grep風コードクローン検索ツールの提案

宮本 裕也[1,a)]　井上 克郎[1,b)]

概要：
コードクローンとは，ソースコード中に含まれている互いに一致している，または類似しているコード片のことである．一般に，コードクローンの存在はソフトウェアの保守や管理を困難にすると言われている．そのため，集約や同時修正など，コードクローンに対する様々な保守や管理の方法や，コードクローンを自動的に検出するためのコードクローン検出手法が提案されている．既存のコードクローン検出ツールの多くは，検索対象から全てのクローンペアを検出するものであるが，指定したクエリのコード片に対し，クローンペアとなるコード片を効率よく検出するように設計されたツールの研究はほとんど無い．
本研究では，コードクローン検索ツール ccgrep を開発した．本ツールは grep に倣った UI を使用し，与えられたクエリにマッチするコード片をコードクローンとして検出する．トークン単位で検索を行い，grep より複雑な検索を簡単に行うことができる．本ツールと他のツールとの検索クエリの作りやすさや検出性能，検索時間などを比較し，本ツールの有用性を確認した．

キーワード：コードクローン，パターンマッチング，grep，正規表現

1. まえがき

コードクローンとは，ソースコード中に含まれている互いに一致している，または類似しているコード片のことである．一般に，コードクローンの存在はソフトウェアの保守や管理を困難にすると言われている [11]．そのため，集約や同時修正など，コードクローンに対する様々な保守や管理の方法が提案されている [1], [12]．しかし，ソースコードの規模が大きくなると，含まれるコードクローンの量も大きくなり，手作業で管理することは難しくなる．そこで，コードクローンを自動的に検出するための様々なコードクローン検出手法が提案されている [5], [11]．既に開発されているコードクローン検出ツールの多くは，検索対象から全てのクローンを検出する全クローン検索である．一方，クエリを指定して，そのクエリとクローンペアを構成するコード片を検出するツールは，バグ修正やコーディングパターン検索などのケースで有用であるが，十分な研究がなされていない．

本研究では，コードクローン検索ツール ccgrep を開発した．本ツールは，与えられたクエリにマッチする文字列を検索するツールである grep に対して，与えられたクエリにマッチするコード片をコードクローンとして検出するコードクローン検索ツールである．また，grep に倣った UI やオプションを採用しているため手軽であり，高速な動作により対話的に使用することができる．

本ツールは，空白やコメントの無視など，grep より複雑な検索を簡単に行うことができる．また，本ツールの検索は，ソースコードをトークン列に分解し，トークン単位でマッチングを取ることで行う．クエリからパーサを構築し，このパーサを使用して検索対象のトークン列とのマッチングを取る．さらに，識別子やリテラルの違いを吸収してタイプ 2 クローンやパラメータ化されたクローンを検索することができ，メタ記号を用いてクエリを指定することで，トークン単位の正規表現や “括弧の釣合いがとれたトークン列” のようなパターンにマッチすることができる．これらの機能により，タイプ 3 クローンまでのコードクローンを検索することができる．

評価として，grep とのクエリの作りやすさの比較，タイプ 1，2 クローン検索の容易性，検出性能，検索に要する時間の比較を行い，本ツールの有用性を確認した．

以降，2 章では研究背景について詳細に述べる．続いて，3 章では開発したコードクローン検索ツールの仕様を，4 章では検索の実装について述べ，5 章では評価実験について述べる．最後に，6 章ではまとめと今後の課題について述べる．

[1] 大阪大学
[a)] yuy-mymt@ist.osaka-u.ac.jp
[b)] inoue@ist.osaka-u.ac.jp

2. 背景

2.1 コードクローン

コードクローンとは，ソースコード中に含まれている互いに一致しているまたは類似しているコード片である．一般に，コードクローンの存在はソフトウェアの保守を困難にすると言われている [11]．コードクローンは主に，既存のソースコードがコピーアンドペーストによって再利用されることで生じる．他の発生要因として，定型処理による発生や，コードの自動生成による発生，偶然一致することによる発生なども挙げられる [7]．また，互いにコードクローンになるコード片の対をクローンペアと呼ぶ．

Roy らはコードクローンの定義として，コードクローン間の違いの度合いに基づいて 4 つのタイプに分類している [9]．

タイプ 1　空白やコメントの有無・コーディングスタイルなどの違いを除き完全に一致する．

タイプ 2　タイプ 1 の違いに加えて識別子名・変数の型・リテラルなどが異なる．

タイプ 3　タイプ 2 の違いに加えて文の挿入・削除・変更が行われている．

タイプ 4　構文は異なるが類似した処理を実行する．

また，変数名などのユーザー定義名や変数の型，リテラルを区別するかという観点で，以下の 2 つの分類も存在する．

リネームされたクローン　変数名などのユーザー定義名，変数の型，リテラルなどがクローン間で異なるコードクローン．以下に示すコード片 1，2，3 はすべて，互いにリネームされたクローンである．

