タイトル	擬似コード生成のための学習 統計的機械翻訳 (T) から ソース コード 使用方法
著者名（敬称略）	小田 祐輔; 札場 広行; Neubig, Graham; 畑 秀明; Sakti, Sakriani; 戸田 知己; 中村 聡史
引用	ASE 2015 : 2015 30th IEEE/ACM International Conference on Automated Software Engineering, 9-13 Nov.2015, Lincoln, NE, USA
発行日	2015
リソースバージョン	作家
権利関係	
ドイ	10.1109/ase.2015.36
URL	http://hdl.handle.net/10061/12734

統計的機械翻訳を用いたソースコードからの擬似コード生成の学習

小田裕介、札場弘之、グラハム・ノイビッグ、秦秀明、
サクリアニ・サクティ、戸田智樹、中村聡
奈良先端科学技術大学院大学 情報科学研究科 630-0192
奈良県生駒市高山8916-5
{oda.yusuke.on9, fudaba.hiroyuki.ev6, neubig, hata, ssakti, tomoki, s-nakamura}@is.naist.jp

概要：自然言語で記述された擬似コードは，不慣れなプログラミング言語のソースコードの理解を助けることができる．しかし、ソースコードの大部分には対応する擬似コードが存在しない。なぜなら、擬似コードは冗長であり、作成に手間がかかるからである。もし、与えられたソースコードから擬似コードを自動的かつ瞬時に生成することができれば、人手をかけずにオンデマンドで擬似コードを生成することができるようになる。本論文では、ソースコードから擬似コードを自動生成する方法として、特に統計的機械翻訳（SMT）の枠組みを採用した方法を提案する。SMTはもともと2つの自然言語間の翻訳を目的として設計されており、ソースコードと擬似コードの組の関係を自動的に学習することができるため、より少ない人手で擬似コード生成器を作成することが可能である。実験では、SMTを用いてPythonの記述から英語または日本語の疑似コードを生成し、生成された疑似コードはほぼ正確であり、コードの理解を助けることを確認した。

キーワードアルゴリズム、教育、統計的アプローチ

I. イントロダクション

ソースコードを理解することは、すべてのプログラマにとって必要不可欠なスキルです。例えば、グループ作業を効率的に行うため、あるいはオープンソースのソフトウェアを統合・修正するために、他の人が書いたコードを読んで理解することが必要だからです。マクロレベルでは、エンジニアがプログラミングプロジェクトの全体構造を理解するためのさまざまなツールがある。例えば，DeLineらは，プログラミングの専門家が大規模な協調ソフトウェア工学プロジェクトを評価するためのツールを提案している[1]．また，Rahmanらは，ソースコードが期待通りに動作しない場合に，その修正方法を推奨するシステムを提案している[2].

もっと細かく言えば、ソースコードの振る舞いを詳細に理解しようとすると、通常はソースコードの各文を注意深く読み、各文が何を行っているかを理解する必要がある。もちろん、既存のソフトウェアのソースコードを徹底的に読み込んで理解することは、ベテランのプログラマーであれば（時間はかかるが）可能である。しかし、初心者のプログラマーや新しいプログラミング言語を学習中のプログラマーにとっては、この作業は非常に困難です。初心者のプログラマーや新しいプログラミング言語を学ぶプログラマーは、手元のソースコードの文法やスタイルを理解していない場合があり、ソースコードを読む負担が大きいのです。

一方、プログラミングに関する教育テキストではプログラムが何を行っているかを明示的に記述するため、初心者でも理解しやすく、かつ、慣れないプログラミング言語よりも読みやすい。

図1は、Pythonのソースコードの例と、ソースコード中の各ステートメントを説明するEnglish疑似コードである。[1]読者がPythonの初心者（あるいはプログラミングの初心者）であれば、図1の左側は、Pythonのソースコードの例です。
1は理解するのが難しいかもしれません。一方、図の右側はほとんどの英語圏の人が簡単に理解できますし、Pythonの具体的な操作の書き方も知ることができます（例えば、変数の型が整数でないことを確認したい場合、「`if not isinstance(`*something*`, int):`」と書くと分かります）。つまり、擬似コードは与えられたソースコードの「ボトムアップ理解」[3]を助けてくれるのです。

しかし、実際のプログラミング環境では、プログラマがプログラミング言語やプロジェクトを十分に理解すれば擬似コードは必要ないため、ソースコードに対応する擬似コードが存在することはほとんどない。また、既存のソースコードに擬似コードを無差別に手挿入することは、プログラマにとって大きな負担となる。一方、擬似コードを自動生成できれば、このような負担を軽減し、実際のプログラミングプロジェクトに応じた擬似コードを作成することができる。自動生成された擬似コードを実用的に利用するためには、以下の4点を満たすことが必要であると言えます。

- 擬似コードは、元のソースコードの振る舞いを記述するのに十分な精度を持っています。
- 読者の要望に応じて、擬似コードを提供する。
- 疑似コードを自動生成してプログラマの負担を軽減すること、また
- 擬似コードを生成する方法は、読者を待たせないように効率的であるべきです。

本研究では、ソースコードから擬似コードを自動生成する手法を提案する。特に、我々の提案する方法は、2つの大きな貢献をしている。

疑似コードとは、プログラム中の文の動作を自然言語（通常は英語、またはプログラマーの母国語）や数式で記述したものです。擬似コードは以下のような補助をします。

[1]擬似コードには多くの種類がありますが、本論文では、図1に示すように、擬似コードはプログラミング言語と自然言語の間の「行間」翻訳であると仮定します。この仮定は、ソースコードと擬似コードの関係を明確に定義するものであり、このタスクに機械翻訳を適用するための便利な第一歩となるものです。

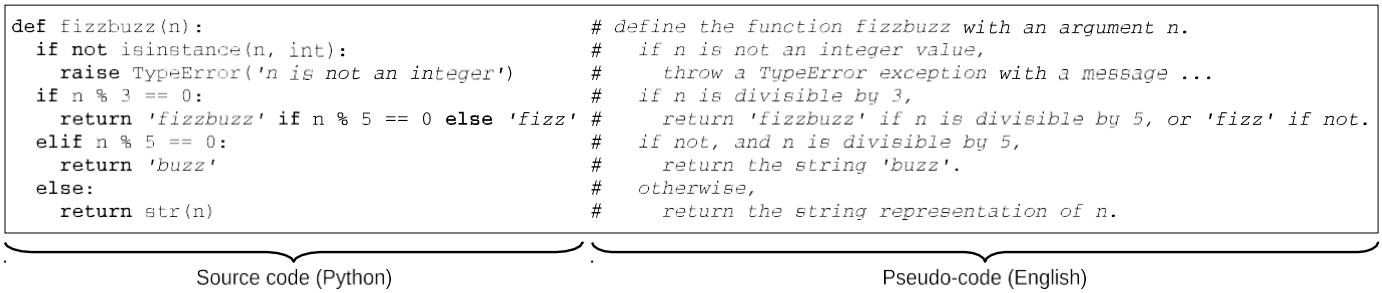


図1．Pythonで書かれたソースコードと、それに対応する英語で書かれた擬似コードの例。

- 私たちの知る限り、これは対応するソースコードを完全に記述する擬似コードを生成するための最初の方法です。このことは、コメント生成に関するこれまでの研究と対照的であり、読むべきソースコードの量を減らすことによって経験豊富なエンジニアを支援することを目的とし、以下のように記述されている。§II.
- 統計的機械翻訳(SMT)を用いて擬似コード生成を行うフレームワークを提案する。SMTは2つの言語間の翻訳方法を自動的に学習する技術であり、英語と日本語のような自然言語間の翻訳のために設計されたものである。これを擬似コード生成に応用することで、図1のようなデータを用意するだけで、他のプログラミング言語と自然言語の組み合わせにも容易に拡張できるようになることが最大のメリットです。

本論文では、まず、コメント自動生成に関する関連研究を参照し、その違いと本手法の利点を明らかにする(§II)。次に、本研究で用いる2つのSMTフレームワークの概要について述べる(§III)。次に、SMTフレームワークを擬似コード生成に適用する方法(§IV)、擬似コード生成システムを学習するためのソースコード/擬似コード並列データの収集方法(§V)、自動的およびコード理解基準を用いて生成された擬似コードを評価する方法(§VI)を説明する。実験では、Pythonから英語、Pythonから日本語の疑似コード生成タスクに我々の疑似コード生成システムを適用し、自動生成された疑似コードを元のソースコードとともに提供することで、プログラミング初心者がコードを理解しやすくなることを発見した(§VII)。最後に、他のソフトウェア工学タスクへの応用を含む、結論と今後の方向性について言及する(§VIII)。[2]

