Hierarchical Code Representations

Elad Nachmias
Hierarchical Code Representations

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Elad Nachmias

Submitted to the Senate
of the Technion — Israel Institute of Technology
Shevat 5782 Haifa January 2022
This research was carried out under the supervision of Prof. Eran Yahav, in the Faculty of Computer Science.

**Acknowledgements**

I would like to thank my advisor, Prof. Eran Yahav, for his tremendous support and guidance throughout my work. I sincerely appreciate the intellectual independence that he fostered in the laboratory, and I hope to emulate his open mindedness, creativity, and scientific rigor in my own career.

My deepest gratitude goes to my family, for their encouragement, patience, and for all the sacrifices they made for me to have such wonderful opportunities. And to my wife, Eva, I am so grateful for her unconditional love, and for encouraging me to strive to fulfill all my hopes and dreams. I thank her for always believing in me.

The generous financial help of the Technion is gratefully acknowledged.
# Contents

List of Figures

<table>
<thead>
<tr>
<th>Abstract</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Towards logging suggestions</td>
<td>3</td>
</tr>
<tr>
<td>1.2 LogVar problem definition</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Approaches for log-related tasks</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Distributed representations for source-code</td>
<td>6</td>
</tr>
<tr>
<td>1.5 Dissertation goals</td>
<td>7</td>
</tr>
<tr>
<td>1.6 Contributions</td>
<td>7</td>
</tr>
<tr>
<td>2 Preliminaries</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Flat representation for text processing using neural networks (NNs)</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Using flat textual representation as a code modeling strategy</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Abstract Syntax Tree (AST)</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Using abstract syntax tree (AST) as a syntactical-structure-aware code modeling strategy</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Control-Flow Graph (CFG)</td>
<td>12</td>
</tr>
<tr>
<td>2.6 Obtaining a semantic-structure-aware code modeling by involving the CFG and PDG together with the AST</td>
<td>13</td>
</tr>
<tr>
<td>3 Methodology</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>15</td>
</tr>
<tr>
<td>3.1.1 Challenge 1: Following complicated control-flow and data-flow</td>
<td>16</td>
</tr>
<tr>
<td>3.1.2 Challenge 2: Long-range relationships</td>
<td>16</td>
</tr>
<tr>
<td>3.1.3 Challenge 3: Long procedures</td>
<td>16</td>
</tr>
<tr>
<td>3.1.4 Challenge 4: Very few data</td>
<td>16</td>
</tr>
<tr>
<td>3.1.5 Example description</td>
<td>17</td>
</tr>
<tr>
<td>3.1.6 Naive prediction approach</td>
<td>18</td>
</tr>
<tr>
<td>3.1.7 Common pattern structure</td>
<td>18</td>
</tr>
<tr>
<td>3.1.8 Modeling chains of relations</td>
<td>19</td>
</tr>
<tr>
<td>3.1.9 Comparison to other representations</td>
<td>19</td>
</tr>
</tbody>
</table>
3.1.10 Hypotheses ........................................ 20
3.1.11 Key aspects ....................................... 20

3.2 Hierarchical micro-macro code encoding .................. 20
3.2.1 Intuitive description ................................ 20
3.2.2 Formal presentation .................................. 21
3.2.3 AST paths-based micro operator ...................... 24
3.2.4 AST top-level expression combiner .................... 25
3.2.5 control-flow graph (CFG) paths macro operator .... 25
3.2.6 Local-global context mixer .......................... 26
3.2.7 Other choices for operators ......................... 26

3.3 Expand-collapse graph encoding framework ............... 26
3.3.1 Overview ............................................. 26
3.3.2 Intuitive description ................................ 27
3.3.3 Formalization ........................................ 28
3.3.4 Folding strategies .................................. 30
3.3.5 Applications in our work ............................. 31

3.4 Sampling as a technique for data-augmentation and regularization .... 31
3.4.1 Overview ............................................. 31
3.4.2 Random sub-identifiers obfuscation ................. 31
3.4.3 Paths sampling ...................................... 34

4 Evaluation ............................................... 37
4.1 Experiments setting .................................... 37
4.1.1 Benchmarks .......................................... 37
4.1.2 Metrics ............................................. 39
4.1.3 Baselines ............................................ 39
4.1.4 Neural models’ architecture ......................... 40
4.1.5 Identifier encoding ................................ 41
4.1.6 Implementation ..................................... 41
4.1.7 Hyper-parameters and neural-network primitives .... 42
4.1.8 Hardware ............................................ 42
4.2 Results .................................................. 42
4.2.1 Scalability .......................................... 43
4.3 Ablation study .......................................... 44
4.3.1 AST leaves-only sequence ........................... 45
4.3.2 Whole-method encoder is a flat sequence of tokens w/o separators / operators / keywords . 48
4.3.3 Un-ordered set of tokens .............................. 49
4.3.4 Fragmentally un-ordered set of tokens ............... 49
4.3.5 No macro operator .................................. 49
4.3.6 Macro operator is an un-ordered set of CFG nodes ... 50
4.3.7 Macro operator is a single sequence of CFG nodes ordered by their textual appearance in code ........................................ 50
4.3.8 Macro operator is an AST paths encoder for the procedure’s AST pruned at expression statement nodes .................. 51
4.3.9 Macro operator is a graph neural network (GNN) applied over the CFG .......................................................... 52
4.3.10 Macro operator is a set of CFG paths, no update of micro encodings with global context ........................................ 52
4.3.11 No second micro operator ........................................ 53
4.3.12 Micro operator is a GNN / TreeLSTM applied locally over the statement sub-AST .............................................. 53
4.3.13 AST paths-based encoder uses only leaf-to-leaf or only leaf-to-root paths ......................................................... 54
4.3.14 Atomic identifier vocabulary instead of sub-identifiers tokenization ................................................................. 54
4.3.15 Single encoding vector for all sub/identifiers ..................... 55
4.3.16 Full sub-identifier obfuscation ..................................... 55
4.3.17 No random sub-identifier obfuscation regularization ............ 56

5 Related work ................................................................. 57
5.1 Code distributed representations ..................................... 57
  5.1.1 AST leaf-to-leaf paths approach ................................. 57
  5.1.2 Extended code property graph (CPG) approach .............. 58
  5.1.3 Tackling GNN’s over-smoothing issue .......................... 58
5.2 Log-related tasks ......................................................... 60
  5.2.1 Traditional research paradigm studies .......................... 61
  5.2.2 Modern data-driven deep-learning-based studies ............ 63
  5.2.3 Log output processing ............................................. 63

6 Discussion ................................................................. 65
6.1 Hypotheses conclusion ................................................. 65
6.2 Reflection of chain-of-relations and data-flow in the hierarchic model ................................................................. 66
  6.2.1 Re-visiting real-life logging example from overview .......... 66
  6.2.2 Comparing basic data-flow modeling to Code2Seq ........... 66
6.3 Justification of the top-level expressions as the hierarchic leveling cutting-point .................................................. 70
6.4 Methodology limitations .............................................. 71
6.5 Dataset limitations ..................................................... 71
  6.5.1 Too small datasets ................................................. 71
  6.5.2 Representation expressiveness as a cause to quick model overfit ................................. 72
  6.5.3 Open Question: Could incorporating transfer-learning approaches help? ................................. 73
6.6 Hierarchic representation as an improvement of the Code2Seq AST path-based approach ..................................................... 73
6.6.1 Open Question: Is the hierarchic method effective in other code-related tasks? ..................................................... 75
6.7 Hierarchic representation advantages ............................................. 75
6.8 Hierarchic framework for general structures ................................. 76
6.8.1 Open Question: Can the hierarchic framework be effective for other tasks with structured data? ................................. 77
6.9 Effectiveness of the expand-collapse as a graph encoding framework . . . 77
6.9.1 Open Question: Can the expand-collapse framework be useful for other graph tasks? ............................................... 77
6.9.2 Open Question: Is it possible to formulate a theoretical claim about the absence of the over-smoothing issue in the expand-collapse methodology? ............................................... 77

A Appendix 79
A.1 Batching variable-sized indexed inputs ................................. 79

Hebrew Abstract i
# List of Figures

1.1 An example of a logging operation (line 4) in a Java method. 4

2.1 Illustration for code flat tokenized representation 10

2.2 Example of an AST of a partial code snippet (taken from an implementation for the binary search algorithm) 10

2.3 Code2Seq AST paths. 11

2.4 Code2Seq pipeline. 12

2.5 Allamanis extended AST. 13

2.6 Allamanis pipeline. 13

3.1 Illustration for the latent data- and control-dependency relations between tokens scattered along the flat representation for the example from Figure 3.3a. 15

3.2 Illustration for a long-distance relation between tokens in the flat representation for a potential variation of the code example from Figure 3.3a. 17

3.3 Two procedures with logging statements (taken from our evaluation set) that demonstrate common pattern. The pattern is a data-flow sequence of data-dependency, control-dependency and control-flow relations. 18

3.4 Split the AST into the upper and lower parts. 21

3.5 High-level illustration of the hierarchic encoding framework. For simplicity, local-global mixture is disregarded here. 22

3.6 Expand-collapse method illustration. A graph of four nodes is being expanded into three paths. A sequential encoder propagates information within each path separately. Then, for each node, its occurrences along the paths are being folded back into node representation. 28

3.7 Illustration of the benefit of the collapse stage for cross-path information flow. Comparison of multi-layer encoders with and without the collapse stage. 29

6.1 Demonstration of how the pattern on the right (originally presented in Figure 3.3b) is represented by the hierarchic model. 67
6.2 Comparison of the reflection of a data-flow sequence in the *Code2Seq* model versus the hierarchic model with our choices for *micro* and *macro* operators.

6.3 Illustration of a partial program’s control-flow graph with *n* consequent *if* statements leading to $\Theta(2^n)$ possible paths from the entry point to the exit point.
Abstract

In recent years, various deep-learning-based techniques have been suggested for reasoning about programs and tackling code-related prediction tasks. A common concern for these methods is the design of the code encoder, which defines the representation of the input code snippet. Recent advances suggest exploiting the unambiguous code’s underlying structure. Indeed, structure-driven modelings became a common practice, enriching the representation with syntactic and semantic relations. However, the standard graph-based approach relies on message-passing protocols, restricting information propagation over the entire program. Furthermore, paths-based approaches incur unbearable computational complexity for lengthy procedures and thus become unscalable for real-life scenarios.

We address the problem of predicting which variables to log at a given logging insertion point within the input program. This task arises challenges like following complicated sequences of latent semantic relations, which are sparsely scattered over lengthy procedures. We show that the present solutions struggle with this setting.

We present a novel code representation approach designed to model sequences of relations. It involves semantic and syntactic characteristics modeled by paths in the procedure’s control-flow graph (CFG) and paths in the abstract syntax tree (AST), respectively. Our work is the first to leverage the powerful paths-based modeling approach for semantic and syntactic relations together.

Our methodology relies on two novelties. The first is a hierarchic encoding technique for code, which decomposes the entire code structure into local and global levels. Encoding each structural level individually (i) brings the procedure’s textually-distant entities closer, (ii) supports lengthy procedures, and (iii) models compositions of local relations. The second novelty is a graph encoding framework called expand-collapse, which extends the paths-based encoder by allowing information to flow across paths.

We compare our approach to a wide variety of other known representations. Our experiments show that our model is more effective than others by up to 10% (accuracy). We show how the lack of scalability in other rich representations causes them to fail, even when compared to the degenerated non-structured approach. Our method enhances them by reducing 95% of the processed tokens without any loss of information. We provide a thorough ablation study to justify the benefit of different components of our solution.
Chapter 1

Introduction

Software that runs in a production environment typically produces a log output. The log provides visibility into the program by capturing essential information about its execution. If the program exhibits incorrect behavior, the log can be used to trace the erroneous execution, diagnose it, and hopefully fix related bugs. Logging is particularly useful in long-running concurrent and distributed systems, where reproducing errors in a lab setting may be very hard or even impossible.

Conceptually, we would like to store the whole program runtime state after each instruction. This way, the log would contain sufficient information to trace the execution and reproduce the error. In practice, however, this approach would induce unbearable performance costs. Therefore, the challenge is to insert sufficient logging to enable diagnosability without incurring significant runtime overhead.

In most cases, it is up to the programmer to determine where to place log operations in the program and what runtime information should be logged. That is, placing log operations in the program typically happens during program development before any failure occurs. Hence, the programmer has to predict, based on his own experience, what parts of the code are more likely to fail and what runtime information would help diagnose it.

1.1 Towards logging suggestions

Programming paradigms like aspect-oriented programming (AOP) \cite{KLM+97} break down software development into distinct tasks. According to this paradigm, placing log statements within the program can be thought of as a complementary technical programming task. Ideally, it could be done automatically without requiring any intervention from the programmer. Taking this burden off the programmer would channel his efforts in favor of developing the core business logic of the program. Indeed nowadays, saving time for the programmer is a priority. A primary step towards automatic logging insertion could be a relaxed logging recommendation model.

An additional advantage of automatic logging is gaining uniformity over the code-
private void removeApplicationAttempt(AppAttemptId appAttemptId, AppAttemptState finalState, boolean keepContainers) {
    Application application = applications.get(appAttemptId.appId());
    AppAttempt attempt = scheduler.getAttempt(appAttemptId);
    if (application == null || !attempt.isValid()) {
        LOG(??);
        return;
    }
    / ... /
}

Figure 1.1: An example of a logging operation (line 4) in a Java method.

base of a project. Manual logging allows different programmers to adopt various logging practices. That could end up in non-uniform manners of logging throughout the code. Establishing and enforcing a strict logging convention is impractical when using classic formatting techniques, as logging decisions vary across scenarios and may depend on the program’s semantics.

Moreover, logging recommendations could prevent omitting logging of essential variables. There are no straightforward means for detecting a potentially missed variable in a logging statement, as logging decisions are typically neither a part of the program specification nor affecting the program’s correctness. Distribution of a program version with a crucial un-logged variable may lead to postmortem debugging difficulties because its value is unrecorded.

To sum up, we presented three motivations for addressing the logging suggestion task: (i) saving time to the programmer; (ii) improving code uniformity; (iii) preventing the omission of important variables in a log statement.

Our work complements a number of previous efforts to infer where to put a logging statement in a program [ZHF^15, LS16, LSS16, SSL16, LSS17b, LSS17a, ZRL^17, LCSH18, LCS20]. One could use these previous works together with our work that infers which variable to log in this given logging insertion point. Together they could form a complete logging recommendation model.

Our work addresses the problem of automatically infer what variables to log at a given logging placement within a program. The prediction is to be based on what we have learned from a large dataset of other programs containing logging operations. Moreover, after the learning phase, the method should produce qualitative predictions over unseen projects. That requirement ensures the solution usability over new scratch projects.
1.2 \textit{LogVar} problem definition

Given a method’s code \( C \) and a point \( P_{\text{log}} \) within the code where a new statement could be inserted, predict a set of variables \( \{V_1, \ldots, V_k\} \) that are recommended to be logged at \( P_{\text{log}} \). Fig. 1.1 demonstrates the problem by an example.

We could formulate the \textit{VarLog} problem as a sequence-to-set translation task - translating from sequence of code tokens to a set of variable names that appeared in the input sequence. Although such formulation is common in other programming-related tasks and similar to previous work (e.g., \cite{LXL+19}), it unnecessarily and implicitly binds the solution to a \textit{flat representation} of the input procedure. We choose to keep the formulation representation-choice free to explore the representation that best copes with the unique challenges characterized by the \textit{LogVar} problem.

1.3 Approaches for log-related tasks

Several techniques have already been proposed for similar log-related tasks, like enhancing a given log message \cite{YZP+12}, suggesting where to place log-statements in the code \cite{ZHF+15, LS16, LS16, SSL16, LSS17b, LSS17a, ZRL+17, LCSH18, LCS20}, and predicting what severity level to associate with a given log-statement \cite{LSH17, LPH+19, LLCS21}. Most of them are mainly based on hand-crafted features and ad-hoc statistical studies conducted by the authors over the source-code of some large open-source projects. Hand-crafted features are time-consuming and generally too specific. In the context of programming-related tasks such as logging, hand-crafted features are not instantly adaptable to different projects, programming languages, or code conventions. That’s why the most recent studies chose to employ more recent learning techniques based on strong language models like recurrent neural networks (RNNs) and gated graph neural networks (GGNNs).

The \textit{LogVar} problem has recently been studied by Liu et al. in their paper \cite{LXL+19}. They formulated the problem as a supervised-learning task and tackled it by employing deep-learning techniques. More specifically, they applied a sequential neural-network model over the procedure’s code that precedes the questioned logging statement. Unfortunately, as we show in Chapter 4, the success of such sequential modeling is limited for some practical logging strategies, like logging variables that are defined in a distant location over the code. Generally, in the context of programming-related tasks, it has already been shown before \cite{AZLY19, ABK18} that the success of such linear representation highly depends on (i) the overall length of the method, (ii) the proximity of meaningful semantic relations within the flat representation, and (iii) the complexity of such relations. In our work, we extend their research by addressing these concrete challenges that arise in the context of logs. We explore how their solution struggles with these challenges and we present an alternative approach that aims to overcome them by leveraging a richer code representation.
1.4 Distributed representations for source-code

In recent years, various deep-learning-based techniques have been suggested for tackling programming-related tasks, like suggesting descriptive names for functions [ABLY19, AZLY19, DAY20], variable-misuse detection [ABK18], semantic code-search [CLK+19], code-completion [HMLJ19, ASLY20] and more. Such techniques typically involve defining a probabilistic model, training it over a large number of programs, and finally employing the trained model to produce predictions over new previously unseen programs.

A common concern for these methods is selecting a strategy for feeding the input code to a neural network. One straightforward approach, adopted from the field of natural language processing (NLP), is to represent the code as a simple linear sequence of tokens. However, unlike natural languages, code possesses distinct characteristics like long distances between related tokens, which imposes difficulties on such representation. Fortunately, programming languages are usually defined by unambiguous grammar, enabling efficient parsing of source code into a unique syntactic structure. Recent advances in the study of code representations suggest structure-driven modelings, taking advantage of these well-defined parsing rules.

One of the recently studied code-representation techniques is based on the program’s AST. The AST holds syntactic information about the code, which has been proven effective for deep-learning-based models [AZLY19]. Alon et al. also showed the effectiveness of paths-based modeling, in which a structure (like a tree) is being regarded as a set of paths within it. In another work [ABK18], Allamanis et al. extended the AST structure by adding edges between nodes like control-flow, data-flow, guarded-by, and first/last-use edges. These edges characterize semantically meaningful relations.

In a follow-up work [HSS+20], the authors pointed out that Allamanis’ architecture is confined to graph-based message passing models, which has been shown to fail dealing with long-range relations due to the over-smoothing issue [CLL+20, OS20, LF21, AY21, LWJ20, RW21]. To overcome this hurdle, they suggested two other modeling approaches (called Graph-Sandwich and GREAT) that take advantage of the strong expressiveness of neural sequential models (e.g.: RNN, Transformer). However, the first still relays on GNN, the second is confined to the Transformer architecture by design, and they both are not solving the scalability issues arising from long procedures and are not designed to leverage the paths-based modeling idea.

Multi-resolution hierarchical modeling [RW21] is another approach suggested to deal with the insufficiency of message-passing models for capturing long-range interactions in graphs. However, it still uses the GNN architecture and doesn’t leverage the powerful paths-based modeling concept.
1.5 Dissertation goals

As the LogVar task poses prominent structural-related challenges, the goal of this work is to explore the effect of structural information over this task and to find an appropriate structural-driven modeling to the problem.

In this work we propose a novel hierarchic code representation that (i) leverages the strength of the paths-based modeling together with the sequential neural models, (ii) embeds both syntactic relations featured by the AST and semantic relations featured by the CFG, (iii) doesn’t suffer from inherent long-range relations capturing limitations, and (iv) efficiently supports long procedures.

We show that our method outperforms all baseline methods over the LogVar task, while nowadays common structure-based modelings utterly fail, even when compared to the naive flat representation. To analyze the model’s expressiveness, we first identify several interesting patterns exhibited by a significant portion of the dataset. Then, we supply an intuitive explanation about how our approach can generalize these patterns better than current techniques.

1.6 Contributions

- Further step towards a logging suggestion tool for real-world programming scenarios.
- Study of complex logging patterns that involve sequences of relations.
- Novel hierarchic code distributed representation approach modeling sequences of relations and efficiently scaling for lengthy procedures.
- Expand-collapse graph encoding framework for long-distance information propagation.
- Study the LogVar task’s structural requirements by thoroughly exploring the contribution of various factors in our and baseline solutions.
- Developing a research methodology to assess the individual contribution of the local and the global code’s structures independently.
- Thorough evaluation and comparison to various baseline approaches (full open source code would become available at github.com/eladn/ndfa).
Chapter 2

Preliminaries

2.1 Flat representation for text processing using NNs

Some of the methodologies referred to in our work use the flat representation as a manner to neurally process textual inputs. Here we explain its fundamentals.

