	Python	Javascript	Go	Ruby
Python	1.00	0.43	0.42	0.79
Javascript	0.43	1.00	0.84	0.39
Go	0.43	0.84	1.00	0.38
Ruby	0.79	0.39	0.38	1.00

Table 5: Pairwise cosine similarities of the learned language embeddings of the CODE TRANSFORMER.

ACKNOWLEDGEMENTS

We are grateful to Dylan Bourgeois for having paved the way to this research contribution with his thesis work (Bourgeois, 2019). We further thank Simon Geisler for his helpful suggestions and proofreading the paper, as well as the anonymous reviewers for their constructive feedback and fruitful discussions.

This research was supported by the TUM International Graduate School of Science and Engineering (IGSSE). Stanford University is supported by DARPA under Nos. N660011924033 (MCS); ARO under Nos. W911NF-16-1- 0342 (MURI), W911NF-16-1-0171 (DURIP); NSF under Nos. OAC-1835598 (CINES), OAC-1934578 (HDR), CCF-1918940 (Expeditions), IIS-2030477 (RAPID); Stanford Data Science Initiative, Wu Tsai Neurosciences Institute, Chan Zuckerberg Biohub, Amazon, JPMorgan Chase, Docomo, Hitachi, JD.com, KDDI, NVIDIA, Dell, Toshiba, Intel, and UnitedHealth Group. Jure Leskovec is a Chan Zuckerberg Biohub investigator.

A APPENDIX

A.1 DISTANCE ENCODING FUNCTION

For encoding scalar relation values via vectors we employ encoding functions $\phi : \mathbb{R} \to \mathbb{R}^d$, where d is the model's embedding dimension. We choose the popular sinusoidal encoding function presented in Vaswani et al. (2017):

$$\phi(r_{i\to j})_{2k} = \sin\left(\frac{r_{i\to j}}{M^{2k/d}}\right) \qquad \phi(r_{i\to j})_{2k+1} = \cos\left(\frac{r_{i\to j}}{M^{2k/d}}\right),$$

where $1 \leq k < d/2$ is the position in the encoding vector and M is some constant; we adopt $M = 10,000$ as chosen by (Vaswani et al., 2017). Note that the distance encoding functions have no trainable parameters.

A.2 MULTILINGUAL REPRESENTATION ANALYSIS

In Table 5, we show the pairwise cosine similarities of the learned language embeddings of the CODE TRANSFORMER. We can see that the pairs Python-Ruby and Javascript-Go have similar language embeddings. This aligns well with roots of language design and common use cases of the languages.

Moreover, in Table 6, we show selected snippets starting with `is`, `main`, or `load` (left) and their best embedding matches from other languages (right).

