一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

転移学習を用いたデコンパイラ歪み修正手法の汎用性の再評価
—複数のプログラミング言語を題材として—

田中　叶也†　　柗本　真佑†　　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{kyo-tank,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　バイナリコードから元のソースコードを復元する技術としてデコンパイラがある．デコンパイラが復元するソースコードには元のソースコードとの差異（歪み）が含まれる．我々の先行研究では事前学習済みモデルに対して転移学習を適用し，画一的に歪みを修正する手法を提案している．しかしながら，先行研究での評価実験は Java に限定されており，プログラミング言語に依らず歪み修正が可能という手法の汎用性を十分に評価できていない．そこで本研究では，デコンパイラ歪み修正手法の汎用性の確認を目的として，Python と C 言語を題材に歪み修正手法の再評価を行った．その結果，どちらの言語に対しても Java と同程度の割合で歪みを修正できることを確認した．一方で新たな歪みを発生させてしまう割合は言語ごとに差があることも確認した．
キーワード　デコンパイラ，転移学習，事前学習済みモデル，歪み

1. はじめに

コンパイラが生成したバイトコードやバイナリコードから，元のソースコードを復元する技術としてデコンパイラがある．デコンパイラはリバースエンジニアリング技術の一種であり，バイナリを対象としたマルウェアの挙動解析 [1] や，ソフトウェアの脆弱性解析 [2] などで活用される．C 言語や Java などの各種プログラミング言語に対して，様々なデコンパイラが存在する．C 言語では Ghidra が，Java では CFR や FernFlower などが広く知られており，当然ながら言語やデコンパイラによって復元性能は異なる [3].

我々の先行研究 [4] では，言語やデコンパイラに依存しない画一的なデコンパイラ歪み修正手法を提案している．歪みとはデコンパイル元ソースコードと，デコンパイル後ソースコードの差異のことであり，デコンパイラによる復元の誤りだといえる．先行研究では多数のソースコードを学習した事前学習済みモデルに対し，歪み修正というタスクに特化するような転移学習を適用する．この際の転移学習は，元ソースコードと復元ソースコードのペアの学習のことであり，復元ソースコードに含まれる誤字や構文誤りを修正する，一種の文法誤り訂正タスクであると見なせる．ソースコードペアを用いた転移学習というアイデアによって，プログラミング言語やデコンパイラの種類にとらわれず，画一的な歪み修正が可能である．

しかしながら，先行研究における評価実験は Java のみを対象としており，言語に依らず歪み修正が可能という手法の汎用性を十分に評価できていない．Java のコンパイル結果はバイトコードであり，デコンパイラによる復元性能は機械語を直接生成する言語と比較して高い傾向にあると考えられる．他方，C 言語のような機械語生成系の言語においても提案する歪み修正手法が効果的かは不明である．また発生する歪みの傾向は言語によって異なると考えられるため，他の言語に対しても評価実験を行うべきである．デコンパイラの活用先はマルウェア解析等のセキュリティ分野が主要であり，特に C 言語を対象とした歪み修正は高い需要があると考えられる．

本研究の目的は，先行研究で提案した歪み修正手法のプログラミング言語に対する汎用性の再評価である．そのために Python と C 言語を対象に歪み修正性能の評価実験を行う．評価実験の結果，Python と C 言語のどちらにおいても Java と同程度の割合で歪みを修正できることを確認した．また新たな歪みを発生させてしまう割合は言語ごとに差があることを確認した．特に C 言語ではその割合が高く，画一的な歪み修正を目指す上での課題であるといえる．

2. 準　　備

2.1 深層学習と事前学習済みモデル

深層学習とは多層のニューラルネットワークを用いてデータを処理する，機械学習の一種である．従来の機械学習では特徴量を手動で設計する必要があったが，深層学習では自動で特徴量を抽出できる. さらに近年では事前学習済みモデルが広く用いられている．事前学習済みモデルはあらかじめ大規模なデータセットで学習を行っているモデルであり，目的に応じてファインチューニング等の転移学習を適用することで汎用的にタスクが処理できる．代表的な事前学習済みモデルとして，BERT [5] や本研究で利用する CodeT5 [6] などがある．

