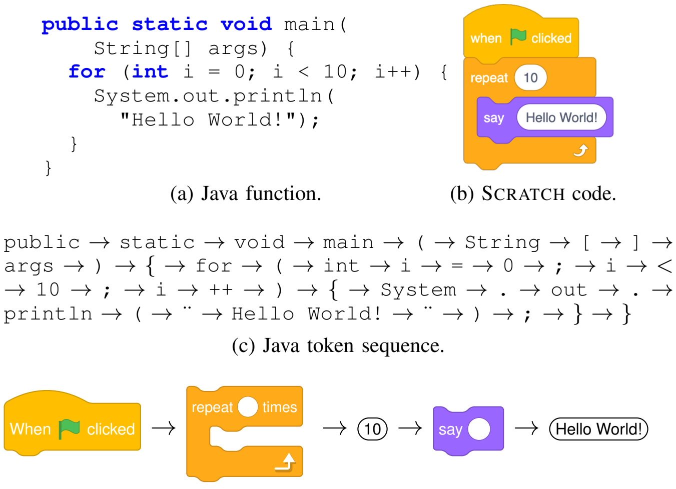
On the Applicability of Language Models to Block-Based Programs

Elisabeth Griebl*, Benedikt Fein*, Florian Obermüller*, Gordon Fraser*, René Just†
*University of Passau, Passau, Germany
†University of Washington, Seattle, USA
{elisabeth.griebl, benedikt.fein, florian.obermueller, gordon.fraser}@uni-passau.de, rjust@cs.washington.edu

***Abstract*—Block-based programming languages like SCRATCH are increasingly popular for programming education and end-user programming. Recent program analyses build on the insight that source code can be modelled using techniques from natural language processing. Many of the regularities of source code that support this approach are due to the syntactic overhead imposed by textual programming languages. This syntactic overhead, however, is precisely what block-based languages remove in order to simplify programming. Consequently, it is unclear how well this modelling approach performs on block-based programming languages. In this paper, we investigate the applicability of language models for the popular block-based programming language SCRATCH. We model SCRATCH programs using n-gram models, the most essential type of language model, and transformers, a popular deep learning model. Evaluation on the example tasks of code completion and bug finding confirm that blocks inhibit predictability, but the use of language models is nevertheless feasible. Our findings serve as foundation for improving tooling and analyses for block-based languages.**

***Index Terms*—Block-Based Programs, Scratch, Natural Language Model, Code Completion, Bugram**



```
public static void main(
    String[] args) {
  for (int i = 0; i < 10; i++) {
    System.out.println(
      "Hello World!");
  }
}
```

(a) Java function.

(b) SCRATCH code.

public → static → void → main → (→ String → [→] → args →) → { → for → (→ int → i → = → 0 → ; → i → < → 10 → ; → i → ++ →) → { → System → . → out → . → println → (→ ¨ → Hello World! → ¨ →) → ; → } → }

(c) Java token sequence.

(d) SCRATCH token sequence.

Figure 1: Example code in text- and block-based format.

I. INTRODUCTION

Block-based programming languages are becoming increasingly popular for education [1] as well industrial applications requiring end-user programming [2]–[4]. The distinguishing feature of these programming languages is that they reduce the syntactic overhead that is common for text-based languages, and instead represent programming constructs using graphical blocks which can only be combined in syntactically valid ways. Figure 1a shows a JAVA function that prints "Hello world!" 10 times; the same functionality can be implemented in the popular block-based language SCRATCH [5] with only three blocks (Fig. 1b). Programming is usually further simplified by explicitly listing all available blocks in the user interface, such that programmers neither need to memorize syntax nor available commands and APIs (recognition over recall).

Just like for programs written in text-based programming languages, there is a need to apply program analysis also to block-based programs: Learners may benefit from automatically generated hints and feedback, and programmers may benefit from code completion or bug detection. A popular approach to implement such analyses is to treat source code like natural language, and thus benefit from the recent proliferation of research on natural language processing (NLP) methods. At the core of these methods lies the concept of language models, which capture the probability distributions over sequences of words. As source code has been observed to exhibit regularities that make it amenable to natural language processing [6], the same models can also be used to capture probability distributions for source code. These models can, for example, predict common code sequences for code completion [7], or identify unusual code sequences for bug detection [8], [9].

Language models are constructed by interpreting the source code as sequences of *tokens*, i.e., individual words or symbols separated by whitespace or deliminator characters. The JAVA program in Fig. 1a would thus be interpreted as the sequence of 39 tokens shown in Fig. 1c. The SCRATCH version of the same program (Fig. 1b) results in a simple stream of only five tokens (Fig. 1d). On the one hand, this difference can be interpreted as strong reaffirmation of just how much block-based programming reduces the cognitive overhead compared to text-based programming languages [10]. On the other hand, it is unclear how this simplification changes the resulting language models. Even when removing "syntactic" tokens [11], the remaining tokens in the JAVA example are intuitively at a lower level of abstraction than the tokens of the SCRATCH program, which contains less technical details such as modifiers or types. Consequently, it is unclear how suitable language models are for program analysis tasks on block-based programs.

In order to investigate whether block-based programs can be modelled and analyzed using language models, we empirically investigate models based on programs written in the SCRATCH programming language [5], which is the most popular block-based programming language and aims at young learners. There is a thriving community of millions of users who share their programs, thus providing large amounts of code, allowing us to perform an extrinsic evaluation of the language models. In detail, the contributions of this paper are as follows:

- We describe and implement the process of creating n-gram models from SCRATCH programs. While there are various alternative neural models, n-gram models have been shown to perform well for many tasks, and a sound understanding requires interpretable models.
- We evaluate the suitability of n-gram models for the common tasks of code completion, i.e., the prediction of which block will be used next in a token stream, using a dataset of publicly shared SCRATCH projects.
- We evaluate the ability of n-gram models to identify erroneous solutions for SCRATCH programming assignments.
- We investigate whether transformers, a popular deep learning model, can improve the performance of the completion task compared to n-gram models.

Our experiments confirm that block-based programs differ fundamentally from text-based programs in a way that negatively affects their predictability. However, there nevertheless are elements of syntax and repetitiveness that make blocks sufficiently predictable to enable the use of natural language models for block-based programming languages.

II. BACKGROUND

Block-based programming languages have recently received increased attention for teaching programming concepts to novices [1] as well as for industrial applications requiring end-user programming [2]–[4]. In this paper, we focus on the popular educational programming language SCRATCH [5].

A. The Scratch Programming Language

SCRATCH [5] is a block based programming language for young learners. SCRATCH programs control the behavior of sprites in an environment (stage); each sprite can implement multiple scripts. Figure 1b exemplifies such a script: Scripts start with event handlers (e.g., When ⚑ clicked) followed by blocks that are executed after the event occurred. To support recognition over recall blocks are color coded based on categories: control structures are orange like the repeat 10 block in Fig. 1b, blocks affecting the visual appearance of sprites are purple (e.g., say Hello World!), etc. Blocks are further divided into different shapes, such as stackable blocks (statements) and round or diamond-shaped reporter blocks that fit into holes in other blocks (expressions). Blocks may have free text spaces for numbers and strings like in say Hello World!, and drop-down menus to select pre-set options. SCRATCH enables a remix culture [10] where users share their programs, and others clone and enhance them.

Even though the block shapes prevent syntactical errors, building programs can nevertheless be challenging: learners may struggle to implement functionality or may have misconceptions [12]–[15], and programmers may miss the convenience and support of modern programming environments. As a consequence, various analysis tools have been proposed, mainly implementing traditional program analyses such as linting (e.g., [16], [17]) or automated testing (e.g., [18], [19]). However, analysis tools using NLP methods are, to the best of our knowledge, not available yet.

