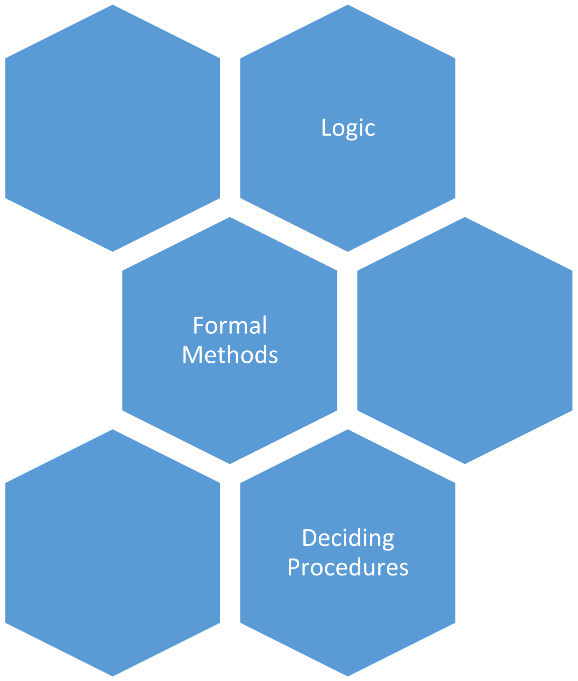
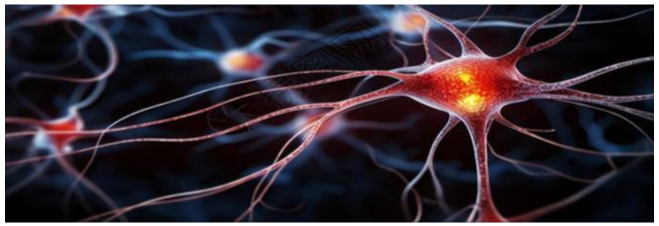
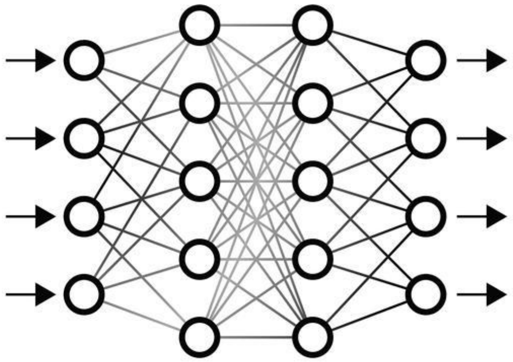
Neural Code Generation Models with Programming Language Knowledge

Yingfei Xiong
Peking University

Two Approaches to AI



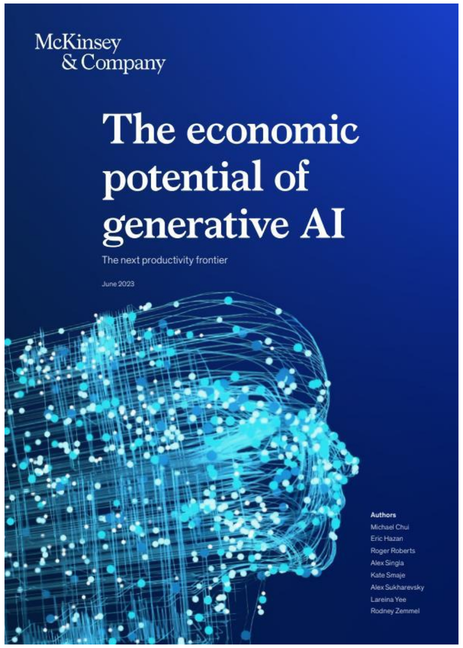




Symbolism
Learn from humans

Connectionism
Learn from human creators

Important LLM Application Assisting Software Development

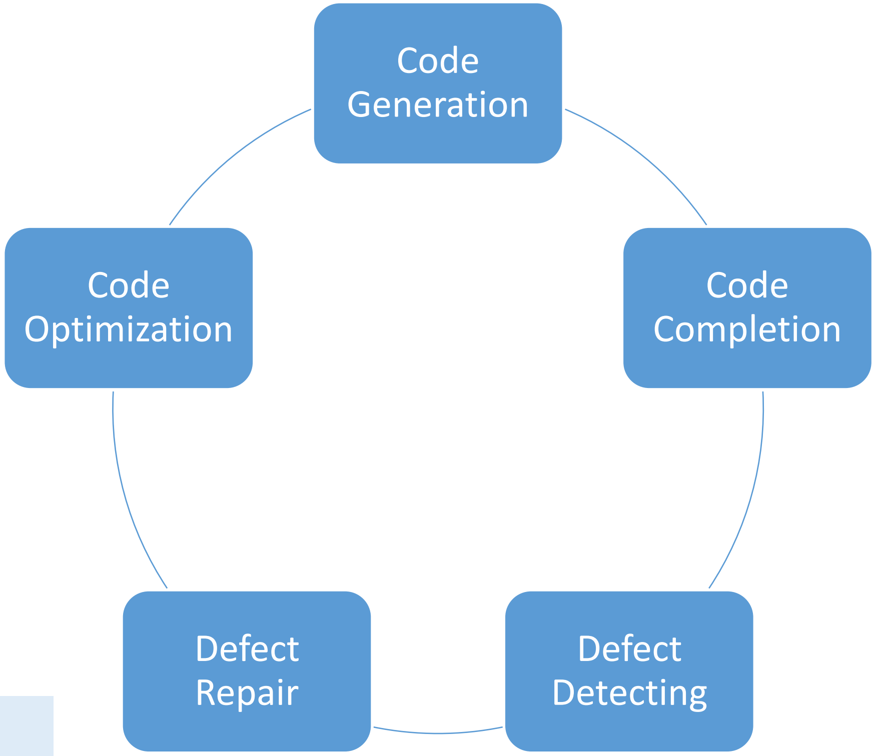

McKinsey & Company

The economic potential of generative AI

The next productivity frontier

June 2023

Authors
Michael Chui
Eric Hazan
Roger Roberts
Alex Singla
Kate Smaje
Alex Sukharevsky
Lareina Yee
Rodney Zemmel



McKinsey: save 20%-45% development cost

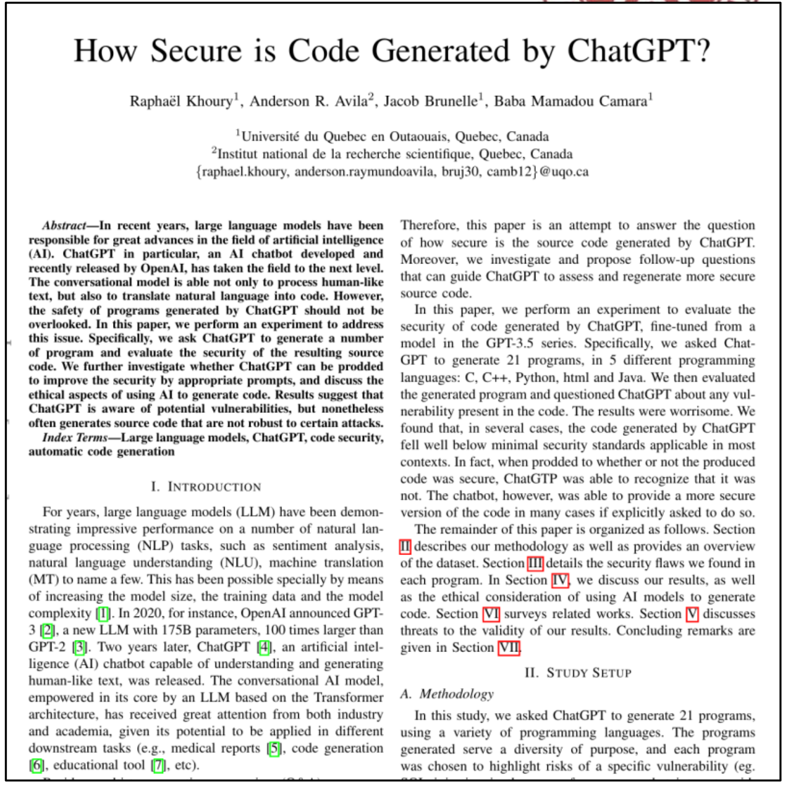
文心快码 Baidu Comate　通义灵码　deepseek　Copilot

Mismatch: connectionism and programs

Programs are symbolic product, with symbolic knowledge:

- Syntactic: ()+5(illegal
- Type: 1+true illegal
- Semantic: use without initialization illegal

These symbolic knowledges are difficult to be learned by neural network



How Secure is Code Generated by ChatGPT?

Raphaël Khoury[1], Anderson R. Avila[2], Jacob Brunelle[1], Baba Mamadou Camara[1]

[1]Université du Quebec en Outaouais, Quebec, Canada
[2]Institut national de la recherche scientifique, Quebec, Canada
{raphael.khoury, anderson.raymundoavila, bruj30, camb12}@uqo.ca

***Abstract*—In recent years, large language models have been responsible for great advances in the field of artificial intelligence (AI). ChatGPT in particular, an AI chatbot developed and recently released by OpenAI, has taken the field to the next level. The conversational model is able not only to process human-like text, but also to translate natural language into code. However, the safety of programs generated by ChatGPT should not be overlooked. In this paper, we perform an experiment to address this issue. Specifically, we ask ChatGPT to generate a number of program and evaluate the security of the resulting source code. We further investigate whether ChatGPT can be prodded to improve the security by appropriate prompts, and discuss the ethical aspects of using AI to generate code. Results suggest that ChatGPT is aware of potential vulnerabilities, but nonetheless often generates source code that are not robust to certain attacks.**

***Index Terms*—Large language models, ChatGPT, code security, automatic code generation**

I. Introduction

For years, large language models (LLM) have been demonstrating impressive performance on a number of natural language processing (NLP) tasks, such as sentiment analysis, natural language understanding (NLU), machine translation (MT) to name a few. This has been possible specially by means of increasing the model size, the training data and the model complexity [1]. In 2020, for instance, OpenAI announced GPT-3 [2], a new LLM with 175B parameters, 100 times larger than GPT-2 [3]. Two years later, ChatGPT [4], an artificial intelligence (AI) chatbot capable of understanding and generating human-like text, was released. The conversational AI model, empowered in its core by an LLM based on the Transformer architecture, has received great attention from both industry and academia, given its potential to be applied in different downstream tasks (e.g., medical reports [5], code generation [6], educational tool [7], etc).

Therefore, this paper is an attempt to answer the question of how secure is the source code generated by ChatGPT. Moreover, we investigate and propose follow-up questions that can guide ChatGPT to assess and regenerate more secure source code.

In this paper, we perform an experiment to evaluate the security of code generated by ChatGPT, fine-tuned from a model in the GPT-3.5 series. Specifically, we asked ChatGPT to generate 21 programs, in 5 different programming languages: C, C++, Python, html and Java. We then evaluated the generated program and questioned ChatGPT about any vulnerability present in the code. The results were worrisome. We found that, in several cases, the code generated by ChatGPT fell well below minimal security standards applicable in most contexts. In fact, when prodded to whether or not the produced code was secure, ChatGTP was able to recognize that it was not. The chatbot, however, was able to provide a more secure version of the code in many cases if explicitly asked to do so.

