修士学位論文

題目

多様なプログラミング言語に対応可能な コードクローン検出ツール **CCFinderSW**

指導教員
井上 克郎 教授

報告者
瀬村 雄一

平成 31 年 2 月 6 日

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 ソフトウェア工学講座

平成 30 年度 修士学位論文

多様なプログラミング言語に対応可能なコードクローン検出ツール CCFinderSW

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し，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．一般的に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ため，開発者は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．したがって，この作業を支援するために現在まで多くのコードクローン検出ツールが提案されている．近年では，実務に使用されるプログラミング言語は多様化し，ある 1 つのプログラミング言語においても，バージョンによってその文法には差異がある．そのため，コードクローン検出ツールも言語やバージョンの多様化に柔軟に対応することが求められる．しかし，既存のコードクローン検出ツールは，限られた数のプログラミング言語にしか対応しておらず，追加の言語を扱うためのシンプルな仕組みを持たない．

字句単位のコードクローン検出ツールである CCFinderX は，C，C++，Java，COBOL，Fortran で記述されたソースコードを対象を対象に，そのプログラミング言語の文法に沿って字句解析を行うことにより，実用的で意味のあるコードクローンを検出できる．字句解析はプログラミング言語の文法に依存するため，CCFinderX の対応言語を増やすには，言語ごとに個別の字句解析を実装しなければならない．これは手間のかかる作業であるが，新たな言語に容易に対応するための仕組みを用いることで，ツール開発者と利用者の手間を減らすことができる．また，既存の多言語対応コードクローン検出ツールでは，ツールの利用者がコードクローンを検出したい言語のコメント文法と予約語の情報を設定ファイルに記述して，ツールの実行時に入力として与えることで，実用的で意味のあるコードクローン検出に必要な正規化処理を行うことが出来る．この仕組みによって，字句解析部を実装する手間が省かれ，新たな言語への対応が容易にできる．しかしこの際にも，ツールの利用者は，コメント文法を専用の設定ファイルを記述し，予約語のリストを用意する手間がかかる．

本研究ではこの問題を解決するために，既存のコードクローン検出ツールにおける字句解析に必要な文法情報を，構文解析器生成系の構文定義記述から自動的に抽出するモジュールを開発した．そしてこのモジュールを用いて，構文定義記述の 1 つである ANTLR の構文

定義記述を入力することで，プログラミング言語のコメント文法と予約語と文字列リテラルの情報を抽出し，字句解析に適用することで，対象言語の文法に沿ったコードクローン検出が可能な CCFinderSW を開発した．

開発したモジュールの評価として，42 言語の構文定義記述からコメント文法と予約語と文字列リテラルの情報を抽出を行い，83 %の言語ですべての情報が抽出可能であることを示した．また，C++で記述されたソースコードに対するコードクローン検出において CCFinderX と出力を比較し，CCFinderSW がほぼ同等の検出能力を持つことを示した．そして，Verilog HDL で記述されたソースコードに対するコードクローン検出を行い，Precision と Recall に関して，既存手法との比較を行った．

主な用語

コードクローン
字句解析
ANTLR

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し，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．そのため，開発者はあるコード片にバグが見つかった場合，そのコード片の全てのコードクローンに対して同一の修正を行うか検討する必要がある [2]．したがって，開発者は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，開発者のコードクローン管理作業を支援するために現在まで多くのコードクローン検出ツールが提案されている [3]．近年では，実務に使用されるプログラミング言語は多様化し，ある 1 つのプログラミング言語においても，バージョンによってその文法には差異がある．そのため，コードクローン検出ツールも言語やバージョンの多様化に柔軟に対応することが求められる [4]．しかし，既存のコードクローン検出ツールは，限られた数のプログラミング言語にしか対応しておらず，追加の言語を扱うためのシンプルな仕組みを持たない．

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 の言語で記述されたソースコードからコードクローンを検出することが可能である [5]．CCFinder では字句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 [6]，これによりソースコードのフォーマット，コメントの有無，変数名や関数名の違いを無視した実用的で意味のあるコードクローンを検出することができる．字句解析はプログラミング言語の文法に依存するため，CCFinder の対応言語を増やすには，言語ごとに個別の字句解析を実装しなければならない．CCFinderX は，CCFinder のバージョンアップとして開発され，字句解析部はツール利用者によ変更可能である[1]．つまりツール利用者は，コードクローンの検出対象のソースコードのプログラミング言語に対応する字句解析部を用意することで，その言語のコードクローン検出が可能になる．しかし，字句解析部を用意するためにはその言語の文法を理解する必要があるため，字句解析部の実装は手間のかかる作業である．このような場合に，新たな言語に容易に対応するための仕組みを利用することで，ツール開発者と利用者の手間を減らすことができる [7]．また，多言語の字句単位のコードクローン検出を行うための，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がある [4]．ツールの利用者がコードクローンを検出したい言語のコメント文法と予約語の情報を

[1] http://www.ccfinder.net/

設定ファイルに記述して，ツールの実行時に入力として与えることで，実用的で意味のあるコードクローン検出に必要な正規化処理を行うことが出来る．この字句解析機構によって，ツールの開発者と利用者の字句解析部を実装する手間が省かれ，新たな言語への対応が容易になっている．しかし，利用者はコメント文法を入力するための専用文法を持つ設定ファイルを記述し，予約語のリストを用意する必要がある．

構文解析器生成系は，字句解析器や構文解析器を自動的に生成するプログラムであり，パーサジェネレータとも呼ばれる．構文解析器生成系は，構文定義記述を用いることで字句解析器・構文解析器を生成することができるため，字句解析器の実装の手間を省くことができる．代表的な構文解析器生成系に，Yacc [8] や JavaCC[2]，ANTLR[3]などがある．ANTLR で利用できる構文定義記述を集めたリポジトリが GitHub 上に存在し[4]，150 以上の構文定義記述が用意され，2019 年現在もコミットが続けられている．

本研究では，多様なプログラミング言語に容易に対応することができるコードクローン検出ツールの開発を目的として，既存ツールにおける字句解析に必要な文法情報を，構文解析器生成系の構文定義記述から自動的に抽出するモジュールを開発した．そしてこのモジュールを用いて，ANTLR の構文定義記述を入力することで，プログラミング言語のコメント文法と予約語と文字列リテラルの情報を抽出し，字句解析に適用することで，多様な言語の文法に沿ったコードクローン検出が可能な CCFinderSW を開発した．CCFinderSW の利用者は，構文定義記述が集められたリポジトリから対象言語の構文定義記述を取得し，ツールの実行時に入力としてそのまま与えることで，新たな言語のコードクローン検出を行うことが出来るため，既存ツールで必要だった作業が不要になる．

また，CCFinderSW を評価する 4 つの実験を行った．1 つ目の実験では，ANTLR で利用できる構文定義記述を集めたリポジトリである grammar-v4 上の，42 のプログラミング言語を表す構文定義記述ファイルを対象に，本研究で開発した構文定義記述解析モジュールがどの程度のコメント文法と予約語一覧と文字列リテラルを抽出可能かを確認した．2 つ目の実験では，ANTLR の構文定義記述から抽出したコメント文法と予約語一覧に相当する情報を，9 つの言語のコードクローン検出へ適用可能であることを確認した．3 つ目の実験では，C++のソースコードに対してコードクローン検出を行い，CCFinderX と CCFinderSW の検出結果を比較し，その差異を分析した．最後に，4 つめの実験では，Verilog HDL のソースコードに対してコードクローン検出を行い，CCFinderSW が検出したコードクローンにおける Precision と，正解コードクローンの検出に関する Recall を測定することで，既存手法と比較を行った．

[2]https://javacc.org/
[3]http://www.antlr.org/
[4]https://github.com/antlr/grammars-v4

以降，2 章では本研究の背景として，コードクローンとその検出技術について，またコードクローン検出ツールである CCFinderX と CCFinderSW の説明を行う．3 章では本研究で対象とした構文解析器生成系の 1 つである ANTLR の利用方法について説明し，次に構文定義記述から文法情報を抽出する手法と，抽出した文法情報を字句単位のコードクローン検出における字句解析へ適用する手法について説明を行う．4 章では本研究で行った 5 つの評価実験について記述する．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，コードクローンとその分類，コードクローンの検出技術についての記述し，字句単位のコードクローン検出ツールである CCFinderX と，既存の多言語対応コードクローン検出ツールについての説明を行う．

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．またクローンペアの 2 つのコード片に対し，それぞれを包含するいかなる字句列も等価でないとき，極大クローンペアと呼ぶ．

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，開発者はバグの修正を行うために修正が加えられたコード片に対してすべてのコードクローンを調査し，同一修正を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の位置などの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提案されている [3]．近年では，実務に使用されるプログラミング言語は多様化し，ある 1 つのプログラミング言語においても，バージョンによってその文法には差異がある．そのため，コードクローン検出ツールも言語やバージョンの多様化に柔軟に対応することが求められる [4]．しかし，既存のコードクローン検出ツールは，限られた数のプログラミング言語にしか対応しておらず，追加の言語を扱うためのシンプルな仕組みを持たない．

2.1　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．

2.2 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．

字句単位の検出

字句単位の検出手法では，検出の前処理として，ソースコードは字句の列に変換される．しきい値以上連続して一致している字句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]．