B. N-gram Models

Probabilistic language models are used to assign probabilities to sequences of tokens in a given language. N-gram language models are based on the Markov assumption, which states that the probability of a sentence s can be estimated based on a chain of probabilities for all its tokens $w_1 \dots w_n$ to occur. N-gram models further simplify this idea and assume that each word actually depends only on its $n-1$ preceding words, that is, on its *local context*. Given $n = 3$ and $s = \langle w_1 w_2 w_3 w_4 \rangle$, an n-gram model thus estimates the probability of s as follows:

$$P(s) \approx P(w_1) \times P(w_2|w_1) \times P(w_3|w_1 w_2) \times P(w_4|w_2 w_3)$$

In contrast to regular Markov chains, the probability of w_4 is estimated considering only the context $w_2 w_3$, not $w_1 w_2 w_3$. The factors of the product are conditional probabilities estimated using the count of occurrences of tokens in the training data. For example, the probability $P(c|ab)$ with local context ab has a probability of 1, if only c follows ab in the training data.

A probability of 1 is rather unrealistic in practice: It is more likely that the training data did not contain all possible tokens that may follow the local context. Therefore, smoothing algorithms shift the raw probabilities based only on the counts of the n-grams, so that other n-grams that are not part of the training data are also assigned probabilities > 0. This way, the model is not "infinitely surprised" by n-grams other than those present in the training data set. A popular smoothing mechanism is modified Kneser-Ney smoothing [20], which has been reported to perform best in natural language contexts [21] and was also used in prior work on programming languages [6].

C. Deep Learning Models: Transformer

While n-gram models are valued for their simplicity and ease of interpretation, research has recently shifted towards neural approaches. Vaswani et al. [22] proposed the transformer architecture to enable capturing long range information during automated natural language translation. The transformer design makes use of the encoder-decoder architecture [22]: In the encoding part the model learns weight matrices for different word relations that encode how strong a word-encoding is influenced by every other word within the sequence [22]. During decoding the next generated token is influenced not only by the previously generated output, but also by the weight matrices over the whole input sequence [22]. Transformers allow for self-supervised learning, e.g., by masking random tokens in input sequences and training the transformer to predict the missing words [23]. These models tend to require a more computationally expensive training process compared to n-gram models, yet it has been shown that a transformer trained on a large dataset (e.g., BERT [23], CodeBERT [24]) can be used without or with only little fine-tuning to assist with tasks in the domain of source code processing [25]–[28].

D. *Program Analysis with Language Models*

Hindle et al. introduced the "naturalness hypothesis" based on which they proposed to use n-gram models to model source code [6]. As an initial application, they presented a simple code completion based on the n-gram model, which suggests a ranking of the most likely tokens based on the local context of the completion. That is, one maximizes the probability of the complete sequence by choosing the most probable n-gram based on the given local context.

The idea of benefiting from concepts of NLP and combining it with other techniques has been taken up successfully in other work, for example, in the area of code completion [7], [29]. However, further investigations of the naturalness hypothesis have also shown that large parts of the naturalness of code are due to syntactic elements such as parentheses or semicolons [11]. Even though source code is less natural than previously thought, regularities can still be found in the source code even after removing certain syntactic elements. In particular, despite the limitations of the naturalness hypothesis, n-gram models have been determined to be capable of representing source code very well, often better than deep neural networks [30], when appropriately configured.

If source code is regular, then irregularities in the source code are suspicious: Ray et al. demonstrated that buggy code has a higher entropy than correct source code [9]. This insight enables the application of n-gram models to identify bugs in source code. In particular, Wang et al. introduced BUGRAM, an automated approach for finding bugs based on n-gram models [8]. Given a specific project, BUGRAM trains an n-gram model on the source code, calculates probabilities for all sequences in the source code, and reports sequences with low probability as potential bugs.

Due to their specific structure, transformers open up further possibilities for code analysis. They can be used to jointly learn from code and natural language by training the model on sequences containing both tokenized code and its documentation to enable code search using natural language descriptions or generating documentation [24]. Those pre-trained models can then also be applied to code-only tasks like identifying buggy code [31] and generating potential fixes for it [32].

III. LANGUAGE MODELS FOR SCRATCH

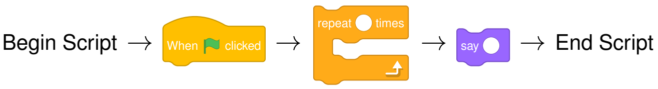
SCRATCH differs from text-based programming languages, to which language models have been previously applied. Thus, we first need to define how to tokenize SCRATCH programs. We then describe how n-gram models are generated and how they are applied for code completion and bug finding. We further describe how we obtain the transformer, and how it can be used for code completion.

A. *Tokenizing* SCRATCH *Programs*

Tokenizing text-based programming languages is straightforward, e.g., by directly lexing the source code. It is less obvious, however, how to tokenize SCRATCH programs: a SCRATCH program consists of a ZIP-file containing resources (images, sounds) as well as a text file in JavaScript Object Notation (JSON) format describing the code. The JSON file describes a program in terms of the *targets* (i.e., stage and sprites), and each target consists of its name, variables, lists, messages, sounds, costumes, scripts, procedures (i.e., custom blocks), and blocks. The blocks are listed in an arbitrary order (e.g., the order in which they were inserted in the program), and each block consists of a unique identifier as well as the identifiers of the parent and successor blocks, as well as any parameter blocks. The block identifiers and their parent/child relations are used in the SCRATCH virtual machine to create a syntax-tree-like representation. Although the JSON format is specific to SCRATCH, other block-based languages represent programs similarly; for example, SNAP! encodes blocks in XML [33]. In order to tokenize a program, we first use the parser provided by the LITTERBOX [16] analysis framework and create the abstract syntax tree for that program. We then traverse the syntax tree in preorder, adding each traversed node that represents a concrete block to the token stream. The resulting sequence of tokens is illustrated in Fig. 1c.

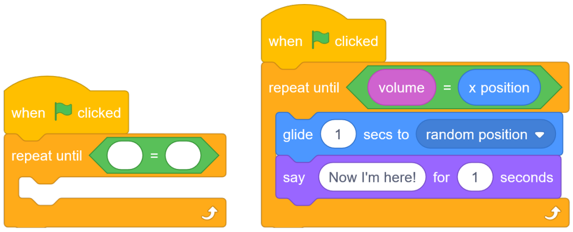
To reduce the vocabulary size and avoid out-of-vocabulary issues [34], we treat literals as follows [35]: First, we do not include string and number literals. On one hand, predicting literals is very difficult; on the other hand, text and numbers entered by users are usually very dependent on the use case. Second, similar to prior work [36], [37], we generalize the occurrence of concrete variables to the occurrence of the generic variable block var and calls to self defined blocks as procedures call call. Since most programs actually define only a few variables and procedures, this is on the one hand potentially not a very large loss of information, but on the other hand simplifies generalization across project boundaries.

Even though SCRATCH treats the drop-down menus that some blocks include as individual blocks in its internal representation, we do not include these as tokens as they are inseparable and are tailored to the specific block and use case. Thus, overall we only include statement and expression blocks as tokens, which results in a vocabulary size of 137 blocks. As usual in NLP, we introduce structural blocks for the beginning and ending of scripts or sprites. Consequently, the remaining token sequence of the script in Fig. 1b looks as follows:



B. *Code Completion using N-gram Models*

N-gram models are the most fundamental type of language models. They work with relatively small amounts of training data compared to more modern deep learning approaches, and have been successfully applied to various software engineering tasks (cf. Section II-D), and are therefore implemented as a baseline for the use of language models in our work. The first task on which we apply the n-gram model is code completion. The idea is to provide code completion for the next block

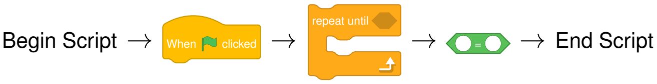

when clicked
repeat until =
when clicked
repeat until volume = x position
glide 1 secs to random position
say Now I'm here! for 1 seconds

(a) Incomplete script during development.

(b) Completed script using an uncommon condition in the repeat block.

