慣用句単位のコード翻訳を応用したプログラミング支援の提案

宮原 和也　山崎 徹郎　千葉 滋

本論文では，書いているプログラミング言語の慣用句がわからないときに，知っている言語でその慣用句を書くと目的の言語に翻訳する手法を提案する．この手法によって，慣れない言語でのプログラミングを支援する．本手法では，入力されたプログラムを LSTM で随時分析することで異なる言語で書かれたコード断片を発見し，GPT を用いて目的の言語に翻訳する．慣用句ごとにコード断片の長さが異なるので，LSTM に与えるトークン列の長さを変えながら，分析を行う．翻訳精度を高めるために，GPT への入力には間違って書かれたコード断片以外の周囲の部分も含める．どこまで周囲に含めるかの判断の基準には，例えば改行や括弧を用いる．

1　はじめに

プログラミングで扱う問題領域が広がったことで，問題領域ごとに別々のプログラミング言語を使う機会も増え，複数のプログラミング言語を使い分ける開発環境が一般的になった．そのような開発環境では，開発者が混乱して，間違った言語の慣用句を書いてしまうことがある．

我々は，間違った言語の慣用句の使用を自動的に正せるような，エディタの新しい入力支援機構を開発している．この入力支援機構は，今書いている本来のプログラミング言語のものではない慣用句の入力を認識して，それを正しい慣用句に翻訳し，それを置換候補として提示する．

今書いている本来の言語と異なる言語で書かれた慣用句を認識する処理は，プログラムを様々な長さのコード片に分割して，それぞれのコード片に対して今書いている言語かどうかの判定をし，それらの結果を突き合わせる方法で行う．コード片が今書いている言語かどうかの判定には，学習済みの LSTM[3] を用いる．慣用句の翻訳には，GPT [1] のような外部の機械学習モデルを用いるが，正しく翻訳できるように慣用句に対応するコード片だけでなく，コード片の周辺文脈も含めてモデルに送る．周辺文脈として送る部分をどのように決定するかは現在検討中であるが，改行や括弧の位置をもとに，範囲を決める方法を考えている．

異なる言語で書かれた慣用句を認識する処理では，まず字句解析をおこなってプログラムをトークン列に変換するが，本来の言語とは異なる言語で書かれた慣用句も含まれるため，本来の言語の字句解析器を用いても，うまくいくとは限らない．そのため，本来の言語が何言語であっても Python の字句解析器を用い，それによる字句解析の結果にヒューリスティックな処理を加えることでトークン化を行う．

提案する入力支援機構は現在開発中であるが，LSTM でコード片が特定の言語かどうかの判定ができることを実験で確かめた．実験は，Python，Java，C++の 3 つの言語を対象に，コード片がその言語で書かれたものなのか否か判定するタスクを教師つきで LSTM に学習させ，判定精度を調べた．どの言語

---

* Proposal for Programming Assistance Applying Idiomatic Code Translation.

Kazuya Miyahara,Tetsuro Yamazaki,Shigeru Chiba, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

に対しても，LSTM で高い精度の判定を行えることが確かめられた．

2 複数言語開発のための入力支援機能

近年，プログラミングで扱う問題領域が広がったが，問題領域ごとに主流のプログラミング言語が異なるため，一人のプログラマが複数の言語を交互に使用するという状況がしばしば見られる．例えば画像認識を行うスマホアプリの開発ではしばしば，画像認識部分は Python で，ユーザーインターフェース部分は Swift や Kotlin などの言語で，と複数の言語を使い分けて開発される．別の例として，複数の開発プロジェクトに参加しているがプロジェクトごとに使用する言語が異なるという状況も考えられる．

複数の言語を使い分ける開発環境では，開発者が混乱して，間違った言語の慣用句を書いてしまうことがある．ここでいう慣用句とは，典型的な処理のためのプログラムの書き方のことである．例えば Python と C 言語では繰り返し処理のために異なる慣用句を用いる．Python 言語では

```
for i in range(10):
```

から始まる `for` 文を用いるであろうし，C 言語では

```
for (int i = 0; i < 10; i++) {
```

から始まる `for` 文を用いるであろう．Python では繰り返し処理に内包表記を用いることもあり，他の言語では `foreach` メソッドのような高階関数で繰り返し処理を書くこともある．これらも慣用句である．複数の言語を使い分けていると，例えば Python のプログラムを書いているときに，C 言語の `for` 文を誤って書いてしまうことがある．同じ処理をするにも言語が異なれば多くの場合に慣用句が異なるので，このような誤りは起こりやすい．