II. 関連作品

automaticのコメントと文書生成に関する先行研究はかなりの量にのぼる。我々の研究と従来の研究との重要な違いは、その動機にある。従来の研究は、読むべきコードの量を「減らす」という考えに基づいているのが普通である。これは、ベテラン技術者にとっては、大量のソースコードを効率的に理解することが目的なので、もっともな考えである。

疑似コードジェネレータの構築と評価に使用した2Dデータセットは、http://ahclab.naist.jp/pseudogen/。

そのため、コメントには、各ステートメントを詳細に記述するのではなく、コードが何を行っているかを簡潔にまとめることが期待される。しかし、技術的な観点からは、疑似コード生成は、ソースコードから自然言語記述を生成する点で、自動コメント生成や自動文書化と類似しています。そこで、ここでは、従来の手法の概要と、本手法との対比を行います。

コメント自動生成には、*ルールベース*アプローチとデータベースアプローチの2つの主要なパラダイムがある。前者については、例えば、Sridharaらは、実際のメソッド定義を人手で定義したヒューリスティックを用いて分析し、Javaメソッドの要約コメント生成を行っている[4], [5]．また，Buseらは，関数が投げる可能性のある例外の指定と，例外が発生するケースを含む文書を生成する方法を提案している[6]．また，Morenoらは，特にクラスの定義に着目して理解を助けるサマリーを生成する手法を開発した[7]．これらのルールベースのアプローチは、ソースコードの構造に密接に関連する詳細な情報を用いるため、言語固有の複雑な構造を扱うことができ、そのルールが与えられたデータに密接に一致する場合には、正確なコメントを生成することができる。しかし、特定の種類のソースコードやプロジェクト、あるいは新しいプログラミング言語や自然言語に対応したシステムを作るために新しいヒューリスティックが必要な場合、システム作成者が自ら手作業でこれらのヒューリスティックを追加しなければなりません。これはコメント生成システムの保守者の負担となり、ルールベースシステムの基本的な制約となる。

一方、ルールベースのシステムとは対照的に、*データベース*のアプローチはコメント生成やコード要約の分野で見られる。Wongらは、情報検索技術を利用して、プログラミングの質問と回答サイトのエントリからコメントを抽出する自動コメント生成手法を提案した[8]。また、自動テキスト要約[9]やトピックモデリング[10]の技術に基づき、場合によっては専門技術者の物理的動作と組み合わせて、コードの要約を生成する手法もある[11]。このようなデータベースのアプローチは、システムの精度を高めようと思えば、システムの構築に用いる「学習データ」の量を増やせばよいという大きなメリットがある。しかし、既存の手法は、既に存在するコメントを検索することが主体であるため、既存のコードを記述したコメントが学習データに既に存在しない場合、正確なコメントを生成する方法がない、という「データの疎性」の問題も大きく抱えている。

次章で紹介するSMTは、ルールベースのアプローチのように詳細なテキスト生成が可能でありながら、データベースのアプローチのようにデータから学習可能という、これらのアプローチの長所を併せ持つものである。

III. 統計的機械翻訳

SMTは自然言語処理（NLP）の応用であり、英語と日本語のような二つの自然言語間の語彙的・文法的関係を発見し、ある自然言語で記述された文を別の自然言語に変換するものである。近年用いられているSMTアルゴリズムは、主に次の2つの考え方に基づいている。

- 新しい入力言語と出力言語の小さな断片の間の関係（通常「ルール」と呼ばれる）を抽出する。
- は、これらの関係を利用して、統計的モデルを用いて新しい入力文の翻訳結果を合成し、どの翻訳が最適かを決定する[12], [13], [14].

SMTフレームワークは、近年、利用可能なデータ量の多さとコンピュータの計算能力の向上を主な理由として、急速に発展してきた。本節では、SMTフレームワークの基礎について説明する。特に、本研究でソースコードを擬似コードに変換するために用いた、主要なSMTフレームワークであるフレーズベース機械翻訳（PBMT）および木-
文字列機械翻訳（T2SMT）の詳細について説明する。

A. 統計的機械翻訳の概要

まず、SMTに基づく擬似コード生成の定式化のために、いくつかの記法を導入する。$\boldsymbol{s} = [s_1, s_2, \ldots, s_{|s|}]$ が入力トークンの配列である「原文」を記述し、$\boldsymbol{t} = [t_1, t_2, \ldots, t_{|t|}]$ が出力トークンの配列である「目的文」を記述するとする。この表記は配列の長さを表している。本稿では、ソースコードから擬似コードへの翻訳を考えているので、$\boldsymbol{s}$は入力ソースコード文のトークンを表し、tは擬似コード文の単語を表している。例えば、図2の小さなPython→英語の例では、$\boldsymbol{s}$はPythonのトークン列["if", "x", "%", "5", "==", "0", ":"] として記述され、$\boldsymbol{t}$は、["if", "x", "is", "divisible", "by", "5"] と記述されています。
|s|=7、|t|=6です。

SMTの目的は、最も確率の高いものを生成することです。

具体的には、原文 $\boldsymbol{s}$ が与えられたら、目標文 $\boldsymbol{t}$を作成する。そこで、$\boldsymbol{s}$ が与えられたときの $\boldsymbol{t}$ の条件付き確率分布、すなわち Pr($\boldsymbol{t}$ | $\boldsymbol{s}$) を指定したモデルを定義し、この確率を最大化する $\hat{\boldsymbol{t}}$ を求めます。

$$\hat{\boldsymbol{t}} \equiv \arg\max_{\boldsymbol{t}} \Pr(\boldsymbol{t} \mid \boldsymbol{s}) \quad (1)$$

とする。

各SMTフレームワークの違いは、以下のように導かれます。

B. フレーズベース機械翻訳

SMTのフレームワークとして最も有名なものの一つがフレーズベース機械翻訳（PBMT）[15]であり、原語と訳語のペア間のフレーズ間の関係を直接利用するものである。ソフトウェア工学の研究では、Karaivanovらがプログラミング言語の変換にPBMTフレームワークを適用する方法を提案し、2つのプログラミング言語の文の部分間の再関係（例えば、Javaの "System.out.println" からC#の "Console.WriteLine" ） を学習しています [16]（英語）。

PBMTモデリングを説明するために、フレーズペアのセット $\phi = [\phi_1, \phi_2, \ldots, \phi_{|\phi|}]$ を導入します。各 $\phi_n = \langle \boldsymbol{s}^{(n)}, \boldsymbol{t}^{(n)} \rangle$ は、ソース文 $\boldsymbol{s}^{(n)}$ の n *番目の*部分文と、対応するソース部分文に関連するターゲット部分文 $\boldsymbol{t}^{(n)}$ を repre- sent しています。例えば、Fig.2 では、$\boldsymbol{s}$ は次のように分割されています。
|φ|=4フレーズです。

$$\phi = \square \begin{matrix} \langle \text{もし」} \rightarrow \text{「 もし 」} \rangle \text{。} \\ \langle \text{"x"} \rightarrow \text{"x"} \rangle \text{.} \\ \langle \text{「\%5」} \rightarrow \text{「5 」} \rangle \text{で} \\ \langle \text{="0 :"} \rightarrow \text{"割り切れる "} \rangle \text{。} \end{matrix} \quad (2)$$

φは、§III-D で説明するように、並列コーパスから抽出された、様々なフレーズとフレーズの関係を確率で含む「フレーズテーブル」を用いて生成される。

しかし、ソース言語とターゲット言語の文法特性（トークンの順序など）は通常異なるため、単純に順序に従って $\boldsymbol{t}^{(n)}$ を連結しても、正確なターゲット文は得られない。例えば、式（2）の目標側フレーズを連結すると、「if x by 5 is divisible」という目標文が得られるが、これは流暢な英語文ではない。

この問題を回避するために、対象文のフレーズの適切な順序を選択する「リオーダリング」を行う必要がある。並べ替えを表現するために、フレーズアライメント $\boldsymbol{a} = [a_1, a_2, \ldots, a_{|\phi|}]$を導入する。ここで、*各*$a_n$ は、原文における*n番目*のフレーズペアの順番を表す整数である。図2では、$\boldsymbol{a} = [1, 2, 4, 3]$とし、1番目と2番目のフレーズペアはそれぞれの位置を維持し、3番目と4番目のフレーズペアはターゲット文の生成前に入れ替えられると仮定しています。

