修士学位論文

題目

複数のプログラミング言語に対するデコンパイラ歪み修正手法の再評価

指導教員

楠本 真二 教授

報告者

田中 叶也

令和 7 年 1 月 28 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 6 年度 修士学位論文

複数のプログラミング言語に対するデコンパイラ歪み修正手法の再評価

田中 叶也

内容梗概

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 と同様に識別子に関する歪みの約 6 割と構文に関する歪みの約 9 割が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り，特に C 言語では他の言語と比べ多くの歪みが新たに発生する傾向が見受けられた．

主な用語

デコンパイラ，転移学習，ファインチューニング，事前学習済みモデル，歪み，大規模言語モデル

目次

1 **はじめに** 1

2 **準備** 2

2.1 深層学習と事前学習済みモデル 2

2.2 デコンパイラと歪み 2

2.3 先行研究 5

3 **実験の目的** 7

4 **実験設定** 8

4.1 歪みの定義 8

4.2 歪みの検出方法 8

4.3 歪み修正性能の評価方法 8

4.4 データセット 9

4.5 事前学習済みモデル 9

5 **実験結果** 11

5.1 歪み修正性能の言語間の比較 11

5.2 歪み修正の内訳 12

5.3 歪み修正の実例 16

5.4 コンパイル可能率とテスト通過率 17

6 **考察** 19

7 **妥当性への脅威** 21

8 **関連研究** 22

9 **おわりに** 23

謝辞 24

参考文献 25

図目次

1 歪みの実例（Java） 3
2 歪みの実例（C 言語） 4
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 [1] 6
4 修正前後の歪み集合の包含関係と評価指標 9
5 識別子歪みに対する歪み修正性能 11
6 構造的歪みに対する歪み修正性能 12
7 歪み修正の実例（C 言語） 17

表目次

1 除去できた識別子歪みの内訳 13
2 3 種類の変数の誤った修正名と頻度（C^{O0}）. 13
3 3 種類の変数の誤った修正名と頻度（C^{O3}）. 14
4 構造的歪みの内訳（Python）. 15
5 構造的歪みの内訳（C^{O0}）. 15
6 構造的歪みの内訳（C^{O3}）. 16
7 コンパイル可能率とテスト通過率 18
8 Java の修正コードにおける主なコンパイルエラー [1] 20

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 [2]．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 [3] や，ソフトウェアの脆弱性解析 [4] 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 [5].

我々の先行研究 [1] 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が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 [6]. 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. したがって手動で特徴量を設定することが困難な画像認識 [7] や音声認識 [8] のような分野で活用される．また，機械翻訳 [9][10] のような自然言語処理の分野でも活用される．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やドメイン適応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，2018 年に Devlin らによって提案された BERT[11] や 2019 年に Liu らによって提案された RoBERTa[12]，2020 年に Brown らによって提案された GPT-3[13]，本研究で利用する Wang らによって提案された CodeT5[14] 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，ループ変数 `i` 以外の変数名は正しく復元されていない．`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歪みが見受けられる．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っている．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振る舞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 での実例には見受けられない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の傾向は異なるため，画一的な歪み修正は容易ではない.

```java
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

```java
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: 歪みの実例（Java）

```c
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

```c
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: 歪みの実例（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 [1]．ソースコード中の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歪みを含む・含まないソースコードペアを用いた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，画一的な歪み修正を実現している．ファインチューニングに用いるデータはテキスト形式であり，プログラミング言語やデコンパイラの種類に依らず歪み修正を行える．

2.3.1 歪み修正の流れ

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作成する．任意のデータソースから得たソースコードの全てに対してコンパイル・デコンパイルを試みることで作成する．次に Step2 では，ファインチューニングによって歪み修正モデルを作成する．Step1 で生成したデータセットを 80%，15%，5% の割合で訓練用，検証用，テスト用と分割し，訓練用と検証用のデータを用いて事前学習済みモデルに対しファインチューニングを行うことで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．

2.3.2 課題

先行研究では Java でしか評価実験を行っておらず，他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．Java では 2 種類のデコンパイラに対して，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．即ち，デコンパイラに対する汎用性が確認されている．しかしながら，プログラミング言語にも依らず歪み修正が可能であるかは明らかになっていない．

