Type Prediction in Source Code with Graph Neural Network Embeddings

Vitaly Romanov

Introduction

Source code analysis with ML

Program Translation

Summarization

Similarity Search

ML for Source Code

Program Element Naming

Classification

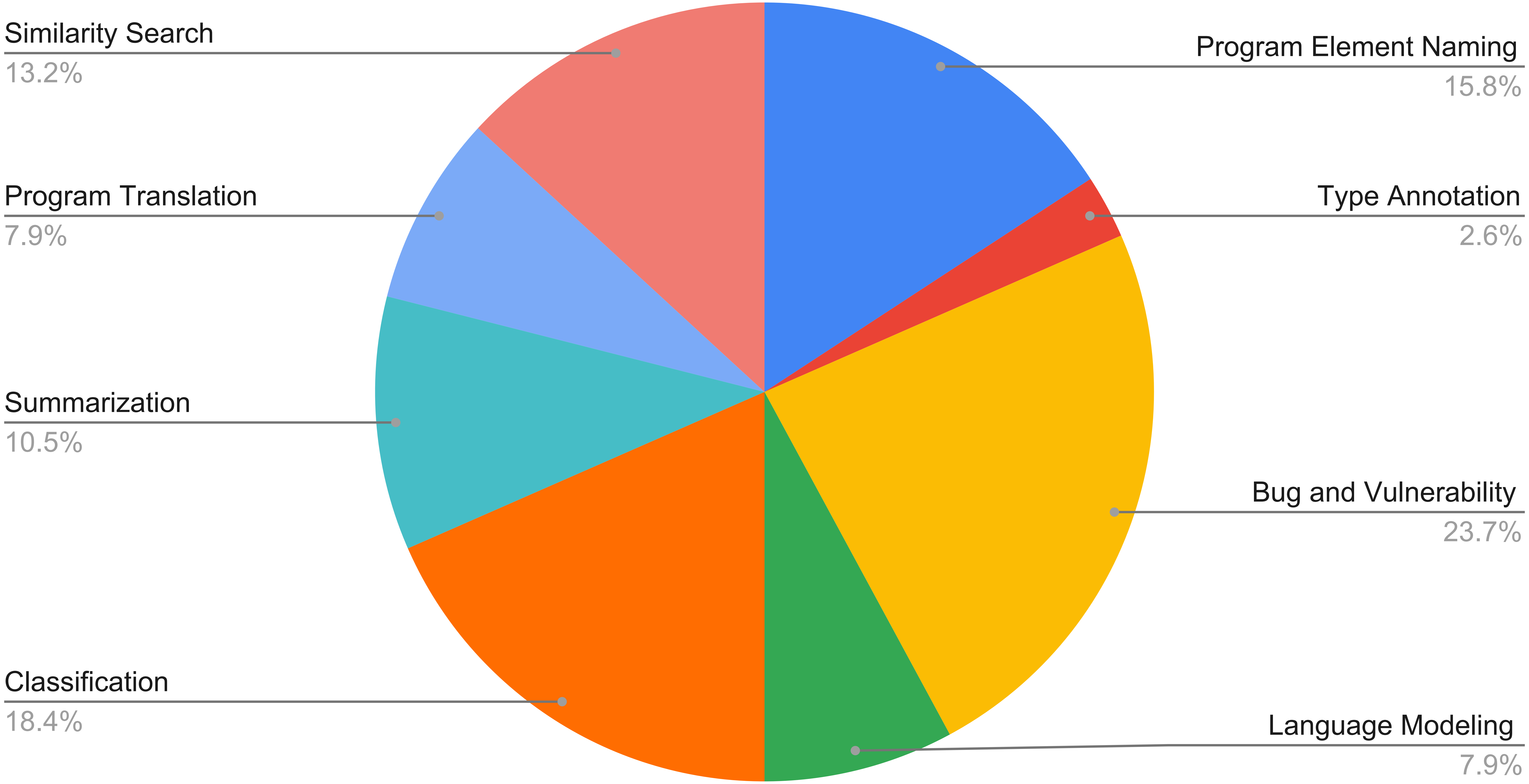
Language Modeling

Type Annotation

Bug and Vulnerability Detection

Introduction

Source code analysis with ML



Introduction

Source code analysis with ML

```
def sum(xs: List[Int]) -> Int:
   val acc = 0
   for x in xs:
      acc += x
   return acc
```

Type prediction

```
def sum(xs: List[Int]) -> ???:
   acc = 0
   for x in xs:
      acc += x
   return acc
```

Variable Name Prediction

```
def sum(xs):
   ??? = 0
   for x in xs:
      ??? += x
   return ???
```

Translation

```
int sum(int [] xs){
   int acc = 0;
   for(x:xs)
      acc += x;
   return acc;
}
```

Clone detection

```
def sum(xs):
   return sum(xs)
```

Summarization

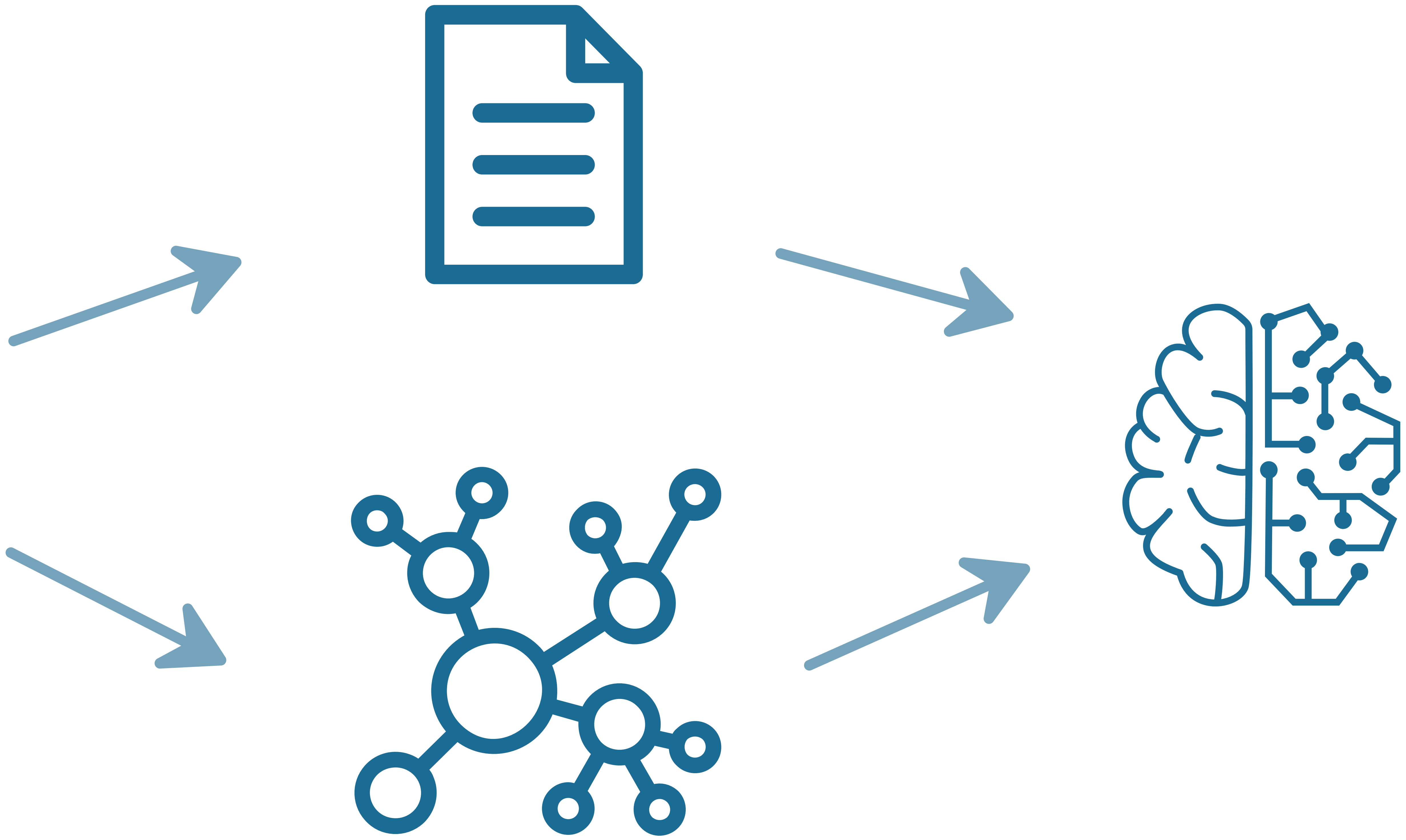
Summary:
A function to compute a sum of elements in a list

Introduction

Source code analysis with ML

Majority of ML for source code is done using token representation of code (Naturalness hypotheses, Allamanis 2017)

```
from ExampleModule import ExampleClass

instance = ExampleClass(None)

def main():
    print(instance.method1())

main()
```



Recent approaches explore using graph representation of source code

Introduction

Summary

- All ML tasks for source code require some level of understanding to process source code
- It would be beneficial to have a shared model
- Such models called pre-trained model
- Pre-trained models are widely adopted for images and text (BERT, ResNet)
- Existing pre-trained models for code are borrowed from NLP
- Pre-trained models specific for source code are not well studied, hard to evaluate
- Good pre-trained models will enable faster development of intelligent tools

Introduction

Contributions

- Trained Graph Neural Network (GNN) embeddings for source code
- Performed experiments with different pre-training objectives
- Tested pre-trained embeddings on the task of type prediction for Python variables
- Studied the impact of GNN embedding dimensionality on the type prediction quality
- Tested a technique for combining existing NLP-based approaches with GNN embeddings

Existing Work

Evaluation Challenges

- Pre-trained models enable faster training
- Pre-trained models allow achieve better prediction quality on various tasks
- They are not straightforward to evaluate due to disconnect between pre-training task and evaluation tasks
- Evaluation tasks for source code are not well defined

Existing Work

Results

Method	Pre-training tasks	Evaluation Tasks
GraphCodeBERT (Guo et al, 2020)	MLM, Data-flow edges, token & data-flow alignment	code search, clone detection, code translation
PLBART (Ahmad et al, 2021)	MLM	code summarization, code generation, code translation, code classification
σ1 (Liu et al, 2021a)	Metapath Random Walk, Heterogeneous Information	method name prediction, link prediction
CuBERT (Kanade et al, 2019)	MLM	Variable Misuse
AWD-LSTM-LM (Trevett et al, 2021)	Language Modeling	Code Search, Method Name Prediction
SLM (Alon et al, 2020)	Predicting AST Nodes, Predicting Subtokens	Autocompletion
TLM (Yang et al, 2019)	Token Prediction	Autocompletion
Ours	Name Prediction (similar to MLM), Edge Prediction, Node Type prediction	Type Prediction

Background

Classical NLP (pre 2008)

- Tokenization
- Feature extraction
- Discrete feature space explosion

Fratures

Token	def	main	(	arg1	,	arg2	)	:	return	1
Suffix	ef	in	(	g1	,	g2	)	:	rn	1
Prefix	de	ma	(	ar	,	ar	)	:	re	1
Token Position -1		def	main	(	arg1	,	arg2	)	:	return
Token Position +1	main	(	arg1	,	arg2	)	:	return	1	
Is number	0	0	0	0	0	0	0	0	0	1
Is punctuation	0	0	1	0	1	0	1	1	0	0

Classifier

O	O	O	U-Int	O	U-Str	O	O	O	O

Background

NLP after ~2005-2008

- To prevent feature space explosion, use dimensionality reduction
- Obtain token embeddings by decomposing token collocation matrix from a large dataset (**pre-training**)

