CAPS: Code Abstraction-Based Pre-training Strategy for Smaller Language Models for Code

Fraol Batole*, Smit S. Patel†, Aashish Yadavally † Tien N. Nguyen †, and Hridesh Rajan*
* *Department of Computer Science, Iowa State University*, Ames, IA, USA
*{fraol, hridesh}@iastate.edu
† *Computer Science Department, The University of Texas at Dallas*, Dallas, TX, USA
*{smitsoneshbhai.patel, aashish.yadavally, tien.n.nguyen}@utdallas.edu

Abstract—**Large Language Models (LLMs) offer powerful capabilities in several domains, including software engineering tasks. However, utilizing the largest models for every use case, such as code-related development, may not be necessary or practical. One reason that hinders its practicality is the model's complexity. Therefore, effective training techniques are needed to harness the power of language models.**

Our work introduces CAPS, a pre-training strategy to efficiently pre-train a transformer-based encoder-decoder model that is less complex but still powerful for source code representation. CAPS achieves comparable performance with LLMs by utilizing source code abstractions to increase code regularity.

CAPS is the first pre-training strategy for LMs that leverages different levels of abstraction in source code, including code sequences (lexical), syntax (structural), token types (part-of-speech), data types, and program dependencies (semantic). CAPS strategy leverages code abstractions for its generalization across different code occurrences. That is, two lexically different statements in two methods/projects might actually possess the same meaning/purpose. Thus, encoding the code sequences with those abstraction annotations helps increase regularity with fewer requirements on complex models and ultra-large training data. Moreover, encoding code dependencies allows a model to learn to distinguish the code occurrences where the lexical sequence is the same but with different semantics.

With CAPS, we pre-trained a small model named LITECODER and fine-tuned it for five downstream tasks: Code Clone Detection, Defect Detection, Code Translation, Code Summarization, and Code Generation. We conducted experiments to show that a much smaller model (60M number of parameters) is able to achieve comparable or relatively better results on all those tasks. The results on clone detection demonstrate that LITECODER has a competitive performance and an F1 score of 95.4%. This improvement is over large models for code, such as CodeT5+ (770M) and CodeGen-multi (350M), while our model has the least model complexity. We showed that explicitly encoding program dependency improves a model's data flow understanding.

I. INTRODUCTION

Recent advances in *deep learning* (DL) and *natural language processing* (NLP) have enabled several large language models to implicitly learn the code patterns for several code-related downstream tasks, including code clone detection, code summarization, code generation, code search, etc. Pre-trained Large Language Models (LLM), such as CodeT5 [1], GPT-4 [2], and CodeGen [3], have emerged as a powerful tool that can significantly impact developers' activities. The ability to complete or generate code is valuable to developers.

While these models offer substantial benefits, their adoption also presents challenges. First, fine-tuning LLMs for specific downstream tasks can be time-consuming and expensive. It requires a significant amount of computational resources specific to the target task. As the models grow, the cost required for fine-tuning increases substantially, thus making it impractical for organizations or developers with limited resources to leverage the benefits of fine-tuning. For example, GPT-3 contains billions of parameters and could cost tens of millions of dollars for pre-training. CodeT5+ and CodeGen-Multi contain 770M and 350M parameters, respectively. Second, the LLMs have limits on the number of input tokens and high computational complexity. LLMs often restrict the length of input text or code sequences that can be processed simultaneously. Moreover, LLMs can have computational complexities that increase quadratically with the length of the input [4]. These limitations can pose challenges when dealing with longer code snippets. Considering these challenges, it is important to carefully evaluate the justification for using ultra-large language models for code-related tasks that developers often use in their daily work on regular workstations, desktops, or laptops without high-power GPUs. While LLMs offer powerful capabilities, utilizing the largest models for every use case may not always be practical. Developers should assess the specific requirements and consider alternative solutions that balance efficiency, cost-effectiveness, and model size.

Several approaches aim to address those challenges from different perspectives. Some propose the techniques to reduce the complexity of a LLM [5], other approaches introduce the techniques to prune the input data to reduce workload [6]. In this paper, we advocate for *an approach that reduces the demands on a Language Model's complexity*, including the number of parameters and memory usage, while still preserving its comparable effectiveness when fine-tuned for a specific task. We introduce CAPS, a Code Abstraction-based Pre-training Strategy designed for source code that is capable of operating with a compact model via encoding code abstractions directly from the input. Our intuition is that a high level of code abstraction empowers a model to learn with reduced complexity due to the higher level of regularity in source code. With the abstractions on source code, the instances of a code pattern with different lexical sequences can be unified, and their regularity can be increased, thus allowing the model to generalize its knowledge across different instances of the same pattern, even with different lexical sequences. By unifying code patterns and extracting their common features, a model can focus on learning the essential aspects of the patterns rather than being overwhelmed by the variations in lexical sequences. Thus, this approach can potentially reduce the complexity requirement on the model and can generalize code patterns across training datasets.

Specifically, our insights are as follows. Source code written in a programming language has unique characteristics regarding the lexical, syntax, and semantic levels. First, *two lexically different statements in two methods or projects might actually possess the same meaning/purpose*. For example, let us consider two statements `int len = str.length()` and `int l = s.length()` in which `len` and `l` are of the same type `int`, and `str` and `s` are of the same type `String`. Both statements are the instances of the same *code pattern* with the same meaning/purpose: *"getting the length of a `String` object and assigning it to an `int` variable"*. Current LLMs are trained with an ultra-large amount of code, however, only at the lexical level. They would need a much larger amount of code to learn that the two statements have shared the same meaning, leading to more complex models.

Second, in contrast, *two statements with the same lexical sequence of code tokens might have different meanings and purposes*. For example, `x.next` can be an access to the field `next` if `x` is a `LinkedList` object. However, `x.next` in a different place could be part of the method call to the method `next()` of a `Scanner` object. As another example, the type `Document` appears in several popular libraries: `com.google.gwt.dom.-client.Document`, `org.eclipse.jface.text.Document`, and `org.w3c.dom.Document`. In different contexts, `Document` can refer to a class in one of the different libraries.

We propose the abstractions for source code to accommodate the above characteristics. In addition to the lexical tokens of source code, we integrate three different levels of abstractions for code: **token types** (syntax/structural), **code sememes** with data types (semantics), and **code dependencies** (semantics). Despite that code abstractions have been applied for other goals, we are the first to introduce them as a pre-training strategy to address the model complexity.

For syntax, the *grammar or syntactic rules* of a programming language enforces the appearance of certain tokens at a position, helping a model learn better the code patterns. For example, the requirement of the open parenthesis '(' for a function call is an indication that the prior token is a function call as in `s.length()`. It helps a model distinguish a function call from field access.

At the semantic level, we integrate both data types and program dependencies. For example, both statements `int len = str.length()` and `int l = s.length()` will be encoded as `VAR[int] = VAR[String].length()`. We expect the model to recognize this code pattern from two lexically different statements. In contrast, program dependencies enable a model to distinguish two code tokens with the same lexeme but different semantics. For example, for `x.next`, we will encode differently in the two cases of the `LinkedList` and `Scanner` classes: `FIELD [LinkedList,next]` and `CALL [Scanner,next]`. The former has a dependency between `x.next` (`CALL [LinkedList, next]`) and `x.hasNext` (`CALL [LinkedList, hasNext]`), while the latter does not have a dependency with either `next` or `hasNext` of `LinkedList`.

