> print(answer)
```

74

Code LLM for Reasoning

Information Retrieval:

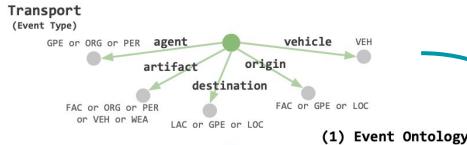


Table QA:

Shirt	Made_in	Sales	Price
Polo	U.S.	500	\$29.8
Luckyland	U.S.	900	\$19.8
Timber Bay	Mexico	400	\$25.9
Office	Turkey	600	\$31.8

Which is the best-selling shirt made in North America and with no chemicals?

Embodied AI:



grab_and_puton('banana', 'plate')

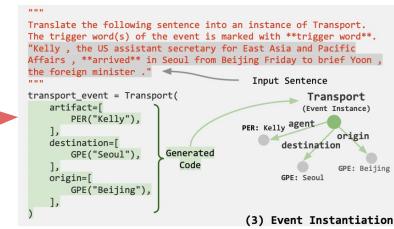
Math:

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

Define the problem as semantic parsing

The code prompt could be **executable** or **symbolic**

Symbolic:



Exec + T:

```

SELECT Shirt FROM T WHERE
  fCodex("No chemicals?",Shirt)='yes' AND
  fCodex("North America?",Made_in)='yes'
  ORDER BY Sales DESC LIMIT 1
  
```

Exec + T:

```

Prompt: instructions import grab_and_puton <obj>-><obj>;
grab_and_puton(<obj>,<obj>); switchoff <obj>; open <obj>; ...
def throw_away_banana():
  objects = ['banana', 'garbage can',...]
  # 1: put banana in garbage can
  grab_and_puton('banana', 'garbagecan');
  # 2: Done

def put_fork_and_spoon_on_the_box():
  objects = ['fork', 'spoon', 'knife',...]
  ...
def put_fork_on_plate_and_spoon_in_box():

def sort_fruits_on_plate_and_bottles_in_box():
  objects = ['banana', 'bottle', 'box', 'plate', 'table', 'drill', 'strawberry']

LLM [GPT-3]
# 1: put banana on plate
grab_and_puton('banana', 'plate')
# 2: put strawberry on plate
grab_and_puton('strawberry', 'plate')
# 3: put bottle in box
grab_and_puton('bottle', 'box')
# 4: Done
  
```

Exec + T:

Model Output

```

A: The bakers started with 200 loaves
loaves_baked = 200
They sold 93 in the morning and 39 in the afternoon
loaves_sold_morning = 93
loaves_sold_afternoon = 39
The grocery store returned 6 loaves.
loaves_returned = 6
The answer is
answer = loaves_baked - loaves_sold_morning
- loaves_sold_afternoon + loaves_returned
>>> print(answer)
74
  
```

Code LLM for Reasoning:



When Use Code LLM for Reasoning:

- Your data structure is hierarchical

inheritance class: Movement → Event

```
class Movement(Event): # Inherit from `Event` class
    ... # omitted for space
    class Transport(Movement):
        """
        self.agent transported self.artifact in self.vehicle vehicle from
        self.origin place to self.destination place.
        """
        def __init__(self,
                     agent: List[GPE | ORG | PER] = [],
                     artifact: List[FAC | ORG | PER | VEH | WEA] = [],
                     destination: List[FAC | GPE | LOC] = [],
                     origin: List[FAC | GPE | LOC] = [],
                     vehicle: List[VEH] = []):
            self.agent = agent
            self.artifact = artifact
            self.destination = destination
            self.origin = origin
            self.vehicle = vehicle
```

The code illustrates the use of inheritance and annotations in Python. The `Movement` class inherits from the `Event` class. The `Transport` class is a subclass of `Movement`. The `__init__` method takes several lists of entity types as parameters: `agent` (GPE, ORG, PER), `artifact` (FAC, ORG, PER, VEH, WEA), `destination` (FAC, GPE, LOC), `origin` (FAC, GPE, LOC), and `vehicle` (VEH). Annotations like `List[GPE | ORG | PER]` are labeled as "Entity Type Annotation". The `Hierarchical Ontology` is shown as a separate block, and the `Event template` is also indicated.

Code LLM for Reasoning:



When Use Code LLM for Reasoning:

- ⌚ Your data structure is hierarchical

inheritance class: Movement → Event