  $A = E \cdot E^T, A_{ij} = 1$ if token t_i collocates with token t_j

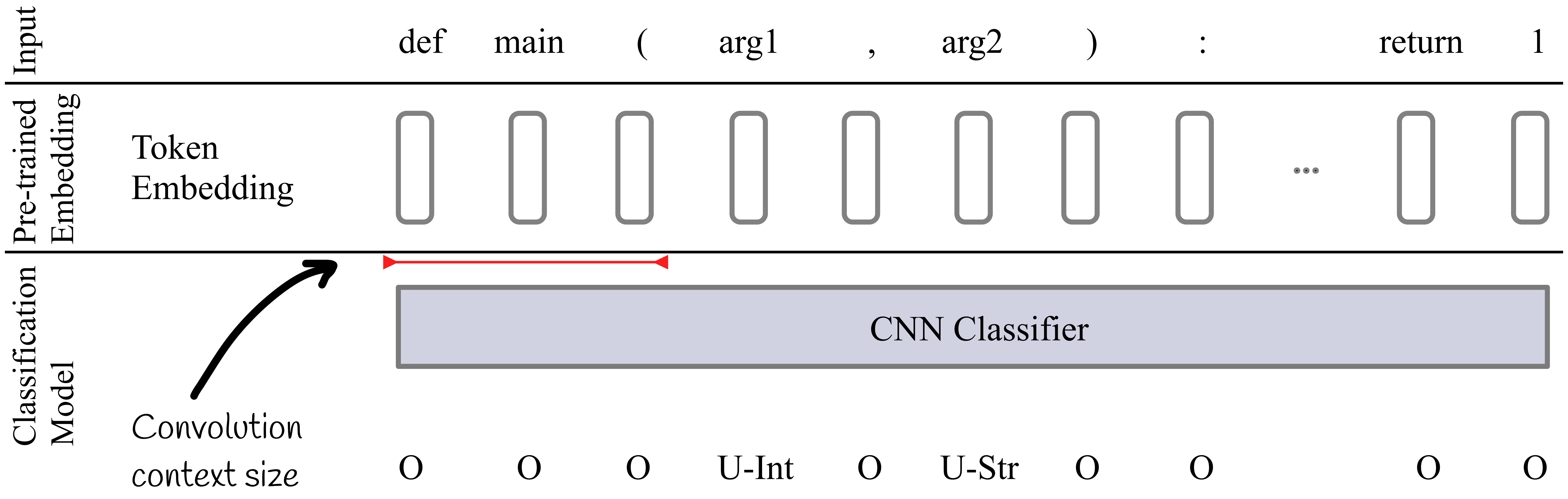
- Rows of E are token embeddings

Background

NLP after ~2008-2010

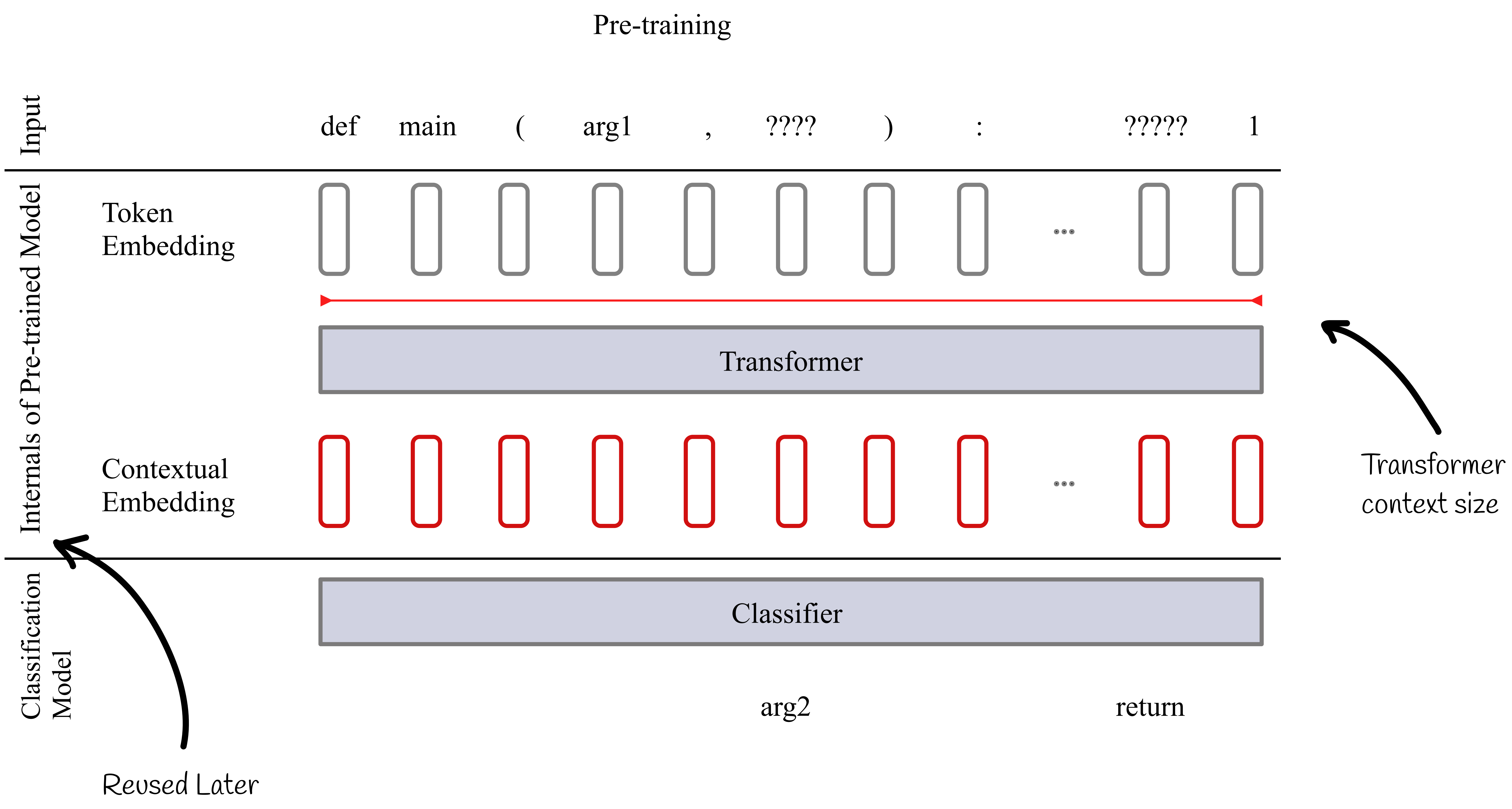
$A = E \cdot E^T, A_{ij} = 1$ if token t_i collocates with token t_j

- $A \in R^{|V| \times |V|}$
- $E \in R^{|V| \times d}$, d - embedding dimensionality
- Rows of E are token embeddings
- Features are no longer explainable, need to learn black box classifier
- Context is limited, but larger than before



Background
NLP state-of-the-art

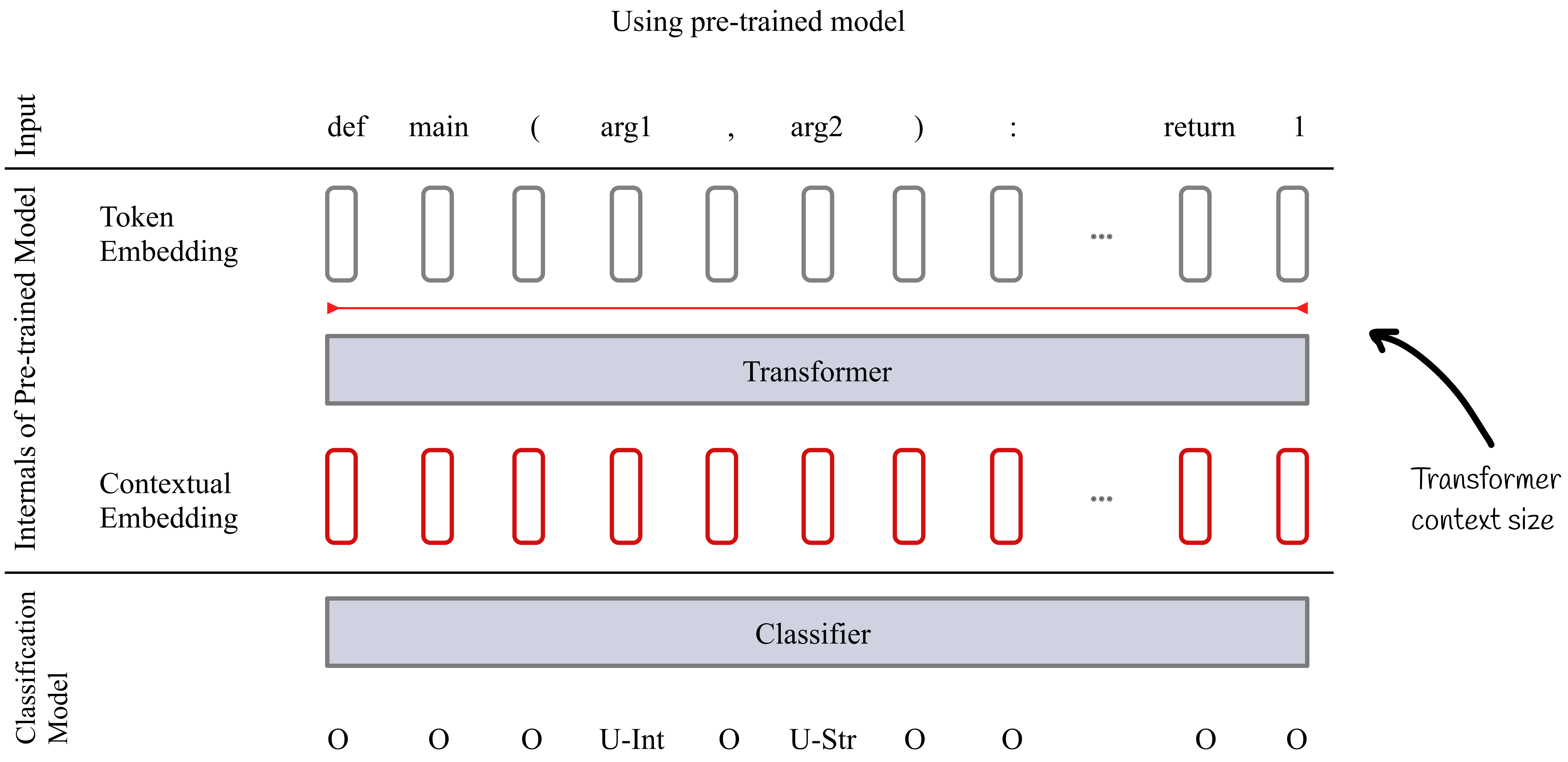
- Modern methods trained with Masked Language Model (MLM) objective
- Context size is large
- Pre-train entire model instead of only token embeddings



Background

NLP state-of-the-art

- Most of the model is reused for new task
- Classifier in the end substituted for task-specific classifier



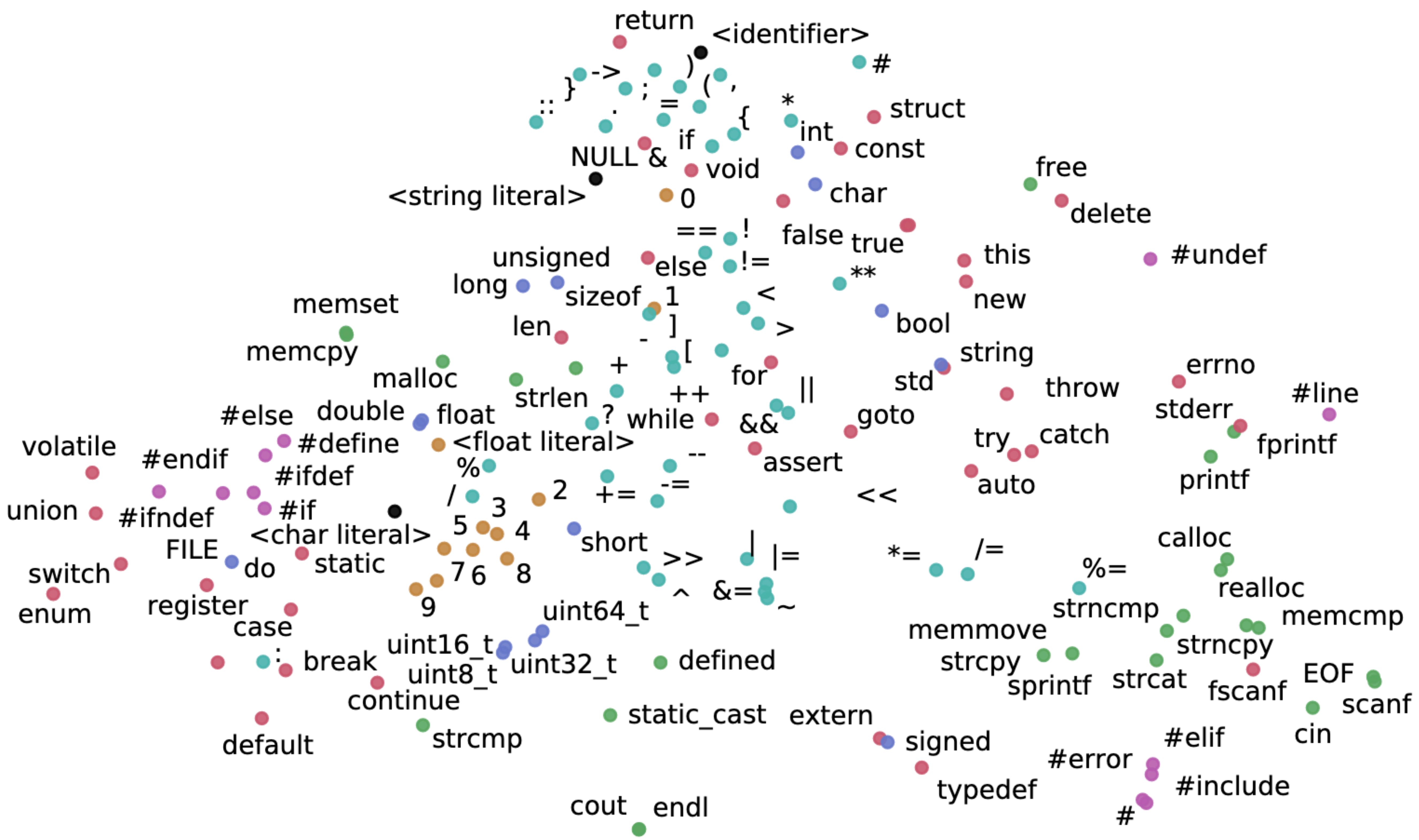
Background

NLP models summary

Classical NLP	Early applications of embeddings	State-of-the-art NLP
High dimensional discrete features	Embedding vectors as **low dimensional** features	Embedding vectors are **contextual** and require a pre-trained model
Features are **hand crafted**	**Embeddings** are trained automatically to recover **word collocations**	Embedding **model** pre-trained to recover **missing tokens**
Trained for **specific task**	Embeddings are pre-trained on large corpus and **reused** across tasks	Embedding model is pre-trained on large corpus and **reused** across tasks
Increasing **context size** leads to **exploding** feature space	**Context size is small** due to limited computational resources	**Context size is large** and requires a lot of computational resources

Background

Examples of Embeddings for Source Code



Example of projecting vectors for tokens in C/C++ source code to 2D space (Harer et al. 2018)

Background

Drawbacks of NLP models for Source Code

Problems:

