Vol.2017-SE-196 No.20
2017/7/20

多言語対応のための字句解析機構を持つコードクローン検出ツールの開発

瀬村 雄一[1,a)]　吉田 則裕[2]　崔 恩瀞[3]　井上 克郎[1]

概要：字句単位のコードクローン検出ツールである CCFinder は，対象のソースコードの字句解析を行っている．対応する言語を増やすにはその言語の字句解析を実装する必要があるが，多くの言語に対して 1 つずつ字句解析を実装するのは，ツールの開発者に各言語への理解が求められ，また手間のかかる作業である．ツールの使用者が，新たな言語に容易に対応できる仕組みを作ることで，開発者の労力を減らすことができる．本研究では，字句解析において言語に依存する部分を，容易なオプションで変更可能な字句解析機構を提案する．適用実験では 175 言語のソースコードに対して，コメントに関するオプションを設定してツールを実行し，90 %以上の言語でコメント除去が可能なことを示した．また，予約語のリストを与えて実行することで，タイプ 2 までのコードクローンが検出可能なことを示した．

キーワード：コードクローン，字句解析，Ngram

A Clone Detection Tool with Flexible Multilingual Tokenization

Yuichi Semura[1,a)]　Norihiro Yoshida[2]　Eunjong Choi[3]　Katsuro Inoue[1]

1. はじめに

コードクローンとは，ソースコード中に存在する互いに一致，または類似したコード片を指す．コードクローンは，主に既存のコード片のコピーアンドペーストによって生成される [2]．その存在は，ソフトウェア保守を困難にしている大きな要因の 1 つとして挙げられている．あるコード片にバグが見つかった場合，そのコード片のコードクローンにもバグが含まれる可能性が高い．そして，開発者はあるコード片にバグが見つかった場合，一致または類似する箇所に対して同一の修正を行うか検討する必要がある [3]．そのため，開発者がコードクローンに関する情報を認識しておく必要があるが，大規模なソフトウェアにおいては，コードクローンにあたる箇所が膨大な数になることにより，その全てを認識しておくことは現実的でない．そこで，開発者を支援するためにコードクローン検出ツールが利用されている [1]．

コードクローン検出ツールの 1 つである CCFinder は，C, C++, Java, COBOL, Fortran の言語で書かれたソースコードのコードクローン検出に対応している [4]．CCFinder ではトークン単位のコードクローンを検出するための前処理として，ソースコードを言語の文法に沿ってトークン単位に分割している．この処理は一般的に字句解析と呼ばれる [8]．同じく CCFinderX は，CCFinder のバージョンアップとして開発されたコードクローン検出ツールであり，字句解析部のユーザによる変更を可能にしている [11]．これはコードクローン検出を行いたい言語に対応する字句解析部をユーザが用意することで，その言語のコードクローン検出が可能になるということを意味する．しかし，字句

[1] 大阪大学
Osaka University
[2] 名古屋大学
Nagoya University
[3] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[a)] y-semura@ist.osaka-u.ac.jp

解析部を用意するには言語の文法を理解することが必要であり，字句解析のコーディングは手間のかかる作業である．

CCFinder では字句解析によってトークンを分割した後に特定のトークンの変換を行っている．これは変数名や関数名などを全て 1 つのトークンに置き換える処理であり，実用的に意味のあるコードクローンだけを検出するために行われる．変数名や関数名で使用できるトークンは、プログラミング言語によって設定されているために，言語によって別の設定を行わなければならない．

本研究では多言語のトークン単位のコードクローン検出を目的として，ユーザによる容易に字句解析部の変更が可能になるような字句解析機構を提案する．字句解析機構の変更箇所を，主にコメント除去のルールと置き換えが必要な字句に限定した．

またコードクローン検出部の実装において高速化を図るため，コードクローン検出ツールである YOCCA のアルゴリズムを参考にした [5]．YOCCA は，ある 1 つのファイルと数百万行以上の規模を持つ巨大なソースコード群の間のコードクローンを高速に検出するツールである．YOCCA では N-gram という手法を使用して巨大なソースコード群をデータベースに保存して，コードクローン検出部の高速化を図っている．しかし，YOCCA は 1 つのファイルと巨大なソースコード群の間のコードクローンを高速に検出できるものの，全てのファイル間のコードクローン検出には適していない．今回は YOCCA で使用されていた N-gram を用いたアルゴリズムを参考に，全てのファイル間のコードクローン検出を行うツールを作成した．