字句単位の検出ツールとして，CCFinder がある．字句解析によって字句を分割した後に特定の字句の変換を行っている．これは変数名や関数名などを全て同一の字句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．

ブロック単位の検出

ブロック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．他の検出技術では検出できない，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．

ブロック単位の検出ツールとして，横井らが開発したブロッククローン検出ツールである CCVolti がある [11]．ブロッククローン検出ツールでは意味的に処理が類似したブロック単位のコードクローンを検出するので，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て，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で，各関数を特徴ベクトルに変換しその類似度を計算し，ブロック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]．Deckard 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．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．

他の検出技術として，行単位，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がされている．

2.3 字句に基づくコードクローン検出ツール: CCFinderX

CCFinderX は，字句単位のコードクローンを検出するツールである．対応言語は，Java，C，C++，COBOL，Visual Basic，C# である[5]．CCFinderX 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字句を同一字句に置き換えることで，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でも実用時間で解析可能である．
- 言語に依存する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，字句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．

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，字句解析部の利用者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利用者が用意することで，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，字句解析部のコーディングは手間のかかる作業である．

2.4 多言語対応コードクローン検出ツール

多言語対応コードクローン検出ツール（以降，既存ツール）は，CCFinderX と同様に字句単位のコードクローンを検出するツールである [4]．既存ツールでは新たな言語に対応するために，コメントと文字列の文法を記述した設定ファイル（以後，コメント設定ファイ

[5] http://www.ccfinder.net/

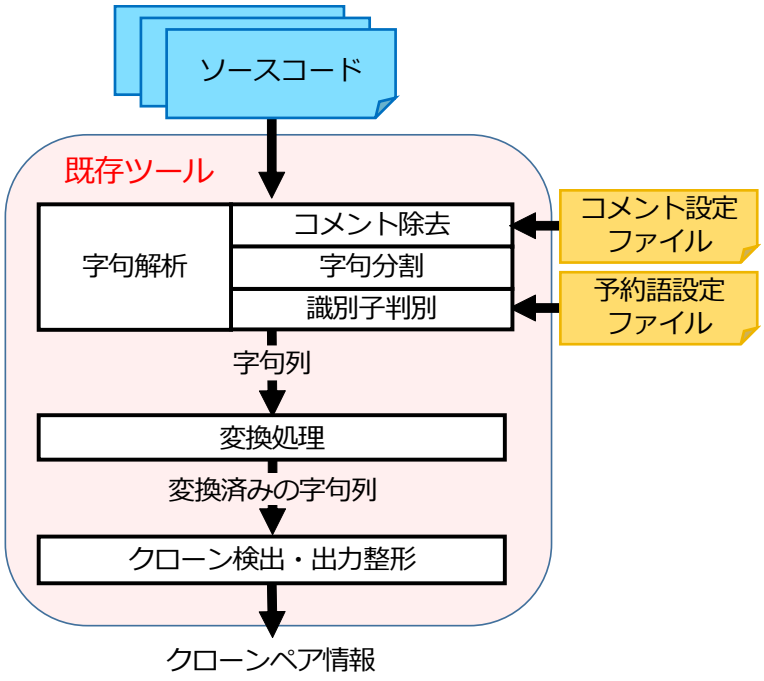

図 1: 既存ツールの処理概要

ル）と予約語の一覧を記述した設定ファイル（以後，予約語設定ファイル）を読み込み，その情報を用いて字句解析を行う仕組みを持つ．これによって，字句解析部の実装の手間が省かれ，新たな言語への対応が容易になっている．

ここで，既存ツールの処理概要について記述する．図 1 は既存ツールの処理概要を図示したものである．既存ツールがソースコードを入力として受け取ってから，クローンペア情報を出力するまでの処理は，4 つの Step に分けられる．以後，各 Step の詳細を説明する．

Step 1:字句解析

入力されたソースコードをプログラミング言語の文法に沿って字句列に変換する．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．字句解析はコメント除去，字句分割，識別子判別といった処理が行われている．入力された 2 つの文法ファイルから検出対象のプログラミング言語の文法情報を受け取る．コメントと文字列の情報はコメント除去で使用され，予約語は識別子判別で使用される．

Step 2:変換処理

分割された字句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同一の字句に置き換える処理である．

Step 3:クローン検出

変換された字句列を比較しコードクローンを検出する．この比較にはN-gram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．

以降，Step 1の詳細として2.4.1節でコメント除去について，2.4.2節で字句分割についての詳細を説明する．2.4.3節では，Step 1の識別子判別とStep 2の変換処理について記述する．2.4.4節では従来の既存ツールの問題点について説明する．

2.4.1 コメント除去

既存ツールの字句解析部では，まず入力されたソースコードからコメントを除去する．これはコードクローンにはコメントや空白は含まれないという，コードクローンの定義に基づいている．既存ツールのコメント除去部では，利用者によるコメント文法の入力を必要とする．除去可能なコメントのルールとして，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，文字列リテラルの5種類のルールが用意されている．利用者は用意されたルールに基づいて，コメント文法を記述したファイルを作成し，実行時に入力として既存ツールに与える．

2.4.2 字句分割

コメント除去を行った後は字句分割を行う．字句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字句とする．
2. 空白，タブ文字と改行の前後で字句を分割する．
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字句とする．
4. それ以外の連続した英数字列は1字句とする．

2.4.3 識別子判別・変換処理

識別子判別では字句分割で英数字列として分割された字句が，識別子か予約語かを判定するものである．これはStep 2の変換処理のために行われる．予約語は変数名や関数名に使

用できない文字列のことを指し，プログラミング言語によって異なっている．利用者は予約語一覧を記述したファイルを作成し，実行時に入力として既存ツールに与える．

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，全て同一の字句に置き換える処理である．

2.4.4 既存ツールの問題点

既存ツールでは，新たな言語に対応するためにはコメントファイルと予約語ファイルを利用者が新しく作成しなければならない．コメントファイルには独自の文法が存在するため，利用者はそれを理解してコメントファイルを作成しなければならない．また，全ての予約語を記述したファイルを用意することは手間がかかる．そのため，これらの利用者による手間を軽減させるための仕組みが必要である．

3　提案ツール

本研究では多様なプログラミング言語に対応したコードクローン検出ツールを開発することを目的として，字句単位のコードクローン検出における字句解析に必要な文法情報を，構文解析器生成系の 1 つである ANTLR の構文定義記述から自動的に抽出するモジュールを開発した．そしてそのモジュールを用いて，ANTLR の構文定義記述を入力として与えることで，多様な言語の文法に沿ったコードクローン検出が可能な CCFinderSW を開発した．CCFinderSW の利用者は，構文定義記述が集められたリポジトリなどから対象言語の構文定義記述を取得し，ツールの実行時に入力としてそのまま与えることでコードクローン検出を行うことが出来るため，既存ツールで必要だった文法ファイルの記述が不要になる．

以降，3.1 節では ANTLR の構文定義記述について，および本研究で利用した理由について説明する．3.2 節では新たに開発した構文定義記述解析モジュールの実装について説明し，その後，3.3 節では抽出した文法に相当する正規表現を用いた字句解析について述べる．

3.1　構文解析器生成系 ANTLR の構文定義記述の利用

本節では，本研究に利用する ANTLR の構文定義記述の文法と，調査を行った構文定義記述ファイルに多く採用されていた表現方法について説明する．

構文解析器生成系は，字句解析器や構文解析器を自動的に生成するプログラムであり，パーサジェネレータとも呼ばれる．構文解析器生成系は，構文定義記述を用いることで字句解析器・構文解析器を生成することができるため，字句解析器の実装の手間を省くことができる．代表的な構文解析器生成系に，Yacc や JavaCC，ANTLR などがある．

ANTLR は，構文木の構築・探索が可能な構文解析器を生成する．Twitter[6]での検索クエリの解析には ANTLR が採用されるなど広く使用されているため，本研究の対象として選択した [14]．ANTLR が構文解析器を生成するターゲットとなる言語は C++，Java，Python，Go などがある．

ここで，Listing 1 に ANTLR の構文定義記述の例を示し，用いられる文法について説明する．まず，以下の構文定義記述の 1 行目では，**grammar Prog** と書くことで構文定義記述が表す文法の名前が **Prog** であることを宣言している．2 行目以降は文法を構成するルールについて記述している．先に示した四則演算式を表す記述の中では，字句解析ルールとして ‘INT’ と ‘WS’ が存在する． ‘INT’ は数字列を表していて，‘WS’ は空白とタブ文字と改行を表している．

ルールの記述法は，**ルール名 : ルールブロック ;** となり，このようなルールを複数組み合わせることで 1 つの文法を表す．ANTLR の文法を定義するルールには，字句解析ルール

[6]https://twitter.com/

```
1 grammar Prog;
2 prog: expr;
3 expr: term ( ( '+' \textbar \ '-' ) term )*;
4 term: factor ( ( '*' \textbar \ '/'  ) factor )*;
5 factor: INT \textbar \ '(' expr ')';
6 INT: [0-9]+;
7 WS: [\textbackslash t\textbackslash r\textbackslash n]+ -> \ skip;
```

Listing 1: 四則演算式を表す構文定義記述

と構文解析ルールの 2 つが存在する．字句解析ルールの名前は大文字から始まり，構文解析ルールの名前は小文字から始まる．本研究で構文解析ルールについてはほとんど使用しないため，本稿では字句解析ルールのみ説明する．