形式的には、PBMT モデルの条件付き確率 Pr($\boldsymbol{t}$ | $\boldsymbol{s}$) は、通常、原文と訳文に対して計算されたいくつかの確率を組み合わせた対数線形モデルを使って推定される[17]。

$$\hat{\boldsymbol{t}} \equiv \arg\max_{\boldsymbol{t}} \Pr(\boldsymbol{t} \mid \boldsymbol{s}) \quad (3)$$

$$\simeq \arg\max_{\boldsymbol{t},\phi,\boldsymbol{a}} \Pr(\boldsymbol{t}, \phi, \boldsymbol{a} \mid \boldsymbol{s}) \quad (4)$$

$$\propto \frac{\exp(\boldsymbol{w}^{\top} \boldsymbol{f}(\boldsymbol{t}, \varphi, \boldsymbol{a}, \boldsymbol{s}))}{\sum_{\boldsymbol{t}'} \exp(\boldsymbol{w}^{\top} \boldsymbol{f}(\boldsymbol{t}', \varphi, \boldsymbol{a}, \boldsymbol{s}))} \quad (5)$$

アークマックス $\boldsymbol{t},\varphi,\boldsymbol{a}$ エクスプローラ $\boldsymbol{t}'$

$\boldsymbol{f}(\boldsymbol{t}', \varphi, \boldsymbol{a}, \boldsymbol{s})$)

Pr($\boldsymbol{t}$ |

$\boldsymbol{s}$)を計算する方法である。この確率は，"並列コーパス"と呼ばれる原文と訳文のペアの集合を用いて推定される．例えば、図1は本研究で対象とする並列コーパスの一例であり、1対1文の

ソースコードと擬似コードの各行が1つずつ対応しています。

$$= \arg\max_{\boldsymbol{t},\varphi,\boldsymbol{a}} \boldsymbol{w}^{\mathsf{T}} \boldsymbol{f}(\boldsymbol{t}, \varphi, \boldsymbol{a}, \boldsymbol{s}), \quad (6)$$

ここで、$\boldsymbol{f}(\boldsymbol{t}, \varphi, \boldsymbol{a}, \boldsymbol{s})$ は翻訳時に計算された特徴関数、$\boldsymbol{w}$ は対応する特徴の重みベクトルであり、翻訳時の重要度を定義している。

を各特徴とする。直感的には、式(6)は、PBMT

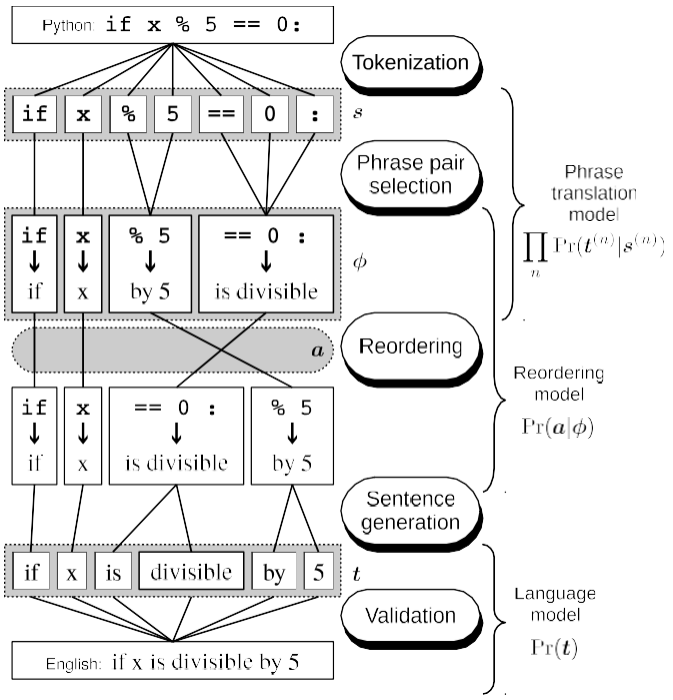


図2. Pythonから英語へのPBMT疑似コード生成の例。

モデルは、特徴の加重和で計算された最も高い「スコア」を持つ文 を見つけます：$\boldsymbol{w}^{\mathsf{T}} \boldsymbol{f}(t, \phi, a, s)$。これらの特徴の典型的な例としては、以下のようなものがある。

- **言語モデル**$\Pr(t)$は、以下のようにターゲット言語下での文tの流暢さを測定する。§III-E.
- **フレーズ翻訳**モデルによる積の計算 文中の個々のフレーズの確率$\Pr(t^{(n)} \mid s^{(n)})$の
- **並べ替えモデル**$\Pr(a \mid \phi)$は、各フレーズが特定の順序で並べられる確率を計算する。

PBMTは短いフレーズを翻訳して並べ替えるというメカニズムが単純な反面、擬似コード生成を直感的にモデル化する表現力に乏しい。例えば、2つのワイルドカード "*X* is divisible by *Y*" を含む英語の文字列は、ソースコード "*X* `%` *Y* `== 0:`" に対応することが直感的に分かるが、PBMTはこれらのワイルドカードを使うことができない。したがって、図2の例のϕは、"`==0:`" "is ⟨→⟩ divisible" のような明らかに「間違った」対応付けを使用しています。また、ソースコードには固有の階層構造があり、フレーズ間の翻訳や並べ替えを行うだけでは、明示的に利用することはできません。

C. 木から文字列への機械翻訳

前節で述べたように、PBMTに基づく擬似コード生成

T2SMTはもともと英語などの自然言語を翻訳するために考案されたもので、自然言語には曖昧さがあるため、sが与えられたときの確率T_s、すなわち$\Pr(T_s \mid s)$［19］、［20］を定義した確率的パーサーを用いて解析木T_sを得ます。したがって、T2SMTの定式化は、この構文解析確率を式（1）に導入することで得ることができる。

$$\hat{t} = \arg\max_{t} \Pr(t \mid s) \quad (7)$$

$$\arg\max_{t, t_s} \Pr(t \mid T_s) \Pr(T_s \mid s) \text{とする。} \quad (8)$$

幸いなことに、すべての実用的なプログラミング言語には、対応するソースコードを決定論的に解析できるコンパイラやインタプリタがあるので、私たちが提案する疑似コード生成法では確率 $\Pr(T_s \mid s)$ を無視することが可能である。したがって、可能な構文解析木は1つだけです。そのため、定式化はそれほど複雑ではありません。

$$\hat{t} \approx \arg\max_{t} \Pr(t \mid T_s) \text{とする。} \quad (9)$$

図3は、T2SMTに基づく擬似コード生成の過程を示している。まず、入力文をトークン化によりトークン配列に変換し、トークン配列を構文解析により構文木にすることで、構文木T_s を得ます。図3の重要な点は、T_s、プログラミング言語の文法によって定義され、「抽象的」な構文木ではないことである。この要件は、T2SMTアルゴリズムの内部動作の特徴に由来するものであり、このトピックは§IVで詳細に説明される。

確率 $\Pr(t \mid T_s)$ は、PBMTの確率と同様に定義されるが（通常、対数線形モデルとして定式化される）、2つの大きな違いがある。

- T2SMTでは、"導出 "と呼ばれる のフレーズペアφ の 代わりに、$[d_1, d_2, \ldots, d_{|d|}]$ を使用します。PBMTです。*各*d_n $\langle T_s(n) \rightarrow t(n) \rangle$ を表します。

では、ワイルドカードや2つの言語間の階層的な対応関係を利用することができません。T2SMTは、この問題を避けるために、図3に示すように、ソーストークンs の代わりにソース文T_s の構文木を利用します[18]。

図 3 の灰色のボックスはソースサブツリーとターゲットフレーズの間の関係をワイルドカードで表現したものである．すべての派生語は元の構文木 T_s の構造に従って接続され、ターゲット文はワイルドカードを対応するフレーズに置き換えることで生成される。

T2SMTの翻訳プロセスでは、ターゲットフレーズ中のワイルドカードの順序が自然にターゲットの順序を定義するため、明示的な順序変更モデルは含まれない。 -

D. SMTルールの抽出

PBMTとT2SMTのモデルを学習するためには、与えられた並列コーパスから、原文と訳文の各部分の関係を定義する翻訳ルールを抽出する必要がある。そのために、両文章間の「ワードアライメント」を用いる。ワードアライメントは、図4に示すように、両言語間の単語間レベルの関係を表す。標準的なSMT学習では、確率モデルと教師なし機械学習の手法を用いて、並列コーパスの統計量からワードアライメントを自動的に計算する[21], [22]。

