修　士　論　文

題　目　Transformerエンコーダの応用による
機械翻訳手法を利用した
プログラム合成モデルの性能の比較

主任指導教員　水野　修　教授

指導教員　崔　恩瀞　助教

京都工芸繊維大学大学院 工芸科学研究科

情報工学専攻

学生番号　19622037

氏　名　舟山　優

令和3年2月10日提出

様式 2 号

学位論文内容の要旨（和文）

令和 3 年 2 月 10 日

京都工芸繊維大学大学院
工芸科学研究科長　殿

工芸科学研究科　情報工学専攻
平成 31 年入学
学生番号　19622037
氏　　名　舟山　優　㊞

（主任指導教員　水野　修　㊞　）

本学学位規則第 4 条に基づき、下記のとおり学位論文内容の要旨を提出いたします。

1. 論文題目

Transformer エンコーダの応用による
機械翻訳手法を利用したプログラム合成モデルの性能の比較

2. 論文内容の要旨（400 字程度）

自然言語の文からプログラムを自動的に生成するプログラム合成では，機械翻訳の技術，特に Encoder-Decoder モデルがよく用いられている．しかし，プログラム合成の先行研究ではデコーダの改良は行われているがエンコーダの改良は行われていない．また，機械翻訳ではエンコーダとデコーダ両方に Transformer を用いた手法が主流であるが，プログラム合成では未だ LSTM を用いた手法が主流である．

本研究では，エンコーダの差異がプログラム合成モデルの性能に与える影響を明らかにするため，既存のプログラム合成モデルのエンコーダを LSTM から Transformer に置き換えた．具体的には，2 種類のエンコーダと 2 種類の既存のモデルを組み合わせて合計 4 種類のモデルを作成し，3 つのデータセットを用いて精度や BLEU スコアを比較した．

実験の結果，プログラム合成では LSTM エンコーダを用いた方が高い性能になりやすく，エンコーダの種類による性能の差はデータセットに依存する傾向があることを確認した．

The Impact of Transformer Encoder for Program Synthesis Model

2021 19622037 *FUNAYAMA Yu*

Abstract

A technique that automatically generates programs from natural language sentences is called *program synthesis*. Program synthesis usually uses machine translation techniques to generate program based on natural language. In particular, program synthesis uses Encoder-Decoder models consisting of *encoder* and *decoder*. While prior studies on the program synthesis have mainly discussed the decoder, the encoder has not been discussed in program synthesis. Therefore, the performance of prior models could be improved by studying the encoder. Actually, LSTM is frequently used as an encoder and a decoder in program synthesis while other models such as Transformer are overlooked.

In this thesis, we replaced the LSTM based encoder of the existing program synthesis models with a Transformer based encoder and clarified the effect of the difference between encoders on the performance of the program synthesis model. For this purpose, we created four approaches using two different encoders and two existing models and compared the accuracy and BLEU scores using three datasets.

Our experiments show that the performance of approaches using the encoder of LSTM tended to be higher, and the performance of such encoders tended to depend on the datasets. This suggests that the performance of the program synthesis models can be improved by using the appropriate encoder for each dataset.

目　次

1. 緒言 **1**

2. 背景と関連研究 **3**

2.1 Encoder-Decoder モデル 3

2.1.1 Seq2Seq 3

2.1.2 Transformer 8

2.1.3 スワップモデル 13

2.2 事前学習 13

2.3 プログラム合成モデル 14

2.3.1 SEQ2TREE 14

2.3.2 TRANX 16

2.4 先行研究の問題点 18

3. 研究の目的 **19**

4. 実験方法 **20**

4.1 実験に使用したモデル 20

4.2 評価指標 21

4.2.1 Accuracy 21

4.2.2 BLEU スコア 21

4.3 データセット 22

4.3.1 使用したデータセット 22

4.3.2 前処理 24

4.3.3 評価方法 25

4.4 学習環境 25

5. 実験結果 **27**

5.1 定量的結果 27

5.2 研究設問への回答 32

5.2.1 RQ1: LSTM エンコーダを用いたプログラム合成モデルと Transformer エンコーダを用いたプログラム合成モデルの性能にはどのような差異があるか 32
5.2.2 RQ2: デコーダの LSTM を 0 で初期化することはプログラム合成モデルの性能に影響するか 32
5.2.3 RQ3: 事前学習を行った場合，プログラム合成モデルの性能はエンコーダの差異によってどのように変化するか 32

6. 考察 33

6.1 研究設問に対する考察 33
6.1.1 RQ1: LSTM エンコーダを用いたプログラム合成モデルと Transformer エンコーダを用いたプログラム合成モデルの性能にはどのような差異があるか 33
6.1.2 RQ2: デコーダの LSTM を 0 で初期化することはプログラム合成モデルの性能に影響するか 33
6.1.3 RQ3: 事前学習を行った場合，プログラム合成モデルの性能はエンコーダの差異によってどのように変化するか 34
6.2 妥当性への脅威 35
6.2.1 構成概念妥当性 35
6.2.2 外的妥当性 35
6.2.3 内的妥当性 36

7. 結言 37

謝辞 **37**

参考文献 **38**

1. 緒言

プログラムを実装する際，多くのプログラマは web を利用して自然言語により情報を検索し，必要な部分をコピー&ペーストして修正している [1,2]．しかし，検索欄にキーワードを入力して必要なソースコード片を探し修正して利用する方法は時間がかかる作業である．検索に用いるキーワードなどからプログラムを直接生成することができれば，実装の効率が向上すると考えられる．

自然言語の文から直接プログラムを生成する技術はプログラム合成と呼ばれる．プログラム合成では自然言語以外にも，入出力例や画像などが入力として用いられている [3,4]．近年の機械学習技術の発展により，プログラム合成では機械学習手法を取り入れた研究が盛んに行われている．特に，自然言語の文を入力とするプログラム合成は自然言語からソースコードに翻訳する問題として扱えるため，機械翻訳の技術を取り入れていることが多い [5–7]．

機械翻訳では Long short-term memory（LSTM）を用いた Sequence-to-Sequence（Seq2Seq）[8] が広く利用されていたが，Attention [9] を用いた Transformer [10] の登場により翻訳の精度が飛躍的に向上した [11,12]．Seq2Seq と Transformer はどちらもエンコーダとデコーダで構成される Encoder-Decoder モデルである．機械翻訳では Encoder-Decoder モデルがよく用いられ，中でも Transformer が現在のデファクトスタンダードとなっている．

プログラム合成においても Transformer を用いた研究は行われているが [13,14]，LSTM を用いた手法が未だ主流である．また，LSTM を用いた手法の多くはデコーダを改良しており，エンコーダについてはほとんど言及されていない．エンコーダには入力文から特徴を抽出する重要な役割があるため，エンコーダの改善はモデル全体の改善につながると考える．機械翻訳ではエンコーダとデコーダのアーキテクチャを複数の組み合わせで比較した研究 [15] があるが，プログラム合成では行われていない．以上のことから，プログラム合成には改善の余地が多く残されていると考える．

本研究では，LSTM と Transformer の 2 種類のエンコーダを用い，エンコーダのアーキテクチャの差異によるプログラム合成モデルの性能への影響を明らかにする．2 種類のエンコーダを適用する実験対象として，既存のプログラム合成モデル

の SEQ2TREE [16] と TRANX [17] を採用した．実験には，生成するプログラムの言語やドメインが異なる 3 つのデータセット [16,18,19] を用い，評価指標には Accuracy と BLEU スコア [20] を用いた．

以降の本稿の構成を紹介する．第 2 章では本研究の背景となる Encoder-Decoder モデルや事前学習，プログラム合成モデル，先行研究の問題点について述べる．第 3 章では本研究の目的と研究設問について述べる．第 4 章では実験に使用したモデルや評価指標，データセット，学習環境について述べる．第 5 章では実験結果を示し，研究設問に回答する．第 6 章では実験結果について考察する．最後に，第 7 章では本研究の結論を述べる．

2. 背景と関連研究

2.1 Encoder-Decoder モデル

Encoder-Decoder モデルは機械翻訳や Q&A 等，様々なタスクに用いられている機械学習モデルの 1 つである [8,21,22]．Encoder-Decoder モデルはエンコーダとデコーダの 2 つのモジュールで構成されている．本研究で扱う，自然言語の文を入力としプログラムを生成するプログラム合成では，エンコーダは自然言語の文の意味を解釈しベクトルに変換する役割を，デコーダはその変換したベクトルを基にプログラムを生成する役割を持つ．

2.1.1 Seq2Seq

Encoder-Decoder モデルは後述する LSTM を基礎としたモデルの Seq2Seq [8] が有名である．ここでは，後述する Attention を取り入れた Seq2Seq を例にして説明する．この Seq2Seq の概要図を図 2.1 に示す．左の破線で囲まれた部分がエンコーダ，右の破線で囲まれた部分がデコーダとなっている．複数の単語からなる文を生成する場合，デコーダは一度にすべての単語を出力するのではなく，1 単語ずつ順次に出力する．単語を 1 つ出力した後にその出力した単語を次の時刻のデコーダへの入力として用いることで，その時刻までに出力した単語を考慮しながら次の単語を出力する．そのため，図 2.1 のデコーダへの入力は「前の時刻の出力」となっている．

図 2.1 の各層について以下に示す．

(1) Embedding

Embedding（埋め込み層）は，正の整数を固定次元のベクトルへ変換する層である．変換されたベクトルは分散表現と呼ばれる．辞書を用いて ID に変換された単語を埋め込み層に入力することで単語の分散表現が得られる．

(2) LSTM

Long short-term memory（LSTM）は再帰型ニューラルネットワーク（RNN）の一種であり，RNN にゲートと呼ばれる仕組みを加えたアーキテクチャである．RNN と

