# Benchmarking Language Models for Code Syntax Understanding

**Da Shen[1], Xinyun Chen[2†], Chenguang Wang[3†], Koushik Sen[4], Dawn Song[4]**
[1]University of Maryland, College Park, [2]Google Research, Brain Team
[3]Washington University in St. Louis, [4]University of California, Berkeley
dashen@terpmail.umd.edu, xinyunchen@google.com, chenguangwang@wustl.edu,
{ksen,dawnsong}@cs.berkeley.edu

## Abstract

Pre-trained language models have demonstrated impressive performance in both natural language processing and program understanding, which represent the input as a token sequence without explicitly modeling its structure. Some prior works show that pre-trained language models can capture the syntactic rules of natural languages without fine-tuning on syntax understanding tasks. However, there is limited understanding of how well pre-trained models understand the code structure so far. In this work, we perform the first thorough benchmarking of the state-of-the-art pre-trained models for identifying the syntactic structures of programs. Specifically, we introduce CodeSyntax, a large-scale dataset of programs annotated with the syntactic relationships in their corresponding abstract syntax trees. Our key observation is that existing language models pretrained on code still lack the understanding of code syntax. In fact, these pre-trained programming language models fail to match the performance of simple baselines based on positional offsets and keywords. We also present a natural language benchmark to highlight the differences between natural languages and programming languages in terms of syntactic structure understanding. Our findings point out key limitations of existing pre-training methods for programming languages, and suggest the importance of modeling code syntactic structures.[1]

## 1 Introduction

Large-scale pre-training of language models has become the de-facto paradigm for a variety of natural language processing tasks. Furthermore, recent studies show that models pre-trained on a massive amount of code also achieve competitive performance on many tasks, e.g., code generation and code classification. These tasks are closely related to natural language (NL) tasks in their problem formulation. Nowadays, the common practice for solving these coding tasks is to utilize the language model architectures and training schemes that are originally designed for NL. The design principle of these neural language models is significantly different from the classic rule-based program generation

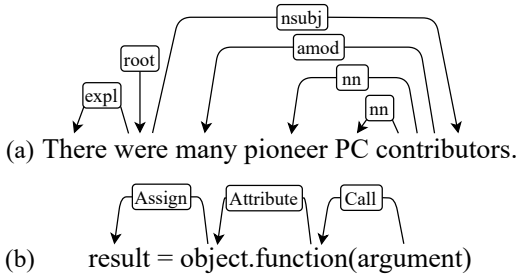



Figure 1: Examples of syntactic relations for (a) natural languages (NL) and (b) programming languages (PL). Each relation is represented by an arrow. The relations in PL represent the syntax of code in a way similar to those in NL.

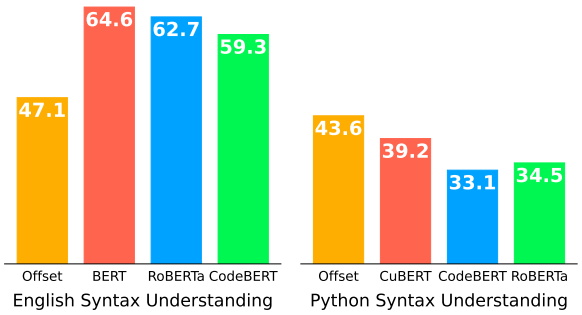



Figure 2: A preview of the model performance comparison on NL and PL syntax understanding tasks. Pre-trained models capture NL syntax relatively well, but perform worse in understanding PL syntax. The *Offset* baseline picks the token using a fixed positional offset. We use BERT-large and RoBERTa-base configurations (corresponding to the configurations of CuBERT and CodeBERT). The plot shows top-1 scores. See Tables 3 and 4 for the full results.

[†]Corresponding authors.
[1]Our code and dataset are available at https://github.com/dashends/CodeSyntax.



*Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 3071–3093
December 7-11, 2022 

systems. Specifically, neural language models take the program as a token sequence, while classic program generation systems utilize the language grammar and code structure. Despite the advanced performance of pre-trained language models on code understanding tasks, what these models have learned from the code corpus remains unclear.

In this work, we investigate whether large-scale pre-training is all we need for code representation learning. In particular, we conduct the first systematic study to analyze how the pre-trained language models understand the syntactic structures of programs. To this end, we introduce `CodeSyntax`, a large-scale benchmark consisting of programs annotated with the syntactic relationships between different tokens. The ground truth syntactic relationships are extracted from edges in the abstract syntax trees (AST) of the programs. Figure 1 shows some examples. These syntactic relations are function-wise similar to dependency relations for NL, where prior work has demonstrated that the attention heads of pre-trained language models can help to identify NL relation types (Clark et al., 2019; Raganato et al., 2018). To measure how well the pre-trained language models capture the code syntactic structures, we adopt the approach to the PL domain. We focus on investigating the zero-shot capability of existing pre-training methods in our experiments, and we evaluate these pre-trained models without finetuning them on our benchmark.

We evaluate the state-of-the-art pre-trained language models for code representation learning, including CuBERT (Kanade et al., 2020) and CodeBERT (Feng et al., 2020). A common characteristic of these models is that they share the same Transformer-based architectural design as NL models (Vaswani et al., 2017; Devlin et al., 2019). This allows us to directly compare their performance in capturing the syntax structure. We present a preview of our key results in Figure 2. Our main observation is that pre-training is insufficient for learning the syntactic relations in code. First, we find that the models pre-trained on code do not always outperform models pre-trained on NL corpus alone. Surprisingly, compared to CodeBERT which is trained on both text and code corpora, RoBERTa achieves better performance without training on any code with identical model architecture. This indicates that pre-training on programs as token sequences does not help learn the syntactic relations. On the contrary, without dependency relations, pre-training still enables language models to understand the NL syntax to some extent.

Moreover, for code syntax understanding, the pre-trained models even perform worse than simple baselines that pick the tokens with a fixed offset. For example, always selecting the (p+2)-th token as the p-th token's dependency yields higher accuracy than any attention head for several relation types. On the other hand, the same model architectures pre-trained on text corpora achieve decent accuracy in identifying the dependency relations in the NL domain, where the performance of the same simple baselines is far behind.

Our analysis reveals several key differences between NL and PL that lead to different capabilities of understanding the syntax for pre-trained models. First, programs are more structured than NL sentences. Programs usually contain hierarchical structures representing long-term dependencies between code tokens. Consequently, a large number of syntactic relation types are between distant tokens, which can be difficult to recognize for attention heads. On the contrary, the dependency relations in NL sentences mostly connect nearby token pairs, and in this case the attention heads are more capable of identifying the correct relations. Meanwhile, language models are good at recognizing keyword-based relations, such as picking the corresponding *else* keyword for an *if* token. Interestingly, we find that the inclusion of tokens such as newlines and semicolons notably affects the performance in the code domain.

Our findings suggest that existing pre-trained models perform quite differently in PL and NL domains in terms of the ability to understand syntax. Thus, directly applying training paradigms developed for NL could be suboptimal for program learning, and we consider designing better approaches to model the code structure as future work.

## 2 `CodeSyntax`: Benchmarking Code Syntax Understanding

We construct the `CodeSyntax` benchmark to evaluate the performance of language models on code syntax understanding. We focus on Python and Java languages, on which the publicly released model checkpoints of both CuBERT (Kanade et al., 2020) and CodeBERT (Feng et al., 2020) are pre-trained. We obtain the code samples from CodeSearchNet (Husain et al., 2019), which is a large-scale dataset consisting of code in different pro-

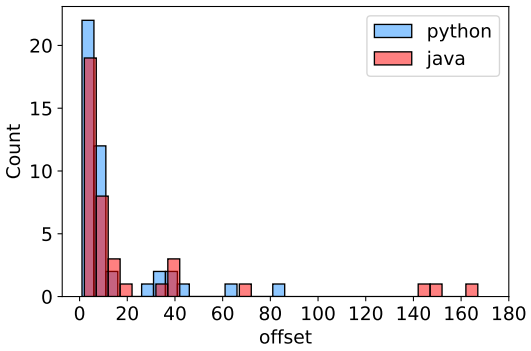
| Relation head→dependent | Count Python | Count Java | Explanation | Code Example Python | Code Example Java |
|---|---|---|---|---|---|
| Assign: target→value | 78,482 | 13,384 | Assigning a value to a target variable. | **target** = **10** | int **target** = **10**; |
| Call: func→args | 110,949 | 50,890 | Calling a function with some arguments. | **function**(**arg**) | **function**(**arg**); |
| For: for→body | 8,704 | 1,864 | A for loop repeatedly executes the body block for some iterations. | **for** target in iter:<br>  **body** | **for** (initializers; test; updaters) {<br>  **body;**<br>} |
| If: if→else | 11,024 | 5,038 | An if statement conditionally executes a body based upon some criteria. The dependent is the `else` keyword. | **if** condition:<br>  body1<br>**else**:<br>  body2 | **if** (condition) {<br>  body1;<br>} **else** {<br>  body2;<br>} |
| If: if→body | 34,250 | 22,392 | An if statement. The dependent is the body block. | **if** condition:<br>  **body1**<br>else:<br>  body2 | **if** (condition) {<br>  **body1;**<br>} else {<br>  body2;<br>} |
| If: body→orelse | 11,024 | 4,976 | An if statement. The head is the body block and the dependent is the body of the else block. | if condition:<br>  **body1**<br>else:<br>  **body2** | if (condition) {<br>  **body1;**<br>} else {<br>  **body2;**<br>} |
| While: test→body | 743 | 975 | The while loop repeatedly executes the body block as long as the specified condition is true. | while **condition**:<br>  **body** | while (**condition**) {<br>  **body;**<br>} |

Table 1: Dataset statistics of selected relation types in `CodeSyntax`. For each relation type, we highlight the head and dependent nodes in the examples in bold, with the head in blue and the dependent in red. We defer the full statistics of all relation types to Table 8 in the appendix.

