特別研究報告

題目

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

指導教員

井上 克郎 教授

報告者

瀬村 雄一

平成 29 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 28 年度 特別研究報告

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

瀬村 雄一

内容梗概

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．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．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．

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．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字句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．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．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．

本研究では，多言語の文法に対応可能なコメント除去の手法を提案し，その手法を用いて多言語のコードクローン検出が可能なツールを開発した．提案手法の適用実験として，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コメント除去が可能だった 50 言語のソースコードに対して実行し，全ての言語でトークン単位のコードクローンが検出されていることを示した．

主な用語

コードクローン

字句解析

N-gram

目 次

1　まえがき

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 [1]．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 [2]．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 [3]．

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 [5]．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 [6]．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．1 つずつ字句解析を実装して対応言語を増やすよりも，各言語に容易かつ柔軟に変更可能な，字句解析と同等の機能を補える仕組みを使って，対応言語を増やしたほうが良い [7]．

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，ユーザによる柔軟かつ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 [8]．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，提案した柔軟な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コメント除去が可能だった 50 言語のソースコードに対して実行し，全ての言語でトークン単位の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，コードクローンと，コードクローン検出ツールである CCFinder と YOCCA の説明する．3 章では本研究で提案する柔軟に変更可能な字句解析の手法と，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　背景

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

2.1　コードクローン

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 [1]．一般的に，互いにコードクローンとなるコード片はクローンペアと呼ばれ，クローンペアにおいて推移関係が成り立つコードクローンの集合はクローンセットと呼ばれる．

コードクローンの存在は，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 [2]．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 [3]．

2.1.1　コードクローンの分類

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 4 つのタイプの分類を用いる [9]．

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．

タイプ 3

タイプ 2 の条件に加えて，文の変更，追加または削除など違いを含むコードクローン．

タイプ 4

同様の処理を行うが，構文的な違いを含むコードクローン．

2.1.2 コードクローンの検出技術

現在までに様々なコードクローン検出技術が提案されている．以下に各技術の特徴と，その技術に関連する研究を示す．

トークン単位の検出

トークン単位の検出手法では，検出の前処理として，ソースコードはトークンの列に変換される．しきい値以上連続して一致しているトークンの列がコードクローンとして検出される．行単位と違い，コーディングスタイルに依存することはないが，プログラムの構造を無視した類似部分を検出することも多い [10]．

トークン単位の検出ツールとして，CCFinder がある．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．CCFinder はタイプ 2 までのコードクローンが検出可能である．

関数単位の検出

関数単位の検出手法では，プログラムの構造的な類似性に着目して，プログラムの意味的な類似度を計算することでコードクローンを検出している．他の手法では検出できないような，for 文と while 文などで構文的な違いはあるが同一処理を行っているコードクローンも検出することができる．

関数単位の検出ツールとして，山中らが開発した関数クローン検出ツールがある [11]．関数クローン検出ツールでは意味的に処理が類似した関数単位のコードクローンを検出する．つまり，タイプ 4 までのコードクローンが検出可能である．情報検索技術を用いることで，ソースコード中の識別子や予約語に用いられる単語に対して重み付けを行うことによって各関数を特徴ベクトルに変換しその類似度を検索することによって，関数クローンの検出を行っている．

抽象構文木を用いた検出

抽象構文木を用いた検出では，検出の前処理としてソースコードに対して構文解析を行い，抽象構文木が構築される．検出されるコードクローンはコーディングスタイルに依存せず，抽象構文木の部分木が検出されるためプログラムの構造を無視した類似部分は検出されない．行単位と字句単位の検出に比べ，時間的及び空間的なコストは高くなるが，実用的な検出法として知られている．

抽象構文木単位の検出ツールとして，Jiang らが開発した Deckard がある [12]．Jiang らは，抽象構文機の各部分木を配列表現に変換し，局所性鋭敏型ハッシュアルゴリズ

ム [13] を用いて，類似配列を求めることにより，コードクローンを検出する手法を提案している．この局所性鋭敏型ハッシュアルゴリズムでは，ある程度配列に違いがあっても同じハッシュ値を割り当てることができる．そのため，タイプ 3 までのコードクローンを検出することができる．

他，プログラム依存グラフ，メトリクスなどの技術を用いた検出の研究もなされている．

2.2 トークンに基づくコードクローン検出ツール:CCFinder/CCFinderX

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．

図 1 は，CCFinder の処理概要である．以下，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．