We have conducted several experiments to evaluate our pre-training strategy, CAPS. First, we used CAPS on the CodeT5-small model and then performed fine-tuning on four different tasks: Code Clone Detection, Code Translation, Code Summarization, and Code Generation. We compared CodeT5-small+CAPS against several baselines.

Our experimental results on the dataset, BigCodeClone [7], show that the resulting model, CodeT5-small+CAPS (let us call it LITECODER) achieved a comparable F1-score (95.4%) as the state-of-the-art LLMs (SOTAs) in code clone detection. Importantly, in comparison with the SOTA large language models CodeT5+ (770M), CodeGen-multi (350M), and CodeT5 (220M), with CAPS pre-training strategy, CodeT5-small, with less complexity (with only 60M parameters) and much less training time, can achieve a comparable performance. Similar results can be observed across all four tasks. We also performed an ablation study to show that all the levels of abstraction in our pre-training strategy contribute to its high performance. Additionally, we show that explicitly encoding code dependencies helps a LM capture better code dependencies. In brief, the contributions of this paper include:

1. CAPS is the first pre-training strategy leveraging *the code abstractions on token types, syntaxes, data types, and program dependencies*. CAPS helps *reduce the demand on model complexity*: with CAPS, a smaller model can achieve comparable performance as more complex LLMs for code.

2. An empirical evaluation of LITECODER in five downstream tasks shows that due to better learning code patterns (code clone detection), a smaller model achieves a comparable performance as the state-of-the-art LLMs.

3. An empirical study shows that all code abstractions in CAPS's strategy contribute to our model's performance.

Code and datasets are available on our website [8].

II. MOTIVATING EXAMPLE AND KEY IDEAS

A. Motivating Examples

We aim to find the abstractions that help a model to better learn code patterns at a higher level with less complexity.

First, a code pattern is defined as a sequence that frequently appears with the same meaning in source code. Some code instances of a code pattern across different locations in the same or different projects could share the same sequence of lexical tokens and maintain the same semantics. Thus, we must capture the lexical code sequences for code representation learning. For example, the code sequence that appears frequently in Java projects is `for (int i = 0; i < n; i++)`.

Observation 1 [Code Sequences]. *A code pattern can be represented in the lexical code sequences. Thus, a model needs to capture the lexical code sequences.*

Second, a code pattern can be specifically involved with a certain statement type. Thus, annotating code tokens with their token types, e.g., the keyword `if`, `for`, etc., or the identifiers, will help a model capture the patterns with those statements or identifiers. For example, in the previous code pattern with the `for` loop, the keyword 'for' will help a model distinguish it with the literal 'for' or the sub-string 'for' in a literal, etc.

Observation 2 [Token Types]. *Token types are helpful for a model to capture code patterns with specific program elements.*

Third, the syntactic rules in a programming language in the source code are important in a code pattern. It helps a model recognize and distinguish between the code tokens with the same lexical value but different meanings and roles in the source code. For example, after a function call, Java requires an open parenthesis '(', allowing a model to know that the previous token is a function name instead of a field's name. Therefore, a model can recognize that `next` in `x.next()` should belong to `LinkedList.next()` rather than `Scanner.next`. Moreover, several syntactic patterns have been reported in previous work [9]. The following is a code pattern called "Loop Collector" to add elements into an `arrayList`.

```
for ( final Element element1 :
    o.getElements()) {
  myElements.add(element1);
}
```

Observation 3 [Syntax]. *The syntax of the programming language is helpful for a model to learn a syntactic pattern with specific syntactic structures.*

Fourth, as explained in Section I, when we abstract a variable V of the type T into `VAR[T]`, it helps a model distinguish the two ambiguous cases. For example, in the code `arr.append("A")`, we will encode it as `ARRAY[String].-append(String)`. In a different project, even with a different variable's name, the model still can recognize the method call `append` for any `ARRAY` variable.

Observation 4 [Variable Names and Data Types]. *The abstraction of the variable names and the data types help a model better learn code patterns with different data types.*

Finally, between two statements, there might exist program dependencies. For example, there is a data dependency between `x.hasNext()` and `x.next()` via the variable `x`. If we encode this dependency, it will help a model distinguish between `x.next` (a field access in `Scanner`) and `x.next` in `x.next()` (a method call in `LinkedList`).

Observation 4 [Code Dependencies]. *Encoding data and control dependencies among program entities could benefit a model in distinguishing between codes with the same lexical sequence but with different meanings.*

From the above observations, we draw the following key ideas.

Key Idea 1 [Pre-training Strategy Using Abstractions on Source Code at Multiple Levels with a Less Complex Language Model]. From our observations, CAPS exploits the sources of information conveyed through multiple abstraction levels: 1) *Token types* (part-of-speech) of the code tokens, 2) *Data types*: if any, 3) *Structure* and *syntax* of the source code, and 4) *Program dependencies* among the program elements. We expect that with abstractions, a less complex model can still learn the regularity of code patterns despite the instances of the *same pattern might have different lexical sequences*.

With the code abstractions, the instances of a code pattern with different lexical sequences can be unified, and their regularity can be increased, thus, allowing the model to generalize its knowledge across different instances of the same pattern, even with different lexical sequences. With the better generalization capability, we expect that a model with less complexity can learn as well as the more complex ones (i.e., with higher numbers of parameters) in downstream tasks.

Key Idea 2 [Code Dependencies to Help Distinguish Same Code with Different Semantics] To help a model distinguish between the code tokens with the same lexical value but having different semantics, we integrate the dependencies among the code tokens. For example, in the class `LinkedList`, the methods `hasNext` and `next` often occur together. Moreover, the class `Scanner` also has the field `next`, but it does not have any field or method named `hasNext`. Therefore, we encode the code dependency between two code tokens `hasNext` and `next`. That helps the model make a distinction between `next` from `LinkedList` and `next` from `Scanner`.

Key Idea 3 [Unified Representation Learning from Multiple Abstraction Levels]. The token types, syntax, and program dependencies can be represented via sequences, trees, or graphs. However, using different types of models for each kind of data would result in a computationally heavy model and further post-processing. Therefore, to reduce the complexity, we encode the data from all channels as sequences.

For the token types, we use a tokenizer to extract the types of each token value. We use a technique in machine translation to encode a syntactic structure via a syntactic symbol called *syntaxeme* (syntactic symbols) [10]. Additionally, we also encode the meaning and data type of each token called *data types*. These two conceptual entities do not overlap but rather complement each other. This is particularly useful in cases where certain information, such as data types, might be scarce in some programming languages like Python. Therefore, in the pre-processing step, we combine them as one abstraction named ***code sememe***. For example, for the code fragment '`for j in range (len(holder))`', we produce the syntaxeme sequence '`FOR target[j] IN starred[range[Call[len, para[holder]]]] begin[:]`.' Syntaxemes have been shown to help capture the syntactic structure for a model to learn the syntactic mappings between two programming languages [10]. We expect them to help CAPS capture the syntactic structure. Lastly, we encode the dependencies of the tokens in different statements. We refer to them as ***code dependency*** (see Section III-A2). Thus, we have a unified representation of learning for all the levels of abstraction used in our pre-training strategy.

