Pythonと日本語変換：Trans-CompilerとTransformerの比較

縫嶋 慧深[1,a)]　秋信 有花[2]　倉光 君郎[1,b)]

概要：近年，深層学習技術の登場により，文書要約や質問応答など，自然言語処理は目覚ましい発展を遂げている．我々は，これらの技術発展をソフトウェア開発においても享受するため，ソースコードを形式化された自然言語に変換する手法を提案する．本発表では，Pythonコードを日本語記述に変換する過程で，従来の形式的な変換に基づくTrans-Compilerと最新のニューラル機械翻訳技法であるTransformerによる変換を比較する予定である．また，逆変換となる自然言語からのコード生成などの展望について述べたい．

キーワード：ソースコード，深層学習，トランスコンパイラ，機械翻訳

1. はじめに

ソフトウェア開発で用いられるソースコードは，数千万行にも及び，それらは膨大なデータ資源をなしている．しかし，膨大なデータ資源にも関わらず，人間による理解と手作業によるメンテナンスに頼っている．近年，ビックコード解析 [1][2] として深層学習技術によるメンテナンスの効率化に期待が集まる理由である．

何がソースコードを深層学習で処理するときの壁になるのだろうか？ソースコードは，プログラミング言語など厳密な意味論によって記述され，形式的に解釈できるようになっている．この形式的な解釈は，深層学習の確率モデルに基づく解釈とは本質的に異なる．

ソースコードを確率モデルで学習できるようにするため，ソースコードの構造的な特徴（抽象構文木）などに着目し，ベクトル表現の研究 [3], [4], [5], [6] が盛んに行われている．これらの研究は，近年高い関心を集めるテーマ [2], [7], [8] となっているが，定番となる手法の発見や確立には至っていない．

我々は，全く新しいアプローチによるソースコードの深層学習への道を探究している．アイディアの骨子は，次の通りである．

- ソースコードは，一旦，自然言語記述に変換する
- 深層学習は，変換された自然言語記述に対して適用する

この利点は，深層学習用の表現として，自然言語を介在させることで，自然言語処理分野で急激に進展する学習手法 [9], [10], [11] が導入できることである．特に，最先端の自然言語処理分野で

[1] 日本女子大学理学部数物科学科
Department of Mathematical and Physical Sciences, Japan Women's University, 2-8-1 Mejirodai, Bunkyo-ku, Tokyo 112–8681, Japan
[2] 日本女子大学大学院理学研究科数理・物性構造科学専攻
Graduate School of Science Division of Mathematical and Physical Sciences, Japan Women's University, 2-8-1 Mejirodai, Bunkyo-ku, Tokyo 112–8681, Japan
[a)] m1716070ne@ug.jwu.ac.jp
[b)] kuramitsuk@fc.jwu.ac.jp

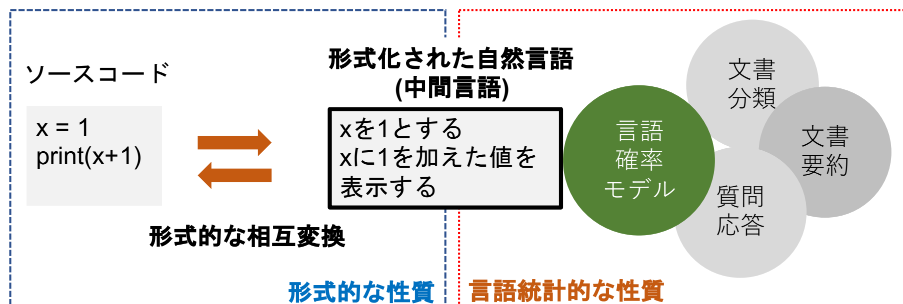


図 1 研究構想図

開発が進む大規模な事前学習済み言語学習モデル (BERT[12], [13], T5[14]) が活用でき，大幅な精度の向上も期待できる．また，最終的な大規模なソースコードの保守を考えたとき，自然言語による文書要約や質問応答はシステムの一部となるため，実用性の面からも期待できるアプローチといえる．

しかし，大きな課題も残っている．「ソースコードから自然言語への変換」は，それ自体が深層学習の応用領域 [15], [16], [17] として取り組まれてきた．このままでは，深層学習のために深層学習が必要となる．