```
class Movement(Event): # Inherit from `Event` class
... # omitted for space
class Transport(Movement): # Hierarchical Ontology
    ...
        self.agent transported self.artifact in self.vehicle vehicle from
        self.origin place to self.destination place.
    ...
    def __init__(self,
        agent: List[GPE | ORG | PER] = [],
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        destination: List[FAC | GPE | LOC] = [],
        origin: List[FAC | GPE | LOC] = [],
        vehicle: List[VEH] = []
    ):
        self.agent = agent
        self.artifact = artifact
        self.destination = destination
        self.origin = origin
        self.vehicle = vehicle
```

Event template

Entity Type Annotation

- ⌚ Your tasks involves a structural plan, or tool calls

yet-to-def: get_mean()

```
def get_mean(xs: List[float]) -> float:
    """Compute the mean of a list of numbers."""
    return sum(xs) / len(xs)
# end of function

def get_abs_diff_between_means(xs0: List[float],
                               xs1: List[float]) -> float:
    """Get the absolute difference between the means of two
    lists of numbers."""
    m0 = get_mean(xs0)
    m1 = get_mean(xs1)
    return abs(m0 - m1)
```

fcodex()

```
SELECT Shirt FROM T WHERE
fcodex("No chemicals?";Shirt)=‘yes’ AND
fcodex("North America?";Made_in)=‘yes’
ORDER BY Sales DESC LIMIT 1
```

math

Model Output

A: The bakers started with 200 loaves
loaves_baked = 200
They sold 93 in the morning and 39 in the afternoon
loaves_sold_morning = 93
loaves_sold_afternoon = 39
The grocery store returned 6 loaves.
loaves_returned = 6

The answer is

```
answer = loaves_baked - loaves_sold_morning
- loaves_sold_afternoon + loaves_returned
```

>>> print(answer)

74



Code LLM for Reasoning:

💡 Why Use Code LLM for Reasoning:

- 💡 Code tends to enforce constraints

inheritance class: Movement → Event

```
class Movement(Event): # Inherit from `Event` class
    ... # omitted for space
    class Transport(Movement): # Hierarchical Ontology
        ...
        self.agent transported self.artifact in self.vehicle vehicle from
        self.origin place to self.destination place.
        ...
    def __init__(
        self,
        agent: List[GPE | ORG | PER] = [], # Entity Type Annotation
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        destination: List[FAC | GPE | LOC] = [],
        origin: List[FAC | GPE | LOC] = [],
        vehicle: List[VEH] = [],
    ):
        self.agent = agent
        self.artifact = artifact
        self.destination = destination
        self.origin = origin
        self.vehicle = vehicle
```

Code LLM for Reasoning:

🟡 Why Use Code LLM for Reasoning:

- 🟡 Code tends to enforce constraints

inheritance class: Movement → Event

```
class Movement(Event): # Inherit from `Event` class
    ... # omitted for space
    class Transport(Movement): # Hierarchical Ontology
        ...
        self.agent transported self.artifact in self.vehicle vehicle from
        self.origin place to self.destination place.
        ...
        def __init__(
            self,
            agent: List[GPE | ORG | PER] = [], # Entity Type Annotation
            artifact: List[FAC | ORG | PER | VEH | WEA] = [],
            destination: List[FAC | GPE | LOC] = [],
            origin: List[FAC | GPE | LOC] = [],
            vehicle: List[VEH] = [],
        ):
            self.agent = agent
            self.artifact = artifact
            self.destination = destination
            self.origin = origin
            self.vehicle = vehicle
```

- 🟡 Code prompt can reduce hallucination

Import action primitives