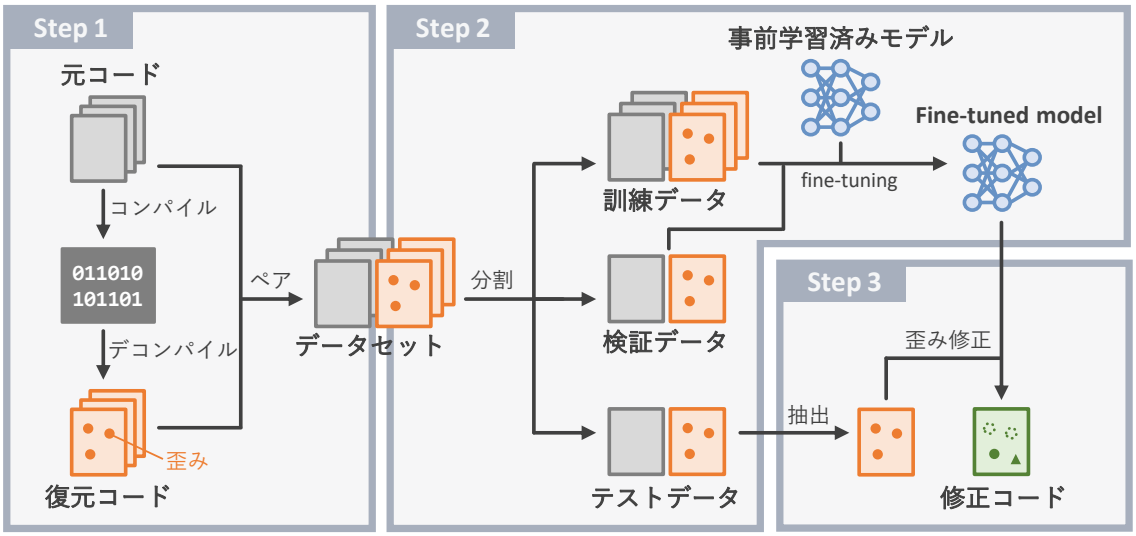

図 3: 転移学習を用いた歪み修正手法による歪み修正の流れ [1]

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，複数言語間で修正性能を比較する．Python は学習が容易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 [15]．そのため，セキュリティ解析で主に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．例えば，インデントによるコードブロックの表現や動的な型付け等の特徴から，Java よりも簡潔なソースコードとなる傾向がある．そのため，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過程で失われる情報が多く，デコンパイラによる復元が比較的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．さらに，デコンパイラの主な活用先としてマルウェアの挙動解析等のセキュリティ分野が挙げられるため，C 言語を対象とした歪み修正の需要は高いと考え，実験の対象言語として選定した．

Python のデコンパイラは Uncompyle6[*1]，C 言語のデコンパイラは Ghidra[*2]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 [16][17]．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 [18] ため採用した．

[*1] https://github.com/rocky/python-uncompyle6

[*2] https://ghidra-sre.org/

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．

4.1.1　識別子歪み

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 等のコンパイル時に識別子名が失われる言語では，デコンパイラによる識別子名の復元は困難である [19]．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．

4.1.2　構造的歪み

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構文の変化は可読性の低下につながり，最悪の場合プログラムの振る舞いも変わってしまう．

4.2　歪みの検出方法

GumTree[20] 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，delete-tree，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，二つの評価指標を定義する．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に対して取り除くことのできた歪みの割合とする．つまり除去率は高いほど良い修正性能であるといえる．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に対して新たに追加された歪みの割合とする．つまり混入率は低いほど良い修正性能であるといえる．