我々はこのループを回避するため，トランスコンパイラ方式の形式的な変換を考えている．ソースコードは，プログラミング言語で記述されているため，形式的な扱いで（マシン語の代わりに）自然言語に変換することができる．加えて，形式的に制限された自然言語を新たに導入することで，自然言語からソースコードへの形式的な変換の実現も目指している．

本稿では，トランスコンパイラの試作ツールと予備実験の結果を報告する．第 2 節では，研究構想の背景を外観し，動機について述べる．第 3 節では，ニューラル機械翻訳の技法を紹介する．第 4 節では，トランスコンパイラの設計と実装を述べる．第 5 節では，変換された日本語についての予備実験について述べる．第 6 節では関連研究を概観する．第 7 節では本研究をまとめる．

2. 動機と構想

まず，本研究の動機と構想を概観しておきたい．

2.1 コードと分散表現

深層学習では，ニューラルネットワーク上で学習を進めるため，入力データと出力データを多次元のベクトルで表現する必要がある．画像認識は，深層学習をリードしてきた応用であるが，画像自体が多次元ベクトルデータといえるため，深層学習とは親和性がよい分野であった．一方，言語は系列をもった非ベクトルのデータであり，多次元ベクトルの表現方法が課題となった．近年，単語分散表現 [18], [19] が積極的に開発され，単語の意味を多次元ベクトルで近似できるようになってきた．

プログラミング言語は，自然言語と同様に字句で構成される系列データである．しかし，word2vec のような単語分散表現をそのまま適用しても，コードの意味を正しく捉えることはできない [7]．プログラミング言語コミュニティでは，構文木などの構造情報を含めた多次元ベクトル化 [5], [8], [20] が盛んに研究されているが，形式言語の意味論をベクトルで近似することは容易な道のりではない．

2.2 自然言語との連携

近年，深層学習による自然言語処理技法は，機械翻訳や文書生成などの応用をみても劇的な進歩を遂げている．これは，単語分散表現の基礎 [18], [19] が進んだだけでなく，その後の深層学習モデル (seq2seq[9], [10], ELMo[21], Transformer[11], BERT[12]) の発展に支えられている．さらに，事前学習による大規模な言語モデルの構築が進み，知識やノウハウを取り込んだ処理も実用的になっている．

我々の着眼点は，ソースコードを一旦，形式的な変換によって自然言語記述に変換してしまう点である．ソースコードが変換できれば，あとは深層学習による自然言語処理技法が適用しやすくなる．図 1 は我々の構想を示している．ポイントは，日本語などの自然言語をそのまま中間言語に用いるのではなく，形式的な操作をしやすくした自然な中間言語を新たに導入する点である．

2.3 形式化された自然言語

我々は，自然言語記述の形式的な操作性，将来的なソースコード間との双方向変換を考慮に入れて，形式言語と自然言語の性質を兼ね備えた中間言語の導入を考えている．このような言語を，現在，日本語ベースの言語設計を考えているため，PJ(Programmable Japanese) と呼んでいる．

PJ は，解析表現文法で文法が形式的に定義されて，日本語に対し制限をかけることで，字句レベルの曖昧さを取り除いている．また，PJ では，形式言語で一般的な結合則のルールを導入する．原則，係り受けは，最も近くに出現する左候補に係ることにする．

- 望遠鏡で子犬が泳ぐのを見た

「望遠鏡で」は右側の動詞節に係る．候補は，「泳ぐ」と「見る」であるが，最も近いのは「泳ぐ」であるため，泳ぐに係ることにする．

もちろん，これは自然な解釈とは異なる．したがって，正しい解釈を促すためには，「望遠鏡で」の位置を変更して書くことになる．

- 子犬が泳ぐのを望遠鏡で見た

PJ では，文節のグループ化の記法 {{ }} [*1]を導入している．これは，数式やプログラミング言語の ( ) と同じように，結合の優先度を指定するものである．グループ化を用いると，語順を変更することなく書き下すことができる．

