目的言語側の構造を考慮した 自然言語からの構文的に正しいソースコード生成

札場 寛之 † 小田 悠介 † 吉野 幸一郎 †§ Graham Neubig ‡† 中村 哲 †

† 奈良先端科学技術大学院大学 ‡Carnegie Mellon University § 科学技術振興機構

{fudaba.hiroyuki.ev6, oda.yusuke.on9}@is.naist.jp, gneubig@cs.cmu.edu,
{koichiro, s-nakamura}@is.naist.jp

1 はじめに

効率的にコンピュータを操るためには、個々の目的に応じたプログラミング言語の習得が求められるが、これは多くの困難を伴う。その原因のひとつとして、個々のプログラミング言語を習得する難しさが挙げられる。各々のプログラミング言語は独自の文法や仕様を持ち、これらをすべて覚えるのは困難である。

一方で、多くの人間は英語や日本語といった自然言語を自由に操れる。通常自然言語は人間と人間の間のコミュニケーションを円滑に行うための手段として使われるが、それと同じように人間とコンピュータの間を取り持つことができれば、新たにプログラミング言語の文法や仕様を学ばなくてもコンピュータをより効率的に扱うことができるようになると考えられる [9]。

この問題の解決手段の一つとして、統計的なモデルによる自動的な変換規則の学習が挙げられる。例えば質問応答の分野では、統計モデルを用いて質問文から SQL クエリを生成する研究がなされている [1]。この研究では、自然言語から統計モデルに基づいて意味表現を生成し、ここからルールによって SQL クエリを生成している。この手法では意味表現が付与された学習データを作成する必要があるものの、自然言語と意味表現を結びつけるルールを自動で学習することができる。人手で曖昧な自然言語を扱うルールを作る必要がないため、相対的に開発のコストが低いと考えられる。

しかし、プログラミング言語には厳密な構文が定義されており、上記の手法はこうした構文を陽に考慮しているわけではない。プログラムの実行は字句解析、構文解析、意味解析、評価の順に行われ、構文を考慮していなければ構文解析でエラーが起きてしまい、以降の処理が行われないようなクエリが生成されてしまう。[1] では、単純な構文を持つ木構造状の意味表現から SQL クエリを生成することで、目的言語側の構造を一部考慮しているが、意味表現と SQL クエリは構文的に等価ではないため、表現可能な範囲が異なる。そのため、意味表現として正しい構造が生成されていても、そこから生成される SQL クエリが SQL の文法を満たすとは言えない。

本研究の目的は、プログラミング言語の仕様を満たすソースコードの生成である。本研究では、上記の先行研究と比較して 2 つの貢献をする。まず、[4] のように、中間意味表現を用いずにプログラミング言語を生成する手法を提案する。こうすることによって、自然言語とプログラミング言語の対訳コーパスさえあれば、中間的な意味表現の考案とアノテーションという手間を省くことができる。また 2 つ目として、プログラミング言語の仕様を満たすソースコードを生成するように制約する手法を提案する 。具体的には、先に挙げられた構文解析不可能なクエリが生成される問題を解決するため、統計的機械翻訳を用いて自然言語文から制約付き構文木を生成し、それを元にソースコードを生成する（図 1）。実験により、統計モデルによって生成を行う際の意味表現の文法に制約を入れることで、高精度で構文解析が成功するソースコードを生成できることを確認した。

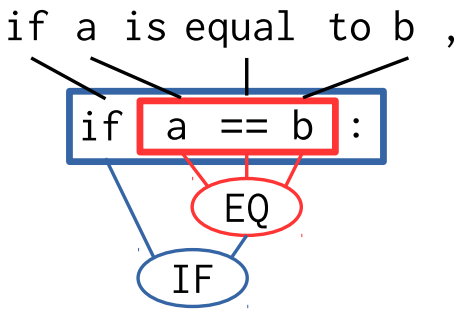

図 1: 自然言語（上）からソースコードの構文木（下）を生成する提案手法

2 プログラミング言語の性質

2.1 厳密な構文

自然言語とは違い、プログラミング言語には厳密な構文が存在する。図 2 に挙げているように、プログラミング言語として実行可能なソースコードを生成するためにはその文法を守る必要がある。これら文法は各プログラミング言語固有のものである。前述したとおり、これを満たさないプログラムは構文解析を行った時点で失敗するので、実行することができない。そのため、その先の意味解析まで処理を続けるためには構文的に正しいプログラムである必要がある。

```
# 括弧の対応が足りない
func(hello, world
```

```
# if は予約語なので変数として扱えない
if + 4
```

```
# 数字は代入先にはなれない
2501 = a + b
```

```
# 代入先のタプルに変数を含んでいない
() = 3
```

