特別研究報告

題目

容易に使用可能な grep 風コードクローン検索ツール

指導教員
井上 克郎 教授

報告者
宮本 裕也

平成 31 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成30年度 特別研究報告

容易に使用可能なgrep風コードクローン検索ツール

宮本 裕也

内容梗概

ソースコードの保守や管理を困難にする要因の一つとして，コードクローンがある．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を困難にすると言われている [10]．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 [2, 11] 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 [7, 10]．既存のコードクローン検出ツールには多くは，検索対象から全てのクローンを検出するものであるが，クエリを指定して，そのクエリにマッチするコード片を検出するためのツールは少ない．

本研究では，grep風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テキストを検索するツールであるgrepと同様に，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．また，grepに倣ったUIを持っており手軽に使用することができる．

本ツールは，ソースコードをトークン分解し，トークン単位で検索する．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，識別子やリテラルの違いを吸収し，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る．さらに，本ツール独自のクエリ記法を使用することで，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．また，本ツールの評価では，他のツールとの検索クエリの作りやすさや検出できるクローンの比較，検索時間の測定を行い，本ツールの有用性を確認した．

主な用語

grep
コードクローン
コード検索

目 次

1 まえがき **3**

2 背景 **4**

2.1 コードクローン 4

2.2 コードクローン検出手法 5

2.3 既存のコードクローン検出手法の問題点 7

3 提案手法 **8**

3.1 検索の仕様 8

3.2 検索クエリの仕様 10

3.2.1 $\$id$ — 識別子の固定 10

3.2.2 $$ — 括弧の釣合いがとれたトークン列 11

3.2.3 正規表現 12

3.3 検索対象の仕様 13

3.4 出力の仕様 14

3.5 検索クエリ・スクリプト例 14

3.5.1 検索クエリ例 14

3.5.2 検索スクリプト例 18

3.6 アルゴリズム 18

3.6.1 クエリからパーサの構築 19

3.6.2 マッチングと検索結果の出力 21

4 評価実験 **24**

4.1 検索コマンドの作りやすさ 24

4.2 検出できるクローンの比較 25

4.3 検索時間 27

5 まとめと今後の課題 **29**

謝辞 **30**

付録 **31**

参考文献 **34**

1　まえがき

ソースコードの保守や管理を困難にする要因の一つとして，コードクローンがある．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を困難にすると言われている [10]．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 [2, 11]．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 [7, 10]．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を行うツールは少なく，十分な研究もなされていない．

本研究では，grep 風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テキストを検索するツールである grep と同様に，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おり，手軽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，識別子やリテラルの違いを吸収し，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る．さらに，本ツール独自のクエリ記法を使用することで，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．

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．また，パラメータ化されたクローンを検出は，マッチング位置ごとに検索クエリと検索対象の識別子の対応関係のテーブルを作成することにより行う．

評価として，grep と比較したクエリの作りやすさ，CCFinderX と比較した検出性能，検索に要する時間を調査した．これにより，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とアルゴリズムについて述べ，4 章では評価実験について述べる．最後に，5 章ではまとめと今後の展望について述べる．

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 [10]．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 [8]．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．

コードクローンの分類

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き以下の 4 つのタイプに分類している [9]．

タイプ 1 クローン

空白やタブの有無，コーディングスタイル，コメントの有無などの違いを除き完全に一致するコードクローン．

タイプ 2 クローン

タイプ 1 の違いに加えて，変数名などのユーザー定義名，変数の型，リテラルなどが異なるコードクローン．

タイプ 3 クローン

タイプ 2 の違いに加えて，文の挿入や削除，変更などが行われているコードクローン．

タイプ 4 クローン

類似した処理を実行するが，構文上の実装が異なるコードクローン．タイプ 4 のコードクローンとして，以下のものが挙げられる．

- 条件分岐処理や繰り返し処理などの制御構造の実装が異なる．
- 中間媒介変数の利用の有無が存在している．
- 文の並び替えが発生している．

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．

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; }

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．前述ののコード片 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．

また，パラメータ化されたクローンはタイプ 2 クローンのみでなく，以下のコード片 4，5 のようなタイプ 3 クローンの場合でも，a，b はパラメータ化され，それぞれ i，j に対応しているとみなす．

コード片 4: if (a < b) { b++ ; a =1; }

コード片 5: if (i < j) { j++ ; i =1; x=2; }

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 [2, 11]．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 [7, 10]．コードクローン検出手法として，字句単位の検出，抽象構文木を用いた検出，識別子名に基づく検出などが存在する．以降，この 3 つの検出手法について説明する．

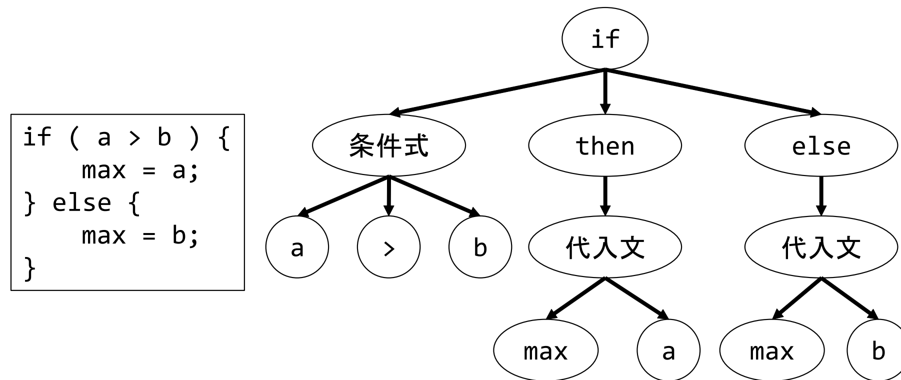

図 1: 抽象構文木の例

字句単位の検出

字句単位の検出は，検出の前にソースコードを字句（トークン）の列に変換する．そして，そのトークン列の中から，一定の条件を満たす部分列をコードクローンとして検出する．