ANTLR で用いられる字句解析ルールでの字句表現について，ANTLR の開発者によるドキュメント [15] が存在する．その中から本稿に必要なものを抜粋し，表 1 に示した．

表 1: ANTLR で使用される主要な字句

字句	説明
‘リテラル’	文字，または文字列にマッチする．
[文字集合]	指定された文字のどれか 1 つにマッチする．a-z のように書くことで範囲を指定することも可能．正規表現で使用されるものと同様.
.	任意の一文字にマッチする
~x	x で記述されている集合にマッチしない任意の一文字にマッチする． x は 1 文字のリテラルや文字集合が指定される．本稿では NOT 演算子と呼ぶ
x*	x の 0 回以上の繰り返しにマッチする．
x?	x の 0 回，または 1 回の出現にマッチする．
x*?	x の 0 回以上の最短の繰り返しにマッチする．
x|y	x または y にマッチする．

次に，ANTLR で使用されるコマンドという機能について説明する．ANTLR は字句解析ルールを用いてソースコード中で出現する字句を定義するが，それぞれの字句にコマンドを指定して特定の操作を施すこともできる．

コマンドは **ルール名** : **ルールブロック** − > **コマンド**; のように記述する． 本研究で扱う，コマンドの種類は 2 つある．1 つ目は skip である．skip で指定された字句は，字句解析によっ

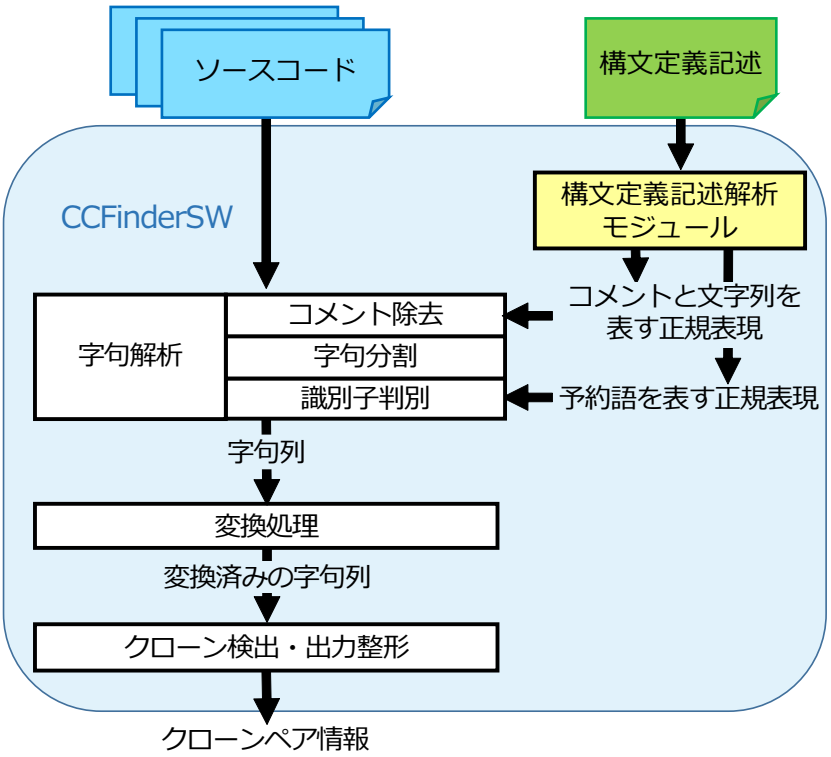

図 2: CCFinderSW の処理概要

て読み飛ばされる．2 つ目は channel コマンドである．channel(x) のように記述し，x にはあらかじめ定義された channel 名が入る．channel コマンドは字句をある channel に渡す．例えばコマンドを channel(HIDDEN) と指定すると，HIDDEN に渡す．特に channel(HIDDEN) は skip と似た働きをしていて，このコマンドを指定された字句は ANTLR の Parser によって無視される [16].

3.2 構文定義記述解析モジュールの開発

図 2 は新たに開発を行った構文定義記述解析モジュールを用いて開発した CCFinderSW の処理概要を表したものである．構文定義記述解析モジュールは，ANTLR の構文定義記述からコメントと予約語を情報を抽出するものである．また，CCFinderSW の字句解析部は，コメントを表す正規表現を用いてコメント除去を行う．これによって，字句分割以降の処理を既存ツールと同じ流れでソースコードを扱うことができる．CCFinderSW に，ソースコードと構文定義記述のみを入力として与えることで，クローンペア情報が出力可能となる．

本研究で開発した構文定義記述解析モジュールは，まず ANTLR の構文定義記述を構文解析して構文木を生成している．この構文解析には，Java で動作する構文解析器を ANTLR

で生成して組み込んだ．

以降，3.2.1 節ではコメント記述ルールの正規表現への変換について説明し，3.2.2 節ではコメント記述ルールの正規表現への変換について説明し，3.2.3 節では予約語一覧の正規表現への変換について説明する．

3.2.1 コメント文法の正規表現への変換

本節では ANTLR の構文定義記述から，コメントを表す正規表現の抽出方法について述べる．文法情報を正規表現で抽出した理由としては，一般的に正規表現は文字列の置換処理などに用いられることが多く，提案ツールの実装言語である Java においても，正規表現を用いた文字列処理に関する標準ライブラリが存在するためである．また，ANTLR の構文定義記述は正規表現と近い表現が用いられており，変換が容易であるためである．さらに，コメント除去に正規表現を用いることで，既存ツールで採用されている分類以外のコメント文法にも幅広く対応出来る．

コメント文法の正規表現への変換は以下の 4 つの Step で行われる．

Step A 全てのルールの中からコメントに関すると考えられるルールを選びだす．

Step B それぞれのルールの中で，別のルールを参照している部分を再帰的に適用する．

Step C 詳細化されたルールを Java で使用される正規表現に変換する．

Step D 生成された正規表現を全て結合して 1 つの表現にする．

各 Step の詳細は以下のとおりである．

Step A まず，全てのルールの中からコメントに関すると考えられるルールを選び出す．これには判断基準を 4 つ設け，そのうちの少なくとも 1 つに当てはまることでコメントに関するルールとして識別する．その 4 つの判断基準を以下に示し，それぞれに当てはまるルールを Listing 2 に例示する．例示された 4 つのルールは全て C 言語の複数行コメントに相当する表現である．

1. ルール名に Comment や COMMENT などの文字が含まれている．
2. コマンドの skip が呼ばれている．
3. コマンドの channel(HIDDEN) が呼ばれている．
4. コマンドの channel が呼ばれていて，channel 名に Comment や COMMENT などの文字が含まれている．

```
1 Comment:'/*'.*?'*/';
2 Block1: '/*'.*?'*/' -> skip;
3 Block2: '/*'.*?'*/' -> channel(HIDDEN);
4 Block3: '/*'.*?'*/' -> channel(BComment);
```

Listing 2: コメントの定義例

Step B 選び出されたコメントに関するルールを再帰的に適用化する．3.1 節で示した四則演算式を表す構文定義記述の例では，term という名前のルールの中で factor というルールが埋め込まれている．このようにルールの中で他のルールが参照されている場合は，Step C で正規表現に変換するために参照先のルールの内容を再帰的に適用する．

Step C 詳細化されたルールを Java で使用可能な正規表現に変換する．これは本研究で用いるモジュールが Java で開発されていることで，ANTLR で使用される表現とは部分的に異なっているためである．Java で使用可能な正規表現と ANTLR の構文定義で用いられる表現の違いで，変換が必要なものは 3 つある．1 つ目はシングルクォーテーションである．ANTLR の構文定義記述では，対象言語に出現する実際のリテラルをシングルクォーテーションで囲んで記述する．正規表現ではこのシングルクォーテーションは不要であるため除去する．2 つ目は NOT 演算子である．ANTLR の構文定義記述では，ある特定の文字集合にマッチしない集合を表すために先頭に，NOT 演算子の役割をする ‘~’ をつける． この NOT 演算子に直接的に相当するものは正規表現には存在しないため，正規表現の否定的先読みを用いて同等の表現に変換する．3 つ目は任意の 1 文字を表す ‘.’ である．ANTLR の構文定義記述ではこの ‘.’ は任意の 1 文字を表すのに対し，Java で使用される正規表現では改行以外の任意の 1 文字を表し，定義がそれぞれで異なっている．この定義の差異を埋めるため，構文定義記述で用いられる ‘.’ を正規表現での同等の表現である ‘[\s\S]’ に変換することで対応している．

Step D 生成された正規表現を全て結合して 1 つの表現にする．これは Step C で生成された正規表現の全ての論理和をとるもので，全ての正規表現の間に ‘|’ を挟んで結合することで行われる．この Step D を終えた時点で，コメント記述ルールの正規表現の変換が終了する．

図 3 は，以上のコメント記述ルールの正規表現への変換例である．この図ではまず，Step A で *BComment* と *LComment* というコメントを表すと考えられるルールを選択する．次に Step B では選択されたルールの詳細化を行う．*BComment* では $CSTART$ と $CEND$ という他のルールへの参照があるため，参照先のルールを代入して他のルール名が含まれない表現にする．Step C では正規表現での同等の表現に変換し，最後に Step D で生成された全ての正規表現を結合する．