```
from actions import walk <obj>,
grab <obj>, switchon <obj>,
open <obj>, standup, find
<obj>, putin <obj> <obj>, ...
```

Code LLM for Reasoning:



Why Use Code LLM for Reasoning:

- Code tends to enforce constraints

Inheritance class: Movement → Event

```
class Movement(Event): # Inherit from `Event` class
    ... # omitted for space
    class Transport(Movement): # Hierarchical Ontology
        ...
        self.agent transported self.artifact in self.vehicle vehicle from
        self.origin place to self.destination place.
        ...
        def __init__(self,
                    agent: List[GPE | ORG | PER] = [],
                    artifact: List[FAC | ORG | PER | VEH | WEA] = [],
                    destination: List[FAC | GPE | LOC] = [],
                    origin: List[FAC | GPE | LOC] = [],
                    vehicle: List[VEH] = []):
            self.agent = agent
            self.artifact = artifact
            self.destination = destination
            self.origin = origin
            self.vehicle = vehicle
```

- Code prompt can reduce hallucination

Import action primitives

```
from actions import walk <obj>,
grab <obj>, switchon <obj>,
open <obj>, standup, find
<obj>, putin <obj> <obj>, ...
```



Suitable for tool calls w/o impacting perplexity of prompts

The New England Journal of Medicine is a registered trademark of [QA("Who is the publisher of The New England Journal of Medicine?") → Massachusetts Medical Society] the MMS.

Out of 1400 participants, 400 (or [Calculator(400 / 1400) → 0.29] 29%) passed the test.

The name derives from "la tortuga", the Spanish word for [MT("tortuga") → turtle] turtle.

The Brown Act is California's law [WikiSearch("Brown Act") → The Ralph M. Brown Act is an act of the California State Legislature that guarantees the public's right to attend and participate in meetings of local legislative bodies.] that requires legislative bodies, like city councils, to hold their meetings open to the public.

PROMPT for Planning

Import action primitives

```
from actions import walk <obj>,
grab <obj>, switchon <obj>,
open <obj>, standup, find
<obj>, putin <obj> <obj>, ...
```

Available objects list

```
objects=[clothesshirt' sink,
pie, apple, fridge, garbagecan,
tv, dishwashingliquid, bed,
bookshelf, salmon, stove, plate,
coffeepot, kitchentable,
wineglass, paper, microwave,
toothbrush, toothpaste,
bathroomcabinet, kitchen, lime,
painkillers, barsoap, ...]
```

Example task(s)

```
def throw_away_lime():
# 0: find lime
find('lime')
...
# 5: close garbagecan
assert('close' to 'garbagecan')
else: find('garbagecan')
assert('garbagecan' is 'opened')
else: open('garbagecan')
close('garbagecan')
...
# 6: Done
```

Next task prompt

```
def microwave_salmon():
```

Too many tool calls will impact perplexity, but not in code prompt !!!

Code LLM for Reasoning:

💡 Tips when Using Code LLM for Reasoning:

💡 Code W/ notations

```
"""
Translate the following sentence into an instance of
Transport. The trigger word(s) of the event is marked
with **trigger word**.           Trigger Marking
"Kelly , the US assistant secretary for East Asia and
Pacific Affairs , **arrived** in Seoul from Beijing
Friday to brief Yoon , the foreign minister ."
"""
transport_event = Transport()
```