字句単位の検出を用いた代表的なツールとして，神谷らが開発した CCFinder[6] がある．CCFinder は，字句解析により，ソースコードをトークンの列に変換する．この時，一定の規則でトークンを追加，削除，変更し，変数名や関数名などのユーザー定義名を特殊トークンに置き換える．そして，閾値以上の長さで共通したトークン列をコードクローンとして検出する．このツールはタイプ 2 までのコードクローンを検出することができる．

抽象構文木を用いた検出

抽象構文木とは，ソースコードの構文構造を木構造で表現したグラフのことである．抽象構文木の例を図 1 に示す．この検出手法は，前処理としてソースコードを構文解析することにより抽象構文木を構築する．そして，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．この検出手法は，抽象構文木を構築する必要があるため，字句単位の検出と比較して時間的，空間的計算量は大きくなるが，実用的な方法として知られている．

抽象構文木を用いた検出手法の代表的なツールには，Jiang らが開発した DECKARD[5] がある．DECKARD は，抽象構文木の各部分木を特徴ベクトルに変換する．そして，LSH（Locality-Sensitive Hashing）[3] を用い，特徴ベクトル間の類似度を求めることによってコードクローンの検出を行う．このツールはタイプ 3 までのコードクローンを検出することができる．

識別子名に基づく検出

識別子とは，ソースコード中の変数や関数などを識別するトークンのことである．この検出手法は，識別子名に基づいた特徴メトリクスを用いて比較し，類似度が閾値以上となった

箇所をコードクローンとして検出する．字句解析や構文解析を行うなどコンパイラの技術を利用した検出手法とは異なり，情報検索技術を利用して検出を行う．

識別子名に基づく検出の代表的な手法には，山中らが開発した関数クローン検出法 [12] がある．山中らの関数クローン検出法では，情報検索技術の 1 つである TF-IDF 法 [1] を利用して関数を特徴ベクトルに変換し，意味的に類似した処理を行うコードクローンを検出する．この手法はタイプ 4 までのコードクローンが検出可能である．

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，クエリを指定して，その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」を行うツールは少なく，十分な研究はなされていない．

既存のコード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，Linux で用いることのできる，正規表現を使用したテキスト検索を行うツールである grep が存在する．この 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（`[a-zA-Z_][a-zA-Z_0-9]*`のように書く必要がある）．
- 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，石尾らが開発した NCDSearch[4] がある．このツールは，クエリのテキストと検索対象の部分テキストから，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テキストをコードクローンとして検出する．しかし，このツールはテキストでの比較を行っており，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．

3　提案手法

本研究では，コード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，検索アルゴリズムを示す．

3.1　検索の仕様

言語

コードクローンを検出することのできる言語は下記の 4 言語である (括弧内は対応拡張子)．ディレクトリ内ファイルを再帰的に検索する場合，検索対象は各言語に対応する拡張子を持つファイルのみとなる．言語はコマンドで明示的に指定する[2]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)

C，C++では，#define などのマクロはすべて無視される．#if，#else，#elif，#endif については，#if のブロックのみが検索対象となり，#else，#elif のブロックは削除される．マクロが削除される例を図 2 に示す．

また，Java は本来，ソースコードの識別子に「$」を使用できるが，本ツールのクエリ記法の都合上，使用することはできない．

[1] http://www.gnu.org/software/grep/
[2] '-l' オプション

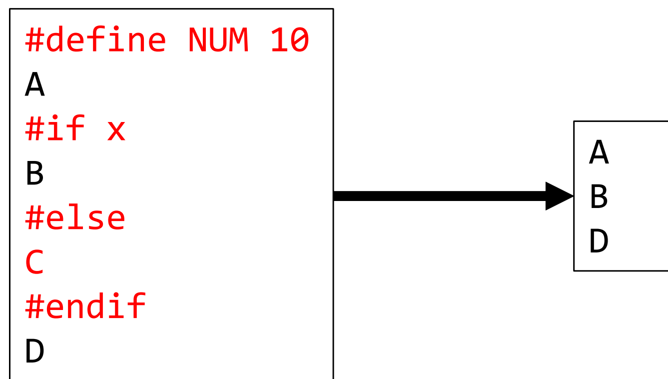

図 2: C，C++で削除されるマクロの例

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，クエリに特殊トークン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．

完全一致 「`-b none`」

完全一致する識別子，リテラルのみにマッチする．図 3 のクエリと検索対象では，対象 1 のみが検出される．

リネーム 「`-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 3 のクエリと検索対象では，対象 1，2，3 すべてが検出される．

パラメータ化（デフォルト） 「`-b consistent`」

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

クエリ	`a = 0; b = 0; a = a + b;`
対象1(完全一致)	`a = 0; b = 0; a = a + b;`
対象2(パラメータ化)	`x = 3; y = 5; x = x + y;`
対象3(リネーム)	`p = 1; p = 2; p = q + p;`