The remainder of this paper is organized as follows. Section II describes our methodology as well as provides an overview of the dataset. Section III details the security flaws we found in each program. In Section IV, we discuss our results, as well as the ethical consideration of using AI models to generate code. Section VI surveys related works. Section V discusses threats to the validity of our results. Concluding remarks are given in Section VII.

II. Study Setup

A. Methodology

In this study, we asked ChatGPT to generate 21 programs, using a variety of programming languages. The programs generated serve a diversity of purpose, and each program was chosen to highlight risks of a specific vulnerability (eg.

76% of the programs generated by GPT contain vulnerabilities

Symbolic PL knowledge is useful

- Correct code generation requires symbolic PL knowledges

```
bool and(bool a, bool b) {
  ________________
}
```

assignment is common and should be used

NN that does not know types

all parameters are Boolean so 'if' is more likely

NN that knows types

Can we guide neural network to learn PL knowledge?

Overview

TreeGen [AAAI20]

- Using transformer to implement L2S
- The first transformer-based code generator
- SOTA 35m model

Grape [IJCAI22]

- Guiding NN to learn grammar rule definitions

Tare [ICSE23]

- Guide NN to learn typing rules

GrammarT5 [ICSE24]

- Applying L2S to pretraining
- SOTA 0.2B model at that time

DeepSeek-Coder [arxiv]

- Guiding NN to learn Def-Use relations
- SOTA open-source code model at that time

GrammarCoder [ACL25-Finding]

- Applying L2S to Decoder-only
- SOTA 1.5B code model

Implements

L2S Framework [TOSEM22]

- Representing code as grammar rule sequences
- Ensuring syntactic correctness
- Allowing easy symbolic analysis

Apply

ACS [ICSE17]

- First program repair approach whose precision > 70%

Recoder [FSE23]

- First neural program repair approach outperforming traditional approaches

ET[APRCOMP24]

- 1st Place in APR-COMP

OCoR [ASE20]

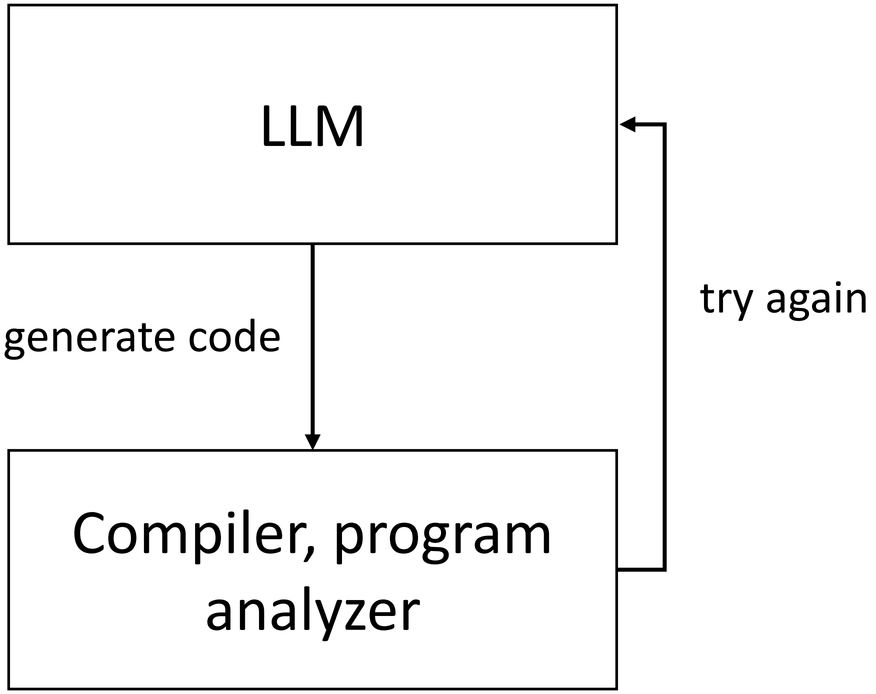
- Code search engine significantly outperforming existing ones

LEAM [ASE22]

- Mutation generation engine that significantly outperforming existing ones

Generating only safe code

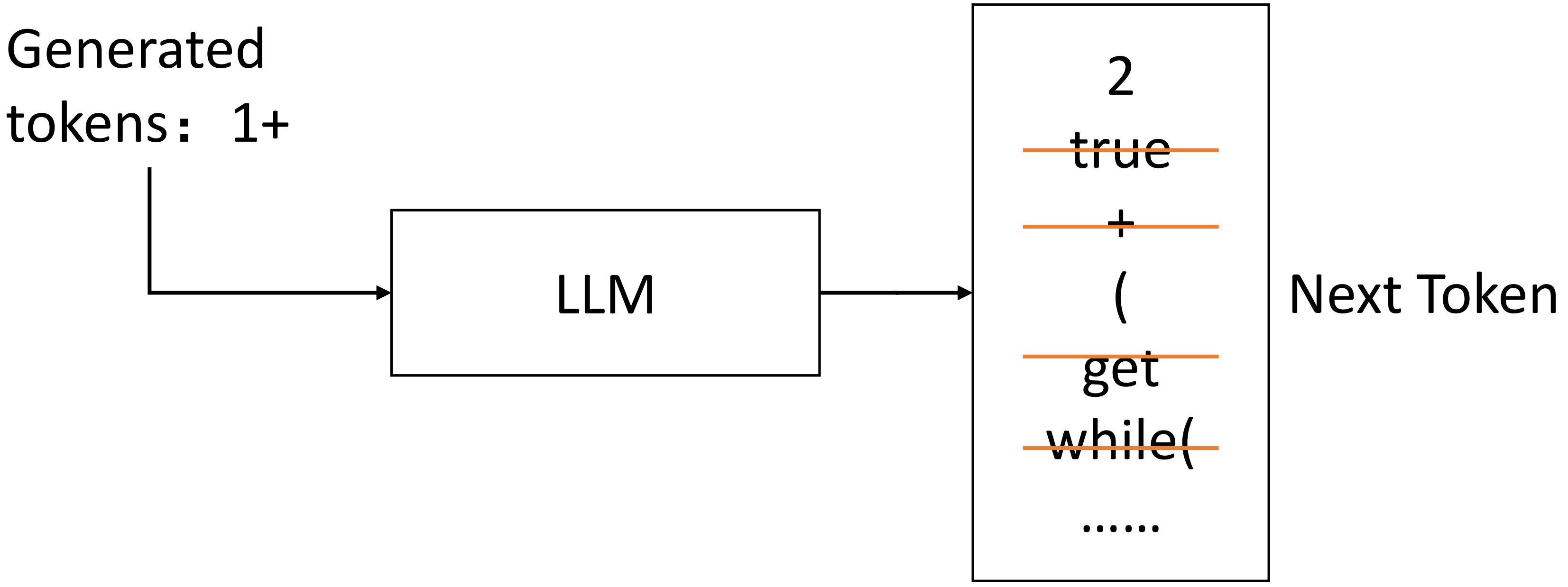
- Attempt 1: Check after generation



Inefficient, may keep generating code with the same fault

Generating only safe code

- Attempt 2: constrained decoding

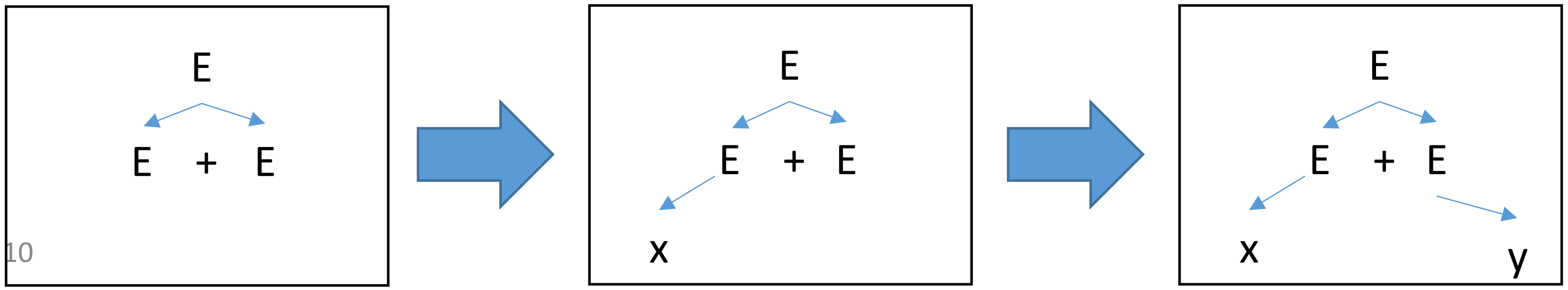


Difficulty： Generated tokens are defined by the BPE algorithm. Lexical analysis is already difficult, let alone parsing.

L2S: representing by grammar rule sequences [TOSEM22]

Program	x+y
Token Sequence	x, +, y
Grammar Rule Sequence	r_1, r_2, r_3

$$r_1: E \rightarrow E + E$$
$$r_2: E \rightarrow x$$
$$r_3: E \rightarrow y$$



Benefit: Constrained Decoding Made Easy

- Ensuring syntactic correctness is trivial
- Type and semantic analysis can be performed on partial AST
  - 1 + “x” + Expr   //Type incorrect
  - if (BoolExpr) then x else x //Semantically incorrect for x=1 → ret=2

- Step:
  - Pre-analysis on grammar rules: get all possibilities for a non-terminal
  - Abstract interpretation on partial program

Benefit: Better Alignment with Semantics [ACL25-Finding]