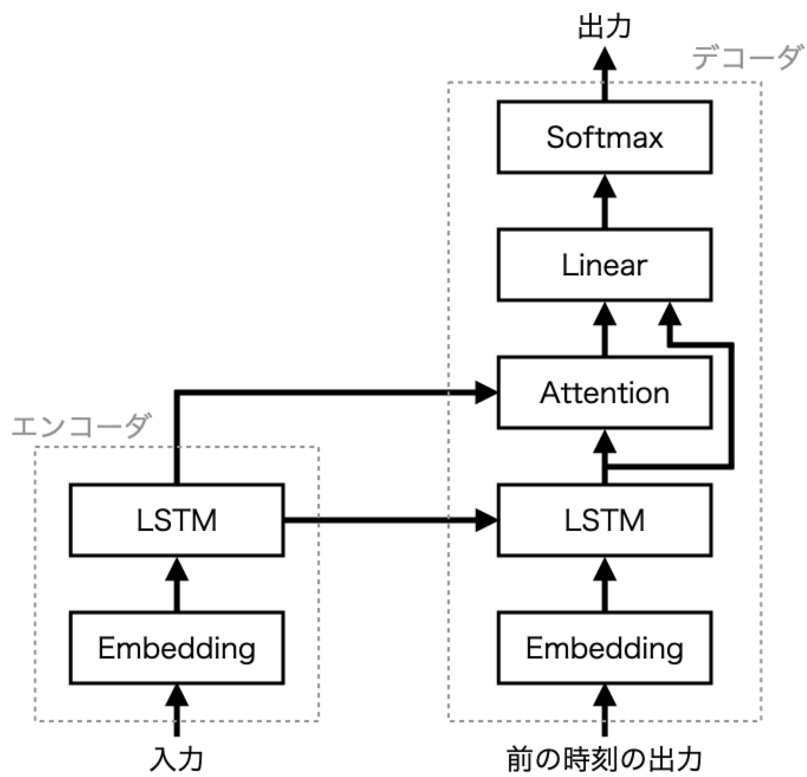

図 2.1　Seq2Seq の概要図

同じく，LSTMは隠れ状態を出力し次の層への入力とするだけでなく，自身への入力にも用いることによって再帰的に計算を行う．LSTMの計算グラフを図2.2に示す．図や以下の式の'σ' はシグモイド関数を表している．

LSTMのゲートには，忘却ゲート，入力ゲート，出力ゲートの3種類がある．

忘却ゲート

図2.2の'f' に相当し，次の式で表される．

$$\mathrm{f} = \sigma(x_t W_x^{\mathrm{f}} + h_{t-1} W_h^{\mathrm{f}} + b^{\mathrm{f}}) \tag{2.1}$$

式中のW_x^{f}は入力x_tに対する重みであり，W_h^{f}は前の時刻の隠れ状態h_{t-1}に対する重みである．b^{f}はバイアスを表す．忘却ゲートは，前の時刻の記憶セルc_{t-1}から不要な情報を捨てる働きをする．

入力ゲート

図2.2の'i' に相当し，次の式で表される．

$$\mathrm{i} = \sigma(x_t W_x^{\mathrm{i}} + h_{t-1} W_h^{\mathrm{i}} + b^{\mathrm{i}}) \tag{2.2}$$

LSTMには忘却ゲートによる不要な情報を捨てる機能だけでなく，新しい情報を得るための機能がある（図2.2の'g'）．これは次の式で表される．

$$\mathrm{g} = \tanh(x_t W_x^{\mathrm{g}} + h_{t-1} W_h^{\mathrm{g}} + b^{\mathrm{g}}) \tag{2.3}$$

入力ゲートはこの機能に対して制御を行う．入力ゲートは追加される新しい情報にどれだけの価値があるのかを判断する．

出力ゲート

図2.2の'o' に相当し，次の式で表される．

$$\mathrm{o} = \tanh(x_t W_x^{\mathrm{o}} + h_{t-1} W_h^{\mathrm{o}} + b^{\mathrm{o}}) \tag{2.4}$$

隠れ状態h_tはc_tにtanh関数を適用することで求められるが，出力ゲートはその際に$\tanh(c_t)$の各要素が次の時刻の隠れ状態としてどれだけ重要であるかを調整する．

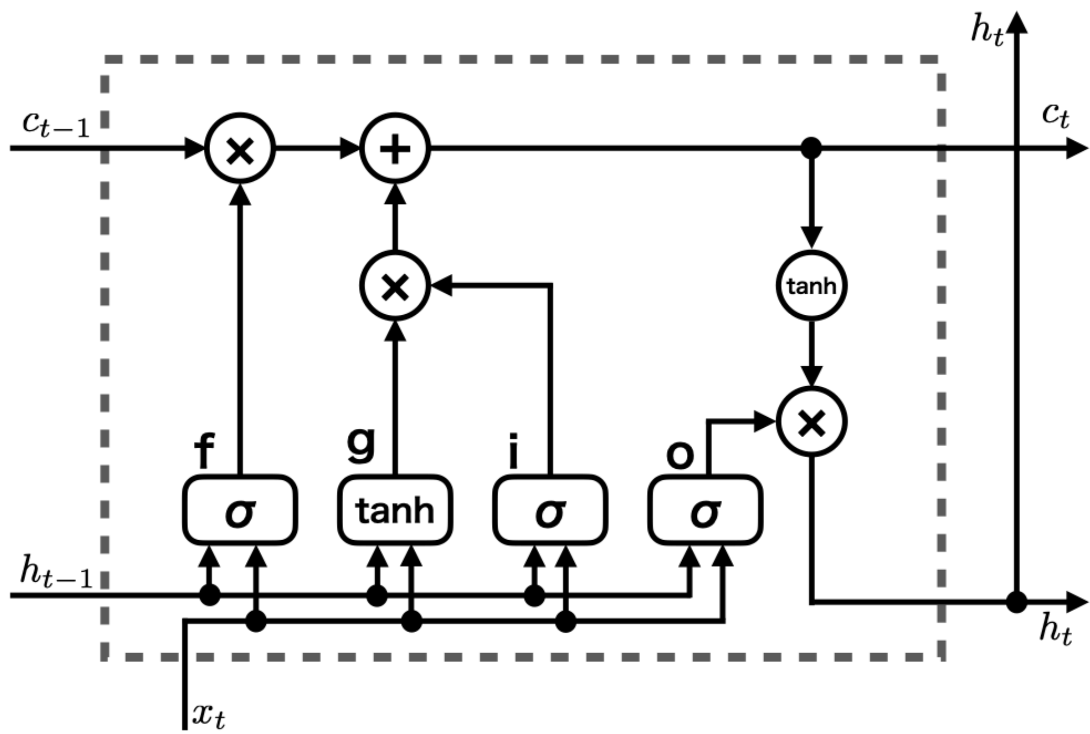

図 **2.2　LSTM** の計算グラフ

これらのゲートを用いて記憶セル c_t と隠れ状態 h_t はアダマール積（$\circ$）を用いて次の式で表される．

$$c_t = \mathrm{f} \circ c_{t-1} + \mathrm{g} \circ \mathrm{i} \tag{2.5}$$

$$h_t = \mathrm{o} \circ \tanh(c_t) \tag{2.6}$$

ゲートと記憶セルを用いることにより，RNN を使用した場合にしばしば問題となる勾配消失を抑制できる．

Encoder-Decoder モデルでは通常，エンコーダの LSTM の隠れ状態と記憶セルは 0 で初期化されるが，デコーダの LSTM の隠れ状態と記憶セルはエンコーダの LSTM の最終出力で初期化される．

(3) Attention

Attention（注意）とは，2 つのベクトル Query と Memory があったとき，それらの類似度を測ることで Memory から重要な情報を抽出する機構のことである [23]．図 2.1 のような Seq2Seq では，エンコーダの LSTM の出力を Memory として，デコーダの LSTM の出力を Query として扱う．エンコーダは入力文の長さにかかわらず固定長のベクトルに圧縮していたが，Attention を用いることで入力の各単語に対応したベクトルも利用できるようになる．

Attention にはいくつかの種類があるが，本研究では内積注意（Dot-Product Attention）[9] を用いる．内積注意は内積を用いて重みを計算する方法であり，パラメータを必要としないためメモリを節約でき，学習も速い．内積注意の計算式を以下に示す．

$$attention_weight = \mathrm{softmax}(QM^{\mathrm{T}}) \tag{2.7}$$

$$context_vector = attention_weight * M \tag{2.8}$$

ここで，Q は Query を，M は Memory を表す．$attention_weight$ は Query と Memory の類似度を表しており，$context_vector$ は Memory から抽出された重要な情報となる．Attention は $context_vector$ を次の層へ出力する．

(4) Linear

本研究で用いる Linear（全結合層）では次のような計算を行う．

$$y = \mathrm{activation}(xW + b) \tag{2.9}$$

ここで，W は重み，b はバイアスである．活性化関数（activation）には tanh や ReLU などから選択できるが，何も指定しなかった場合は活性化関数の引数をそのまま出力する．全結合層は Attention や後述する Transformer の Feed Forward 層などでも用いられており，用途によって指定される活性化関数は異なる．

(5) Softmax

Softmax（ソフトマックス関数）は，入力された数値の合計が 1 になるように変換する関数である．以下にソフトマックス関数の式を示す．

$$y_k = \frac{\exp(x_k)}{\sum_{i=1}^{n} \exp(x_i)} \tag{2.10}$$

2.1.2 Transformer

Transformer とは，Vaswani らが 2017 年に発表したアーキテクチャである [10]．Transformer は自然言語処理においてデファクトスタンダードとなっており，そこからさらに派生したモデルも登場している [11, 12]．

