On-the-Fly Syntax Highlighting using Neural Networks

Marco Edoardo Palma
University of Zurich
Switzerland
marcoepalma@ifi.uzh.ch

Pasquale Salza
University of Zurich
Switzerland
salza@ifi.uzh.ch

Harald C. Gall
University of Zurich
Switzerland
gall@ifi.uzh.ch

ABSTRACT

With the presence of online collaborative tools for software developers, source code is shared and consulted frequently, from code viewers to merge requests and code snippets. Typically, code highlighting quality in such scenarios is sacrificed in favor of system responsiveness. In these on-the-fly settings, performing a formal grammatical analysis of the source code is not only expensive, but also intractable for the many times the input is an invalid derivation of the language. Indeed, current popular highlighters heavily rely on a system of regular expressions, typically far from the specification of the language's lexer. Due to their complexity, regular expressions need to be periodically updated as more feedback is collected from the users and their design unwelcome the detection of more complex language formations. This paper delivers a deep learning-based approach suitable for on-the-fly grammatical code highlighting of correct and incorrect language derivations, such as code viewers and snippets. It focuses on alleviating the burden on the developers, who can reuse the language's parsing strategy to produce the desired highlighting specification. Moreover, this approach is compared to nowadays online syntax highlighting tools and formal methods in terms of accuracy and execution time, across different levels of grammatical coverage, for three mainstream programming languages. The results obtained show how the proposed approach can consistently achieve near-perfect accuracy in its predictions, thereby outperforming regular expression-based strategies.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → *Automated static analysis.*

KEYWORDS

Syntax highlighting, neural networks, deep learning, regular expressions

ACM Reference Format:
Marco Edoardo Palma, Pasquale Salza, and Harald C. Gall. 2022. On-the-Fly Syntax Highlighting using Neural Networks. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549109

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549109

1 INTRODUCTION

Today, software developers often turn to online web applications for support on several aspects concerning their source code manipulation tasks. Source code repository hosting services, e.g., GitLab, BitBucket, are typically concerned with managing version control instances, DevOps lifecycles, code reviews, continuous integration, and deployment pipelines. Some extend these functionalities by including issue tracking, knowledge bases, and chats, among other non-software-related features. Also, some Q&A platforms, e.g., StackOverflow, provide the possibility to query the community about code-related issues.

With the ability to boost productivity [32], code syntax highlighting (SH) is popular in online scenarios such as these described. Formally, SH is a form of secondary notation in which portions of the text are displayed in different colors, each representing some feature of the language. Due to the majority of features only being inferable from the grammatical structure of the input, the task of deciding what color should annotate what portion is non-trivial. Therefore, resolvers infer the color assignments from some internal grammatical representation of the code. Intuitively, the more this analysis restricts the belonging of a subsequence to some grammatical productions, the higher is the accuracy of its computation. As a result of the higher the number of such productions it can recognize, and therefore annotate, the higher is the strategy's coverage.

Unfortunately, there are two main challenges in performing such analysis in this context. First, there is a varying level of grammatical validity of the code highlighted. Due to online code being embedded in multiple contexts, its grammatical correctness cannot be guaranteed. Indeed, although in version control iterations source code might tend towards being of higher quality, in other cases, such as discussions in code review or chats, this might not carry a valid language derivation, i.e., an Abstract Syntax Tree (AST) might not be derivable [18, 31, 37–39]. This inherently induces SH strategies in being less reliant on the ability to derive a complete and well-formed representation of the code.

A brute-force (BF) approach towards performing accurate SH is to use the language's grammar for the derivation of ASTs, binding a color to each token, based on its location in the tree. However, not only is this often a computationally expensive strategy, but it also cannot be easily ported to effectively or deterministically recover errors in scenarios of severely incorrect or incomplete language derivations, e.g., code snippets [11, 12, 22]. Also, given the rich syntax of modern mainstream programming languages, parsing strategies better suited for dealing with noisy language derivations, e.g., *island parsing*, would expect developers of SH tools to produce viable encodings of the languages' original grammars [24, 25], while still requiring to execute a parsing routine.

As for the second challenge, only a small time delay is allowed for this frequent process to terminate, which BF approaches might

```
public class ClassDecl extends ClassAType implements InterAType<Type0, Type1> {
    public static final String var0 = "StringLiteral";
    private Type0<Type1<Type2, Type3>> member0 = new com.example.ClassBType<>();
    @Deprecated(since = "StringLiteral", forRemoval = true)
    public ClassEType<Class0Type> methodDelarationIdentifer() {
        ClassCType var1, var2 = new ClassDType<>();
        if (member1.methodCall(1) == var3 && !this.member2.methodCall())
            for (int index = member2.size() - 1; index >= 0; index--)
                methodCall(var3, index); // Inline comment.
        return null;
    } /** Multiline comment. **/
    private com.another.example.ClassEType<Class0Type> member3;
}
```

(a) Pygments, 64.76 % character accuracy.

```
public class ClassDecl extends ClassAType implements InterAType<Type0, Type1> {
    public static final String var0 = "StringLiteral";
    private Type0<Type1<Type2, Type3>> member0 = new com.example.ClassBType<>();
    @Deprecated(since = "StringLiteral", forRemoval = true)
    public ClassEType<Class0Type> methodDelarationIdentifer() {
        ClassCType var1, var2 = new ClassDType<>();
        if (member1.methodCall(1) == var3 && !this.member2.methodCall())
            for (int index = member2.size() - 1; index >= 0; index--)
                methodCall(var3, index); // Inline comment.
        return null;
    } /** Multiline comment. **/
    private com.another.example.ClassEType<Class0Type> member3;
}
```

(b) BRNN(16), 100.00 % character accuracy.

Figure 1: An example of task T4 of Java SH, using the state-of-practice and proposed approaches.

exceed. "On-the-fly" SH refers to code being highlighted as this is being retrieved by the user. The adherence to such computational schema results in SH resolvers having an indirect impact on user experience [16, 19]. For the above-mentioned reasons, state of practice SH strategies are mainly built around *(per-language) ad-hoc* lexers, which heavily rely on systems of regular expressions (regexes). Such design allows to achieve excellent computational performances whilst providing a SH capable of capturing some contained number of grammatical structures and accuracy levels. An example of the effects of this strategy is visible in Figure 1. Here, the SH produced by a popular regexes-based resolver (Figure 1a) is compared to one producing perfect highlighting for a grammatical coverage resembling those found on Integrated Development Environments (IDEs) (Figure 1b). The low coverage of the former is perceived by its inability to detect identifiers for types, method and variable declarations. In addition, it cannot distinguish severely distant grammatical constructions such as field accesses, method invocations, and reference types. In turn, this contributes to low annotation accuracy. Moreover, the specification of these lexers is often far from that of the language, inducing a tedious and error-prone regexes design process, with the generalizability of the final product relying on the manual compilation of test cases and multiple iterations of user feedback.

Therefore, motivated by these challenges and shortcomings, it is desirable to have an approach that is: (1) *simple to implement*, providing a deterministic, reusable, and low-effort process for developers to create and customize highlighters; (2) *able to reach high grammatical coverages*, enabling efficient highlighting of more complex grammatical structures than those computed in nowadays online highlighters; (3) *highly accurate*, closely reproducing the highlighting accuracy of a formal AST analysis process; (4) *input flexible*, reaching high accuracy on correct and incomplete/invalid derivations of the target programming language.

