ソースコードの構文木表現による構造類似性を用いた自動関数生成方式

北 椋太† 岡田 龍太郎† 峰松 彩子† 中西 崇文†

†武蔵野大学データサイエンス学部データサイエンス学科 〒135-8181 東京都江東区有明 3-3-3

E-mail: s2022007@stu.musashino-u.ac.jp,

{ryotaro.okada, ayako.minematsu, takafumi.nakanishi}@ds.musashino-u.ac.jp

あらまし 本稿では，ソースコードの構文木表現による構造類似性を用いた自動関数生成方式について示す．プログラミングにおいて構造を簡略化することは，生産性やのちの保守性の向上において重要であり，これを自動化することは，プログラマにとって労力の軽減につながる．構造を簡略化する方法の一つとして，コードクローンを関数としてまとめることがあげられる．本方式は，任意のソースコードを入力として，入力したソースコードを構文木として表現した上で，各部分木間の類似度を計量することにより，コードクローンを検出し，新たに生成した関数に置き換える．一般的に，関数化によるリファクタリングでは，構造の一致するコードクローンでなければ適用することができないが，本方式では，構造を部分的に抽象化することで，適用範囲を拡張し，完全一致でない類似したコードクローンに対する構造の簡略化を図る．本方式を実現することで，より簡略化した保守性の高いソースコードを書くことが可能になる．

キーワード コードクローン，ソースコード，リファクタリング，構文木，関数生成

1. はじめに

近年，ソフトウェアの利用分野の拡大にともない，プログラムの不具合やそれにともなうシステムの故障が社会的に問題となることが多くなっている．ソフトウェア開発ライフサイクル全体において，保守作業のコストが占める割合は非常に高く，プロジェクトの大規模化にともなって，その割合はさらに増加する．そこで，ソフトウェアの保守作業の効率化が重要な課題となっている．ソフトウェアの開発や保守作業を行うにあたり，開発者は，ソースコードからプログラムの構造を理解し，修正する必要があり，この作業がソフトウェアのメンテナンスの多くの割合を占めている．そのため，ソフトウェアの保守作業に要するコストを削減するためには，ソースコードの理解や，修正を容易にすることが重要である．これらの作業を困難にする要因の一つとして，コードクローンがあげられる．コードクローンとは，ソースコード中の一部分(コード片)のうち，類似または一致したコード片がソースコード中に複数存在するもののことである．あるコード片にて欠陥が発見された場合には，そのコード片のすべてのコードクローンを探し，検査や修正を検討する必要がある．しかし，大規模なソフトウェア開発において．ソースコード中のコードクローンを手作業で探し，修正を行うには，非常に大きな労力が必要となる．一般にコードクローンを検出する方法として，キーワード検索や，コードクローン検出ツールが用いられる．

キーワード検索では，欠陥を含むコード片から抽出したキーワードから，grep[1]などを用いて検索する．しかし，キーワード検索では，キーワードと完全に一致するコード片を出力するため，空白やコメントを含む場合や，識別子が異なる場合では検出することが難しい．また，効率的な検索を行うための適切なキーワードを選定するためには，対象となるソースコードを十分に理解している必要がある．一方，CCFinder[2]などのコードクローン検出ツールでは，空白やコメントを含むコードクローンや，識別子が異なるコードクローンにおいても検出が可能であるが，構造の異なるコードクローンを検出することは難しい．

本稿では，ソースコードを構文木で表現することで，構文木内の類似した構造を含むコード片を検出し，抽出したコードクローンを再利用可能な関数に置き換える手法を提案する．

本方式では，構文木内の構造に基づいてコードクローンを検出することにより，識別子の違いなどの処理に影響を及ぼさない差異を無視したコードクローンの検出を可能にする．検出したコードクローンのうち，構造が完全に一致するものを関数に置き換えることにより，コードクローンの解消を実現する．また，構造が異なる類似したコードクローンに対して，構造を部分的に抽象化し，構造を同一化することで，本方式の適用範囲を拡張する．

本稿の構成は，次の通りである．2 章では，関連研

究について紹介する．3 章では，本提案方式であるソースコードの構文木表現による構造類似性を用いた自動関数生成方式について示す．4 章では，本方式を実現するシステムを構築し，実験を行う．5 章で本稿をまとめる．

2. 関連研究

本章では，本方式に関連する研究について述べる．本節の構成は，次の通りである．2.1 節では，コードクローン検出に関する研究について述べる．2.2 節では，ソースコードの簡略化に関する研究について述べる．