2.2 デコンパイラと歪み

バイナリコードやバイトコードから元のソースコードを復元する技術としてデコンパイラが存在する．デコンパイラの課題として，復元したソースコードと元のソースコードの差

This article is a technical report without peer review, and its polished and/or extended version may be published eleswhere.
Copyright ©2025 by IEICE

異，すなわち歪みが挙げられる． 図 1 と 図 2 に Java のデコンパイラ CFR と C 言語のデコンパイラ Ghidra それぞれによって生成された歪みの実例を示す．

図 1(b) に示す Java デコンパイラの CFR による歪みについて説明する．識別子名に着目すると，numbers や occurence のような意味のある変数名は失われてしまっている．また，プログラム構造にも失われている箇所がある．元のソースコードでは番兵である-1 の要素を if 文を用いて発見しているが，デコンパイラによる復元コードでは for 文の条件式に組み込まれてしまい可読性が低下している．さらに，単項演算子 ++ は全て後置から前置に変換されている．ここで示した 図 1 の例ではソースコードのふるまいは変化していないが，歪みによって可読性が低下しているといえる．

図 2 に示す C 言語での歪みの実例では， 図 1 の Java での実例とは異なる歪みが発生している．具体的には，配列へのアクセスが numbers[i] という形式からポインタを用いた形に変換されており，配列の先頭アドレス param_1 を基準に演算したアドレスを参照している．ポインタの概念がない Java ではこの歪みは発生しない． このようにプログラミング言語によって歪みは異なるため，画一的な歪み修正は容易ではない．

```
int count(int[] numbers) {
  int occurence = 0;
  for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] == -1) // found sentinel
      break;
    occurence++;
  }
  return occurence;
}
```

(a) 元のソースコード

```
int count(int[] arrn) {
  int n = 0;
  for (int i = 0; i < arrn.length
               && arrn[i] != -1; ++i) {
    ++n;
  }
  return n;
}
```

識別子歪み
構造的歪み

(b) デコンパイラ CFR による復元コード

図 1: 歪みの例（Java）

```
int count(int *numbers, int size) {
  int occurrence = 0;
  for (int i = 0; i < size; i++) {
    if (numbers[i] == -1) // found sentinel
      break;
    occurrence++;
  }
  return occurence;
}
```

(a) 元のソースコード

```
int count(long param_1,int param_2){
  int local_10;
  int local_c;
  local_10 = 0;
  local_c = 0;
  while ((local_c < param_2 &&
         (*(int *)(param_1 +
          (long)local_c * 4) != -1))) {
    local_10 = local_10 + 1;
    local_c = local_c + 1;
  }
  return local_10;
}
```

(b) デコンパイラ Ghidra による復元コード

図 2: 歪みの例（C 言語）

2.3 先 行 研 究

先行研究では画一的な歪み修正を目的として，転移学習を用いた歪み修正手法を提案している [4]．歪みを一種の文法誤りとみなし，事前学習済みモデルに対して翻訳タスクとしてファインチューニングを適用することで，プログラミング言語やデコンパイラに依存しない画一的な歪み修正を実現している．

図 3 に転移学習を用いた歪み修正手法による歪み修正の流れを示す．大きく 3 ステップに分けて流れを説明する．Step1 では，学習に用いる元コード・復元コードのペアのデータセットを，任意のデータソースから得たソースコードを全てコンパイル・デコンパイルすることで作成する．次に Step2 で，Step1 で生成したデータセットを用いたファインチューニングにより歪み修正モデルを生成する．最後に Step3 では，テストデータから抽出した復元コードを Step2 で生成されたモデルに入力して歪み修正を試みる．

先行研究の課題として，Java 以外のプログラミング言語に対する歪み修正性能を確認できていない点がある．Java では識別子歪みの約 6 割と構造的歪みの約 9 割を修正できたが，他の言語においても同様に修正できるかは不明である．