This paper proposes a solution that exploits lightweight Recurrent Neural Networks (RNNs) models to encode the highlighting behaviors of formal SH brute-force (BF) methods. A BF approach is user-defined and exploits the language's existing lexing and parsing tools to assign each token in the source code to a SH class, i.e., an abstraction of the SH color, based on its location in the AST. Therefore, it is a formal process, which, if well-formed to match the intended highlighting scheme, is always guaranteed to generate the correct SHs for files carrying a valid derivation of the language. After having used BF to compile highlighting assignments (SH) for multiple sample files, an RNN is trained to bind sequences of tokens to sequences of SH classes. The training process only occurs once and produces an RNN model that is reusable for all future SH tasks. For the training hardware used for the experiments in this work, all the proposed models can be trained in the order of minutes, and comfortably within the one hour mark. In the case of the non-bidirectional flavors, the delay is cut in half compared to their bidirectional counterpart. This delay substitutes to today's *state-of-practice* resolvers which involve the development of tedious systems of Regex. Once trained, the RNN computes the SH of source code by inferring SH classes to the token stream produced by the language's original lexer.

This novel approach to on-the-fly SH is tested with regards to its accuracy across four types of grammatical coverages, exploring the detection of different combinations of lexical features and various grammatical constructions for identifiers, declarations, and annotations. To support the suitability of the proposed approach in the deployment scenarios previously envisioned, this is also tested with regards to its execution time when predicting. Moreover, the same metrics are measured across three mainstream programming languages: Java, Kotlin, and Python. All the metrics are also computed for a highly popular SH tool, i.e., Pygments [8], based on the well-establish regex strategy used by a large number of online vendors such as GitLab, BitBucket, and Wikipedia.

To summarize, the main contributions of this paper are:

- a dataset for SH benchmarking for three popular programming languages, i.e., Java, Kotlin, and Python, obtained through formal BF strategies;
- the design of an Neural Network (NN)-based approach for SH, with near-perfect highlighting accuracy and suitable prediction delays;
- the comparison with the state of practice SH strategy in terms of accuracy, coverage, and execution time;
- the performance analysis of the approaches in case of incorrect/incomplete source codes.

The implementation, benchmark datasets, and results are available in the replication package [27] and published at the address https://hlnn.netlify.app.

The rest of this paper is structured as follows. Section 2, presents the design of the approach. Section 3 describes the experimental setup, whereas Section 4 shows and discusses the results. Section 5 surveys the related work, and Section 6 concludes with a summary of the findings and contributions, as well as an outlook on future research in this area.

2 APPROACH

The strategy designed to tackle the challenges raised in this paper aims at deriving Neural Networks (NNs) capable of statistically inferring the perfect behavior of brute-force (BF) models. For this purpose, an oracle of SH solutions is generated using the language's BF resolver. The following section presents in detail the specification of both BF and NN models, as well as providing some motivations for the design.

2.1 Oracles for Syntax Highlighting

Brute-force (BF) refers to the deterministic process of producing the correct token classification, or syntax highlighting (SH), for some language derivation from which an Abstract Syntax Tree (AST) is derivable. These are the sole components for the generation of the SH oracle and are created by reusing the language's existing lexing and parsing tools. The two components respectively represent the input source code as a token stream and order them into an AST. Subsequently, a tree walker exploits the structural information of the AST to assign each token to its SH class. This process assumes that a BF resolver is guaranteed to compute the correct SH of any valid input file, hence setting the highest achievable highlighting accuracy for any coverage specification. It is important to note that such a design merely requires the developer to implement a walker consisting of only a handful of detection rules, as reported in the replication package [27]. As a result, the process of producing a BF highlighter is deterministic and only asks for a basic understanding of the language's grammar, as it already exists. It is a significant departure from the tedious and error-prone workflow of defining systems or regular expressions.

The BF algorithm is integral in the generation of the *oracle*, i.e., a collection of language's source code files and respective SH. For this purpose, each sample file is piped through the language's lexer and then tokenized. From each token a new entity is derived in the form: $ETA = \{i_s, i_e, t, tr\}$ where the *Extended Token Annotation (ETA)* object is a tuple of: (1) i_s and i_e, denoting the token's character start and end indexes respectively, according to the file that contains it; (2) t, the exact text the token references; (3) *tr*, the token's *Token Rule*, encoded as a natural number, or in other words, the *ID* the language's lexer consistently assigns, through a dictionary, to tokens of the same type, among all types defined in the lexer (e.g., a token of text `{` might corresponds to a lexer type `OPEN_BRACE` hence to the token rule, or unique *ID*, `20`). For example, `String lang = "Java.";` might result in the set of ETA: `{0, 5, String, 102 }`, `{7, 10, lang, 102}`, `{12, 12, =, 73}`, `{14, 20, "Java.", 55}`, and `{21, 21, ;, 63}`. This representation allows the generalization of SH patterns based on the sequence of language features in the form of token types. It does it by abstracting away the otherwise "noise", injected by the tokens' specific text features, transparent to the parsing of the file.

Subsequently, the language's parser organizes the tokens into an AST. Walking the AST through patterns such as, *Visitor* or *Listener*, all previously computed ETAs are mapped to *Highlighted Extended Token Annotation (HETA)* objects. These extend ETAs to include a *Highlighting Class hc*, corresponding to the grammatical SH class to which the token being referenced is part of. Tokens that are not part of any grammatical construction are bounded to the unique *hc* `ANY`, representing text, i.e., no highlighting. As a result $HETA = \{i_s, i_e, t, tr, hc\}$. Continuing on the above-mentioned example, the following HETA set might be computed as: `{0, 5, String, 102, 1}`, `{7, 10, lang, 102, 2}`, `{12, 12, =, 73, 0}`, `{14, 20, "Java.", 55, 3}`, `{21, 21, ;, 63, 0}`, where *hc* of: `0` decodes to some not highlighted tokens, `1` to type identifiers, `2` to variable declaration identifiers, and `3` to string literals.

A BF resolver for some language L is a function of the form: $bl_L : \{c\}, le_L, l_L, p_L, ws_L \rightarrow \{HETA\}$, where: (1) for the *lexer encoder* $le_L : l_L, \{c\} \rightarrow \{ETA\}$, (1.1) l_L is the lexer of L, (1.2) $\{c\}$ is the character set of the input file, (1.3) $\{ETA\}$ the resulting set of ETAs. (2) for the *parser* $p_L : l_L, \{c\} \rightarrow AST_L$, AST_L is the derived AST of the input file, (3) and for the *walking strategy* $ws_L : AST_L, \{ETA\} \rightarrow \{HETA\}$, $\{HETA\}$ is the oracle for the input's file

2.2 RNNs for Syntax Highlighting

In order to efficiently perform SH for a given file, this approach seeks to obtain a Neural Network (NN) model capable of mapping a sequence of token rules $\{tr\}$ to a sequence of SH classes $\{hc\}$, as performed by some BF resolver. Hence, the process of computing SH becomes a statistical inference on the expected grammatical structure of the token sequence in input.

The motivation behind the use of NNs for such a task relies on the highly structured nature of programming languages' files. Indeed, the flow of the incoming characters is: (1) represented as an entity stream selected from a finite set of terminal symbols $\{tr\}$, and (2) ordered by an underlying pure ordering function as a formal grammar. SH can be viewed as the grammar for which there always exists a correct language derivation whenever there exists a valid derivation of the original grammar. This is true as for some grammar g, its highlighter is the grammar hg that sequentially parses sub-productions s_{hg} of g, which are enough to discriminate a *tr* subsequence to some target highlighting construction; or otherwise, map every token not consumable by any s_{hg} to a terminal symbol. In this novel approach to SH, the effort of producing such SH grammar is lifted from the shoulders of the developers and instead delegated to the NN which infers it from the behavior observed from some BF. The task of SH is reduced to a "sequence-to-sequence" translation task [36], i.e., from $\{tr\}$ to $\{hc\}$.