Transformer の特徴として，LSTM などの RNN を用いず Attention を基礎としていることが挙げられる．再帰的な計算を行わないため入力文のすべての単語を同時に処理できるため，並列化が可能となり学習時間が短縮される．

本研究では Transformer のエンコーダ部分のみを扱うため，Transformer のエンコーダ部分について述べる．Transformer エンコーダの概要図を図 2.3 に示す．

(1) Positional Encoding

Positional Encoding（位置エンコーディング）とは，入力の単語列に位置情報を付与するための行列である．RNN では単語列を順次に入力していたが Transformer で

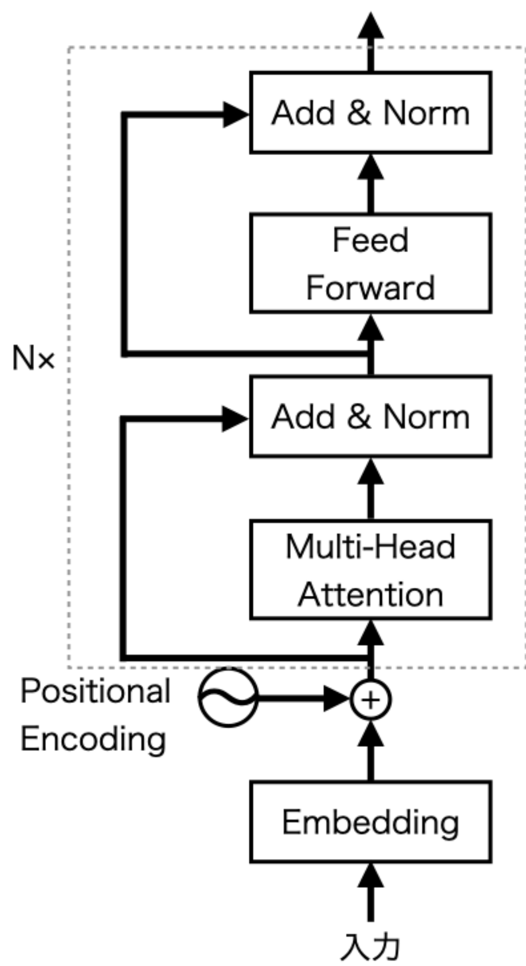

図 2.3　**Transformer エンコーダの概要図**

は同時に入力するため，別途位置情報を与える必要がある．そこで，入力単語の分散表現の各要素に位置エンコーディングを加算する．位置エンコーディング PE は次の式で計算される．

$$PE_{pos,2i} = \sin(pos/10000^{2i/d_{model}}) \tag{2.11}$$

$$PE_{pos,2i+1} = \cos(pos/10000^{2i/d_{model}}) \tag{2.12}$$

ここで，pos は各単語の位置，i は成分の次元，d_{model} はすべての層の出力の次元数である．

(2) Multi-Head Attention

Multi-Head Attention とは，Attention を並列に並べた機構である．その概要図を図 2.4 に示す．また，Multi-Head Attention で用いられている Scaled Dot-Product Attention の概要図を図 2.5 に示す．

Scaled Dot-Product Attention は以下の式で算出される．

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.13}$$

ここで，Q，K，V はそれぞれ Query, Key, Value を表しており，d_k は K の次元数を表している．Key と Value は 2.1.1(3) 節で述べた Attention の Memory と対応している．

図 2.5 の各層について説明する．MatMul 層では内積を計算している．Scale 層では内積を $\sqrt{d_k}$ で除算している．これは d_k が大きい場合，内積が大きくなりソフトマックス関数の勾配が非常に小さくなってしまい，Attention がうまく機能しなくなるためである．Mask 層では，バッチごとの入力文の長さを統一するために用いられる padding など，無視したい部分の重みをソフトマックス関数を適用した際に 0 にするため，入力されたベクトルの一部を $-\infty$ に置き換えている．

Multi-Head Attention では上記の Scaled Dot-Product Attention を用いて，以下の式のように出力ベクトルを算出する．

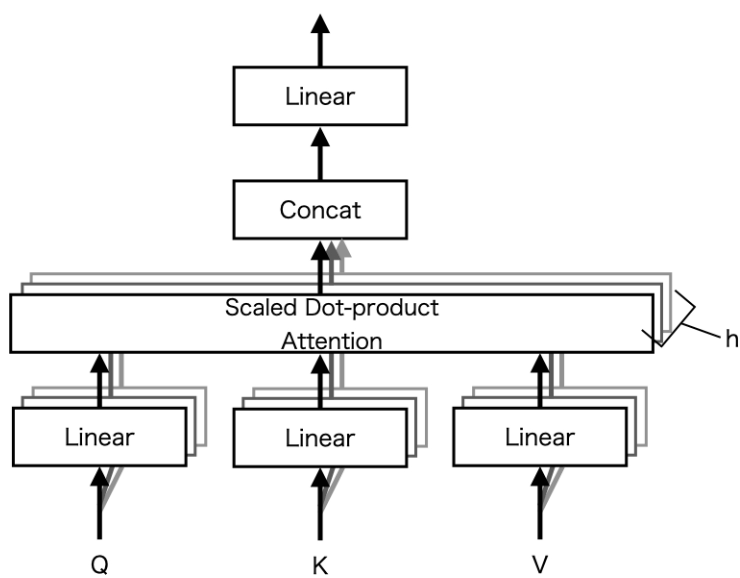

図 2.4　**Multi-Head Attention の概要図**

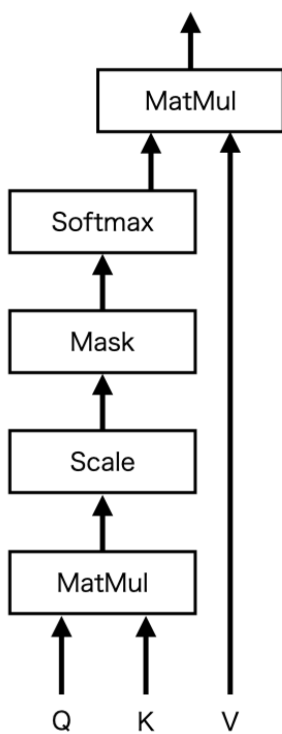

図 2.5　**Scaled Dot-Product Attention の概要図**

$$\mathrm{MultiHead}(Q,K,V)=\mathrm{Concat}(\mathrm{head_1},...,\mathrm{head_h})W^O \tag{2.14}$$

$$\text{where } \mathrm{head}_i=\mathrm{Attention}(QW_i^Q,KW_i^K,VW_i^V) \tag{2.15}$$

ここで，V の次元数を d_v とすると，それぞれの重みは，$W_i^Q\in\mathbb{R}^{d_{model}\times d_k}$，$W_i^K\in\mathbb{R}^{d_{model}\times d_k}$，$W_i^V\in\mathbb{R}^{d_{model}\times d_v}$，$W^o\in\mathbb{R}^{hd_v\times d_{model}}$ である．単一の Attention で計算するのではなく，図 2.4 のように Q，K，V を h 個に分割してそれぞれで Attention を計算する（2.15 式）．その後，分割する前と同じ次元になるように h 個の head_i を結合し，最後に全結合層へ入力する（2.14 式）．

図 2.3 の通り，Transformer エンコーダの Multi-Head Attention への入力は 1 つである．これは，Q，K，V が同じベクトルであることを表す．このような Attention は自己注意（Self-Attention）と呼ばれる．

(3) Feed Forward

図 2.3 の Feed Forward は Position-wise Feed-Forward Network（位置毎のフィードフォワードネットワーク, FFN）を表しており，2 層の全結合層で構成されている．中間層である 1 層目の活性化関数には ReLU を用いる．FFN の計算式は以下の通りである．

$$\mathrm{FFN}(x)=\max(0,xW_1+b_1)W_2+b_2 \tag{2.16}$$

ここで，FFN の中間層の次元を d_{ff} とすると，それぞれの重みは，$W_1\in\mathbb{R}^{d_{model}\times d_{ff}}$，$b_1\in\mathbb{R}^{d_{ff}}$，$W_2\in\mathbb{R}^{d_{ff}\times d_{model}}$，$b_2\in\mathbb{R}^{d_{model}}$ である．

(4) Add & Norm

Transformer では残差スキップ接続 [24] と Layer Normalization [25] を採用しており，図 2.3 の Add & Norm でこれらを行う．残差スキップ接続は変換前の入力の情報を保持しておく手法である．また，Layer Normalization は正規化手法の 1 つである．どちらの手法も深層学習を行う上で問題となる勾配消失や勾配爆発を抑えるために用いられる．

2.1.3 スワップモデル

スワップモデルとは根石らが提案した，Seq2Seq と Transformer のエンコーダ及びデコーダを互いに交換したモデルである [15]．根石らは Seq2Seq，Transformer，エンコーダに LSTM を用いデコーダに Transformer を用いたスワップモデル，エンコーダに Transformer を用いデコーダに LSTM を用いたスワップモデル，合計 4 種のモデルで機械翻訳タスクを行い性能を比較した．その結果，Transformer のエンコーダとデコーダの影響力は大きく，2 種類のスワップモデルの翻訳結果は Seq2Seq よりも Transformer に類似することなどを示した．

エンコーダに LSTM を，デコーダに Transformer を用いたスワップモデルでは，全時刻の LSTM の出力をデコーダに入力する．これは Seq2Seq における Attention への入力と同じものである．

エンコーダに Transformer を，デコーダに LSTM を用いたスワップモデルでは，エンコーダの出力はデコーダの Attention へ入力される．このとき，デコーダの LSTM の初期状態を初期化するベクトルが存在しないため，0 で初期化する．