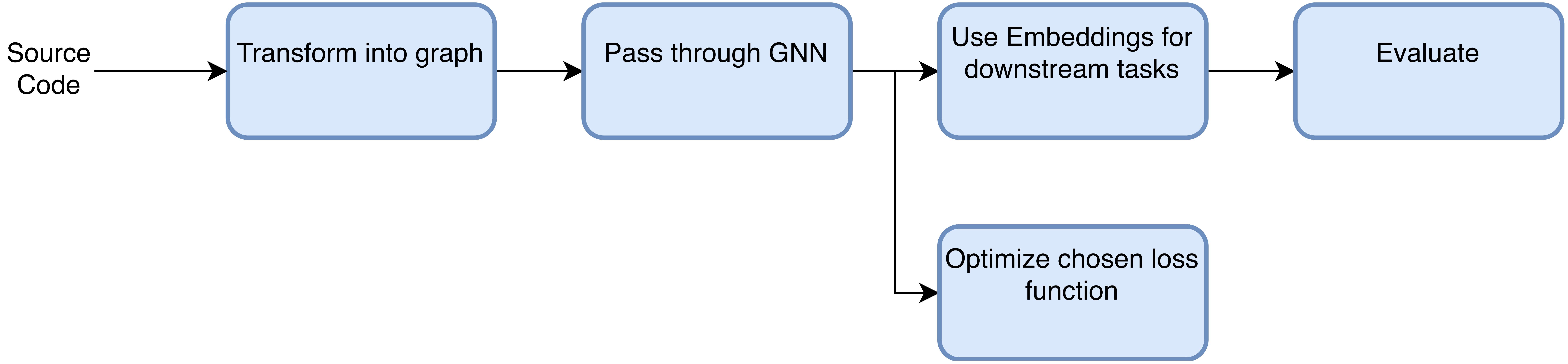
- Source code is represented as a sequence of tokens
- Sequential representation does not reflect source code execution order
- Context size is limited and cannot incorporate imports and calls
- Elements of source code are processed as strings and not as aliased entities

Solution

- Use Graph Neural Networks

Method Description

Source Code Processing Pipeline



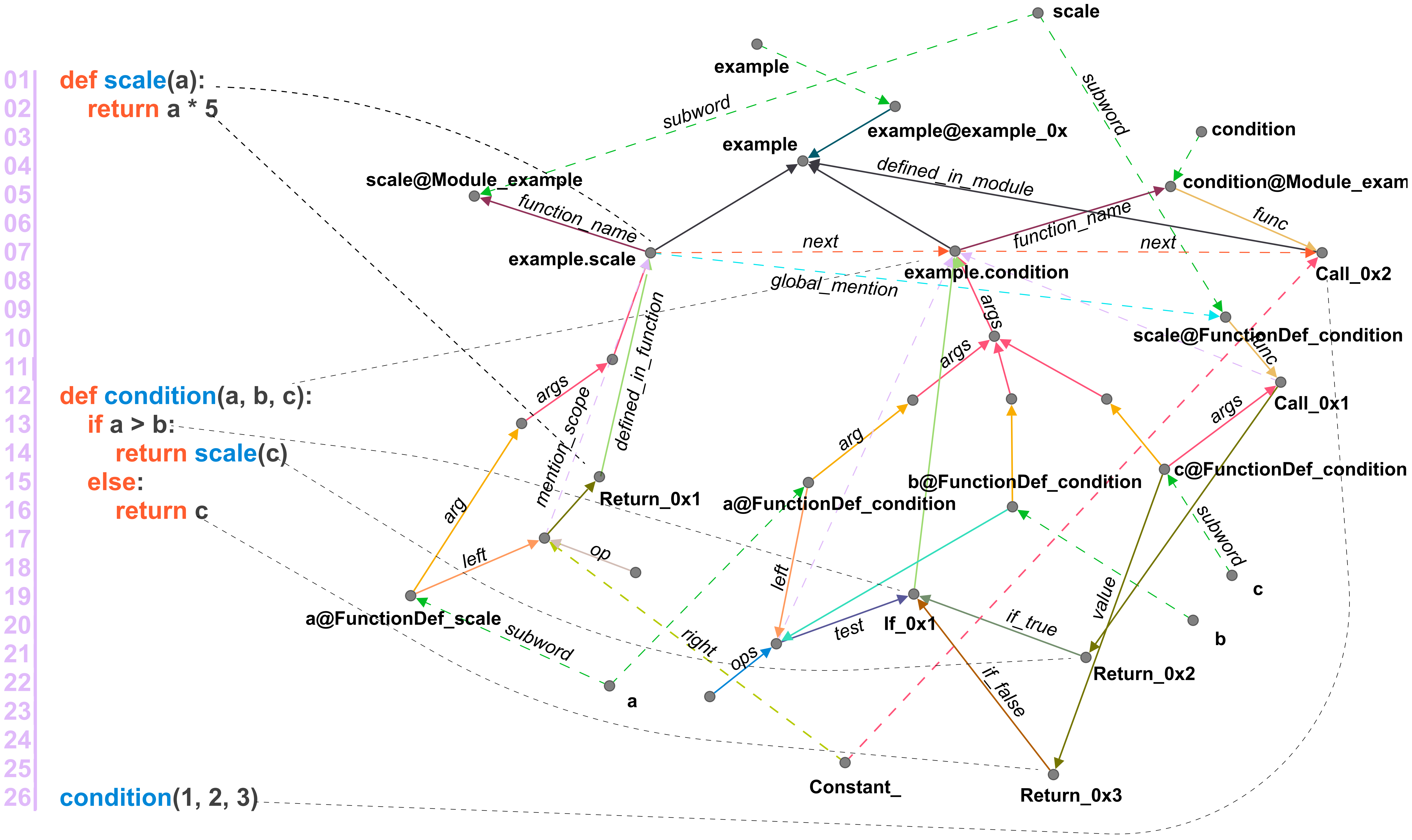
Method Description

Transform Source Code Into a Graph

- Abstract Syntax Tree (AST) was chosen as a starting point for building graph representation of source code
- Additional edges added to connect different parts of source code (inheritance, calls, imports)
- Nodes that represent the same variable in a function are connected together
- Reduce the number of parameters with subword tokenization

