修士学位論文

題目

事前学習済みモデルを用いたデコンパイラの歪み修正手法の提案

指導教員

楠本 真二 教授

報告者

開地 竜之介

令和 6 年 2 月 1 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 5 年度 修士学位論文

事前学習済みモデルを用いたデコンパイラの歪み修正手法の提案

開地 竜之介

内容梗概

バイトコードからソースコードを復元するための手法としてデコンパイラが数多く提案されている．デコンパイラはプログラムの解析を目的として広く利用されている．しかし，コンパイル時に変数名など一部の情報が失われるため，完全な復元は難しく元のソースコードとの差異，すなわち歪みが発生してしまう．歪みはソースコードの可読性の低下のみならず，プログラムの振る舞いの変化にもつながる可能性がある．また，デコンパイラの種類によって異なる歪みが発生する．そこで本研究では，自然言語処理の分野で用いられる文法誤り訂正のアイデアを活用したデコンパイラの歪み修正手法を提案する．文法誤り訂正の中でも特に深層学習ベースの手法を用いることで，プログラミング言語やデコンパイラの種類によらずソースコード復元における歪み修正が可能となる．実験では，識別子歪みと構造的歪みの 2 種類の歪みに関して，提案手法でそれぞれどの程度修正が可能かを検証した．その結果，識別子歪みでは約 5 割を，構造的歪みでは約 9 割を修正できることを確認した．

主な用語

デコンパイラ，深層学習，事前学習済みモデル，ファインチューニング，歪み，文法誤り訂正

目次

1 はじめに 1

2 準備 3

2.1 文法誤り訂正 3

2.2 深層学習 3

2.3 デコンパイラと歪み 3

3 提案手法 6

3.1 概要 6

3.2 提案手法の流れ 6

4 実験設定 8

4.1 実験目的 8

4.2 歪みの定義 8

4.3 歪みの検出方法 9

4.4 歪み修正能力の評価方法 10

4.5 データセット 11

4.6 事前学習済みモデル 11

5 実験結果 13

5.1 歪み修正性能 13

5.2 具体的な修正内容 17

5.3 コンパイル可能率とテスト通過率 19

5.4 識別子に対する後処理 20

6 考察 23

7 妥当性の脅威 24

8 おわりに 25

謝辞 26

参考文献 27

図目次

1 デコンパイラによって発生した歪みの例 4
2 提案手法による歪み修正の流れ 6
3 GumTree を用いた歪み検出の例 9
4 2 つの歪み集合の包含関係 10
5 識別子歪みのベン図（CFR） 14
6 構造的歪みのベン図（CFR） 14
7 識別子歪みのベン図（Fernflower） 15
8 構造的歪みのベン図（Fernflower） 15
9 ReCa のある題材における歪み修正の実例 16
10 提案手法によって修正できた歪みの例 17
11 提案手法によって修正できなかった歪みの例 18
12 提案手法によって新たに追加された歪みの例 19
13 文字列置換によるコンパイルエラー修正の例 21

表目次

1 本実験におけるハイパーパラメータ 11
2 コンパイル可能率とテスト通過率 19
3 修正コードにおける主なコンパイルエラー 20
4 文字列置換による修正後のコンパイル可能数とテスト通過数 22

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から，その生成元となった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 [1]．デコンパイラは一種のリバースエンジニアリングツールであり，様々な目的で利用される．一つの活用方法は，ソースコードへアクセスできない環境下での対象の振る舞いの把握である．IntelliJ や Eclipse をはじめとする多くの IDE はデコンパイラを搭載しており，利用中のライブラリの具体的な処理内容を確認することが可能である．また，デコンパイラはバイナリを対象としたセキュリティ解析 [2] の重要な技術の一つであり，Android アプリに対するマルウェア検出手法 [3] [4] も提案されている．

デコンパイラが抱える一つの課題として，復元したソースコードに含まれる元のソースコードとの差異が挙げられる．本稿ではこの差異を歪みと呼ぶ．バイトコード等の中間言語には，元のソースコードに含まれていた識別子名は含まれていない．よって情報量の観点からも識別子名の完全な復元は不可能である [5]．識別子名はプログラム理解において重要な役割を占める [6] ことが知られており，その適切な復元はライブラリ理解という活用において重要な課題であるといえる．さらに識別子名のみならず，プログラム構造に関する歪みが発生することもある．これは中間言語と高水準言語の命令は 1 対 1 で対応するとは限らないことが原因であり，例えば，繰り返し処理を `for` 命令と `while` 命令のどちらに復元するべきかは一種の推論問題となる．場合によっては，この構造的な歪みによってプログラムの挙動が異なってしまうことも指摘されている [7]．

また，発生する歪みは適用するデコンパイラによって異なる．Java では CFR や Procyon，Jad，Fernflower など様々なデコンパイラが開発されており，各種デコンパイラによって異なる歪みが発生する [7]．デコンパイラの復元性能には差があり [8]，また復元可能な対象言語のバージョンも様々である．Java の場合，言語バージョン改定に伴って新たな言語仕様が追加されることも多く，その対応状況はデコンパイラによって異なる．これらの点から，最良なデコンパイラを確定することは難しく，画一的な歪み修正方法の実現も容易ではないといえる．

本研究の目的は，デコンパイラの種類に依存しないソースコードの歪み修正の実現である．そのために，深層学習ベースの文法誤り訂正手法（Grammatical Error Correction; GEC）のアイデアに基づく歪み修正手法を提案する．GEC とは，自然言語の文章を入力として，その文章に含まれる文法的な誤りを検出し修正する手法である．本稿ではソースコード中の歪みを一種の文法誤りだと見なすことで，GEC の転用を可能とする．提案手法の利点の一つは，メタアプリケーションである深層学習の活用により，特定のデコンパイラに特化しない歪み修正が可能という点にある．また，学習に用いるソースコードのペア（元コードと復元コード）を全自動で生成できるため，深層学習に必要な大量のデータを確保しやすいという利点も存在する．評価実験として，競技プログラミングのデータセット ReCa [9]

に対して提案手法の適用を試みた．その結果，CFR による復元コードにおいては約 55% の識別子歪みと約 91% の構造的歪みを，Fernflower による復元コードにおいては約 59% の識別子歪みと約 88% の構造的歪みを取り除くことができた．

2　準備

2.1　文法誤り訂正

文法誤り訂正（Grammatical Error Correction; GEC）とは，自然言語で記述された文章の中から文法的な誤りを自動で検出し修正する技術である．文法誤り訂正のアプローチは 3 種類に大別できる [10] [11]．一つはルールベースの手法であり，文法ルールやパターンに基づいて特定の文法エラーを検出し修正する．もう一つは統計的なアプローチである．統計的なアプローチでは，大規模な文法付きコーパスを使用して統計モデルをトレーニングし，文法エラーの検出と修正を行う．近年は，深層学習を利用した機械翻訳（Neural Machine Translation; NMT）ベースの手法 [12] [13] が主流となっている．本研究では，NMT ベースの文法誤り訂正のアプローチを参考にしたデコンパイラの歪み修正手法を提案する．

2.2　深層学習

深層学習とは，多層構造のニューラルネットワークを用いている機械学習の手法の一つである [14]．従来の機械学習では特徴量の設計を人間が行う必要があったが，深層学習では特徴量を自動で抽出できる．そのため，人の手で特徴量を指定することが困難な非構造化データを扱う画像認識 [15] や音声認識 [16] といった分野でよく活用される．また，機械翻訳 [17] [18] や対話システム [19] [20] といった自然言語処理の分野でも活用されるようになった．

さらに近年では，事前学習済みモデルが広く利用されるようになってきている．事前学習済みモデルとは，大規模なデータセットを用いて事前に学習を行っており汎用的なタスクに対応可能となっている深層学習モデルのことである．学習データの収集が困難であるタスクに対しても、少ないデータで事前学習済みモデルをファインチューニングすることにより高い精度のモデルを得ることが可能となる．代表的な事前学習済みモデルとしては，2018 年に Devlin らによって提案された BERT [21] や 2019 年に Liu らによって提案された RoBERTa [22]，2020 年に Brown らによって提案された GPT-3 [23]，また今回活用する Wang らによって提案された CodeT5 [24] などがある．

2.3　デコンパイラと歪み

バイトコードからソースコードを復元するための手法としてデコンパイラが存在する．デコンパイラが抱える一つの課題として，元のソースコードへと完全に復元することはできず，歪みが発生する点が挙げられる．

図 1 に 2 つのデコンパイラ CFR と JAD によって生成された実際の歪みの例を示す．左の元のソースコードと比べて，デコンパイル後のコードには様々な歪みが発生している．ここでは歪みを識別子