2.2 事前学習

事前学習とは，学習に使用する本来のデータとは別の大規模なデータを用いて事前にモデルを学習させる手法である [11, 26]．事前学習を行った後に本来のデータでもう一度学習することをファインチューニング（fine tuning）と呼ぶ．ファインチューニングを行う際モデルの重みはすべて更新されるが，一部の重みだけを更新する手法は転移学習と呼ばれる．本研究ではファインチューニングを用いる．

モデルの重みは通常何らかの方法で初期化されるが，ファインチューニングでは事前学習で獲得した重みを初期値として利用することにより，モデルの精度を向上させることができる．あるタスクのデータ数が少ない場合，似たタスクの大規模なデータを事前学習用のデータとして利用できるが，プログラム合成ではプログラムと自然言語の文などがペアとなったデータが必要なため用意が難しい．Xu らは Stack Overflow やプログラミング言語のドキュメントから集めたデータを用いて事前学習を行う手法を提案した [27]．

2.3 プログラム合成モデル

プログラム合成とは，入力された高レベルの仕様（例：自然言語の文やテストケース，画像，それらを組み合わせたものなど [3,4,28–31]）から自動的にプログラムを生成する技術である．

自然言語の文を入力とするプログラム合成には，生成する対象として SQL など特定の言語のみを生成するモデル [14,32,33] や，Python などの汎用言語を含む複数の言語を生成するモデルがある [5–7,16,17,28,29,34]．

以降，複数の言語を生成できるモデルの中から本研究で使用した 2 つのモデルについて述べる．

2.3.1 SEQ2TREE

SEQ2TREE は Attention を用いた Seq2Seq を基礎として，生成する言語の木構造を利用できるように改良されたモデルである [16]．入力された自然言語の文を木構造のクエリへ変換する意味解析（Semantic Parsing）タスクにおいて Seq2Seq を上回る性能となった．SEQ2TREE ではエンコーダへの入力文の順序を反転させて入力する手法 [8] を採用している．

SEQ2TREE のモデルについて説明する．SEQ2TREE のエンコーダは 1 層の通常の LSTM を使用しているが，デコーダは 1 層の特殊な LSTM を使用している．通常の LSTM は単語列を順に処理するが，SEQ2TREE のデコーダの LSTM は単語列の木構造を表すため複数回に分けて階層的に出力を生成する．SEQ2TREE のデコーダの LSTM の概要図を図 2.6 に示す．図の左上の灰色の LSTM はエンコーダの LSTM の最終時刻のものを表している．白色の LSTM の上の単語はその時刻で生成した出力である．出力のうち，<n> は木の非終端記号を，</s> は単語列の終了を表すトークンである．また，破線は parent-feeding を表しており，親の情報を子に与える役割を持つ．parent-feeding は LSTM への入力ベクトルに親の隠れ状態を結合することで実現されている．

図 2.6 の例では次のクエリを生成している．なお，この図では括弧は省略されている．

lambda $0 e (and (>(departure time $0) 1600:ti) (from $0 dallas:ci))

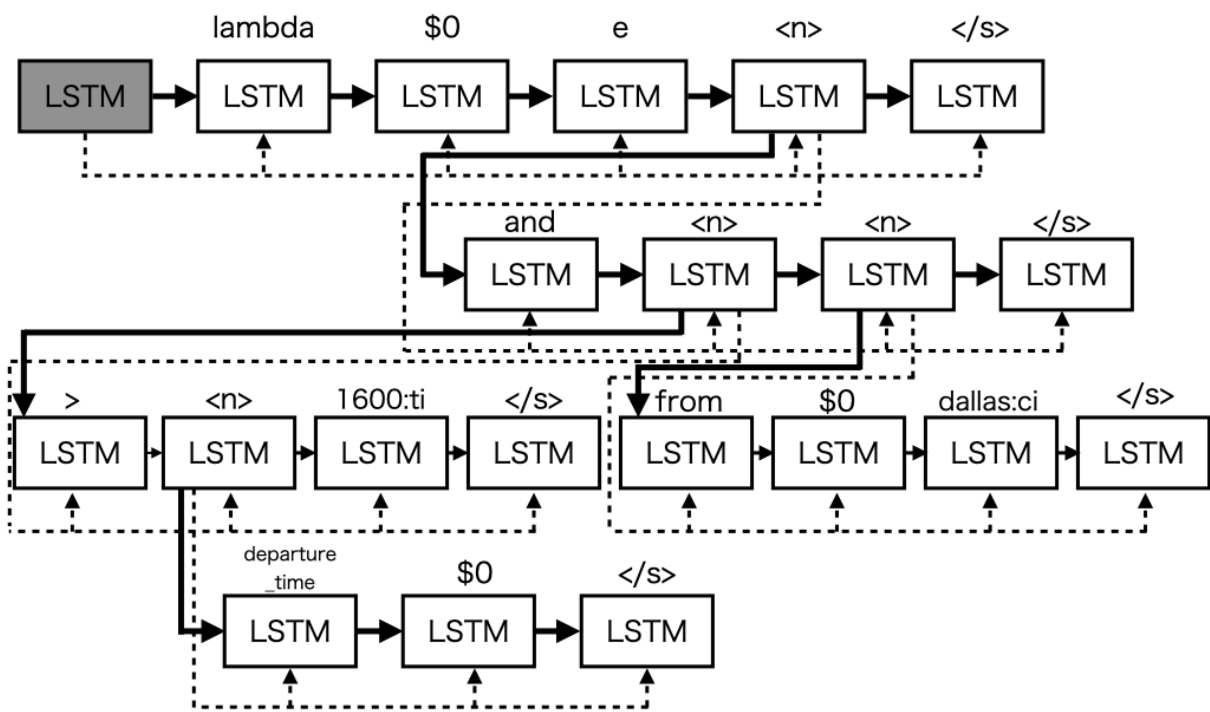

図 2.6　**SEQ2TREE のデコーダの LSTM の概要図**

このクエリは括弧を用いることで木構造を表現している．デコーダはクエリを一度にすべて生成するのではなく，括弧の中身（子）を推測するのは後回しにしておき，代わりに非終端記号を生成する．その階層の生成が終わると，非終端記号の情報を用いて次の階層の生成を始める．これを非終端記号がなくなるまで繰り返し，最後に結合することでクエリを完成させる．

SEQ2TREE は生成対象の言語特有の情報を使用しないため，括弧など木構造を表すものがあれば様々な言語を生成できる．

2.3.2 TRANX

TRANX はプログラミング言語の Abstract Syntax Description Language（ASDL）(注 1) を利用してプログラムを合成する，Seq2Seq を基礎としたモデルである [17]．TRANX は複数のプログラム合成タスクにおいて最良またはそれに近い結果となった．TRANX が自然言語の文からプログラムを生成するまでの概要図を図 2.7 に示す．この図の実線の部分に機械学習を用いる．

TRANX は入力された自然言語の文から図 2.7 に示されるような Actions を生成することを学習する．Actions は ASDL の抽象構文木（AST）を作成するための命令であり，順番に Actions を実行することで ASDL の AST を拡張する仕組みである．また，Actions は生成するプログラミング言語の ASDL 文法ファイルに基づいて定義されるため，プログラミング言語ごとに Actions の内容は異なる．TRANX は Actions を生成した後，生成した Actions から ASDL の AST を作成し，さらにプログラミング言語の AST へと変換する．変換されたプログラミング言語の AST は，Python3 の場合は ast ライブラリ (注 2) を用いることでソースコードに変換できる．また，ソースコードから Actions までの逆変換も可能である．

TRANX は Attention を用いた Seq2Seq を基礎としており，エンコーダには 1 層の通常の双方向 LSTM を用いるが，デコーダは 1 層の特殊な LSTM を用いる．デコーダの LSTM への入力に入力トークンの分散表現だけでなく，前の時刻の Attention の出力である 2.1.1(3) 節で説明した *context_vector* と親の情報を与える parent-feeding

(注 1): Python3 の ASDL はここから参照できる．
`https://github.com/python/cpython/blob/3.9/Parser/Python.asdl`
(注 2): `https://docs.python.org/ja/3/library/ast.html`