パラメータ化されたクローン　リネームされたクローンのうち，変更に一貫性があるコードクローン．コード片 1，2 は互いにパラメータ化されたクローンであるが，コード片 3 は，i と j の位置が入れ替わっているためパラメータ化されたクローンでない．

コード片 1
```
if (a < b) { b++ ; a = 1; }
```

コード片 2
```
if (i < j) { j++ ; i = 1; }
```

コード片 3
```
if (i < j) { i++ ; j = 1; }
```

2.2 コードクローン検出手法

コードクローンが存在すると，ソフトウェアの保守が困難になる．このため，コードクローンに対して，集約や同時修正など保守や管理が行われる [1], [12]．しかし，ソースコードの規模が大きくなると，コードクローンの量も大きくなり，手作業で管理することは難しくなる．そこで，コードクローンを自動的に検出するための種々のコードクローン検出手法が提案されている [5], [11]．検出手法には以下のものが存在する．

トークン単位の検出　ソースコードをトークンの列に変換し，条件を満たす部分列をコードクローンとして検出する．神谷らの CCFinder[4] など．

抽象構文木を用いた検出　ソースコードを構文解析して抽象構文木を構築し，抽象構文木上にある同形の部分木をコードクローンとして検出する．Jiang らの DECKARD[3] など．

プログラム依存グラフに基づく検出　プログラムの制御やデータの依存関係を表すプログラム依存グラフ（PDG）を構築し，部分グラフの同型判定を行うことによりコードクローンを検出する．Komondoor らの手法 [6] など．

2.3 既存のコードクローン検出手法の問題点

前述の CCFinder など，一般に知られているコードクローン検出ツールの多くは，検索対象から全てのクローンを検出する “全クローン検索” を行う．一方，クエリにマッチするコード片をコードクローンとして検出する “特定クローン検索” は以下のようなケースで有用であるが，これを行うツールは少なく，十分な研究がなされていない．

- ソフトウェア中でバグを含むコード片を見つけたときに，そのコードクローンにもバグがあると考えられるため検索して修正する．
- 特定のコーディングパターンを検索してリファクタリングなどを行う．

既存の全クローン検出ツールは，一般にインストールや実行の設定，準備に手間がかかり，安易に特定クローン検索を行うことはできない．また，多くの環境で用いることのできる，正規表現を使用した文字列検索を行うツールである `grep` が存在する．`grep` を特定クローン検索のために使用できるが，以下のような問題があり，コードクローンに特化した検索は難しい．

- 空白・コメントを無視できない．
- 識別子やリテラルへのマッチングが複雑．
- 入れ子になった括弧へのマッチングができない．

既存の特定クローン検索を行うツールとして，Li と Ernst が開発した CBCD（Cloned Buggy Code Detector）[8] がある．このツールはバグのある既知のコード片から，類似したコード片をバグの可能性のあるコード片として検出するツールである．コード片と対象システムのプログラム依存グラフ（PDG）を構築し，部分グラフの同型判定を行うことにより検索を行うが，PDG の構築のために長い時間を必要とする．

石尾らは特定クローン検索ツールとして NCDSearch[2] を開発した．このツールは，クエリの文字列と検索対象の部分文字列から，デフレートアルゴリズムを用いて標準圧縮距離を計算し，距離が閾値以下であるような文字列をコードクローンとして検出する．しかし，このツールは文

字列の圧縮距離での比較を行うため，検出されるコードクローンのタイプや構造を詳細に制御することはできない．また，出力は検出したコード片の位置と圧縮距離のみであり，コード片の内容をすぐに確認できない．

このように，既存ツールには以下に示す要求を満たすものは無い．

- クエリに対するコードクローンを対象ソースコードから検索する．
- トークン列の粒度でマッチングを行い，空白などをクエリで指定する必要がない．
- メタ記号などを用いたクエリにより詳細に制御された検索ができる．
- 検索の結果をすぐに確認でき，対話的な検索ができる．

3. 提案手法

本研究では既存の特定クローン検索ツールでは不可能な，柔軟で手軽に使用できる特定クローン検索ツールとして ccgrep を開発した．本ツールは，ソースコードから検索クエリにマッチするコードクローンを検索することができるコードクローン検索ツールであり，GNU grep[*1] を参考にして作られた UI やコマンドオプションで使用できる．また，コメントや空白を無視することができるため，パラメータ化されたクローンのみを検索するなど，コードクローンに特化した検索を行うことができる．さらに，正規表現や，“括弧の釣合いがとれたトークン列” へのマッチングなどにより，タイプ 3 クローンまでのコードクローンを検索することができる．本ツールの実装には Java を使用しており，多様な実行環境で稼働できる．