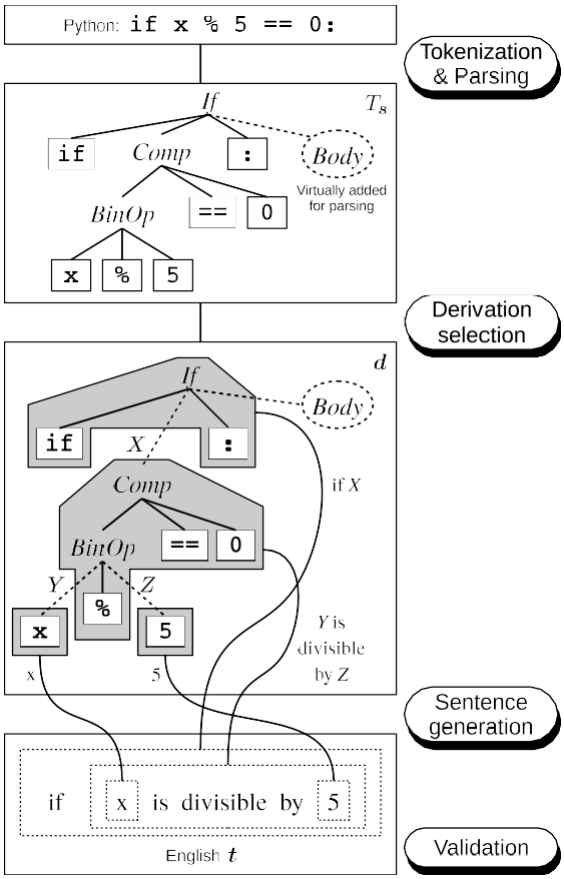


図3. Python to English T2SMT擬似コード生成の例。

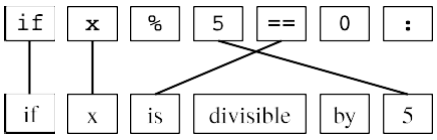


図4.2つのトークン文字列間のワードアライメント。

並列コーパスの各文章の単語アライメントを取得した後、以下の条件を満たすフレーズをPBMTフレームワークのフレーズペアとして抽出できると仮定する。

- 両フレーズ内の一部の単語が揃っており
- フレーズ外の単語がどちらのフレーズ内の単語にも揃うことはない。

例えば、図5は、1つのフレーズペアの抽出φ=0.

⟨ == 0 :" → "割り切れる"
⟩。

T2SMTフレームワークの場合、GHKMアルゴリズム[23]として知られる方法を用いて、木から文字列までの翻訳ルールである。GHKMアルゴリズムは、まず原文の構文木をアラインメントに従っていくつかのサブツリーに分割し、サブツリーとそれに対応する目標文の単語の

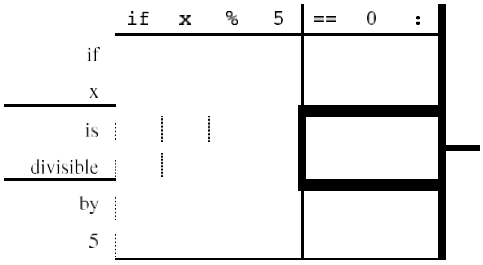


図5. 単語アライメントに応じたPBMT翻訳ルールの抽出。

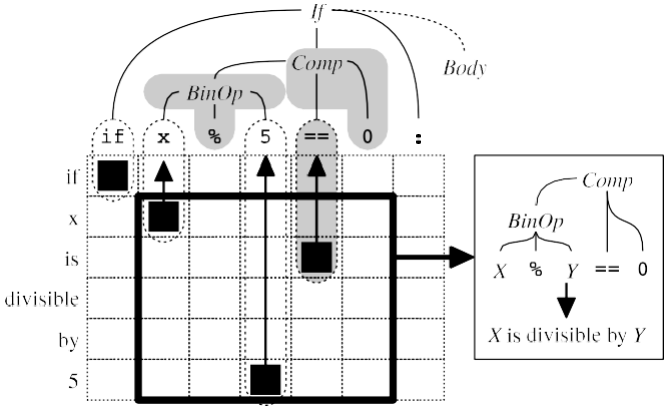


図6. 単語アライメントに応じたT2SMT翻訳ルールの抽出。

図6は、"*X* is divisible by *Y* "というワイルドカードを含むフレーズに対応する翻訳ルールを、"x "と "5 "に対応する2つのルールをそれぞれワイルドカードに置き換え、太枠内の最小ルールを組み合わせることによって抽出したものである。

抽出されたPBMTルールとT2SMTルールは、いくつかの指標を用いて評価され、その評価スコアとともに各フレームワークのルールテーブルに格納される。これらのスコアは確率 Pr(***t s***) や Pr(***t*** T_s) を計算する際に用いられる特徴関数を算出するために利用される。例えば，並列コーパスにおけるルールの出現頻度，フレーズの長さ，そして複雑なサブツリーも特徴としてよく使われる。

E. 言語モデル

SMTのもう一つの重要な機能は「言語モデル」であり、これは対象言語の文の流暢さを評価する。言語モデルは、対象文***t***が与えられたとき、***t***の各単語が前の単語から受ける確率の積として定義される。

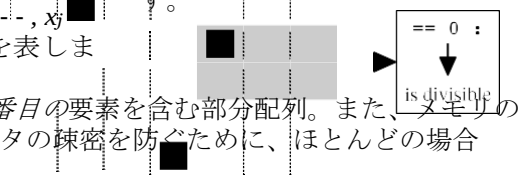
$$\Pr(t) \equiv \prod_{i=1}^{|t|} \Pr(t_i | t_1, t_2, \ldots, t_{i-1}) \quad (10)$$

ペアを "最小ルール $\prod_{i=1}^{|t|} \Pr(t_i | t_1, t_2, \ldots, t_{i-1}) \equiv$ "として抽出する。 (11)

ここで、***x***という表記は

i 1
1
$i=1$

次に、このアルゴリズムは、元の構文木に従っていくつかの最小ルールを組み合わせて、より大きなルールを生成する。例えば



= [x_i , x_{i+1} , - - , x_j]は、以下を表しま す。

***x*の**

*i*番目から*j番目*の要素を含む部分配列。また、メモリの節約とデータの疎密を防ぐために、ほとんどの場合

実用的な言語モデルは、"*N-gram*モデル "を使用します。

$$\Pr(t_i \mid t_1^{i-1}) : \Pr(t_i \mid t^{i-1}), \quad \text{(12)}$$

ここで、*n-gram* は *n* *個の*連続した単語として定義される。この近似は、次の単語 t_i - が前の (*n* 1) 単語のみを条件とすることを意味する。最も単純な*n-gram*モデルは、*各n-gram*の出現回数を用いて、ターゲット言語のテキストから単純に計算することができる。

$$\Pr(t_i \mid t^{i-1}) \equiv \frac{\text{count}(t^i)}{\text{count}(t^{i-1})} \quad \text{(13)}$$

ここで、count(***x***)は与えられたコーパスにおけるシーケンス***x*の**出現回数である。例えば、1-gram の "is" が 10 回出現し、2-gram の "is divisible" が 3 回出現する場合、 Pr(*t*ᵢ = "divisible" *t* ᵢ1 = "is") = 3/10 と見積もることができる。さらに、Kneser-Ney smoothing [24]と呼ばれる近似法を用いて、すべての*n-gram*の確率を平滑化することで、以下のことを防ぐことができる。
の問題を解決し、任意の文の確率を正確に計算することができるようになった。

また、*N-gram*モデルはあらゆる種類のシーケンスデータに容易に適用でき、ソフトウェア工学に頻繁に利用されており、典型的にはソースコードの自然さを測定したり[25], [26]、ソースコードの特徴を区別するために[27], [28]使われていることに注目すべきです。

