A Study on the Impact of Common Code Structures on CodeParrot's Autocompletion Performance

Razvan-Mihai Popescu

Supervisors: Arie van Deursen, Maliheh Izadi, Jonathan Katzy

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Razvan-Mihai Popescu
Final project course: CSE3000 Research Project
Thesis committee: Arie van Deursen, Maliheh Izadi, Jonathan Katzy, Azqa Nadeem

Abstract

In recent years, deep learning techniques, particularly transformer models, have demonstrated remarkable advancements in the accuracy and efficiency of language models. These models provide the foundation for many natural language processing tasks, including code completion. The effectiveness of code completion models has been the subject of a variety of empirical studies. However, none of the existing literature has explicitly investigated the potential impact of common code structures on the performance of large language models during code completion. This paper evaluates the influence of common code structures on the code completion performance of CodeParrot, a state-of-the-art natural language processing model. Using the tuned lens method, we show that typical code structures lead to a higher completion accuracy compared to uncommon code structures, due to their frequent occurrence, consistent syntax, clear semantics, and contextual clues. Finally, we perform an attention investigation to assess the significance of the common code structures and reveal potential data patterns across low- and high-resource languages.

Index Terms – code completion, attention

1 Introduction

Large language models (LLMs), particularly those based on deep learning techniques such as transformer models, have shown significant promise in improving the accuracy and effectiveness of code completion, alongside other natural language processing (NLP) applications [1]. These models can produce more relevant and contextually appropriate suggestions than conventional rule-based techniques since they are trained on enormous volumes of code data. Moreover, they are useful for a range of tasks in addition to their capacity for generative analysis, as demonstrated by previous NLP research. Word embeddings that have already been trained can be used in downstream tasks [2], while recurrent neural networks (RNN) performance could be enhanced by LLM pre-training [3].

Many state-of-the-art language models have been proposed to improve the field of code completion, including GPT-2 [4], CodeParrot [5], CodeBERT [6], InCoder [7], RoBERTa [8], and others. These foundation models can be adapted to solve many tasks, and they move away from the paradigm of narrow experts within deep learning. However, most models are only trained in one language or multiple very common languages at the same time. Thus, many of the plugins being developed, such as GitHub Copilot [9], will function best with popular languages such as Python or Java, but may struggle to operate on low-resource languages such as Kotlin, Julia, or Go. State-of-the-art models, such as CodeBERT or GPT-2, have been trained on a large corpus of data that also includes low-resource languages. Nevertheless, their performance in these languages is generally lower compared to high-resource languages due to limited training data and the absence of specific tuning for such cases [10]. Additionally, one important aspect that has not been studied in isolation before is whether common code structures (CCS) have an impact on the performance of LLMs during code completion. The performance of code completion models depends on various factors, including the model architecture, the quality of the training data, and fine-tuning methods. However, common code structures, such as conditional statements, loops, functions, and sequential structures, can also represent a significant factor.

This study aims to fill the existing knowledge gap by analyzing the variation in the code completion performance of CodeParrot across different programming languages. That is, we describe the relationship between common code structures and the depth of the first correct completion (DoFCC) of our model among low- and high-resource languages. To achieve this, we are using a multi-language test bench, comprising six programming languages, namely Python, Java, Kotlin, C++, Julia, and Go. We employ the CodeParrot model together with the tuned lens method [11] for token completion. This allows us to visualize the inner mechanism of our model at each computation stage and identify potential patterns in the data that highlight the influence of common structures during code completion. Furthermore, we conduct an attention investigation to evaluate the implications of the CCS on the attention mechanism of our model, concerning their impact on the outcome of code completion. This analysis provides insights into the performance of CodeParrot, and supports us in answering the **research question** of this study: *How do common code structures relate to the depth of the first correct completion?*

Our findings demonstrate that CodeParrot achieves higher accuracy in code completion for CCS when compared to less common structures. The influence of these structures is evident across all programming languages. However, it becomes significantly more pronounced in high-resource languages due to a higher amount of training data. Finally, both the common code structures and the uncommon code structures (UCS) have a similar impact on the attention mechanism. The ratio of null attention heads (NAHs) to other head types remains consistent across languages, with slight variations based on resource level, language complexity, and size of training data. To summarize, this paper's contributions can be stated as follows:

- We show that frequent code structures yield higher completion accuracy when compared to standard structures, across high- and low-resource languages;
- We illustrate that both common and uncommon structures demonstrate comparable and consistent implications on CodeParrot's attention mechanism;
- We reveal that the completion accuracy of CCS strongly correlates with the attention distribution within the model's initial layers;
- We map the performance variations between languages and provide a foundation of knowledge for the development of future models;
- We make our source code, datasets, and fine-tuned models public[1].

[1]The GitHub Repository

2 Background

The process of code completion can be categorized into several types based on the context in which it is applied.

- Token completion suggests single code tokens, representing the smallest meaningful units in a programming language, such as variables, functions, keywords, or operators. In this study, we utilize token completion to assess the distribution and features of individual tokens.
- Statement completion offers code completions at the level of a single line, aiding in the completion of partially written statements by suggesting missing code structures like function arguments, variable assignments, or method invocations.
- Block completion handles larger code blocks, specifically methods or functions, and plays a vital role in generating boilerplate code snippets, such as method signatures.

These different code completion types offer varying levels of granularity and assistance, addressing different stages of code writing.

2.1 CodeParrot Language Model

Understanding code completion involves a thorough investigation of the CodeParrot model, revealing its capability for correctly predicting and producing relevant code suggestions. CodeParrot is a multi-language code generation model that was trained on the GitHub Code dataset[2], which consists of 115M code files in 32 programming languages, including all the languages stated in the introduction except Kotlin. CodeParrot is based on the GPT-2 model and uses a Byte-Pair Encoding (BPE) tokenizer with a context length of 1024 tokens. The multi-language version of this model used in this study has 110M parameters and employs a transformer architecture that is specifically designed as a decoder-only model. This deep neural network architecture is based solely on the mechanics of attention, completely disregarding recurrence and convolution. Transformers, as opposed to sequential models, are far more expressive and more suited to the parallel computing capabilities of current GPUs, enabling language model pre-training to be carried out with larger models and more data [12].

Unlike traditional transformer models that consist of both encoder and decoder components, CodeParrot exclusively focuses on the decoder aspect. This architecture, depicted in *Figure* 1, is specifically tailored for GPT models and sets them apart from other transformer-based architectures. It incorporates a stack of 12 decoder layers, augmented with a linear and softmax layer on top. Each layer in the stack consists of a masked multi-head self-attention layer, followed by a feed-forward neural network. Additionally, there are residual connections and normalization layers [12]. The self-attention layer is essential for incorporating contextual information from relevant tokens into the current token being processed [12]. Furthermore, the masking prevents the model from constructing a token's representation by looking forward in the sequence, as typically done in the encoder.

This serves as a key justification for why the decoder-only architecture is a suitable choice for tasks such as code completion. In the process of code completion, the anticipation of subsequent tokens should be avoided while predicting the current one. The masked self-attention layer allows for an auto-regressive architecture that can repeatedly predict the next token in a sequence. In other words, the output of the model at time t is utilized as the input at time $t + 1$ [12]. Moreover, through the use of multiple heads, the model can jointly attend to data from various representation subspaces at different positions [12]. The residual connection adds the input to the output of the sub-layer to make the propagation of the information through the layers more efficient and effective [13]. The normalization layer helps to balance the activations of the model across various input sequence positions and layers. The feed-forward neural network layer is meant to transform the input sequence's representations and identify complex, and non-linear patterns. After the input is transformed into a sequence of embedding vectors, together with their positional encoding, it flows through the decoder stack and reaches the final two layers. The linear layer takes the output of the top decoder layer and transforms it into a logit vector of the same size as the vocabulary. This vector captures the initial unnormalized scores assigned to each potential token. Finally, the softmax layer takes this vector and converts it into a probability distribution over the vocabulary, resulting in the final prediction.

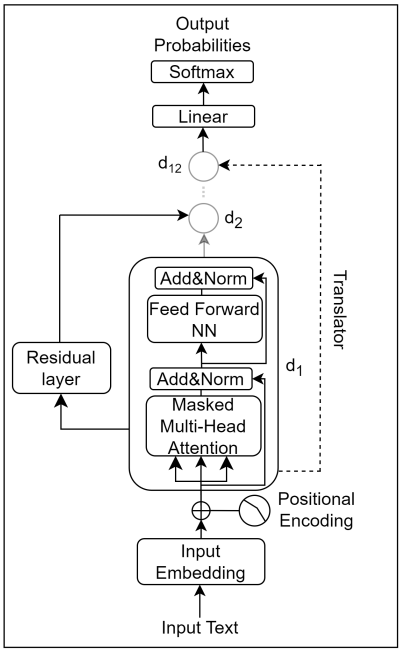


Figure 1: The tuned lens technique and decoder architecture

2.2 Tuned Lens Method

Despite a clear representation of the model's structural framework, a complete understanding of its mode of operation, internal mechanism, and computing capabilities is only partially attained. This inherent constraint results from the model's "black box" design, which makes it challenging to understand its internal workings and intricate details. Achieving transparency is of utmost importance while examining these common code structures. Such transparency is obtained through the application of the tuned lens methodology, thereby facilitating a more comprehensive analysis. This technique captures the iterative computations per-

[2]The GitHub Code Dataset

formed by a transformer to predict subsequent tokens, enabling us to analyze the model's latent predictions during a forward pass.