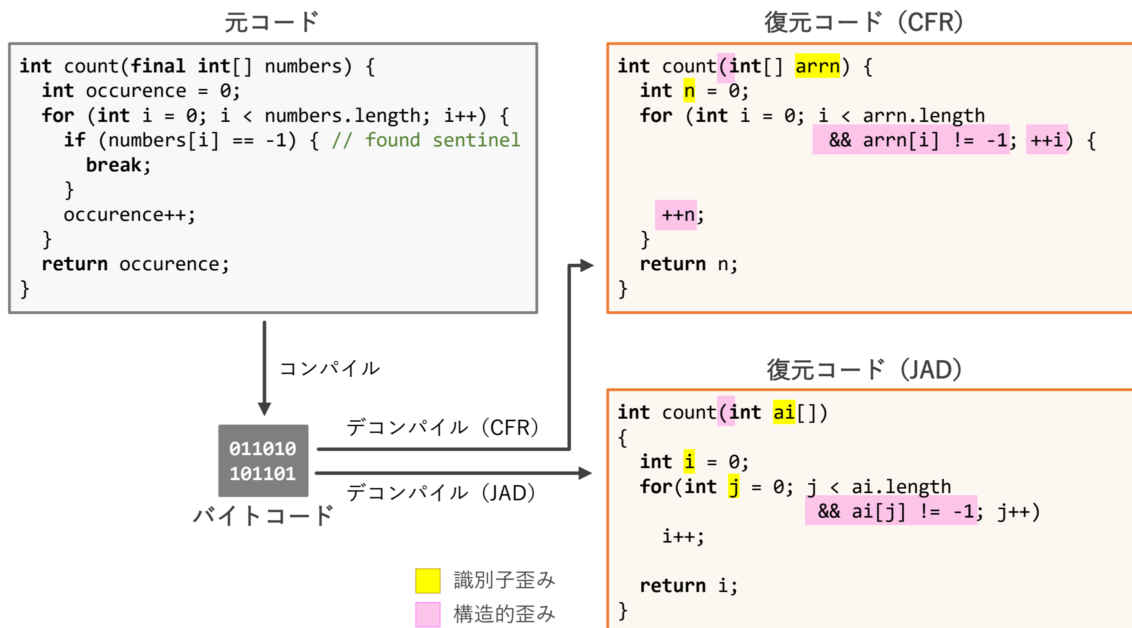

図 1　デコンパイラによって発生した歪みの例

名に関する歪みとプログラム構造に関する歪みの 2 種類に大別する．単純化のために，まずは右上の CFR による復元コードの歪みについて説明する．

まず識別子歪みに着目すると，ローカル変数名のほとんどは適切に復元できていない．唯一 `for` 文のインデックス `i` は歪んではいないものの，頻度を表す `occurence` や数値集合を表す `numbers` といった，プログラムの振る舞いの理解に役立つ情報のほとんどが失われている．

次に構造的歪みに着目する．第一に `final` 修飾子が省略されており，その値や参照が不変であるという開発者の意図や制約は読み取れない．`final` 修飾子やメソッド修飾子はコンパイラに対する制約の明言であり，コンパイル後のバイトコードには含まれない．よってその完全な復元は難しい．さらに，番兵（`-1` の要素）の発見条件は `for` 命令の繰り返し条件に統合されている．この統合に伴って，番兵発見条件の反転（`==`から`!=`への反転）という歪みも発生している．また軽微な歪みではあるものの，単項演算子の `++` は全て後置から前置に変換されている．全体的なソースコードそのものの振る舞いは元コードと等価ではあるものの，復元コードからは開発者が意図的に記述した狙いが読み取りにくくなっているといえる．

次に 2 つのデコンパイラ（CFR と JAD）の復元ソースコードを比較すると，異なる歪みを生成していることが分かる．特に識別子名は JAD は CFR とは異なる戦略で復元している点が読み取れる．識別子名はバイトコードには含まれていないため，変数の型やその用途などの情報から復元せざるを得ない．その復元の戦略が異なると全く異なる識別子歪みが発生する．デコンパイラ Fernflower の場合は，型すら無視して全て `var0`，`var1` のような連番の名前を付与する．構造的歪みという観点では，番兵条

件の for 文への統合という構造の大幅な歪みは CFR と共通しているが，単項演算子の歪みは発生していない．このようにデコンパイラによって発生する歪みは異なるため，ルールベース等の画一的な手法での歪みの修正は容易ではない．

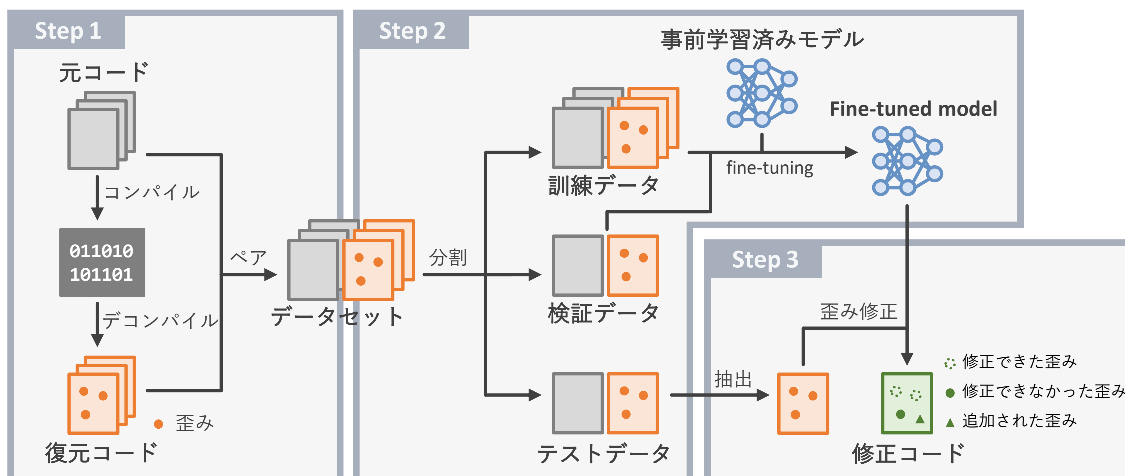

図 2　提案手法による歪み修正の流れ

3　提案手法

3.1　概要

本研究ではデコンパイラの種類に依存しない歪み修正の実現を目的として，GEC のアイデアに基づく歪み修正手法を提案する．ソースコード中の歪みを一種の構文的誤りだと見なすことで，ルールを用いない汎用的な歪み修正を実現する．具体的な手法としては，Transformer ベースの事前学習済みモデルである CodeT5 [24] を用い，歪みを含む・含まないソースコードのペアを用いて翻訳タスクとしてファインチューニングを適用する．生成されたモデルは歪みを含む入力コードから，歪みを含まないコードへ翻訳するようにソースコード変換を実施する．なお本手法は，特定の言語やデコンパイラに依存しない汎用的な手法である．学習対象となるデータセットがテキスト形式のソースコードペアであり，様々な言語やデコンパイラへの適用が可能である．

3.2　提案手法の流れ

図 2 に提案手法による歪み修正の流れを示す．提案手法は次の 3 つのステップから構成される．

Step 1: データセットの作成

このステップでは，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作成する．まず，GitHub や公開データセット等の任意のデータソースからソースコードの集合を得る．次に全てのソースコードをコンパイルしてバイトコード（言語によっては機械語）を生成する．さらに任意のデコンパイラを用いてバイトコードからソースコードの復元を試みる．この時点で復元コードには一定の歪みが

含まれている．最後に，元コードと復元コードをペアにしてデータセットを生成する．

Step 2: ファインチューニング

本ステップでは，次に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．Step 1 で生成したデータセットを訓練データ，検証データ，テストデータの 3 種に分割する．分割の割合は順に 0.80, 0.15，0.05 である．さらに訓練データと検証データを用いて，事前学習済みモデルに対するファインチューニングを実施する．この際の学習タスクは，ペアのデータセットを用いた翻訳タスクとなる．最終的に歪みを文法誤りのように見なし修正するモデルが得られる．

Step 3: 歪みの修正

最後に生成されたモデルを使って歪み修正を試みる．テストデータに含まれる復元コードをファインチューニングで生成したモデルに入力する．結果として一定の歪みが取り除かれた修正コードが得られる．

4　実験設定

4.1　実験目的

提案手法の歪み修正性能を確認するための実験を行う．復元コードに含まれていた識別子歪みと構造的歪みが，それぞれどの程度修正されたのかを検証する．まず初めに元コードと復元コード，元コードと修正コードとの AST の差分を検出し，復元コードと修正コードに含まれる歪みの集合をそれぞれ得る．その後，2 つの集合の包含関係を分析し，復元コードに含まれる歪みがどの程度修正できたのか検証する．

提案手法は，特定のプログラミング言語やデコンパイラに特化しない歪み修正が可能である．ただし本稿では，プログラミング言語として Java を対象に実験を行う．また，デコンパイラには CFR[*1]と Fernflower[*2]を利用する．これらのデコンパイラは，復元コードのコンパイル可能率が高く元コードとの AST の差分が小さい [7] など，Java のデコンパイラの中でも優れているため採用した．