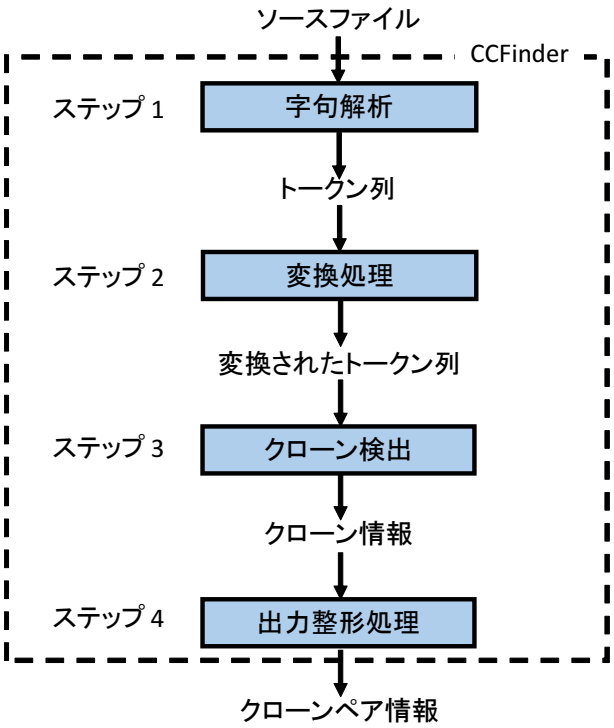


図 1: CCFinder の処理概要

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 [6]．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，字句解析部を柔軟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．

2.3 ハッシュ関数を用いたコードクローン検出ツール:YOCCA

コードクローン検出部の実装にあたり，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 [8]．YOCCA は，ある 1 つのファイルと 100 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．Livieri らは，Debian GNU/Linux などの FOSS（Free and Open Source Software）に対し，クローン検出を行うのは数ヶ月の時間がかかるか，またはリソース不足のためにクラッシュする可能があるという問題を指摘している．その解決策として，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たと記述している．YOCCA では N-gram という手法を使用して，コードクローン検出部の高速化を図っている．

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 [14]．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 [15]，文字列の検索に用いられることもある．

図 2 は，日本語の文章を隣り合う数文字単位で分割する例として，「今日は良い天気ですね」という文に 3-gram を使用する場合を示す．文章の始めの隣り合う 3 文字である「今日は」から文章の終わりの「ですね」まで，8 つの 3-gram が現れている．

今	日	は	良	い	天	気	で	す	ね
今	日	は							
	日	は	良						
		は	良	い					
			良	い	天				
				い	天	気			
					天	気	で		
						気	で	す	
							で	す	ね

図 2: 3-gram の例

YOCCA は，巨大なソースコード群に対して N-gram をあらかじめ計算して，データベースに保存している．そのデータベースを使用して，ある 1 つのファイルとのコードクローンを検出することから，クエリとなるファイルの一部を巨大なソースコード群から検索するという側面の強いツールといえる．Livieri らは論文 [8] 内で，クエリとなるファイルと巨大なソースコード群の間に存在するコードクローンを検出するアルゴリズムを例を用いて説明している．この例では 4-gram を使用している．

被検索文字列を $\sigma = \{sethesetheses\}$ とする．検索の準備として，この文字列から 4-gram を作成し，あらかじめ全てデータベースに保存しておく．これを出現順に並べた集合を $\omega(\sigma)$ とすると，(1) のように 4-gram が 10 個出現することになる．

$$\omega(\sigma) = \{seth, ethe, thes, hese, eset, seth, ethe, thes, eses\} \tag{1}$$

この 10 個の 4-gram から重複のない集合を作成し，これを $\omega^U(\sigma)$ とする．

$$\omega^U(\sigma) = \{seth, ethe, thes, hese, eset, eses\} \tag{2}$$

またこの重複のない集合の要素に対し，その要素が被検索文字列で何番目に出現するかを記録しておく．全体で 1 つ目にあたる $seth$ の出現箇所を 0 番目とすると，$p^\sigma(seth) = \{0, 5\}$

のようにおける．これを他の要素に対しても記録すると，

$$\begin{aligned} p^{\sigma}(seth) &= \{0,5\} \\ p^{\sigma}(ethe) &= \{1,6\} \\ p^{\sigma}(thes) &= \{2,7\} \\ p^{\sigma}(hese) &= \{3,8\} \\ p^{\sigma}(eset) &= \{4\} \\ p^{\sigma}(eses) &= \{9\} \end{aligned} \tag{3}$$