2.1. コードクローン検出に関する研究

コードクローン検出の研究は，ソースコードをさまざまなモデルで表現することにより実現されてきた[3]．

本節では，主にテキストやトークンの並びでソースコードを表現する字句を用いた検出手法と，構文木やグラフでソースコードを表現する構文を用いた検出手法の二つに分類し，関連する研究について述べる．

2.1.1. 字句を用いた検出手法に関する研究

神谷ら[2]は，プログラムテキストをトークンの並びで表現し，そのトークン列の等価性に基づいてコードクローンを検出する CCFinder を開発した．

吉田ら[4]は，コード片に含まれる識別子の類似性に基づいてコードクローンを検出する手法を提案している．この研究では，ソースコード内の関数に含まれる語を抽出し，共起関係に基づいて類義語を特定し，入力コード片が含む語と一致する，もしくは類似する語を含む関数を提示している．

2.1.2. 構文を用いた検出手法に関する研究

Gabel ら[5]は，ソースコードから program dependence graph(PDG)の形で構造化することにより，コード片の検出を部分グラフの類似度計量の問題にすることでコードクローンを検出する手法を提案している．

Jiang ら[6]は，ソースコードを抽象構文木で表現し，木構造の類似度を算出することでコードクローンを検出する DECKARD を開発した．

2.2. ソースコードの簡略化に関する研究

肥後ら[7]は，複数のオブジェクトに共通するメソッドを親クラスへ引き上げる手法や，コードクローンをメソッドとして定義する手法を提案している．

石原ら[8]は，ファイル単位にまたがって存在するコードクローンのライブラリ化を目的に，大規模なソフトウェア群を対象に，メソッド単位のコードクローンを検出する手法を提案している．

Baxter ら[9]は，抽象構文木を用いたコードクローンの検出手法を提案する中で，部分木の類似度に対して任意の閾値を設定し，閾値以上の類似度を持つコードクローンをマクロとしてまとめ，それを更に再帰的に繰り返すことで，より大きな構造に対してコードクローンを検出することを可能にした．

2.3. 本研究の位置付け

本研究では，入力されたソースコードを構文解析し，抽象構文木で表現する．抽象構文木内の各ノードには，親子ノードやトークンの属性をあらわす抽象文法名などの情報が保持されている．本研究では，さらに，各ノードに，そのノードの高さを保持している．この情報を用いて，任意の高さ以上の部分木を抽出したのちに，一致または類似する構造を含む部分木の組み合わせを検出する．検出した部分木に該当するコード片をコードクローンとみなし，これを関数に置き換えることによりコードクローンを解消する．

これまでのコードクローン検出に関する研究では，類似するコード片を検出し，位置情報または抽出したコード片を出力とすることが主であった．そのため，あるコード片にて不具合が発見された場合には，コードクローンの検索は効率化されるものの，最終的には手作業で修正を行う必要がある．それを解消するために提案されてきたリファクタリング手法についても，既にメソッドとして定義されているコード片や，完全に一致している構造をもつコードクローンを対象としている．これにより，ほとんどの手法では識別子レベルの差異までしか対応することができない．一方，本研究では，抽象構文木を用いて検出したコードクローンに対して，部分的に抽象化を行うことでリファクタリングの適用範囲を拡張した．また，本研究では抽出する部分木の高さをパラメータとして任意の値で設定することで，コードクローンを検出するスコープを変更することが可能である．

3. ソースコードの構文木表現による構造類似性を用いた自動関数生成方式

本章では，提案方式である，ソースコードの構文木表現による構造類似性を用いた自動関数生成方式について示す．

3.1. 提案手法の概要

本節では，提案手法の全体像について述べる．提案システムの全体像を図 1 に示す．

本方式では，入力データとして任意のソースコードを与え，構文解析を行い，抽象構文木を生成する．生成した抽象構文木から，指定した高さパラメータ以上の高さである部分木をすべて抽出し，部分木の集合を作成する．そして，部分木間の類似度を計量し，類似度の高いものをコードクローンとして検出する．その後，類似度が 0 より大きく，任意の閾値 ε 以下である場合については，差異部分を抽象化することにより，構造を同一化する．抽象化したコードクローンおよび