Figure 2: Example of a SCRATCH script.

when interpreting the preceding code linearly (i.e., in the order of the token sequence). Fig. 2a shows a simple SCRATCH program during development. The existing code tokens would be interpreted in the following order:



For a 3-gram model, repeat until and = represent the local context to be completed by the model. Thus, in this case the code completion shall suggest blocks that are usually used for equality comparisons in while-loops, such as var, costume number, or round. To this end, we build a "general" n-gram model using modified Kneser-Ney smoothing on a large set of SCRATCH programs as described in Section III-A.

A special case for code completion in SCRATCH exists in the case of procedure definition blocks: Analogous to method headers in JAVA, these blocks build the header of a newly created custom block. Since procedure definitions are not added using drag and drop like other blocks, but using a dedicated dialogue to set the name and select possible parameters, we exclude procedure definitions from predictions. Another special case is the behavior of code completion when the model predicts the end of a block sequence based on the End Script blocks that were observed during training. In this case, instead of this prediction, which has no value for a user, the completion returns a prediction for a new first block, i.e., the completion for the context at the beginning of a script.

To the best of our knowledge, this is the first implementation of code completion for block-based programming languages. Therefore, we implemented a simple code completion based purely on the probabilities of the n-gram model, which allows us to evaluate how well the language model itself represents the language, and provides a baseline for further research.

C. Bug Finding using N-gram Models

Language models can reflect that buggy code is less regular than non-buggy code [9]. For example, Fig. 2b shows a SCRATCH script including a very unlikely condition in the loop block repeat until: Comparing the audio volume volume to the x position of a sprite x position is very unlikely to be a meaningful comparison. Ideally, a bug finding approach would be capable of capturing these and other irregularities in the source code.

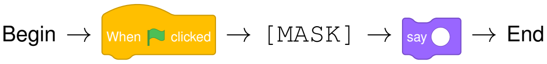
This principle has been applied in the BUGRAM approach [8] (cf. Section II-D). As SCRATCH projects are much smaller than JAVA or PYTHON projects, and likely contain less repetitive API usages, BUGRAM is not directly applicable. However, a common scenario in an educational context is that students implement a task for which there are one or more model solutions. Thus, we train an n-gram model on model solutions or known good student solutions, and then assess the probabilities of all sequences in the student solutions, reporting those with a particularly low probability. Since this model is not trained on the entire code base (i.e., all student solutions), unlike in the BUGRAM approach, the use of a smoothing algorithm is necessary, for which we again use modified Kneser-Ney smoothing. Additionally, we skip any preprocessing applied by BUGRAM, such as including the whole path in method names or skipping tokens with a particularly low count as proposed by Wang et al. [8] when parsing the source code, due to the simple structure of SCRATCH. Thus, we structurally use the same model for code completion and the bug finding task.

D. Code Completion using a Transformer

While n-gram models are simple, light weight, and easily interpretable, transformers are more contemporary and often yield better results [25], [38]. However, they are limited to tasks where very large amounts of training data are available, which in our study includes only the code completion task.

Using the script tokenization (Section III-A) we generate one token sequence $[t_0, \ldots, t_n]$ per sprite by concatenating the token sequences from all procedure definitions followed by scripts. Since users can place their blocks freely on the canvas, we tokenize the scripts in the order which they appear in the project file. This usually represents the order in which the user created them. For training we used the RoBERTa [39] implementation provided in the PYTHON *transformers* library [40]. As identifiers are removed from the token sequence, the model does not have to handle out-of-vocabulary scenarios and the vocabulary is reduced to 137 different tokens. Therefore, a simple word-level tokenizer that assigns a numeric identifier for each token is used.

The model is trained using a masked language model [23] with the default RoBERTa approach of randomly masking tokens [39]. In this approach tokens are randomly replaced by a placeholder `[MASK]` for which the original token then has to be predicted based on the surrounding context:



When using the trained model for code completion, a sequence of up to the last $m-1$ tokens of the existing code is extracted and the additional `[MASK]` token appended. The model then predicts a probability for each token to replace this mask, so that a top-x selection of suggestions can be presented to the user. Analogous to the completion with n-gram models, procedure definitions are excluded as described in Section III-B, and predictions for the end of a script are replaced by suggestions for a new script.

IV. Evaluation

As a baseline to gain an understanding of the applicability of language models in the context of block-based programming languages, we experimentally examine n-gram models on Scratch programs from two opposite angles: First, we consider how well n-gram models capture the regularities of Scratch programs by looking at the highest probabilities encoded in the model, using the task of code completion. Second, we consider how well n-gram models detect deviations from common patterns by looking at the lowest probabilities encoded in the model, using the task of bug finding. Finally, we investigate whether predictions can be improved using deep learning models. This leads to the following research questions:

- **RQ1, Completion:** How well does code completion based on n-gram models perform on Scratch source code?
- **RQ2, Bug Finding:** How well does bug finding based on n-gram models perform on Scratch source code?
- **RQ3, Model Comparison:** Do transformer-based deep learning models improve over n-gram models?

A. RQ1: Code Completion with N-gram Models

1) Experimental Setup: To create a general model for the purpose of code completion, we trained an n-gram model on 100 000 randomly selected Scratch programs. Between May 2021 and February 2022 we retrieved the 10 000 most recently publicly shared Scratch programs each day using the REST API of the Scratch website, resulting in a total 2.7 million projects. From these, we filtered projects with less than 10 blocks, as these very often represent projects in which children focused on arts and drawing, e.g., drawing a background and arranging sprites on it, rather than coding. Furthermore, we excluded remixes (i.e., copied and modified programs). From the resulting 1.1 million projects, we then randomly sampled 110 000, which we split into a training set of 100 000 projects, and an evaluation set of 10 000 projects. Comparing the number of blocks between projects in the training and evaluation sets shows that there is no significant difference ($p = 0.191$ using a Mann-Whitney U test [41]), thus confirming that the two datasets are drawn from the same overall distribution.

To choose a suitable value for the sequence length n, we trained the model for $n = \{1, 2, 3, 4\}$. We used 4 as the upper bound for two reasons: First, we observed only marginal improvements for larger n, which is in line with findings in prior work [6], [30]. Second, the models require substantially more memory and computation time as n increases. This is particularly relevant in the bug-finding use case, where a custom model is trained, e.g., for a given set of programs, in order to identify unusual sequences. In the context of finding bugs in programming assignments, which tend to use relatively small and simple Scratch programs, training a complex model may not be worthwhile. We use the same 100 000 randomly selected Scratch programs for each n-gram model.

The projects from the evaluation data set are broken down into sets of local contexts, each consisting of $n-1$ blocks. Every context is given to the completion engine that is asked to predict the next block for this context. The completion engine returns the top $x = \{1, 2, 3, 5, 10\}$ blocks ranked by their probability, and we evaluate the code-completion suggestions in terms of top-x accuracy, i.e., the ratio of suggestions that contained the actual block in the original program. We consider top-x accuracy for varying x and n, and we evaluate the influence of block frequency, category, and shape on top-x accuracy.

Table I: Accuracy of code completion in top x suggestions for different values of n.

	Top 1	Top 2	Top 3	Top 5	Top 10
1-gram	6.24 %	12.27 %	17.17 %	25.88 %	44.38 %
2-gram	23.22 %	34.56 %	42.54 %	54.03 %	69.44 %
3-gram	31.41 %	43.87 %	52.08 %	62.82 %	76.07 %
4-gram	36.31 %	49.05 %	56.77 %	66.76 %	78.35 %
transformer	33.83 %	43.78 %	49.91 %	57.79 %	69.32 %

2) Threats to Validity: Threats to external validity arise as results may not generalize to projects outside our dataset. We confirmed that in terms of size the sample is a valid representation of publicly shared projects; however, unfinished, unshared programs might have other properties. To avoid skewing results with very similar code we used only original projects and excluded remixes. Threats to internal validity may arise from our implementation: Although we tested and validated all code thoroughly, our implementation may confound the studied measurements and relationships. For example, rare aspects of the Scratch program representation not encountered during testing may be misrepresented. Threats to construct validity may arise from our choice of top-x accuracy as metric rather than precision or recall. This choice is based on the use case: We assume that each suggestion in the top-x is equally useful. Indeed a deployed code-completion engine can suppress low-confidence predictions, and a user does not have to accept incorrect suggestions.