となる．これで検索の準備は終了である．

次に検索のアルゴリズムの説明をする．検索対象となる文字列を $s = heset$ とする．この s に対しても被検索文字列と同様に，出現する 4-gram の集合を用意する．この集合 c を，

$$c = \{hese, eset\}, \quad c_0 = hese, \quad c_1 = eset \tag{4}$$

とする．ここで c の 0 番目の要素である $c_0 = hese$ に対する $p^{\sigma}(c_0)$ は，

$$p^{\sigma}(c_0) = p^{\sigma}(hese) = \{3,8\} \tag{5}$$

であるから，検索対象の s の最初の 4 文字は被検索文字列の 3 番目と 8 番目に出現することがわかる．

次に c の 1 番目の要素である $c_1 = eset$ に対する $p^{\sigma}(c_1)$ は，

$$p^{\sigma}(c_1) = p^{\sigma}(eset) = \{4\} \tag{6}$$

であるから，ここで $c_1 = eset$ の出現箇所は 4 とわかり，

$$p^{\sigma}(c_0) = \{3,8\}, \quad p^{\sigma}(c_1) = \{4\} \tag{7}$$

に着目すると，c_0 と c_1 の出現箇所が 3 と 4 で連続していることがわかる．

つまり，被検索文字列で 3 番目と 4 番目にあたる 4-gram が，検索対象となる文字列の 0 番目と 1 番目の 4-gram と一致したことになる．そして，これは検索対象となる文字列の全てであるから，被検索文字列で 3 番目から始まる文字列が検索対象と一致するとわかる．以上で N-gram を用いた検索アルゴリズムの説明を終える．

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対応していない．本研究では，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．

コードクローン探索を行う具体的なアルゴリズムについては，3 章で述べる．

3　提案ツール

本章では，本研究で開発を行った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処理概要と，その各ステップの詳細を述べる．ツールに関する，全ての実装は Java で行った．

図 3 は開発したツールの処理概要である．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　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; //ここはコメント
```

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．

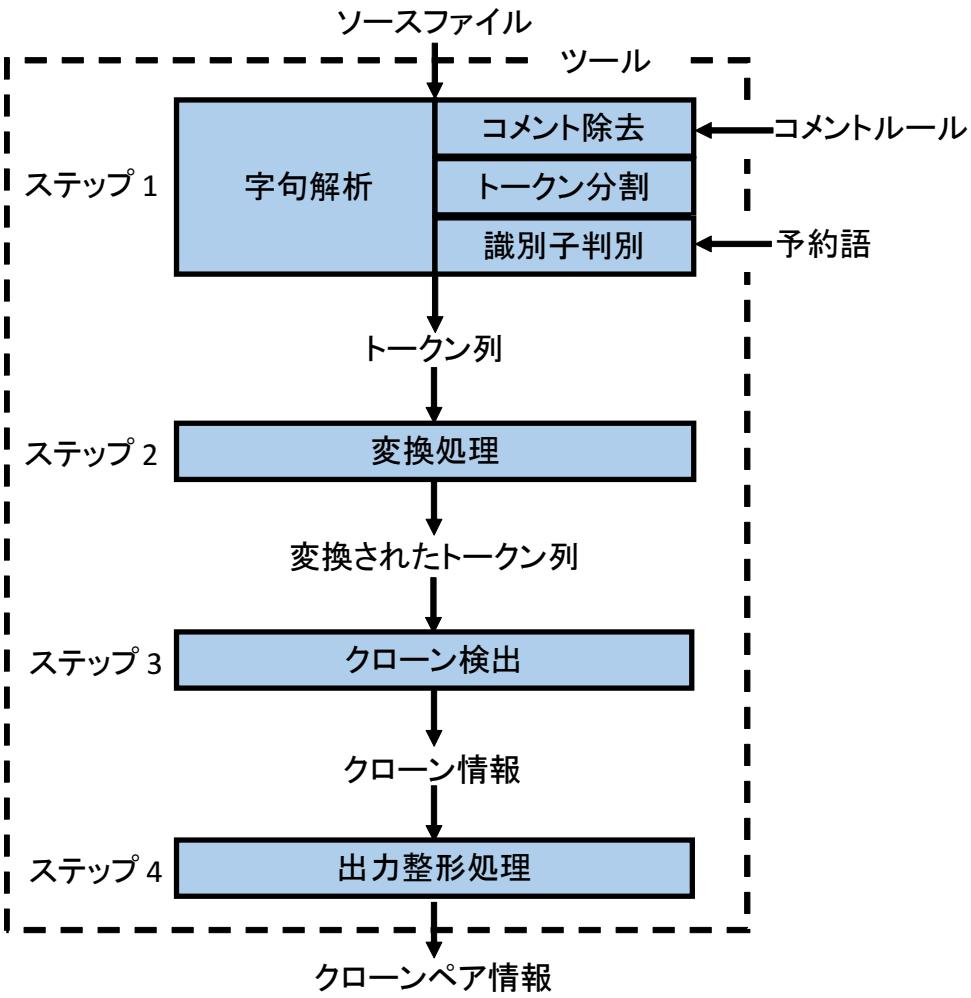


図 3: 開発したツールの処理概要

複数行コメント

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

複数行コメントにおいては，ネストを許す場合がある．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，ネストを許す場合と，許さない場合について説明する．

ネストを許さない場合