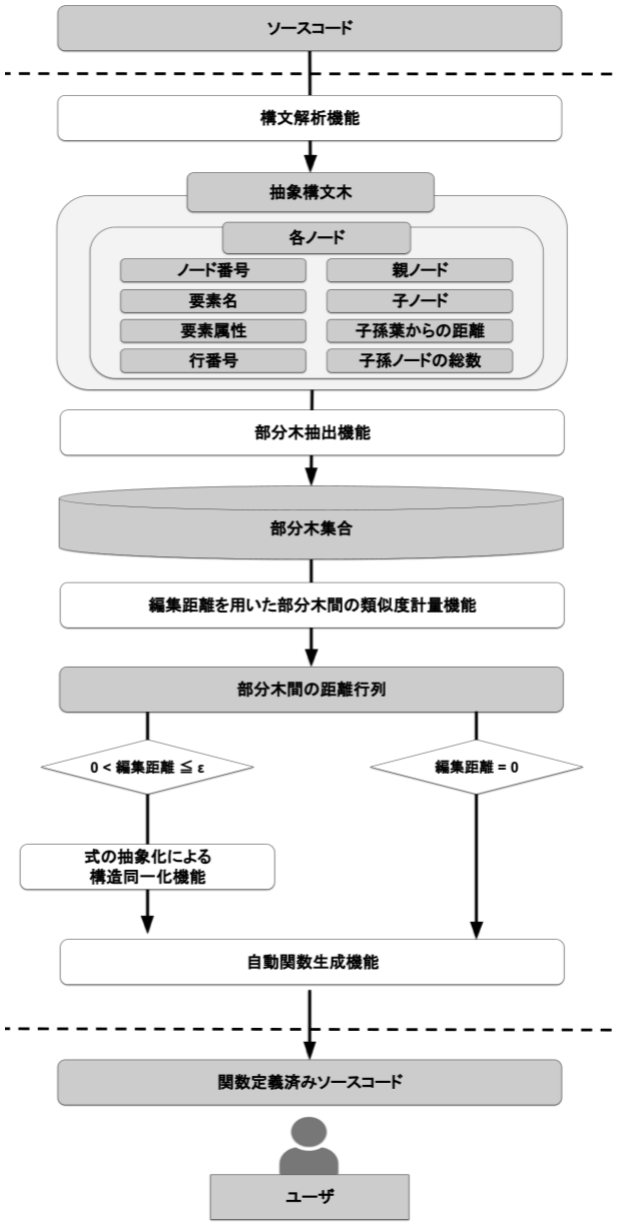

図 1 提案システムの全体像

類似度が 0 であったコードクローンから関数を生成し，置き換える．

本手法は，構文解析機能，部分木抽出機能，編集距離を用いた部分木間の類似度計量機能，式の抽象化による構造同一化機能，自動関数生成機能からなる．

3.2. 構文解析機能

本節では，ソースコードから抽象構文木を生成する構文解析について述べる．構文解析は，ソースコードから要素の最小単位であるトークンを取り出す字句解析を行なったのち，そのトークン間の関係を明確にする手続きのことである．構文解析の結果を木構造で表現したものが構文木であり，特にトークンの詳細な情報を取り除き，動作の意味に関係ある情報のみを取り出した構文木を抽象構文木と呼ぶ．本方式では，構文木の構造から類似度計量を行うため，識別子などの動作に関与しない要素を省略した抽象構文木を用いる．また，本方式で生成する抽象構文木の各ノードには抽象文法名や親子ノードの他に，部分木の抽出時に使用する構文木の高さや，関数生成時に構文木からソースコードへ変換するために使用する行番号や列番号といった位置情報，プログラムテキスト，根や葉といったノードの位置属性を保持する．

3.3. 部分木抽出機能

本節では，抽象構文木から部分木を抽出する方法について述べる．関数生成を行うにあたって，小さなコード片を関数に置き換えても可読性や保守性の向上には繋がらないため，関数化の対象となるコードクローンは，ある程度の大きさの処理のまとまりである必要がある．そこで，本方式では，抽出する最小高さを設定することで，指定した最小高さ k 以上の部分木のみを抽出し，部分木の集合を作成する．これを実現するため,各ノードは子ノードを再帰的に探索することで,葉との最大距離を算出し，部分木の高さとして保持する．また，本方式の自動関数生成機能では，行の一部分だけを関数に置き換えることができない.そのため,根ノードが文となる部分木のみを抽出する．文とは，プログラムの部分構造であり，トークンを組み合わせることで処理を行う実行単位である．