A.3 DATA PREPROCESSING

A.3.1 TEXTUAL CODE SNIPPET PREPROCESSING

1. **Tokenize** code snippets with Pygments language-specific tokenizer.
2. **Remove comments** (both multi-line, single-line and doc comments). The comment token types. `pygments.token.Comment` and `pygments.token.Literal.String.Doc` that are generated by Pygments are used to identify comments.

```javascript
function _isEqualArray(a, b) {
        if (a === b) {
            return true;
        }
        if ((a === undefined) ||
            (b === undefined)) {
            return false;
        }
        var i = a.length;
        if (i !== b.length){
            return false;
        }
        while (i--) {
            if (a[i] !== b[i]) {
                return false;
            }
        }
        return true;
    }
```

```go
func areSameFloat32Array(a, b []float32) bool {
  if len(a) != len(b) {
    return false
  }
  for i := 0; i < len(a); i++ {
    if a[i] != b[i] {
      return false
    }
  }
  return true
}
```

```javascript
function main() {
    var rawData = $('.HeaderTexture[data-login-
                     user-email]').data();
    if (rawData) {
        me = {
            name: rawData.loginUserName,
            mail: rawData.loginUserEmail
        };
        getCartId(function (cart) {
            me.cart = cart;
            injectMenu();
            refreshUsers(actions.updateUsers);
            listenOnChanges(onChange);
            listenOnOrderConfirm(onConfirm);
        });
    } else {
        callback('no user');
    }
}
```

```go
func TaskSayHello(t *tasking.T) {
  username := t.Flags.String("name")
  if username == "" {
    user, _ := user.Current()
    username = user.Name
  }

  if t.Flags.Bool("verbose") {
    t.Logf("Hello %s, the time now is %s\n",
           username, time.Now())
  } else {
    t.Logf("Hello %s\n", username)
  }
}
```

```python
def _load_rule_file(self, filename):

        if not (os.path.exists(filename)):
            sys.stderr.write(
              "rflint: %s: No such file or "
              "directory\n" % filename
            )
            return
        try:
            basename = os.path.basename(filename)
            (name, ext) = os.path.splitext(basename)
            imp.load_source(name, filename)
        except Exception as e:
            sys.stderr.write(
              "rflint: %s: exception while "
              "loading: %s\n" % (filename, str(e))
            )
```

```go
func Backup(filename string) error {

  info, err := os.Stat(filename)
  if err != nil {
    if os.IsNotExist(err) {
      return nil
    }
    return err
  }
  if info.Size() == 0 {
    return nil
  }

  files, err := filepath.Glob(
    filename + _BACKUP_SUFFIX
  )
  if err != nil {
    return err
  }

  numBackup := byte(1)

  if len(files) != 0 {
    lastFile := files[len(files)-1]
    numBackup = lastFile[len(lastFile)-2] + 1
    if numBackup > '9' {
      numBackup = '1'
    }
  } else {
    numBackup = '1'
  }

  return Copy(filename,
              fmt.Sprintf("%s+%s~", filename,
                          string(numBackup)))
}
```

Table 6: Selected snippets starting with `is`, `main`, or `load` (left) and their best embedding matches from other languages (right).

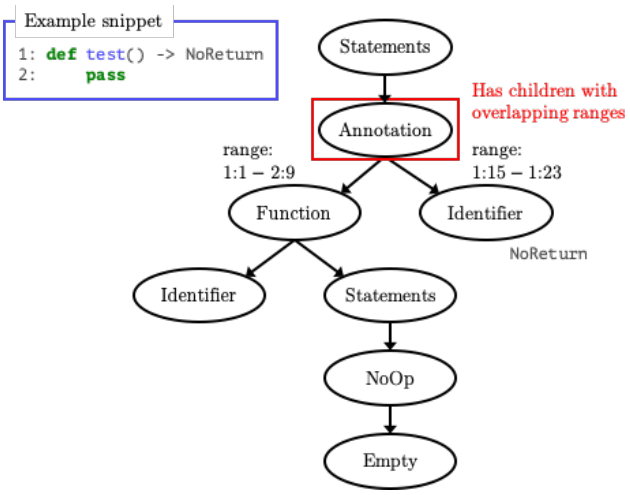

Figure 6: Example snippet and its corresponding AST obtained from GitHub Semantic.

3. **Empty lines** are removed.
4. **Hard coded strings and numbers** are replaced with a special `[MASK_STRING]` and `[MASK_NUMBER]` token.
5. **Indentation** style of the code snippet is detected and whitespace characters at the beginning of a line are replaced with a single `[INDENT]` or `[DEDENT]` token when indentation changes.
6. Tokens are further **split into sub tokens**, e.g., setBottomHeight → ['set', 'bottom', 'height']. Throughout our experiments, we use 5 input sub tokens. If a token consists of less than 5 sub tokens, the remaining spaces are filled with a special `[PAD]` token.
7. Any remaining tokens that only consist of **white spaces are removed**. The only white space characters that are kept are line breaks '\n'.
8. Any code snippets where the Pygments tokenizer **cannot parse a token** are **discarded**.