```
1   /*
2     /*
3       ここはコメント
4     */
5     ここはコメントではない
6   */
```

ネストを許す場合

```
1   /*
2     /*
3       ここはコメント
4     */
5     ここはコメントである
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       ここはコメント
*       ここもコメント
```

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．以下はその例である．

複数行全体コメント

```
=begin ここから
   puts ”Comment 1”   コメント継続中 1
   puts ”Comment 2”   コメント継続中 2
=end   ここまでコメント
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 = */";
```

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2にまとめた．必要なら○，不必要なら×で表している．

表 1: 各ルールに必要な情報

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	○	○	×

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．

Java で使用される文法に対し，このルールを使用した例を図4と図5に示す．

図4では例文が7トークンに分割されている．今回はソースコード上で「"」から始まって「"」で終わる文字列を文字列リテラルとみなし，ルールの1つ目によって1トークンとしている．

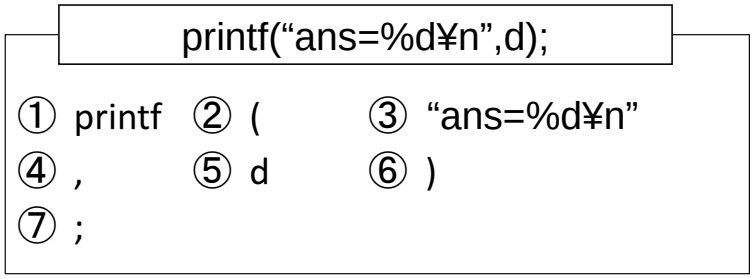


図 4: トークン分割の例 1

図 5 では例文が 11 トークンに分割されている．例文内の“ == ”は，二文字で一つの意味を表すが，3 番目のルールによって，1 文字でそれぞれ 1 トークンとしている．

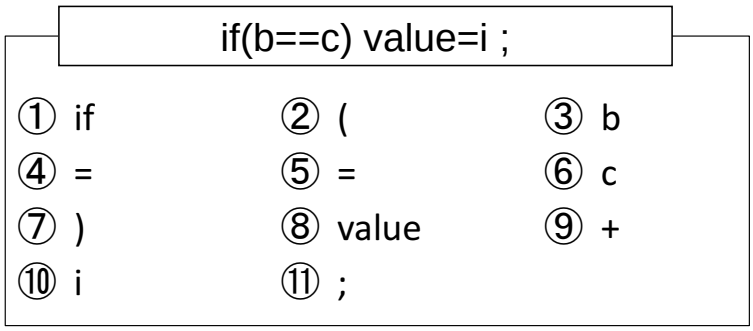


図 5: トークン分割の例 2

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，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 6 は C++と Java の予約語の一部であり，言語間の違いを示している．

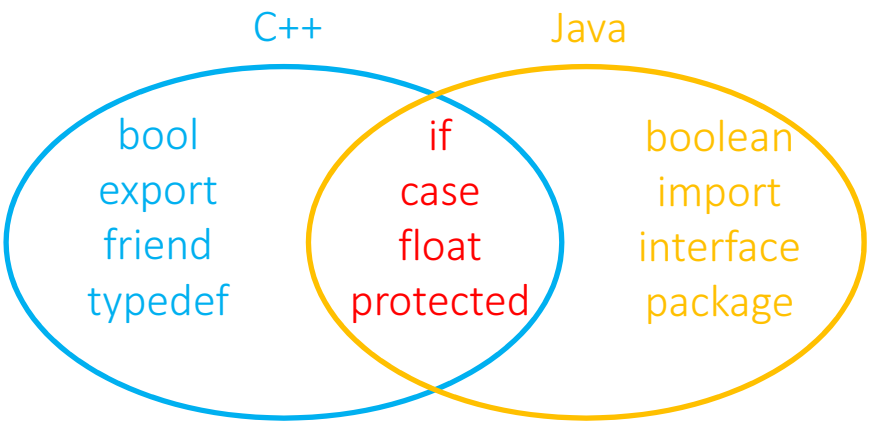


図 6: 予約語

例として，図 5 の例文と分割されたトークンに対して，変換処理を行ったものを図 7 に示す．今回は識別子の変換先を“a”に設定している．アルファベットで構成されたトークンの中で, 1 トークン目の“if”のみ予約語であるため，他は全て“a ”に変換される．

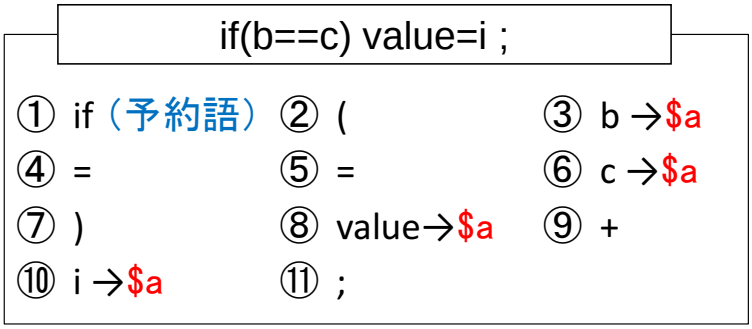


図 7: トークンの変換処理

3.4 N-gram を用いたコードクローン検出アルゴリズム

コードクローン検出部の実装にあたり，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 N-gram を使用したコードクローン検出アルゴリズムから発想を得て，実装を行った．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．

実際に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最低しきい値である．

簡単のため，今回は最低トークン数を 4 とする．図 7 の変換処理を行ったトークンの，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．

表 2: 作成された 4-gram

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

次に，N-gram のリストを作成し，N-gram からコードクローンを探索するアルゴリズムを説明する．ソースコード全体を *sethesetheses* とする．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，今回は文字単位で N-gram を作成する．検出するコードクローンの最低しきい値を 4 とし，4 文字以上の一致箇所を見つける．

最低しきい値が 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 番目という順番で設定している．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図 8 の左側の表に示した．このリストを出現箇所順リストと呼ぶ．

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 という関数を用いて，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図 8 の右側の表に示した．このリストを，ユニーク・リストと呼ぶ．

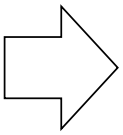
