修士学位論文

題目

変数のデータフローを考慮した API 利用コード例の検索手法

指導教員
井上 克郎 教授

報告者
竹之内 啓太

平成 29 年 2 月 7 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 28 年度 修士学位論文

変数のデータフローを考慮した API 利用コード例の検索手法

竹之内 啓太

内容梗概

ソフトウェア開発の効率を向上させるため，ライブラリなどの API (Application Programming Interface) を利用する形でソフトウェアを再利用することは一般的である．一方で，巨大化・複雑化した API の利用は必ずしも容易ではないことが知られている．それに対し，API の利用方法の理解を支援する手法が数多く提案されてきた．なかでも，コード検索による API の理解は一般的に普及している．

本研究ではコード例の検索により API の理解を支援する手法を提案する．提案手法の特徴として，(1) 「変数のデータフロー」を (独自の検索クエリを用いて) 指定し，API の理解に有益なコード例を検索する点，(2) 検索対象となるソースコードを既存のコード検索エンジンから取得することで，さまざまな API の検索に対応している点，(3) 有限オートマトンを利用した軽量なアルゴリズムを用いることでウェブアプリケーションとしての実装を実現している点などが挙げられる．また，コード例のランキングや整形方法も先行研究には見られない独自の方式を用いている．

提案手法の評価のため被験者実験を行い，その有効性や独自の検索クエリの評価，検索に要する時間等を調査した．その結果，提案手法が API の理解を有効に支援する場合があることや，検索クエリの記述が比較的容易であること，検索時間は実用的な範囲に収まることを確認した．

主な用語

API 利用例
データフロー
コード検索

目 次

1 はじめに

ソフトウェア開発において，ライブラリなどの API (Application Programming Interface) の利用は欠かせないものとなっている [1]．API を利用することで，第三者が過去に作成したソフトウェアを再利用することができ，開発効率の向上につながる．API の仕様はドキュメントとして記載されている場合が一般的であり，開発者はそれを参照することで API の利用方法を学習することができる．しかしながら， API が巨大化・複雑化している場合は，その利用が困難となることが知られている [2][3]．そこで， API の利用方法の理解を支援する手法が数多く提案されてきた．先行研究では，具体的な API の利用例を開発者に提示するもの ([8], [10], [11], [12], [15]) や，API メソッドの利用パターンのような抽象的な情報を提示するもの ([9], [1], [13], [14]) が存在する．また，コード例検索により実際に API を利用しているコード例を見つけ出し，それを見て学習することは一般的である．コード検索エンジンはウェブサービスとして利用できるため導入の手間がいらず，検索結果を即座に取得できるため，広く利用されている [4]．

しかしながら， API に対する検索の要望は常に 1 つであるとは限らない．そのため，一般的なコード検索エンジンで利用されている単語単位の検索では，要望を上手く表現することが容易ではない場合がある．たとえば，`Java` の標準ライブラリに含まれる `Statement` インターフェースの `executeQuery` メソッドはデータベースを操作する SQL 文を実行するためのメソッドである．このメソッドの実行には，次の 2 つの実現方法を理解する必要がある．

1. メソッド呼び出しのレシーバオブジェクトとなる， `Statement` インターフェースの実装クラスのインスタンスを生成する．
2. `execute` メソッドの戻り値である `ResultSet` 型の変数から SQL 文の実行結果を取り出す．

一般的なコード検索エンジンを利用する場合，“`executeQuery`” というメソッド名で検索し，これらの実現方法を調査することになる．検索結果は一般的に検索ワードにマッチした行と前後の行程度の範囲のみが表示されるため，結局元のソースコード全体を見て検索の要望にマッチしているかどうかを判断することになる．これはソースコード全体から興味のある一部分のコード片にたどり着く手間がかかり，効率的とは言えない．また，メソッド名がコメント行に含まれている場合など，有益なコード片がそもそも含まていない場合も考えられる．つまり，一般的なコード検索エンジンは，ソースコード中に “`executeQuery`” という文字列が含まれている以上のことは保証していないのである．

そこで本研究では，開発者にとって有益なコード例を適切に提示する手法を考案した．具体的には，独自の検索クエリを利用することで「変数のデータフロー」を表現し検索するという方式を提案する．この検索クエリは基本的には検索対象のプログラミング言語のトー

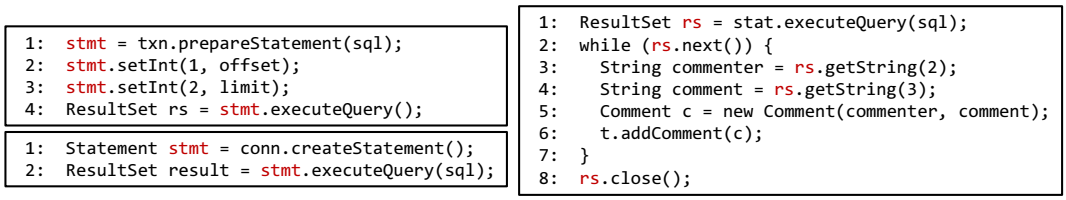

図 1: 提案手法の検索結果の例

クンの列であるが，3 つの特殊な意味を持つワイルドカードを導入することで検索に柔軟性を持たせた．たとえば，上記の 1. の実現方法を検索するには executeQuery メソッドのレシーバとなる変数への代入を検索すればよく，それは次のような検索クエリによって表現できる．

```
$st = ?? $st.executeQuery
```

ここで，“$st” は $変数と呼ばれるワイルドカードであり，1 つの変数名にマッチする．また，“??” は改行含む任意のトークン列にマッチするワイルドカードであるこの検索の結果，図 1 の左側のようなコード例が検出される．$st には 1 つの変数名のみがマッチし，それを赤字で表現している．これらのコード例から createStatement や preparedStatement といったメソッド呼び出しによって，Statement インターフェースの実装クラスのオブジェクトが取得できることが読み取れる．これらのコード例では変数 stmt にオブジェクトが「代入」された後，メソッド呼び出しに「使用」されることが分かる．このような変数の代入から使用までの流れのことを，本研究では「変数のデータフロー」と呼んでいる．すなわち，変数のデータフローを考慮することにより複数の API メソッドの関係を表現し検索することが本研究の目的である．

上記の 2. の実現方法を検索するには executeQuery メソッドの戻り値がその後どのように扱われるのかを調査すればよい．たとえば，以下のような検索クエリが考えられる．

```
$rs = _ . executeQuery ?? $rs
```

ここで，“_” は 1 つの文の内部におけるトークン列にマッチするワイルドカードである．その結果，図 1 の右側のようなコード例が検出される．このコード例における変数 rs のデータフローにより，next, getString, close とった SQL の実行結果を取り出すための一連のメソッド呼び出し列を知ることができる．

本手法は検索対象となるソースコードを既存のコード検索エンジンから取得するため，さまざまな API の検索の要望に対応している．また，検索クエリの設計と検索のマッチングアルゴリズムを工夫し，検索の計算コストを抑えることで，応答性のよいウェブアプリケーションとしての実装を実現している．

以降，初めに 2 章では研究背景について詳細に述べる．続いて，3 章では提案手法の概要について，4 章ではそのアルゴリズムについて述べる．5 章では評価実験について述べ，最後に 6 章ではまとめと今後の展望について述べる．

2 背景

2.1 API の利用

API (Application Programming Interface) はソフトウェア開発において用いられる，再利用可能なコンポーネント群のことである．ソフトウェアの開発において，開発者自身が作成するソフトウェアをすべてを開発するということは少なく，サードパーティが開発したライブラリ等の API を利用することが一般的である [1]．API の利用は開発のコストの削減につながるだけでなく，十分にテストされた信頼できるコンポーネントを再利用することが可能となり，ソフトウェア品質の向上にもつながる．

API には，それを使用するためのドキュメントが存在することが一般的である．たとえば，Java の標準ライブラリのドキュメント [1] には，標準ライブラリに含まれる全クラスの階層構造や，メソッドの仕様等が記述されている．これらの情報によって，開発者は API の利用方法を学習することができる．しかしながら，ドキュメントに問題がない場合であっても巨大化・複雑化した API の利用は容易ではない [2] [3]．たとえば， API のドキュメントにより個々のメソッドの動作が既知の場合であっても，複数のメソッドを組み合わせて利用しなければならない場合，開発者がその利用方法を学習することは容易ではない．また，使用すべきクラスが既知であるものの，そのインスタンスの生成方法が不明な場合なども同様である．

2.2 コード例検索

API の利用が困難となる原因として，4 分の 1 以上の開発者が API の利用方法を示したコード例が存在しないことを挙げている [2]．したがって，コード例は API を理解するための重要な役割を果たしており，コード例検索は API の利用のための重要なツールであると考えられる．また，多くの開発者にとってコード検索は一般的な行為であり， Caitlin ら [5] の Google における調査では，開発者は 1 日に平均値で 12 回，中央値で 6 回のコード検索を行うことを明らかにしている．そのため，開発者をコード検索により支援することは，開発者への負担が少なく有効な手段であると考えられる．

2017 年 2 月現在利用可能なコード検索エンジンとして，Krugle [6] や searchcode.com [7] などが挙げられる．これらのコード検索エンジンは，インターネット上に存在する膨大な数のオープンソースリポジトリにおけるさまざまなプログラミング言語のソースコードを検索対象としている．調べたいメソッド名やクラス名を入力することで，それらを実際に使用しているコード例が提示される．Krugle では検索のオプションとして，あるメソッド名に対

[1] https://docs.oracle.com/javase/jp/8/docs/api/

して「呼び出しを行っている箇所」のみを検索するなどのフィルター機能を持つ．このフィルター機能により，開発者にとって興味のない，たとえば同名のメソッドを「定義」している箇所などを検索結果から排除できる．また，2013 年まで存在していたコード検索エンジン Google Code Search は検索クエリとして正規表現が使用でき，複数のメソッド名を同時に検索する検索や，ソースコードの一部の差を無視した検索が可能となっていた．これらの機能により，開発者の要望をより正確に表現する検索を実現していた．