本章では，本ツールの検索クエリとマッチングの仕様，出力の仕様について説明する．また，検索例を示す．

3.1 検索の仕様

3.1.1 言語

本ツールが検出することのできる言語は以下に示す 4 言語である（括弧内は対応する拡張子）．ディレクトリ内のファイルを再帰的に検索する場合，各言語に対応する拡張子を持つファイルが検索対象となる．言語は’-l’ オプションで明示的に指定するか，クエリをファイルで指定する場合は言語をクエリファイルの拡張子から推論される．デフォルトでは Java が指定される．

- C （c，h）
- C++ （cpp，cc，c++，cxx，c，h，hpp）
- Java （java）
- Python （py）

3.1.2 検出するクローン

本ツールが検出できるクローンは，タイプ 1，タイプ 2，タイプ 3 クローンである．コマンドオプションやクエリによって，リネームされたクローンやパラメータ化されたクローンなど，どのようなクローンを検出するかを制御することができる．デフォルトのコマンドオプションで，クエリにメタ記号を使用しない場合は，タイプ 2 のパラメータ化されたクローンのみを検出する．

識別子やリテラルの区別においてトークンは以下の 2 つのグループに分けられ，同じグループのトークン同士がリネーム，パラメータ化の対象となる．

識別子　ユーザ定義の識別子，int などの組み込み型．

リテラル　整数，浮動小数，文字，文字列などのリテラル．

識別子やリテラルの区別の度合いは，-b オプションによって以下に示す完全一致，リネーム，パラメータ化の 3 つが指定できる．

-b none　完全一致する識別子，リテラルのみにマッチする．図 1 のクエリと検索対象を使用した場合，対象 1 のみが検出される．

-b full　識別子・リテラルはすべて同一視し，リネームされたクローンにマッチする．図 1 のクエリと検索対象を使用した場合，対象 1，2，3 すべてが検出される．

-b consistent （デフォルト）　識別子はパラメータ化，リテラルはリネームされ，パラメータ化されたクローンにのみマッチする．図 1 のクエリと検索対象を使用した場合，対象 1，2 は検出されるが対象 3 は検出されない．

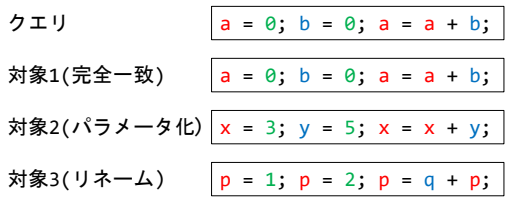

図 1　-b オプションで検出されるコードクローンの違い

3.2 検索クエリの仕様

検索に使用するクエリには任意のコード片が使用できる．クエリはコマンド内文字列，ファイル，標準入力から指定できる．

文字列
```
ccgrep 'int a = 0;'
```
ファイル
```
ccgrep -f query.c
```
標準入力
```
ccgrep -s
```

本ツールでは，クエリ内で “$” から始まるメタ記号を導入し，クエリ内で “識別子の固定”，“括弧の釣合いがとれたトークン列”，“正規表現” を指定するために使用できる．

3.2.1 $id — 識別子の固定

デフォルトのコマンドオプションでは，タイプ 2 クローンを検出するために識別子の文字列は無視されるが，特定の識別子を完全一致させたい場合は，$id のように$に続

[*1] http://www.gnu.org/software/grep/

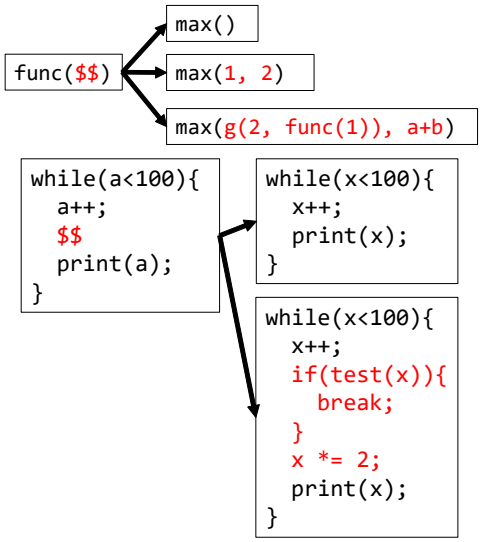

図 2　$$によるマッチングの例

```
1: func() {
2:    int a = 1;
3:    if(flag) {
4:      a *= 2;
5:    }
6:    print(a);
7: }
```