次にコードクローンの探索を行う．作成したユニーク・リストの一行目の，ハッシュ値 (1234) に対応した出現箇所の集合である，$\{0, 5\}$ に注目する。この集合の各要素に対し，出現箇所順リストで対応する行をみると同じハッシュ値が出現しているということがわかる．つまり，出現箇所 0 と 5 に登場する 4-gram から生成されたハッシュ値は同値であり，よって 4-gram が同値であると考えられる．言い換えれば，元の文字列で 0 文字目から 3 文字目までの 4 文字と，5 文字目から 8 文字目までの 4 文字は同値であると考えられ，これをコードクローンとする．また，ハッシュ値 (2342) 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．最低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．

出現箇所順リスト

{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)



ユニーク･リスト

(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}

図 8: N-gram

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．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 [16] [17][18].

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[1]．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．

4.2.1　適用対象と手法

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．26 種類のうち，1 種類は複数行コメントのネストの許可に関する設定を行うものとし，他の 25 種類はコメントのルールを加算方式とす

[1]GitHub - RosettaCode Data Project:https://github.com/acmeism/RosettaCodeData

る．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．

4.2.2　26 種類のオプション

オプション名にはそれぞれアルファベットが与えられている．ルールの列には，本研究で提案した 5 つのコメントルールを記述し，それぞれに対し開始記号と終了記号を設定する．行コメントの場合は「行」，複数行コメントの場合は「複数行」，行全体コメントの倍は「行全」，複数行コメントの場合は「複数行全」，文字・文字列リテラルの場合は「リテラル」と記述した．

表 3 と表 4 は作成したオプションである．z のルールを追加した場合は，複数行コメントのネストを許す．

表 3: a から p までのオプション

オプション名	ルール	開始記号	終了記号
a	リテラル	’	’
b	リテラル	”	”
c	行	//	なし
	複数行	/*	*/
	複数行	/+	+/
	複数行	<!–	– >
d	行	;	なし
e	行	#	なし
f	行	–	なし
g	行	%	なし
h	行	!	なし
	行	#!	なし
i	行	’	なし
	行	‘	なし
j	行	NB.	なし
k	複数行	{	}
l	複数行	[	]
m	複数行	(	)
n	複数行	”	”
o	複数行	(*	*)
	複数行	(:	:)
	複数行	(#	#)
p	複数行	#|	|#
	複数行	#=	=#
	複数行	#{	}#
	複数行	###	###
	複数行	#cs	#ce
	複数行	# ~	~#

表 4: q から y までのオプション

オプション名	ルール	開始記号	終了記号
q	複数行	{|	|}
	複数行	{-	-}
r	複数行	%{	%}
s	行	->	なし
	行	–##	なし
	複数行	–[[	]]
t	行	COMMENT	なし
	行	IGNORELINE	なし
	複数行	'COMMENT'	;
u	行	*>	なし
	行	*|	なし
v	行	ⓒ	なし
	行全	*	なし
	行全	NOTE	なし
	行全	C	なし
	行全	c	なし
	行全	:	なし
w	複数行	%REM	%END REM
	行	REM	なし
	行	rem	なし
	行	Rem	なし
x	複数行全	==comment	=cut
	複数行全	=begin	=end
	複数行全	=pod	=cut
y	複数行	#;	なし
	複数行	NIL	なし
	複数行	###	なし
	複数行	end.	なし

オプション y の複数行コメントの終了文字がなしと書いている理由は，除去対象のコメントが，ある開始文字が現れてからはファイルの終わりまでコメントとみなされるというルー

ルにもとづいているからである．終了記号をソースコードでは現れることのない文字列にすれば，複数行コメントのルールで除去が可能である．

4.2.3 実験結果

4.2.2 節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い，その結果を表 5 から表 10 に示す．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，150 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 5: 適用実験の結果 1

言語	オプション
360-Assembly	可能
4D	ci
6502-Assembly	d
8086-Assembly	d
ACL2	dp
Ada	f
ALGOL-60	t
ALGOL-W	可能