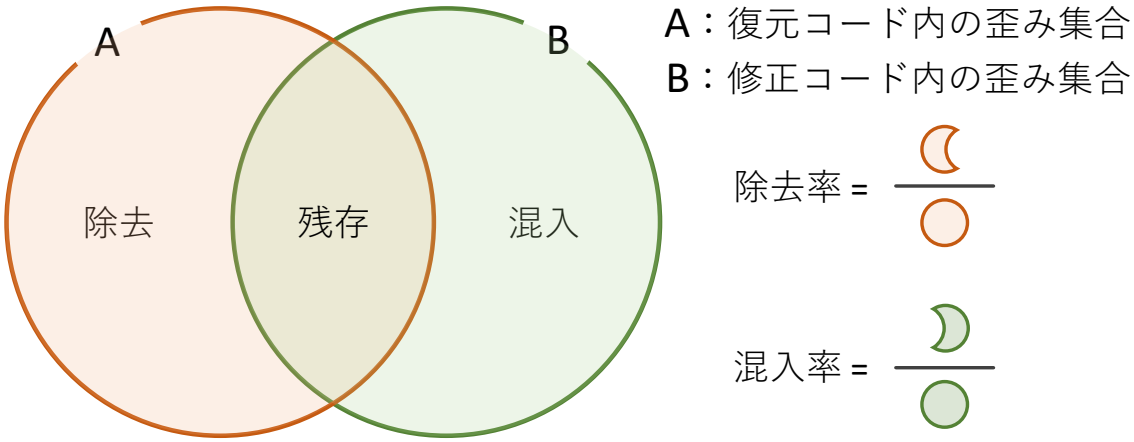

図 4: 修正前後の歪み集合の包含関係と評価指標

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[21] 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 でコンパイルした C 言語を C^{O0}，最適化オプション O3 でコンパイルした 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 未満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．C 言語では強い最適化が施されている C^{O3} の方が収集できたデータ数が少ない．これは最適化の程度が強くなった結果，デコンパイルに失敗する場合や復元コードのサイズが 2KB 以上となって除外される場合が増加したためである．収集したソースコードを 80%，15%，5% で分割し，それぞれ訓練用，検証用，テスト用として実験に用いる．

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[14] とする．CodeT5 は CodeSearchNet[22] のデータセットで事前学習された Transformer[23] 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．コード関連のタスクに対応している他のモデルと比較すると，CodeT5 はサイズや時間，メモリの効率が優れているため，本研究において採用した [24]．また，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 つの言語に対する歪み修正性能を示す．

まず 図 5 の識別子歪みに対する歪み修正性能に着目する．C^{O0} と C^{O3} 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 と比較すると混入率が高く，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．

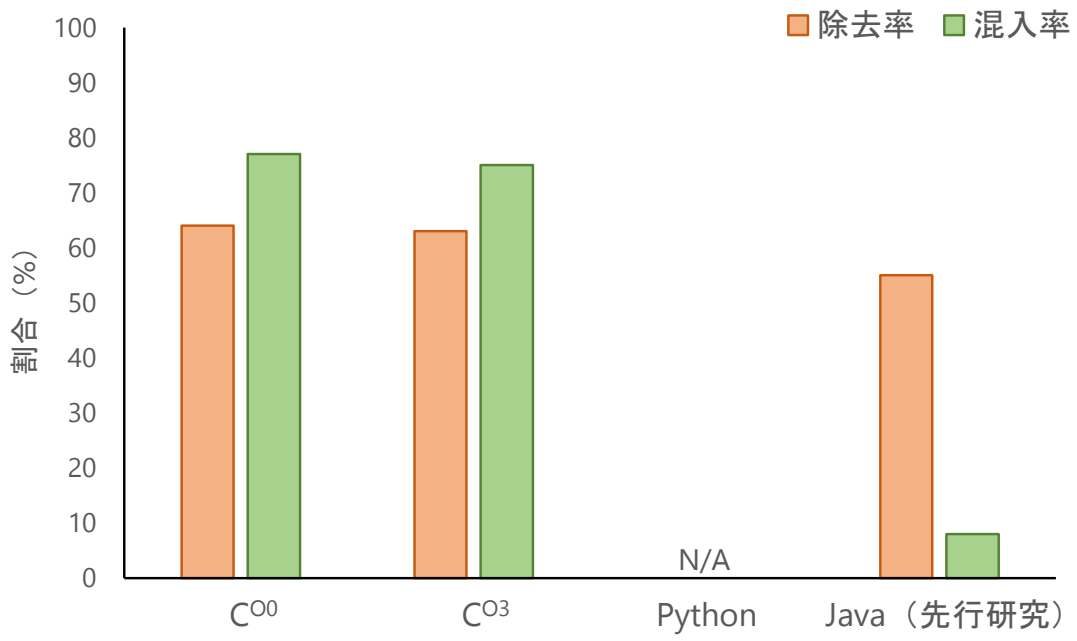

図 5: 識別子歪みに対する歪み修正性能

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とわかる．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インデントでコードブロックを区切るといった Python のソースコードの簡潔さが考えられる．