Moreover, the tuned lens approach possesses the property of stimulus-response alignment, indicating that features that exert the most influence on the output of the tuned lens also hold significant influence on the model itself [11]. On top of that, in a transformer with n layers, the tuned lens method enables the replacement of the final m layers with an affine translator. As it can be seen in *Figure* 1, the translator enables us to directly map the output from the first decoder layer (d_1) to the linear layer, bypassing the remaining decoder layers. This pattern holds for the entire stack and helps in obtaining the optimal prediction based on the model's intermediate representations. Each translator is trained to minimize the Kullback-Leibler (KL) divergence between its prediction and the final output distribution of the original model. The necessity of these translators arises due to the possibility of rotations, shifts, or stretches occurring in the representations across different layers [11]. Finally, the utilization of affine translators serves the purpose of establishing a consistent representation of the output at each layer, aligning it with the final layer output. This alignment not only facilitates the acquisition of coherent predictions but also ensures their relevance throughout the network.

2.3 Null Attention Heads

For the attention investigation, we are using the BertViz library[3] to visualize the attention mechanism of CodeParrot. To be more specific, our focus lies in examining the proportion of null attention heads to other attention heads, as this can expose how CCS are processed by our model. An attention head attends to different parts of the input sequence, enabling the model to focus on relevant information during processing. Conversely, a null attention head focuses all the attention on the first token in the sequence [14].

2.4 The Stack Dataset

A multi-language test bench was used for this research, containing source code files for both low- and high-resource languages. We extracted six subsets from The Stack dataset,[4] which includes over 6 TB of licensed source code files covering 358 programming languages. Each subset consists of 100,000 source code files per programming language. However, due to computational overhead, only 512 files were used for each language.

2.5 DelftBlue Supercomputer

To enable the tuned lens and attention investigations on such extensive datasets, the usage of DelftBlue[5] was imperative. We made use of one GPU, 4 CPUs per task, and up to 100 GB of memory to carry out this analysis. TU Delft's supercomputer was purposefully designed to accommodate the increasing computational demands and proved to be an ideal fit for this research. It provided us with the extensive computing power necessary to successfully conduct and conclude this study.

[3]The BertViz Library

[4]The Stack Dataset

[5]The DelftBlue Supercomputer

3 Related Work

The success of transformer models can be attributed to their ability to leverage large amounts of code data during training, enabling them to capture intricate programming patterns and enhance code generation. As a result, the application of deep learning techniques, especially transformer models, in code completion is an active and dynamic area of research within the broader domain of NLP, holding great practical promise. While a significant body of research exists in this field, none of the papers have specifically examined the influence of CCS on the attention mechanism and the code completion performance of LLMs. Instead, the focus has predominantly been on the general performance across diverse code structures and possible ways of improving the accuracy of code completion tools.

Using CodeBERT and GraphCodeBERT [15], Wan *et al.* [16] highlighted the importance of syntax structure in the pre-training phase of language models for source code. In this study, they explore probing word embeddings, conduct syntax tree induction, and perform an attention analysis to gain valuable insights into these models. Key findings include the strong alignment of attention with code syntax, the preservation of syntax structure in intermediate representations, and the model's ability to induce syntax trees. Throughout our analysis, we observed that the significance of the syntax structure can be derived from the superior completion performance achieved by CCS as opposed to UCS. This is due to their consistent syntax and high frequency across all six languages.

Existing probing methods aim to uncover the internal workings and representations of pre-trained LLMs by analyzing specific layers, neurons, or attention mechanisms. However, when it comes to code pre-trained models (CodePTMs), these methods fail to consider the unique characteristics of code. To address this, Chen *et al.*[17] introduced a novel quantitative probing technique called CAT-probing, which interprets how CodePTMs attend to code structure. It involves denoising input code sequences based on pre-defined token types and defining a metric called CAT-score to measure the commonality between attention scores and pair-wise distances of AST nodes. Furthermore, this study examines the common token types that CodePTMs prioritize across various programming languages such as Go, Java, Python, and JavaScript. The findings indicate distinct and frequent token types for each language. For instance, Java includes "*public*", "*s_literal*", and "*return*", as frequent token types, while Python consists of "*for*", "*if*", and "*)*". CodePTMs prioritize token types differently from programmers, showing a greater emphasis on code tokens such as brackets. Our investigation also revealed that spaces, brackets, and punctuation signs have the highest completion accuracy. This is due to their widespread utilization in constructing more complex components in most programming languages. A limitation of this study is the exclusive focus on encoder-only CodePTMs such as CodeBERT, GraphCodeBERT, or RoBERTa in the adopted probing approaches. On the other hand, we are using a decoder-based model and the first layer where the completion was correct, as an evaluation metric.

Recent advancements in LLMs for code completion and code synthesis, such as Codex [9], have shown significant potential, but the lack of publicly available models raises

questions about their design choices. To address this, Xu *et al.*[5] systematically evaluated existing models, including Codex, GPT-J,[6] GPT-Neo [18], GPT-NeoX-20B [19], and CodeParrot, across different programming languages. The study demonstrates that open-source models achieve similar results to Codex in some languages, although they are primarily designed for natural language modeling. Moreover, the need for a large open-source multilingual code model is emphasized [5]. Results from the HumanEval benchmark and token analysis during training are used to compare different models. Among the models studied, PolyCoder, trained on a mixture of GitHub repositories in 12 languages, performs worse than the similarly sized GPT-Neo and even the smaller Codex 300M. Lastly, CodeParrot shows weak performance in languages other than Python, attributed to its exclusive training on Python data in this study, which differs from the multi-language version used in our research.

To assess the significance of specific tokens during code completion, conducting an attention analysis becomes essential. Utilizing the GPT-2 small pre-trained model, Jesse Vig [14] explores attention-syntax relationships and reveals that different layer depths target distinct parts of speech, with middle layers emphasizing dependency relations. Moreover, deeper layers capture distant relationships, while exemplar sentences illustrate attention patterns specific to certain heads. Furthermore, the study reveals that the performance of the model is minimally affected by individual attention heads. Hence, the analysis excludes NAHs as they do not contribute any significant information. Additionally, the analysis reveals that, on average, 57% of attention is allocated to the initial token, and this pattern is particularly found in the upper layers of the network. This analysis was conducted using sentences from English Wikipedia, which the model was not trained on, whereas our attention investigation used source code files from six different languages.

Finally, using the BertViz visualization method and the GPT-2 model, Jesse Vig [20] conducts an attention investigation to identify possible recurring patterns. The findings show that many attention heads in the initial layers often exhibit position-based behavior, focusing on the same or previous tokens. Moreover, a consistent pattern is represented by null attention heads, which concentrate solely on the first token. These attention heads signify the absence of a linguistic property in the input text, providing valuable insights into the model's decision-making process. The attention heads used in this investigation were extracted from a single sentence. In contrast, our predictions involve a greater left context, resulting in a modified structure of the NAHs, as explained in *Subsection* 5.2.

4 Methodology

Based on their functionality, common structures can be categorized into multiple distinct groups, namely: control structures, functions, classes, objects, exceptions, data structures, and input and output (I/O) structures. The control structures comprise constructs used to control the flow of execution in a program, such as "*if*" statements or loops, which include "*for*" and "*while*" statements. Data structures are represented by various types of containers for organizing and storing data, such as arrays, stacks, queues, trees, and many others. Lastly, the I/O structures facilitate interactions between the program and the external environment, including user input and output, or reading and writing data to files. To gain a more profound understanding of the implications associated with these structures, our analysis is divided into two parts: the *tuned lens investigation* and the *attention investigation*.

4.1 Tuned Lens Approach

For the tuned lens investigation, we start by creating a collection of the most common code structures for each language. Following this, the CodeParrot model is used to perform token completion on the pre-processed data. At the same time, the tuned lens method is applied to acquire insights into the internal representations of our model. Various statistical techniques are utilized to evaluate the intermediate outcomes of the model. Ultimately, numerous visualization methods are deployed to extract relevant information and identify potential trends within the data.

4.2 Attention Approach

When it comes to the attention investigation, we initiate the process by collecting attention heads for common and uncommon structures, using distinct datasets. Therefore, by employing diverse statistical methodologies, our investigation revolves around the NAHs aspect across all aforementioned structures and languages. As a result, we aim to enhance our understanding of how the model directs its focus toward particular contextual information. Finally, we analyze the correlation between the results of the two investigations, aiming to comprehend the overall impact and significance of these CCS and to understand the decision-making process of the model.

5 Experimental Setup

The quantity of training data constitutes an important factor influencing the overall performance of CodeParrot in code completion tasks. As illustrated in *Table* 1, high-resource languages possess a greater volume of training data in contrast to low-resource languages. In addition, it is noteworthy that Kotlin, unlike other languages, was excluded from the model training process.

Table 1: Training data for the CodeParrot model

Language	Data Size (GB)
Java	107.70
C++	87.73
Python	52.03
Go	19.28
Julia	0.29
Kotlin	0

5.1 Tuned Lens Investigation

A distinct collection of the most prevalent tokens was created for each language utilized in this research, to strengthen the reliability of our findings. The process involved carefully examining code samples and leveraging domain knowledge and experience to identify the tokens that appeared most frequently. To avoid potential result distortions, tokens that represent standalone brackets or whitespaces were excluded

[6]The GPT-J Model

from the analysis. Given their substantial presence across all languages, these tokens demonstrate the highest accuracy among all tokens, which can lead to biased results. The soundness of these collections is supported by a comparative analysis and the experimental results presented in *Section* 6.