Before the training, a fixed vocabulary is being created from the most frequent words in the dataset carpus (the maximum vocabulary size is usually pre-defined). Each word in the vocabulary is embedded in $\mathbb{R}^d$ during the training process. More specifically, in the beginning of the training an embeddings matrix from $\mathbb{R}^{(|V| \times d)}$ is being initialized (where $|V|$ is the vocabulary size). By so, the i-th row in the embeddings matrix is the embedding vector represents the i-th row in the vocabulary.

When feeding a sentence to the network, its words are embedded individually; that is, each word is looked-up in the vocabulary to find its index, then the row of that concrete index is being taken from the embeddings matrix. This produces a sequence of vectors. Sequential neural models are designed to accept these sequences as inputs and process them. Two prominent sequential models are the LSTM [HS97] and the Transformer [VSP+17] which became widely used in the last few years.

During the learning process, upon backpropagation, the gradients flow back to this matrix and thus the embedding vectors are being updated on every optimizer step.

2.2 Using flat textual representation as a code modeling strategy

Adapted from the field of NLP, this approach represents the code snippet as a linear sequence of tokens in accordance with its textual appearance. It is sometimes considered as an naive approach, as it doesn’t take advantage of the well-defined underlying code’s structure which can be retrieved from the source code at a considerably low computational cost. The unsuitability of this representation strategy to code is being exemplified later in Sections 3.1.1 to 3.1.3.

A token can be one of the followings: (i) a built-in keyword (e.g: for, int), (ii)
2.3 Abstract Syntax Tree (AST)

Programming languages are defined by a context-free grammar that any code snippet should follow to be considered as syntactically valid for this language. The language’s grammar is formally specified as a set of inductive rules written in declarative syn-
A parsing tree is used to describe the steps of applying the grammar’s derivation rules to generate a given valid code. Programming languages usually use specific grammars classes with efficient parsing algorithms that produce these parsing trees from input code. In the context of programming languages, parsing trees are called Abstract Syntax Tree (AST). For unambiguous grammars, which are adopted by most programming languages, there is a unique parsing tree for any given code snippet.

Each inner-node in the AST symbolizes a rule that should be applied over the node’s children as operands to produce its associated code. For example, the code `array.getSize() - 1` matches the application of the rule `BinaryMinusExpr` over the two expressions `array.getSize()` and `1` (which themselves can be described as applications of derivation rules). Thus, it would be parsed into an inner node with the attribute `BinaryMinusExpr` and two children sub-trees of the inductively parsed operand expressions. The leaves of the tree are terminals from the original code string after tokenization. Figure Figure 2.2 demonstrates the constructing of a parsing-tree from a partial code snippet.

### 2.4 Using AST as a syntactical-structure-aware code modeling strategy

Alon et al. presented the Code2Seq code representation approach [ABLY19]. The author’s goal was to take advantage of the well-defined syntactical structure of the input source-code. They chose to express these syntactical relations as paths in the procedure’s AST. In their work, they illustrated how certain procedures can be characterized by unique paths over its AST. More technically, their method produces a distributed representation vector for a given procedure, by encoding paths over its AST. It is comprised of four main stages described hereby: (i) All the leaf-to-leaf paths are being extracted from the procedure’s AST, as illustrated in Figure 2.3. (ii) Each AST node is embedded into $\mathcal{R}^d$. Inner nodes are embedded by their type (ex: `BinaryAdd`, `MethodCall`) and their location among their siblings. Terminals (leaves) are embedded...
by their associated identifier. (iii) The embedded paths are fed to a neural sequential model (the authors used two layers of an LSTM \cite{HS97}). (iv) A weighted average is being calculated over the encoded paths of the procedure, thus, resulting with its final distributed representation. Figure 2.4 illustrates the pipeline of Code2Seq.

### 2.5 Control-Flow Graph (CFG)

The Control-Flow Graph (CFG) represents the various possibilities of control transfer between program instructions during its execution. A program’s execution can be described by a path in the CFG. The CFG is used by the compiler for static program analysis and for optimizations. Methods like Data-Flow Analysis are performed over the CFG to systematically gather semantic information about the propagation of computed values along the program. Unlike the AST, which conveys purely syntactic features, the CFG and its variations all represent semantic characteristic of the program’s execution.

In a statement-level CFG, the program statements or conditions are considered to be the atomic computational units represented as nodes, sometimes referred to as basic blocks. Apart from statements and conditions, its nodes set also consists of two additional procedure-entry and procedure-exit nodes.

Directed edges describe the possible control transfer between statements. Within a block of statements, the control is transferred deterministically from one statement to the following, without indeterministic branches. This case induces a linear graph of the shape of a string. However, the transfer of control out of conditions’ branches is unknown prior to the program’s execution. Thus, conditional nodes have an out-degree of at least two.

The Program Dependence Graph (PDG) is another familiar variation to the CFG. It is produced by replacing the control-flow edges by data- and control-dependency edges. A data-dependency (or use-def) edge connects a statement having a symbol definition to a reachable statement that uses this symbol (provided that there is at least one path from the source statement to the target one without another definition to the same symbol in-between). A control-dependency edge connects a conditional statement to a statement which the control-flow into it is dependent over the condition. The PDG can be calculated from the original CFG.
2.6 Obtaining a semantic-structure-aware code modeling by involving the CFG and PDG together with the AST

In their paper [ABK18], Allamanis et al. extended the AST into a more comprehensive program graph representation by enriching it with supplemental edges characterizing semantic relations like control-flow, control- and data-dependency. Figure 2.5 exemplifies application of these edges over the example from Figure 2.2. The edges are attributed by their types (e.g: ASTChild, LastRead, LastWrite, LastLexicalUse, ComputedFrom, GuardedBy). This kind of extended code graph is sometimes called Code Property Graph (CPG), as originally suggested by Fabian et al. [YGAR14] for advanced code analysis purposes. The authors applied a graph neural network (GNN) to neurally process their new graph representation. Figure 2.6 illustrates the entire pipeline of their approach.
Chapter 3

Methodology

In this section we present two methodologies regarding code modeling. First, in Section 3.1, we outline the key challenges of the LogVar task for which our methodology is inspired by. Then, in Section 3.2, we present the hierarchic encoding framework, which consists our main modeling concept. Later, in Section 3.3, we introduce the expand-collapse graph encoding framework, which is an extension to the basic paths-based tree representation approach. Finally, in Section 3.4 we describe our ad-hoc regularization techniques and discuss their significance in the light of our task’s constraints.

3.1 Overview

In Chapter 1 we explained how addressing the LogVar problem may assist programmers. The need for a solution is well-established, considering that prior works have already tried tackling it. Still, solving the LogVar problem with learning techniques raises technical challenges that previous works struggle to cope with. In this section we identify those challenges and present an especially-hard subset of the problem that exhibits them. Then we explain how our novel method is designed to overcome these challenges.

Figure 3.1: Illustration for the latent data- and control-dependency relations between tokens scattered along the flat representation for the example from Figure 3.3a.
3.1.1 Challenge 1: Following complicated control-flow and data-flow

It is common practice to log variables that affect control-flow decisions in a program (for example, did a certain event occur). In [LLCS21] the authors conducted a manual survey in which they partitioned the logging-statements into five different categories. They named the biggest category *Logic-Branch*, and it contains logging-statements that are guarded by a condition of an unexpected execution branch. Intuitively, deciding which variables to log at a certain point sometimes involves tracking the computation of their values. Formulating these concerns as patterns is possible by following the program’s data- and control-flow. Figure 3.1 illustrates how such relations are being latent with respect to the flat representation. Afterwards, Figure 3.3 provides a suggestion for a more explicit and simplified manner to represent such a concatenation of relations.

3.1.2 Challenge 2: Long-range relationships

The *LogVar* task could be thought as measuring a variable’s relevance to the runtime state at some given code point. As it turns out, this quality is attributed to long-range relationships in the program. For example, in Fig. 1.1, the logged variable is declared in line 1, while the log statement is in line 5. Generally, such textual distance between the method’s entry point and a point within it spans over big amount of tokens (in the program’s flat representation), where many of them might be irrelevant at the logging point. The declaration includes the variable type, which might hint about its likeliness to be logged. In Figure 3.2, the code snippet from Figure 3.3a has been modified by inserting a conditioned code block right after the definition of *name*. This piece of code can, for example, handle some corner-case for special values of *name*. In this case, the distance between the definition of *name* and its use in *container* definition has been enlarged.

3.1.3 Challenge 3: Long procedures

Logging is often present in long procedures. An average procedure in our dataset has 228 (±201) tokens (90% percentile is 500) and 36 (±32) lines (90% percentile is 81). Unfortunately, the nowadays code encoding techniques are limited to short procedures by design. The flat representation is affected by the sequential model’s limitations w.r.t the code’s textual length. More specifically, RNNs become less effective for lengthy sequences due to the *vanishing gradients* issue, and the *Transformer* architecture induces quadratic complexity w.r.t the sequence length. The *Code2Seq* [ABLY19] structure-based representation suffer from quadratic complexity w.r.t the procedure’s AST size.

3.1.4 Challenge 4: Very few data

In Section 4.1.1 we point-out the number of extracted logging examples from each benchmarks. In average, there are 923 valid logging examples per code project. It
implies that a substantial amount of projects has to be processed to construct a considerable dataset. However, the number of accessible qualitative open source projects is limited. This leaves us with practically very few training data to learn from. In Section 6.5.2, we address the difficulties arising from the shortage of data when training expressive models and the measures we take in our method to overcome them. Prior logs study [LLCS21] also mentioned this shortage in the context of the log-level prediction task.

### 3.1.5 Example description

In the rest of this section, we present two code examples taken from our validation set and simplified for readability purposes. We show that although they may look different from a textual and a syntactic perspective, it appears that they do share a very similar pattern. This pattern also exhibits some of the challenges presented above.

We then demonstrate our code-representation method on these examples. The produced representations support the idea that our approach can theoretically detect this pattern. Finally, we explain why other techniques that were used before struggle to spot such patterns.

To establish an initial intuition, we reflect the logging process as it is performed by a human programmer over the procedure `unregisterSPI()` from Figure 3.3a. The logging statement in line 6 is performed on some fail-branch, where the `container` of `spi` could not be found. In that case, the developer chose to log the variables `spi` and `name`. It makes sense, as these are the only informative variables that may help tracking the problem’s cause after the execution. The same analysis applies for procedure `getWcRoot()` from Figure 3.3a.
Figure 3.3: Two procedures with logging statements (taken from our evaluation set) that demonstrate common pattern. The pattern is a data-flow sequence of data-dependency, control-dependency and control-flow relations.

3.1.6 Naive prediction approach

Note that in the general case, an educated guess might have been to log variables that occurred in the condition of the IfStmt that switched the control into this fail-branch. However, in our case, logging the variable container that occurred in the IfStmt condition in line 5 makes no sense, as it does not hold valuable information and therefore does not convey sufficient usable information for the postmortem debugging. Hence such simple rule-based heuristic might not be generally effective. Of course, one could formulate a set of ad-hoc rules to catch this specific case, but it won’t catch other examples that exhibit slightly different patterns. Yet, we would like to find a common underlying structure that inherently allows learning such patterns.

3.1.7 Common pattern structure

Figures 3.3b and 3.3d represent a pattern in the procedures from Figures 3.3a and 3.3c accordingly. If we look at Figure 3.3a, we first observe that the logging point is laid within a fail-branch, which ends in a Return statement. The entrance to this fail-branch is conditioned by the variable container. Its value depends on name, which
itself depends on \textit{spi} (which are the actual two logged variables). Virtually, we log the "origins" of \textit{container} which just tested to be null. Overall, we end up with a \textit{chain of relations}, as presented by the pattern chart in Figure 3.3b. A similar analysis applies for the second procedure as well. Note that for both procedures the two matching patterns hold a similar structure.

A common generalized structure for the two patterns can be described as the following chain of relations: (i) data-dependency relation between some variable \(x\) and some variable \(y\); (ii) data-dependency relation between the variable \(y\) and some variable \(z\); (iii) is-null control-dependency relation between the variable \(z\) and the log statement; (iv) control-flow relation between the log statement to the procedure’s exit. Note that this pattern is presented here as an example among many other patterns exhibited by different examples. We are not looking for such explicit patterns, but we only show them here as a background motivation for the rest of the explanation.

### 3.1.8 Modeling chains of relations

Our aim is to find a code-representation technique where such chains of relations can be modeled. To do that, let’s first understand where these mentioned relations come from. Remember that a data-dependency relation actually originates from an assignment statement, and the null-test originates from a condition expression. That way, each stage in the above-mentioned pattern is a program statement or a condition of some control-structure. Additionally, the order of these stages is an essential property of the overall structure. In fact, it is conveyed by the control-flow of these statements. The overall chain can be modeled by a path in the CFG, whereas each node along the path can be modeled by the inner relations within its associated statement.

Indeed, in Section 3.2 we describe in detail how our method models a procedure by employing two different stages. The first stage models each statement independently by its inner relations. Eventually, the second stage models the entire chain by propagating the obtained statement’s encodings along paths in the CFG. Finally, in Section 6.2 we show how the pattern from Figure 3.3b is modeled by our hierarchic approach, and how general data-flows are reflected in this modeling.

### 3.1.9 Comparison to other representations.

Although the shown pattern is by no means considered complicated, none of the commonly-known code representation approaches models it.

The flat representation, as incorporated for the \textit{LogVar} task in [LXL+19], lacks of any explicit modeling for latent code relations. In Figure 3.1, we illustrate the insufficient reflection of these latent relations in the flat representation.

In [ABK18], Allamanis et al. model the procedure as local areas of both syntactic and semantic relations. Although it takes into account the kind of relations we use, its
ability to detect sequences of relations is very limited due to the over-smoothing issue in GNNs [CLL⁺20, OS20, LF21, AY21, LWJ20, RW21] which the method is based on.

In [AZLY19, ABLY19], Alon et al. model the procedure as a set of syntactic relations, where each relation involves only two terminals (program identifiers; e.g.: variable names, type names). We aim to model an ordered chain of relations. Although each atomic relation in the chain might involve only two terminals, the overall chain that connects these relations may consist multiple relations. Additionally, the chain itself is connected via semantic control-flow relations, which are not employed at all by Alon’s approach. Later, in Section 6.2.2 we explain the lack of reflection of data-flows in the Code2Seq model accompanied by a detailed example.

We further discuss both approaches in Sections 2.4 and 2.6 and compare our results to theirs in Chapter 4.

3.1.10 Hypotheses

Our work is driven by two main hypotheses. In Sections 4.2 and 4.3 we conduct experiments to test these hypotheses, and later, in Section 6.1 we conclude these hypotheses with respect to the findings made throughout the work.

**Hypothesis 1:** Grasping code structural information is essential to effectively solve of the LogVar task.

**Hypothesis 2:** Our hierarchic representation framework can achieve better results than other representations over the LogVar task thanks to its structural modeling.

3.1.11 Key aspects

The illustrated example highlights several key aspects of our approach:

- The application is learning which variables to log.
- Practically, reasoning about which variables should be logged brings-up simple explanations that depend on semantic relationships like control-flow, control.Dependency and data-dependency.
- The approach is modeling sequences of the program’s relations using paths in the statement-level CFG. Such representation allows expressing the desired patterns.

3.2 Hierarchical micro-macro code encoding

3.2.1 Intuitive description

We propose a two-stage hierarchical code encoder. We conceptually separate the program’s structure into two hierarchic levels. The local level consists of individual top-level expressions associated with the program’s CFG nodes; it can be, for example, an
expression-statement or a condition expression of a control structure (like while loops or if statements). Each such local element is related to a CFG node that represents it. The global level consists of the inter-relations between the standalone local statements. Figure 3.4 demonstrates this local-global partitioning over a procedure’s AST.

During the first stage of the hierarchic encoding process, we apply a micro operator over each top-level expression independently. The micro operator produces a single vector in $\mathbb{R}^d$ for each statement; it will be called the local encoding. In the second stage, we apply the macro operator over the local statements’ encodings (yielded by the micro operator). The macro operator is responsible for passing contextual information globally between statements via inter-statements relationships such as control-flow or cross-statements data-dependency. Figure 3.5 illustrates the hierarchic flow.

### 3.2.2 Formal presentation

**Top-level expression** An expression is called top-level expression if its parent node in the AST is not an expression. The statement-level CFG is defined over the procedure’s top-level expression. Namely, each top-level expression is associated with a CFG node that represents it. For example, top-level expression can be an ExpressionStatement, an expression for a ReturnStatement / YieldStatement, a condition of a control statement (like a while-loop), or an initialization / update statement of a for-loop. The lower AST partition, illustrated in Figure 3.4, consists of the procedure’s top-level expressions.

**Expression inner elements** A top-level expression is defined by a code snippet. The inner elements are the components comprising that code snippet. The type of the inner elements varies between different code representations. Practically, if we choose to represent an expression as a flat sequence of code tokens, then the tokens are considered...
Figure 3.5: High-level illustration of the hierarchic encoding framework. For simplicity, local-global mixture is disregarded here.

to be the inner elements of this expression. Alternatively, if we use the expression’s sub-AST to represent it, then its nodes or its paths can form the set of inner elements of the expression.

**Input procedure terminology** Given a procedure $\mathcal{P}$, we denote its top-level expressions as $\mathcal{T} = \{t_1, \ldots, t_{|\mathcal{T}|}\}$, the statement-level CFG as $\mathcal{G}_{\text{CFG}} = (\mathcal{V}_{\text{CFG}}, \mathcal{E}_{\text{CFG}})$. The set of CFG nodes can be written as $\mathcal{V}_{\text{CFG}} = \{v_{\text{CFG}}^{\text{enter}}, v_{\text{CFG}}^{\text{exit}}\} \cup \{v_{\text{CFG}}^1, \ldots, v_{\text{CFG}}^{\|\mathcal{T}\|}\}$, where $v_{\text{CFG}}^{\text{enter}}, v_{\text{CFG}}^{\text{exit}}$ are the procedure’s entry and exit point, and for each $i \in [|\mathcal{T}|]$, the node $v_{\text{CFG}}^{t_i}$ is associated with the top-level expression $t_i$.

**Abstract operators terminology** We first present the hierarchic algorithm in a generic form, by keeping out the implementation details of the operators applied at each step. The operators’ requirements are listed below. Later, we describe the concrete specification of our operators’ choices. In practice, our operators are implemented using common neural architectures with weights that are learnt during the model’s training.

- The *embedder* conforms to the functional form $o_{\text{embedder}} : \mathcal{T} \to \langle \mathcal{R}^d, \ldots, \mathcal{R}^d \rangle$. Given a top-level expression, this operator matches distributed representations for its inner elements. Practically, it means that each code token or AST node or AST path (which of these depends on the chosen representation) is assigned with a vector in $\mathcal{R}^d$.

- The *micro operator* conforms to the functional form $o_{\text{micro}} : \mathcal{T} \times \langle \mathcal{R}^d, \ldots, \mathcal{R}^d \rangle \to \langle \mathcal{R}^d, \ldots, \mathcal{R}^d \rangle$. It matches new distributed representation for a given statement $t_i \in \mathcal{T}$ and the distributed representations of its inner elements (that have been
calculated in previous steps). Practically, our suggestions for this operator perform some information propagation between the inner elements of the expression along some structural paths, as commonly done by existing code representation approaches. Note that the micro operator is applied over a stand-alone contextless expression, without involving information from other parts of the input procedure.

- The expression combiner conforms to the functional form $o_{\text{combiner}}: \mathcal{T} \times (\mathbb{R}^d, \ldots, \mathbb{R}^d) \rightarrow \mathbb{R}^d$. Given a top-level expression $t_i$ and its set of inner encodings, the combiner produces a local representation vector in $\mathbb{R}^d$ for the according CFG node $v_i^{\text{CFG}}$.

- The macro operator conforms to the functional form $o_{\text{macro}}: \mathcal{G}_{\text{CFG}} \times (\mathbb{R}^d, \ldots, \mathbb{R}^d) \rightarrow (\mathbb{R}^d, \ldots, \mathbb{R}^d)$. It receives the local representations of the CFG nodes and returns new global-aware representations for them.

- The local-global mixer conforms to the functional form $o_{\text{mixer}}: \mathcal{T} \times \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^d$. It operates over a top-level expression $t_i$, the global-aware representation of its CFG node $v_i^{\text{CFG}}$, and a local encoding vector of some inner element of $t_i$. It produces a new global-aware encoding vector of the same expression’s inner element.

