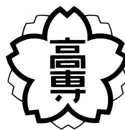

卒 業 研 究 報 告 書

令和7年度

研究題目

Path Contextを用いた
プログラム理解時の視線パターン分析

指導教員　上 野 秀 剛　准教授

氏　　名　升 岡 瑞 葉

令和8年2月27日　提出

奈 良 工 業 高 等 専 門 学 校　情 報 工 学 科

Path Contextを用いた
プログラム理解時の視線パターン分析

上野研究室　升岡　瑞葉

プログラム理解は，人がプログラムを読む際の理解過程を明らかにすることを目的とした研究分野であり，ソフトウェア開発やプログラミング教育において有用な知見を提供してきた．プログラム理解時の視線を分析する研究では，ディスプレイに対する視線を計測する視線計測装置を用いることが一般的である．従来研究では，ディスプレイ座標やソースコード上の行・列番号に基づいて視線位置を可視化し，その結果をもとに分析が行われてきた．さらに近年では，座標単位で取得された視線移動を構文木上のノード間の遷移に変換する手法が提案され，構文要素ごとの注視時間とプログラム理解との関係が示唆されている．しかし，視線遷移を構文要素間の移動として扱うにとどまり，構文木の構造に基づく理解単位の大きさ（粒度）については十分に検討されていない．本研究は，プログラム理解時に読み手が注目しているプログラム構造の粒度である興味範囲を定量化することを目的とする．先行研究の手法を適用し，視線移動を構文木上のノード間の遷移に変換する．連続する2つの視線に対応する構文木上の2ノード間の経路であるPath Contextから，Path Contextを構成する2ノードの共通祖先のうち最も深いノードである最小共通祖先を抽出する．さらに，この最小共通祖先の深さを興味範囲の指標とする手法を提案する．最小共通祖先の深さが浅い場合は大きな構造単位間を移動している状態を，深い場合はより細かな構造内を移動している状態を表すと考えられる．本研究では，RQ1：連続する2つの視線から最小共通祖先を自動的に抽出できるのか?，RQ2：最小共通祖先の時系列変化から読み手の興味範囲の変化を識別できるのか?という2つの問いを設定した．提案手法の有効性を検証するために，最小共通祖先の深さの変化がどのような理解行動を反映しているのか，著者が目視で読み取る．具体的な対象として，構造が単純なプログラムと複雑なプログラムを対象としたタスクにおいて，正解者の視線データを選択した．分析の結果，RQ1については，先行研究の提案手法で求めた構文要素からPath Contextを算出することで，連続する2つの視線から最小共通祖先を自動的に抽出できることを確認した．また，RQ2については，構造が単純なプログラムに対する最小共通祖先の深さの時系列変化を見ると，被験者02は深さ4〜16の範囲で視線遷移をして，クラス構造からwhile文内部まで粒度を細かく変化させて読んでいた．それに対し，被験者12は深さ1〜12の浅い遷移が多く，より粗い粒度で読んでいたことが確認された．複雑なプログラムでは被験者03は粗い粒度から細かい粒度さらに再び粗い粒度へと視線の着目単位が変化しており，最小共通祖先の時系列変化から読み手の興味範囲の変化を識別できた．本研究の今後の課題は，最小共通祖先の深さのみでは読み手の興味内容を十分に特徴づけられない場合があることが確認された．このため，今後は最小共通祖先の深さに加えてノード種別やPath Contextの長さも考慮することで，読み手の理解過程をより詳細に捉えられる手法への改善が考えられる．

目次

1 はじめに **2**

2 関連研究 **4**

3 準備 **5**

3.1 視線計測 5

3.2 視線移動の構文木へのマッピング手法 5

3.3 Path Context 6

4 提案 **10**

5 実験 **14**

5.1 視線データ 14

5.2 タスク 14

5.3 分析 15

6 結果と考察 **17**

6.1 最小共通祖先の自動抽出 17

6.2 時系列変化による興味範囲識別 19

6.2.1 Task1 19

6.2.2 Task10 24

7 おわりに **28**

参考文献 **30**

1　はじめに

プログラム理解は人がプログラムを読む際の理解過程を明らかにすることを目的とした研究分野であり，ソフトウェア開発の現場で有用な知見を提供してきた．例えば，データ構造やデータ操作に対する理解が深いほど，それらに起因するバグの発見率が高くなることが報告されている．そのため，プログラム理解はバグ発見における必須スキルであることが示唆されている[1]．また，プログラム理解の知見を得ることはプログラミング教育においても重要である．近年の視線計測を用いた研究では，成績優秀者とそうでない者は視線パターンが異なることが示されており[2, 3]，学習者のつまずきを把握して支援に役立てることが期待される．

プログラム理解時の視線を分析する研究では，ディスプレイに対する視線を計測する視線計測装置を用いることが一般的である．従来研究では，ディスプレイ座標やソースコード上の行・列番号に基づいて視線位置をプロットし，その可視化結果をもとに分析が行われてきた．しかし，可視化結果に基づいた分析において，視線は停留点としてのみ可視化され，それらがどの構文要素に属するかという構造的情報が付与されず，視線が注がれた箇所がif文の条件式や関数呼び出しなど，プログラム構造上のどこに対応するのか一意に判断できない．そのため，視線移動が同一の構文単位内で生じたものなのか，あるいは異なる制御構造や関数間を跨いだものなのか，プログラム構造に基づいて解釈することが難しく，人間が目視で視線移動を確認し，対応する構文構造を判断する必要がある．