To reduce the computational overhead, the data is initially filtered by removing the comments from each file. The comments are not pertinent to this study as they do not provide insights into CCS. This is because they typically provide additional information about the code logic, rather than representing the code structures themselves. Afterward, the files with less than 500 tokens are removed, and for large files, only a random slice of approximately 1750 tokens is retained. This filtering enhances computational efficiency and ensures sufficient context for accurate predictions. Hence, we start with a minimum left-context of 512 tokens and progressively increase it up to a maximum of 1024, the upper limit supported by CodeParrot. Following that, data undergoes tokenization using the GPT-2 BPE tokenizer, which effectively captures a wide range of linguistic variations. Furthermore, batching is applied to process multiple inputs simultaneously, reducing the prediction overhead and optimizing memory usage.

To maximize prediction accuracy, we apply the tuned lens method to exclusively concentrate on the final token within each layer. This is done to disregard any right-context information that may lead to biased completion outputs. This process enables us to extract the intermediate results generated by CodeParrot at each specific layer of our prediction. To ensure the consistency of the stored results, we create a data structure that enhances accessibility for further computations. For every file, we store the input code, the tokenized input, the intermediate layer predictions, the generated attention heads, and multiple metadata elements including the file ID and the number of tokens.

As an evaluation method for our predictions, we are using the *depth of the first correct completion* (DoFCC). This refers to the layer at which the intermediate result aligns with the expected token. We contrast the mean and standard deviation of the DoFCC between common and uncommon code structures across all six languages. Lastly, we perform a comparative analysis of printing statements, a widely used code structure, among all programming languages. To accomplish this, we isolate the specific context in which these structures occur and retain only the print statements that contain a maximum of 16 tokens within their scope. Therefore, we can illustrate the impact of CCS from both a global and local perspective.

5.2 Attention Investigation

The identical procedure of filtering, tokenizing, and batching is applied for the attention investigation. However, in this case, only the attention heads are stored rather than the predictions. By utilizing the pre-established prevalent collections, we create separate attention datasets for both common and uncommon structures for each language. To mitigate any potential biases in our data and ensure diversity, we limit the token prediction to one per file and store only a randomly selected attention head per layer. Hence, our analysis involves a total of 18,000 attention heads, specifically 3,000 for each language, equally distributed among the two token types.

As mentioned earlier, to enhance the precision of our outcomes, we adopted a left-context of 1024 tokens. Consequently, the structure of our attention heads is denoted by a matrix of dimensions 1024×1024, wherein each token carries a cumulative attention weight of 1. The task of identifying and visualizing null heads, where the first token receives the entirety of the attention weight (specifically, 1024), becomes increasingly challenging in a broader context. This difficulty arises due to a considerable number of tokens spreading lower attention weights among several other tokens, particularly in the initial layers, as shown in *Subsection* 6.2. Therefore, we define the null attention head as the head wherein a minimum of half of the attention (specifically 512), is directed toward the first token.

Subsequently, we conduct a comparative analysis of the proportion of null attention heads to other types of heads across common and standard structures for each language. Furthermore, we examine the frequency of null attention heads per layer. In addition, we contrast the standard deviation and average attention score of each null head per layer for both types of structures. This score represents the total amount of attention weight that the first token receives. Finally, we explore the potential relationship between the outcomes derived from both investigations, aiming to identify any possible correlations. As a result, we can illustrate how common code structures impact the completion process and the attention mechanism of CodeParrot.

6 Experiment Results

CodeParrot was trained on large amounts of code, which includes numerous instances of these common code structures. Therefore, the model has more exposure to the patterns and contexts associated with these structures, enabling it to learn their usage accurately.

6.1 Tuned Lens Results

The influence of the CCS on completion performance is apparent across all six languages, particularly in high-resource languages such as Java, Python, and C++. This effect is more pronounced in these languages due to the availability of a larger volume of training data. The analysis presented in *Figure* 2 demonstrates a notable disparity in completion accuracy between CCS and UCS in Java. Specifically, the common structures lead to nearly twice the level of accuracy compared to the uncommon structures. The observed DoFCC for common structures is 3.53, in contrast to 6.27 for uncommon structures, while the standard deviations for both cases are relatively similar. Comparable findings were observed for Python, with a slightly increased average DoFCC for common structures. Furthermore, the average DoFCC for common code structures in C++ is just one layer lower compared to the DoFCC for standard structures, which still denotes a substantial performance improvement. A more detailed view of these results can be found in the *Appendix* A.1.

In the context of low-resource languages, the significance of typical code structures is less pronounced due to the limited availability of training data. However, an exceptional case arises with Go, as it demonstrates remarkable performance comparable to that of Java, as it can be seen in *Figure* 3. Due to its focus on simplicity and ease of use, Go features a more minimalistic syntax in comparison to Java, thereby compensating for the scarcity of training data. For example, the implicit absence of *"while"* loops necessitates the utilization of their desugared versions by using

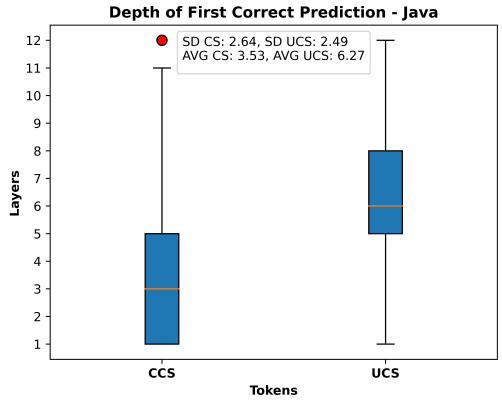


Figure 2: The mean and standard deviation of the DoFCC for common and uncommon code structures in Java

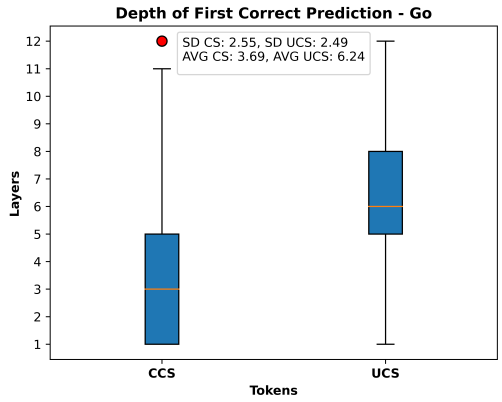


Figure 3: The mean and standard deviation of the DoFCC for common and uncommon code structures in Go

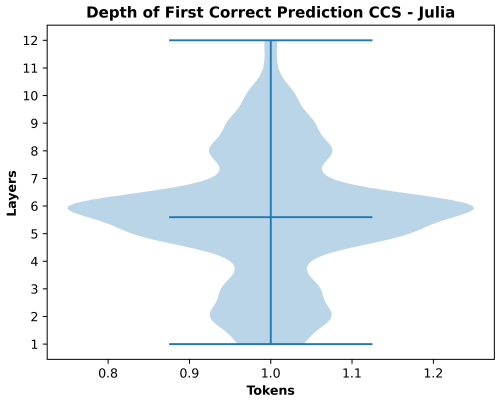


Figure 4: The mean and standard deviation of the DoFCC for common code structures in Julia

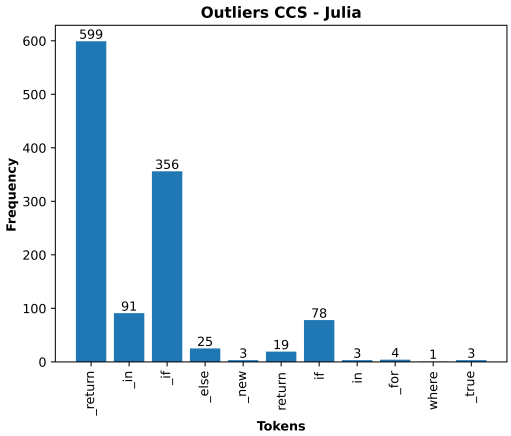


Figure 5: Low outliers of common code structures in Julia

"if" or *"for"* statements. Therefore, this leads to a smaller collection of commonly used structures in Go when compared to other high-resource languages, which explains this level of accuracy. Conversely, the influence of typical structures is most minimal in the context of Julia, as illustrated in *Figure* 4.

The predictions exhibit a concentration within layers 4 to 7, with a mean value of approximately 6, further highlighting the vital role of training data in shaping performance. Surprisingly, Kotlin displays consistency in both the mean and standard deviation across all structures, leading to similar outcomes to Julia, despite our model not being trained on it. These results can be attributed to the fact that Kotlin's syntax incorporates a substantial number of Java constructs, thereby emphasizing the importance of syntax in the context of code completion. As a result of the aforementioned reason, our analysis outcomes for Kotlin do not demonstrate any notable differences compared to the other languages for which training data was accessible.

An interesting pattern is depicted in *Figure* 5, where common tokens such as *"_return"* or *"_if"* lead to outstanding completion performance in Julia, with a DoFCC value of just one. This reinforces the fact that maintaining a proper indentation and adhering to whitespace conventions, leads to a higher completion performance. However, this is not the case for Python, since it enforces correct indentation for code execution. This trend was consistently observed in all programming languages, with a more pronounced impact on languages that adopt curly brackets to denote code blocks, where the inclusion of whitespace within those brackets is optional. Further elaboration on these results can be found in the *Appendix* A.2, offering a more extensive perspective on the findings.