このような慣用句の間違いを自動的に正せるような，エディタの入力支援を実現したい．例えば Python のプログラムを書いているときに開発者が C 言語の `for` 文を誤って書いてしまったら，エディタが自動的にそれを検出して正しい Python の `for` 文を置換候補として提示できるとよい．あるいは文字列を出力するために誤って C 言語風に

```
printf("Hello World!");
```

と書いてしまったら，正しい

```
print("Hello World!")
```

を置換候補として提示できるとよい．コード補完の機能と同様に，開発者が書いているコードを随時分析することで，間違った言語で書いた慣用句を開発者が書き始めたら，その慣用句に該当するコードを判定し，それを正しい慣用句に翻訳したものを置換候補として提示する入力支援が望まれる．

このようなエディタの入力支援機能は，既存の機械学習手法の単純な組み合わせでは実現できない．異なる言語の慣用句を正しい言語の慣用句に翻訳するタスクだと考えると，既存の機械学習によるプログラムの翻訳手法を用いればよいように思える．しかし，この入力支援機能の実現に必要なのは，慣用句の単純な翻訳ではない．入力途中の不完全な慣用句が，今書いているプログラムの言語とは異なる言語の慣用句であると認識し，今書いている言語の対応する慣用句に翻訳しなければならない．

3 慣用句単位でのコード翻訳

我々は前章の問題意識に基づき，プログラム・エディタのための新たな入力支援機構を開発している．この入力支援機構は，入力された（あるいは部分的に入力された）慣用句が今書いているプログラミング言語のものではない場合に，それを認識して正しい慣用句に翻訳し，それを置換候補として提示する．エディタの利用者が置換候補を選ぶだけで誤って入力したコード片を修正することができることを目指している．

図 1 に，開発している入力支援機構の概要を示す．図ではエディタの利用者が Python のプログラムを書いており，途中で `for` 文の一部を誤って C 言語風に書いてしまった場合を例として処理の流れを示している．まずエディタ上で入力されているプログラムの中から，今書いている本来のプログラミング言語とは異なる言語で書かれた慣用句を認識し，対応する部分のコード片を抽出する．この処理にはエディタの一部