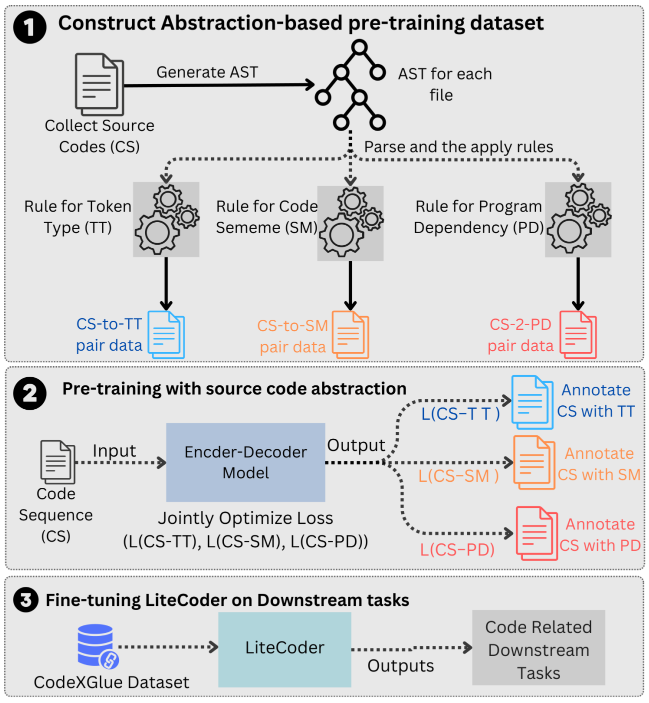
1 Construct Abstraction-based pre-training dataset
Generate AST
Collect Source Codes (CS)
AST for each file
Parse and the apply rules
Rule for Token Type (TT)
Rule for Code Sememe (SM)
Rule for Program Dependency (PD)
CS-to-TT pair data
CS-to-SM pair data
CS-2-PD pair data
2 Pre-training with source code abstraction
Input
Code Sequence (CS)
Encder-Decoder Model
Output
Jointly Optimize Loss (L(CS-TT), L(CS-SM), L(CS-PD))
L(CS−T T)
L(CS−SM)
L(CS−PD)
Annotate CS with TT
Annotate CS with SM
Annotate CS with PD
3 Fine-tuning LiteCoder on Downstream tasks
CodeXGlue Dataset
LiteCoder
Outputs
Code Related Downstream Tasks

Fig. 1: Overview of CAPS's workflow.

III. Methodology

This section presents a high-level overview of our approach, CAPS. As illustrated in Fig. 1, CAPS encompasses three core modules: (A) constructing abstraction-based pre-training data, (B) context learning via abstraction-based pre-training, and (C) fine-tuning CAPS on downstream tasks.

A. Construct Abstraction-based Pre-training Data

To facilitate effective pre-training, we construct a dataset that captures fundamental abstractions inherent in source code. Our goal is to learn representations that encode not just the program syntax but also their semantics and control/data flow. We extract three types of code features: **Token Types** (TT), **Code Sememes** (SM), and **Program Dependencies** (PD). These abstractions were chosen to provide a comprehensive, multi-level representation of code spanning its syntax, semantics, and structure. Fig. 2 illustrates each abstraction.

1) Preliminaries: We first define the key concepts used in our abstraction techniques:

Definition 1 (**Token Types (TT)**). *Token types represent the fundamental building blocks of a programming language's syntax, encompassing keywords, identifiers, literals, operators, punctuation, and comments.*

Definition 2 (**Syntactic Units (SU)**). *Syntactic units, or syntaxemes, are the fundamental building blocks of a programming language's grammar, representing the smallest meaningful units of syntax within a program [10].*

Definition 3 (**Data Type (DT)**). *Data types specify the kind of data a variable or expression can hold.*

Definition 4 (**Program Dependencies (PD)**). *Program dependencies capture the relations between code elements, revealing how different parts of the code influence each other. The two main types of dependencies are control and data dependencies.*

2) Extracting Code Abstractions: We extract the three code abstractions, TT, SM, and PD, in the following process:

a) Token Type Abstraction: Given a code snippet CS_i, we first apply lexical analysis using a language-specific tokenizer $\mathcal{T}$ to break CS_i into a token sequence $\mathcal{T}(CS_i) = [t_1, t_2, ..., t_m]$. Each token t_k is then mapped to its token type $\mathcal{F}_{TT}(t_k)$ based on the language grammar. Formally, we define a token type mapping function $\mathcal{F}_{TT} : t \rightarrow TT$ that assigns each token to one of six basic types: keyword, identifier, literal, operator, punctuation, or comment. Extracting token types allows the model to learn syntactic roles and composition patterns. Fig. 2(b) shows the result of token type extraction.

b) Code Sememe Extraction: While TT captures syntactic structure, they do not encode the rich semantic information inherent in code. To address this, we introduce the concept of code sememes, inspired by the notion of sememes in natural language as fundamental units of meaning and prior work on LLMs training [11]. Code sememes provide a semantic abstraction that captures both syntactic structure and types. By exposing these semantic properties to the model during pre-training, we enable it to learn repetitive code patterns.

We define a code sememe as a composite representation that encodes a code token's data type and syntactic unit role within the program context. Formally, for a code token t_k in a code snippet CS_i, its code sememe $SM(t_k)$ is a tuple:

$$SM(t_k) = \langle SU(t_k), DT(t_k) \rangle \tag{1}$$

where $SU(t_k)$ represents the syntactic unit (or syntaxeme) of t_k and $DT(t_k)$ represents its data type.

To extract code sememes, we parse CS_i into an AST $AST(CS_i)$, traverse it, and based on the AST context, we map each t_k to its syntaxeme $SU(t_k)$ and data type $DT(t_k)$. Fig. 2(c) shows the result of code sememe extraction. Code sememes enable the model to learn rich semantic patterns.

c) Program Dependency Abstraction: The final abstraction we extract is program dependencies, which capture the relationships between code elements in terms of control flow and data flow. This has been shown to be crucial for tasks that require deep reasoning about program behavior, such as code generation, translation, and bug detection [12].

Given a code snippet CS_i, we define the program dependency set $PD(t_k)$ for each token $t_k \in CS_i$ that has a direct control or data dependency with t_k. Formally, let $\mathcal{T}(CS_i) = [t_1, t_2, \ldots, t_m]$ be the set of tokens in CS_i. Then:

$$PD(t_k) = \{t_j \in \mathcal{T}(CS_i) \setminus \{t_k\} \mid t_j \rightsquigarrow_C t_k \vee t_j \rightsquigarrow_D t_k\} \tag{2}$$

where $\rightsquigarrow_C$ represents a control dependency and $\rightsquigarrow_D$ represents a data dependency.