近年の研究では，Yoshiokaらが座標単位で取得された視線移動を構文木上のノード間の遷移に変換する手法を提案し，構文要素に着目した分析が行われている．各構文要素ごとの注視時間の割合を正解者と不正解者の間で比較し，プログラム理解の有無と理解のために重要な構文要素の間に関係があることが示唆された[4]．また，堀川は構文要素間の視線遷移を頻出パターンとして抽出し，理解正否との関係を分析した[5]．しかし，これらの研究では，視線遷移を構文要素間の移動として扱っており，構文木の構造に基づく理解単位の大きさについては検討されていない．

本研究はプログラム理解時に読み手が注目しているプログラム構造の粒度である興味範囲を定量化する事を目的とする．興味範囲を定量化することにより，これまで主観的に解釈されてきた理解の単位を数値として表現することが可能となる．さらに，本手法により，プログラミングの熟練者と初心者の視線パターンの比較や理解戦略の違いを定量的に分析することが可能になる．さらに，本手法により，プログラミングの熟練者と初心者の視線パターンの比較や理解戦略の違いを定量的に分析することが可能になる．プログラム理解における視線遷移をPath Contextとして表現した時の最小共通祖先を求め，その深さを求める．

Path Contextとは連続する2つの視線に対応する構文木上の2ノード間の経路のことである．共通祖先とは，構文木において，Path Contextを構成する2つのノードをともに子孫としてもつノードのことである．また，最小共通祖先とは，共通祖先のうち，最も深さが大きいノードである．本研究では，根ノードを深さ0とする．最小共通祖先の深さが浅い場合は比較的大きな構造単位間を移動している状態，深い場合は，より細かな構造内を移動している状態を表すと考えられる．最小共通祖先の深さの変化をグラフで表現することで，読み手の興味範囲の変化をプログラム構造と対応付けて識別できるかを検討する．

本研究は以下の問いに答えることを目的とする．

RQ1: 連続する2つの視線から最小共通祖先を自動的に抽出できるか?
RQ2: 最小共通祖先の時系列変化から，読み手の興味範囲の変化を識別できるか?

以下，2章では関連研究について述べ，3章では分析に関する準備について説明する．4章では，本研究の提案について述べる．5章の実験では使用するデータと分析方法について説明し，6章で分析結果を示し考察を行う．最後に，7章では結論と今後の課題について述べる．

2 関連研究

視線計測を用いてプログラム理解過程やコードレビュー過程の特徴を分析する研究が多く行われている．Crosby と Stelovsky はソースコード中の単語や構文要素に対応する領域を関心領域 (AOI) として定義し，プログラム理解過程の視線を計測した．その結果，初心者はコメント部分に視線が集中する傾向があるのに対して，熟練者はコメントへの視線が少なく，ソースコード本体をより見ていることが明らかになった [6]．Uwano らはコードレビュー中の作業者の視線を計測・記録するための環境を構築した．その環境を用いてソースコードレビューの実験を行い，誤りを発見する際の視線の動きを行単位で分析した．その結果，レビュー開始時に十分にコード全体を眺めた被験者はバグを早く見つける傾向があることが明らかになった [7]．Busjahn らはプログラマの読み順序を決定する要因を明らかにすることを目的として視線計測実験を行った．分析の結果，読み順序はコードの物理的な配置だけでなく，制御構造や依存関係といった論理構造に強く影響されることが示された．さらに，熟練者は初心者と比べて，より構造を意識した読み方を行う傾向があることが報告されている [9]．Yoshioka らは座標単位で取得された視線移動を構文木上のノード間の遷移に変換する手法を提案した．各構文要素ごとの注視時間を正解者と不正解者の間で比較し，正解したタスクでは if の条件式を有意に多く注視，メソッドの仮引数や print 文は有意に少なく注視していたことが確認された．また，再帰を含む複雑なタスクでは，if 文の条件式よりも，if 文内の文をより多く注視していたことがわかった [4]．他にもデータ依存関係に基づく視線遷移の特徴分析 [8]，各構文要素に対する注視割合や視線の遷移とプログラム理解タスクの正否の因果関係分析 [5] などが行われている．これらの研究では視線遷移を特定の構文要素間の移動として扱っており，構文木の構造に基づく理解単位の大きさ（粒度）については検討されていない．

プログラムの構造的特徴を表現する手法として，Path Context が提案されている．浦井らは Path Context を用いて学習者のプログラム間の類似度を算出し，自動採点における部分点付与する手法を提案した．評価実験の結果，提案手法による部分点は人手による採点結果にある程度近いことが示された [11]．Path Context は AST 上の 2 つのノード間の経路を表現するデータ構造であり，これまで主に機械学習の入力や静的解析に利用されてきた [12, 13]．しかし，視線行動の分析に応用した研究はほとんど行われていない．本研究では，Path Context を機械学習の入力ではなく，視線移動の構造的表現として用いる点が異なる．

3 準備

3.1 視線計測

視線計測はアイトラッキングとも呼ばれ，人がどこを見ているのかを計測することで視覚的注意などを明らかにする生体計測手法である．視線計測装置はアイトラッカーとも呼ばれ，眼球に直接接触する接触型と眼球に直接接触しない非接触型の2種類に大別される．本研究では，被験者の疲労が少ない非接触型の計測装置を用いる．また，非接触型の計測装置には据え置き型と装着型があるが，本研究ではスクリーンの下部に設置する据え置き型を使用する．

非接触型の計測装置の多くは，角膜反射法を採用している．この手法は角膜に近赤外線を照射することで瞳孔の位置を確認し，瞳孔と赤外線の反射点により，眼球の向きを推定し，視線を計算する．しかし，人によって目の形状に個人差があり，また周囲の照明環境によって赤外線の反射や瞳孔の見え方が変化する．そのため，視線計測する前には，被験者ごとにキャリブレーション（補正）を行う．眼球の形状，光の屈折，反射特性に関する情報から正しい眼球の位置を計算する．視線は装置のサンプリングレートに応じて秒間数十回から数千回の頻度でディスプレイ上の2次元座標や，カメラを原点とした3次元座標として取得される．