3) Results: Table I shows the top-x accuracy of code completion for $n = \{1, 2, 3, 4\}$. Top-x accuracy is defined as the sum of all true positive predictions per block divided by the total number of predictions. A prediction is considered a true positive if the set of top-x suggestions contains the actual block to be predicted. By definition, top-x accuracy monotonically increases with increasing x. We also observe that it increases with increasing n. Since 4-grams perform best, we use 4-grams for the rest of RQ1. Furthermore, we use top-3 accuracy for subsequent evaluations as 3 is a reasonable number for suggestions in the Scratch user interface (e.g., in the "backpack" of code snippets) satisfying the design philosophy of Scratch to keep the cognitive load low [10].

The overall best predicted block is When ⚑ clicked with a top-3 accuracy of 95.52 %. With an occurrence rate of 6.03 % this is the second most frequent block overall in the evaluation data. Consequently, it is likely that occurrence has an influence on the performance of the prediction. We note that the top blocks in other categories show a substantially lower top-3 accuracy, thus there appears to be an influence also of the category. Finally, we observe that generally oval and diamond shaped blocks all have

Table II: Completion accuracy by category for n=4, x=3.

Group	Occurrences	Accuracy	Acc. Transformer
sound	2.5 %	32.9 %	44.7 %
pen	0.9 %	34.0 %	25.1 %
myblocks	0.9 %	41.3 %	60.6 %
event	15.3 %	48.0 %	30.4 %
motion	12.2 %	51.5 %	48.9 %
looks	20.7 %	53.9 %	64.2 %
control	21.2 %	58.8 %	31.0 %
operator	9.6 %	66.3 %	53.2 %
data	11.4 %	68.0 %	66.3 %
sensing	5.3 %	73.6 %	75.9 %

Table III: Completion accuracy by shape for n=4, x=3.

Shape	Occurrences	Accuracy	Acc. Transformer
end	1.3 %	29.9 %	58.9 %
hat	13.7 %	47.6 %	16.9 %
stack	52.0 %	51.8 %	56.3 %
c	12.2 %	59.9 %	14.0 %
oval	11.5 %	74.9 %	67.6 %
diamond	9.4 %	75.2 %	70.7 %

particularly high accuracy values, suggesting that the shape of blocks also contributes to the prediction. In order to better understand what determines the overall prediction performance, we therefore investigate the influence of these three aspects: (1) the frequency with which blocks occur in practice; (2) the category the blocks belong to (e.g., motion, looks, . . .); and (3) the shape of the blocks (e.g., regular stackable blocks, event handler blocks, . . .).

Figure 3a summarizes these three aspects and their influence on the top-3 accuracy for $n = 4$: The plot is split into facets based on the 10 different block categories in which the blocks are sorted in the SCRATCH user interface. For each category, blocks are plotted based on the number of occurrences in the training data (x-axis) and the resulting top-3 accuracy (y-axis). Data points are color-coded based on their shape. In particular, *hat blocks* represent event handlers, *stack blocks* are regular statements, *oval* blocks represent reporters returning numerical or textual data, *diamond* shaped blocks represent Boolean values, *c*-shaped blocks are control structures such as loops and if-conditions, and *stop* blocks are statement blocks that cannot have successors. The "myblocks" category only contains one block because we generalize identifiers (see Section III-B), and only predict calls to these self-created blocks, not their creation. Consequently, data points for procedure definition blocks and their possible parameters are not included. To better understand the influence of block frequency, category, and shape on accuracy, we used multiple linear regression to model this relationship. We include occurrence as a continuous variable and category and shape as categorical variables. Since we are interested in whether the accuracy for particular categories and/or shapes, independently of occurrence, differs significantly from the average accuracy, we used deviation coding—comparing each level to the grand mean. Figure 3b shows the results of the regression analysis.

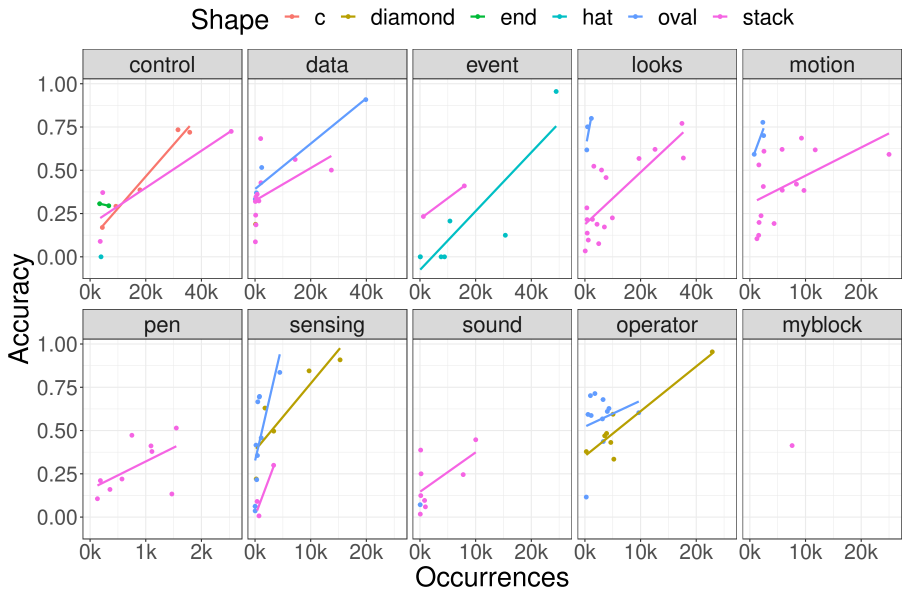
Table IV: Top 3 predictions for the example in Fig. 2a.

Block	Confidence
var	80.31 %
costume number ▾	5.67 %
answer	5.44 %

Figure 3a suggests differences between categories, which are summarized in Table II: Blocks of the sensing, data, operator, and control categories are predicted with a substantially higher accuracy than, for example, blocks from the sound or pen categories. One of the reasons for this lies in differences in the frequency of occurrence. For example, blocks of the pen or sound categories appear much less frequently than, for example, blocks from the motion or looks categories. The importance of occurrence can also be observed within categories, not just across categories. The fitted lines in Fig. 3a very clearly demonstrate that across all categories and shapes, the number of occurrences has a positive influence on the accuracy: The more frequently a block occurs in practice, the higher its probability of being predicted correctly. This is, for example, confirmed by the high occurrence and accuracy of the best predicted block of the event category (cf. Table II), i.e., When 🏁 clicked. Figure 3b confirms that the occurrence has a significant effect on the accuracy of the prediction, although the small coefficient of the regression model indicates the influence is small.

However, not all differences between the categories can be explained through the numbers of occurrences. For example, the categories sensing and operators can be predicted relatively well (Table II), even though blocks in these categories occur less frequently overall compared to, e.g., motion blocks. Figure 3b confirms some influence of the category; in particular, the motion category has a significant influence on the prediction accuracy. However, we observe that the well-predictable sensing and operator categories differ from others in an important property: they all contain a particularly large proportion of diamond-shaped and oval-shaped blocks, which represent expressions rather than statements (100 % expression blocks in operator, 83 % in sensing). Pen and sound blocks, for example, consist of almost only regular stack blocks. Figure 3a generally demonstrates with the different colors and fitted lines that stack blocks appear to be more difficult to predict than, for example, oval blocks (cf. categories data, looks, motion, sensing) or diamond-shaped blocks (cf. sensing). Table III supports this impression using the top-3 accuracy values grouped by the shape of the blocks. Figure 3b confirms that oval and diamond shapes have a significant positive influence on the prediction, whereas hat blocks have a significant negative influence.