2.3 その他の API 理解支援

コード例検索技術の他にも，多くの API 理解支援の手法が提案されている．

Strathcone [8] は，開発者が特定の API を利用してコードを実装しているとき, 関連するコード例をリポジトリから検索し提示する手法である．開発者は実装中のコードに類似したコード例を見ることができるため，API の利用方法を学習することができる．検索クエリは実装中のコード片から自動的に抽出されるため，開発者自身が検索クエリを記述する必要はない．

PROSPECTOR [9] は，入力の型 T_{in} と出力の型 T_{out} を (T_{in}, T_{out}) という 2 つ組の検索クエリにより指定すると，型 T_{in} から型 T_{out} を生成するようなコード片を合成する手法である．この手法では API のシグネチャから，型名をノード，型から型への変換をエッジとして表現したグラフを構築し，グラフにおけるパスを計算することで検索クエリに合致するコード片を合成している．

XSnippet [10] は，型名を入力としてインスタンスの生成方法を示したコード例を提示する手法である．型名とともに検索の文脈を選択することで，開発者の目的に合わないものを出力から除外している．

PARSEWeb [11] は，PROSPECTOR と同様に入出力の型を検索クエリとして指定し，コード例やメソッドの呼び出しの列を提示する手法である．それまでの手法と異なり，検索対象となるコード集合はコード検索エンジン (Google Code Search) からウェブ経由で収集される．これにより，特定の API を利用しているクライアントサイドのプログラムを収集する準備の手間がなくなり，より幅広い API やフレームワークの検索が可能となっている．一方で，出力するコード例を選定するためのアルゴリズムの計算コストが高く，あらかじめコード検索エンジンからソースコードをダウンロードし計算を行っておく必要がある．結果として，ソースコード検索エンジンからコードを収集することによるスケーラビリティを最大限に生かしきれないという課題がある．

SNIFF [12] は，自然言語 (英語) を検索クエリとして，それにマッチするコード例を提示する手法である．自然言語の検索クエリを用いているため，開発者が使用すべきクラス名やメソッド名が分かっていない状態でも対応が可能であるという利点を持つ．

MAPO [1] は，コード集合から抽象的な API の使用パターンをマイニングする手法である．この手法により，複数のメソッドをどのような順序で呼び出すべきかという，API のドキュメントに不足しがちな情報を補うことができる．一方で，マイニングアルゴリズムの計算コストは高く，20 個の Java プロジェクトからパターンを検出するのに数時間を要するなど，スケーラビリティに課題がある．

UP-Miner [13] は MAPO と同様に抽象的な API の利用例をマイニングする手法である．検出されたパターンの質を評価する 2 つのメトリクスを導入することで，MAPO と比較して，より簡潔で網羅性の高いパターンを検出することが可能となっている．また，手法の出力として，合成されたコード例だけでなく probabilistic graph という，あるメソッドの次に使用されるメソッドを確率的に表現したグラフを採用しているもの特徴的である．

MLUP [14] は MAPO, UP-Miner と同様に API の利用パターンを検出する手法である．ともに使用されるメソッド同士の関係性を,「必ずともに使用される場合」と「選択的にともに使用される場合」の 2 つのレベルに分類していることが先行研究と異なる点である．複数のレベルの関係性を考慮することで，より質の高いコード例を提示することに成功している．

MAPO, UP-Miner, MLUP のような API の利用パターンをマイニングする手法は，提案手法と同様に質の良いコード例を提示することを目的としている．しかしながら，これらの手法を利用する前準備として，特定の API を利用している複数のクライアントサイドのプログラムをあらかじめ収集し，数時間をかけてマイニングを行っておく必要がある．そのため，特定の API を利用している十分な数のプログラムを収集するのが困難な場合や，API の中でもごく一部のものについてのみ学習したい場合には適していないと言える．

MUSE [15] は使用する API のドキュメントの各メソッドの説明欄に，そのメソッドが使用されているコード例を埋め込む手法である．API を用いた開発ではドキュメントは主要な情報源であり，ドキュメントにコード例を記載することで開発者の有効な支援につながる．出力するコード例を求めるアルゴリズムでは，メソッドに関係のない部分はプログラムスライシング [16] によって除外される．また，類似したコード例が複数表示されることを防ぐため，コードクローン検出 [17] に基づいたフィルタリングを行っている．

3 提案手法

本研究では，API の利用方法の理解を支援するため，変数のデータフローを考慮したコード例検索手法を提案する．一部の先行研究 [11] と同様に，検索対象となるコード集合をコード検索エンジン (searchcode.com) から取得する．その結果，前準備なしに様々な API の検索に対応するが可能である．コード検索エンジンを利用することによるスケーラビリティを最大限に活かすため，コードの収集をあらかじめ行うのではなく，検索クエリに応じてコード検索エンジンから必要なコード片を収集し開発者にコード例を提示する．具体的には，検索クエリ中に含まれるすべての識別子名を抽出することで searchcode.com の単語単位の検索クエリに変換し，検索対象となるソースコードを取得している．したがって，コード検索エンジンから取得可能な任意のコード片を開発者に提示する候補とすることが可能である．一方で，このような方式を取る場合は，開発者が検索クエリを送信してから結果が返ってくるまでの応答時間が長くなる傾向にある．そこで本研究では，提示するコード例のマイニング等による選定を行わず，開発者の要望を検索クエリにより表現するという方式をとる．

既存のコード検索エンジンである Google Code Search では正規表現によるコードの検索が可能であったものの，開発者の API 利用例の検索におけるさまざまな要望を表現するには不十分であると考えられる．本研究では，検索クエリによりコード中に現れる「変数のデータフロー」を指定できるようにすることで，従来のコード検索エンジンに比べ，より限定的な文脈を持つ検索を実現した．データフローという言葉は，元々データフロー解析というプログラムの静的解析手法において見られるものである．データフロー解析は，変数がある箇所で定義され他の箇所で使用される場合に対し，定義された部分から使用される部分までを変数の流れとして捉えるような手法である [18]．本研究では，1 章の例のような特定の API とクライアントサイドの変数のデータフローの関係が見て取れるコード例を提示するすることが，開発者の API 理解を支援する手法として有益であるという立場をとっている．一方で，データフローという考え方は開発者にとって一般的なものではなく，望みのデータフローを直感的に精緻に指定するのは容易でないと考えられる．そこで本研究では，3 つの特殊な意味を持つワイルドカードを用いたトークン列のマッチングにより，近似的なデータフローを指定する方式をとっている．正規表現を書くように，特定のトークン列の並びを指定するだけでデータフローを表現した検索クエリを記述することが可能となっている．

提案手法はウェブアプリケーションとして実装を行った．ウェブアプリケーションとして実装することで，開発者にツール導入の手間が無くなるという利点がある．また，より幅広い開発者を支援するため， `Javva`, `C/C++`, `C#`, `Ruby`, `JavaScript` の 5 つのプログラミング言語のコード例の検索に対応した．先行研究では，ソースコードの静的解析を行うため検索対象となるプログラミング言語は 1 種類であるものが多い．提案手法はトークン列のマッチ

ングという簡易的な手法を用いているため，新たなプログラミング言語を追加するコストは先行研究より低いと言える．

本章では，検索クエリの仕様とマッチングの仕様について述べる．具体的なアルゴリズムについては 4 章で述べる．

3.1 検索の仕様

検索対象となるソースコードは，コメントや空白，改行のようなレイアウト情報を取り除いたトークン列として扱う．これは，CCFinder [19] のようなトークン単位のコードクローン検出ツールと同様である．また，プログラミング言語の構文についての情報として，出現順序に対応関係の制約を持つトークン（たとえば “{” と “}” ， “(” と “)”）と，変数への代入を意味するトークン (“=”)，文の区切り文字 (“;”) についての情報は与えられるものとする．トークン列への変換は字句解析器，あらかじめ与えるトークンの情報は文法定義にのみ依存するため，多くのプログラミング言語に対して適用可能である．本論文では `Java` を例として説明する．

本手法では，検索クエリを指定することにより，検索対象から特定のパターンを持つトークン列を検出する．以下に述べる 3 つの特殊トークン以外は，トークンの種類や識別子名が完全に一致した場合にのみ，2 つのトークンが等しいと判定する．たとえば，検索クエリ中で “`foo`” という識別子を持つトークンが指定されると，そのクエリは “`bar`” や “`FOO`” という識別子を持つトークンにはマッチしない．したがって，検索クエリとして特殊トークンを使用せずに検索を行った場合，コードクローンの中でもタイプ 1 [17] のクローン検出と同等の検索となる．

以下に特殊トークンについて述べる．これらはマッチする対象が互いに異なるワイルドカードである．

$変数 – 変数にマッチするワイルドカード

> “`$`” の後に識別子名が続くようなトークンであり，任意の 1 つの識別子名にマッチするワイルドカードである．たとえば，“`$a`” や “`$list`” のようなトークンは `$`変数であり，それぞれ 1 つの識別子名にマッチする．検索クエリにおいて，同一の `$`変数が複数箇所に出現する場合，すべての出現箇所に対し同じ識別子名がマッチしなければならない．この `$`変数により，同じ変数名の変数が複数の箇所に出現するようなコード例を検索することができる．すなわち，変数のデータフローを指定することが可能となる．