AmigaE	cs
Asymptote	c
AutoHotkey	cd
AutoIt	dp
AWK	e
Babel	可能
BASIC	iw
Batch-File	vw
BBC-BASIC	uw
Befunge	不可能
Blast	e
Brainf—	不可能
Brat	可能
Brlcad	e
Burlesque	不可能
C	ce
C++	ce
Chapel	c
Chef	不可能
Clean	cz
Clojure	不可能
COBOL	u

表 6: 適用実験の結果 2

言語	オプション
CoffeeScript	ep
ColdFusion	c
Common-Lisp	d
Component-Pascal	cz
D	cz
Deja-Vu	beh
Delphi	c
Deluge	c
DWScript	clo
Dylan	c
E	可能
Eiffel	f
Ela	c
Elixir	e
Emacs-Lisp	d
Erlang	g
Euphoria	cf
Factor	h
Falcon	c
FALSE	k
Fancy	be
Fish	不可能
Forth	可能
Fortran	v
Frink	c
Gambas	i
GAP	e
Gema	h
GML	c
Go	c

表 7: 適用実験の結果 3

言語	オプション
Golfscript	e
GW-BASIC	w
Haskell	fq
Haxe	c
HicEst	h
Icon	e
IDL	d
Inform-7	lz
Io	ce
J	j
Java	c
JavaScript	c
JCL	可能
Joy	eo
Julia	pe
K	可能
KonsolScript	c
Lang5	e
LaTeX	g
Liberty-BASIC	iw
Lilypond	g
Logo	d
Logtalk	cg
LotusScript	iw
LSE64	e
Lua	sf
M4	不可能
Maple	eo
Mathematica	oz
MATLAB	g

表 8: 適用実験の結果 4

言語	オプション
Maxima	cz
MAXScript	cf
MBS	ch
Metafont	g
Mirah	ce
Modula-2	oz
Modula-3	oz
MOO	c
Neko	c
Nemerle	c
NetRexx	cf
NewLISP	d
Nimrod	e
NSIS	cde
Oberon-2	oz
Objeck	pe
OCaml	oz
Octave	egr
Openscad	c
OxygenBasic	cid
Oz	cg
Pascal	ko
PASM	e
Perl	ex
Perl-6	可能
PHP	ce
PicoLisp	epy
Pike	c
PL-I	c
PL-SQL	cf

表 9: 適用実験の結果 5

言語	オプション
Pop11	cdz
PostScript	g
PowerShell	可能
ProDOS	t
Prolog	cg
Protium	可能
PureBasic	d
Python	e
R	e
Racket	pez
Raven	e
REALbasic	ci
REBOL	ek
Retro	m
REXX	abcfz
RLaB	ce
Ruby	ex
Run-BASIC	iw
Rust	可能
SAS	可能
Sather	f
Scala	c
Scheme	dpz
Scilab	c
Seed7	eoz
SETL	f
Simula	h
Slate	n
Smalltalk	n
SNOBOL4	可能

表 10: 適用実験の結果 6

言語	オプション
SQL	f
Squirrel	ce
Standard-ML	oz
Tcl	e
TI-83-BASIC	v
TI-89-BASIC	v
Toka	h
TorqueScript	c
TPP	f
TUSCRIPT	可能
TXR	可能
UNIX-Shell	e
Ursala	eoz
VBA	i
VBScript	i
Verilog	c
VHDL	f
Vim-Script	不可能
Visual-Basic	iw
Visual-Basic-.NET	iw
Vorpal	e
XPL0	不可能
XQuery	o
XSLT	c
XUL	c

4.3 開発したツールの有用性の評価

本研究で開発したツールの有用性を示すためのコードクローンを検出する適用実験を行った．

4.3.1 適用対象と手法

4.2 節の適用実験で，26 種類のオプションによるコメントの除去が可能だった言語の中から，RosettaCode 全体で実装されているタスクが多い言語を上位 50 言語を選び出した．その 50 言語で書かれた RosettaCode 全体のソースコードに対して各言語でコードクローン検出を行い，コードクローンが検出されるかを調べた．このコードクローン検出は，予約語の入力を行っていないので，タイプ 1 までのコードクローンが出力される．コードクローンの最小トークン数は 30 とした．

4.3.2 実験結果

結果，50 言語全てでコードクローンが検出された．表 11 と表 12 に言語名，RosettaCode 全体でその言語で実装されたタスク数と検出されたクローンペア数を示した．

表 11: 検出されたクローンペア数 1

言語名	タスク数	クローンペア数
Tcl	739	4334
Racket	735	122
Python	704	363
Ruby	668	650
J	661	58093
Go	661	391
C	659	647
D	648	933
Perl	646	74
PicoLisp	637	96
Mathematica	633	44
Haskell	631	92
REXX	612	12221
Ada	591	197
Java	582	406
AutoHotkey	555	164
Common-Lisp	525	100