To tackle this new problem reduction, the following proposes the use of Recurrent Neural Networks (RNNs) [9], for the learning of SH sequence bindings. These offer a base approach to sequence translation by iterating through each value of the input sequence while outputting a unit of translation and carrying forward differentially optimized information to aid the prediction of future inputs. Furthermore, for those grammars producing sequence distributions for which the binding of an *hc* for some *tr* may require the look ahead of an arbitrary number of tokens, this approach resorts to the use of Bidirectional Recurrent Neural Networks (BRNNs) [33] in place of traditional RNNs. Indeed, these also aim at addressing this specific issue by behaving as traditional RNNs, however inferring the translation of each input from the extra information carried from navigating the input sequence in reverse. Finally, the model is designed to output for each *tr*, a categorical probability distribution over the set of available *hc*. The absolute values of such distributions are normalized by a *softmax* function, resulting in the sequence of *hc* for some sequence of *tr* being the set of *max* values of the

distribution computed for each *tr*. Consequently, with regards to SH, an RNN model M is a function of the form: $M : \{tr\} \rightarrow \{hc\}$.

Although base RNNs are no longer the state of the art in many translation applications, with current solutions mostly utilizing convolutional layers, the encoder-decoder architectures, or relying on the attention mechanism [4, 5, 14, 21, 36, 42], these still offer a lightweight model compared to more recent techniques. Moreover, as it is later shown, the number of well-formed structural features NN are expected to infer from the SH oracle samples is small. It means that the extra infrastructure of deeper networks would result in no appreciable SH accuracy increases, but rather in computational overheads and non-trivial hyperparameter/training configurations. Instead, RNNs and BRNNs provide a baseline solution for this novel challenge, delivering predictions with contained overheads. In addition, the training behavior of such models allows this approach to maintain a constant training configuration. Not only does it result in stable performances across different languages and coverage settings, but also in a solution that is accessible to a broader audience of developers [3].

3 EXPERIMENTS

The effectiveness of this proposed approach is evaluated in terms of its prediction accuracy and speed for four types of SH coverage. Moreover, in the interest of providing a clearer view on how the performances of this approach might generalize, all experiments were conducted on three mainstream programming languages: Java, Kotlin, and Python. To represent the state of practice approach using regexes, the Pygments SH library [8] is also evaluated against the same metrics. Pygments is highly popular in online and offline scenarios and found in an array of tools such as GitLab, BitBucket, and Wikipedia. The following research questions are considered for the formal analysis of the solution:

RQ1 *How accurately can the proposed NN approach replicate the SH behaviour of a BF model?*

This question aims at evaluating, in terms of SH accuracy, for all the defined coverage levels, to what extent the proposed approach can be a substitute to pure brute-force methods.

RQ2 *How does the proposed NN approach compare to nowadays state of practice, or regex, approaches?*

This question needs the computation of the SH accuracy, for all the defined coverage levels, to understand to what extent the proposed approach can be a substitute to the state of practice.

RQ3 *How do the speed of computation of the three approaches, NN, BF , and regex compare?*

It provides insights into the time delays required when performing SH with the proposed NNs, regex-based, and BF approaches.

RQ4 *How accurately can the proposed NN approach perform SH of incomplete language derivations, compared to the regex and BF approaches?*

An advantage of both the proposed and regex-based approaches is their natural portability to estimate SH schemes for incorrect/incomplete sequences of tokens. Hence, this question evaluates, in terms of accuracy, for all the defined coverage levels, how these approaches compare to the theoretical perfect SH solution.

3.1 Coverage Tasks Definition

Although an infinite number of coverage schemes could be generated and tested for, the initial iteration of this novel approach to SH investigates the highlighting of language features as done in the most common IDEs for the selected languages, such as IntelliJ IDEA, PyCharm, and Visual Studio Code.

Each *Coverage Task* (T) is therefore created by combining one or many of the following language feature groups. Each feature represents a unique *hc* (*Highlighting Class*, see Section 2.1), or in visual terms, a color.

`Lexical`: this group includes token classes that are lexically identifiable, meaning that for a given token, nothing but its *tr* value is required to bind it or not to any of such classes:

- `KEYWORD`, thereby only referring to strong keywords, as soft keywords may also be used as user-defined identifiers in some allowed language contexts. In this class, also tokens of primitive types, e.g., `int`, `float`, are included if the language identifies them as such;
- `LITERAL`, any literal value of the language, e.g., numbers (integers, floating, binary, hexadecimals), boolean values (`true`, `false`), null constants (`null`, `None`);
- `CHAR_STRING_LITERAL`, any user-defined string or character literals, including those part of string interpolation sequences;
- `COMMENT`.

For this group, all classes are assigned using the same criterion that is applied to all the selected programming languages.

`Identifier`: the group includes classes for special types of identifiers:

- `TYPE_IDENTIFIER`, matching all the identifier tokens within all the languages' productions representing a type entity;
- `FUNCTION_IDENTIFIER`, all the identifiers used in function or methods calls;
- `FIELD_IDENTIFIER`, referring to those identifiers that the grammars understand being references to an attribute of an object or entity. These are usually preceded by a entity navigation operator, e.g., in Java's `Object o = a.b.c().d;`, `b` and `d` are such `FIELD_IDENTIFIER`, whereas `c` might be considered a `FUNCTION_IDENTIFIER`.

`Declarator`: it includes classes for the classification of token identifiers that carry the name of new top-level features of programs:

- `CLASS_DECLARATOR`, referencing identifiers bounded to some newly defined declaration of any form of class, objects, enumerations, data classes, structures, etc.;
- `FUNCTION_DECLARATOR`, for identifiers bounded to some newly defined method or function;
- `VARIABLE_DECLARATOR`, to some newly defined variable. Note the exclusion of this class from Python experiments due to its intrinsic ambiguity of value to identifier assignments.

`Annotation`: this includes the base annotation components:

- `ANNOTATION_DECLARATOR`, as it is common practice to markup annotations in all three selected languages, this class references the token identifiers and prefixed symbols such as the `@`, of an annotation.

Finally, *hc* `ANY` gathers all tokens not belonging to any of the categories mentioned above.

Table 1: Metrics for Java, Kotlin, and Python normalized SH oracles

Metric	Java					Kotlin					Python				
	Mean	SD	Min	Median	Max	Mean	SD	Min	Median	Max	Mean	SD	Min	Median	Max
Chars	6239	11575	0	2932	504059	2455	4385	80	1490	176176	7390	34324	0	3398	3987090
Whitespaces	1207	2417	0	529	72702	575	1276	6	282	47495	1999	12941	0	829	1465856
Lines	190	332	0	94	14628	70	121	1	43	4734	208	873	0	104	89373
Tokens	882	1745	1	371	45229	737	1559	23	327	72484	1161	4997	1	525	448562

From the *hc* groups, four coverage tasks are defined to evaluate the flexibility of the RNN approach to comply with some arbitrary SH coverage. The four *Coverage Tasks* are defined to demand the identification of the following groups:

- T1: {`ANY`}, `Lexical`, and `Declarator`;
- T2: {`ANY`}, `Lexical`, and `Identifier`;
- T3: {`ANY`}, `Lexical`, `Declarator`, and `Identifier`;
- T4: {`ANY`}, `Lexical`, `Declarator`, `Identifier`, and `Annotation`.

It is important to note that, for the reported tasks configuration, given the oracle O_{T4} carrying all of the language classification groups, the oracle of any other class $O_{T[1..3]}$ can be derived directly from O_{T4} through means of a *Task Adapter* $TA_{T4,\,T[1..3]}$. For any task $O_{T_I \mid i \in \{1..3\}}$ a TA_{T4,T_I} maps every target class *hc* to itself if it is a possible target class for T_I, otherwise to the *hc* class `ANY` (text).

More details about the above language groups, and their detection strategy for all three investigated languages, are available in the replication package [27].

3.2 Data Collection and Preprocessing

The following describes the procedure produce the datasets used in the experiments. The full details, together with the downloadable data, are available in the related replication package [27].