図 3: -b オプションで検出されるコードクローンの違い

みマッチする．図 3 のクエリと検索対象では，対象 1，2 は検出されるが対象 3 は検出されない．

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テキスト，ファイル，標準入力から指定できる．

テキスト： `ccgrep 'int a = 0;'` [3]

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「$」から始まる特殊トークンが使用できる．これは，正規表現などの特殊な記号を表すために使用する．

- 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

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テキストは無視される．変数名や関数名など，特定の識別子をクエリ内で指定したもののみにマッチさせたい場合は，$\$id$ のように，$ に続けて識別子 id を記述することで，id にのみ

[3] 変数展開などを抑制するために，シングルクォートで囲む．

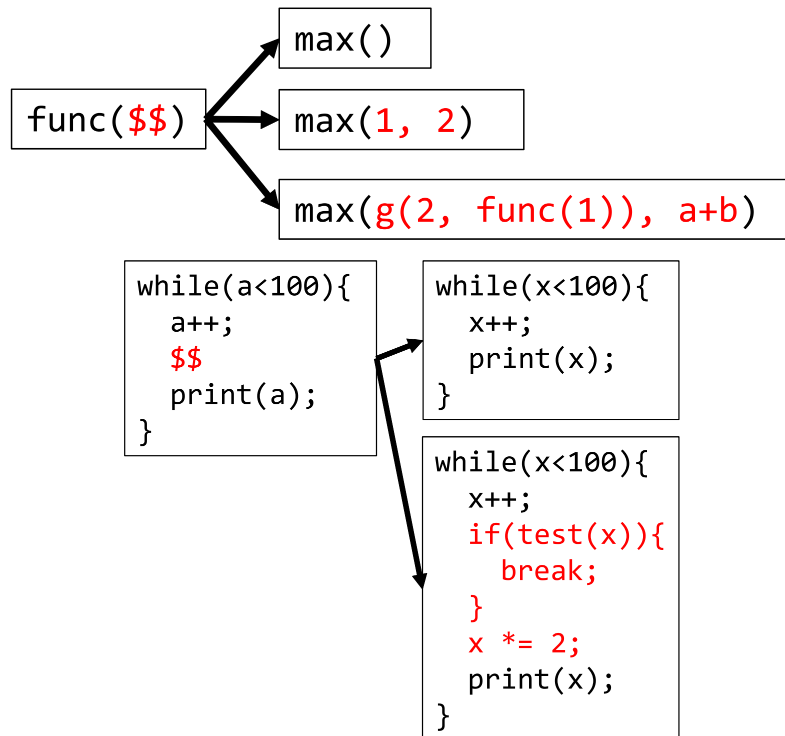

図 4: $$によるマッチングの例

マッチさせることができる[4]．例えば，変数 value の宣言は「`T $value;`」のようにして検索できる．

コマンドオプション「`-b none`」を指定することで，すべての識別子を固定したものとして扱うことができる．例えば，クエリを「`a = a + c;`」とすると,「`$a = $a + $c;`」として扱われる．また,「`--fix=`*id*」を指定すると，クエリ内の識別子 *id* をすべて固定させることができる．例えば，クエリを「`a = a + c;`」として「`--fix=a`」を与えると,「`$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 4）．

$$の中で開いた括弧がすべて閉じた状態で，クエリでその$$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(){$$}`をクエリとすると図 5 に記すコード片全体にマッチする．このクエリで記述された$$は } が現れるとマッチングを終了するが，最初

[4] クエリ内の同名の識別子で，$で固定しているものと固定していないものが混在している場合，$のあるもののみが固定される．

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

図 5: func(){$$} にマッチするコード片

```
1: func():
2:   $$
```

```
1: func():
2:   a = 1
3:   if flag:
4:     a *= 2
5:   print(a)
```

図 6: Python における func(){$$}

に } が現れる 5 行目では，3 行目で開いた { を閉じるために使われるため，$$のマッチングを終了しない．2 回目に } が現れる 7 行目では，$$のマッチング中に開いた括弧はすべて閉じているため，$$のマッチングを終了する．その結果，`func(){$$}`は 1 行目から 7 行目までマッチする．

また，言語を Python に指定した場合は，`()`，`{}`，`[]` に加えて，Python のブロック開始のインデントとブロック終了のインデントの組も括弧として認識する．そのため，Python のブロックも，多言語の`{}`で作られるブロックと同様に認識することができる．図 6 に記すコード片で前述と同等の検索ができる．

3.2.3 正規表現

本ツールでは，以下に示す記号が正規表現として使用できる．

$| — 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 + b $| a + b - c`」，対象を「`x + y - z`」とした場合，「`x + y`」ではなく「`x + y - z`」全体にマッチする．

$* $+ $? — 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．

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．

また，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．

$($) — 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; $) $+

$. — **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．検索対象は複数指定することができる．検索対象の字句解析器はクエリで使用するものと同じものを使用しているが，特殊トークンが含まれていてはならない．

それぞれの指定例を示す．

ファイル： `ccgrep 'query' file`

ディレクトリ： `ccgrep 'query' directory/ -r` [5]