図 3　func(){$$} にマッチするコード片

けて識別子 id を記述することで，id にのみマッチさせることができる．例えば，変数 index の宣言は “T $index;” で検索できる．すべての識別子を固定する場合は，コマンドオプション “-b none” を指定する．また “--fix=id” を指定すると，クエリに含まれる識別子 id のみをすべて固定することができる．例えば，クエリを “a = a + c;” として “--fix=a” を与えると，“$a = $a + c;” として扱われる．

3.2.2　$$ — 括弧の釣合いがとれたトークン列

クエリ内に “$$” と記述すると，括弧（()”，{}，[]）の釣合いがとれた任意長のトークン列に最短マッチする（図 2）．“$$” は開いた括弧がすべて閉じた状態で，クエリでその “$$” の次に記述されたトークンが現れるまでマッチし続ける．例えば，“func(){$$}” をクエリとすると図 3 に記すコード片全体にマッチする．5 行目の “}” の時点でマッチを終了すると 3 行目の “{” が開いたままとなってしまうため無視され，7 行目の “}” の時点ではすべての括弧が閉じているためマッチを終了する．

3.2.3　正規表現

以下に示す 4 種の記号は正規表現として使用できる．

3.2.3.1　$| — 選択

“$|” で区切られたそれぞれのパターンのうち最長のものにマッチする．例えばクエリを “a() $| a+b $| a+b-c”，対象を “x + y - z” とした場合，“x+y-z” にマッチする．

3.2.3.2　$* $+ $? — 繰返し

直前に指定されたパターンの繰返しにマッチする．それぞれ以下の繰り返しに対応する．

$*　0 回以上の繰り返し．

$+　1 回以上の繰り返し．

$?　0 回または 1 回の繰り返し．

パラメータ化されたクローンを検出する場合，繰り返しのマッチング内で初めて登場する識別子は，繰り返し 1 回のマッチングを終えると，次の同じ繰り返しや，その繰り返し以降のクエリで同じ識別子が登場しても，別のものとして扱われる．例えば，“$(a a$)$* ; a” をクエリとした場合，“x x y y z z ; c” のような対象にマッチできる．

制限として，繰返しのマッチングは直前に指定されたパターンにマッチする限り続く．例えば，“a$*a” をクエリとすると “aaa” にはマッチしなくなる．クエリの “a$*” の部分が “aaa” 全体にマッチし，クエリ最後の “a” がマッチしないためである．

3.2.3.3　$($) — グルーピング

$(と $) で囲まれた部分を 1 つのパターンとしてまとめる．選択（$|）の範囲を制限したり，繰り返し（$*）の対象を指定したりするために使用できる．

選択の範囲を制限　`for(;a<10; $( a++ $| ++a $) )`

繰り返しの対象を指定　`$( T x = 0; $) $+`

3.2.3.4　$. — 任意 1 トークン

任意のトークン 1 つにマッチする．トークンが存在しない場合はマッチしない．

3.3　検索対象の仕様

検索対象には任意のコード片が使用できる．ファイル，ディレクトリ，標準入力で与えることができる．ディレクトリを指定する場合は，そのディレクトリ内のファイルで，指定された言語の拡張子を持つもののみが検索対象となる．

ファイル　`ccgrep 'query' file`

ディレクトリ　`ccgrep 'query' directory/ -r`

標準入力　`ccgrep 'query' -`

3.4　出力の仕様

検出されたクローンは標準出力へ出力される．デフォルトの出力内容は，1 つのクローンにつき 1 行で，そのクローンのファイル名と 1 行目のテキストである．コマンドオプションにより，行番号の出力や，コードクローンのテキスト全体の出力も可能である．外部のスクリプトやツールで処理しやすいように，JSON 形式や XML 形式での出力にも対応している．

“ccgrep 'for($$){$$}' -l java -r ccgrep/” に出力オプションとして，“行番号の出力（-p n）”，“行番号の出力とコードクローン全体の出力（-p nf）” を指定し

た場合の出力例を図 **4**，図 **5** に示す．“`-p nf`” の出力形式では，コードクローンのファイル名が出力され，その次の行からそのコードクローンのテキスト全体が出力される．

また，コマンドの終了コードとして，コードクローンを 1 つ以上検出した場合は 0 を，コードクローンが検出できなかった場合は 1 を，エラーが発生した場合は 2 を返す．