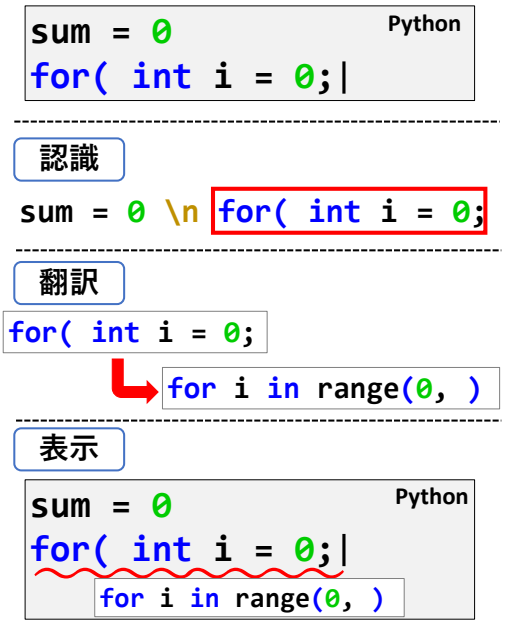

図 1　提案する入力支援機構

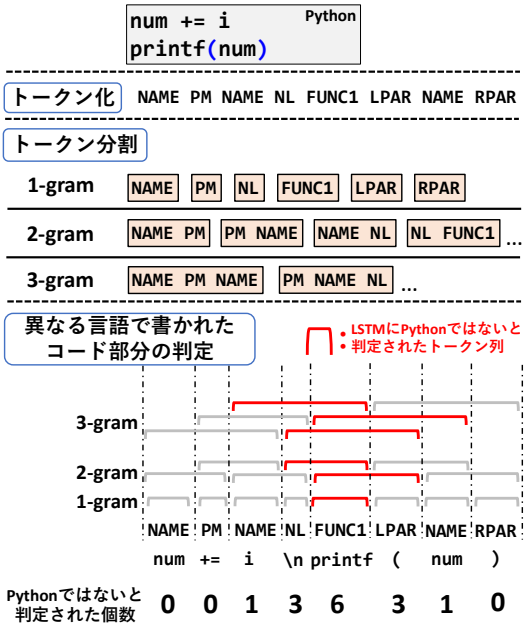

図 2　間違った慣用句の認識と抽出

として動く機械学習モデル LSTM [3] を用いる．次にコード片として抽出した慣用句を GPT [1] のような外部の機械学習モデルに送り，本来の正しい言語で書かれた慣用句に翻訳し，置換候補として提示する．この際，正しく翻訳できるように慣用句に対応するコード片だけでなく，コード片の周辺も含めて外部の機械学習モデルに送る．

3.1　間違って書かれた慣用句の認識

まず，プログラムの中から，本来のプログラミング言語とは異なる言語で書かれた慣用句を認識し，対応する部分のコード片を抽出する部分の解析手法について述べる．慣用句の長さは様々であるので，プログラムを様々な長さのコード片に分割して調べ，それらの結果を突き合わせることで目的の慣用句を見つける．

図 2 にこの解析手法の概要を示す．図は，Python で文字列を出力する関数 `print` を間違えて C 言語風に `printf` と書いてしまった場合を例として概要を示している．まず解析対象となるプログラムを字句解析によってトークンの列に変換する．次に与えられた n について，トークン列をトークンを単位として 1-gram から n-gram で分割する．図 2 では，最大 3-gram で分割した場合を示している．各 n-gram は，それぞれが今書いている言語のコード片であるか否かを，あらかじめ学習済の LSTM で判定する．最後に，各トークンについて，そのトークンを含む n-gram のうち，今書いている言語ではないと判定された個数を数える．この個数が判定基準以上であるトークンの連続する並びからなるコード片が，間違った言語で書かれた慣用句と認識される．例えば図 2 では，判定基準が 4 から 6 であれば間違った言語で書かれた慣用句は `printf` となる．判定基準が 2 または 3 であれば，

```
\n printf (
```

が間違った言語で書かれた慣用句となる．さらに判定基準が 1 であれば，

```
i \n printf ( num
```

が間違った言語で書かれた慣用句となる．

3.2 字句解析

間違った言語で書かれた慣用句を認識・抽出するために，我々が開発している手法ではまず字句解析をおこなってプログラムをトークン列に変換する．この字句解析は，今書いている本来のプログラミング言語の字句解析器を用いても，うまくいくとは限らない．本来の言語とは異なる言語で書かれた慣用句も字句解析できなければならないからである．

そこで我々は，本来の言語が何言語であってもPython の字句解析器を用い，それによる字句解析の結果にヒューリスティックな処理を加えることで，この問題に対処している．例えば本来の言語が C++ であり，プログラムの大半が C++ で書かれていても，Python の字句解析器を用いて解析をおこなう．Python の字句解析器を使うのは以下の理由からである．

- Python の一行コメントの記号として使われる `#` は，C++のディレクティブの接頭記号など，他言語での使用用途が限定的である．
- Python の複数行コメントの記号として使われる `"""` は，他の言語ではあまり使われない．
- C++や Java の複数行コメントの記号として使われる `/*` は Python では使われない．
- Python は一重引用符と二重引用符のどちらでも文字列を囲める．C++や Java では一重引用符で囲めるのは単一の文字なので，C++や Java の字句解析器では一重引用符に囲まれた Python の文字列を解析するときにエラーになる可能性がある．

Python の字句解析器による解析結果に加えるヒューリスティックな処理は次のようなものである．例えば Python の字句解析器は次のような行を字句解析すると

```
#include <iostream>
```

`#` に続く部分を全てコメントトークンに変えてしまうので，`#` を接頭記号とするディレクティブを含む上の行は次のようなトークン列に変換される．

```
HASH COMMENT
```

この行は本来は次のようなトークン列に変換されるのが望ましい．

```
HASH INCLUDE NAME
```

そこで，コメントトークンとしてトークン化されたものは，トークン化前の文字列の最初の部分を参照し，`include` や `define` といった文字列と一致した場合，`#` を除いた部分でトークン化し直す．

また，Python の字句解析器はプログラム中の `//` を整数除算の記号として認識してしまうため，`//` をコメントアウトの記号として使用する言語を字句解析した場合に，コメント部分がコメントトークンとしてトークン化されない．コメント部分もそれ以外の部分と同じようにトークン化されてしまう．例えば次の行は

```
// Written in C
```