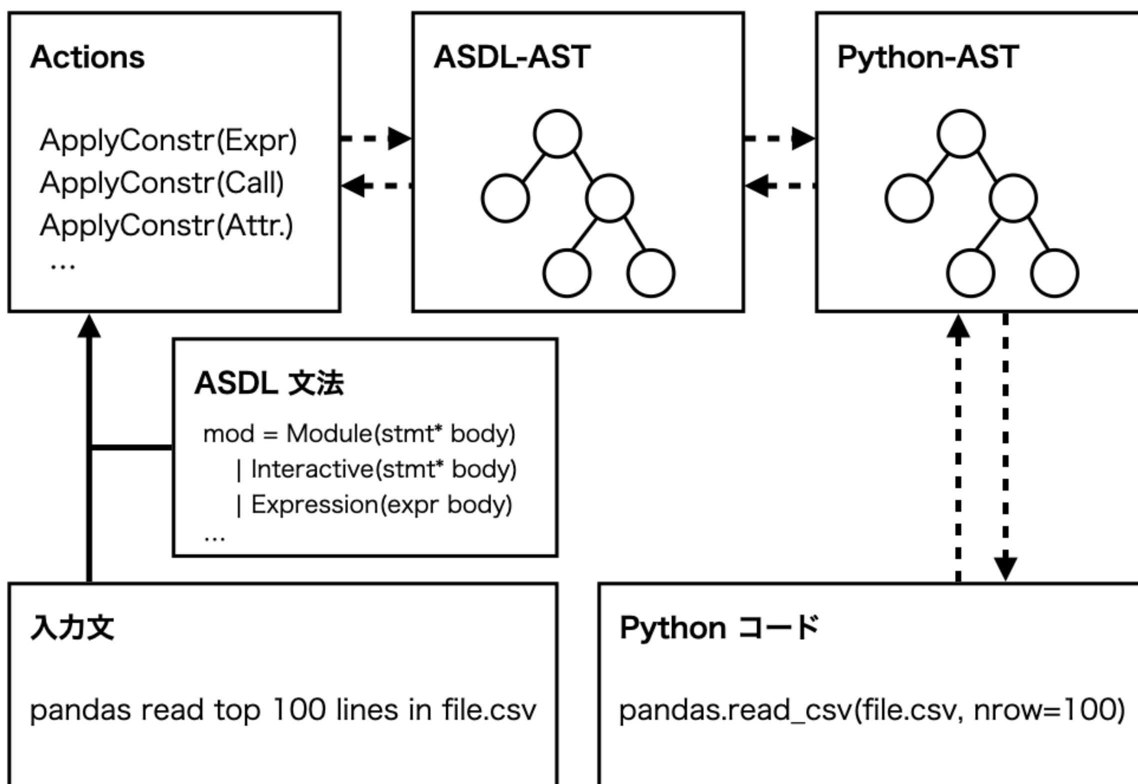

図 2.7　TRANX の概要図

を結合する．Actions 自体は木構造を持たないが，Actions から生成された ASDL の AST を利用して親の情報を取得できるため，SEQ2TREE のように LSTM で木構造を表現しない．また，TRANX ではプログラムを生成する際に入力された自然言語の文から変数名などの単語をコピーする機構である pointer network [35] を取り入れている．

TRANX が使用する言語情報は ASDL に集約されているため，ASDL 文法ファイルを用意することができれば様々な言語を生成できる．

2.4 先行研究の問題点

SEQ2TREE や TRANX のようなプログラム合成の先行研究には問題点が 2 つある．

1 つは，LSTM を使用していることである．機械翻訳において，Transformer は LSTM や畳み込みニューラルネットワーク（CNN）を用いたモデルより高い精度で翻訳しており [10]，また，Transformer を基礎としたモデルは翻訳の精度をさらに向上させている [12,36]．プログラム合成に Transformer を採用している例はとしては，通常の Transformer を Seq2Seq 等と比較した研究 [14] や，位置エンコーディングを改良した研究 [13] がある．しかしその数は少なく，未だプログラム合成では Transformer がデファクトスタンダードであるとは言い難い．

もう 1 つは，2.3 節で紹介した先行研究のほとんどはデコーダのみを改良していることである．エンコーダを含むモデル全体を改良している例としては，CNN を用いた Encoder-Decoder モデルの研究 [7] が挙げられるが，本研究ではプログラム合成で広く用いられている LSTM を用いたモデルを対象とする．

デコーダはエンコーダの出力を利用してプログラムを生成するため，エンコーダの性能も最終的なモデル全体の性能に影響すると考えられる．2.1.3 節で紹介した通り，エンコーダやデコーダのアーキテクチャを置き換えることによって機械翻訳モデルの性能が変化した例もある．SEQ2TREE や TRANX はデコーダの LSTM の隠れ状態を利用してプログラムの構造を捉えているが，Transformer には隠れ状態がないため置き換えることができない．しかし，エンコーダは通常の LSTM を使用しているため改良の余地があるが，我々が知る限り，エンコーダを改良したプログラム合成モデルは存在しない．

3. 研究の目的

2.4 節で述べたように，プログラム合成の先行研究ではエンコーダに対する改良を行っていないため，エンコーダの改良がモデル全体の性能に影響するか調査する必要がある．そこで，先行研究で用いられている LSTM エンコーダを Transformer エンコーダに置き換えることでエンコーダの改良を試みる．本研究の目的は，Encoder-Decoder モデルに広く用いられている LSTM と Transformer に着目し，プログラム合成において，これらを用いたエンコーダのアーキテクチャの差異によるモデルの性能への影響を明らかにすることである．

そのため，本研究では以下に示す研究設問（RQ）について検証を行う．

RQ1 LSTM エンコーダを用いたプログラム合成モデルと Transformer エンコーダを用いたプログラム合成モデルの性能にはどのような差異があるか

RQ2 デコーダの LSTM を 0 で初期化することはプログラム合成モデルの性能に影響するか

RQ3 事前学習を行った場合，プログラム合成モデルの性能はエンコーダの差異によってどのように変化するか

それぞれの研究設問を設定したモチベーションは以下の通りである．

RQ1 は，どちらのエンコーダがプログラム合成に適しているかを調査するため，それぞれのエンコーダを用いたモデルでプログラム合成を行う実験をする．

RQ2 は，2.1.3 節で述べたように，Transformer エンコーダを用いたときにデコーダの LSTM の初期状態を 0 で初期化することが RQ1 の結果に影響しているか調査するため，デコーダの LSTM を 0 で初期化する LSTM エンコーダを用いたモデルを追加して実験する．

RQ3 は，事前学習の有無による性能の変化量はエンコーダのアーキテクチャの差異によって異なるか調査するため，それぞれのエンコーダを用いたモデルで事前学習とファインチューニングを行う実験をする．

4. 実験方法

本章では，3 章で説明した RQ に回答するために行った実験に使用したプログラム合成モデルやデータセットなどについて述べる．

4.1 実験に使用したモデル

本研究では，2.3.1 節で説明した SEQ2TREE と 2.3.2 節で説明した TRANX の 2 つのプログラム合成モデルを対象に実験を行った．これらのモデルを選んだ理由は，2 つのモデル間の手法が大きく異なっていることと，実装が公開されていることである．

RQ1 に回答するため，本研究では LSTM エンコーダを用いた SEQ2TREE と TRANX に加え，Transformer エンコーダを用いた SEQ2TREE と TRANX の合計 4 種のプログラム合成モデルを作成した．さらに，RQ2 の実験をするため，デコーダの LSTM を 0 で初期化する LSTM エンコーダを用いた SEQ2TREE と TRANX も作成した．

実装は Pytorch[注 1] を用いて，公開されている実装を基に Transformer を追加する形で行った．Transformer の実装には Pytorch の TransformerEncoderLayer クラスを利用した．また，位置エンコーディングの実装は Pytorch のチュートリアル[注 2] を参考にした．

各モデルに用いるハイパーパラメータなどについて説明する．最適化手法は元の実装に従ったためモデルごとに異なり，SEQ2TREE では RMSProp [37] を，TRANX では Adam [38] を使用した．RMSProp の学習率は 0.007 を，Adam の学習率は 0.001 を用いた．埋め込み層の次元 $embedding$ と LSTM の次元 $hidden$，Transformer の次元 d_{model} は {64, 128, 256, 512} を，埋め込み層と Attention のドロップアウト率 dr と Transformer のドロップアウト率 dr_{trans} は {0.1, 0.3} を，バッチサイズは 64 を用いた．また，Transformer の Multi-Head Attention の head 数 n_{head} は {2, 4, 8} を，Feed Forward 層の次元 d_{ff} は {128, 256, 512, 1024} を，Transformer エンコーダの層の数 n_{layer} は {1, 2, 4, 6} を用いた．これらのハイパーパラメータの中からいくつかの組み合わせで実験を行った．

(注 1): `https://pytorch.org/`
(注 2): `https://pytorch.org/tutorials/beginner/transformer_tutorial.html`

4.2 評価指標

本研究では，自然言語処理で用いられる代表的な評価指標の Accuracy と BLEU スコアを用いた．それぞれの評価指標について説明する．

4.2.1 Accuracy

Accuracy とは精度や正解率のことであり，全データのうちモデルが正しく予測できたデータの割合を表す，分類問題などで用いられている指標である．Accuracy は以下の式で算出される．

$$Accuracy\ (\%) = \frac{\text{正しく予測できたデータ数}}{\text{全データ数}} * 100 \tag{4.1}$$

4.2.2 BLEU スコア

Bilingual Evaluation Understudy（BLEU）スコアとは，正解文と生成文の類以度を測る指標であり，機械翻訳など自然言語処理の分野で広く用いられている指標である [20]．正解文と生成文の単語列をそれぞれ R，C とすると，2 つの文の一致度 p_n は次の式で表される．

$$p_n = \frac{R \text{ と } C \text{ で一致した n-gram 数}}{C \text{ の全 n-gram 数}} \tag{4.2}$$

ここで，n-gram とは連続する n 個の単語のまとまりのことである．例えば，”I play piano”という 3 つの単語からなる文があるとき，2-gram（n=2）は”I play” と”play piano”の 2 つとなる．また，n-gram 数を数えるときに同じ n-gram が出現する場合があるが，n-gram は 1 度だけ使用し，同じものは無視される．

C が R よりも短い場合に p_n が高くなりすぎることを防ぐため，BLEU スコアを計算する際に課すペナルティ係数 Brevity Penalty（BP）を用いる．R の単語数を r，C の単語数を c とすると，BP は以下のように算出される．

$$BP = \begin{cases} 1 & (c > r) \\ e^{1-\frac{r}{c}} & (c \leq r) \end{cases} \tag{4.3}$$

p_n と BP を用いて，BLEU スコアは次のように算出される．

$$\text{BLEU スコア } (\%) = BP * \exp(\frac{1}{N}\sum_{n=1}^{N} \log p_n) * 100 \tag{4.4}$$