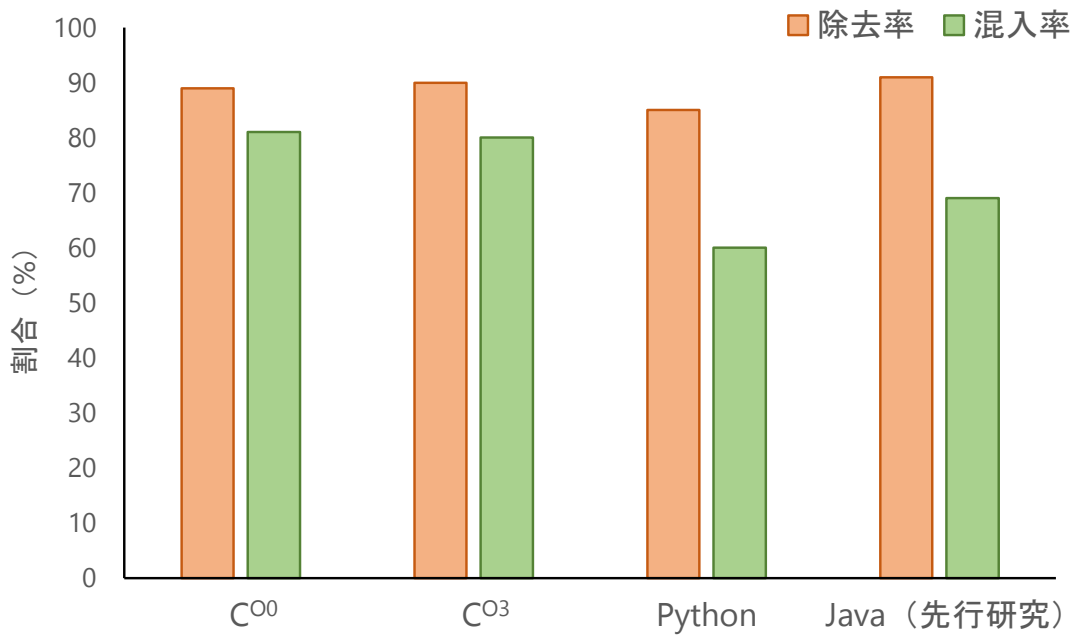

図 6: 構造的歪みに対する歪み修正性能

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．

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

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，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．

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

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, delete-tree, 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ため，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の一つであった．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確認できたことから，転移学習を用いた歪み修正手法の汎用性が確認できる．ただし，C 言語では言語特有の歪みが新たに混入してしまう場合も見受けられた．除去できた歪みと比較すると頻度は少ないものの，画一的な歪み修正を目指す上での課題といえる．

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

5.3 歪み修正の実例

転移学習を用いた歪み修正手法によって具体的にどのような歪み修正が行われるかについて確認する．図 7 に 2.2 節の 図 2 で示した C 言語のデコンパイラ Ghidra における歪みの実例に対して歪み修正を適用した結果を示す．

黄色のハイライトで示される識別子歪みは復元コードと修正コードで同じ数存在しており，歪み修正がうまくできていないように見える．しかし，`param_1`，`param_2`，`local_10` といった識別子名から，`arr`，`n`，`count` といった直感的に役割が分かりやすい識別子名に変換されており，可読性が向上している．

次に，ピンクのハイライトで示される構造的歪みはその多くが除去に成功している．例えば復元コードには `for` 文が `while` 文に変換されてしまう歪みや単項演算子が二項演算子に変換される歪み，番兵発見のための if 文が繰り返し分の条件式に組み込まれてしまう歪みなどが存在した．修正コードではそれらの歪みが全て除去されており，歪み修正によって可読性が向上しているとわかる．ただ，修正コードではプログラムの振る舞いが変化してしまっている．元コードでは番兵までの要素数が出力されるが修正コードでは番兵以外の要素数，すなわち値が `-1` でない全ての要素の数を計算して出力してしまう．このように，歪み修正によって振る舞いが変化してしまう場合がある．当然だが歪み修正は振る舞いが変化しないように行われる必要があり，歪み修正手法の汎用性を評価するためには歪み修正がプログラムの振る舞いに与える影響についても確認するべきである．