A notable aspect presented in *Figure* 6 is the consistent average DoFCC observed across all tokens within the scope of a print statement in Python. Our findings reveal that the majority of the predictions occur in layer 5, with a higher occurrence of outliers observed for later tokens due to their lower frequency. This pattern of consistency is further supported by the standard deviation, predominantly falling within the range of 2 to 3. This demonstrates that CodeParrot, given their frequent occurrence and consistent syntax, is capable of effectively learning these CCS. As a result, common structures achieve a higher completion accuracy compared to standard structures, which, on average, exhibit a DoFCC of 6 in Python. Conversely, this consistency is not observed in the case of print statements in C++, primarily due to their more intricate structure. As depicted in *Figure* 7, the average DoFCC for print statements in C++ ranges between 6 and 7, with a few minor exceptions. The influence of common

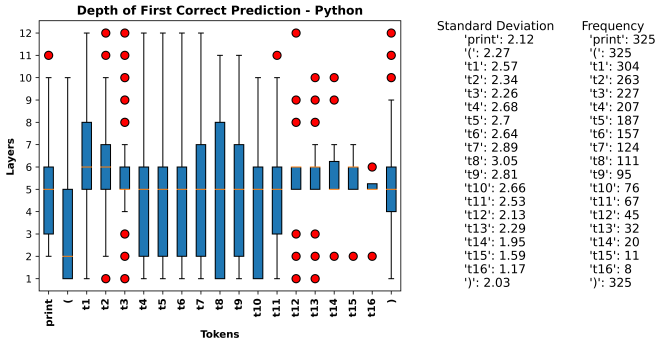


Figure 6: The mean and standard deviation of the DoFCC for print statements in Python

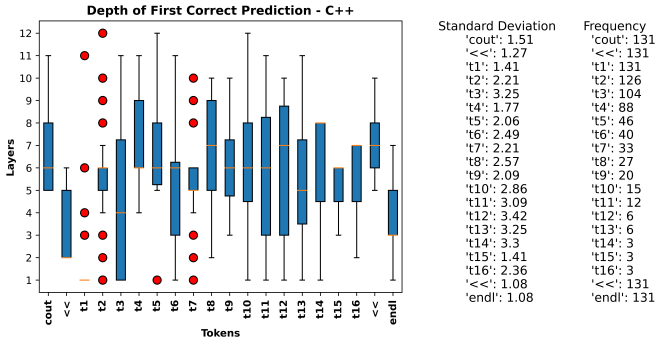


Figure 7: The mean and standard deviation of the DoFCC for print statements in C++, using the *std* namespace import

structures on completion accuracy is not as straightforward as in Python, since the uncommon structures in C++ also achieve an average DoFCC of approximately 6. However, this outcome can be attributed to the lower frequency of tokens and the syntactic complexity associated with printing operations.

The decision-making process of the model can be influenced by the various forms in which a print statement can be expressed in C++, either by using the *"std"* and *"::"* tokens or not. Moreover, the presence of nested print statements, involving multiple occurrences of the "<<" token, can result in fluctuations in the average DoFCC values. This arises due to the potential confusion faced by CodeParrot, as it can interpret the usage of "<<" as an application of the bitwise operator, which uses the same token in C++. Additionally, the oscillations in DoFCC are also associated with the pattern of consecutive common structures. This pattern emerges when clusters of common tokens, forming specific structures, appear successively. Once the model encounters the initial token, subsequent common tokens tend to receive significantly improved predictions due to the learned patterns during training. This pattern becomes apparent at the start of the print statement in Python, where the opening bracket exhibits higher accuracy in completion compared to other tokens. In the context of C++, this trend is observable both at the beginning and the end of the print statement, and potentially within the body, particularly in nested cases. Moreover, if there are any other common structures present within the print statements, they can also introduce fluctuations in the values of DoFCC. Finally, the pattern of consecutive common structures can also be observed in the print statements of the other languages, as stated in the *Appendix* A.3.

6.2 Attention Results

The implications of the common structures on the attention mechanism of CodeParrot appear to be less evident when compared to their impact on the completion performance. Our results show that the ratio of NAHs to other head types remains consistent across all languages for both CCS and UCS. This ratio varies between 0.04 and 0.14 indicating a relatively small occurrence of null attention heads in each language. As discussed in *Subsection* 5.2, this can be attributed to the use of a high context value, making it challenging to identify null attention heads where the first token receives all the attention. Therefore, we imposed a new attention threshold, requiring at least half of the attention to be allocated to the first token. In this scenario, a noticeable pattern can be observed among all structures and languages, particularly in the lower layers. *Figure* 8 visually demonstrates a clear absence of null attention heads within the initial four layers. This implies that the attention is predominantly distributed across a wide range of tokens, or among specific tokens, with little to no weight assigned to the first token.

At the layer level, the findings also demonstrate consistent results. The mean attention score varies from 533 to 714, without reaching the maximum attention value of 1024. Additionally, the standard deviation of the attention score for each NAH remains stable across all languages, with a maximum value of 95. Nevertheless, regarding the frequency of null attention heads per layer, a distinct pattern emerges in upper layers, for both types of structures. Noticeable frequency peaks can also be found in layers 5,6,8, and 12, although the majority of null attention heads are concentrated in layers 10 and 11, the latter one being more predominant. Go achieves the highest number of NAHs for common code structures, with a value of 67 heads in layer 11. In terms of uncommon code structures, Python attains the highest count of 64 NAHs in layer 11, closely followed by Java with 63, and again Go with 62 null attention heads. Furthermore, Julia exhibits the lowest quantity of NAHs among both common and uncommon structures, while consistently maintaining a minimal count across all layers. This observation can be directly related to the limited availability of training data. Finally, Kotlin attains a higher amount of NAHs in comparison to Julia, and this can be attributed to its pronounced resemblance to Java, which possesses the highest volume of training data.

The two investigations unveil a possible correlation between the obtained results. These findings provide valuable insights into the connection between layers and code structures in the occurrence of NAHs. The results of the tuned lens analysis showed that the correct predictions for the CCS usually take place in the first half of the layers, across all languages. This happens predominantly in the first four layers but with some exceptions in layers 5 and 6. The behavior of CodeParrot potentially signifies a higher emphasis on the processing of predictions in the lower layers. Instead of allocating a significant attention weight to the initial token, the model distributes its attention across more contextually relevant tokens. As demonstrated earlier, the number of NAHs begins to rise in the latter half of the layers following the prediction. This implies that model's focus on the prediction

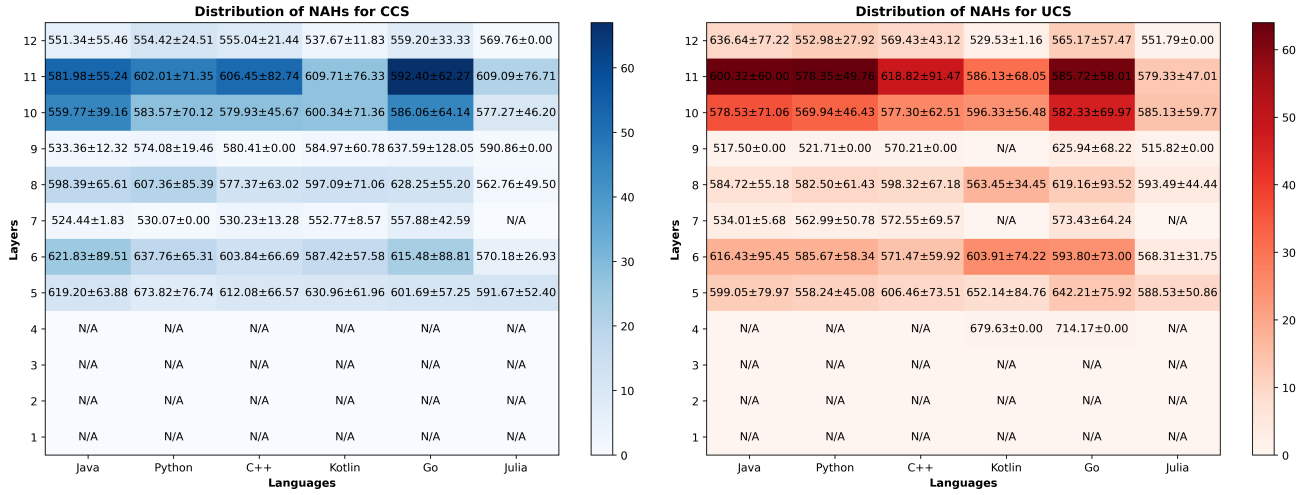


Figure 8: The frequency of null attention heads across each layer and within each language. The mean and standard deviation of the attention score for each null attention head are also displayed. The left heatmap corresponds to common code structures, while the right heatmap represents uncommon code structures.

diminishes in the upper layers. Additionally, the first token in the sequence receives most of the attention, whereas the other tokens are either neglected or assigned minimal attention weights.

According to Jesse Vig [20], the null attention heads fail to reveal the presence of the linguistic property within the input text, here represented by CCS. Moreover, it was observed that the focus on the initial token appears in the absence of relevant tokens elsewhere in the sequence [14]. Therefore, in our scenario, these remarks suggest that focusing a substantial amount of attention on the first token does not yield pertinent contextual information for the prediction. Hence, this observation potentially sheds light on the disparity between lower and upper layers, as well as the intricate relationship connecting NAHs and the impact of common code structures on the model's behavior.