- 望遠鏡で {{ 子犬が泳ぐのを }} 見た

このように自然言語を形式化しておくことで，既存のプログラミング言語と PJ の間では，構文的な違いを超えて形式的に変換できることが期待される．

[*1] {{ }} の記号は， Latex に由来している．文書生成される．

3. ニューラル機械翻訳

機械翻訳は，ルールベース機械翻訳 (RMT)，統計的機械翻訳 (SMT)，ニューラル機械翻訳 (NMT) に大別される．近年，深層学習技術の発展にともない，大量の対訳文を教師データとした NMT が大きく発展し，seq2seq や Transformer などの優れた学習モデルが登場した．本節では，これらを概観する．

3.1 Seq2seq

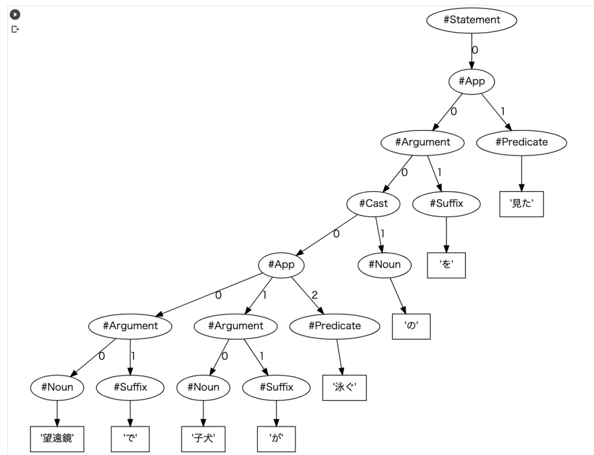
Seq2seq は，ニューラルネットワークを備えた Encoder と Decoder を用いて，系列データを別の系列データに変換するモデルである．Encoder では，ある入力を固定長の特徴量ベクトルに符号化し，Decoder でエンコードされた特徴量ベクトルを復号化することによって新しい系列データを出力する．特徴は，ゲート付きの RNN (Recurrent Neural Network) を導入することで，より長い時系列データに対しても効率よく学習できる点である．主要なゲート付き RNN には，LSTM (Long-Short Term Memory) や GRU (Gated Recurrent Unit) がある．

提案当初の seq2seq では，Encoder が入力系列の長さによらず固定長ベクトルに変換してしまうため，長い系列の特徴を捉えることが難しかった．現在の実用的な seq2seq では，Decoder 側に出力系列と入力系列の参照を行う Attention 機構が導入されることが多い．Attention 機構を導入することによって，Encoder 側の入力系列の長さを考慮し，より長い文にも対応できるようにしている．

3.2 Transformer

Transformer は，RNN を使用せず Attention 機構のみを用いた Encoder-Decoder モデルである．Transformer の特徴は，self-attention 機構を使用している点である．Self-attention 機構では，入力データの単語のみを用いて単語間の関連度を計算している．これにより，従来モデルに比べてより適切な単語依存関係を把握することに成功した．

入力: 望遠鏡で子犬が泳ぐのを見た



入力: 望遠鏡で {{ 子犬が泳ぐの }} を見た

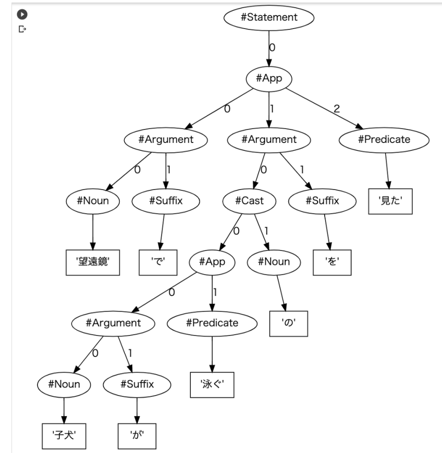


図 2　PJ パーサによる構文木の出力

また，個々のデータがデータ全体から並列的に依存関係を参照するため，より広範囲の依存関係を把握できるようになった．

4.　トランスコンパイラ

Kotonoha は，コンパイラ方式の意味解析に基づいて，ソースコードから日本語記述を出力するトランスコンパイラである．

4.1　概要

自然言語処理において，機械翻訳が難しい理由のひとつは，自然言語の曖昧さにある．つまり，人間にとって自明な解釈であっても，構文レベルから曖昧さがあり，さらに語彙レベルでも多義的かつ文脈依存性がある．したがって，自然言語を形式的に変換すると，おかしな訳になることが多い．