A.3.2 Stage 1 Preprocessing (Generation of ASTs)

1. Stripped code snippets are used to **generate language-specific ASTs**. For Java, we use the AST parser from the **java-parser** project. The ASTs contain node types and source ranges. For Python, JavaScript, Ruby and Go, we use **semantic**.
2. Snippets that lead to an **AST parse error are discarded**.
3. We calculate a **mapping between tokens and nodes in the AST**. Every token is assigned to the node in the AST with shortest source range that still encompasses the source range of the token.
   To find such a node, we originally intended to make use of the assumption that source ranges of child nodes do not overlap. Then, one could easily find the node with smallest encompassing source range by greedily selecting at every layer in the AST the child that encompasses the token's source range (there can only be at most one child that fulfills this). However, this assumption does not hold for all ASTs (see Figure 6 for an example). As a heuristic, we greedily select the child node with the shorter source range in case there were multiple child nodes with encompassing source ranges. This approximation seems to be sufficient in our case, and limits runtime as we do not have to consider multiple paths in the AST. It is also sufficient to stop when no child node encompasses the source range of the token, as in ASTs the source ranges of child nodes are always contained in the source ranges of their parent.

A.3.3 STAGE 2 PREPROCESSING (CALCULATION OF DISTANCE MATRICES)

1. Tokens are **vocabularized**. Any token occurring less than 100 times in the training set is **replaced by an** `<unk>` token.
2. We calculate multiple **pair-wise relations** between nodes in the AST:
   - Personalized Page Rank (PPR)
     We interpret the negative logarithm of PPR as a distance. We use a teleport probability of $\alpha = 0.15$ and a threshold of e^{-5}, i.e., anything with $-\log PPR > 5$ is considered unreachable
   - Shortest path length between two nodes
   - Ancestor shortest paths (bidirectional). That is, the parent has an ancestor shortest path distance of 1 to all its children and the child has a distance of -1 to its parents. We consider nodes that are not ancestors or descendants of a node (i.e. not reachable by following only parent or only child relations) as not connected in the ancestor shortest paths relation. We encode this with a very large value in their distance; we have found a value of $1,000$ to work well in practice.
   - Next sibling shortest paths (bidirectional, analogous to the ancestor shortest paths)

   Note that the ancestor shortest paths and next sibling shortest paths are required because treating the AST as a normal graph leads to ambiguity. In a graph, the neighbors of a node have no ordering; however in the AST, the order of the children of a node reflects their order in the code. Therefore, we explicitly include the next sibling shortest paths. The ancestor shortest paths would not be required if we treated the AST as a directed graph; in this case, however, a leaf node could not reach any other node in the AST, and therefore both PPR and shortest path length are not useful in this case. Therefore, we model the AST as undirected and inject the ancestor / child edges to avoid ambiguity.
3. **Distance values are binned** into 32 bins using area-based exponential binning with a growth factor of 1.3, i.e., the area of a bin's rectangle (x: bin range, y: number of values in bin) will be approximately 1.3 times bigger for the next bin (going away from the bin that contains the zero value). Additionally, for discrete distance measures (such as sequence distance or shortest path length), we hard-code 9 values around 0 to have their own bins. For instance, on the sequence distance the values $-4, -3, \ldots, 4$ have their individual bins, and around those values we employ the exponential binning.
4. Punctuation tokens (such as points or brackets) are removed from the input sequence, as experiments showed that their presence does not improve performance but slows down training due to bigger input sizes.
5. Snippets that are longer than `MAX_NUM_TOKENS` after punctuation tokens are removed are discarded from the training set. Throughout our experiments, we use `MAX_NUM_TOKENS = 512`. During evaluation on the test set, we use `MAX_NUM_TOKENS = 1000`.