7 Discussion

The findings of our study indicate that common structures exhibit a higher completion accuracy compared to uncommon structures across all programming languages considered in this research. Furthermore, we observed variations in performance level across both high and low-resource languages, primarily influenced by the availability of training data. The relationship between the amount of training data and the syntax structure plays a significant role in shaping the performance of CodeParrot. This observation was noted in the context of Kotlin, which demonstrated comparable results to the other languages, despite the limited availability of training data. This outcome can be attributed to Kotlin's robust syntax similarity to Java.

The attention analysis revealed similar outcomes for CCS and UCS. However, a closer examination of the frequency of NAHs at the layer level unveils intriguing patterns. In Jesse Vig's investigation [14], it was demonstrated that certain attention heads exhibited an average attention concentration of over 97% on the first token. This finding emphasizes that the upper layers of the model display the highest proportion of attention directed toward the initial token. Our findings provide further support for this observation, highlighting the consistent pattern where the majority of NAHs are predominantly concentrated in the upper layers of the model, rather than the lower layers. However, for our analysis, we used attention heads extracted from source code, as opposed to those used in Vig's study, which typically focused on Wikipedia sentences following a similar encyclopedic format and style [14]. Additionally, we saw that these attention trends exhibit a potential correlation with the completion accuracy of CCS. This provides valuable insights into the decision-making process employed by our model and the relationship between DoFCC and the distribution of attention throughout various stages of the prediction. Nevertheless, the primary limitation of our study lies in the scarcity of data available to support this observation. At the same time, this presents a promising avenue for future investigation. By isolating the common constructs according to specific contexts, a more comprehensive understanding of their impact on the completion performance of CodeParrot can be obtained. This would provide valuable information to reinforce our observations and enhance the significance of our findings.

8 Responsible Research

We prioritize reproducibility and transparency in all facets of our study. These core values guide our efforts to maintain the integrity, trustworthiness, and accessibility of our research findings. To achieve this, we publish all the necessary resources for conducting the experiment, such as the source code, fine-tuned model, and datasets. In addition, we place a strong emphasis on thorough code documentation, ensuring that it is well-structured and extensively commented. This leads to highly-readable code, enabling easy understanding and facilitating reproducibility.

Furthermore, we provide clear and concise explanations of our research methodology, experimental setup, data collection processes, and analysis techniques. Additionally, we incorporate a comprehensive description of the tools employed to carry out this research. Finally, to mitigate bias in the data processing algorithm, we employ a range of techniques detailed within this paper, such as the random selection of

NAHs per layer or the random selection of slices from large input files. This extensive documentation of the research process serves as a roadmap, enabling others to navigate and understand the trajectory of our research.

9 Conclusion

In this work, we brought to light the impact of common code structures on the attention mechanism and completion performance of CodeParrot.

We showed that common structures yield higher completion accuracy compared to less common structures, with the outcomes varying depending on the resource level, language complexity, and quantity of training data. This can be attributed to their frequent occurrence, consistent syntax, clear semantics, and contextual clues. These factors significantly enhance the model's ability to learn and assimilate the patterns associated with these structures more effectively.

We also discovered that the implications of the common structures on the attention mechanism are less pronounced compared to their influence on the completion performance. Both CCS and UCS achieved consistent and similar attention outcomes across all six languages studied. We established a strong correlation between the DoFCC of CCS and the scarcity of NAHs in the initial four layers of the network. Moreover, our observations indicated that the majority of NAHs originating from the upper layers do not contribute relevant contextual information to the predictions. Nevertheless, our findings present opportunities for further research and exploration in this domain.

References

[1] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[2] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In *"Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing"*, pages 670–680, 2017.

[3] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *"Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)"*, pages 328–339, 2018.

[4] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. 2019.

[5] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A systematic evaluation of large language models of code. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[7] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, et al. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

[8] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, et al. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692, 2019.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, et al. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, et al. Codebert: A pre-trained model for programming and natural languages. *ArXiv*, abs/2002.08155, 2020.

[11] Nora Belrose, Zach Furman, Logan Smith, Danny Halawi, Igor Ostrovsky, et al. Eliciting latent predictions from transformers with the tuned lens. *ArXiv*, abs/2303.08112, 2023.

[12] Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, et al. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008, 2017.

[13] He et al. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] Jesse Vig and Yonatan Belinkov. Analyzing the structure of attention in a transformer language model. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: Student Research Workshop*, pages 158–165, 2020.

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, et al. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[16] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hairong Jin. What do they capture? - a structural analysis of pre-trained language models for source code. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2377–2388, 2022.

[17] Nuo Chen, Qiushi Sun, Renyu Zhu, Xiang Li, Xuesong Lu, and Ming Gao. CAT-probing: A metric-based approach to interpret how pre-trained models for programming language attend code structure",. In *"Findings of the Association for Computational Linguistics: EMNLP 2022"*, pages 4000–4008, 2022.

[18] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Rose Biderman. GPT-Neo: Large-scale autoregressive language modeling with mesh-tensorflow. 2021.

[19] Sid Black, Stella Biderman, Eric Hallahan, Connor Leahy, Kyle McDonell, et al. GPT-NeoX-20B: An

open-source autoregressive language model. In *"Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models"*, pages 95–136, 2022.

[20] Jesse Vig. Visualizing attention in transformer-based language representation models. *ArXiv*, abs/1904.02679, 2019.

A Extended Tuned Lens Results

A.1 General Structure Results

This subsection showcases the remaining results of the tuned lens analysis, which were used to evaluate the completion accuracy of both common and uncommon structures. These results comprise the mean and standard deviation of the DoFCC across the languages used in this study. The violin plots are utilized to gain a comprehensive and detailed perspective of the data distribution.

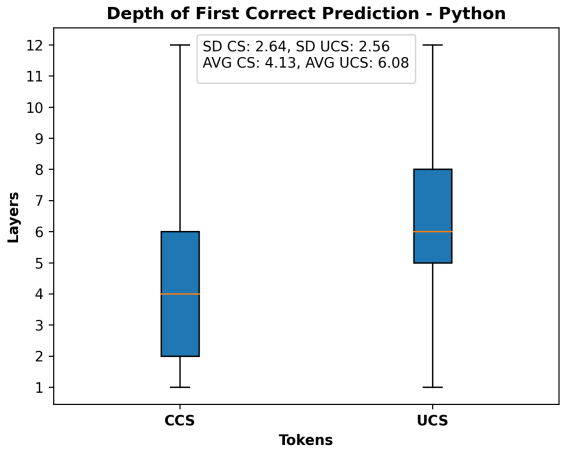


Figure 9: The mean and standard deviation of the DoFCC for common and uncommon code structures in Python

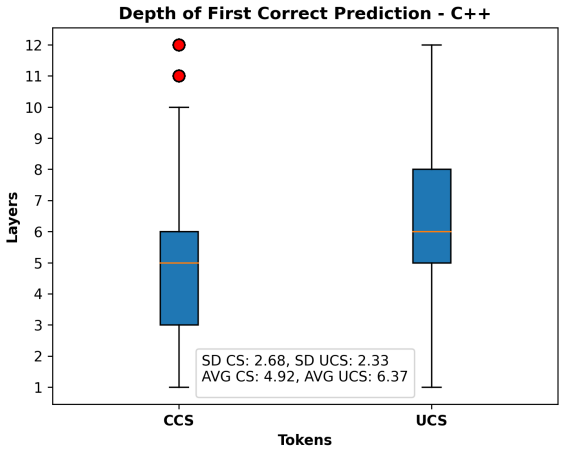


Figure 10: The mean and standard deviation of the DoFCC for common and uncommon code structures in C++

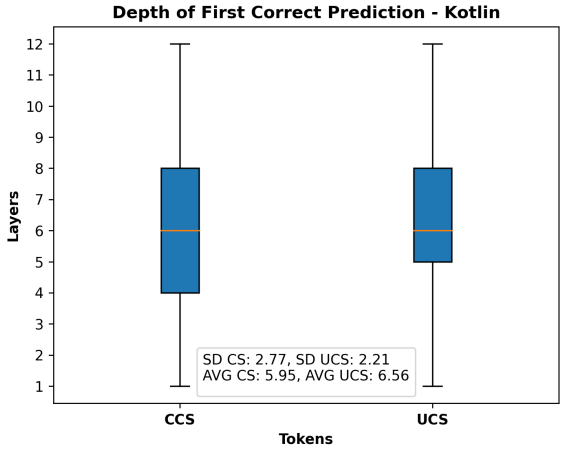


Figure 11: The mean and standard deviation of the DoFCC for common and uncommon code structures in Kotlin

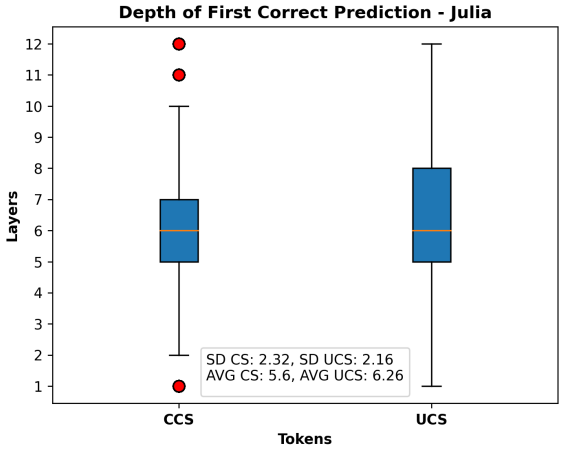


Figure 12: The mean and standard deviation of the DoFCC for common and uncommon code structures in Julia