標準入力： `ccgrep 'query' -`

[5] ディレクトリを指定する場合は「-r」を指定する必要がある．

```
ccgrep¥antlr¥c¥CLexer.java:130:                 for (int i = 0; i < tokenNames.length; i++) {
ccgrep¥antlr¥c¥CLexer.java:672:                 for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
ccgrep¥antlr¥cpp14¥CPP14Lexer.java:149:                 for (int i = 0; i < tokenNames.length; i++) {
ccgrep¥antlr¥cpp14¥CPP14Lexer.java:768:                 for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
```

図 7: 「-p n」の出力例

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/ -m 2`」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）」「JSON 形式の出力（–json）」を指定した場合の出力例を図 7，8，9 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・スクリプト例

3.5.1 検索クエリ例

本ツールで使用できる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; }
```

```
ccgrep¥antlr¥c¥CLexer.java
130:            for (int i = 0; i < tokenNames.length; i++) {
131:                    tokenNames[i] = VOCABULARY.getLiteralName(i);
132:                    if (tokenNames[i] == null) {
133:                            tokenNames[i] = VOCABULARY.getSymbolicName(i);
134:                    }
135:
136:                    if (tokenNames[i] == null) {
137:                            tokenNames[i] = "<INVALID>";
138:                    }
139:            }
ccgrep¥antlr¥c¥CLexer.java
672:            for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
673:                    _decisionToDFA[i] = new DFA(_ATN.getDecisionState(i), i);
674:            }
ccgrep¥antlr¥cpp14¥CPP14Lexer.java
149:            for (int i = 0; i < tokenNames.length; i++) {
150:                    tokenNames[i] = VOCABULARY.getLiteralName(i);
151:                    if (tokenNames[i] == null) {
152:                            tokenNames[i] = VOCABULARY.getSymbolicName(i);
153:                    }
154:
155:                    if (tokenNames[i] == null) {
156:                            tokenNames[i] = "<INVALID>";
157:                    }
158:            }
ccgrep¥antlr¥cpp14¥CPP14Lexer.java
768:            for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
769:                    _decisionToDFA[i] = new DFA(_ATN.getDecisionState(i), i);
770:            }
```

図 8: 「-p nf」の出力例

```
{
 "startTime":"2019-02-07T12:18:12.867157+09:00",
 "language":"JAVA9", "blindLevel":"CONSISTENT",
 "queryCode":"for($$){$$}",
 "clonesPerFile":[
  {
   "fileName":"ccgrep¥¥antlr¥¥c¥¥CLexer.java",
   "countLine":651, "countToken":2848, "countClone":2,
   "clones":[
    {
     "startLine":130, "startColumn":3, "startToken":1588, "endLine":139,
"endColumn":3, "endToken":1658,
     "code":"¥t¥tfor (int i = 0; i < tokenNames.length; i++) {¥n¥t¥t¥ttokenNames[i]
= VOCABULARY.getLiteralName(i);¥n¥t¥t¥tif (tokenNames[i] == null)
{¥n¥t¥t¥t¥ttokenNames[i] = VOCABULARY.getSymbolicName(i);¥n¥t¥t¥t}¥n¥n¥t¥t¥tif
(tokenNames[i] == null) {¥n¥t¥t¥t¥ttokenNames[i] = ¥"<INVALID>¥";¥n¥t¥t¥t}¥n¥t¥t}"
    }
    ,    {
     "startLine":672, "startColumn":3, "startToken":2811, "endLine":674,
"endColumn":3, "endToken":2848,
     "code":"¥t¥tfor (int i = 0; i < _ATN.getNumberOfDecisions(); i++)
{¥n¥t¥t¥t_decisionToDFA[i] = new DFA(_ATN.getDecisionState(i), i);¥n¥t¥t}"
    }
   ]
  }
  ,  {
   "fileName":"ccgrep¥¥antlr¥¥cpp14¥¥CPP14Lexer.java",
   "countLine":747, "countToken":3240, "countClone":2,
   "clones":[
    {
     "startLine":149, "startColumn":3, "startToken":1826, "endLine":158,
"endColumn":3, "endToken":1896,
     "code":"¥t¥tfor (int i = 0; i < tokenNames.length; i++) {¥n¥t¥t¥ttokenNames[i]
= VOCABULARY.getLiteralName(i);¥n¥t¥t¥tif (tokenNames[i] == null)
{¥n¥t¥t¥t¥ttokenNames[i] = VOCABULARY.getSymbolicName(i);¥n¥t¥t¥t}¥n¥n¥t¥t¥tif
(tokenNames[i] == null) {¥n¥t¥t¥t¥ttokenNames[i] = ¥"<INVALID>¥";¥n¥t¥t¥t}¥n¥t¥t}"
    }
    ,    {
     "startLine":768, "startColumn":3, "startToken":3203, "endLine":770,
"endColumn":3, "endToken":3240,
     "code":"¥t¥tfor (int i = 0; i < _ATN.getNumberOfDecisions(); i++)
{¥n¥t¥t¥t_decisionToDFA[i] = new DFA(_ATN.getDecisionState(i), i);¥n¥t¥t}"
    }
   ]
  }
 ],
 "countAllFile":2,
 "countAllLine":1398,
 "countAllToken":6088,
 "countCloneFile":2,
 "countAllClone":4,
 "procTime":0.533
}
```


図 9: 「–json」のときの出力例

setter 関数宣言

```
T f(T v) { return 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 {$$} $) $?
```

while 文

```
while ($$) {$$}
```

インデックスを使用する for 文

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

インデックスを使用する while 文