3.5 検索クエリ例

本ツールで使用可能な検索クエリの例を以下に列挙する．

引数なしの関数 max の呼び出し

```
$max()
```

0 個以上の引数を持つ関数 max の呼び出し

```
$max($$)
```

第 1 引数が変数 buf である関数 print の呼び出し

```
$print($buf, $$)
```

任意の関数宣言

```
T f($$) { $$ }
```

getter 関数宣言

```
T f() { return this.v; }
```

setter 関数宣言

```
void f(T v) { this.v = v; }
```

関数 max の宣言

```
T $max($$) { $$ }
```

return 文のみの関数の宣言

```
T f($$) { return $$; }
```

引数のオブジェクトに処理を委譲する関数の宣言

```
T f(U obj) { return obj.m(); }
```

変数 num への代入

```
$num = $$;
```

2 つの変数を比較し大きい方を返す 3 項演算子の式

```
a < b?  b:  a
```

if 文

```
if ($$) {$$}
```

最後に return する if 文

```
if ($$) { $$ return $$; }
```

else ブロックのある，またはない if 文

```
if ($$) {$$}  $(else {$$} $) $?
```

制御変数を持つ for 文

```
for(i=0; i<$$; i++) {$$}
```

for 文または while 文

```
for($$){$$} $| while($$){$$}
```

4. 実装

本ツールの検索アルゴリズムは以下のステップに分けられる．それぞれのステップについて説明する．

手順 1　パーサの構築

手順 2　マッチングと検索結果の出力

表 1　マッチング手順例 (対象 1)

手順	位置	ノード	状態
1	1(x)	1(a)	成功．“a↔x” をテーブルに保持．
2	2(<)	2(<)	成功．
3	3(y)	4(b)	成功．“b↔y” をテーブルに保持．
4	4(?)	5(?)	成功．
5	5(y)	6(b)	成功．“b↔y” にマッチ．
6	6(:)	7(:)	成功．
7	7(x)	8(a)	成功．“a↔x” にマッチ．全体が成功．

位置 = 検索対象のマッチングトークン位置
ノード = パーサの現在ノード

4.1 パーサの構築

この手順では，検索のマッチングに使用するパーサを構築する．まず，与えられたクエリを，ANTLR（ver. 4.7.1）[*2] を使用してトークンに分解する．ANTLR は，独自の文法によりトークンの規則を定義することで，ソースコードのトークン分解を行うことができるツールである．本ツールでは，検索対象のプログラミング言語の文法に，クエリに使用するメタ記号（`$Id`，`$$`，`$(`，`$)`，`$|`，`$*`，`$+`，`$?`，`$.`）を追加した文法による字句解析器を使用している．クエリをトークン分解した後，そのトークン列から正規表現などのメタ記号により構文解析を行い，パーサを構築する．マッチングはこのパーサを使用した最左導出により行う．ここまでの処理の手順を図 **6** に示す．

4.2 マッチングと検索結果の出力

手順 2 では，以下の処理をファイル単位で繰り返す．

手順 2A　コードクローンのマッチング

手順 2B　検索結果の出力

4.2.1 コードクローンのマッチング

コードクローンのマッチングは，検索対象のファイルを ANTLR でトークンに分解し，そのトークン列の前方から，1 トークンずつ先頭位置をずらしながらパーサがマッチするか確かめていく．

パラメータ化されたクローンの判定には，マッチングの先頭位置ごとに識別子の対応テーブルを作り，出現した識別子の対応関係を作っていくことにより行う．選択のマッチングでは，各パターンのマッチングが失敗すると，次のパターンをマッチングする前に，テーブルを選択のマッチング開始時点の状態に戻す．繰返しのマッチングでは成功・失敗に関わらず，各パターンのマッチングが終了するたびにテーブルを繰返しのマッチング開始時点の状態に戻す．

マッチングの進行は，図 **7**，図 **8** に示す検索クエリと検索対象を使用した場合，それぞれ表 **1**，表 **2** に示す手順で行われる．

4.2.2 検索結果の出力

ファイル内を検出し終えると，コマンドオプションで指

[*2] https://www.antlr.org/

```
ccgrep/src/antlr/CLexer.java:144:    for (int i = 0; i < tokenNames.length; i++) {
ccgrep/src/antlr/CLexer.java:696:    for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
```

図 4　“-p n” の出力例

```
ccgrep/src/antlr/CLexer.java
144:    for (int i = 0; i < tokenNames.length; i++) {
145:        tokenNames[i] = VOCABULARY.getLiteralName(i);
146:        if (tokenNames[i] == null) {
147:            tokenNames[i] = VOCABULARY.getSymbolicName(i);
148:        }
149:
150:        if (tokenNames[i] == null) {
151:            tokenNames[i] = "<INVALID>";
152:        }
153:    }
ccgrep/src/antlr/CLexer.java
696:    for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
697:        _decisionToDFA[i] = new DFA(_ATN.getDecisionState(i), i);
698:    }
```