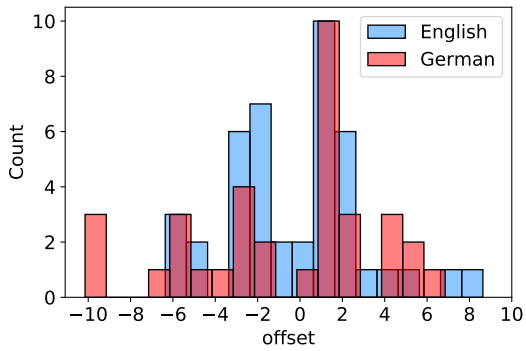
gramming languages. Its training set is also part of the pre-training data of CodeBERT, so we remove the data samples that are included in the pre-training data of either CuBERT or CodeBERT. Thus, none of the programs in `CodeSyntax` has been seen by CuBERT or CodeBERT in the pre-training phase.

In total, `CodeSyntax` contains 18,701 code samples annotated with 1,342,050 relation edges in 43 relation types for Python, and 13,711 code samples annotated with 864,411 relation edges in 39 relation types for Java. Each code sample is an entire function consisting of multiple statements, which is analogous to a paragraph in NL. Each relation corresponds to an edge in the program AST; specifically, we utilize the Python ast module (Foundation, 2021) and the Java org.eclipse.jdt.core.dom.ASTParser class (Contributors, 2014) to parse a code sample into an AST. We present some examples of relation types in Table 1, and we defer the description of all relation types to Table 8 in the appendix. More details about relation extraction are discussed in Appendix A. Note that we can easily extend the dataset to cover more languages since the workflow for extracting relations is automated and AST parsers are available for most popular programming languages.

We observe several characteristics of relations in `CodeSyntax`. First, the keywords in PL play an important role in recognizing the code structure. Specifically, some relation types have fixed keywords as the edge nodes, such as the `If:if→else` relation. Meanwhile, compared to the dependency relations in NL, the relation edges in the program AST tend to connect nodes that are much farther away from each other. As shown in Figure 3, the average offset between head and dependent nodes is no more than 10 for dependency relations in NL, while the average offset for a relation type can be more than 100 tokens in programs. Specifically, in `CodeSyntax`, there are 22 near dependency types whose average offsets are less than 10, and 12 far




(a) CodeSyntax.




(b) Natural language corpus.

Figure 3: Offset distribution of relation types in (a) CodeSyntax and (b) NL corpus. The x axis is the average positional offset distance between heads and dependents for each relation. The y axis is the number of relations that has the average offset value. See Section 3 for more details on the NL corpus.

dependency types whose average offsets are above 10.

## 3 Evaluation Setup

Do pre-trained language models capture the code structure without direct supervision of the syntactic information? To investigate this question, we evaluate several pre-trained language models without finetuning, and compare their performance in understanding the syntax for NL and PL.

**Natural language benchmark.** To compare the performance on CodeSyntax to NL syntax understanding, we construct the NL benchmark that includes English and German. Specifically, we use the English News Text Treebank: Penn Treebank Revised (Bies et al., 2015) labeled with Stanford Dependencies (de Marneffe and Manning, 2008a,b), and German Hamburg Dependency Treebank (Foth et al., 2014) labeled with Universal Dependencies (de Marneffe et al., 2021). In total, the English dataset has 48,883 sentences, 43 relation types, and 1,147,526 relation edges; the German dataset has 18,459 sentences, 35 relation types, and 307,791 relation edges.

**Attention probing approach.** Some prior works demonstrate that a Transformer architecture (Vaswani et al., 2017) pre-trained on a text corpus, such as BERT (Devlin et al., 2019), contains attention heads that specialize in certain dependency relations in NL (Raganato et al., 2018; Clark et al., 2019). Specifically, in the Transformer architecture, each vector $e_i$ for an input token is transformed into the query and key vectors $q_i$ and $k_i$ via some linear transformations, and the transformations vary among different attention heads. For the $i$-th token, the attention weight assigned to the $j$-th token is

$$\alpha_{i,j} = \frac{\exp(q_i^T k_j)}{\sum_l \exp(q_i^T k_l)}$$

The attention weight indicates how important the $j$-th token is with respect to the $i$-th token.

Typically, different attention heads learn different weights between input tokens. Therefore, to measure the correctness of recognizing a relation type r, for each edge <h, t, r> in the program AST where h is the head node and t is the dependent node, we enumerate all attention heads to compute the attention weight $\alpha_{h,t}$. If an attention head tends to assign high attention weights that connect the pair of tokens belonging to the relation type r, we consider the relation type to be captured. We defer more implementation details of attention map extraction to Appendix B.

**Metrics.** We use the unlabeled attachment score (UAS) to measure the syntax understanding performance, and we consider top-k scores with different values of k. To compute top-k scores for language models, for each attention head, given the head token h in a relation edge <h, t, r>, we compute the attention weight over all tokens in the input code, and we consider the prediction to be correct if the attention weight over the dependent token t is among the top-k tokens with the highest attention weights. For each relation, we select the best-performing attention head and use its score as the model's score for that relation. We calculate a model's average score over all relations as the final score of the model.

In NL dependency parsing problems, the dependent node t usually corresponds to a single word. However, in PL, the dependent can be a block that

contains multiple code tokens. For example, in the `If:if`→body relation, the head is the keyword `if`, while the dependent is the entire body block. Therefore, we measure three metrics. *First-token metric* and *last-token metric*: the prediction is deemed correct if it successfully predicts the first and last token of the dependent block, respectively; *Any-token metric*: the prediction is considered correct if it can predict any token within the dependent block. While we agree that these are not perfect metrics and one single metric may be incomplete, we observe that our findings generally hold for all the three metrics we evaluated. Note that the first-token metric is stricter than the any-token metric by design. Unless otherwise specified, we report the top-k scores using the first-token metric by default.

**Model architectures.** Table 2 summarizes the models evaluated in this work. For language models over code, we consider CuBERT (Kanade et al., 2020) and CodeBERT (Feng et al., 2020), and we evaluate their released pre-trained checkpoints. Both of them are based on architectures initially designed for NL. Specifically, CuBERT utilizes the BERT (Devlin et al., 2019) architecture, and CodeBERT (Feng et al., 2020) utilizes the RoBERTa (Liu et al., 2019) architecture. For NL models, we also evaluate multilingual variants of BERT and RoBERTa on the German dataset, i.e., Multilingual BERT (Pires et al., 2019) and XLM-RoBERTa (Conneau et al., 2020). Both of the two code language models are cased, so we also evaluate the cased versions of the NL models.

| Programming Languages | Natural Languages |
|---|---|
| CuBERT | BERT |
| | Multilingual BERT |
| CodeBERT | RoBERTa |
| | XLM-RoBERTa |

Table 2: Model architectures evaluated on PL and NL benchmarks. Models in the same row share the same architecture, but are pre-trained on different corpora.

**Baselines.** To examine how well the attention performs through comparisons, we design a simple offset baseline and a simple keyword baseline. The offset baseline with an offset value of $i$ always selects the token after $i$ positions of the input token as its prediction when $i > 0$, and selects $i$ positions before the input token when $i < 0$. The keyword baseline with a keyword of $key$ always predicts the next $key$ token as its prediction. In our experiments, we evaluate offset baselines with each possible offset value between 0 and 512 for PL, and -512 to 512 for NL. We use all Python and Java keywords for the keyword baselines on Python and Java datasets respectively, including tokens such as `if`, `for`, `in`, etc. To evaluate the top-k scores for baselines where $k \geq 2$, we combine k simple baselines with different offset (keyword) values to give k predictions. To select k offset (keyword) values, we repeatedly and greedily include the next value that yields the highest performance increase for the relation type under consideration.

## 4 Experiments

In this section, we present the results of pre-trained language models for both PL and NL syntax understanding tasks, and discuss the key observations that distinguish PL from NL.

### 4.1 Main Results

| Language | Model | Top-k Score k=1 | k=3 | k=10 | k=20 |
|---|---|---|---|---|---|
| Python | Offset | 43.6 | 63.7 | 87.3 | 94.9 |
| | Keyword | 15.7 | 21.9 | 23.6 | 23.8 |
| | Combined | **49.4** | **69.7** | **90.1** | **96.3** |
| | CuBERT | 39.2 | 58.4 | 81.3 | 91.4 |
| | CodeBERT | 33.1 | 51.8 | 78.6 | 89.2 |
| | RoBERTa | 34.5 | 56.9 | 82.5 | 91.3 |
| Diff (Model - Baseline) | | -10.2 | -11.3 | -8.8 | -4.9 |
| Java | Offset | 52.7 | 71.5 | 87.1 | 94.3 |
| | Keyword | 22.4 | 27.3 | 30.2 | 30.6 |
| | Combined | **60.4** | **77.2** | **90.0** | **96.1** |
| | CuBERT | 39.7 | 59.8 | 80.0 | 90.2 |
| | CodeBERT | 36.3 | 57.1 | 78.3 | 88.8 |
| | RoBERTa | 34.7 | 57.8 | 80.3 | 90.5 |
| Diff (Model - Baseline) | | -20.7 | -17.4 | -10.0 | -5.9 |

Table 3: Top-k scores for code syntax understanding. For each language, the upper block contains the results of baselines, including: (1) *Offset*: always picking the token with a fixed positional offset; (2) *Keyword*: matching a fixed keyword nearby; and (3) *Combined*: combining the best option from *Offset* and *Keyword*. Score differences are calculated as the best attention score - best baseline score for each language, where a positive value indicates that the language model surpasses the baseline.

We present our main results to compare the performance in syntactic relation understanding on PL and NL in Tables 3 and 4, respectively. First, on `CodeSyntax`, language models generally perform worse than simple offset baseline and its combination with the keyword baseline, which indicates

| Language | Model | Top-k Score k=1 | k=3 | k=10 | k=20 |
|---|---|---|---|---|---|
| English | Offset | 47.1 | 72.7 | 91.0 | 96.6 |
| | BERT-large | **64.6** | 83.2 | 96.3 | 99.3 |
| | RoBERTa-base | 62.7 | **84.3** | **96.9** | **99.4** |
| | CodeBERT | 59.3 | 79.7 | 95.2 | 99.1 |
| Diff (Model - Baseline) | | 17.5 | 11.6 | 5.9 | 2.8 |
| German | Offset | 36.3 | 58.0 | 83.1 | 95.1 |
| | Multilingual BERT | 62.6 | 81.9 | 96.5 | 99.6 |
| | XLM-RoBERTa-base | **67.4** | **85.5** | **97.1** | **99.7** |
| Diff (Model - Baseline) | | 31.1 | 27.5 | 14.0 | 4.6 |

Table 4: Top-k scores for NL syntax understanding. Note that BERT-large and CuBERT share the same model configuration, and CodeBERT and RoBERTa-base have the same model architecture. Unlike Table 3, we exclude *Keyword* and *Combined* baselines because they do not add upon the *Offset* baseline in terms of the performance.

