情報処理学会論文誌　プログラミング　Vol.17　No.4　1–9 (Aug. 2024)

LLMによる慣用句の言語間翻訳を用いたコード入力の支援

宮原 和也[1,a)]　山崎 徹郎[1,b)]　千葉 滋[1,2,c)]

受付日 2024年2月16日, 採録日 2024年5月14日

概要：複数言語を使い分けて開発する環境では，言語間の慣用句の違いによってプログラマが混乱することがあるため，異言語の慣用句の使用を認識して，その修正候補を提示する入力支援の開発が望まれる．本論文では，大規模言語モデル（LLM）を用いたコード翻訳に，異言語の慣用句の混入を判定する前処理と，翻訳前と翻訳後のコードから慣用句部分を抽出する後処理を組み合わせて，この入力支援を実現する手法を提案する．この提案手法は，LLM のサーバーへの問合せ回数を減らしつつ，コード中の慣用句部分だけの翻訳を可能にする．また本論文は，提案手法による異言語の慣用句の混入を判定する性能と，混入が判定されたときの翻訳結果の質とを，実験によって測定する．

キーワード：入力支援，LLM，コード翻訳，機械学習

Coding Assistance with Interlanguage Translation of Idioms by LLM

Kazuya Miyahara[1,a)]　Tetsuro Yamazaki[1,b)]　Shigeru Chiba[1,2,c)]

Received: February 16, 2024, Accepted: May 14, 2024

Abstract: When multiple programming languages are used for software development, programmers tend to be confused by differences in idiomatic expressions across languages. Such confusion will be avoided by coding assistance that suggests a correct expression when it recognizes that a programmer accidentally writes an idiomatic expression in a wrong foreign language. This paper proposes a new technique for developing such coding assistance that uses code translation by a Large Language Model (LLM). It combines an LLM with a pre-process and a post process. The pre-process detects whether an idiomatic expression in a wrong foregin language is included in given code. The post-process extracts an idiomatic expression from the original code and the translated code by an LLM. The proposed technique reduces the number of inquiries to the LLM server, and it enables code translation of only idiomatic expressions included in given code. This paper also presents the result of our experiments for measuring the proposed technique with respect to the performance for determining whether an idiomatic expression in a wrong foreign language is included in given code and the quality of the translation of the detected idiomatic expression.

Keywords: coding assistant, LLM, code translation, machine learning

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が一般的になった．そのような開発では，開発者が混乱して，今書いているプログラミング言語とは異なる間違った言語，つま

[1] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8654, Japan
[2] 情報処理学会プログラミング研究会
Information Processing Society of Japan Special Interest Group on Programming, Chiyoda, Tokyo 101–0062, Japan
[a)] chokazumiya9926@csg.ci.i.u-tokyo.ac.jp
[b)] yamazaki@csg.ci.i.u-tokyo.ac.jp
[c)] chiba@acm.org

り異言語，の慣用句をコード中に書いてしまうことがある．あるいは正しいものを思い出せず，間違った異言語でしかその慣用句を書けないことがある．そのためプログラムを入力中に，間違った異言語の慣用句の使用を自動的に検知し修正候補をユーザーに提示する，エディタの新しい入力支援が望まれる．

大規模言語モデル（LLM）のコード翻訳を用いれば，そのような入力支援が簡単に実現できるように思える．しかし，この入力支援はコード編集中に何度も呼び出す必要があるため，LLM のサーバーに負荷を与え，LLM の使用料が増大する可能性がある．また，LLM はコード内の異言語の慣用句を翻訳できるが，そのときに不必要な書き換えも合わせて行うことがある．

我々は，前述した入力支援を開発するために，LLM のコード翻訳に，異言語の慣用句が混入し翻訳を必要とするかを判定する前処理と，翻訳後のコードから慣用句を抽出する後処理を組み合わせる手法を提案する．異言語の慣用句が混入しているかを判定する前処理によって，異言語の慣用句が混入していないと判定した場合は，LLM によるコード翻訳を行わない．これにより，異言語の慣用句が混入している場合のみ LLM を呼び出すことになり，LLM のサーバーの負荷は軽減され，使用料も減少する．また，LLM によるコード翻訳を行った後に慣用句を抽出する後処理を加えることで，慣用句単位のコード翻訳を実現し，LLM による不必要なコードの書き換えを排除することができる．

提案手法の性能を測定するために，今用いている言語が Python であると仮定し，混入した異言語の慣用句に対する判定精度および修正精度を測定した．判定精度の測定では，入力支援システムがコードに異言語の慣用句が混入しているか否かの判定をどの程度の精度で行えるかを調べた．修正精度の測定では，提案手法が出力する修正候補が，正解として用意した修正候補にどれだけ近いかを調べた．