3. 実験の目的

本研究における実験の目的は，転移学習を用いた歪み修正手法のプログラミング言語に対する汎用性の確認である．そのために，Python と C 言語を対象に歪み修正手法を適用し，言語間で修正性能を比較する．Python は学習が容易でかつドキュメントやツールが豊富であり，多くのマルウェア作成者が利用している [7]．そのため，セキュリティ解析で用いられる技術であるデコンパイルの需要も高い．また，Python は Java と構文特徴が大きく異なるため，汎用性の確認に有効であると考え採用した．C 言語は Java や Python と違い，コンパイル結果として機械語を生成する．バイトコードを生成する Java や Python と比べてコンパイル時に失われる情報が多く，よりデコンパイルが困難だと考えられる．したがって歪み修正の必要性も高いと考え採用した．

Python のデコンパイラは Uncompyle6，C 言語のデコンパイラは Ghidra を用いる．Uncompyle6 は広範囲の Python バージョンをサポートしており，精度も比較的高いため採用した [8]．Ghidra は頻繁にメンテナンスされており，オープンソースの C デコンパイラの中で精度が比較的高い [9] ため採用した．

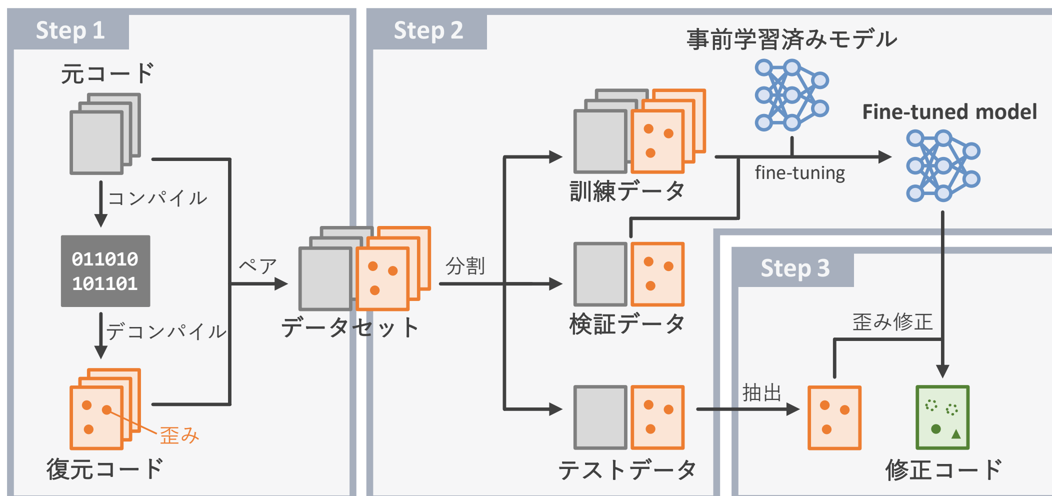

図 3: 転移学習を用いた歪み修正手法による歪み修正の流れ [4]

4. 実験設定

4.1 歪みの定義

本研究では歪みを復元コードと修正コードの抽象構文木（AST）の差と定義する．歪みは識別子歪みと構造的歪みの 2 種類を定義する．

識別子歪みはメソッド名や変数名などの識別子名の変更に関する歪みである．Java や C はコンパイル時に識別子名が失われるため，デコンパイラによる復元は困難である．デコンパイラによって復元されたソースコードでは意味のある識別子名が失われ，可読性の低下につながる．

構造的歪みはソースコードの構文の変化に関する歪みである．構造的歪みの発生による冗長な構造は可読性の低下につながり，最悪の場合プログラムのふるまいも変化してしまう．

4.2 歪みの検出方法

GumTree [10] というツールを用いて復元コードと修正コードの AST 差分を検出する．GumTree はプログラム間の AST 差分を検出し，変更内容を AST に対する 7 種類の操作に分類して出力するツールである．その 7 種類は match，update-node，insert-node，delete-node，insert-tree，deletetree，move-tree である．操作が update-node かつ操作対象が identifier である変更を識別子歪み，それ以外の変更は構造的歪みに分類する．

4.3 歪み修正性能の評価方法

GumTree によって歪みを検出した後，復元コードに含まれる歪み集合と修正コードに含まれる歪み集合の包含関係を分析する． 図 4 は A を復元コードに含まれる歪み集合，B を修正コードに含まれる歪み集合としたベン図である．$A \cap \bar{B}$ の集合は歪み修正によって取り除けた歪みの集合を表す．$A \cap B$ の集合は取り除けなかった歪みの集合を表す．$\bar{A} \cap B$ の集合は歪み修正によって新たに追加されてしまった歪みの集合を表す．