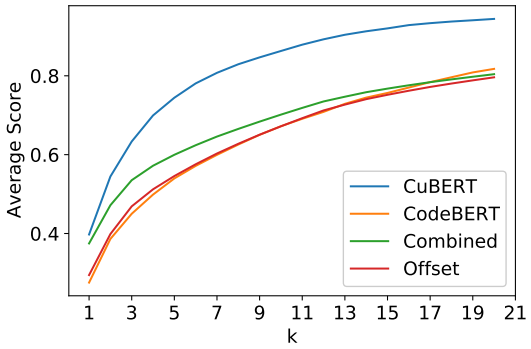
| Language | Model | Top-k Score (Any-token Metric) k=1 | k=3 | k=10 | k=20 |
|---|---|---|---|---|---|
| Python | Offset | 63.6 | 85.4 | 96.7 | 98.9 |
| | Keyword | 22.2 | 31.3 | 34.9 | 35.2 |
| | Combined | **66.8** | **88.4** | **98.2** | **99.6** |
| | CuBERT | 64.3 | 82.7 | 96.1 | 99.2 |
| | CodeBERT | 56.0 | 76.5 | 93.5 | 97.9 |
| | RoBERTa | 49.4 | 74.7 | 94.4 | 98.5 |
| Diff (Model - Baseline) | | -2.5 | -5.7 | -2.1 | -0.4 |
| Java | Offset | 69.4 | 86.5 | 96.8 | 99.0 |
| | Keyword | 40.9 | 44.9 | 46.7 | 47.0 |
| | Combined | **75.7** | **90.0** | **98.2** | **99.6** |
| | CuBERT | 72.1 | 87.4 | 97.5 | 99.5 |
| | CodeBERT | 62.7 | 81.1 | 93.9 | 97.6 |
| | RoBERTa | 59.8 | 81.4 | 94.9 | 98.4 |
| Diff (Model - Baseline) | | -3.6 | -2.6 | -0.7 | -0.1 |

Table 5: Top-k scores for code syntax understanding using the any-token metric.

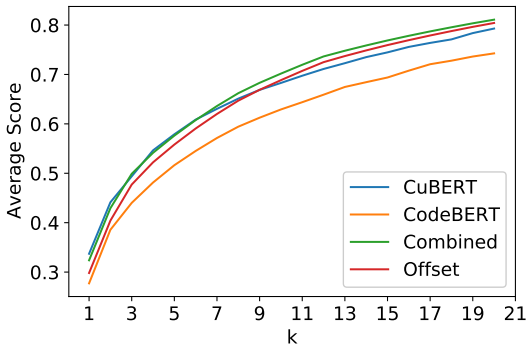
that the attention heads of PL pre-trained models do not effectively capture the syntactic relations in programs. The comparison between CodeBERT and RoBERTa further shows that pre-training on a large-scale code corpus, in addition to the text corpus for RoBERTa pre-training, does not yield a notably better understanding of code syntax. In comparison, language models substantially outperform offset baselines in recognizing the dependency relations in NL, demonstrating that the attention heads learn to be specialized for different relation types via large-scale pre-training on text.

Meanwhile, we present the any-token results on `CodeSyntax` in Table 5. Although the best combined baseline still outperforms language models, the performance gap shrinks drastically. In particular, CuBERT achieves better scores than the offset baseline, and the improvement on Java is more notable. We defer the full results of different top-k scores on both PL and NL benchmarks to Appendix D. In the following sections, we discuss the key factors that affect prediction performance.

### 4.2 Case Studies: The Effect of Keywords




(a) With semicolons (default).




(b) Without semicolons.

Figure 4: Top-k scores for Java syntax understanding using the last-token metric.

To examine why the offset baseline outperforms CodeBERT and CuBERT, and why the relative performance differences get smaller when using the any-token metric, we conducted case studies and error analysis in Section 4.2 and Section 4.3, which both quantitatively and qualitatively categorize the error patterns.

Firstly, we investigate the most frequently attended code tokens, and we observe that the attention heads tend to recognize the reserved tokens and keywords in PL. For example, CuBERT and CodeBERT get an improved score on Java because the semicolon token is part of the ground truth dependent node, which is a popular token attended to by language models. Based on this observation, we perform an ablation study on the presence of the semicolon in ground truth annotations. When the semicolon tokens are removed from ground

truth dependent nodes, we also disable the language models to attend to semicolons in the input code. Since the semicolon appears at the end of each Java statement, here we compute the last-token score which may be significantly affected by semicolons. As shown in Figure 4, CuBERT substantially outperforms baselines when semicolons are included in the ground truth labels. On the other hand, CuBERT reaches lower scores than baselines when semicolons are excluded from ground truth labels and predictions. The comparison suggests that attention heads are more capable of identifying frequent keywords in the model input. We defer the full ablation study on both Python and Java to Appendix F.

We further discuss the breakdown results with respect to relation types, and we select some representative relations for Python that highlight the performance differences between CuBERT and the offset baseline in Table 6. First, the attention is highly capable of performing keyword matching, which leads to decent accuracy on relations that connect popular keywords, such as `If:if→else`. However, when the head and dependent tokens are diverse, it becomes challenging for the language model to recognize the relation. For example, in relation types `Assign:target→value` and `Call:func→args`, both head and dependent nodes can take various identifier names defined by different programmers. In particular, CuBERT can not effectively utilize the relative positions of tokens to learn the relations, even if the dependent node is near the head node. In such situations, the offset baseline with a fixed offset value of 2 already surpasses the pre-trained model. The full breakdown results of all relation types on both Python and Java can be found in Appendix G.

| Relation | Score | | Offset | Diff |
|---|---|---|---|---|
| | CuBERT | Offset | | |
| If:if→else | 92.7 | 5.7 | 17 | 87.1 |
| If:body→orelse | 29.2 | 7.1 | 12 | 22.0 |
| If:if→body | 31.5 | 23.1 | 7 | 8.4 |
| For:for→body | 30.4 | 32.7 | 7 | -2.3 |
| Assign:target→value | 39.8 | 71.2 | 2 | -31.4 |
| While:test→body | 16.2 | 48.5 | 4 | -32.4 |
| Call:func→args | 59.3 | 93.2 | 2 | -33.9 |

Table 6: The comparison of top-1 first-token scores between CuBERT and the offset baseline with the best fixed offset for selected relation types on Python dataset.

### 4.3 Error Analysis

| Relation | Error Situation | Count | |
|---|---|---|---|
| | | Python | Java |
| If: if→else | Nested if statements or multiple if statements close to each other. | 34 | 42 |
| | Predicts other keywords inside body block, e.g., `if` and `while`. | 11 | 4 |
| | Other. | 5 | 4 |
| If: body→orelse | Predicts another token with the same name as head token itself. | 38 | 21 |
| | Predicts keywords inside body block, e.g., `if`, `;` and `while`. | 0 | 14 |
| | Predicts a long string or docstring. | 7 | 7 |
| | Other. | 5 | 8 |
| If: if→body | Predicts blank space or tab. | 32 | 0 |
| | Predicts { or }. | 0 | 27 |
| | Predicts `return`. | 0 | 19 |
| | Predicts `\n` or `:`. | 15 | 0 |
| | Other. | 3 | 4 |
| For: for→body | Predicts blank space or tab. | 46 | 0 |
| | Predicts { or }. | 0 | 29 |
| | Other. | 4 | 21 |
| Assign: target→value | Predicts a token that comes before =, e.g. `a[0]` and `a.b` | 16 | 30 |
| | Predicts =. | 22 | 5 |
| | Other. | 12 | 15 |
| While: test→body | Predicts a token in the test block. | 48 | 36 |
| | Other. | 2 | 14 |
| Call: func→args | Predicts ( or ). | 45 | 37 |
| | Predicts another token with the same name as head token itself. | 0 | 10 |
| | Other. | 5 | 3 |

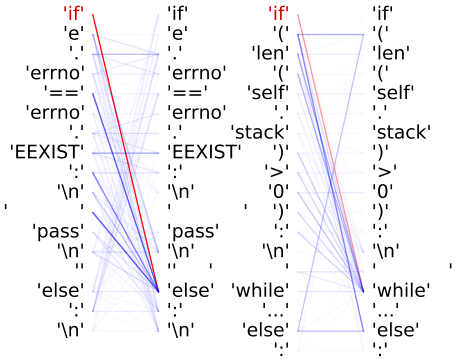
Table 7: Error analysis using CuBERT.

To categorize the wrong predictions of the attention, we manually examine 50 error cases for each relation selected in Table 6, and present the error situations in Table 7. Again, we observe that the attention often incorrectly selects frequently occurring tokens such as brackets. Moreover, the model has difficulty capturing the hierarchical code structure, thus it often attends to nearby keywords regardless of logical code blocks.

Take the relation `If:if→else` as an example, on which the language model generally achieves the best performance. Shown in Figure 5 are two sample `if`-statements, where the first one does not contain nested flow control blocks while the second one contains a keyword `while` inside the `if`-body. "..." denotes that some code is omitted. Visualizing their corresponding attention weights of the attention head that performs the best on the relation `If:if→else`, we observe that the attention head

```
if e.errno == errno.EEXIST:
    pass
else:
    ...
if (len(self.stack) > 0):
    while ...
        ...
else:
    ...
```

(a) Python code.




(b) Attention weights.

Figure 5: Two sample cases for the relation `If:if→else` and corresponding attention weights of CuBERT's head 17-2.

correctly attends to the `else` token in the first example, while it wrongly attends to the `while` token inside the `if`-body in the second example. More examples like these can be found in Appendix E.