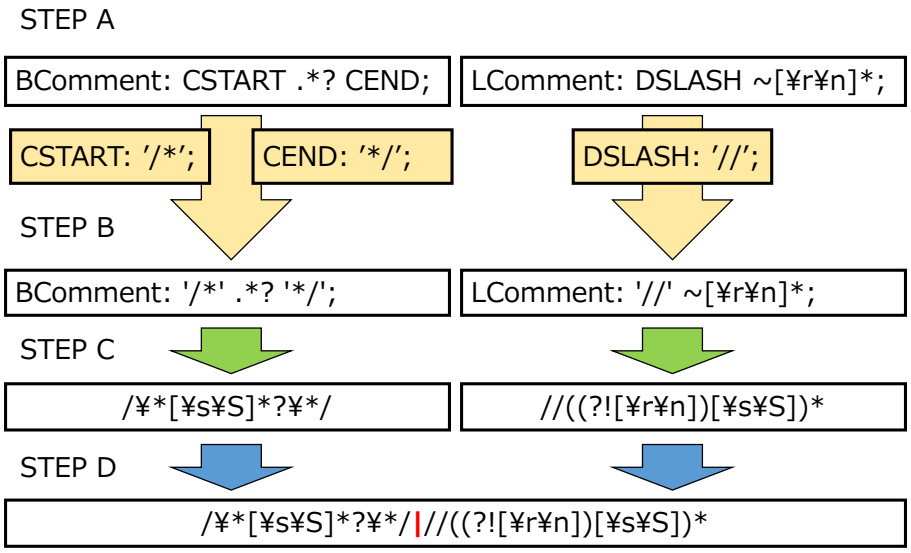

図 3: コメント記述ルールの正規表現への変換例

3.2.2 文字列リテラル文法の正規表現への変換

本節では ANTLR の構文定義記述から文字列リテラル文法を抽出し，正規表現へ変換する方法について述べる．

文字列リテラルはリテラルの 1 種であり，ソースコード中の文字列オブジェクトを表すものである．例えば，Java ではダブルクォーテーションに囲まれた文字列が文字列リテラルとして認識される．CCFinderSW に設定可能なコメント文法の 1 つに文字列リテラルを定義しているため，今回の拡張においても同様のコメント除去を実現するためには，CCFinderSW へ文字列リテラルを入力として与える必要がある．したがって，構文定義記述解析モジュールは構文定義から文字列リテラルを抽出し，コメントと同様に正規表現に変換する．正規表現は CCFinderSW のコメント除去部で使用される．

本ツールを開発する際に，ANTLR の構文定義記述内での文字列リテラルの記述法を調査した．Listing 3 に示すのは，調査対象の中に多く存在した構文定義である．

```
1 StringLiteral: QUOTE StringCharacters? QUOTE;
2 STRING : 'string';
```

Listing 3: 文字列リテラルの定義例

1 行目の構文定義は文字列リテラルの定義である．2 行目は定義名に “STRING” が含まれているが， ソースコード中に出現する “string” という予約語の定義である．このような調査をもとに，文字列リテラルに関する構文定義を抽出するための基準を設けた．その基準

は,「定義名に “STRING” または “string” などという文字が含まれるもののうち，予約語の定義ではないルール」である

そして基準に当てはまる構文定義を抽出し，正規表現へ変換するための実装を行った．変換方法はコメントの正規表現に用いられたものと同じであるため，説明を省略する．

3.2.3 予約語一覧の正規表現への変換

本節では ANTLR の構文定義記述から予約語一覧を抽出し，正規表現へ変換する方法について述べる．

本ツールを開発する際に，ANTLR の構文定義記述内での予約語の記述法を調査した結果，大きく分けて 2 種類の記述法があった．以下の Listing 4 に，*while* という予約語の定義に対する 2 種類の記述法を例示する．

```
1 WHILE:'while';\\
2 WHILE:[wW][hH][iI][lL][eE];
```

Listing 4: 予約語の定義例

1 行目の記述法では，$WHILE$ というルール名に *while* という単語が紐付けられている．この記述法は最も広く使われているものである．そして 2 つ目の記述法では，正規表現で使用されるような文字クラスを用いた書き方がされている．これは *while* に含まれる 5 文字のそれぞれに大文字と小文字のどちらでもマッチングすることを許す記述法である．1 つ目の記述法の右辺も正規表現としてみなせるため，予約語一覧もコメント文法と同様に正規表現で生成することにした．予約語一覧の抽出は以下の 5 つの Step で行われる．これらの多くの Step は 2.4.1 で説明したコメント抽出の Step と同様である．

Step α 全てのルールで，出現しているリテラルのうち英字列に一致するものを抽出する．

Step β 全てのルールで，別のルールを参照している部分を再帰的に詳細化する．

Step γ Step β で詳細化されたルールを Java で使用可能な正規表現に変換する．

Step δ Step γ で変換された正規表現で，英字列を表しているものを選出する．

Step ϵ Step α で抽出されたリテラルと Step δ で選出された正規表現を全て結合して 1 つの表現にする．

Step α は，全てのルールブロックに出現するリテラルのうち英字列に一致するものを全て抽出し予約語であるとみなす．Step β は 2.4.1 節で説明したコメント抽出の Step A と同様であり，Step γ はコメント抽出の Step C と同様である．Step δ は Step γ で生成された

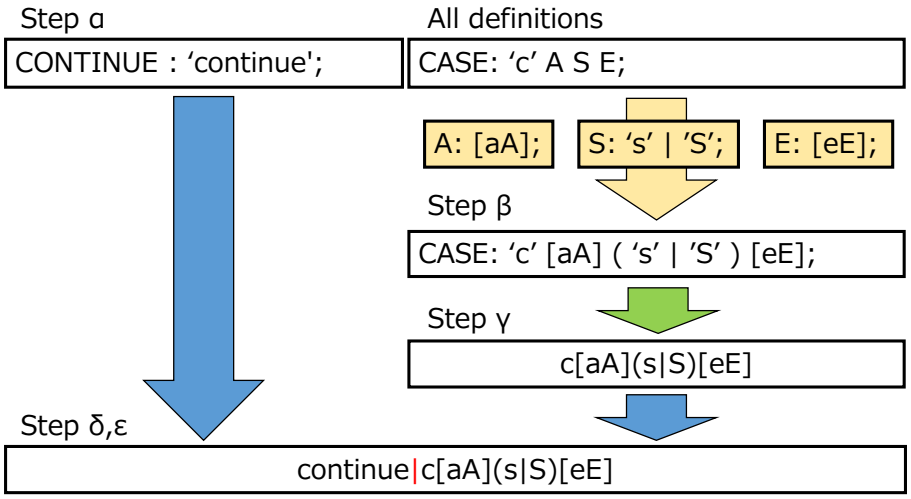

図 4: 予約語一覧の正規表現への変換例

正規表現が，英字列を表しているものを選び出す．Step ϵ はコメント抽出の Step D と同様に '|' を用いた結合を行う．

Step α と Step δ では，予約語は英字列であるという前提に基づいている．本研究で行った構文定義記述への調査では，予約語に含まれる文字として英字列以外に '_' や '@' などの記号も含まれることがあったため，モジュールに抽出したい予約語に含まれる文字を指定する機能を実装した．

図 4 は，以上の予約語一覧の正規表現への変換例である．まず Step α で $CONTINUE$ という英字列を表すルールを選択する．次に Step β では全てのルールの詳細化を行う．例えば $CASE$ というルールにおいては，A と S と E という他のルールへの参照があるため，参照先のルールを代入して他のルール名が含まれない表現にする．Step γ では正規表現での同等の表現に変換する．次に Step δ は Step γ で生成された正規表現が，英字列を表しているものを選び出す．最後に Step ϵ では全ての予約語を '|' を用いて結合する．

3.3 正規表現を用いたコメント除去手法

本節では，構文定義記述解析モジュールが出力したプログラミング言語の文法を表す正規表現を用いて，CCFinderSW の字句解析部が行う処理について記述する．

一般的に，正規表現は文字列の検索や置換などで使用される．Friedl は著書の中で，C 言語のソースコードに対するコメント除去の方法について記述している [17]．この本の中では，C 言語のコメント除去手法が，Perl で用いられる正規表現の文法を用いて説明されていて，本研究ではそこから基本的な考え方を参考にした．本ツールの開発は Java で行っているため，Java で実装可能な手法を記述する．本ツールの実装では，Java の標準的な正規表現ライ

ブラリである java.util.regex パッケージの中の Pattern クラスと Matcher クラスを用いた，また，文字列の保持に用いられる String クラスの replace メソッドと replaceAll メソッドを用いることで，正規表現を用いた文字列の置換を容易に行うことが出来る．

以降，具体的な除去手法について，C 言語のコメントと文字列を表す正規表現を用いて，C 言語のソースコード中のコメントを除去する手法について説明する．

まず，C 言語におけるコメントに相当する正規表現は，$/\backslash*[\backslash s\backslash S]*?\backslash*/|//((?![\backslash r\backslash n])[\backslash s\backslash S])*$ であり，文字列に相当する正規表現は $((?!")[\backslash s\backslash S])*?$ であると定義する．これらの正規表現を用いて，ソースコード中からコメントにマッチする部分を正しく認識し，除去することが目的となる．以下の説明では便宜上，コメントの正規表現を COM と記述し，文字列の正規表現を STR と記述する．