3.4. 編集距離を用いた部分木間の類似度計量機能

本節では，作成した部分木の集合から各部分木間の類似度を計量する方法について述べる．構文木同士の類似度を計量する手法として，完全一致する部分木の個数から計量する手法や，部分木の編集距離から計量する手法などがあげられる．本方式では，構造の差異を明確にするために，追加，削除，更新の操作の最小回数である編集距離を用いて類似度を計量する手法を用いる.本システムでは,編集距離の算出手法として,Zhang と Shasha によって提案されたアルゴリズム[10]を用いる．3.3 節で抽出した部分木の組み合わせに類似度計量を行い，部分木間の距離行列を作成する．

3.5. 式の抽象化による構造同一化機能

本節では，類似度が 0 より大きく，任意の閾値 ε 以下である部分木間の構造を同一化する方法について述べる．自動関数生成は，基本的には部分木が完全一致するものについてしか行うことが出来ない．しかし，類似する部分木の一部を変形することによって同一の構造に変換することができれば，自動関数生成を適用する範囲を拡張することが可能である．ここでは，任意に定める閾値 ε 以下の編集距離を持つ完全一致ではない部分木の組のうち，式に対応する部分木を置換す

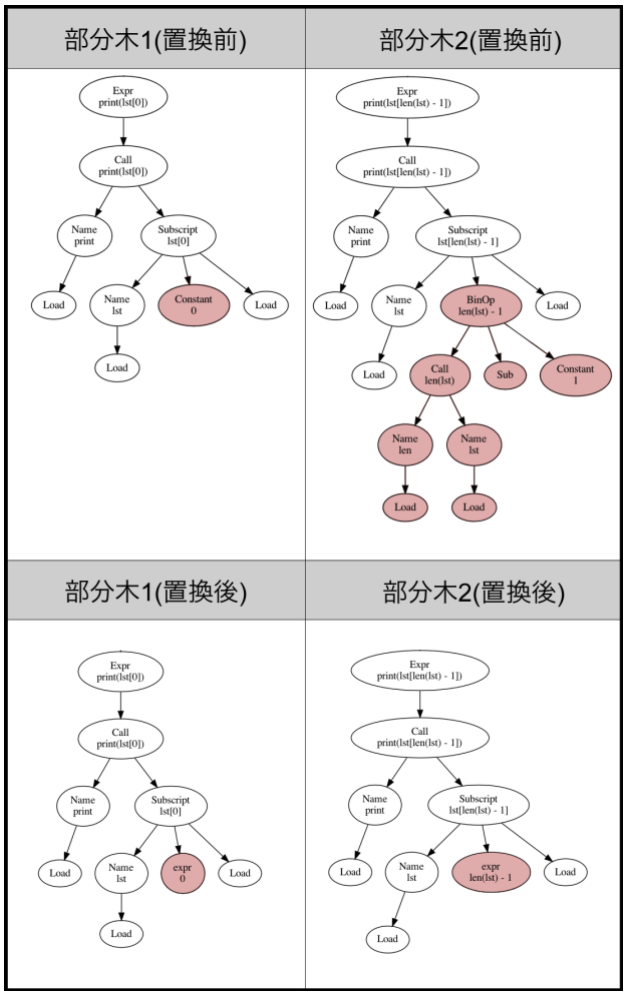

図 2 式の抽象化による構造同一化機能の実行例

ることによって構造の同一化を実現する．式とは，プログラムの部分構造であり，評価後に値に変換されるものである．例えば，変数やリテラル，それを組み合わせた演算子式などが式にあたる．本手法は，3.4 節で編集距離の算出時に編集操作を行なったノードを対象とする．また，編集操作を行なったノードが連なっている場合，編集操作を行なったノード群のうち，最上位に位置するノードを対象とする．本手法は，対象となるノードが式であり，編集操作の前後で対象となるノードの兄弟ノード数が変化していない場合にのみ本手法を適用することができる．前述した条件を満たす部分木の組を，新たに用意した式をあらわす抽象文法名を持つノードに置換する．本手法の実行例を図 2 に示す．

3.6. 自動関数生成機能

本節では，構造が完全に一致する部分木から関数の生成を行う方法について述べる．はじめに，ソースコードの先頭に引数，処理，戻り値を空欄とした関数部

```
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3

for _ in range(cannon1):
    if rate1 > random.random():
        cannon2_ -= 1
    if cannon2_ == 0:
        break
for _ in range(cannon2):
    if rate2 > random.random():
        cannon1_ -= 1
    if cannon1_ == 0:
        break

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

図 3 一致するコードクローンを含むソースコード