- Same semantics

    -
      ```
      if (x<0) y=y+1;
      ```

    -
      ```
      if (x < 0) {
         y = y + 1;
      }
      ```

Similar in Grammar Rule Representation
Different in Token Representation

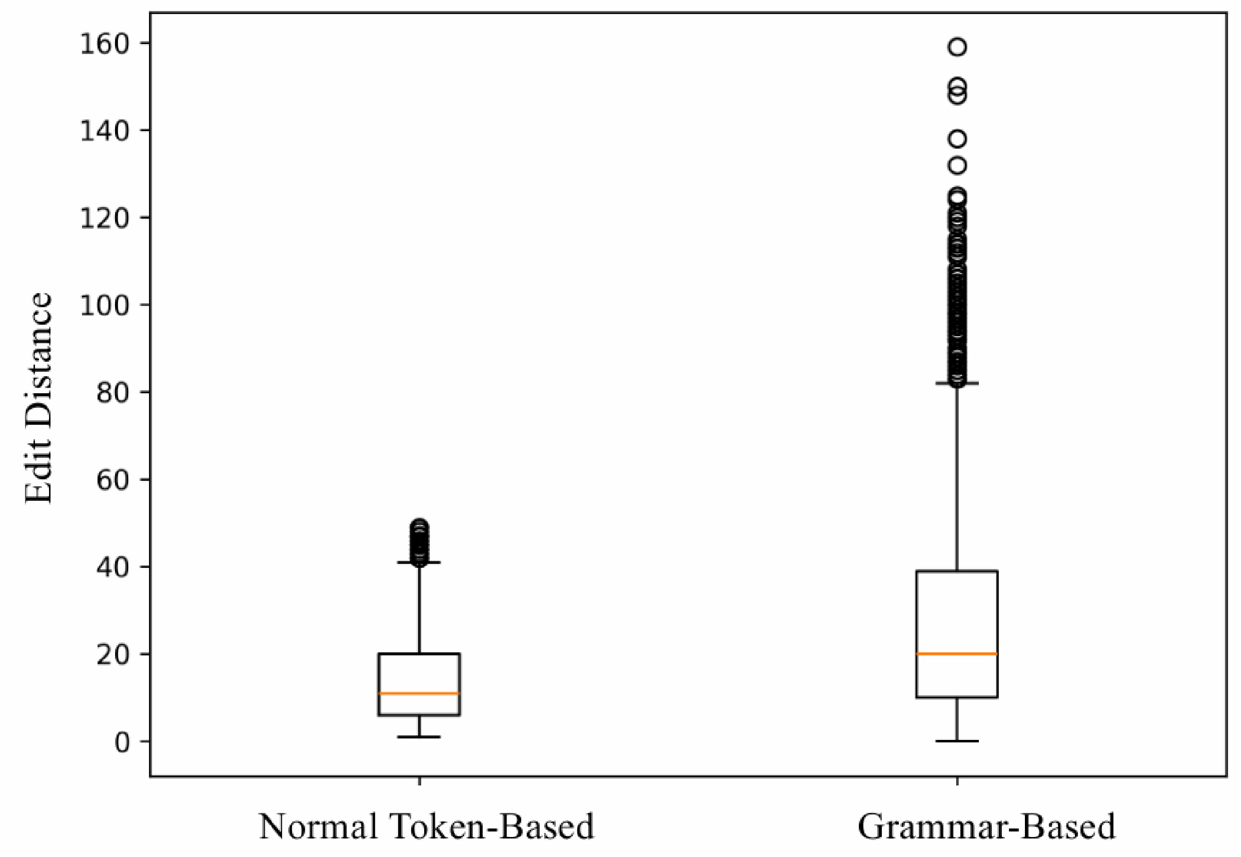
- Different semantics

    -
      ```
      for i in range(1, 6):
        x =  x + 1
        sum = sum + x
      ```

    -
      ```
      for i in range(1, 6):
        x = x + 1
      sum = sum + x
      ```

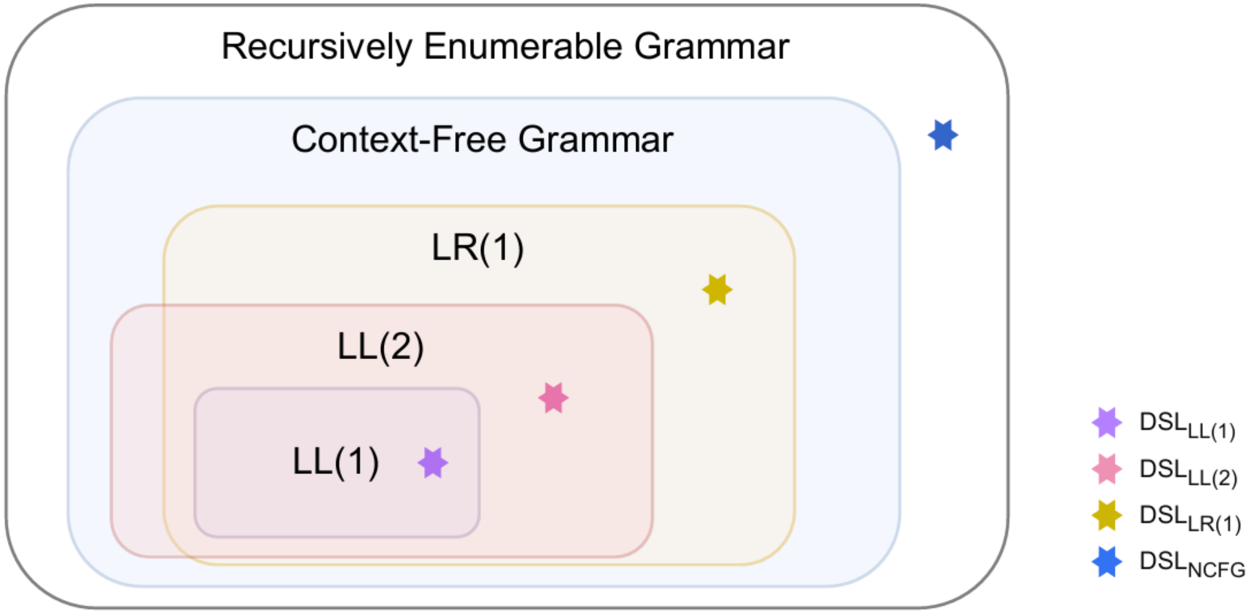
Different in Grammar Rule Representation
Similar in Token Representation

Edit Distance between Semantically Different Code



Benefit: Easier Parsing [Submitted]

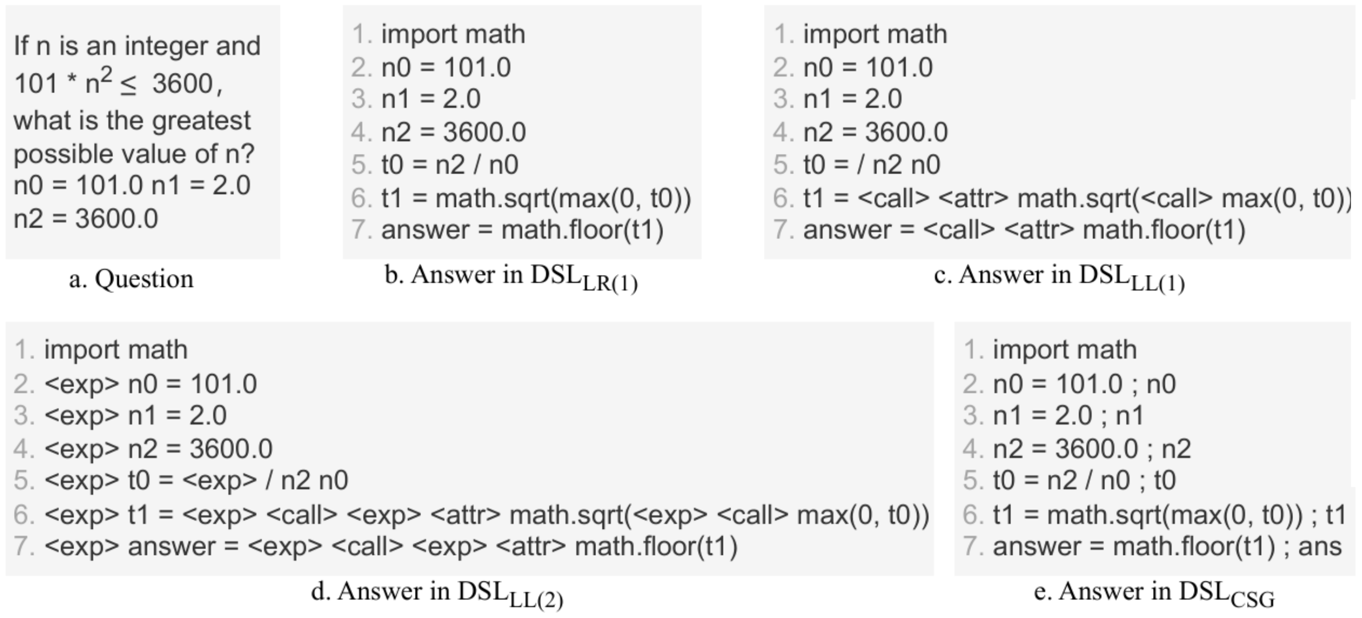
- The easier the language is to parse, the better the performance of the neural model.



Language	Mean(%)
$DSL_{LL(1)}$	82.00
$DSL_{LL(2)}$	81.74
$DSL_{LR(1)}$	81.14
DSL_{NCFG}	80.41

Benefit: Easier Parsing [Submitted]

- The easier the language is to parse, the better the performance of the neural model.



```
If n is an integer and
101 * n^2 ≤ 3600,
what is the greatest
possible value of n?
n0 = 101.0 n1 = 2.0
n2 = 3600.0
```

a. Question

```
import math
n0 = 101.0
n1 = 2.0
n2 = 3600.0
t0 = n2 / n0
t1 = math.sqrt(max(0, t0))
answer = math.floor(t1)
```

b. Answer in $\text{DSL}_{LR(1)}$

```
import math
n0 = 101.0
n1 = 2.0
n2 = 3600.0
t0 = / n2 n0
t1 = <call> <attr> math.sqrt(<call> max(0, t0))
answer = <call> <attr> math.floor(t1)
```

c. Answer in $\text{DSL}_{LL(1)}$

```
import math
<exp> n0 = 101.0
<exp> n1 = 2.0
<exp> n2 = 3600.0
<exp> t0 = <exp> / n2 n0
<exp> t1 = <exp> <call> <exp> <attr> math.sqrt(<exp> <call> max(0, t0))
<exp> answer = <exp> <call> <exp> <attr> math.floor(t1)
```

d. Answer in $\text{DSL}_{LL(2)}$

```
import math
n0 = 101.0 ; n0
n1 = 2.0 ; n1
n2 = 3600.0 ; n2
t0 = n2 / n0 ; t0
t1 = math.sqrt(max(0, t0)) ; t1
answer = math.floor(t1) ; ans
```

e. Answer in DSL_{CSG}

Language	Mean(%)
$\text{DSL}_{LL(1)}$	82.00
$\text{DSL}_{LL(2)}$	81.74
$\text{DSL}_{LR(1)}$	81.14
DSL_{NCFG}	80.41

Benefit: Easier Parsing [Submitted]

- Grammar rule representation is in LL(1)
- Significantly easier than most languages
  - Python: not context-free
  - Java: LR

Overview

TreeGen [AAAI20]
- Using transformer to implement L2S
- The first transformer-based code generator
- SOTA 35m model

Grape [IJCAI22]
- Guiding NN to learn grammar rule definitions

Tare [ICSE23]
- Guide NN to learn typing rules

GrammarT5 [ICSE24]
- Applying L2S to pretraining
- SOTA 0.2B model at that time

DeepSeek-Coder [arxiv]
- Guiding NN to learn Def-Use relations
- SOTA open-source code model at that time

GrammarCoder [ACL25-Finding]
- Applying L2S to Decoder-only
- SOTA 1.5B code model

Implements

L2S Framework [TOSEM22]
- Representing code as grammar rule sequences
- Ensuring syntactic correctness
- Allowing easy symbolic analysis

Apply

ACS [ICSE17]
- First program repair approach whose precision > 70%

Recoder [FSE23]
- First neural program repair approach outperforming traditional approaches

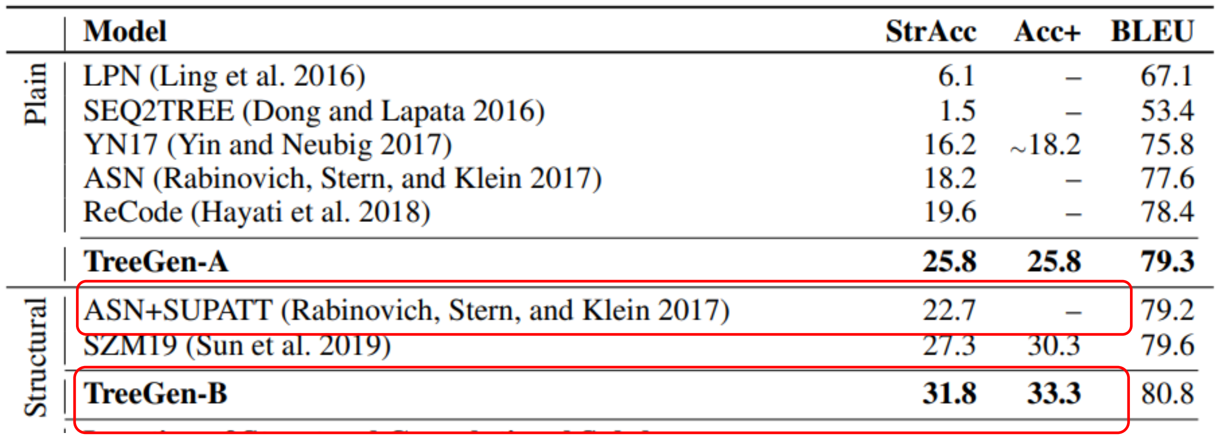
ET[APRCOMP24]
- 1st Place in APR-COMP

OCoR [ASE20]
- Code search engine significantly outperforming existing ones

LEAM [ASE22]
- Mutation generation engine that significantly outperforming existing ones