Intuitively, a token t_j has a control dependency on t_k if the execution of t_k depends on the branching behavior at t_j, such as in an `if` statement or loop. A token t_j has a data

```
...
if ( i < nums.length ) {
   info = holder.get();

   list.append(info);
}

...
```

(a) Original Code

```
...
if[binary_operation[var[i], <, CALL
    [nums.length]]] begin[{]
   assign[ var[info], op[=],
       CALL[holder.get()] ]
   CALL[list.append(info), PARAM[info]] ]
end[}]
...
```

(c) Code Sememe Annotation

```
...
keyword [if] ( id[i] < id[nums.length] ) {
   id[sum] op[+] op[=] id[nums[I]];
   id[info] op[=] id[holder.get()] op[;]
   id[list] op[.] id[append(info)] op[;]
}

...
```

(b) Token Type Annotation

```
...
if ( i < nums.length ) {
   info = holder.get(); # depends on
       [control=['if'], data=[holder]]
   list.append(info); # depends on
       [control=['if'], data=[info]]
}
...
```

(d) Dependency Annotation

Fig. 2: Three Abstraction-based Pre-training Data

dependency on t_k if the value of t_k depends on the value of t_j, such as when t_j is assigned to a variable that is later used in t_k. The result of dependency extraction on CS_i is thus a sequence with an encoded comment as presented in Fig. 2(d).

To extract and encode program dependencies, we follow these steps: First, we parse the code snippet CS_i into its AST representation $AST(CS_i)$ and perform a static analysis over the AST to identify control and data dependencies between the tokens. Then, for each token t_k, we annotate it with a comment indicating its dependency set $PD(t_k)$. The comment is placed immediately after the token t_k in the original code. In the case of nested control flow structures (e.g., an ‘if’ statement within a ‘for’ loop), we annotate each statement within the nested structures with the appropriate dependencies. The final result is a sequence of tokens from CS_i with the encoded program dependency annotations, as shown in Fig. 2(d).

The effectiveness of our abstraction-based pre-training approach is validated through extensive experiments on downstream tasks (see Section V). Combining token types, code sememes, and program dependencies provides a comprehensive, multi-level code representation that enables the model to learn meaningful patterns on syntax, semantics, and structure.

B. Context Learning via Abstraction-based Pre-training

CAPS proposes three pre-training tasks to enhance the model's understanding of code abstractions: Code Sequence to Token Type (CS-TT), Code Sequence to Code Sememe (CS-SM), and Code Sequence to Program Dependency (CS-PD). These objectives are designed to capture different abstraction levels through a unified sequence-to-sequence learning.

Central to all objectives is the sequence-to-sequence modeling approach, where the model learns to map an input code sequence X to an output sequence Y. This objective design is inspired by related work on naturalizing source code [13]. The loss function for each task T is defined as:

$$\mathcal{L}_T = -\sum_{i=1}^{N} log(P(y_t|y_{<t}, X)) \tag{3}$$

Here, X is the input code sequence, y_i is the target output at position i, $y_{<i}$ represents the previous outputs, and N is the length of the output sequence. By minimizing this loss, the model learns to generate the desired output sequence conditioned on the input code and previous predictions.

1) Objective CS-TT: The CS-TT objective aims to build an understanding of code tokens and their token types. The input X is a sequence of code tokens, and the target Y is the corresponding sequence of token types. By learning to predict token types, the model acquires knowledge of the syntactic roles and composition of code elements.

2) Objective CS-SM: The CS-SM objective captures the syntactic and semantic structure of code by learning to generate code sememes. Unlike prior work, e.g., CodeFill [11] that models sememes as a next token prediction task, we formulate it as a sequence-to-sequence problem. The input X is a code sequence, and the target Y is the corresponding sequence of code sememes. By generating code sememes, the model learns the abstract syntactic and semantic structure beyond tokens.

3) Objective CS-PD: The CS-PD objective teaches the model to reason about data and control dependencies. While existing methods like GraphCodeBERT [12] explicitly encode token connections by modifying the encoder architecture, we model dependencies as a structured output sequence. The input X is a code sequence, and the target Y is the same code sequence augmented with dependency annotations.

4) Multi-task Pre-training with CAPs Loss: CAPS employs a multi-task learning approach to jointly optimize the three pre-training objectives. The final pre-training loss $\mathcal{L}_{CAPs}(\theta)$ is a sum of the individual task losses:

$$\min_{\theta}(\mathcal{L}_{CS-TT}(\theta) + \mathcal{L}_{CS-SM}(\theta) + \mathcal{L}_{CS-PD}(\theta)) \quad (4)$$

where θ represents the model parameters. Multi-task learning enables the model to share knowledge across objectives and learn robust code representations that capture multiple levels of abstraction [14].

The choice of pre-training tasks is motivated by their complementary nature in capturing different facets of code understanding. CS-TT focuses on low-level syntactic information, CS-SM captures abstract syntactic and semantic structure, and CS-PD encodes program-level dependencies. By jointly learning these objectives, CAPS aims to develop a comprehensive understanding of code beyond surface-level patterns.

C. Fine-Tuning CAPS on Downstream Tasks

The main goal of fine-tuning is to take the pre-trained model with contexts and train it for any code-related downstream tasks. Unlike the pre-training phase, the fine-tuning objective exclusively utilizes code sequences for downstream tasks. In this work, we extensively evaluate the effectiveness of CAPS on five downstream tasks: (1) clone detection, (2), defect detection, (3) code translation, (4) text-to-code generation, and (5) code summarization (the details are in Section IV).

IV. Empirical Evaluation

Our evaluation seeks to answer the following questions:

RQ1. How well does LiteCoder perform compared to state-of-the-art LLM approaches in **code clone and defect detection**? This will validate whether our pre-training strategy with code abstraction helps the model learn code patterns more effectively.

RQ2. How well does LiteCoder perform compared with the state-of-the-art LLMs on **code translation**? The answer will assess whether the pre-trained strategy with abstraction provides an advantage in understanding and translating code structures, syntax, semantics, and data flow across languages.

RQ3. How well does LiteCoder perform compared with the state-of-the-art LLM approaches on **text-to-code generation**? This evaluation can help us answer whether the pre-training objectives are beneficial for generative tasks.

RQ4. How well does LiteCoder perform compared with the state-of-the-art approaches on **code summarization**? Similar to RQ3, this evaluation will help us assess the effectiveness of CAPS on generative tasks.

The answers to RQ1–RQ4 (in)validate our hypothesis that with our pre-training objectives, a smaller model like CodeT5-small can perform at the comparable level as the larger LLMs in the downstream tasks after being fine-tuned in the same dataset. If LiteCoder performs at the comparable level as much larger LLMs, we can conclude that CAPS *helps reduce the requirement on the model complexity*. That is, the same level of performance can be achieved with a smaller model that was pre-trained with CAPS strategy.

RQ5. Does *explicitly encoding code dependencies* help a language model model capture better code dependencies? The answer will confirm that abstractions, dependencies among program elements help with code-related tasks.

RQ6. How do different pre-training objectives in CAPS affect its overall performance? The answer will confirm each designed component in CAPS. It will also confirm our hypothesis that the abstractions unify the instances with different lexical sequences of the same code patterns.

A. Experimental Methodology

Pre-training Procedure. We initialize CAPS from CodeT5-Small [15] and further pre-trained it using our pre-training objectives. LiteCoder has 60M parameters. We implement the pre-training step using Pytorch [16]. We use the following pre-training hyperparameters: batch size of 32, learning rate of 5e-5, and maximum input/target length of 1024. We then pre-trained the model using CAPS objectives for 30K steps. To avoid catastrophic forgetting, we alternate between each language and each task after 5k steps. For example, if we begin with the CS-TT task in Python, the next 5k training data points will be for CS-TT in Java. Through an ablation study, we confirmed that the order of tasks or language does not affect performance. Therefore, we pre-train CAPS in the following order: (1) CS-TT, (2) CS-SM, and (3) CS-PD. The empty token index is assigned '-100' to ensure padding tokens are excluded during loss computation. We used 4 A100 GPUs with 40GB memory to pre-train and fine-tune LiteCoder on different downstream tasks.