Data mining. In order to generate SH oracles for testing the approaches with regards to their accuracy, speed of evaluation, and training of the RNN models, samples for the three programming languages selected are mined from GitHub's public repositories, through GitHub's Application Program Interface (API). In this process, the repositories are pulled by filtering per programming language and sorting by descending order of stars rating. For every main branch, files matching the language's file extension are downloaded in their natural order.

With the ultimate goal of converting each file to its equivalent set of HETA, the data collection process filters only files for which the BF strategy can derive an AST. Of all files, only one instance of the same token rule (*tr*) sequence is kept: this prevents giving an advantage to the RNN approach, which works at a *tr* sequence level instead of at a character level. Indeed, two program files might carry different text but share the same structure; notice how these two Python code are structurally equal: `a = b.c[3].d()` and `u_field = user.files[0].normalised()`.

For each programming language, the data collection pipeline runs until it has sampled 20 000 files. This sample size is in the interest of creating oracles that are both of large statistically meaning, for the average file contents of each language, but could also allow for the execution of extensive accuracy and performance testing. Statistics on the number of characters, whitespace, lines of code, and tokens, of the datasets collected for each language are summarized in Table 1.

Brute-Force and Oracle Generation. To create an oracle for each language, given a set of valid input files, a BF method must be created. As one of the goals of this proposed approach is to reutilize the existing lexing and parsing strategies, the ANTLR4 [29] parser generator tool is used, pooling the respective official ANTLR4 lexer and parser grammars of each language. Using ANTLR4 proved to be a winning solution to kick-start the creation of all three oracles. Not only is it a widely popular parser generator, but also used by official language specifications, such as Kotlin, and benefits of an active community developing grammars for most of the mainstream programming languages. However, it is essential to note that the operability of this approach does not strictly rely on this particular tool, as any preprocessing program could be used if mildly adapted to output the required and largely generic oracle information.

The obtained lexers and parsers, of which version details are available in the replication package [27], are kept largely unchanged. The most significant changes interest the lexers, which were instructed to push the skipped tokens, e.g., comments, through the lexers' hidden channel. Such a (minor) modification enables the approach to obtain tokens for these otherwise dropped entities, which might still require highlighting, as reported in Section 2. Should this workflow not be available in a language's parsing implementation, or should its introduction cripple the structure of the parser, tokens can be lexed by a dedicated lexer.

In addition to the pipeline for obtaining the ETAs set and AST for a given file, a tree walker is created, which aids the conversion of each ETA into its grammatically highlighted HETA derivative. Although multiple walking strategies are available, for the highlighting of the grammatical features considered in this first iteration of this novel RNN approach to SH, this can most easily be achieved through the "listener pattern." It limits the process to providing highlighting logic for the productions that are expected to contain tokens belonging to any of the target SH classes. All other tokens are instead implicitly mapped to the `ANY` class. As the reporting of the fine details of such implementations would lead to a large and mainly uninteresting listing of tree analysis rules, this can instead be consulted in the replication package [27].

For each language, the BF methods are created for the coverage specified by task T4. This leads to the generation of an oracle carrying highlighting targets for each SH classes present in any given source files. The *Task Adapter* method described earlier is therefore used to derive the oracles for the other sub coverages of task T1, T2 and T3. This method not only has no effect on the correctness of the derived oracles but it also avoids the definition

of a new tree walker and respective time and space expansion for computing further oracles.

Data organization. As the proposed RNN approach involves the training of NNs, it is important to report on what strategies are put in place to ensure that not only the generalizability of the solution is verified but that there also exists an unbiased setting when its accuracy is compared with the other approaches.

For these reasons, the oracles are randomly shuffled and then split into three folds. Folds ensure that 33 % of the oracle's samples are used for testing only, whereas of the remaining 66 %, 90 % is reserved for training and 10 % for validation. These three sets never intersect, according to the data collection strategy employed. Moreover, all folds used in the experiments are constant and persisted. This helps ensure reproducibility and allows each RNN model to be compared when trained on equal datasets.

Incomplete files generation. Although both the proposed RNN approach and the state of practice, based on regexes, are capable of computing SH for incorrect program files, their accuracy in these cases cannot be checked exactly, as deterministic oracles for such files are not always derivable. For this reason, the focus is shifted from mining for incorrect file derivations towards generating invalid language derivations from the set of valid sampled files.

In order to compare the accuracy of the SH computed by the proposed and regex-based strategies, when fed files carrying incomplete (hence invalid) language derivations against the target SH computed by a pure process with access to required extra file structure, the files in each test fold are sampled line-wise to generate one code snippet sized files. These are drawn from the test datasets, as in this first iteration of this approach, the network is not trained on these incomplete files but only tested; however, sampling for training datasets of the folds might have given an unfair advantage to the RNN approach.

At this stage, it is also important to note that it would not have been tractable to sample such snippet-sized file from distributions of natural snippets generating processes, as the BF method would not have been available for the formal computation of the correct SH, for the reasons highlighted above. Therefore, with the target number of newly generated files of 5000 from each fold test set, thus 15000 per language, each test file is drawn randomly, and from it, a random sub-sequence of lines is chosen.

The lengths of the snippets are drawn normally according to the language's mean, standard deviation, minimum and maximum number of snippets lines, determined by number of lines found by querying the StackExchange Data Explorer [35], focusing on snippets from StackOverflow. In particular, at the time of the experiments, these numbers were (mean, standard deviation, minimum, maximum): 17.00, 28.75, 1, and 1117 for Java; 15.00, 22.05, 1, and 703 for Kotlin; 14.00, 20.39, 1, and 1341 for Python.

Both test files and lines are sampled with replacement. Given the lines selected, the process gathers the set of HETAs in range and produces a new oracle instance.

3.3 Compared Approaches

Multiple variations of baseline RNNs models are investigated.

An initial configuration for the RNNs and training was derived by improving the convergence of the networks on the validation set of only the first fold of the Java dataset. The initial embedding layer was kept at 128, i.e., the smallest power of two larger than the number of token ids for the languages, while the hidden units were added in increasing power of two. With a constant learning rate of 10^{-3} and Adam optimizer, 16 and 32 (B)RNNs were found to produce near-perfect accuracy, with the latter not improving in wider models. Accuracy converged after the second epoch. A final investigation involved the common practice of reducing the learning rate after convergence by a factor of 10, i.e., 10^{-4}. It further helped improve the accuracy of the model, which again was observed to converge within the following two epochs.

As a result, the RNN models evaluated consist of a fixed 128 embedding layer. The output of the embedding layer is mapped to a single layer RNNs or BRNNs, of widths evaluated among 16 and 32 hidden units. The output of all the RNNs or BRNNs is passed through a fully connected linear layer reducing it to a categorical distribution of the available *hc*, depending on the *Coverage Task*. This results in the testing of four models, identified by its directionality and width of RNN layer: RNN(16), RNN(32), BRNN(16), and BRNN(32). Every model is trained sequentially on each training sample, with cross-entropy loss and Adam optimizer. The training session for any SH RNN, language and coverage, was accordingly set to train for two epochs with a learning rate of 10^{-3}, and for a subsequent two epochs with a learning rate of 10^{-4}. It is in respect of the approach's initial guarantee of delivering a training configuration capable of achieving the performance advertised without the tweaking of the training session by expert developers. All models commence the training process from a randomly initialized state, according to the deep learning framework utilized, i.e., PyTorch [30], while a constant seed ensures the reproducibility of the experiments.

To contextualize the performances produced by the RNNs approaches, the vastly popular and well-established regex-based syntax highlighter Pygments is tested [8] using its latest available version at the time of testing *2.10.0* . In the following, Pygments is being referred to as Regex. Its output was manually adjusted to output the same classes included in T4, of which details can be found in the replication package [27]. Hence, the same *Task Adapter* used during the conversion of the oracle to any other task is used to map each Pygments' prediction to its task-specific class.