## 5 Related Work

Transformer-based language models have been widely used for natural language processing (Devlin et al., 2019; Liu et al., 2019; Wang et al., 2020, 2021; Shen et al., 2022; Wang et al., 2022). Hewitt and Manning (2019) show that syntax trees are implicitly embedded in BERT's word representation space via a structural probe. Another line of work studies what is learned by the attention in language models (Clark et al., 2019; Raganato et al., 2018; Voita et al., 2019; Michel et al., 2019; Vig, 2019; Burns et al., 2018; Marecek and Rosa, 2018; Voita et al., 2018). In particular, Clark et al. (2019) evaluate the attention heads of BERT on dependency parsing tasks using the English Penn Treebank corpus, where the attention significantly outperforms offset baselines. On the contrary, we demonstrate that attention-based models largely perform worse than offset baselines on code syntax understanding.

The success of Transformer-based models for natural language processing leads to their application in the PL domain (Kanade et al., 2020; Feng et al., 2020; Rozière et al., 2020, 2021; Clement et al., 2020; Dehghani et al., 2019). Chen et al. (2021) evaluate the model performance by measuring the functional correctness on unit tests. Chirkova and Troshin (2021) empirically shows that Transformers can utilize syntactic information to make predictions in some code processing tasks, while we analyze attention's ability to understand syntactic relations. Karmakar and Robbes (2021) probe pre-trained models on four code understanding tasks. They focus more on code classification, e.g., they train a classifier for predicting the AST node tag and the code length. On the contrary, we probe the attention heads for syntactic relation understanding, and we aim to present a comprehensive study of the differences between pre-trained language models on NL and PL for capturing the syntax structures.

There have been some efforts that try to take code structure into account during pre-training of Transformer-based models for code. For example, GraphCodeBERT (Guo et al., 2021) utilizes data flow for pretraining; i.e., the relation of "where-the-value-comes-from" for variables. On our Python benchmark, GraphCodeBERT achieves a top-1 first-token score of 39.3, which is better than 33.1 of CodeBERT, and comparable to 39.2 of CuBERT. However, such a score is still worse than 43.6 of the offset baseline. This trend is consistent when evaluating with other metrics. These results show that pre-training on data flow helps improve the model's ability to understand code syntax, but there is still large room for improvement.

## 6 Conclusion

In this work, we introduce `CodeSyntax`, a large-scale benchmark for measuring the performance of code syntax understanding. Based on `CodeSyntax`, we conduct the first comprehensive study to analyze the capability of pre-trained language models on understanding the code syntactic structures without further finetuning. We demonstrate that while the attention heads of pre-trained language models are able to identify dependency relations in NL to some extent, they have difficulty recognizing the syntactic relations in programs. Pre-trained models even generally perform worse than simple offset baselines, and they tend to attend to frequently occurring nearby tokens without taking the hierarchical code structure into consideration.

We also analyze the differences between NL and

PL from the perspectives of pre-trained models. Our evaluation suggests that PL has unique characteristics that distinguish them from NL, such as the long-term dependency between code tokens, and the hierarchy in the syntactic structures. Therefore, simply taking a program as a token sequence is insufficient for modeling the program structure, which could eventually limit the potential of language models for code understanding tasks. We consider developing new model architectures and pre-training algorithms to leverage and represent the code structure and dependency graph as important future work.

## 7 Limitations

For the limitations of our benchmark, the gold annotations are based on the AST parsers. Adding new programming languages whose parsers are unavailable will require additional labeling efforts. A limitation in our experimental setup is that we have only benchmarked six models across two kinds of natural languages and programming languages. Finally, the main focus of our study is to probe the language models for code understanding. As a result, we have not proposed models that could deal with the code syntax in natural language and programming language applications. Future work could include developing such models that capture both semantics and structures.

## 8 Ethical Considerations

We hereby acknowledge that all of the co-authors of this work are aware of the provided *ACM Code of Ethics* and honor the code of conduct. The followings give the aspects of both our ethical considerations and our potential impacts to the community. This work creates a benchmark to test the code syntax understanding of pre-trained language models. Instead of natural language, the programming language is used for pre-training. We do not anticipate the production of harmful outputs after using our benchmark and existing models, especially towards vulnerable populations.

## 9 Environmental Considerations

We use several pre-trained language models. According to the estimation in (Strubell et al., 2019), pre-training a model with a similar size as used in the work costs 1,507 kWh·PUE and emits 1,438 lb $CO_2$. This work focuses on inference. Therefore, our energy cost and $CO_2$ emissions are relatively small.


## Acknowledgements

We would like to thank the anonymous reviewers for their suggestions and comments. This material is in part based upon work supported by Berkeley DeepDrive and Berkeley Artificial Intelligence Research.


## References


Ann Bies, Justin Mott, and Colin Warner. 2015. English news text treebank: Penn treebank revised. Philadelphia: Linguistic Data Consortium.

Kaylee Burns, Aida Nematzadeh, Erin Grant, Alison Gopnik, and Thomas L. Griffiths. 2018. Exploiting attention to reveal shortcomings in memory models. In *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP 2018, Brussels, Belgium, November 1, 2018*, pages 378–380. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Nadezhda Chirkova and Sergey Troshin. 2021. Empirical study of transformers for source code. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 703–715, New York, NY, USA. Association for Computing Machinery.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. 2019. What does bert look at? an analysis of bert's attention. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. The Association for Computational Linguistics.

Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 9052–9065. Association for Computational Linguistics.

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 8440–8451. Association for Computational Linguistics.

Eclipse Contributors. 2014. Rational software architect realtime edition 9.5.0. `https://www.ibm.com/docs/en/rsar/9.5?topic=SS5JSH_9.5.0/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/package-use.html`.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008a. Stanford dependencies manual.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008b. The stanford typed dependencies representation. In *Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation@COLING 2008, Manchester, UK, August 23, 2008*, pages 1–8. Coling 2008 Organizing Committee.

Marie-Catherine de Marneffe, Christopher D. Manning, Joakim Nivre, and Daniel Zeman. 2021. Universal dependencies. *Comput. Linguistics*, 47(2):255–308.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. 2019. Universal transformers. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*.

Kilian A. Foth, Arne Köhn, Niels Beuck, and Wolfgang Menzel. 2014. Because size does matter: The hamburg dependency treebank. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation, LREC 2014, Reykjavik, Iceland, May 26-31, 2014*, pages 2326–2333. European Language Resources Association (ELRA).

Python Software Foundation. 2021. Python 3.10.0 documentation, ast — abstract syntax trees. `https://docs.python.org/3/library/ast.html`.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

John Hewitt and Christopher D. Manning. 2019. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4129–4138. Association for Computational Linguistics.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR.

Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code? In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1332–1336. IEEE.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

David Marecek and Rudolf Rosa. 2018. Extracting syntactic trees from transformer encoder self-attentions. In *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP 2018, Brussels, Belgium, November 1, 2018*, pages 347–349. Association for Computational Linguistics.

Paul Michel, Omer Levy, and Graham Neubig. 2019. Are sixteen heads really better than one? In *Advances in Neural Information Processing Systems*

*32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14014–14024.

Telmo Pires, Eva Schlinger, and Dan Garrette. 2019. How multilingual is multilingual bert? In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4996–5001. Association for Computational Linguistics.

Alessandro Raganato, Jörg Tiedemann, et al. 2018. An analysis of encoder representations in transformer-based machine translation. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. The Association for Computational Linguistics.

Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Baptiste Rozière, Jie M. Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *CoRR*, abs/2110.06773.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Jianhao Shen, Chenguang Wang, Linyuan Gong, and Dawn Song. 2022. Joint language semantic and structure embedding for knowledge graph completion. In *COLING*.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. In *ACL*, pages 3645–3650.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Jesse Vig. 2019. Visualizing attention in transformer-based language representation models. *CoRR*, abs/1904.02679.

Elena Voita, Pavel Serdyukov, Rico Sennrich, and Ivan Titov. 2018. Context-aware neural machine translation learns anaphora resolution. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 1264–1274. Association for Computational Linguistics.

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 5797–5808. Association for Computational Linguistics.

Chenguang Wang, Xiao Liu, Zui Chen, Haoyun Hong, Jie Tang, and Dawn Song. 2021. Zero-shot information extraction as a unified text-to-triple translation. In *EMNLP*.

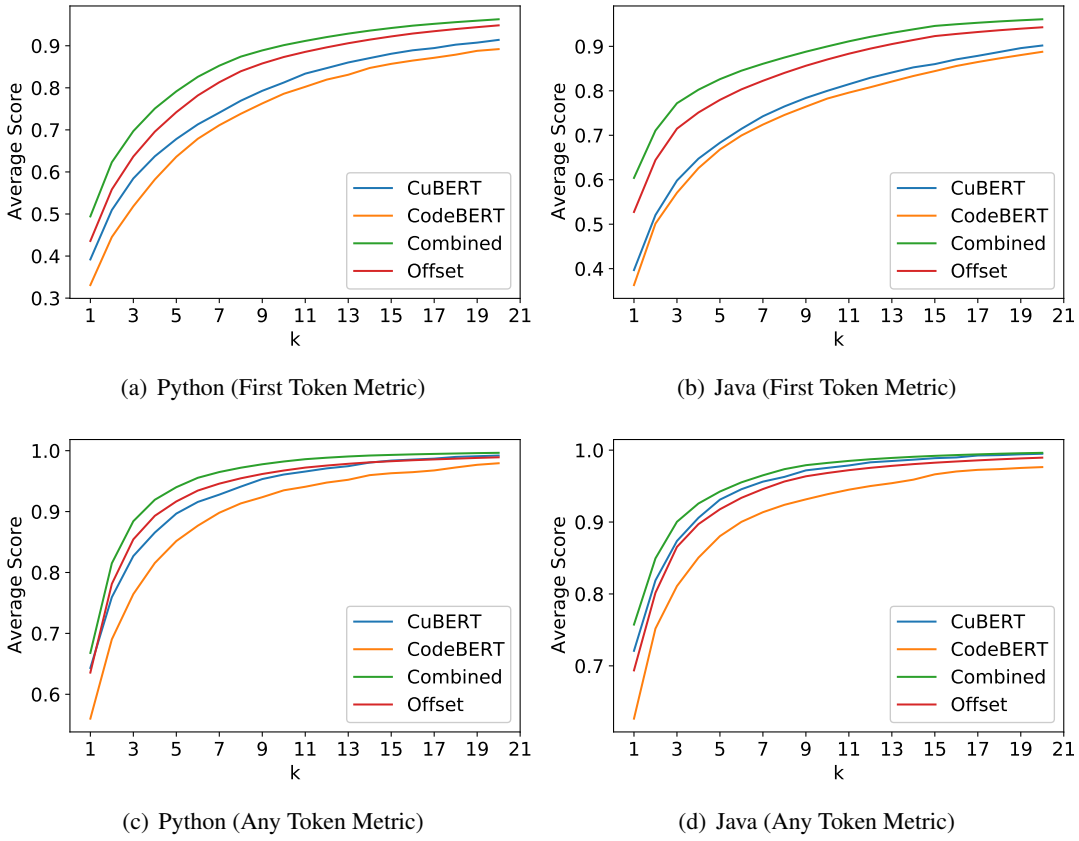
Chenguang Wang, Xiao Liu, Zui Chen, Haoyun Hong, Jie Tang, and Dawn Song. 2022. DeepStruct: Pretraining of language models for structure prediction. In *ACL*.