3.2 視線移動の構文木へのマッピング手法

Yoshiokaらは視線計測装置が出力する画面座標単位の視線遷移を，ソースコードから生成される構文木のノード間の視線遷移に変換する手法を提案した[4]．図1にYoshiokaらの手法の構成を示す．ソースコードを読んでいる被験者の視線移動は視線計測装置によって計測され，ディスプレイ上の(X,Y)座標として時刻と共に記録される．座標-行/列変換モジュールは，座標単位の視線移動とソースコードを入力として受け取り，各視線をソースコード名とソースコード上の行・列番号に変換して出力する．このとき，ソースコード上の行・列番号からソースコードの単語または文字を抽出し，構文解析で得られた構文木上のノードと対応をとる．また，同じ単語，文字に対する連続した注視は1つに統合する．構文木/視線結合モジュールはパーサが生成した構文木と，行・列単位の視線移動を受け取り，構文ノード単位の視線移動を出力する．パーサが出力する構文解析の結果には，ソースコード中の各単語の位置を表す行・列番号と，文字数，およびその単語の構文種類が含まれる．構文木/視線結合モジュールは行・列番号に変換された視線移動を構文解析の各ノードが持つ行・列番号と対応付けることで，視線移動を構文木上のノード単位に変換する．

Yoshiokaらの手法は視線移動を構文木ノード間の遷移として表現することで，ソースコードの表示位置やフォーマットによる違いといった表層的な要因の影響を排除できる．一方で，その遷移がプログラム構造上のどの程度の範囲を跨いで生

経過時間	X座標	Y座標
28:54.1	121	313
28:54.1	123	351
28:54.2	159	363

経過時間	ファイル	行	列
28:54.1	Main.java	1	13
28:54.1	Main.java	2	12
28:54.2	Main.java	2	21

経過時間	構文情報
28:54.1	classDeclaration
28:54.1	methodDeclaration
28:54.2	expression

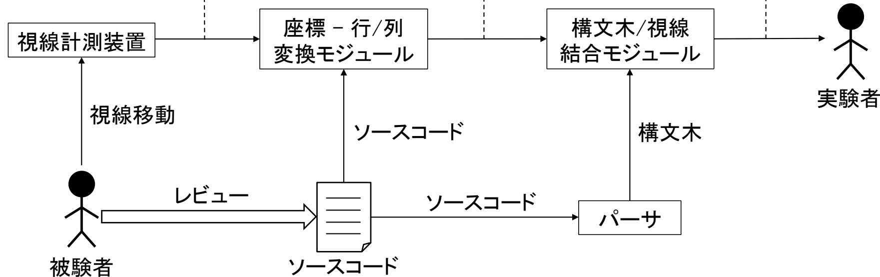

図1 Yoshiokaの提案手法の構成[4]

じたのか，同一文内の移動なのか，ブロックやメソッド，あるいはクラスを跨ぐ移動なのかといった「理解単位の大きさ」については扱っていない．本研究はYoshiokaらの手法によって構文木ノード単位に変換された視線遷移を前提として，連続する2つの視線に対応する構文木ノード間の経路をPath Contextとして表現する．さらに，そのPath Contextを構成する2ノードの最小共通祖先の深さを算出することで，視線遷移をプログラム構造の階層に基づく粒度として定量化する．

3.3 Path Context

Path Contextは，Alonらが提案したニューラルモデルcode2vecにおいて用いられる，プログラムの構造的特徴表現である[12]．code2vecはプログラムを構文木として表現し，構文木上の2つの終端ノードとそれらを結ぶ非終端ノードの列からなる経路を抽出する．この経路はノードの種類および親子関係を示す上下方向の記号列として表現され，その両端に対応する終端ノードと経路の組をPath Contextと呼ぶ．

図2に例として用いるJavaのソースコードを，図3に対応する構文木を示す．丸は各構文要素ノード，矢印は各構文要素間の親子関係を表し，親ノードから子ノードに向かって伸びている．また，赤文字は構文要素(非終端ノード)，青文字はソースコード中のトークン(終端ノード)を表している．図4にPath Contextの例を示す．緑の波線がPath Contextを示しており，両端のノードがaとbであることを示す．このPath Contextを文字列で表現すると下記の通りとなる．

a, (var.Dec.Id, ↑,formalParam., ↑, formalParam.List, ↓,formalParam., ↓,var.Dec.Id),b

括弧で囲まれた文字列(var.Dec.Id, ↑, ..., ↓,var.Dec.Id)は構文木上で経路に含まれる各ノードの構文要素を表す．また，↑および↓はそれぞれ親ノードおよび子ノードへの移動を示し，ノードaからノードbに至る構文木上の経路を表す．なお，

```
1: class A {
2:   int f(int a, int b){
3:     return a < b ? a : b;
4:   }
5: }
```

図 2 Java のソースコード

各構文要素の名称は読みやすさを考慮して，適宜略記する．例の Path Context はメソッド f の仮引数 a と b の間の構文的関係を表す．識別子 a に対応する var.Dec.Id ノードから親方向へ遡って，formalParam，formalParamList に到達し，子方向に formalParam を経由して識別子 b に対応する vara.Dec.Id ノードへ下降する構文上の経路を表現している．

Path Context は構文木上の 2 つの終端ノード間の関係を，それらを結ぶ非終端ノードの列として明示的に表現できる点に特徴がある．この表現により，2 つのノードを単に語やトークンの並びとしてではなく，プログラムの階層構造や構文的な位置関係を保持したまま表現することが可能となる．これにより，本研究のように視線データを用いたプログラム理解過程の分析にも応用できる．連続する視線移動を構文木上のノード間遷移として対応づけることで，各視線移動を Path Context として表現でき，視線移動をプログラムの構造を保ったまま分析できる．また，Path Context は，構文的情報を含むため，同一のトークン同士の関係であっても，それらがどの構文単位に属しているかを区別して表現できる．例えば，メソッド宣言部における仮引数 a から b への Path Context と return 文中の条件式 a < b ? a : b における真の場合の戻り値 a から偽の場合の戻り値 b への PathContext を示す.