Whenever there is a block in the local context that is usually followed by an expression block, this significantly limits the actual choice of matching blocks. The code example from Fig. 2a illustrates this phenomenon: repeat until ◆ has room only for diamond-shaped expression blocks. Round expression blocks are intended to go into the round placeholders of the ○ = ○ block, which reduces the successor blocks. The top blocks

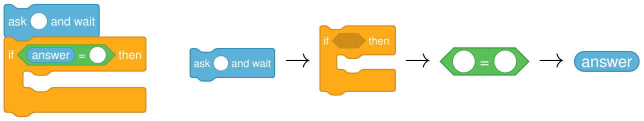


(a) Top-3 accuracy vs. occurrence of blocks, grouped by category and shape of blocks. (Note the differently scaled x-axes.)

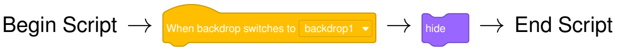
Variable	Coefficient	p
Intercept	2.94×10^{-1}	< 0.001
Occurrences	1.49×10^{-5}	< 0.001
Category		
control	-1.05×10^{-1}	0.216
data	9.81×10^{-3}	0.834
event	-4.62×10^{-2}	0.634
looks	5.69×10^{-3}	0.897
motion	1.08×10^{-1}	0.019
pen	5.72×10^{-2}	0.342
sensing	-2.68×10^{-2}	0.622
sound	-9.49×10^{-2}	0.108
operator	1.28×10^{-2}	0.831
myblock	7.96×10^{-2}	0.612
Shape		
c	-2.34×10^{-2}	0.797
diamond	1.55×10^{-1}	0.034
end	3.62×10^{-2}	0.763
hat	-2.86×10^{-1}	0.003
oval	1.92×10^{-1}	0.002
stack	-7.35×10^{-2}	0.146

(b) General linear model.

Figure 3: Influence of block occurrence, category, and shape on code-completion accuracy.



(a) Code for processing user input (left) and corresponding 4-gram (right). The probability of this 4-gram is 97.56 %.



(b) Hiding a sprite on change of the backdrop as an isolated script with no other blocks following. The 4-gram has a probability of 95.48 %.

Figure 4: Common language idioms as captured by the model.

suggested by the model for this scenario are shown in Table IV. The model is 80.58 % confident that a var should be inserted into the ○ = ○ block. costume number ▾ and answer have a comparably low probability of 5.69 % and 5.29 % compared to the variable block. While costume number ▾ is a sensible suggestion for this scenario, the answer block would only be usable if preceded by a ask ○ and wait block to which the answer block could refer to (cf. Fig. 4a for a usage example), but in Fig. 2a there is no such block. Nevertheless, all 3 suggestions do have in common that their shape makes them a syntactically correct building block. This effect is comparable to the prior observation that syntactic elements contribute substantially to the predictability of source code [11]. Accordingly, for categories like pen, sound, or motion, which do not contain much syntactical constraints, completion performs worse.

Finally, some regularity can be attributed to recurring idioms in common SCRATCH code. Considering n-grams to which the model assigns very high probabilities and excluding known patterns such as starting with a When 🏳 clicked block, we see that the model has learned some idioms that go beyond purely syntactic regularities. Similar to traditional programming languages, there are certain token sequences that repeat across the boundaries of specific programs. For example, the 4-gram in Fig. 4a has a probability of 97.56 %, and describes requesting a user input and reacting based on a comparison of this. While this sequence occurs much less frequently than programs beginning with When 🏳 clicked (7 465 vs. 499 146 occurrences), the model is very confident that answer follows as last block for this local context. This can only be partially explained by occurrence, category, and shape: Although answer is an oval block and thus fits syntactically well, there are numerous other oval blocks in SCRATCH. However, the *total* probability for *all* other blocks of all shapes to follow this local context is less than 2.5 %.

Other examples of idioms we observed include isolating functionality in very short scripts (c.f. Fig. 4b) and repetitions of the same blocks (e.g., inserting elements into a list). These idioms show a connection of the blocks on a semantic level and therefore indicate further repetitive, predictable structures apart from pure syntactic connections, i.e., shapes, and block frequency, i.e., occurrences. These idioms are comparable to those discovered in traditional programming languages.

Summary (RQ1, Completion): The best model (4-gram) achieves a top-3 accuracy of 56.77 %. Prediction quality is influenced by block frequency, shape, and category, but we also found recurring idioms influencing regularities.

B. RQ2: Bug Finding with N-gram Models

For RQ2 we use the n-gram model in the inverse way compared to RQ1: Rather than predicting the most likely blocks, we are interested in the least likely sequences of blocks.

In contrast to the evaluation in the Bugram paper [8], we assume that for the small student solutions we can identify *all* actual errors in the programs using tests. Thus, in our paper, a test suite acts as ground truth to evaluate whether the most unlikely sequences actually contain erroneous code. In the original paper, the authors looked for *undetected* bugs and refactoring possibilities in very large projects. Instead of comparing the code with a ground truth like existing tests, it was manually examined for improvement possibilities.

1) Experimental Setup: The application scenario of the bug finding task is a programming assignment given in an educational context. We use the dataset provided with the replication package of the Whisker [19] paper on testing Scratch programs, consisting of 41 student solutions and one model solution of a Fruit Catching game (c.f. Fig. 5a). The objective of the game is to catch as many apples and bananas as possible in 30 seconds by moving a bowl at the bottom of the screen. For bananas that touch the ground, the player loses points. The game is lost if an apple drops on the ground.

We trained a 3-gram model with modified Kneser-Ney smoothing on the model solution and one student solution which was deemed almost correct using automated tests [19]. We use $n = 3$ based on prior results of Wang et al., who found that 3-gram models perform best in finding bugs and refactoring opportunities [8].

Using this model, we determined the probabilities for all occurring sequences for the 41 student programs (that is, including the best student solution). Intuitively, sequences with lower probability assigned by the model are more likely to contain bugs. When extracting sequences we exclude “loose” code, i.e., blocks and scripts not connected to an event handler which are never executed. Since the ideal sequence length for this analysis has not previously been investigated in our context, we performed the evaluation for sequence lengths from 3 to 6. Longer sequence lengths are unlikely to be useful for the generally small Scratch programs, as the sequences otherwise would frequently exceed script boundaries.

To investigate whether low probability sequences indicate bugs, we randomly selected 10 of the 41 student solutions for manual validation, considering only those with at least 10 sequences, as they are otherwise unlikely to fully implement any aspects of functionality. For each of these 10 programs, we manually classified the 10 sequences with the lowest probabilities, for each of the sequence lengths in the range of 3 to 6. Two authors independently evaluated whether the corresponding sequences contained bugs or not. As ground truth for the existence of bugs we use the extensive Whisker test suite provided by Stahlbauer et al. [19] that fully covers the program behavior. We consider a sequence to contain a bug exactly if it causes the failure or skipping of one or more test cases. Thus for each sequence, we determined (1) whether the sequence contains at least one bug, and (2) for each failing test whether the sequence contributes to the failure. In the case of disagreement of the two independent classifications, these individual cases were discussed again until a consensus was reached. Thus overall, 400 sequences were manually classified.

Table V: Percentage of found bugs for each sequence length.

Sequence length	3	4	5	6
Bugs found (%)	57.58	62.88	64.39	56.06

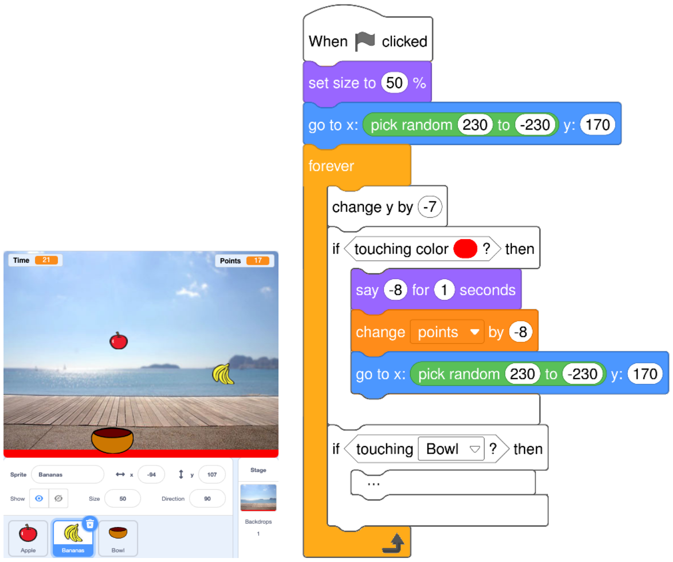
Table VI: For sequences of length 4, the bottom 10 most unlikely sequences and 10 random sequences, the Precision@10 for each sequence to contain at least one bug, as well as the proportion of found bugs in total, and the total number of bugs in the program.