2. 複数言語開発環境向けの入力支援

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，ソフトウェア開発において，1 人のプログラマが複数の言語を使い分ける状況がしばしば見られる．たとえば，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．他にも，プログラマの人数が少ない組織で，JavaScript や HTML/CSS を用いて Web フロントの部分の開発をし，VBA や Google Apps Script を用いて業務ロジックの部分の開発をする，ということを 1 人のプログラマが任されることがある．

プログラミング言語が異なると，同じ処理をする場合でも慣用句が異なる．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全般のことで，制御構文の書き方やオブジェクトの定義・処理の表現の仕方，ライブラリ関数の呼び出し方などを指す．例として，配列の要素を標準出力にすべて出力する処理を Python，C 言語，JavaScript で書いたコードを，それぞれ Listing 1，Listing 2，Listing 3 に示す．繰り返し処理に Python と C 言語では異なる構文の `for` 文を用いており，JavaScript は `for` 文を用いずに高階関数である `foreach` メソッドを使用している．また標準出力に値を出力する関数の名前は，Python では `print`，C 言語では `printf`，JavaScript では `console.log` と言語によって異なる．他にも配列の定義の仕方も言語によって異なる．

Listing 1　配列の要素をすべて標準出力する Python コード

```
array = ["a", "b", "c"]
for element in array:
    print(element)
```

Listing 2　配列の要素をすべて標準出力する C 言語のコード

```
const char* array[] = {'a', 'b', 'c'};
int array_count
    = sizeof(array) / sizeof(array[0]);
for(int i=0; i<array_count; i++){
    printf(array[i]);
}
```

Listing 3　配列の要素をすべて標準出力する JavaScript のコード

```
const array = ['a', 'b', 'c'];
array.forEach((element) => console.log(element
    ));
```

Listing 4　Java の Stream-API を用いて配列の奇数を 2 倍にして取得するコードの例

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5,
    6, 7, 8, 9, 10);
List<Integer> result
  = numbers.stream().filter(n -> n % 2 != 0)
                .map(n -> n * 2)
                .collect(Collectors.toList())
```

この言語間の慣用句の違いによって，複数言語を使い分ける開発環境では，プログラマ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．プログラミングに慣れていないプログラマであれば，簡単な制御構造やよく使う関数の名前などでさえ混乱する可能性がある．また慣れているプログラマでも，複雑な慣用句を他の言語でどう書けばよいか分からなることがある．たとえば，Java の Stream-API を用いて配列の奇数を 2 倍にして取得する Listing 4 を他の言語でどのように書くかをすぐに思い出

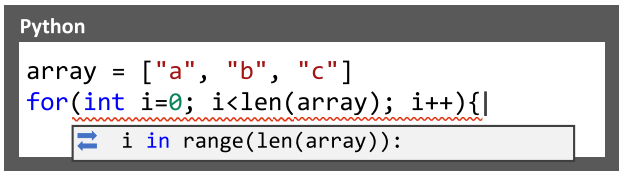

図 1　望ましい入力支援
Fig. 1　Desirable coding assistance.

すのは難しい．

この問題を解決するために，我々は，プログラマが異言語で慣用句を書いてしまった場合に，それを今書いている正しい言語に修正する置換候補をユーザーに即座に提示するような入力支援が必要だと考える．たとえば，**図 1** のように，Python でプログラムを書いているときにプログラマが 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` 文に直すための置換候補を提示できればよい．この入力支援によって，うっかり異言語で慣用句を書いてしまった場合にそれを修正することができる．また，今書いている言語での慣用句を忘れてしまった場合に，プログラマが知っている言語でその慣用句を書くことで，書きたかった慣用句を提示させることができる．

大規模言語モデル（LLM）は自然言語処理だけでなく，コード処理の面でも優れた性能を持ち，コード翻訳を非常に高い精度で行うことができる．たとえば，LLM の代表的なモデルである GPT-3.5 [1] の入力プロンプトに Python へ翻訳する指示文を含めて，以下の C 言語の `for` 文をモデルに送ると，

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

Python に翻訳された以下の `for` 文を得られる．

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

そのため，LLM を用いれば，前述した入力支援が簡単に実現できるように思える．しかし，実際には 2 つの課題が存在する．

1 つ目の課題は LLM のサーバーに大量のリクエストを送ってしまい，LLM のサーバーが高負荷になり，使用料が増大する可能性があることである．目標とする入力支援は，プログラマが異言語の慣用句を書いてしまったら，即座に翻訳と修正候補の提示を行うよう，プログラマが書いているコードを随時分析する必要がある．すなわち，プログラマがコード編集を行っている間，1 文字，1 単語，あるいは 1 文をタイプする度に何度も LLM を呼び出すことになる．一定量のコード片がタイプされた後にまとめて LLM を呼び出して置換候補を示す方式では，対話的な入力支援としては不十分であろう．外部の LLM のモデルを使用するとき，そのモデルの API を通して使用するが，現在公開されている LLM の API は入力プロンプトや出力文のトークン数，単語数単位で使用料が設定されている．そのため，大量のリクエストを送ると，使用料が増大してしまう．

Listing 5　LLM に送る翻訳元のプログラム