メソッド宣言：a, (var.Dec.Id, ↑,formalParam., ↑, formalParam.List, ↓,formalParam., ↓,var.Dec.Id),b
return 文：a, (primary, ↑,ext., ↑, ext., ↓,ext., ↓,primary),b

同じ a から b というトークン間の関係であっても，それらが仮引数同士の関係なのか，あるいは同一式構造内の異なる部分式同士の関係なのかを Path Context によって区別して表現でき，人手による判断を介さずに単なるトークン一致ではなく構造レベルで比較，分析することが可能となる．このように，Path Context はプログラムの意味構造を反映した構造的特徴表現として有用である．

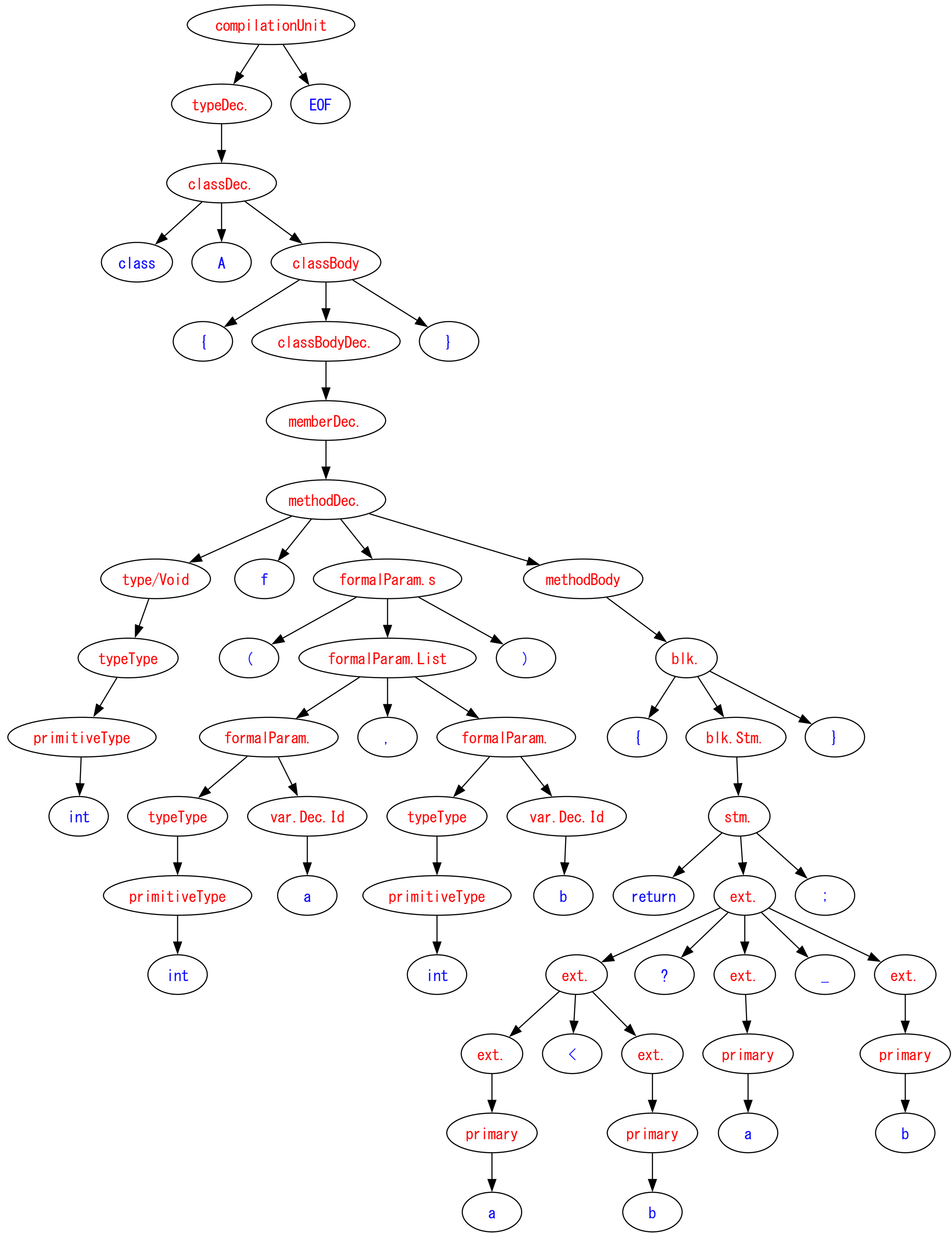

図 3 構文木

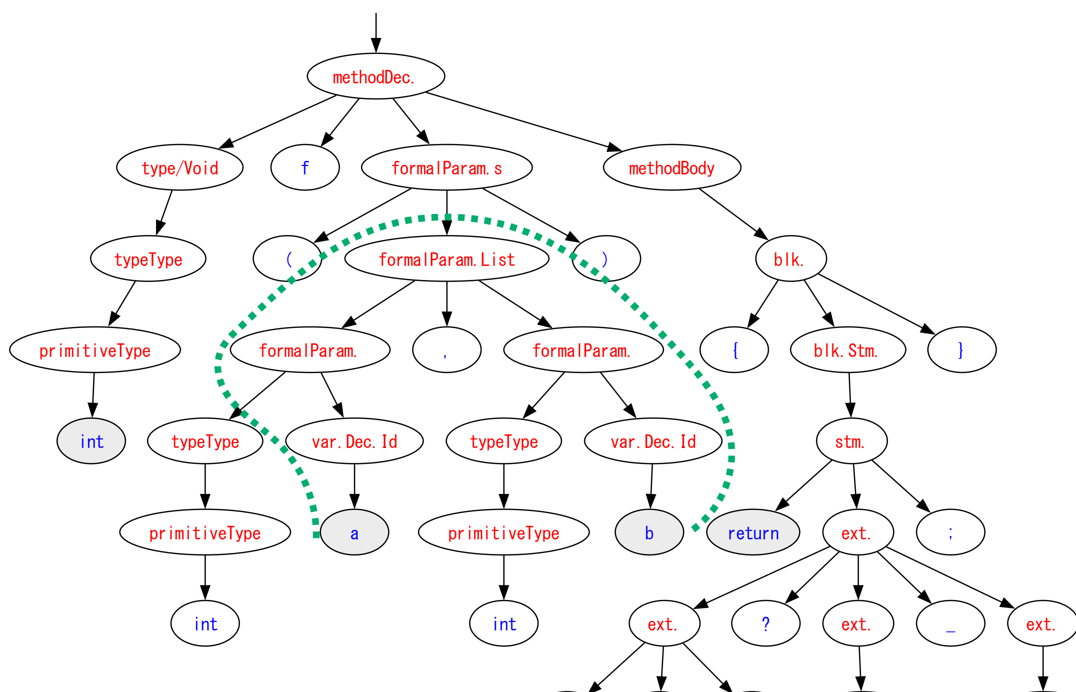

図 4 Path Context の 例

4 提案

本研究は視線移動のPath Contextから最小共通祖先を抽出することで，どの構文単位内で視線移動が生じたか求める手法を提案する．本研究ではPath Contextに含まれる2つのノードの最小共通祖先が，連続した視線移動に共通する細粒度の構文単位を示す点に着目する．

図5に図2のソースコードに対する2種類のPath Contextと，対応する最小共通祖先を示す. 緑色の破線が示す「a」から「b」へのPath Contextを構成する2つのノードの最小共通祖先はformalParameterListノードとなる．図2の2行13列目の「a」に対する視線は，次の異なる粒度として捉えることができる．

- aという単語に対する視線
- int aという文に対する視線
- int a , int bという引数に対する視線
- int f (int a,int b) というメソッド宣言に対する視線
- Aクラスに対する視線

2行20列目の「b」に対する視線も，同様の複数の粒度で捉えることができる．このとき，aからbへの視線移動に対応するPath Contextの最小共通祖先であるformalParameterListは，メソッド宣言における仮引数の並びを表す構文要素である．

同様に，図2の青色の破線が示すPath Contextを構成する2ノードと最小共通祖先であるmethodDec.はそれぞれ以下の意味合いを持つ．

- int：メソッド宣言における戻り値の型
- return：メソッド本体中で計算結果を返すreture文
- methodDec.：戻り値型，メソッド名，引数リスト，メソッド本体を含むメソッド宣言全体