A.4 INPUT EMBEDDINGS TO THE MODEL

Besides its five subtokens (e.g., `['get', 'data', '[PAD]', '[PAD]', '[PAD]']`), each input token has a token type (coming from the Pygments tokenizer) and an AST node type. The AST node type is the type of the node assigned to each respective token, as described in Section A.3.2. We concatenate the embeddings of the five subtokens, the token type, and the AST node type. Then, we apply a linear layer (without activation function) to project down to the model's embedding dimension.

A.5 INPUT TO THE GREAT BASELINE

As mentioned in the main text, we also compare with GREAT Hellendoorn et al. (2020). Since their preprocessing pipeline is proprietary and could not be shared with us even after contacting the authors, we provide to GREAT the same AST distances as our model. Since GREAT uses edges instead of distances to encode relations in the Structure, we essentially threshold the ancestor, sibling, and shortest-paths distances and provide the edges where the distances are equal to 1 (including their edge types) to the model.

(a) CODE TRANSFORMER

Hyperparameter	Value
Activation	GELU
Input Nonlinearity	tanh
Num. layers	3
d	1024
d_{FF}	2048
p_{dropout}	0.2
Num. heads	8

(b) GREAT (Hellendoorn et al., 2020)

Hyperparameter	Value
Activation	GELU
Num. layers	3
d	1024
d_{FF}	2048
p_{dropout}	0.2
Num. heads	8

Table 7: Code Summarization hyperparameters

A.6 EXPERIMENTAL SETUP

Table 7 shows hyperparameters of our models for code summarization. For all our experiments, we use a Transformer Decoder with one layer and teacher forcing to generate 6 output sub tokens. We also employ label smoothing of 0.1. As optimizer, we use Adam with a learning rate of $8e^{-5}$ and weight decay of $3e^{-5}$. Batch size during training is 8 with a simulated batch size of 128 achieved by gradient accumulation.

Apart from comparing the CODE TRANSFORMER to baselines, we performed the following hyperparameter comparisons and ablation studies:

- CODE TRANSFORMER (structure-only)
  Using only AST information as input, i.e., masking all tokens that do not correspond to a leaf of the AST, and removing the token distance as a relation to be used by the model. Further, token types are not fed into the model.
- CODE TRANSFORMER (context-only)
  Here, we do not include any information on the AST (i.e. node types and distances on the AST). This is effectively the XLNet backbone plus encoding of the token type returned by the tokenizer.
- CODE TRANSFORMER (Max-Dist.)
  Applying a Max Distance Mask of 5 to the shortest paths distance (i.e., model cannot see a node that is more than 5 hops away no matter how small the other distances are). Early results showed that, as expected, results deteriorate substantially when limiting our model's receptive field. Hence, we do not include these results in this work.
- Using 16 and 64 bins instead of 32 bins. This had no noticeable effect on performance.

A.7 CODE SUMMARIZATION EXAMPLES

In the Tables 8, 9, 10, 11, 12, 13, 14 and 15 we present example functions from the Java-small dataset along with the different models' predictions for the function name.

```
public Summation next() {
    return parts[i++];
}
```

Model	Prediction
GREAT	get x map
code2seq	get parts
Ours w/o structure	get
CODE TRANSFORMER	get next
Ground Truth	next

Table 8: The CODE TRANSFORMER is the only model to correctly identify the notion of getting the *next* entry.

```
private Path findCacheFile(Path[] cacheFiles, String fileName) {
    if (cacheFiles != null && cacheFiles.length > 0) {
        for (Path file : cacheFiles) {
            if (file.getName().equals(fileName)) {
                return file;
            }
        }
    }
    return null;
}
```

Model	Prediction
GREAT	get path
code2seq	find file
Ours w/o structure	get file
CODE TRANSFORMER	find cache
Ground Truth	find cache file