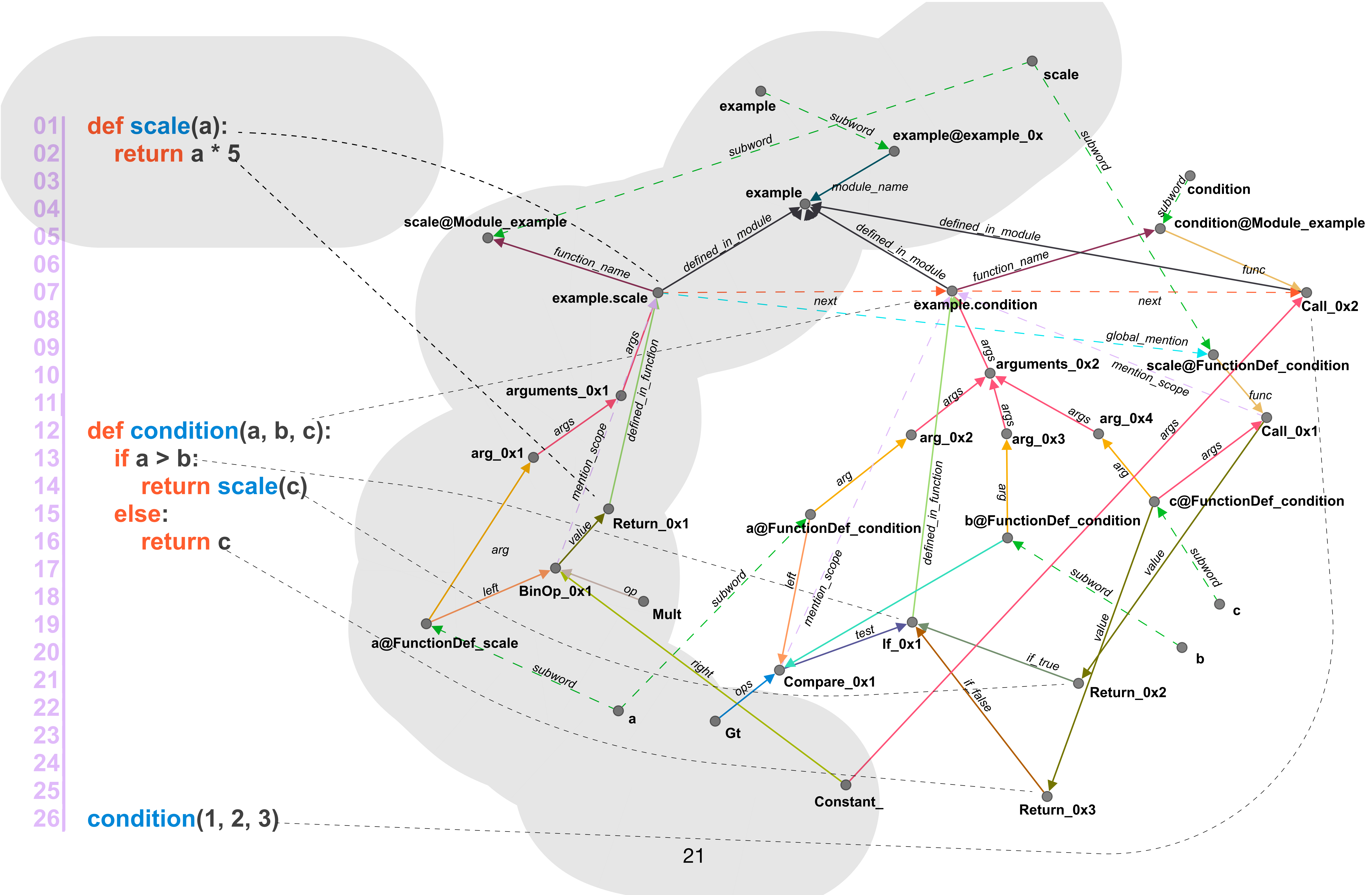
Method Description

Transform Source Code Into a Graph



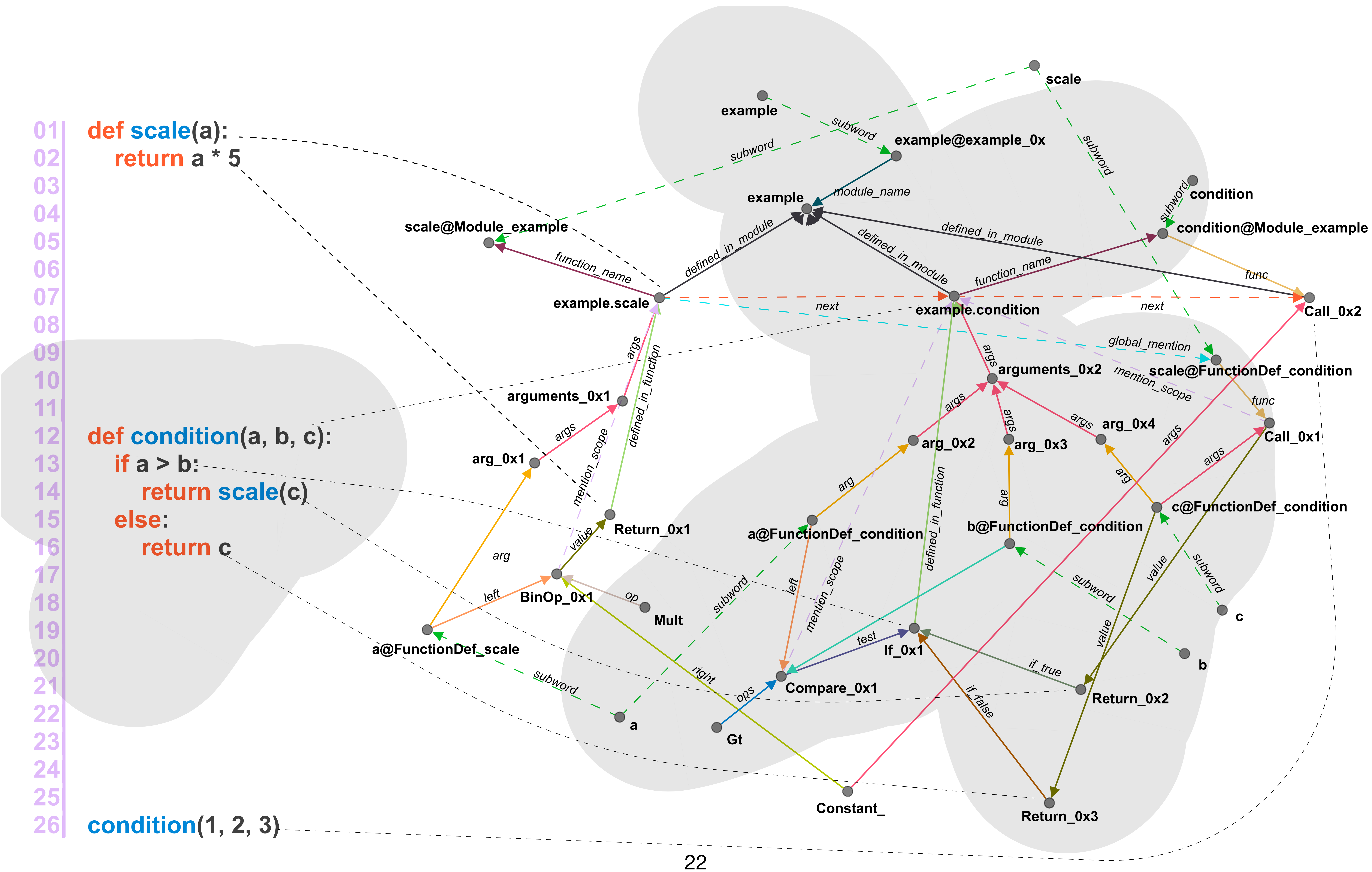
Method Description

Transform Source Code Into a Graph



Method Description

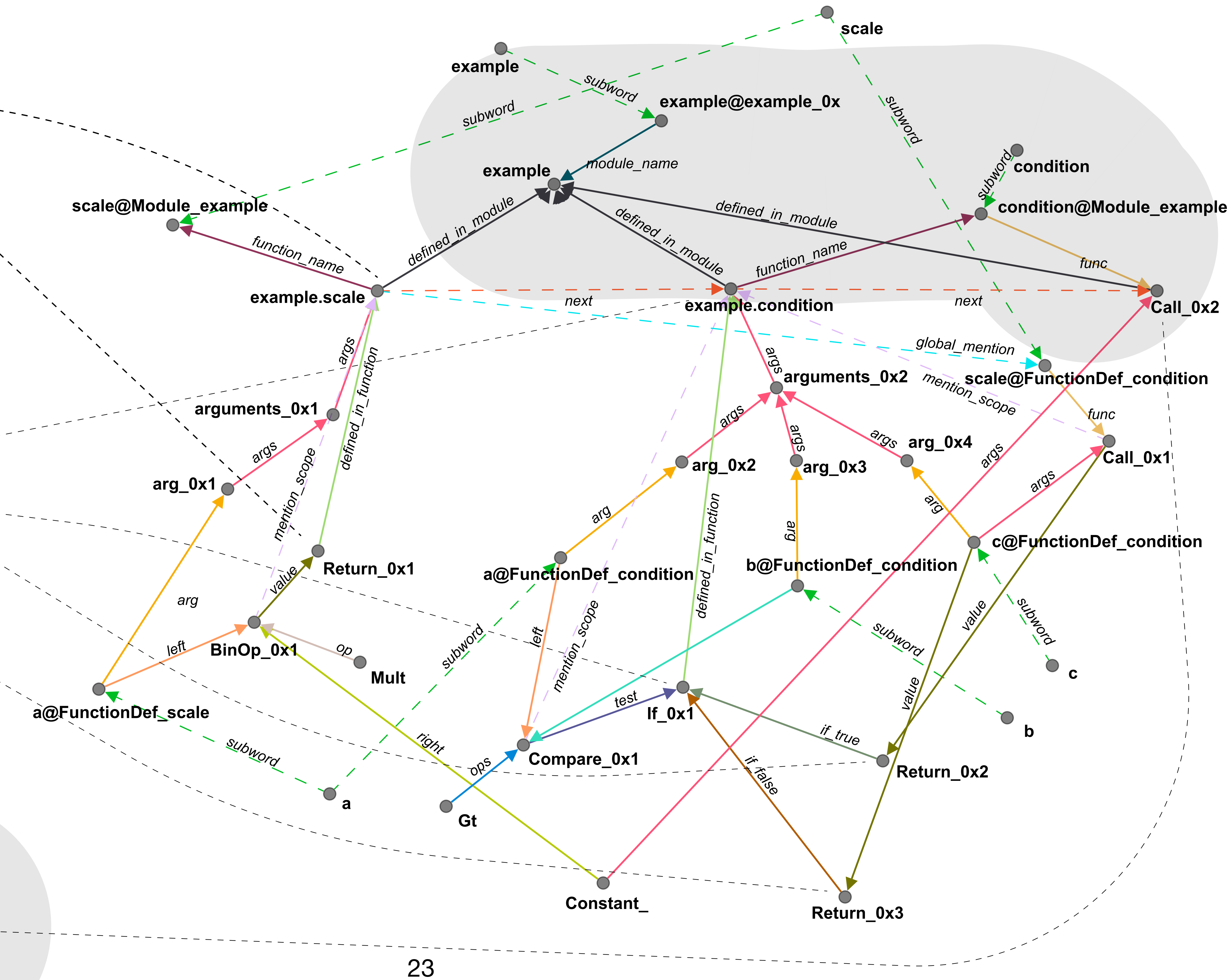
Transform Source Code Into a Graph



Method Description

Transform Source Code Into a Graph

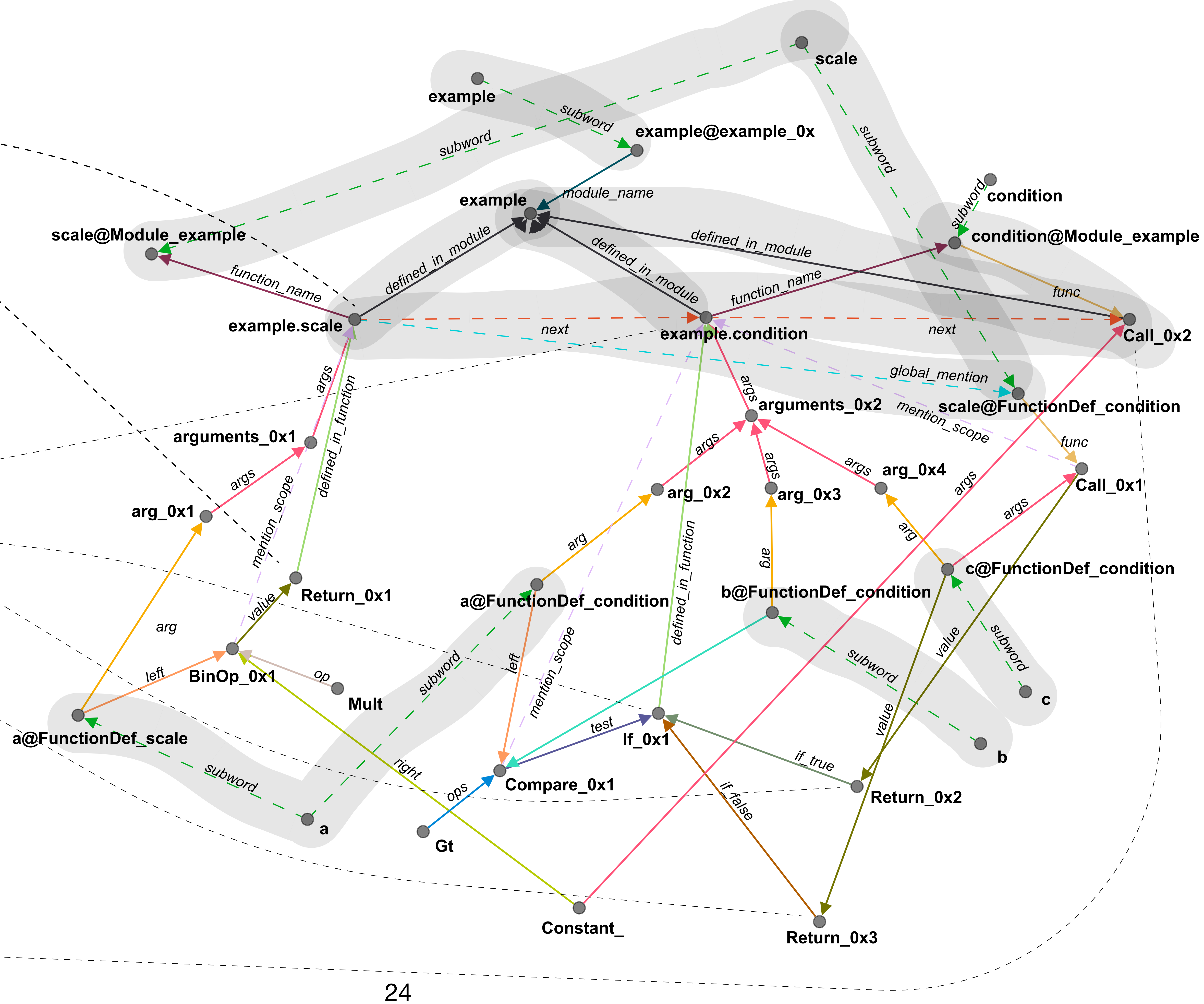
```python
def scale(a):
    return a * 5




























def condition(a, b, c):
    if a > b:
        return scale(c)
    else:
        return c










condition(1, 2, 3)
```



Method Description

Transform Source Code Into a Graph

```python
def scale(a):
    return a * 5









def condition(a, b, c):
    if a > b:
        return scale(c)
    else:
        return c









condition(1, 2, 3)
```



Method Description

GNN

- Graph Neural Network (GNN) is a class of Neural Network that takes graph as an input
- GNNs are based on the principle of message passing
- Messages are vector representations of nodes

Method Description

GNN Message Passing Algorithm

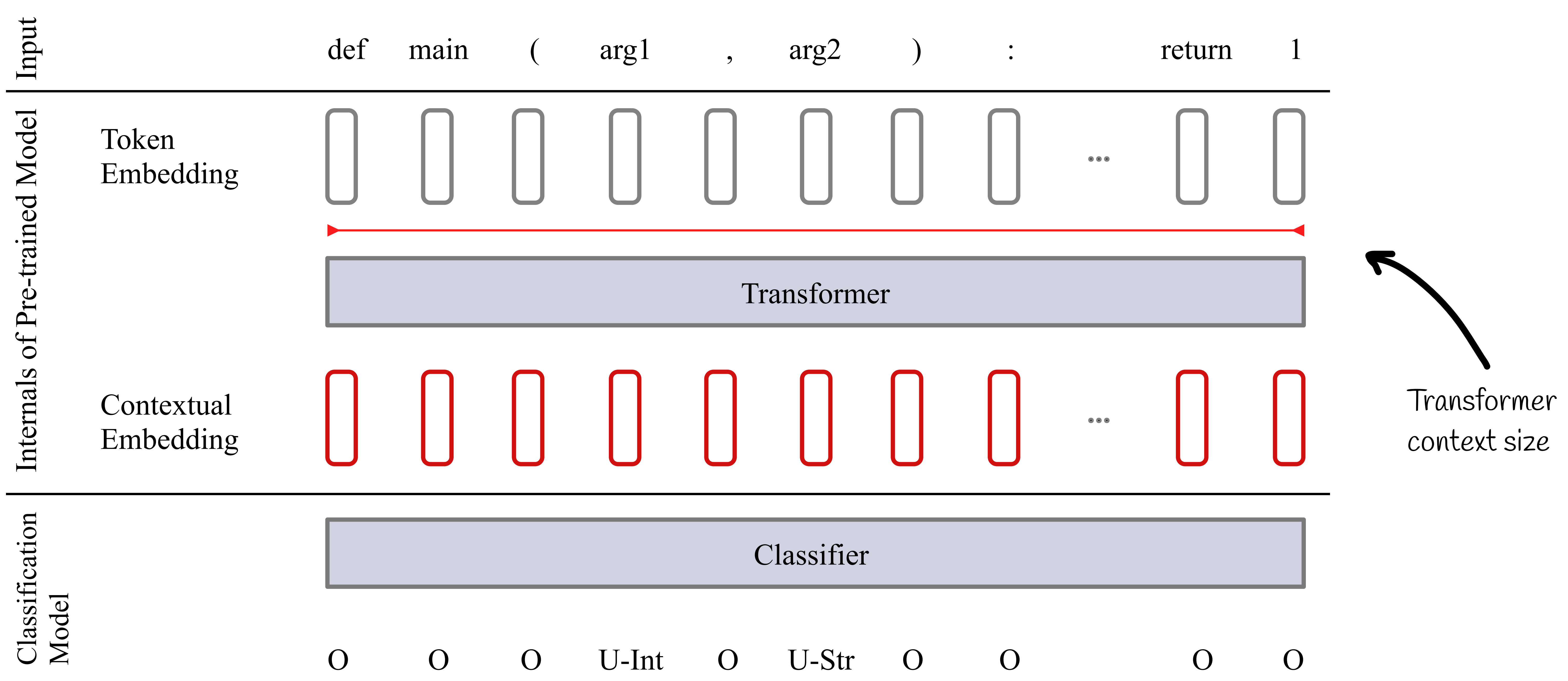
Algorithm 1: Message passing algorithm for GNN

```
Data: Input graph
Result: Model that produces embeddings for nodes in the graph
initialize node embeddings randomly;
while training do
    for n message passing steps do
        for each node, prepare message from it's own embedding;
        for each node, pass message to neighbours;
        for each node, aggregate messages from neighbours;
        for each node, update it's own embedding;
    end
    for nodes in training set, compute loss function;
    to minimize loss, update embeddings and GNN parameters;
end
```