```
# Exchange Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n):
   for j in range(i+1, n):
      if A[j] < A[i]:
         A[i], A[j] = A[j], A[i]
print(A)

# Bubble Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n-1):
   for j in range(n-1):
      if A[j+1] < A[j]:
         A[j], A[j+1] = A[j+1], A[j]
print(A)
```

図 4 類似するコードクローンを含むソースコード

のテンプレートを挿入する．次に，構造が完全に一致する部分木のノードが保有するプログラム上の位置情報を参照し，それぞれの部分木に該当するコード片を抽出し，削除したのち，関数の呼び出し文を挿入する．抽出したコード片に含まれる式のうち，すべての変数および二つの部分木間で識別子が異なる式を，関数部で使用する変数として定義し，置換する．置換したコード片を関数部のテンプレートの処理部に加える．関数部で使用する変数として定義した式のうち，定数およびプログラム上で既に値が割り当てられているものを関数の呼び出し文の引数として使用する．また，定

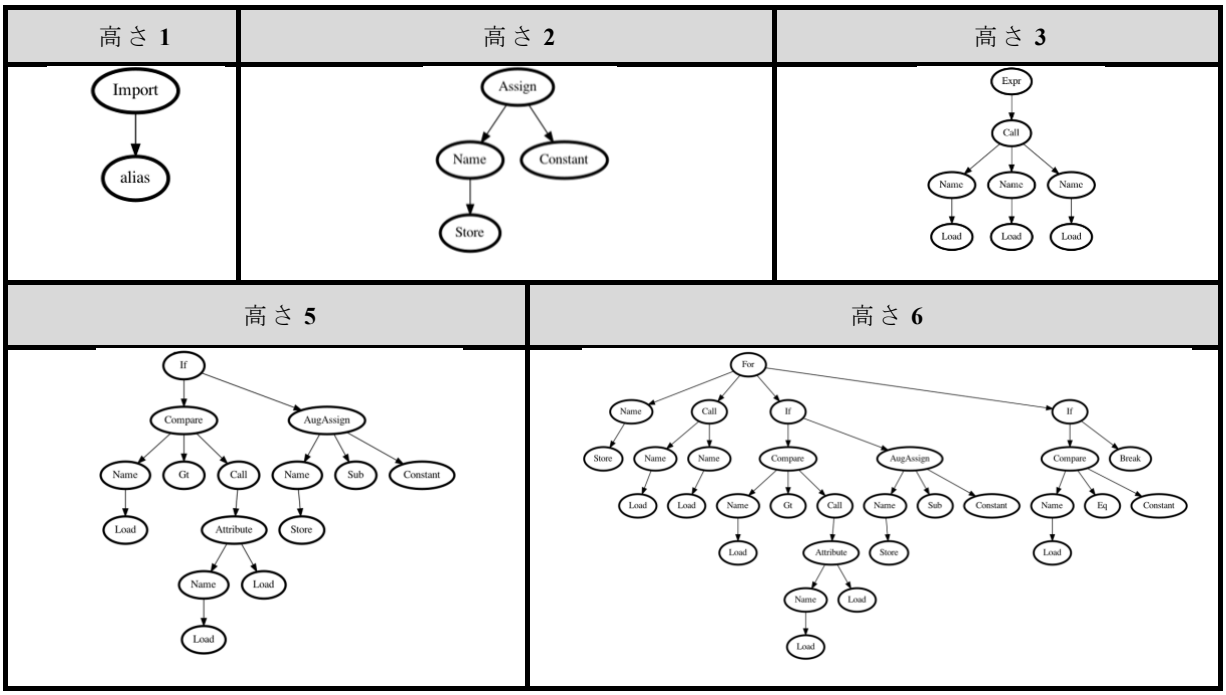

図 5　実験 1 で抽出した抽象構文木の部分木の例

義した変数を関数で使用する引数として定義し，関数部のテンプレートの引数部に加える．次に，コード片の処理において，値の再代入やメソッドの呼び出しなどにより値に何らかの影響を受ける変数を関数の呼び出し文の被代入変数として使用する．また，関数で使用する戻り値として定義し，関数部のテンプレートの戻り値部に加える．

4. 実験

本章では，本手法の評価実験について述べる．4.1 節では実験環境について述べる．4.2 節では，実験 1 として，用意した一致するコードクローンを含むソースコードに対して構文解析機能を適用し，抽出された構文木を観察することで，関数における置き換えのための最適な高さについて検討する．4.3 節では，実験 2 として，提案方式である自動関数生成方式を一致するコードクローンを含むソースコードに対して適用し，提案方式の有効性を検証する．その際，置換する部分木の高さパラメータを変化させて実験を行い，最適な高さについて検討を行う．4.4 節では，実験 3 として，提案方式である自動関数化方式を類似するコードクローンを含むソースコードに対して適用し，提案方式の有効性を検証する．

4.1. 実験環境

本方式は Python を用いて構築した．

実験 1 および実験 2 の対象である一致するコードクローンを含むソースコードを図 3 に示す．抽象構文木の抽出には Python の標準ライブラリである ast を用いた．実験 3 の対象である類似するコードクローンを含むソースコードを図 4 に示す．

4.2. 実験 1

4.2.1. 実験目的

実験 1 として，用意したソースコードに対して構文解析機能を適用し，抽出された構文木を観察することで，関数化における置き換えのための最適な高さについて検討する．

4.2.2. 実験結果

構文解析機能によって抽出された抽象構文木のうち，各高さの部分木の例を図 5 に示す．ノード数 112，最大高さ 7 の抽象構文木が生成された．本ソースコードから抽出した抽象構文木のうち，主要な処理をあらわす部分木における高さについて，ライブラリのインポートは高さ 1，変数の定義は高さ 2，packing を使用した変数のスワップは高さ 3，for 文による繰り返し処理は高さ 6，if 文による条件分岐は高さ 5，print 関数による標準出力は高さ 3 となった．

4.2.3. 考察

本研究の目的は，コードクローンを検出し，関数に置き換え，一元的に管理することで保守作業の効率を上げることである．図 5 を確認すると，高さ 3 以下の部分木は，ソースコードにおける代入文などの 1 行の処理を表している．これらを関数に置き換えることは，

(関数部)