本研究では，提案した柔軟かつ容易に変更可能な字句解析機構の適用実験として，多言語のソースコードに対して，コメント除去を行う前後のソースコードを比較を行った．この結果，約 92 %の言語でコメント除去が可能だった．また提案手法を用いて開発したツールを実行することで，全ての言語で正しくコードクローンが検出されていることを示した．

2 章では本研究の背景として，コードクローン検出ツールである CCFinder の説明を行う．3 章では本研究で提案する字句解析の手法と，N-gram を用いたコードクローン検出部のアルゴリズムについて説明する．4 章では開発したツールの有用性を示した適用実験をまとめ，5 章ではまとめと今後の課題について述べる．

2. コードクローン検出ツール:CCFinder

本章では本研究の背景として，コードクローン検出ツールである CCFinder についての説明を行う．

コードクローンには，普遍的定義は存在しない．本論文では，コードクローンの定義として以下の 2 つのタイプの分類を用いる [7].

タイプ 1

空白，タブ文字，改行やコメントなどを除いて一致するコードクローン．

タイプ 2

タイプ 1 の条件に加えて，リテラル，型，識別子を除いて一致するコードクローン．

CCFinder は，字句単位のコードクローンを検出するツールである [4]．CCFinder の特徴として以下のようなものが挙げられる．

- 変数名や関数名などのトークンを置き換えることで，タイプ 2 までのコードクローンを検出できる．
- 数百万行規模のシステムにも実用時間で解析可能である．
- 言語依存部分を取り替えることでさまざまな言語に対応している．
- しきい値を与えることで，トークン数がしきい値未満のコードクローンを検出しないようにできる．

ここで，CCFinder の処理概要について記述する．4 つの Step に分けられ，各 Step の詳細を説明する．

Step 1:字句解析

ソースコードをプログラミング言語の文法に沿ってトークン列に変換する．その際，空白とコメントは機能に影響しないので無視される．

Step 2:変換処理

分割されたトークン列から実用的に意味のあるコードクローンだけを検出するための変換を行う．これは変数名や関数名などを全て 1 つのトークンに置き換える処理である．

Step 3:クローン検出

変換されたトークン列を，比較しコードクローンを検出する．この比較には suffix-tree という木構造を用いたアルゴリズムを採用している．

Step 4:出力整形処理

最後に出力整形処理を行い，検出されたクローンペアについて，もとのソースコードでどのファイルに存在するか，行番号と列番号が出力される．

CCFinderX は CCFinder のバージョンアップであり，同様にトークン単位のコードクローンを検出するツールである．CCFinder と比べると性能が向上していて，字句解析部のユーザによる変更が可能になっている [11]．これはコードクローン検出を行いたい言語に対応する字句解析部をユーザが用意することで，その言語のコードクローン検出が可能になるということを意味する．しかし，字句解析部を用意するには言語の文法を理解することが必要であり，字句解析のコーディングは手間のかかる作業である．

本研究は，字句解析部を言語に合わせて容易に変更できるように開発し，多言語のコードクローン検出を行うことを目標とした．

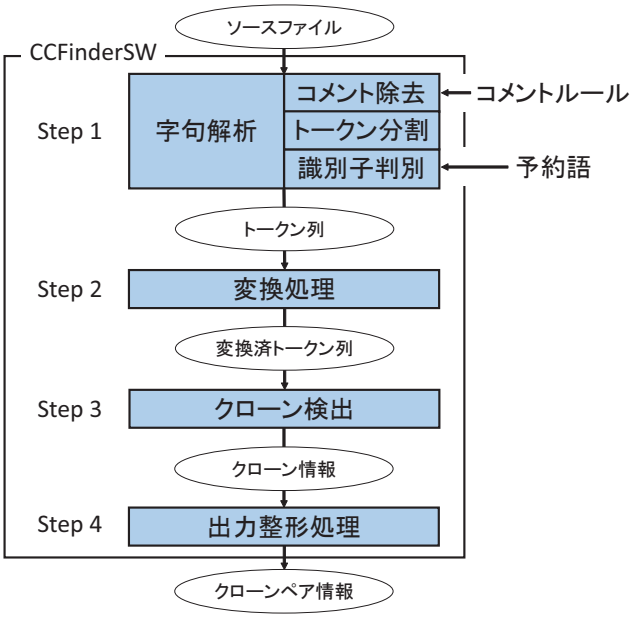

図 1　CCFinderSW の概要
Fig. 1　An Overview of CCFinderSW

3. 提案ツール：CCFinderSW

本章では，本研究で開発を行った容易に変更可能な字句解析機構を持つコードクローン検出ツールである CCFinderSW の処理概要と，その各ステップの詳細を述べる．ツールの実装は Java で行った．