“_” (アンダースコア) – 文内のトークン列にマッチするワイルドカード

> “`_`” は長さ 0 以上の開き括弧と閉じ括弧の対応関係が取れたトークン列にマッチす

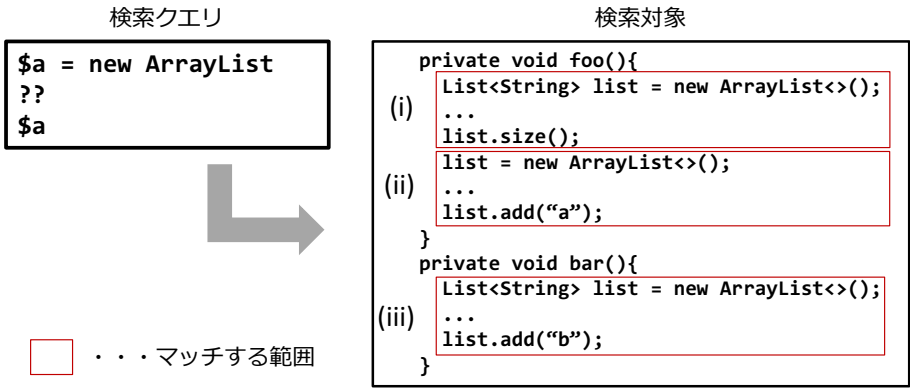

図 2: “??” がマッチする範囲の例．$変数の生存区間を考慮するため，(i), (ii), (iii) の 3 つのコード片にマッチする．

るワイルドカードである．ただし，マッチする範囲は 1 つの文の中に限る．たとえば，“= _.get” という検索クエリは，“=” の後ろに長さ 0 以上のトークン列が続き，そのトークン列で開いた括弧がすべて閉じられた後に “.get” にマッチする．そのため，“= foo.get” や “= getFoo().get” のようなトークン列にマッチする．一方で，“= 0 ; a = foo.get” のような複数の文にまたがる (“;” を含む) トークン列にはマッチしない．このトークンにより，文の中におけるトークン列の差を吸収する検索クエリを記述することが可能である．

“??” – 任意のトークン列にマッチするワイルドカード

長さ 0 以上の任意のトークン列にマッチするワイルドカードである．“??” は $変数のスコープが有効である範囲において最長マッチする．すなわち，検索結果として得られるコード片は “{” より多くの “}” を含まず，$変数への再代入が行われないようなものである．これにより，複数の箇所に出現する同じ名前を持つ変数が，異なるスコープを持つようなコード片を出力から排除することが可能である．たとえば，図 2 では，(i), (ii), (iii) の 3 つのコード片にマッチする．(i), (ii) と (iii) のコード片はそれぞれ異なるメソッドのスコープを持つため，別のコード片としてマッチする．また，(i) と (ii) が別のコード片としてマッチするのは，(ii) の最初の行において $変数にマッチした変数 list への代入が行われている (list の後に “=” が続く) ため，$変数にマッチした変数の生存区間がひと続きでないからである．このようなマッチングの仕様によって，検索クエリに $変数と “??” を記述するだけで自然に変数の生存区間を考慮した検索が可能となる．すなわち，変数のデータフローに注目した検索が可能となる．ただし，検索クエリに $変数が含まれない場合，巨大なトークン列が検索クエリにマッチするのを防ぐため，最短マッチするものとする．

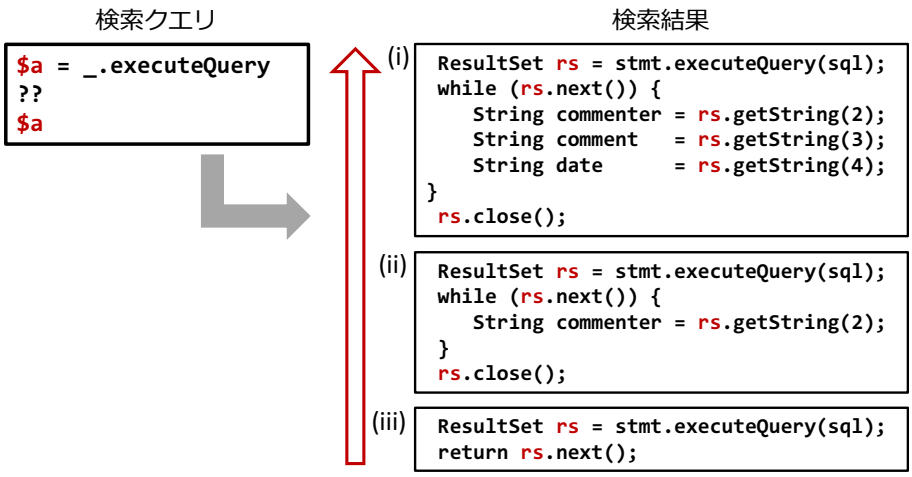

図 3: 検索のランキングの例．赤字の変数が “$a” にマッチした変数である．

3.2 コード例の表示形式

検索クエリにマッチしたコード片は，検索結果のコード例として開発者に提示される．先行研究の中には，開発者にとってより有益なコード例を提示するため，コード例のランキングや整形が行われるものが存在する．提案手法でも同様に，検索結果のランキングとコード例の整形を行う．ただし，これらの処理は，主に検索クエリに $変数が少なくとも 1 つ含まれる場合を主に対象としている．そうでない場合，開発者にとってコード例のどの部分に関心があるかは状況によって異なると考えられるため，これらの処理は行わない．コード例のランキングや整形のための処理が高コストになる場合，ウェブアプリケーションとしての応答時間が大きくなってしまう．そのため提案手法では，検索クエリに含まれる $変数がデータフローを追いかけたい変数であるという性質を利用し，低コストなランキングと整形を実現している．それぞれについて以下に述べる．

まず，検索結果のコード例のランキングについて説明する．検索クエリが $変数を含む場合，検索クエリにマッチしたコード例のうち $変数にマッチした変数を多く含むものの方が有用性が高いと考える．たとえば，図 3 は検索クエリに対し 3 つのコード例 (i), (ii), (iii) が検出された例である．(i) には 検索クエリ中の “$a” にマッチした変数が 6 個，(ii) には 4 個，(iii) には 2 個含まれている．したがって，上から (i), (ii), (iii) という順序で検索結果が表示される．上位のコード例ほど，executeQuery メソッドの戻り値に対する操作についての情報量が大きく，開発者にとって有益であると考えられる．

つぎに，検索結果のコード例の整形について説明する．先行研究において一般的なコード例の整形は，開発者にとって無関係と考えられる部分をコード例から除外するというものである．本研究でも同様に，開発者の関心とは関連性が小さいと考えられる部分をコード

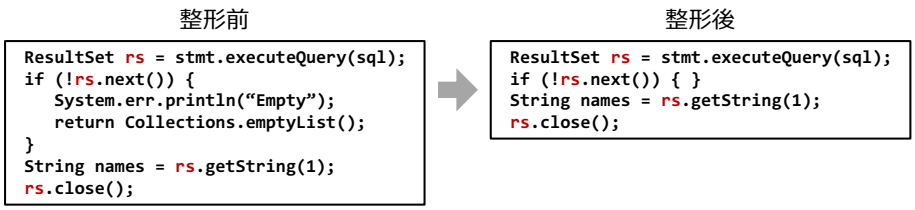

図 4: コード例の整形の例．赤字の変数が $変数にマッチした変数である．

例から省略する．まず，検索クエリにマッチしたコード片から空行とコメント行が除外される．これは，開発者にとって空行やコメント行が， API の理解のために有益な情報を持つことは少ないと考えられるからである．また，検索クエリにマッチしたコード片から $変数にマッチした変数が再帰的に (すなわち入れ子になった子ブロックの中にも) 1 つも含まれないブロックの表示は省略される．このようなブロック中では $変数に対する操作が存在しないうえ，$変数のスコープよりさらに狭いスコープをもつ．そのため，このブロックの中で新たに定義された変数はブロックの外に影響を及ぼすことはなく，除外してもコード例の可読性が損なわれる可能性は低いと考えられる．たとえば，図 4 はコード例の整形の例である．この例では `if` 文のブロックの中に $変数がマッチしたものが存在しないため，整形後はブロックの内部が省略されている．整形後の方が開発者が変数 `rs` のデータフローを追いかけるのに無駄がないコード例となっている．なお，実装したアプリケーションでは，検索結果として得られたコード例から元のソースコードのページへのハイパーリンクを提供することで，省略された情報も必要に応じて参照可能となっている．

4　アルゴリズム

本章では，提案手法のアルゴリズムについて述べる．アルゴリズムは主に以下の 4 つの手順からなる．

- 手順 1．非決定性有限オートマトンの構築
- 手順 2．コードの取得と非決定有限オートマトンへのマッチング
- 手順 3．検索結果の選択
- 手順 4．コード例の整形

それぞれの手順について述べる．

4.1　手順 1．非決定性有限オートマトンの構築

まずはじめに検索クエリのトークン化を行う．トークン化には構文解析ツール ANTLR v4[2] を使用した．ANTLR v4 はトークンの規則を独自の文法により定義することで，その規則に応じたトークン化を行うことが可能なツールである．本手法の検索クエリは，検索対象言語のトークン集合と 3 つの特殊トークンの並びで構成される．特殊トークンをトークンとして認識するため，検索対象のプログラミング言語の構文規則に 3 つの特殊トークンに対応する規則を追加し，検索クエリに対する字句解析器を生成した．具体的には，“??” と “_” はソースコードの前後を考慮することなくトークンとして認識できるため，それぞれに対応する 1 つの構文規則を追加する．すなわち，プログラミング言語の既存の構文規則に手を加える必要はない．一方で， $変数は “$” の後に既存の識別子名が続くようなものをすべて認識する必要があり，プログラミング言語自体の規則に手を加える必要がある．たとえば，既存の識別子名の規則が “Identifier” という名前で定義されている場合，既存の識別子名の規則を “OrdinaryIdentifier” (すなわち既存の “Identifier” に相当)，$変数の規則を “DollarIdentifier” という名前とすると，識別子名がいずれかであるという規則を追加する．たとえば，

```
Identifier       : OrdinaryIdeitifier | DollarIdentifier ;
DollarIdentifier : '$' OrdinaryIdeitifier ;
```