IV. 疑似コード生成へのSMTの適用

前節では、2つのSMTフレームワークについて説明した。PBMTとT2SMTである。これらのフレームワークを用いる重要な利点は、SMTフレームワークが統計的アプローチであり、プログラミング言語から自然言語への翻訳ルールを学習データから自動的に取得できるため、疑似コード生成器を更新する際に新しい翻訳ルールを明示的に記述する必要がないことである。このため、技術者の擬似コード生成器作成・保守の負担が大幅に軽減される。新しいケースに対応する擬似コードを生成する場合、新しいケースごとに専用のルールを作成するのではなく、目的の文に対応する擬似コードを検索・作成するだけでよいのです。例えば、"if *X* is divisible by 2 "ではなく、"`if X % 2 == 0:` "というソースコードに対して、「*X*が偶数なら」というルールを作りたい場合、この例を含む文、例えば "if something is an even number" や "`if something % 2 == 0:`" をコーパスに追記すればよいのである。この作業は、ルールを明示的に記述するよりも明らかに簡単である。この種のデータは、我々の特定の疑似コード生成システムに精通していないプログラマでも作成できるし、既存のデータから採取できる可能性もあるからだ。

本稿で紹介したSMTベースの擬似コード生成器を任意のプログラミング言語と自然言語のペアで構築する場合、以下のデータとツールを準備する必要がある。

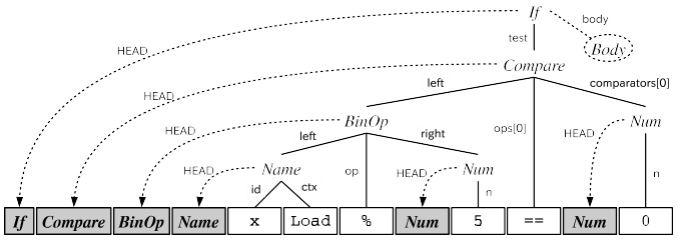


図7. ヘッド挿入の様子。

SMTによる擬似コード生成器の学習用**ソースコード/擬似コード並列コーパス。**

- **対象となる自然言語のトークン化**もし私たちがコン

Stanford Tokenizer[3]
などのメソッドを使用します。スペースがない言語（日本語など）を対象とする場合は、文中の各単語の間に明示的に区切り記号を挿入する必要があります。幸い、自然言語のトークン化は自然言語処理の研究分野として発達しているので、主要な言語のトークナイザーは簡単に見つけることができます。

ソースプログラム言語のトークン化器。通常、プログラミング言語の文法から具体的な定義を得ることができ、言語自体の標準ライブラリの一部としてトークナイザーモジュールが提供されていることが多い。例えば、Pythonでは`tokenize`モジュールでトークン化メソッドとシンボル定義が提供されている。

プログラミング言語ソースのパーサー。トークン化器と同様に、プログラミング言語のパーサーは、ソースコードをコードの構造を記述する解析木に変換し、言語の文法で明示的に定義されます。Pythonも`ast`モジュールで抽象的な構文解析を行う方法を提供しています。しかし、このモジュールの出力は、次のセクションで説明するように、直接使用することができません。

を、単語と単語の間にスペースを置く言語と見なす。英語）、より単純なルールベースのトークン化を使用することができます

A. 抽象構文木の表面改質

Pythonのライブラリ`ast`と同様に、プログラミング言語の標準ライブラリ
が提供する構文解析手法は、記述形式よりも言語の実行時セマンティクスに
よって定義される「抽象的」な構文解析手法であることが多い。前節で述べたように、木から文字列への変換規則を抽出するGHKMヒューリスティックは、構文木の葉の上で定義される単語の整列を利用する。抽象構文木はソースコードに存在するキーワードや演算子を葉ではなく内部ノードとして用いることが多いが、これらの表層語はターゲット言語の特定の語と強く関連している可能性がある（例えば、Pythonのトークン「`if`」は英語の「if」に対応する）。

もちろん、特定のプログラミング言語の文法定義から新しいパーサーを開発することもできるが、これは必ずしも容易ではない（例えば、C++の文法は公式仕様書の数百ページを使って定義されている）。本論文では、以下の2つのプロセスを用いて、抽象的な構文木から構文解析に似た木を生成する、より合理的な方法を適用します。

1) ヘッド挿入。まず、抽象構文木に、内部ノードのラベルを葉として含む「HEAD」と呼ばれる新しいエッジを追加する。図7は

3http://nlp.stanford.edu/software/tokenizer.shtml

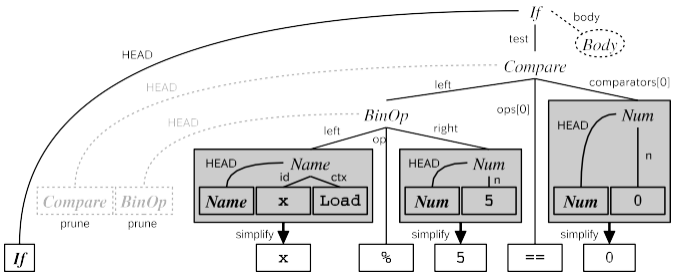


図8. 刈り込みと簡略化の処理。

というソースコードから生成された木に対して、Pythonのastモジュールを用いて頭出し処理を行った。この処理を適用することで、"if
"などの抽象構文木を作る過程で消えてしまった単語を単語整列の候補として扱うことができる。この処理は単純であり、抽象構文解析器を用いて抽象構文木を生成できれば、あらゆるプログラミング言語の抽象構文木に容易に適用できる。一方、見出し挿入処理には、HEADノードを木のどこに配置するかというオプションがある。本論文では、すべてのHEADノードを、英語では親の左端の子、日本語では右端の子に置き、対象言語の見出し語の順序に対応させる。

*2) 刈り込みと簡略化。*展開木では、見出し挿入後の葉の数が対象文の単語数より大幅に多くなり、自動的な単語アライメントにノイズを与える可能性がある。この問題を解決するため、いくつかの刈り込みと簡略化のヒューリスティックを適用し、木の複雑さを軽減する。我々は20個の手書きルールを開発し、言語の表面形式に関係しないと思われるノードを削除することで、頭挿入木の刈り込みと簡略化を行った。図8は、図7の結果に対してこの処理を適用した後の例である。

我々の擬似コード生成手法では、これらの変換手法のみ人為的な
工夫が必要である。しかし、これらの方法は比較的単純であり、擬似コードの言語にほとんど依存しないため、ルールベースのシステムを開発するよりも容易である。

B. 擬似コード生成器の学習過程

ここまでで、擬似コード生成システムの各部の詳細を説明した。

図9に擬似コード生成器の全体処理、図10にPBMTとT2SMTフレームワークの学習処理を示す。本論文では、4種類の擬似コード生成方式を比較した。*PBMTは*、PBMTフレームワークのプログラミング言語と自然言語のトークンを直接利用する方法である。*Raw-T2SMTは*、前節で説明した修正を行わない生のプログラミング言語の抽象構文木を用いて学習するT2SMTベースの手法である。*Head-T2SMTは、同じく*T2SMTに基づく手法であり、頭部挿入処理のみを行った修正木を用いて学習させる。*Reduced-T2SMTは*最後のT2SMTに基づく手法であり、頭部挿入、刈り込み、および簡略化の各処理を用いる。

は、システム構築を支援するオープンソースツールを利用することができる。本研究では以下のツールを使用する。MeCabは日本語文のトークン化
[29]、pialignは単語アライメントの学習 [22]、Kneser-Ney平滑言語モデルの学習
[30]、MosesはPBMTモデルによる目標文の学習・生成
[31]、TavatarはT2SMTモデルによる目標文の学習・生成
[32]、である。

V. ソースコードから擬似コードへ並列コーパス

IIIで述べたように、§SMTに基づく擬似コード生成器の学習には、ソースコードと擬似コードの並列コーパスが必要である。本研究では、既存のコードに擬似コードを追加するプログラマを雇い、Python-to-English およびPython-to-Japanese
の並列コーパスを作成することに成功した。

Python-
英語コーパスは、Webアプリケーションフレームワークであるの疑似コード作成を1名のエンジニアに依頼し、Pythonの文と英語の疑似コードのペア18,805件を含むコーパスを取得した。Pythonから日本語への変換は、まずエンジニア1名にプログラミング練習用の算数問題サイト Project Euler[4]の問題解答のPythonコード作成を依頼した。その結果、算数問題の解法に関連する関数定義177個を含む722個のPython文が得られた。これを用いて，VI-B, VI-Cで説明する人間評価実験を行った．このコードを別の日本人技術者に見せ、各文章に対応する日本語の擬似コードを作成させた。なお，この擬似コードはいずれも本研究とは無関係の別の技術者が作成したものであり，提案システムの学習データ作成に特別な知識は必要ないことを示している．