Finally, the same BF methods used for the generation of the oracles are reused for the outlined comparisons with the RNNs and regex-based approaches. The use of ANTLR4 is not only induced by the large availability of language grammar, but also by its highly efficient *LL(*)* parsing [28] strategy, and native error recovery logic, both of which undermine the real-world performance advantage of, otherwise theoretically regarded as most efficient, Parsing Expression Grammar (PEG) parsers [13] in both fronts [6, 17, 22].

3.4 Evaluation Metrics

The quality of an SH can be measured with regards to its coverage, accuracy, and speed, described in the following.

Coverage. The absolute number of unique grammar constructions the highlighter is able to recognize.

Accuracy. Given a coverage specification, the degree to which the highlighter can bind each character in the input text to its correct SH class. It also resolves the issue of BF and Regex strategies possibly producing different tokenizations of the same file.

Speed. The time delay for the computing of SH. Prediction speed for all methods evaluated during the experimentations is measured as the absolute time in nanoseconds required to predict the SH of an input file once this has been supplied. For each SH method, the following time delays are measured:

- BF: the time to natively parse the input file and perform a SH walk of the obtained AST;
- Regex: the time to compute the output vector of SH classes, once given the file's source text, but excluding the time required to format the output to any specification. The latter is achieved by defining a new Pygments *Formatter* object which accepts the computed SH, but does not invest computational time into outputting it, hence removing the added time complexity any specific format might introduce, thereby highlighting the complexity of the approach's underlying SH strategy;
- RNNs: the time for the ANTLR4 inherited lexer (the same used by the BF approach) to tokenize the input file into a sequence of token rules, plus the time for the RNN model to create the input tensor, and predict the complete output vector of SH classes.

3.5 Execution Setup

All RNN models are trained on a machine equipped with an AMD EPYC 7702 64-Core CPU clocked at 2.00 GHz, 64 GB of RAM, and a single Nvidia Tesla T4 GPU with 16 GB of memory. Instead, all performance testing for all of the compared approaches was carried out on the same machine with an 8-Core Intel Broadwell CPU clocked at 2.00 GHz with 62 GB of RAM.

3.6 Threats to Validity

With regards to the problem statement raised in this paper, i.e., on-the-fly SH, ANTLR4 undoubtedly represents the package of technologies and strategies required not only for the definition of BF models but also their evaluation. Despite the best intention to consider all viable options, one should not exclude the existence of, perhaps language-specific, parsing tools that might scale the performance of BF resolvers.

The impossibility to generate testing oracles from snippets produced by online user processes, resulted in a first experiment setup which synthetically generates incomplete/incorrect language derivations from the set of parsable derivations. Therefore, it is crucial to note that RQ4 only intends to provide an initial perspective on how the three approaches might perform on file segments, and at that the formal measure of closeness between this synthetic process to that observable in online code snippets is unknown. Moreover, human annotator processes are likely to employ their statistical inference about the missing context of some code fragment. Hence, one may argue that conducting such an assessment with a manually composed, and therefore inconsistent, dataset would instead validate a model's ability to meet the level of *program-comprehension* of the sample of users that created the dataset. Instead, the synthetic dataset created here indirectly validates the model's ability to infer the statistically most likely missing context.

Pygments provides syntax highlighting for 534 languages. However, it is a collection of implementations of language-specific Regex SH, and not a single generic SH resolver. This work compares with three of such highlighters, i.e., Java, Kotlin, and Python, but promises to be applicable to other languages, as language-specific BF can be used to train new language models. The validation of the proposed approach across all the languages supported by the Regex-based counterpart would extensively assess the generalizability of the strategy. Therefore, this aspect is considered a limitation of the experimental setup, which does not prove the absolute generic performances of this novel strategy but instead delivers seminal evidence of its applicability.

Benchmarks for prediction delays might only give a general perspective of the performances of such tools, but exclude specific implementation optimizations that developers might design. It may also include file size limits for online consumption, which might be platform dependent. Other variables might concern the efficiency of the integration of SH resolvers with the rest of the service, caching strategies, or hardware specifications. For example, the proposed RNN solution might perform differently if run on more production-focused deep learning libraries [1], or on GPUs.

4 RESULTS

Developing from the experiment setups described in Section 3, this section individually addresses the performance of the proposed approach with regards to the four research questions identified. For each question, its specific validation workflow is described, and the results are presented and discussed.

To compare the observations, the "Kruskal-Wallis H" test [23] was applied with the "Vargha-Delaney $\hat{A}_{12}$" test [40], for the effect size to characterize the magnitude of such differences. For this reason, the following reports the evaluation metrics in terms of median values, being these tests based on the median differences.

4.1 RQ1 – Comparison with BF's Accuracy

RQ1 aims at evaluating the SH accuracy of the proposed approach when compared to the theoretical perfect BF resolver, on language derivation for which an AST is derivable. Such aspect is validated regarding all three programming languages, as well as to the four *Coverage Tasks*. Every candidate RNN model is first individually trained on the training set of each fold, and its accuracy is recorded about its predictions on the corresponding test set.

As reported in Table 2, for all the languages and coverage tasks selected in this experiment, the proposed approach is capable of producing near-perfect SH solutions. The bidirectional variants prove to be the most eclectic model, which, even in the narrowest tested configuration (RNN(16)), achieve a perfect score more consistently than any base RNN model, across all languages and tasks. It is as expected, with bidirectionally extending the context around each token. Hence, it enables the resolution of ambiguous syntactical structures of which type is dependent on the next tokens.

Furthermore, the BRNN variant promotes a significant improvement in the stability of this strategy, with the accuracy distribution more concentrated around the perfect mark and the outliers being not only fewer but also of generally higher accuracy than otherwise obtainable with base RNNs. This is clearly visible in Figure 2.

Table 2: Median values over 3 folds for the accuracy. The maximum scores per task are highlighted

Model	Java				Kotlin				Python			
	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4
Regex	0.8662	0.7606	0.7233	0.7230	0.8009	0.6998	0.6787	0.6781	0.9364	0.8189	0.8189	0.8165
RNN(16)	0.9987	0.9716	0.9676	0.9668	1.0000	0.9627	0.9598	0.9605	1.0000	0.9560	0.9559	0.9550
RNN(32)	1.0000	0.9751	0.9710	0.9706	1.0000	0.9648	0.9640	0.9631	1.0000	0.9572	0.9571	0.9570
BRNN(16)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
BRNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

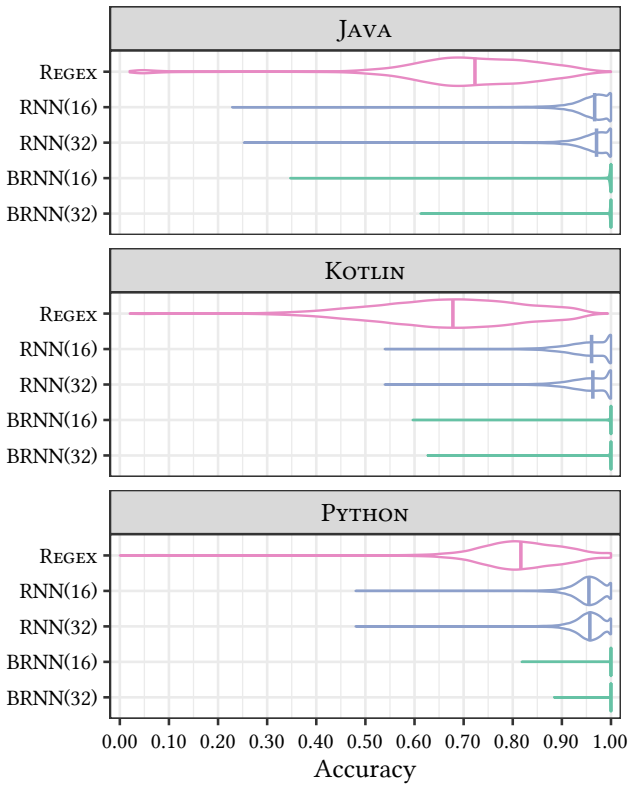

Figure 2: Accuracy values comparison for T4.