Path Contextを構成する2ノードの最小共通祖先は，2つの連続した視線に共通する，細粒度の構文単位であり，2つの視線が見た要素を共通する1要素で表現する．例えば，図5の緑色は連続する2つの視線がメソッド宣言部の引数一覧を見ていることを示し，青色は連続する2つの視線がメソッド宣言全体を見ていることを示す. また，最小共通祖先はASTの一部であるため，根からの距離（深さ）を持つ．最小共通祖先の深さはソースコード中で対応する構文単位の抽象度を表す. 本研究において，構文木の根に近い上位構造にあるノードほど深さが浅い，葉に近い下位構造にあるノードほど深さが深いと表現する．例えば，図5の緑色（深さ8）は青色（深さ6）より深さが深く，より細かい粒度に着目した視線移動である

図 5 PathContext と最小共通祖先

ことを表す．最小共通祖先の深さが深い場合は，読み手が単語や式といった局所的な構文要素に注目していることを意味し，最小共通祖先の深さが浅い場合は引数リストやメソッド，全体構造といった，より大きな構文単位を意識してコードを読んでいることを意味する．そのため，最小共通祖先の深さを算出することで連続する2つの視線がどのような粒度でソースコードを読んでいるのか，すなわち興味範囲を定量的に表現できる．

提案手法は視線移動をPath Contextから求めた最小共通祖先の系列に変換し，最小共通祖先の深さの変化として表現することで，読み手の興味範囲の変化を捉える．視線移動は2つ以上の連なり(ABCD...)として表され，それに対応してPath Contextも連なり(AB,BC,CD,...)をつくる．さらに，各Path Contextから最小共通祖先を求めることで系列(ABの最小共通祖先,BCの最小共通祖先,CDの最小共通祖先,...)を得られる．プログラムを理解するためには，クラスやメソッド構成などの全体像の理解と個々の文やブロックなど細部の理解の両方が必要である．これらを理解する際には，ソースコードを読む順序や注目する構文単位の粒度が異なると考えられる．

```
 1: public class Main {
 2:     public static void main(String[] args) {
 3:         int x = 3;
 4:
 5:         System.out.println(method1(x));
 6:     }
 7:
 8:     static int method1(int a) {
 9:         if (a > 0) {
10:             return a + 1;
11:         } else {
12:             return 0;
13:         }
14:     }
15: }
```

図6 サンプルコード

図6に例を説明するためのサンプルコードを，図7に例に対応する最小共通祖先を示すため，構文木の一部を抜粋したものを示す.