Chenguang Wang, Xiao Liu, and Dawn Song. 2020. Language models are open knowledge graphs. *arXiv preprint arXiv:2010.11967*.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

## A More Details on `CodeSyntax` Construction

Since the code search net dataset does not come with syntactic relation labels, we come up with a way of extracting syntactic relations. We first utilize python's tokenize module and javalang module to produce code tokens from source code, and then label these code tokens with syntactic relations by using AST parsers on source code. We utilize Python ast module (Foundation, 2021) and Java org.eclipse.jdt.core.dom.ASTParser class (Contributors, 2014) to parse source code into ast nodes. The AST structure captures syntactical relations. An AST node has children AST nodes and a name that denotes its class. We use the class of the node as label and children nodes as dependents and heads when generating annotations. For example, the source code `A = B`, which means assigning value `B` to target variable `A`, is parsed into the




(a) Python (First Token Metric)

(b) Java (First Token Metric)

(c) Python (Any Token Metric)

(d) Java (Any Token Metric)

Figure 6: PL Top-k Scores On Test Set

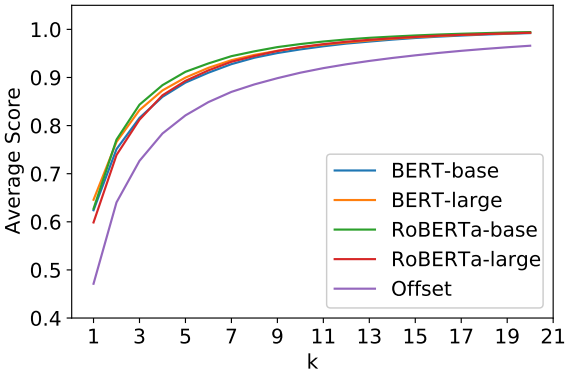



Figure 7: English Top-k Scores

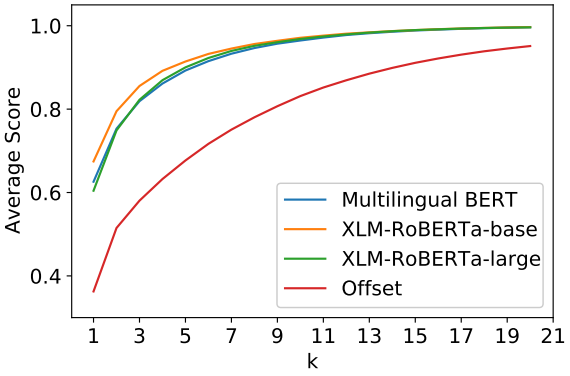



Figure 8: German Top-k Scores

AST node `Assign(targets=[Name(id='A')], value=Name(id='B'))`. It gives us a syntactic relation whose head is `A` and dependent is `B`, annotated with the relation type label `Assign`. Full statistics of `CodeSyntax` are displayed in Table 8.

## B More Details on Attention Map Extraction for Code Language Models

Our experiments follow the work of Clark et al. (2019). They evaluate the attention heads of BERT on dependency parsing tasks on an English dataset, while we extend the work to the PL domain. We adopt and extend some of their code, such as the functions for extracting attention from BERT and plotting attention weights. The main differences between our work and theirs are that we construct a novel dataset for syntax understanding tasks for PL and come up with related evaluation metrics to accommodate the characteristics of PL.

### B.1 Model Input

Each of our code samples is an entire Python or Java function. To prepare the input to be fed to the models, we run CuBERT and CodeBERT tokeniza-

tion to obtain sequences of input ids for each code sample. We insert a `[CLS]` token at the beginning and append a `[SEP]` token at the end. If the input length is longer than 512 tokens (the maximum number of tokens allowed), we discard that code sample. We never split a long code sample into several input sentences because the span of some dependency relations is very long within a function. For example, for an `if` statement, the `else` block may be far away from the keyword `if`. If we split them into two input sentences, then attention will not be able to understand and predict the relation between them. To avoid uncommon data points, we remove a code sample from both CuBERT and CodeBERT's input if it is longer than 512 tokens after either one of CuBERT or CodeBERT's tokenization.

### B.2 Token Alignment And Word-level Attention

BERT uses WordPiece tokenization (Wu et al., 2016) and RoBERTa uses byte-level Byte-Pair Encoding (BPE) (Sennrich et al., 2016), which may split a word into several subtokens. Additionally, CuBERT imposes some special rules when producing program vocabulary. However, our dataset's labels use code tokens generated by the tokenize module and the javalang module. Therefore, there exists a need to align CuBERT/CodeBERT subtokens with code tokens in order to evaluate the models on our dataset. We first generate such an alignment that maps each code token to a set of CuBERT/CodeBERT subtokens, and then convert the original subtoken-level attention to word-level attention. We follow (Clark et al., 2019) to combine the attention weights of subtokens, i.e., we sum up their attention weights.

## C More Reproducibility Information

Here we provide more information according to the EMNLP 2022 Reproducibility Criteria.

- Train/validation/test splits for datasets used: We do not finetune the pre-trained models on our benchmark. The validation set of `CodeSyntax` contains the code samples that come from the validation set of CodeSearchNet, and our test set contains the samples from CodeSearchNet's test set. We use our test partition to probe the pre-trained attention heads while the validation set is not used.
- Number of parameters in each model: Cubert and BERT-large have 340M parameters. CodeBert and RoBERTa-base have 125M parameters. XLM-RoBERTa-base has 250M parameters. Multilingual BERT-base has 110M parameters.
- The average runtime for each model or algorithm: Running the pipeline to construct the `CodeSyntax` dataset takes about four hours assuming that dependencies and required datasets have been downloaded. The algorithm to probe a pre-trained model on one programming language of `CodeSyntax` takes about twelve hours on our machine using one Nvidia 1080Ti GPU.

## D More Results on Top-k Scores

PL top-k scores are plotted in figure 6. NL scores are plotted in figure 7 (English) and figure 8 (German).

## E Examples of Correct and Incorrect Predictions

In this section, we present some visualization examples where attention correctly or incorrectly predicts the dependents. The heads chosen in these examples are the best-performing heads of CuBERT evaluated using the first-token metric. We feed the entire function as input to the transformer, however, we only present relevant snippets here for simplicity. In the source code displayed, "..." denotes that the remaining part of the code is omitted. As a result, the attention from a token may not sum up to one in these figures because the rest of the function is omitted.

**Relation `Call: func` → `args`.** The corresponding attention weights are visualized in Table 9 for Python and 10 for Java.

- Python correct case.

```
n = len(x)
n_fft = len(win_sq)
```

  Attention correctly predicts the arguments `x` and `win_sq`, respectively.
- Python error case.

```
re.findall(pattern,text)
```

  The function `findall` is called. The correct prediction should be the first argument, which

is `pattern`; however, attention incorrectly predicts the parenthesis `(`.

- Java correct case.

```
subscriber.onError(ex);
```

The token `ex` has the largest weight in attention, which is a correct prediction.

- Java error case.

```
isBug(error)
```

The function `isBug` is called. The correct prediction should be the argument, `error`; however, attention incorrectly predicts `)`.

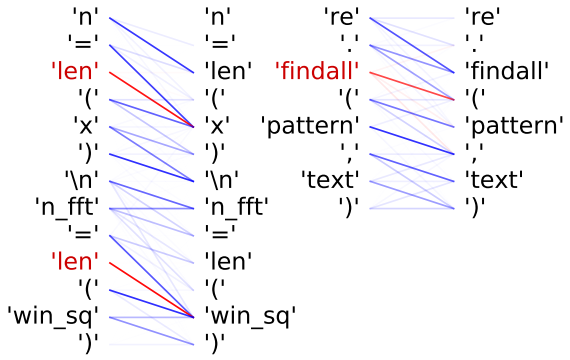



Figure 9: Python Head 15-11 Call: func → args.

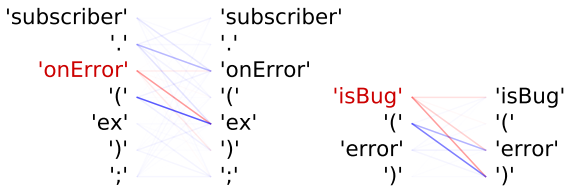



Figure 10: Java Head 19-9 Call: func → args

**Relation `Assign`: `target` → `value`.** The corresponding attention weights are visualized in Table 11 for Python and 12 for Java.

- Python correct case.

```
value = round(value,
              precision)
```

The assigned value `round` is correctly predicted.

- Python error case.

```
d["_text"] = r.text
```

The value assigned is `r.text`, but attention incorrectly predicts `[`.

- Java correct case.

```
int p = parallelism();
```

Attention has the largest weight for the head token `parallelism`, which correctly predicts the relation.

- Java error case.

```
this.defaultProcessor
              = processor;
```

The value assigned is `processor`, but attention incorrectly predicts `;`.