Therefore, for the average case, the proposed RNN strategy to SH is most often able to perform as well as the pure BF strategy. Nevertheless, being this a nondeterministic approach, some contained levels of inconsistency should be expected.

4.2 RQ2 – Comparison with Regex's Accuracy

Addressing RQ2 allows for the contextualization of the accuracy values obtainable by the proposed strategy, with what is achievable with today's state of practice, i.e., regexes. Such a research question is therefore tackled by evaluating the SH accuracy of Pygments on the same test datasets used to estimate the generalizing accuracy of the RNN models in RQ1.

As supported by the evidence displayed in Table 2, which reports the median accuracy values per SH method, the regex-based strategy consistently performs the worst across all tested scenarios. It is also essential to notice how the Regex approach is significantly more prone to variability in its level of accuracy, compared to any of the RNN models tested, as visualized in Figure 2. Pygments yields its best performance across all languages when its output is evaluated about coverage task T1.

Another observation concerns Pygments's accuracy decaying significantly for all tasks other than T1. Compared to the other tasks, T1 requires the identification of only lexical features and declarator identifiers. However, unlike declarations, lexical components are always deterministically identifiable through lexing, except soft keywords. T1 is, therefore, the least complex task out of all of those tested as, per file, only a handful of declaration identifiers are found, requiring the resolvers to identify mainly lexical features. Hence, the accuracy of Regex resolver converges considerably for tasks T2, T3 and T4, as all other grammatical features are reasonably consistently bounded to incorrect *hc* values.

Overall, the evidence collected for RQ2 supports the fact that the proposed approach is capable of quite consistently boosting the SH accuracy otherwise achievable with the state of practice.

4.3 RQ3 – Speed Comparison

The investigation into the prediction speed of all the available approaches aids in contextualizing at what responsiveness costs the proposed approach to SH can deliver its coverage and accuracy performances. Thus, each resolver is set to produce SH for each language's oracle 30 times, and their prediction delays are recorded. The experiments are carried out on the same machine, and no GPU is used for the evaluation of the execution of NN based resolvers. From the results obtained and summarised in Table 3, several observations can be made.

The RNN based approaches provide significant speed-ups over the BF resolvers. In fact, in the case of Java prediction delays are 25 times smaller for RNNs of both 16 and 32 hidden units; and 13 times smaller in the case of the bidirectional variants. Moreover, the standard deviation of the prediction delays of the proposed solution is also significantly smaller than the BF counterparts. Both RNN models reduce this metric by a factor of 38 and the BRNN models by a factory of 25. Kotlin leads to similar conclusions, although with the BF solution yielding better performances, but still worst compared to the proposed solution. In particular, the gains in favor of the RNN models, which do remain consistent with the delays recorded in Java, decrease to an average speed-up of 4 for the RNN models, and 2 for the BRNN models. Standard deviation is also down by a factor of 3 and 2 for the RNN and BRNN respectively.

Table 3: Descriptive statistics of execution time (ms)

Model	Java					Kotlin					Python				
	Mean	SD	Min	Median	Max	Mean	SD	Min	Median	Max	Mean	SD	Min	Median	Max
BF	225.684	894.046	0.004	45.903	49618.222	30.950	87.893	0.011	8.080	14119.526	52.798	242.363	0.033	24.022	23628.056
Regex	0.015	0.040	0.004	0.011	22.975	0.010	0.047	0.004	0.009	27.468	0.016	0.030	0.003	0.013	7.048
RNN(16)	9.195	18.704	0.206	3.877	689.178	8.383	31.805	0.370	3.612	12755.019	66.313	288.904	0.182	32.597	27164.357
RNN(32)	9.202	18.581	0.195	3.887	677.833	8.439	30.231	0.384	3.666	12067.893	63.522	276.867	0.176	31.682	26279.598
BRNN(16)	17.506	36.176	0.270	7.241	1269.607	14.997	40.814	0.586	6.537	12120.509	75.235	333.959	0.217	35.742	32334.076
BRNN(32)	17.728	36.565	0.278	7.396	1341.984	15.664	42.090	0.605	6.829	12243.090	76.895	344.068	0.219	36.301	32475.535

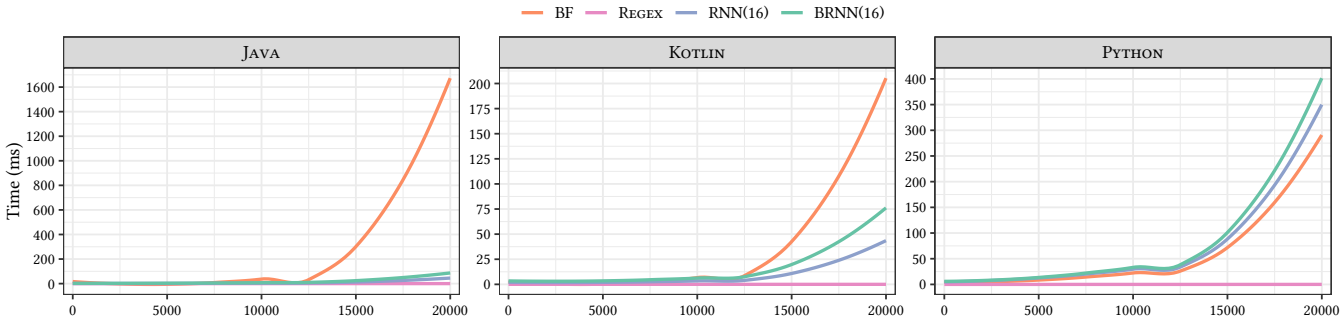

Figure 3: Execution time (ms) values trends comparison for T4.

Nevertheless, such a narrative changes when comparing the performance of the NN approach to the BF resolver for Python. According to Table 3, the proposed RNN approach is not superior to the BF approach. In fact, the parsing proves to be significantly more efficient than it is in the cases of Java and Kotlin. With the technologies constant for all BF resolvers, this suggests the grammar of the Python language is the main promoter for the efficiency gains observed. Nonetheless, the proposed approach proves capable of nearing such stellar performance of the BF resolver, however with some contained slowdowns: 1.3 and 1.4 on average for the RNN and BRNN models respectively. Standard deviation is also mildly down by 1.2 and 1.4 for the RNN and BRNN models.

As expected, the computational overheads of the proposed NN approach are more significant compared to the ones that accompany Regex. However, with the RNN strategy focused on delivering greater SH accuracy and coverage, and a significantly smaller development effort for developers, the focus is shifted on the suitability of this approach to the task. Considering the average delays recorded during this experiment, these are found to be relatively small. For the RNN approaches predictions are on average delivered in 9*ms*, 8*ms*, and 66*ms*, for Java, Kotlin and Python respectively; and the medians 4*ms*, 4*ms* and 33*ms*. Such computational delays would most comfortably belong with the *Seow*'s response-time categorization of *instantaneous* [34]. In this category includes human and computer interactions that are expected to complete within 100 ms and 200 ms, e.g., clicking and typing; whilst longer delays, within 500 ms and 1000 ms, being categorized as immediate, this last one including navigation actions [10, 34].

Figure 3 shows a smoothed line plot to represent the execution times for all the experiments. As it shows, the proposed approach is capable of delivering SH results well within the average human deadlines, with these requiring delays to be within 2 s to 5 s to maintain flow [10, 34], and tolerating a webpage response of 2 s [26].