図7の黄色で示したノードは5行目に書かれたmethod1の呼び出し部と8行目に書かれたmethod1の宣言部に対応する最小共通祖先を示す. また，緑で示したノードは9行目のaの参照部と10行目のreturnに対応する最小共通祖先を表す. このソースコードにおいてmainメソッドとmethod1メソッドの関係性を理解する際には，mainのメソッド内でmethod1を呼び出している5行目とmethod1の宣言部である8行目を交互に見ている視線が計測される．このとき，最小共通祖先は両メソッドの共通要素であり，根に近く，深さが浅い最小共通祖先が続く（例：5行目のmethod1→8行目のmethod1の最小共通祖先は深さ3）．一方，同じコード内でmethod1内のif文など特定ブロックの処理を理解する場合，同ブロック内の各要素に対する視線が計測される．このときの最小共通祖先は各行の共通要素であり，葉に近い，深さの深い最小共通祖先が続く（例:9行目のa→10行目のreturnの最小共通祖先は深さ10）．以上より，Path Contextの最小共通祖先の系列は，プログラム理解中の視線移動における興味範囲の変化を定量的に示すことができる．

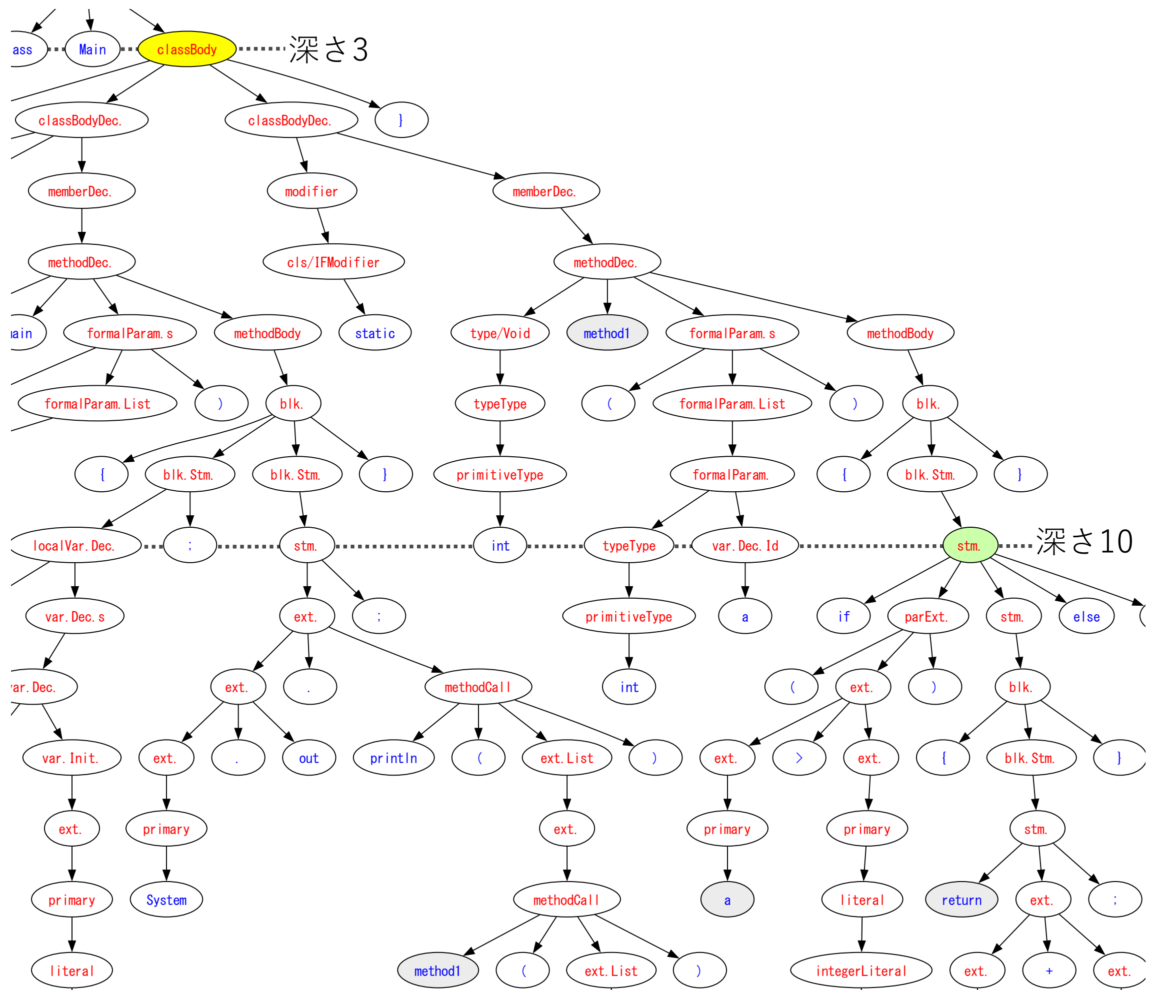

図７視線移動に対応する最小共通祖先を示した構文木

5 実験

5.1 視線データ

本研究では，石田の研究[14]で計測された視線データを使用する．日本語とJavaで記述されたプログラム課題を被験者に提示し，処理内容を理解する間の視線を計測する．被験者は奈良工業高等専門学校の19歳から21歳の学生16人で，全員がJavaによるプログラミングの基礎講義を受講済みである．実験は被験者1名と実験者2名のみがいる静かな部屋で実施し，視線計測装置，タスク提示用PC，記録用PCを使用する．なお，視線計測に非接触型の視線計測装置であるTobii社製Tobii Eye Tracker 4Cを用いる．