4.4 式の N は n-gram の n の最大値を表す．一般的に $N = 4$ が使用されており，本研究でもこれを使用する．

4.3 データセット

4.3.1 使用したデータセット

本研究では以下の 3 つのデータセットを用いる．また，それらのデータセットに含まれているデータの例を表 4.1 に示す．この表の各行の上段が自然言語の文で，下段がクエリまたはソースコードを示す．

ATIS ATIS は 5,373 個のクエリとそれに対応する自然言語の質問文からなる航空券予約システムのデータセットである [16]．クエリはラムダ計算式で表されている．4,434 個の学習データ，491 個の検証データ，448 個のテストデータで構成される．評価指標には主に Accuracy が用いられている．

DJANGO DJANGO は Django フレームワーク(注 3)から抽出した 18,805 行の Python2 のソースコードと各行に注釈付られたコメントのペアのデータセットである [18]．1 行のソースコードとそれに対応するコメントを 1 つのデータとし，16,000 個の学習データ，1,000 個の検証データ，1,805 個のテストデータで構成される．評価指標には主に Accuracy が用いられている．

CoNaLa CoNaLa は Stack Overflow からマイニングした 2,878 個の Python3 のソースコードとそれらに対応する手動で追加されたコメントのデータセットである [19]．2,178 個の学習データ，200 個の検証データ，500 個のテストデータで構成される．評価指標には BLEU スコアが用いられている．また，CoNaLa は自動で作成された 60 万個のデータも提供している．

上記のデータセットに含まれている学習データに対して，自然言語の文の長さや重複している同じデータの数を調査したものを表 4.2 に示す．

RQ3 の実験で行う事前学習に用いるデータセットについて説明する．本研究で行う事前学習は Xu ら [27] が提供しているデータセットを用いて行った．このデータセットは，CoNaLa データセットが提供している 60 万個のデータのうち，品質を考

(注 3): https://www.djangoproject.com/

表 4.1　データセットの例

データセット	データの例
ATIS	list the al0 flight arriv in ci0 `(lambda $0 e (and (flight $0) (airline $0 al0) (to $0 ci0)))`
DJANGO	return path_info.decode method return value called with UTF_8 as the argument. `return path_info.decode(UTF_8)`
CoNaLa	check if directory ‘directory ‘ exists and create it if necessary `if (not os.path.exists(directory)):\n    os.makedirs(directory)`

表 4.2　各データセットの自然言語の文の単語数と重複データの数

	ATIS	DJANGO	CoNaLa
データ数（全て）	5,373	18,805	2,878
学習データ数	4,434	16,000	2,178
学習データ中の自然言語の文の単語数（平均値）	10.6	14.3	10.3
学習データ中の自然言語の文の単語数（中央値）	10	12	10
学習データ中の重複している自然言語の文の数	1,125	4,162	88
学習データ中の重複している自然言語の文の割合	25%	26%	3%

慮した10万個のデータとPython3.7.5のAPIドキュメントから収集したデータを基に作成されており，そのデータ数の合計は114,327個である．この事前学習用データはCoNaLaデータセットでファインチューニングするために作成されたものであるため，RQ3はCoNaLaデータセットのみで実験した．

4.3.2 前処理

本節ではSEQ2TREEとTRANXそれぞれのモデルで用いるデータセットに対して行った前処理について説明する．

まず，SEQ2TREEで用いるデータセットに対して行った前処理について説明する．2.3.1節で説明したように，SEQ2TREEは括弧などを用いて木構造を表現している．表4.1からわかるように，ATISデータセットのクエリは括弧で木構造を表現する言語であるため，前処理を行う必要がない．しかし，DJANGOやCoNaLaデータセットの言語はPythonであるため前処理を行う必要がある．そこで本研究では，DJANGOやCoNaLaデータセットに含まれているPythonのソースコードに対して，TRANXで用いられているASDL文法ファイルを利用して前処理を行い，自然言語の文とASDL-AST文のペアのデータを作成した．2.3.2節で述べた通り，図2.7の一連の処理は一部逆変換できる．従って，本研究ではDJANGOとCoNaLaデータセットのPythonのソースコードをASDLのASTへと逆変換し，さらにTRANXで実装されているASDLのASTを文字列に変換する機能を用いて文字列へと変換した．便宜のため，この変換した文字列をASDL-AST文と呼ぶ．例えば，表4.1の"`return path_info.decode(UTF_8)`"のASDL-AST文は

```
(Return (expr?-value (Call (expr-func (Attribute (expr-value (Name
(identifier-id pathinfo) (expr_context-ctx (Load)))) (identifier-attr decode)))
(expr*-args (Name (identifier-id UTF_8) (expr_context-ctx (Load))))
(keyword*-keywords))))
```

となる．なお，紙面の都合上改行しているが実際は1行の文である．

TRANXでは元の実装通り，3つのデータセットのクエリやソースコードをActionsへ逆変換する前処理を行い，自然言語の文とActionsのペアのデータを作成した．

4.3.3 評価方法

SEQ2TREE で ATIS データセットを用いて学習する場合，モデルはクエリを生成するため，生成したクエリと正解データのクエリを用いて Accuracy で評価した．しかし，DJANGO と CoNaLa データセットを用いて学習する場合，モデルは ASDL-AST 文を生成するが，ASDL-AST 文から AST への逆変換はできないため，生成した ASDL-AST 文と正解データの ASDL-AST 文を用いて Accuracy または BLEU スコアで評価した．

TRANX で本研究で使用した 3 つのデータセットを用いて学習する場合，モデルはそれぞれの Actions を生成するが，生成した Actions を図 2.7 のようにクエリやソースコードへ変換し，それらをトークナイズしたものとトークナイズした正解データを用いて Accuracy または BLEU スコアで評価した．

用いる評価指標によって実験結果が変わる可能性があるため，RQ1 の実験では 3 つのデータセットそれぞれに対して Accuracy と BLEU スコアの両方で評価したが，RQ2 と RQ3 の実験では時間的な制約から 4.3.1 節で述べた評価指標のみで評価した．

あるソースコードをトークナイズして BLEU スコアを計算した結果と ASDL-AST 文に変換して BLEU スコアを計算した結果は異なることに注意が必要である．例えば，"`return pathinfo.decode(UTF_8)`" と "`return pathinfo.decode(UTF_16)`" の BLEU スコアを計算すると，トークナイズして評価した場合は 64.3%となり，ASDL-AST 文に変換して評価した場合は 95.2%となる．よって，SEQ2TREE と TRANX の間での BLEU スコアの比較には意味がない．また，本研究の目的は異なるエンコーダを用いた場合にプログラム合成モデルの性能がどのように変化するか調査することである．そのため，本研究では 4 種類のモデルのうちどれが最も優れているか比較するのではなく，2 種類のエンコーダを用いた SEQ2TREE 同士での比較と 2 種類のエンコーダを用いた TRANX 同士での比較に分けた．

4.4 学習環境

本研究では 4 つの計算機を用いて実験した．以下にそれぞれの計算機の構成を示す．OS はすべて Ubuntu Server 18.04 LTS である．

- Intel®Xeon®CPU E5-2630 v4 @ 2.20GHz 2 基，GeForce®GTX 1080 Ti 11GB 1

基，RAM 64GB

- Intel®Core™i7-6700K CPU @ 4.00GHz 1 基，GeForce®GTX 1080 8GB 1 基，RAM 32GB
- Intel®Xeon®CPU E5-1620 v3 @ 3.50GHz 1 基，GeForce®GTX TITAN X 12GB 1 基，RAM 64GB
- Intel®Xeon®Silver 4210 CPU @ 2.20GHz 1 基，GeForce®RTX 2080 Ti 11GB 1 基，RAM 128GB

5. 実験結果

5.1 定量的結果

まず最初に RQ1 の実験結果を示す．本実験で作成した 4 つのモデルを 4.3.1 節で説明した 3 つのデータセットを用いて実験した結果を表 5.1 に示す．この表の左端の列はモデルの名前を表しており，”-LSTM”が付いているものは LSTM エンコーダを用いたモデル，”-Transformer”が付いているものは Transformer エンコーダを用いたモデルである．表 5.1 からわかるように，SEQ2TREE では，”SEQ2TREE-LSTM”の Accuracy と BLEU スコアは”SEQ2TREE-Transformer”より高い結果となった．TRANX では，ATIS と DJANGO では”TRANX-LSTM”の Accuracy と BLEU スコアは”TRANX-Transformer”より高い結果となり，CoNaLa では評価指標によって異なる結果となった．

次に RQ2 の実験結果を示す．RQ1 の 4 つのモデルに 4.1 節で説明したデコーダの LSTM を 0 で初期化する 2 つのモデルを追加した 6 つのモデルを用いて実験した結果を表 5.2 に示す．この表の”-LSTM0”が付いているモデルがデコーダの LSTM を 0 で初期化したモデルである．比較のため，表 5.1 の結果も一部再掲している．表 5.2 からわかるように，SEQ2TREE では，”SEQ2TREE-LSTM0”の Accuracy と BLEU スコアは”SEQ2TREE-LSTM”より低い結果となった．TRANX では，”TRANX-LSTM0”の DJANGO の Accuracy は”TRANX-LSTM”より低い結果となったが，ATIS の Accuracy と CoNaLa の BLEU スコアは”TRANX-LSTM”より高い結果となった．

最後に RQ3 の実験結果を示す．事前学習を行い CoNaLa でファインチューニングした場合の実験結果を表 5.3 に示す．この表からわかるように，SEQ2TREE では事前学習を行うと性能が低下する結果となった．一方，TRANX では事前学習を行うと性能が向上する結果となった．また，事前学習を行うことで，”SEQ2TREE-LSTM”と”SEQ2TREE-Transformer”の結果の差は 2.4%から 1.3%へと小さくなり，”TRANX-LSTM”と”TRANX-Transformer”の結果の差も 1.1%から 0.3%へと小さくなった．