のような規則に書き換える．

つぎに，図 5 のように検索クエリから NFA (非決定性有限オートマトン) を構築する．この手順は正規表現検索の一般的なアルゴリズムに類似している．NFA の入力記号は検索対

[2] http://www.antlr.org/

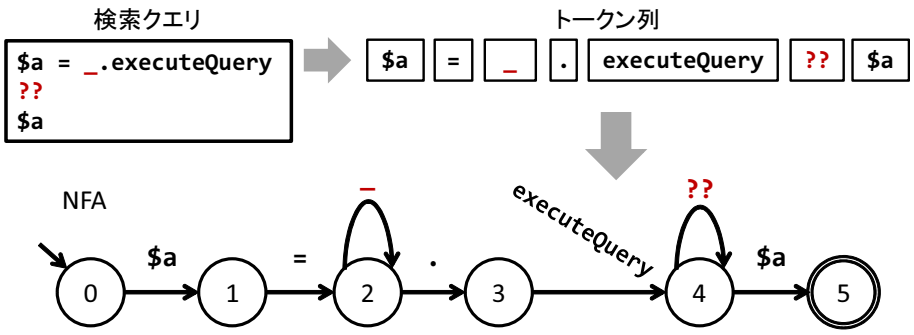

図 5: 検索クエリからの非決定性有限オートマトンの構築

象言語のトークン集合である．NFA の初期状態を生成した後，検索クエリのトークン列を先頭から読み込み，読み込んだトークン t に応じて新たな状態と遷移を追加する．直前に作成した状態を s_p とする．トークン t が “??” または “_” であるとき， s_p の任意の入力に対する遷移先として状態 s_p 自身を設定する．トークン t が “??”, “_” 以外であるとき，新たな状態 s_n を作成し， s_p の入力トークン t に対する遷移先として s_n を追加する．トークン列を末尾まで処理した後，最後に作成した状態を受理状態とすれば NFA の構築が完了する．

4.2 手順 2. コードの取得と非決定有限オートマトンへのマッチング

本手順ではまず初めに，コード検索結果エンジンからソースコードを取得する．本研究で用いるコード検索エンジン searchcode.com では，一般的なメソッド名やクラス名などの単語単位での検索を行うための Web API が存在している．まず，検索クエリに含まれる識別子名をトークン化の段階で認識しておき，その識別子名すべてが含まれるような複数のソースコードを API 経由で取得する．そして，取得したコードを `ANTLR v4` を用いてトークン化する．トークン集合に対し，検索クエリに含まれる識別子名がすべて含まれているかを確認する．たとえば，検索クエリ “`Result $a = _.execute`” から，識別子名 “`Result`”, “`execute`” を抽出しておき，検索対象のトークン列がこれらをすべて含むかを確認する．すべては含まれていない場合，検索クエリにマッチするコード例が検出されることはないため，そのソースコードは NFA へのマッチングをスキップする．このような場合が存在するのは，検索クエリ中に含まれる識別子名がソースコードのコメントとして含まれている場合や，識別子名の文字列の一部として含まれている場合 (たとえば，検索クエリ中の識別子名 “`execute`” にソースコード中の識別子名 “`executeAll`” がマッチする場合など) が存在するからである．一般的なコード検索エンジンはソースコードを文字列として認識し検索を行うため，このような状況が生まれる．

つぎに，ソースコードをトークン化し，NFA へのマッチングを行っていく．本手法で使用する NFA は一般的な NFA と同様，入力に応じてアクティブな状態が遷移するモデルである．検索対象ファイルのトークン列を先頭から NFA へ入力していくが，トークンを入力するたびに NFA の初期状態にアクティブな状態を 1 つ追加する．そして，すべてのトークン列を読み終えるまでに，NFA の受理状態に到達するアクティブな状態の集合を求めると，その集合が出力候補となるコード片の集合に対応する．NFA へのマッチングの例を図 6 に示す．この例では，検索対象のソースコードの 2-3 行目がマッチする．

ただし,「アクティブな状態」は以下の 5 つの情報を保持し，その状況に応じて消滅するルールを持つ．すなわち，検索対象へのマッチングが NFA の能力のみを用いて実現されているというわけではない．また，同一の情報を保持するアクティブな状態同士は区別しないものとする．つまり，アクティブな状態の遷移先に，同一の情報を保持するアクティブな状態がすでに存在する場合，それらのアクティブな状態は 1 つのアクティブな状態として扱われる．これにより，結果として複数の同一内容のコード例が検出されることを防ぐほか，アクティブな状態数が指数的に増加することを抑えることができる．入力トークンに対する遷移先が存在しない場合は，そのアクティブな状態は消滅するものとする．

以下に，アクティブな状態が保持する 5 つのデータと，アクティブな状態を消滅させる条件について説明する．ただし，4, 5, 6 のデータに関してはアクティブな状態を消滅させる条件は存在しない．

1．括弧の対応関係の情報

データ内容と変化するタイミング

ブロックの開始と終了を意味する括弧の情報を積むスタック．$変数のスコープを考慮するために使用する．入力トークンが開き括弧 (‘{’) であるときにプッシュ，閉じ括弧 (‘}’) であるときポップされる．なお，検索クエリ自体に，対応する開き括弧を持たないような閉じ括弧が含まれている場合，閉じ括弧に対応する数の開き括弧をあらかじめ初期状態としてスタックに積んでおく．これにより，括弧の対応が取れていない検索クエリに対する検索を実現している．

消滅を引き起こす条件

このスタックが空の状態でポップが行われる場合，そのアクティブな状態は消滅する．これにより，コード片が独立した複数のブロックにまたがるようなコード例へのマッチングを防いでいる．受理状態においてこのスタックが空でない場合，ブロックが閉じられていないというだけで含まれる変数のスコープに問題があるわけではないため，アクティブな状態は消滅しない．

2. “_” にマッチするコード片の括弧の対応関係の情報

データ内容と変化するタイミング

検索クエリ中の 1 つの “_” へのマッチングを行う， 1. と同様のスタック．“_” へ括弧の釣り合いがとれたトークン列のみをマッチさせるために使用する．アクティブな状態が “_” へのマッチを開始したときに空のスタックとして作成され，“_” へのマッチを終えると消去される．入力トークンが開き括弧 (‘{’, ’(’, ’[’, ’<’) のときにプッシュ，閉じ括弧 (‘}’, ’)’, ’]’, ’>’) のときにポップされる．

消滅を引き起こす条件

“_” へのマッチを終えたとき，このスタックが空でなければそのアクティブな状態が消滅する．また，このスタックが存在するときに，入力トークンが文と文の区切り文字 (‘;’) であれば，アクティブな状態が消滅する．

3. $変数の束縛情報

データ内容と変化するタイミング

検索クエリに含まれる各 $変数の束縛情報．これにより，検索クエリの $変数の条件にマッチしないものを出力から除外している．初期状態は空である．ある状態から $変数 ”$a” の遷移があり，入力トークンが識別子名 “id” であったとき，”$a=id” のように $変数と識別子の束縛情報を追加する．

消滅を引き起こす条件

入力トークンにより $変数 ”$a” の遷移が起こるとき，すでに ”$a” が入力トークンと異なる識別子名によって束縛されている場合，このアクティブな状態は消滅する．

4. $変数へのマッチした回数

データ内容と変化するタイミング

入力トークンが $変数 にマッチした回数．コード例のランキングアルゴリズムに使用する．新たな $変数に束縛が起こったとき，またはすでに束縛されている $変数と矛盾が起こらない識別子トークンが入力されたときに 1 だけ増やす，