```
1 for i in range(20):
2     sum_val += math.sqrt(i)
3 printf(sum_val)
```

Listing 6　GPT-3.5 によって Listing 5 を翻訳したプログラム

```
1 import math
2 sum_val = 0
3 for i in range(20):
4     sum_val += math.sqrt(i)
5 print(sum_val)
```

2 つ目の課題は LLM がコード翻訳を行うときに，不必要な書き換えも同時に行ってしまうということである．LLM は渡されたコード内に変数の初期化式やライブラリのインポート文が書かれていないときに，それを勝手に補ってしまう．たとえば，Listing 5 を GPT-3.5 によって Python にコード翻訳すると，出力として Listing 6 が得られる．このコード翻訳によって，C 言語の `printf` 関数が Python の `print` 関数に修正されている．しかし，それとは別に，渡した入力プロンプトのコード内に書かれていない変数 `sum_val` への初期値の代入文や，ライブラリ `math` のインポート文が追加されている．目標とする入力支援は，異言語で書いてしまった慣用句だけを修正するものであり，それ以外の箇所ではユーザーの書いたコードに修正を加えないことが理想である．そのため，余計な代入文やインポート文を含めずに，異言語で書かれた慣用句だけを今書いている言語に修正してユーザーに提示したい．これは LLM だけを用いた手法では難しい．

3.　LLM を応用した入力支援システムの提案

我々は，LLM によるコード翻訳の処理の前後にローカルな処理を加えることで，目標とする入力支援システムを実現する手法を提案する．この手法では，LLM によるコード翻訳の前処理として，ローカルな LSTM（Long-short term memory）[6] を用いて，異言語の慣用句の混入を判定する処理を行う．また，LLM によるコード翻訳の後処理として，元のコードから異言語の慣用句，翻訳後のコードから修正された慣用句をそれぞれ抽出して対応づける処理を行う．

図 2 に提案する入力支援システムの実現方法を示す．まず，カーソル付近の数行のコードを取得し，そのコードに対して異言語の慣用句の混入を判定する前処理を行う．もし，異言語の慣用句が混入していないと判定された場合，次の処理に進まない．異言語の慣用句が混入してると判定された場合のみ，LLM によるコード翻訳の処理に進む．ま

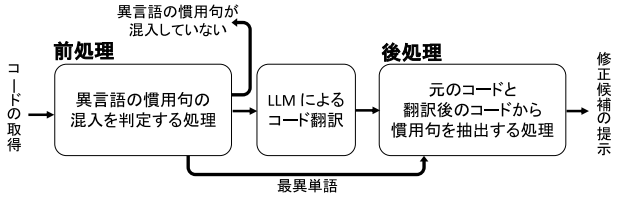

図 2　提案する入力支援システム
Fig. 2　Proposal coding assistance system.

た，この処理では，後処理に用いるためにコード内で「最も異言語として疑わしい単語」の特定も同時に行う．本論文ではこの単語を最異単語と呼ぶ．

LLM に送る翻訳元コードは，最異単語の前後数単語だけでなく，ユーザーから取得したカーソル周辺の数行のコードを含めたものである．LLM 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．たとえば，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，最異単語の前後数単語として