ここで文字列リテラルの定義が必要な理由について，Listing 5 を用いて説明する．これは，C 言語のソースコードの一部である．このソースコードに対し，コメントの正規表現 COM のみを用いてマッチを行った場合，コメントを表す部分に正しくマッチしない．マッチングしたい文字列は緑字の部分であるが，実際にマッチした部分の開始地点はダブルクオーテーションに囲まれた $/*$ になってしまう．この対策として，文字列の正規表現が必要となる．

```
1 char *CommentStart = "/*"; /* Comment1 */
2 char *CommentEnd = "*/"; /* Comment2 */
```

Listing 5: コメントが正確に除去できないソースコード

本ツールの開発では，この問題の対策としてコメントの正規表現と文字列の正規表現を $|$ を用いて結合して，ソースコードに対して検索を行う方法を用いた．つまり，検索に用いる正規表現は $COM|STR$ となる．この正規表現を用いてマッチを行うと，コメントを表す部分か文字列を表す部分のどちらかにマッチするため，文字列の中でコメントの開始地点と認識されて，予期せずマッチすることがなくなる．そして，マッチした部分をもう一度 COM を用いてマッチを行い，マッチするものはコメントを表す部分なので除去対象とし，マッチしないものは文字列を表す部分であるためそのまま残すと判断することで，期待した通りに認識することが出来る．

ここでコメントとして認識された部分を，Java の String クラスの replace などを用いてそのまま空文字列に置換してしまうと，字句分割を行う際に問題が生じてしまう．CCFinderSW の処理では，コメント除去の次に字句分割が行われる．字句分割では，コメント除去が行われたソースコードしか読み込まないため，もとのソースコードでどの位置にコメントがあったか，などの情報は失われている．もし複数行にまたがるコメントを，そのまま空文字列に置換した場合，本来存在したインデントの情報や改行文字が失われてしまうことがある．字

句分割で得られた字句は，その字句の出現行や出現列などの情報とともに保持する必要があるため，本来存在したインデントや改行文字が失われた状態では，字句情報が誤って保持されてしまう．このため，後のコードクローン検出・出力整形でコードクローンの位置を誤って出力してしまう可能性がある．このため，コメントの中のインデントや改行を表す部分をそのまま残し，それ以外の部分は空白文字に変換する，という処理が必要となる．

以上のことをふまえて，本ツールでのコメント除去の実装を，Java のソースコードを用いて具体的に説明する．

以下の Listing 6 に示す removeComment メソッドは，ソースコードを表す String 型のデータと，コメントの正規表現を表す String 型のデータと，文字列の正規表現を表す String 型のデータを引数に渡すと，コメントが除去されたソースコードが返されるものである．正規表現の例を，2 行目と 3 行目に緑字でコメントとして例示している．このメソッドで引数に与えた文字列の正規表現全体を，正規表現のグループ名を割り当ててグループ化する必要がある．4 行目では，str というグループ名を設定し，5 行目で文字列の正規表現をそのグループ名を用いてグループ化している．6 行目では，変換後ソースコードを貯める StringBuffer を定義する．7 行目ではコメントと文字列の正規表現を結合したものを Pattern 型にコンパイルし，Matcher のインスタンスを生成している．8 行目から 13 行目の while ループでは，ソースコード中で正規表現にマッチした部分に対する処理を前から順に行っている．9 行目は前回マッチした次の部分から今回マッチした直前の部分までの文字列を StringBuffer に貯める処理である．10 行目では 12 行目は三項演算子を用いた式である．10 行目の条件式は，マッチした部分がコメントであれば真になり，11 行目の値をバッファに追加する．文字列であれば偽になり 12 行目の値をバッファに追加する．真ならば，正規表現を用いて改行や空白を表す文字以外を空白に置換した文字列が返され，偽ならば文字列をそのまま追加する．14 行目はマッチが終了したあとの残りの文字列をバッファに貯める処理である．最後に 15 行目に，バッファに貯めた文字列を String 型で返り値としている．

```
1  public static String removeComment(String source, String COM, String STR) {
2    //COM = "/\\*[\\s\\S]*?\\*/|//((?![\\r\\n])[\\s\\S])*";
3    //STR = "\\\"((?!\\\")[\\s\\S])*?\\\"";
4    String groupName = "str";
5    STR = "(?<"+ groupName + ">" + STR + ")";
6    StringBuffer sb = new StringBuffer();
7    Matcher m = Pattern.compile(COM + "|" + STR).matcher(source);
8    while (m.find()) {
9        m.appendReplacement(sb, "");
10       sb.append(m.group(groupName) == null
11               ? m.group().replaceAll("\\S", "␣")
12               : m.group());
13   };
14   m.appendTail(sb);
15   return sb.toString();
16 }
```

Listing 6: removeComment メソッド

4　評価実験

本章では，構文定義記述解析モジュールを含め，本研究で開発した CCFinderSW の有用性を確認するために行った 4 つの実験について説明する．

4.1　構文定義記述解析モジュールを用いた文法情報抽出実験

本研究で開発したモジュールを用いて，どの程度の構文定義記述ファイルにおいてコメントと予約語の情報が抽出可能であるかを確認するための実験を行った．

実験の対象となるファイルは Github のリポジトリである grammars-v4[7]を使用した．このリポジトリは ANTLR の構文定義記述ファイルを集めたものであり，約 150 種類が含まれている．ANTLR の開発者である Parr も含めて，170 人以上の貢献者が存在し，現在でも更新が続けられている．実験で使用したリポジトリのスナップショットは 2017 年 12 月 14 日時点のものである．

本研究では，リポジトリに含まれている 154 種類の文法の中から，Github の Advanced Search[8]の検索対象に登録されている 42 言語の構文定義記述ファイルを実験対象とした．次にその 42 言語の構文定義記述ファイルからコメントと予約語と考えられる定義についてあらかじめ記録しておき，構文定義記述解析モジュールがコメントと予約語についての情報を抽出できるかどうか判定した．

本研究における予約語の定義について記述する．一般的に予約語とは，変数名や関数名に使用できない文字列のことを指し，それぞれのプログラミング言語によって定められている．一方，プログラミング言語におけるキーワードとは，あるプログラミング言語で使用される特別な意味を持つ文字列のことを指す．予約語とキーワードは似通った存在と言われているが，言語によってはキーワードであっても予約語ではない文字列が存在し，そもそも予約語が存在しない言語も存在する．例として，実験対象言語の 1 つである Fortran には予約語は存在せず，キーワードも変数名に使用できる．この実験では，各言語で使用されるキーワードがそのまま予約語になると定義して実験を行った．

表 2 は，各言語の構文定義記述ファイルに対してコメントと予約語についての情報を抽出結果を示したものである．各言語に対し，モジュールがコメントと予約語の情報が抽出可能かどうかを示している．○は抽出可能，×は抽出不可能を示し，「ー」は構文定義記述ファイルにコメントと文字列と予約語と考えられる記述がなかったことを示している．表 2 で分かるように，本研究で開発したモジュールでは，プログラミング言語の文法を表す 42 の構文定義記述ファイルのうち，コメントは 93 %，予約語は 98 %，文字列は 88 %から抽出する

[7]https://github.com/antlr/grammars-v4
[8]https://github.com/search/advanced

ことができた．もともと文法情報が定義されていないものも含めて，3 つとも抽出出来たのは 34 言語で，これは全体の 81 %にあたる．

次に，抽出不可能となった理由についてそれぞれ記述する．

css3 予約語情報の抽出ができない．css の構文定義記述の中では，抽出したい予約語の中に改行が含まれることを許すような記述がされているなど，特殊な書き方がされているため抽出不可能と判断した．

csharp 文字列情報の抽出ができない．文字列定義の種類が多いことに加え，文字列の文法が ANTLR の pushMode と popMode を用いた表現がされているため，抽出方法が複雑な実装になると判断したため．

kotlinlexer 文字列情報の抽出ができない．文字列の文法が ANTLR の pushMode と popMode を用いた表現がされているため，抽出方法が複雑な実装になると判断したため．

lua コメント情報の抽出ができない．lua に用いられるコメント記号が構文定義記述の中で複雑に定義されており，正規表現での抽出は不可能と判断した．

phplexer コメント情報と文字列情報の抽出ができない．コメントと文字列の文法が ANTLR の pushMode と popMode を用いた表現がされているため，抽出方法が複雑な実装になると判断したため．

prolog 文字列情報の抽出ができない．文字列に再帰的な文法が含まれているため．

protobuf 文字列情報の抽出ができない．文字列を定義するルール名が ‘StrLit’ であったため，抽出できなかった．

rexx コメント情報の抽出ができない．構文定義記述でネストを許す複数行コメントの定義が，2 つのルールをループすることで表現されているため，他の言語の構文定義記述とは違った特殊な書き方がされているため，抽出は不可能と判断した．