5. 開始行番号

データ内容と変化するタイミング

初期状態から遷移を引き起こした入力トークンの行番号．出力するコード片の開始行に一致する．

6. 終了行番号

データ内容と変化するタイミング

受理状態への遷移を引き起こした入力トークンの行番号．出力するコード片の終了行に一致する．

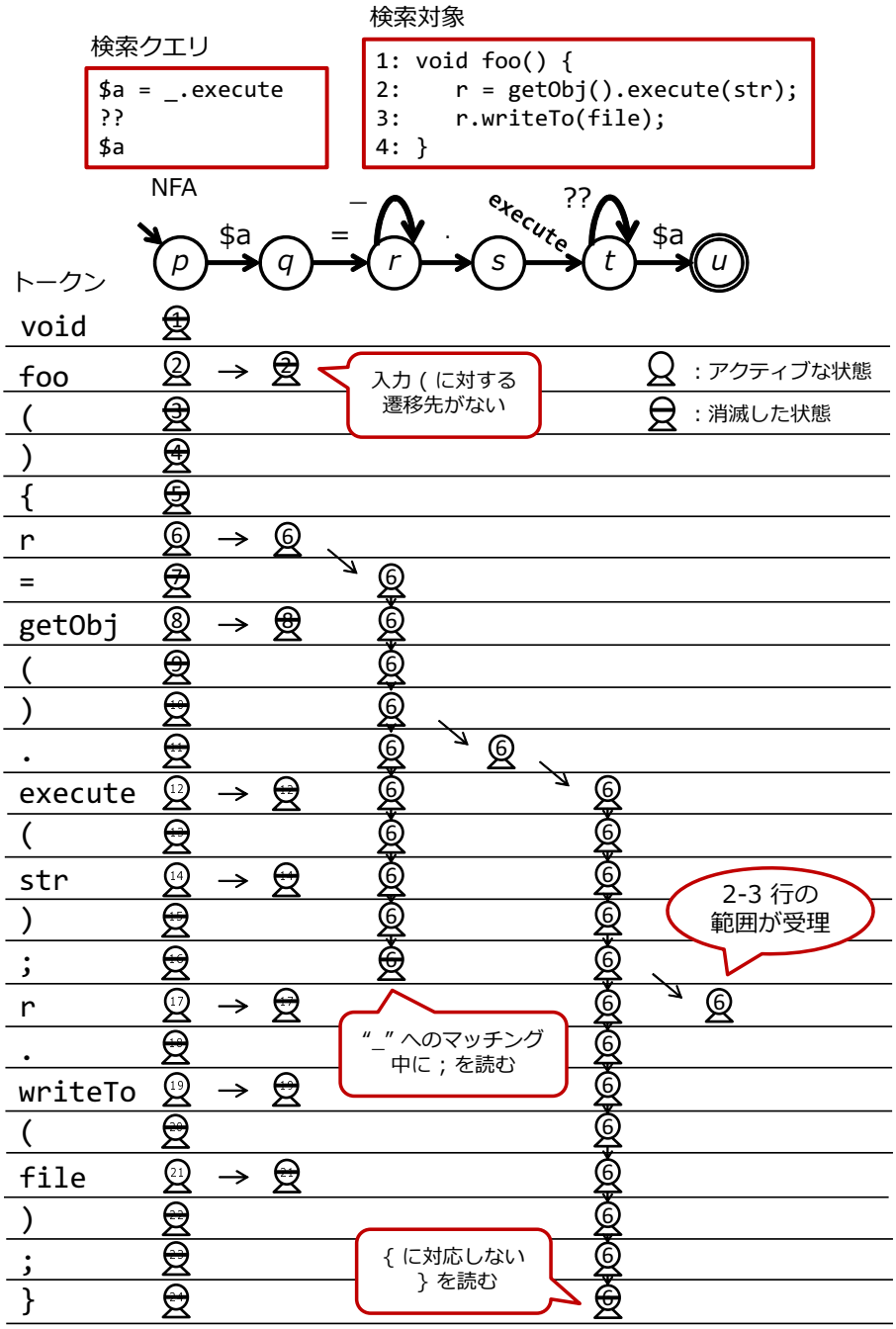

図 6: NFA へのマッチングの例．アクティブな状態に記載されている番号は新たに初期状態へ追加された順序である．

4.3 手順 3. 検索結果の選択

この手順では，手順 2 で受理状態に到達したアクティブな状態集合から検索結果として表示するコード例の選定を行う．基本的には，受理状態に到達したアクティブな状態に対応するコード例を検索結果として表示すればよいが，コード例が $変数の生存区間に対応していない場合や，互いに包含関係にあるような場合が存在する可能性があるため，そのようなコード例を除外する．ここでは，受理状態に到達したすべてのアクティブな状態に対応するコード例を「出力候補」と呼ぶことにする．

まず，出力候補から $変数への再代入が行われているコード例を除外する．たとえば，検索クエリ “$a = execute ?? $a” に対して，

```
1:  r = execute();
2:  r.dump();
3:  r = execute();
4:  r.dump();
```

という出力候補が存在する場合，$変数 “$a” にマッチした変数名 “r” へ 1 行目と 3 行目で代入が行われている．このようなコード例は $変数の生存区間が考慮できていないため，除外する．具体的なアルゴリズムとしては，束縛情報から $変数へマッチした識別子名 “r” を取得し，出力候補のトークン列を先頭から見ていったとき “r”， “=” というトークン並びが 2 度出現した時点で出力候補から除外する．この処理は手順 2. において NFA へのマッチング中にも行うことができるが，アクティブな状態が保持する情報と消滅する条件が複雑化するのを避けるため，出力候補のフィルタリングという形をとっている．

つぎに， $変数の生存区間がより長いものを検索結果として表示するため，コードの行数が重複している出力候補同士の中でもっとも大きいもののみを出力する．そのため，受理状態に到達した 2 つのアクティブな状態 a_1, a_2 の包含関係として，以下の半順序関係 $\subseteq$ を定義する．ただし， $a.start$ はアクティブな状態 a の開始行番号， $a.end$ は a の終了行番号とする．

$$\begin{aligned} a_1 \subseteq a_2 \quad \Leftrightarrow \quad & a_1.start \geq a_2.start \\ \wedge \quad & a_1.end \leq a_2.end \end{aligned}$$

この半順序関係 $\subseteq$ により，受理状態に到達したすべてのアクティブな状態集合は半順序集合となる．この半順序集合における極大元にあたるアクティブな状態が，検索結果として表示するコード例となる．この処理により，より長い $変数の生存区間を持つ，すなわち開発者にとってより有益なコード例を検索結果として表示することができる．たとえば，図 7 は

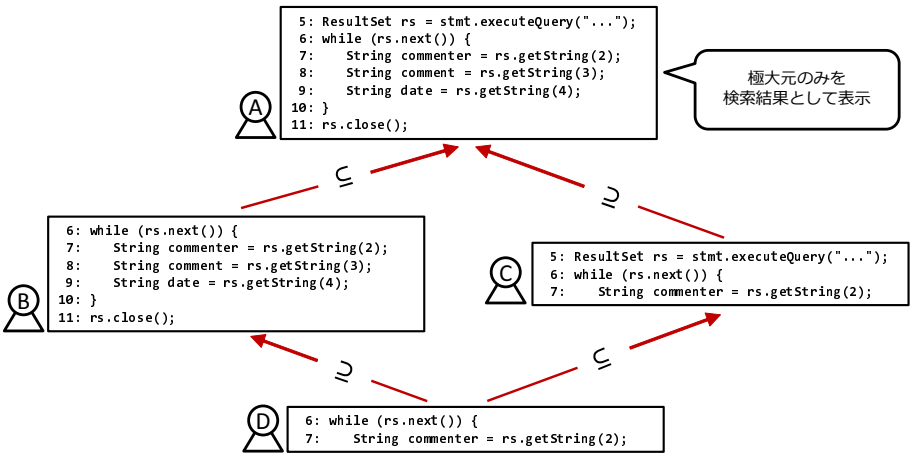

図 7: アクティブ状態集合の $\subseteq$ における半順序集合のハッセ図

半順序集合の例である．この例ではアクティブな状態 A のコード例が極大元であり，検索結果として表示するコード例となる．

4.4 手順 4. コード例のランキングと整形

最後の手順として，検索結果として得られたそれぞれのコード例を開発者にとってより有益であると考えられるものを上から並び替え，より可読性の高いコード例へと整形を行う．準備として，検索結果として得られたコード例が互いに同一である，すなわちタイプ 1 のクローンの関係である場合，1 つのコード例を残して他のコード例は検索結果から除外する．これにより，検索結果が複数の同一コード例により冗長になるのを防いでいる．

まず，検索結果として得られたコード例のランキングによる並び替えを行う．3 章でも述べた通り，検索クエリに $変数が含まれる場合，ランキングアルゴリズムは $変数にマッチした変数の出現回数に基づく．すなわち，アクティブな状態が保持する 4 番目の情報「 $変数へマッチした回数」によって，回数が多いものを上位としてに並べ替えを行う．ただし，実装したウェブアプリケーションでは，検索結果のコード例から元のソースコードへのハイパーリンクが存在する．そのため，検索結果のコード例はファイル単位 (すなわちソースコード単位) でひとまとまりに並んでいる方が望ましいと考えられる．したがって，提案手法では上記の並べ替えをファイル単位で行う．コード例単位のランキングからファイル単位のランキングへの拡張方法はシンプルであり，ファイルごとの $変数へマッチした回数の平均値を求め，その値が大きいものから上位に表示するというものである．その結果，開発者にとって有益であると考えられるコード例がより多く含まれるファイルから上位に表示することが可能となる．

最後に，表示するコード例の整形を行う．3 章の通り，本手順ではコメント行や空行は削

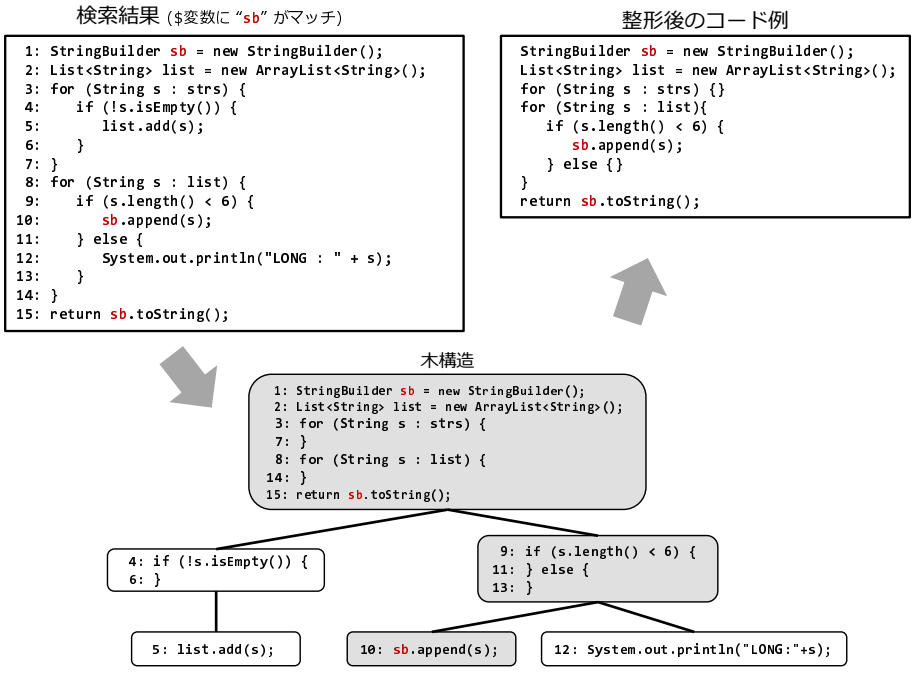

図 8: コード整形のための木構造の例

除し，$変数にマッチした変数が再帰的に存在しないブロックは表示を省略する．ブロックの省略の具体的な処理は以下の通りである．コード例のブロックの入れ子関係は階層構造として捉えることができる．コード例の先頭のトークンを含むブロックを最上位の階層とすると，それに含まれるブロックはひとつ下位の階層と見なすことができる．このように階層を再帰的に定義することで，コード例に含まれるすべてトークンはどこかの階層に属することになる．すなわち，トークン集合をノード，ブロックの入れ子関係をエッジとする木構造として，コード例を表現することが可能である．この木の根を始点として深さ優先で走査する．あるノードが $変数にマッチした識別子名を含んでいる場合，根からそのノードへの経路上に存在するすべてのノードにマークを付ける．走査が終了したとき，マークが付いているノードに含まれるトークン集合が検索結果として表示するコード例となる．一方で，マークが付いていないノードが整形によって表示が省略される部分である．図 8 はコードの整形の例である．この図ではトークン集合を行単位で表現しており，灰色のノードが走査によりチェックが付いたものである．整形後のコード例は StringBuilder のインスタンスの操作を学習するのに無駄が少ないものとなっている．