```
reverse = true
```

が認識されたとする．このコード片のみを LLM に送った場合，LLM はこのコード片をメソッドの引数ではなく，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も含めて LLM に送った場合，LLM 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．

LLM のコード翻訳の後に，元のコードから異言語の慣用句を，翻訳後のコードから修正された慣用句を抽出する後処理を行う．この処理では，前処理で求めた最異単語の位置情報を用いる．翻訳前と翻訳後でコードのどの部分が同じで，どの部分が異なるか差分を調べる．そして最異単語の位置を含む異なる部分だけを抽出する．抽出した部分の翻訳前のコードが異なる言語で書かれた慣用句で，翻訳後のコードが修正された慣用句であると判定する．これによって，エディタ上で異言語の慣用句の範囲にエラー破線を表示し，修正した慣用句をユーザーに提示するような入力支援を実現する．

LLM によるコード翻訳を行う前に，異言語の慣用句の混入を判定する処理を行うことで，LLM に送ってコード翻訳を行う必要があるかどうかを判定でき，LLM へリクエストを送る回数を抑えられる．それにより，サーバーへの負荷を軽減し，LLM の API 利用にかかる使用料も減少させることができる．また，前処理で特定した最異単語の情報を用いて，LLM がどこからどこまでのコード片を翻訳したかを判別する．異言語で書かれた慣用句と正しい言語にそれを翻訳したものだけを抽出し提示し，LLM による不必要な書き換えは提示から除外する．

3.1　トークン化

提案する入力支援システムの実現方法はトークン単位の解析をベースにしている．一般にトークンの定義はプログラミング言語ごとに決まっているため，単一の言語で書かれたコードであれば，その言語のトークンの定義に従ってトークン化すればよい．しかし今回解析の対象としているコード中には，複数言語のコード片が混在している可能性があるため，1 つの言語のトークンの定義によって全体のトークン化を行うことができない．

我々は，複数言語のコード片が混在している場合でも正しくトークン化できるように新しいトークン定義を考案した．トークンの定義のベースとして Python の字句解析器のトークンの定義を用いる．これに Python 以外の言語に対応できるように，以下の例外的なトークンの定義を加える．

- キーワード，標準ライブラリのメソッド名や関数名，出現回数の多い単語は，独立したトークンとする．
- コメント・トークンのうちディレクティブだと考えられるものは，ディレクティブ・トークンとする．
- // に続くトークン列が整数除算として考えにくい場合，そのトークン列はコメント・トークンとする．
- *と*\に囲まれた範囲はコメント・トークンとする．

Python の字句解析器のトークン定義をベースに用いる理由は，Python の字句解析器は扱える演算子のトークンの種類が多く，他の言語のコードをトークン化する場合でもエラーが少ないからである．また，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 のトークンの定義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．Python の字句解析器を用いれば，一重引用符に囲まれた文字列をその長さに関係なくトークン化することができる．

3.2　前処理：異言語の慣用句の混入を判定する処理

LLM によるコード翻訳を行う前に，プログラマが書いたコードに対して異言語の慣用句の混入を判定する前処理を行う．この処理によって，プログラマが書いたコードを LLM に送る必要があるかどうかを判定する．また，この処理では後処理で用いる最異単語の特定も同時に行う．

前処理の手順を説明する．まず，ユーザーから取得した

コードを前節の字句解析手法でトークン化したあと，様々な長さの n-gram に分解し，各 n-gram が今書いている言語であるかどうかを学習済の LSTM で判定する．n-gram とは，何らかの列から連続する n 個の部分列を抜き出したものの集合である．本手法では，字句解析によってコードから変換されたトークン列から，n を 2 から 12 に変えながらそれぞれの n について n-gram を抽出する．その n-gram それぞれすべてについて，事前学習した LSTM モデルを用いて「異言語である」か「異言語ではない」かの判定を行う．

次に，各 n-gram に対する LSTM の判定結果を突き合わせることで，コード中の各単語がどのくらい異言語として疑わしいかの値を算出する．本論文ではこの値を *diff* 値と呼ぶ．*diff* 値は最小値 0，最大値 1 の指標であり，数値が大きいほどそのトークンが異言語である可能性が高いことを示す．*diff* 値を求める計算式は，以下のようになる．

$$diff = \frac{1}{|N|} \sum_{n \in N} \sum_{gram \in G_n} \frac{gram_{wrong}}{|G_n|} \tag{1}$$

ここで，判定に用いる n-gram の長さの集合 $\{2, 3, \ldots, 12\}$ を N，*diff* 値を求めるトークンが含まれる長さ n の n-gram の集合を G_n，集合 G_n の元を *gram*，その *gram* が LSTM によって異言語であると判定されたかを $gram_{wrong}$ と表す．LSTM によって「異言語である」と判定されている場合 $gram_{wrong} = 1$，「異言語ではない」と判定されている場合 $gram_{wrong} = 0$ となる．また，集合 A の元の個数は $|A|$ と表す．

最後に，求めた *diff* 値によって異言語の慣用句の混入の判定を行う．最大の *diff* 値が 0.1 を超えている場合，異言語の慣用句が混入していると判定する．このとき，その最大の *diff* 値を持つ単語が最異単語となる．

3.3　後処理：コードから慣用句を抽出する処理

LLM によるコード翻訳を行った後，元のコードから異言語の慣用句，翻訳後のコードからそれを修正した慣用句をそれぞれ抽出する．この処理によって，LLM による不必要な書き換えを修正候補から排除する．