図 2: 構文的に正しくないソースコードの例

2.2 中間表現である抽象構文木

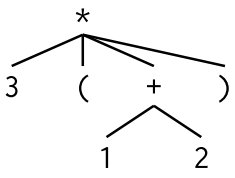

図 3: 構文木の例

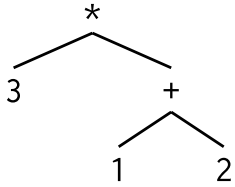

図 4: 抽象構文木の例

目的言語側であるプログラムの構造を考慮するためには、プログラムを何らかの構造へ対応付けておき、プログラムを生成する際には、まずその構造を生成し、その構造からプログラムへと変換する手法が考えられる。その場合、プログラムを表現する構造としてはそれが持つトークンや意味をすべて木構造として表した構文木が最も望ましい（図 3）。しかし、プログラムを実行する上では構文木は不要な情報が多く、コンパイル時は通常抽象構文木と呼ばれる木を生成する。抽象構文木とは、言語の意味に関係のある情報、つまりその後の意味解析に必要な情報を木構造状に表現したデータ構造である（図 4）。そのため、あるプログラムに基づいて生成される抽象構文木は、そのプログラムの動作を表現していると考えられる。本研究では抽象構文木を用いて構文木の代替とする。

3 目的言語側の構造を考慮した生成

3.1 String-to-Tree 翻訳

我々の手法は String-to-Tree 翻訳 [2] に基づくものである。String-to-Tree 翻訳は統計的機械翻訳の手法の一つで、単語列を入力とし、木構造を出力する。木構造は同期木文法に従い導出される。

String-to-Tree 翻訳モデルは同期木の書き換えを行うことで木を生成する。X を非終端記号、γ を終端器号と非終端記号、α を各葉ノードが終端・非終端記号の木構造、$\sim$ を γ と α の一対一対応とすると、同期木の書き換えは式の右側のペアの置き換えであると言える。この同期木ルールは、自然言語とプログラミング言語のペアの集合であるパラレルコーパスから自動で学習できる。

$$X \rightarrow \langle \gamma, \alpha, \sim \rangle \tag{1}$$

String-to-Tree 翻訳は、原言語 F を目的言語の木 T_E へ変換するものであるが、この変換の過程は前述したルールを用いた導出 D によって表される。したがって、String-to-Tree 翻訳は自然言語から木を生成する導出を求める問題に帰着する。導出の質を評価するスコア関数を $s(D)$ とすると、これは以下の式を最大化する $\hat{D}$ を求めることにほかならない。

$$\hat{D} = \underset{D \text{ s.t. } f(D)=f}{\arg\max} \; s(D). \tag{2}$$

ここで、$s(D)$ は導出の特徴の重み付き和である。

$$s(D) = \sum_i w_i \phi_i(D). \tag{3}$$

ここで用いる同期木文法は文脈自由文法であり、文脈自由文法の範囲に収まる文法を生成することができる。しかし逆に、文脈に依存するような文法は完全に取り扱うことができない。プログラミング言語の文法は一般に文脈自由文法によって記述されるが、プログラミング言語によっては文脈依存な仕様も存在し、構文エラーとなる範囲が広い場合がある。その場合、文脈自由文法を用いた枠組みではこの構文エラーを解決することができない。

3.2 抽象構文木に対する制約

本研究では、String-to-Tree 翻訳を用いて自然言語から抽象構文木を生成する。しかし、以下に示すように抽象構文木とプログラミング言語の文法の表現力には違いがあるため、プログラミング言語の処理系が生成する抽象構文木の文法をそのまま String-to-Tree モデルで学習して得た抽象構文木への変換ルールは、必ずしもプログラムの自動生成に適切ではない。そのため、この変換ルールに制約を入れる必要がある。

3.2.1 未知語への対応

プログラムのソースコード中には変数名や文字列リテラルが多く出現するため、自然言語の文中に比べ未知語が頻出する。したがって、未知語が入力にあった場合にこれを目的言語側の適切な場所に挿入する必要がある。ここで、プログラミング言語には変数名に制約がある場合がある点問題になる。例えば、Python では変数名の最初の文字は数字以外でなくてはならない。また予約語やキーワード、具体的には`'if'`や`')'`といったような構文的に特別な意味を持つ語と同じ名前を使うことができない。ある語が予約語かどうかの判定は言語仕様によって判別することができる。本研究では、プログラミング言語の処理系に備わっているパーザを用いてこれを判別する。未知語をパーザで解析した結果を元に、その語が予約語なのか、変数名として正しいのか、そのいずれでもないかを判別できる。これによって、未知語の不適切な場所への挿入を防ぐことができるようになる。例えばもし変数名として正しくない未知語が現れたときは、それを変数として扱う導出を選ばないようにする。