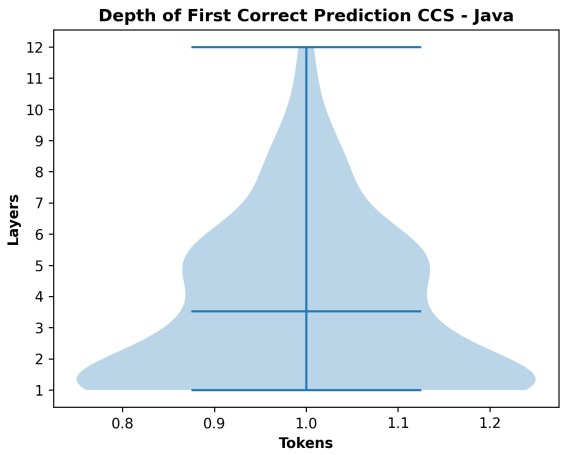


Figure 13: The mean and standard deviation of the DoFCC for common code structures in Java

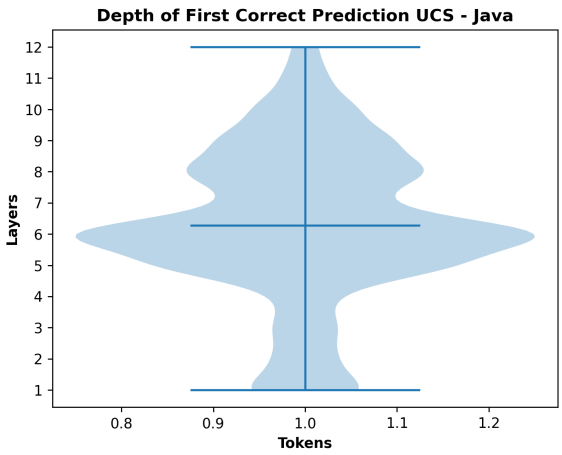


Figure 14: The mean and standard deviation of the DoFCC for uncommon code structures in Java

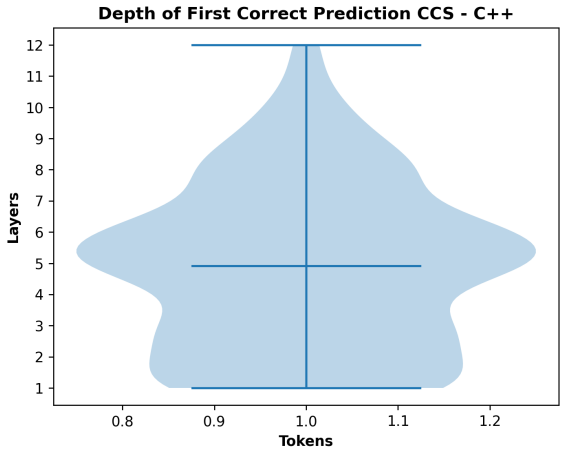


Figure 15: The mean and standard deviation of the DoFCC for common code structures in C++

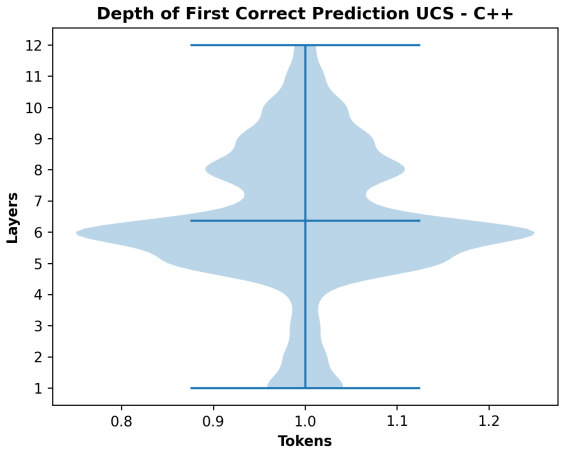


Figure 16: The mean and standard deviation of the DoFCC for uncommon code structures in C++

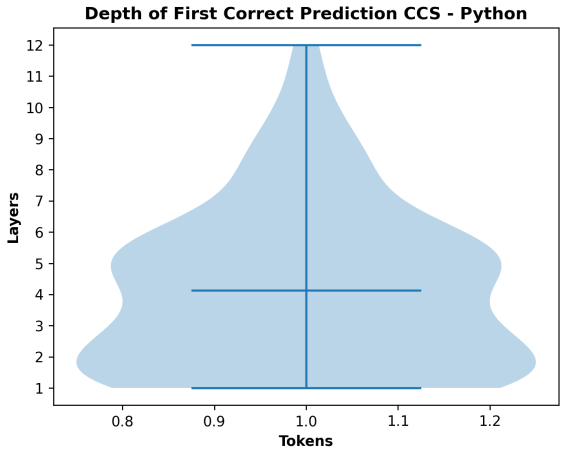


Figure 17: The mean and standard deviation of the DoFCC for common code structures in Python

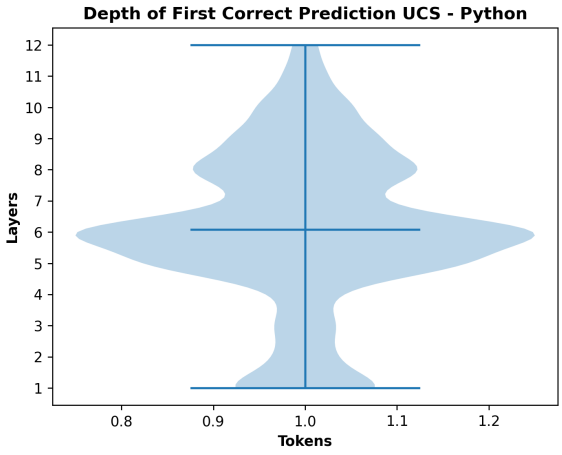


Figure 18: The mean and standard deviation of the DoFCC for uncommon code structures in Python

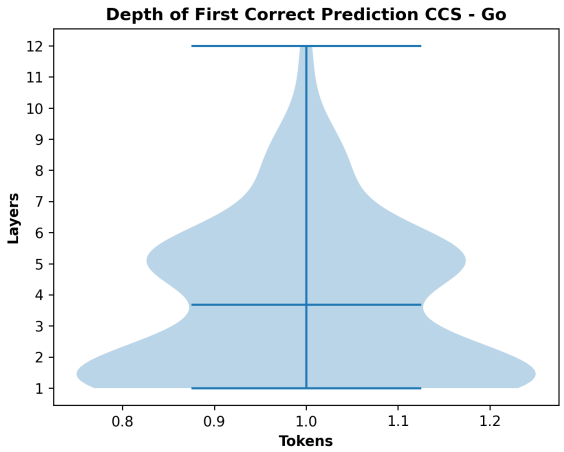


Figure 19: The mean and standard deviation of the DoFCC for common code structures in Go

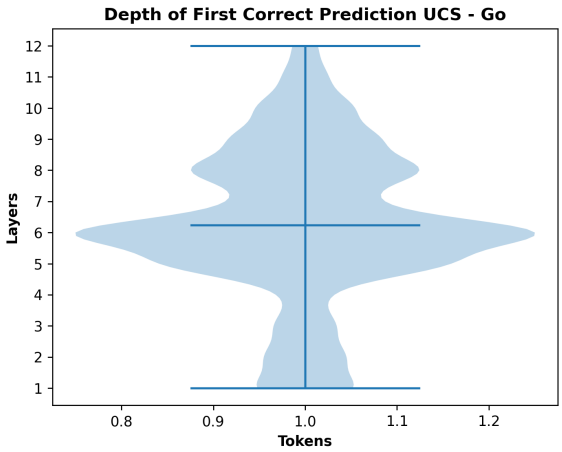


Figure 20: The mean and standard deviation of the DoFCC for uncommon code structures in Go

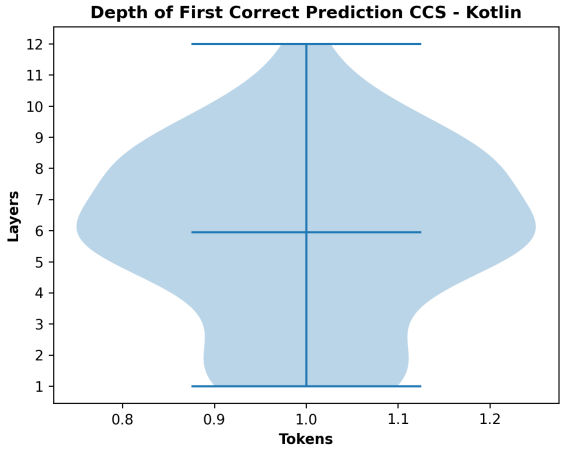


Figure 21: The mean and standard deviation of the DoFCC for common code structures in Kotlin

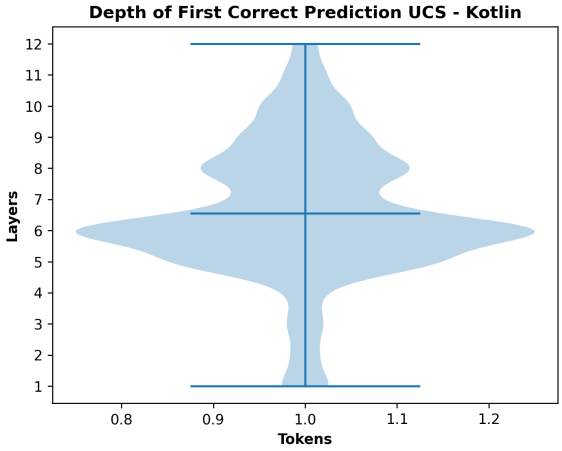


Figure 22: The mean and standard deviation of the DoFCC for uncommon code structures in Kotlin