```
def function1(var1, var2):
    if var1 > random.random():
        var2 -= 1
    return var2
```

(実行部)

```
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3

for _ in range(cannon1):
    cannon2_ = function1(rate1, cannon2_)
    if cannon2_ == 0:
        break
for _ in range(cannon2):
    cannon1_ = function1(rate2, cannon1_)
    if cannon1_ == 0:
        break

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

(a) k = 5 の場合

(関数部)

```
def function1(var2, var3, var4):
    for var1 in range(var2):
        if var3 > random.random():
            var4 -= 1
        if var4 == 0:
            break
    return var4
```

(実行部)

```
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3

 cannon2_ =   function1(cannon1, rate1, cannon2_)
 cannon1_ =   function1(cannon2, rate2, cannon1_)

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

(b) k = 6 の場合

図 6 実験 2 の実行結果

(関数部)

```
def function1(var1, var2, var3):
  if var1[var2] < var1[var3]:
    var1[var3],var1[var2] = var1[var2],var1[var3]
  return var1
```

(実行部)

```
# Exchange Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n):
  for j in range(i+1, n):
    A = function1(A, j, i)
print(A)

# Bubble Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n-1):
  for j in range(n-1):
    A = function1(A, j+1, j)
print(A)
```

図 7 実験 3 の実行結果

保守作業の効率化に有用でない．そのため，実験 1 より，本ソースコードにおいて抽出する部分木の最大高さは 5 以上が適切である．

4.3. 実験 2

4.3.1. 実験目的

実験 2 として，一致するコードクローンをソースコードに自動関数生成機能を適用し，提案方式の有効性を検証する．また，本研究では，コードクローン検出の対象となる部分木について最大高さ k を任意の値として設定する．このとき，k の変化に対して，実行結果にどの程度違いがあるかを調査する実験を行った．

本実験では，実験 1 の結果を踏まえて最大高さ k を 5 としたときと，6 としたときの場合についてシステムを実行した．