**Algorithm 3.1 Hierarchic code encoder**

<table>
<thead>
<tr>
<th>Input: $\mathcal{T}, \mathcal{G}_{\text{CFG}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Encodings for the inner elements of each $t \in \mathcal{T}$.</td>
</tr>
</tbody>
</table>

1: for $t_i \in \mathcal{T}$ do
2: $\langle e_{i1}^t, \ldots, e_{||t_i||}^t \rangle \leftarrow o_{\text{embedder}}(t_i)$
3: $\langle l_{i1}^t, \ldots, l_{||t_i||}^t \rangle \leftarrow o_{\text{micro}}(t_i, \langle e_{i1}^t, \ldots, e_{||t_i||}^t \rangle)$
4: $l_{i}^{lc} \leftarrow o_{\text{combiner}}(t_i, \langle l_{i1}^t, \ldots, l_{||t_i||}^t \rangle)$
5: end for
6: $\langle g_1^t, \ldots, g_{||\mathcal{T}||}^t \rangle \leftarrow o_{\text{macro}}(\mathcal{G}_{\text{CFG}}, \langle l_{1}^{lc}, \ldots, l_{||\mathcal{T}||}^{lc} \rangle)$
7: for $t_i \in \mathcal{T}$ do
8: $\langle m_1^t, \ldots, m_{||t_i||}^t \rangle \leftarrow o_{\text{mixer}}(t_i, g_i^t, l_{i1}^t, \ldots, o_{\text{mixer}}(t_i, g_i^t, l_{||t_i||}^t))$
9: $\langle f_1^j, \ldots, f_{||t_i||}^j \rangle \leftarrow o_{\text{micro}}(t_i, \langle m_1^t, \ldots, m_{||t_i||}^t \rangle)$
10: end for
11: return $\{\langle f_1^j, \ldots, f_{||t_i||}^j \rangle | j \in [\mathcal{T}] \}$

**Generic algorithm explanation**

The pseudo-code for our hierarchic code encoder is given in Algorithm 3.1. Here we provide a further explanation of its different stages.

The embedding (line 2) constitutes the initial numerical distributed representation for the input code. Each expression’s element (token or AST node) is embedded separately according to its type (e.g: BinaryAnd, SimpleName, etc.). The term $e_i^t$ denotes
the local embedding of the $i$-th element of the top-level expression $t_j$. In this stage, neither local (inter-expression) nor global (cross-expression) information propagation has been applied.

The first information flow occurs in the application of the first micro operator in line 3. Each top-level expression is independently processed to obtain its local inner encodings. Note that these encodings are not global-aware, meaning that information has not been shared across top-level expressions. The term $l^j_i$ denotes the produced local encodings of the $i$-th element of the top-level expression $t_j$ as calculated by the first micro operator.

Just before the global information propagation step, we make sure to have a single vector encoding for each top-level expression, which will constitute the representation of the CFG node associated to it. The term $l^j_c$ (defined in line 4) denotes the local encoding of the CFG node $v^j_{CFG}$. In line 6 the macro operator globally propagates information across the CFG nodes. The term $g^j$ denotes the global-aware encoding of the CFG node $v^j_{CFG}$.

In this stage, the local and the global encodings are detached, in the sense that the micro encodings of the expression’s inner elements are not global-aware. The final output is expected to be these element’s encodings, as they are the most detailed code’s component exhibiting fine-grained data resolution. Hence, we want to integrate the global-aware information back to the micro encodings. Indeed, in line 8 we propagate the global contextual information down to the expression’s inner elements encodings. The term $m^j_i$ denotes the encoding of the $i$-th element of the top-level expression $t_j$ mixed with the global-aware encoding of the containing CFG node $v^j_{CFG}$. Then, in line 9 we perform another information propagation within each standalone top-level expression, but this time after the global information is also taken into account. This stage has found to be essential in the current hierarchical framework (shown in Section 4.3). Finally, the term $f^j_i$ denotes the final encoding of the $i$-th element of the top-level expression $t_j$.

Note that we use two micro operators, $o^{(1)}_{micro}$ (line 3), and $o^{(2)}_{micro}$ (line 9). In practice, they might share the same architecture and differ only in the learnt weights. Later, in Section 4.3 we justify the different choices made in the algorithm’s design.

### 3.2.3 AST paths-based micro operator

A micro operator is given with a program’s standalone top-level expression. It is presumed to perform some kind of data propagation across its inner elements, and produce the resulted local-aware encodings of its inner elements. The inner elements of an expression can be, for example, its tokens, its related AST nodes, or its related AST paths. Our suggested paths-based AST micro operator uses the expand-collapse graph encoding approach, presented in Section 3.3, which itself is based on Code2Seq [ABLY19], as described in Section 2.4. The given top-level expression is associated with a sub-AST
that represents it. We employ the Code2Seq method over this sub-AST as a standalone tree (without regarding its ancestors or other siblings in the entire AST), and with some modifications as described hereby. We observed that 8% of the top-level expressions have a single leaf (the sub-tree is a degenerated string; e.g. a SimpleName of an identifier that appears as the condition of a control structure). In such cases, the set of leaf-to-leaf paths is empty. Hence, apart from the leaf-to-leaf paths, we also extract the leaf-to-root paths in this sub-AST (where the root is considered to be the root of this sub-tree and not the root of the entire AST). We empirically justify the integration with the leaf-to-root paths later in Section 4.3. The second modification regards the final encodings. Unlike the original Code2Seq, here, after applying the sequential encoder over the paths, we perform the collapse stage, as presented in Section 3.3. Therefore, rather than resulting in a set of AST paths encodings as with the original Code2Seq method, in our version, we obtain new representations for the AST nodes. The motivation for this choice is also described in Section 3.3. We empirically justify this choice later in Section 4.3. We use the general attention mechanism to fold the paths into nodes. That is, for each AST node, all of its occurrences (scattered along the AST paths) are attended with respect to a general learnt query vector. Notice that the attention is applied multiple times in parallel on a single top-level expression, as it is used on each node separately. Finally, the micro operator produces the resulted AST nodes representations as the inner encodings of the given top-level expression.

3.2.4 AST top-level expression combiner

The combiner’s role is to produce a single representing vector for a top-level expression out of its inner elements encodings, which, in our case, are a set of AST nodes encodings. For this purpose, we apply a general attention mechanism over the set of AST nodes. We also checked other alternatives like using mean of nodes, sum of nodes, or attention over the nodes with the root of this sub-AST as the query vector. Using the general attention yielded the best results.

3.2.5 CFG paths macro operator

We employ the expand-collapse approach (presented in Section 3.3) over the procedure’s CFG. In the expand stage, we take all paths from the procedure’s entry point $v_{enter}^{CFG}$ to the procedure’s exit point $v_{exit}^{CFG}$. In Section 6.4 we address the theoretical and practical complexity of the number of such paths in a procedure.

Paths-based approaches have been used before in Code2Seq over syntactical relations. Here, in the macro operator, we chose to use it over semantic control-flow relations. The motivation to adopt the paths-based approach in this context, rather than using other graph encoding strategies, is due to the ability of the sequential models to grasp long-range relations in a sequential input. As mentioned in Section 3.1.2, our task is characterized by long-range relations over the procedure’s code that we aim
to shorten.

Another choice is using the control-flow edges as a route for the local information of the top-level expressions to propagate across. The motivation for this choice is related to the aim for a chain-of-relations modeling, as explained in Section 3.1.8. Later, in Section 4.3 we explore other kinds of macro operators to justify these choices.

### 3.2.6 Local-global context mixer

For a top-level expression $t_i \in \mathcal{T}$, a global-aware encoding $g^i \in \mathcal{R}^d$ of its CFG node $v^i_{\text{CFG}}$, and the local encoding of its $j$-th inner element $l^i_j$, the mixed encoding of this inner element is defined as follows:

$$o_{\text{mixer}}(t_i, g^i, l^i_j) \triangleq \text{LeakyReLU}(W_m \times [g^i || l^i_j])$$

Where $[\cdot || \cdot]$ marks vector concatenation, $W_m \in \mathcal{R}^{(d,2d)}$ is a learnt weight that constitutes a linear projection layer, $\times$ marks matrix-vector dot product, and the LeakyReLU is the common element-wise leaky rectified linear unit activation function [MHN+13]. Hence the mixer yields a vector in $\mathcal{R}^d$ (the same domain as the original micro encodings). We tried multiple versions for the context mixer, including addition and applying a gated RNN cell. However, the presented version generally yields the best results.

### 3.2.7 Other choices for operators

In Section 4.3 we suggest other micro and macro operators and compare them to the ones presented here. Our implementations for the micro operator take into account mostly syntactical features, as it is confined by design to local information on the operand top-level expression. The macro operator, on the other hand, has access to the global context of the input method. Therefore, it can take advantage of semantic features that involve relations between multiple statements. Practically, our architecture can be described as a hierarchy of encoding stages.

### 3.3 Expand-collapse graph encoding framework

#### 3.3.1 Overview

Generally, one of our main goals is to find better code representations that benefit from the code’s underlying structural information. Fortunately, programming languages well define parsing rules to produce such structures from source code. Thus, we feed the deep neural network (DNN) with such structured data. To neurally process such input forms, we adopt the paths-based modeling approach presented by Alon et al. [ABLY19], where the authors originally applied it over trees. We employ their paths-based method both on trees and on general graphs. For some cases this method is found to be more effective than other neural graph processing methods like TreeLSTM.
or message-passing GNNs, as shown both in Alon’s mentioned paper and also by us in Section 4.3. The path-based approach profits from the power of the sequential models to propagate information over long distances. In that way, it is designed to tackle the well-known over-smoothing limitation \[\text{CLL}^+20, \text{OS}20, \text{LF}21, \text{AY}21, \text{LWJ}20, \text{RW}21\] of the GNNs’ message-passing graph encoding framework.

However, this approach, in its original form, is subjected to three main deficits. The first relates to the desired operation requirements. For some applications, given a graph \(G(V,E)\), we want to process it and be left with a representation for each node \(v \in V\). The original paths-based approach, as described by the authors in [ABLY19], expands the input tree into paths, performs a sequential processing over them, and then averages the resulted paths encodings to yield a single distributed representation for the entire tree. By so, this approach, as is, does not meet the above-mentioned requirement to produce node-level representations.

The second deficit regards the potential loss of structural information. After the expansion of the graph into paths, a certain node can occur in more than one path. Hence, during the sequential processing of the paths, information of a certain node participating in several paths couldn’t be propagated across these paths containing it. Moreover, the model has no way to know the occurrences belong to the same node. In other words, the model is neither supplied with any information about the relation between these occurrences nor has the suitable architecture primitives to grasp these relations. This decoupling may result in loss of the overall contextual information of a node and the entire structure. Figure 3.7 illustrates this deficit.

The third deficit regards paths sampling. In Section 3.4 we discuss the importance of paths sampling for overfitting avoidance in the context of small datasets. If we were to apply multiple stacked layers of the paths-based encoder, the current design doesn’t support using different sample paths set per layer. Instead, the same sampled paths set would have to be fed to all layers, as the encodings of the concrete paths are the ones that are being produced from a layer and being fed to the next one. In Section 6.5.2 we discuss the potential benefits of per-layer paths set sampling.

### 3.3.2 Intuitive description

The original above-mentioned paths-based approach expands the input graph into a set of paths. Our expand-collapse method adopt this expanding technique as the first stage of the algorithm. To deal with the three described issues of the original paths-based approach, in our improvement, the expand stage is followed by an additional collapse stage. This additional stage is responsible for re-constructing the graph, by calculating concrete node representations. This is achieved by folding each node’s occurrences into a single vector in \(\mathcal{R}^d\). This method is illustrated in Figure 3.6.

The eventual generation of node representations meets the design requirements, and by so resolves the first mentioned deficit. The aggregation of the node’s occurrences into
a single node representation conserves the structural information (addressing the second deficit). In fact, the collapse stage can be seen as a means to inform the model about the association between the scattered occurrences and their representative node. Moreover, for the case of multiple stacked expand-collapse layers, the information about a node participating in several paths could be propagated across these paths. After the first layer, a node representation can theoretically contain information from all the paths it occurs in. Thus, at the second layer, after re-expanding the node representations (generated by the first layer) into paths, an occurrence of a node in some path can contain information from all the other paths this node occurs in (and not only from this certain path). This opposes to stacked layers of the original paths-based version (without the collapse stage), where the sequential encoder is just applied multiple times over each expanded path separately without information flow across paths. The two cases are illustrated in Figure 3.7.

Additionally, per-layer paths-set sampling becomes possible in the revised algorithm, as only the node representations are being passed between the layers, which allows per-layer re-expansion of the graph into a set of paths that is potentially different from layer to layer. Later, in Section 4.3 we show the expand-collapse approach’s superiority over the original expand-only approach for the cases of AST and CFG encoding.

3.3.3 Formalization

Input terminology Let \( G \) be the set of all graphs \( G = (V^G, E^G) \), where we denote \( V^G = \{v^G_1, ..., v^G_{|V|}\} \). For \( G \in \mathcal{G} \) we denote \( \mathcal{PATH}(G) \) to be the set of all paths in \( G \) by \( \mathcal{PATH}(G) \triangleq \{(v^G_{k_1}, ..., v^G_{k_l}) | l \in \mathcal{N}, \forall i \in [l] : k_i \in |[V]|, \forall i \in [l-1] : (k_i, k_{i+1}) \in E\} \). Also let...
(a) When the collapse step is not involved (in accordance with the original Code2Seq), multi-layer encoding can be performed by consequently applying sequential layers stacked on top of each other over the expanded graph. In such way, there isn’t information flow between the two distinct paths. Here, the purple node, for example, participates in two paths, but the information from its occurrence in the upper path doesn’t propagate to its occurrence in the lower path and vice versa. Thus, for any arbitrary $k$ there is no information flow between nodes $C$ and $D$, even after $k$ layers.

(b) After collapse #1, the new representation of the purple node is being determined with respect to its participation in paths (i) and (ii). Thus, after expand #2, the information about the participation of the purple node in path (ii) is being propagated into the encoding of this node in the expanded path (i). Then, after sequential #2, the information of the purple node from path (ii) propagated along path (i) and vice versa. Finally, that forms a complete information flow path between nodes $C$ and $D$.

Figure 3.7: Illustration of the benefit of the collapse stage for cross-path information flow. Comparison of multi-layer encoders with and without the collapse stage.
\( \mathcal{P} \mathcal{A} \mathcal{T} \mathcal{H}(\mathcal{G}) \triangleq \{ \mathcal{P} \mathcal{A} \mathcal{T} \mathcal{H}(G) | G \in \mathcal{G} \} \). For a graph \( G \in \mathcal{G} \) we denote \( o^G_{\text{embed}} : V^G \rightarrow \mathcal{R}^d \) to be a mapping from a node to some distributed representation of it.

**Operators** The *expand-collapse* graph encoding framework is a general algorithm that depends on the definition of three operators: \( o_{\text{expand}} \), \( o_{\text{sequential}} \) and \( o_{\text{collapse}} \).

- \( o_{\text{expand}} : \mathcal{G} \rightarrow \mathcal{P}(\mathcal{P} \mathcal{A} \mathcal{T} \mathcal{H}(\mathcal{G})) \). For a given graph \( G \), the *expand* operator yields a subset of \( \mathcal{P} \mathcal{A} \mathcal{T} \mathcal{H}(G) \). That is, it expands the graph into some set of paths. Note \( o_{\text{expand}} \) is free to choose any subset out of the entire \( \mathcal{P} \mathcal{A} \mathcal{T} \mathcal{H}(G) \) for every \( G \in \mathcal{G} \). We only require the target set to be finite, that is \( \forall G \in \mathcal{G} : |o_{\text{expand}}(G)| < \infty \) to theoretically ensure the algorithm’s completeness. Of course, practical implementations would meet stricter complexity requirements.

- \( o_{\text{sequential}} : \langle \mathcal{R}^d, ..., \mathcal{R}^d \rangle \rightarrow \langle \mathcal{R}^d, ..., \mathcal{R}^d \rangle \). For a given sequence of embedded vertices, the *sequential* operator yields a new distributed representation for each vertex occurrence along the given path. Practically, this operator is expected to perform some sort of data propagation of the input representations along the sequence. By so, the obtained *path-aware* representation of a vertex occurrence contains information of other vertices on the path it occurred in. In our applications, this operator is implemented using DNN primitives that include weights that are learnt during the training.

- \( o_{\text{collapse}} : \langle \mathcal{R}^d, ..., \mathcal{R}^d \rangle \rightarrow \mathcal{R}^d \). For a set of occurrences encodings of a certain node, it matches a single representation vector for this node.

### Algorithm 3.2 Expand-collapse graph encoder

**Input:** \( G \in \mathcal{G}; o^G_{\text{embed}}(\cdot) \in (\mathcal{R}^d)^{|V^G|} \)

**Output:** \( \langle f_1, ..., f_{|V^G|} \rangle \in (\mathcal{R}^d)^{|V^G|} \)

1: \( \text{paths} \leftarrow o_{\text{expand}}(G) \)
2: \( \langle e_1, ..., e_{|V^G|} \rangle \leftarrow \langle o^G_{\text{embed}}(v_1), ..., o^G_{\text{embed}}(v_{|V^G|}) \rangle \)
3: \( \text{embeddedPaths} \leftarrow \{ \{e_{k_1}, ..., e_{k_l}\} | \{v_{k_1}, ..., v_{k_l}\} \in \text{paths} \} \)
4: \( \text{encodedPaths} \leftarrow \{ o_{\text{sequential}}(p) | p \in \text{encodedPaths} \} \)
5: \( \langle f_1, ..., f_{|V^G|} \rangle \leftarrow \langle o_{\text{collapse}}(\{ o_k_i | p \in \text{encodedPaths}, o_{k_i} \in p \} | i \in [|V^G|]) \rangle \)
6: return \( \langle f_1, ..., f_{|V^G|} \rangle \)

### 3.3.4 Folding strategies

In the collapse stage, we explored several manners for the folding of the node’s occurrences back to a single representation for this node. The folding operator has to take variable number of occurrences’ encodings (each is a vector in \( \mathcal{R}^d \)), perform some kind of aggregation over them, and produce a single encoding vector in \( \mathcal{R}^d \). We tried applying mean, sum, general attention (where the attention’s query is a general learnt vector) and attention (with the node’s previous encoding as the query vector). Using general attention yielded the best results.
3.3.5 Applications in our work

We employ the expand-collapse encoder in two different parts of our work where a graph has to be processed. The first usage is in the AST paths-based micro operator, as described in Section 3.2.3. In that case \( o_{\text{expand}} \) returns the leaf-to-leaf and leaf-to-root paths in the given sub-AST. The second is the CFG paths macro operator, as described in Section 3.2.5. In that case \( o_{\text{expand}} \) returns all the CFG paths from the procedure’s entry point to its exit point without visiting twice in a loop’s body (re-visit is allowed only for loops’ condition, as for some loop types it is required to visit their condition twice to exit). We discuss the theoretical and practical complexity implications of our \( o_{\text{expand}} \) operators later in Section 6.4. For both of our applications we show in Section 4.3 their superiority over the simpler set-of-paths algorithm (expand only, without collapse).

3.4 Sampling as a technique for data-augmentation and regularization

3.4.1 Overview

In this work, we explore rich code representations inspired by our task characteristics. On the one hand, expressive representations are designed to theoretically contribute to the model’s capability of grasping complex relations in the input procedures. On the other hand, their expressive power also makes them prone to overfitting, as it empowers the model with memorizing capacity. When trained over little amounts of data, such model may be able to simply memorize the entire training data more easily than models with less-expressive representations. We further discuss the expressiveness tradeoff later in Sections 6.5.2 and 6.5.2. Unfortunately, as pointed out in Section 3.1.4, there is not much training data accessible for the LogVar task; certainly not enough to effectively train such over-parameterized models.

One common way to avoid overfitting is to use regularization techniques. Dropout [SHK+14] is a commonly-known example for regularization in DNNs. In this section, we describe two representation-specific regularization techniques that we devised and integrated in our model. The first, described in Section 3.4.2, relates to intentional identifiers embedding obfuscation. The second, in Section 3.4.3, relates to sampling of AST paths.

3.4.2 Random sub-identifiers obfuscation

Motivation for distinguishable out-of-vocabulary (OOV) sub-identifiers In software source-code, it is a common practice having variables names that are project-unique, module-unique, or even procedure-unique. In many cases such identifier can be formed as an abbreviation or a portmanteau of other software components’ names.
In our data, it results in 23% and 25% of the sub-identifiers being out-of-vocabulary in the training and evaluation sets accordingly.

Both in NLP and in current code representation approaches, a common practice to handle these out-of-vocabulary words, is having a dedicated OOV learnable embedding vector, that is used whenever an out-of-vocabulary word is being fed to the NN. As a result, when multiple different out-of-vocabulary words are being fed together to the model, they all get the same embedding, and thus could not be effectively distinguishable one from another by the model. In the context of source code, identifiers might be crucial to distinguish between, as they might hold fundamental semantic information about the program. Unfortunately, as the identifiers' statistics reveal, intermingling dissimilar different sub-identifiers is inevitable using today’s OOVs handling technique.

In contrast, in average 6 (9%) among the procedure’s sub-identifiers are both unknown and participate in multiple identifiers. The joint participation of a sub-identifier in various identifiers may hold information about the relation between these identifiers. Thus, assigning the same embedding to all unknown sub-identifiers not only incurs loss of information about distinctiveness, but also leads to disassociation of related identifiers.

We propose a new manner to handle these OOV sub-identifiers, that allows the model to distinguish between different ones and, at the same time, unify unknown sub-identifiers that recur in several identifiers throughout the procedure.