図 1 は開発した CCFinderSW の処理概要である．CCFinder と同じく，タイプ 2 までの字句単位のコードクローンを検出する．CCFinder の処理概要を参考に実装を行っているため，おおまかな処理の順番は同じである．変更可能な字句解析部の実装における主要なポイントとして，コメント除去とトークン分割の方法を挙げる．

ツールを動かす際のオプションの入力として，コメントルールと予約語に限定した．コメントルールは字句解析部のコメント除去処理で使用し，予約語は識別子判別で使用する．

3.1 節，3.2 節で字句解析部の詳細について記述する．3.3 節では識別子判別と変換処理にあたる部分を記述する．3.4 節では Step3 で実際に用いられている手法の詳細を記述し，3.5 節で出力整形処理とクローンペア情報の解析について記述する．

3.1 コメントの除去

プログラミングにおけるコメントは，人間が読むことを目的にメモとしてソースコード中に挿入される注釈のことを指す．そのため，プログラムの機能には関係がない．タイプ 1 のコードクローンの定義より，コメントを除いたコードクローンを検出する．

本研究では，コメントのルールを 5 つに分類した．ルールは幾つでも追加できるが，先に追加したルールを優先的に処理するので，競合するルールが存在する可能性もある．

以下に、5 つのルールについての詳細をそれぞれ例を用いて記す．例の中の赤い文字が，字句解析において無視される部分である．

行コメント

ある記号から行末までをコメントとして扱うコメントを，行コメントと呼ぶ．C 言語や Java では，“//”以降が行コメントとみなされる．

行コメント
```
v=v+i; //comment
```

複数行コメント

ある開始記号から終了記号までに出現する文字をコメントとして扱うコメントを，複数行コメントまたはブロックコメントと呼ぶ．C 言語や Java では，“/*”と“*/”で囲われた文字が複数行コメントとして扱われる．

複数行コメント
```
v=v+i;    /*comment
printf(”Hello world”);    ← comment */
printf(”Hello world”);      /*comment*/
```

複数行コメントのネストを許す言語が存在する．コメントのネストとは，複数行コメントで囲われた中に，さらに複数行コメントの記述が存在する場合である．具体的な例を用いて，ネストを許す場合のコメントについて説明する．

ネストを許す場合
```
1  /*
2    /*
3      comment
4    */
5    comment
6  */
```

1 行目に出現する/*に対応するのは，ネストを許さない場合は 4 行目の*/であり，2 行目の/*は無視されている．それに対してネストを許す場合で，1 行目に出現する/*に対応するのは，6 行目の*/である．その違いは 2 行目の/*もコメントの開始記号とみなされているからである．4 行目の“*/”によって 2 行目から始まるコメントは終了されているが，5 行目は 1 行目から始まるコメントがまだ継続されているため，コメントとみなされる．

このようなネストの有無は言語のよって違いがあるので，複数行コメントにネストを許可するかどうかの

オプションも存在する．

行全体コメント

ある開始記号が先頭に存在する行全体をコメントとして扱うコメントを定義し，本研究ではこれを行全体コメントと呼ぶ．Fortran のコメントでは，行頭に開始記号“C”または“*”がある行全体をコメントと扱っているため，このようなコメントを無視するためにルールを設けた．行コメントとは違い，行の先頭ではない場所に記号が現れてもコメントの開始記号として認識しない．

行全体コメント

```
c       This is a comment
*       This is a comment
```

複数行全体コメント

ある開始記号が先頭に存在する行から終了記号が先頭に存在する行までをコメントして扱うコメントを定義し，本研究ではこれを複数行全体コメントと呼ぶ．Ruby のコメントでは，“=begin”で始まる行から，“=end”で始まる行までをコメントと扱っている．

複数行全体コメント

```
=begin comment
   puts ”Comment 1”    comment
   puts ”Comment 2”    comment
=end    comment
puts ”Hello world”
```