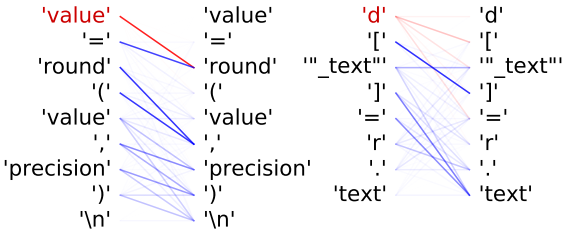



Figure 11: Python Head 15-10 Assign: target → value

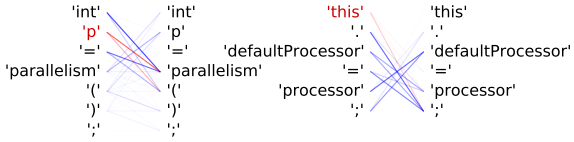



Figure 12: Java Head 20-10 Assign: target → value

**Relation `If`: `if` → `else`.** The corresponding attention weights are visualized in Table 13 for Java.

- Java correct case.

```
if (t instanceof Error) {
    throw (Error) t;
} else {
    ...
```

It correctly identifies the keyword `else`.

- Java error case.

```
if(error.addThrowable(ex)) {
    if ...
} else {
    ...
```

There is another `if` statement inside the body of the first if statement. The correct prediction should be keyword `else`, but it predicts the inner `if`.

**Relation `For`: `for` → `body`.** The corresponding attention weights are visualized in Table 14 for Python and 15 for Java.

- Python correct case.

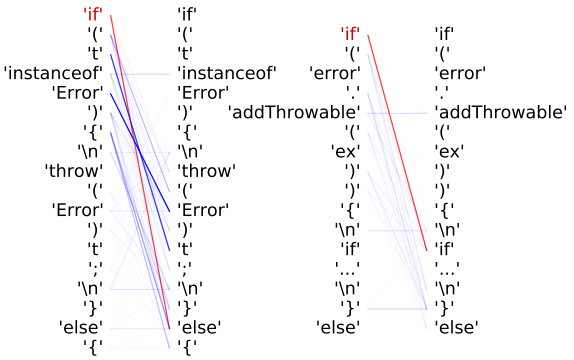



Figure 13: Java Head 9-10 If: if → else

```
for el in predictions:
    if 0 in el:
        ...
```

- Python error case.

```
for pass_ in
        self.working_list:
    ret.append(...
```

The correct prediction should be the first token within the body, which is `ret`; however, attention incorrectly predicts the blank space " " before `ret`.

- Java correct case.

```
for(BehaviorSubscription<T>
                s : array) {
    if (...
```

- Java error case.

```
for (;;) {
    CacheSubscription ...
```

The correct prediction should be the first token within the body, which is `CacheSubscription`; however, attention incorrectly predicts `{`.

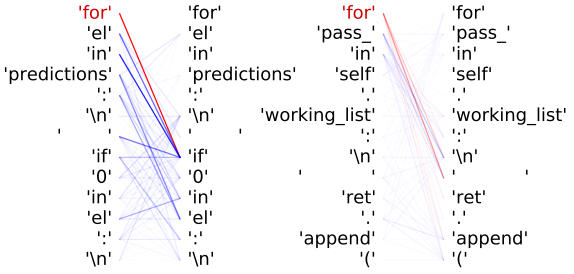



Figure 14: Python Head 18-4 For: for → body

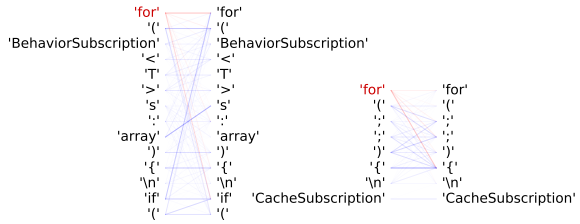



Figure 15: Java Head 16-5 For: for → body

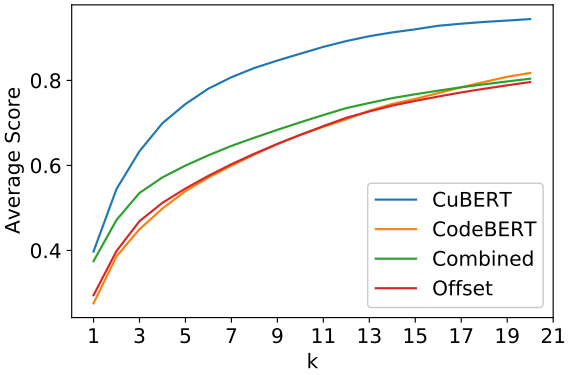
## F More Results on the Ablation Study of Delimiter Tokens

The ablation study on java dataset is shown in Figure 17 (any-token metric) and Figure 16 (last-token metric). Results with first-token metric are not affected at all because semicolons and newlines are never used as the first token of dependents. We found that attention performs very well with last-token metric because it can find semicolons and newlines.

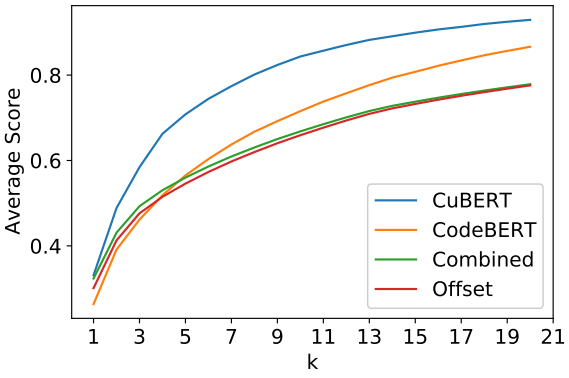
The ablation study on python dataset is shown in Figure 18 (any-token metric) and Figure 19 (last-token metric).

## G More Breakdown Results on Different Relation Types

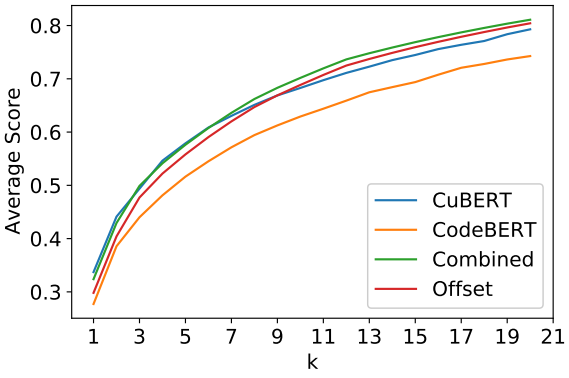
More results on comparisons of top-1 scores between CuBERT and the offset baseline are presented in Tables 9, 10, 11, and 12.




(a) Original

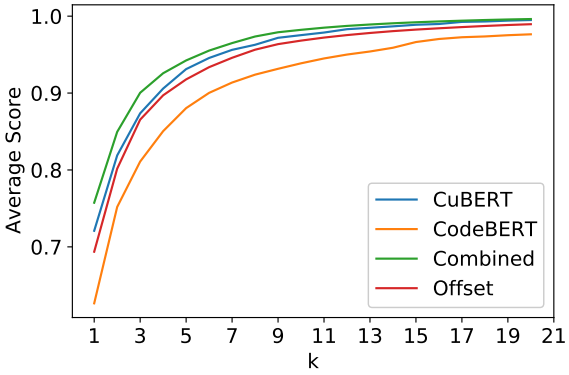



(b) With Newline

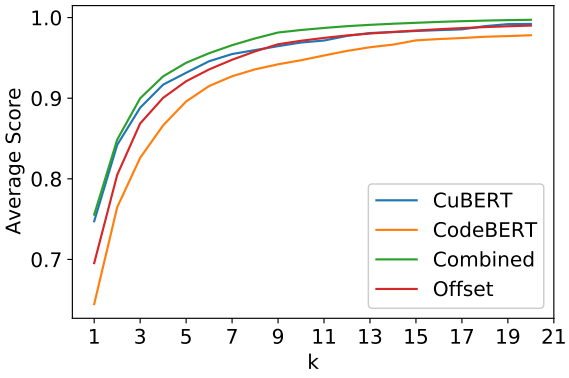



(c) Without Semicolons

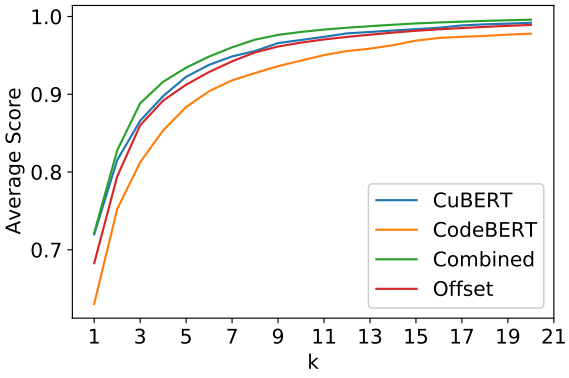
Figure 16: Top-k scores for Java syntax understanding using the last-token metric.




(a) Original

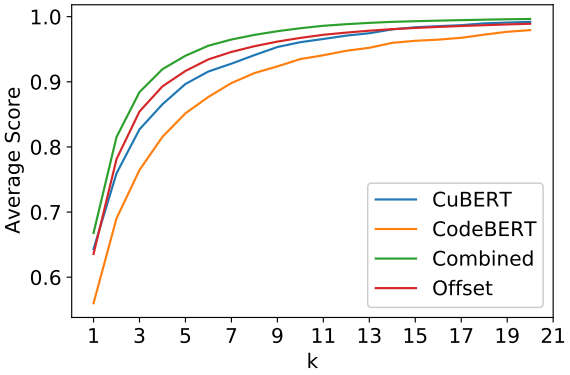



(b) With Newline

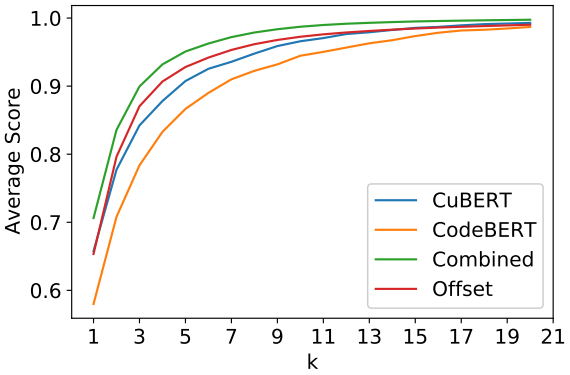



(c) Without Semicolons

Figure 17: Ablation Study (Java Any-Token Metric).