次のようなトークン列に変換される．

```
DSLASH NAME NAME NAME
```

この行はコメントなので，次のようなトークン列に変換されるのが望ましい．

```
DSLASH COMMENT
```

これを実現するため，`//` から改行までのトークン列を分析し，それが整数除算としての `//` に続くものとして考えにくいトークン列だった場合，そのトークン列全体を 1 つのコメントトークンに変換する．

このようなヒューリスティックスでも機能するのは，字句解析器の結果得られるトークン列が機械学習モデルの入力として使われるだけだからである．Python のプログラムを書いているときに誤って C++ の慣用句を書いてしまったからといって，C++ の慣用句の部分を C++ の字句解析器と同様に解析する必要はない．適当なトークン列に分解することができ，その慣用句のプログラミング言語を機械学習モデルがそのトークン列から推定できればよい．

3.3 慣用句の翻訳

我々が開発している入力支援機構は，慣用句の翻訳に GPT などの外部の機械学習モデルを用いる．この支援機構は 3.1 節の方法で誤った言語で書かれた

慣用句を認識・抽出した後，それを GPT などを使って対応する正しい言語の慣用句に翻訳し，置換候補としてエディタの利用者に提示する．

モデルに送る翻訳元のコードは，3.1 節の方法で間違った言語であると認識した慣用句だけでなく，その周辺も含めて送る．GPT などの生成系モデルは，慣用句の周辺文脈を伝えないと，不適切な翻訳を行う可能性があるからである．例えば，Java の `Arrays.sort` メソッドで逆順ソートするコードを間違えて

```
Arrays.sort(src, reverse = true);
```

のように書いてしまったときに，間違った言語の慣用句として

```
reverse = true
```

が認識されたとする．このコード片のみをモデルに送ってしまった場合，モデルはこのコード片をメソッドの引数としてではなく，単純な代入文だと理解してしまい，

```
boolean reverse = true;
```

と翻訳してしまう．このコード片では適切な置換を行うことができない．周辺文脈としてメソッド全体をモデルに送った場合，モデルはメソッドの引数が間違っていると理解でき，

```
Arrays.sort(src, Collections.
    reverseOrder());
```

と目的としている翻訳結果が得られる．

周辺文脈として送る部分をどのように決定するかは現在検討中である．改行や括弧の位置をもとに，範囲を決める方法を考えている．

4 実験

3.1 節で述べた解析手法の重要部分として，実際に LSTM で与えられた慣用句のコード片が対象の言語で書かれたものか否かを判定できることを確かめる実験をおこなった．この実験では，Python，Java，C++の 3 つの言語で書かれたコード片がどの言語のものであるか判定するタスクを教師つきで LSTM に学習させ，学習後の判定制度を調べた．各言語ごとに別々の LSTM を用意し，入力されたコード片がそれぞれの言語のものであるか否かの二値分類で判定させた．LSTM が入力として受け取るコード片のトークン列の長さは最大 20-gram である．

各言語についてそれぞれ，GitHub[†1] から 8,000 ファイルのソースコード，競技プログラミングのデータセットである Project CodeNet[†2] から 2,000 ファイルのソースコードを集めた．それらのソースコードをトークン列化したあと，ファイル全体のトークン列を 1-gram から 20-gram で分割した．その中から 1,000,000 の様々な長さの n-gram を選択してデータセットとした．データセットには，LSTM が判定の対象としている言語の n-gram が 500,000 個と，それ以外の言語の n-gram が 500,000 個含まれた．また，データセットのうち 800,000 個の n-gram を訓練データ，200,000 個の n-gram を検証データとして使用した．LSTM が判定の対象としている言語と n-gram の言語が同じか違うかで，各 n-gram に正解ラベルを与えた．学習にあたり，各モデルは最大エポック数を 50 に設定したが，過学習を防ぐために，検証データの正答率が 2 連続で下がった場合，学習を終了した．

図 3 はそれぞれのモデルの学習データと検証データに対する正答率の変化を表している．どのモデルも検証データに対する正答率が 92 から 96% の間に収まり，比較的高い精度を示した．