4.2　歪みの定義

本稿では，元のソースコードとの AST の差を歪みと定義する．また，以下のように 2 種類の歪みを定義する．

識別子歪み

識別子歪みは識別子名の変更に関する歪みである．AST においてラベルが SimpleName か QualifiedName であるノードの更新を識別子歪みに分類する．図 1 において黄色のハイライトで示した歪みは識別子歪みの一つである．コンパイル時に識別子名の情報が失われることから，通常デコンパイラでは識別子名の復元は困難である．元コードで変数に意味のある名前がつけられていたとしても，デコンパイル後には名前が変わってしまうため可読性の低下につながる．

構造的歪み

構造的歪みとはソースコードの構文の変化に関する歪みのことである．識別子歪み以外の差分をすべて構造的歪みに分類する．図 1 において紹介した，番兵発見条件の反転は構造的歪みの一つである．識別子歪みと同様に可読性の低下につながる．また，最悪の場合プログラムの振る舞いにも影響を与えてしまう．

[*1] https://www.benf.org/other/cfr/

[*2] https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine

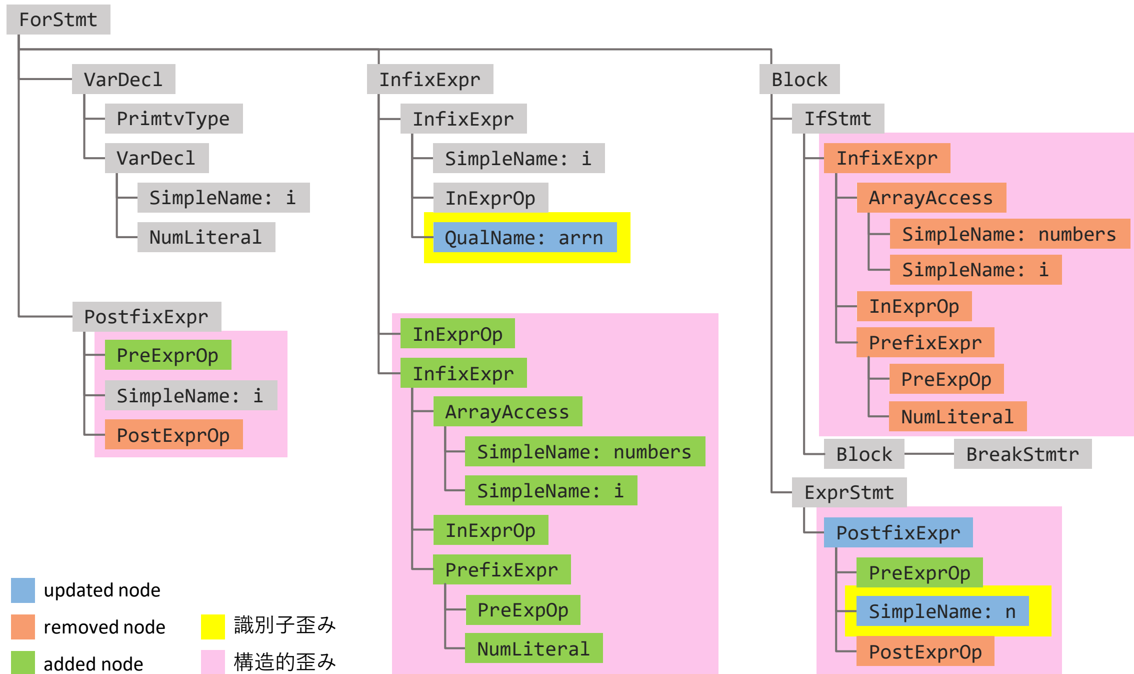

図 3　GumTree を用いた歪み検出の例

4.3　歪みの検出方法

復元コードと修正コードそれぞれに含まれる歪みを検出するために，GumTree [25] を使用して元コードとの AST の差分を検出する．GumTree はプログラムのバージョン間での変更箇所を特定し，それらの変更内容を 7 種類に分類可能なツールである．その 7 種類は，match, update-node, insert-node, delete-node, insert-tree, delete-tree, move-tree である．

図 3 に，GumTree を使って AST の差分を可視化した例を示す．これは，図 1 における元コードと CFR による復元コードのうち，それぞれの `for` 文に絞った例である．黄色のハイライトが識別子歪みを表しており，ピンク色のハイライトが構造的歪みを表している．この時，識別子歪みは単一ノードの更新に基づくため，黄色でハイライトされたノードの数と GumTree で実際に検出される歪みの数が一致する．一方で，構造的歪みではピンク色でハイライトされたうちの各ノードや部分木の変更が歪みに関与する．そのため，GumTree によって一つのブロック内から複数の歪みが検出される．このように，本稿の実験では構造的歪みを細かく検出している．今後は，ブロック単位で構造的歪みを検出した場合の実験を検討している．

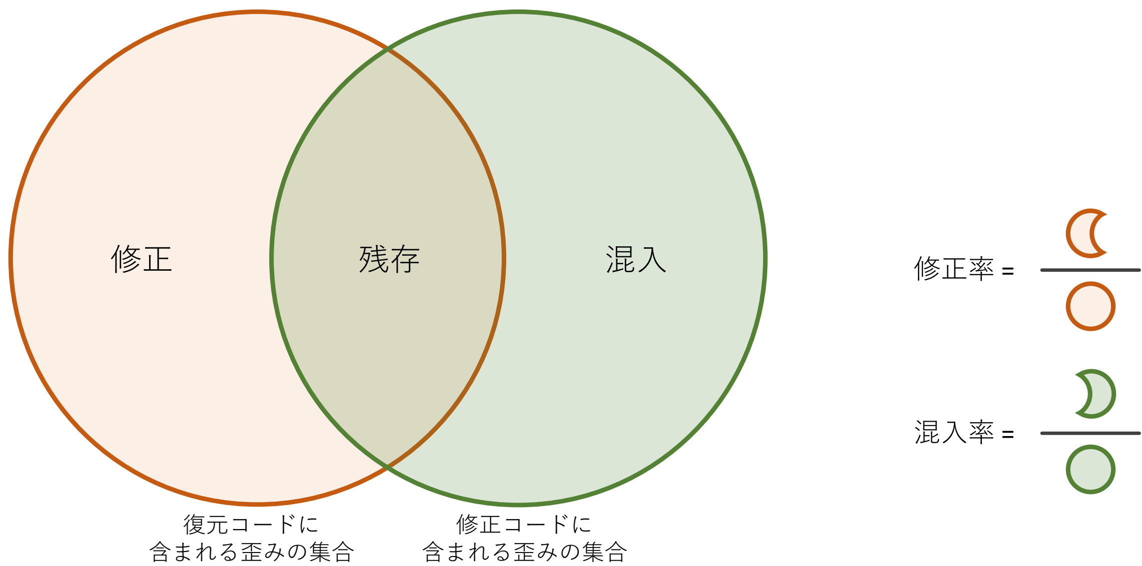

図 4　2 つの歪み集合の包含関係

4.4　歪み修正能力の評価方法

GumTree によって歪みの検出を行った後に，復元コードと修正コードそれぞれに含まれる歪みの集合の包含関係を分析し，提案手法によってどの程度歪みを修正できたかを評価する．

図 4 に示すベン図は復元コードに含まれる歪みの集合 A と修正コードに含まれる歪みの集合 B の包含関係を表すベン図である．このベン図において $A \cap \bar{B}$ の集合は，復元コードには含まれて修正コードには含まれない歪みの集合である．すなわち，提案手法による修正で取り除くことができた歪みといえる．同様にして考えると，$A \cap B$ の集合は提案手法では取り除けなかった歪みの集合，$\bar{A} \cap B$ の集合は提案手法により新たに追加されてしまった歪みの集合であるとわかる．

以上のように，復元コードと修正コードそれぞれに含まれる歪みの集合の包含関係を考えることで，提案手法によってどの程度歪みを修正できたかを評価する．

また，2 つの評価指標を定義した．一つ目が，修正率である．修正率は復元コードに含まれていた歪みのうち提案手法によって修正することができた歪みの割合である．そのため，修正率が高いほど良い結果であるといえる．二つ目が，混入率である．混入率は修正コードに含まれる歪みのうち，提案手法による修正で新たに追加されてしまった歪みの割合である．したがって，混入率が低いほど良い結果であるといえる．

4.5　データセット

本稿の実験では，競技プログラミングの解答として提出されたソースコードを収集したデータセットである ReCa [9] を用いる．ReCa には，C，C++，Python，Java の 4 種類のプログラミング言語のソースコードが含まれているが，今回はその中でも比較的容易にデコンパイル可能である Java だけを対象とした．また，復元コードを取得するためにはコンパイラによって生成されたバイトコードが必要である．そのため，コンパイル可能であるソースコードのみを抽出した．さらに，その中でもファイルサイズが 2,000 バイト以下のコードのみを抽出して実験対象とした．当初，ファイルサイズによる選別は行わずに実験を行った結果，メモリの問題が発生しモデルのファインチューニングが途中で止まってしまった．そのため，メモリの問題を引き起こさないこととファインチューニングに用いるデータ数が十分であることの両方を考慮し，最終的に 2,000 バイト以下のソースコードを対象とすることに決定した．