**Motivation for partial sub-identifiers obfuscation**

Common semantic characteristics can be shared across variables defined and used in different procedures or different places within a procedure. We aim to design a model that is able to grasp these hidden semantic features. Such features can be reflected in the code both by the variable’s name and also by hidden patterns in the contextual structure in which the variable lays.

As pointed out above, the name is subjected to variability across and within implementations, and sometimes isn’t even listed in the vocabulary at all. For example, let’s consider a variable that stores a counter and being incremented in a loop. It has the characteristic of being a counter. Identifying this characteristic might help catching higher-order code patterns. However, its name can vary across implementations, depending on the kind of elements it counts or on the purpose of the counting (for example: checking whether some threshold has exceeded or calculating the expected time for a process to run). Fortunately, although the variable name could be unknown, the counting characteristic of this variable might potentially be identified using a structural pattern that includes the variable’s occurrences.

Hence, the identifiers themselves shouldn’t be considered as a principal reliable modeling feature for a variable. Instead, we would ideally prefer a variable modeling which is mostly based on its underlying contextual structure patterns, while regarding the identifiers essentially as a supplementary hint that is sometimes available for the
Current embedding methods theoretically allow the model to store any information at the embedded words during training. Unfortunately, for small datasets, this can assist the model memorizing the training examples using the architectural learnt weights. Such memorization may result in a model that relies on shallow features, and thus prone to overfit.

To prevent the model from developing any sort of dependency on certain variable names during training, we employ an ad-hoc regularization technique over the identifiers embeddings. We augment the training examples by randomly treating some portion of the known identifiers as unknown. In each recurrence of an example during training, different identifiers would be obfuscated, and different embeddings would be assigned to these obfuscated identifiers (out of a definite learnable set of OOV embeddings, which is also used for the identifiers that are unknown in the first place). This way, the model struggles to store example-specific information on variable names embeddings straightforwardly (like information that indicates that a variable is logged). The goal is to force the model to propagate the variable’s information throughout the code’s structural paths, independently of its specific name. In such manner, better learning generalization can be potentially achieved.

**Technical description** Here we suggest and examine an improvement that aims to address the two difficulties stated above. Firstly, instead of having a single OOV embedding vector for unknown sub-identifiers, we learn 64 different OOV embeddings \{OOV₁,...,OOV₆₄\}. When an example is fed to the model (both during training and inference), we randomly map the set of unknown sub-identifiers to the set of OOVs embeddings, such that: (i) within a certain example, two distinct sub-identifiers are mapped to different OOV embeddings; (ii) within a certain example, all of the occurrences of a sub-identifier are mapped to the same OOV embedding, even across different identifiers; (iii) the mapping randomization is example-dependant - that is, a certain sub-identifier might be assigned with different OOV embeddings across different examples it participates in; (iv) the mapping randomization is batch-dependant - that is, a certain sub-identifier in a certain example might be assigned with different OOV embeddings across different batches the example participates in.

Additionally, during training only, we treat 20% of the known identifiers as unknown. This augmentation is expected to compensate for the inevitable gap between the ratio of OOVs in the training set compared to the evaluation set.

As desired, this technique supplies the model with the necessary information to distinguish between different OOV sub-identifiers, and the information about the participation of a common sub-identifier in multiple identifiers. The per-batch mapping randomization is intended to hinder the model from overfitting.
3.4.3 Paths sampling

Motivation Our suggested hierarchic representation decomposes the entire procedure’s code into smaller and simpler elements. Each hierarchic encoding level has at least as expressive power as the non-hierarchic encoding approaches, while its responsibility is confined to less complex data comprised of a more restrained structure.

More specifically, let’s first consider the paths-based AST micro encoder. Its input, a top-level expression, is typically small: in average, its textual length is 9, and its sub-AST has 10 nodes and 7.8 leaf-to-leaf paths of average length of 6 (±3.8). Additionally, it is comprised only by well-defined expression grammar parsing rules (leaving out cross-statement rules). Hence, the variance among the single top-level expressions distribution is significantly smaller (by construction) than the variance of entire procedures distribution (which contains the first). The micro encoder’s sole responsibility is for a standalone top-level expression. Compared to a flat representation, where the entire textual code is being processed by a sequential model, our micro encoder particularly contains the same sequential model and even extra power (attention is used for the folding at the collapse stage), but is processing significantly simpler data elements containing less information. Moreover, the same information of the top-level expression is being delivered to the micro encoder after being pre-parsed as an AST. This might reduce even more the required neural resources of the model, leaving more learning capacity available.

The exact same claim also stands for the macro encoder. Overall, a more powerful model learns less complex data. In such cases, when trained on small datasets, the model is prone to overfit. This makes sense, as over-parameterized models can easily memorize small amounts of simply structured data.

To avoid overfitting in the context of the AST micro encoder, we slightly augment the data during training. That way, in each recurrence of a training example, its shallow features would be different enough to hinder the model from unintelligently memorizing them. We achieve this aim by sampling a subset from the entire set of AST paths. Where in each train example recurrence a different subset is being sampled. Therefore, from a shallow point of view, the input data seems different across repetitions. However, the set of all leaf-to-leaf paths contains information redundancy, and hence the remaining sampled paths subset could potentially be sufficient to convey deeper essential meanings of the structure. These persisting profound features shared across repetitions can be theoretically more suitable for better generalization.

The paths sampling concept is comparable to DropAttention suggested by Zehui et al. [ZLH19]. While DropAttention can be regarded as random removal of an edge, we randomly remove an entire path. It doesn’t necessarily mean that all of the edges on this path are being removed altogether, as they might participate in other paths which haven’t been removed.
During training, we sample 80% of the procedure’s paths. This sampling ratio has been determined by a trial-and-error exploration over the validation set. For each train batch an example participates in, another sample is taken for this example. During evaluation and inference the entire paths set is being processed, without drawing any sample. We chose to cancel the sampling at inference both for determinism of the prediction technique on a trained model (to obtain consistent results across experiments), and also for having full utilization of the evaluated data. The sampling is irrelevant during inference, as its aim is to prevent overfitting.

We also tried another stochastic sampling method where the sample ratio itself is first being drawn from some distribution, just before taking each set-of-paths sample. In this approach, the sampling ratio is defined to be a random variable over the range (0, 1]. Practically, in our explorations, we used the following RV: \( \text{SamplingRatio} \triangleq \min(\max(S, \frac{1}{\#\text{paths}}), 1) \) where \( S \) is a random variable with a significant density within the interval (0, 1]; for example, \( S \sim \mathcal{N}(\mu = 0.8, \sigma = 0.15) \). For each example we first draw a realization \( \text{ratio} \) from \( \text{SamplingRatio} \), and only then draw a set-of-paths sample according to this sampled \( \text{ratio} \). This means that, apart from having an individual paths subset per recurrence of some example during training, the subset’s size varies as well. The motivation for varying the size of the paths subsets is to prevent the model from getting accustomed to certain (and unrealistic) amount or ratio of node occurrences, as upon inference we feed it with a different ratio. Empirically, the results of this approach on our data added only a minor advantage compared to the simpler fixed ratio sampling approach. We mention it here as it might be more relevant for other datasets. Additionally, we tried gradually changing the distribution parameters during the training (both ascending and descending), but it was not found to add value over the non-dynamic sampling method.

When using multiple expand-collapse encoding layers stacked on top of each other, the paths sampling can be performed per-layer. That is, for each individual layer we can draw a separate paths subset sample. The motivation for per-layer sampling is increasing the amount of input data to the training, while still hindering each concrete layer from overfitting the training data. In this case, even a smaller sample for each layer could be sufficient, without overall loss of information. Although we haven’t explored this variant in our work, we believe it should be considered in future works.

Although we only used the paths sampling over the top-level expressions’ sub-ASTs in this work, the same sampling technique could be applied over the CFG paths in a similar manner.

Comparison to paths sampling applied by Code2Seq In the original paths-based Code2Seq encoder [ALBY19], Alon et al. also incorporated paths sampling practice. They reported sampling of 1K paths per procedure during pre-processing and sampling of 200 paths during data loading. Code2Seq extracts the set of leaf-to-leaf
paths over the entire procedure’s AST, which turns out to be too big to fit in nowadays accelerated computing hardware when working with standard batch sizes. Thus, while we use sampling to avoid overfit, in their case, sampling is used as a mean to overcome computational complexity limitations. Note that, unlike Code2Seq, we are not obligated to sample for our algorithm to run on the same hardware, as the set of paths is significantly smaller in the first place.

Alon’s Code2Seq sampled sets have a fixed absolute size determined with respect to the hardware limitations, without taking into account the size of the original paths set. This might lead to over-sampling for long procedures. Later, in Section 6.6 we discuss loss of essential information caused by the original Code2Seq over-sampling on our data. In contrast, we sample by ratio with respect to the original paths set size. It means that in our approach the size of the sampled paths set is relative to the size of the original set. Particularly, even paths sets of small absolute size in the first place are sampled.

Additionally, note that the privilege to revoke sampling on inference is not possible in the original Code2Seq approach, as the sampling is required there anyway due to unavoidable computational complexity considerations.
Chapter 4

Evaluation

First, in Section 4.1 we describe our research methodology. Then, in Section 4.2, we present the results for our method and for the compared baselines over our benchmarks. Finally, in Section 4.3, we conduct an ablation study, where we justify the various elements of our solution and we explore different aspects of the task.

4.1 Experiments setting

4.1.1 Benchmarks

We evaluated the examined methods over three datasets that include code snippets written in Java. We present them below.

Java-small and java-med

The first two datasets are based on java-small and java-med that were presented and used by Alon et al. in their papers [ABLY19, ASLY20]. These datasets have been used to address other code-related tasks like predicting a name for a code snippet and code synthesis. Table 4.1 presents quantitative statistics for the datasets.

Apache-java

We collected the third dataset from open-source well-maintained Java projects, publicly listed in the Apache organization’s GitHub profile. It will be referred as apache-java. We chose to focus on Apache’s projects as they are used as benchmarks of previous log-related works [LXL+19, LLCS21]. As we compare our approach to the work of Li et al. [LCS20], apache-java extends their benchmark; namely, it includes all the projects they evaluated on and more. To collect this dataset, we automatically selected all Apache’s repositories with highest ranking with respect to popularity, number of commits, number of contributors, number of forks, number of releases, etc..
Table 4.1: Benchmarks statistics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Repositories</th>
<th>#Log statements</th>
<th>#Filtered log stmts</th>
</tr>
</thead>
<tbody>
<tr>
<td>java-small</td>
<td>10</td>
<td>29,717</td>
<td>5,092</td>
</tr>
<tr>
<td>java-med</td>
<td>276</td>
<td>135,448</td>
<td>23,178</td>
</tr>
<tr>
<td>apache-java</td>
<td>190</td>
<td>274,344</td>
<td>56,706</td>
</tr>
</tbody>
</table>

Extracting LogVar examples from datasets

Given a Java method, we first look for logging operations in it. These are method calls, where the method name is one of: {info, warning, error, fatal, debug, trace}, and the call scope name contains the word "log" (e.g: AppLogger.info(...)). Then, we extract the used symbols in the logged expression. More specifically, we search for local variables that are defined in the scope of the method, parameters of the method’s declaration, scoped parameter of a catch clause or class fields that are used in other places in the procedure. We filter out logging operations without symbols holding these requirements or with more than four distinct valid symbols. For example, for the following logging operation Logger.warning("Missing required account authorization for application " + app.getInfoForAccount(accountId)) the extracted set of target symbols is {app, accountId}, given that they both hold the above-mentioned requirements.

Data folding

First, we split each dataset into two sets: train and test, in a 2:1 ratio. During model development we used 5-fold cross-validation over the training set to evaluate our models. The results reported here are for models trained over the entire training set and evaluated over the test set, which was held-out during development.

The dataset is divided into sets / folds by projects; Namely, each project is entirely contained in exactly one fold. This is done due to two main reasons: (i) avoiding data leakage between folds, as a certain project may contain similar examples; (ii) examine the model’s ability to learn from a certain codebase, and then predict over examples from possibly new projects, as this is the most reasonable use-case for such model. This folding-by-project is adopted from Alon et al. [ABLY19, ASLY20].

We used only java-small during models development, so java-med and apache-java can be regarded as our test sets. That way, we test our method instead of testing the trained model itself.

Filtering

The original dataset contains many (>50%) naive examples, like logging the exception variable that has just been caught in the catch clause or logging the single procedure’s parameter at its entry. These examples can be easily learnt and predicted by much
simpler models. As a previous work already exists, in this work we aim to improve it by extending the range of cases it can solve. Therefore, each original dataset is filtered. The filter is designed to take these basic cases out. Our filter is composed of the following predicates: (i) logging statement logs with at least 1 variable and at most 4; (ii) distance from the logged variable definition to the logging statement is at least 6 lines of code; (iii) there are at least 5 symbols (local variables / method’s parameters / class fields) accessible in the method; (iv) method has at least 4 statements.

4.1.2 Metrics

The expected output of our task is a set of variables. Given the ground-truth variables set $V_{gt}$ and the predicted variables set $V_{pred}$, we consider two metrics: (i) $F1$ score, where the elements used for precision/recall calculation are the variables. This metric scores accordingly predictions that are partially similar to the ground-truth set but not exactly correct (where $V_{gt} \cap V_{pred} \neq \emptyset$ but also $V_{gt} \Delta V_{pred} \neq \emptyset$); and (ii) accuracy which gives score of 1 to exact sets (un-ordered) match and zero otherwise, as $I[V_{gt}=V_{pred}]$.

4.1.3 Baselines

We compared our results over the benchmarks to seven different baseline approaches. Here we describe each one of them. The neural-based approaches differs from one another by the manner in which they model the distributed representation of the given input procedure.

**Flat tokens sequence representation**

The flat (or tokens sequence) representation is the most elementary approach we explore. As opposed to other more-advanced methods, it doesn’t involve any intentional modeling for the code’s underlying structure. We present this method in more detail in Section 2.2. In their work, Liu et al. [LXL+19] have employed this method over the LogVar task, and therefore we regard it as the main baseline for our work. Its results are presented in row 1 on table 4.3. We used an LSTM [HS97] as the sequential model, as originally done by Liu et al. Using two LSTM layers produced the nest results over all explored benchmarks (we tried with 1-4 layers).

To adapt the method to the supervised LogVar task, the entire log statement is replaced by a special $<$LOG MASK$>$ token. The masking provides the model with the information about the log’s location within the code and hides the ground-truth labels (the actual variables that are being used there).

**Leaf-to-leaf AST paths representation**

This baseline is based on Code2Seq [ABLY19], as described in Section 2.4. Here we use this method as a baseline when applied over the entire method’s AST. Its results are
presented in row 2 on table 4.3. Note that the sub-AST of the log statement is being replaced with a single AST node of a special type \texttt{<LOG\_MASK>} similarly to the masking performed by the flat representation.

A follow-up work [ASLY20] to \texttt{Code2Seq} proposed \texttt{GEN}, which aims to automatically complete some missing code in a given partial procedure. Similarly to \texttt{Code2Seq}, \texttt{GEN} is also based on paths in the procedure’s AST. In fact, for the case of \texttt{LogVar} these two approaches are almost equivalent. Predicting a sequence of variables for a given logging statement can be regarded as completing a sub-AST of the method call \texttt{logger.log(...)}. Hence, it can be seen as a degenerated flat \texttt{GEN} completion site. For such cases, \texttt{GEN}’s architecture is similar to \texttt{Code2Seq}, except for the \textit{copy attention} of the target symbol. In its output generator, \texttt{Code2Seq} predicts the produced output from the fixed vocabulary, while \texttt{GEN} adds the dynamic vocabulary from the input. It means that \texttt{GEN} can predict a name that does not present in the fixed vocabulary. This technique is commonly known as the \textit{copy mechanism} [GLLL16]. Here, we evaluate over the option with the copy mechanism, as all of the other baselines also use it. Although adding the copy mechanism increases the power of the original \texttt{Code2Seq}, throughout this work we still refer this model as ”\texttt{Code2Seq with copy}” and sometimes even as \texttt{Code2Seq} rather than \texttt{GEN}, because \texttt{GEN} is a more extensive model that is generally intended for tasks with richer target structure.

\textbf{AST TreeLSTM representation}

\texttt{Code2Seq} represented tree-structured data by extracting it into paths. Another manner to do so is by using the \textit{TreeLSTM} [TSM15]. Similarly to applying \textit{LSTM} units over a linear sequence, \textit{TreeLSTM} applies \textit{LSTM} cells over a tree from the leaves towards the root of the tree. More specifically, when applying the tool\textit{TreeLSTM} cell over a node, the already-calculated states of its immediate children is being aggregated (summed up for example) and being inserted into a classic \textit{LSTM} cell. We use this method as a baseline when applied over the entire procedure’s AST. Its results are presented in row 3 on table 4.3.

\textbf{4.1.4 Neural models’ architecture}

Each one of the presented neural-based models provides a technique to encode the input procedure while masking the target logging statement. After the encoding stage the model takes the produced encoding vectors and feed it to a decoder. The decoder is responsible to produce a sequence of output symbols. For the decoder, we use two \textit{LSTM} layers with attention over the encoder output vectors. This is a common decoding strategy, suggested by Loung et al. [LPM15] and used by neural machine translation (NMT) tools like \textit{OpenNMT} [KKD+17]. As the \texttt{LogVar} output is expected to be a set of symbols, for each output word the decoder points (using the pointing mechanism [VFJ15]) on a symbol, out of a dynamic set of symbols encodings. This
technique is commonly known as the *copying mechanism* [GLLL16]. The symbols encodings are the encoder outputs for identifier tokens or AST leaves that are parsed as symbols from the point of view of the programming language’s grammar. Multiple occurrences of the same symbol are being summed up, in accordance with the dynamic vocabulary copy mechanism technique applied by *OpenNMT*. We use the same decoding architecture for all of the presented neural models.

### 4.1.5 Identifier encoding

All explored methods require embedding of the program’s identifiers. Identifiers can be variable names, a type name or any name of the program’s symbols. It is customary among today’s code models to tokenize the identifier into its sub-parts. To keep the embedding technique uniform across all methods, in our implementation, each identifier is encoded by applying a sequential model other its sub-identifiers. The number of sub-parts in an average identifier is 2 (±1.1). For example, the identifier `getUserId` is tokenized into three sub-tokens: (i) `get`, (ii) `user`, (iii) `id`. Each one of them is being embedded using trainable embeddings for the vocabulary of sub-identifiers. Then, an LSTM layer processes this sequence of sub-identifiers. The output encodings forms the eventual identifier encoding. Occurrences of an identifier (as an AST terminal or as a token in a flat represented code expression) is encoded using this pre-calculated identifier encoding.

### 4.1.6 Implementation

We implemented an end-to-end system that covers all explored approaches and all the involved pre- and post- processing and evaluating stages. This strategy helps unifying all non method-specific components like pre-processing, normalizations, optimization method choices (optimizer, learning rate scheduling, effective batch size), weights initialization, sampling seeding, decoder choices, target vocabulary copy-mechanism, evaluation metrics, and other neural-network primitives choices. By so, it makes the comparisons between the pure conceptual methods less dependant on implementation details, and hence more reliable. [Technically, to run an experiment, the user executes the system with the desired parameters (describing the selected method to use), and the system incorporates the relevant modules within the same general framework.]

To implement the hierarchic method we had to cope with non-trivial technical batching difficulties. To overcome these difficulties, we implemented a package called `TensorsDataClass`. See more details in Section A.1.

We used PyTorch together with pytorch-scatter as the neural-network backend. We used RocksDB embedded persistent key-value store library to support fast random access for loading shuffled batches of pre-processed data during training.

We haven’t yet published the implementation’s source-code, as we still aim for publication of our work. Hence, for the time being, we kept our full implementation
in a private repository. After publication, our implementation would become available at github.com/eladn/ndfa. All the reported results are fully reproducible with our implementation.

4.1.7 Hyper-parameters and neural-network primitives

We used today’s common optimization and regularizations techniques with their typical values. Apart for the learning rate, none of the hyper-parameters has been optimized. We chose to use the same hyper-parameters for all experiments to ensure general stability for the entire experimental set and to avoid overfitting on the one hand, and while keeping an equal and fair chance for each experiment on the other hand. As we conducted a considerable amount of experiments, to save computational resources, the learning rate has been adapted to be about the maximum possible (over the simplest flat tokens sequence baseline) so that the network would converge and the training would be as fast as possible.

For all encoded elements (sub-identifiers, identifiers, tokens, AST nodes, CFG nodes, symbols) we used embedding dimension of \( d = 256 \). We used the AdamW optimizer [LH19] with learning rate of 3e-4, weight decay of 1e-4, learning rate decay of 0.02 (scheduled epoch-wise), gradient normalization of 0.5, and an effective batch size of 64 (the number of examples whose gradients are accumulated for a single optimizer step). During training, we applied dropouts (after linear layers) with drop rate of 0.3. We also used the LeakyReLU activation after linear layers. We used PyTorch’s default weights initialization and set the random number generator seed to 0.

For the AST paths sampling we used the constant rate of 0.8. We used the obfuscation rate of 0.3 for the sub-identifiers embeddings. To avoid overfitting, we limited the sub-identifiers vocabulary to be the minimum between total size of 5K words or when the least frequent word has at least 40 occurrences (different datasets reach different limitations).