次に、このデータを実験用に別のセットに分割した。Python-to-Englishコーパスは、16,000文、1,000文、1,805文の3つに分割されました。16,000の要素は、SMTルールの抽出と言語モデルの学習に使用する「学習」データです。次の1,000個は、対数線形モデルの重みベクトル$\boldsymbol{w}$を最適化するための「開発」データである。最後の1,805件は、学習したSMTモデルを評価するための「テスト」データです。Python-to-Englishコーパスは Python-to-Englishコーパスよりも小さいので、90%を学習データ、開発データなし（wは各SMTフレームワークのデフォルト）、10%をテストデータとして10重クロスバリデーションを行っています。

VI. 擬似コード生成の評価

擬似コードを生成したら、その精度と有用性を評価したい。そのために、SMTの文献から採用した翻訳精度の自動評価指標、コード生成の精度の手動評価指標、および疑似コードがコード理解にどの程度貢献できるかの評価指標を使用します。以下の節で説明するように、英語と日本語の擬似コー
ドについては自動評価尺度を、日本語の擬似コードにつ

いては手 動評価精度とコード理解度を算出する。

SMTシステムを並列から学習させるためのアルゴリズム

コーパスは複雑すぎて、自分たちで開発することはできません。幸いなことに、私たちは

4https://projecteuler.net/

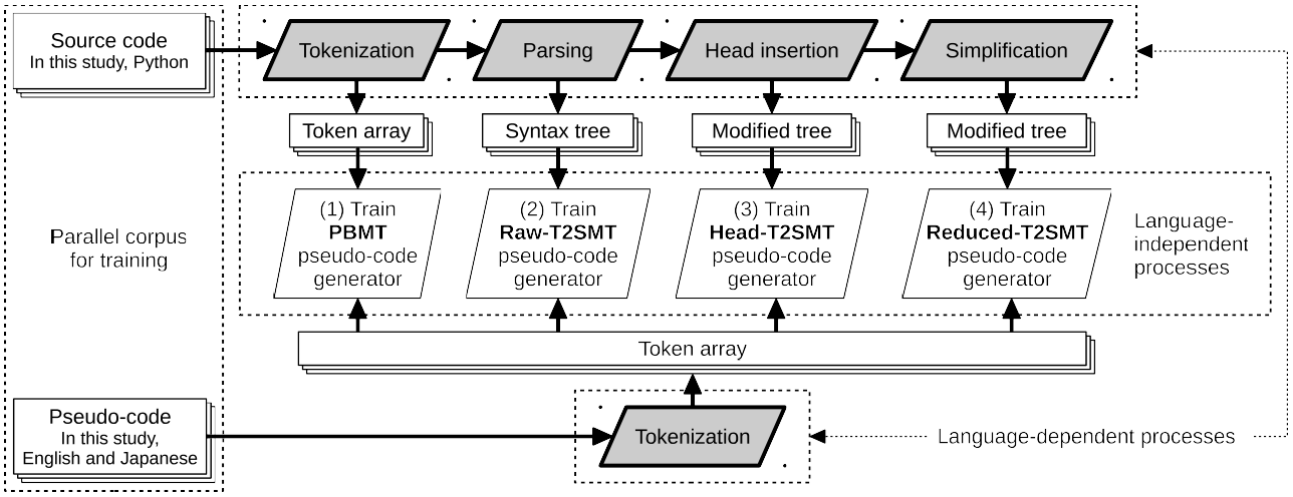


図9 提案手法の学習プロセス　各提案手法の学習プロセス全体（太枠は言語依存のプロセスを示す）。

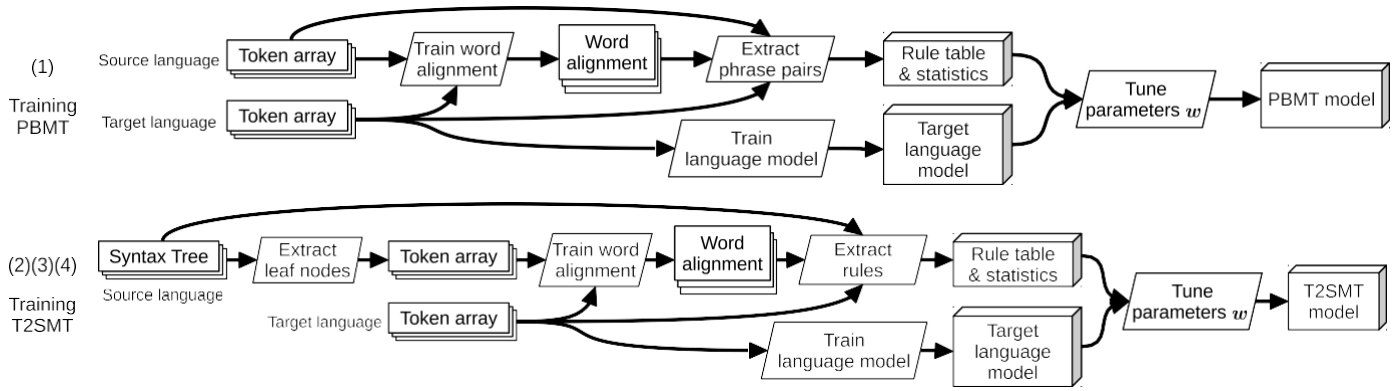


図10.　PBMTとT2SMTフレームワークの学習過程。

A. 自動評価-BLEU

まず，擬似コード生成の精度を自動的に測定するために，
機械翻訳の研究で広く使われている生成翻訳の自動評価指標である　BLEU（Bilingual　Evaluation　Understudy）［33］
を用いる．BLEUは、生成された翻訳と人間が作成した参照翻訳の類似度を自動的に計算します。BLEUは「*n-gram*精度」と「brevityペナルティ」の積として定義される。*n-*
gram精度は、システムが生成した長さ*n*の単語列が、人間の参照でも生成される割合を測定し、brevityペナルティは、システムが過度に短い仮説（高い*n-gram*精度を持っているかも）を生成しないようにするペナルティである。BLEUは範囲[0,1]の特定の実数値を与え、通常パーセントで表現されます。翻訳結果が参考文献と完全に一致する場合、BLEUスコアは1になります。

本研究では，各文章の人間が記述した擬似コードを参考に，生成された英語と日本語の擬似コードの品質をBLEU で評価した．

B. 人間評価（1）-受容性

BLEUは、参考文献をもとに生成された翻訳の精度を自動的かつ迅速に算出することができます。しかし

表 1. 受容性の定義

	レベルの意味
AA	(5)文法的に正しく、流暢である。
A	(4)文法的に正しく、流暢でない。
B	(3)文法的に正しくなく、理解しやすい。
C	(2)文法的に正しくなく、理解しにくい。
E	(1)理解できない、または重要な単語がいくつか欠けている。

は、各翻訳結果がソース文に対して意味的に正しいことを完全に保証するものではありません。また、翻訳の品質をより正確に測定するために、 表 I に示す受入能力基準[34]に従って、人間の注釈者による評価も行っている。また、Python の専門家 5 名を採用し、生成された日本語擬似コードの文単位での許容度を評価した。

C. 人間評価(2)-コード理解

特に初心者のプログラマにとって、疑似コードは対応するソースコードの理解を助ける可能性がある。この効果を調べるために、我々は、経験豊富な日本人Pythonプログラマと未経験の日本人Pythonプログラマを対象に、Webインターフェースを通じてコード理解の人間評価を実施した。

まず、関数の定義のサンプルを表示し

表　　2．理解しやすさの定義

レベル	意味
5	非常に分かりやすい
4	わかりやすい
3	わかりやすくも難しくもない
2	わかりにくい
1	非常にわかりにくい
0	理解できない

図1のような擬似コードを未経験のプログラマに読ませ，プログラマはそのコードの理解度を6段階で評価する．その結果，プログラマは各サンプルに対して，コードをどの程度理解したかの印象を6段階で評価する．この印象は、表IIで説明したリッカート尺度に類似している。評価インターフェースは，これらのスコアと，サンプルを提案してからプログラマがスコアを提出するまでの経過時間を記録する．この経過時間は，プログラマがサンプルを理解するために必要な時間であると考えることができる．次に，プログラマが読んだ関数の動作を母国語で記述してもらう．この結果は、Python経験1年未満（未経験を含む）の学生6名を含む14名の学生がこの実験課題を実施したことにより得られたものである。