Table 9: The CODE TRANSFORMER is the only model to both recognize that the task is to *find* a file as well as the fact that it is about the cache. However, it did not correctly predict the *file* part of the method name.

```
public int compare(Pair<LoggedJob, JobTraceReader> p1,
                   Pair<LoggedJob, JobTraceReader> p2) {
    LoggedJob j1 = p1.first();
    LoggedJob j2 = p2.first();
    return (j1.getSubmitTime() < j2.getSubmitTime()) ? -1
            : (j1.getSubmitTime() == j2.getSubmitTime()) ? 0 : 1;
    }
```

Model	Prediction
GREAT	run
code2seq	get submit time
Ours w/o structure	compare
CODE TRANSFORMER	compare
Ground Truth	compare

Table 10: The CODE TRANSFORMER and the its context-only variant are the only models correctly recognizing the 'compare' template in the method body.

```
public static MNTPROC fromValue(int value) {
    if (value < 0 || value >= values().length) {
        return null;
    }
    return values()[value];
}
```

Model	Prediction
GREAT	get value
code2seq	get value
Ours w/o structure	to
CODE TRANSFORMER	from value
Ground Truth	from value

Table 11: The CODE TRANSFORMER is the only model to recognize that the snippet is similar to a static factory method which is often preceded with *from*.

```
private Iterable<ListBlobItem> listRootBlobs(String aPrefix,
                                             boolean useFlatBlobListing,
                                             EnumSet<BlobListingDetails> listingDetails,
                                             BlobRequestOptions options,
                                             OperationContext opContext)
                                      throws StorageException, URISyntaxException {
    CloudBlobDirectoryWrapper directory = this.container.getDirectoryReference(aPrefix);
    return directory.listBlobs(null, useFlatBlobListing,
                               listingDetails, options, opContext);
}
```

Model	Prediction
GREAT	list blobs
code2seq	list blobs
Ours w/o structure	list blobs
CODE TRANSFORMER	list blobs by prefix
Ground Truth	list root blobs

Table 12: All models could correctly identify the *listBlobs()* call in the return statement. However, the CODE TRANSFORMER additionally comprehended that the specified prefix is quite important.

```
private static void dumpOpCounts(EnumMap<FSEditLogOpCodes, Holder<Integer>> opCounts) {
    StringBuilder sb = new StringBuilder();
    sb.append("Summary of operations loaded from edit log:\n  ");
    Joiner.on("\n  ").withKeyValueSeparator("=").appendTo(sb, opCounts);
    FSImage.LOG.debug(sb.toString());
}
```

Model	Prediction
GREAT	append
code2seq	add
Ours w/o structure	log
CODE TRANSFORMER	log op counts
Ground Truth	dump op counts

Table 13: Only the CODE TRANSFORMER could correctly identify that it is the *op counts* that should be logged.

```java
static String execCommand(File f, String... cmd) throws IOException {
    String[] args = new String[cmd.length + 1];
    System.arraycopy(cmd, 0, args, 0, cmd.length);
    args[cmd.length] = f.getCanonicalPath();
    String output = Shell.execCommand(args);
    return output;
}
```

Model	Prediction
GREAT	get canonical path
code2seq	exec
Ours w/o structure	get output
CODE TRANSFORMER	exec command
Ground Truth	exec command

Table 14: Only the CODE TRANSFORMER and code2seq could identify that the relevant part of the method is concerned with executing a command instead of returning something.

```java
protected void subView(Class<? extends SubView> cls) {
    indent(of(ENDTAG));
    sb.setLength(0);
    out.print(sb.append('[').append(cls.getName()).append(']').toString());
    out.println();
}
```

Model	Prediction
GREAT	print
code2seq	print
Ours w/o structure	print
CODE TRANSFORMER	print sub view
Ground Truth	sub view

Table 15: Only the CODE TRANSFORMER was able to link the *print* functionality to the object that should be printed, which can only be inferred from the object's class in the method parameters.