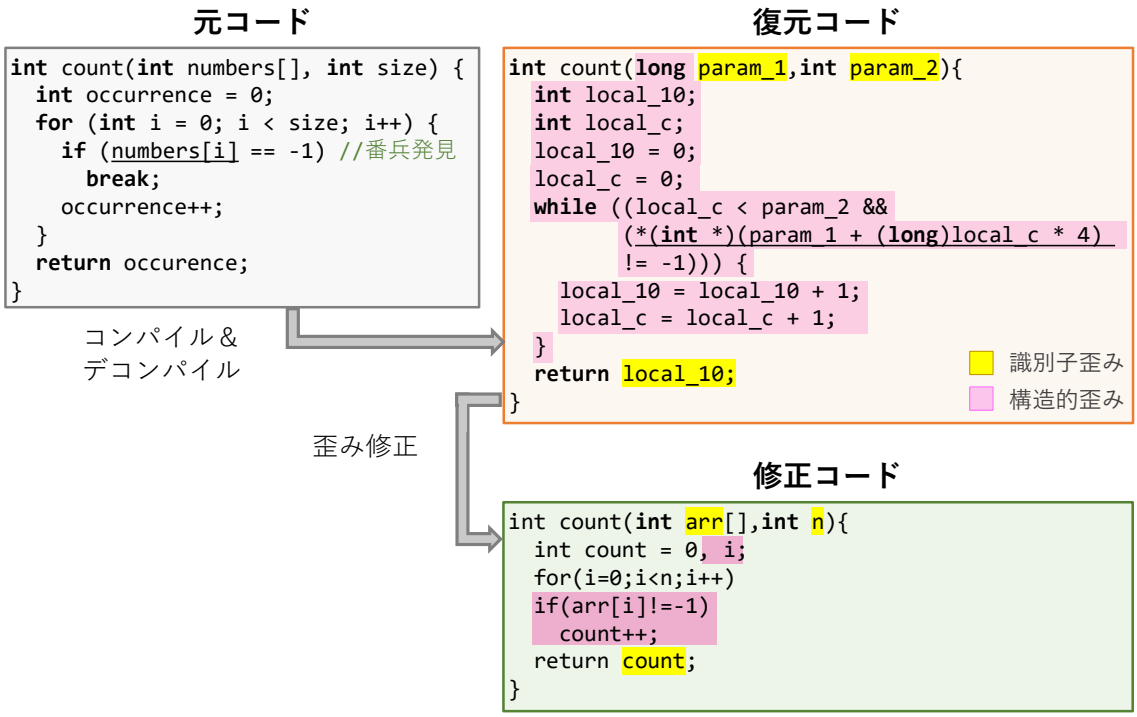

図 7: 歪み修正の実例（C 言語）

5.4 コンパイル可能率とテスト通過率

5.1 節と 5.2 節では複数のプログラミング言語に対する歪み修正性能とその内訳を分析したが，歪み修正がプログラムの振る舞いに与える影響は確認していない．5.3 節で示した歪み修正例でも見受けられたように，歪み修正によってプログラムの振る舞いが変化してしまう場合がある．歪み修正手法の汎用性を評価する上で，そのような振る舞いの変化が生じる割合は把握しておく必要がある．本節では各言語において歪み修正前後のコンパイル可能率とテスト通過率を確認し，歪み修正がプログラムの振る舞いに与える影響を分析する．

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 0.00% 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低いが，歪み修正前は 0.00% であることを鑑みれば歪み修正手法は効果的といえる．ただし，修正コードではコンパイル可能率と比べてテスト通過率が低くなっており，振る舞いが変化してしまっている傾向が見受けられる．その原因として C 言語の構造的歪みに対する修正の難しさが挙げられる．5.1 節で示

した通り，他の言語と比べると C 言語の構造的歪みは修正が難しい．そのため，振る舞いが変化してしまうような修正が行われてしまっていると考えられる．

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，歪み修正が振る舞いの改善に対して有効だとわかる．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，さらに識別子歪みが発生しないことが原因として考えられる．

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	0.00% (0/691)	0.00% (0/691)
	修正	42.5% (294/691)	9.99% (69/691)
C^{O3}	復元	0.00% (0/525)	0.00% (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失われていたマクロ定義がある程度修正される場合が多く，その際に変数名が歪んでいると新たに識別子歪みが混入したと判断されてしまう．また，C 言語における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目視で確認した結果，Java や Python よりも構造的歪みのサイズが大きく，数が多い傾向が見受けられた．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追加をはじめ，変数の宣言および初期化の位置が統一される歪みや配列がポインタを用いて表される歪み，for 文が while 文に変換される歪みなど C 言語では構文が大きく変わる歪みが多い．構造が複雑に歪む場合，識別子歪みは構造的歪みに取り込まれてしまう．具体的には識別子の周囲の構文が大きく歪んでいる場合を考える．この際，本研究で歪み検出に用いている GumTree の性質上，抽象構文木の差分は識別子名の更新ではなくその識別子を含めた周囲の文の削除と新たな文の挿入と表される傾向がある．識別子名の更新以外は構造的歪みに分類されるため，元コードの変数と同様な役割を持つ変数の名前が歪んでいても，構造が大きく異なれば構造的歪みとして検出されてしまう．その一方で，修正コードでは復元コードと比較して構造が元コードに類似するため，復元コードにおいて構造的歪みに取り込まれていた識別子歪みが新たに検出され，混入率が高くなる．