以上の条件を満たすコードを抽出した結果，最終的に 17,220 のソースコードが集まった．これを 0.80:0.15:0.05 の比率で分割し，それぞれ訓練データ，検証データ，テストデータとして実験に使用した．

4.6　事前学習済みモデル

本稿の実験で利用する事前学習済みモデルは，Wang らによって提案された CodeT5 [24] である．CodeT5 は，CodeSearchNet [26] のデータセットで事前学習された Transformer [27] 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 には異なるサイズのモデルが複数ある．しかし，サイズの大きいモデルはメモリの問題から利用できなかったため，今回は最もサイズの小さい CodeT5-small を利用した．

本実験で CodeT5-small を利用する際のハイパーパラメータは表 1 の通りである．ほとんどのパラ

表 1　本実験におけるハイパーパラメータ

ハイパーパラメータ	数値
フィードフォワード次元	2,048
埋め込みサイズ	512
エポック数	20
バッチサイズ	16
レイヤ数	6
ヘッド数	8

メータをデフォルトのままの数値で利用した．エポック数は，損失関数の推移を確認して値が収束し始めた回数を適用した．バッチサイズに関しては，メモリエラーが発生しない最も大きいサイズを選択した．これらのパラメータのもとファインチューニングを行った結果，要した時間はおよそ 1 時間 30 分であった．

5　実験結果

5.1　歪み修正性能

図 5 と図 6 に CFR を使用した場合の，図 7 と図 8 に Fernflower を使用した場合の識別子歪みと構造的歪みそれぞれに対する修正結果を示す．各円の縁から延びる数値は，復元コードと修正コードそれぞれにおいてテストデータとして用いた全ソースコード（861 個）に含まれる歪みの総計を表している．

初めに図 5 に着目すると，CFR による復元コードに含まれる識別子歪みの合計は 31,835 個であった．このうち，提案手法により取り除くことができたのは 17,503 個であり，約 55% の歪み修正に成功した．一方で，提案手法によって，新たに 1,210 個の歪みが追加されてしまっている．これは，修正コードに含まれる識別子歪みのうち 8% と比較的少数であった．次に，図 6 から CFR による復元コードに含まれる構造的歪みの合計は 50,645 個とわかる．このうち，提案手法により取り除くことができたのは 46,219 個であり，約 91% と大部分の歪みを修正できた．一方で，新たに追加された構造的歪みは 9,999 個で修正コードに含まれる歪みのうち約 69% となり，識別子歪みの場合よりも増加した．

次に図 7 と図 8 に着目すると，デコンパイラに Fernflower を使用した場合も 2 つの歪みそれぞれで CFR の場合と似た傾向の結果となっており，どちらのデコンパイラで復元されたコードに対しても同様の精度で歪み修正できることを確認した．

図 9 に，提案手法による歪み修正の実例を示す．ただし，修正性能が分かりやすくなるようにソースコードの一部を省略している．この実例において一番に着目したいのはピンクのハイライトがなされている構造的歪みである．まず初めに，元コードでは `boolean` 型の変数 `f` をフラグとしており，これによって最終的な出力が決定される．一方で，復元コードでは `int` 型の変数 `n` の値をフラグ代わりにしているが，この変数の役割を直感的に理解することは難しく，可読性が低下している．さらに，`for` 文の中の `if` 文の条件式が反転したことにより，人間が普通は書かないようなコードになっている．それはまさしく可読性の低下の原因である．それに対して，修正コードではフラグとして `boolean` 型の変数を使っており，その役割を明確にしている．また，`if` 文に関してもより理解のしやすい元コードの書き方へ修正しており，可読性が高くなっていると考える．

また，黄色のハイライトで示される識別子歪みに着目すると，復元コードと修正コードで歪みの数に差がないためうまく修正できていないように見える．しかし，修正コードでは配列が `arr`，フラグが `flag` というようにプログラム中での役割が直感的に理解可能な名前に変化している．確かに，元コードと同じ名前への修正はできていないため今回の定義においては修正失敗であるが，可読性の観点では修正コードの方が優れていると考える．この点は，提案手法の強みの一つである．