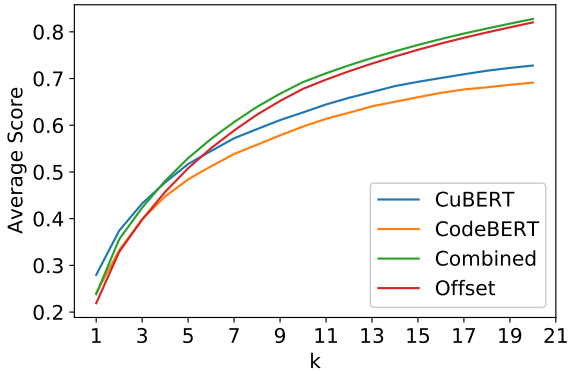



(a) Original

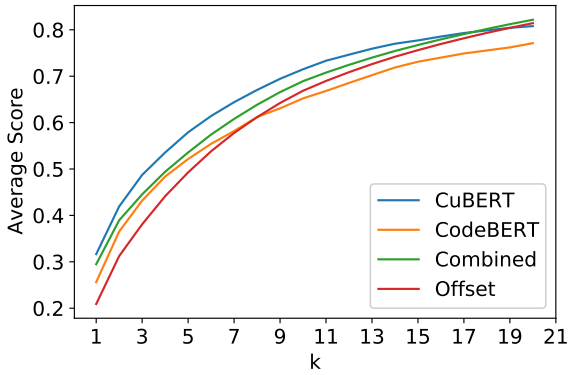



(b) With Newline

Figure 18: Ablation Study (Python Any-Token Metric).




(a) Original




(b) With Newline

Figure 19: Ablation Study (Python Last-Token Metric).

| Relation:head→dependent | Count Python | Count Java | Explanation | Code Example Python | Code Example Java |
|---|---|---|---|---|---|
| Assign:target→value | 78482 | 13384 | Assigning a value to a target variable. | **target** = **10** | int **target** = **10**; |
| Attribute:value→attr | 158797 | 84215 | Accessing the attribute (member field or member function) of an value. | **value**.**attribute** | **line**.**setLength**(2); |
| AugAssign:target→value | 3150 | / | An assignment augmented with an operation. For Java, this case is included in Assign:target→value. | **x** += **2** | / |
| BinOp:left→right | 26035 | / | A binary operation. | **a** + **b** | / |
| BoolOp:value→value | 5783 | / | A boolean operation | **True** or **False** | / |
| Call:args→keywords<br>Call:func→args<br>Call:func→keywords | 9256<br>110949<br>16274 | /<br>50890<br>/ | Calling a function with some arguments (and keywords). | **function**(**arg**, **key=1**) | **function**(**arg**); |
| Compare:left→comparator | 25852 | / | A comparison between values. | **a** < **b** | / |
| Dict:key→value | 7787 | / | Initializing a dictionary. | {**count** : **10** } | / |
| DictComp:key→generator<br>DictComp:key→value<br>DictComp:value→generator | 359<br>359<br>359 | /<br>/<br>/ | Dictionary comprehension. | {**i**: **2*i** for **i in list**} | / |
| Do:body→test<br>Do:do→body<br>Do:do→test | /<br>/<br>/ | 38<br>45<br>38 | The do loop repeatedly executes the body block as long as the condition is true. | / | **do**<br>  **statement;**<br>while (**condition**); |
| For:for→body<br>For:for→initializers<br>For:for→iter<br>For:for→target<br>For:for→test<br>For:for→updaters<br>For:initializers→body<br>For:initializers→test<br>For:initializers→updaters<br>For:iter→body<br>For:target→body<br>For:target→iter<br>For:test→body<br>For:test→updaters<br>For:updaters→body | 8704<br>/<br>8704<br>8704<br>/<br>/<br>/<br>/<br>/<br>8704<br>8704<br>8704<br>/<br>/<br>/ | 1864<br>1650<br>/<br>/<br>1296<br>1682<br>1781<br>1286<br>1670<br>/<br>/<br>/<br>1789<br>1678<br>1685 | A for loop repeatedly executes the body block for some iterations. | **for target** in **iter**:<br>  **body** | **for** (**initializers**;<br>  **test**; **updaters**) {<br>  **body;**<br>} |
| GeneratorExp:elt→generator | 685 | / | A generator expression. | (**2*i** for **i in list**) | / |
| If:body→orelse<br>If:if→body<br>If:if→else<br>If:if→test<br>If:test→body<br>If:test→orelse | 11024<br>34250<br>11024<br>34250<br>34250<br>11024 | 4976<br>22392<br>5038<br>19323<br>22392<br>5007 | An if statement conditionally executes a body based upon some criteria. | **if condition**:<br>  **body1**<br>**else**:<br>  **body2** | **if** (**condition**) {<br>  **body1;**<br>} **else** {<br>  **body2;**<br>} |
| IfExp:body→orelse<br>IfExp:body→test<br>IfExp:test→body<br>IfExp:test→orelse | 1262<br>1262<br>/<br>1262 | 1173<br>/<br>1218<br>1173 | An if expression (conditional expression). | **x** if **condition** else **y** | (**condition**) ? **x** : **y** |
| InfixExpr:left→right | / | 35170 | Infix expression of the form leftOperand InfixOperator right-Operand. | / | **a** + **b** |
| InstanceofExpr:expr→type | / | 1367 | Checking whether an expression is some type. | / | **input** instanceof **String** |
| LabeledStatement:label→body | / | 10 | A statement labeled with an identifier. | / | **Identifier** : **Statement** |
| ListComp:elt→generator | 2691 | / | List comprehension. | [**x** for **x in list1**] | / |
| SetComp:elt→generator | 67 | / | Set comprehension. | {**x** for **x in list1**} | / |
| Slice:lower→upper | 731 | / | A slice used in subscript of lists. | A[**2**:**6**] | / |
| Subscript:value→slice | 39271 | 4555 | Accessing parts of an array or data structure through subscript. | **A**[**2**:**6**] | **A**[**0**] |

Continued on next page.

Continued from previous page.

| Relation:head→dependent | Count Python | Count Java | Explanation | Code Example Python | Code Example Java |
|---|---|---|---|---|---|
| Switch:expr→statement<br>Switch:switch→expr<br>Switch:switch→statement | /<br>/<br>/ | 385<br>320<br>385 | A switch statement chooses a branch to execute based upon conditions. | / | **switch** (**inputExpr**) {<br>  **Statement;**<br>} |
| Try:body→finalbody<br>Try:body→handler<br>Try:body→orelse<br>Try:handler→finalbody<br>Try:handler→orelse | 135<br>3020<br>181<br>48<br>181 | 474<br>2011<br>/<br>186<br>/ | A try statement for handling exceptions. | try:<br>  **body1**<br>except **Exception:**<br>  **body2**<br>else:<br>  **body3**<br>finally:<br>  **body4** | try {<br>  **body1;**<br>} **catch (Exception e)** {<br>  **body2;**<br>} finally {<br>  **body3;**<br>} |
| While:test→body<br>While:while→body<br>While:while→test | 743<br>743<br>743 | 975<br>975<br>416 | The while loop repeatedly executes the body block as long as the condition is true. | **while condition**:<br>  **body** | **while** (**condition**) {<br>  **body;**<br>} |
| With:item→body | 1239 | / | A with statement with built-in context manager. | with **open("file") as f**:<br>  **content = f.read()** | / |
| children:parent→child | 652417 | 569499 | Any pair of AST nodes that are parent and child in the parse tree. | / | / |
| comprehension:target→iter | 3881 | / | A for clause to iterate over some sequences. | [x for **x** in **list1**] | / |

Table 8: Full dataset statistics table of `CodeSyntax`. For each relation type, we highlight the head and dependent nodes in the examples in bold, with the head in blue and the dependent in red. If a node can be either head or dependent in different relations, we color it in green. For more explanation about the syntax, please refer to the documentation of Python ast module (Foundation, 2021) and Java org.eclipse.jdt.core.dom.ASTParser (Contributors, 2014)

| Relation | Score | | Offset | Difference |
|---|---|---|---|---|
| | CuBERT | Offset | | |
| If:if→else | 92.7 | 5.7 | 17 | 87.1 |
| IfExp:body→orelse | 46.4 | 13.6 | 6 | 32.8 |
| Try:body→handler | 39.1 | 7.9 | 8 | 31.3 |
| If:body→orelse | 29.2 | 7.1 | 12 | 22.0 |
| BoolOp:value→value | 33.3 | 22.4 | 2 | 10.9 |
| Try:body→finalbody | 20.5 | 10.3 | 7 | 10.3 |
| If:if→body | 31.5 | 23.1 | 7 | 8.4 |
| Call:func→keywords | 38.0 | 34.1 | 2 | 4.0 |
| If:test→orelse | 7.5 | 5.1 | 18 | 2.4 |
| Compare:left→comparator | 46.5 | 45.7 | 2 | 0.8 |
| If:if→test | 98.8 | 98.8 | 1 | -0.0 |
| For:for→target | 99.4 | 99.4 | 1 | 0.0 |
| While:while→test | 99.3 | 99.3 | 1 | 0.0 |
| Try:body→orelse | 10.5 | 10.5 | 33 | 0.0 |
| For:for→body | 30.4 | 32.7 | 7 | -2.3 |
| Try:handler→orelse | 13.2 | 15.8 | 16 | -2.6 |
| IfExp:test→orelse | 23.2 | 26.7 | 2 | -3.5 |
| children:parent→child | 26.3 | 30.7 | 2 | -4.4 |
| Attribute:value→attr | 76.7 | 82.3 | 2 | -5.6 |
| For:target→body | 26.4 | 32.7 | 6 | -6.3 |
| If:test→body | 16.3 | 23.2 | 6 | -6.9 |
| Subscript:value→slice | 59.4 | 66.4 | 2 | -7.1 |
| BinOp:left→right | 31.5 | 45.2 | 2 | -13.7 |
| Try:handler→finalbody | 8.3 | 25.0 | 21 | -16.7 |
| For:target→iter | 58.3 | 77.4 | 2 | -19.1 |
| AugAssign:target→value | 49.4 | 70.3 | 2 | -21.0 |
| For:iter→body | 11.7 | 34.8 | 4 | -23.1 |
| IfExp:body→test | 17.2 | 42.1 | 2 | -24.9 |
| While:while→body | 22.1 | 48.5 | 5 | -26.5 |
| Assign:target→value | 39.8 | 71.2 | 2 | -31.4 |
| While:test→body | 16.2 | 48.5 | 4 | -32.4 |
| Call:func→args | 59.3 | 93.2 | 2 | -33.9 |
| Call:args→keywords | 20.6 | 54.9 | 2 | -34.4 |
| For:for→iter | 34.1 | 77.4 | 3 | -43.3 |