```
a = 0;  while(a < $$) { $$ a++; }
```

for 文または while 文

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

3.5.2　検索スクリプト例

以下に示すような「ディレクトリ$DIR内のファイルクローンの全検索を行うスクリプト」といったスクリプトを作ることができる．

```
for FILE in `find $DIR -type f -name "*.c"` do
    ls $FILE
    ccgrep -l c -f $FILE -r $DIR -x -p l
    echo
done
```

このスクリプトでは，例えばsource1a.c，source1b.c，source2a.c，source2b.c，source3.cの5つのファイルのうち，source1a.cとsource1b.c，source2a.cとsource2b.cの2組がファイルクローンであるとすると次のように出力される．

```
dir/source1a.c
dir/source1a.c
dir/source1b.c

dir/source1b.c
dir/source1a.c
dir/source1b.c

dir/source2a.c
dir/source2a.c
dir/source2b.c

dir/source2b.c
dir/source2a.c
dir/source2b.c

dir/source3.c
dir/source3.c
```

3.6　アルゴリズム

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 クエリからパーサの構築

手順 2 ファイル単位でのコードクローンのマッチングと検索結果の出力

3.6.1 クエリからパーサの構築

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）[6]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．開発したツールでは，検索対象のプログラミング言語の文法に，以下の 9 個のトークン（言語の識別子を表す規則が `Identifier` の場合）を追加した文法による字句解析器を使用している．ただし，Java や Perl のように元々「$」を使用する言語は，追加された規則が元の文法と競合するため，そのまま追加することができない．そのため，本ツールで Java を検索対象とする場合は，識別子に「$」を使用できないという制約を設けている[7].