後処理の手順を説明する．まず，元のコードと翻訳コードの中で一致する単語の組合せを求める．任意の箇所に 1 単語を追加する操作と，任意の 1 単語を削除する操作を用いて，元のコードを翻訳後のコードに変えるときの，最も操作回数が少ない操作手順を求める．これには，レーベンシュタイン距離 [7] の最小値を求めるときに用いられる動的計画法と同様のアルゴリズムを用いる．この操作手順の中で 2 つの操作のどちらも加わっていなく，元のコードと翻訳後のコードの両方に存在する単語の組をすべて求める．

次に，一致する単語の組合せを用いて，元のコードと翻訳コードの中の一致範囲と不一致範囲を求める．一致する単語の組合せの組が連続で 3 組以上続くコード範囲を一致範囲とする．それ以外のコード範囲を不一致範囲とする．

最後に，元のコードから異言語の慣用句，翻訳後のコードから修正された慣用句を抽出する．元のコードの不一致範囲のうち，最異単語が含まれる範囲が異言語で書かれた慣用句と判定される．また，その異言語の慣用句の範囲に対応する，翻訳後のコードの不一致範囲が修正後の慣用句となる．この対応関係は，一致する単語の組合せによって求められる．

4.　実験

本論文で提案した手法を実装し，その性能を測定する実験を行った．本実験は，今ユーザーが書いている言語が Python であると仮定して行った．また，コード翻訳を行うための LLM には，GPT 3.5 Tubo [*1] を用い，以下の 2 つの実験を行った．

- 異言語の慣用句の混入を判定する性能の測定
- 混入された異言語の慣用句を抽出し，正しく翻訳する性能の測定

4.1　データセット

実験には，陽性データセットと陰性データセットの 2 種類のデータセットを用いた．陽性データセットとは，そのコード片の中の 1 つの慣用句だけが Java もしくは C++ で書かれているが，残りが Python で書かれているコード片を集めたものである．つまり異言語の慣用句が含まれているコード片を集めたものである．陽性データセットの各コード片には，Java もしくは C++ で書かれた慣用句部分を Python に直したコード片が正解データとして付随する．一方，陰性データセットとは，すべて Python で書かれたコード片を集めたものである．つまり異言語の慣用句が含まれて「いない」コード片を集めたものである．陰性データセットに含まれているコード片は 5 行ほどの長さである．

公開されているデータセットで，陽性データセットのようなものは我々の知る限り存在しない．そのため，我々はこれらのデータセットを作成した[*2]．まず，陽性データの中に含める異言語の慣用句を，長さや複雑さによって以下の 3 種類とした．

簡単な関数やメソッド

`print` や `append` など，言語によって関数名やメソッド名が異なるだけ，もしくは少し文法を変えるだけで良い，簡単に書きかえることができる慣用句．

制御構造

`for` 文や `if` 文，サブルーチンの定義などの制御構造．

[*1] https://platform.openai.com/docs/models
[*2] https://zenodo.org/doi/10.5281/zenodo.11110697 で公開されている．

複雑な慣用句

Java の Stream API など，翻訳のときに関数名やメソッド名を書きかえるだけではなく，文法を全体的に変える必要がある慣用句.

これらの慣用句の種類に属する 89 の慣用句を選び，各慣用句が書かれた Python のソースコードを GitHub [*3] もしくは Project CodeNet [*4] から探して，慣用句が書かれた行とその前後数行を抜き出した．もし，慣用句が書かれた Python のソースコードが見つからなければ，その慣用句が含まれる 5 行前後のコード片を作成した．これらのコード片の集合を正解データとして扱う．次に，正解データのそれぞれのコード片の中で，慣用句にあたる部分を Java もしくは C++ の慣用句に書き換えた．これらの作業により，各慣用句に対して，最大 20 の慣用句を含むコード片とその正解データのペアを作成した．全体ではペアを 859 個含む陽性データセットを作成した.

陰性データセットは，Github と Project CodeNet から無作為に選んだファイルの Python ソースコードそれぞれから，連続した 2 から 4 行ほどのコード片を無作為に抽出することで作成した．全体でコード片を 5,000 個含む陰性データセットを作成した.

4.2　異言語で書かれた慣用句の混入の判定

前処理が異言語の慣用句の混入を判定する性能を測定する実験を行った．陽性データセットに含まれるコード片に対して正しく「異言語の慣用句が混入している」と判定できた割合を示す *Recall* と，陰性データセットのコード片に対して正しく「異言語の慣用句が混入していない」と判定できた割合を示す *Specificity* の 2 つの評価指標を求めた.

事前に，前処理に用いる LSTM の教師付き学習を行った．この LSTM は入力された n-gram が Python が書かれたものか否かを判定する．この LSTM は PyTorch [*5] の LSTM を用いて実装した．学習用のデータセットを作成するために，Github と Project CodeNet から集めた Python，C++，Java の各言語のソースコード 12,000 ファイルを，1 から 20 の長さの n-gram に分割した．Python のソースコード由来の n-gram には，Python で書かれていることを表すラベルを付随した．C++ もしくは Java のソースコード由来の n-gram のうち，Python のソースコード由来の n-gram に含まれていないものには，異言語で書かれていることを表すラベルを付随した．LSTM の学習のデータセットには，それぞれのラベルが付随している n-gram を 6,570,090 個ずつ用いた．データセットの 8 割を学習用に，2 割をテスト用に使用し，入力された n-gram に対してそのラベルを当てるように LSTM の学習を行った．学習はテストデータに対する判定精度が収束するまで全部で 17 エポック行った.

[*3] https://github.com
[*4] https://developer.ibm.com/exchanges/data/all/project-codenet/
[*5] https://pytorch.org

表 1　異言語の慣用句の混入に対する判定精度
Table 1　Detection accuracy of inclusion of foreign idioms.

評価指標	値
Recall	0.75
Secificity	0.94

Listing 7　異言語の慣用句と同名の関数やメソッドが Python に存在する例