図 5　“-p nf” の出力例

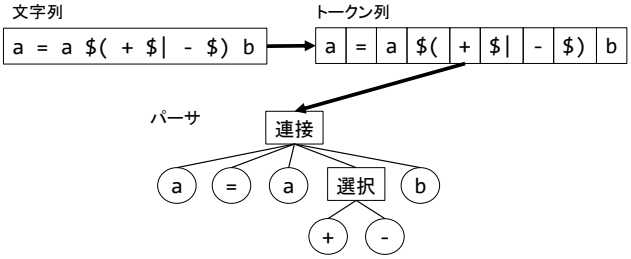

図 6　パーサの構築

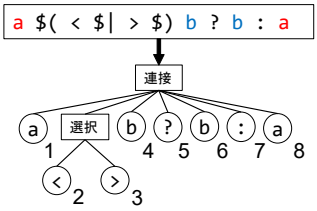

図 7　マッチング例の検索クエリとそのパーサ

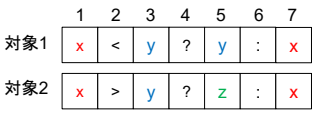

図 8　マッチング例の検索対象

表 2　マッチング手順例 (対象 2)

手順	位置	ノード	状態
1	1(x)	1(a)	成功．“a↔x” をテーブルに保持．
2	2(>)	2(<)	失敗．選択 2 つ目のパターンへ．
3	2(>)	3(>)	成功．
4	3(y)	4(b)	成功．“b↔y” をテーブルに保持．
5	4(?)	5(?)	成功．
6	5(z)	6(b)	失敗．“b↔y” に非マッチ．全体失敗．

定された出力形式で検出したコードクローンを出力する．

5. 評価実験

grep や他のクローン検出ツールとの比較することにより，本ツールの評価を行った．評価項目は検索コマンドの作りやすさ，タイプ 1，2 クローン検索の容易性，検出性能・検索時間の 3 項目である．

5.1 検索コマンドの作りやすさ

同等のコードクローンを検索するための検索クエリやコマンドオプションを，本ツールと grep でどのような違いがあるかを例示することにより，本ツールの検索コマンドの作りやすさを示す．

5.1.1 空白やコメントの処理

```
grep '\s*int\s+a\s*=\s*b\s*;' -r src/
```

```
ccgrep '$int $a = $b;' -r src/
```

これは初期値を b とする int 変数 a の宣言を検索するクエリである．grep では，空白を処理するために “\s*” などの記述を挟む必要があり，コメントを処理するためにはさらに記述を増やす必要がある．一方，本ツールは自動的に空白やコメントを無視するため記述は不要である．

5.1.2 特定の関数の呼び出しへのマッチング

```
grep 'time(' -r src/ -w
```

```
ccgrep '$time($$)' -r src/
```

これは time 関数の呼出しを検索するクエリである．grep では，time の関数名と関数呼び出しの開き括弧によって time 関数の呼び出しへのマッチングを行っている．また，counttime のような異なる識別子にマッチしないように “-w” によりワード単位でマッチするように指定している．他にも，コメントなどに含まれているものにもマッチしてしまうなど，マッチングを適切に制限しなければ関係の無いものが大量にマッチしてしまう．一方，本ツールは，関数名がリネームされないように “$” で固定する記述が必要になるが，time と開き括弧の間に空白があってもマッチし，また “$$” によって関数呼び出しの引数部分へのマッチさせることもできる．

5.1.3 リネームされたクローンへのマッチング

```
grep '[a-zA-Z_][a-zA-Z_0-9]*
    [a-zA-Z_][a-zA-Z_0-9]*;' -r src/
```

```
ccgrep 'T a;' -r src/ -b full
```

これは任意の変数宣言を検索するクエリである．grep では識別子へのマッチングごとにパターンを記述しているが，本ツールではコマンドオプション “-b full” を指定するだけでクエリ内すべての識別子がリネームされ，リネームされたクローンを検出することができる．

5.1.4 パラメータ化されたクローンへのマッチング

```
grep '\([a-z_][a-z_0-9]*\)() {
   return [a-z_][a-z_0-9]*.\1(); }'
   -r src
```

```
ccgrep 'f()  return a.f(); ' -r src/
```

これは他のオブジェクトに処理を委譲する関数を検索するクエリである．grep では，タグ付き正規表現 "\(\)" を使用して過去にマッチした文字列と同じもののみにマッチさせることができる．しかし，識別子の量が増えると記述量も多くなってしまう．本ツールは，デフォルトでパラメータ化されたクローンへのマッチングとなる．