```
CCG_SPECIAL_ID     : '$' Identifier;
CCG_SPECIAL_SEQ    : '$$';
CCG_SPECIAL_LPAR   : '$(';
CCG_SPECIAL_RPAR   : '$)';
CCG_SPECIAL_ORLNG  : '$|';
CCG_SPECIAL_MORE0  : '$*';
CCG_SPECIAL_MORE1  : '$+';
CCG_SPECIAL_EITH   : '$?';
CCG_SPECIAL_ANY    : '$.';
```

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特殊記号により構文解析を行い，パーサを構築する．ここまでの処理の手順を図 10 に示す．

「$$」を用いた括弧の釣合いがとれたトークン列へのマッチングは，図 11 に示すパーサを使用して行う．

これらのパーサは，連接や選択など小さなパーサを木構造に組み合わせることで，大きな 1 つのパーサを構築することで作られる．マッチングは，この木構造のパーサを深さ優先で走査し，各ノードに検索対象のトークン列と部分的なマッチングを取らせることで行う．葉ノードが 1 トークンへのマッチングを行い，その親のノードが連接や選択などの複数トークンへのマッチングに統合することで，全体のマッチングが行われる．次の手順であるコードクローンのマッチングは，このパーサを使用して行う．

[6]https://www.antlr.org/

[7]Java において「$」は，主に匿名クラスなど自動生成される識別子に使用されるため問題ないと考えられる．

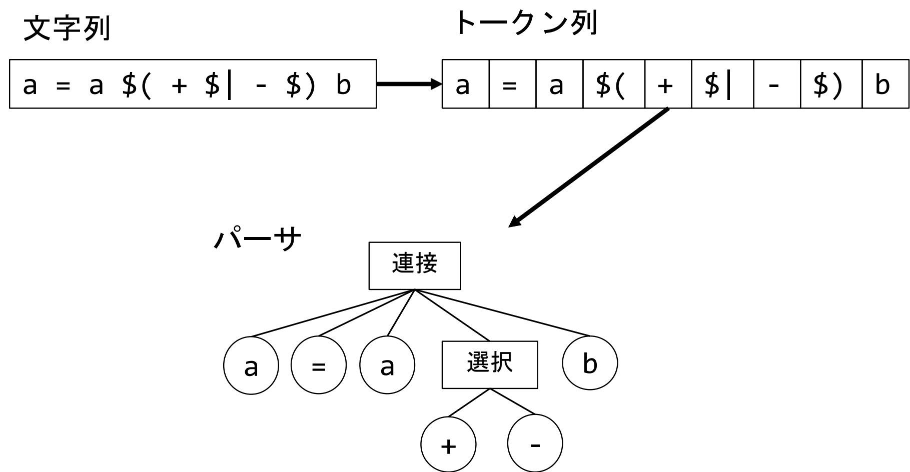

図 10: パーサの構築

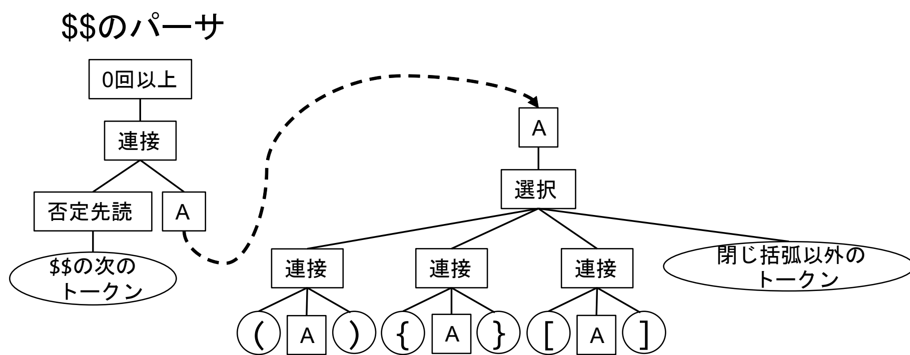

図 11: 括弧の釣合いがとれたトークン列「$$」のパーサ

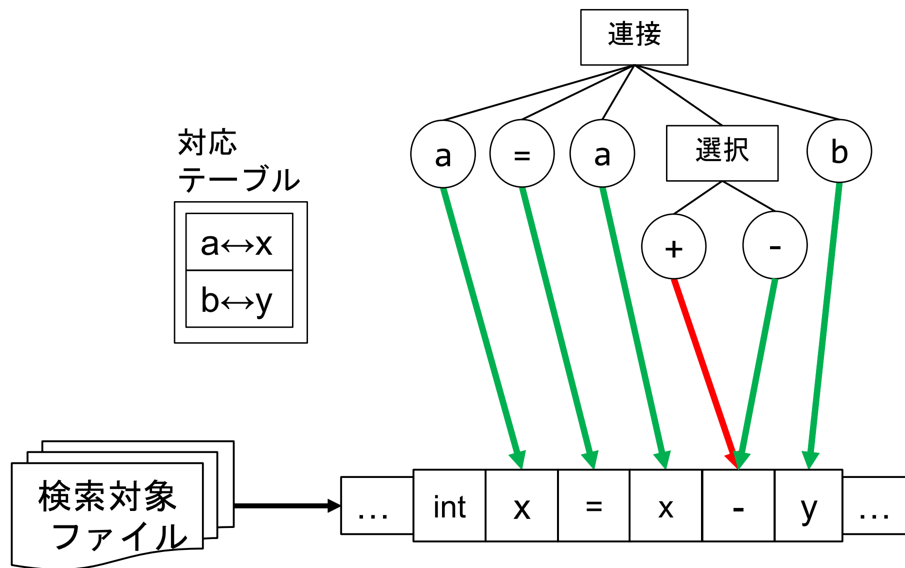

図 12: ファイル単位でのマッチングの例

3.6.2 マッチングと検索結果の出力

この手順では，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

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．ファイル単位でのマッチングの例を図 12 に示す．

詳細なマッチングの例

1 つのマッチング先頭位置におけるマッチングの例を，図 13，14 に示す検索クエリと検索対象により説明する． 対象 1 の場合は以下のようにマッチングが行われる．

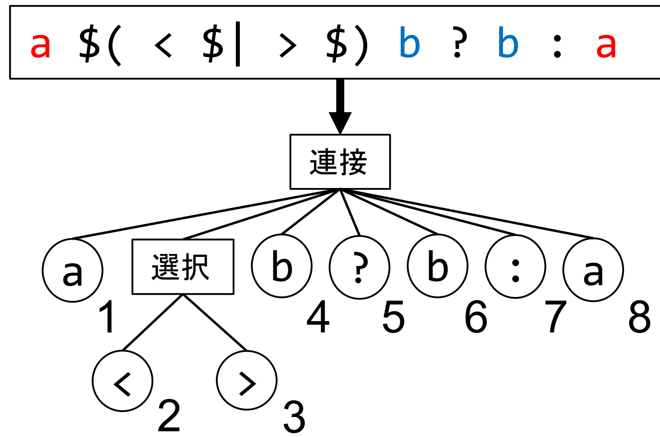

図 13: マッチング例の検索クエリとそのパーサ

	1	2	3	4	5	6	7
対象1	x	<	y	?	y	:	x
対象2	x	>	y	?	z	:	x

図 14: マッチング例の検索対象

（対象のマッチング）位置 **1**,（パーサの）ノード **1** 成功.「a↔x」の関係をテーブルに保持.

位置 **2**，ノード **2** 成功.

位置 **3**，ノード **4** 成功.「b↔y」の関係をテーブルに保持.

位置 **4**，ノード **5** 成功.

位置 **5**，ノード **6** 成功.「b↔y」の関係にマッチ.

位置 **6**，ノード **7** 成功.

位置 **7**，ノード **8** 成功.「a↔x」の関係にマッチ．マッチング全体が成功.

対象 2 の場合は以下のようにマッチングが行われる.

位置 **1**，ノード **1** 成功.「a↔x」の関係をテーブルに保持.

位置 **2**，ノード **2** 失敗．選択 2 つ目のパターンへ.

位置 **2**，ノード **3** 成功.「b↔y」の関係をテーブルに保持.

位置 **3**，ノード **4** 成功.

位置 **4**，ノード **5** 成功.

位置 **5**，ノード **6** 失敗.「b↔y」の関係にマッチしない．マッチング全体が失敗.

検索結果の出力

ファイルからコードクローンを検出し終わると，コマンドオプションで指定された出力形式でそのコードクローンを出力する.

4 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．評価項目は次の 3 つである．

- 検索コマンドの作りやすさ
- 検出できるクローン
- 検索時間

4.1 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．

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/
```

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る．また，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，特に記述は必要としない．

time 関数の呼び出しへのマッチング

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

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

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ングを取ることもできる．

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
```

grep では識別子へのマッチングごとに「`[a-zA-Z_][a-zA-Z_0-9]*`」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．

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/
```

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へのマッチを取ることができる．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．