これらの結果を算出したときのモデルのハイパーパラメータを表 5.4，5.5 に示す．

表 **5.1　3** つのデータセットの実験結果

	ATIS		DJANGO		CoNaLa	
	Accuracy	BLEU	Accuracy	BLEU	Accuracy	BLEU
SEQ2TREE-LSTM	76.3	90.7	20.4	70.0	0.0	58.6
SEQ2TREE-Transformer	57.1	83.8	15.7	49.9	0.0	56.2
上記 2 つのモデルの結果の差	19.2	6.9	4.7	20.1	0.0	2.4
TRANX-LSTM	87.1	95.7	77.9	64.5	2.9	28.0
TRANX-Transformer	68.8	92.0	42.8	48.7	2.4	29.1
上記 2 つのモデルの結果の差	18.3	3.7	35.1	15.8	0.5	1.1

表 **5.2**　デコーダの **LSTM** の初期状態を **0** で初期化した場合の実験結果

	ATIS(Accuracy)	DJANGO(Accuracy)	CoNaLa(BLEU)
SEQ2TREE-LSTM	76.3	20.4	58.6
SEQ2TREE-LSTM0	70.8	20.3	55.2
上記 2 つのモデルの結果の差	5.5	0.1	3.4
TRANX-LSTM	87.1	77.9	28.0
TRANX-LSTM0	88.4	75.3	28.2
上記 2 つのモデルの結果の差	1.3	2.6	0.2

表 **5.3**　事前学習の有無による実験結果

	CoNaLa(BLEU)	
	事前学習あり	事前学習なし
SEQ2TREE-LSTM	32.9	58.6
SEQ2TREE-Transformer	34.2	56.2
上記 2 つのモデルの結果の差	1.3	2.4
TRANX-LSTM	29.6	28.0
TRANX-Transformer	29.9	29.1
上記 2 つのモデルの結果の差	0.3	1.1

表 **5.4 LSTM** エンコーダのモデルのパラメータ

		embedding	*hidden*	*dr*
SEQ2TREE-LSTM	ATIS（Accuracy）	256	256	0.3
	ATIS（BLEU）	256	256	0.3
	DJANGO（Accuracy）	256	256	0.3
	DJANGO（BLEU）	256	256	0.3
	CoNaLa（Accuracy）	256	256	0.3
	CoNaLa（BLEU）	256	256	0.3
	CoNaLa（事前学習）	256	256	0.3
SEQ2TREE-LSTM0	ATIS（Accuracy）	256	256	0.3
	DJANGO（Accuracy）	256	256	0.3
	CoNaLa（BLEU）	256	256	0.3
TRANX-LSTM	ATIS（Accuracy）	256	256	0.3
	ATIS（BLEU）	256	256	0.3
	DJANGO（Accuracy）	256	256	0.3
	DJANGO（BLEU）	256	256	0.3
	CoNaLa（Accuracy）	256	256	0.3
	CoNaLa（BLEU）	128	128	0.3
	CoNaLa（事前学習）	512	512	0.3
TRANX-LSTM0	ATIS（Accuracy）	256	256	0.3
	DJANGO（Accuracy）	256	256	0.3
	CoNaLa（BLEU）	128	128	0.3

表 5.5　Transformer エンコーダのモデルのパラメータ

		d_{model}	d_{ff}	n_{head}	n_{layer}	dr	dr_{trans}
SEQ2TREE-Transformer	ATIS（Accuracy）	256	1024	8	1	0.3	0.1
	ATIS（BLEU）	256	1024	8	1	0.3	0.1
	DJANGO（Accuracy）	64	256	4	1	0.3	0.1
	DJANGO（BLEU）	64	256	4	1	0.3	0.1
	CoNaLa（Accuracy）	256	1024	2	2	0.3	0.1
	CoNaLa（BLEU）	256	1024	2	2	0.3	0.1
	CoNaLa（事前学習）	256	1024	8	1	0.3	0.1
TRANX-Transformer	ATIS（Accuracy）	128	128	8	2	0.3	0.1
	ATIS（BLEU）	128	256	8	2	0.3	0.1
	DJANGO（Accuracy）	128	256	4	4	0.3	0.1
	DJANGO（BLEU）	128	256	8	2	0.3	0.1
	CoNaLa（Accuracy）	256	512	8	2	0.3	0.1
	CoNaLa（BLEU）	256	512	8	1	0.3	0.1
	CoNaLa（事前学習）	256	1024	8	1	0.3	0.1

5.2 研究設問への回答

5.2.1 RQ1: LSTM エンコーダを用いたプログラム合成モデルと Transformer エンコーダを用いたプログラム合成モデルの性能にはどのような差異があるか

表 5.1 から，エンコーダの優劣はデータセットに依存する傾向があり，ATIS と DJANGO ではどちらのモデルでも LSTM エンコーダの方が良い性能となることがわかった．

5.2.2 RQ2: デコーダの LSTM を 0 で初期化することはプログラム合成モデルの性能に影響するか

表 5.2 から，それぞれのモデルの”-LSTM”と”-LSTM0”を比較すると，Accuracy では 5.5%，2.6%，1.3%，0.1%の差が，BLEU スコアでは 3.4%，0.2%の差が確認され，デコーダの LSTM を 0 で初期化することはモデルに無視できない影響を及ぼすことがわかった．また，その影響は必ずしも性能の低下を招くものではないことがわかった．

5.2.3 RQ3: 事前学習を行った場合，プログラム合成モデルの性能はエンコーダの差異によってどのように変化するか

表 5.3 から，事前学習を行うことで LSTM エンコーダのモデルと Transformer エンコーダのモデル間の性能の差が小さくなることがわかった．

6. 考察

6.1 研究設問に対する考察

6.1.1 RQ1: LSTM エンコーダを用いたプログラム合成モデルと Transformer エンコーダを用いたプログラム合成モデルの性能にはどのような差異があるか

表 5.1 において，全体的に Transformer エンコーダのモデルの方が劣る結果となった理由として，今回の実験で Transformer の性能を十分に引き出せなかったことが考えられる．Transformer の提案論文では $n_{layer} = 6$ が用いられており，Transformer を採用している多くの研究でも同様またはそれ以上のパラメータが用いられている．しかし本研究では表 5.5 に示すように，最も良い結果となったモデルの n_{layer} は 2 以下のものが多く，$n_{layer} = 6$ で実験した場合はほとんどのモデルでうまく学習できなかった．その理由としては，データセットの数や，デコーダの LSTM が 1 層だけで構成されており複雑度が不足していたことが考えられる．

また，プログラム合成では Transformer エンコーダと LSTM デコーダの相性が良くない可能性も考えられる．プログラム合成と機械翻訳のデータセットでは入力文の語彙や文の長さなど異なる点があり，適するアーキテクチャが機械翻訳と異なる可能性がある．

LSTM エンコーダの方が優れた結果が多かったことから，今後プログラム合成で Transformer を基礎としたモデルを扱う場合にエンコーダを LSTM にすることで，プログラム合成モデルの性能が向上する可能性があると考える．

6.1.2 RQ2: デコーダの LSTM を 0 で初期化することはプログラム合成モデルの性能に影響するか

表 5.2 に示すように，”TRANX-LSTM”と”TRANX-LSTM0”の結果の差は全体的に”SEQ2TREE-LSTM”と”SEQ2TREE-LSTM0”の結果の差より小さい．この理由として，SEQ2TREE はクエリやソースコードを直接生成するが，TRANX は Actions を生成する文法を利用した手法を用いていることが考えられる．そのため，文法を利用したプログラム合成手法を用いることで，Transformer エンコーダを用いる場合にデコーダの LSTM を 0 で初期化することによる影響を軽減できると考えられる．

TRANX ではエンコーダに双方向 LSTM が用いられている．双方向 LSTM を用いた場合，デコーダの LSTM を初期化するベクトルには入力文の最初と最後の単語の情報が多く含まれることになる．”-LSTM0”の方が優れた結果となったときの理由として，この最初と最後の単語情報がノイズとなっていた可能性がある．プログラム合成の入力文には命令文が多く，その場合最初の単語の情報は有用であるが，最後の単語は最初の単語に比べて重要度が低い可能性がある．一方，SEQ2TREE では単方向の LSTM が用いられているが，2.3.1 節で説明したようにエンコーダの入力文を反転させて入力している．そのため，SEQ2TREE のエンコーダの LSTM の最終時刻の出力には入力文の最初の単語の情報が多く含まれる．”SEQ2TREE-LSTM0”ではこの最初の単語情報を利用できなくなったため”SEQ2TREE-LSTM”より性能が低下したと考える．

本研究では Transformer エンコーダを用いる場合はデコーダの LSTM を 0 で初期化しており，この初期化方法はモデルの性能に影響を与えていることがわかった．デコーダの LSTM をより良い方法で初期化することで，Transformer エンコーダを用いたモデルの性能も変化する可能性があると考える．

6.1.3 RQ3: 事前学習を行った場合，プログラム合成モデルの性能はエンコーダの差異によってどのように変化するか

表 5.3 に示すように，事前学習を行うことで LSTM エンコーダのモデルと Transformer エンコーダのモデルの性能の差が小さくなった．機械翻訳では，学習データの数を増やすと Seq2Seq と Transformer，2.1.3 節で説明した 2 つのスワップモデルの合計 4 つのモデルの性能の差が小さくなったことが根石ら [15] により報告されている．これらのことから，本研究では約 11 万個のデータで事前学習を行ったが，より大きなデータで事前学習を行うことで LSTM エンコーダのモデルと Transformer エンコーダのモデルの性能の差はさらに小さくなる可能性があると考えられる．最終的なモデルの性能の差が小さい場合，性能ではなく学習時間やメモリの使用量などを重視してエンコーダを設計できる．

