文法誤り訂正手法の転用による デコンパイラの歪み修正手法の提案

開地 竜之介[1]　柗本 真佑[1]　楠本 真二[1]

概要：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．この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を転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 55%，構造的歪みでは約 91%を修正できることを確認した．

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（Quirk）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，プログラム構造の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，デコンパイラの種別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，深層学習に基づく文法誤り訂正手法（Grammatical Error Correction; GEC）を転用した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

[1] 大阪大学大学院情報科学研究科 大阪府吹田市

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，識別子名に関する歪みの 55%，プログラム構造に関する歪みの 91%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．GEC 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．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を転用する．

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．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や音声認識といった分野でよく活用される．また，機械翻訳や対話システム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な深層学習モデルのことである．学習データの収集が困難なタスクに対しても、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[14]，2020 年に Brown らによって提案された GPT-3[15], また今回活用する CodeT5[16] 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．デコンパイラの問題点として，元のソースコードを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 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等価ではあるものの，開発者の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．

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 を転用した歪み修正手法を提案する．ソースコード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[16] を用い，歪みを含む・含まな

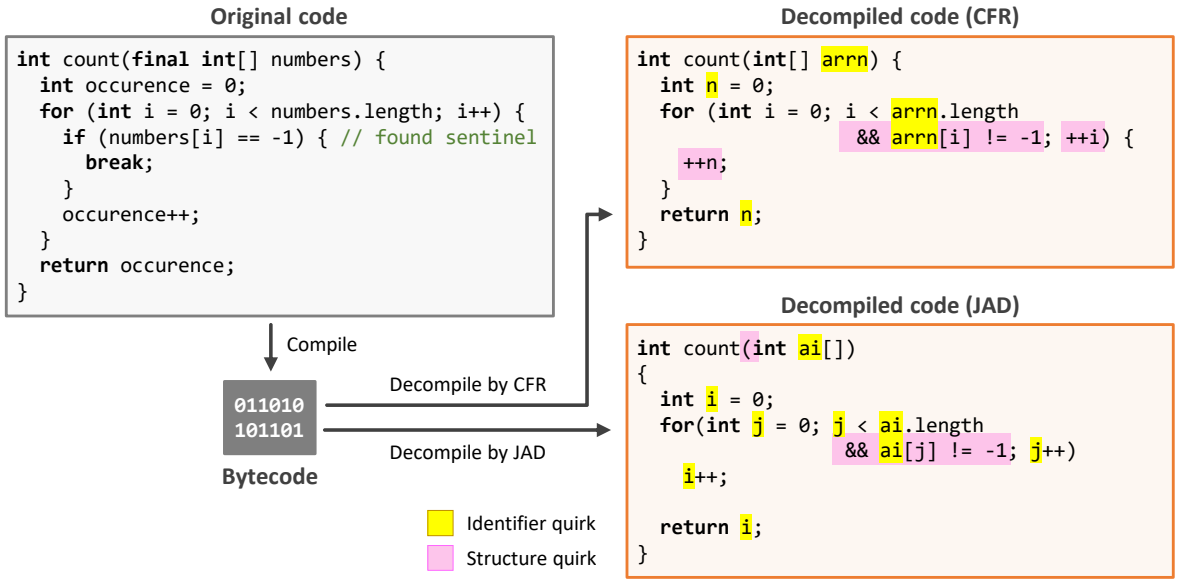

図 1: デコンパイラによって発生した歪みの例

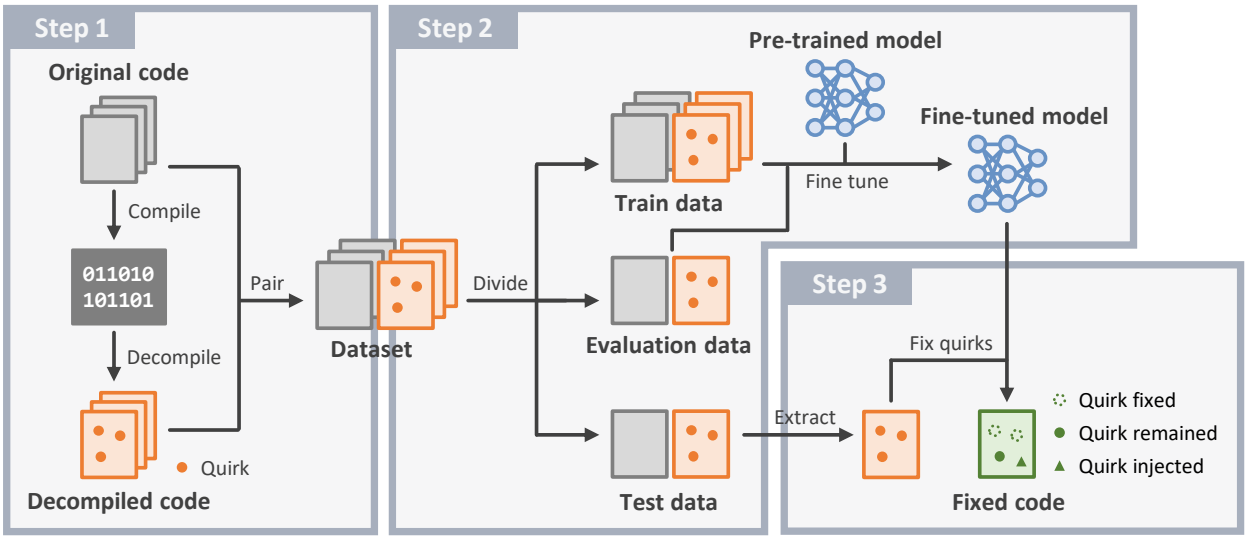

図 2: 提案手法による歪み修正の流れ

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，CodeT5 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

提案手法の有用性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]を利用する．CFR は，復元コードのコンパイル可能性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，復元ソースコードと元のソースコードの抽象構文木の差分を歪みと定義する．また，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/

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[17] 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文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．

4.4 歪み修正能力の評価方法

GumTree によって得られた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．

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]