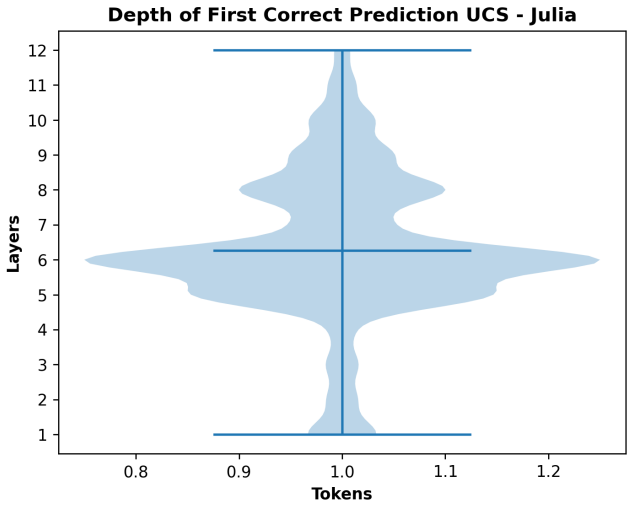


Figure 23: The mean and standard deviation of the DoFCC for uncommon code structures in Julia

A.2 Indentation Results

This subsection presents the results derived from the indentation analysis conducted for common code structures. These results are particularly applicable to languages such as Java, Kotlin, and Go, where the usage of curly brackets to define code blocks eliminates the need for proper indentation. For this investigation, we examined two highly prevalent constructs found in all of these languages, specifically *"if-else"* statements and *"for"* loops. We evaluate the completion accuracy of these structures by comparing their performance with and without proper indentation. Precisely, we examine the mean and standard deviation of the DoFCC per token. The findings of our study indicate that, in nearly all instances, the utilization of common tokens with proper indentation yielded higher accuracy in code completion. However, the excessive usage of white spaces should be avoided, as evident in the case of the *"int"* and *"i"* tokens in Java and Kotlin. These tokens frequently appear within the body of *"for"* loops, representing a common pattern across these languages. It is noteworthy that introducing unnecessary white spaces before these tokens results in a higher mean value of the DoFCC.

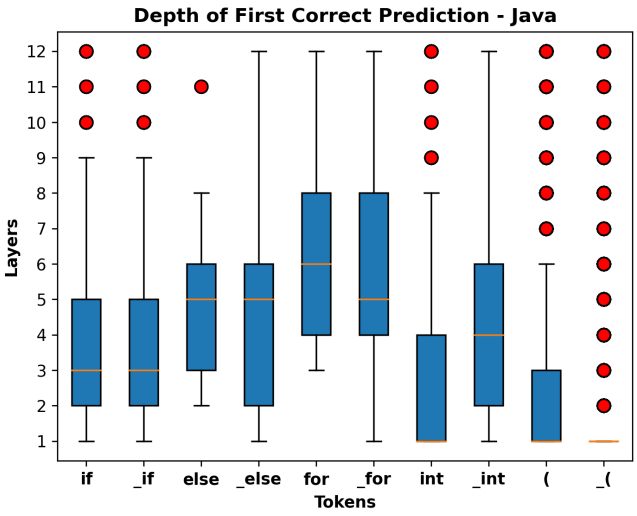


Figure 24: The mean and standard deviation of the DoFCC for common code structures with and without proper indentation in Java

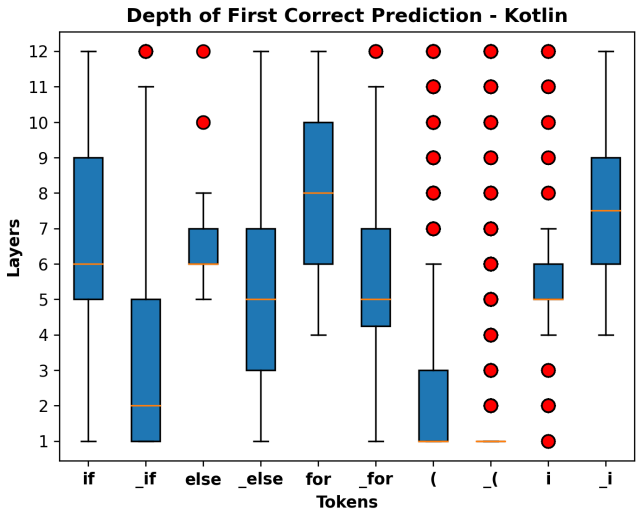


Figure 25: The mean and standard deviation of the DoFCC for common code structures with and without proper indentation in Kotlin

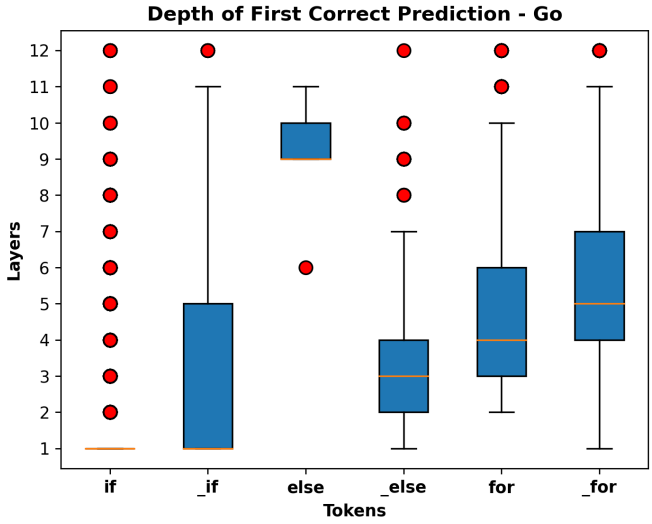
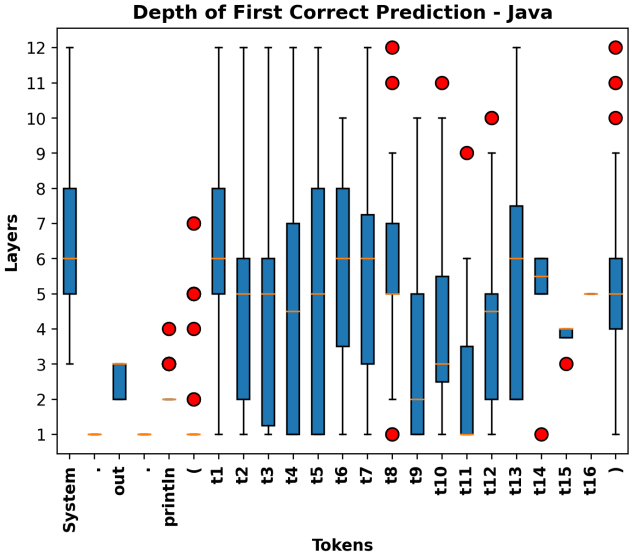


Figure 26: The mean and standard deviation of the DoFCC for common code structures with and without proper indentation in Go

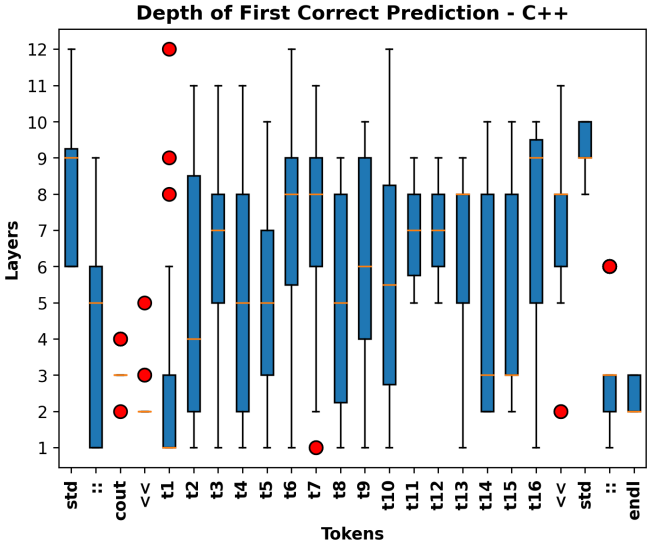
A.3 Print Statement Results

This subsection illustrates the additional results utilized in the comparative analysis of print statements across the languages investigated in this research. Since there are multiple approaches for implementing a print statement in certain languages, we also included the results of these different cases. Moreover, the results for Kotlin exhibit limited granularity, primarily attributed to the infrequent occurrence of print statements in the code. The presence of these syntax variations can influence the decision-making process of our model and reveal potentially meaningful patterns.