C++	520	931
BBC-BASIC	519	137
Scala	515	255
OCaml	483	111
PureBasic	477	348
Icon	473	59
JavaScript	422	467
Julia	404	54

表 12: 検出されたクローンペア数 2

言語名	タスク数	クローンペア数
Erlang	402	47
Seed7	389	47
Lua	382	28
Fortran	381	208
PL-I	378	120
R	359	24
PHP	345	62
MATLAB	327	117
AWK	320	34
Liberty-BASIC	312	82
Elixir	307	9
Scheme	276	31
NetRexx	274	148
Oz	271	20
Run-BASIC	268	10
Factor	264	8
PowerShell	263	85
Pascal	261	75
UNIX-Shell	251	18
E	250	1177
Prolog	244	148
Smalltalk	232	7
Delphi	231	211
Euphoria	229	27
Objeck	225	17

4.4 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価する実験を行った.

4.4.1 適用対象と手法

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の再現率を算出した．また，検出されたコードクローンそれぞれが，タイプ 1 かタイプ 2 のコードクローンであることを確認し，適合率を算出した．

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いことも挙げられる．

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．

4.4.2 検出精度の指標の定義

本実験では検出精度の指標として，適合率，再現率の 2 つの指標を用いて評価を行った．適合率（precision），再現率（recall）を以下のように定める．

$$
\begin{aligned}
precision &= \frac{|D \cap C(n)|}{|D|} \\
recall &= \frac{|D \cap CL(n)|}{|CL(n)|}
\end{aligned}
$$

DC は，検出されたコードクローンを指す．n はコードクローン検出における最小トークン数である．$C(n)$ はソースコード上に含まれる長さ n トークン以上のコードクローン全体の集合を表す．$CL(n)$ はあらかじめリストアップした長さ n トークン以上のコードクローンの集合を指す．また，$|D|$ は，集合 D の要素数を表している．

再現率の分母が $C(n)$ にすることが出来ないのは，本研究で用いたトークン分割の定義を用いたコードクローン検出ツールがなく，他のコードクローン検出ツールと比較が出来ないからである．

4.4.3 実験結果

実験の結果を，表 13 に示す．結果，検出されたコードクローンは全て正しく，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．

表 13: コードクローン検出の精度

言語名	検出数	母数	適合率	再現率
C	9	9	1.0	1.0
C++	7	7	1.0	1.0
Go	17	17	1.0	1.0
Java	8	8	1.0	1.0
Python	3	3	1.0	1.0
Ruby	3	3	1.0	1.0

5　まとめと今後の課題

本研究では，柔軟に変更可能な字句解析機構を持つ，トークン単位のコードクローン検出ツールを開発した．コメントルールと予約語のユーザによる理解があれば，字句解析のコーディングを行わなくても，コードクローン検出が行える仕組みを持っている．

適用実験によって，本研究で開発したコードクローン検出ツールを使用して，コメントオプションによって多くの言語でのコメント除去が可能であることを示した．また複数言語でタイプ 1 のコードクローン検出が可能であり，予約語を入力することでタイプ 2 のコードクローン検出が可能であることを示した．

今後の課題として，字句解析が出来なかった言語に対して対応するためにツールを改良することが考えられる．具体的には，文法に関する様々なオプションの作成がある．しかし，オプションの種類を増やすと，どのオプションを設定すれば対象言語のコードクローン検出が可能か不明瞭になってしまい，最終的にユーザに求める労力が増大してしまうのが問題である．

また，ANTLR[1]などのパーサジェネレータを使用して，コメントルールを読み出し，オプションを自動生成する仕組みを作ることが挙げられる．これにより，パーサジェネレータで対応している言語でコードクローン検出が可能になる．

[1]ANTLR:http://www.antlr.org/

謝辞

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 井上 克郎 教授には，研究に関する多くの御指導及び御助言を賜りました．研究の初期段階においての，ツール開発の方針についての御助言はとても大きなものでした．井上 教授に心より深く感謝いたします．

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 松下 誠 准教授には，研究の各段階において多くの御助言を賜りました． 松下 准教授に心より深く感謝いたします．