```
1 n=int(input())
2 nums=[]
3 for i in range(n):
4     nums.add(int(input()))
5 count=0
6 for i in range(n):
```

実験結果を**表 1** に示す．*Recall* が 0.75，*Specificity* が 0.94 となった．異言語の慣用句で使われるものと同名の関数やメソッドが Python に存在する場合に，誤って判定されることが多かった．たとえば，Listing 7 がその例である．このコード片の 4 行目にリスト `nums` に要素を追加するときに，Java などでリスト型に要素を追加する `add` メソッドが使われているが，Python ではリスト型に要素を追加するために `append` メソッドを用いるので，この `nums.add` は異言語の慣用句である．しかし，提案手法の LSTM はプログラムの意味を考慮せずに，前後数トークンの字句の情報だけで判定を行う．Python では `add` メソッドは `set` 型に要素を追加するメソッドであるので，この部分は異言語の慣用句ではないと判定されてしまうと考えられる.

4.3　異言語と判定された慣用句の翻訳

異言語で書かれていると判定された慣用句が提案手法によって正しく抽出され，LLM を用いて適切に翻訳されるかを測定する実験を行った．評価指標には BLEU Score [9] を用いた．陽性データセットのコード片のうち，前処理によって「異言語の慣用句が混入している」と判定されたコード片に対して，抽出した異言語の慣用句を LLM で翻訳された慣用句に置換したコード片を BLEU Score の翻訳文として用いる．そのデータに対応する正解データのコード片を BLEU Score の参照訳として用いる．LLM には Listing 8 をプロンプトとして与えた.

実験結果を**図 3** 示す．図は BLEU Score の度数分布を表す．翻訳文が参考訳と一致する場合，BLEU Score は 1 となる．このようなデータのペアは全体の 30%存在した．また，データのペアのうち 74%は，BLEU Score が 0.7 以上であった.

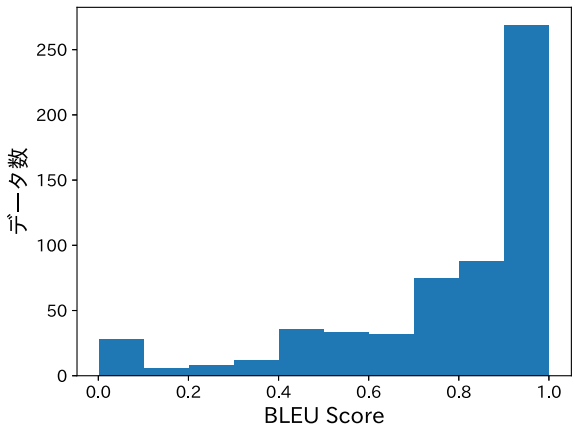

図 3　異言語の慣用句が混入していると判定されたコード片を提案手法で修正した後のコード片と，それに対応する正解のコード片の間で計算された BLEU Score の度数分布

Fig. 3　Distribution of the BLEU scores of code fragments after the correction by the proposed method when the fragments are detected as the ones including foreign idioms. The fragments after the correction are compared to the truly corrected ones to compute the BLUE scores.

Listing 8　LLM に与えたプロンプト