Using Transformer to implement L2S [AAAI20]

- The earliest work that applies Transformer for code generation
  - TreeGen: a Transformer model designed for grammar rule sequences



	Model	StrAcc	Acc+	BLEU
Plain	LPN (Ling et al. 2016)	6.1	–	67.1
	SEQ2TREE (Dong and Lapata 2016)	1.5	–	53.4
	YN17 (Yin and Neubig 2017)	16.2	~18.2	75.8
	ASN (Rabinovich, Stern, and Klein 2017)	18.2	–	77.6
	ReCode (Hayati et al. 2018)	19.6	–	78.4
	TreeGen-A	**25.8**	**25.8**	**79.3**
Structural	ASN+SUPATT (Rabinovich, Stern, and Klein 2017)	22.7	–	79.2
	SZM19 (Sun et al. 2019)	27.3	30.3	79.6
	TreeGen-B	**31.8**	**33.3**	80.8

TreeGen has been widely applied to decompilation, program repair, code search, automating editing by different researchers

Overview

TreeGen [AAAI20]
- Using transformer to implement L2S
- The first transformer-based code generator
- SOTA 35m model

Grape [IJCAI22]
- Guiding NN to learn grammar rule definitions

Tare [ICSE23]
- Guide NN to learn typing rules

GrammarT5 [ICSE24]
- Applying L2S to pretraining
- SOTA 0.2B model at that time

DeepSeek-Coder [arxiv]
- Guiding NN to learn Def-Use relations
- SOTA open-source code model at that time

GrammarCoder [ACL25-Finding]
- Applying L2S to Decoder-only
- SOTA 1.5B code model

Implements

L2S Framework [TOSEM22]
- Representing code as grammar rule sequences
- Ensuring syntactic correctness
- Allowing easy symbolic analysis

Apply

ACS [ICSE17]
- First program repair approach whose precision > 70%

Recoder [FSE23]
- First neural program repair approach outperforming traditional approaches

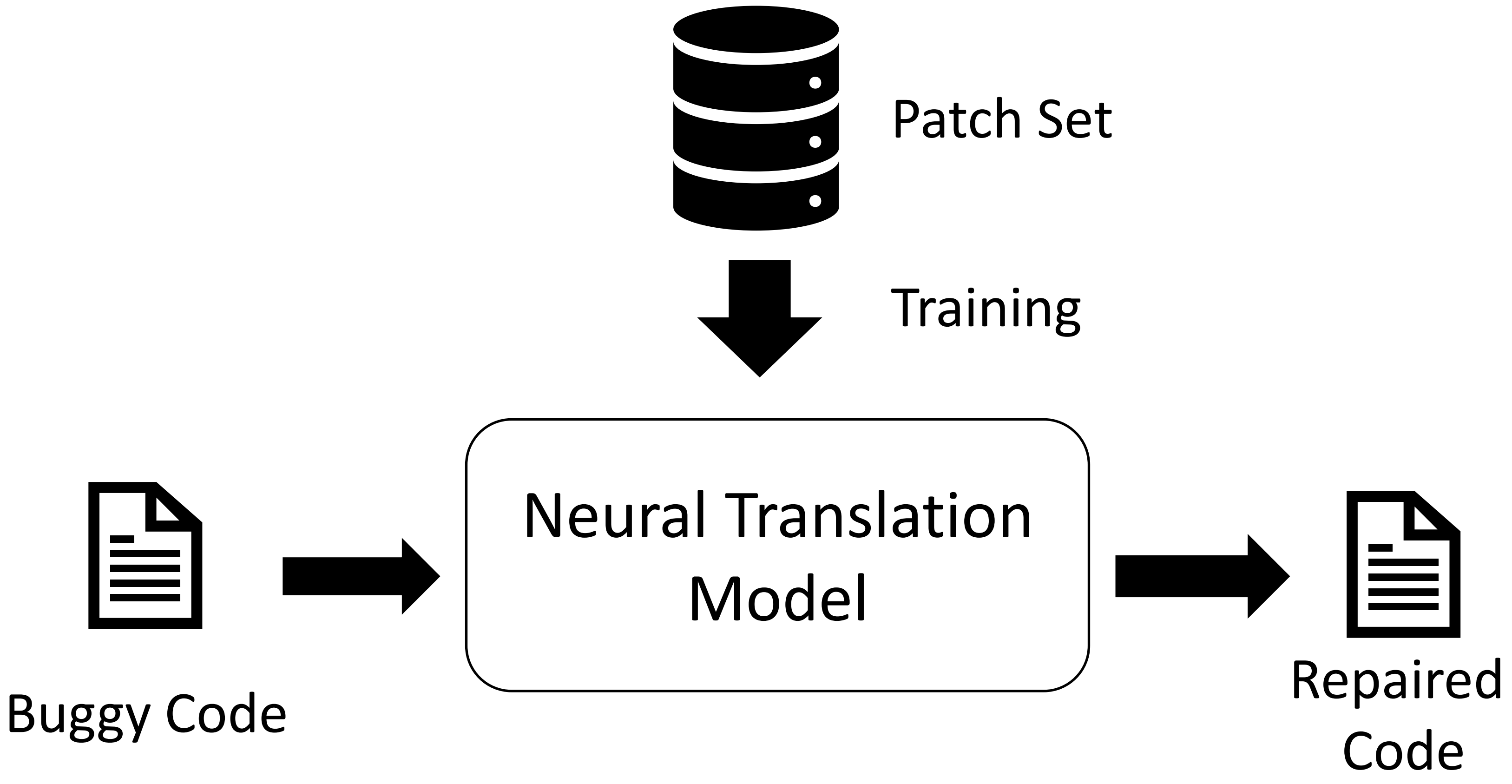
ET[APRCOMP24]
- 1st Place in APR-COMP

OCoR [ASE20]
- Code search engine significantly outperforming existing ones

LEAM [ASE22]
- Mutation generation engine that significantly outperforming existing ones

Existing Neural Program Repair

- Treating a patch as a pair of code



A finding in bidirectional transformation [Models'11 MIP]

- State-based representation is ineffective

```
cfa.createEdge(fromNode, Branch.UNCOND, finallyNode);
```

```
cfa.createEdge(fromNode, Branch.ON_EX, finallyNode);
```

1. Need to learn diff during training
2. Repr is long (13 tokens)

- Delta-based representation is more desirable

```
modify(9, ON_EX)
```

1. Change is directly given
2. Repr is short (3 tokens)

A grammar of change

1.	*Edits*	$\rightarrow$	*Edit*; *Edits* | `end`
2.	*Edit*	$\rightarrow$	*Insert* | *Modify*
3.	*Insert*	$\rightarrow$	`insert(`⟨*HLStatement*⟩`)`
4.	*Modify*	$\rightarrow$	`modify(` ⟨*ID of an AST Node with a NTS*⟩, ⟨*the same NTS as the above NTS*⟩`)`
5.	⟨*Any NTS in HL*⟩	$\rightarrow$	`copy(`⟨*ID of an AST Node with the same NTS*⟩`)`
		|	⟨*The original production rules in HL*⟩
6.	⟨*HLIdentifier*⟩	$\rightarrow$	`placeholder`
		|	⟨*Identifiers in the training set*⟩

Ensuring the changed code is still syntactically correct.

Recoder [ESEC/FSE'21]

- TreeGen for generating changes
- Neural program repair outperformed traditional approaches for the first time