5 関連研究

Matteo らは，RoBERTa モデルを用いて，様々な粒度のトークン列を対象にコード補完を行い，コード補完の限界と能力を探る実験をおこなっている [2]．トークン単位，for 文の完全な条件定義や関数呼び出しの引数定義などの構成要素単位，波括弧で囲まれたブロック単位の 3 つの粒度を対象としている．このようなコード補完は，入力済みの数語からコードの残りを予測するものであるため，コード片を翻訳する本研究のコード補完とは異なる．

Roziere らは，Python，Java，C++の 3 言語間をコード翻訳する TransCoder の開発した [4]．この研

†1 https://github.com

†2 https://developer.ibm.com/exchanges/data/all/project-codenet/

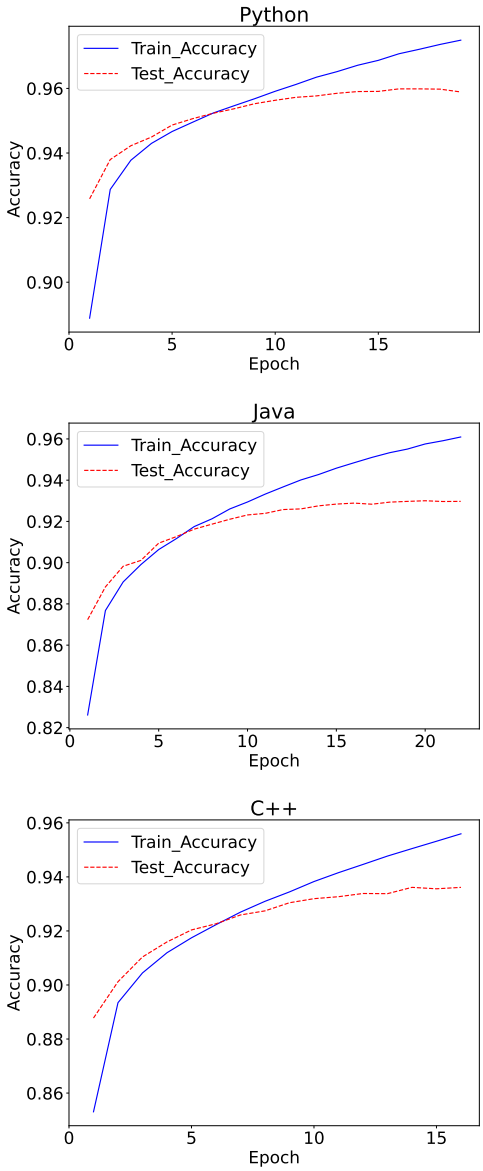

図 3　LSTM の正答率

究では，Transformer を伴うエンコーダとデコーダによって構成される，Attention 付きの sequence-to-sequence (seq2seq) モデルが使用され，30 から 70% の翻訳性能が得られている．しかし，この研究の翻訳対象はメソッドであり，我々の研究が必要としている短いコード片を別の言語に翻訳する処理には使えない．

6　まとめ

本論文では，入力されたプログラムを LSTM で随時分析することで本来とは異なる言語で書かれたコードの範囲を判定し，そのコードを GPT などの外部の機械学習モデルを用いて本来の言語に翻訳する入力支援を提案した．これにより，誤って本来とは異なる言語で書いた慣用句のようなコード片を，本来の正しいコード片に翻訳させることができる．提案した入力支援機構は現在開発中であるが，LSTM でコード片が特定の言語かどうかの判定ができることを実験で確かめた．Python，Java，C++の 3 つの言語に対して，LSTM で 92 から 96% の比較的高い正答率での判定を行えることが確かめられた．

今後の課題は，提案している入力支援機構の実装を進めて完成させることである．また完成した入力支援機構の有用性を実験により評価することも今後の課題である．

参 考 文 献

[1] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D.: Language Models are Few-Shot Learners, 2020.

[2] Ciniselli, M., Cooper, N., Pascarella, L., Poshyvanyk, D., Penta, M. D., and Bavota, G.: An Empirical Study on the Usage of BERT Models for Code Completion, *CoRR*, Vol. abs/2103.07115(2021).

[3] Hochreiter, S. and Schmidhuber, J.: Long Short-Term Memory, *Neural Computation*, Vol. 9, No. 8(1997), pp. 1735–1780.

[4] Roziere, B., Lachaux, M.-A., Chanussot, L., and Lample, G.: Unsupervised Translation of Programming Languages, *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Curran Associates Inc., 2020.