文字リテラル，文字列リテラルなど

このルールはコメントのルールでは無いが，コメントに優先されるルールとして設けているものである．Java では，シングルクォーテーションで囲まれた文字を文字リテラル，ダブルクォーテーションで囲まれた文字を文字列リテラルとされている．リテラルとはデータそのものを指し，この中に書かれた文字は値としてプログラムの中に存在している．よって，この中にコメントルールで使用される文字が出現しても，コメントの開始点と終了点に含まれない．以下は Java で使用される文字リテラル，文字列リテラルの例である．ダブルクォーテーションで囲まれた文字の中に，複数行コメントの開始記号が含まれているが，複数行コメントの開始とはみなされない．

文字リテラル，文字列リテラル

```
String x = ”    Line Comment start = /*   ”;
String y = ”    Line Comment end = */   ”;
```

表 1　各ルールに必要な要素
Table 1　Necessary elements for each rule

ルール	開始記号	終了記号	ネストの有無
行コメント	○	×	×
複数行コメント	○	○	○
行全体コメント	○	×	×
複数行全体コメント	○	○	×
文字，文字列リテラル	○	○	×

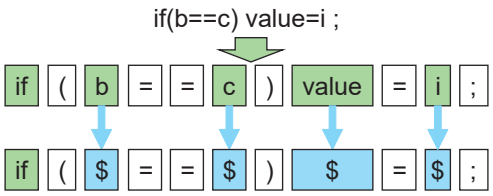

図 2　トークン分割と変換処理
Fig. 2　Tokenization and Transformation

以上のルールを定義したところで，各ルールを追加する際にどの情報が必要になるかを表 1 にまとめた．必要なら○，不必要なら×で表している．

3.2　トークン分割

本研究で提案する手法では，トークン分割はコメント除去の後に行われる．トークン分割で使われるルールは以下の通りである．番号が小さいルールほど優先される．

(1) 文字，文字列リテラルは 1 トークンとする．
(2) 空白と改行の前後でトークンを分割する．
(3) 記号は 1 文字ずつで分割する．記号が複数文字で 1 つの意味を表す場合でも，1 文字で 1 トークンとする．
(4) それ以外の連続したアルファベットまたは数字の列は 1 トークンとする．

3.3　変換処理

変換処理は，実用的に意味のあるコードクローンだけを検出するために行われる．これは字句解析で識別子と判別された変数名や関数名などを，全て 1 つのトークンに置き換える処理である．変数名や関数名に使用できない文字列はプログラミング言語によって定められていて，これは予約語と呼ばれる．if や while などプログラムの流れの制御に使用される単語は，指定されていることが多い．

図 2 はあるソースコード中の一文がトークン分割され，変数名を“$”という文字に変換する流れを示している．グリーンのボックスは英字列を表していて，ブルーのボックスは，変換された変数名を表している．

3.4　N-gram を用いたコードクローン検出アルゴリズム

CCFinderSW は，コードクローン検出を高速に検出するため，YOCCA と同様に N-gram を使用している [5]．De-

表 2　4-grams の例
Table 2　An Example of 4-grams

番号	4-gram
0	if($=
1	($==
2	$==$
3	==$)
4	=$)$
5	$)$+
6	)$=$
7	$=$;

bian GNU/Linux などの大規模な FOSS（Free and Open Source Software）からクローンを検出する場合，膨大な検出時間が必要か，またはリソース不足のためにクラッシュする可能性がある．Livieri らは，その問題の解決策として，既知のソースコードの集合とある 1 つのファイル間の，トークン単位のコードクローン検出を行うための時間的かつ空間的に効率的な手法を提案し，YOCCA を開発した．

N-gram とは，あるテキストの総体を前から順に任意の N 個の文字列または単語の組み合わせで分割したものである [10]．N の値に応じて，1-(uni)gram，2-(bi)gram，3-(tri)gram，と呼ばれる．N-gram を使用した言語モデルは，隣り合う文字列の出現頻度を調査するのに使用され，自然言語学の統計的な分野で利用されることが多く [9]，文字列の検索に用いられることもある．

CCFinderSW で，コードクローン検出に用いたアルゴリズムについて，説明する．字句解析して生まれたトークンから，N-gram を作成する．この時の N の数字はユーザによって設定されたものであり，検出されるコードクローンのトークン数のしきい値である．簡単のため，今回は最低トークン数を 4 とする．2 で変換処理が行われたトークン列の，隣合う 4 トークンを結合して 4-gram を作成する．その結果，表 2 のようになる．

ここからは簡単のため，実際のアルゴリズムではトークン単位で N-gram を作成しているところを，文字単位で N-gram を作成する．ソースコード全体を *sethesetheses* とする．検出するコードクローンのトークン数のしきい値を 4 とし，4 文字以上の一致箇所を見つける．