また，二つの指標を定義する．一つ目が除去率である．これは復元コードに含まれる歪みの集合に対して取り除くことのできた歪みの集合の割合とする．つまり除去率は高いほど良いと言える．二つ目は混入率である．これは修正コードに含まれる歪みの集合に対して新たに追加された歪みの集合の割合とする．つまり混入率は低いほど良いと言える．

4.4 データセット

実験には競技プログラミングの解答ソースコードが収集されたデータセットである ReCa [11] を用いる．ReCa には C，C++，Python，Java の 4 言語のソースコードが含まれている．本研究では Python と C のソースコードを対象とする．なお，最適化の強さを考慮するため，C 言語は gcc コンパイラの 2 つの最適化オプション（O0，O3）でそれぞれソースコードをコンパイルしてデータセットを作成する．以降，最適化オプション O0 の C 言語を C^{O0}，最適化オプション O3 の C 言語を C^{O3} と表す．デコンパイラによる復元コードを収集するにはバイナリコードもしくはバイトコードが必要なため，コンパイルできないソースコードは除外した．さらに，GPU メモリの不足によりファインチューニングが中断されることを防ぐため，2KB 以下のソースコードのみを対象とした．以上の条件で抽出した結果 Python は 15,753，C^{O0} は 13,820，C^{O3} は 10,024 のソースコードが収集できた．収集したソースコードを 80%，15%，5% で分割し，それぞれ訓練，検証，テストに用いる．

4.5 事前学習済みモデル

実験に用いる事前学習済みモデルは CodeT5 [6] とする．CodeT5 は CodeSearchNet [12] のデータセットで事前学習された Transformer ベースのモデルであり，コードの生成や変換，修正などマルチタスクに対応している．CodeT5 には複数のモデルサイズがある．本研究では可用性の確認を目的とするた