4.1.8 Hardware

We executed our experiments over a single AWS p3.2xlarge instance with a Tesla V100 (16GB memory) GPU accelerator. We used mini-batches of size 32.

4.2 Results

Our principal method is the hierarchical approach with the expand-collapse over AST (both leaf-to-leaf and leaf-to-root paths) as micro operator and with the expand-collapse over CFG as the macro operator. It includes random AST paths sampling (for the micro operator), and random OOV sub-identifiers obfuscation. The scores of our principal method are presented in Table 4.2. The results of the baseline approaches (described in Section 4.1.3) are presented in Table 4.3.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>F1 (P./R.)</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java-Small</td>
<td>0.57 (0.61/0.53)</td>
<td>0.40</td>
</tr>
<tr>
<td>Java-Medium</td>
<td>0.66 (0.68/0.64)</td>
<td>0.50</td>
</tr>
<tr>
<td>Apache Projects</td>
<td>0.71 (0.75/0.66)</td>
<td>0.57</td>
</tr>
</tbody>
</table>

Table 4.2: Results for our hierarchic approach with leaf-to-leaf & leaf-to-root (combined) AST paths micro operator and full CFG paths macro operator over our benchmarks.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Java-Small</th>
<th>Java-Med</th>
<th>Apache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Linear tokens sequence</td>
<td>0.47 0.33</td>
<td>0.63 0.46</td>
<td>0.68 0.54</td>
</tr>
<tr>
<td>(with symbols copy)</td>
<td>0.53/0.42</td>
<td>0.65/0.61</td>
<td>0.73/0.65</td>
</tr>
<tr>
<td>2 AST leaf-to-leaf paths</td>
<td>0.41 0.29</td>
<td>0.51 0.37</td>
<td>0.58 0.43</td>
</tr>
<tr>
<td>(GEN or Code2Seq w. symbols copy)</td>
<td>0.49/0.36</td>
<td>0.57/0.46</td>
<td>0.63/0.53</td>
</tr>
<tr>
<td>3 AST TreeLSTM</td>
<td>0.36 0.25</td>
<td>0.41 0.30</td>
<td>0.46 0.33</td>
</tr>
<tr>
<td></td>
<td>0.43/0.31</td>
<td>0.49/0.36</td>
<td>0.49/0.44</td>
</tr>
<tr>
<td>4 AST GNN (GCN x8)</td>
<td>0.41 0.30</td>
<td>0.49 0.33</td>
<td>0.53 0.39</td>
</tr>
<tr>
<td></td>
<td>0.50/0.35</td>
<td>0.51/0.47</td>
<td>0.58/0.48</td>
</tr>
<tr>
<td>5 AST GNN (GAT x8)</td>
<td>0.41 0.29</td>
<td>0.50 0.35</td>
<td>0.54 0.40</td>
</tr>
<tr>
<td></td>
<td>0.47/0.37</td>
<td>0.53/0.47</td>
<td>0.58/0.50</td>
</tr>
<tr>
<td>6 AST GNN (GATv2 x8)</td>
<td>0.42 0.30</td>
<td>0.49 0.35</td>
<td>0.54 0.41</td>
</tr>
<tr>
<td></td>
<td>0.50/0.36</td>
<td>0.53/0.45</td>
<td>0.59/0.50</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison with non-hierarchic (entire method) baselines.

Our method outperforms all baseline approaches with a significant gap of 10%, 3% and 3% in F1 score and 7%, 4% and 3% in accuracy over the three tested benchmarks accordingly. Additionally, among the evaluated baselines, the degenerated flat representation yields the highest scores. Unfortunately, all AST-based baseline approaches (including Code2Seq) have shown very poor performance comparing to the flat model (over all benchmarks). In fact, our method is the only structural-aware model to surpass the flat model.

### 4.2.1 Scalability

One of the notable advantages of the hierarchic framework is its support for long procedures. In average, our the hierarchic model (with our default micro and macro operators) processes 3,141 tokens per examples, where for comparison, Code2Seq requires 123.5K processed tokens in average (both without incorporating sampling). This is due to the quadratic number of leaf-to-leaf paths (w.r.t the number of AST leaves), which leads to 9.5K paths in an average procedure’s AST. As in today’s hardware it is
possible to process only about 300 sequences for reasonable batch sizes, long procedures require over-sampling of AST paths, which causes loss of essential information. The omission of the AST paths that cross top-level expressions in the hierarchic model leads to massive reduction of 96% of these leaf-to-leaf paths. As it turns out from the results, these cross-statement AST paths are not required to achieve even higher results.

In addition, when comparing to the flat representation, the average length of a path in a top-level expression’s sub-AST is 6 (±3.8) and the average length of a path in the CFG is 9, while the average length of a tokenized procedure is 283 (±213). As the sequential models are naturally more effective for shortest sequences, it makes the hierarchic model friendly for long procedures more than the flat model.

In Section 6.6 we further discuss the scalability concerns of the paths-based AST model and the potential power of the hierarchic model to overcome them.

### 4.3 Ablation study

In Section 3.1.10, we hypothesize about (a) the importance of the code’s structural information for the solution of the LogVar task; and (b) the ability of our hierarchic model to benefit from some sort of structural information better than other models. The results from the previous section support the second hypothesis. To further address both of these hypotheses, we conduct here a comprehensive set of ablation experiments. By modifying the examined models, each time in a different manner, we get a finer perspective over both the models and the problem. These glimpses allow us to figure out to what extent different components of the various methods contribute to the power of the models to express any underlying structures of the code, and which parts of the code’s structure are actually needed by the LogVar task.

The ablations of the flat representation would show that its success was with the aid of some of the structural information that is embedded in the textual code and the flat representation managed to grasp, thus supporting hypothesis (a). The ablations of the hierarchical model show that intentionally deducting structural modeling from the hierarchic model hurts the performance. Thus, supporting hypothesis (a).

As we later explain in Section 6.7, the hierarchic framework helps exploring the contribution of the structure in a finer resolution, by separating the modeling of the local and the global structures.

Specifically, we raise four distinct and viable explanations for the reason that the baselines with rich structural representations have not dominated over the flat model: (i) structural information is not needed at all to solve the LogVar task; (ii) the simplest flat model manages to infer all the required structural information for solving the LogVar task; (iii) none of the explored models possess enough expressive power to take advantage of the required structural information / there are not enough structural features in the data; (iv) our training datasets are too little for the suggested models for effective learning and to fully realize their potential expressive power.
Flat tokens sequence ablation

<table>
<thead>
<tr>
<th></th>
<th>Java-Small F1 Acc.</th>
<th>Java-Med F1 Acc.</th>
<th>Apache F1 Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P./R.</td>
<td>P./R.</td>
<td>P./R.</td>
</tr>
<tr>
<td>1 w/o copy attention @ decoder</td>
<td>0.20 0.13</td>
<td>0.32 0.20</td>
<td>0.46 0.34</td>
</tr>
<tr>
<td>(identifier embeddings target word)</td>
<td>0.23/0.17</td>
<td>0.34/0.30</td>
<td>0.48/0.44</td>
</tr>
<tr>
<td>2 Un-ordered set of tokens</td>
<td>0.35 0.24</td>
<td>0.40 0.29</td>
<td>0.43 0.31</td>
</tr>
<tr>
<td>3 Fragmentally shuffled</td>
<td>0.45 0.31</td>
<td>0.57 0.42</td>
<td>0.65 0.50</td>
</tr>
<tr>
<td>[Fragment] ∼ U[2, 5]</td>
<td>0.50/0.40</td>
<td>0.63/0.52</td>
<td>0.69/0.61</td>
</tr>
<tr>
<td>4 w/o keywords</td>
<td>0.47 0.33</td>
<td>0.62 0.47</td>
<td>0.69 0.54</td>
</tr>
<tr>
<td>5 w/o operators</td>
<td>0.47 0.31</td>
<td>0.62 0.48</td>
<td>0.68 0.54</td>
</tr>
<tr>
<td>6 w/o separators</td>
<td>0.44 0.32</td>
<td>0.60 0.44</td>
<td>0.66 0.51</td>
</tr>
<tr>
<td>7 Identifiers only (order kept)</td>
<td>0.42 0.31</td>
<td>0.54 0.38</td>
<td>0.58 0.44</td>
</tr>
<tr>
<td>(no keywords, operators, separators)</td>
<td>0.50/0.36</td>
<td>0.58/0.50</td>
<td>0.63/0.54</td>
</tr>
</tbody>
</table>

Table 4.4: Ablations for non-hierarchic (entire method) flat code tokens sequence approaches.

In the following subsections we describe experiments to empirically check the above-mentioned explanations.

### 4.3.1 AST leaves-only sequence

**Description:** In this model, given an AST (either of the entire procedure or the local sub-tree belonging to a concrete statement), only its leaves are being considered. The sequence of leaves (in infix order) are being processed by a sequential model and constitutes this AST’s representation.

<table>
<thead>
<tr>
<th></th>
<th>Java-Small F1 Acc.</th>
<th>Java-Med F1 Acc.</th>
<th>Apache F1 Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P./R.</td>
<td>P./R.</td>
<td>P./R.</td>
</tr>
<tr>
<td>1 AST leaf-to-root paths</td>
<td>0.41 0.29</td>
<td>0.49 0.35</td>
<td>0.57 0.43</td>
</tr>
<tr>
<td></td>
<td>0.47/0.36</td>
<td>0.55/0.45</td>
<td>0.61/0.54</td>
</tr>
<tr>
<td>2 AST leaves sequence only</td>
<td>0.47 0.34</td>
<td>0.57 0.43</td>
<td>0.63 0.48</td>
</tr>
<tr>
<td></td>
<td>0.55/0.41</td>
<td>0.63/0.51</td>
<td>0.67/0.60</td>
</tr>
<tr>
<td>3 Un-ordered set of AST leaves</td>
<td>0.35 0.25</td>
<td>0.42 0.30</td>
<td>0.46 0.33</td>
</tr>
<tr>
<td></td>
<td>0.43/0.30</td>
<td>0.48/0.38</td>
<td>0.52/0.41</td>
</tr>
</tbody>
</table>

Table 4.5: Ablations for non-hierarchic (entire method) AST paths-based approaches.
<table>
<thead>
<tr>
<th>Micro operator</th>
<th>Java-Small F1 Acc. P./R.</th>
<th>Java-Med F1 Acc. P./R.</th>
<th>Apache F1 Acc. P./R.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Flat tokens sequence</td>
<td>0.55/0.50 0.61/0.50</td>
<td>0.64/0.50 0.70/0.60</td>
<td>0.70 0.55 0.74/0.65</td>
</tr>
<tr>
<td>2 Un-ordered tokens set</td>
<td>0.50 0.37 0.59/0.43</td>
<td>0.62 0.47 0.69/0.56</td>
<td>0.66 0.52 0.72/0.60</td>
</tr>
<tr>
<td>3 AST TreeLSTM</td>
<td>0.48 0.31 0.48/0.47</td>
<td>0.58 0.43 0.62/0.55</td>
<td>0.65 0.51 0.68/0.61</td>
</tr>
<tr>
<td>4 AST GCN (x4)</td>
<td>0.51 0.34 0.52/0.49</td>
<td>0.63 0.46 0.64/0.62</td>
<td>0.68 0.48 0.70/0.66</td>
</tr>
<tr>
<td>5 AST GAT (x4)</td>
<td>0.49 0.32 0.50/0.48</td>
<td>0.61 0.42 0.60/0.61</td>
<td>0.68 0.53 0.70/0.65</td>
</tr>
<tr>
<td>6 AST GATv2 (x4)</td>
<td>0.48 0.31 0.49/0.46</td>
<td>0.62 0.45 0.62/0.61</td>
<td>0.65 0.51 0.68/0.62</td>
</tr>
</tbody>
</table>

Table 4.6: Hierarchic model micro ablations: different choices of micro operators, while having the same macro operator of all full CFG paths.

<table>
<thead>
<tr>
<th>AST paths micro operator ablations</th>
<th>Java-Small F1 Acc. P./R.</th>
<th>Java-Med F1 Acc. P./R.</th>
<th>Apache F1 Acc. P./R.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 AST leaves sequence only</td>
<td>0.55 0.38 0.59/0.51</td>
<td>0.64 0.49 0.69/0.60</td>
<td>0.69 0.54 0.73/0.64</td>
</tr>
<tr>
<td>2 AST leaves un-ordered set</td>
<td>0.51 0.36 0.58/0.45</td>
<td>0.62 0.47 0.69/0.56</td>
<td>0.66 0.52 0.72/0.62</td>
</tr>
<tr>
<td>3 AST leaf-to-leaf paths only</td>
<td>0.56 0.40 0.61/0.53</td>
<td>0.64 0.50 0.70/0.60</td>
<td>0.69 0.55 0.74/0.64</td>
</tr>
<tr>
<td>(no leaf-to-root paths)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 AST leaf-to-root paths only</td>
<td>0.56 0.38 0.58/0.53</td>
<td>0.64 0.50 0.70/0.58</td>
<td>0.70 0.55 0.73/0.66</td>
</tr>
<tr>
<td>(no leaf-to-leaf paths)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 AST paths w/o collapse</td>
<td>0.55 0.38 0.59/0.51</td>
<td>0.64 0.48 0.68/0.60</td>
<td>0.69 0.54 0.74/0.64</td>
</tr>
<tr>
<td>(keep paths expanded w/o node representations)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Ablations of the AST paths-based micro operator. Hierarchic model is used with all full CFG paths macro operator.
### Table 4.8: Hierarchic model macro ablations: different choices of macro operators, while all having the same micro operator of paths-based AST.

<table>
<thead>
<tr>
<th>Macro operator</th>
<th>Java-Small</th>
<th>Java-Med</th>
<th>Apache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F1 Acc.</td>
<td>F1 Acc.</td>
<td>F1 Acc.</td>
</tr>
<tr>
<td></td>
<td>P./R.</td>
<td>P./R.</td>
<td>P./R.</td>
</tr>
<tr>
<td>1 None (no information flow between CFG nodes)</td>
<td>0.47 0.33</td>
<td>0.55 0.40</td>
<td>0.58 0.44</td>
</tr>
<tr>
<td></td>
<td>0.53/0.42</td>
<td>0.58/0.51</td>
<td>0.62/0.55</td>
</tr>
<tr>
<td>2 Un-ordered set of CFG nodes</td>
<td>0.47 0.34</td>
<td>0.54 0.40</td>
<td>0.59 0.45</td>
</tr>
<tr>
<td></td>
<td>0.54/0.42</td>
<td>0.60/0.50</td>
<td>0.64/0.54</td>
</tr>
<tr>
<td>3 CFG nodes single sequence (ordered by textual appearance)</td>
<td>0.54 0.38</td>
<td>0.63 0.50</td>
<td>0.70 0.56</td>
</tr>
<tr>
<td></td>
<td>0.60/0.49</td>
<td>0.70/0.58</td>
<td>0.73/0.66</td>
</tr>
<tr>
<td>4 Upper pruned AST leaf-to-leaf paths</td>
<td>0.50 0.36</td>
<td>0.62 0.48</td>
<td>0.68 0.54</td>
</tr>
<tr>
<td></td>
<td>0.57/0.44</td>
<td>0.67/0.58</td>
<td>0.72/0.64</td>
</tr>
<tr>
<td>5 CFG GNN (GCN x4 layers)</td>
<td>0.49 0.34</td>
<td>0.59 0.44</td>
<td>0.63 0.50</td>
</tr>
<tr>
<td></td>
<td>0.54/0.44</td>
<td>0.65/0.54</td>
<td>0.68/0.59</td>
</tr>
<tr>
<td>6 CFG GNN (GAT x4 layers)</td>
<td>0.50 0.34</td>
<td>0.58 0.44</td>
<td>0.64 0.50</td>
</tr>
<tr>
<td></td>
<td>0.54/0.46</td>
<td>0.63/0.54</td>
<td>0.68/0.60</td>
</tr>
<tr>
<td>7 CFG GNN (GATv2 x4 layers)</td>
<td>0.50 0.35</td>
<td>0.59 0.43</td>
<td>0.64 0.50</td>
</tr>
<tr>
<td></td>
<td>0.54/0.46</td>
<td>0.62/0.56</td>
<td>0.68/0.60</td>
</tr>
</tbody>
</table>

### Table 4.9: Hierarchic model architectural ablations: remove stages from the hierarchic model (the default micro & macro operators are used).

<table>
<thead>
<tr>
<th>Hierarchic framework ablation</th>
<th>Java-Small</th>
<th>Java-Med</th>
<th>Apache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F1 Acc.</td>
<td>F1 Acc.</td>
<td>F1 Acc.</td>
</tr>
<tr>
<td></td>
<td>P./R.</td>
<td>P./R.</td>
<td>P./R.</td>
</tr>
<tr>
<td>1 No macro collapse; no local-global mix; no 2nd micro; decoder gets CFG paths</td>
<td>0.47 0.31</td>
<td>0.56 0.42</td>
<td>0.62 0.47</td>
</tr>
<tr>
<td></td>
<td>0.51/0.43</td>
<td>0.61/0.61</td>
<td>0.66/0.57</td>
</tr>
<tr>
<td>2 No local-global mix; no 2nd micro; decoder gets folded CFG nodes</td>
<td>0.48 0.32</td>
<td>0.56 0.42</td>
<td>0.62 0.48</td>
</tr>
<tr>
<td></td>
<td>0.52/0.44</td>
<td>0.62/0.51</td>
<td>0.68/0.57</td>
</tr>
<tr>
<td>3 No 2nd micro</td>
<td>0.55 0.39</td>
<td>0.65 0.50</td>
<td>0.68 0.54</td>
</tr>
<tr>
<td></td>
<td>0.62/0.49</td>
<td>0.70/0.60</td>
<td>0.75/0.62</td>
</tr>
</tbody>
</table>
Possible outcomes and their implications: We compare the results of this ablation to our strongest AST-based encoder with full inner structure information. Obtaining lower results on this ablation would indicate that the structure of the statement is essential for our tackled task and that our AST model indeed manages to use it for some extent. This provides negative indication for the above-mentioned explanation (i). Essentially, the information of the sequence of leaves is contained in the information of the entire AST (that also includes the inner nodes and their structure). Hence, we expect the full model to perform at least as well as the leaves sequence. Nevertheless, obtaining even higher results on this ablation may indicate either flaws in the full model (like under-sampling data) preventing it representing the input code or that there is not enough training examples for the full model to become effective and thus it rather overfits the sparsely overly-detailed inputs.

Results for the case of hierarchic micro operator: The paths-based AST model as a micro operator (Table 4.2) yields higher results comparing to this ablation (row 1 on table 4.7), indicating the statement’s structure is essential for the task (disproving above-mentioned explanation (i)).

Results for the case of whole-method encoder: The leaves sequence for the entire procedure’s AST (row 2 on table 4.5) yields significantly higher results compared to Code2Seq paths-based AST model (row 2 on table 4.3). It indicates that Code2Seq’s inevitable paths under-sampling is too aggressive. We further discuss it in Section 6.6.

4.3.2 Whole-method encoder is a flat sequence of tokens w/o separators / operators / keywords

Motivation: Validate the flat model’s ability to grasp any kind of the code’s inherent structural information. Having the separators in the code, for example, is typically required to unambiguously parse the code. By removing them, the resting code cannot convey some of its original structural information.

Possible outcomes and their implications: We compare the ablation’s results to the results of the full flat representation. If similar results are obtained, it would indicate that the code’s structural information is not necessarily used by the flat representation. That would provide a negative indication for explanation (ii). For the case of whole-method, both cross-statements control structures and within expression structures are being questioned. For the case of hierarchic micro operator, only the latter is being questioned.

Results: For no-separators and identifiers-only ablations (rows 6 and 7 on table 4.4) we observe a notable degradation in scores w.r.t the original flat representation (row 1 on table 4.3). This suggests that even the flat model manages to take advantage of
the code structure by processing its mere textual representation. However, keeping out only keywords or only operators (rows 4 and 5 on table 4.4) seems not affecting the model. As these kinds of code tokens hold non-negligible information about the program, we conclude that even though the flat representation apparently manages to grasp structural information to some extent, it still lacks realizing the full potential of the structural information.

4.3.3 Un-ordered set of tokens

**Motivation:** That is yet another manner of obfuscating the code’s grammar to isolate between the code’s structure and the existing tokens in the code.

**Possible outcomes and their implications:** When used as a micro operator in the hierarchic model, if reaches similar results to the flat representation it would indicate that the flat representation doesn’t necessarily uses the expression’s structure. When used as a whole-method encode, we would like to expect the results to be lower than the flat representation, indicating the model does somehow grasping distances over the code (searching for tokens near the logging statement for example).

**Results:** We see in row 2 on table 4.4 a significant degradation in scores w.r.t the original flat representation (row 1 on table 4.3). This additionally supports the ability of the flat representation to grasp some of the code structural information.