しきい値が 4 のため，4-gram を使用する．まず，ソースコード全体から 4-gram を作成する．その 4-gram をリスト化する際には，出現箇所，文字列，ハッシュ値をセットでリスト化を行う．出現箇所は，初めに現れる 4-gram を 0 番目として，次に現れる 4-gram を 1 番目という番で設定している．ハッシュ値は，文字列をハッシュ化したものである．このハッシュ化は，文字列の比較より数値の比較のほうが速いことを利用して，高速化を図ったものである．このように出現箇所，文字列，ハッシュ値をセットでリスト化したものを出現箇所順リストと呼ぶ．表 3 に，出現箇所順リストの例を示した．

今回，説明のために図示しているハッシュ値は，ツールに実装されているものとは違っている．実際のハッシュ化は，Java の String クラスに用意された hashcode というメソッドを用いて，String 型の文字列のハッシュ値を生成している．また，この説明ではハッシュ値の衝突は考えないとする．つまりハッシュ値が同値ならば，もとの文字列も同値であると考える．

この出現箇所順リストのハッシュ値の集合から，重複のない全てのハッシュ値を選び出す．その選び出されたハッシュ値の各々に対応する，出現箇所を全て選び出す．ハッシュ値から出現箇所の対応は，1 対 1 ではないため，多数存在する場合もある．ハッシュ値と出現箇所の集合をセットで記録したものを，ユニークリストと呼ぶ．表 4 に，ユニークリストの例を示した．

次にコードクローンの探索を行う．作成したユニークリストの一行目の，ハッシュ値 (1234) に対応した出現箇所の集合である，$\{0,5\}$ に注目する。この集合の各要素に対し，出現箇所順リストで対応する行をみると同じハッシュ値が出現しているということがわかる．つまり，出現箇所 0 と 5 に登場する 4-gram から生成されたハッシュ値は同値であり，よって 4-gram が同値であると考えられる．言い換えれば，元の文字列で 0 文字目から 3 文字目までの 4 文字と，5 文字目から 8 文字目までの 4 文字は同値であると考えられ，これをコードクローンとする．また，ハッシュ値 (2342) に対応した出現箇所の集合 $\{1,6\}$ と，ハッシュ値 (3421) に対応した出現箇所の集合 $\{2,7\}$ でも同じことがいえるため，この文字列からはクローンセットが 3 組出現しているといえる．トークン数がしきい値のコードクローンはこのようにして検出される．

しきい値よりも大きいコードクローンも存在するため，その探索を行うアルゴリズムを説明する．あるクローンセットの要素のそれぞれの，次の出現箇所のハッシュ値を見る．例えば，4 文字のクローンセット $\{0,5\}$ の要素それぞれに 1 を足し $1,6$ として，出現箇所順リストで出現箇所 1 と出現箇所 6 に対応するハッシュ値を見ると，どちらも (2342) で一致している．つまり出現箇所が 0 と 5 の 4-gram が一致し，1 と 6 の 4-gram が一致しているので，出現箇所 0 と 5 から始まる 5 文字がコードクローンになっていると言える．こうして 4 文字のクローンセット $\{0,5\}$ は，5 文字のクローンセット $\{0,5\}$ に書き換えられる．このようにクローンセットの出現箇所要素の，次の出現箇所のハッシュ値を見ることで新たなクローンセットを探索する．この作業を，ユニークリストの全てのハッシュ値に対し実行することでコードクローン探索を終える．

3.5　CCFinderSW の出力形式

CCFinderSW は Step4 として出力整形処理を行い，ファ

表 3　出現箇所順リストの例
Table 3　An Example of a List in Order of Appearance

{ 出現箇所 }	[文字列]	(ハッシュ値)
{0}	[seth]	(1234)
{1}	[ethe]	(2342)
{2}	[thes]	(3421)
{3}	[hese]	(4212)
{4}	[eset]	(2123)
{5}	[seth]	(1234)
{6}	[ethe]	(2342)
{7}	[thes]	(3421)
{8}	[hese]	(4212)
{9}	[eses]	(2121)

表 4　ユニークリストの例
Table 4　An Example of a Unique List

(ハッシュ値)	{ 出現箇所 }
(1234)	{0,5}
(2342)	{1,6}
(3421)	{2,7}
(4212)	{3,8}
(2123)	{4}
(2121)	{9}