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 石尾 隆 助教には，研究において常に貴重な御意見を賜りました． 石尾 助教に心より深く感謝いたします．

名古屋大学大学院情報科学研究科附属組込みシステム研究センター/ 情報システム学専攻 吉田 則裕 准教授には，常に研究に関する直接の御指導を賜りました．たくさんの熱心な御指導により，本論文を完成することができました．吉田 准教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科 春名 修介 特任教授には，常に適切な御指導及び御助言を賜りました． 春名 特任教授に心より深く感謝いたします．

奈良先端科学技術大学院大学情報科学研究科ソフトウェア設計学研究室 崔 恩瀞 助教には，研究に関する多くの貴重な御助言を賜りました．崔 恩瀞助教に心より深く感謝いたします．

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 中村 勇太 氏， 沼田 聖也 氏には，本論文の修正と添削を行っていただくなど，日頃から研究に関してたくさんのご協力を頂きました．心より深く感謝いたします．

最後に，研究室内での生活において私を支えてくださった， 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井上研究室の皆様に，心より深く感謝いたします．

参考文献

[1] 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.

[2] 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.

[3] 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.

[4] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 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] 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.

[6] 神谷年洋. the archive of CCFinder Official Site. http://www.ccfinder.net/.

[7] Kazunori Sakamoto, Kiyofumi Shimojo, Ryohei Takasawa, Hironori Washizaki, and Yoshiaki Fukazawa. Occf: A framework for developing test coverage measurement tools supporting multiple programming languages. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pp. 422–430. IEEE, 2013.

[8] Simone Livieri, Daniel M. German, Katsuro Inoue. A needle in the stack: efficient clone detection for huge collections of source code. Technical report, 2010.

[9] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 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.

[10] 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, Vol. 28, No. 4, pp. 43–56, 2011.

[11] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく高速な関数クローン検出. 情報処理学会論文誌, Vol. 55, No. 10, pp. 2245–2255, oct 2014.

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pp. 96–105. IEEE Computer Society, 2007.

[13] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pp. 253–262. ACM, 2004.

[14] 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, mar 2007.

[15] 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.

[16] RosettaCode. http://rosettacode.org/wiki/Rosetta_Code.

[17] プログラミング言語版ロゼッタストーン「Rosetta Code」. http://www.softantenna.com/wp/webservice/rosetta-code/.

[18] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in rosetta code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pp. 778–788, Piscataway, NJ, USA, 2015. IEEE Press.