4.3.4 Fragmentally un-ordered set of tokens

**Motivation:** Yet another experiment to check to what extent the whole-method flat representation manages to take advantage of the code’s grammar to grasp its hidden structure.

**Description:** We cut the entire sequence into sub-sequences of random lengths between 5-25. Than, we shuffle the tokens inside of each sub-sequence, while keeping the order between the sub-sequences.

**Results:** We see in row 3 on table 4.4 a significant degradation in scores w.r.t the original flat representation (row 1 on table 4.3). However, these results are higher than the ablation of the fully un-ordered set of tokens. The performance order between these three versions settles with the actual amount of information omitted in each ablation.

4.3.5 No macro operator

**Motivation:** Validate the task’s necessity of global information propagation. Form a lower bound for assessing the contribution of other macro operators.
**Description:** In this ablation we employ the hierarchic model, but without applying any kind of macro operator. It means that there is not cross-statement information propagation at all. The micro operator is kept to be our default choice. The goal is to assess an upper bound for the performance of the local operator solely. In other words, we answer the question to what extent does the lack of global information propagation limits the model. This can be regarded as a lower bound for the assessment of other macro operators.

**Results:** We see in row 1 on table 4.8 a noticeable degradation in scores w.r.t our method (table 4.2). This ablation is even inferior to the flat representation. It suggests that the global information indeed makes a difference.

### 4.3.6 Macro operator is an un-ordered set of CFG nodes

**Description:** The locally-calculated encodings of the CFG nodes are being processed by a model that represents un-ordered sets to propagate the information globally between the nodes, but without any information about their inner order - neither textually nor with regard to the control flow.

**Motivation:** Both in the full control-flow paths macro operator and the previous ablation, most of the related CFG nodes are expected to be close by representation. Here we validate the more-general importance of any kind of proximity between related CFG nodes with respect to the LogVar task.

**Possible outcomes and their implications:** We expect the results to be lower than the previous ablation, indicating there is an importance to the order of the statements and that the model can use this order when it is given.

**Results:** It seems the model doesn’t manage to take advantage of the arbitrary global information mix, as the results (in row 2 on table 4.8) are similar to the none macro operator case (in row 1 on table 4.8). Hence, without the having any information about the near contextual neighbour of a CFG node, the model takes advantage only on separated local statements.

### 4.3.7 Macro operator is a single sequence of CFG nodes ordered by their textual appearance in code

**Description:** The CFG nodes are arranged in a single sequence, ordered by their matching statements appearance in the code. Their locally-calculated encodings are being inserted in this fashion into a sequential model to propagate the information globally between the nodes.
**Motivation:** Validates the full model’s ability to use the control-flow paths over the given training data to solve to *LogVar* task.

**Possible outcomes and their implications:** We compare the ablation’s results to the results of the hierarchic model with the same micro operator and all full CFG paths as the *macro* operator.

**Results:** Surprisingly, as we see in row 3 on table 4.8, this ablation obtains significantly better results w.r.t the ablation of no *macro* operator and to the flat representation baseline. This suggests that even the rough contextual information in the CFG holds significant information of the code’s *global* structure. However, it should not be extremely surprising, as the induced order of the CFG nodes’ textual appearance many times approximates well the control-flow topological order. Lastly, our CFG paths-based *macro* operator is nevertheless still superior to all the mentioned cases.

### 4.3.8 Macro operator is an AST paths encoder for the procedure’s AST pruned at expression statement nodes

**Description:** The procedure’s AST can be divided into two parts. The *upper* part contains the root node and the inner nodes, while pruning sub-trees that represent a CFG node’s statement (only its root node remains). That way, the *upper* part characterises the control structure of the program, containing inner nodes like `BlockStmt` and `IfStmt`. The *lower* part contains the pruned-out sub-ASTs of the statements. In this ablation we replace the CFG paths macro operator with a paths-based AST encoder that is applied over the *upper* AST structure. As a *macro* operator, it receives the locally-created encodings of the CFG nodes and assigns them to the matching leaves of the pruned tree. By so, the global information propagation is performed along syntactical paths.

**Motivation:** We would like to question the contribution of the control-flow paths as a global information propagation means in the hierarchic encoder. As an alternative for comparison, we propose the syntactic AST paths.

**Effective alternative to Code2Seq:** The whole-method paths-based AST encoder from the previous section produced poor results. One explanation is information loss because of under-sampling of paths. In Section 6.6 we further discuss this issue. Here we also validate this assumption by suggesting a resemblance modeling that doesn’t require heavy sampling and therefore doesn’t suffer from information loss. Note that the same kind of information presents here and in the whole-method paths-based AST encoder. Figure 3.4 illustrates this lower-upper separation. In that way, this model can be regarded as an effective approximation for the whole-method paths-based AST.
Outcomes and their implications: This model (row 4 on table 4.8) mitigates the scores gap between the Code2Seq (row 2 on table 4.3) and our hierarchic model (table 4.2). On the one hand, the ablation results are higher than Code2Seq, providing an evidence for the hypothesis about the information-loss and the effectiveness of the hierarchic model in this context. On the other hand, the ablation results are lower than the hierarchic model with the control-flow paths macro operator, indicating that the control-flow information is of greater importance than the control syntactical structure, at least for our task. Furthermore, even the simple single CFG nodes sequence as the macro operator (row 3 on table 4.8) performs better than this ablation, even more strengthening the claim about the lack of information in the global syntactical structure.

4.3.9 Macro operator is a GNN applied over the CFG

Motivation: Validating the efficiency of the path-based encoding technique over the CFG.

Description: In this ablation, we replace the paths-based CFG macro encoder with a GNN. We attempt several types of GNNs and for different number of layers. We compare the results to the results of our main configuration for the hierarchic model, which is the paths-based CFG macro encoder. By so, we put the choice of the path-based encoder to the test. Notice that the two alternatives are given with the same input information as a graph, once represented as expanded paths and once as an explicit graph. Hence in this ablation we do not check the choice of using the control-flow information, but we check only the choice of representing it as extended paths. This could also be regarded as another evidence for the hypothesis about the locality of the information propagation in GNNs.

Results: The results for all of the GNN-based macro operators (rows 5 to 7 on table 4.8) yield lower results than our path-based operator. These results matches our preliminary expectations based on the well-discussed GNNs locality limitation.

4.3.10 Macro operator is a set of CFG paths, no update of micro encodings with global context

Motivation: Validating the necessity of the two stages: (i) folding the CFG paths into nodes representations, and (ii) mixing it with local encodings.

Description: In the hierarchic encoding algorithm 3.1, the macro operator in line 6 produces new representations for the CFG nodes after processing them with their global context. Then, in line 8, these context-aware CFG nodes encodings are being used to update the inner micro encodings of the expression’s tokens. The choice of the macro operator’s output form is required by design to perform the just-mentioned mixture of
the local encodings with their global contextual information. Here we want to justify these choices. Firstly, we explore a much simpler alternative, where the macro operator produces a set of encodings for the CFG paths. In this alternative, the collapse of the CFG paths into CFG node representation is omitted, together with the following steps in algorithm 3.1 (update of the micro encodings and applying the micro operator in the second time). The CFG paths encodings are being passed to the decoder. As a second alternative, we propose a finer modification, where the CFG nodes folding is performed as done originally, but there is neither update of the micro encodings with their global context, nor usage of a second micro operator. The CFG nodes encodings are being passed to the decoder.

**Results:** We see in row 1 on table 4.9 a significant degradation in scores w.r.t our method (table 4.2), suggesting some of the ablated parts of the hierarchic framework are indeed essential.

4.3.11 No second micro operator

**Motivation:** Validating the necessity of the second micro operator.

**Description:** In line 9 of the hierarchic encoding algorithm 3.1, a second micro operator is applied over the updated micro encodings (mixed with their global context). In this ablation this step is omitted. The mixed micro encodings \{\langle m_1^i, ..., m_{||S_i||}^i \rangle \}_{i \in [n]} are being passed to the decoder.

**Results:** We see in row 3 on table 4.9 a notable degradation in scores w.r.t our method (table 4.2), supporting the essentiality of the second micro operator.

4.3.12 Micro operator is a GNN / TreeLSTM applied locally over the statement sub-AST

**Motivation:** Validate the efficiency of the paths-based AST micro operator for local top-level expressions.

**Description:** Here we replace the paths-based AST micro operator in the hierarchic model with several other models. Both TreeLSTM and GNNs are capable of processing tree- or graph-structured data. Hence, we do not check here the choice of representing the statement as an AST, but we only question the technical means we use for processing this AST representation; that is, the compared architectures differ in the manner they propagate the information over the same input structure.

**Results:** We see in rows 3 to 6 on table 4.6 notable degradation in scores w.r.t our paths-based AST micro operator (Table 4.2). This suggests that the paths-based
Table 4.10: Ablations for identifier embedding approaches applied over the hierarchic model.

<table>
<thead>
<tr>
<th>Identifier ablation</th>
<th>Java-Small F1 Acc.</th>
<th>Java-Med F1 Acc.</th>
<th>Apache F1 Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 vocabulary of atomic identifiers (no identifier sub-tokenization)</td>
<td>0.56 0.40</td>
<td>0.64 0.49</td>
<td>0.69 0.55</td>
</tr>
<tr>
<td>2 single common identifier word (no vocabulary)</td>
<td>0.55 0.41</td>
<td>0.63 0.48</td>
<td>0.68 0.54</td>
</tr>
<tr>
<td>3 single common sub-identifier word (no vocabulary)</td>
<td>0.55 0.40</td>
<td>0.62 0.49</td>
<td>0.69 0.54</td>
</tr>
<tr>
<td>4 full sub-identifiers obfuscation (draw random vocab word for sub-identifier)</td>
<td>0.55 0.39</td>
<td>0.62 0.49</td>
<td>0.69 0.54</td>
</tr>
<tr>
<td>5 sub-tokenization; 64 OOV words; disable obfuscation regularization</td>
<td>0.53 0.38</td>
<td>0.65 0.48</td>
<td>0.69 0.56</td>
</tr>
<tr>
<td>6 sub-tokenization; single OOV word; disable obfuscation regularization</td>
<td>0.53 0.36</td>
<td>0.65 0.47</td>
<td>0.69 0.56</td>
</tr>
</tbody>
</table>

4.3.13 AST paths-based encoder uses only leaf-to-leaf or only leaf-to-root paths

**Motivation:** Is it necessary having leaf-to-leaf paths? When both are necessary?

**Description:** Code2Seq and GEN use leaf-to-leaf AST paths over the entire procedure’s AST. As we already mentioned, processing leaf-to-leaf paths comes with quadratic computational complexity. Here we want to validate its necessity. Additionally, when we apply the AST paths-based encoder as a micro operator, we use both leaf-to-leaf and leaf-to-root paths. We do so because 8% of the expression statements have only one leaf (and hence have no leaf-to-leaf paths at all). Here we validate the necessity of using leaf-to-root in that setting.

**Results:** We observe in rows 3 and 4 on table 4.7 a notable degradation in scores w.r.t our method (table 4.2), suggesting the essentiality of involving both leaf-to-leaf and leaf-to-root paths.

4.3.14 Atomic identifier vocabulary instead of sub-identifiers tokenization

**Motivation:** Validate the necessity of identifier tokenization into sub-elements.
**Description:** With accordance to today’s code models, in our original model, each identifier is encoded by applying a sequential model other its sub-identifiers as described in Section 4.1.5. In this ablation, we question whether this process is necessary. We do so by replacing it with a simpler embedding mechanism. The identifiers are regarded as atomic tokens, and simply inserted into a vocabulary intact without decomposing them into sub-elements.

**Results:** We see in row 1 on table 4.10 a minor degradation in scores w.r.t the results of our method, suggesting the identifiers sub-tokenization benefits the model, but not significantly. Maybe the mere minor contribution is due to the small number of training examples. It is possible that training upon bigger datasets would make a more significant difference.

### 4.3.15 Single encoding vector for all sub/identifiers

**Motivation:** Validate the dependence of the model with any separation between the identifiers.

**Description:** All the identifiers are replaced with the same trainable embedding vector.

**Results:** We see in rows 2 and 3 on table 4.10 a minor degradation in scores w.r.t our method, suggesting that the vocabulary does slightly benefit the model. However, the model doesn’t take a significant advantage of the given ability to distinguish between different variables. Instead, the overall code’s structure is sufficient for the model. Additionally, in this case the identifier sub-tokenization becomes unhelpful, as the two results are very close. The latter effectively means that knowing simply how many sub-parts an identifier have doesn’t contribute the model.

### 4.3.16 Full sub-identifier obfuscation

**Motivation:** Validating that the model doesn’t relay on memorization of common logged symbols names.

**Description:** Instead of using a fixed vocabulary of identifiers, where an identifier is mapped to an embedding vector, here we keep a vocabulary of obfuscated identifiers \{OBFUSCAED_1,...,OBFUSCAED_{|V|}\} where \(|V| = 64\). The identifiers in the batch are being randomly mapped to different obfuscation words. In such configuration, the same identifier might be mapped to another obfuscation word when presented in different examples and/or different batches. That way, we completely prevent the model from memorizing any special meaning to an identifier.
Results: We see in row 4 on table 4.10 a minor degradation in scores w.r.t our method; Meaning that the original model doesn’t rely on sub-identifiers memorization. More specifically, the model is able to learn to predict reasonably without relying over the logged variable names. Additionally, The results are same as previous identifier ablations, suggesting that adding only the ability to distinguish between sub-identifiers still without having their vocabulary doesn’t help the model at all. In other words, the slight extra score the not-ablated model has is because of the sub-identifiers’ learnt vocabulary and not because of the sole ability to distinguish them from another.

4.3.17 No random sub-identifier obfuscation regularization

Motivation: Justify the regularization technique suggested in Section 3.4.2 and used by the model.

Description: In our default model we augment the training data by randomly replacing 30% of the known identifiers with ad-hoc unknown vectors. Here we cancel this regularization to evaluate its effect on the model.

Results: We see in row 5 on table 4.10 that disabling the sub-identifiers obfuscation regularization leads to a slight degradation (but consistent over benchmarks) in scores w.r.t our method. This supports the effectiveness of the suggested regularization. Additionally, in row 6 on table 4.10 we also disable the multiple OOV words regularization. It seems that the multiple OOVs alone (without random obfuscation) has no effect on the model.
Chapter 5

Related work

As our work aims to fill the gap between the study of the log prediction task and the study of code representations, we refer to studies from both niches.

5.1 Code distributed representations

5.1.1 AST leaf-to-leaf paths approach

As described in Section 2.4, Code2Seq [ABLY19] models the program using a weighted set of AST leaf-to-leaf paths. In their work, Alon et al. demonstrated the superiority of this representation for the task of function name prediction when restricted for short functions. They showed that a set of a few paths can represent concepts like counting occurrences, reversing an array, and checking for membership. Our approach is similar to theirs in the way that we both basically use end-points paths as a modeling approach. However, the two main differences are: (i) the graphs we operate on; and (ii) the explicit separation between local (intra-statement) and global (cross-statement) modeling.

In Section 6.6 we explain how our model can be practically seen as an improvement for Code2Seq, and in Section 6.2 we exemplify the lack of explicit reflection of data-flows in Code2Seq as opposed to their straightforward reflection in our hierarchic model.

In a follow-up work, Structural Language Model (SLM) [ASLY20] Alon et al. employed the paths-based modeling for the code completion task. The authors treat any expression sub-AST as a valid potential completion site; that is, their proposed model is expected to be able to generate any missing expression from a given procedure. Projecting our case over their work, the LogVar task can be formulated as an any-code completion task, where the completion site is a modified version of the log statement, as the model should predict only the used variables and not the entire logged expression. Thus, the original logging statement should be replaced with a flat sub-AST having a special AST node of the type <LOG> as its root, and the logged variable identifier AST nodes as its direct children. If fact, because this completion site is degenerated, their former Code2Seq model is virtually equivalent to SLM, except for the copy attention. Note that predicting only the variables is a valid task itself, as deciding which variables
hold crucial information about the program runtime involves non-trivial perceiving of the program’s characteristics. On the other hand, the entire logged expression notably varies depending on the peripheral project’s codebase; moreover, it often involves calling methods that are not used in the procedure’s scope and thus becomes unpredictable in our setting.

5.1.2 Extended code property graph (CPG) approach

As described in Section 2.6, Allamanis et al. suggested taking advantage of the code’s well-defined underlying structure by (i) representing it as a code property graph (CPG) enriched with semantic code relations, and (ii) processing it using a GNN [ABK18]. Their work has been widely adopted by the community and evaluated for multiple programming-related tasks, including prediction of log level [LPH+19].

However, this solution has been shown to suffer from the message-passing over-smoothing issue [HSS+20], limiting its ability to propagate information along long distances over the program graph. Consequently, this issue confines the expressive power of the model for local relations. Unfortunately, our desired pattern discussed in Section 6.2 could not be counted as a local relation over the proposed program graph, as the graph’s backbone is basically still the program’s AST, and the target pattern involves multiple statements.

Indeed, as we demonstrate in Section 4.3, the GNN-based methods are incompetent for grasping relations that are scattered over long distances in such big graphs and are less effective even for restricted local structures. Our modeling solves this issue by the explicit local-global separation and the reliance of sequential models to propagate information along distances both over the local and the global cases.

In particular, the CFG itself has also been used in the context of reverse engineering [DAY20]. However, the authors haven’t needed neither the rich syntactical structure nor any hierarchic modeling, as they used the instruction-level CFG, which means that each CFG basic-block is a primitive intermediate representation (IR) instruction, as opposed to a statement-level CFG, where a basic-block is a more high-level statement or condition. An IR instruction consists of an operator and a constant number of operand registers. Thus, the encoding of a CFG node is based on elementary concatenating of few embedded dictionary tokens. In contrast, the encoding of a statement-level CFG node in the hierarchic model involves the compound modeling of the syntactical structure the associated top-level expression.

5.1.3 Tackling GNN’s over-smoothing issue

GraphSandwiches [HSS+20] is an attempt to alleviate the de facto locality of the message-passing protocols for programming-related tasks. Their approach is based on applying a sequential model over the encodings of the graph nodes as obtained by a classic message-passing iteration. It exhibits better performance than only applying
GNN over the entire graph. This result shouldn’t come as a surprise, as the sequential models are known for their greater ability to propagate information through long distances. However, the manner the authors used the sequential model completely disregarded the graph’s structure, and they eventually still relied on GNNs to overcome this concern. Their second approach, GREAT, doesn’t involve GNNs but is restricted to the Transformer architecture. Thus, it requires large datasets to train on (incompatible with our task as mentioned in Section 3.1.4) and induces quadratic memory complexity w.r.t the number of nodes in the program’s graph. In our solution, we also employ a sequential model, but over much shorter sequences and via the expand-collapse approach, which is designed to conserve the graph’s structural information. In Section 4.3, we compare our method to a single sequence of nodes (both for the local and the global cases) to show the superiority of expand-collapse.

HGNet [RW21] is a hierarchical neural network architecture also aims to tackle the insufficiency of the message-passing models for capturing long-range interactions in graphs. They use the heuristic Louvain method for graphs clustering to constructed an abstract level on top of the original graph. They repeat this operation multiple times until they are left with a fixed sized graph. That way, an hierarchic structure is obtained. In the first stage of the processing, they perform an down-up computation, by apply a GNN iteration over each hierarchic level, pool its clustered nodes to produce node encodings for the next level, and so on until the most upper level has been computed. Now, they perform an up-down processing to propagate the global information towards the original graph. This is similar to our local-global mix where we pass the information from the global level to the local one. They evaluated HGNet over molecular property prediction benchmarks and shown overall superiority w.r.t classic GNN approaches. However, each hierarchic level in their solution is still confined to the GNN’s limitations. In Section 4.3, we show that for our hierarchy leveling choice (which we justified in Section 6.3) the GNN is less effective as the per-level structure processing (both for local and global). In our solution, we leverage the powerful paths-based modeling to encode each hierarchic layer. In addition, our code hierarchic framework is described in generic terms, leaving out modeling choices for the hierarchic levels. By so, the most adaptable modeling can be chosen for each hierarchic level, without particularly restricting it to GNNs. Lastly, our hierarchic cutting-point is chosen carefully and intentionally to suit our modeling objectives, while HGNet performs heuristic graph clustering to choose these cutting-points.

GraphTrans [JWW+21] is another attempt to propagate information globally over graphs by using a sequential model. However, they still rely on GNN as their sole structure-aware information propagation technique, while the sequential model is applied over an unordered nodes set, utterly regardless to the graph structure. Alon and Yahav [AY21] tried to address the graph bottleneck issue by applying an additional GNN layer over all nodes connected as a clique. Their solution also relies on non-structure-aware patches.
The expand stage of our expand-collapse algorithm adopts the idea of encoding graphs by applying sequential models over extracted paths, which is suggested both by Taheri et al. [TGBW18, TGBW19] and by Alon et al. in Code2Seq [ABLY19]. However, the main novelty of our approach is rather the integration of the collapse stage. The essentiality justification of the collapse stage is intuitively explained in Section 3.3.2 and empirically evaluated in Section 4.3. In fact, the idea of extracting sequences from a graph to encode it originates back in even more traditional approaches like the Node2Vec [GL16], where they extended the use of n-grams-based encoding methods from flat text to a general graph structure. However, these classic methods were limited compared to today’s sequential models, as they rely on the recurrence of these exact n-grams or skip-grams in the training set.