イルに出力する際に CCFinder，または CCFinderX と同じ形式で出力することができる．CCFinder はコードクローン分析ツールである Gemini[13] で使用できる形式で出力され，CCFinderX は同じくコードクローン分析ツールである GemX に対応した形式で出力されている [11]．つまり，CCFinderSW で出力されたクローンペア情報は Gemini と GemX で分析できる．

図 3 は Python のソースコードに対して CCFinderSW を実行した結果を，Gemini で表示したものである．図の左側に表示されたファイルの 62 行目から 73 行目と，右側に表示されたファイルの 52 行目から 63 行目がクローンペアになっていて，それぞれ赤と青でハイライトされている．2 つのクローンペアは関数名と変数名が違っており，また左側のファイルの 66 行目にはコメントが含まれているため，タイプ 2 のコードクローンが検出されていることがわかる．CCFinder と CCFinderX は Python のコードクローン検出に対応していないため，本研究で開発した CCFinderSW で新たに対応できたことが示されている．

4.3 節の適用実験では，結果の確認を Gemini で行った．

4. 適用実験

本章では，開発したツールの有用性を示すために行った適用実験について説明する．

4.1 サンプルコード集:RosettaCode

RosettaCode は，同一のタスク，または例題を様々なプログラミング言語で実装し，そのソースコードをウェブ上に公開しているウィキ形式のウェブページである[*2]．同じタスクへの実装を様々な言語で行うことで，それぞれのプログラミング言語がどのように類似し，どのように異なっているかを提示することを目的としている [6]．

また本研究で使用する RosettaCode のソースコードは，RosettaCode のウェブサイトに掲載されているものが Github 上のリポジトリにミラーリングされているものを使用している[*3]．また，そのリポジトリの 2015 年 11 月 18 日のコミットによるものである．

4.2 コメント除去の有用性の評価

本研究で提案するコメントルールとその除去手法についての，有用性を示すための適用実験を行った．適用対象となるのは，RosettaCode の“Comments”という名前のタスクである．このタスクはソースコードに必ず各言語のコメントが含まれているため，今回の実験に適している．

実験の手順は以下の通りである．

(1) Comments タスクで使用されている，提案手法で削除可能な各言語のコメントルールを列挙する．

この中から，字句解析の範囲で可能であったが，今回の提案手法では除去できないコメントを除去不可能とした．

(2) オプションを 26 種類作成する．このオプションによって多くの言語のコメントルールを網羅出来るようにする．

ユーザによるコメントルールの設定を更に容易にするための仕組みとして，アルファベット列を与えるだけでコメントルールの設定を行えるという仕組みを作成した．26 種類という限定には，ツールを実行する際に，引数としてコメントルールを記述しやすいということを見込んでいる．コメントのルールは加算方式であり，例えば “adf” という引数を与えてツールを実行すると，a と d と f に設定されたコメントルールでコメント除去を行う．

表 5 に，各オプションを示した．z に対応するルールは複数行コメントのネストの許可のオプションを割り当てるため，表には載せていない．左から二列目の数字は使用したコメントの分類を表しており，1 は行コメント，2 は複数行コメント，3 は行全体コメント，4 は複数行全体コメント，5 は文字・文字列リテラルのルールである．なお，各言語に対するオプションの適用結果の詳細は，文献 [12] に掲載している．

このオプションを使用して，RosettaCode の“Comments”が実装された 175 言語に対してコメント除去を行った．適用出来なかった言語には，オプションの列に不可と記述

[*2] http://rosettacode.org/wiki/Rosetta_Code
[*3] https://github.com/acmeism/RosettaCodeData