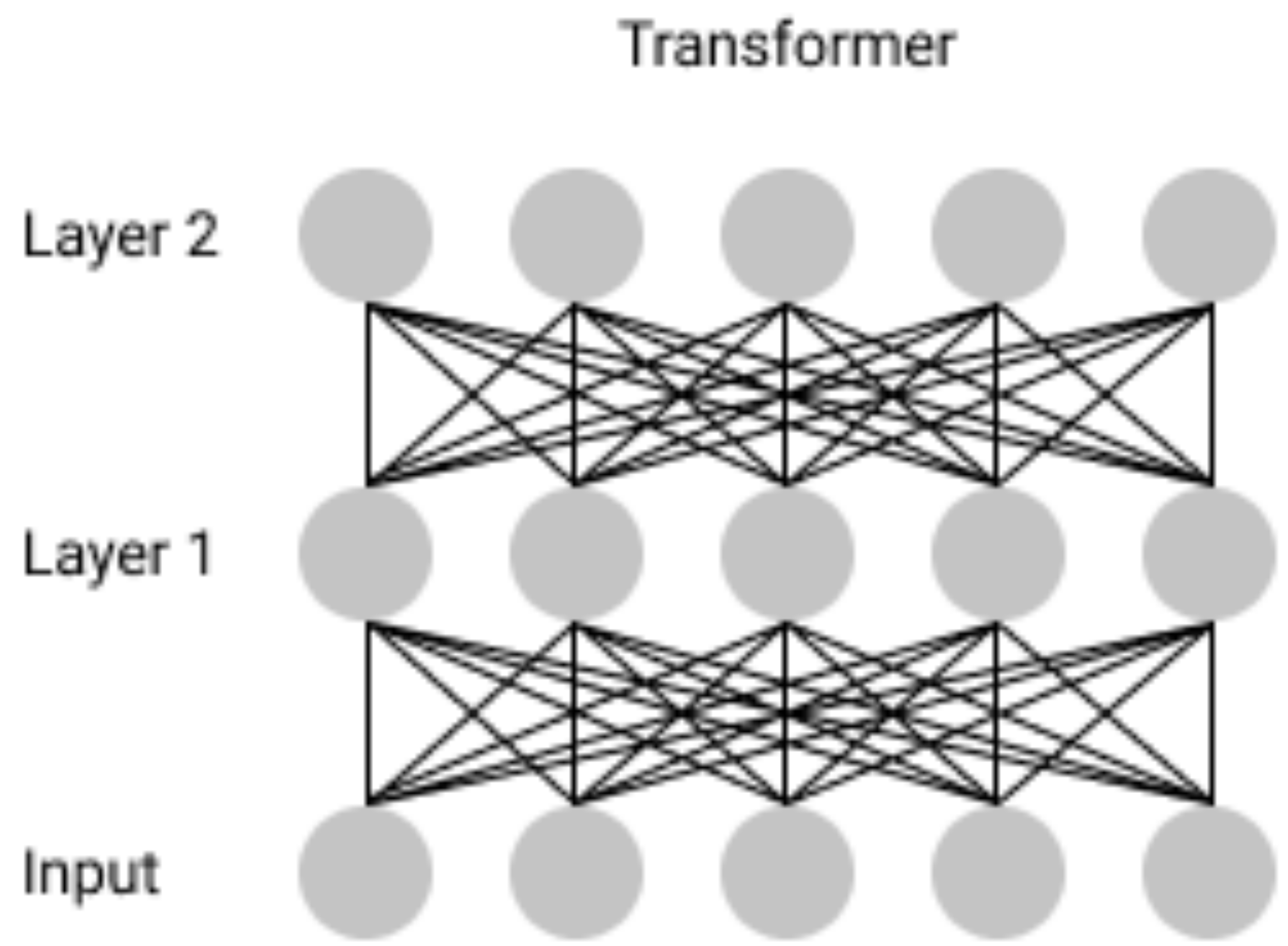
Method Description

Difference between GNN and Transformer

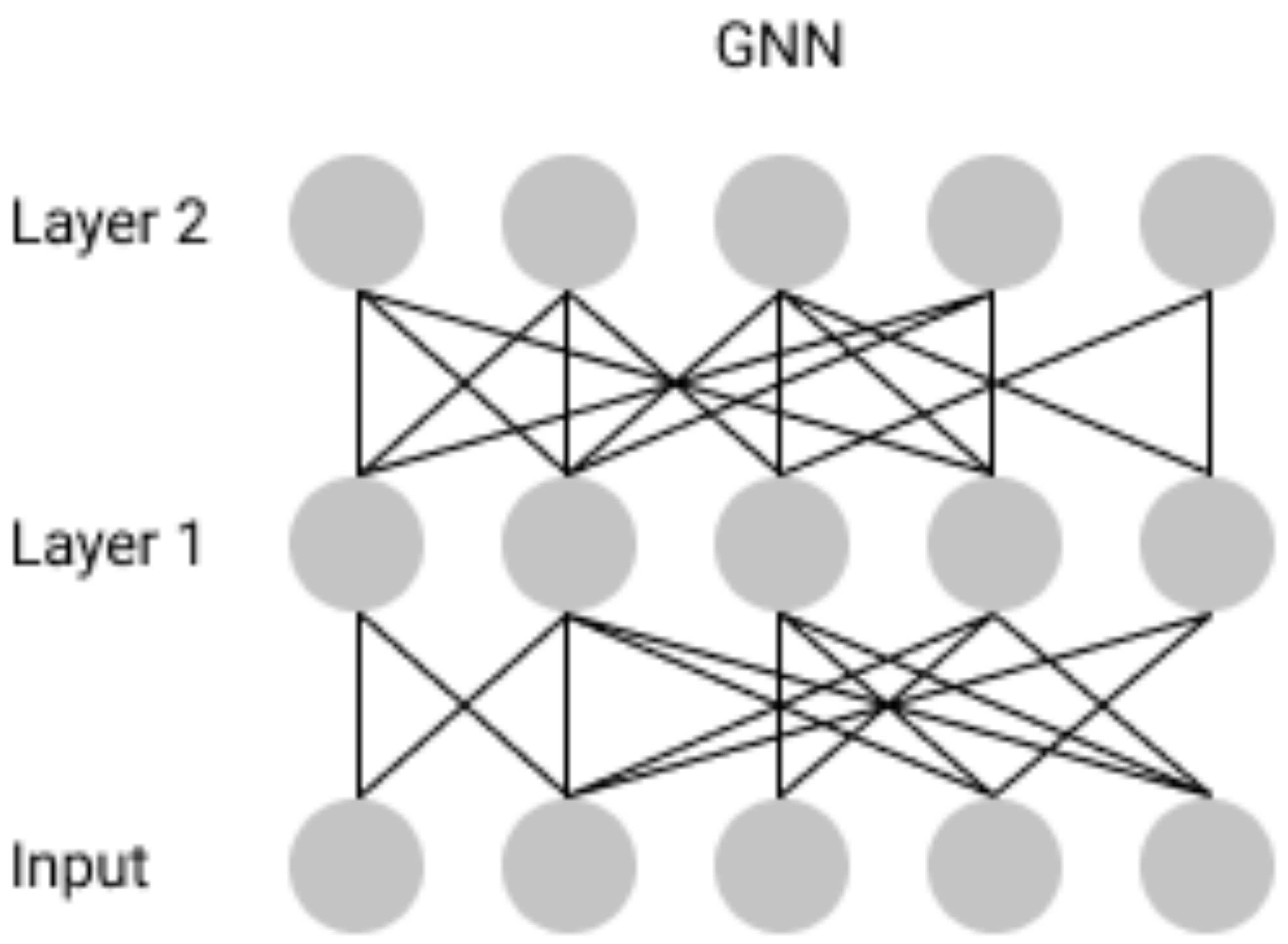
Using pre-trained model



Both models aim at producing embeddings for source code. These models share similarity
In how embeddings are computed



Transformer creates fully connected graph between each token in current context. Embeddings created for tokens.



GNN model uses only connections defined in the original graph. Embeddings created for nodes.

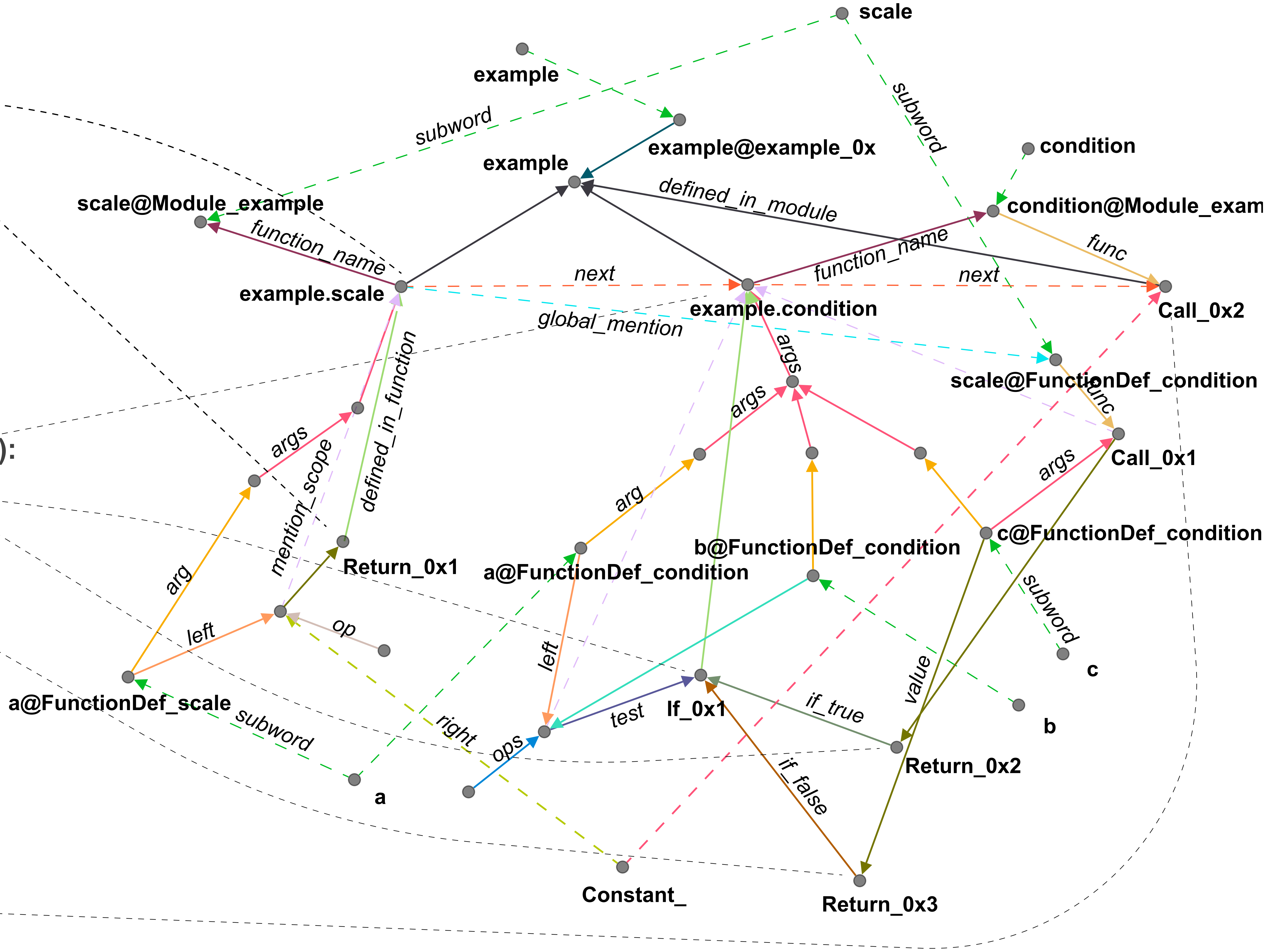
Method Description

Pre-training objectives

We experiment with several pre-training objectives

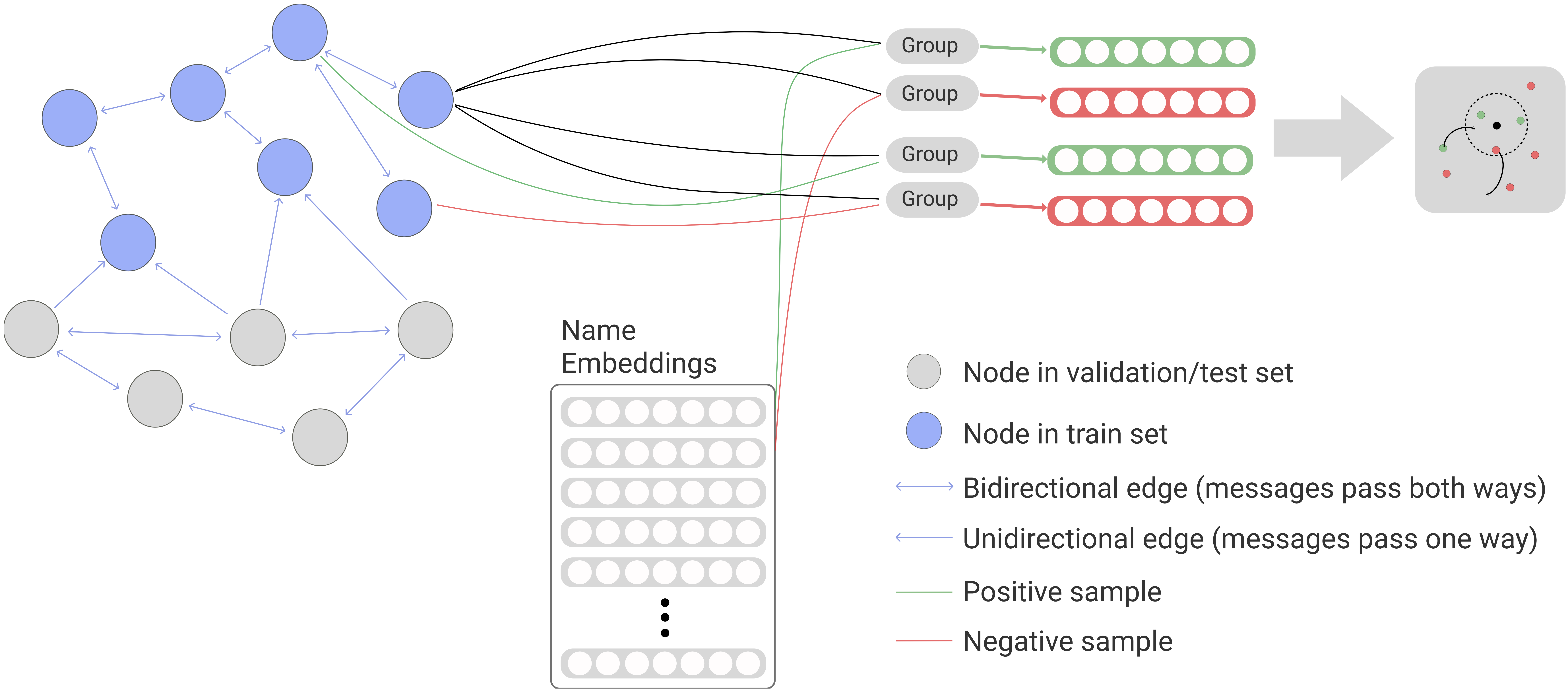
- Name prediction
- Edge prediction
- Node type prediction
- TransR objective (variant of edge prediction)

```python
def scale(a):
    return a * 5









def condition(a, b, c):
    if a > b:
        return scale(c)
    else:
        return c










condition(1, 2, 3)
```