SEQ2TREE では事前学習を行うと，事前学習を行わなかった場合よりも性能が低下した．事前学習を終えた段階でのテストデータの結果を確認すると，BLEU スコアは 40%程度あり，ファインチューニングがうまく行えていないことがわかった．事

前学習用データの品質は手動で集められたデータより劣るため，SEQ2TREE には適さなかったと考える．

6.2　妥当性への脅威

6.2.1　構成概念妥当性

構成概念妥当性では実験における評価手法の妥当性について議論する．

本実験の評価指標には Accuracy と BLEU スコアを用いた．これらの指標は生成した文が正解の文と同じまたは似ているかを測る指標である．しかし，プログラムは同じ動作をするものであっても複数の書き方がある．例えば Python であれば，”`x = x + 1`”と”`x += 1`”は同じ動作をするが，Accuracy や BLEU スコアではこれらの式を同一とみなす評価ができない．そのため，プログラムの実行結果を評価するなど，プログラムの意味を評価する新たな評価指標の導入が必要である．

また，RQ1 の実験では 3 つのデータセットに対して Accuracy と BLEU スコアを用いて評価したが，RQ2 と RQ3 の実験では時間的な制約から 4.3.1 節で説明した評価指標のみを用いて評価した．表 5.1 から，ATIS と DJANGO データセットに関しては片方の評価指標のみを用いても結果に大きな影響はないと考えられるが，CoNaLa データセットに関しては評価指標によりエンコーダの優劣が逆転した結果があった．また，1 つの評価指標で評価したことにより，RQ2 と RQ3 の比較のためのデータが少なくなった．そのため，RQ2 と RQ3 を 1 つの評価指標を用いて評価したことは妥当性への脅威となりうる．

6.2.2　外的妥当性

外的妥当性では実験結果の一般性についての妥当性を議論する．

本実験で使用したプログラム合成モデル SEQ2TREE と TRANX はどちらもあらゆる言語に対応できる汎用的なモデルである．プログラム合成には SQL など特定の言語のみに特化したモデルも存在しており，それらのモデルを用いた場合，本実験と異なる結果になる可能性がある．

6.2.3 内的妥当性

内的妥当性では本実験の方法の妥当性について議論する．

ATIS と DJANGO の評価指標に BLEU スコアを用いた実験と CoNaLa の評価指標に Accuracy を用いた実験，デコーダの LSTM を 0 で初期化する実験に関しては，時間的な制約から十分なパラメータ調整が行えていない．そのため，パラメータ調整を行うことで結果が変わる可能性があると考える．

また，本実験で用いたモデルは公開されている実装を基に著者が実装した．デバッグやテストを行いながら実装したが，不具合が残っている可能性は否定できない．

7. 結言

本研究ではLSTMを用いた2つのプログラム合成モデルのエンコーダをTransformerに置き換え，エンコーダの差異によりプログラム合成モデルの性能が変化するか比較実験を行った．単純な比較に加え，デコーダのLSTMを0で初期化することで条件を近くした比較や，事前学習の有無による比較も行った．実験対象のモデルには，木構造を意識して設計されたSEQ2TREEと，生成対象の言語の文法を利用したTRANXを採用した．実験の評価には異なる言語やドメインの3つのデータセットを使用し，評価指標にはAccuracyとBLEUスコアを用いた．

実験の結果，デコーダにLSTMを用いる場合はエンコーダにもLSTMを用いた方が良い結果となりやすく，その結果はデータセットに依存する傾向があることがわかった．TransformerエンコーダよりLSTMエンコーダを用いたモデルの方が良い性能となった理由の1つに，デコーダのLSTMの初期化方法が考えられる．また，事前学習を行うことでエンコーダの差異によるモデルの性能への影響が小さくなることがわかった．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学・人間科学系水野修教授，崔恩瀞助教に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻洪浚通先輩，近藤将成先輩，西浦生成先輩をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] M. Kim, L.D. Bergman, T.A. Lau, and D. Notkin, “An ethnographic study of copy and paste programming practices in OOPL,” In Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE), pp.83–92, 2004.

[2] J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, and S.R. Klemmer, “Two studies of opportunistic programming: interleaving web foraging, learning, and writing code,” In Proceedings of the 27th International Conference on Human Factors in Computing Systems (CHI), pp.1589–1598, 2009.

[3] M. Balog, A.L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, “Deepcoder: Learning to write programs,” In Proceedings of the 5th International Conference on Learning Representations (ICLR), 2017.

[4] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, “Machine learning-based prototyping of graphical user interfaces for mobile apps,” IEEE Trans. Software Eng., vol.46, no.2, pp.196–221, 2020.

[5] W. Ling, P. Blunsom, E. Grefenstette, K.M. Hermann, T. Kociský, F. Wang, and A.W. Senior, “Latent predictor networks for code generation,” In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL), pp.599–609, 2016.

[6] P. Yin and G. Neubig, “A syntactic neural model for general-purpose code generation,” In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL), pp.440–450, 2017.

[7] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, “A grammar-based structural CNN decoder for code generation,” In Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI), pp.7055–7062, 2019.

[8] I. Sutskever, O. Vinyals, and Q.V. Le, “Sequence to sequence learning with neural networks,” In Proceedings of 28th Conference on Neural Information Processing Systems (NeurIPS), vol.27, pp.3104–3112, 2014.

[9] T. Luong, H. Pham, and C.D. Manning, “Effective approaches to attention-based neural machine translation,” In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp.1412–1421, 2015.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L.u. Kaiser, and I. Polosukhin, “Attention is all you need,” In Proceedings of the 31st Conference on Neural Information Processing Systems (NeurIPS), vol.30, pp.5998–6008, 2017.

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” In Proceedings of the 17th Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), vol.1, pp.4171–4186, 2019.

[12] T.B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D.M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” In Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS), 2020.

[13] V.L. Shiv and C. Quirk, “Novel positional encodings to enable tree-based transformers,” In Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS), pp.12058–12068, 2019.

[14] D. Dalal and B.V. Galbraith, “Evaluating sequence-to-sequence learning models for if-then program synthesis,” CoRR, vol.abs/2002.03485, 2020.

[15] 根石将人，吉永直樹，“英日翻訳タスクにおけるスワップモデルを通したseq2seqとtransformerの比較,” 言語処理学会 第25回年次大会 発表論文集，pp.1117–1120，2019.

[16] L. Dong and M. Lapata, “Language to logical form with neural attention,” In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL), p.33–43, 2016.

[17] P. Yin and G. Neubig, "TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation," In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp.7–12, 2018.

[18] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (T)," In Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp.574–584, 2015.

[19] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," In Proceedings of the 15th International Conference on Mining Software Repositories (MSR), pp.476–486, 2018.

[20] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), pp.311–318, 2002.

[21] Z. Yan, D. Tang, N. Duan, S. Liu, W. Wang, D. Jiang, M. Zhou, and Z. Li, "Assertion-based QA with question-aware open information extraction," In Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), pp.6021–6028, 2018.

[22] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," In Proceedings of 7th International Conference on Learning Representations (ICLR), 2019.

[23] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," In Proceedings of the 3rd International Conference on Learning Representations (ICLR), 2015.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," In Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp.770–778, 2016.

[25] L.J. Ba, J.R. Kiros, and G.E. Hinton, "Layer normalization," CoRR,

vol.abs/1607.06450, 2016.

[26] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, "A survey on deep transfer learning," In Proceedings of 27th International Conference on Artificial Neural Networks (ICANN), vol.11141, pp.270–279, 2018.

[27] F.F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL), pp.6045–6052, 2020.

[28] M. Allamanis, D. Tarlow, A.D. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," In Proceedings of the 32nd International Conference on Machine Learning (ICML), vol.37, pp.2123–2132, 2015.

[29] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL), pp.1139–1149, 2017.

[30] I. Polosukhin and A. Skidanov, "Neural program search: Solving programming tasks from description and examples," In Proceedings of the 6th International Conference on Learning Representations (ICLR), 2018.

[31] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," In Proceedings of the 6th International Conference on Learning Representations (ICLR), 2018.

[32] C. Liu, X. Chen, E.C.R. Shin, M. Chen, and D.X. Song, "Latent attention for if-then program synthesis," In Proceedings of 30th Conference on Neural Information Processing Systems (NeurIPS), pp.4574–4582, 2016.

[33] T. Yu, Z. Li, Z. Zhang, R. Zhang, and D.R. Radev, "Typesql: Knowledge-based type-aware neural text-to-sql generation," In Proceedings of the 16th Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), pp.588–594, 2018.

[34] L. Dong and M. Lapata, "Coarse-to-fine decoding for neural semantic parsing," In

Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL), pp.731–742, 2018.

[35] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," In Proceedings of 29th Conference on Neural Information Processing Systems (NeurIPS), pp.2692–2700, 2015.

[36] J. Zhu, Y. Xia, L. Wu, D. He, T. Qin, W. Zhou, H. Li, and T. Liu, "Incorporating BERT into neural machine translation," In Proceedings of 8th International Conference on Learning Representations (ICLR), 2020.

[37] T. Tieleman and G. Hinton, Neural networks for machine learning lecture 6a overview of mini-batch gradient descent, （オンライン），入手先 ⟨http://www.cs.toronto.edu/ hinton/coursera/lecture6/lec6.pdf⟩（参照 2021-1-30）.

[38] D.P. Kingma and J. Ba, "Adam: A method for stochastic optimization," In Proceedings of 3rd International Conference on Learning Representations (ICLR), 2015.