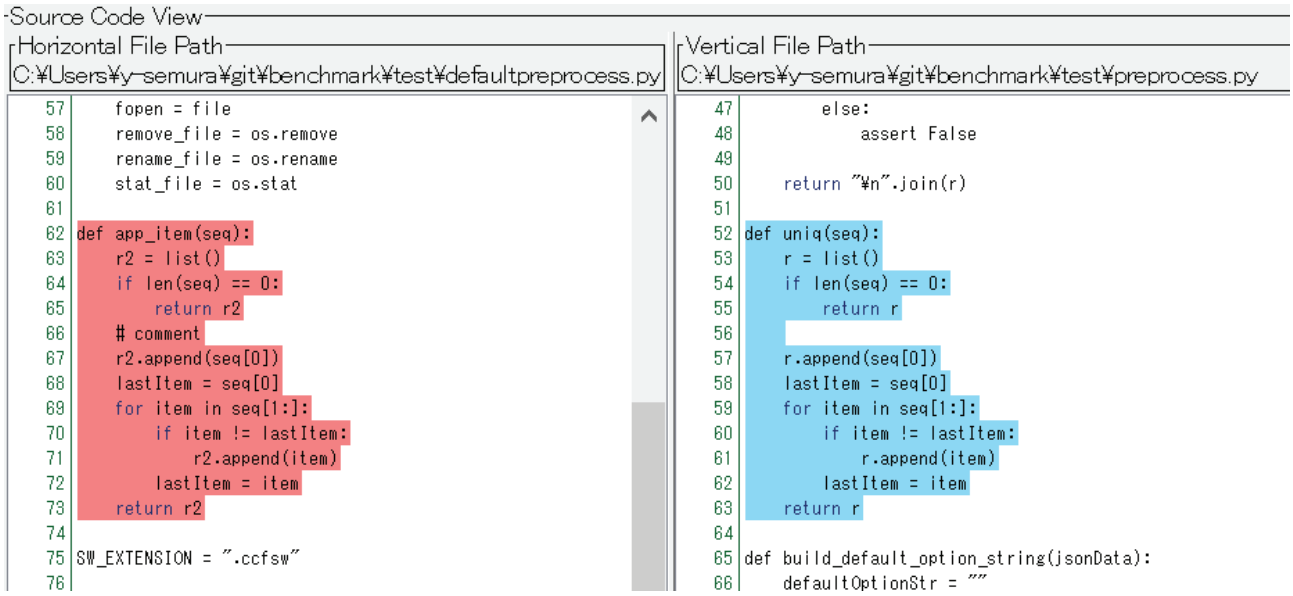

図 3　Gemini を用いたコードクローンの表示
Fig. 3　Visualization of Code Clone Using Gemini

表 5　26 種類のオプション (z を除く)
Table 5　26 Options (exclude z)

	分類	開始	終了
a	5	’	’
b	5	”	”
c	1	//	なし
	2	/*	*/
	2	/+	+/
	2	<!–	– >
d	1	;	なし
e	1	#	なし
f	1	–	なし
g	1	%	なし
h	1	!	なし
	1	# !	なし
i	1	’	なし
	1	‘	なし
j	1	NB.	なし
k	2	{	}
l	2	[	]
m	2	(	)
n	2	”	”

	分類	開始	終了
o	2	(*	*)
	2	(:	:)
	2	(#	#)
p	2	#|	|#
	2	#=	=#
	2	#{	}#
	2	###	###
	2	#cs	#ce
	2	# ˜	˜#
q	2	{|	|}
	2	{-	-}
r	2	%{	%}
s	1	->	なし
	1	–##	なし
	2	–[[	]]
t	1	COMMENT	なし
	1	IGNORELINE	なし
	2	’COMMENT’	;

	分類	開始	終了
u	1	©	なし
	1	*>	なし
	1	*|	なし
	3	NOTE	なし
	3	*	なし
v	3	*	なし
	3	C	なし
	3	c	なし
	3	:	なし
w	2	%REM	%END REM
	1	REM	なし
	1	rem	なし
	1	Rem	なし
x	4	==comment	=cut
	4	=begin	=end
	4	=pod	=cut
y	3	#;	なし
	3	NIL	なし
	3	###	なし
	3	end.	なし

した．

この実験結果より，175 言語のうち 166 言語が提案手法で可能であり，151 言語が 26 種類のオプションによって対応が可能であることが示された．提案手法で対応不可能な言語が生まれる理由としては，コメントが構文解析をしなければ全く除去出来ない場合や，言語が難解な文法を持っている場合などがあった．また，オプションで対応出来ない言語に関しては，コメントルールが独特で他の言語と共通することがない場合や，コメントルールが多すぎる場合などがあった．

表 6 は，166 言語中の各オプションの使用言語数である．

4.3　コードクローン検出の精度の評価

開発したツールのコードクローン検出の精度について評

表 6　各オプションの使用言語数
Table 6　The Numbers of Languages for Each Options

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	3	55	25	40	14	10	7	11	1	3	2	1	2	14	8	1	1	2	2	2	4	9	2	1	17