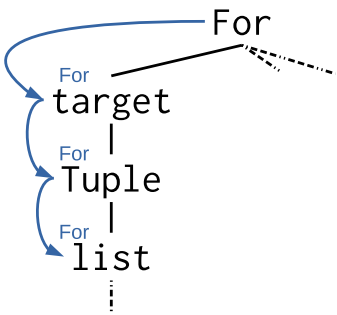

図 5: `for a, b in c:`の抽象構文木に制約

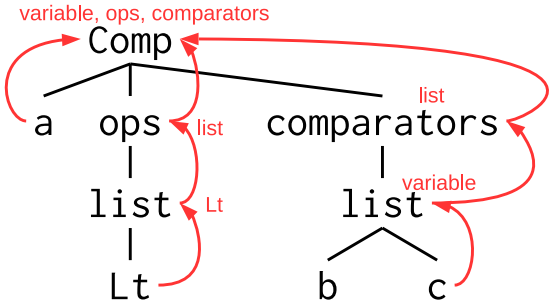

図 6: `a < b < c` の抽象構文木に型情報を付与

3.2.2 構文に現れない約束

例えばPythonには変数への代入は許されているが、数値への代入は許されていない。もし数値への代入を試みると構文エラーとなる（図 2）。このような問題は他にも多く見られ、`for` 文や内包表記にはこのような制約が見られる。しかしこれらは抽象構文木上には現れない約束である。この暗黙の約束は、抽象構文木の文法がソースコード側から生成されるために設計されており、通常の利用では問題にならないと考えられる。具体的にいえば、プログラムから生成された抽象構文木は構文的に正しいソースコードからしか生まれないのであって、誤りを含まない。そのため、抽象構文木の文法はプログラミング言語の文法より単純で制約の少ないものであったとしても、その後の処理に影響しないと考えられる。であるから、例えば代入を表現する抽象構文木の文法の被演算子に関して何の制約もない場合がある。故に数値への代入といった構造も抽象構文木としては正しい場合がある。その結果、抽象構文木として文法的に正しい構造であっても、それから生成されたソースコードが正しいとは限らないということになる。

この問題に対応するため、抽象構文木の文法を学習する際にルールを導入して解決する。例えば、代入のように暗黙の約束が存在する構造に対しては、その構造内にあるすべての非終端記号に対して代入の構造内にあるという情報を再帰的に付与していく（図 5）。このようにすることで、デコード時に生成される代入の構造内にあるすべての要素が代入の文法に従うことを保証することができる。その結果として、構造にとって不適切なルールが導出されることがなくなる。

3.2.3 型情報の付与

String-to-Tree 翻訳モデルは文脈自由文法を学習する。そのため、役割が異なっていても同一の名前を持つ非終端記号の区別ができない。例えば、Python の抽象構文木にはたびたび `list` という非終端記号が現れる。これは構造が複数の子を持つ場合、これを表すために使われる。Python では複数の子を持つ構造が多くあり、一例として二項演算子がある。`a < b < c` のようなソースコードは、二項演算子`<`を 2 つ持つ `list` を用いて表される。しかし同様に、`f(x, y)` のような関数の引数 `x, y` も 2 つ変数を持つ `list` として表されてしまう。したがって、この 2 つの `list` はデコード中区別されることがないので、関数の引数 `x, y` のルールが二項演算子として扱われてしまうことがある。その結果、`a x b y c` のような構文的に正しくないソースコードが生成される。これを解決するため、すべての非終端記号に対し、その直接の子ノードの型情報を再帰的に付与する（図 6）。これを行うことで、`list` のように役割の異なる同一の名前を持つ非終端記号をデコード中区別することができる。つまり、先ほどの例を挙げるなら、それぞれの `list` は `list-of-Operators` と `list-of-Variables` として区別されるようになる。

3.3 抽象構文木からソースコードへ変換

本研究では抽象構文木からソースコードへ変換するために小さなルールセットを用いる。Python では、抽象構文木が与えられた時、それに対応するソースコードを出力するライブラリが提供されており、本研究ではこれを用いた。これが提供されていないプログラミング言語が対象の場合開発者が自分で用意する必要があるが、限られた文法によって記述される抽象構文木からソースコードを生成するルールを用意することは、自然言語からプログラムを生成するルールと比べれば容易である。