4.3.2. 実験結果

図 3 に示した一致するコードクローンを含むソースコードに対して，k を 5 として自動関数生成機能を適用した結果を図 6 (a)に，k を 6 として自動関数生成機能を適用した結果を図 6 (b)に示す．k を 5 とした場合は，if 文による条件分岐文のみが関数化された．この

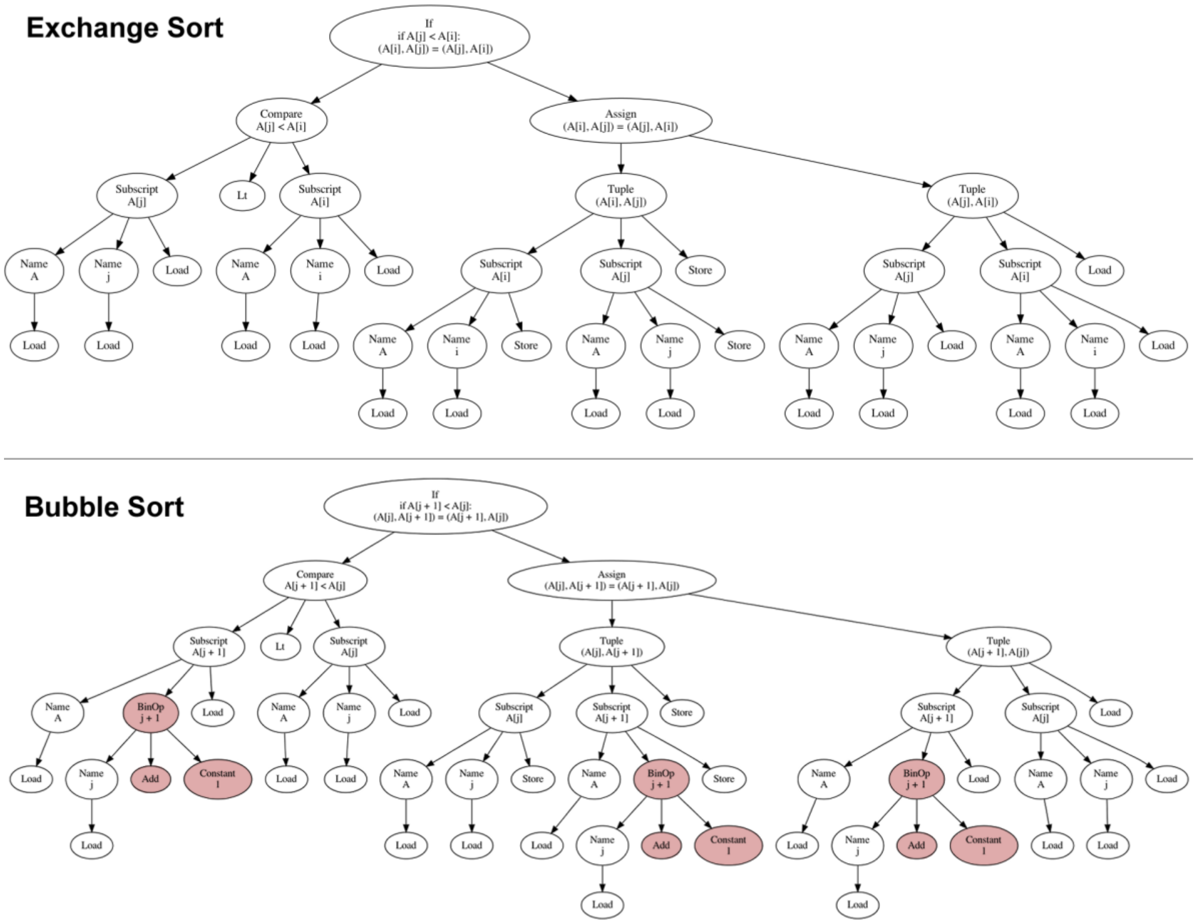

図 8 コードクローン部分の編集操作結果

とき，関数は二つの引数，一つの戻り値を取る．k を 6 とした場合には，先ほどの条件分岐文を含む for 文による繰り返し文が関数化された．このとき，関数は三つの引数，一つの戻り値を取る．関数化によってソースコードが簡略化できたかを考えるために，空行を除く行数を比較する．元のソースコードの行数は 15 行であった．k を 5 としたときのソースコードは，関数部 4 行，実行部 13 行の計 17 行となった．k を 6 としたときのソースコードは，関数部 7 行，実行部 7 行の計 14 行となった．

4.3.3. 考察

本実験では，k を 5 としたときと，k を 6 としたときの場合について，どちらも正常に置き換えることに成功した．また，機能のまとまりを関数に置き換えることに成功している．空行を除く行数を比較すると，k が 5 のときは行数の総計は増加したが，k が 6 のときは行数の総計を減少させることに成功した．これらのことから,適切な k の値を設定することができれば,ソースコードの保守性を向上させることが可能になると考えられる．

4.4. 実験 3

4.4.1. 実験目的

実験 3 として，類似するコードクローンをソースコードに自動関数生成機能を適用し，提案方式の有効性を検証する．

4.4.2. 実験結果

図 4 に示した類似するコードクローンを含むソースコードに対して，k を 5，式の抽象化による同一構造化機能を適用する閾値 ε を 10 として自動関数生成機能を適用した結果を図 7 に示す．結果として，値の大小比較および更新部分が関数化された．このとき，関数化した部分木間の編集距離は 9 であった．編集操作を行なった箇所については図 8 に示す．k を 6 以上にした場合，関数化されなかった．また，元のソースコードが 16 行であったのに対して，実行部は関数部 4 行，実行部 14 行の計 18 行となり，行数の総計としては増えたが，実行部の行数は減少した．

4.4.3. 考察

本実験では，k を 5 としたときについて，正常に置き換えることに成功した．また，機能のまとまりを関

数に置き換えることに成功している．行数の総計は増えてしまったが，実行部の行数は減っており，長いソースコードに対しては関数化の恩恵が得られる可能性があると考えられる．また，k を 6 以上としたときについて関数化されなかったのは，本ソースコード中の Exchange Sort の二つ目の for 文では range 関数の引数が二つあるのに対して，Bubble Sort では一つの引数であることが要因である．これを関数化する方法として，関数の取る引数に応じて，デフォルト引数を加えることがあげられる．関数の取る引数の数の違いに対応することで，より汎用的に関数化を行うことができる．プログラムの記述において，一般的にネストが深いほど可読性は低下するとされる．そのため，ネストを深める for 文を関数に置き換えることは可読性の向上に寄与すると考えられ，for 文で頻繁に用いられる range 関数の引数の違いに対応することは，ソースコードの簡略化に寄与すると考えられる．

5. まとめ