5.2 Log-related tasks

The LogVar task has been studied lately by Liu et al. [LXL+19]. To our knowledge, this is the only study to explore this task. In their work, they employed the flat representation (explained in Section 2.2) to model the input procedure without the target logging statement. In our dissertation, we consider this work as the main baseline. Our main hypotheses (in Section 3.1.10) refer to the importance of structural information incorporation in the context of the LogVar task, and to the insufficiency of structural information in the flat code modeling approach. In fact, their work was the first to generally exemplify the capability of basic deep-learning-based solution for this task. However, they have not covered the aspect of representation exploration at all. Moreover, our work is the the first to point out the concrete challenges that makes the LogVar task an appealing case-study for code representations exploration. Additionally, in our work, we significantly enlarged the dataset Liu et al. benchmarked their model over, and in contrast to their per-project evaluation scheme, we favored the more practical and prevalent cross-project evaluation scheme (as mentioned in Section 4.1.1).

Apart from LogVar, other log-related tasks have also been studied over the years, like log message enhancement, predicting where to insert log statements in a given program, and predicting what message severity level to associate with a given log statement. Being inspired by manual statistical studies, most of the proposed solutions are based on hand-crafted features and heuristics applied over restricted case studies (like try-catch blocks or conditional branches controlled by specific families of conditions).

In the few recent years deep-learning-based solutions has begun pervading the study of log-related tasks. Being data-driven expands these solutions for the general case, rather than being restricted to specific case-studies. Additionally, learning from data allows adapting the solution to project-wise or organization-wise conventions without requiring extra research efforts. However, these studies mainly demonstrated the general potential of incorporating deep-learning techniques for these tasks, without ques-
tioning about the representation. In the rest of this section we elaborate on log-related studies from the two mentioned disciplines.

5.2.1 Traditional research paradigm studies

_LogEnhancer_ [YZP+12] by Ding Yuan et al. is a tool for programmers that suggests additional vital variables to append to the log message. It does so by first performing data-flow and control-flow analysis to the program and then using its result to make additional analysis for finding variables that store information that is causally-related to the execution path of the observed log-statement. The idea is to reduce that uncertainty about this execution path as much as possible. A variable would be added only if it helps reducing this uncertainty in a way that the original variables could not. They looked at modifications in the source-code where the programmer added variables to an existing log-statement. The _LogEnhancer_ was able to detect 95% of these variables.

In another study, Ding Yuan et al. [YPZ12] performed a case study on logging practices in big Java projects. They collected statistics about code modifications of log-printing-stmts in the history of the projects’ revision control management. These statistics might indicate, as the authors believe, where exactly developers spend significant efforts, and what developers are confused about regarding their logging decisions. They found out that logging-statement movement or deletion is rare. In 26% of the updates the log-level was changed, which means that, as explained by the authors, the developers changed their mind about the criticality of an error event or they were confused when estimating the cost overhead. In 27% of the updates, the variables to log were modified, and among these changes, the majority (62%) is adding variables. The authors propose that it may indicate that developers need additional runtime information (that they originally hadn’t thought they would need) to diagnose a failure.

_LogAdvisor_ [ZHF+15], suggested by Jieming Zhu et al., is a tool for solving a variant of the _Where-To-Log_ task. The authors conducted a user study, by looking at the source-code of four different projects. They found out that the vast majority of the logging statements are placed within catch-blocks or in the taken branch of a conditional if-statement that checks whether a returned value (from a previous function call) equals one of the values in \{-1, null, false, empty\}. Their task: given such a snippet (without logging), determine whether or not it should be logged. They extracted hand-crafted features from the relevant examples and employed a feature selection methods to reduce the feature-vector dimension. They used classic machine-learning classifiers over the generated data (SVM, Naive-Bayes, decision trees and logistic regression). Their work is based on previous empirical study on logging practices [FZH+14], that includes both source-code analysis and a questionnaire survey with 54 developers. In that study they detected five categories of snippets that are statistically likely to contain a log-statement, and name these with the following tags: assertion-check, return-value-check, exception, logic-branch, observing-point.
LogOpt [LS16] was suggested by Lal Sangeeta et al. to address a special case of the Where-To-Log task, where only for catch-blocks are considered. In this work they considered the within-project variant. They observed catch-blocks and manually identified 46 features extracted from the source-code. Their features list includes, for example: (i) is there asserts in the catch block? (ii) is there Thread.Sleep() in the catch block? (iii) how many methods are called from the catch block? (iv) properties of the caught exception type and so on. They used Random Forest (RF) classifier as a learning algorithm. Later, they suggested a new tool named LogOptPlus [LSS16] that is an extension of their previous tool LogOpt, but here they not only considering catch-blocks, but also if-blocks.

Logger4U [SSL16] by Srishti Saini et al. also tackled the Where-To-Log task for the case of if-blocks only. The authors extracted 28 hand-crafted features from the source code. They used Support Vector Machine (SVM) classifier once with the standard linear version and once with the 1 RBF kernel. They also tried the Multilayer Perceptron (MLP) and the Random Forest (RF) classifier. The SVM with the RBF kernel performed the best and RF has the second best performance.

ECLogger [LSS17b] by Lal Sangeeta et al. addressed again the catch-blocks case of the Where-To-Log task, but now for the cross-projects variant. They trained multiple machine-learning ensembles of classifiers. In a later work [LSS17a], they extended their previous work for if-blocks as well, this time using LogOpt, LogOptPlus, and ECLogger from previous works.

The tool Log20 [ZRL+17] suggested by Zhao Xu et al. for solving the Where-To-Log task. They used the Entropy measurement from the field of Information Theory for making logging decisions that would reduce the uncertainty about the execution path of the program. In that way they showed the potential profit of using the CFG paths in the context of logs. It turns out that finding an optimal log placing is a hard problem. Their algorithm effectively approximates this objective using dynamic programming.

In their work [LSH17], Heng Li et al. addressed the task to predict which is the most suitable log level for a given log statement in a program. They used classic regression models. They checked the correlation between log-level used and the block type where the log-statement resides in (if, else, catch, for, while). In a later work [LCIS18], the same authors addressed the task of Where-To-Log (for a general code-block). They defined a fixed set of code-snippet categories (called them “topic”s) that describes the content of the code. Given a code-snippet, they used the names of variables in that block and the method name to predict a topic label which (they believe) approximates the “functionality” of this code-snippet. Then, they checked for each topic its probability to contain a log printing statement. They got that only a small number of topics much more likely to be logged.
5.2.2 Modern data-driven deep-learning-based studies

In their study [LPH+19], Mingzhe Li et al. addressed the log-level prediction task by suggesting a NN architecture based on Allamanis’ graph program representation processed via GGNN. They compared their results to Heng Li’s model [LSH17], and shown their superiority over it. However, their best model integrated the GGNN approach together with the simpler sequential approach of an LSTM, suggesting the expression power of the GGNN-based approach may not be optimal for program representation. Later, a follow-up study [LLCS21] by Heng Li et al. re-visited this task, now by using a simpler LSTM-based model integrated with simplified AST contextual information.

In *Where Shall We Log?* [LCS20], Li et al. suggest an LSTM-based deep-learning architecture to predict relevant places in program to insert logging statements. However, they rely on an extremely involved and primitive feature engineering to represent the program’s code, without a decent usage of the code’s underlying structure.

5.2.3 Log output processing

Various peripheral studies [FLWL09, XHF+09, LFY+10, FRZ+12, FLL+13, DLZS17] suggest techniques for retrospectively inspecting the produced logging output script and trying to find anomalies in the program execution and/or correlation between the log output script with known system failure events. In [DLZS17] they used *Recurrent Neural Networks* with LSTM cells to learn logging patterns from examples where a normal execution took place, and later to detect anomalies by finding deviations in log patterns. These works relates to ours, as they all address log-related tasks. However, this certain task is not one of the direct tasks we explicitly address.
Chapter 6

Discussion

In this section, we discuss our approach and its limitations. First, in Section 6.1, we conclude our base hypotheses. Then, in Section 6.2, we exemplify the reflection of data-flows in the hierarchic model, and compare it to Code2Seq’s modeling. Next, in Section 6.3, we justify our choice of the top-level expressions as the cutting points of the hierarchic leveling. Afterwards, in Section 6.4, we discuss the limitations our methodology. In addition, in Section 6.5, we outline the limitations of our dataset and how they could potentially be improved in future work. Later, in Section 6.6, we discuss the general applicability of our hierarchical representation. Further, in Section 6.7, we outline the advantages of the hierarchic modeling, as indicated from our findings. After that, in Section 6.8, we suggest applying the hierarchic approach to other tasks. Finally, in Section 6.9, we conclude the effectiveness of the expand-collapse framework and propose to explore it for other graph tasks.

6.1 Hypotheses conclusion

In Section 3.1.10, we brought up the two main hypotheses that guide our work. Here, we conclude them in the light of our findings throughout the work.

Regarding our first hypothesis, in Section 4.3 we conducted multiple experiments to verify the importance of the structure for the LogVar task. We deliberately modified the flat models, the AST-based models and the hierarchic model to show that whenever parts of the structure are eliminated the overall performance of the model deteriorates. The obtained results support our claim.

Our hierarchic model yields better performance than any other explored model over all tested benchmarks. Modifying its architecture in manners that restricts the usage of the underlying code structure causes a decrease in the performance. These results testify for the prominent contribution of the selected code structure elements in the hierarchic model, and thus strengthen our second hypothesis.
6.2 Reflection of chain-of-relations and data-flow in the hierarchic model

6.2.1 Re-visiting real-life logging example from overview

In Sections 3.1.7 and 3.1.8 we exemplified how a chain-of-relations modeling could represent code patterns that are characterized by non-trivial composition of relations. Figure 6.1 demonstrates how the pattern, originally presented in Section 3.1.7, is being reflected in our hierarchic model with our choices for micro and macro operators (AST leaf-to-leaf paths and CFG paths respectively).

The simplified pattern in Figure 6.1b is characterized by chaining three kinds of binary relations: (i) data-dependency (two first edges); (ii) control-dependency (3rd edge); and (iii) control-flow (last edge). In Figure 6.1a we present the statement-level control-flow path that consists this pattern. Then, we expand one of its top-level expressions to demonstrate its reflection by the micro operator.

At the first stage, the micro operator can model data-dependency binary relations. Such a relation is induced by an assignment, which is contained in a single top-level expression. Thus, a leaf-to-leaf AST path within this expression (originating from the assignee variable, passing through the assignment operator and terminating in the used variable) can represent this relation. In the figure, the assignment into the variable container is expanded, with the path between name and container highlighted. Other paths within the expression could add information about the relation’s characteristics.

At the second stage, after the inner data-dependency relations have been encoded locally within their associated statement-level nodes, the macro operator models CFG paths. The first edge symbolizes the concatenation of two binary data-dependency relations (thus, forming the ternary relation (spi, name, container)). The second edge symbolizes the def-use relation between the definition of container and its use in the non-nullish condition expression. The third edge symbolizes a control-dependency relation (the variable container affects the transferring of the program’s control to the logging operation), and the fourth edge symbolize the last control-flow relation. When a CFG paths is being expanded by our macro operator, special tokens are inserted between every pair of consequent CFG nodes to symbolize the edge kind between them (e.g: the type of the third edge is IfTrue). Therefore, the entire sequence in Figure 6.1a is finally being modeled.

6.2.2 Comparing basic data-flow modeling to Code2Seq

Here, we compare the hierarchic model versus Code2Seq’s [ABLY19] set of leaf-to-leaf paths model in the aspect of their power to express data-flows. Figure 6.2a demonstrates a simple code snippet exhibiting a sequence of three data-dependency and def-use relations: (a→b), (b→c), (c→f). By concatenating these relations, a new quartary relation could be induced: (a→b→c→f). In the general case, such telescopic concatenation of
(a) On the left there is a CFG path of top-level expressions. On the right we expand a top-level expression to

Figure 6.1: Demonstration of how the pattern on the right (originally presented in Figure 3.3b) is represented by the hierarchic model.

\[ \text{container} = \text{containers}.\text{find(name)} \]

(b) The pattern originally presented in Figure 3.3b. Each edge symbolizes a binary relation of data-dependency, control-dependency or control-flow.

\[ \text{spi} \rightarrow \text{name} \]
\[ \text{name} \rightarrow \text{container} \]
\[ \text{container is null} \]
\[ \text{LOG} \]
\[ \text{return} \]

(a) Example of a code snippet exhibiting data-flow sequence of \( a \rightarrow b \rightarrow c \rightarrow f \).

(b) Illustration of the data-flow reflection in the hierarchic model. The basic binary data-flow relations are first being modeled by the micro operator. Then, the macro operator induces a control-flow path that conveys the entire data-flow sequence.

(c) Illustration of the lack of explicit reflection of the data-flow sequence in Code2Seq’s modeling. There is no single path that involves all of the symbols of the desired data-flow sequence.

Figure 6.2: Comparison of the reflection of a data-flow sequence in the Code2Seq model versus the hierarchic model with our choices for micro and macro operators.
n − 1 binary relations \( \langle a_1 \rightarrow a_2 \rangle, \langle a_2 \rightarrow a_3 \rangle, \ldots, \langle a_{n-1} \rightarrow a_n \rangle \) can be interpreted as a formation of an \( n \)-ary relation \( \langle a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \ldots \rightarrow a_{n-1} \rightarrow a_n \rangle \). As motivated in Section 3.1.7, considering relations of arity bigger than two might enrich the expressive power of the model, and thus help it to grasp more complicated data-flows.

Figure 6.2b demonstrates how the mentioned data-flow sequence is reflected in the hierarchic model. The \textit{micro} operator can locally model data-dependencies by leaf-to-leaf paths in their associated top-level expressions. Each such relation is encoded into a vector in \( R^d \) that represents this statement-level CFG node. The \textit{macro} operator concatenates these locally encoded relations into a sequence of relations, which is effectively equivalent to the desired data-flow.

In Figure 6.2c, we partially list the set of leaf-to-leaf paths (of the entire AST) for the same code snippet (in accordance to \textit{Code2Seq} approach). Note that, in contrast to the hierarchic model from Figure 6.2b, here the paths can cross between statements and pass through the statement-block inner AST node (marked as \{\ldots\}). We first observe that each leaf-to-leaf path involves exactly two terminals, and by so, models a binary relation between them. As the presented data-flow sequence is a quartary relation, it definitely cannot be expressed via a single leaf-to-leaf path.

As the set of leaf-to-leaf paths model is undoubtedly not explicitly designed to express such composed relations, its ability to do so nonetheless is questionable. We identify below three complexities that this task arises. This leads us to claim that extracting compound relations from the set of leaf-to-leaf paths is not a trivial task, and it might require an adequate modeling.

**Lost statements ordering**  Lets first consider the subset having only a single path per statement (for example, the first three listed paths in Figure 6.2c). When these paths are given in a set, their inter-order is undefined, their locality with respect to each other is unknown and their scoping is also not given. In Section 4.3 we empirically proved that this information is crucial. In the terms of our example, losing the order between the three intra-statement paths would require re-ordering them to construct the desired quartary relation. An average procedure from our dataset has 27 (±18.7) different statements, which means there are about 1.1e+28 possible permutations, suggesting rediscovering the correct order is not trivial. Even though in our simplified example there could be enough information to perform such re-ordering, such task requires non-trivial computational and learning capacity. In a more realistic example, there could be other statements involving some of the variables \( a, b \) and \( c \), which would make this task even harder. Particularly, potential re-assignments (to the same variable or assignments to another variable from another scope with the same name) could increase the perplexity about which is the right definition that precedes the \( a \) usage, and thus break the \textit{def-use} certainty that, for comparison, is obvious from the control-flow structure.

Now, lets consider paths that cross between statements and pass through a statement-
block inner AST node (marked as \{\ldots\}). Each such path passes through exactly two statements. We question how the mutual ordering of $>2$ statements can be retrieved from these paths. For example, observe the three highlighted paths in Figure 6.2c. Every path always starts from a preceding statement and ends at a later one. Thus, seemingly, all these paths together could form their total order. However, as we state in the following listed difficulties, each path is actually decoupled from its origin context, and hence such ordering re-construction becomes not necessarily possible. Even if it were computationally possible extracting the ordering information from the entire set of paths, this extraction is not a trivial task, and may require non-negligible amount of computation and learning capacity.

Lost context of intra-statement paths  

Let’s now consider the subset of all the paths that don’t cross between statements. An average statement (with more than one leaf) has 7.8 ($\pm 8.2$) inner leaf-to-leaf paths. It induces more than $1e+200$ possible matchings of all inner paths to different 27 ($\pm 18.7$) statements. As explained in Section 3.3.1, an inner node of a path is detached from some of its contextual information. That is, the same inner node can participate in multiple paths, but the information propagates only over each path separately, and hence doesn’t propagate across paths. Hence, the Code2Seq model is not given with any information to associate related paths (from the same statement or that sharing common inner AST nodes). An average statement includes 4 ($\pm 2.3$) leaves. They might have meaningful inter-relations of arity $>2$. However, as the leaf-to-leaf paths are decoupled one from each other, the model might not be able to grasp such relations.

Lost context of cross-statement paths  

The same path decoupling issue mentioned before for the intra-statement case also applies in the cross-statement case. Here, for example, for paths with the prefix $a \rightarrow \{\ldots\}$, it is not given that the node $\{:=\}$ is associated with the assignment $b := a$. This is because other assignments also produce paths with the node type $\{:=\}$, which are not explicitly distinguishable from the modeling point-of-view. Let’s take for example the path from $a$ to $f$. Allegedly, it could tell us that $f$ depends on $a$. However, as noted, the model is not informed about which assignment (or statement) the prefix of this path originates from.

To conclude, on the one hand information is missing from each separate path as it relates only two leaves, and on the other hand the paths are decoupled and therefore re-constructing the missing information from the entire set of paths is not a straightforward task. Code2Seq applies a weighted average over the set of all AST paths to produce the final representation. In the light of the above-mentioned difficulties, it is safe to question whether this sole operation has enough expressive power to extract the desired quartary relation out of these un-ordered and decoupled paths.
6.3 Justification of the top-level expressions as the hierarchic leveling cutting-point

Generally, deciding to break a given structure into two hierarchic levels brings up the question of where are the optimal cutting-points to make the leveling separation upon. In this work, we proposed and explored the top-level expressions as the hierarchic separating spot. Alternatively, we could have considered other suggestions for such cutting-points. For example, breaking the AST by sub-trees height; that is, letting the local level be sub-ASTs of height of at most \( h_{\text{max}} \in \mathbb{N}^+ \). Here, we claim that our choice is not arbitrary, and we justify it by pointing out its prominent benefits.

**Control flow** As explained in Section 6.2, control-flow can convey essential semantic compositions of relations. These control relations are defined over the top-level expressions in the statement-level CFG. Hence, for these control relations to be modeled, their operands (which are top-level expressions) should be represented as entities of the model. Having them as the cutting-points allows the model to learn local relations within the expression scope via the *micro* operator, and then to apply their control relations via the *macro* operator.

In fact, compilers perform data-flow analysis over this structure to track the spread of calculated values across the program for optimization purposes. The information flow performed by our *macro* operator propagates in the same manner over the program’s global structure.

Empirically, we indeed observe in Section 4.3 that the CFG paths modeling as the *macro* operator performs significantly better when compared to other global structures, like the upper AST paths modeling.

Moreover, a model with integrated control-flow propagation mechanism can ideally have sufficient expressive power to grasp other cross-statements relations (e.g: data-dependency, control-dependency), as these relations are usually obtained by data-flow analysis that is also based on information propagation over the CFG.

**Language grammar rules clustering** Typically, imperative programming languages (like Java, C#, python) separates between the statements-related and the expressions-related sets of grammar parsing rules. Control statements kinds (like a StmtBlock, a IfStmt or a WhileStmt) inductively have their own statement children. Similarly, expressions kinds (like a FieldAccessExpr, a MethodCallExpr or a BinaryExpr) inductively include children expressions. However, a statement can be an expression statement, which includes an expression child, but dissimilarly in most cases, an expression cannot have a statement descendant. This makes it possible to partition the AST into distinct upper and lower parts, as demonstrated by Figure 3.4. As a result, these sets of parsing rules may hold different kinds of relations and different structural characteristics. For comparison, the average branching factor of a top-level expression
sub-AST is 2 (±1.4), while for the entire procedure’s AST it is 10 (±8.2). It suggests that different modelings might be optimal for each structural part. In addition, their purpose also differs, as while the AST’s upper part represents control structures, the bottom part represents functional computations. As the hierarchic model gives the opportunity to choose an ad-hoc modeling per each hierarchic level, it makes sense exploring different modelings for each distinct structure type.