5.1.5 括弧の釣合いがとれたトークン列へのマッチング

```
ccgrep 'if(a == b) { $$ }' -r src/
```

これは "a == b" を条件とする if 文を検索するクエリである．grep では，任意の深さで入れ子になった括弧へマッチさせることはできないが，本ツールは，"$$" と記述するだけで，括弧の釣合いがとれたトークン列へマッチさせることができる．

以上に示すように本ツールは，空白やコメントの処理が不要であり，既存コード片をそのままクエリとすることでタイプ 1，2 クローンを検索でき，パラメータ化されたクローンや括弧の釣合いがとれたトークン列へのマッチングを使用して検索を制御できるなど，コード検索において grep より簡単に検索クエリを作成することができた．

5.2 タイプ 1，2 クローン検索の容易性

本ツールのタイプ 1，2 クローン検索の容易性を検証する．検証にはデータセットとして BigCloneBench[10] を使用した．このデータセットに含まれるタイプ 1，2 のクローンペアに対して，その一方のコード片をそのままクエリとしてもう一方のコード片を検索する．この検出ができ，また逆方向にも検出できたクローンペアを検出できたクローンペアとする．

結果を**表 3** に示す．検出できなかった 7 個のクローンペアのコード片を確認すると，実際はタイプ 1，2 クローンではなかった．これはデータセットのエラーであると考えられるため，これらを無視すればすべてのタイプ 1，2 クローンを検出できている．ccgrep は既存コード片をそのままクエリとして与えるだけで，タイプ 1，2 のクローンペアを全て検索できることが確認できた．これらのクローン以外にも ccgrep はメタ記号などを使用することで検索をさらに制御することができる．

5.3 検出性能・検索時間

表 4 に示す 5 個の対象プロジェクトに対し，次に示すクエリ 1〜4 を使用して本ツールの検索時間を計測した．コマンドオプションはデフォルトのものを使用している．

表 3 BigCloneBench から検出できたクローン

タイプ	総ペア数	検出ペア数	非検出ペア数
タイプ 1	48,116	48,111	5*
タイプ 2	4,234	4,232	2*
合計	52,350	52,343	7*

*実際のクローンペアではない．

表 4 検索時間の実験対象プロジェクト

対象	言語	ファイル数	行数
ANTLR	Java	678	59,511
Ant	Java	1,272	138,396
Git	C	339	90,495
PgSQL	C	904	177,174
Linux	C	15,123	3,756,212

ANTLR: ANTLR4 v.4.7.2, Ant: Apache Ant v.1.10.5, Git: v.1.6.4.3, PgSQL: PostgreSQL v.6.5.3, Linux: Linux kernel v.2.6.14rc2

表 5 実験環境

OS	Windows 10 Pro for Workstations 64bit
CPU	Intel(R) Xeon(R) CPU E5-1603 v4 @ 2.80GHz
RAM	32.0GB

クエリ 1 `a < b?  a:  b` 三項演算子を用いた変数の比較

クエリ 2 `T1 f(T2 a) { return $$; }` すぐに return する 1 引数関数

クエリ 3 `f($$, $$, $$);` 3 個以上の引数を持つ関数

クエリ 4
```
for(a = 0; a < $$; a++) { $$ }
$| a = 0; while(a < $$) { $$ a++; }
```
制御変数を持つループ文

また，比較ツールとして grep と NCDSearch[2] の検索時間を計測した．NCDSearch は標準圧縮距離を利用してコード片を検索するツールである．grep のクエリには，次に示すクエリ 5 とコマンドオプションを使用しており，改行を含む対象を検出できないことを除いてクエリ 1 と同等である．NCDSearch のクエリには，本ツールと同様のクエリ 1 を使用した．実験環境は**表 5** に示す．

クエリ 1'
```
([a-zA-Z_][a-zA-Z_0-9]*)\s*<\s*
([a-zA-Z_][a-zA-Z_0-9]*)\s*\?\s*
\1\s*:\s*\2
```

オプション `-w --include='*.[ch]'`

本ツールの実験結果を**表 6** に，grep と NCDSearch の実験結果を**表 7** に示す．

grep は本ツールと比較すると 4.11〜7.55 倍高速である．これは，grep が文字列ベースでありトークン分解などの処理が不要であることや，高速なアルゴリズムを使用していることが要因であると考えられる．また，grep は改行を含むコード片を検出できなかった．一方，NCDSearch は 0.07〜0.28 倍の検索時間であり，本ツールの方が高速に検索することができた．また，NCDSearch は正しくない検

表 6　本ツールの検索時間

対象		ANTLR	Ant	Git	PgSQL	Linux
クエリ 1	検出数	0	2	8	3	48
	時間 [s]	1.56	2.10	1.51	2.23	25.29
クエリ 2	検出数	159	161	7	27	543
	時間 [s]	1.55	2.18	1.53	2.27	26.42
クエリ 3	検出数	1,710	2,487	5,717	10,603	187,653
	時間 [s]	1.69	2.48	1.69	2.55	31.79
クエリ 4	検出数	1	13	442	621	10,754
	時間 [s]	1.62	2.40	1.70	2.47	30.90