Pre-training Data. Following related works [1], [12], [15], we used the CodeSearchNet [17] as the pre-training dataset for LiteCoder. Specifically, we filtered that dataset to select Python and Java files that can be tokenized and parsed. As a result, the total number of files we used is 453K Python and 495K Java files. We then used the Tokenizer, JavaLang, and TreeParser libraries to extract the token type, code sememe, and program dependency. Note that the pre-training dataset for different models could be different; however, the fine-tuning dataset has to be similar for a fair evaluation. We explain the fine-tuning dataset in the following part.

Fine-Tuning Procedure. We selected five experiments for downstream applications (three tasks, each for code understanding and generative tasks). In particular, for code understanding tasks, we use code clone detection, defect detection, and code translation tasks. For generative tasks, we use text-to-code generation and code summarization tasks. We follow the standard fine-tuning protocol and implementation provided by CodeXGlue [7] to ensure a fair evaluation. This entails utilizing the same fine-tuning dataset, hyperparameters, and evaluation metrics as those employed in the baselines. In subsequent sections, we provide details regarding the dataset, evaluation metrics, and basic hyperparameters. We direct interested readers to prior literature on fine-tuning details and the specifics [7].

Fine-Tuning Data. We use the following canonical datasets for the five downstream tasks. We use the BigClone dataset for code clone detection [18]. For defect detection, we use the Devign [19] dataset. Defect detection is a classification task to identify if a code may attack software systems, like resource leaks, use-after-free vulnerabilities, and DoS attacks. We use the dataset provided by CodeXGlue [7] for the code translation. The dataset contains two formats. The first is for translating Java to C#, while the second is for C# to Java. We used the CONCODE text-to-code generation provided by CodeXGlue [7] for the code generation task. It has textual descriptions with the goal of generating Java code. Lastly, we used the CodeXGlue code-to-text on Python and Java code for the code summarization task [7].

Evaluation Metrics. We use the following metrics to evaluate the code translation task. Exact Match (EM) computes accuracy based on perfect similarity between prediction and ground truth. Syntax Match (SM) measures the similarity of prediction and ground truth AST, focusing on the code's syntactic structure. Dataflow match (DM) measures the similarity of dataflow edge between candidate and ground truth. CodeBleu integrates the other metrics for a collective measure.

V. Empirical Results

A. Comparison on Code Clone and Defect Detection (RQ1)

Baselines. The baselines include encoder-decoder models like CodeT5 (60M) [15], CodeT5+ (220M) [20], and encoder-only models like CodeBert [21] and GraphCodeBert (125M) [12]. They represent recent advances in code representation learning and have demonstrated strong performance on various code understanding tasks, making them relevant comparisons for LiteCoder. For CodeT5-small, we run fine-tuning for all experiments and present both the reported and rerun results.

Experimental Setup. In this research question, we compare the effectiveness of our approach with ten state-of-the-art language models for code clone and defect detection. To evaluate the efficacy of our pre-training strategy, we fine-tune LiteCoder pre-trained model for the code clone and defect detection tasks. For clone detection, we use a batch size of 16, a maximum source length of 320, a target length of 128, and fine-tune for 1 epoch with an initial learning rate of 5e-5. For defect detection, we change the maximum source length to 512, target length to 3, and fine-tune for 10 epoch.

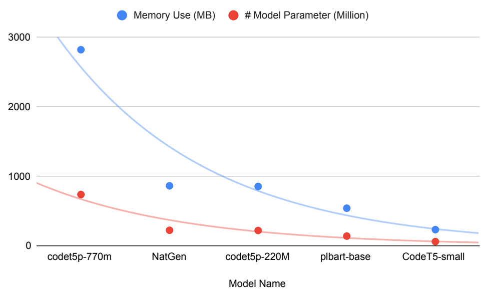

Fig. 3: Memory and # Model Parameter Comparison.

Model complexity. Fig. 3 illustrates the significant difference in complexity between CodeT5-small/LiteCoder and the

TABLE I: Results on Code Clone Detection

Model Name	Clone Detection			Defect Detection
	Recall	Precision	F1-score	EM
CodeBERT (125M)	94.7	93.4	94.1	62.1
GraphCodeBERT (125M)	94.8	95.2	95.0	-
UniXcoder (125M)	92.9	**97.6**	95.2	-
CodeGen-multi (350M)	94.1	93.2	93.6	63.1
PLBART (140M)	94.8	92.5	93.6	63.2
CodeT5 (220M)	95.1	94.9	95.0	65.8
CodeT5p (220M)	96.4	94.1	95.2	**66.1**
CodeT5p (770M)	**96.7**	93.5	95.1	66.7
CodeT5p-small **(60M)** (Reported)	94.0	93.3	93.6	63.4
CodeT5p-small **(60M)**	94.5	94.6	94.3	62.7
LiteCoder **(60M)**	95.6	95.3	**95.4**	63.8

baseline models. CodeT5-small and LiteCoder has 60M parameters, which is 95.45% smaller than CodeT5+ (770M), the largest model in the comparison. Moreover, CodeT5-small exhibits the least memory usage among all the models, making it suitable for deployment on computationally less intensive hardware. This highlights the potential of CAPS to achieve competitive performance with a much smaller model size.

Results of Code Clone Detection. Table I presents the result in code clone detection. We can see that **CodeT5-small pre-trained with CAPS (LiteCoder) achieves the same level of performance comparable to the baseline models in all metrics**. All the results are above 90%, indicating a high level of performance. This highlights that pre-training CodeT5-small with CAPS helps capture semantically equivalent programs, which is important for clone detection.

Since we pre-trained the CodeT5-small model with CAPS, comparing it against the same model that has not been pre-trained with CAPS is insightful. As seen in Table I, although CodeT5-small has the same architecture and number of parameters, its results are lower than pre-training with CAPS. Thus, the results show the advantages of CAPS in those tasks.

Example 1 (different lexemes but same meaning) (Clones)

```
public static byte[] ComputeForBinary(String ThisString)
    throws Exception {
   byte[] Result;
   MessageDigest MD5Hasher;
   MD5Hasher = MessageDigest.getInstance("MD5");
   MD5Hasher.update(ThisString.getBytes("iso-8859-1"));
   Result = MD5Hasher.digest();
   return Result;
}
```

```
private static String digest(String buffer) {
      try {
         MessageDigest md5 =
               MessageDigest.getInstance("MD5");
         md5.update(buffer.getBytes());
         return new String(encodeHex(md5.digest(key)));
      } catch (NoSuchAlgorithmException e) {
      }
      return null;
}
```

Fig. 4: Example of Code Clones Detected by LiteCoder

To better understand CAPS's effectiveness, let us consider the example in Fig. 4, where the model pre-trained with CAPS was able to correctly identify the clones. In this example, the two programs are lexically different, e.g., different function names in line 1. The functions operate on `MD5Hasher` and `md5` in line 3, which have different variable names. Thus, to detect clones precisely, it is important to recognize patterns even though the statements include different lexemes.