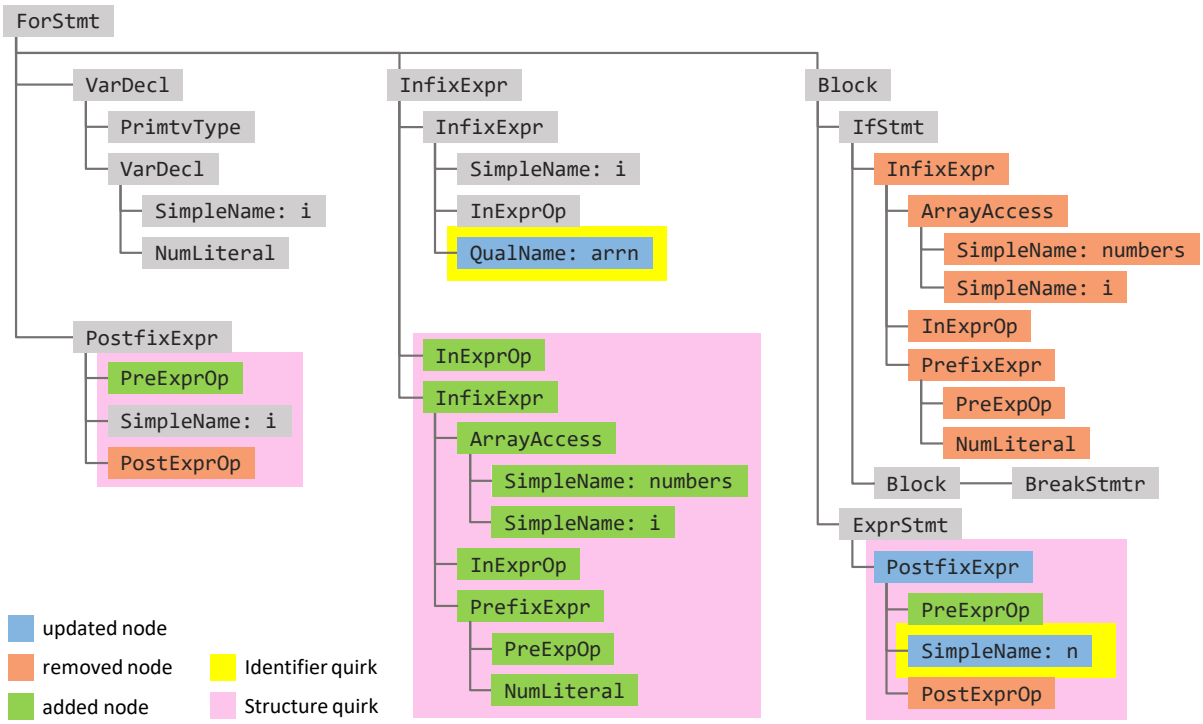

図 3: GumTree を用いた歪み検出の例

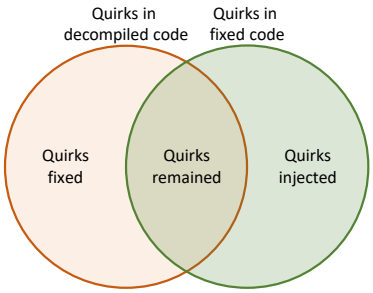

図 4: 2 つの歪み集合の包含関係

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 Java だけを対象とした．また，復元コードを得るためにはコンパイラにより生成されたバイトコードが必要である．そのため，コンパイル可能な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 2KB 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．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かの両方を考慮し，最終的に 2KB 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，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[16] である．CodeT5 は，CodeSearchNet[18] 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 には異なるサイズのモデルが複数あるが，メモリの問題から今回は CodeT5-small を利用した．

本実験で CodeT5 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．また，ファインチューニングに要した時間はおよそ 1 時間 30 分であった．

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

5. 実験結果と考察

5.1 実験結果

図 5 と**図 6** に識別子歪みの集合と構造的歪みの集合それぞれに関するベン図を示す．ベン図の中の数値はテストデータとして用いた全ソースコード (861 個) の歪みの総計を表している．まず図 5 の識別子歪みに着目すると，復

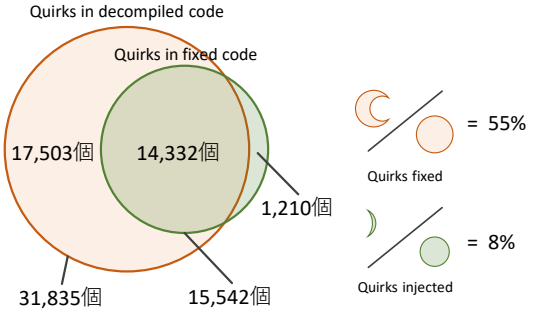

図 5: 識別子歪みのベン図

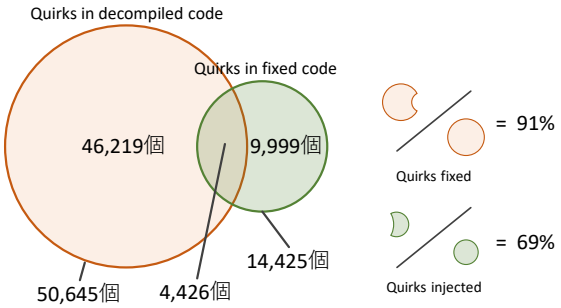

図 6: 構造的歪みのベン図

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%とかなり少数であった．

次に，図 6 から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 個であり，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．

図 7 に，提案手法による歪み修正の実例を示す．ただし，修正による効果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．

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，むしろ可読性の観点では修正コードの方が優れている．この点は，提案手法の強みの一つであると考える．

5.2 考察

提案手法による修正で，識別子歪みのうち約 55%を修正できたのに対して，構造的歪みは約 91%と識別子歪みよりも大幅に修正成功率が高くなった．このような結果とな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それほど開発者への依存度は高くないと考える．今回使用した事前学習済みモデルの事前学習用のデータセットとファインチューニングに使用したデータセットは，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．