次に，Python では Java と比べて修正コードのコンパイル可能率が高い原因について考察する．表 8 に Java のコンパイル不可能な修正コードにおいて発生したコンパイルエラーのうち，発生数の上位 5 件とその主な原因を示す．発生数 1 位のエラー（cannot find symbol）と発生数 3 位のエラー（variable X is already defined）は Python では起こり得ない．Python では識別子歪みが発生せず，変数名の誤りや重複が起こらないためである．発生数 2 位のエラー（X expected）と発生数 4 位のエラー（reached end of file while parsing）は Python では起こりにくいと考えられる．Python のソースコードはインデントでコードブロックが区切られるため，区切り文字の過不足は起こらない．さらにソースコードが Java よりも比較的簡潔なことから，括弧の対応関係を事前学習済みモデルが学習しやすく，閉じ括弧の不足が起こりにくいと推測できる．最後に発生数 5 位のエラー（incompatible types）についてだが，Python は動的型付け言語であり型の不一致が発生する可能性は低い．以上から Python では Java で多く見受けられたコンパイルエラーが発生しにくいため，修正コードのコンパイル可能率が Java より高いと考えられる．

表 8: Java の修正コードにおける主なコンパイルエラー [1]

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

7　妥当性への脅威

転移学習を用いたデコンパイラ歪み修正手法の汎用性を評価するにあたり，競技プログラミングの解答ソースコードを収集した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平均数十行程度と比較的行数が少なく [21]，単純な構文が多い．そのため，より行数が多く複雑なソースコードを含むデータセットを用いた場合，得られる結果が異なる可能性がある．さらに，実験ではファインチューニングを行う際に GPU メモリが不足することを防ぐため，サイズが 2KB 未満のソースコードのみを利用している．2KB 以上のソースコードも含めてファインチューニングを行った場合，歪み修正性能が異なる可能性がある．

本研究では C 言語と Python のどちらに対しても 1 種類のデコンパイラしか使用していない．他のデコンパイラに対する歪み修正性能は異なる可能性がある．特に C デコンパイラの一種である Hex-Rays[*3]は本研究で用いた Ghidra と比べて性能が良く，再コンパイル可能率が高いことで知られている [25]．先行研究においてデコンパイラに対する手法の汎用性を示した際に用いられた 2 種類の Java デコンパイラ（CFR，Fernflower）は，性能に大きな差が無い [5]．そのため，Ghidra と Hex-Rays のように性能が異なる場合，生じる歪みに対する修正性能が同等であるとは限らない．

本研究において，モデルにファインチューニングを行う際のハイパーパラメータはエポック数とバッチサイズのみ調節した．エポック数は損失関数の値が収束し始めた回数を，バッチサイズはメモリ不足が発生しない最も大きい値を選択した．他のハイパーパラメータについても調節を行うことで歪み修正モデルの性能が変化し，得られる評価結果が変わる可能性がある．

また，本研究では歪みを検出する際に，GumTree という AST 差分検出ツールを利用した．GumTree の出力は直感に沿わないケースがある．そのため，実験において過剰に歪みを検出してしまっている場合や，残存であるはずの歪みが混入歪みとして検出されてしまう場合が考えられる．より精度の高い AST 差分検出方法を用いた場合，異なる評価結果が得られる可能性がある．

[*3] `https://hex-rays.com/decompiler`

8 関連研究

デコンパイラの歪みを修正する研究が存在する．Lacomis らは機械学習を用いた識別子名復元手法である DIRE を提案している [26]．DIRE はソースコードから語彙情報（トークン化されたコード）と構造情報（抽象構文木）をそれぞれ取得し，識別子名の復元に利用する．ただし，DIRE は特定のデコンパイラ上で動作するプラグインとして設計されており，他のデコンパイラに対してはうまく機能しないという課題がある．機械学習を用いた他の識別子名復元手法として，Nitin らによって提案された DIRECT がある [27]．DIRECT は Transformer ベースのモデルの活用により，DIRE よりも高い精度で識別子名の復元に成功している．DIRE と DIRECT はいずれも識別子歪みの修正のみを目的とした手法であり，構文に関する歪みは対象としていない．