プログラミング言語は，構文的な曖昧さは取りのぞかれ，解釈においてもプログラム意味論として曖昧さがない．したがって，ソースコードを形式的な操作で扱っても，自然言語のような曖昧さや多義性の影響は受けない．

Kotonoha は，コンパイラ方式の意味解析に基づいて，日本語訳を出力するトランスコンパイラである．機械翻訳的には，ルールベース (RMT) に相当するが，プログラミング言語の形式的性質から，

Python	日本語
`[]`	空のリスト
`A + B`	A に B を加えた値
`A == B`	A と B は等しいかどうか
`A = B`	B を A とする
`A,B = B,A`	A と B を入れ替える
`A,B = C`	C から値を取り出し， A と B とする
`int(X)`	X の整数値
`print(A)`	A を表示する
`print(end='')`	改行せずに

図 3　コーパス・ルール（抜粋）

決定的な構文解析にもとづき，抽象構文木に変換される．そのあと，あらかじめ用意された変換規則に基づいて自然言語記述に帰納的に変換される．

4.2　コーパス・ルール

Kotonoha は，パラメータ化した対訳コーパスを用いて，変換規則を表現する．本稿では，便宜上，$A, B, \ldots$ をパラメータとする．図 3 は，定義された変換規則から抜粋した例である．

Kotonoha は，ソースコードと自然言語の対訳を出力するが，これは次のステージの深層学習からみると，対訳コーパスを出力することになる．したがって，変換規則のことを（対訳コーパスを生

成するという点から）コーパス・ルールと呼ぶ．

コーパス・ルールは，再帰的構成可能である必要がある．つまり，パラメータ化された部分は，文脈に依存することなく，再帰的にコーパス・ルールが適用されるため，自然言語として可読性が保たれるように対訳を与える必要がある．我々は，次のように対訳を与えている．

- 評価値のある式や関数は名詞で終わる
- bool 型の式は「かどうか」で終わる
- 文は基本的に動詞の終止形

Kotonoha は，より適切な対訳になるように，構文レベルで一部，最適化処理を行っている．例えば，多重代入は通常の代入に異なり，より具体的な訳が与えられる．一方，型によってより適切な対訳に切り替えることも考えられるが，現在はほとんど型情報を活用した対訳生成を行っていない．（今後の検討課題となっている．）

日本語対訳は，解析表現文法で形式的に定義された PJ 文法で記述している．原理的には，PJ からソースコードへの変換も，形式的に行えることが期待されるが，語彙の多義性が残っているためにルールの適用が一意に決まらないことがある．例えば，「表示する」のコードは，必ずしも `print()` 関数とは限らない．PJ から逆変換を，どのようにすべきかは現時点でオープンクエスチョンである．

4.3 Kotonoha の試作実装

最後に，Kotonoha の試作実装を紹介する．現在の実装は，Python コードを対象にして実装されている．構文解析器は，PEG パーサ生成器の一種である PEGTree[22] を用いて，抽象構文木に変換し，トップダウンでコーパスルールを適用して変換する．型解析は行っていない．

図 4 は，Kotonoha によるモンテカルロ法のソースコードの変換例を示したものである．原則，変換されたフレーズは PJ コーパスによる．工夫としては，式 1+2 は，{{ 1 に 2 を足す }} のように動詞で終わるのではなく，{{ 1 に 2 を足した値 }} のように名詞で終わるようになっている．Kotonoha は，活用形の形式的な操作によって自然な日本語に整形して出力している．

5. 予備実験

本節では，Kotonoha が変換した日本語訳，そしてそれらをコーパスとしたニューラル機械翻訳に関して評価する．なお，本節で報告される実験結果は，予備的なものであり，評価もインフォーマルな段階であることをあらかじめお断りしておきたい．

我々は，Kotonoha を用いて，Python ソースコードを PJ 文に変換した．ここで，ソースコードは，オイラー問題の解法に対して日本語コメントを加えた Euler コーパス [23] を用いた．図 4 は，Kotonoha による PJ 文の例である．下段カッコ内は，Euler コーパスの人手による対訳文である．人手の場合の方がよりコンパクトな対訳を書くことができるが，Kotonoha であっても同品質の対訳を生成することができる．