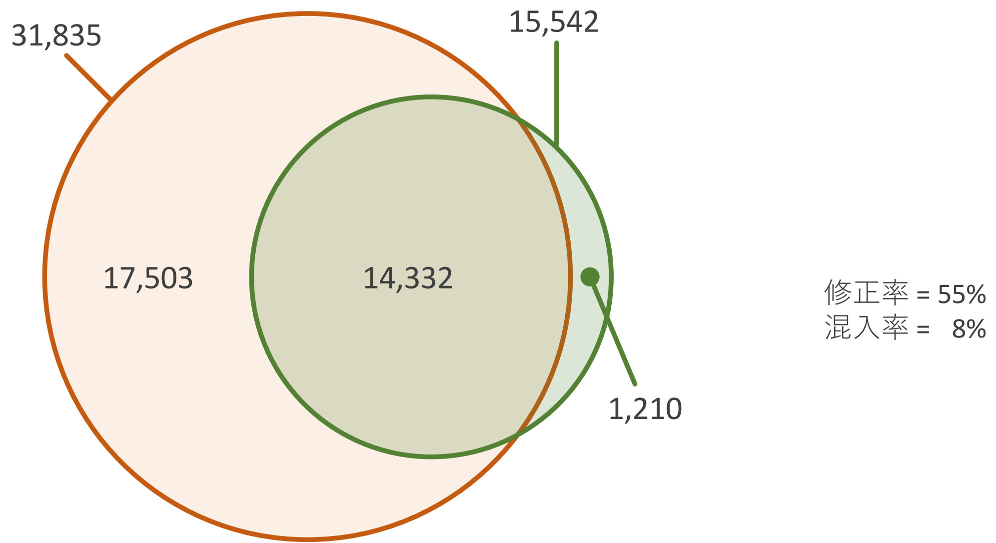

図 5　識別子歪みのベン図（CFR）

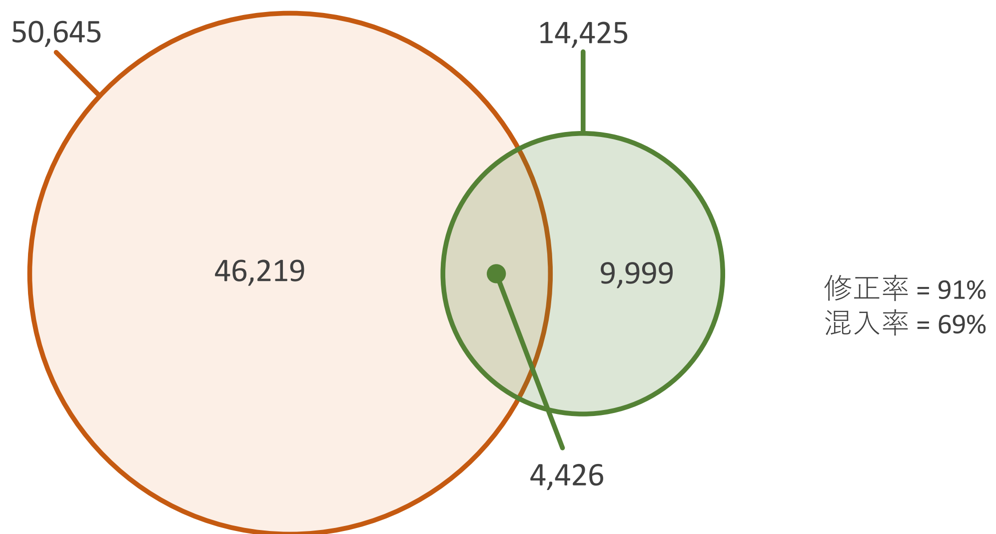

図 6　構造的歪みのベン図（CFR）

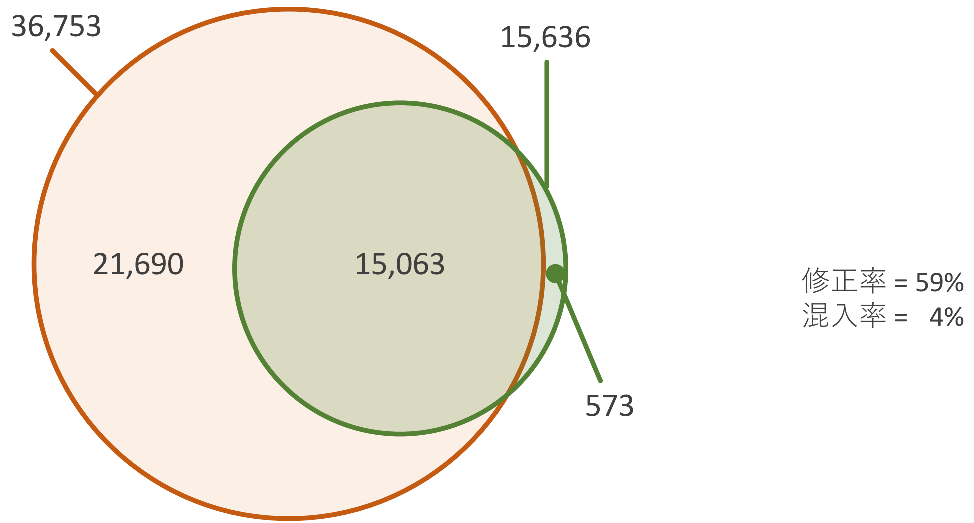

図 7　識別子歪みのベン図（Fernflower）

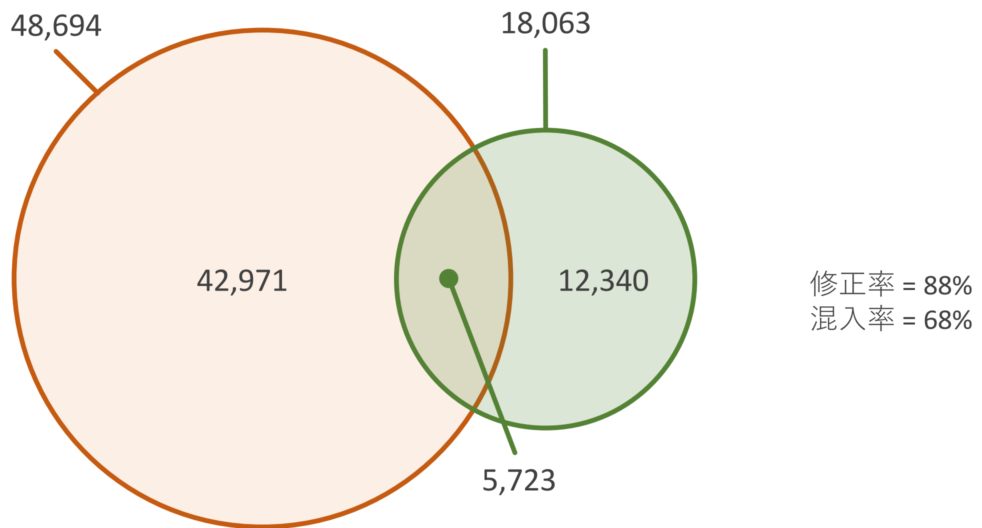

図 8　構造的歪みのベン図（Fernflower）

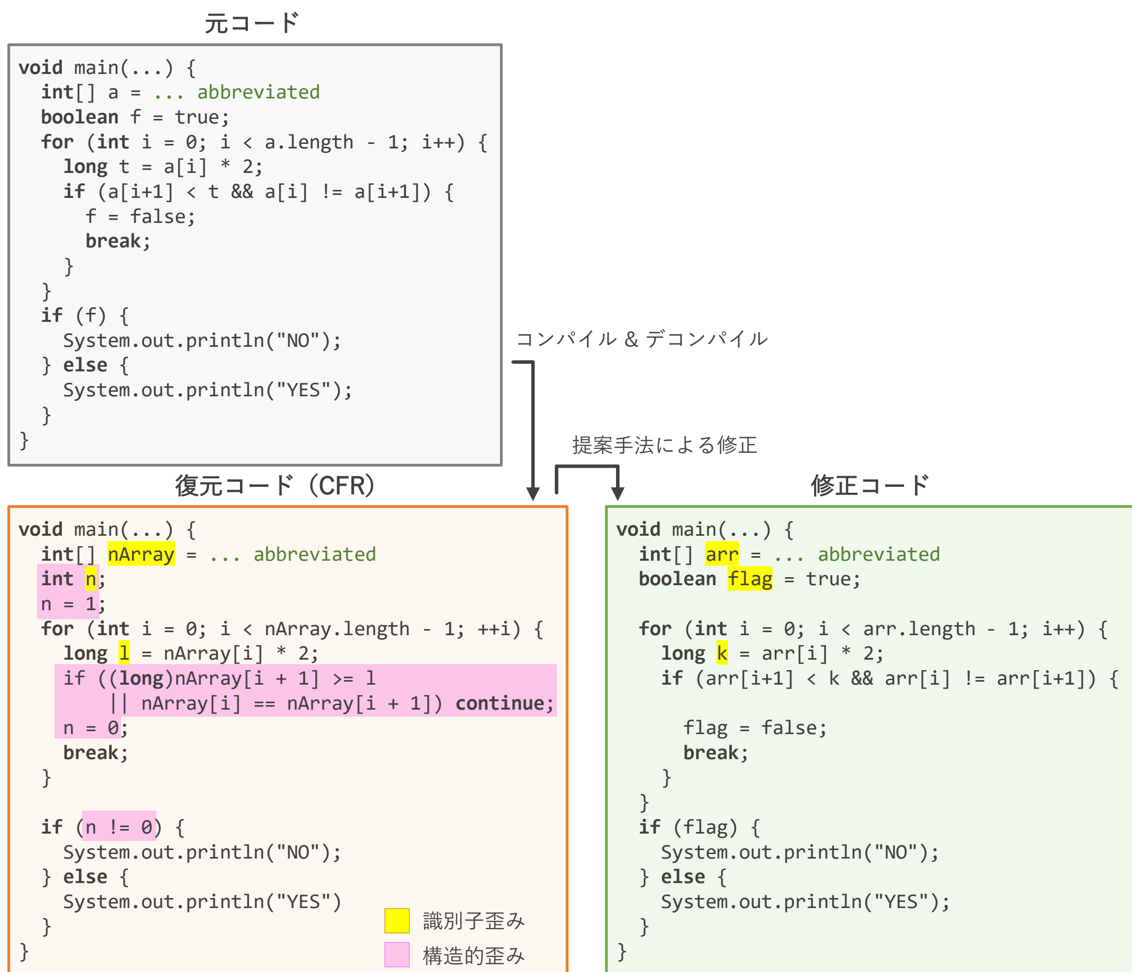

図 9　ReCa のある題材における歪み修正の実例

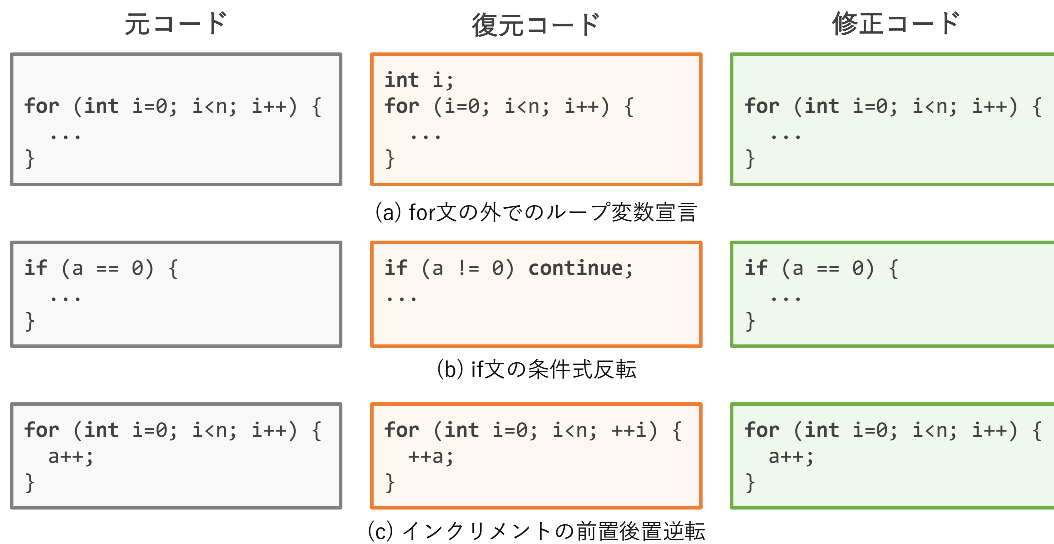

図 10　提案手法によって修正できた歪みの例

5.2　具体的な修正内容

提案手法はどのような歪みを修正でき，どのような歪みを修正できなかったのか，そしてどのような歪みが新たに追加されてしまったのかについてそれぞれ確認した．

5.2.1　修正できた歪み

図 10 に提案手法によって修正できた歪みの例を 3 つ示す．まず一つ目は，for 文の外でループ変数の宣言を行う歪みの修正である．通常，ループ変数は for 文の初期化式で宣言を行い，初期化する．しかし，復元コードでは for 文の外で宣言を行っていた．for 文外でこの変数を使用しない場合には，初期化式内で宣言することが望ましいとされているため，正しく修正できたといえる．次に二つ目は，if 文の条件式が反転してしまう歪みの修正である．図 10（b）に示すように，復元コードではしばしば if 文の条件式を反転させている場合があった．この歪みに関しては，プログラムの振る舞いに影響を与えることはほとんどないが，自然な書き方ではないため可読性の低下に繋がる．提案手法により，元コードと同じ書き方へと修正できた．最後に三つ目は，インクリメントの前置後置に関する歪みの修正である．この歪みは可読性やプログラムの振る舞いに特に影響はないものの，元コードと同じ書き方へと修正できた．