Results of Defect Detection. Table I presents the results of defect detection, where the effectiveness is measured by the Exact Match (EM) metric. LiteCoder achieves an EM score of 63.8%, outperforming 4 out of the 10 baseline models, including CodeBERT (125M), CodeGen (350M), and PLBART (140M). Notably, LiteCoder surpasses its initialization model, CodeT5-small, by 1.1 percentage points despite having the same architecture and number of parameters. This improvement can be attributed to our pre-training strategy, which enhances the model's ability to capture code semantics.

TABLE II: Code translation result. EM: exact match; SM: syntax match; DM: data-flow match; CB: CodeBlue

Approach	**Java → C#**				**C# → Java**			
	EM	**SM**	**DM**	**CB**	**EM**	**SM**	**DM**	**CB**
PBSTM (-)	12.5	-	-	42.7	16.1	-	-	43.5
CodeBERT (119M)	59.0	-	-	85.1	58.8	-	-	79.4
SPT-Code (262M)	64.1	-	-	-	60.2	-	-	-
PLBART (350M)	64.6	-	-	87.9	65.0	-	-	**85.3**
CodeT5 (reported)	65.9	-	-	-	66.9	-	-	-
GraphCodeBert (125M)	59.4	-	-	-	58.8	-	-	-
CodeT5 (60M) (Reported)	64.1	-	-	-	65.0	-	-	-
CodeT5 (220M)	65.9	90.4	91.9	87.8	66.0	90.4	88.9	84.4
NatGen (220M)	**66.2**	**91.0**	**92.0**	**88.1**	**67.3**	**91.0**	89.8	**85.2**
CodeT5 (60M)	56.0	86.2	87.8	82.6	60.2	86.7	87.9	81.6
LiteCoder (60M)	62.0	88.0	90.0	83.9	62.0	88.4	**90.2**	84.0

B. Comparison on Code Translation Tasks (RQ2)

Baselines. For code translation tasks, we compare our approach against the following state-of-the-art baseline models, selected based on the work by Chakraborty et al. [13] such as SPT-Code (262M) [22], a transformer-based model; and NatGen (220M) [13], a language model for code generation.

Experimental setup. For this experiment, we use the CodeT5-small (60M parameters) model pre-trained with CAPS. We fine-tune this pre-trained model on the CodeXGlue dataset [7], which includes Java-to/from-C# translation tasks. It is noteworthy that while the pre-training corpus did not include C# code, this evaluation also serves to assess whether CAPS pre-training leads to catastrophic forgetting or retains knowledge from prior training. To fine-tune, we use the learning rate as 5e-5, the batch size is 32, the source length is 320, and the target length is 128. We run the model for 30 epochs until it converges.

Model complexity. We utilize the same model with same complexity as RQ1.

Results of Code Translation Tasks. Table II presents the results for Java-to-C# and C#-to-Java translation tasks. LiteCoder demonstrates competitive performance compared to the state-of-the-art models, despite having fewer parameters.

In the Java-to-C# translation task, LiteCoder achieves an exact match (EM) score of 62.0%, which is 4.2 percentage points lower than the best-performing model, NatGen (220M). However, it is important to note that NatGen has nearly four times more parameters than LiteCoder (60M). When compared to the models with a similar number of parameters, such as CodeBERT (119M) and GraphCodeBERT (125M), LiteCoder improves over them by 3.0 and 2.6 percentage points, respectively.

For the C#-to-Java translation task, LiteCoder exhibits a similar trend. It achieves an EM score of 62.0%, which is 5.3 percentage points lower than NatGen (220M). Nevertheless, LiteCoder surpasses CodeBERT and GraphCodeBERT by 3.2 percentage points each, demonstrating its effectiveness in code translation despite its smaller size.

Notably, LiteCoder shows a relatively better data-flow match (DM) score of 90.2% in the C#-to-Java translation task compared to the state-of-the-art model NatGen, which achieves 89.8%. This improvement can be attributed to the program dependency component in CAPS, which explicitly captures control and data dependencies during the pre-training phase. By teaching the smaller model to recognize these dependencies, LiteCoder enhances its ability to understand and preserve data flows during the translation process.

Furthermore, LiteCoder outperforms the vanilla CodeT5 model with the same number of parameters (60M) by 6.0 and 1.8 percentage points in the EM scores for Java-to-C# and C#-to-Java translation tasks, respectively. This improvement demonstrates the effectiveness of the CAPS pre-training strategy in enhancing the model's code translation capabilities.

Furthermore, it is worth highlighting that LiteCoder also achieves competitive results in other metrics while exhibiting significantly lower memory usage and number of parameters as seen in Fig. 3. In conclusion, these results underscore that the pre-training of CodeT5-small with CAPS retains knowledge. The results indicate that pre-training a smaller model with CAPS for different programming languages and downstream tasks is a promising avenue for development.

C. Comparison on Text-to-Code Generation (RQ3)

TABLE III: Text-to-Code Generation result. EM: Exact Match, SM: Syntactic Match, DM: Dataflow Match, CB: CodeBleu. '-' indicates that the model parameters are not present.

Approach	**Metrics**			
	EM	**SM**	**DM**	**CB**
Seq2Seq (-)	3.05	-	-	26.39
Guo et al. [23] (-)	10.05	-	-	29.46
Iyer et al. [24] (-)	12.2	-	-	-
GPT-2 (117M)	17.3	-	-	29.69
CodeGPT (124M)	20.1	-	-	35.98
PLBART (140M)	18.75	-	-	38.52
CodeT5-base (220M)	**22.3**	-	-	43.0
CodeT5-small (**60M**) (Reported)	21.55	-	-	41.39
NatGen (220M)	**22.3**	45.59	**46.87**	43.73
CodeT5-small (**60M**)	21.05	74.9	29.26	59.76
LiteCoder (**60M**)	21.30	**74.34**	45.44	**62.80**

Baselines. We compare our approach against large models used as baselines in recent SOTA models on code generation task [13]. The baselines are fine-tuned and evaluated using the same canonical benchmark as our work. Therefore, the results reported are taken from the CodeXGLUE [7] leaderboard and NatGen work [13]. The benchmark has the text description to generate Java code. We have included the following decoder-based models: Seq2Seq (-) [13], Guo et al. [23], Iyer et al. [24], GPT-2 (117M) [7], and CodeGPT (124M) [25].

Experimental setup. We fine-tune LITECODER using the same procedure and benchmark as in CodeXGlue [7]. In particular, the learning rate is 5e-5, the batch size is 32, the source length is 320, and the target length is 128. We run the model for 30 epochs until it converges.

Model complexity. We utilize the same model with same complexity as RQ1.

Results of Text-to-Code Generation Tasks. Table III presents the results for the text-to-code generation tasks. LITECODER achieves an exact match (EM) score of 21.30%, which is comparable to the state-of-the-art models CodeT5-base (22.3%) and NatGen (22.3%), despite having significantly fewer parameters (60M vs. 220M). Importantly, LITECODER shows comparable results and outperforms vanilla CodeT5-small (60M) by 3 percentage points on the CodeBleu score, demonstrating the effectiveness of the CAPS pre-training strategy.

In terms of syntactic match (SM), LITECODER achieves a score of 74.34%, surpassing all the baseline models except for CodeT5-small, which achieves a slightly higher score of 74.9%. This strong performance in syntactic matching can be attributed to the inclusion of code sememes in the CAPS pre-training strategy, which explicitly exposes the model to the syntactic structure of the programming language.