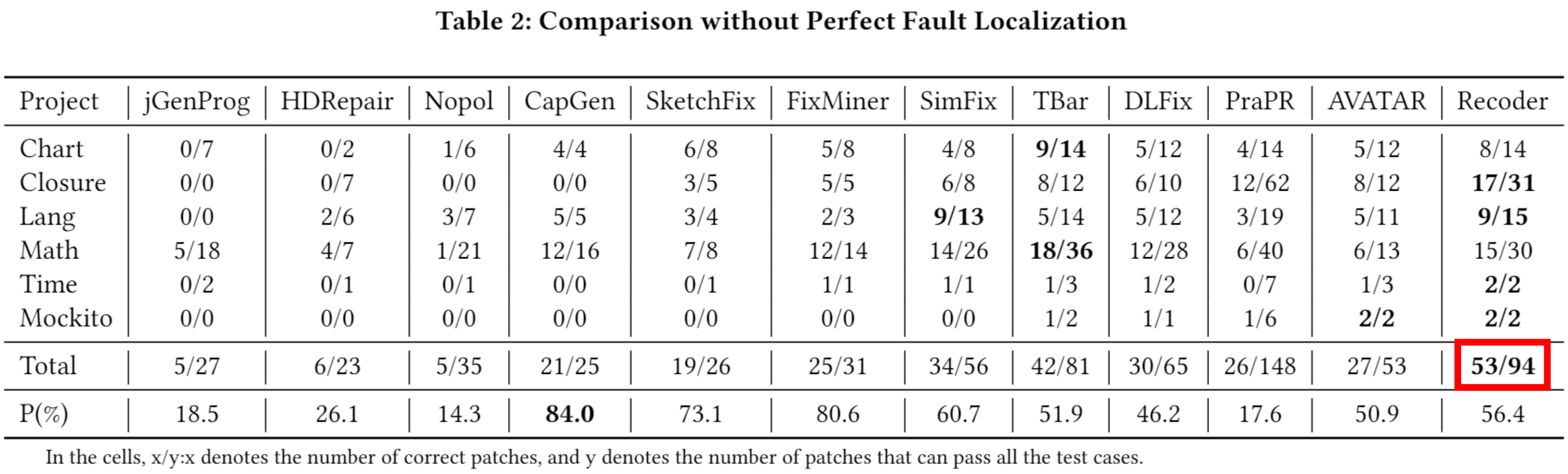
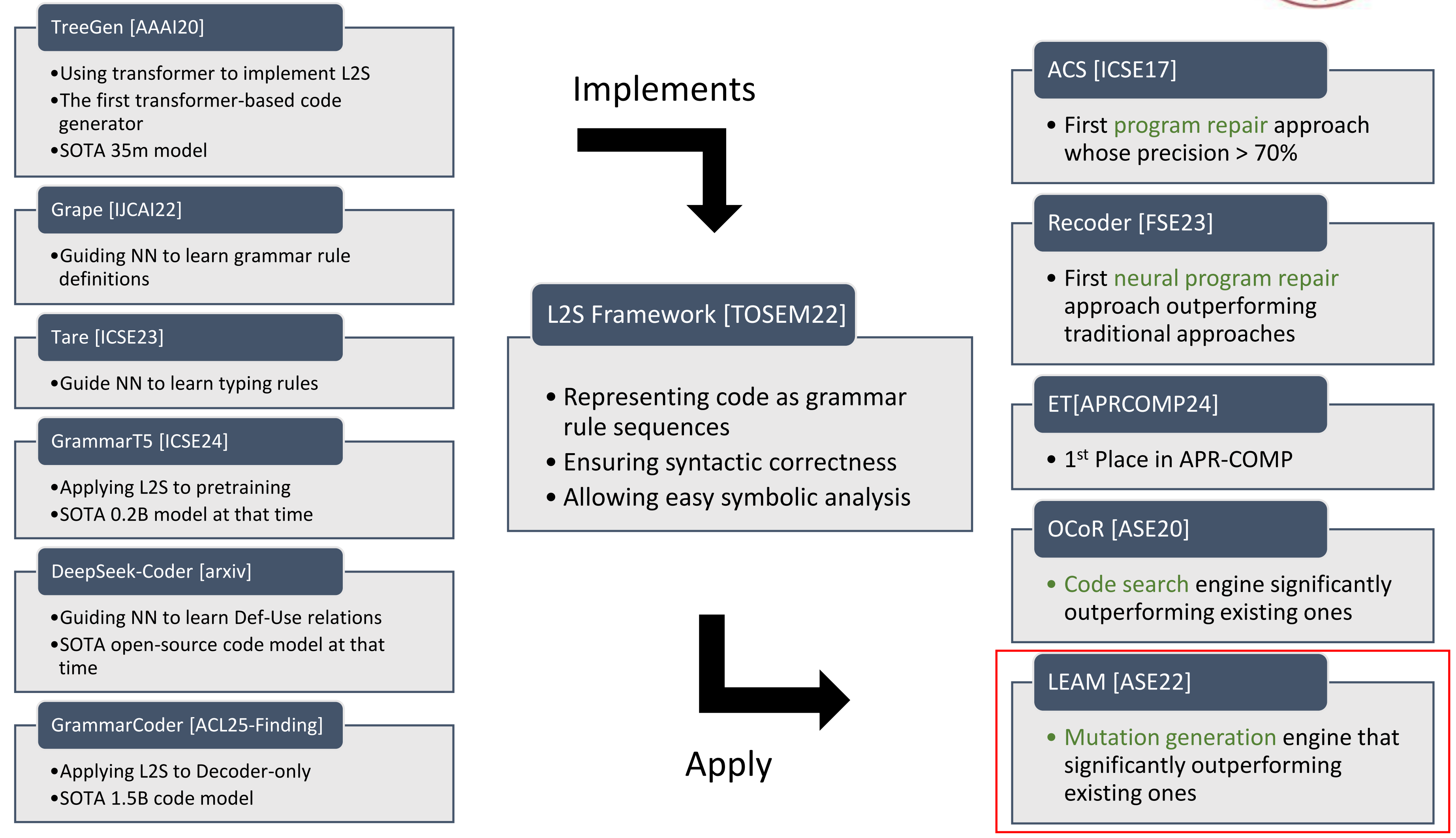


Table 2: Comparison without Perfect Fault Localization

Project	jGenProg	HDRepair	Nopol	CapGen	SketchFix	FixMiner	SimFix	TBar	DLFix	PraPR	AVATAR	Recoder
Chart	0/7	0/2	1/6	4/4	6/8	5/8	4/8	**9/14**	5/12	4/14	5/12	8/14
Closure	0/0	0/7	0/0	0/0	3/5	5/5	6/8	8/12	6/10	12/62	8/12	**17/31**
Lang	0/0	2/6	3/7	5/5	3/4	2/3	**9/13**	5/14	5/12	3/19	5/11	**9/15**
Math	5/18	4/7	1/21	12/16	7/8	12/14	14/26	**18/36**	12/28	6/40	6/13	15/30
Time	0/2	0/1	0/1	0/0	0/1	1/1	1/1	1/3	1/2	0/7	1/3	**2/2**
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/2	1/1	1/6	**2/2**	**2/2**
Total	5/27	6/23	5/35	21/25	19/26	25/31	34/56	42/81	30/65	26/148	27/53	**53/94**
P(%)	18.5	26.1	14.3	**84.0**	73.1	80.6	60.7	51.9	46.2	17.6	50.9	56.4

In the cells, x/y:x denotes the number of correct patches, and y denotes the number of patches that can pass all the test cases.

Overview



TreeGen [AAAI20]

- Using transformer to implement L2S
- The first transformer-based code generator
- SOTA 35m model

Grape [IJCAI22]

- Guiding NN to learn grammar rule definitions

Tare [ICSE23]

- Guide NN to learn typing rules

GrammarT5 [ICSE24]

- Applying L2S to pretraining
- SOTA 0.2B model at that time

DeepSeek-Coder [arxiv]

- Guiding NN to learn Def-Use relations
- SOTA open-source code model at that time

GrammarCoder [ACL25-Finding]

- Applying L2S to Decoder-only
- SOTA 1.5B code model

Implements

L2S Framework [TOSEM22]

- Representing code as grammar rule sequences
- Ensuring syntactic correctness
- Allowing easy symbolic analysis

Apply

ACS [ICSE17]

- First program repair approach whose precision > 70%

Recoder [FSE23]

- First neural program repair approach outperforming traditional approaches

ET[APRCOMP24]

- 1st Place in APR-COMP

OCoR [ASE20]

- Code search engine significantly outperforming existing ones

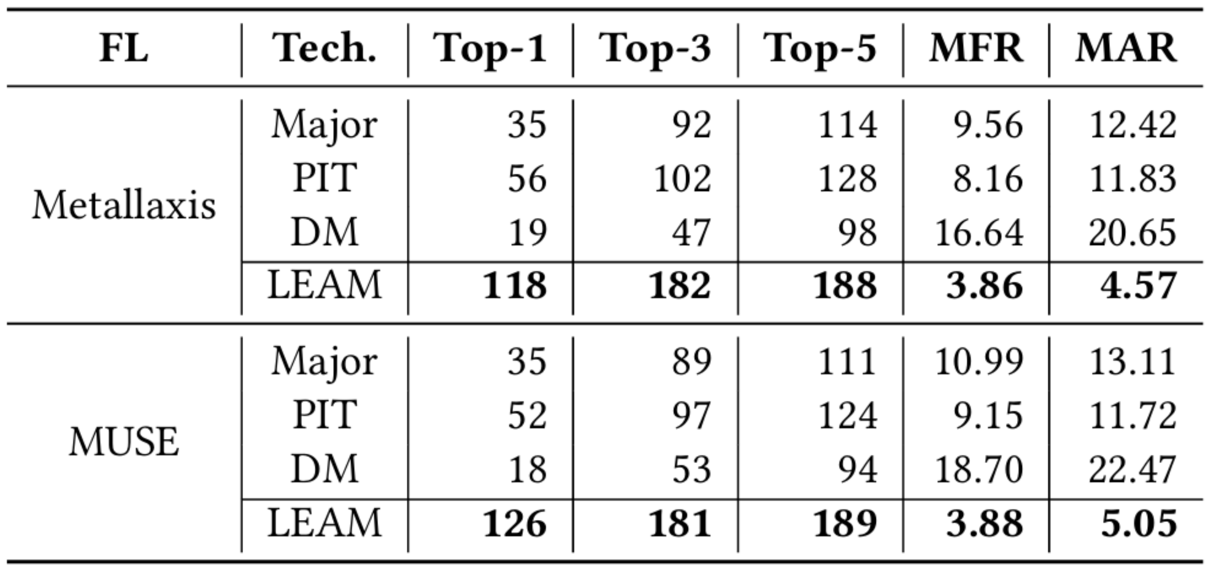
LEAM [ASE22]

- Mutation generation engine that significantly outperforming existing ones

LEAM [ASE’22 Distinguished]

- From Junjie Chen and Lingming Zhang’s group
- Exchange the input and output of Recoder
- Program Repairer -> Bug Seeder

Table 4: Overall effectiveness in mutation-based FL



FL	Tech.	Top-1	Top-3	Top-5	MFR	MAR
Metallaxis	Major	35	92	114	9.56	12.42
	PIT	56	102	128	8.16	11.83
	DM	19	47	98	16.64	20.65
	LEAM	**118**	**182**	**188**	**3.86**	**4.57**
MUSE	Major	35	89	111	10.99	13.11
	PIT	52	97	124	9.15	11.72
	DM	18	53	94	18.70	22.47
	LEAM	**126**	**181**	**189**	**3.88**	**5.05**

Overview

TreeGen [AAAI20]

- Using transformer to implement L2S
- The first transformer-based code generator
- SOTA 35m model

Grape [IJCAI22]

- Guiding NN to learn grammar rule definitions

Tare [ICSE23]

- Guide NN to learn typing rules

GrammarT5 [ICSE24]

- Applying L2S to pretraining
- SOTA 0.2B model at that time

DeepSeek-Coder [arxiv]

- Guiding NN to learn Def-Use relations
- SOTA open-source code model at that time

GrammarCoder [ACL25-Finding]

- Applying L2S to Decoder-only
- SOTA 1.5B code model

Implements

L2S Framework [TOSEM22]

- Representing code as grammar rule sequences
- Ensuring syntactic correctness
- Allowing easy symbolic analysis

Apply

ACS [ICSE17]

- First program repair approach whose precision > 70%

Recoder [FSE23]

- First neural program repair approach outperforming traditional approaches

ET[APRCOMP24]

- 1st Place in APR-COMP

OCoR [ASE20]

- Code search engine significantly outperforming existing ones

LEAM [ASE22]