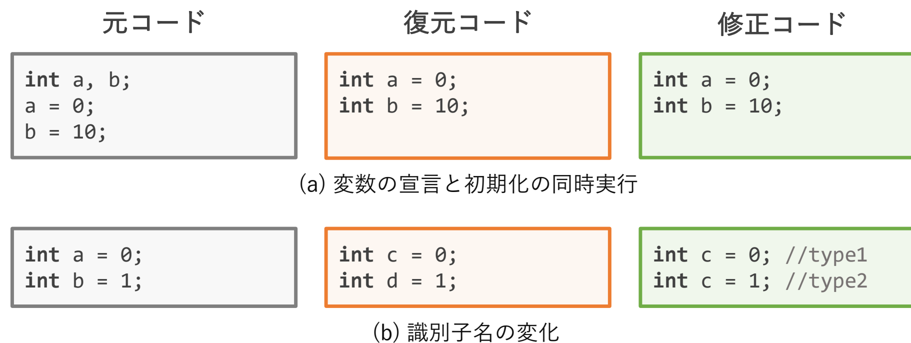

図 11　提案手法によって修正できなかった歪みの例

5.2.2　修正できなかった歪み

図 11 に提案手法によって修正できなかった歪みの例を 2 つ示す．一つ目の例は変数の宣言と初期化に関する歪みである．元コードでは変数の宣言のみ行った後に初期化を行っていたのに対して，復元コードでは変数の宣言と初期化を同時に行っている．修正コードにおいても復元コード同様に，変数の宣言と初期化を同時に行っており修正できなかった．これは構造的歪みに該当する．復元コードに含まれる構造的歪みのおよそ 10% が提案手法によって修正できなかったが，その多くが図 11（a）に示される歪みである．この歪みは，可読性やプログラムの振る舞いへの影響はない上に，修正コードの方が自然であると考える．そのため，構造的歪みのうち修正できなかった約 10% は対策を必要とする問題ではないといえる．さらに，この例から提案手法は単に元コードへ復元するのではなく，自然なコードへと復元していることが分かる．これは，提案手法が事前学習済みモデルを活用していることによる利点と考える．事前学習済みモデルは，膨大な量のデータを用いてプログラミング言語の一般的な構文やコーディングのパターンを学習し理解している．そのため，提案手法は単に元コードへ復元するのではなく，自然なコードに戻すような振る舞いをしていると考えられる．次に二つ目の例は，識別子歪みに関する例である．修正できなかった識別子歪みには 2 種類存在する．一つは復元コードと同じ変数名のままで修正できなかった場合で，もう一つは復元コードとも異なる変数名が付けられてしまう場合である．特に後者の場合，図 11（b）の例に示すように変数名が重複する問題が発生する可能性がある．変数名の重複はコンパイルエラーを引き起こすため，今後対策を考えるべき問題である．

5.2.3　追加された歪み

図 12 に提案手法によって新たに追加された歪みの例を 2 つ示す．まず，一つ目は for 文の省略可能な中括弧の省略である．元コードにおいて for 文中の処理が一行であった場合に，修正コードでは中括弧を省略した書き方に変化することがあった．本稿の定義においてはこの変化は歪みとなってしまうも

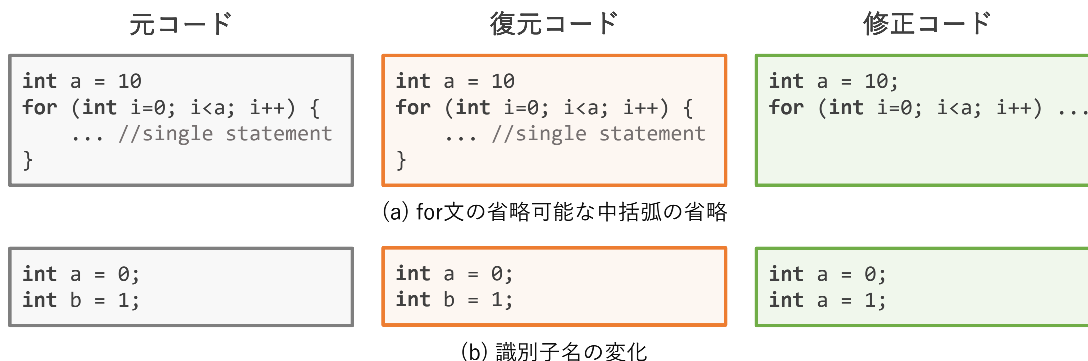

図 12 提案手法によって新たに追加された歪みの例

のの，可読性やプログラムの振る舞いへの影響はない．5 章 1 節において修正コードに含まれる歪みの約 70% が提案手法によって追加されたものであるという結果を示したが，その多くがこの例で示すような悪影響のない歪みであった．したがって，構造的歪みの混入率が高いことに関しては，対策を必要とする問題ではないと考える．二つ目は識別子歪みに関する例である．修正コードでは，元コードとは異なる変数名がつけられることがある．その際に同じスコープ内に同じ名前の変数が複数宣言されてしまう場合があり，図 11（b）の例で述べたのと同様に変数名の重複問題が発生する．そのため，今後対策を考えるべき問題である．

5.3 コンパイル可能率とテスト通過率

表 2 に復元コードと修正コードそれぞれにおいてコンパイル可能であったコードの割合とテストを通過したコードの割合を示す．CFR によって復元されたコードは 861 個のうち 848 個のコードがコンパイル可能であり，さらにそのうちの 792 個のコードがテストを通過した．一方で，提案手法によって生成された修正コードに関しては，コンパイル可能であったコードは約 51% の 438 個であり，復元コードと比較して大幅に減少した．テストを通過したコード数に関しても，約 45% の 391 個と減少した．デコンパイラに Fernflower を利用した場合もコンパイル可能なコードの割合とテストを通過するコー

表 2 コンパイル可能率とテスト通過率

デコンパイラ	コード	コンパイル可能率（A）	テスト通過率（B）	B/A
CFR	復元コード	98.5%（848/861）	92.0%（792/861）	93.4%
	修正コード	50.9%（438/861）	45.4%（391/861）	89.3%
Fernflower	復元コード	97.4%（839/861）	91.3%（786/861）	93.7%
	修正コード	53.0%（457/861）	48.2%（415/861）	90.8%

ドの割合ともに似た傾向の結果となった．表 2 の一番右の列は，コンパイル可能であったコードのうちテストも通過したコードの割合を示している．この結果から，どちらのデコンパイラの場合も復元コードと修正コードで大きく差があるわけではないことが分かる．すなわち，提案手法の抱える課題の一つはコンパイル可能なコードの生成数を増加させることにある．

そこで，修正コードにおいて発生したコンパイルエラーについて調査を行った．表 3 に，コンパイル不可能であった 423 個の修正コード（デコンパイラは CFR を利用）で発生した全 1,249 個のエラーのうち，数の多かった上位 5 つのコンパイルエラーとその主な原因について示す．X は任意の識別子や区切り文字を表す．コンパイルエラーの原因が複数考えられるエラーもあるが，今回の実験において大部分を占めていた原因のみを記載している．表 3 から，ほとんどのエラーが識別子か区切り文字に関するエラーであることがわかる．事前学習済みモデルを用いてコンパイルエラーを修正する手法を提案した関連研究 [28] では，区切り文字の過不足を修正するためのモデルや誤った識別子を適切な識別子へと修正するためのモデルというように特定のエラーの修正を目的にファインチューニングしたモデルを複数組み合わせて利用している．関連研究を参考に識別子や区切り文字の修正に特化したモデルを組み込む，もしくは識別子や区切り文字の修正用である学習データを追加してファインチューニングを行うことで，修正コードにおけるコンパイル可能率を向上させられると考える．

表 3　修正コードにおける主なコンパイルエラー

エラータイプ	エラー数	原因
cannot find symbol	286	変数名の誤り
X expected	185	区切り文字の過不足
variable X is already defined	156	変数名の重複
reached end of file while parsing	94	閉じ括弧の不足
incompatible types	85	型の不一致

5.4 識別子に対する後処理

提案手法においてコンパイルエラーを引き起こす原因の多くが識別子や区切り文字に関係することが分かった．その中でも特に多かった変数名の誤りや重複によるエラーは比較的容易に修正できるエラーである．そこで，コンパイルエラーとなった修正コードのうち，発生するエラーの種類が cannot find symbol と variable X is already defined のどちらか 1 種類，もしくはその両方のみであったコードに関して，簡単な文字列置換による修正を行った．