D. Comparison on Code Summarization Task (RQ4)

In this experiment, we evaluate the effectiveness of CAPS on a generative task that distinguishes itself from the preceding tasks as it involves creating textual descriptions.

Baselines. We use the following baselines: PLBart, CodeT5, and NatGen [13].

Experimental setup. As with the previous RQs, we ran LITECODER on code summarization task. Following baseline models, we use the CONCODE [7] dataset to fine-tune and evaluate LITECODER. Since our pre-training strategies were only introduced for Java and Python, we only evaluated the code summarization approach in those two languages. To fine-tune, we use the learning rate as 5e-5, the batch size is 32, the source length is 320, and the target length is 128. We run the model for 15 epochs until it converges.

Model complexity. We utilize the same model with same complexity as RQ1.

Results of Code Summarization Tasks. Table IV presents the results of LITECODER on the code summarization task. Although LITECODER achieves competitive performance compared to some of the baselines, it falls short on Python and Java compared to the larger models such as CodeT5+ and NatGen. Code summarization is a challenging task that involves generating textual descriptions from code, which is not explicitly emphasized in CAPS's objectives.

TABLE IV: Results on Code Summarization in BLUE scores

Languages	Approach						
	PLBart (140M)	**CodeT5-base (220M)**	**CodeT5+ (770M)**	**NatGen (220M)**	**CodeT5 (60M) (Reported)**	**CodeT5 (60M)**	**LITECODER (60M)**
Python	19.3	19.56	**20.47**	20.09	20.04	19.44	19.45
Java	18.45	20.31	**20.83**	20.38	20.09	19.35	19.54

E. Probing of CS-PD Pre-training Objective (RQ5)

Experimental setup. Similar to [26], we adopt a token perturbation-based probing approach for this purpose. Code dependencies are important in our pre-training strategy as they help a model distinguish the code tokens with the same lexical sequence but having different meanings. In this experiment, we aim to show that a model learns the code dependencies via our pre-training objective.

Specifically, we show the performance gains by the inclusion of the Program Dependency pre-training objective, i.e., between rows (3) and (4) in Table V. As a result, we chose to analyze and interpret the structural knowledge encoded within LITECODER. Due to its parameter-free design paired with better interpretability, we adopt a *token perturbation*-based probing approach for this purpose. The primary concept behind this probing technique is to evaluate the influence that one code token has on another within a dependency relation. For example, consider the code sequence: $\mathbf{c} = \{c_1, c_2, ..., c_N\}$. Among these, let us assume a program dependence relation between the code tokens $c_i \rightarrow c_j$. When $\mathbf{c}$ is input to a pre-trained model M with network parameters θ, it contextualizes all tokens in $\mathbf{c}$, wherein the latent representation for c_i can be denoted with $M^\theta(\mathbf{c})_i$. The impact of c_i on c_j can be perturbedly computed as follows: first, replace c_i in $\mathbf{c}$ with `[MASK]` to compute $M^\theta(\mathbf{c} \setminus \{c_i\})_i$; next, replace both c_i and c_j in $\mathbf{c}$ with `[MASK]` to compute $M^\theta(\mathbf{c} \setminus \{c_i, c_j\})_i$; finally, compute the distance between both these latent representations for c_i, i.e., between $M^\theta(\mathbf{c} \setminus \{c_i\})_i$ and $M^\theta(\mathbf{c} \setminus \{c_i, c_j\})_i$. Mathematically:

$$Impact(c_i, c_j) = \mathcal{L}_p \left\{ M^\theta(\mathbf{c} \setminus \{c_i\})_i, \quad M^\theta(\mathbf{c} \setminus \{c_i, c_j\})_i \right\} \tag{5}$$

where $\mathcal{L}_p\{.\}$ denotes p-norm distance.

TABLE V: Impact of Program Dependency pre-training objective via token perturbation-based probing

Pre-Training Objective	Metric	Program Dependence Relations			
		for	*if-else*	*try-except*	Overall
w/o CS-PD	$\mathcal{L}_1\{.\}$	44.91	22.2	26.68	25.18
	$\mathcal{L}_2\{.\}$	3.53	1.75	2.1	1.98
w/ CS-PD	$\mathcal{L}_1\{.\}$	60.55	22.2	30.47	**37.23**
	$\mathcal{L}_2\{.\}$	4.75	1.75	2.39	**2.92**

Results. In Table V, we present the average impact thus computed for three frequently occurring token-level control dependency relations: `for`, `if-else`, and `try-except`. For convenience, we measure the impact of a code token split into multiple sub-tokens by considering just the impact given by the first token. We employ both $\mathcal{L}_1$ and $\mathcal{L}_2$-norm distance metrics for two pre-training objectives in CAPS with different network parameter settings: (a), without CS-PD (say, M^{θ_1}), i.e., CS-TT + CS-SM; (b) with CS-PD (say, M^{θ_2}), i.e., CS-TT +

TABLE VI: Impact of pre-training objectives

Method	Tasks		
	Defect Detection (EM)	**Translate (EM/SM/DM/CB)**	**Clone Detection (Rec/Pr/F1)**
LiteCoder	**63.8**	**62.0/88.4/90.2/84.0**	95.1/**95.3/95.4**
- w/o CS-TT	55.6	58.6/82.5/78.8/76.0	**95.4**/87.5/91.3
- w/o CS-SM	56.6	57.6/82.2/79.2/75.8	95.2/87.8/91.3
- w/o CS-PD	57.2	57.9/82.7/79.5/76.2	95.1/88.5/91.7

CS-SM + CS-PD. We can see that LITECODER with the pre-training setting that includes CS-PD records a higher impact than that without CS-PD for all dependencies. Moreover, the approach is also distance metric-invariant, as we observe an overall improvement of 47.86% and 47.47% for both $\mathcal{L}_1$ and $\mathcal{L}_2$-norm distance metrics, respectively. Thus, based on the impact scores computed via two-stage perturbation, we can see that **explicitly encoding CS-PD helps successfully learn data and structural dependencies between the code tokens**. Such an analysis can also be extended to other relations in future work.

F. Impact of Pre-training Objectives (RQ6)

Experimental setups. We conducted an ablation study focused on three variants of LITECODER, each without one pre-training objective. Specifically, the models are trained as follows: (1) w/o CS-TT task, (2) w/o CS-SM, and (3) w/o PD. Prior works on LMMs for code [12], [20] inspire the experimental design to construct the models. We select three downstream tasks (i.e., defect detection, clone detection, and translation) to evaluate the ablation study.

Results. Table VI presents the results of the ablation study, showcasing the impact of each pre-training objective on the model's performance in different tasks. The complete LITECODER achieves the best results across all tasks, demonstrating the effectiveness of the unified pre-training strategy.

While removing the CS-TT objective leads to the most significant performance drop, with a decrease of 8.2 points in the defect detection EM score and an average decline of 7.2 points across the translation metrics, the CS-SM and CS-PD objectives also play crucial roles. The CS-SM objective, which captures semantic information by combining syntactic units and data types, results in a 7.2-point drop in the defect detection EM score and an average decrease of 7.5 points in the translation metrics when removed. On the other hand, the CS-PD objective, which encodes PD, demonstrates its effectiveness in capturing code structure and behavior, as evident from the 10.7-point drop in the translation DM score when removed.