Python-to-Japaneseコーパスに含まれる117個の関数定義をランダムに3つの設定に分割し（分割は被験者ごとに異なる）、それぞれ異なる種類の疑似コードで表示したものを使用しています。

- ソースコードそのもののみを表示し、疑似コードは表示しない*コード*設定。
- 人間が作成した擬似コード（擬似コード生成器の学習データ）を表示する*参照*設定。
- *Reduced-T2SMT*方式で自動生成されたコードを表示する*自動*設定です。

VII.　実験結果および考察

まず、Python-EnglishとPython-Japaneseのデータセットに対する提案手法のBLEUスコアと、Python-Japaneseのデータセットに対する各手法の平均受容度スコアをTABLE IIIに示す。

これらの結果から、BLEUスコアは比較的高く（Pythonから英語へのデータセットにおけるPBMTを除く）、提案手法は両方のデータセットに対して比較的正確な結果を生成していることがわかります。参考までに、現在の最新のSMTシステムは、比較的簡単なフランス語から英語へのペアにおいて約48のBLEUスコアを達成しており[35]、ソースコードから疑似コードへの翻訳は、自然言語間の翻訳よりも簡単であることを示唆しています。この結果は、自然言語ペアを対象とするSMTシステムが、ソース自然言語文のトークン化または構文解析の曖昧さによって引き起こされるノイズを常に含むのに対し、我々の疑似コード生成器はそのような入力曖昧さを持たないため、予想されることである。

また、Python-to-JapaneseデータセットのBLEUスコアは、Python-to-Englishデータセットより高いことがわかります。これは、それぞれのデータセットの特徴に起因していると考えられます。Python-to-Japaneseデータセットの元のソースコードは、一人のエンジニアがプログラミング練習用の演算問題から生成したものであり、セット内のすべての入力コードは似たようなプログラミングスタイルを共有しています。一方、Python-to-Englishデータセットのソースコードは、Djangoから抽出されたものであり

表 III.各仮想コード生成器の許容度（bleu%）と平均値。

擬似コードジェネレーター	BLEU% (英語)	(日本語)	平均値 受容体 (日本語)
ビープイエムティー	25.71	51.67	3.627
生T2SMT	49.74	55.66	3.812
ヘッド-T2SMT	47.69	59.41	4.039
リデュースド T2SMT	**54.08**	**62.88**	**4.155**

は、様々な目的のために多くの技術者を開発するため、ばらつきが大きくなります。Python-to-EnglishのPBMTと*Head-T2SMT*の結果は、このような特徴を反映している。III-Bで述べたように、PBMTベースの擬似コード生成器は文法的に複雑な文章を十分に扱うことができず、これが精度を低下させる原因であると思われる。各T2SMTは*PBMT*よりも有意に高いBLEUを達成しており、コメント生成前にプログラム構造を適切に解析することが、正確な擬似コード生成に不可欠であることを示している。

また、Python-to-Englishデータセットでは、*Head-T2SMT*システムが*Raw-T2SMT*よりも低いスコアになっていることに注目できる。この結果は、人間が作成した擬似コードと単語のアライメントのばらつきが原因であると思われます。頭出し処理によって構文木に多くの新しいトークンが導入され、その中にはプログラミングと自然言語の関係を表現する情報がないものがあり、自動単語アライメントに問題が生じています。しかし、Pythonから日本語へのデータセットでは、データのばらつきが少ないため、この問題は表面化しませんでした。*Reduced-T2SMT*システムは、両言語のすべての設定において最高のBLEUスコアを達成しており、この手法は、頭出し構文木の冗長構造を減らすことでこの問題を回避できることを示しています。

統計的有意性を評価するために、これらの結果からランダムに抽出した10,000セットの評価文に対してペアワイズブートストラップ検定[36]を実施した。この結果、すべての評価文において$p<0.001$以下の統計的有意性が得られた。*PBMT*システムに対するT2SMTシステムの比較、および$p<0.001$は、他のすべてのシステムに対する*Reduced-T2SMT*システム。

また、TABLE IIIでは、Python-to-Japaneseデータセットに対する疑似コードの受容度スコアも、BLEUの向上と相関して向上していることが分かります。特に*Reduced-T2SMT*は平均受容度4以上を達成しており、最先端の手法*Reduced-T2SMT*によって生成された疑似コードの多くが評価者によって文法的に正しいと判断されたことを意味しています。

図11は、各システムの可用性分布である。どのシステムも50%の文に対して、文法的に正しく、流暢な擬似コードを生成できることがわかります。さらに、各T2SMTベースのシステムは、"中間的な受容性"を持つ擬似コードをより少なく生成していることがわかります。これは興味深い結果ですが、T2SMTベースのシステムは、プログラミング言語の構文木を通じた文法情報を明示的に利用するため、ルールテーブルが入力文をカバーしていれば正確な擬似コードを生成できることが直感的に理解できます。

最後に，表 IVにコードアンダースタンディングの実験結果を示す．この表では，異なる評価者グループによって算出された 3 つの結果を示している．*経験者*グループには，14 名の評価者のうち，1 年以上の経験を持つ評価者が 8 名含まれている．

§

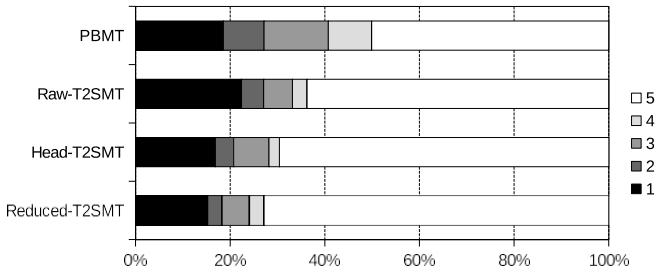


図11. 各システムの受容性分布

表4. 平均的な理解度の印象と平均的な理解までの時間。

グループ	設定	平均印象	平均時間
経験豊富な	*コード*	2.55	41.37
	参考 自動	3.05	35.65
	化された	2.71	46.48
未経験者	*コード*	1.32	24.99
	参考 自動	2.10	24.97
	化された	1.81	39.52
すべて	*コード*	1.95	33.35
	参考 自動	2.60	30.54
	化された	2.28	43.15

この結果、「参照」設定の結果は、全設定の中で最も理解しやすく、かつ読了までの時間が短いという結果を得た。この結果から、*Reference*設定の結果が、全設定の中で最も高い理解度と最短の読書時間を達成していることがわかります。これは、正しい擬似コードを提案することで、読者がソースコード全体を詳細に読もうとしたときに、コードの読みやすさと効率が向上することを意味しています。また、「*自動化*」設定の結果は、「*コード*」設定の結果よりも良い印象を与えることができます。しかし、他の設定よりも読上げ時間が長くなっています。これは、我々のジェネレータから生成された数行の奇妙な擬似コード（例えば、受容性においてスコア1である擬似コード）が、読者がソースコードを解釈しようとする際に混乱させる結果であると推測されます。*Automated* 法における生成誤差を減らすことで、原理的には*Reference*
の結果と同様に、この時間的ロスを減らすことができる。

図12に提案手法による3組の例文を示す。T2SMTを用いたシステム（特に*Reduced-T2SMT*）は、*PBMT*システムよりも精度の高い英文を生成していることが分かる。

VIII. 結論と今後の課題

本論文では、我々の知る限り、この種のものとしては初めての擬似コード生成手法を提案した。本手法は統計的機械翻訳（SMT）技術、特にフレーズベース機械翻訳（PBMT）と木-
文字列機械翻訳（T2SMT）に基づいており、文単位の疑似コード生成器を自動的に学習し、生成器の作成と更新に要する人手を少なくすることが可能である。提案手法は、Python-英語、Python-
日本語のプログラミング言語と自然言語のペアに対して、文法的に正しい擬似コードを生成することを実験的に示し、擬似コードをソースコードとともに提案すること

Python	`for node in graph.leaf_nodes(app_name):`
PBMT	for node in graph.leaf_nodes with an argument app_name,
Raw-T2SMT	for every node in, return value is the return value of the graph.leaf_nodes app_name,
Head-T2SMT	for every node in graph.leaf_nodes app_name,
Reduced-T2SMT	for every node in return value of the graph.leaf_nodes with an argument app_name,

Python	`if self._isdst(dt):`
PBMT	if self.call the method _isdst with 2 arguments dt, if it evaluates to true,
Raw-T2SMT	self._isdst with an argument dt, if it evaluates to true,
Head-T2SMT	if self._isdst with an argument dt, return the result.
Reduced-T2SMT	call the method self._isdst with an argument dt, if it evaluates to true,

Python	=
PBMT	
Raw-T2SMT	
Head-T2SMT	
Reduced-T2SMT	

で、未知のプログラミング言語に対するプログラマのコード理解力を向上させることを明らかにした。SMTフレームワークを利用するために、文(または文節)を含むパラレルコーパスを準備しました。

context[self.var_name] = obj
self.call the method self.var_name context [] = obj.
call the method obj, substitute it for value under the ' key of the self.var_name key of the context dictionary.
substitute obj for value under the self.var_name key of the context dictionary.
substitute obj for the value under the self.var_name key of the context dictionary.