4.4 RQ4 – Incomplete Derivations Highlighting

RQ4 considers SH accuracy of the highlighters with incomplete/incorrect language derivations. Likewise, for RQ1 and RQ2, all approaches are set to produce highlighting for all three languages and four coverage tasks. The dataset used for this RQ4 is the generated snippet dataset, for which perfect target solutions are known.

As it possible to notice by comparing Table 4, related to RQ4, with Table 2, related to RQ1, the results show how the RNN-based approaches are capable of maintaining accuracy performances on par with those obtainable on language derivations for which an AST is derivable. In fact, also in this scenario the RNN models compute SH with an accuracy within 94 % to 96 %, and the bidirectional variants always reaching a perfect median accuracy value. The state of practice, i.e., Regex, registers a decrease in accuracy, which, similarly to RQ2, is considerably far from those obtainable with the proposed NN models. It is especially noticeable for tasks with larger grammatical coverage, such as T4.

Figure 4 informs best about not only how consistently poorer the results of the Regex approach are compared to those of the RNNs and BRNNs, but also how much more variable they can be expected to be. Instead, the BF resolvers proved to be the least eclectic strategy. For Java, median performance values are close to those obtainable with the Regex resolver, however, at the cost of much greater variability than the latter. BF strategy performs the worst with Kotlin, yielding 0 median accuracy value and yet again a significant accuracy variance. Finally, in the case of Python, the BF approach is capable of outperforming both Regex and the

Table 4: Median values over 3 folds for the accuracy for snippets. The maximum scores per task are highlighted

Model	Java				Kotlin				Python			
	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4
BF	0.9211	0.7421	0.6586	0.6440	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000
Regex	0.8700	0.6859	0.6346	0.6340	0.8117	0.6577	0.6285	0.6279	0.9338	0.7890	0.7890	0.7860
RNN(16)	1.0000	0.9582	0.9512	0.9506	1.0000	0.9503	0.9469	0.9467	1.0000	0.9605	0.9595	0.9587
RNN(32)	1.0000	0.9634	0.9557	0.9555	1.0000	0.9534	0.9513	0.9512	1.0000	0.9618	0.9614	0.9617
BRNN(16)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
BRNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

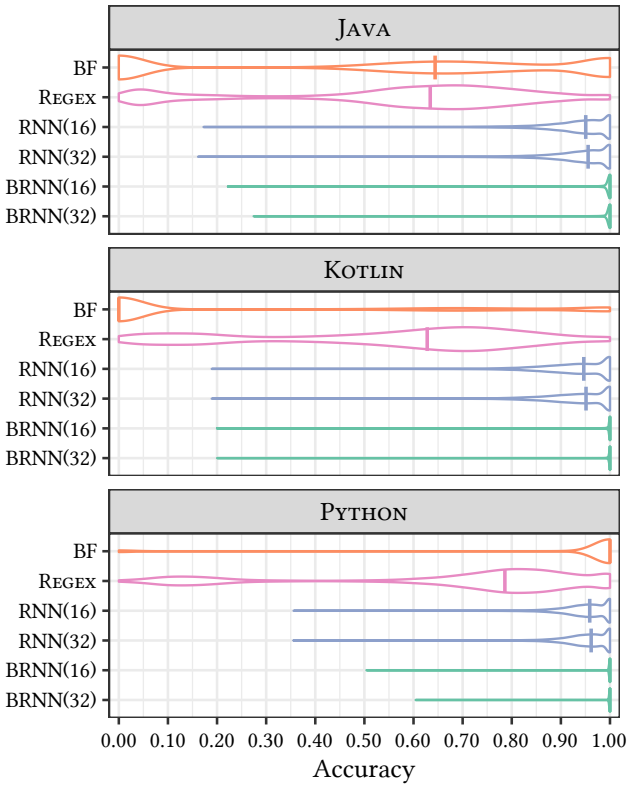

Figure 4: Accuracy values comparison for incomplete language derivations.

base RNN approaches, nearing the predictions of the BRNN models. However, the latter presents a mildly smaller number of outliers.

5 RELATED WORK

The main goal of this proposed approach is to show that deep learning can be used to perform syntax highlighting effectively and efficiently. In the following, the current state-of-the-art approaches that most relate to the proposed approach are listed.

Deep learning type inference. Similar applications of deep learning (DL) models have been utilized in the field of *Type Inference*; an example of this is DeepTyper [15]. In this case, motivated by the maintainability and readability benefits of a statically typed codebase, the model aims at aiding developers in the transition of code of dynamically typed languages supporting type annotation to their annotated equivalent. Similar to how the proposed approach to SH learns to infer the behavior of a parser on token ID sequences, DeepTyper aims at statistically inferring the compiler's type inference process. Such capability becomes especially useful in languages such as JavaScript, which cannot deterministically handle *duck-typing* even during runtime. The architecture used in DeepTyper is also based on BRNNs, however including extra infrastructure for the handling of more complex predictions. In fact, this consists of bidirectional Gated Recurrent Unit (GRU) [9], with 2 hidden layers of 650 hidden units each. To proxy between the two hidden layers, an extra layer is introduced: the *Consistency Layer*. This pushes forward an extra input for the second BRNN layer, in the form of the average token representation (embeddings) of the first BRNN layer, thereby promoting the model to use long-range values in the input. Furthermore, the model maps its input vector through an embedding layer of size 300. Finally, DeepTyper maps the values of its output layer through a *softmax* function to obtain for each input token a categorical probability distribution over the types in some vocabulary. The oracle is also generated analytically, with TypeScript files first annotated by the compiler and then stripped of their type annotations to obtain JavaScript files.

Unlike the approach proposed in this paper, DeepTyper uses tokens as inputs, complete of identifiers: this also allows it to compute type names. However, this extra information is not needed in the SH scope, in which structure is directly dependent on the sequence of token rule or type, i.e., *tr*. The adaptation of the DeepTyper model to the task of SH, although obviously possible, is vain due to the evidence being reported. Base BRNN models are never saturated in their ability to reach perfect SH accuracy. It means the extra infrastructure of a DeepTyper model would likely not generate better results but would lead to larger and slower models.

Learning lenient parsing and typing via indirect supervision. TypeFix is a transformer [41] decoder network developed as part of an approach to leniently parse and type Java code fragments [2]. It develops from the architecture and task of DeepTyper, and derives a deeper model based on a 6 layer decoder network, with each layer having multi-head attention and feed-forward. By design, such flavors of encoder-decoder models promote the output of each inner layer to be a function of all combinations of units in the previous layer. It promotes the learning of generalizable reductions of the relationships among elements in the input sequence. More levels of relationships between the inputs may also be learned through *multi-headed attention*, by adding more attention layers to the model.

Moreover, this mechanism allows the model to be more easily trained on long sequence, unlike RNN models, which, due to their recurrent evaluation of an input vector, suffer from vanishing gradients [7]. Similarly to the proposed strategy to SH, and DeepTyper, TypeFix is trained over a synthetically derived oracle. In particular, this consists of bindings of Java token identifiers and the respective deterministically derived type. Hence, the model is trained to bind a categorical probability distribution over some fixed type vocabulary.

The reasons for which such architecture is not being evaluated in this first iteration towards on-the-fly SH, are in line with those given for DeepTyper.

Generating robust parsers using island grammars. Island grammars [24] are grammars which define both *island* and *water* productions. *island* rules define how to consume specific subsequences of some input sequence. Instead, *water* rules define how to consume all of those tokens that could not be bounded to any *island* rule. Such grammar structure might be used for the task of SH. In fact, given a language, one can define the set of *island* rules as the collection of those sub-productions which consume highlightable sequences and map every other token to a particular production that consumes any terminal symbol.