視線データは経過時間とディスプレイ上のXY座標から構成され，装置のサンプリングレートである秒間90Hzで情報を取得する．一定時間かつ一定範囲内に視線が留まる状態を停留(fixation)と呼び，その中心座標を停留点と呼ぶ．

5.2 タスク

視線計測時に被験者に与えられたタスク内容について説明する．被験者1人に対して低難易度のタスクと高難易度のタスクを各8問，計16問を与える．被験者にはプログラムの仕様書，ソースコード，質問の3つを提示し，それぞれタスク提示ツール上の異なるタブに表示する．プログラムの仕様書には,「変数bに格納した数字の階乗を計算し表示する.」のように，プログラムの概要が日本語で記述されている．ソースコードはJavaで記述されており，1つのファイルに1つのクラスが含まれている．質問には「6行目の文について，2回目に実行された後のaの値を答えてください.」のように，プログラムの動作を正しく理解していることを確認する質問が書かれている．被験者はタスク開始と共に3つの提示物を読み始め，質問に口頭で制限時間2分30秒以内に解答する．各提示物の読み方や読む順序などは指示しない．解答が用意した答えと一致した場合はプログラムの動作を理解しているとみなし，不一致または制限時間を超過した場合はプログラムを理解していないとみなす. なお，解答の正誤は被験者には伝えない．

低難易度のタスクはすべてmainメソッドのみを使用し，一重の繰り返し文や条件文で構成された，理解が容易と思われるソースコードを用いる．高難易度のタスクは複数メソッドの使用や再帰構造など，複雑なアルゴリズムを使用した，制限時間内での理解が難しいと思われるソースコードを用いる．次の表1にタスク一覧を示す.

表1タスク一覧

難易度	番号	タスク名	仕様
低難易度	1	Factorial	階乗の計算
	2	SearchMax	配列の中の最大値検索
	3	PrimeNum	素数判定
	4	SearchMedian	3つの変数の中央値検索
	5	Power	累乗の計算
	6	Swap	2つの変数の入れ替え
	7	Contained_Substring	特定の文字列を含むのか判定
	8	ReverseString	文字列を反転させる
高難易度	9	TowerOfHanoi	ハノイの塔
	10	NumOfRoute	経路数を求める
	11	MakePermutation	順列を全列挙する
	12	Combination	組み合わせを漸化式から求める
	13	PayMoney	支払う硬貨の組み合わせを求める
	14	StrCombination	文字列の組み合わせを求める
	15	CloudSim	雲の移動シミュレーション
	16	20.lcm_gcd	最小公倍数と最大公約数を求める

5.3 分析

提案手法を用いてプログラム理解時の視線移動を分析する．単語と単語を分離する空白(space)，空行(newLine)，インデント(indent)に対する視線は除外した．

本研究では，提案手法が出力するPath Contextから求めた最小共通祖先とその深さを用いて，読み手の興味範囲の変化を定量的に示す手法の有効性を確認するために，最小共通祖先の深さの変化がどのような理解行動を反映しているのか，著者が目視で読み取る．具体的な対象として5.2節で説明したタスクのうち，構造が単純なプログラムと複雑なプログラムの両方に対する分析を行うために，Task1とTask10を対象に，正解者の視線データを選択する．

図8にTask1のソースコードを示す. Task1は4の階乗を求めるプログラムで，1つのクラス，1つのメソッドから構成される．while文による単純な繰り返しを含む，制御構造が単純なプログラムである．図9にTask10のソースコードを示す. Task10は始点(0,0)から終点(X,Y)までの経路数を求めるプログラムで，1つのクラス，4つのメソッドから構成される．条件分岐と再帰構造を含む，制御構造が複雑なプログラムである．

```
 1: public class Main {
 2:   public static void main(String arg[]) {
 3:     int a = 1, b = 4;
 4:
 5:     while (b > 1) {
 6:       a = a * b;
 7:       b--;
 8:     }
 9:
10:     System.out.println(a);
11:   }
12: }
```

図 8 Task1 の ソ ー ス コ ー ド

```
 1: public class Main {
 2:   static int X, Y;
 3:
 4:   public static void main(String arg[]) {
 5:     X = 5;
 6:     Y = 4;
 7:
 8:     method1(X, Y);
 9:   }
10:
11:   static void method1(int a, int b) {
12:     System.out.println("route = "
13:                         + method2(0,0));
14:   }
15:
16:   static int method2(int a,int b) {
17:     if((a > X) || (b > Y)) {
18:       return 0;
19:     }
20:
21:     if((a == X) && (b == Y)) {
22:       return 1;
23:     }
24:
25:     return method2(a+1, b)
26:                     + method2(a, b+1);
27:   }
```

図 9 Task10 の ソ ー ス コ ー ド

6 結果と考察

6.1 最小共通祖先の自動抽出

連続する2つの停留点に対して，対応する構文木上の終端ノードから最小共通祖先を自動的に抽出できるかを検証した．表2にtask1における視線をYoshiokaの手法により構文要素に変換した結果を示す. IDは各視線に付与した識別番号を表す. 構文要素は視線に対応づけられた構文木上のノード種別，文字列は視線に対応するトークン，行:列はトークンの開始位置における行・列番号を表す. 全構文は構文木の根ノードから当該構文要素ノードまでの経路を示す. なお，構文要素名の末尾に付与されている「@数字」は構文木上におけるノードの通し番号を示す. 表2は3つの連続した視線を表しており, 図8の2行目の「main」,「void」,「static」へと右から左に視線が移動していることを示している．