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/
```

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．

以上に示す通り，空白やコメントの処理，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など，本ツールはコードクローンの検索において grep より簡単に検索クエリを作成することができた．

4.2 検出できるクローンの比較

他のツールで検出されたコードクローンが本ツールでもコードクローンとして検出できるかを検証する．まず，CCFinderX でクローンペアを検出する．そして，検出されたクローンペアの一部に対して，そのクローンペアの一方のコード片をクエリとして，もう一方のコード片が本ツールで検出できるか調査した．調査対象は，C 言語で書かれた 3 つのプロジェクト Git，PostgreSQL，Linux である．結果は表 1 に示す．

図 15 に示すクローンペアは，互いにコードクローンとして検出できたクローンペアの例である．この例では，識別子 `wm_controls` がリネームされて `stac_controls` となり，本ツールでも検出できた．一方，図 16 に示すクローンペアは，互いにコードクローンとして検出

表 1: 本ツールで検出できるクローンの比較

プロジェクト	Git	PostgreSQL	Linux
ファイル数	339	904	15,123
行数	127,043	317,664	6,272,939
総クローンペア数	1360	9754	154842
実験に使用したクエリ数	400	400	400
検出できたクエリ数	249	274	214
検出できなかったクエリ数	151	126	186

```
for (i = 0; i < ARRAY_SIZE(wm_controls); i++) {
    err = snd_ctl_add(ice->card, snd_ctl_new1(&wm_controls[i], ice));
    if (err < 0)
        return err;
    }
    return
```

```
for (i = 0; i < ARRAY_SIZE(stac_controls); i++) {
    err = snd_ctl_add(ice->card, snd_ctl_new1(&stac_controls[i], ice));
    if (err < 0)
        return err;
    }
    return
```

図 15: コードクローンとして検出できたクローンペア

```
bt87x_t *chip = snd_kcontrol_chip(kcontrol);
u32 old_control;
int changed;

spin_lock_irq(&chip->reg_lock);
old_control = chip->reg_control;
chip->reg_control = (chip->reg_control & ~CTL_A_SEL_MASK)
    | (value->value.enumerated.item[0] << CTL_A_SEL_SHIFT);
snd_bt87x_writel(chip, REG_GPIO_DMA_CTL, chip->reg_control);
changed = chip->reg_control
```

```
bt87x_t *chip = snd_kcontrol_chip(kcontrol);
u32 old_control;
int changed;