表 7　比較ツールの検索時間

対象		ANTLR	Ant	Git	PgSQL	Linux
ccgrep	検出数	0	2	8	3	48
(クエリ 1)	時間 [s]	1.56	2.10	1.51	2.23	25.29
	速度比*	1.00	1.00	1.00	1.00	1.00
grep	検出数	0	1	8	2	44
(クエリ 1')	時間 [s]	0.38	0.51	0.20	0.42	4.66
	速度比*	4.11	4.12	7.55	5.31	5.43
NCDSearch	検出数	3	21	22	80	21,047
(クエリ 1)	時間 [s]	5.56	11.79	8.32	16.57	361.55
	速度比*	0.28	0.18	0.18	0.13	0.07

*速度比=（Query1 を用いた ccgrep の検出時間）/（比較ツールの検出時間）.

索結果を多く含んでいた.

本ツールの検索時間は Linux kernel のような多量のファイルを含んだ大規模なプロジェクトに対しても 30 秒程度であり，十分に対話的な使用が可能である．また，grep は改行のあるコード片にマッチできず，NCDSearch は正しくないコード片が多く検出されたため，検出精度でも比較ツールに対して本ツールに優位性がある.

6. まとめと今後の課題

本研究ではコードクローン検索ツール `ccgrep` を開発した．本ツールは，`grep` と同様に，与えられたクエリにマッチするものをコードクローンとして検出する．`grep` に倣った UI を採用しており，手軽に使用できる．本ツールによる検索は，トークン単位でソースコードのマッチングを取ることにより行う．空白やコメントの無視や，識別子のパラメータ化などにより，`grep` より複雑なコードクローンに特化した検索を簡単に行うことができる．また，独自のクエリ記法を使用することにより，タイプ 3 クローンまでのコードクローンを検索することができる．評価として，`grep` と比較したクエリの作りやすさの比較，検出性能，検索時間を調査し，本ツールの有用性を確認した.

今後の課題として，以下が挙げられる.

検索アルゴリズムの改善　検索アルゴリズムを改善することにより，検索の高速化や省メモリ化を行う.

検索機能の追加　正規表現機能やその他の検索機能を拡充して，より適切な検索を可能にする.

言語の追加　検索対象にすることのできる言語を追加する．字句解析に ANTLR を使用しているため，ANTLR の文法ファイルがあれば比較的容易に言語を追加できる.

評価実験　被験者実験や他ツールとの比較などの実験を行い，本ツールの有用性や扱いやすさを評価する.

謝辞　本研究は JSPS 科研費 18H04094 の助成を受けたものです.

参考文献

[1] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, MA, USA (1999).

[2] Ishio, T., Maeda, N., Shibuya, K. and Inoue, K.: Cloned Buggy Code Detection in Practice Using Normalized Compression Distance, *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, IEEE Computer Society, pp. 591–594 (online), DOI: 10.1109/ICSME.2018.00022 (2018).

[3] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones, *IN ICSE*, pp. 96–105 (2007).

[4] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code, *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7, pp. 654–670 (online), DOI: 10.1109/TSE.2002.1019480 (2002).

[5] Kaur, H., Kaur, R. and Bathinda, T. S.: A Review: Clone Detection in Web Application Using Clone Metrics.

[6] Komondoor, R. and Horwitz, S.: Using Slicing to Identify Duplication in Source Code, *Static Analysis* (Cousot, P., ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 40–56 (2001).

[7] Koschke, R.: Large-Scale Inter-System Clone Detection Using Suffix Trees, *2012 16th European Conference on Software Maintenance and Reengineering(CSMR)*, Vol. 00, pp. 309–318 (online), DOI: 10.1109/CSMR.2012.37 (2012).

[8] Li, J. and Ernst, M. D.: CBCD: Cloned buggy code detector, *2012 34th International Conference on Software Engineering (ICSE)*, pp. 310–320 (online), DOI: 10.1109/ICSE.2012.6227183 (2012).

[9] 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 (online), DOI: https://doi.org/10.1016/j.scico.2009.02.007 (2009).

[10] Svajlenko, J. and Roy, C. K.: Evaluating Clone Detection Tools with BigCloneBench, *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, Washington, DC, USA, IEEE Computer Society, pp. 131–140 (online), DOI: 10.1109/ICSM.2015.7332459 (2015).

[11] 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).

[12] 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（オンライン），DOI: 10.11309/jssst.28.4_43 (2011).