```
def throw_away_lime():
    # 0: find lime
    find('lime')
    ...
    # 5: close garbagecan
    assert('close' to 'garbagecan')
    else: find('garbagecan')
    assert('garbagecan' is 'opened')
    else: open('garbagecan')
    close('garbagecan')
    ...
    # 6: Done
```

💡 Code models:

- 💡 Open-source models: Code-llama, Starcoder, etc
- 💡 Closed-source models: code-davinci-002, text-davinci-003, gpt-3.5-turbo
gpt-4, gpt-4-32k (**completion model**) (*chatcompletion may lead to more hallucination*)

Summarization

Summarization:

CodeLMs

💡 Close-Sourced and Open-Sourced CodeLMs

Closed-Source

- GPT-C (2020)
- Codex (2021)
- LaMDA (2022)
- PaLM-Coder (2022)
- PanGu-Coder (2022)
- AlphaCode (2022b)
-

Open-Source

- GPT-NEO (2021)
- GPT-J (2021)
- CodeGen (2023)
- InCoder (2023)
- StarCoder(2023)
- Code Llama(2023)
-

💡 What makes LMs successful in NL2Code?

- 🟡 Large Size
- 🟡 Premium Data
- 🟡 Expert Tuning

Summarization:

Datasets and Benchmarks:

High-quality data will lead development of CodeLM:

Large-Scale Unlabelled Datasets

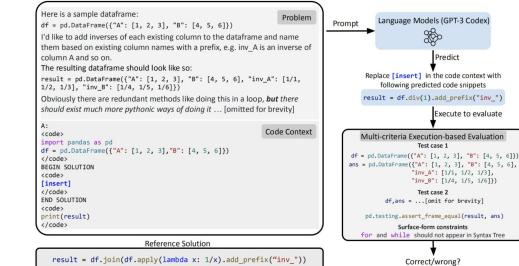
- CodeParrot's corpus (2021) 😊
- GitHub-Code (2021) 😊
- Stack (2022) 😊
- JulCe (2019) W
- APPS (2021) 🎯
- CodeNet (2021) IBM
- BigQuery (2016) G

Benchmarks

- Human Eval (2021) 🤖
- MBPP (2021) G
- DS-1000 (2022) 🏆
- MultiPL-E (2021) Microsoft
-

BigQuery

Cloud data warehouse to power your data-driven innovation



Why and Wen to use CodeLLM for Reasoning?:

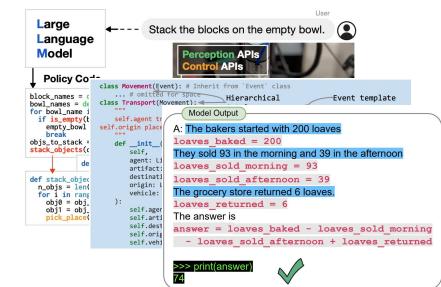
Motivations:

Why Use CodeLM for Reasoning?

- Enforce Constraints
- Reduce Hallucination
- Suitable for Tool Calls

When use CodeLM for Reasoning?

- Data is Hierarchical
- Tasks involve Structural Plans
- Tasks need Many Tool API Calls



Summarization:

💡 How to Enhance Code Generation Capability:

- 🟡 Boosting Semantic Parsing Capability In One Turn
- 🟡 Teach LLMs to Self-Debugging

Summarization:

💡 How to Enhance Code Generation Capability:

- 🟡 Boosting Semantic Parsing Capability In One Turn
- 🟡 Teach LLMs to Self-Debugging
 - When executable env is available: unit test feedback
 - When executable env is private: explanation feedback (rubber duck, exec trace)

Summarization:

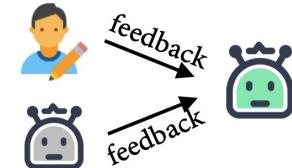


How to Enhance Code Generation Capability:

- 🟡 Boosting Semantic Parsing Capability In One Turn
- 🟡 Teach LLMs to Self-Debugging
 - When executable env is available: unit test feedback
 - When executable env is private: explanation feedback (rubber duck, exec trace)
- 🟡 Balance the Text and Code, then Interactively Optimize Code via Feedback

Follow Feedback

Understand complex feedback from human / agents and convert them into symbolic executable sequences



Summarization:

💡 How to Enhance Code Generation Capability:

- 🟡 Boosting Semantic Parsing Capability In One Turn
- 🟡 Teach LLMs to Self-Debugging
 - When executable env is available: unit test **feedback**
 - When executable env is private: explanation **feedback** (rubber duck, exec trace)
- 🟡 Balance the Text and Code, then Interactively Optimize Code via **Feedback**