spin_lock_irq(&chip->reg_lock);
old_control = chip->reg_control;
chip->reg_control = (chip->reg_control & ~CTL_A_GAIN_MASK)
    | (value->value.integer.value[0] << CTL_A_GAIN_SHIFT);
snd_bt87x_writel(chip, REG_GPIO_DMA_CTL, chip->reg_control);
changed = old_control
```

図 16: コードクローンとして検出できなかったクローンペア

クエリ1
```
a < b? a: b
```
クエリ2
```
T1 f(T2 a) { return $$; }
```
クエリ3
```
f($$, $$, $$);
```
クエリ4
```
for(a = 0; a < $$; a++) { $$ }
 $|
a = 0; while(a < $$) { $$ a++; }
```

図 17: 本ツールの時間計測に使用したクエリ

クエリ5
```
'([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]'
```

図 18: grep の時間計測に使用したクエリ

できなかったクローンペアの例である．この例において，CCFinderX は，ドット演算子やアロー演算子で繋がった複数の識別子を 1 つの識別子として扱うためこのコード片がコードクローンとして検出されるが，本ツールではそのような処理を行わないため，検出できなかった．

以上のように CCFinderX は本ツールと同様にトークン単位での検出を行うが，トークン列に対する事前処理により，括弧の省略を無視したり，メンバ参照や配列リテラルなどのトークン列を結合して同一視したりするため，本ツールでは検出できないコードクローンが存在した．しかし，本ツールは正規表現や括弧の釣合いがとれたトークン列を使用することによりタイプ 3 クローンを検出することができるため，この点では本ツールに優位性がある．

4.3 検索時間

前述の 3 つの対象プロジェクト Git，PostgreSQL，Linux に対して，図 17 に示す 4 つのクエリを使用して本ツールの要した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．また，比較ツールとして grep と NCDSearch の検索時間を計測した．grep のクエリには，図 18 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，本ツールと同様に図 17 に示すクエリ 1 を使用した．ただし，検索アルゴリズムの違いのため，検出されるクローンは異なる． 実験環境は表 2 に示す．

本ツールの実験結果を表 3 に，grep と NCDSearch の実験結果を表 4 に示す．検索時間の

表 2: 実験環境

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

表 3: 本ツールの検索時間

プロジェクト		Git	PostgreSQL	Linux kernel
ファイル数		339	904	15,123
行数		127,043	317,664	6,272,939
クエリ 1	検出数	8	3	48
	時間 [s]	1.236	1.824	19.896
クエリ 2	検出数	7	27	543
	時間 [s]	1.293	1.856	19.550
クエリ 3	検出数	5,717	10,603	187,653
	時間 [s]	1.444	2.163	24.913
クエリ 4	検出数	442	621	10,754
	時間 [s]	1.880	2.493	24.907

比は，各比較ツールの検索時間に対する，クエリ 1 を使用した本ツールの検索時間の比である． 本ツールは，grep と比較すると 6〜11 倍の検索時間を要した．これは，grep がテキストベースでの処理を行っており，トークン分解などの処理が不要であるためと考えられる．一方，NCDSearch と比較すると検索時間は 0.054〜0.12 倍であり，高速に検索することができた．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 25 秒程度であり，十分実用的に使用できる．

表 4: 比較ツールの検索時間

プロジェクト		Git	PostgreSQL	Linux kernel
grep（クエリ 5）	検出数	8	2	44
	時間 [s]	0.110	0.280	3.260
	検索時間の比	11.236	6.514	6.103
NCDSearch（クエリ 1）	検出数	22	80	21,047
	時間 [s]	10.310	21.820	367.930
	検索時間の比	0.120	0.084	0.054

5 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，CCFinderX と比較した検出性能，検索に要する時間を調査した．これにより，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

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．ただし，本ツールは特殊トークンを表すために$を使用しているため，Perl など頻繁に$を使用する言語では工夫が必要になる．

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

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，研究に関する適切な御指導及び御助言を賜りました．井上教授の御指導及び御助言のおかげで本論文を完成させることができました．井上教授に心より深く感謝いたします．

島根大学総合理工学部神谷年洋教授には，研究に関する貴重な御助言を賜りました．貴重な御意見を頂いた神谷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，研究室での発表の機会において多くのご意見をいただき，研究をより洗練することができました．松下准教授に心より感謝申し上げます．

奈良先端科学技術大学院大学先端科学研究科石尾隆准教授には，研究においてたくさんの貴重な御意見を賜りました．多くの御助言を頂いた石尾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，研究室での発表の際に様々なご意見をいただき，研究をより洗練することができました．多くの御助言を頂いた春名特任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神田哲也助教には，研究室での発表において大変貴重な御意見を賜りました．多くの御助言を頂いた神田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瀬村雄一氏，横井一輝氏，溝内剛氏には，研究に関する相談に乗っていただき，また本論文の修正に御協力していただくなど研究の様々な場面で御助力いただきました．有意義な研究室生活を送りながら本論文を完成させることができたことは先輩方のおかげであると，心より深く感謝いたします．

最後に，御指導，御助言を頂き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．

付録

本研究で開発したツール ccgrep のコマンドの用例とオプションリストを以下に示す．

コマンドの用例

クエリをコマンド内で指定する場合 `ccgrep 'int a = 1;' target/ -r`

クエリをファイルで指定する場合 `ccgrep target/ -f query.java -r`

クエリを標準入力で指定する場合 `ccgrep target/ -s -r`

クエリを「**-e**」オプションで指定する場合 `ccgrep target/ -e 'int a = 1;' -r`

オプションリスト

検索クエリのオプション

-e $PATTERN$
: $PATTERN$ を検索クエリとして指定する．複数回指定でき，その場合はそれぞれのパターンを$| で繋いだものがクエリとして使用される．

-s,–stdin-query
: 検索クエリを標準入力で指定する．

検索対象のオプション

–exclude $FILE_PATTERN$
: ファイル名が $FILE_PATTERN$ にマッチするファイルを検索対象から除外する．

-f,–file $FILES$
: 検索クエリをファイルで指定する．

–include $FILE_PATTERN$
: ファイル名が $FILE_PATTERN$ にマッチするファイルのみを検索対象とする．

-r,–recursive
: ディレクトリ内のファイルを再帰的に検索する．

検索方法のオプション

-b,–blind $LEVEL$
: 識別子や，リテラルの違いをどれだけ吸収するか設定する．

none 完全一致のみ．

consistent パラメータ化されたクローン（デフォルト）．

full リネームされたクローン．

–fix *ID*

クエリ内の識別子 *ID* を固定する．

-l,–language *LANG*

検索対象の言語を *LANG* とする．

- c
- c++
- java
- python

-m,–max-count *NUM*

コードクローンを *NUM* 個検出した時点で検索を終了する．

-x,–file-match

ファイル全体がクエリにマッチする場合のみコードクローンとして検出する．

出力形式のオプション

–json

検索結果を JSON 形式で出力する．

-p,–print *OPTION*

検索結果の出力形式を指定する．以下に示すオプションを「-p nf」のようにして指定できる．

c 検出したコードクローンの総数のみを出力する．

l コードクローンが検出したファイルのファイル名のみを出力する．

h ファイル名を出力しない．

n 行番号を出力する．

f コードクローンのテキスト全体を出力する．

e ファイル名や行番号など，コードクローンのテキスト以外をコメントアウトする．

–xml

検索結果を XML 形式で出力する．

その他のオプション

-h,–help
ヘルプを表示する．

参考文献

[1] R. (Ricardo) Baeza-Yates and Berthier de Araújo Neto Ribeiro. *Modern information retrieval : the concepts and technology behind search*. Pearson, 2nd ed edition, 2011.

[2] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[3] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. pp. 604–613, 1998.

[4] Takashi Ishio, Naoto Maeda, Kensuke Shibuya, and Katsuro Inoue. Cloned buggy code detection in practice using normalized compression distance. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pp. 591–594. IEEE Computer Society, 2018.

[5] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *IN ICSE*, pp. 96–105, 2007.

[6] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 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, July 2002.

[7] Harpreet Kaur, Rupinder Kaur, and Talwandi Sabo Bathinda. A review: Clone detection in web application using clone metrics.

[8] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *2012 16th European Conference on Software Maintenance and Reengineering(CSMR)*, Vol. 00, pp. 309–318, mar 2012.

[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, 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.

[11] 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, oct 2011.

[12] 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.