我々は，2 つのゲート付き回帰型ユニット (GRU) と Attention 機構で構成されるニューラル機械翻訳モデル (以下，seq2seq) を用いて，Kotonoha とコード変換を比較した．表 2 は，Euler コーパスを教師データとして学習した seq2seq モデルで，Python のコードから PJ 文への変換を試みた結果である．Seq2seq では，なかなか期待通りの変換が得られないことがわかる．表 3 は，同じ教師データを用いて，PJ 文から Python のコードを出力した結果である．現時点は，結論を導出しにくい結果であるが，我々は PJ 文から Python コードの方が，確率的言語モデルと相性がよく，変換に期待が持てると評価している．

6. 関連研究

伝統的な機械学習アルゴリズム (決定木 [24], CRF[25], 確率 CFG[26]) は，深層学習が脚光を浴びる前から広くソースコードの解析に応用されてきた．深層学習が登場すると，コードのベクトル表現（分散表現 [8]）は，プログラム理解の中心的なテーマとなった．初期の研究では，コードを単純に字句の列 [7], [27] と見なした．これらは，ソースコードに深層学習が適用できることで大きな一歩となったが，コードを構造的に学ぶチャンスを

```python
import random
c = 0
n = 100
for i in range(n):
    x = random.random()
    y = random.random()
    if x**2 + y**2 < 1.0:
        c += 1

print(c*4/n)
```

```
random モジュールを用いる
c を 0 とする
n を 100 とする
{{0 から n 未満までの数列}}を先頭から順に i として，以下を繰り返す
x を{{0.0 から 1.0 までの乱数}}とする
y を{{0.0 から 1.0 までの乱数}}とする
もし{{{{x を 2 乗した値}}に{{y を 2 乗した値}}を加えた値}}が 1.0 より小さいとき，
c を 1 だけ増加させる

{{{{c を 4 で掛けた値}}に n で割った値}}を表示する
```

図 4　Python コードと変換された PJ（例：モンテカルロ法）

表 1　Python から PJ への変換例 (Kotonoha)

	Python　入力	Kotonoha 出力 (Euler コーパス作者の人手による対訳)
(1)	`sd = 0`	sd を 0 とする (sd に 0 を代入)
(2)	`c += 1`	c を 1 だけ増加させる (c に 1 を足す)
(3)	`s += int(x)`	s を x の整数値だけ増加させる (s に x の表す整数を足す)
(4)	`return s`	s が関数出力となる (s を返す)
(5)	`for i in range(n):`	{{ 0 から n 未満までの数列 }} を先頭から順に i として，以下を繰り返す (以下の処理を n 回繰り返す)
(6)	`if n <= 20:`	もし n が 20 以下のとき， (もし n が 20 以下であれば)
(7)	`if i % y == 0:`	もし {{ i を y で割った余り }} が 0 と等しいとき， (もし i が y で割り切れるなら)
(8)	`while n != 1:`	もし n が 1 と等しくないとき，以下を繰り返す (n が 1 でない間)

逃していた．

続く，研究グループは，抽象構文木 (AST) やプログラムの構造に基づいたベクトル表現 [3], [4], [5], [8], [20], [26], [28] を探究した．これらの成果により，ソースコードを深層学習するときのオプションは広がったが，コードの意味を正しくベクトル表現で近似することは難しく [6], [29]，定番的な手法の確立には至っていない．

7.　むすびに

近年，深層学習技術の登場により，文書要約や質問応答など，自然言語処理は目覚ましい発展を遂げている．我々は，これらの技術発展をソフトウェア開発においても享受するため，ソースコードを形式化された自然言語に変換する手法を提案した．Kotonoha は，ソースコードを変換するため，コンパイラ方式に基づく自然言語対訳生成ツールである．本発表では，Python コードを日本語記述に変換する過程で，Kotonoha と最新のニューラル機械翻訳技法による翻訳を比較した．