表 2: 文法情報抽出実験の対象言語と実験結果

ファイル名	コメント	予約語	文字列	ファイル名	コメント	予約語	文字列
agc	○	○	ー	modelica	○	○	○
antlrv4lexer	○	○	○	m2pim4	○	○	○
apex	○	○	○	objective-c	○	○	○
asm6502	○	○	○	pascal	○	○	○
aspectjlexer	○	○	○	phplexer	×	○	×
c	○	○	○	plsqllexer	○	○	○
clojure	○	○	○	prolog	ー	ー	×
cobol85	○	○	○	protobuf3	○	○	×
cool	○	○	○	python3	○	○	○
cpp14	○	○	○	r	○	ー	○
csharp	○	○	×	rexx	×	○	○
css3	○	×	○	scala	○	○	○
ecmascript	○	○	○	smalltalk	○	○	○
erlang	○	○	○	smtlibv2	○	○	○
fortran77	○	○	ー	swift3	○	○	○
golang	○	○	○	vba	○	○	○
htmllexer	○	ー	○	verilog2001	○	○	○
idl	○	○	○	vhdl	○	○	○
java9	○	○	○	visualbasic6	○	○	○
kotlinlexer	○	○	×	webidl	○	○	○
lua	×	○	○	xmllexer	○	ー	○

4.2 構文定義記述を用いた正解コードクローン検出実験

本節では，ANTLR の構文定義記述から抽出したコメントと文字列に相当する正規表現と予約語一覧が，コードクローン検出へ適用可能であることを確認するために行った実験について記述する．

実験の手順について説明する．まず，ある言語のソースコードを 1 つ用意し，そのソースコードのコピーを同時に作成する．この時点では 2 つのソースコードは同一のため，タイプ 1 のコードクローンといえる．次にコピーのソースコードに対して，適切な箇所にコメントを追加して変数名を書き換える．このような処理によって，手動でタイプ 2 のコードクローンを作成する．最後に，このオリジナルとコピーしたソースコードと，構文定義記述から抽出した情報を入力として CCFinderSW に渡し，コードクローン検出を行い，タイプ 2 のコードクローンとして検出されることを確認する．

本実験の対象として 9 言語を選出した．この言語は 4.1 節の実験でコメントと予約語のどちらも抽出可能となった言語のうち，変数名の変更とコメントの追加が十分に可能と考えられたものを複数選択した．

コードクローン検出の結果を，表 3 に示した．全ての言語においてタイプ 2 のコードクローンの検出が可能であることが確認できた．

表 3: 正解コードクローン検出の結果

ファイル名	コードクローン検出
c	○
cpp14	○
csharp	○
golang	○
java9	○
python3	○
smalltalk	○
vhdl	○
visualbasic6	○

4.3 GitHub リポジトリに対する構文定義記述を用いたコードクローン検出実験

本節では，ANTLR の構文定義記述を用いて，GitHub のリポジトリ上のソースコードへのコードクローン検出が実行可能であることを確認するために行った実験について説明する．

実験の手順について説明する．まず，4.1 節の実験で対象となった 42 の構文定義記述のそれぞれに対し，GitHub のリポジトリを 1 つ選択する．これには，GitHub のコード検索エンジンを用いて上位に 10 件以内に提示されるリポジトリの中で，対象言語で記述されたソースコードが十分に存在するものを選択した．次に選択したリポジトリのソースコードと対象言語の構文定義記述を提案ツールに入力として与え，正常に実行が終了したことを確認する．

表 4 は，実行対象としたリポジトリの一覧である．42 言語のうち 1 言語については，GitHub 上にソースコードが存在しなかったため，該当なしと表記している．残りの 41 言語については，すべての言語について提案ツールによる正常にコードクローン検出を行うことできた．

表 4: 対象リポジトリの一覧

agc	https://github.com/chrislgarry/Apollo-11
antlrv4lexer	https://github.com/antlr/grammars-v4
apex	https://github.com/twilio/twilio-salesforce
asm6502	https://github.com/jefftranter/6502
aspectjlexer	https://github.com/krimple/spring-roo-in-action-examples
c	https://github.com/git/git
clojure	https://github.com/clojure/clojurescript
cobol85	https://github.com/neopragma/cobol-unit-test
cool	https://github.com/bjkail/cool-cool
cpp14	https://github.com/udoprog/c10t
csharp	https://github.com/StackExchange/Dapper
css3	https://github.com/FezVrasta/bootstrap-material-design
ecmascript	https://github.com/eserozvataf/laroux.js
erlang	https://github.com/ninenines/cowboy
fortran77	https://github.com/pymc-devs/pymc
golang	https://github.com/mholt/caddy
htmllexer	https://github.com/google/material-design-lite
idl	該当なし
java9	https://github.com/airbnb/lottie-android
kotlinlexer	https://github.com/worker8/TourGuide
lua	https://github.com/cedlemo/blingbling
modelica	https://github.com/lbl-srg/modelica-buildings
m2pim4	https://github.com/congdm/Patchouli-Compiler
objective-c	https://github.com/dzenbot/DZNEmptyDataSet
pascal	https://github.com/cheat-engine/cheat-engine
phplexer	https://github.com/WordPress/WordPress
plsqllexer	https://github.com/utPLSQL/utPLSQL
prolog	https://github.com/sebschub/FontPro
protobuf3	https://github.com/dotabuff/yasha
python3	https://github.com/django/django
r	https://github.com/tidyverse/dplyr
rexx	https://github.com/trothr/znetboot
scala	https://github.com/apache/spark
smalltalk	https://github.com/redline-smalltalk/redline-smalltalk
smtlibv2	https://github.com/eclsnowman/Eustathios-Spider-V2
swift3	https://github.com/onevcat/Kingfisher
vba	https://github.com/VBA-tools/VBA-Web
verilog2001	https://github.com/ejrh/cpu
vhdl	https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner
visualbasic6	https://github.com/ImminentFate/CompactGUI
webidl	https://github.com/aduros/webidl-externs
xmllexer	https://github.com/magro/msm-sample-webapp

4.4 C++におけるコードクローン検出結果の CCFinderX との比較実験

本節では，C++で記述されたソースコードに対するコードクローン検出において，同じ字句単位のコードクローン検出ツールである CCFinderX との検出結果の比較と分析を行う．この実験の目的は，CCFinderX が対応しているプログラミング言語において，CCFinderSW で検出されるコードクローンが，ほぼ同等の検出能力を持つことを示すことである．

まず，実験の手順について記述する．同一のソースコードに対して，CCFinderX と CCFinderSW でコードクローン検出を行う．この際，2 つのツールで用いられるパラメータや条件は可能な限り揃える．次に，それぞれのツールで検出されたクローンペアを比較し，同一または類似したクローンペアが検出されているかどうかを確認する．本研究では，2 つのツールで検出されたクローンペアが同一またはあるしきい値以上の類似度を持つことで一致しているとみなし，特に同一であるとみなされたものは完全一致と表現する．一致するものが存在しない場合，つまり片方のツールでしかクローンペアが検出されていない場合は，もう片方のツールで検出されない原因を分析する．また，それぞれで検出されたクローンペア数と，マッチしたクローンペア数について測定し，その割合を算出する．

本実験の対象言語は，CCFinderX の対応言語の 1 つである C++を選択した．実験対象となるソースコードは，git[9]の 2018 年 12 月 21 日時点でのスナップショットを使用した．

次に，CCFinderX と CCFinderSW に与えるオプションについて説明する．共通のオプションとして検出クローン片の最低字句数のしきい値を 100 にした．CCFinderX では実用的なクローンを出力するために，メトリクスなどを用いた様々なフィルタリングや，構文的な解析に基づく正規化を行っている．このため CCFinderX は，多言語に適用可能なように定義された字句分割とマッチングを用いて検出を行う CCFinderSW よりも，少ない量のクローンペアを出力する．本実験では，出来るだけ CCFinderX と CCFinderSW との検出条件を揃えるために，CCFinderX に与えるオプションのうちから 3 つを，デフォルトの値から変更した．P-match に関する検出オプションをデフォルトでオンのところをオフにし，検出されるクローン片に含まれる最低字句種類数のしきい値をデフォルト値である 12 から 0 に変更し，Block Shaper に関するオプションをデフォルト値で 2 が用いられるところを 0 に変更した．また CCFinderSW の検出についての変更として，CCFinderX の構文的な解析によってコードクローン検出に含まれない部分を，CCFinderSW に与えるソースコードから前処理的に取り除いてから検出を行った．

次に，検出されたクローンペアの比較方法の詳細と一致の定義について説明する．あるファイル A とファイル B 間に存在するクローンペアについて，CCFinderSW と CCFinderX で検出されたものを比較を行うとする．まずクローン片のそれぞれの名前についての定義として，