```
Translate this program to Python.Some part of the code
could be written in Java or C++. Follow the rule below.
- Adding lines to define variables and objects that was
not written in the original code should be written first
in the output code.
For example translation of

max = 10
printf(sum)

should be

sum = 0
max = 10
print(sum)

not

max = 10
sum = 0
print(sum)

- Don't write any natural language.
- Show the code part you changed, however don't use natural
language nor comments.
- If nothing to translate, return only "Correct".
```

5.　議論

すでに述べたように，エディタでプログラムをタイプしている最中に頻繁に LLM を呼び出す方式では，LLM の利用料が大きくなりすぎる恐れがある．極端な例として，1 文字タイプする度に LLM を呼び出すのでは，応答性の観点からも望ましくない．これを避けるためには，現在タイプされているコード片が異言語の慣用句である可能性が高そうな場合に限って LLM を呼び出して，置換候補を示すのがよい．

本論文で提案する手法では，エディタを動かしている手元のマシンで LSTM を用いて異言語の慣用句である可能性を判定し，その可能性が高い場合に LLM を呼び出す．しかしながら，異言語の慣用句であるか否かを判定する方法は LSTM によるものだけではない．たとえば入力されているプログラムに構文エラーや型エラーが見つかった場合，そのエラーは異言語の慣用句をタイプしたために発生したと考えて LLM を呼び出す方法も考えられる．たとえば前章の Listing 7 の例は，静的解析によって意味エラーとして検出できるかもしれない．ただ構文や型，意味エラーが即，異言語の慣用句をタイプしたことを示しているとは限らない．また LSTM より強力なモデルを使う方法も考えられる．たとえば手元のマシンでも動かせるパラメタ数が少ない LLaMa [13] のような低性能な LLM を使う方法もあるだろう．本論文で提案する LSTM を用いる方法と，これらの方法の比較は今後の研究課題である．また本論文で提案する方法とこれらの方法を組み合わせて用いて，より精度よく異言語の慣用句を検出する方法も考えられるかもしれない．

将来 GitHub Copilot などのコード補完の機能が強化されると，そもそも異言語による慣用句を誤って書くことがなくなるのではないか，という疑問もある．たとえば `for` とタイプした後に Python の `for` 文が補完候補として示されれば，誤って C 言語の `for` 文をタイプすることはないだろう．しかし，プログラマがこれからタイプしようとしているコードを完璧に予測できるコード補完が可能であるかは，まだ議論の余地があるだろう．また本論文で提案する異言語の慣用句を修正する支援は，自分が知っている言語の記法を用いて，今何をするコード片を書こうとしているかの意図をエディタ側に伝えられる，という利点もある．

前章の実験結果によれば，本論文で提案する手法は異言語の慣用句が混入している場合，*Recall* 0.75 で検出できる．混入していない場合は *Specificity* 0.94 で誤検出しない．見落としはあるものの，おおむね LLM を呼び出して異言語の慣用句の修正候補を示せるといえる．一方，修正候補の 30%は実験データの参考訳と完全に一致しているが，大きく異なる場合もある．この結果が実際のプログラミングにおいて，どの程度有用であるかの議論は今後の課題である．また用いる LLM が今後より強力なものになれば，修正候補の正しさも改善する可能性がある．

実験に用いたデータセットは，著者らが人工的に作ったもので，実際にプログラムをタイプする際の履歴から作成したものではない．またデータセット中のコード片は，完成したプログラムの断片を取り出して加工したもので，実際にタイプしている途中のコード片ではない．このため，

本論文が想定する利用用途の現実に近いデータセットで実験した場合，本論文で提案する手法による異言語の慣用句の検出精度や，修正候補の正しさが大きく変わる可能性がある．この点は本論文で示した実験の限界である．

6. 関連研究

LLM や大規模機械学習モデルを用いたコード補完に関する研究や，それを活用した入力支援システムはすでに存在する．また，LLM の開発が進む以前にも，機械学習モデルによってコード翻訳を試みる研究が行われている．

Github Copilot [4] は，書きかけのコードやコメントからユーザーがどのようなコードを書きたいのかをモデルが予測して続きのコードを補完する LLM を用いた入力支援ツールである．この入力支援ツールは主要なコードエディタや統合開発環境からも使用することができ，多くの人が利用している．GitHub Copilot のコード補完は，入力済みのコードやコメントから続きのコードを予測するものである．そのため，入力済みのコードを正しいコードに修正するコード入力支援を目的とする本論文とは異なる．Github Copilot Chat [*6]の機能を用いると，ユーザーが誤った言語による慣用句を含むコード片を指定して，それを正すように LLM に明示的に依頼することができる．本論文はそのようなコード片を自動的に発見して LLM に修正を依頼することを目的とする点が異なる．

Ciniselli らは，BERT [3] をベースとしたモデル RoBERTa [8] に対して，コード補完の限界と能力を探る実験を行った [2]．主に，粒度，特異性，抽象性 3 つの要素がコード補完の性能に与える影響を検証した．この研究も GitHub Copilot と同様に入力済みの数語からコードの残りを予測するコード補完に対するものであるため，入力済みのコードを正しいコードに修正するコード入力支援を目的とする本論文とは異なる．

Roziere らは，Attention [14] 付きの Encoder-Decoder モデル [12] を用いて，プログラミング言語間の関数単位でのコード翻訳を可能にする TransCoder を開発した [11]．TransCoder は，関数単位の比較的大きな粒度のコード片を翻訳対象としたモデルである．また，TransCoder に入力として与える翻訳の元となるコードは単一の言語で書かれていることが前提となっている．そのため，複数の言語が混在したコードの中で慣用句単位の翻訳を行う本研究のコード翻訳には LLM と同様にそのままでは利用できない．

本論文の手法は LLM によるプログラムの誤り修正（automatic program repair: APR）ととらえることもできる．これについては，たとえば Prenner らが GPT-3 による APR の性能を測定している [10]．機械学習モデルを使った APR は長く研究されており，初期の研究としては DeepFix [5] などがある．

[*6] https://docs.github.com/en/copilot/github-copilot-chat