4.5 計算量

本アルゴリズムでもっとも計算コストが大きいのは手順 2. における，トークン列の NFA へのマッチングの処理である．この処理における計算量について考察する．NFA へのマッチングにおいて，計算量はアクティブな状態の数に比例する．アクティブな状態の数が最大となるのは NFA 中の全状態が自己への遷移を持つときである．この時のアクティブな状態数を見積もるため，NFA の状態数を M, 入力トークン数を N とする．$変数は高々数個であると仮定しその影響を無視すると，アクティブな状態は，マッチを開始したトークンの位置（開始行番号），入力済みのトークン列，その時点での NFA の状態の 3 つ組によって一意に定まる．したがって，$n(1 \leq n \leq N)$ 番目のトークンを読んだ時に生成された（初期状態としてマッチを開始した）アクティブな状態は，その後 $(N - n)$ 回のトークンの入力によって数が増加していくが，その総数はアクティブな状態が NFA の全 M 状態に出現する場合の $M \times (N - n)$ より大きくなることはない．つまり， n 番目に生成されたアクティブ状態がもたらす計算量は $O(MN)$ で抑えられる．これがすべての n について同様に言えるため，計算量は $O(MN^2)$ となる．ただし， “??” にマッチする範囲はマッチング開始時のブロックに限り，また “_” にマッチする範囲は “_” へのマッチング開始時の文の内部に限る．そのため， N は高々1 ブロック内や 1 文内におけるトークン列の長さであり，実際の計算コストは小さく抑えられる．

5　評価実験

本研究では，提案手法の有用性を考察するため，評価実験を行った．評価する項目として，以下の 3 つのリサーチクエスチョンを設定した．

RQ1. 提案手法は API 理解支援として有用か

提案手法は，変数のデータフローを考慮することで，より有益な API の利用コード例を提示することを目的としている．既存のコード検索エンジンと比較して，提案手法は API 理解のために有用であるかを調査する．

RQ2. 検索クエリの記述は容易か

提案手法は，変数のデータフローを指定するための独自の検索クエリを定義している．提案手法が実用的であることを確認するため，検索クエリの記述が容易であるかどうかを調査する．

RQ3. 検索にかかる時間はどれくらいか

提案手法では，検索クエリをもとにコード検索エンジンからソースコードを取得し，その場で提示するコード例の選定を行っている．選定の時間が大きい場合，ウェブアプリケーションとしての応答時間が長くなり，手法の実用性が低くなると言える．そのため，提案手法が現実的な応答時間で検索を終えることができるかを調査する．

これらのリサーチクエスチョンを考察するため，本研究では，API の理解をともなうタスクを設定し，被験者実験を行った．本章では，実験の設計とその結果について述べる．

5.1　実験の設計

本研究では，3 つののリサーチクエスチョンを考察するため被験者実験を行った．被験者は，研究室に所属する修士課程の学生 8 人であり，M1 が 4 人, M2 が 4 人という構成である．被験者は主に研究に利用するツール等の開発のため `Java` によるプログラミング作業を日常的に行っている．そのため，実験のタスクに使用するプログラミング言語は `Java` とした．

`RQ1` に答えるため，実験では提案手法と既存手法の比較を行った．すなわち，対照実験の形式をとる．比較対象となる既存手法として，コード検索エンジン serachcode.com [7] を選択した．コード検索エンジンの利用は API 理解のためのツールとして一般的であり [5] , searchcode.com はキーワード (メソッド名やクラス名など) による検索など，コード検索エンジンとして標準的なものである．また，提案手法のソースコードの取得元であるため，検索対象となるソースコード集合に差がなく，対照実験の比較対象として適切であると考えられる．

表 1: 実験のグループ分け． T はタスク，S は既存手法，P は提案手法を意味する．

	グループ A	グループ B	グループ C	グループ D
セッション 1	T1 - P	T1 - S	T2 - P	T2 - S
セッション 2	T2 - S	T2 - P	T1 - S	T1 - P

RQ1 に答えるため，実験に使用するタスクは API の理解を伴う必要がある．恣意的なタスク設定を避けるため，実験に使用するタスクは Laura ら [14] の実験に用いられた 2 つの実験タスクを以下の変更を加えて使用した．

1. 元のタスクは英語であるが，今回は被験者が全員日本人であるためタスクの和訳を行った．

2. 先行研究では Google 検索等の検索エンジンの使用等にまったく制限がないが，本実験では検索エンジンにおいて特定の API 名を検索しコード例を閲覧する行為は禁止とした．これにより，提案手法または比較対象を利用する代わりに外部のコード例を見ることで API の利用方法を学習することを防ぎ，コード検索ツールとしての性能の差がより結果に反映されやすいようにした．

3. 先行研究ではタスクにおいて使用する API 名のみが記載されていたが，本実験ではタスクにおいて使用する API のクラス名・インターフェース名をヒントとして記載した．これにより「どの API を利用すべきか」というコード例検索に関係のない項目の調査に作業時間が割かれるコストを抑えた．

4. 先行研究では制限時間が 1 つのタスクにつき 60 分であったが，本実験では上記のヒントの影響を考慮して制限時間を 40 分とした．

なお，2 つのタスクは本論文の付録に記載している．

2 つのタスクをこなすため，作業時間を 40 分間の 2 つのセッションに分割した．先行研究と同様，被験者を表 1 の A∼D の各グループに 2 人ずつ割り当てた．表中における T1, T2 はそれぞれタスク 1, タスク 2 に対応しており，P は提案手法，S は既存手法 (searchcode.com) に対応している．たとえば，グループ A は，セッション 1 においてタスク 1 を提案手法を用いて行い，セッション 2 においてタスク 2 を既存手法を用いて行うということを意味する．

各タスクに取り組む前，使用するコード検索エンジンの使用方法を学習する時間を 3 分間設けた．提案手法のウェブアプリケーションには，特殊トークンのマッチングの規則や検索クエリの例を記載しており，それらを確認する形で被験者は学習を行った．また，既存手法は標準的な単語単位でのコード検索エンジンであり，検索クエリを入力するフォームと検索結果の見方の確認を行った．

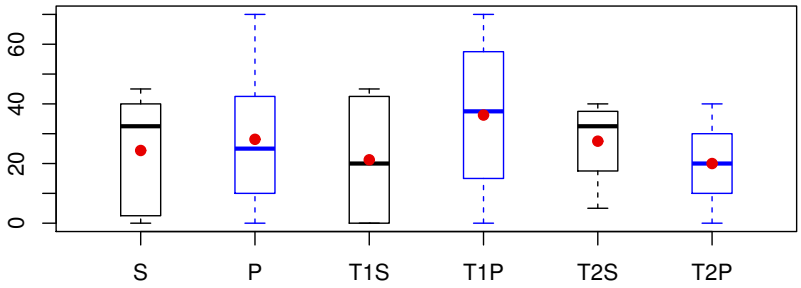

図 9: 各タスクのスコアの箱ひげ図．左端の S, P はタスク 1, 2 の合計である．また，赤色の点は平均を示している．

各セッションが終了するごとに被験者が作成したプログラムの収集を行った．タスクにおける各サブタスクの配点は先行研究に従うものとする．ただし，より詳細な採点を行うため，各サブタスクに部分点を導入した．部分点は，そのタスクの正解コードにおけるすべてのメソッド呼び出しのうち，いくつのメソッド呼び出しを実装できているかに基づいて採点する．実装方法が一通りでない場合は，正解コードのメソッド呼び出しに相当する処理が実装できてるかという観点で採点を行った．２つのセッションが終了後，アンケートによる調査を行った．質問項目は以下の通りである．

1. 「提案手法においてクエリの記述は難しいと感じましたか (4 段階で評価)？また，その理由を教えてください.」
2. 「タスクで使用した API について，このタスクを行うより前に使用したことがあるものがあれば選択してください.」

また， RQ3 に答えるため，送信された検索クエリとその応答時間をデータログを残す形で収集した．

5.2 実験結果と考察

この節では，3 つのリサーチクエスチョンについて実験結果に基づいて考察する．アンケートの 2 つ目の質問に対する回答から，タスクに使用した API を以前に使ったことがある被験者は存在しないことが分かった．

5.2.1 RQ1. 提案手法は API 理解支援として有用か

実施した実験の各タスクのスコアを，図 9 の箱ひげ図として示す．この図は，横軸がタスクと使用したコード検索手法の組み合わせ (T がタスク，S が既存手法， P が提案手法を

ERAService.java

```
1:   HttpClient httpClient = new DefaultHttpClient();
2:   HttpGet httpget = new HttpGet(service + id);
3:   httpget.setHeader("Accept", "application/json");
4:   HttpResponse response = httpClient.execute(httpget);
```

AnsServices.java

```
1:   HttpClient httpClient = new DefaultHttpClient();
2:   HttpResponse response = httpClient.execute(httpPost);
```