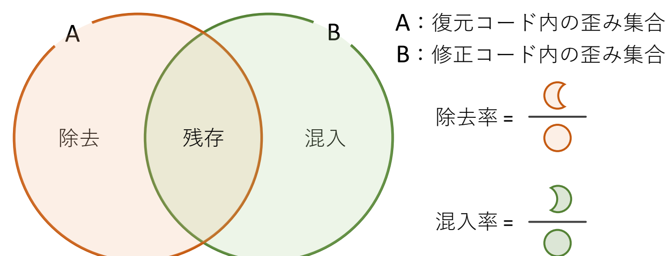

図 4: 修正前後の歪み集合の包含関係と評価指標

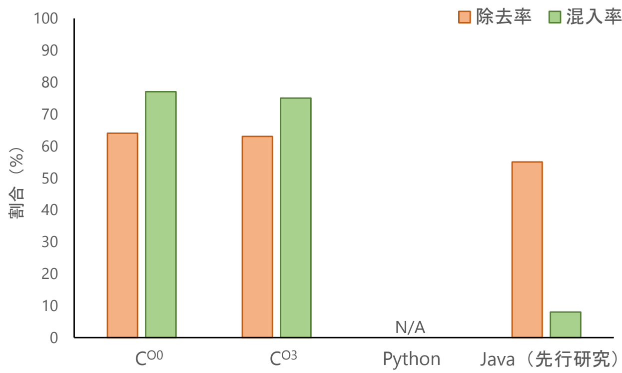

図 5: 識別子歪みに対する歪み修正性能

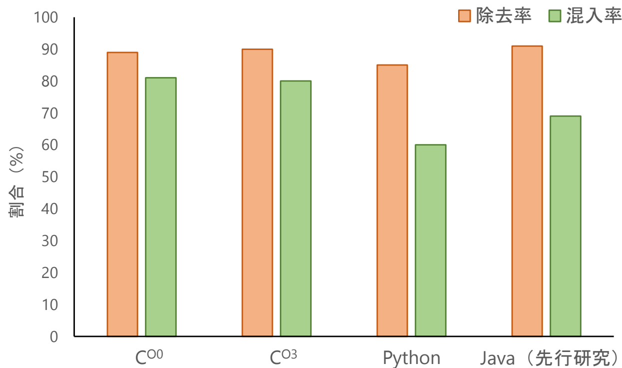

図 6: 構造的歪みに対する歪み修正性能

め，最もサイズの小さい CodeT5-small を利用した．

5. 実 験 結 果

5.1 歪み修正性能の言語間の比較

先行研究での実験対象である Java を含めた 3 つの言語間で歪み修正性能を比較する．ただし C 言語は 2 通りの最適化オプションでそれぞれ実験を行うため，歪み修正性能は 4 つの設定で比較する．また，Python はコンパイル過程で識別子名が失われないため，識別子歪みが発生しない．したがって Python は構造的歪みのみ分析する．以上を踏まえて 図 5 と 図 6 に 3 つの言語間における歪み修正性能を示す．

まず 図 5 の識別子歪みに着目する．C^{O3} と C^{O0} には除去率と混入率ともに違いがないため，最適化の強さが修正性能に与える影響は少ないとわかる．最適化の影響が少ない一つの要因として，最適化の強さに関わらず識別子名が失われることが考えられる．C 言語を Java と比較すると，明らかに修正性能が低い．Java のようなコンパイル結果がバイトコードの言語と比べ，C 言語のような機械語を生成する言語はコンパイル時に失われる情報が多いため復元しづらく，歪み修正も困難と推測できる．

次に 図 6 の構造的歪みに着目する．どの設定においても除去率が高く，最適化の強さやプログラミング言語によらず歪みが除去できていることがわかる．先行研究での考察に，プログラムの構文は同じ言語であれば開発者への依存度が低いため構造的歪みの除去率が高くなるというものがあった．その考察が他の言語においても当てはまると推測できる．また，Python は Java や C 言語と比較すると混入率が低い．その要因の一つとして Python のソースコードの簡潔さが考えられる．インデントでコードブロックを区切る Python は，Java や C 言語で発生しうる括弧の過不足が起こり得ない．

5.2 歪み修正の内訳

本節では C 言語と Python における歪み修正結果の具体的な内訳を，識別子歪みと構造的歪みそれぞれで分析する．5.1 節では歪み修正性能を割合で評価した．加えて歪みの修正内訳も確認し，各言語での歪み修正性能を具体的に分析する．

5.2.1 識別子歪みの修正内訳

除去できた識別子歪みと，誤って修正された場合の識別子名の内訳を分析する．まず除去できた識別子歪みの内訳を確認する． 表 1 に除去できた識別子歪みのうち，頻度の上位 5 件を示す．Python では識別子歪みが発生しないため，C 言語での修正内訳のみ示している．どちらの最適化オプションにおいても同様の傾向であり，ループ変数である `i` と `j` や，入力に用いられる `scanf` が含まれている．これらは競技プログラミングの解答で頻繁に出現する識別子であり，出現頻度の高い識別子の歪みの除去が多いとわかる．

次に誤って修正された場合の識別子名の内訳を確認する．ここでは 3 種類の変数を選び，それぞれ誤って修正された場合の名前と頻度を分析する．3 種類の変数は元のソースコードに含まれる変数のうち，修正コードにおいて歪んでいる頻度が高いかつ互いに特徴の異なるものを選択した． 表 2 と 表 3 に 3 種類の変数それぞれの誤って修正された場合の名前を，出現頻度の上位 6 件まで示す． 表 2， 表 3 では意味的に誤った修正が見受けられる．例えば 表 2(a) と 表 3(a) に示す変数 `n` は，ループ変数である `i` や `k` へ変換されるケースがある．題材が競技プログラミングの解答であるため，変数 `n` は入力値を格納する目的で用いられることが多い．したがって `i` や `k` への修正は意味的に誤っており，可読性の低下を招く．一方で意味的に類似した修正も確認できる．例えば 表 2(b) と 表 3(b) に示す変数 `i` は，`j` への修正が最も多い．`i` と `j` はどちらもループ変数としてよく用いられるため，`j` への修正は意味的に類似した修正結果であり，可読性への悪影響は小さい．