Method Description

Pre-training objectives



Method Description

Pre-training outcome

- The outcome of pre-training process is a GNN model that can be used for computing GNN embeddings
- GNN embeddings are not the same as token embeddings in NLP approaches
- GNN embeddings can be used for classifying individual nodes and for classifying subgraphs
- At the current stage, experiments completed only for classifying variables by type (node classification)

Type prediction

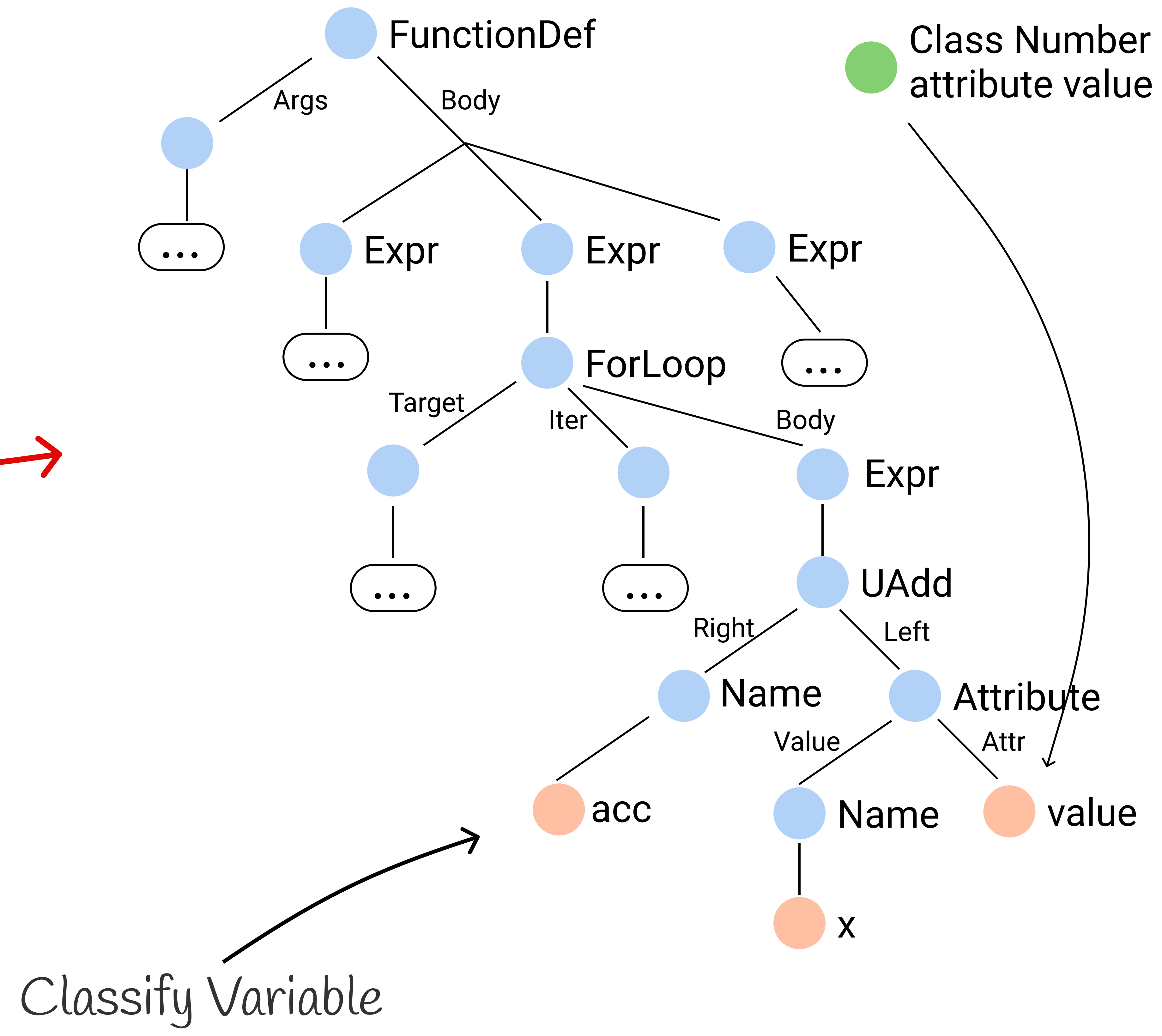
Approaches for Type Prediction

- Classify embedding for node that represents variable (Type Prediction I)
- Use hybrid model to take advantage from sequential and graph representations (Type Prediction II)

Type prediction

Approach for Type Prediction I

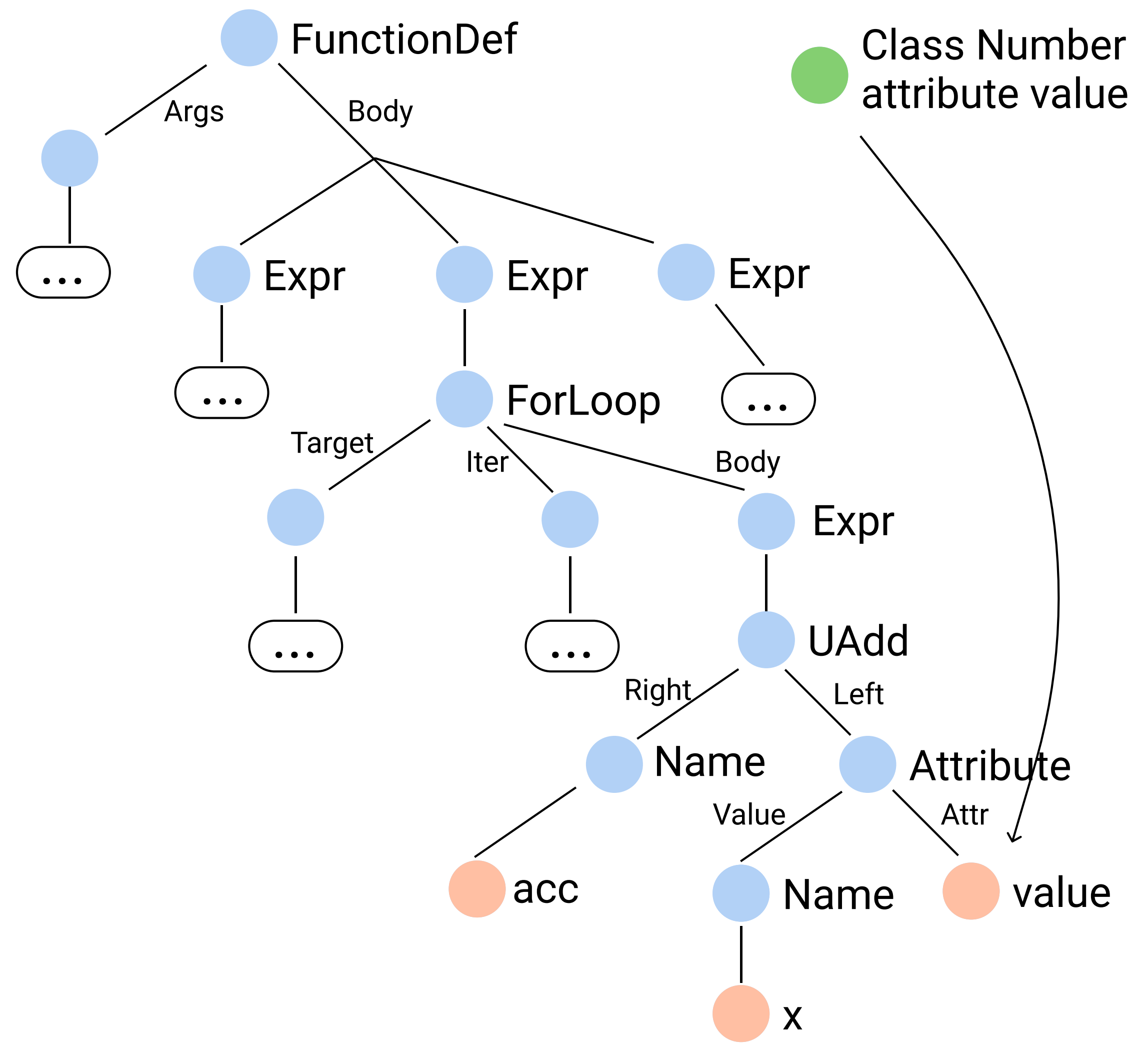
```
def sum(xs: List[Numbers]) -> int:
    """
    Takes a list of integers,
    returns an integer
    """
    acc: int = 0
    for x in xs:
        acc += x.value
    return acc
```



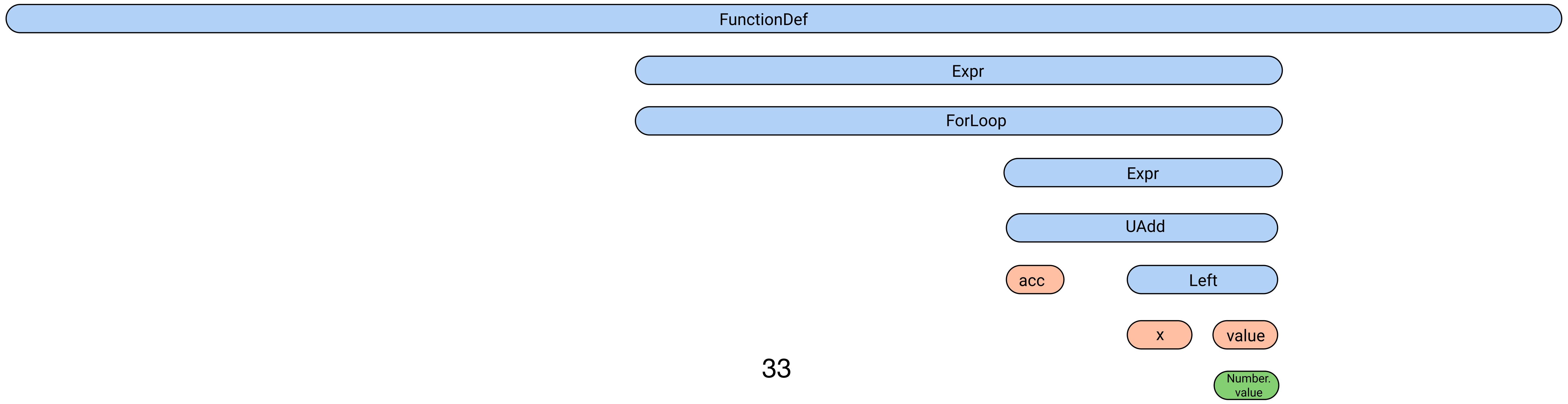
Type prediction

Approach for Type Prediction II

```python
def sum(xs: List[Numbers]) -> int:
    """
    Takes a list of integers,
    returns an integer
    """
    acc: int = 0
    for x in xs:
        acc += x.value
    return acc
```



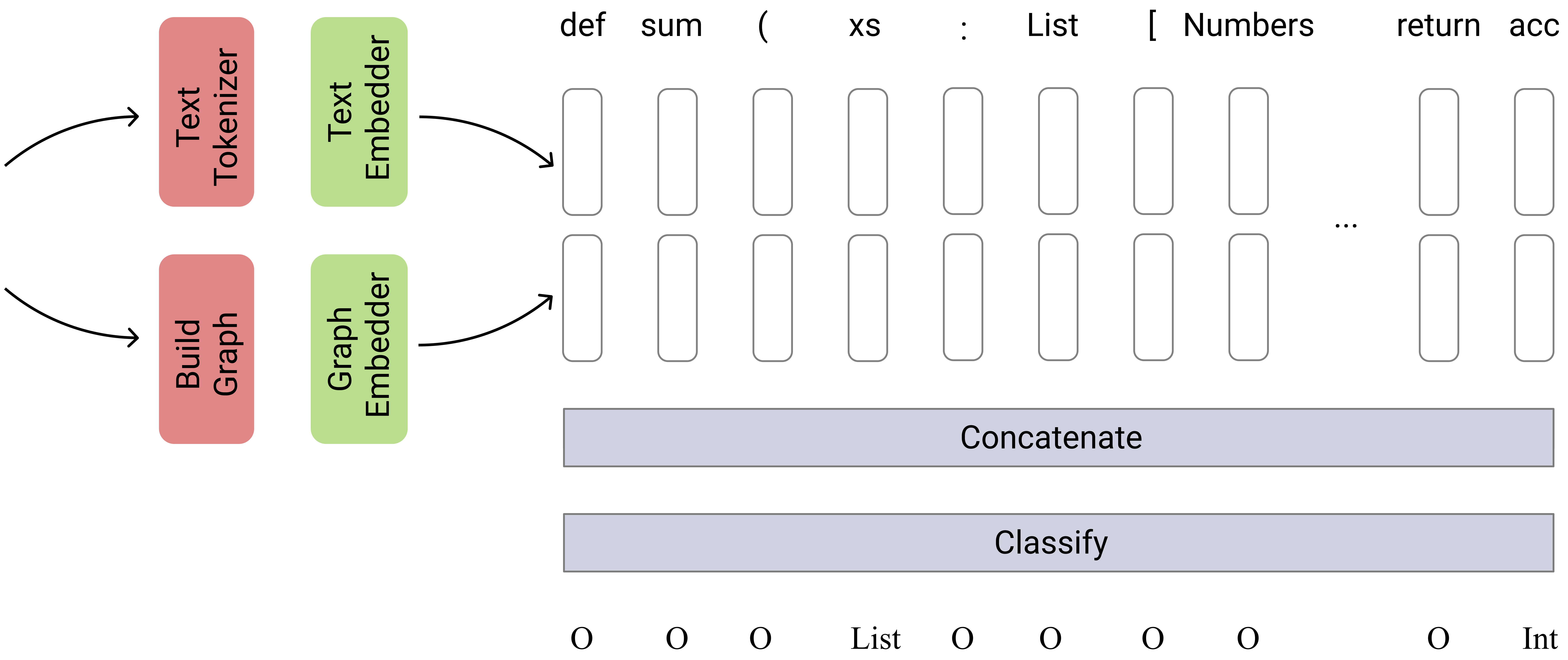

```
def sum ( xs ) : \n \t acc = 0 \n \t for x in xs: \n \t \t acc += x . value \n \t return acc
```