図 10: 検索クエリ “HttpClient $a = ?? $a.execute” の検索結果例

意味する)，縦軸がその得点率である．また，左端の S, P は 2 つのタスクの結果を合計したものであり，赤色の点はそれぞれにおける平均値を示している．この結果より，タスク 1 において (図中の T1S, T1P を参照)，平均値・中央値ともに提案手法の方がよい傾向が見られる．一方で，タスク 2 では (図中の T2S, T2P を参照)，その傾向は見られず，むしろ既存手法の方がよいという傾向が得られた．2 つのタスクを合わせた結果では，平均値は提案手法の方がよく，中央値は既存手法の方がよいという結果になった．

これらの結果について考察する．

まず，タスク 1 において提案手法の方がよい結果が出たのは，このタスクが HttpClient インターフェースの実装クラスを特定するところから始まることが影響していると考えられる．既存手法を用いて HttpClient インターフェースの実装クラスを探そうとすると，API のドキュメントに記載されている複数の実装クラスを順番に見ていくか，コード検索エンジンで「HttpClient」というキーワードで検索し，周辺のコードを見るかの選択肢に限られる．一方で，提案手法において記述された検索クエリの中には以下のようなものが見られた．

```
HttpClient $a =
??
$a.execute
```

この検索クエリは，HttpClient インターフェースの execute メソッドの呼び出しがある箇所を “$a.execute” と表現し，そのレシーバとなるオブジェクトの生成を “HttpClient $a =” と表現したものであると推測される．どちらも共通の $変数 “$a” を指定することで，これにマッチする変数のデータフローを考慮した検索が可能となっている．実験終了後，この検索クエリを用いて検索すると，検索クエリを送信してから約 5 秒程度で検索結果が返され，上から 2 つ目と 3 つ目のファイルのコード例として，図 10 が見られた．赤字の変数が $変数にマッチした変数である．これらのコード例の 1 行目はともに同じであり，

```
HttpClient httpClient = new DefaultHttpClient();
```

CustomHttpClient.java

```
1:   HttpClient client = getHttpClient(context);
2:   HttpResponse response = client.execute(httpRequest);
3:   if ($b.getStatusLine().getStatusCode() != HttpStatus.SC_OK) { }
4:   HttpEntity resEntity = response.getEntity();
```

HttpClient.java

```
1:   HttpClient clinet = new DefaultHttpClient();
2:   HttpPost post = new HttpPost(URL);
3:   List<NameValuePair> nvList = new ArrayList<NameValuePair>();
4:   BasicNameValuePair bnvp = new BasicNameValuePair("json", data);
5:   nvList.add(bnvp);
6:   post.setEntity(new UrlEncodedFormEntity(nvList));
7:   HttpResponse resp = clinet.execute(post);
8:   InputStream is = resp.getEntity().getContent();
```

図 11: “HttpClient $a = ?? HttpResponse $b = $a.execute ?? $b.” の検索結果例

という代入文によりインスタンスが生成されていることが読み取れる．実際，このコード例に習って実装すれば，タスクを完了することが可能である．

上記の検索クエリはその後，次のような検索クエリへと変化したことがデータログから読み取れた．

```
HttpClient $a =
??
HttpResponse $b = $a.execute
??
$b.
```

これは先ほどの execute メソッドの戻り値 (HttpResponse という型をもつ) を，“$b” に代入し，その後に続く “$b” へのメソッド呼び出しを ”$b.” と表現しているものであると考える．これは， HttpResponse の戻り値から HTTP リクエストの結果を取り出す操作を検索する目的であると推測される．

その結果，8 秒程度で検索結果が表示され，上から 2 番目と 5 番目に図 11 のコード例が見られた．HttpResponse に対するメソッド呼び出しとして，”response.getEntity()” や “resp.getEntity().getContent()” などが存在することが分かる．これらコード例もタスクを完了するために有益なものであると言える．以上のような検索クエリを記述できた場合は，提案手法の方がより有効に API への理解の支援を行うことができたと考えられる．

タスク 2 については，提案手法の有用性が見られなかった．その原因として，タスク 2 は主にユーティリティメソッドを使用するサブタスクから構成されるため，どのようにメソッドを使用するかより，どのメソッドを使用すべきか考慮することに多くの時間が割かれたことが考えられる．また，ユーティリティメソッドの多くが static メソッドであるため，オ

ブジェクトの状態等を考える必要がなく，使い方はそれほど難しくはない．そのため，コード例による API の理解支援を必要とせず，提案手法と既存手法の差がほとんど生まれなかったと考えられる．

本研究での各タスクの得点率は先行研究に比べ全体的に低かった．これは先行研究の被験者が実際の開発者であるのに対し，本研究の被験者は研究室の修士課程の学生であることが原因であると考えられる．本研究の被験者の学生は普段からプログラミング作業を行っているものの，その多くが研究のためツール開発であり，慣れない API を使用するようなプログラミングをしているというわけではない．研究のためのアルゴリズムを実装するよりも，企業等でのソフトウェア開発の方が API の利用が盛んに行われるため，実際の開発者に大きなアドバンテージがあったと考えられる．実験後にヒアリング調査を行ったところ，多くの学生がインタフェースの扱いに不慣れであることが判明した．たとえば，タスク 1 では `HttpClient` インターフェースの実装クラスのインスタンスを取得する必要があるが，

```
HttpClient client = new HttpClient();
```

のように，インタフェースを `new` するような操作が見られたり，`HttpClient` インターフェースを実装するクラスを自作しようとしているコードが見られた．より手法を正確に評価するためには，より経験のある開発者を被験者とした実験を行う必要があると考えられる．

`RQ1` の調査結果は以下のようにまとめられる．

- インタフェースの実装クラスの特定など，API の使用方法が自明でないとき提案手法の方がより有効にはたらく例が見られた．
- 一方で，提案手法は既存手法より API 理解支援技術として常に優れていると主張できる結果は得られなかった．
- より経験のある開発者を被験者とした追加実験の検討が必要であると考えられる．

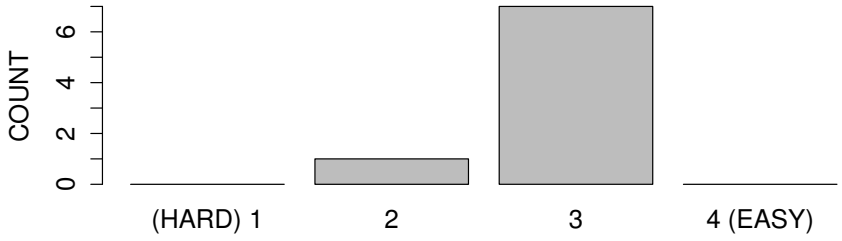

図 12: 検索クエリの記述についてのアンケート結果

5.2.2 RQ2. 検索クエリの記述は容易か

タスク終了後，アンケートにより提案手法の検索クエリの記述は容易かを 4 段階で評価してもらった．その結果は図 12 となった．横軸がその評価値であり,「1」ほど検索クエリの記述が困難,「4」ほど容易であることを意味する．結果として，8 人の被験者のうち 7 人が「3」，つまりどちらかというと容易であると回答した．実験後のアンケートによる調査では，提案手法の検索クエリに対し「最初は検索クエリの記述の仕方が分からなかったが，使っているうちに理解してきた」という意見が多かった．また,「検索に慣れてくると欲しいコード例を検索できて便利だと思った」という意見も見られた．一方で，検索クエリの記述が難しい理由として「一般的な検索エンジンのようにキーワードをスペースで区切って並べるような検索をしても結果が得られないので戸惑った」というものがあった．この意見のとおり，提案手法は，検索クエリにより検索したいものをある程度厳密に指定する必要があるという点で従来手法と異なる．たとえば，”foo”, “bar” という 2 つのキーワードを含むコード例を検索したいとき，検索クエリを “foo bar” のようにスペース区切りで記述すると，“foo” という識別子名の直後に “bar” という識別子名が続くようなコード例のみが検出されることになる．このような検索結果がコード検索の意図とは異なることは明らかである．検索クエリの仕様を，提案手法を初めて利用する人にいかに理解してもらうかは今後の課題と言える．

RQ2 の調査結果は以下のようにまとめられる．

- 検索クエリの記述は 40 分間のタスクの中で理解できる程度の比較的容易なものであることが確認できた．
- 一方で，従来の検索エンジンのような単語単位での検索ではないという点を，より分かりやすく理解してもらう方法を検討する必要がある．

表 2: 提案手法の検索に要する時間

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
時間 (秒)	2.3	3.7	4.3	5.0	5.0	25.4

5.2.3 RQ3. 検索にかかる時間はどれくらいか

タスク中において検索クエリを送信してから検索結果を表示するまでに要した時間を表 2 にまとめる．これらの時間は，マッチング対象となるソースコードを searchcode.com からダウンロードしてくる時間も含まれたものである．この時間は多くの場合で 1 ～ 2 秒であることを確認している．タスクの中の検索結果として，合計で 2448 個のコード例を検索結果として表示した．ただし，この表は検索結果を少なくとも 1 つ表示した場合の検索時間のみをまとめたものである．これは，検索クエリの仕様を理解していないことにより，検索結果が 1 つも表示されないような検索が見られたからである．たとえば，先に述べたように “`foo bar`” のような検索クエリを記述した場合，2 つの識別子名が連続して出現するようなコード例のみを検索結果として表示する．しかしながら，このようなコード例は一般的に存在しないと考えられるため，検索に時間を要する一方で検索結果は表示されない．検索クエリを正しく理解した上での検索に要する時間を評価することに意味があると考え，このような検索の影響は除外した．また，提案手法のウェブアプリケーションの中には検索クエリの例が記載されており，タスク中にそれらの検索クエリが送信されている場合があるため，送信されたすべての検索クエリがタスクの内容に直接関係があるとは限らない．