[9]https://github.com/git/git

CCFinderSW で検出されたクローンペアのクローン片を *SWA* と *SWB* として，CCFinderX で検出された *XA* と *XB* とする．本研究で用いる手法は行単位の一致率の計算を行う．比較するときに用いる数値としては，*SWA* と *XA* のそれぞれのクローン片の開始行と終了行，そして *SWB* と *XB* のそれぞれのクローン片の開始行と終了行となる．ここでそれぞれのクローン片の開始行と終了行の値を *start* と *end* を名前の後につけることで表現し，クローン片の長さを *len* をつけることで表現する．以上のことから，それぞれのクローン片の長さは以下のような式となる．

$$SWAlen = SWAend - SWAstart + 1 \quad (1)$$

$$SWBlen = SWBend - SWBstart + 1 \quad (2)$$

$$XAlen = XAend - XAstart + 1 \quad (3)$$

$$XBlen = XBend - XBstart + 1 \quad (4)$$

次に，このクローン片の一致率を計算式を説明する．*SWA* と *XA* で一致している部を *MA* と表現し，ファイル B においても同様とする．このとき，*MA* の長さの定義を，与えられた 2 つの値で大きい側の値の返す関数 Max と，与えられた 2 つの値で小さい側の値を返す Min を用いて，以下のように定義できる．*MB* に対しても同様の計算をする．

$$MAlen = Min(SWAend, XAend) - Max(SWAstart, XAstart) + 1 \quad (5)$$

$$MBlen = Min(SWBend, XBend) - Max(SWBstart, XBstart) + 1 \quad (6)$$

ここから *SWA* と *XA* の一致率を表す $MatchA(\%)$ を計算し，ファイル B においても $MatchB$（%）を計算する．最後に CCFinderSW と CCFinderX で検出されたクローンペアの一致率を表す $MatchSWX(\%)$ を計算する

$$MatchA = \frac{MAlen * 100}{Max(SWAlen, XAlen)} \quad (7)$$

$$MatchB = \frac{MBlen * 100}{Max(SWBlen, XBlen)} \quad (8)$$

$$MatchSWX = Min(MatchA, MatchB) \quad (9)$$

表 5 は実験の結果を示したものである．この表は，CCFinderX と CCFinderSW の検出クローンペア数，その 2 つのツールで検出されたクローンペアの一致数を表したものである．2 つのツール間で完全一致したクローンペア数は 1806 個である．検出ペア中の一致した割合は，一致した数を検出ペアの数で割った値である．検出ペア中の一致した割合の列を見ると，CCFinderX で検出されているクローンペアの約 98 %が CCFinderSW で検出されており，逆も同様に約 98 %であることから，CCFinderSW は CCFinderX とほぼ同等の検出能

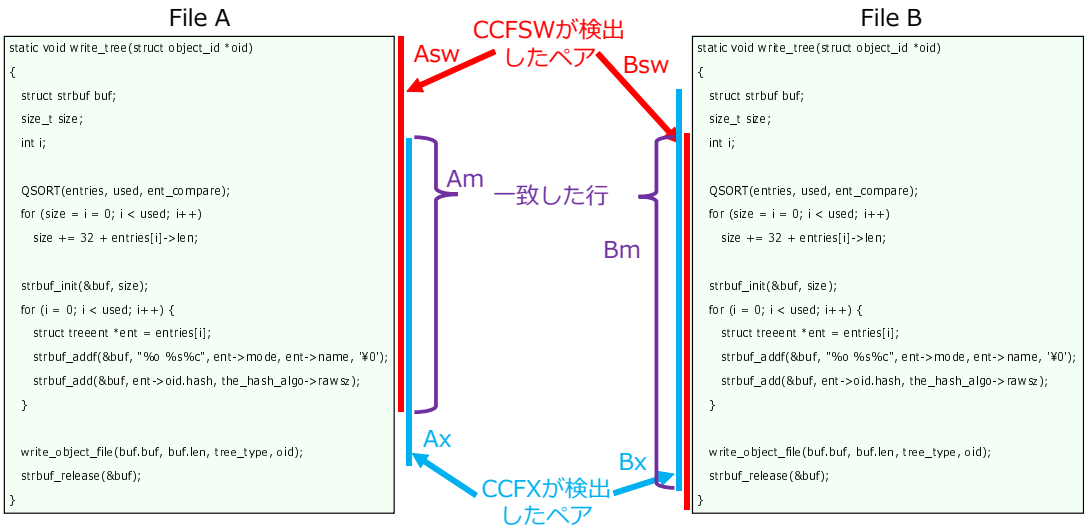

図 5: 2 つのツールで検出されたクローンペア例

力を持っているといえる．また，2 つのツールで検出されたクローンペアの包含関係を可視化するために，図 6 に示した．CCFinderX で検出されたもののうち一致していない 122 個のクローンペアの内訳は，8 割以上一致しているものが 90 個，一致していないものが 32 個である．CCFinderSW で検出されたもののうち一致していない 138 個のクローンペアの内訳は，8 割以上一致しているものが 98 個，一致していないものが 40 個である．

表 5: CCFinderX との出力クローンペアの比較結果

	検出ペア	一致したペア	検出ペア中の一致した割合	一致なし
CCFinderX	1928	1896	0.983	32
CCFinderSW	1944	1904	0.979	40

次に，CCFinderX と CCFinderSW のそれぞれで一致が見られなかったクローン片を目視で確認し，分析した結果について述べる．この説明ために，CCFinderX で検出されるクローンペアについて補足する．CCFinderX はクローンペアに含まれる 2 つのコード片の順序関係も記録している．つまり，A と B という 2 つの類似したコード片があった場合，CCFinderX は {A,B}{B,A} という 2 つのクローンペアを記録する．このことから，今回検出された CCFinderX のみで検出された 32 個のクローンペアは，16 個の類似したコード片の組み合わせになっており，それが 2 つの順序で記録されているものである．CCFinderSW においても，CCFinderX と同様に 2 つのコード片の順序関係を記録している．本実験での，分析結果はコード片の順序を無視してクローンペアを数えている．

まず，CCFinderX のみで検出された 16 個のクローンペアについて，CCFinderSW で検

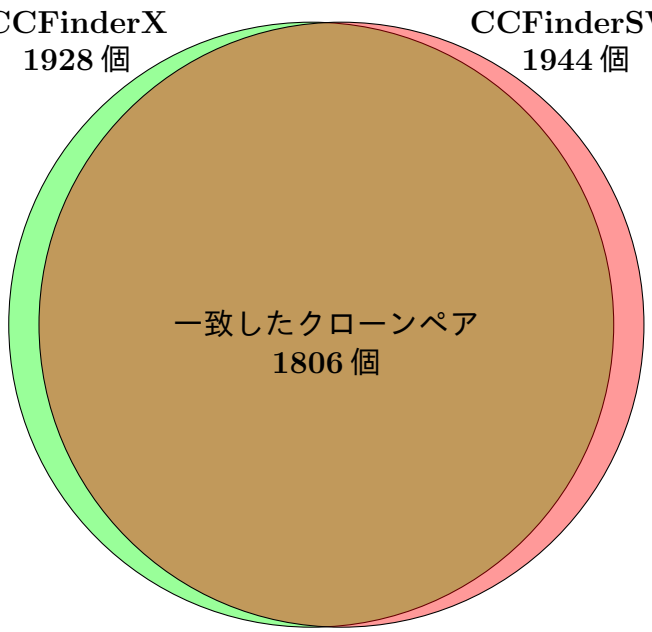

図 6: 2 つのツールで検出されたクローンペア数とマッチ数

出されなかった原因を説明する．

極大クローンペアではない

8 個存在した．何らかの理由で CCFinderX で検出されるコードクローンが極大クローンではないものであったため，CCFinderSW で検出されたコードクローンと一致しなかった．

字句定義の違い

4 個存在した．このクローンペアは CCFinderX では最低字句数のしきい値が 100 になっているが，字句定義の際のために CCFinderSW では字句数が 100 以下であるため検出されなかった．言い換えれば，このクローンペアは CCFinderSW での検出を行う際に最低しきい値を下げることで検出可能なクローンである．

unsigned を含む型定義

2 個存在した．C++ではいくつかのプリミティブ型定義の前に，**unsigned int** のように **unsigned** という予約語を入れることで，その型が符号なしであることを明示する機能がある．CCFinderSW はこの **unsigned int** を 2 つの字句で認識するが，CCFinderX では 1 つの型を表す字句として認識される．この違いにより，CCFinderSW ではコードクローンとして検出されなかったものである．

文字列の補完

1 個存在した．CCFinderX では，複数の文字列が和の形で複数行にまたがっている場合にも，1 つの字句として認識する機能があるが，CCFinderSW ではその機能がないために検出できなかった．

前処理のミス

1 個存在した．CCFinderSW で検出対象のソースコードに行った前処理が誤っていたために，出力されたクローンペアに違いが生まれたものである．

次に CCFinderSW で検出され，CCFinderX で検出されなかった 20 個のクローンペアについてについて，分析した結果を説明する．

字句定義の違い

11 個存在した．このクローンペアは，CCFinderSW では最低字句数のしきい値が 100 になっているが，CCFinderX では字句数が 100 以下であるため検出されなかった．言い換えれば，このクローンペアは CCFinderX での検出を行う際に，クローン片の最低字句数のしきい値を下げることで検出可能なクローンである．

極大クローンペアではない

8 つ存在した．何らかの理由で CCFinderX で検出されるコードクローンが極大クローンではないものであったため，CCFinderSW で検出されたコードクローンと一致しなかった．

前処理のミス

1 つ存在した．CCFinderSW で検出対象のソースコードに行った前処理が誤っていたために，出力されたクローンペアに違いが生まれたものである．

以上の結果から，CCFinderX と CCFinderSW の検出能力はほぼ同等であると判断される．また検出結果に差異があったものに関しては目視で確認し，原因を突き止めることができた．