Type prediction

Approach for Type Prediction II

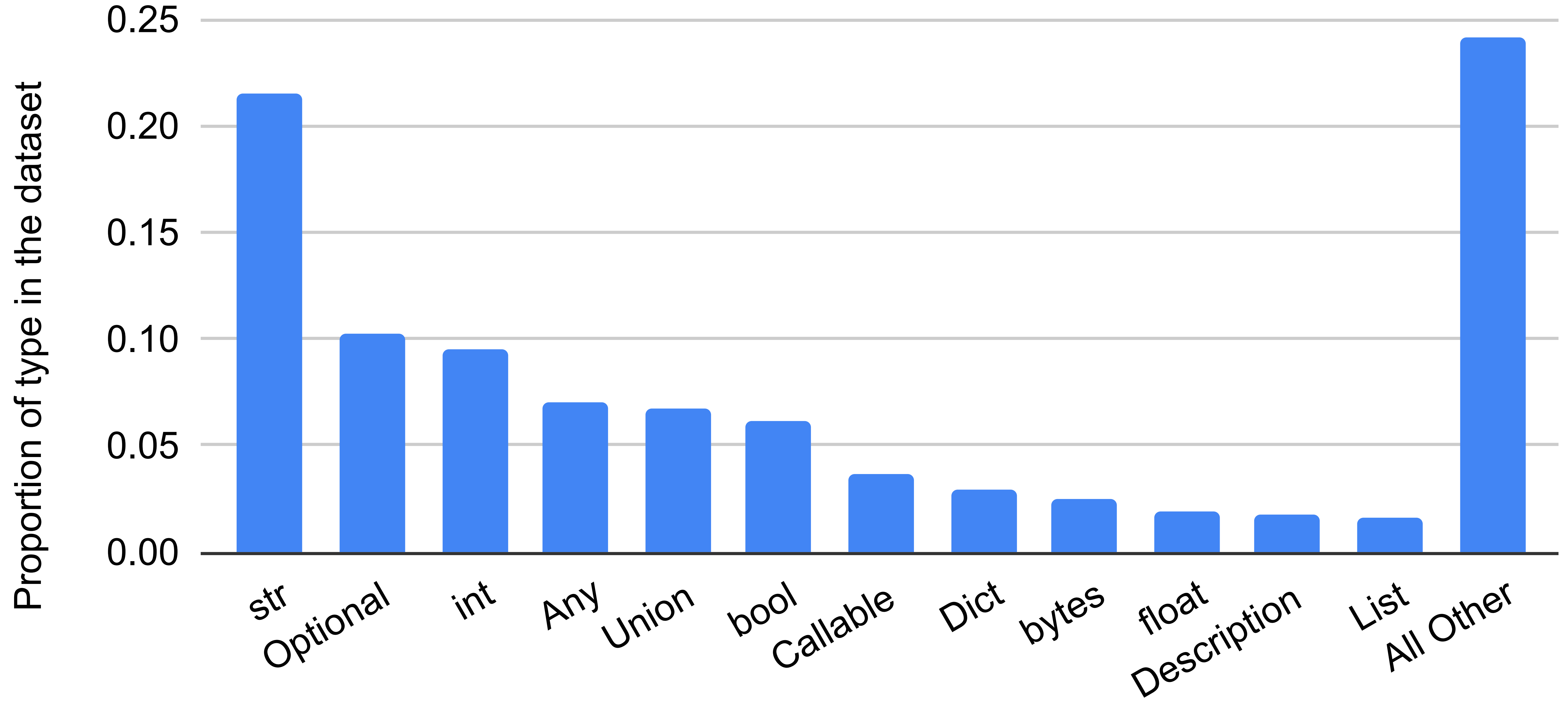
```
def sum(xs) -> int:
    """
    Takes a list of integers,
    returns an integer
    """
    acc: int = 0
    for x in xs:
        acc += x.value
    return acc
```



Experiments

Data Preparation

- Remove type annotations and default values from code and graph
- Pre-train GNN model
- Compute embeddings for nodes that represent variables
- Simplify type annotations (List[Int] -> List)
- Overall 2767 examples and 89 unique type labels
- Use top 20 type labels (80%) as popular types



Experiments

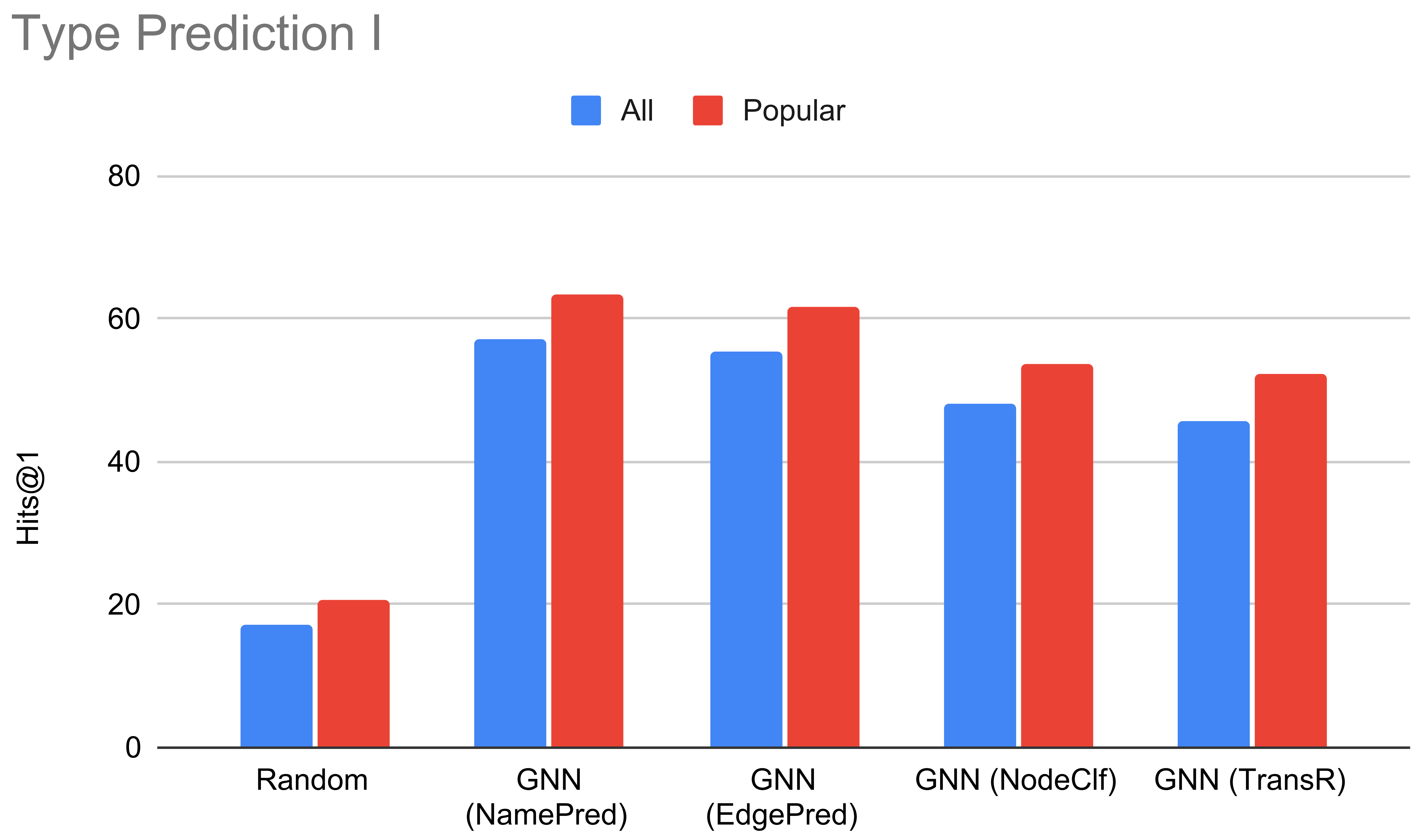
Need to determine:

1. Are pre-trained graph embeddings useful for type prediction?
2. How type prediction quality is affected by embedding dimensionality?
3. Can GNN embeddings be used together with sequential embeddings?
4. Do GNN embeddings improve training speed?

Experiment

Are pre-trained graph embeddings useful for type prediction?

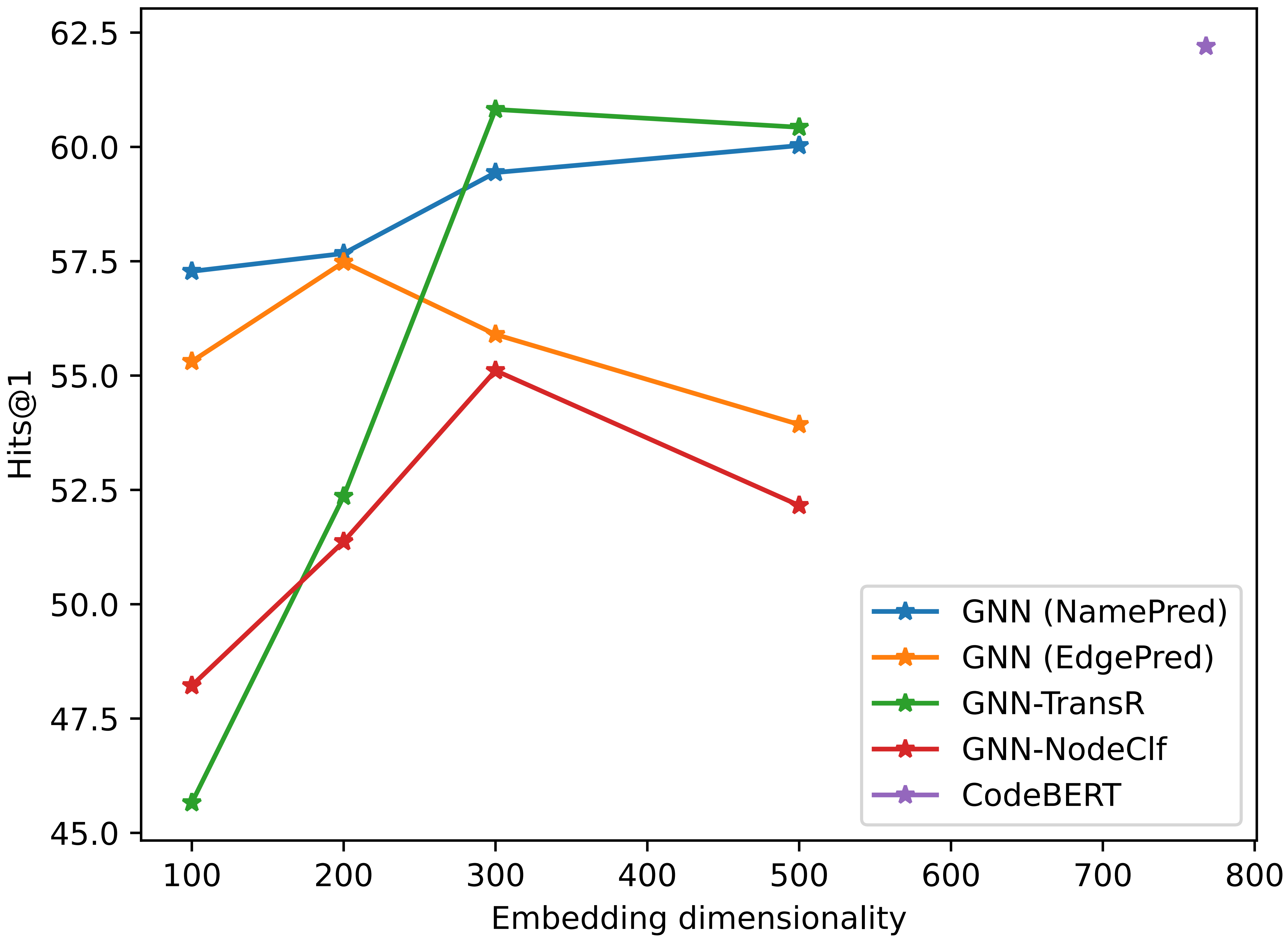
- Pre-train GNN embeddings using pre-training objective
- Pre-training objective is not related to type prediction
- Take embeddings for nodes that correspond to variables
- Classify embeddings into Python types
- Random embeddings are used as baseline



Experiments

How type prediction quality is affected by embedding dimensionality?