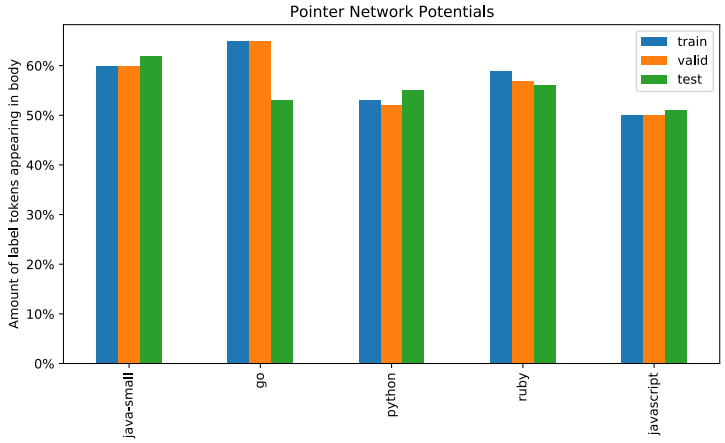

Figure 7: Share of tokens in the labels also occurring in the bodies of methods.

Model	Python			Javascript			Ruby			Go		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
code2seq	-	-	-	-	-	-	-	-	-	-	-	-
GREAT	34.93	31.12	31.61	29.69	24.24	25.55	25.69	21.49	22.18	48.38	45.97	45.71
Ours w/o structure	36.87	32.17	32.97	31.30	25.03	26.64	31.43	25.34	26.63	49.78	46.73	46.69
Ours w/o pointer net	**38.77**	31.72	33.27	32.70	25.50	27.33	**32.12**	**30.17**	**29.36**	**53.09**	**48.70**	**49.26**
Ours	36.68	**33.86**	**33.84**	**33.36**	**27.55**	**29.02**	31.53	24.72	26.43	52.00	47.35	47.93
code2seq (Multilanguage)	-	-	-	-	-	-	-	-	-	-	-	-
GREAT (Multilanguage)	35.73	30.81	31.74	31.49	26.17	27.41	29.72	24.20	25.43	50.32	47.94	47.66
Ours w/o structure (Mult.)	36.78	29.92	31.58	32.60	26.02	27.74	31.71	26.07	27.24	51.91	47.58	48.15
Ours w/o pointer (Mult.)	37.18	30.52	32.04	33.95	25.92	28.11	32.76	25.04	27.01	53.50	48.54	49.35
Ours (Multilanguage)	**38.10**	**33.32**	**34.18**	**34.29**	**28.69**	**30.08**	**33.30**	**28.33**	**29.29**	**53.86**	**50.46**	**50.61**
Ours (Mult. + Finetune)	38.29	32.41	33.65	34.43	28.28	29.91	32.89	27.15	28.49	53.85	50.85	50.81
Ours (Mult. + LM Pretrain)	**38.97**	**34.77**	**35.34**	**35.23**	**30.26**	**31.38**	**33.73**	**29.15**	**29.94**	**55.31**	**52.03**	**52.13**

Table 16: Code summarization results on the CSN dataset (sample-F1).

A.8 Estimation of Pointer Network Potential

In Table 2 we observe that the pointer network improves the F1 score for all languages except Go, where counterintuitively it leads to reduced performance as measured by F1 score on the test set (while it improves by about 3 points on validation). To investigate this, in Figure 7 we plot the share of tokens in the labels also occurring in the bodies of methods in the different languages. Intuitively, this gives an indication on how much gain we can expect from using a pointer network. If the share were zero, then *no* token in the labels ever occur in the bodies of the methods, so the pointer network cannot improve the prediction by pointing at the input. We see that for Go, there is a strong mismatch between the test partition and the train/validation partitions, which much fewer tokens from the labels occurring in the bodies of methods on test compared to train/validation. Thus, we attribute the drop in performance observed by adding a pointer network on Go to this apparent violation of the i.i.d. assumption.

A.9 Code Summarization Results on the CSN dataset (sample-F1)

In Table 16, we present our results on the CSN dataset as measured by the sample-F1 score.