4.5 Verilog HDL に対するコードクローン検出精度に関する実験

本節では，Verilog HDL で記述されたソースコードに対して開発した CCFinderSW を用いてコードクローン検出を行い，その検出精度を評価する実験について説明する．この実験は，CCFinderX などのコードクローン検出ツールが一般的に対応していない言語に対しても，CCFinderSW を用いることでタイプ 2 のコードクローンが検出可能であることを示すために行った．

上村らは，代表的な HDL である Verilog HDL のコードクローンの検出手法を提案し，10 件のプロジェクト中のコードクローンについて調査している [18]． この提案する検出手法は，Verilog HDL のソースコードを幾つかの変換規則に基づいて疑似 C++に変換し，CCFinderX で変換後のソースコードを C++のソースコードとしてコードクローン検出を行うものである．

本研究における，提案ツールである CCFinderSW の検出手法の評価に，上村らが用いたコードクローン検出ツールの評価手法を用いる [19]．コードクローンの検出精度は，検出されたコードクローンのうち正解の割合（Precision）と，すべての正解のうち検出できたコードクローンの割合（Recall）で評価することができる．Svajenko らは，自明なコードクローンを埋め込むことで正解集合を構築する手法を提案している [20]．この手法では，まずソースコード中から関数やコードブロック単位でコード片を複数選択し，このコード片に対して複数の変異を適用し，変異コード片をソースコード中に埋め込んでいる．そして，この変異コード片を文法上問題のない，ランダムな位置に挿入し，自明なコードクローンを生成し，正解コードクローンとして記録する．最後に，評価対象のコードクローン検出ツールが，全ての自明な正解コードクローンからそれを正しく検出できた割合を測定し，その値を Recall とする．上村らは，Svajenko らが用いた変異手法に変更を加えて変異コード片を生成して，Verilog HDL のソースコードに対するコードクローン検出の提案手法を評価している．本研究の提案ツールである CCFinderSW の評価にも，上村らが用いたデータセットと変異コード片と同じものを用いる

Svajenko らは，抽出するコード片の粒度を関数およびブロック単位とすると述べている．これに基づいて，上村らは Verilog HDL でよく用いられるブロックである，**module**，**always**，**if**，**case** の 4 種類のブロックを対象にしている．この評価対象には，Verilog HDL のプロジェクトの ridecore[10]が選ばれている．

本研究では，上村らが作成した変異コードが埋め込まれたソースコードに対して，CCFinderSW を用いてコードクローン検出を行い，Precision と Recall を測定する．この際，CCFinderSW に与える構文定義記述ファイルは， grammars-v4 の Verilog2001.g4 を使用して検出を行った．4.1 節の実験結果にある通り，CCFinderSW の構文定義記述解析モジュールはこ

[10]https://github.com/ridecore/ridecore.git

の構文定義記述ファイルからコメント文法と予約語と文字列の情報を正規表現として抽出することが出来る．

Precision は CCFinderSW によって検出されたコードクローンを手作業で正解であると判定された割合を表している．このコードクローン検出では，CCFinderSW で最低字句数 50 以上の条件を与えて検出されたコードクローンを，CCFinderX のメトリクスによるフィルタリング機能で，クローン片に含まれる字句種類数の最低しきい値を 12 としてフィルタリングを行ったコードクローンを対象とした．これは，上村らが用いた手法でも行われているフィルタリングであるため，コードクローンを確認する作業を減らすために採用した．

Recall は，変異コード片を用いて埋め込まれた正解コードクローンのうち検出されたものの割合を表している．上村らは，**module**，**always** ブロックのコード片に対しては最低クローン長を 50，**if**，**case** ブロックのコード片に対しては最低クローン長を 25 としている．本実験の CCFinderSW を用いた検出でも，この値を採用する．

上村らが行った実験結果と本ツールで検出した結果の Precision を表 6 に，Recall を表 7 に示す．総計はタイプ 1 とタイプ 2 を区別なく集計した結果である．本ツールの検出結果では，全体の Precision は 100 %，Recall は 99 %となり高い数値となっている．Recall が 100 %にならない理由として，Verilog HDL では，変数名にグレーブアクセント（’）を含めることが出来るが，CCFinderSW で行われている字句分割においてグレーブアクセントは変数名に用いられる文字に設定されていないため，変数名の正規化がうまくいかなかったことがある．

表 6: Precision の測定結果

Recall	上村らの手法					CCFinderSW				
(%)	module	always	if	case	合計	module	always	if	case	合計
タイプ 1	100	100	100	97	99	100	100	100	100	100
タイプ 2	99	100	100	95	98	100	100	100	100	100
総計	99	100	100	96	99	100	100	100	100	100

表 7: Recall の測定結果

Recall	上村らの手法					CCFinderSW				
(%)	module	always	if	case	合計	module	always	if	case	合計
タイプ 1	92	95	98	89	94	99	99	99	99	99
タイプ 2	74	94	98	94	89	98	100	100	100	99
総計	86	95	98	91	93	99	99	99	99	99

以上の結果から，Verilog HDL で記述されたソースコードに対するコードクローン検出において，CCFinderSW は既存手法よりも高い精度で検出可能であった．

5　まとめと今後の課題

本研究では多様なプログラミング言語に対応したコードクローン検出ツールを開発することを目的として，字句単位のコードクローン検出における字句解析に必要な文法情報を，構文解析器生成系の構文定義ファイルから自動的に抽出するモジュールを開発した．そしてそのモジュールを用いて，ANTLR の構文定義記述を入力として与えることで，対象言語の文法に沿ったコードクローン検出が可能な CCFinderSW を開発した．

開発したモジュールに対する評価実験では，プログラミング言語の文法を表す ANTLR の構文定義記述ファイルのうち，コメントは 93 %，予約語は 98 %，文字列は 88 %のファイルから抽出することができた．次に提案ツールである CCFinderSW への評価実験では，C 言語のソースコードに対する CCFinderX との比較においては，CCFinderX で出力されるコードクローンの 98 %が CCFinderSW でも検出可能であることを示した．また，Verilog HDL で記述されたソースコードに対して，CCFinderSW の Precision と Recall を測定し，既存手法より優れた精度で検出が行えていることを示した．

構文定義記述解析モジュールの問題点として，構文定義記述は対象となるプログラミング言語の文法に依存するが，書き手にも依存する．つまり，同じ文法であっても複数の記述法で表現することができる．本研究での開発は，構文定義記述の調査において多く存在した記述法に対して行われたものである．新たに異なる記述法に対応するためには，モジュールを拡張しなければならない．

文法情報を正規表現で抽出することに対する問題点として，提案モジュールによって自動的に抽出された正規表現が長くなった場合に，既存手法のコメント除去の処理と比較して，処理時間が数十倍以上になることが挙げられる．また，使用するスタックも増大するため，本ツールに与えられたスタック容量を超過し，プログラムが停止してしまう可能性が考えられる．このような問題点から，冗長な正規表現に最適化を施すか，別の表現を用いたコメント除去を行う必要がある．

本研究で開発したモジュールで抽出される文字列はプログラミング言語のキーワードであり，予約語かどうかの判定は行われていないため，抽出されたキーワードが予約語であるか判定することが挙げられる．また，追加の評価実験として，より多くの言語に対し CCFinderSW を適用していくことが挙げられる．

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，研究に関する多くの御指導及び御助言を賜りました．研究の各段階において，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 神田 哲也 助教には，研究生活においても常に貴重な御意見と様々なサポートを賜りました． 神田 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，常に研究に関する様々な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先端科学技術研究科情報科学領域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．

奈良先端科学技術大学院大学先端科学技術研究科情報科学領域ソフトウェア設計学研究室 上村 恭平 氏には，本論文の評価に用いるためのデータセットや実験結果の提供を，快く受け入れてくださいました．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] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Ccfindersw: Clone detection tool with flexible multilingual tokenization. In *Proc. of APSEC 2017*, pp. 654–659. IEEE, 2017.

[5] 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.

[6] 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.

[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 *Proc. of ICST 2013*, pp. 422–430. IEEE, 2013.

[8] Stephen C Johnson. *Yacc: Yet another compiler-compiler*, Vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[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] 横井一輝, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく細粒度ブロッククローン検出. コンピュータ ソフトウェア, Vol. 35, No. 4, pp. 16–36, 2018.

[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] Terence Parr. About The ANTLR Parser Generator. http://www.antlr.org/about.html.

[15] Terence Parr. antlr4/index.md at master antlr/antlr4. https://github.com/antlr/antlr4/blob/master/doc/index.md.

[16] Terence Parr. *The definitive ANTLR 4 reference*. 2013.

[17] Jeffrey EF Friedl. *Mastering regular expressions*. 2002.

[18] Kyohei Uemura, Akira Mori, Kenji Fujiwara, Eunjong Choi, and Hajimu Iida. Detecting and analyzing code clones in HDL. In *Proc. of IWSC 2017*, pp. 1–7. IEEE, 2017.

[19] 上村恭平, 森彰, 藤原賢二, 崔恩瀞, 飯田元. ハードウェア記述言語におけるコードクローンの定量的調査. 情報処理学会論文誌, Vol. 59, No. 4, pp. 1225–1239, apr 2018.

[20] Jeffrey Svajlenko and Chanchal K Roy. Evaluating modern clone detection tools. In *Proc. of ICSME 2014*, pp. 321–330. IEEE, 2014.