7. まとめ

本論文では，異言語の慣用句を入力してしまった場合に，その修正候補をユーザーに提示する入力支援システムを開発するための手法を提案した．これは，LLM を用いたコード翻訳に，異言語の慣用句の混入を LSTM で判定する処理と，翻訳元コードと翻訳後コードから慣用句部分を抽出する処理を組み合わせる手法である．このような手法で入力支援を実現すれば，複数言語を使い分けるような開発で，言語間の慣用句の違いによってユーザーが起こす混乱に対して，ユーザーを支援することができる．

また提案手法を評価するために実験用のデータセットを作成した．そしてそのデータセットを用いて，与えられたコードが異言語の慣用句を含むか否かを判定する性能を実験で測定した．異言語の慣用句が混入しているときに正しく判定できている割合を示す *Recall* が 0.75，異言語の慣用句が混入していないときに正しく判定できている割合を示す *Specificity* が 0.94 となった．また，提案手法によって翻訳して得られる修正候補が，正解とする修正候補にどの程度近いかを BLEU Score を用いて実験で測定した．実験結果によれば BLEU Score が 1 となるデータは全体の 30%存在した．また，データ全体の 74%は，BLEU Score が 0.7 以上であった．

謝辞　本研究は JSPS 科研費 JP20H00578，JP24H00688 の助成を受けたものです．

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., et al.: 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) (online), available from ⟨https://arxiv.org/abs/2103.07115⟩.

[3] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2019).

[4] GitHub Inc.: GitHub Copilot, available from ⟨https://github.com/features/copilot⟩.

[5] Gupta, R., Pal, S., Kanade, A. and Shevade, S.: DeepFix: fixing common C language errors by deep learning, *Proc. 31st AAAI Conference on Artificial Intelligence*, *AAAI'17*, pp.1345–1351, AAAI Press (2017).

[6] Hochreiter, S. and Schmidhuber, J.: Long Short-Term Memory, *Neural Computation*, Vol.9, No.8, pp.1735–1780 (online), DOI: 10.1162/neco.1997.9.8.1735 (1997).

[7] Levenshtein, V.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals, *Soviet Physics Doklady*, Vol.10, p.707 (1966).

[8] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L. and Stoyanov, V.: RoBERTa: A Robustly Optimized BERT Pretrain-

ing Approach (2019).
[9] Papineni, K., Roukos, S., Ward, T. and Zhu, W.-J.: BLEU: A method for automatic evaluation of machine translation, *Proc. 40th Annual Meeting on Association for Computational Linguistics*, *ACL '02*, pp.311–318, Association for Computational Linguistics (online), DOI: 10.3115/1073083.1073135 (2002).
[10] Prenner, J.A., Babii, H. and Robbes, R.: Can OpenAI's codex fix bugs? an evaluation on QuixBugs, *Proc. 3rd International Workshop on Automated Program Repair*, *APR '22*, pp.69–75, Association for Computing Machinery (online), DOI: 10.1145/3524459.3527351 (2022).
[11] Roziere, B., Lachaux, M.-A., Chanussot, L. and Lample, G.: Unsupervised Translation of Programming Languages, *Proc. 34th International Conference on Neural Information Processing Systems*, *NIPS '20*, Curran Associates Inc. (2020).
[12] Sutskever, I., Vinyals, O. and Le, Q.V.: Sequence to Sequence Learning with Neural Networks, *CoRR*, Vol.abs/1409.3215 (2014) (online), available from ⟨http://arxiv.org/abs/1409.3215⟩.
[13] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E. and Lample, G.: LLaMA: Open and Efficient Foundation Language Models (2023).
[14] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I.: Attention Is All You Need, *CoRR*, Vol.abs/1706.03762 (2017) (online), available from ⟨http://arxiv.org/abs/1706.03762⟩.

宮原 和也

2022 年東京大学工学部化学システム工学科卒業．2024 年同大学大学院情報理工学系研究科創造情報学専攻修士課程修了．機械学習や生成 AI に興味を持つ．

山崎 徹郎　（正会員）

2017 年東京大学情報理工学系研究科創造情報学専攻修士課程修了．2021 年同専攻博士課程修了，博士（情報理工学）．同大学特任助教を経て，現在，同大学助教．メタプログラミングの応用，FFI 環境下でのガベージコレクションの研究に従事．

千葉 滋　（正会員）

1991 年東京大学理学部情報科学科卒業，1996 年同大学理学系研究科情報科学専攻より博士（理学）．2012 年東京大学情報理工学系研究科創造情報学専攻教授．プログラミング言語，システムソフトウェアの研究に従事．