6. 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，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%を提案手法により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，実験で用いるデコンパイラの種類を増やすことである．デコンパイラの問題点として歪みの発生があるが，この歪みはデコンパイラの種類で異なってくる．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デコンパイラの種類によらず網羅的に歪みの修正を行うことが可能と考える．そのため，複数のデコンパイラを用いた実験を行う必要がある．2 つ目は，異なる観点の評価指標を用いることである．今回は，歪み修正の能力を評価するにあたり，AST の差分のみに着目している．しかし，実用性を考えると，修正コードがコンパイル可能であるか，元コードのテストケースを通過するかといった観点での評価を行う必要がある．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

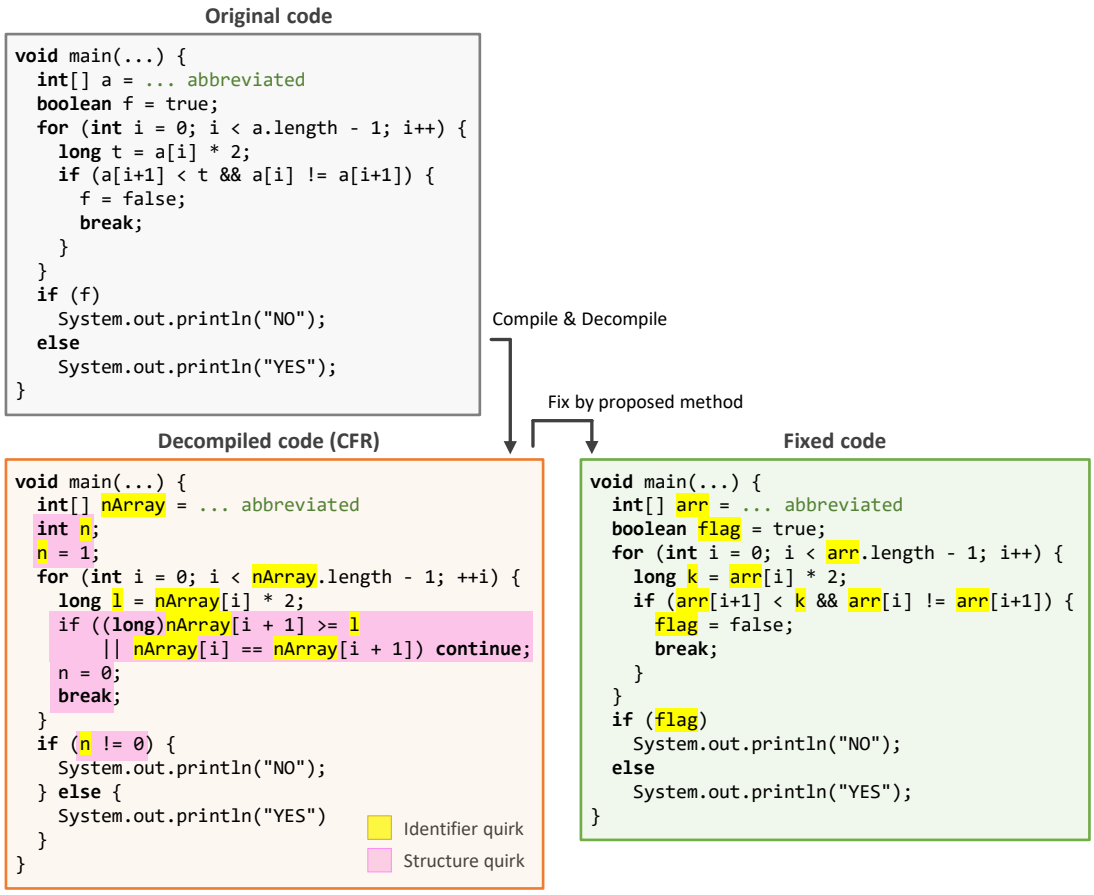

図 7: ReCa のある題材における歪み修正の実例

する [19][20]．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 本研究の一部は，JSPS 科研費（JP21H04877，JP20H04166，JP21K18302，JP21K11829）による助成を受けた．

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, *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, *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, *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, *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, *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, *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, *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, *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] 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).

[15] 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).

[16] 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).

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

[18] 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).

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

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