我々の構想は研究プロジェクトとして始まったばかりである．今後は，ソースコード要約や機械翻訳，質問応答システムなど，応用を見据えて，自然言語によるソースコード深層学習の基盤を開拓

表 2　Python から PJ への変換例 (seq2seq)

	Python　入力	Py2PJ 出力
(1)	`sd = 0`	res を 0 と する
(2)	`c += 1`	c を 1 だけ 増加 さ せる
(3)	`s += int(x)`	s を s の だけ 増加 さ せる
(4)	`return s`	s が 関数 出力 と なる
(5)	`for i in range(n):`	{{ 0 から n 未満 まで の 数列 }} を 先頭 から 順に i として ， 以下 を 繰り返す
(6)	`if n <= 20:`	もし {{ が b より 小さい とき ， 以下 を 繰り返す
(7)	`if i % y == 0:`	もし {{ を x で 割っ た 余り }} が 0 と 等しい とき ， 以下 を 繰り返す
(8)	`while n != 1:`	もし n が 関数 出力 と なる

表 3　PJ から Python への変換例 (seq2seq)

	PJ　入力	Python 出力
(1)	sd を 0 とする	`m = 0`
(2)	c を 1 だけ増加させる	`c += 1`
(3)	s を x の整数値だけ増加させる	`s += x`
(4)	s が関数出力となる	`return s`
(5)	{{ 0 から n 未満までの数列 }} を先頭から順に i として，以下を繰り返す	`for i in range(2,`
(6)	もし n が 20 以下のとき，	`if n < 2:`
(7)	もし {{ i を y で割った余り }} が 0 と等しいとき，	`if i % i == 0:`
(8)	もし n が 1 と等しくないとき，以下を繰り返す	`while n >= N`

していきたい．

謝辞　本研究を進めるに辺り，有意義なコメントをいただいた情報処理学会プログラミング研究会（第 131 回）の参加者の皆様に感謝します．

参考文献

[1] Allamanis, M., Barr, E. T., Devanbu, P. and Sutton, C.: A Survey of Machine Learning for Big Code and Naturalness, *ACM Comput. Surv.*, Vol. 51, No. 4 (online), DOI: 10.1145/3212695 (2018).

[2] Le, T. H. M., Chen, H. and Babar, M. A.: Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges, *ACM Comput. Surv.*, Vol. 53, No. 3 (online), DOI: 10.1145/3383458 (2020).

[3] Alon, U., Zilberstein, M., Levy, O. and Yahav, E.: A General Path-Based Representation for Predicting Program Properties, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, New York, NY, USA, Association for Computing Machinery, p. 404–419 (online), DOI: 10.1145/3192366.3192412 (2018).

[4] Raychev, V., Bielik, P., Vechev, M. and Krause, A.: Learning Programs from Noisy Data, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, New York, NY, USA, Association for Computing Machinery, p. 761–774 (online), DOI: 10.1145/2837614.2837671 (2016).

[5] Alon, U., Zilberstein, M., Levy, O. and Yahav, E.: Code2vec: Learning Distributed Representations of Code, *Proc. ACM Program. Lang.*, Vol. 3, No. POPL (online), DOI: 10.1145/3290353 (2019).

[6] Wang, K. and Su, Z.: Blended, Precise Semantic Program Embeddings, *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, New York, NY, USA, Association for Computing Machinery, p. 121–134 (online), DOI: 10.1145/3385412.3385999 (2020).

[7] Hellendoorn, V. J. and Devanbu, P.: Are Deep Neural Networks the Best Choice for Modeling Source Code?, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, New York, NY, USA, Association for Computing Machinery, p. 763–773 (online), DOI: 10.1145/3106237.3106290 (2017).

[8] Mou, L., Li, G., Zhang, L., Wang, T. and Jin, Z.: Convolutional Neural Networks over Tree Structures for Programming Language Processing, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, AAAI

Press, p. 1287–1293 (2016).

[9] Sutskever, I., Vinyals, O. and Le, Q. V.: Sequence to Sequence Learning with Neural Networks, *CoRR*, Vol. abs/1409.3215 (online), available from ⟨http://arxiv.org/abs/1409.3215⟩ (2014).