5.2.2 構造的歪みの修正内訳

次に構造的歪みの修正内訳を，GumTree が出力する AST 差分の内容を用いて分析する．ここでは 4.2 節で示した AST に対する 7 種類の操作のうち match を除いた 6 種類の操作（update-node，insert-node，delete-node，insert-tree，deletetree，move-tree）と，その操作対象となるノードの組み合わせを集計して構造的歪みの修正内訳を確認する． 表 4， 表 5， 表 6 に C 言語と Python それぞれにおいて除去できた構造的歪みと混入した構造的歪みの内訳を示す．

まず 表 4 に示す Python の構造的歪みの内訳に着目する．表 4(a) の除去できた歪みには他の言語にはない Python 特有の歪みを確認した．例えば頻度が 3 位の歪み（ins-node else:）と 4 位の歪み（ins-node else-clause）が該当する．これらの歪みは for ブロックの直後への不必要な else: の挿入が原因で発生して

表 1: 除去できた識別子歪みの内訳

(a) C^{O0}

識別子名	頻度
i	569
scanf	324
n	239
j	103
a	96

(b) C^{O3}

識別子名	頻度
i	448
scanf	251
n	186
a	83
j	61

表 2: 3 種類の変数の誤った修正名と頻度（C^{O0}）

(a) 変数 n

識別子名	頻度
a	56
t	31
i	25
k	22
m	20
num	16

(b) 変数 i

識別子名	頻度
j	77
a	18
n	14
m	14
x	13
k	12

(c) 変数 ans

識別子名	頻度
sum	15
k	14
s	9
count	8
min	6
max	6

表 3: 3 種類の変数の誤った修正名と頻度（C^{O3}）

(a) 変数 n

識別子名	頻度
a	40
i	24
t	18
m	16
k	16
num	12

(b) 変数 i

識別子名	頻度
j	53
a	17
m	14
x	13
k	10
l	7

(c) 変数 ans

識別子名	頻度
k	14
sum	12
s	7
count	6
m	4
M	3

おり，for-else 構文は Python 特有の構文であることから Python 特有の歪みといえる． 表 4(b) の混入した歪みのうち上位 3 件の歪みは，if 文の条件式が複数ある場合に and で一つにまとめるかネストして複数の if 文にするかの違いから発生する歪みであった．どちらで書くかは開発者の好みに依存するので，新たに混入する歪みとして多く発生したと考えられる．

続いて， 表 5 と 表 6 に示す C 言語の構造的歪みの内訳に着目する． 表 5(a) と 表 6(a) の除去できた歪みのうち頻度が 2 位の歪み（ins-tree expr-stmt）は C 言語特有の歪みであった．この歪みが発生する原因の一つに，スタック保護のためのソースコードの追加がある．実験で用いている gcc コンパイラは，スタックが破壊されていないか確認する処理をコンパイル時に挿入するため，その挿入された処理が歪みとなる．これは手動でメモリにアクセスできる C 言語特有の歪みと考えられる． 表 5(b) と 表 6(b) の混入した構造的歪みのうち上位 2 件はグローバル領域の不完全な修正が原因であった．例えばマクロ定義はコンパイル時に失われ，デコンパイラによる復元コードには含まれない．したがって修正コードでのマクロ定義に変数の過不足があると， 表 5(b) と 表 6(b) の上位 2 件に示

表 4: 構造的歪みの内訳（Python）

(a) 除去できた構造的歪み

操作	操作対象	頻度
ins-node	block	864
mov-tree	expr-stmt	859
ins-node	else:	757
ins-node	else-clause	755
mov-tree	block	277

(b) 混入した構造的歪み

操作	操作対象	頻度
mov-tree	expr-stmt	118
ins-node	parenthesized-expr	113
mov-tree	comparison-operator	100
ins-node	identifier	96
mov-tree	block	75

表 5: 構造的歪みの内訳（C^{O0}）

(a) 除去できた構造的歪み

操作	操作対象	頻度
ins-tree	declaration	4,656
ins-tree	expr-stmt	3,463
del-tree	expr-stmt	1,199
ins-node	;	935
ins-node	(	909