図 13 に文字列置換によるコンパイルエラー修正の例を示す．初めに，変数名重複の修正方針について説明する．図 13（a）に示すように変数名 `a` の重複宣言があった場合，変数名を `a_$`へと置換する．

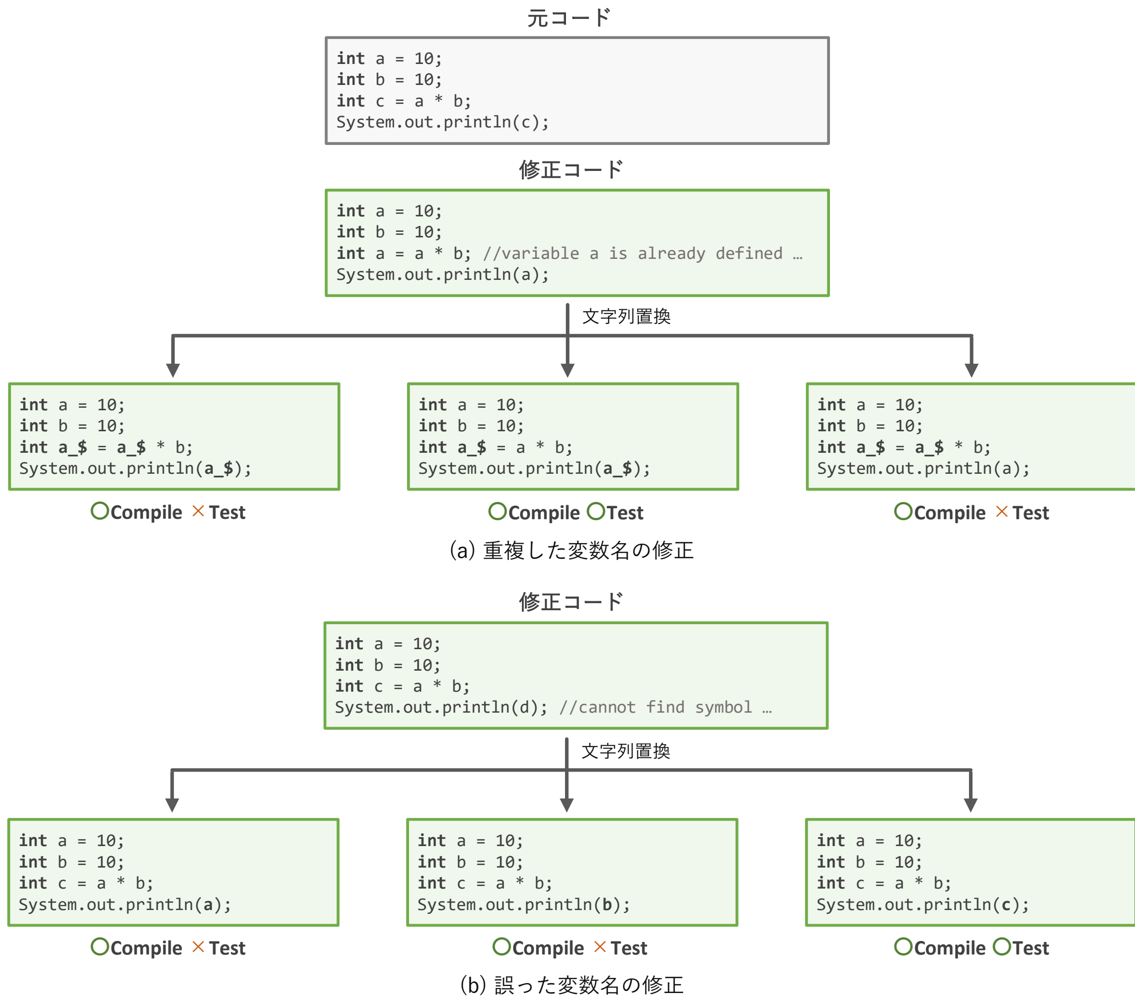

図 13　文字列置換によるコンパイルエラー修正の例

その後，重複宣言以降で使用されるすべての変数 a に関して，a_$に置換するか否かの 2 択からなる全パターンの置換を行う．仮に重複宣言以降で n 回重複変数が現れる場合，すべて置換しないパターンを除いた $2^n - 1$ 通りの置換を行う．図 13（a）の例では重複変数 a が 2 回現れるため，3 パターンの置換を行う．実際には，1 パターン置換するごとにコンパイルとテストを実行し，どちらもクリアした時点で置換を終了する．次に，変数名誤りの修正方針について説明する．図 13（b）に示すように宣言されていない変数 d が使用されていた場合，それ以前で宣言されているすべての変数への置換を試す．今回の例では，int 型の変数が 3 つ宣言されているため，3 パターンの置換を行う．変数重複の修正と同様に，1 パターン置換するごとにコンパイルとテストを実行し，どちらもクリアした時点で置換を終了する．以上の方針に基づき 2 種類のコンパイルエラーの修正を試みる．

表 4 に，文字列置換による修正でどの程度のコードがコンパイル可能となりテストを通過したのかを示す．CFR の場合，文字列置換による修正の対象コードは 119 個であった．そのうち，91 個のコード

がコンパイル可能となり，さらにそのうち 45 個のコードがテストを通過した．同様に Fernflower の場合では，110 個の対象コードのうち 82 個がコンパイル可能となり，47 個がテストを通過した．非常に単純な文字列置換でもある程度の数のコードを元コードと同じ振る舞いに修正できることから，5 章 3 節で述べたような識別子や区切り文字の修正に特化した学習データによるファインチューニングを追加することで，提案手法による修正コードのコンパイル可能率とテスト通過率が向上する可能性は高いと考える．

表 4　文字列置換による修正後のコンパイル可能数とテスト通過数

		文字列置換後	
デコンパイラ	修正対象コード数	コンパイル可能数	テスト通過数
CFR	119	91	45
Fernflower	110	82	47

6　考察

提案手法による修正で，どちらのデコンパイラの場合も構造的歪みは約 9 割修正できたのに対して，識別子歪みはどちらも約 6 割ほどしか修正できなかった．このように構造的歪みに比べて識別子歪みの修正がうまくいかなかった要因を考察する．

識別子名のつけ方は，ある程度の共通認識は存在するが開発者への依存度が高く，同じ意味の変数であっても異なる名前がつけられることが多々ある．一方で，ソースコードの構文に関しては同じ言語であれば開発者への依存度は比較的低いと考える．今回使用した事前学習済みモデルの事前学習用とファインチューニング用のデータセットは，いずれも複数の開発者によって作成されたソースコードから成る．したがって，事前学習済みモデルは開発者によって正解が大きく変化しうる識別子歪みの修正よりも，ある程度正解が定まっている構造的歪みの修正の方が適しているのだと考える．

提案手法によって生成された修正コードでは，復元コードと比較してコンパイル可能率が大幅に減少した．5 章 3 節の結果から，コンパイルエラーとなる原因の大部分が識別子や区切り文字に関係していることが分かった．修正コードにおいて識別子や区切り文字に関するエラーが多く発生してしまった要因について考察する．

はじめに，識別子に関するコンパイルエラーが多く発生したのは，前述したとおり識別子の命名は開発者への依存度が高いことが要因の一つであると考える．変数名に意味を持たせる習慣がなく，単にアルファベット一文字の名前しかつけない開発者や異なるスコープにおいては何度も同じ名前を利用するという開発者もいる．このような開発者により作成されたソースコードをファインチューニングに使用した場合，同じ変数名が一つのコード内で頻繁に出現するようになり，最悪の場合それらのスコープが重複してしまうことでコンパイルエラーに繋がっている可能性が考えられる．ファインチューニング用の学習データとして利用するソースコードに対して，変数の命名方法や同じ変数名の使用回数などに制約を設けることで，識別子に関するコンパイルエラーの発生を抑えられる可能性があると考える．次に，区切り文字に関するコンパイルエラーが多く発生した要因の一つは，ファインチューニングの形式にあると考える．今回ファインチューニングのステップでは，ソースコード全体をそのまま利用して翻訳タスクによる学習を行った．特にクラスやメソッドのブロックを表す中括弧においては，ソースコードの規模が大きくなるほど始まりと終わりの距離が離れるため，事前学習済みモデルがそれらの対応を把握することが困難になるのではないかと考える．そのため，学習単位をメソッド単位にするなど規模を小さくすることで，区切り文字に関するコンパイルエラーの発生を抑えられる可能性があると考える．また，区切り文字の過不足解消に特化したデータによるファインチューニングを行うことも解決策の一つとして考えられる．

7 妥当性の脅威

提案手法の歪み修正性能を評価するにあたり，競技プログラミングのデータセットである ReCa を利用した．ReCa に含まれるソースコードは比較的行数も小さく，含まれる構文も単純なものが多い．実プロジェクトに見られるような，より規模の大きく複雑なソースコードから成るデータセットを用いた場合，得られる実験結果が異なる可能性がある．