- Mutation generation engine that significantly outperforming existing ones

Limit of L2S

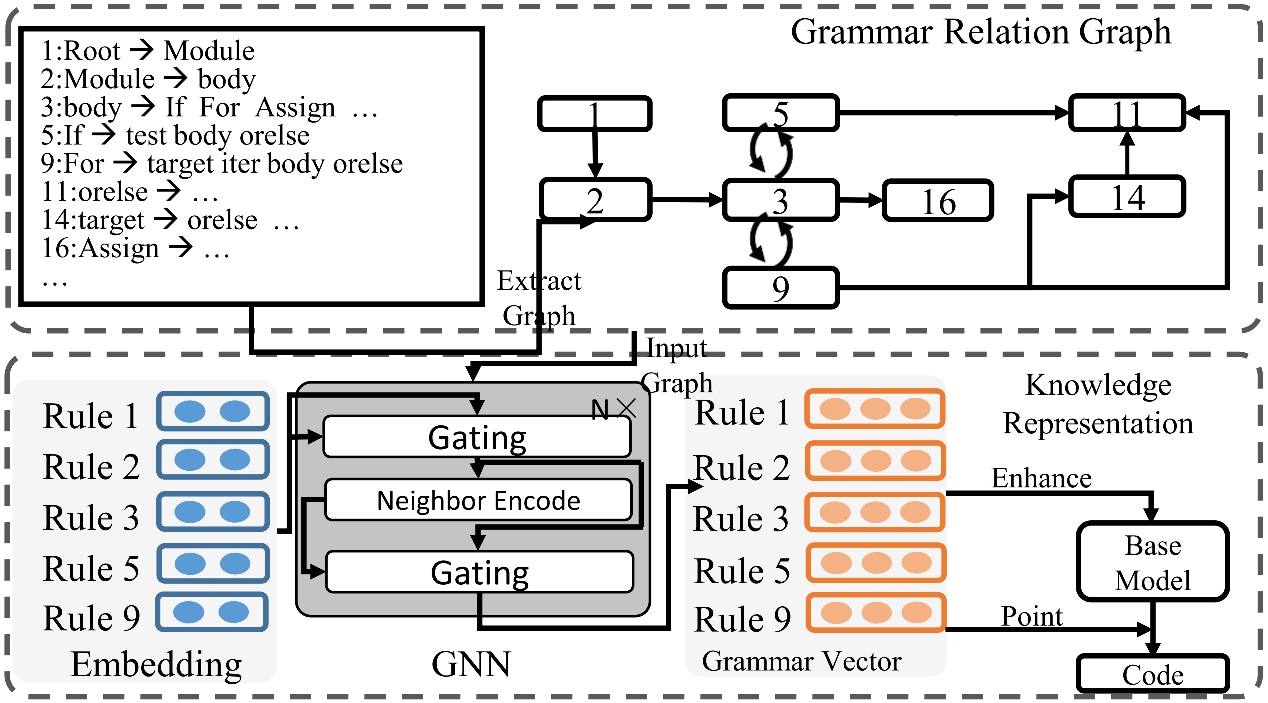
- Force syntactical and other constraint from outside
- NN does not learn their definitions

ifstmt -> ‘if’ ‘(‘ boolExpr ‘)’ stmt	10
whilestat -> ‘while’ ‘(‘ boolExpr ‘)’ stmt	11
boolExpr -> andExpr	12
boolExpr -> orExpr	13

Grammar rules are encoded as numbers without content.
NN could predict impossible sequences such as 10, 11.

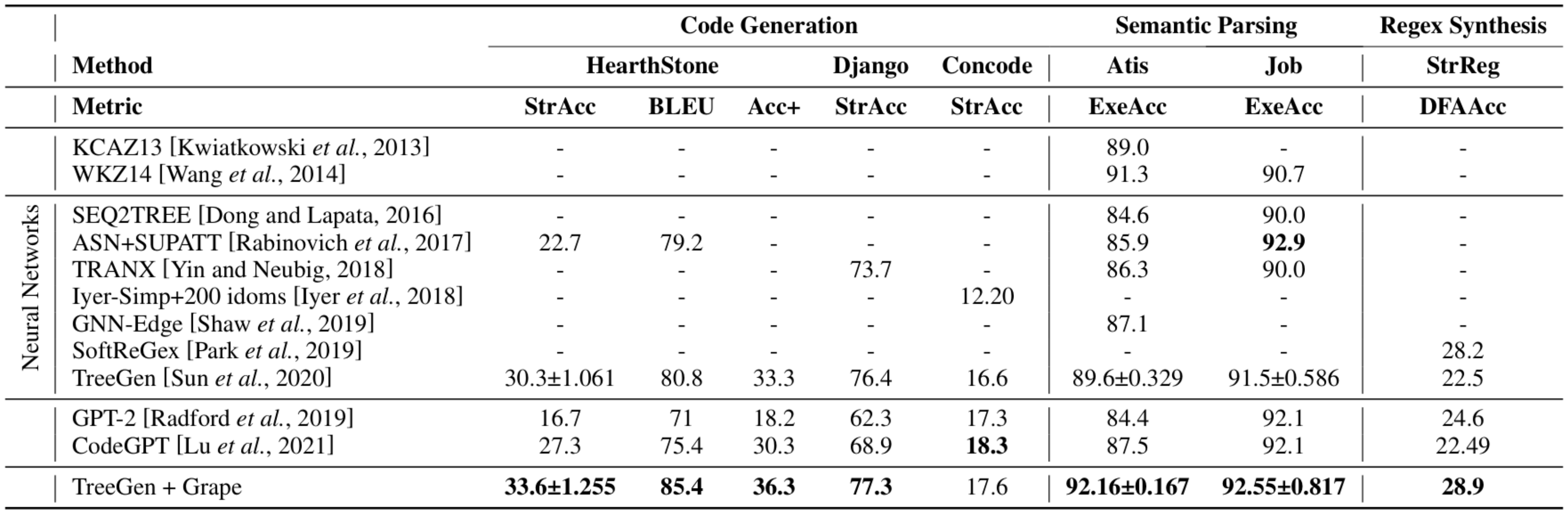
Learning Grammar Rules [IJCAI22]

- Guide the NN to learn grammar definitions
- Word2Vec: assign each token a vector
- Grape: assign each grammar rule a vector, learned with its definition structure



Learning Grammar Rules [IJCAI22]

- Improve the performance of TreeGen up to 5 percentage points
- Outperforms larger pre-training models



		Code Generation					Semantic Parsing		Regex Synthesis
	Method	**HearthStone**			**Django**	**Concode**	**Atis**	**Job**	**StrReg**
	Metric	**StrAcc**	**BLEU**	**Acc+**	**StrAcc**	**StrAcc**	**ExeAcc**	**ExeAcc**	**DFAAcc**
	KCAZ13 [Kwiatkowski *et al.*, 2013]	-	-	-	-	-	89.0	-	-
	WKZ14 [Wang *et al.*, 2014]	-	-	-	-	-	91.3	90.7	-
Neural Networks	SEQ2TREE [Dong and Lapata, 2016]	-	-	-	-	-	84.6	90.0	-
	ASN+SUPATT [Rabinovich *et al.*, 2017]	22.7	79.2	-	-	-	85.9	**92.9**	-
	TRANX [Yin and Neubig, 2018]	-	-	-	73.7	-	86.3	90.0	-
	Iyer-Simp+200 idoms [Iyer *et al.*, 2018]	-	-	-	-	12.20	-	-	-
	GNN-Edge [Shaw *et al.*, 2019]	-	-	-	-	-	87.1	-	-
	SoftReGex [Park *et al.*, 2019]	-	-	-	-	-	-	-	28.2
	TreeGen [Sun *et al.*, 2020]	30.3±1.061	80.8	33.3	76.4	16.6	89.6±0.329	91.5±0.586	22.5
	GPT-2 [Radford *et al.*, 2019]	16.7	71	18.2	62.3	17.3	84.4	92.1	24.6
	CodeGPT [Lu *et al.*, 2021]	27.3	75.4	30.3	68.9	**18.3**	87.5	92.1	22.49
	TreeGen + Grape	**33.6±1.255**	**85.4**	**36.3**	**77.3**	17.6	**92.16±0.167**	**92.55±0.817**	**28.9**

Parameters：TreeGen+Grape: 35M GPT-2、CodeGPT：110M

Overview

TreeGen [AAAI20]
- Using transformer to implement L2S
- The first transformer-based code generator
- SOTA 35m model

Grape [IJCAI22]
- Guiding NN to learn grammar rule definitions

Tare [ICSE23]
- Guide NN to learn typing rules

GrammarT5 [ICSE24]
- Applying L2S to pretraining
- SOTA 0.2B model at that time

DeepSeek-Coder [arxiv]
- Guiding NN to learn Def-Use relations
- SOTA open-source code model at that time

GrammarCoder [ACL25-Finding]
- Applying L2S to Decoder-only
- SOTA 1.5B code model

Implements

L2S Framework [TOSEM22]
- Representing code as grammar rule sequences
- Ensuring syntactic correctness
- Allowing easy symbolic analysis

Apply

ACS [ICSE17]
- First program repair approach whose precision > 70%

Recoder [FSE23]
- First neural program repair approach outperforming traditional approaches

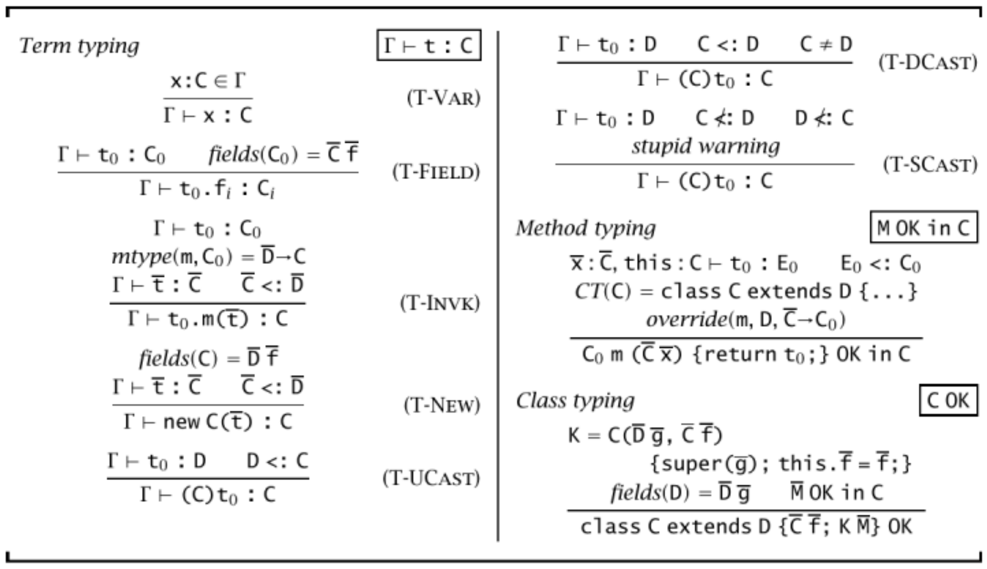
ET[APRCOMP24]
- 1st Place in APR-COMP

OCoR [ASE20]
- Code search engine significantly outperforming existing ones

LEAM [ASE22]
- Mutation generation engine that significantly outperforming existing ones

Learning Typing Rules [ICSE23]

- Full type system is difficult to learn from data



Term typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{C}}$

$$\frac{\mathtt{x}:\mathtt{C} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{C}} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{f}_i : \mathtt{C}_i} \quad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \\ \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) : \mathtt{C}} \quad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \\ \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{t}}) : \mathtt{C}} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \quad \text{(T-UCAST)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \quad \text{(T-DCAST)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \\ \mathit{stupid\ warning}\end{array}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \quad \text{(T-SCAST)}$$

Method typing $\boxed{\mathtt{M}\ \mathtt{OK\ in\ C}}$

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{t}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\} \\ \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}_0)\end{array}}{\mathtt{C}_0\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathtt{return}\ \mathtt{t}_0;\}\ \mathtt{OK\ in\ C}}$$

Class typing $\boxed{\mathtt{C\ OK}}$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}) \\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ in\ C}\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}}$$

Figure 19-4: Featherweight Java (typing)

- Only 30%-40% programs generated by Recoder is typable

Learning Typing Rules [ICSE23]

- A single rule is much easier to learn
  - T-Graph: present the input of a typing rule to the NN
  - T-Grammar: force NN to predict the output of a typing rule