Hu らは LLM を活用することで特定のデコンパイラに依存せずにデコンパイラ出力を改善する，DeGPT という手法を提案している [28]．DeGPT はプロンプトエンジニアリングをベースに実現されており，識別子名だけでなくプログラム構造に対しても改善を行い可読性を向上させる．しかしながら，DeGPT を構成する機構の一部で用いられている構文解析器は特定の言語に依存しているため，他の言語に対する適用は困難と考えられる．他に LLM を活用する手法として，Wong らが DecGPT を提案している [29]．DeGPT と同様にプロンプトエンジニアリングがベースだが，目的が異なる．DecGPT は，コンパイル可能かつ機能等価なソースコードへデコンパイラ出力を修正することを目的とした手法である．したがって可読性に関する評価は行われていない．また，本研究で用いている歪み修正手法と同様に DecGPT は特定のデコンパイラや言語に依存しない手法だが，他のデコンパイラや言語での評価は行われていない．

9　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，二つ挙げられる．一つ目は学習に用いるデータの見直しである．本研究で用いたデータセットは複数の開発者が作成したソースコードから成るため，変数名の命名方法に一貫性がない．変数名の命名方法や同じ変数名の使用制限を設けることにより，識別子歪みに対する修正性能やコンパイル可能率が向上する可能性がある．また，本稿の実験ではソースコード全体を学習単位としたが，より規模が小さい関数単位での学習によって区切り文字の対応関係が取りやすくなり，コンパイル可能率や構造的歪みに対する修正性能が改善できる可能性がある．さらに，学習データに含まれる構文の種類を調整し，含まれる数が少ない歪みや複雑な歪みをより多く学習することで，構造的歪みに対する修正性能を向上させられると考える．二つ目は他の歪み修正手法との比較である．機械学習を利用した識別子歪みの修正手法 [26][27] や，LLM を用いた他の歪み修正手法 [28][29][30] が存在する．これらの手法との性能比較を検討している．

謝辞

本研究を遂行するにあたり，多くの方々にご指導とご支援を賜りました．

楠本真二教授には，研究に対して客観的なご意見をいただき大変参考になりました．特に中間報告会と修士論文発表の練習会では，質疑によって自分では考えが至らなかった点に気づかせていただき，研究の改善に繋がりました．また，差し入れ等のご支援は円滑な研究活動に繋がりました．冬にお持ち下さった蜜柑の味は忘れられません．心より感謝申し上げます．

柗本真佑准教授には，研究活動の全ての過程を通して丁寧で的確なご支援をいただきました．研究方針の相談では私の意を汲んだご対応をして下さりました．毎週の進捗報告や個別相談の場では理に適ったご指摘をしていただき，方針が逸れている場合にはその都度気づかせて下さりました．論文執筆や発表準備の際には，読者や聴衆に分かりやすく伝えるための様々なご助言をいただきました．また，社会人として必要な多くの学びを得ることができ，研究活動を行う上での基礎力が養われました．人としての礼儀や伝える力，スケジューリングの大切さ，大人としての考え方などを根拠も含めてお教えいただいたことは人生を通して有用な財産です．深く感謝申し上げます．

事務補佐員の橋本美砂子氏には研究活動を行うにあたり，備品の購入をはじめとした研究環境に関する多くのご支援をいただきました．環境作成以外の点においても，出張手続きや授業に関する事務処理，大学生活上の必要手続きなど様々な点でご支援いただき，ストレスなく研究活動を行うことができました．入学当初には気さくに話しかけていただいたおかげで，修士課程からの入学ということもあり強く感じていた新生活への不安が薄れました．心より感謝いたします．

楠本研究室の同期の方々には，あらゆる面から研究室生活を支えていただきました．研究活動に行き詰まった時は親身に寄り添って貴重な意見をくださり，修士課程からの配属である私にとっては特に大きな助けとなりました．疲れた時は他愛もない会話によって活動を続けていくための英気を養うことができました．優秀かつ心優しい同期の方々はかけがえのない存在です．深く感謝いたします．

楠本研究室の後輩方は，研究で困った際に心優しく相談に乗って下さりました．発表準備や研究方針に悩んだ際は，多くの時間を費やして価値のあるご意見やご指摘をいただきました．論文執筆の際には，執筆に不慣れな私に対して丁寧な添削をして下さりました．先輩からの頼みは断り辛く，忌憚なく意見を述べるというのも難しいかと思います．その上で真摯な対応をしていただいたことに感謝いたします．大変お世話になりました。

最後に，人生において誰よりも長く，そして近くで自分を見てきた存在である家族に感謝いたします．生活の援助や時折かけていただいた励ましの言葉は，二年間の研究活動において大きな支えとなりました．

研究活動を行うにあたりお世話になった全ての方々に心より感謝の意を表します．

参考文献

[1] 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).