また，本研究においてハイパーパラメータはバッチサイズとエポック数のみ変更して実験を行った．他のハイパーパラメータに関しても調節することで，モデルの精度が変化して得られる結果が変わる可能性がある．

8　おわりに

本研究では，デコンパイラで復元されたソースコードに含まれる 2 種類の歪みの修正方法として，事前学習済みモデルを活用した手法を提案した．結果として，CFR による復元コードに含まれる識別子歪みの約 55%，構造的歪みの約 91% を修正でき，Fernflower による復元コードに含まれる識別子歪みの約 59%，構造的歪みの約 88% を修正できることを確認した．

また，今後の課題として以下の 3 つを挙げる．1 つ目は，ファインチューニングに利用する学習データの見直しである．今回行った実験から，ファインチューニングの際に識別子や区切り文字に特化した学習データも追加することで，歪み修正性能の向上につながると考える．また，今回の実験では複数の開発者が作成したソースコードから成るデータセットを利用した．開発者ごとに異なるルールでプログラムを書いていた場合，ファインチューニングの精度に悪影響を及ぼす可能性がある．そのため，ある一定のルールの下で書かれたソースコードの集合を学習データに用いることで，より高い性能を実現できると考える．2 つ目は実験対象とするプログラミング言語の種類を増やすことである．提案手法は，ファインチューニングの際のデータセットを変えることでプログラミング言語の種類によらず網羅的に歪みの修正を行うことができる．そのため，複数のプログラミング言語を用いた実験を行う必要がある．3 つ目は，既存手法との比較である．デコンパイラにより発生する識別子歪みの修正手法が存在する [29] [30]．今後は，提案手法とこれらの手法との比較を検討している．

謝辞

本研究の遂行にあたり，多くの方々にご指導とご支援を賜りました．

丁寧で暖かなご指導を賜りました，楠本真二教授に心より感謝申し上げます．また，研究指導の他にも差し入れ等，様々なご支援をいただき大変お世話になりました．

担当教員として 2 年間，熱心かつ丁寧なご指導をいただきました，柗本真佑助教に心より感謝申し上げます．研究の方向性などで迷った際には，個別に時間を取って相談に乗っていただきました．また，論文執筆の際の添削や研究発表の練習にも多くの時間を割いていただき，研究以外でも必要となる能力の向上に繋がりました．

ソフトウェア工学講座の肥後芳樹教授には，本研究に関する有益かつ的確なご助言をいただきました．研究の方向性や研究発表における改善案など多数の指摘をいただきました．心より感謝申し上げます．

事務補佐員の橋本美砂子氏には，研究活動を円滑に行うための様々なご支援をいただきました．特に，国内外の学会へ参加するための事務処理では大変お世話になりました．深く感謝申し上げます．

楠本研究室の皆様には，活発な議論や息抜きの会話など様々な面で支えていただきました．誠にありがとうございました．

最後に，本研究の遂行中，常に励まし応援していただいた友人や家族へ心より感謝の意を表します．

参考文献

[1] Cifuentes, C. and Gough, K. J.: Decompilation of binary programs, *Software: Practice and Experience*, Vol. 25, No. 7, pp. 811–829 (1995).

[2] Cifuentes, C., Waddington, T. and Van Emmerik, M.: Computer security analysis through decompilation and high-level debugging, in *Working Conference on Reverse Engineering (WCRE)*, pp. 375–380 (2001).

[3] Milosevic, N., Dehghantanha, A. and Choo, K.-K. R.: Machine learning aided Android malware classification, *Computers and Electrical Engineering*, Vol. 61, pp. 266–274 (2017).

[4] Cen, L., Gates, C. S., Si, L. and Li, N.: A probabilistic discriminative model for android malware detection with decompiled source code, *Transactions on Dependable and Secure Computing (TDSC)*, Vol. 12, No. 4, pp. 400–412 (2014).

[5] Jaffe, A., Lacomis, J., Schwartz, E. J., Goues, C. L. and Vasilescu, B.: Meaningful variable names for decompiled code: A machine translation approach, in *International Conference on Program Comprehension (ICPC)*, pp. 20–30 (2018).

[6] Hofmeister, J., Siegmund, J. and Holt, D.: Shorter identifier names take longer to comprehend, in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 217–227 (2017).

[7] Harrand, N., Soto-Valero, C., Monperrus, M. and Baudry, B.: The strengths and behavioral quirks of Java bytecode decompilers, in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 92–102 (2019).

[8] Mauthe, N., Kargén, U. and Shahmehri, N.: A large-scale empirical study of android app decompilation, in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 400–410 (2021).

[9] Liu, H., Shen, M., Zhu, J., Niu, N., Li, G. and Zhang, L.: Deep Learning Based Program Generation From Requirements Text: Are We There Yet?, *Transactions on Software Engineering (TSE)*, Vol. 48, No. 4, pp. 1268–1289 (2022).

[10] Felice, M., Yuan, Z., Andersen, Ø. E., Yannakoudakis, H. and Kochmar, E.: Grammatical error correction using hybrid systems and type filtering, in *Conference on Computational Natural Language Learning: Shared Task (CoNll)*, pp. 15–24 (2014).

[11] Yuan, Z. and Felice, M.: Constrained grammatical error correction using statistical machine translation, in *Conference on Computational Natural Language Learning: Shared Task*

(CoNLL), pp. 52–61 (2013).

[12] Yuan, Z. and Briscoe, T.: Grammatical error correction using neural machine translation, in *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, pp. 380–386 (2016).

[13] Wang, Y., Wang, Y., Dang, K., Liu, J. and Liu, Z.: A comprehensive survey of grammatical error correction, *Transactions on Intelligent Systems and Technology (TIST)*, Vol. 12, No. 5, pp. 1–51 (2021).

[14] Hinton, G. E., Osindero, S. and Teh, Y.-W.: A fast learning algorithm for deep belief nets, *Neural Computation*, Vol. 18, No. 7, pp. 1527–1554 (2006).

[15] Krizhevsky, A., Sutskever, I. and Hinton, G. E.: Imagenet classification with deep convolutional neural networks, *Advances in Neural Information Processing Systems*, Vol. 25, (2012).

[16] Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., et al.: Deep speech: Scaling up end-to-end speech recognition, *arXiv preprint arXiv:1412.5567* (2014).

[17] Bahdanau, D., Cho, K. and Bengio, Y.: Neural machine translation by jointly learning to align and translate, *arXiv preprint arXiv:1409.0473* (2014).

[18] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al.: Google's neural machine translation system: Bridging the gap between human and machine translation, *arXiv preprint arXiv:1609.08144* (2016).

[19] Shang, L., Lu, Z. and Li, H.: Neural Responding Machine for Short-Text Conversation, *arXiv preprint arXiv:1503.02364* (2015).

[20] Zhang, Y., Sun, S., Galley, M., Chen, Y.-C., Brockett, C., Gao, X., Gao, J., Liu, J. and Dolan, B.: DialoGPT: Large-Scale Generative Pre-training for Conversational Response Generation, in *ACL, system demonstration* (2020).

[21] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding, *arXiv preprint arXiv:1810.04805* (2018).

[22] 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 pretraining approach, *arXiv preprint arXiv:1907.11692* (2019).

[23] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners, *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33, pp. 1877–1901 (2020).

[24] Wang, Y., Wang, W., Joty, S. and Hoi, S. C.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, *arXiv preprint arXiv:2109.00859* (2021).

[25] Falleri, J., Blanc, andXavier F. M., Martinez, M. and Monperrus, M.: Fine-grained and accurate source code differencing, in *International Conference on Automated Software Engineering (ASE)*, pp. 313–324 (2014).

[26] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M. and Brockschmidt, M.: Codesearchnet challenge: Evaluating the state of semantic code search, *arXiv preprint arXiv:1909.09436* (2019).

[27] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. and Polosukhin, I.: Attention is all you need, *Advances in Neural Information Processing Systems*, Vol. 30, (2017).

[28] Ahmed, T., Ledesma, N. R. and Devanbu, P.: SynShine: Improved Fixing of Syntax Errors, *Transactions on Software Engineering (TSE)*, Vol. 49, pp. 2169–2181 (2021).

[29] Lacomis, J., Yin, P., Schwartz, E., Allamanis, M., Le Goues, C., Neubig, G. and Vasilescu, B.: Dire: A neural approach to decompiled identifier naming, in *International Conference on Automated Software Engineering (ASE)*, pp. 628–639 (2019).

[30] Nitin, V., Saieva, A., Ray, B. and Kaiser, G.: DIRECT: A Transformer-based Model for Decompiled Identifier Renaming, in *Workshop on Natural Language Processing for Programming (NLP4Prog)*, pp. 48–57 (2021).