[10] Luong, T., Pham, H. and Manning, C. D.: Effective Approaches to Attention-based Neural Machine Translation, *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, Lisbon, Portugal, Association for Computational Linguistics, pp. 1412–1421 (online), DOI: 10.18653/v1/D15-1166 (2015).

[11] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I.: Attention Is All You Need, *CoRR*, Vol. abs/1706.03762 (online), available from ⟨http://arxiv.org/abs/1706.03762⟩ (2017).

[12] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota, Association for Computational Linguistics, pp. 4171–4186 (online), DOI: 10.18653/v1/N19-1423 (2019).

[13] Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P. and Soricut, R.: ALBERT: A Lite BERT for Self-supervised Learning of Language Representations (2020).

[14] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W. and Liu, P. J.: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (2020).

[15] Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T. and Nakamura, S.: Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation, ASE '15, IEEE Press, p. 574–584 (online), DOI: 10.1109/ASE.2015.36 (2015).

[16] Allamanis, M., Tarlow, D., Gordon, A. D. and Wei, Y.: Bimodal Modelling of Source Code and Natural Language, ICML'15, JMLR.org, p. 2123–2132 (2015).

[17] Zilberstein, M. and Yahav, E.: Leveraging a Corpus of Natural Language Descriptions for Program Similarity, *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, New York, NY, USA, Association for Computing Machinery, p. 197–211 (online), DOI: 10.1145/2986012.2986013 (2016).

[18] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. and Dean, J.: Distributed Representations of Words and Phrases and Their Compositionality, *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, USA, Curran Associates Inc., pp. 3111–3119 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2999792.2999959⟩ (2013).

[19] Pennington, J., Socher, R. and Manning, C. D.: GloVe: Global Vectors for Word Representation, *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543 (online), available from ⟨http://www.aclweb.org/anthology/D14-1162⟩ (2014).

[20] Allamanis, M., Brockschmidt, M. and Khademi, M.: Learning to Represent Programs with Graphs, *International Conference on Learning Representations*, (online), available from ⟨https://openreview.net/forum?id=BJOFETxR-⟩ (2018).

[21] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K. and Zettlemoyer, L.: Deep contextualized word representations, *CoRR*, Vol. abs/1802.05365 (online), available from ⟨http://arxiv.org/abs/1802.05365⟩ (2018).

[22] Kuramitsu, K.: Nez: Practical Open Grammar Language, *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, New York, NY, USA, ACM, pp. 29–42 (online), DOI: 10.1145/2986012.2986019 (2016).

[23] Fudaba, H., Oda, Y., Akabe, K., Neubig, G., Hata, H., Sakti, S., Toda, T. and Nakamura, S.: Pseudogen: A tool to automatically generate pseudo-code from source code, *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, pp. 824–829 (2015).

[24] Raychev, V., Bielik, P. and Vechev, M.: Probabilistic Model for Code with Decision Trees, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, New York, NY, USA, Association for Computing Machinery, p. 731–747 (online), DOI: 10.1145/2983990.2984041 (2016).

[25] Raychev, V., Vechev, M. and Krause, A.: Predicting Program Properties from "Big Code", *Proceedings of the 42nd Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, New York, NY, USA, Association for Computing Machinery, p. 111–124 (online), DOI: 10.1145/2676726.2677009 (2015).

[26] Bielik, P., Raychev, V. and Vechev, M.: PHOG: Probabilistic Model for Code, *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, JMLR.org, p. 2933–2942 (2016).

[27] Hindle, A., Barr, E. T., Gabel, M., Su, Z. and Devanbu, P.: On the Naturalness of Software, *Commun. ACM*, Vol. 59, No. 5, p. 122–131 (online), DOI: 10.1145/2902362 (2016).

[28] DeFreez, D., Thakur, A. V. and Rubio-González, C.: Path-Based Function Embedding and Its Application to Error-Handling Specification Mining, ESEC/FSE 2018, New York, NY, USA, Association for Computing Machinery, p. 423–433 (online), DOI: 10.1145/3236024.3236059 (2018).

[29] Kang, H. J., Bissyandé, T. F. and Lo, D.: Assessing the Generalizability of Code2vec Token Embeddings, ASE '19, IEEE Press, p. 1–12 (online), DOI: 10.1109/ASE.2019.00011 (2019).