Table 9: Attention vs. offset baseline with fixed offset for each relation on **Python** dataset using **first**-token metric. In the score column, we present the accuracy score for CuBERT and offset baseline. In the offset column, the chosen offset is shown. Score differences are calculated as CuBERT score - offset baseline score for each relation, where a positive value indicates that the language model surpasses the baseline performance. Since CuBERT always outperforms Codebert, we only include results for CuBERT.

| Relation | Score | | Offset | Difference |
|---|---|---|---|---|
| | CuBERT | Offset | | |
| If:if→else | 92.7 | 5.7 | 17 | 87.1 |
| IfExp:body→orelse | 52.4 | 21.3 | 10 | 31.1 |
| For:target→iter | 98.8 | 77.4 | 2 | 21.4 |
| For:target→body | 97.2 | 81.4 | 14 | 15.8 |
| If:if→body | 77.0 | 61.8 | 11 | 15.2 |
| If:body→orelse | 60.7 | 48.7 | 18 | 12.0 |
| For:for→body | 93.2 | 81.3 | 15 | 11.8 |
| Compare:left→comparator | 56.4 | 45.7 | 2 | 10.7 |
| Try:body→orelse | 57.9 | 52.6 | 62 | 5.3 |
| While:while→body | 95.6 | 92.6 | 14 | 2.9 |
| children:parent→child | 45.3 | 42.6 | 2 | 2.7 |
| If:if→test | 100.0 | 98.8 | 1 | 1.1 |
| BinOp:left→right | 46.0 | 45.2 | 2 | 0.8 |
| For:for→target | 99.4 | 99.4 | 1 | 0.0 |
| While:while→test | 99.3 | 99.3 | 1 | 0.0 |
| Try:body→finalbody | 25.6 | 25.6 | 56 | 0.0 |
| Assign:target→value | 78.2 | 79.7 | 4 | -1.5 |
| Call:func→keywords | 62.4 | 64.7 | 4 | -2.3 |
| If:test→body | 59.2 | 61.7 | 10 | -2.5 |
| BoolOp:value→value | 47.8 | 50.6 | 8 | -2.8 |
| AugAssign:target→value | 66.8 | 70.3 | 2 | -3.6 |
| Subscript:value→slice | 61.7 | 66.4 | 2 | -4.8 |
| For:for→iter | 72.2 | 77.4 | 3 | -5.2 |
| Attribute:value→attr | 76.7 | 82.3 | 2 | -5.6 |
| IfExp:body→test | 34.8 | 42.1 | 2 | -7.3 |
| While:test→body | 85.3 | 92.6 | 13 | -7.4 |
| If:test→orelse | 36.5 | 44.9 | 31 | -8.4 |
| Try:body→handler | 41.6 | 51.2 | 17 | -9.7 |
| IfExp:test→orelse | 30.0 | 40.7 | 4 | -10.7 |
| Try:handler→orelse | 31.6 | 47.4 | 25 | -15.8 |
| Call:func→args | 75.3 | 93.2 | 2 | -17.8 |
| For:iter→body | 63.6 | 83.5 | 12 | -19.8 |
| Call:args→keywords | 49.2 | 74.4 | 4 | -25.2 |
| Try:handler→finalbody | 16.7 | 58.3 | 21 | -41.7 |

Table 10: Attention vs. offset baseline with fixed offset for each relation on **Python** dataset using **any**-token metric.

| Relation | Score | | Offset | Difference |
|---|---|---|---|---|
| | CuBERT | Offset | | |
| If:if→else | 87.0 | 7.0 | 15 | 80.0 |
| Switch:switch→statement | 100.0 | 75.2 | 5 | 24.8 |
| If:if→body | 48.8 | 26.2 | 7 | 22.5 |
| For:test→updaters | 76.0 | 53.4 | 4 | 22.5 |
| If:body→orelse | 28.7 | 9.4 | 10 | 19.3 |
| Try:body→handler | 24.4 | 5.5 | 8 | 18.9 |
| Do:body→test | 13.3 | 6.7 | 74 | 6.7 |
| Try:body→finalbody | 7.0 | 4.3 | 9 | 2.6 |
| Do:do→test | 6.7 | 6.7 | 34 | 0.0 |
| IfExp:test→orelse | 24.3 | 24.5 | 7 | -0.2 |
| InstanceofExpr:expr→type | 89.8 | 91.9 | 2 | -2.1 |
| For:for→initializers | 97.5 | 100.0 | 2 | -2.5 |
| Attribute:value→attr | 81.2 | 83.9 | 2 | -2.7 |
| children:parent→child | 34.2 | 36.8 | 2 | -2.7 |
| If:test→orelse | 2.6 | 6.8 | 15 | -4.2 |
| For:initializers→updaters | 39.8 | 45.7 | 9 | -6.0 |
| Subscript:value→slice | 72.3 | 78.8 | 2 | -6.5 |
| IfExp:body→orelse | 44.2 | 52.5 | 2 | -8.2 |
| For:for→updaters | 36.2 | 45.5 | 11 | -9.3 |
| Try:handler→finalbody | 19.5 | 29.9 | 16 | -10.4 |
| InfixExpr:left→right | 50.7 | 61.6 | 2 | -10.9 |
| Switch:switch→expr | 89.0 | 100.0 | 2 | -11.0 |
| If:test→body | 11.3 | 26.2 | 5 | -14.9 |
| IfExp:test→body | 21.2 | 37.5 | 5 | -16.3 |
| For:initializers→body | 21.0 | 37.5 | 13 | -16.6 |
| Switch:expr→statement | 58.1 | 75.2 | 3 | -17.1 |
| For:for→body | 16.1 | 36.0 | 15 | -19.9 |
| While:while→body | 13.2 | 42.7 | 9 | -29.6 |
| While:test→body | 9.4 | 42.7 | 7 | -33.3 |
| Assign:target→value | 35.3 | 68.7 | 2 | -33.4 |
| Call:func→args | 63.6 | 98.7 | 2 | -35.1 |
| For:test→body | 13.9 | 49.2 | 8 | -35.3 |
| If:if→test | 58.2 | 96.5 | 2 | -38.3 |
| While:while→test | 41.9 | 82.0 | 2 | -40.1 |
| For:updaters→body | 19.4 | 88.9 | 4 | -69.5 |
| For:initializers→test | 15.1 | 85.4 | 5 | -70.3 |
| Do:do→body | 26.7 | 100.0 | 2 | -73.3 |
| For:for→test | 10.5 | 84.8 | 7 | -74.3 |

Table 11: Attention vs. offset baseline with fixed offset for each relation on **Java** dataset using **first**-token metric.

| Relation | Score | | Offset | Difference |
|---|---|---|---|---|
| | CuBERT | Offset | | |
| If:if→else | 87.0 | 7.0 | 15 | 80.0 |
| For:test→updaters | 86.9 | 53.6 | 5 | 33.3 |
| If:if→body | 79.6 | 62.1 | 11 | 17.5 |
| If:body→orelse | 69.1 | 52.1 | 16 | 17.0 |
| While:while→test | 98.8 | 82.0 | 2 | 16.9 |
| Assign:target→value | 81.4 | 68.7 | 2 | 12.6 |
| Try:body→finalbody | 20.0 | 13.0 | 37 | 7.0 |
| IfExp:body→orelse | 59.5 | 52.5 | 2 | 7.0 |
| For:for→updaters | 53.6 | 46.8 | 11 | 6.7 |
| Do:body→test | 33.3 | 26.7 | 81 | 6.7 |
| Do:do→test | 33.3 | 26.7 | 81 | 6.7 |
| For:for→test | 95.3 | 89.5 | 9 | 5.8 |
| IfExp:test→orelse | 45.9 | 42.4 | 10 | 3.5 |
| While:while→body | 91.4 | 89.0 | 17 | 2.4 |
| If:if→test | 98.7 | 96.5 | 2 | 2.3 |
| children:parent→child | 44.0 | 42.4 | 2 | 1.6 |
| InstanceofExpr:expr→type | 93.1 | 91.9 | 2 | 1.1 |
| For:initializers→updaters | 47.2 | 47.1 | 9 | 0.1 |
| Switch:switch→statement | 100.0 | 100.0 | 21 | 0.0 |
| For:for→initializers | 99.4 | 100.0 | 2 | -0.6 |
| Try:body→handler | 44.0 | 44.7 | 25 | -0.7 |
| Subscript:value→slice | 77.9 | 78.8 | 2 | -0.9 |
| Switch:switch→expr | 99.0 | 100.0 | 2 | -1.0 |
| While:test→body | 87.4 | 89.5 | 15 | -2.2 |
| Attribute:value→attr | 81.2 | 83.9 | 2 | -2.7 |
| For:updaters→body | 96.8 | 99.6 | 9 | -2.8 |
| Switch:expr→statement | 96.6 | 100.0 | 12 | -3.4 |
| InfixExpr:left→right | 58.2 | 61.6 | 2 | -3.4 |
| For:test→body | 92.3 | 96.1 | 14 | -3.9 |
| If:test→body | 57.4 | 62.1 | 9 | -4.7 |
| For:for→body | 90.3 | 94.9 | 23 | -4.7 |
| If:test→orelse | 42.3 | 47.7 | 27 | -5.4 |
| Do:do→body | 93.3 | 100.0 | 2 | -6.7 |
| For:initializers→body | 87.7 | 94.9 | 21 | -7.2 |
| Call:func→args | 91.1 | 98.7 | 2 | -7.6 |
| For:initializers→test | 74.5 | 90.2 | 7 | -15.6 |
| IfExp:test→body | 25.9 | 50.0 | 5 | -24.1 |
| Try:handler→finalbody | 26.0 | 53.2 | 20 | -27.3 |

Table 12: Attention vs. offset baseline with fixed offset for each relation on **Java** dataset using **any**-token metric.