	Precision@10		% Bugs Found		Bugs
	Bottom	Random	Bottom	Random	Total
K6_S01	90.0	40.0	96.3	37.04	27.0
K6_S12	60.0	90.0	80.8	26.9	26.0
K6_S15	60.0	30.0	62.5	31.3	16.0
K6_S18	60.0	50.0	50.0	50.0	6.0
K6_S31	30.0	10.0	75.0	12.5	8.0
K7_S03	60.0	40.0	27.3	45.5	11.0
K7_S10	20.0	0.0	28.6	0.0	7.0
K7_S14	30.0	20.0	60.0	20.0	5.0
K7_S17	10.0	10.0	50.0	50.0	2.0
K7_S24	70.0	50.0	33.33	20.8	24.0
Average	49.0	34.0	62.88	28.79	13.2
p	0.073		0.003		
$\hat{A}_{12}$	0.69		0.84		

As a baseline, we further selected and classified 10 random sequences per program using the best sequence length determined by the classification of low probability sequences, which allows us to determine if 10 most unlikely sequences are more likely bugs than random sequences.

2) Threats to Validity: Threats to external validity arise as our experiments are based on one task and student solutions from two school classes, so the results may not generalize to other tasks or classes. To avoid threats to internal validity, we randomized the selection of projects to avoid bias. As manual classification may be influenced by subjective interpretation, each sequence was independently classified by two authors of this paper to minimize the influence on the results (inter-rater reliability: 88.8 %). Furthermore, the same authors classified all sequences to ensure that the results are consistent and comparable to one another. To ensure construct validity of our evaluation, we rely on accepted measures for bug finding, considering the number of buggy sequences as well as the number of unique bugs.

3) Results: Table V lists the overall percentage of bugs found ($b = \#\text{bugs found}/\#\text{bugs in total}$) for different sequence lengths. Sequences of length 5 find the most bugs overall, closely followed by sequences of length 4; sequences of length 6 identify the fewest bugs. The minor difference between sequences of length 4 and 5 originates only from a single program, for which sequence length 5 finds significantly more bugs (K7_S03: 27.27 % vs. 90.91 %). In all other programs, the results of sequence length 4 are equal or even better. Since sequences of length 4 narrow down the source of the problem/bug better than sequences of length 5, all further results are based on a sequence length of 4 tokens.



```
When 🏴 clicked
set size to 50 %
go to x: pick random 230 to -230 y: 170
forever
    change y by -7
    if touching color ● ? then
        say -8 for 1 seconds
        change points ▾ by -8
        go to x: pick random 230 to -230 y: 170
    if touching Bowl ▾ ? then
        ...
```

(a) The Fruit Catching game.

(b) Student solution K7_S10 for Banana sprite. Colored blocks are examples for reported sequences that cause the failure of tests.

Figure 5: Example for Fruit Catching game with buggy student solution.

Table VI lists the results per program based on the least likely and random sequences. Precision@10 is given in terms of the number of sequences in the bottom 10 containing at least one bug, i.e., the probability of a sequence in the bottom 10 to contain an actual bug. The table also shows the proportion of all bugs found in each program. Since the programs are relatively small, there is frequently more than one bug per sequence; for reference, the table also lists the total number of bugs, which corresponds to the number of failed tests.

In terms of the percentage of buggy sequences, on average half the sequences among the least likely ones contain at least one bug, in contrast to only 34 % of the randomly selected sequences. The difference is not significant at $\alpha = 0.05$ ($p = 0.073$), but note that, since the sequences are randomly selected, there is some overlap with the 10 least likely sequences: on average 16 % of the randomly selected sequences are also among the bottom 10. However, randomly selected sequences of length 4 are only capable of finding 29 % of bugs in total, and they find no more than 50 % of the bugs in any program. In contrast, the least likely sequences find an average of 63 % of the bugs in total. The improvement of the least likely sequences over random sequences is statistically significant, with a large Vargha-Delaney effect size of $\hat{A}_{12} = 0.84$ and $p = 0.003$ using the Mann-Whitney U test. This confirms that indeed the n-gram model captures the expected structure of the solution.

For example, Fig. 5b shows student solution K7_S10 of the Banana sprite. The colored parts of the source code are two sequences of length 3 that are among the 10 least likely according to the model. For both examples, the code actually leads to failing test cases: For the first sequence, the solution violates the task specification that the Banana should start *one second after* the flag was clicked. The program, however, starts immediately with the for-loop that moves the Banana. For the second code sequence, the Banana is supposed to *disappear for one second* after the point deduction as a time penalty, but the buggy implementation causes the Banana to start dropping from the top again immediately.

We generally observed that blocks not included in the training data have a strong influence on the probability of a sequence. In particular, not only are such sequences assigned particularly low probabilities, but the same unusual block tends to influence the probability of several surrounding sequences. As a consequence, the least likely 10 sequences may in the worst case cover only a fraction of the program. For example, solution K6_S01 violates the task specification by starting the game (i.e., each script in each sprite) with pressing the up arrow key instead of the green flag. The model has not seen this block in the training data. As a result, 9 of 10 sequences reported least likely by the model, contain the unknown When up arrow ▾ key pressed block. We noticed the same behavior with uncommon, yet correct, blocks as well.

The original BUGRAM paper suggests filtering rare tokens [8]. Due to the much more limited vocabulary in SCRATCH we decided not to implement such an approach; however, the experimental results suggest that this could be a useful addition when implementing this approach in practice, although a challenge for this will be to not discard too large parts of the rather small student solutions, which would also cause potential bugs to go undetected.

Program K7_S17 is the overall best student solution, which was included in the training data of the model. The fact that one of the two bugs in this program was identified (Table VI) shows that, similar to the BUGRAM [8] approach, errors can even be found in programs which served as training data.

Summary (RQ2, Bug Finding): For 9 out of 10 programs, the number of bugs found using the model is greater than or equal to that of randomly selected sequences; this improvement is statistically significant.

C. RQ3, Comparison: Code Completion with Transformer

1) Experimental Setup: We trained the RoBERTa model [39] on a dataset obtained using the same procedure as described in Section IV-A1, but by sampling 500 000 programs with 517 431 sprites. To limit sequences to the maximum length m that can be processed by the model, we split them into subsequences: The first generated sequence for a program with n tokens is $[t_0, \ldots, t_{\min(n,m)}]$. Then the first script not fully included in this sequence is taken as the starting point for the next one. To embed the script into maximally possible context, tokens preceding and following this script are added symmetrically to fill the sequence up to a length of m. This is repeated until all tokens have been included in at least one subsequence. By not splitting sequences at script-level, it is also possible to predict when a script should finish and a new one should be started. Applying the sequence splitting resulted in 4 493 833 sequences for training. The same 10 000 programs as for RQ1 were used for evaluation.

We used the default RoBERTa hyperparameters as starting point for further tuning [39]. As the language to be modeled is smaller, and programs also tend to be small, the tuning decreased the maximum sequence length to 256, the number of hidden layers (12 to 2), their sizes (hidden size 256, intermediate size 512), and number of attention heads (12 to 4). The other parameters remained unchanged.

2) Threats to Validity: The same threats to validity apply as described in Section IV-A2. An additional threat to construct validity arises from the splitting into smaller subsequences. To mitigate this risk, we compared the results when splitting the sequence into one padded sub-sequence per script, and using non-overlapping chunks of tokens of the model's sequence length. We chose the strategy described in Section III-D as the others did not improve the prediction accuracy. Our results achieved using the RoBERTa model [39] might not generalize to alternative transformer-based models. This matches our aim of not maximizing any particular metric of model performance, but instead providing a baseline and evidence that investigating different models as part of future work is warranted.