(b) 混入した構造的歪み

操作	操作対象	頻度
ins-node	identifier	789
del-node	identifier	664
ins-node	,	577
ins-tree	expr-stmt	575
mov-tree	expr-stmt	574

表 6: 構造的歪みの内訳（C^{O3}）

(a) 除去できた構造的歪み

操作	操作対象	頻度
ins-tree	declaration	3,299
ins-tree	expr-stmt	2,364
del-tree	expr-stmt	836
del-node	identifier	657
ins-tree	if-stmt	642

(b) 混入した構造的歪み

操作	操作対象	頻度
ins-node	identifier	525
del-node	identifier	434
ins-tree	expr-stmt	420
ins-node	,	420
mov-tree	expr-stmt	368

す歪みの混入となる．

C 言語と Python のどちらにおいても，言語特有の歪みの除去が見受けられたことから，歪み修正手法の汎用性が確認できる．ただ，C 言語では言語特有の歪みが混入する場合も見受けられた．除去できた歪みと比べて頻度は少ないものの，画一的な歪み修正を目指すうえでの課題といえる．

5.3　コンパイル可能率とテスト通過率

5.1 節と 5.2 節では各言語において歪みがどれほどの割合で修正できるかを確認し，その内訳を分析した．しかし，歪み修正がプログラムのふるまいに与える影響は確認できていない．本節ではプログラムのふるまいという観点から，歪み修正前後でコンパイル可能率とテスト通過率を確認し，言語間で比較する．

表 7 に先行研究の対象言語である Java も含めた 3 言語における，歪み修正前後のコンパイル可能率とテスト通過率を示す．C 言語では，復元コードのコンパイル可能率はどちらの最適化オプションにおいても 0% となっている．これは実験で用いているデコンパイラの Ghidra が，厳密には C の疑似コードを生成しており，再コンパイルが困難であるためだと考えられる．修正コードではどちらの最適化オプションにおいてもコンパイル可能率とテスト通過率が上昇している．他の言語

に比べると値はかなり低いが，歪み修正前は 0% であることを鑑みれば歪み修正手法は効果的といえる．

次に Python に着目すると，復元コードのコンパイル可能率が 87.6%，修正コードは 89.1% となっている．歪み修正によってわずかではあるがコンパイル可能率は上昇している．テスト通過率は復元コードが 45.1%，修正コードは 59.6% であり，歪み修正が有効だとわかる．この結果は Python と同じくコンパイル結果がバイトコードである Java よりも比較的良い．Python は Java と比べてソースコードが簡潔であり，さらに識別子歪みは発生しない．したがって歪み修正モデルが歪みと元ソースコードの対応関係をとりやすく，構文的に誤った修正が行われにくかったと考えられる．

表 7: コンパイル可能率とテスト通過率

		コンパイル可能率	テスト通過率
C^{O0}	復元	0.0% (0/691)	0.0% (0/691)
	修正	42.5% (294/691)	9.99% (69/691)
C^{O3}	復元	0.0% (0/525)	0.0% (0/525)
	修正	47.0% (247/525)	10.9% (57/525)
Python	復元	87.6% (686/783)	45.1% (353/783)
	修正	89.1% (698/783)	59.6% (467/783)
Java (先行研究)	復元	98.5% (848/861)	92.0% (792/861)
	修正	50.9% (438/861)	45.4% (391/861)

6. 考　　察

Java と比べて C 言語での識別子歪みの混入率が高い原因について考察する．C 言語ではコンパイル時にマクロ定義が失われるため，デコンパイラによる復元コードにはマクロ定義が含まれない．一方で修正コードではマクロ定義がある程度復元される場合があるため，その際に識別子名歪みが新たに混入する可能性がある．また，構造的歪みの複雑さが影響していることも考えられる．5.1 節の結果から，C 言語では構造的歪みの修正が他の言語より困難であることがわかる．実際に今回用いている C デコンパイラの Ghidra による復元コードを確認したところ，構文の歪みが複雑であった．例えば 5.2.2 節で述べたスタック保護のためのソースコードの追加が挙げられる．スタックが破壊されていないかの確認のために if 文が挿入され，さらにそれに伴って関数中で宣言される変数の数も増えることで構造的歪みが発生する．他にも変数を一文で複数宣言している箇所は個別の宣言文へとなる歪みや配列がポインタとなる歪み，for 文から while 文となる歪みなど様々な構造的歪みが絡み合う．構造が複雑に歪む場合，GumTree は識別子の対応関係が正しく取れず，更新ではなく削除と挿入によって差分を表現する傾向がある．そのため復元コードの識別子歪みが検出されず，本来は残存である修正コードの識別子歪みが混入扱いになったため混入率が高くなった可能性がある．