### Scalability

From a practical point of view, the empirical sizes of the top-level expressions are on the one hand small enough to allow effective application of the paths-based approach over it (having 7.8 (±8.2) leaf-to-leaf paths in average per expression), and on the other hand rich enough to possess essential relations for this modeling to be beneficial (having 8 (±4.4) AST nodes in average). Additionally, the optimal modeling for the macro structure (as found in Section 4.3) is also happened to be efficient (having 13.7 (±16.8) control-flow paths in average). These sizes facilitate the encoding complexity of lengthy procedures.

### 6.4 Methodology limitations

The number of paths in the CFG can potentially be exponential with respect to the number of nodes (= code statements). For example, consider the graph from Figure 6.3. Let’s assume it has \( n \) diamonds. To construct a path, in each diamond one has to choose which one of the diamond’s vertical nodes to pass through. This results in \( 2^n \) paths.

Such structure could be demonstrated by code with \( n \) consecutive if statements with an else clause each. Our CFG-paths macro operator extracts all the paths in the CFG. Therefore, for such case we would be left with exponential number of paths. For example, a procedure containing 12 consecutive if statements would create a CFG with at least 4,096 paths.

However, in practice, in the datasets we worked this issue was not prominent at all: in each one of the datasets apache-java, java-med, java-small, for 96% of the examples there were below 400 paths.

For the very few cases where this limitation is met, it is possible to apply paths sampling over the graph. It is possible to conduct an efficient paths sampling with uniform distribution over all existing paths.

### 6.5 Dataset limitations

#### 6.5.1 Too small datasets

Practically we have witnessed that for big datasets with millions of procedures there are only small amounts of \( \log \text{Var} \) examples. Unfortunately, deep learning models typically requires big amounts of training examples to learn from. This raises a question
whether the fancy models even manage to take any advantage over their inherent rich representation. Alon et al. reported dataset sizes of hundreds of thousands to millions for training the AST path-based models. In their constellation, their suggested model did overpower the flat representation. In our case, their AST-paths approach shows poor performance; surprisingly, significantly worse than much simpler methods like the flat representation. Additionally, our ablation study shows that even with as little information as the mere sequence of AST leaves (which has only identifiers, without any syntactic information), the model reaches higher results than other structure-based models like the AST paths-based model. Therefore, it is a viable hypothesis that the advanced models we present couldn’t manage to realize their full potential power over the given training data, and could maybe become more effective when being trained on bigger amounts data.

6.5.2 Representation expressiveness as a cause to quick model overfit

When small datasets are used to train models that are based on highly-expressive representation, the model might easily memorize the entire dataset during training and thus reach overfit. Our hierarchic micro operator is based on a powerful method (that proved itself as very effective in multiple tasks [ABLY19]), but in the hierarchic constellation is responsible only for small and restricted structures. Similarly, our hierarchic macro operator processes relatively small structure by regarding the statement-level CFG nodes as atomic in the macro context. We believe the hierarchic model enables specialization of each part with the concrete input element type it processes. Hence, as a side effect, also the memorization can become easier. Memorizing the entire method at-once using a non-hierarchic modeling can be harder than memorizing smaller elements that has been set apart in advance. Indeed, our empirical findings support this claim, as employing ad-hoc regularization techniques (sub-identifiers obfuscation and AST paths sampling as described in Section 3.4) manages to postpone the overfit. These regularizations can be seen as a data augmentation technique. The idea behind them is to make it a bit harder for the model to memorize the data and by so improves the generalization. On the other hand, trying to integrate additional path
kinds over the sub-ASTs in the micro operator (AST leaves sequence, AST siblings with parent sequences) leads to a degradation in performance. This also strengthens our claim, as additions of non-garbage data could be ignored by the model via the attention mechanism used in the folding stage without affecting the performance.

6.5.3 Open Question: Could incorporating transfer-learning approaches help?

In the recent years, transfer-learning approaches have been vastly used in the deep learning community, and particularly in the field of NLP [DCLT19]. Transfer-learning is especially applicable for tasks for which there is too few learning data to train big deep neural networks on. It is mainly done by pre-training parts of the model over other related tasks sharing similar input structure and characteristics with the original task’s data. Assuming training data for these auxiliary tasks is easily obtainable, using it for pre-training provides a better initialization point for the eventual desired model of the original task.

A follow-up work could propose a pre-training stage for the hierarchic method encoder over other code-related tasks. Examples for such auxiliary tasks could be predicting a name of a masked identifier (sequence of sub-tokens), predicting which symbol should appear in a mask location, predicting an AST/CFG edge type, and predicting symbols data-flow relations. For all these tasks, a labeled data can be automatically generated from pure source code. Pure source-code is available in big amounts today, as opposed to the case of labeled log statements. This make such approach feasible.

6.6 Hierarchic representation as an improvement of the Code2Seq AST paths-based approach

The number of leaf-to-leaf paths in the AST is asymptotically quadratic with respect to the number of AST leaves, as a tree has a path between each pair of leaves. In our datasets, the average number of leaf-to-leaf paths over the entire procedure’s AST is 9.5K. Loading and processing the complete set of paths using today’s GPUs would require unreasonable amount of computing resources.

Thus, in their implementation of Code2Seq [ABLY19], Alon et al. sample 1K paths per procedure during preprocessing, and out of them they sample 200 paths during training and inference (the latter-mentioned sample is being re-drawn per each recurrence the example during training). Additionally, they also restrict their work to short procedures only. Similarly, in this work we were able to sample at most 300 paths per example, when training the Code2Seq model over a single Tesla V100 (16GB memory) using batch size of 16.

Unfortunately, our findings in Chapter 4 demonstrate the severe incompatibility of the Code2Seq method for our task. We suspect its poor results might be related to the
distinctly lengthy procedures characterized by our task (as mentioned in Section 3.1.3), leading to the extreme amount of leaf-to-leaf paths in a procedure. Sampling 300 out of 9.5K paths incurs random loss of 97% of the paths. This immediately raises the question of whether a uniformly-drawn sample of 300 paths is sufficient for soundly representing procedures with thousands of leaf-to-leaf AST paths, and to what extent essential information is lost due to the under-sampling.

After sampling 300 paths in Code2Seq, the average number of within-statement paths in a given statement is 0.44. It means that less than one path is sampled for a given statement. This under-sampling could affect the grasping of local relations (between the 4 tokens in an average statement), that is shown to be effective in Section 4.3. The ratio between within-statement paths and cross-statement paths is 1:26; that is, less than 4% of the overall sampled paths are within-statement paths. This might not be an optimal choosing of paths, as in Section 4.3 we show that the local (statement-level) AST structure is more beneficial to the task than the global upper AST.

Our ablations study (in Section 4.3) shows that even the degenerated model of AST leaves-only sequence unexpectedly outperforms Code2Seq. However, Code2Seq (w/o sampling) strictly contains the information of the AST leaves sequence model, and therefore is seemingly suppose to perform better when the sampling is not considered. Thus, the aggressive sampling can be regarded as a compelling reason for the drop in performance. This strengthens the information loss claim.

Apart from the information loss, the quadratic number of paths induces high training costs when compared to other code representations (flat / graph-based). Additionally, the preprocessed data itself requires high amounts of storage capacity. On our datasets we experienced x27 more than the flat representation, and x6 more than the hierarchic model.

Fortunately, these issues can be mitigated by the hierarchic model using the paths-based AST as a micro operator. Unlike the bare Code2Seq model, the average total number of paths processed by the hierarchic model is practically acceptable. In our datasets, the average number of CFG paths is 107. In addition, the average total number of leaf-to-leaf paths in the sub-ASTs of all statements in the procedure is $E[\sum_{stmt} (#Leaf2LeafPaths(AST_{stmt}))] = 363$. In overall, the average number of paths the model has to process is 470 ($\approx 9.5K$). Thus, for 96% of the cases (see Section 6.4 for more details), this can be done without any sampling at all. Hence, no information loss should be incurred.

In conclusion, the hierarchic modeling allows incorporating Code2Seq’s fundamental paths-based AST modeling concept for individual statements without under-sampling and while inducing lower runtime, memory and storage costs. Indeed, our evaluation shows significant better results for the hierarchic model with the AST paths micro encoder than the original Code2Seq.

In addition, in our ablation study, we explore the case of using the upper AST
structure as the *macro* operator. That is, apart from applying the AST leaf-to-leaf paths-based modeling for local top-level expressions via the *micro* operator, the same modeling is also applied over the rest of the AST (the remaining structure after pruning off the sub-trees of the top-level expressions from the entire AST) via the *macro* operator. This configuration can be seen as an approximation for the original *Code2Seq*. As expected, this approach also yields better results than *Code2Seq*. However, the hierarchic model with our default *macro* operator (CFG paths) still yields significantly better results than all others. Moreover, it is enough for the *macro* operator to be a single sequence of all CFG nodes for the hierarchic model to outperform the model with the upper pruned AST as a *macro* operator. This result suggests that the upper AST structure is completely useless for our task.

6.6.1 **Open Question: Is the hierarchic method effective in other code-related tasks?**

The paths-based AST representation for code has proved itself by previous works as effective for various code-related tasks. In our work we show that its ineffectiveness for long procedures. We propose a novel model that employs the paths-based concept and theoretically mitigates some of the deficiencies of the original paths-based AST approach. We would be glad to see further works evaluating our hierarchic model over other code-related tasks.

6.7 **Hierarchic representation advantages**

We outline below five benefits of the hierarchic framework. Here they are presented in the context of code encoding, but could also be relevant for other tasks with well structured data.

**Specialization** As has been pointed out in Section 6.3, a structured data, like code, could be formed of sub-structures of different characteristics. It is possible that different optimal modelings suit different kinds of sub-structures. Additionally, different scopes of the entire structure might carry different latent patterns (e.g: the language grammar diverges for expressions and statements as mentioned in Section 6.3). The forced nomination of a designated sub-model for a concrete restricted sub-structure might increase the utility of its learning capacity. In that way, the elected sub-model is being forced to specialize in the predefined scope.

**Fine-grained modeling exploration** The freedom to choose different modeling for each hierarchic level separately allows exploring the data necessities in a finer manner. By comparing various representations for the local and the global structures independently, we can better assess the individual contribution of each one for the addressed
task. In Section 4.3 we conducted such experiments by replacing the *micro* and *macro* operators. For our task, it helped us detect the unnecessity of the upper AST part, which lead to sparing 96% of cross-statement AST paths. Generally, sometimes a costly modeling can be helpful only for certain sub-parts of the entire structure. Designating it only for these relevant sub-structures can save unnecessary computations.

**Paths-based modeling** The paths-based approach proved itself as powerful both in previous work [ABLY19] and in our work. The separate application of the *micro* and *macro* operators over two different structures having different kinds of entities makes it possible to incorporate the paths-based modeling both for the AST and for the CFG altogether. Trying to employ the paths-based approach in a non-hierarchic manner over a single graph containing both structures joined together (similar to Allamanis’ extended AST [ABK18]) could be challenging, as it would induce enormous number of paths. By breaking the entire structure into distinct hierarchic levels, we effectively chose the relevant paths out of the complete graph and also scheduled the information propagation accordingly. In Section 6.6, we explain why simply sampling paths from the entire graph intrinsically creates an unwanted bias towards the less-informative paths. The intentional choice of the hierarchic levels cutting-point helps prioritizing the essential paths.

**Modeling compositions of local relations** As described in Section 6.2, the hierarchic approach enables the modeling of composed relations and data-flows. This provides an effective modeling for relation of arity $> 2$.

**Shortening distances because of higher abstraction** The enforced definition of the higher hierarchic level may involve deliberate creation of entities of a more general essence. Such a high-level entity represents a cluster of lower-level entities. A relation between atomic high-level entities can be a projection of relations originated from the complete structure. This abstraction makes the projected relation span over shorter distances. In our case, the CFG nodes in the *macro* level abstract away the sub-structure of their associated top-level expression. By so, relations between two expression tokens of consequent statements becomes closer in the *macro* operator.

### 6.8 Hierarchic framework for general structures

In Section 3.2, we formulate the hierarchic framework for code-related tasks, where the local micro elements are chosen to be top-level expressions and the global macro elements are their associated CFG nodes. In Section 6.3 we justify this choice of the separation point in the context of our task. However, the hierarchic concept is not limited to the scope of code, and could be generalized for other problems with structured inputs.
6.8.1 Open Question: Can the hierarchic framework be effective for other tasks with structured data?

For a given graph problem, we require a pre-defined function that partitions an input graph into local sub-graphs. This function’s definition may be based on our preliminary knowledge of the characteristics of the input graphs.

6.9 Effectiveness of the expand-collapse as a graph encoding framework

In our work, we use the expand-collapse both as the hierarchic micro operator for encoding of sub-ASTs of top-level expressions, and also as the hierarchic macro operator for encoding of the CFG. For both uses, in Section 4.3, we compare their results to other graph encoders (GNNs, TreeLSTM) and to the method ablation where the collapse stage is removed. In all of the conducted experiments, the full expand-collapse method produced the highest results over all of our tested benchmarks.

6.9.1 Open Question: Can the expand-collapse framework be useful for other graph tasks?

The significantly promising results of the expand-collapse in our use cases, brings up the question whether it could be useful for other graph tasks. An inevitable related question is which expand techniques are effective for general graphs. Our graphs were special in the sense that we could extract paths easily and effectively (the CFG is a directed graph with reasonable number of paths from its single source to its single sink, and the top-level expression’s sub-ASTs typically have only few leaf-to-leaf paths).

6.9.2 Open Question: Is it possible to formulate a theoretical claim about the absence of the over-smoothing issue in the expand-collapse methodology?

As been pointed out in Chapter 1, the popular GNN message-passing approach suffers from the well-known over-smoothing issue. Practically, it withholds the GNNs-based approaches from effectively propagating information over long distances along the graph. Fortunately, the expand-collapse framework is based on sequential models, which are known for their increased ability to propagate information along the input sequence. On this background, it is worth to question whether the over-smoothing issue becomes irrelevant for the case of expand-collapse.

Let’s consider two arbitrary-chosen nodes and trace the required computation to propagate a message between them in both models. In the paths-based approach, if the expand operator emits a path that includes both of these nodes, the message propagation involves a single update of each node occurrence on this path. On the
other hand, in a message-passing approach, in order to propagate a message between nodes of distance \( d \) from each other, at least \( d \) communication iterations are required. In each iteration all the nodes in the graph are being updated. Particularly on each path between these two nodes, each node is being updated \( d \) times. These redundant updates are not required for the propagation of the message. As the excessive updates are the main cause for the GNN’s over-smoothing issue, avoiding them could entitle expand-collapse to be a plausible solution. Note that multiple node occurrences along other paths allegedly do induce additional updates to the same node. Still, performing these updates in parallel doesn’t have the same repetitive effect as the consequent updates that cause the over-smoothing effect in the GNN’s setting.
Appendix A

Appendix

A.1 Batching variable-sized indexed inputs

Our input data is formed of several types of elements; that is, each pre-processed example contains multiple kinds of entities: CFG nodes, AST nodes, tokens, symbols, identifiers, and paths. Some of the hierarchic method’s calculations involve elements of multiple kinds, and an element can participate in multiple calculations. We typically use the index of an element to address it in the computation. For example, constructing the local encodings of a CFG node requires combining all AST paths of the top-level expression sub-AST associated with it. To do so, we store dedicated mappings from the CFG node indices to their associated sub-AST node indices. Another example is a mapping between the identifiers’ indices and the AST leaves (terminals) they appear in. Moreover, the AST paths themselves are stored as sequences of AST nodes indices. The same goes for the CFG paths as well.

Usually, training and evaluating neural networks is performed over batches of examples, following the SIMD (single instruction multiple data) computational scheme to maximize the utility of the accelerated processing units and make the training feasible under the available resources. However, the preprocessed example is stored on its own, while it should reoccur in various batches during training. Therefore, the batching takes place during data loading. Whenever a collection of examples are being collated into a batch, continuous tensors are being created containing all the elements in the batch. As a result, the indices of these elements are updated. Thus, the references to them have to be fixed accordingly to retain the indexing consistency.

Unfortunately, there is neither out-of-the-box nor a known third-party solution for these issues. Devising a straightforward solution is unrealistic, especially given the variability in the number of elements for each entity kind between examples. Hence, we had to implement ad-hoc batching utilities to overcome these difficulties.

Although this implementation hasn’t brought up any substantial algorithmic difficulty or novelty, it still addresses non-trivial engineering challenges and can be valuable for future research projects with similar characteristics. We generalized and open-
sourced this implementation under github.com/eladn/tensors_data_class.
Bibliography


[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors,


Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In


עדל חזרה, המשימה האוטומטית של פעולות ניטור נחקקה עבור שיטה妁.Images התוכן
בבל. בימパーティー, התוכן הוחזק על ידי פנוות מקוד, ומאפשר את מסעויות של אוטומציה של פעולות ניטור.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למעשה, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל מבנה של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל המדות של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל המדות של פנוות בבל, ומתקדם את השיטה השומרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל המדות של פנוות בבל, ומתקדם את השיטה השמרית.
למשל, בעדכוניות האוטומטית של פעולות ניטור, נבדל המדות של פנוות בבל, ומתקדם את השיטה השמרה.
תקציר

עלאף התקדמות有意思的 בבריאות מבקרת-איכות ובואימות של תוכנה, לא만 הדבר הנミニום בשתייהLTR א務だろうそれは, על כן פועל על כל התוכנות ולהודעת את התוכנית. ליירש של ידître ייעוד התוכנית, יانخفاض הפרוטו של התוכנית, וול להשתייך לנהוג של אפקט התוכנית בודק, ולא להתקין פועלוطلعностי, ואילו חלוקות מצטבר.

הнтכניות התועדו.

ностנדו, וא באינטרנט את מימוש החישובה האוטומטית של פעולות יוט לשנים Zub, בתוכנית מתוונה.

עד שמול פעולות יוטו בתוכנת ע holster, בפרהיסטוריה מיתוג יועדו למתן להלכאת פעולות יוטו בתוכנת ההלכת, התרחשה אוטומטית של קוד דוג, שיעד את מבנהו של היגיון של תוכנת ההלכת, בפי מרידות,

למה יتشكח קצרות של היצירת של התוכנית, עוד לח הינו היגיון, וקוד בתוכנת היגיון הינו חלום, لمدة במ ожи מודל המתוכנן הבינן.

היום יטורג מהנדס חוסות של נמלים מהודר."}

בماتראקסבר, רב่อยילקטרוניקה ו başında ובאילונים של מلاء מחוון-

ו-פסק, הוחלט את שאלת היגיון. כלומר, לכל השלכות, המבוססות כאלו על תכונות שונות של מ@updateה של משותף לשני פעמים. שים פעימה, ההלכת מתחלף צלע בשפה של למגנימים. עפולה, בזמנו התוכנית של פעימה של ממגנימים, קוד מתאמה בתוכנת היגיון, למושג של ידית ומומש לשני פעימות חנויות ובית בתי שיוור מנקות.

המ纪委监ות ידוע, לרתק בתוכנות וב伸びים של ידית ומומש לשני פעימות חנויות ובית בתי שיוור מנקות, ידוע ינטילוק במשתטיב מאיו ידית ומומש ובית בתי שיוור מנקות.

למרבה המזל, שפת התוכנה מוגדרת על ידי דקדוק פורתיק, עבורי קיימים אלגוריתמים יעילים

המאפרסיים השתק ענני התוכנה לש הקטן קד רבון. שיסותiveau לקוד מתוינת בתוכנות שמתוינות ובית בתי שיוור מנקות, לתוך התוכנה, בתוכנה ובשלבים בתוכנה ומושלביולוק השתייך לבראנו של ידית ומומש ובית בתי שיוור מנקות.

ל릴ן על יני שיסות פורום סולטקה בתוכנה, יידא לשני מיני תכונת.
המחקר בוצע בהנחייה של פרופסור ערן נב, בפקולטה למדעי המחשב.

תודה

אני רוצה להודות לכל הנחה Sheila, פרופ' ערן נב, על התמיכה האкова-ספość וההנחייה המקצועית במחלקה להמחקרים. אני מעריך את האעדים והמתחביחות שנרגש למחצית אוצר, אנו מקוונים אליהם ואת התפקידים המקצועיים, היצירתיים, וה죠ונים המוחמצים שהועמודו בו במחול.

הממש הקריירה המקצועית שלי.

משפחתי הקדש, תודה על עמיות, הגיהום והמודל, הסבלנות, על הרגעיםbrero מצייד ואפשרות leaning לעתונות, התמיכה והמשתתפות של האמה, אווה, והגרות, על השרשורים ואלי שערת עלי משמרות את הת İşte, ועל השילוב בין뎠ה של האוור המחורי של שחר. תודה.

שאף להרשים אתお勧めים ב.

אני מודה לנטנינ על התמיכה והנסיון הנדיב ובasyarakatיות.
ייצוגים היררכיים לקוד

Տոբר על מחקר

לשם מוללי חלקי של הדרישות לקבלת התואר
מוניטור למדעי במדעי המחשב

אלעד נחמיאס

הנה למשתתף הטכנולוגיה — מרכז טכנולוגיה לישראל
שבט החשף"ב חיפה
ינואר 2022
ייצוגים היררכיים לקוד

آلעד נחמיאס