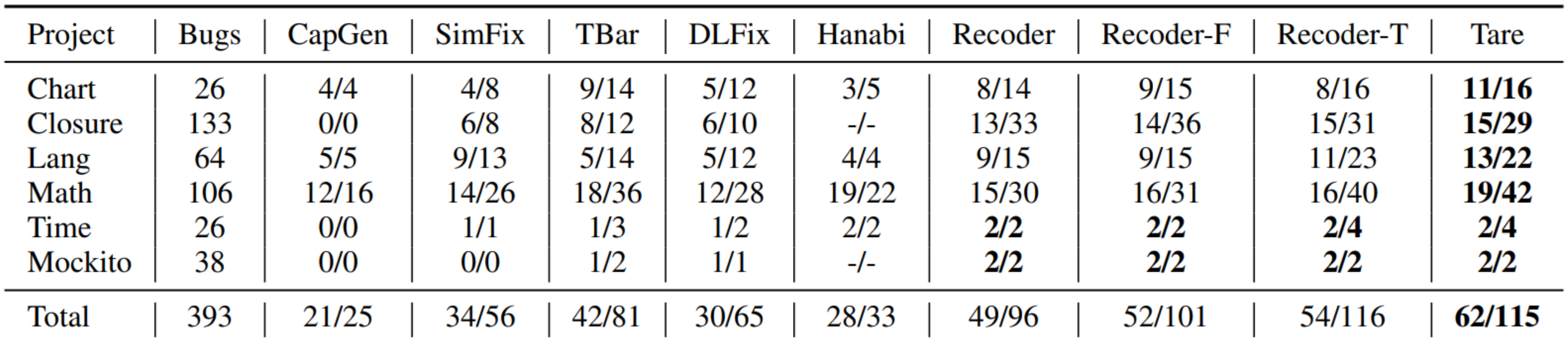
T-Graph: Representing typing relations

- types of AST nodes
- types of variables
- subtyping relations

T-Grammar:
E -> E && E becomes
[Bool]E -> [Bool]E && [Bool]E

Learning Typing Rules [ICSE23]

- Applying to program repair, forming Tare



Project	Bugs	CapGen	SimFix	TBar	DLFix	Hanabi	Recoder	Recoder-F	Recoder-T	Tare
Chart	26	4/4	4/8	9/14	5/12	3/5	8/14	9/15	8/16	**11/16**
Closure	133	0/0	6/8	8/12	6/10	-/-	13/33	14/36	15/31	**15/29**
Lang	64	5/5	9/13	5/14	5/12	4/4	9/15	9/15	11/23	**13/22**
Math	106	12/16	14/26	18/36	12/28	19/22	15/30	16/31	16/40	**19/42**
Time	26	0/0	1/1	1/3	1/2	2/2	**2/2**	**2/2**	**2/4**	**2/4**
Mockito	38	0/0	0/0	1/2	1/1	-/-	**2/2**	**2/2**	**2/2**	**2/2**
Total	393	21/25	34/56	42/81	30/65	28/33	49/96	52/101	54/116	**62/115**

Tare+ExpressAPR(efficient patch validation tool) got the first place in the Java functional bug track of APR-COMP'24.

Overview

TreeGen [AAAI20]
- Using transformer to implement L2S
- The first transformer-based code generator
- SOTA 35m model

Grape [IJCAI22]
- Guiding NN to learn grammar rule definitions

Tare [ICSE23]
- Guide NN to learn typing rules

GrammarT5 [ICSE24]
- Applying L2S to pretraining
- SOTA 0.2B model at that time

DeepSeek-Coder [arxiv]
- Guiding NN to learn Declare-Use relations
- SOTA open-source code model at that time

GrammarCoder [ACL25-Finding]
- Applying L2S to Decoder-only
- SOTA 1.5B code model

Implements

L2S Framework [TOSEM22]
- Representing code as grammar rule sequences
- Ensuring syntactic correctness
- Allowing easy symbolic analysis

Apply

ACS [ICSE17]
- First program repair approach whose precision > 70%

Recoder [FSE23]
- First neural program repair approach outperforming traditional approaches

ET[APRCOMP24]
- 1st Place in APR-COMP

OCoR [ASE20]
- Code search engine significantly outperforming existing ones

LEAM [ASE22]
- Mutation generation engine that significantly outperforming existing ones

A Era of LLMs

Bard

ChatGPT

Meta AI

LLMs (=pretrained large models) exhibit superior performance

Can we use grammar-based representation in LLMs?

Challenges

- Big vocabulary
  - User-defined identifiers can be added to the grammar when the training set is small
  - Pre-training sets are too large
- Heterogeneous grammars
  - Existing models: One programming language
  - Pretraining models: Many programming languages
- Pretraining Tasks
  - Self-supervised training tasks are needed
  - Tasks are expected to guide the neural network to learn the grammar structure

Big vocabulary

- Existing approaches
  - IDEN -> isodd | iseven
- Our approach
  - Using BPE (Byte Pair Encoding) to find a small set of subtokens
    - is, odd, even
  - Integrating them into the grammar
    - IDEN -> is IDEN | odd IDEN | even IDEN
      | #is | #odd | #even
    - # indicates the ending tokens
    - Leads to significantly shorter encoding than the standard sequence encoding
      - IDEN -> is IDEN | odd IDEN | even IDEN
        | ϵ

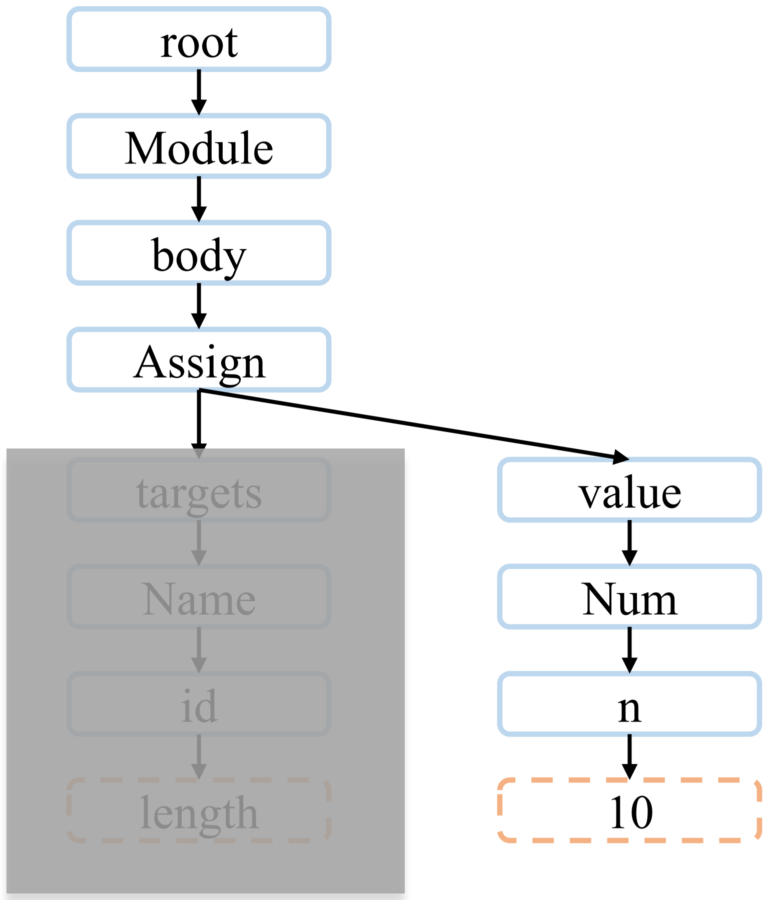
Heterogeneous grammars

- A hyper grammar that includes all grammars
  - Root -> Root@Python | Root@Java | …
- Experimentally has better performance than sharing some of non-terminals
  - While -> while ‘(‘ BoolExpr ‘)’ Statements
  - BoolExpr -> BoolExpr@Java | BoolExpr@C# | …

Pretraining Tasks

- Given a rule sequence, predicting the parent of a rule
  - $$\begin{tikzcd}1 & 2 & 3 & \boxed{10} & 11 & 13 & 64 & \boxed{18} \arrow[llll, bend left] & 19 & 8\end{tikzcd}$$

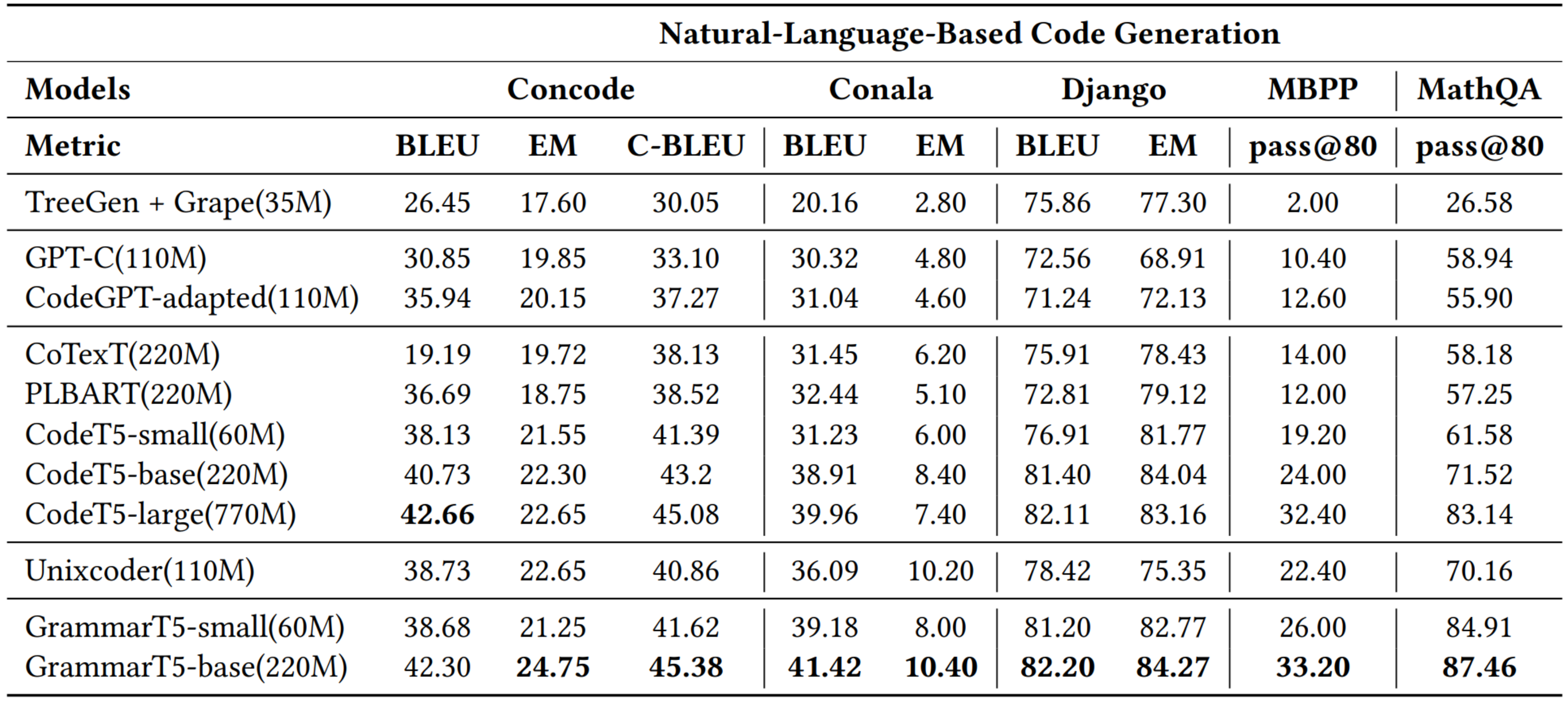
- Predicting some subtree of an AST



Learning Declare-Use Relation

- Existing pre-training models sort files randomly
- LLMs may see a function or a variable before its declaration
- Dependency parsing:
  - Extract declaration-use relationship from files
  - Sort the files so that declarations appear before use