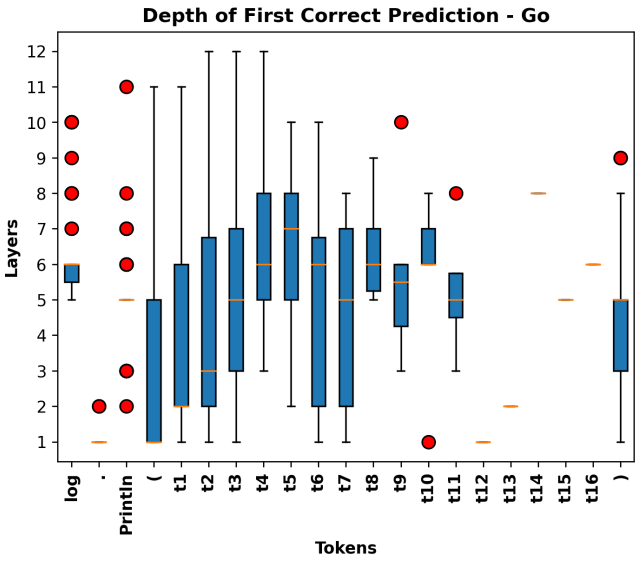
Standard Deviation	Frequency
'System': 1.68	'System': 69
'.': 0.0	'.': 69
'out': 0.47	'out': 69
'.': 0.0	'.': 69
'println': 0.46	'println': 69
'(': 1.79	'(': 69
't1': 2.39	't1': 66
't2': 2.82	't2': 61
't3': 2.82	't3': 58
't4': 3.18	't4': 52
't5': 3.06	't5': 49
't6': 2.93	't6': 39
't7': 2.95	't7': 36
't8': 2.58	't8': 28
't9': 2.48	't9': 24
't10': 2.83	't10': 19
't11': 2.37	't11': 16
't12': 3.13	't12': 14
't13': 3.26	't13': 11
't14': 1.77	't14': 6
't15': 0.43	't15': 4
't16': 0.0	't16': 3
')': 2.19	')': 69

Figure 27: The mean and standard deviation of the DoFCC for print statements in Java



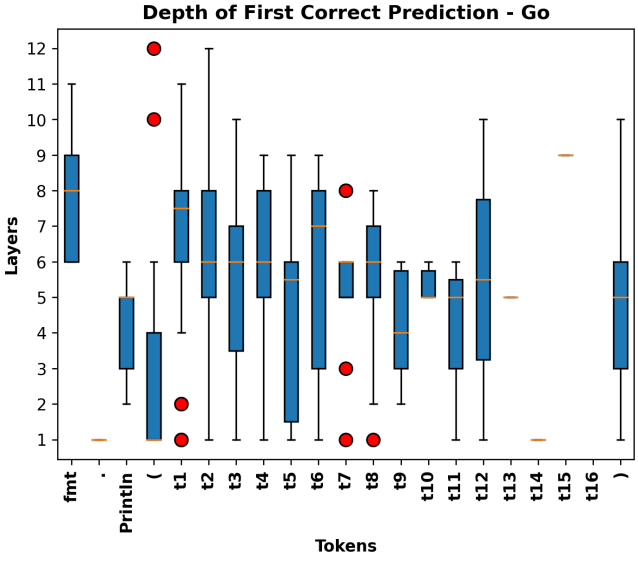
Standard Deviation	Frequency
'std': 1.75	'std': 32
'::': 2.43	'::': 32
'cout': 0.35	'cout': 32
'<<': 0.57	'<<': 32
't1': 2.71	't1': 31
't2': 3.49	't2': 31
't3': 2.4	't3': 27
't4': 3.18	't4': 23
't5': 2.6	't5': 22
't6': 2.91	't6': 19
't7': 2.91	't7': 19
't8': 2.9	't8': 18
't9': 3.1	't9': 15
't10': 3.32	't10': 12
't11': 1.36	't11': 8
't12': 1.41	't12': 6
't13': 2.93	't13': 5
't14': 3.35	't14': 5
't15': 3.19	't15': 5
't16': 4.03	't16': 3
'<<': 1.91	'<<': 32
'std': 0.63	'std': 32
'::': 1.5	'::': 32
'endl': 0.47	'endl': 32

Figure 28: The mean and standard deviation of the DoFCC for print statements in C++, without using the *std* namespace import



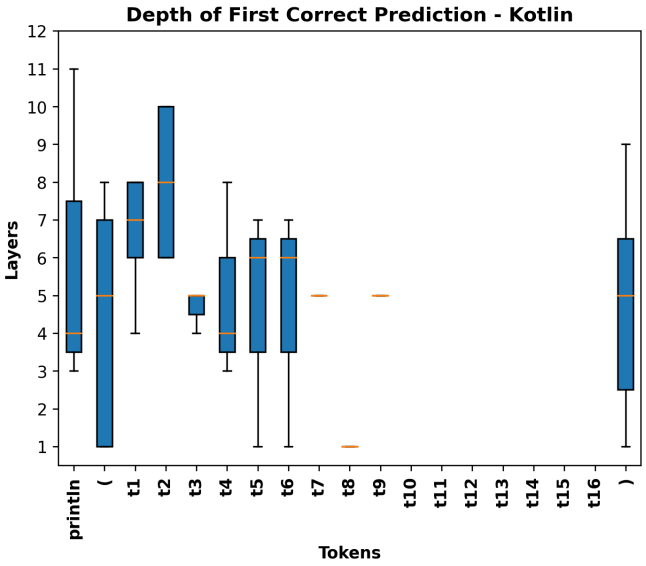
Standard Deviation	Frequency
'log': 1.32	'log': 59
'.': 0.18	'.': 59
'Println': 1.39	'Println': 59
'(': 2.2	'(': 59
't1': 2.76	't1': 59
't2': 3.12	't2': 50
't3': 2.91	't3': 47
't4': 2.16	't4': 22
't5': 1.99	't5': 15
't6': 2.96	't6': 14
't7': 2.47	't7': 13
't8': 1.28	't8': 10
't9': 2.21	't9': 6
't10': 2.42	't10': 5
't11': 1.79	't11': 4
't12': 0.0	't12': 1
't13': 0.0	't13': 1
't14': 0.0	't14': 1
't15': 0.0	't15': 1
't16': 0.0	't16': 1
')': 1.69	')': 59

Figure 29: The mean and standard deviation of the DoFCC for print statements in Go, using the *log* package



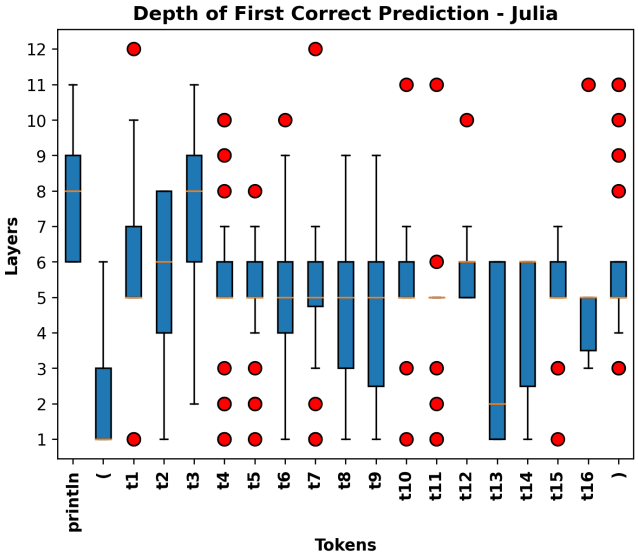
Standard Deviation	Frequency
'fmt': 1.59	'fmt': 53
'.': 0.0	'.': 53
'Println': 1.07	'Println': 53
'(': 2.38	'(': 53
't1': 2.25	't1': 52
't2': 2.62	't2': 49
't3': 2.45	't3': 43
't4': 2.65	't4': 32
't5': 2.5	't5': 22
't6': 3.11	't6': 15
't7': 1.85	't7': 12
't8': 2.2	't8': 9
't9': 1.57	't9': 6
't10': 0.47	't10': 6
't11': 2.16	't11': 3
't12': 4.5	't12': 2
't13': 0.0	't13': 2
't14': 0.0	't14': 1
't15': 0.0	't15': 1
't16': N/A	't16': 0
')': 2.34	')': 53

Figure 30: The mean and standard deviation of the DoFCC for print statements in Go, using the *fmt* package



Standard Deviation	Frequency
'println': 2.91	'println': 7
'(': 2.96	'(': 7
't1': 1.49	't1': 6
't2': 2.0	't2': 4
't3': 0.47	't3': 3
't4': 2.16	't4': 3
't5': 2.62	't5': 3
't6': 2.62	't6': 3
't7': 0.0	't7': 2
't8': 0.0	't8': 1
't9': 0.0	't9': 1
't10': N/A	't10': 0
't11': N/A	't11': 0
't12': N/A	't12': 0
't13': N/A	't13': 0
't14': N/A	't14': 0
't15': N/A	't15': 0
't16': N/A	't16': 0
')': 2.76	')': 7

Figure 31: The mean and standard deviation of the DoFCC for print statements in Kotlin



Standard Deviation	Frequency
'println': 1.73	'println': 40
'(': 1.81	'(': 40
't1': 2.09	't1': 39
't2': 2.12	't2': 37
't3': 1.94	't3': 34
't4': 2.43	't4': 30
't5': 1.51	't5': 30
't6': 2.27	't6': 30
't7': 2.12	't7': 28
't8': 2.12	't8': 24
't9': 2.01	't9': 23
't10': 1.74	't10': 21
't11': 2.0	't11': 20
't12': 1.15	't12': 18
't13': 2.36	't13': 16
't14': 2.15	't14': 16
't15': 1.46	't15': 16
't16': 2.69	't16': 6
')': 1.83	')': 40

Figure 32: The mean and standard deviation of the DoFCC for print statements in Julia