4 実験

4.1 実験設定

対訳コーパスとして英語の指示文と Python のソースコード 17000 行を用いた。このコーパスは Python のウェブフレームワークである Django のソースコードであり、アノテータによってその動作を説明する英文を行単位で付与されている [7]。対訳コーパスのうち 15000 行を学習データ、1000 行を開発データ、1000 行をテストデータとした。実験の評価は BLEU[8]、構文的な正しさと意味的な正しさによって評価する。構文的な正しさとは Python の ast パーザが正常に解析できるソースコードの割合である。意味的な正しさとは主観的評価であり、ユーザが意図した動作をするソースコードの割合によって評価した。BLEU によって評価する理由は、意味的な正しさを評価するのは手間がかかるため、機械翻訳の評価指標で代用できるかを調べるためである。ベースラインとして、同じ統計的機械

Method	BLEU	構文的正しさ	意味的正しさ
PBMT	66.0	51.5	**62**
T2S	**76.8**	72.7	34
ANN	62.8	90.6	16
S2T	15.1	54.9	9
S2T w/ NS	44.8	84.0	22
S2T w/ NSR	46.9	94.7	21
S2T w/ NSRT	44.4	**99.5**	20

表 1: 評価結果 (%)

リファレンス	`read(size=None)`
S2T w/ NSRT	`read(size=None, size=None)`

図 7: 構文的に正しくないソースコードの例

翻訳であるフレーズベース翻訳 (PBMT)[3]、Tree-to-String 翻訳 (T2S)[6]、注意型ニューラル翻訳 (ANN)[5] の 3 手法を比較対象とした。これらはいずれも目的言語側の構造を考慮しない手法である。String-to-Tree 翻訳モデル (S2T) は制約をつけた場合とそうでない場合を比較するため、制約を入れない場合 (S2T)、未知語への対応を考慮した場合 (S2T w/ NS)、加えて構文に現れない約束を考慮した場合 (S2T w/ NST)、更に型情報を付与した場合 (S2T w/ NSRT) で比較する。

4.2 実験結果

表 1 は実験の評価結果である。String-to-Tree 翻訳は 99.5%以上の場合で構文的に正しいソースコードを生成していることが分かる。100%ではない理由はいくつかあり、構文エラーとなった失敗した例を図 7 に示す。

Python の関数は引数としてキーワード引数を受け取ることができる。しかし、同じキーワードを同時に 2 つ以上渡すと構文エラーとなる。これは文脈に依存した文法であり、Python の構文エラーとなる範囲が文脈自由文法でカバーできる範囲を超えていることが原因である。これを解決するためには文脈依存文法を取り扱えるモデルが必要となるが、String-to-Tree 翻訳モデルは文脈自由文法に基づきデコードするので、これを解決できない。

意味的な正しさでは PBMT が最も高かった。他のモデルでは、モデルの複雑さに対して用いたデータが小さく、学習したルールがスパースになってしまったために PBMT が最も高いということになったのではないかと考えられる。

BLEU に関しては、意味的な正しさとの関係はなかった。このため、意味的な正しさを自動的にはかる新たな指標が必要であることがわかった。

5 終わりに

本研究では String-to-Tree 翻訳モデルを用いて自然言語から 99.5%以上の確率で実行可能なソースコードを生成した。比較対象である PBMT や T2S、ANN と比べても高いことから、目的言語側の構造を考慮する必要があることが示された。また、単純に String-to-Tree 翻訳モデルを適用した場合、構文上に現れない約束が影響し、これらを考慮した場合と比べて著しく精度が落ちることが示された。今後の課題として、意味的な正しさを高めていく必要がある。また、文脈依存文法を扱えるモデルを考える必要もある。なお、本研究では特に英語から Python への翻訳を試みたが、用いた手法を他言語間の翻訳に適用することは可能である。

謝辞：本研究の一部は JSPS 科研費 JP16H05873 の助成を受けたものです。

参考文献

[1] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, p. 1257, 1985.

[2] David Chiang. A hierarchical phrase-based model for statistical machine translation. ACL '05, pp. 263–270, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[3] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proc. ACL*, 2007.

[4] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Toma´sKocisky´, Andrew Senior, Fumin Wang, Phil Blunsom. Latent predictor networks for code generation. pp. 599–609, Berlin, Germany, USA, August 2016. Association for Computational Linguistics.

[5] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. pp. 1412–1421, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[6] Graham Neubig. Travatar: A forest-to-string machine translation engine based on tree transducers. In *ACL (Conference System Demonstrations)*, pp. 91–96. The Association for Computer Linguistics, 2013.

[7] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. Lincoln, Nebraska, USA, November 2015.

[8] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pp. 311–318, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[9] Kenji Sagae, Gwen Christian, David De Vault, and David R. Traum. Towards natural language understanding of partial speech recognition results in dialogue systems. pp. 53–56, Boulder, Colorado, USA, June 2009. Association for Computational Linguistics.