GrammarT5 [ICSE24]

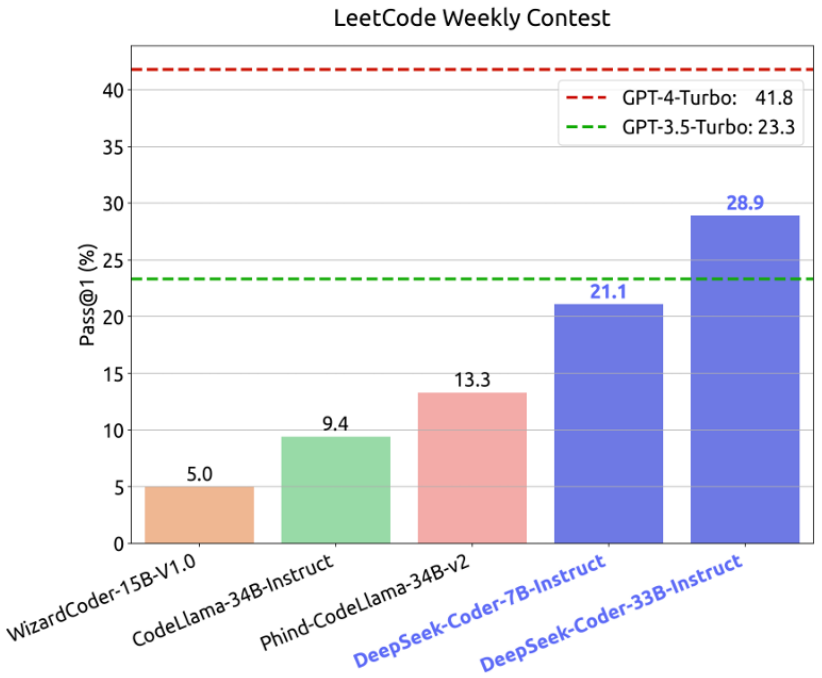


	Natural-Language-Based Code Generation								
Models	**Concode**			**Conala**		**Django**		**MBPP**	**MathQA**
Metric	**BLEU**	**EM**	**C-BLEU**	**BLEU**	**EM**	**BLEU**	**EM**	**pass@80**	**pass@80**
TreeGen + Grape(35M)	26.45	17.60	30.05	20.16	2.80	75.86	77.30	2.00	26.58
GPT-C(110M)	30.85	19.85	33.10	30.32	4.80	72.56	68.91	10.40	58.94
CodeGPT-adapted(110M)	35.94	20.15	37.27	31.04	4.60	71.24	72.13	12.60	55.90
CoTexT(220M)	19.19	19.72	38.13	31.45	6.20	75.91	78.43	14.00	58.18
PLBART(220M)	36.69	18.75	38.52	32.44	5.10	72.81	79.12	12.00	57.25
CodeT5-small(60M)	38.13	21.55	41.39	31.23	6.00	76.91	81.77	19.20	61.58
CodeT5-base(220M)	40.73	22.30	43.2	38.91	8.40	81.40	84.04	24.00	71.52
CodeT5-large(770M)	**42.66**	22.65	45.08	39.96	7.40	82.11	83.16	32.40	83.14
Unixcoder(110M)	38.73	22.65	40.86	36.09	10.20	78.42	75.35	22.40	70.16
GrammarT5-small(60M)	38.68	21.25	41.62	39.18	8.00	81.20	82.77	26.00	84.91
GrammarT5-base(220M)	42.30	**24.75**	**45.38**	**41.42**	**10.40**	**82.20**	**84.27**	**33.20**	**87.46**

SOTA 0.2B model in 2024

Standard L2S was applied

DeepSeek-Coder [arxiv24]

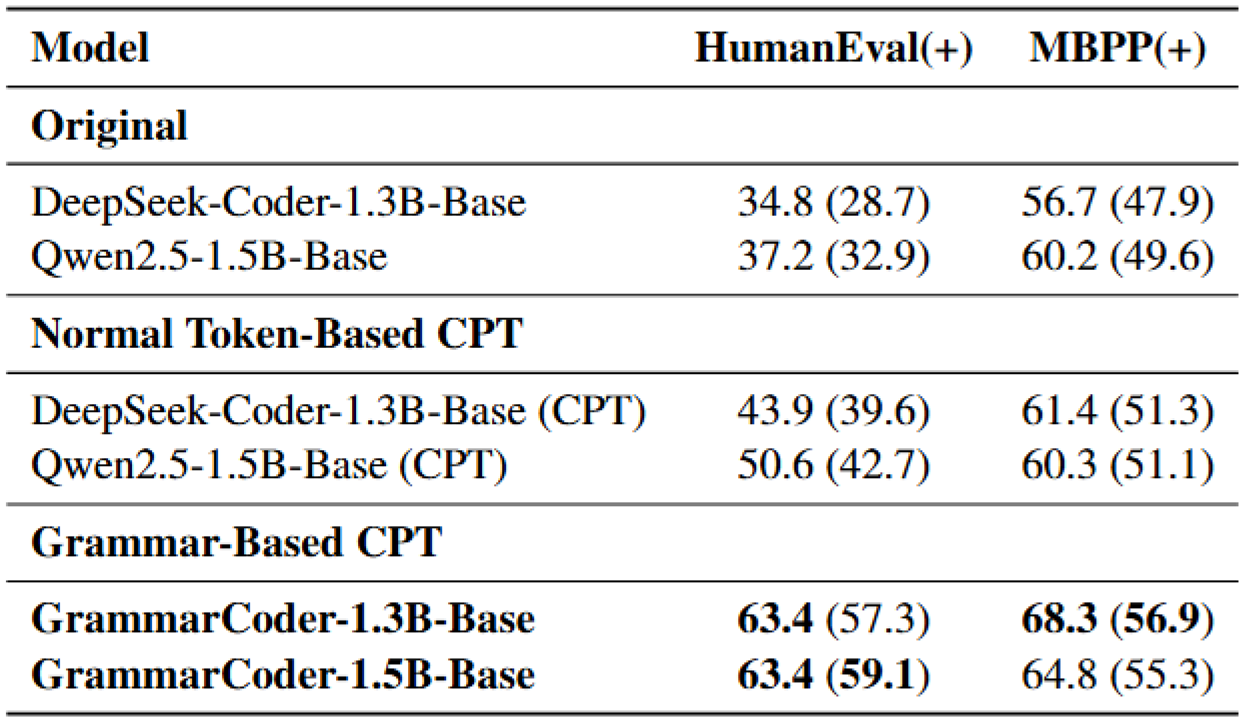


Cooperation with DeepSeek

SOTA open source code model in 2024, still widely used as a base model for fine-tuning

Dependency parsing was applied

GrammarCoder [ACL25-Finding]



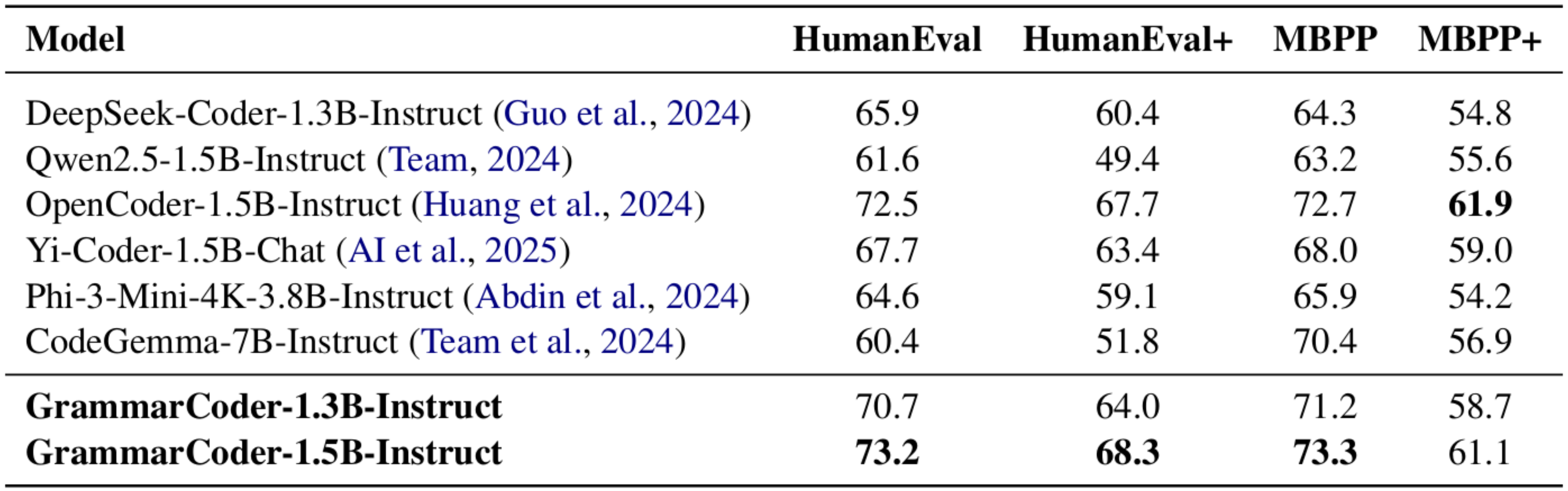
Model	HumanEval(+)	MBPP(+)
Original		
DeepSeek-Coder-1.3B-Base	34.8 (28.7)	56.7 (47.9)
Qwen2.5-1.5B-Base	37.2 (32.9)	60.2 (49.6)
Normal Token-Based CPT		
DeepSeek-Coder-1.3B-Base (CPT)	43.9 (39.6)	61.4 (51.3)
Qwen2.5-1.5B-Base (CPT)	50.6 (42.7)	60.3 (51.1)
Grammar-Based CPT		
GrammarCoder-1.3B-Base	**63.4** (57.3)	**68.3 (56.9)**
GrammarCoder-1.5B-Base	**63.4 (59.1)**	64.8 (55.3)

Cooperation with Kuaishou

SOTA 1.5B model

Standard L2S was applied

GrammarCoder [ACL25-Finding]



Model	HumanEval	HumanEval+	MBPP	MBPP+
DeepSeek-Coder-1.3B-Instruct (Guo et al., 2024)	65.9	60.4	64.3	54.8
Qwen2.5-1.5B-Instruct (Team, 2024)	61.6	49.4	63.2	55.6
OpenCoder-1.5B-Instruct (Huang et al., 2024)	72.5	67.7	72.7	**61.9**
Yi-Coder-1.5B-Chat (AI et al., 2025)	67.7	63.4	68.0	59.0
Phi-3-Mini-4K-3.8B-Instruct (Abdin et al., 2024)	64.6	59.1	65.9	54.2
CodeGemma-7B-Instruct (Team et al., 2024)	60.4	51.8	70.4	56.9
GrammarCoder-1.3B-Instruct	70.7	64.0	71.2	58.7
GrammarCoder-1.5B-Instruct	**73.2**	**68.3**	**73.3**	61.1

Cooperation with Kuaishou

SOTA 1.5B model

Standard L2S was applied

Conclusion

- Code has structure
  - Syntax
  - Types
  - Semantics
- These structures are defined by symbolic rules
- LLMs do not learn the symbolic rules well
- Guiding them to learn improves their performance
- Future: more genetic ways to learn more software knowledge