本稿では，ソースコードの構文木表現による構造類似性を用いた自動関数生成方式について述べた．本方式では，ソースコードを構文木で表現し，構造類似性に基づいて，コードクローンを検出した．その後，構造が完全に一致するコードクローンを生成した関数に置き換えることで，ソースコードの構造の簡略化を実現した．構造の一致しないコードクローンについても，部分的に抽象化することで構造を変形することで関数生成機能の適用範囲を拡張した．

また，本方式を検証するための実験システムを構築し，本方式の有効性を検証する実験を行った．

ソースコードの構文木表現による構造類似性を用いた自動関数生成方式を実現したことによって，同一機能をもつコードクローンを一元的に管理することが可能になり，保守作業を効率化させることができると考えられる．

今後の課題としては，関数の取る引数の違いに対応することと，抽出する部分木の高さや自動関数生成機能を適用する閾値のパラメータを自動調整する機能の実現，識別子の適切な命名機能の実現，本方式の適用前と適用後におけるソースコードの等価性検証があげられる．

参 考 文 献

[1] GNU Grep, http://www.gnu.org/software/grep/.

[2] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, 28(7), pp. 654-670, 2002.

[3] 神谷年洋，肥後芳樹，吉田則裕，“コードクローン検出技術展開”，コンピュータソフトウェア，28(3), pp.28‐42, 2011.

[4] 吉田則裕，服部剛之，早瀬康裕，井上克郎，"類義語の特定に基づく類似コード片検索法"，情報処理学会論文誌，50(5), pp.1506-1519, 2009.

[5] M. Gabel, L. Jiang, Z. Su, "Scalable detection of semantic clones", In Proceedings of the 30th International Conference on Software Engineering, pp. 321-330, 2008.

[6] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones", In 29th International Conference on Software Engineering, pp. 96-105, 2007.

[7] 肥後芳樹，吉田則裕，“コードクローンを対象としたリファクタリング”，コンピュータソフトウェア，28(4)，pp.43‐5, 2011.

[8] 石原知也，堀田圭佑，肥後芳樹，井垣宏，楠本真二，"大規模ソフトウェア群に対するメソッド単位のコードクローン検出"，電子情報通信学会技術研究報告，111(481), pp. 31-36, 2012.

[9] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, "Clone detection using abstract syntax trees", In Proceedings of International Conference on Software Maintenance, pp. 368-377, 1998.

[10] Kaizhong zhang, Dennis Shasha, "SIMPLE FAST ALGORITHMS FOR THE EDITING DISTANCE BETWEEN TREES AND RELATED PROBLEMS", In SIAM Journal of computing, 18(6), pp. 1245-1262, 1989.