💡 Open Questions:

- 🟡 How to test real semantic parsing capabilities of CodeLMs out of memorization?

Follow Feedback

Understand complex feedback from human / agents and convert them into symbolic executable sequences



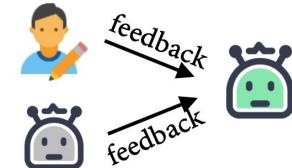
Summarization:

💡 How to Enhance Code Generation Capability:

- 🟡 Boosting Semantic Parsing Capability In One Turn
- 🟡 Teach LLMs to Self-Debugging
 - When executable env is available: unit test **feedback**
 - When executable env is private: explanation **feedback** (rubber duck, exec trace)
- 🟡 Balance the Text and Code, then Interactively Optimize Code via **Feedback**

Follow Feedback

Understand complex feedback from human / agents and convert them into symbolic executable sequences



💡 Open Questions:

- 🟡 How to test real semantic parsing capabilities of CodeLMs out of memorization?
- 🟡 Do you trust that the model can self-debug by themselves?
Which group do you support?

TEACHING LARGE LANGUAGE MODELS TO SELF-DEBUG

Xinyun Chen¹ Maxwell Lin² Nathanael Schärlí¹ Denny Zhou¹
¹Google DeepMind ²UC Berkeley
{xinyunchen,schaerli,dennyzhou}@google.com, mxlin@berkeley.edu

LARGE LANGUAGE MODELS CANNOT SELF-CORRECT REASONING YET

Jie Huang^{1,2,*} Xinyun Chen¹ Swaroop Mishra¹ Huaixiu Steven Zheng¹ Adams Wei Yu¹
Xinying Song¹ Denny Zhou¹
¹Google DeepMind ²University of Illinois at Urbana-Champaign

Summarization:

💡 How to Enhance Code Generation Capability:

- 🟡 Boosting Semantic Parsing Capability In One Turn
- 🟡 Teach LLMs to Self-Debugging
 - When executable env is available: unit test **feedback**
 - When executable env is private: explanation **feedback** (rubber duck, exec trace)
- 🟡 Balance the Text and Code, then Interactively Optimize Code via **Feedback**

Follow Feedback

Understand complex feedback from human / agents and convert them into symbolic executable sequences



💡 Open Questions:

- 🟡 How to test real semantic parsing capabilities of CodeLMs out of memorization?
- 🟡 Do you trust the model can self-debug by themselves?
- 🟡 Can Code Prompt be applied in your domain? *Symbolic or Exec + T?*

TEACHING LARGE LANGUAGE MODELS TO SELF-DEBUG

Xinyun Chen¹ Maxwell Lin² Nathanael Schärlí¹ Denny Zhou¹
¹Google DeepMind ²UC Berkeley
{xinyunchen,schaerli,dennyzhou}@google.com, mxlin@berkeley.edu

LARGE LANGUAGE MODELS CANNOT SELF-CORRECT REASONING YET

Jie Huang^{1,2,*} Xinyun Chen¹ Swaroop Mishra¹ Huaixiu Steven Zheng¹ Adams Wei Yu¹
Xinying Song¹ Denny Zhou¹
¹Google DeepMind ²University of Illinois at Urbana-Champaign

References:

- [1] Code as Policies: Language Model Programs for Embodied Control
- [2] CODE4STRUCT: Code Generation for Few-Shot Event Structure Prediction
- [3] PROGPROMPT: Generating Situated Robot Task Plans using Large Language Models
- [4] Binding Language Models in Symbolic Languages
- [5] PAL: Program-aided Language Models
- [6] Lemur: Harmonizing Natural Language and Code for Language Agents
- [7] Toolformer: Language Models Can Teach Themselves to Use Tools

Thank You For your Time

Have a nice weekend