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

[3] Durfina, L., Kroustek, J. and Zemek, P.: PsybOt malware: A step-by-step decompilation case study, in *Working Conference on Reverse Engineering*, pp. 449–456 (2013).

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

[5] 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*, pp. 92–102 (2019).

[6] 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).

[7] Krizhevsky, A., Sutskever, I. and Hinton, G. E.: ImageNet classification with deep convolutional neural networks, *Communications of the ACM*, Vol. 60, No. 6, pp. 84–90 (2017).

[8] Purwins, H., Li, B., Virtanen, T., Schlüter, J., Chang, S.-Y. and Sainath, T.: Deep Learning for Audio Signal Processing, *IEEE Journal of Selected Topics in Signal Processing*, Vol. 13, No. 2, pp. 206–219 (2019).

[9] Sutskever, I., Vinyals, O. and Le, Q. V.: Sequence to sequence learning with neural networks, in *International Conference on Neural Information Processing Systems*, p. 3104 – 3112 (2014).

[10] Gu, J., Neubig, G., Cho, K. and Li, V. O.: Learning to Translate in Real-time with Neural Machine Translation, in *Conference of the European Chapter of the Association for Computational Linguistics*, pp. 1053–1062 (2017).

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

[12] 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, `https://arxiv.org/abs/1907.11692` (2019).

[13] 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*, Vol. 33, pp. 1877–1901 (2020).

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

[15] Koutsokostas, V. and Patsakis, C.: Python and Malware: Developing Stealth and Evasive Malware Without Obfuscation, in *International Conference on Security and Cryptography*, pp. 125–136 (2021).

[16] Wiedemeier, J., Tarbet, E., Zheng, M., Ko, S., Ouyang, J., Cha, S. K. and Jee, K.: PYLINGUAL: Toward Perfect Decompilation of Evolving High-Level Languages, in *Symposium on Security and Privacy (SP)*, pp. 52–52 (2025).

[17] You, K., Bai, R., Cao, M., Wang, J., Stoica, I. and Long, M.: depyf: Open the Opaque Box of PyTorch Compiler for Machine Learning Researchers, `https://arxiv.org/abs/2403.13839` (2024).

[18] Liu, Z. and Wang, S.: How Far We Have Come: Testing Decompilation Correctness of C Decompilers, in *International Symposium on Software Testing and Analysis*, pp. 475–487 (2020).

[19] 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*, pp. 20–30 (2018).

[20] 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*, pp. 313–324 (2014).

[21] 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*, Vol. 48, No. 4, pp. 1268–1289 (2022).

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

[23] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I.: Attention is all you need, in *International Conference on Neural Information Processing Systems*, pp. 6000–6010 (2017).

[24] Jiang, N., Liu, K., Lutellier, T. and Tan, L.: Impact of Code Language Models on Automated Program Repair, in *International Conference on Software Engineering*, pp. 1430–1442 (2023).

[25] Cao, Y., Zhang, R., Liang, R. and Chen, K.: Evaluating the Effectiveness of Decompilers, in *International Symposium on Software Testing and Analysis*, pp. 491–502 (2024).

[26] Lacomis, J., Yin, P., Schwartz, E. J., 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*, pp. 628–639 (2020).

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

[28] Hu, P., Liang, R. and Chen, K.: DeGPT: Optimizing Decompiler Output with LLM, in *Network and Distributed System Security Symposium*, pp. 1–27 (2024).

[29] Wong, W. K., Wang, H., Li, Z., Liu, Z., Wang, S., Tang, Q., Nie, S. and Wu, S.: Refining Decompiled C Code with Large Language Models, *CoRR*, Vol. abs/2310.06530, (2023).

[30] Tan, H., Luo, Q. and Zhang, Y.: LLM4Decompile: Decompiling Binary Code with Large Language Models, in *Conference on Empirical Methods in Natural Language Processing*, pp. 3473–3487 (2024).