図12. 各システムから生成された擬似コードの例。

構文木）のペアを互いに関連付け、並列コーパスをSMTに適した形式に調整するためのいくつかのアルゴリズムについて説明しました。

将来的には、提案するSMTフレームワークを用いて、コメント自動生成の標準的な設定に近い、複数文を扱える疑似コード生成器を開発する予定である。そのためには、ソースコードと、ソースコード中の複数の文に対応するコメントの高品質な並列コーパスを見つけるか作成する必要がありますが、これは行間コメントの作成よりも整形の難しい問題であり、将来の興味深い課題です。また、大規模なソフトウェアプロジェクト環境において、経験豊富なプログラマによる自動擬似コード生成の利用についても調査する予定です。例えば、自動生成された擬似コードは、プログラマが自分の書いたコードが実際に期待通りに動作しているかどうかを確認したり、ソースコード中の既存の（一行）コメントが古くなり更新が必要な場合に確認するのに利用できるかもしれません。

ACKNOWLEDGMENT

この研究の一部は、「戦略的国際ネットワーク形成推進事業」によるものである。また、本研究に関して有益な助言をいただいた井原秋則氏に感謝する。

参考文献

[1] R.このような状況において，「コードマップを用いたソフトウェア開発」， *Commun.ACM*, vol.53, no.8, pp.48-54, 2010.

[2] M.M. Rahman and C. K. Roy, "Surfclipse:イドのコンテキストアウェアメタ検索」、*Proc. ICSME*, 2014、pp.617-620。

[3] M.-A.Storey, "Theories, tools and research methods in program comprehension:このような場合、「*ソフトウェア品質ジャーナル*」、vol.14、no.3、pp.187-208、2006年。3, pp.187-208, 2006.

[4] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proc.ASE*, 2010, pp.43-52.

[5] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proc. ICSE*, 2011, pp.101-110.

[6] R.P. Buse and W. R. Weimer, "Automatic documentation inference for exceptions," in *Proc. ISSTA*, 2008, pp.273-282.

[7] L.また、このような場合、「自然言語要約の自動生成」を行うことができる。

[8] E.Wong, J. Yang, and L. Tan, "Autocomment:Mining question and answer sites for automatic comment generation," in *Proc.ASE*, 2013, pp.562-567.

[9] S.Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On use of automated text summarization techniques for summarizing source code," in *Proc.WCRE*, 2010, pp.35-44.

[10] B.P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques," "ソースコード要約技術の評価:レプリケーションと拡張、"*Proc.ICPC*, 2013, pp.13-22.

[11] P.Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S'D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proc. ICSE*, 2014, pp.390-401.

[12] P.Koehn, *Statistical Machine Translation*.ケンブリッジ大学出版局, 2010.

[13] A.ロペス, "統計的機械翻訳," *ACM Computing Surveys*, vol. 40, no.3, pp.8:1-8:49, 2008.

[14] P.F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer, "The mathematics of statistical machine translation:パラメータ推定」『*Computational Linguistics*』19巻2号、263-311頁、1993年。

[15] P.Koehn, F. J. Och, and D. Marcu, "Statistical phrase-based translation," in *Proc.NAACL-HLT*, 2003, pp.48-54.

[16] S.このような場合、「逐語的な翻訳」であることが望ましい。*Onward!*, 2014, pp.173-184.

[17] F.J. Och and H. Ney, "The alignment template approach to statistical machine translation," *Computational Linguistics*, vol. 30, no.4, pp.417-449, 2004.

[18] L.Huang, K. Knight, and A. Joshi, "Statistical syntax-directed translation with extended domain of locality," in *Proc.AMTA*, vol.2006, pp.223-226。

[19] D.クライン、マニング、"Accurate unlexicalized parsing", in *Proc.ACL*, 2003, pp.423-430.

[20] S.このような場合、「曖昧さ」を解消するために、「曖昧さ」を解消した上で、「曖昧さ」を解消するために、「曖昧さ」を解消した上で、「曖昧さ」を解消した上で、「曖昧さ」を解消した上で、「曖昧さ」を解消した上で、「曖昧さ」を解消した上で、「曖昧さ」を解消した上で、「曖昧さ」を解消した上で、「曖昧さ」を解消した上で、「曖昧さ」を解消した上で、「曖昧さ」を解消する。

[21] P.F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer, "The mathematics of statistical machine translation:パラメータ推定」『*Computational Linguistics*』vol.19, no.2, pp.263-311, Jun.1993.

[22] G. Neubig, T. Watanabe, E. Sumita, S. Mori, and T. Kawahara, "An unsupervised model for joint phrase alignment and extraction," in *Proc.ACL-HLT*, Portland, Oregon, USA, 6 2011, pp.632-641.

[23] M.Galley, M. Hopkins, K. Knight, and D. Marcu, "What's in a translation rule?" in *Proc.NAACL-HLT*, 2004, pp.273-280.

[24] R.Kneser and H. Ney, "Improved backing-off for m-gram language modeling," in *Proc. ICASSP*, 1995, pp.181-184.

[25] A.また，このような場合，「ソフトウェアとその周辺環境との関係性」，「ソフトウェアとその周辺環境との関係性」，「ソフトウェアとその周辺環境との関係性」，「ソフトウェアとその周辺環境との関係性」，「ソフトウェアの自然さについて」，「ソフトウェアの自然さについて」，「ソフトウェアの自然さについて」，「ソフトウェアの自然さについて」，「ソフトウェアの自然さについて」，「ソフトウェアの自然さについて」，「"*Proc.*

[26] T.T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. Nguyen, "A statistical semantic language model for source code," in *Proc.FSE*, 2013, pp.532-542.

[27] Z.Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proc.FSE*, 2014, pp.269-280.

[28] A.T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proc. ICSE*, 2015.

[29] T.工藤 拓也, 山本 和彦, 松本 恭代, "条件付き確率場の日本語形態素解析への応用", 日本音響学会論文集, pp.*EMNLP*, vol.4, 2004, pp.230-237.

[30] K.Heafield, I. Pouzyrevsky, J. H. Clark, and P. Koehn, "Scalable modified Kneser-Ney language model estimation," in *Proc.ACL*, Sofia, Bulgaria, August 2013, pp.690-696.

[31] P.Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N.ベルトルディ、B・コーワン、W・シェン、C・モラン、R・ゼンス、C・ダイアー、O・ボジャール。A.Constantin, and E. Herbst, "Moses:統計的機械翻訳のためのオープンソースツールキット", in *Proc.ACL*, 2007, pp.177-180.

[32] G. Neubig, "Travatar: A forest-to-string machine translation engine based on tree transducers," in *Proc.ACL*, Sofia, Bulgaria, August 2013, pp.91-96.

[33] K.K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu:機械翻訳の自動評価のための方法」、*Proc.ACL*, 2002, pp.311-318.

[34] I.後藤, K. P. Chow, B. Lu, E. Sumita, and B. K. Tsou, "Overview of patent machine translation task at the ntcir-10 workshop," in *NTCIR-10*, 2013.

[35] O.Bojar, C. Buck, C. Federmann, B. Haddow, P. Koehn, J. Leveling, C.Monz, P. Pecina, M. Post, H. Saint-Amand, R. Soricut, L. Specia, および A.Tamchyna, "Findings of the 2014 workshop on statistical machine translation," in *Proc.WMT*, 2014, pp.12-58.

[36] P.このような場合、「機械翻訳の評価における統計的有意差検定」、P. Koehn, "Statistical significance tests for machine translation evalua- tion," in *Proc.EMNLP*, 2004, pp.388-395.