3) Results: Table I shows the top-x accuracy of code completion. The accuracy for the first three suggested tokens is similar to the one for the 3-gram model, i.e., the second best n-gram model according to the results for RQ1.

Tables II and III give a more detailed insight which types of blocks can be predicted accurately. The transformer outperforms the 4-gram model (i.e., the best n-gram model) for predictions of end blocks (delete this clone and stop ▾) and performs similarly on regular statements (stack), Boolean expressions (diamond), and placeholders (oval). For script starts (hat) and branching blocks (c) the accuracy is worse compared to the n-gram model.

The discrepancy between being able to predict ends of scripts and new starts is noteworthy. Scripts in SCRATCH do not have to use an "end" block as terminal statement. Therefore, in most cases no clear indicator exists when a new script should start. Within the "hat" group of blocks the ones with the best accuracy are when backdrop switches to ▾ (35.3 %), when key ▾ is pressed (25.3 %), and when I receive ▾ (22.5 %). In those cases it is likely that corresponding statements to change the background or send messages are placed near the end of previous scripts, which can then be interpreted as hints that new scripts should start. Note that the "end" blocks are not necessarily placed at the end of a sprite sequence. Instead, the scripts within the sequence are saved in the same order as they were created by the user. Hence, the model cannot use the number of tokens following the "end" block to learn if a script should end.

For branching blocks (c-blocks) the transformer has substantially lower accuracy (14.0 %) compared to the n-gram model (59.5 %). This may be caused by the bidirectional attention that is applied during training: Using the surrounding context from both sides of the masked c-block, the model learns that they are in most cases, except for forever-loops, followed by a Boolean condition token. As this information is missing during the code completion task, there is not enough context to reliably predict the correct token. However, modifying the attention mechanism to only allow unidirectional attention on tokens preceding the masked one resulted in worse accuracy.

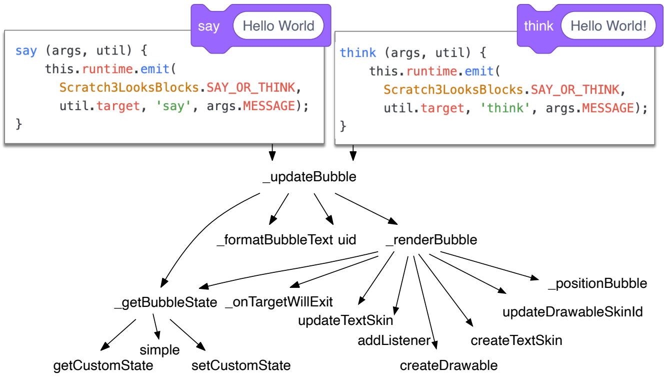
Overall, the prediction accuracy of the transformer is worse than the best n-gram model. Transformers can handle large vocabularies in the order of tens of thousands of different words [23], but this advantage is of little use as the tokenized SCRATCH language only has 137 words. Additionally, the usefulness of long range information to the next token prediction is not clear. The scripts are ordered in the sequence in which the user inserted the first block contained in it while creating the program and do not depend on each other in the program flow except for passed messages in between. For example, in the context within the same sprite some scripts might handle user inputs and the resulting movement while others are triggered on interactions with other sprites to play sounds or change the look. Therefore, we expect that the next token mostly depends on the short-range local context and the unrelated blocks of the other scripts act as distracting factor.

Summary (RQ3, Model Comparison): The transformer model performs comparable to the 3-gram model and thereby worse than our best n-gram model ($n = 4$). We conjecture that this is caused by the small vocabulary and the importance of short-range over long-ranged information for code completion.

V. DISCUSSION

Recent program analyses frequently build on the "natural hypothesis" [6], which assumes that programming languages have similar regularities as natural language, and source code is therefore amenable to natural language processing techniques. It has been shown that a certain degree of this observed regularity in source code is due to syntactic overhead in text-based programming languages. Specifically, Rahman et al. showed lower regularities when filtering common syntax tokens such as delimiters or nesting tokens [11]. Our investigation is related in that block-based programming languages explicitly *avoid* such syntax tokens, thus also making the language potentially less repetitive and predictable.

Compared to JAVA, the code completion for SCRATCH appears to perform slightly worse. For example, in a related study [29] based on a 3-gram model for JAVA, the top-1 accuracy was almost 20 % higher than the corresponding top-1 accuracy for the 4-gram model in SCRATCH. This study modeled the source code as a stream of lexical tokens including identifiers, keywords, or symbols, specified by the programming language [29]. Further, this study evaluated a completion task, using local context such as `<if, (, node>` and completion suggestions such as "`!=` `null`", "`== null`", and "`.isRoot()`" [29]. The task design is comparable to that of our completion task. Accordingly, we assume that the results are comparable to the extent possible across programming language boundaries. A difference in prediction accuracy of almost 20 % therefore suggests that there are substantial differences either in the properties of block-based vs. text-based programming languages or in the characteristics of their programs.



```
say (args, util) {
    this.runtime.emit(
      Scratch3LooksBlocks.SAY_OR_THINK,
      util.target, 'say', args.MESSAGE);
}
```

```
think (args, util) {
    this.runtime.emit(
      Scratch3LooksBlocks.SAY_OR_THINK,
      util.target, 'think', args.MESSAGE);
}
```

Figure 6: Two different SCRATCH blocks vs. their shared implementation and call-tree in JAVASCRIPT.

Based on recent trends in NLP and software engineering, one might expect deep learning models to make a difference here, but our results suggest this is not the case (cf. Section IV-C3). In a similar experiment on JAVA code by Ciniselli et al. the transformer model achieved a better prediction accuracy than their best n-gram model [25]. This suggests that there is a difference in how the model is able to use long-range information in block-based vs. text-based programming languages. To mitigate this, it may be possible to integrate more information into a deep learning model [38], [42]. For example, using the flat sequences as the input to the transformer removes all structural information from the code. Guo et al. [42] adapted the attention mechanism of their GraphCodeBERT model to focus on related code elements determined by the data flow graph. Similarly, for SCRATCH information about the relation of scripts (e.g., passed messages, changed sprite attributes) could be extracted [16] to help the transformer focus on scripts related to the one that should be completed and ignore other ones. This could improve the capture of long-range information in a long sequence containing several short scripts.

As part of RQ1 and RQ3 (Sections IV-A and IV-C), we discussed some aspects of block-based programs and their influence on predictability. Besides these aspects, there is another important difference between traditional, text-based programming languages and block-based programming languages that likely has an impact on repetitiveness and predictability: the level of abstraction at which source code is defined. Figure 6 shows the actual JAVASCRIPT code that is executed by a say or a think block, as well as the corresponding call tree of the corresponding callback functions in the SCRATCH virtual machine code. On the one hand, the example shows how abstract SCRATCH code is compared to traditional programming languages. Not only is syntactic overhead removed, but to some extent the code is also streamlined. On the other hand, this causes repetitions in the JAVASCRIPT code and thus may be interpreted as introducing code regularities: in the JAVASCRIPT implementation both the say and think blocks create, format, and render a speech bubble, and the call to `_updateBubble` is essentially the same for both blocks. However, in the SCRATCH code, the two blocks are distinct without any repetitiveness. The JAVASCRIPT code is thus more repetitive and predictable.

VI. CONCLUSIONS

A recent trend in software engineering research is to apply language models and NLP techniques to text-based programming languages for a multitude of different tasks. A niche of programming languages excluded from this trend so far is represented by block-based languages, which differ from text-based languages by some of the very properties that make NLP techniques applicable to source code. In order to shed light on the applicability of language models to block-based programs, we empirically studied n-gram and transformer models for SCRATCH. Although our results demonstrate that block-based languages are more challenging to predict, they nevertheless demonstrate that the approach is viable.