Nevertheless, this strategy is outside the goals of this work. Producing an *island* grammar would induce a development workflow similar to the current state of practice, requiring developers to have a deep understanding of the grammatical structure and undertake a tedious process for the definition of productions with high coverage and accuracy. It is significantly more challenging than providing a tree walker for relevant constructions of the original grammar, which by design correctly consumes all the valid iterations for the same feature. Moreover, the *island* approach would still leave the handing of incomplete language derivations in the hands of the developer. Similarly to the state of practice, *island*-like solutions represent the workflow this paper wishes to avoid.

6 CONCLUSIONS AND FUTURE WORK

The proposed approach is capable of consistently computing perfect SH schemes for the average input files for all the mainstream languages considered. Thereby, it comfortably outperforms the SH accuracy achievable with the here tested state of practice. Furthermore, this solution to SH is capable of producing such outputs in expected time delays significantly faster and with lower variance than formal approaches, i.e., brute-force (BF), capable of equal outputs. However, it is verified that for cases in which the language's grammar results in an efficient parsing of the input, as it is true for Python, the deep strategy does not represent a superior alternative to the BF with regards to the prediction delays, with both solutions yielding time delays suitable for these scenarios.

Future work might investigate further the accuracy with regard to the distribution of online snippets: an aspect that, due to the strict design of a BF method, at this stage was not achievable. For this purpose, the automated APIzation protocol of code fragments presented by Terragni and Salza [39], might be used for the construction of grammatically correct versions of online snippets, from which a formal oracle could be derived. Moreover, the native parallelisation of Convolutional Neural Networks (CNNs) [20], already employed in sequence to sequence translation tasks [14], may be exploited for the achieving of smaller prediction delays.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the Swiss National Science Foundation (SNSF) project "Melise - Machine Learning Assisted Software Development" (SNSF204632).

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org

[2] Toufique Ahmed, Premkumar Devanbu, and Vincent J Hellendoorn. 2021. Learning Lenient Parsing & Typing Via Indirect Supervision. *Empirical Software Engineering* 26, 2 (2021), 1–31.

[3] Kanav Anand, Ziqi Wang, Marco Loog, and Jan van Gemert. 2020. Black Magic in Deep Learning: How Human Skill Impacts Network Training. *arXiv:2008.05981 [cs.CV]* (2020). https://arxiv.org/abs/2008.05981

[4] Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. 2018. Unsupervised Neural Machine Translation. In *International Conference on Learning Representations (ICLR)*.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *International Conference on Learning Representations (ICRL)*.

[6] Ralph Becket and Zoltan Somogyi. 2008. DCGs+ Memoing= Packrat Parsing but Is It Worth It?. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*. 182–196.

[7] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning Long-Term Dependencies with Gradient Descent Is Difficult. *Ieee Transactions on Neural Networks* 5, 2 (1994), 157–166.

[8] Georg Brandl. 2022. *Pygments*. https://pygments.org

[9] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1724–1734.

[10] Jim Dabrowski and Ethan V Munson. 2011. 40 Years of Searching for the Best Computer System Response Time. *Interacting with Computers* 23, 5 (2011), 555–564.

[11] Sérgio Queiroz de Medeiros, Gilney de Azevedo Alvez Junior, and Fabio Mascarenhas. 2020. Automatic Syntax Error Reporting and Recovery in Parsing Expression Grammars. *Science of Computer Programming* 187 (2020).

[12] Sérgio Queiroz de Medeiros and Fabio Mascarenhas. 2018. Towards Automatic Error Recovery in Parsing Expression Grammars. In *Brazilian Symposium on Programming Languages (SBLP)*. 3–10.

[13] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 111–122.

[14] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. 2017. Convolutional Sequence to Sequence Learning. In *International Conference on Machine Learning (ICML)*. 1243–1252.

[15] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 152–162.

[16] John A Hoxmeier and Chris DiCesare. 2000. System Response Time and User Satisfaction: An Experimental Study of Browser-Based Applications. In *America's Conference on Information Systems (AMCIS)*.

[17] Luke AD Hutchison. 2020. Pika Parsing: Reformulating Packrat Parsing as a Dynamic Programming Algorithm Solves the Left Recursion and Error Recovery Problems. *arXiv:2005.06444 [cs.PL]* (2020). https://arxiv.org/abs/2005.06444

[18] Jiwoon Jeon, W Bruce Croft, Joon Ho Lee, and Soyeon Park. 2006. A Framework to Predict the Quality of Answers with Non-Textual Features. In *International ACM Conference on Research on Research and Development in Information Retrieval (SIGIR)*. 228–235.

[19] Yu Kang, Yangfan Zhou, Min Gao, Yixia Sun, and Michael R Lyu. 2016. Experience Report: Detecting Poor-Responsive Ui in Android Applications. In *International Symposium on Software Reliability Engineering (ISSRE)*. 490–501.

[20] Yann LeCun and Yoshua Bengio. 1995. Convolutional Networks for Images, Speech, and Time Series. *The Handbook of Brain Theory and Neural Networks* 3361, 10 (1995).
[21] Minh-Thang Luong, Ilya Sutskever, Quoc V Le, Oriol Vinyals, and Wojciech Zaremba. 2015. Addressing the Rare Word Problem in Neural Machine Translation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. 11–19.
[22] Sérgio Medeiros and Fabio Mascarenhas. 2018. Syntax Error Recovery in Parsing Expression Grammars. In *ACM/SIGAPP Symposium on Applied Computing (SAC)*. 1195–1202.
[23] Douglas C Montgomery. 2017. *Design and Analysis of Experiments*. Wiley.
[24] Leon Moonen. 2001. Generating Robust Parsers Using Island Grammars. In *Working Conference on Reverse Engineering (WCRE)*. 13–22.
[25] Leon Moonen. 2002. Lightweight Impact Analysis Using Island Grammars. In *International Workshop on Program Comprehension*. 219–228.
[26] Fiona Fui-Hoon Nah. 2004. A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait? *Behaviour & Information Technology* 23, 3 (2004), 153–163.
[27] Marco Edoardo Palma, Pasquale Salza, and Harald C. Gall. 2022. *On-the-Fly Syntax Highlighting Using Neural Networks – Replication Package*. https://doi.org/10.5281/zenodo.6958312
[28] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL (*) Parsing: The Power of Dynamic Analysis. *ACM SIGPLAN Notices* 49, 10 (2014), 579–598.
[29] Terence Parrm. 2022. *ANTLR*. https://www.antlr.org
[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NIPS)*. 8024–8035.
[31] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. 2014. Improving Low Quality Stack Overflow Post Detection. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 541–544.
[32] Advait Sarkar. 2015. The Impact of Syntax Colouring on Program Comprehension. In *Annual Meeting of the Psychology of Programming Interest Group (PPIG)*.
[33] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional Recurrent Neural Networks. *Ieee Transactions on Signal Processing* 45, 11 (1997), 2673–2681.
[34] Steven C Seow. 2008. *Designing and Engineering Time: The Psychology of Time Perception in Software*. Addison-Wesley Professional.
[35] Stack Exchange, Inc. 2022. *StackExchange Data Explorer*. https://data.stackexchange.com
[36] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *International Conference on Neural Information Processing Systems (NIPS)*. 3104–3112.
[37] MohammadReza Tavakoli, Abbas Heydarnoori, and Mohammad Ghafari. 2016. Improving the Quality of Code Snippets in Stack Overflow. In *ACM/SIGAPP Symposium on Applied Computing (SAC)*. 1492–1497.
[38] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 118–129.
[39] Valerio Terragni and Pasquale Salza. 2021. APIzation: Generating Reusable Apis from StackOverflow Code Snippets. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 542–554.
[40] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Conference on Neural Information Processing Systems (NIPS)*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). 5998–6008.
[42] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation. *arXiv:1609.08144 [cs.CL]* (2016). https://arxiv.org/abs/1609.08144