結果として，中央値で 4.3 秒，平均値で 5.0 秒の検索時間を要することが分かった．また，1 つのコード例を提示するのに平均値で約 0.4 秒の時間を要するという結果となった．この数値はシンプルなコード検索ツールを除く既存の API 理解支援の技術に比べると桁違いに高速である．検索結果を 1 秒以内で表示するようなコード検索エンジンに比べると劣るが，一般的な開発において十分実用的な範囲であると考えている．また，コード例検索においてインタラクティブな検索は一般的であることが知られている [5] が，この要望にも答えることができると考えている．一方で，10 個のコード例を提示するのに検索時間が 25 秒を超える場合が存在することが分かった．このような検索では，検索クエリ中の `$`変数や “`??`” にマッチするものの自由度が非常に高い形で用いられることで，出力候補となるコード例が多数存在している状態であると考えられる．提案手法のアルゴリズムではコード例の包含関係による選定を NFA へのマッチング後に行っている．この処理を NFA へのマッチング中に行うことで，アクティブ状態な状態数が極端に増加することを抑えることができ，より高速にマッチングを終えることが可能であると考えている．ただし，提案手法は検索対象となるソースコードを既存のコード検索エンジンから取得しており，そのダウンロードに要する時

間は 1 ～ 2 秒程度であることが分かっている．このような方式をとっている以上，既存のコード検索エンジンのパフォーマンスが提案手法のパフォーマンスの上限となっている．

RQ3 の調査結果は以下のようにまとめられる．

- 検索に要する時間は平均的に 4 ～ 5 秒であり，実用的な範囲内に収まっていると考えられる．
- 一方で検索に数十秒を要する場合が存在することが明らかとなったが，マッチングアルゴリズムの改善によりある程度の短縮が可能であると考えている．

6　まとめと今後の展望

本研究では API 理解を支援するための技術として，変数のデータフローを指定することによるコード例検索手法を提案した．検索対象となるソースコードは既存のコード検索エンジンから取得することで，幅広い API の検索に対応している．また，有限オートマトンをベースとした低コストなマッチングアルゴリズムにより高速な検索が可能となり，ウェブアプリケーションとしての実装を実現している．評価実験を行った結果，提案手法が API の理解のために有効な場合があることを確認した．また，独自の検索クエリの記述は比較的容易であり，検索にかかる時間は実用的な範囲であることを確認した．

今後の課題としては，より高速な検索のためマッチングアルゴリズムを改良することや，検索クエリの記述方法を分かりやすく伝える方法を検討することが挙げられる．また，実装したウェブアプリケーションを外部に公開し，実際の開発者からのフィードバックを受け取ることや，より質の良い実験データを収集することが今後の目標である．

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，本研究のみならず研究室に配属されてからの 3 年間，さまざまな場面でお世話になりました．自由な雰囲気の中で伸び伸びと研究ができたおかげで，楽しく有意義な研究生活を送ることができました．井上研究室での経験を生かして，今後も尽力していきたいと思います．今まで本当にありがとうございました．

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．本研究のみならず，先生の鋭いご指摘はいつも大変勉強になりました．今までありがとうございました．心より感謝申し上げます．

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，技術的なことから論文の書き方まで，本当に細かく熱心な指導していただきました．3 年間の指導を通して，以前には想像もしていなかったほどさまざまな力を付けることができたと思います．今まで本当にお世話になりました．深く感謝申し上げます．

最後に，その他様々な御指導，御助言等を頂い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感謝いたします．

付録

付録として，次のページから評価実験に使用した 2 つの実験タスクの詳細とその解答コード例を示す．実際のタスクには，使用するデータ一覧の中に自明なもの (プログラム中で作成するファイルやディレクトリなど) が含まれていたため，その表示を省略している．

実験タスク 1

このタスクで作成するプログラムには，以下のデータを使用します．

- PDF ドキュメントの URL．今回は，
  “http://sel.ist.osaka-u.ac.jp/lab-db/betuzuri/archive/1068/1068.pdf”
  を利用してください．

以下の 3 つの処理を行う Java プログラムを作成してください．ただし，例外の処理については特に指定しません．

1. PDF ファイルを指定された URL からダウンロードし，output ディレクトリ内に保存してください．保存ファイル名は問いません．
2. PDF ファイルからメタデータ [1] を抽出し，コンソールに **PDF が作成された日付** を表示してください．表示のフォーマットは問いません．
3. PDF ファイルからテキスト内容を抽出し，コンソールに表示してください．表示のフォーマットは問いません．

使用ライブラリの制約

上記のタスクを行うときは，以下の使用ライブラリの制約を守ってください．

1. PDF ファイルをダウンロードするときは Apache HttpClient ライブラリを使用してください．また，HttpClient インターフェースは， HTTP リクエスト実現するインターフェースです．
2. PDF からメタデータとテキスト内容を抽出するときは Apache Tika ライブラリを使用してください．また，PDFParser クラスは PDF データをパースするためのクラスです．

使用する **API** のドキュメントへのリンク集

- Apache HttpClient 4.5.2 API
- Apache Tika 1.14 API
- Java(tm) Platform, Standard Edition 8 API 仕様

[1] PDF のメタデータには PDF の作者や保存されているデータ，作成にあたって使用されたツールなどの情報が含まれています．

実験タスク 1 解答コード例 (main 部のみ)

```
public static void main(String[] args) throws Exception {
  String outputFile = "output/saved.pdf";
  String url        = "http://sel.ist.osaka-u.ac.jp/lab-db/"
                        + "betuzuri/archive/1068/1068.pdf";

  HttpClient     client   = HttpClients.createDefault();
  HttpUriRequest httpGet  = new HttpGet(url);
  HttpResponse   response = client.execute(httpGet);
  HttpEntity     entity   = response.getEntity();
  try(OutputStream out = new FileOutputStream(outputFile)) {
    entity.writeTo(out);
  } catch (IOException e) { }

  PDFParser          parser   = new PDFParser();
  InputStream        stream   = new FileInputStream(new File(outputFile));
  BodyContentHandler handler  = new BodyContentHandler();
  Metadata           metadata = new Metadata();
  ParseContext       context  = new ParseContext();
  parser.parse(stream, handler, metadata, context);
  System.out.println(metadata.get(TikaCoreProperties.CREATED));
  System.out.println(handler.toString());
}
```

実験タスク 2

このタスクで作成するプログラムには，以下のデータを使用します．

- Task2_TargetDirectory ディレクトリ (Java プロジェクト中に含まれています)．このディレクトリには，OpenFright の空港データベースの csv データを含む，さまざまな拡張子のファイルとサブディレクトリが入っています．

以下の 5 つの処理を行う Java プログラムを作成してください．ただし，例外の処理については特に指定しません．

1. 出力ディレクトリ outputDirectory と出力ファイル output-file.csv を (プログラムにより) 作成してください．作成するパスは問いません．
2. Task2_TargetDirectory ディレクトリ以下に含まれるすべての csv データを，データ構造 (たとえば File オブジェクトのリストなど) として読み込んでください．
3. 読み込んだ csv データのすべての行がちょうど 11 個の項目を持つことを確かめてください．以下では，このようなファイルを「正しい」ファイルと呼ぶことにします．ただし，csv データにおいては項目と項目は ',' によって区切られますが，" " で挟まれた ',' は文字列の一部であることに注意してください．
4. すべての「正しい」ファイルを outputDirectory にコピーしてください．
5. すべての「正しい」ファイルの先頭 100 行のみを，出力ファイル output-file.csv に書き込んでください．

使用するライブラリの制約

上記のタスクを行うときは，以下の使用するライブラリの制約を守ってください．

1. 処理 2, 4, 5 を実装するときは，Apache Commons IO ライブラリを使用してください．また，FileUtils クラスはファイル操作のためのユーティリティクラスです．
2. 処理 3 を実装するときは，Apache Commons Lang ライブラリを使用してください．また，StrTokenizer クラスは区切り文字に基づいて文字列を分割するためのクラスです．

使用する API のドキュメントのリンク集

- Apache Commons IO 2.5 API
- Apache Commons Lang 3.5 API
- Java(tm) Platform, Standard Edition 8 API 仕様

実験タスク **2** 解答コード例 (main 部のみ)

```
public static void main(String[] args) throws Exception {
   File outDir = new File("outputDirectory");
   outDir.mkdir();
   File outFile = new File("output-file.csv");
   outFile.createNewFile();

   File     inDir      = new File("Task2_TargetDirectory");
   String[] extensions = new String[]{"csv"};
   Collection<File> files = FileUtils.listFiles(inDir, extensions, true);

   for (File f : files) {
      List<String> lines = FileUtils.readLines(f, "UTF-8");
      boolean isCorrect = true;
      for (String line : lines) {
         StrTokenizer tokenizer = StrTokenizer.getCSVInstance(line);
         if (tokenizer.size() != 11) {
            isCorrect = false;
            break;
         }
      }
      if (isCorrect) {
         FileUtils.copyFileToDirectory(f, outDir);
         List<String> subList = lines.subList(0, 100);
         FileUtils.writeLines(outFile, subList, true);
      }
   }
}
```

参考文献

[1] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009.

[2] Martin P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[3] Martin P. Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[4] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Eng. Methodol.*, 21(1):4:1–4:25, 2011.

[5] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How developers search for code: A case study. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 191–201, New York, USA, 2015.

[6] Krugle. `http://opensearch.krugle.org`.

[7] searchcode.com. `https://searchcode.com`.

[8] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, pages 117–125, 2005.

[9] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–61, 2005.

[10] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 413–430, 2006.

[11] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213, 2007.

[12] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, Held As Part of the Joint European Conferences on Theory and Practice of Software*, pages 385–400, 2009.

[13] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 319–328, 2013.

[14] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining multi-level api usage patterns. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 23–32, 2015.

[15] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering*, pages 880–890, 2015.

[16] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[17] 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*, 74:470–495, 2009.

[18] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., 1st edition, 2009.

[19] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.