7. おわりに

本研究では転移学習を用いた歪み修正手法のプログラミング言語に対する汎用性を評価した．Python と C 言語を対象に実験を行った結果，どちらにおいても Java と同様の割合で歪みを除去でき，歪み修正手法の汎用性が確認できた．しかし，新たに追加される歪みの割合にはプログラミング言語によって差が見受けられ，特に C 言語では他の言語よりも高い割合で歪みが混入した．汎用性を高めるため，新たな歪みの混入を防ぐ方法を検討する必要がある．

今後の課題として，大規模言語モデルを用いた他の歪み修正手法 [13] との性能比較が挙げられる．また，近年はデコンパイルに特化した大規模言語モデルの開発が取り組まれている [14]．そのようなデコンパイラの歪みをどれほど修正できるか調査することで，より汎用的なデコンパイル技術への有効な知見が得られる可能性がある．

謝辞　本研究の一部は，JSPS 科研費（JP24H00692, JP21H04877，JP21K18302）による助成を受けた．

文　　献

[1] L. Durfina, J. Kroustek, and P. Zemek, “PsybOt malware: A step-by-step decompilation case study,” Working Conference on Reverse Engineering, pp.449–456, 2013.

[2] A. Di Federico, P. Fezzardi, and G. Agosta, “rev.ng: A Multi-Architecture Framework for Reverse Engineering and Vulnerability Discovery,” International Carnahan Conference on Security Technology, pp.1–5, 2018.

[3] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, “The Strengths and Behavioral Quirks of Java Bytecode Decompilers,” International Working Conference on Source Code Analysis and Manipulation, pp.92–102, 2019.

[4] 開地竜之介，柗本真佑，楠本真二，“大規模言語モデルを用いたデコンパイラ歪みの自動修正，” 情報処理学会論文誌，vol.65, no.11，pp.1576–1585，2025（出版予定）．

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” Conference of the North American Chapter of the Association for Computational Linguistics, pp.4171–4186, 2019.

[6] Y. Wang, W. Wang, S. Joty, and S.C. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” Conference on Empirical Methods in Natural Language Processing, pp.8696–8708, 2021.

[7] V. Koutsokostas and C. Patsakis, “Python and Malware: Developing Stealth and Evasive Malware Without Obfuscation,” International Conference on Security and Cryptography, pp.125–136, 2021.

[8] A. Ahad, C. Jung, A. Askar, D. Kim, T. Kim, and Y. Kwon, “PYFET: Forensically Equivalent Transformation for Python Binary Decompilation,” Symposium on Security and Privacy, pp.3296–3313, 2023.

[9] Z. Liu and S. Wang, “How Far We Have Come: Testing Decompilation Correctness of C Decompilers,” International Symposium on Software Testing and Analysis, pp.475–487, 2020.

[10] J. Falleri, F.M. andXavierBlanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” International Conference on Automated Software Engineering, pp.313–324, 2014.

[11] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, and L. Zhang, “Deep Learning Based Program Generation From Requirements Text: Are We There Yet?,” Transactions on Software Engineering, vol.48, no.4, pp.1268–1289, 2022.

[12] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search,” https://arxiv.org/abs/1909.09436, 2019.

[13] P. Hu, R. Liang, and K. Chen, “DeGPT: Optimizing Decompiler Output with LLM,” Network and Distributed System Security Symposium, pp.1–27, 2024.

[14] H. Tan, Q. Luo, and Y. Zhang, “LLM4Decompile: Decompiling Binary Code with Large Language Models,” https://arxiv.org/abs/2403.05286, 2024.