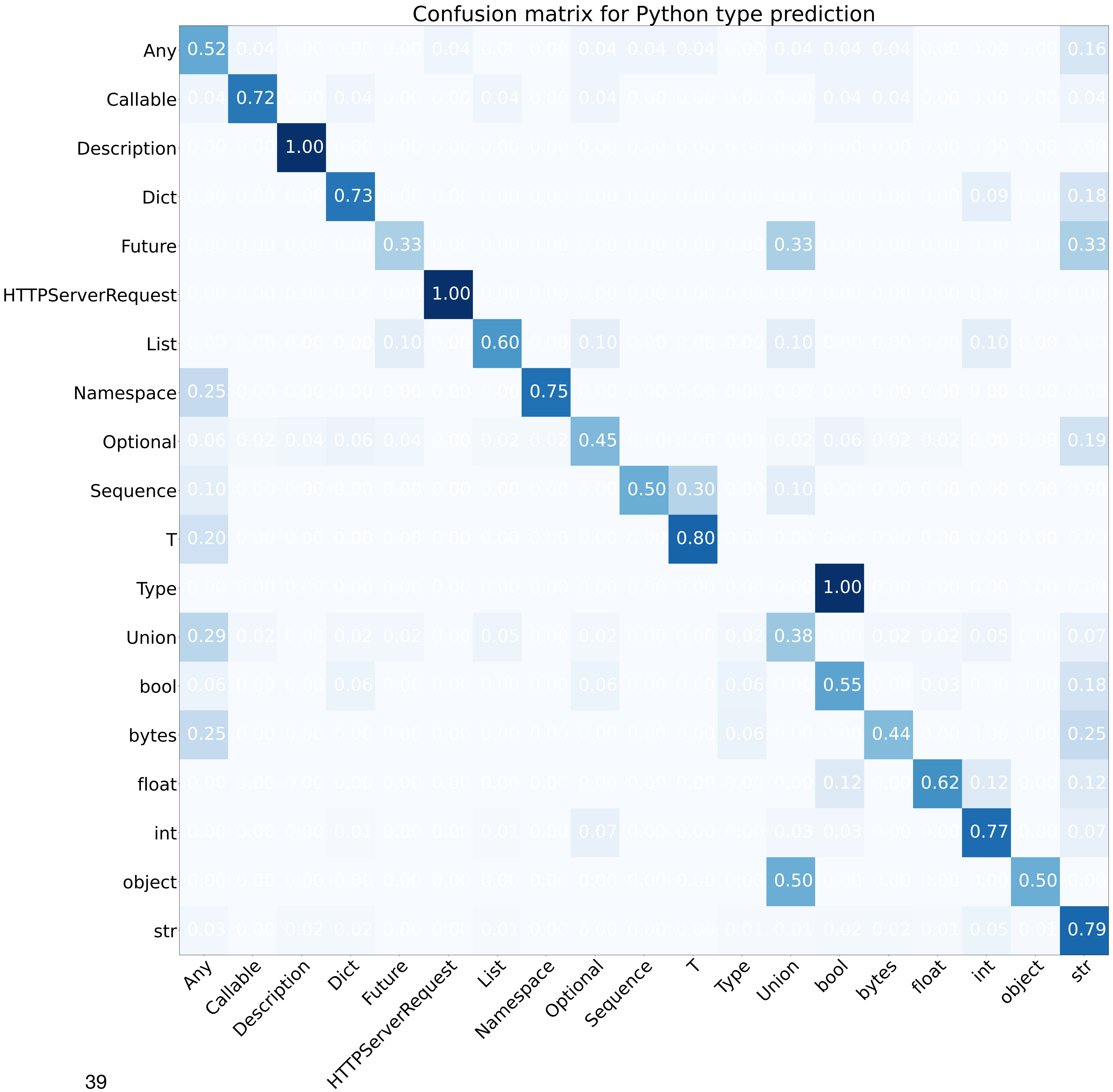
- Dimensionality of GNN embeddings can be controlled
- Increasing dimensionality leads to better performance
- CodeBERT (dimensionality 768) provided for reference



Experiments

What are common mistakes?

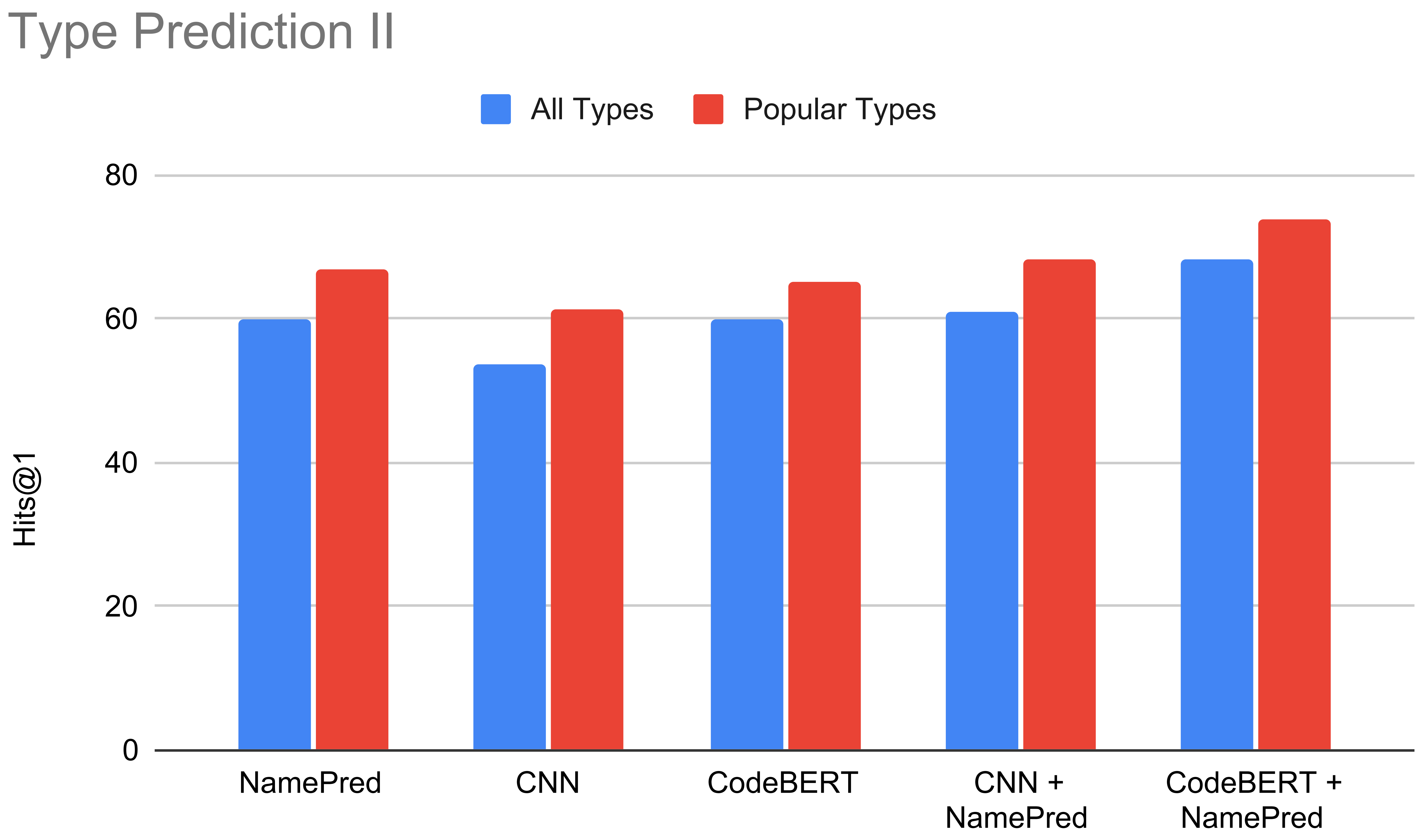
- Vertical axis - correct type
- Horizontal - predicted type
- A lot of confusion with popular types (str, any)
- Confusion among ambiguous types (Union, List)



Experiments

Can GNN embeddings be used together with sequential embeddings?

- Need to determine
- Are pre-trained graph embeddings useful for type prediction?
- How type prediction quality is affected by embedding dimensionality?
- Can GNN embeddings be used together with sequential embeddings?
- Do GNN embeddings improve training speed?



Experiments

Type Prediction Examples

Predicted

```
def _update_inplace( self  FRAMEORSERIES ,  result  STR ,  verify_is_copy   BOOL ) :

        # NOTE: This does *not* call __finalize__ and that's an explicit
        # decision that we may revisit in the future.

        self._reset_cache()
        self._clear_item_cache()
        self._data = getattr(result, "_data", result)
        self._maybe_update_cacher(verify_is_copy=verify_is_copy)
```

Ground Truth

```
def _update_inplace(self, result,  verify_is_copy   BOOL_T ) :

        # NOTE: This does *not* call __finalize__ and that's an explicit
        # decision that we may revisit in the future.

        self._reset_cache()
        self._clear_item_cache()
        self._data = getattr(result, "_data", result)
        self._maybe_update_cacher(verify_is_copy=verify_is_copy)
```

Experiments

Type Prediction Examples

Predicted

```
def __init__(self, string STR ) :
    if not isinstance(string, str):
        raise TypeError("IsEqualIgnoringCase requires string")
    self.original_string = string
    self.lowered_string = string.lower()
```

Ground Truth

```
def __init__(self, string STR ) :
    if not isinstance(string, str):
        raise TypeError("IsEqualIgnoringCase requires string")
    self.original_string = string
    self.lowered_string = string.lower()
```

Experiments

Type Prediction Examples

Predicted

def parse_config_file(path STR , final BOOL) :

return options.parse_config_file(path, final=final)

Ground Truth

def parse_config_file(path STR , final BOOL) :

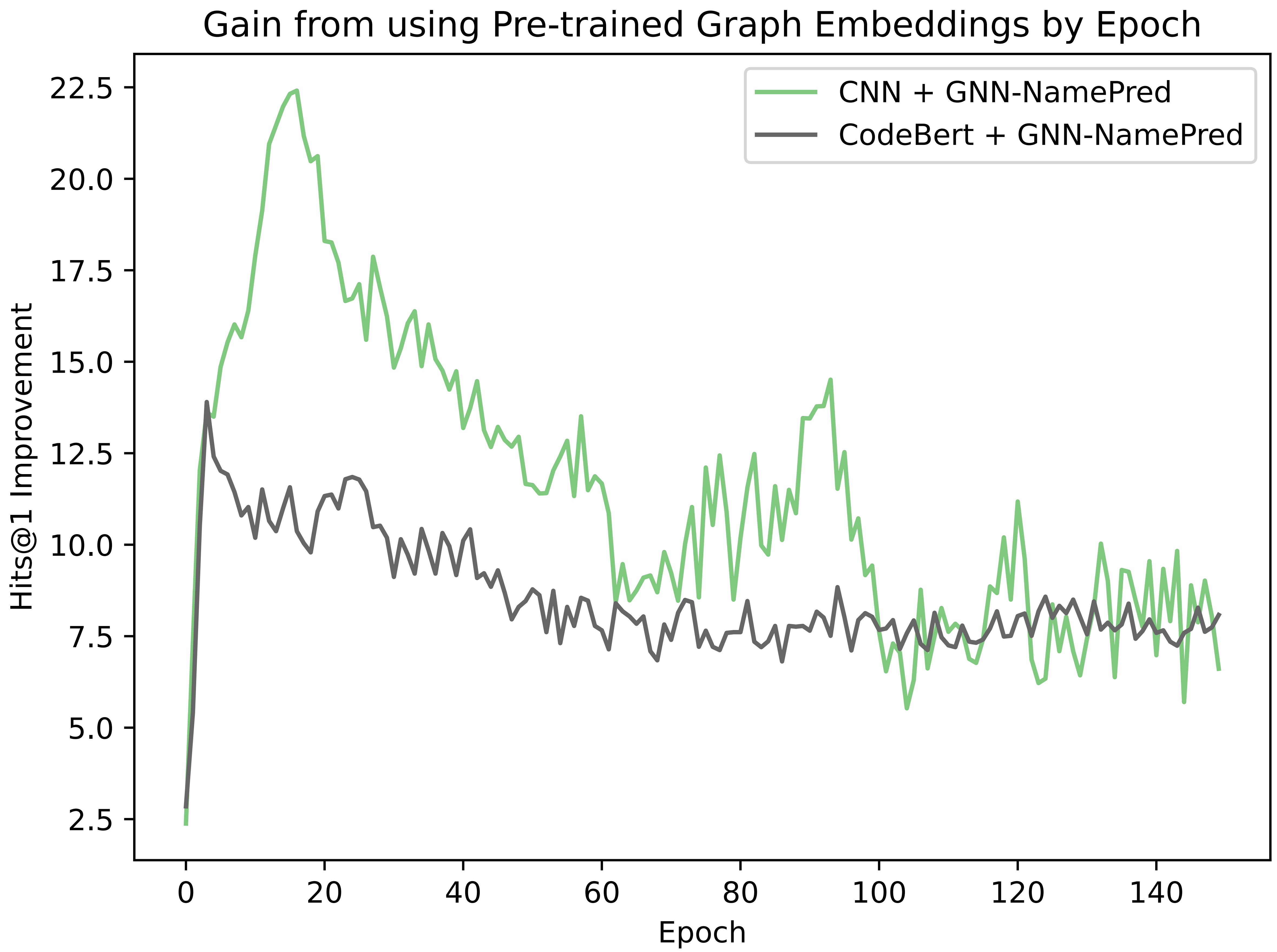
return options.parse_config_file(path, final=final)

Experiments

Do GNN embeddings improve training speed?

- ML models are trained in iterations (epochs)
- Pre-trained model improves training speed for Type Prediction II
- Top prediction accuracy improved

$$\text{Improvement}_i = \text{Score\_with\_GNN}_i - \text{Score\_without\_GNN}_i$$



Summary

- Created a GNN model for creating source code embeddings
- Designed an approach to use sequential representation of source code (NLP) together with graph representations
- Evaluated pertained embeddings on the task of type prediction for Python variables. Compared results with CodeBERT
- Using CodeBERT together with GNN embeddings allow to improve type prediction accuracy
- GNN embeddings improve training speed
- Pre-training gives best results using Name Prediction objective

Future Work

- Better to test on larger dataset
- Current results for the prediction achieved for simplified types. Can explore approaches for predicting full type
- Need to evaluate on other tasks (variable misuse, search)

Thank you