表 7　コードクローン検出の再現率
Table 7　Recall of the Code Clone Detection

言語名	検出数	母数	再現率
C	9	9	1
C++	7	7	1
Go	16	16	1
Java	8	8	1
Python	3	3	1
Ruby	3	3	1

価する実験を行った．RosettaCode のあるタスクのソースコードのうち数言語を選んだ．その言語のソースコードの中を読み，タイプ 2 までのコードクローンであるものをリストアップする．その後コメントルールと予約語の入力をして開発したツールを実行し．検出されたコードクローンの中にあらかじめリストアップしたコードクローンが含まれていることを確認した．リストアップしたコードクローンを正解集合として， 検出したコードクローンとの再現率を算出した．

今回は，適用対象として RosettaCode 内の “Sudoku” という名前のタスクを選択した．このタスクを選んだ理由は，RosettaCode の他のタスクと比べて一つ一つのソースコードが長く (それぞれ 60 から 200 行程度)，コードクローンが含まれている可能性が高いことが挙げられる．また，パズルを解くという内容のタスクであり，パズルの構造上，for 文などの制御文が多用されている可能性が高く，コードクローンも存在しやすいと考えられることも挙げられる．

今回は対象言語を 6 言語として，最小トークン数を 15 としてツールを実行した．“Sudoku” が実装された 40 言語のうち，予約語を用意できた言語を選び，さらに実際にトークン単位のコードクローンが存在したものを選んだ．

実験の結果を，表 7 に示す．結果，リストアップしたコードクローンを全て検出することが出来た．リストアップしたコードクローンを母数，そのうちでツールが検出したコードクローン数を検出数として，再現率を算出している．

5. まとめと今後の課題

本研究で開発したコードクローン検出ツールでは，多くの言語での字句単位でのコードクローンを検出できることを示した．しかし，字句解析が出来なかった言語に対して対応するオプションを作成することで，ユーザによる設定がさらに複雑になってしまう可能性がある．

トークン分割に対しては文字リテラルと文字列リテラル以外のオプションの設定を行わなかったが，トークン分割に関するオプションを用意することが今後の課題といえる．

謝辞　本研究は JSPS 科研費 25220003，16K16034，15H06344 の助成を受けた．

参考文献

[1] 肥後芳樹，楠本真二，井上克郎：コードクローン検出とその関連技術，電子情報通信学会論文誌，Vol. J91-D, No. 6, pp. 1465–1481 (2008).
[2] 井上克郎，神谷年洋，楠本真二：コードクローン検出法，コンピュータソフトウェア，Vol. 18, No. 5, pp. 47–54 (2001).
[3] 肥後芳樹，神谷年洋，楠本真二，井上克郎：コードクローンを対象としたリファクタリング支援環境，電子情報通信学会論文誌，Vol. Vol.J88-D-I, No. 2, pp. 186–195 (2005).
[4] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670 (2002).
[5] Livieri, S., German, D. M., Inoue, K. : A needle in the stack: efficient clone detection for huge collections of source code, Technical report (2010).
[6] Nanz, S. and Furia, C. A.: A Comparative Study of Programming Languages in Rosetta Code, *Proceedings of the 37th International Conference on Software Engineering*, pp. 778–788 (2015).
[7] Roy, C. K., Cordy, J. R. and Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495 (2009).
[8] 植田泰士，神谷年洋，楠本真二，井上克郎：クローン検出ツールを用いたソースコード分析ツールの試作，電子情報通信学会技術研究報告，Vol. 101, No. 240, pp. 17–24 (2001).
[9] 山田崇仁：N-gram モデルを利用したテキスト分析，http://www.shuiren.org/chuden/teach/n-gram/index-j.html.
[10] 山田崇仁：N-gram 方式を利用した漢字文献の分析，立命館白川靜記念東洋文字文化研究所紀要，No. 1, pp. 1–23〈オンライン〉，入手先 〈http://ci.nii.ac.jp/naid/40016415257/〉 (2007).
[11] 神谷年洋：the archive of CCFinder Official Site, http://www.ccfinder.net/.
[12] 瀬村雄一：柔軟に変更可能な字句解析機構を持つコードクローン検出ツールの開発，大阪大学基礎工学部情報科学科 卒業論文，(オンライン)，入手先 (http://sel.ist.osaka-u.ac.jp/lab-db/Bthesis/contents.ja/157.html) (2017).
[13] 肥後芳樹，吉田則裕，楠本真二，井上克郎：産学連携に基づいたコードクローン可視化手法の改良と実装，情報処理学会論文誌，Vol. 48, No. 2, pp. 811–822 (2007).