Prior work on code completion suggests various ways in which our baseline model could potentially be improved: For example, it is conceivable that other models, such as statistical graph models [7], [11] or neural models (e.g., [43], [44]), could improve performance. Further filtering of the vocabulary, e.g., to filter rare blocks [8], might lead to performance improvements [34], [35]. The performance could also be improved by taking additional context into account [45]; for example, for SCRATCH the context could be provided by the sprite or stage being edited. Similarly, it might be possible to build models for different types of programs; for example, games might differ fundamentally from animation or art projects in SCRATCH, thus leading to different models.

Besides the performance of the models, an important question for future work concerns the application of these models. For example, unlike text-based programming there is no text-cursor at which to display code completion, thus creating a usability challenge. Nevertheless, the inclusion of a code completion system into the SCRATCH user interface should be feasible, since the blocks are already presented grouped by their type. A modification of the interface could show the recommended blocks as such a group. However, the educational application domain may also suggest the need for custom models that take the education level into account; for example, code completion should not recommend blocks that require concepts a learner is not yet aware of, which could be determined using computational thinking metrics for SCRATCH [46]. Furthermore, beyond our initial bug finding task, we envision many possible applications of language models in the educational domain.

To support replication and future work, the source code of the tokenizer and the language models, the datasets, analysis scripts, and the raw results can be found at

https://doi.org/10.6084/m9.figshare.19382588.v1

ACKNOWLEDGEMENTS

This work is supported by BMWK (50RM2100B, “ANUKI”), Bayerische Forschungsstiftung (AZ-1520-21, “DeepCode”), and DFG (FR2955/4-1, “STUNT”).

REFERENCES

[1] M. M. McGill and A. Decker, "Tools, languages, and environments used in primary and secondary computing education," in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 103–109. [Online]. Available: https://doi.org/10.1145/3341525.3387365

[2] D. Weintrop, D. C. Shepherd, P. Francis, and D. Franklin, "Blockly goes to work: Block-based programming for industrial robots," in *2017 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 2017, pp. 29–36.

[3] N. Ritschel, V. Kovalenko, R. Holmes, R. Garcia, and D. C. Shepherd, "Comparing block-based programming models for two-armed robots," *IEEE Transactions on Software Engineering*, 2020.

[4] C. Mayr-Dorn, M. Winterer, C. Salomon, D. Hohensinger, and R. Ramler, "Considerations for using block-based languages for industrial robot programming - a case study," in *3rd IEEE/ACM International Workshop on Robotics Software Engineering, RoSE@ICSE 2021, Madrid, Spain, June 2, 2021*. IEEE, 2021, pp. 5–12. [Online]. Available: https://doi.org/10.1109/RoSE52553.2021.00008

[5] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, pp. 16:1–16:15, 2010. [Online]. Available: https://doi.org/10.1145/1868358.1868363

[6] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 837–847. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227135

[7] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, pp. 419–428. [Online]. Available: https://doi.org/10.1145/2594291.2594321

[8] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 708–719. [Online]. Available: https://doi.org/10.1145/2970276.2970341

[9] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the 'naturalness' of buggy code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 428–439.

[10] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. A. Turbak, "Learnable programming: blocks and beyond," *Commun. ACM*, vol. 60, no. 6, pp. 72–80, 2017. [Online]. Available: https://doi.org/10.1145/3015455

[11] M. Rahman, D. Palani, and P. C. Rigby, "Natural software revisited," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 37–48.

[12] F. Hermans, K. T. Stolee, and D. Hoepelman, "Smells in block-based programming languages," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2016, pp. 68–72. [Online]. Available: http://ieeexplore.ieee.org/document/7739666/

[13] F. Hermans and E. Aivaloglou, "Do code smells hamper novice programming? A controlled experiment on Scratch programs," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[14] P. Techapalokul and E. Tilevich, "Understanding recurring quality problems and their impact on code sharing in block-based software," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2017, pp. 43–51. [Online]. Available: http://ieeexplore.ieee.org/document/8103449/

[15] C. Frädrich, F. Obermüller, N. Körber, U. Heuer, and G. Fraser, "Common bugs in Scratch programs," in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 89–95. [Online]. Available: https://doi.org/10.1145/3341525.3387389

[16] G. Fraser, U. Heuer, N. Körber, F. Obermüller, and E. Wasmeier, "LitterBox: A linter for Scratch programs," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2021, pp. 183–188.

[17] P. Techapalokul and E. Tilevich, "Quality Hound — an online code smell analyzer for Scratch programs," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2017, pp. 337–338.

[18] D. E. Johnson, "ITCH: Individual testing of computer homework for Scratch assignments," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 223–227.

[19] A. Stahlbauer, M. Kreis, and G. Fraser, "Testing Scratch programs automatically," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 165–175.

[20] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," *Comput. Speech Lang.*, vol. 13, no. 4, pp. 359–393, 1999. [Online]. Available: https://doi.org/10.1006/csla.1999.0128

[21] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson, "One billion word benchmark for measuring progress in statistical language modeling," in *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, H. Li, H. M. Meng, B. Ma, E. Chng, and L. Xie, Eds. ISCA, 2014, pp. 2635–2639. [Online]. Available: http://www.isca-speech.org/archive/interspeech_2014/i14_2635.html

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[23] J. Devlin, M.-W. Chang, K. Lee, and K. 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, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[24] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[25] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of BERT models for code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2021, pp. 108–119. [Online]. Available: https://doi.org/10.1109/msr52588.2021.00024

[26] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 703–715. [Online]. Available: https://doi.org/10.1145/3468264.3468611

[27] R. Degiovanni and M. Papadakis, "μbert: Mutation testing using pre-trained language models," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2022, pp. 160–169.

[28] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: code generation using transformer," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1433–1443. [Online]. Available: https://doi.org/10.1145/3368089.3417058

[29] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 532–542. [Online]. Available: https://doi.org/10.1145/2491411.2491458

[30] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 763–773.

[31] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using CodeBERT model," *Applied Sciences*, vol. 11, no. 11, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/11/4793

[32] E. Mashhadi and H. Hemmati, “Applying CodeBERT for automated program repair of Java simple bugs,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 505–509.

[33] B. Harvey, D. D. Garcia, T. Barnes, N. Titterton, D. Armendariz, L. Segars, E. Lemon, S. Morris, and J. Paley, “Snap!(build your own blocks),” in *Proceeding of the 44th ACM technical symposium on Computer science education*, 2013, pp. 759–759.

[34] R. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, “Big code != big vocabulary: open-vocabulary models for source code,” in *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1073–1085. [Online]. Available: https://doi.org/10.1145/3377811.3380342

[35] H. Babii, A. Janes, and R. Robbes, “Modeling vocabulary for big code machine learning,” *arXiv preprint arXiv:1904.01873*, 2019.

[36] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, “Compilation error repair: for the student programs, from the student programs,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, 2018, pp. 78–87.

[37] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, “Commit message generation for source code changes,” in *IJCAI*, 2019.

[38] S. Kim, J. Zhao, Y. Tian, and S. Chandra, “Code prediction by feeding trees to transformers,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 150–162.

[39] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “RoBERTa: A robustly optimized bert pretraining approach,” 2019.

[40] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, “Transformers: State-of-the-art natural language processing,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://doi.org/10.18653/v1/2020.emnlp-demos.6

[41] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *The annals of mathematical statistics*, pp. 50–60, 1947.

[42] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “GraphCodeBERT: Pre-training code representations with data flow,” 2020. [Online]. Available: https://arxiv.org/abs/2009.08366

[43] J. Li, Y. Wang, M. R. Lyu, and I. King, “Code completion with neural attention and pointer networks,” in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI’18. AAAI Press, 2018, p. 4159–25.

[44] F. Liu, G. Li, Y. Zhao, and Z. Jin, “Multi-task learning based pre-trained language model for code completion,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.

[45] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, “Context-sensitive code completion tool for better API usability,” in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 621–624.

[46] J. Moreno-León, G. Robles, and M. Román-González, “Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking,” *RED. Revista de Educación a Distancia*, no. 46, pp. 1–23, 2015.