VI. LIMITATIONS AND THREATS TO VALIDITY

Noise introduced by the parser: We only build the parser to extract features from common Python and Java statements. However, it's important to acknowledge that this focused approach may introduce noise if unsupported statements are parsed using our tool. To comprehensively understand the improvement, it will be crucial to expand the parser's capabilities to accommodate a broader range of statements in the future.

CAPS's Parameters and Training: CAPS's parameters were initialized from the CodeT5-small model and then further trained for 30K steps. One might question whether the improvement in the performance is due to the increased training or the introduction of our pre-training objectives. However, as noted by [13], CodeT5 did not show improved performance with further training using the original CodeT5 objective.

External Threats: It is important to note that CAPS is exclusively pre-trained on Python and Java benchmarks similar to PLBart [27] and DietCode [6], which could potentially limit applicability to other PL's. Nevertheless, it is feasible to develop a parser for other languages to extract the abstractions. Currently, we only evaluated our proposed pre-training strategy in one model CodeT5-small and one fine-tuning dataset. More experiments are needed to evaluate the impact of CAPS pre-training on different models and other datasets.

VII. RELATED WORK

Language models for Code Understanding: Several approaches have leveraged deep learning to detect code clones and code patterns [28]–[31]. While some use graph-based techniques [32], [33], others utilize a Neurosymbolic-based approach [34] and large language model [35]. GraphCodeBert [12] introduces a data-flow aware encoder-based pre-trained model. This model encodes source code, comments, and data flow information to predict data flow edges between variables. In contrast, our approach introduces a pre-training objective that extends beyond variables to detect control dependencies in the code.

Language models for Code Generation: Several approaches have applied ML/DL for code representation. DNN4C [36] uses three code sequences at lexical, syntactical, and semantic levels with the DNN language model for CC. TravTrans+ [37] encodes the AST information to feed it into a transformer to take advantage of code structures. In contrast, CAPS further abstracts the AST to take sequences as input. CodeFill [11] aims to integrate both the natural language channel (token value sequences) and the syntactic channel (token type sequences). Compared to our technique, DNN4C and CodeFill introduce a causal language modeling objective to predict the next token, where CAPS designs the objective as a sequence-to-sequence language modeling. [38] propose to learn to complete code with a sketch. The goal of their model is to learn to minimize the number of holes. UniXCoder [39] introduces a unified learning approach to generate code based on the AST and code comments. Compared to our work, UniXCoder directly converts the AST into a sequence, while our approach first abstracts away the tree. NatGen [13] proposes a pre-training objective to "naturalize code" using a transformer encoder-decoder model. Lastly, DietCode [6] introduced an approach to pruning the input to the model with the goal of reducing the training time of pre-trained models. While DietCode focuses on minimizing input size for more efficient training, our goal is pre-training with code abstractions.

VIII. Conclusion

This paper introduces CAPS, a novel pre-training objective designed to enhance source code representation for a smaller language model called LiteCoder. By explicitly encoding programming language features, it improves LiteCoder's ability to represent source code with reduced complexity. CAPS achieves competitive performance across various downstream tasks, including code clone detection, defect detection, code translation, code summarization, and text-to-code generation. Using a 60M-parameter encoder-decoder transformer architecture pre-trained with CAPS on Python and Java, LiteCoder demonstrates promising results, including a 95.4% F1 score in code clone detection. Despite its lower complexity, LiteCoder performs comparably well in five different code-related tasks. Future work may explore additional programming language-based abstractions and evaluate different architectures and model sizes.

References

[1] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[2] "OpenAI," https://openai.com/.

[3] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint*, 2022.

[4] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," *arXiv preprint arXiv:2312.00752*, 2023.

[5] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[6] Z. Zhang, H. Zhang, B. Shen, and X. Gu, "Diet code is healthy: Simplifying programs for pre-trained models of code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1073–1084. [Online]. Available: https://doi.org/10.1145/3540250.3549094

[7] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[8] "Replication package for caps." 08 2023. [Online]. Available: https://zenodo.org/records/10858998

[9] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 819–830.

[10] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 585–596. [Online]. Available: https://doi.org/10.1109/ASE.2015.74

[11] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. Association for Computing Machinery, 2022, p. 401–412. [Online]. Available: https://doi.org/10.1145/3510003.3510172

[12] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[13] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "Natgen: generative pre-training by "naturalizing" source code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 18–30.

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

[15] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[17] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[18] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[19] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[20] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint*, 2023.

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

[22] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: Sequence-to-sequence pre-training for learning source code representations," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2006–2018. [Online]. Available: https://doi.org/10.1145/3510003.3510096

[23] D. Guo, D. Tang, N. Duan, M. Zhou, and J. Yin, "Coupling retrieval and meta-learning for context-dependent semantic parsing," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 855–866. [Online]. Available: https://aclanthology.org/P19-1082

[24] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 1643–1652. [Online]. Available: https://aclanthology.org/D18-1192

[25] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[26] Z. Wu, Y. Chen, B. Kao, and Q. Liu, "Perturbed masking: Parameter-free probing for analyzing and interpreting bert," *arXiv preprint arXiv:2004.14786*, 2020.

[27] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668. [Online]. Available: https://aclanthology.org/2021.naacl-main.211

[28] M. Zubkov, E. Spirin, E. Bogomolov, and T. Bryksin, "Evaluation of contrastive learning with various code representations for code clone detection," *arXiv preprint arXiv:2206.08726*, 2022.

[29] D. Perez and S. Chiba, “Cross-language clone detection by learning over abstract syntax trees,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 518–528.

[30] M. A. Yahya and D.-K. Kim, “Cross-language source code clone detection using deep learning with infercode,” *arXiv preprint arXiv:2205.04913*, 2022.

[31] S. N. Pinku, D. Mondal, and C. K. Roy, “Pathways to leverage transcompiler based data augmentation for cross-language clone detection,” *arXiv preprint arXiv:2303.01435*, 2023.

[32] D. Yu, Q. Yang, X. Chen, J. Chen, and Y. Xu, “Graph-based code semantics learning for efficient semantic code clone detection,” *Information and Software Technology*, vol. 156, p. 107130, 2023.

[33] A. Nair, A. Roy, and K. Meinke, “funcgnn: A graph neural network approach to program similarity,” in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[34] K. Hasija, S. Pradhan, M. Patwardhan, R. K. Medicherla, L. Vig, and R. Naik, “Neuro-symbolic zero-shot code cloning with cross-language intermediate representation,” *arXiv preprint arXiv:2304.13350*, 2023.

[35] A. Zhang, L. Fang, C. Ge, P. Li, and Z. Liu, “Efficient transformer with code token learner for code clone detection,” *Journal of Systems and Software*, vol. 197, p. 111557, 2023.

[36] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, “A deep neural network language model with contexts for source code,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 323–334.

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

[38] D. Guo, A. Svyatkovskiy, J. Yin, N. Duan, M. Brockschmidt, and M. Allamanis, “Learning to complete code with sketches,” in *International Conference on Learning Representations*, 2022.

[39] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “UniXcoder: Unified cross-modal pre-training for code representation,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 7212–7225. [Online]. Available: https://aclanthology.org/2022.acl-long.499