表3に表2からPath Contextを求め，最小共通祖先とその深さを算出した結果を示す. IDは表2におけるIDの組を表し，どのIDが2つの連続した視線移動に対応するかを示す. 元および先はそれぞれ視線移動の開始点と終了点に対応するトークンを示す. 最小共通祖先は対応する2つの終端ノードの最小共通祖先となる構文木上ノード, 深さは最小共通祖先ノードの構文木における根からの距離を示す. 3つの連続した視線移動からは2つのPath Contextが得られる．ID 1→2は図8の2行目22列「main」から2行目17列「void」への視線移動を表す. 図10に算出されたPath Contextに対応する構文木を示す. 灰色で示したノード1「main」とノード2「void」の最小共通祖先は黄色で示したmethodDec.であり, その深さが6であることが確認できる．同様にID2→3に対応する最小共通祖先はcls.BodyDec.であり, その深さが4であることが確認できる．同様の処理をすべての連続する2つの視線移動に対して行い，最小共通祖先およびその深さを，人手による操作を介さず一意に算出で

表2 task1の視線移動

ID	構文要素	文字列	行:列	全構文
1	methodDec.@11	main	2:22	*compilationUnit*@0 → *typeDec.*@1 → *cls.Dec.*@3 → *cls.Body*@4 → *cls.BodyDec.*@5 → *memberDec.*@10 → *methodDec.*@11
2	type/Void@12	void	2:17	*compilationUnit*@0 → *typeDec.*@1 → *cls.Dec.*@3 → *cls.Body*@4 → *cls.BodyDec.*@5 → *memberDec.*@10 → *methodDec.*@11 → *type/Void*@12
3	cls/IFModifier@9	static	2:10	*compilationUnit*@0 → *typeDec.*@1 → *cls.Dec.*@3 → *cls.Body*@4 → *cls.BodyDec.*@5 → *modifier*@8 → *cls/IFModifier*@9

表 3 PathContext と最小共通祖先

ID	元	先	最小共通祖先	深さ	Path Context
1 → 2	main	void	methodDec.@11	6	methodDec.@11, ↓, type/Void@12
2 → 3	void	static	classBodyDec.@5	4	type/Void@12, ↑, methodDec.@11, ↑, memberDec.@10, ↑, classBodyDec.@5, ↓, modifier@8, ↓, cls/IFModifier@9

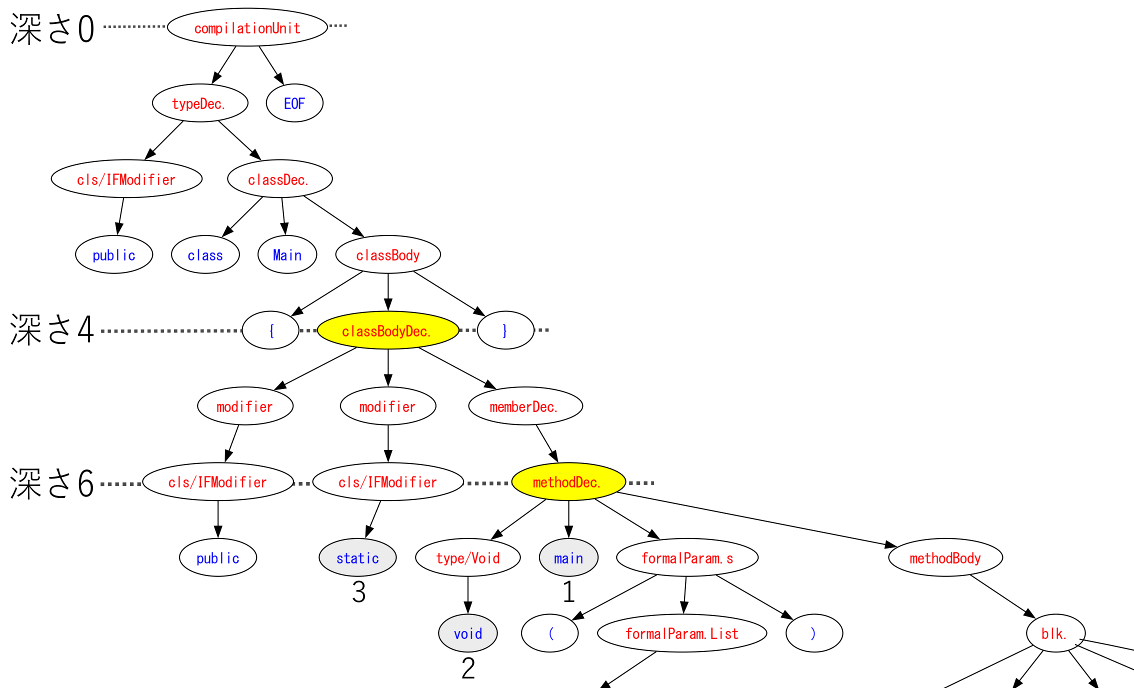

図 10 表3に対応する構文木（一部）

きることを確認した．以上の結果から，RQ1 に回答する．

RQ1: 連続する2つの視線から最小共通祖先を自動的に抽出できるか?

RQ1への回答：Yoshioka の手法で求めた構文要素から Path Context を算出することで，連続する2つの視線から最小共通祖先を自動的に抽出できた．

6.2 時系列変化による興味範囲識別

6.2.1 Task1

この節ではTask1に対する検証を示す.

図11にTask1における被験者02と被験者12の最小共通祖先の深さの時系列変化を示す. 青線が被験者02, 赤線が被験者12を示す. 被験者02は読み始めのstep2〜6で深さ4の浅い遷移が見られたものの, 大部分を深さ8〜16の範囲での遷移が占めている.

図12にa)step1〜step6までの視線移動と, b)step2とstep3の最小共通祖先を示す. a)において, 丸は視線を表し, 線分は連続する視線移動(遷移)を表す. また, 各線分に付与された番号は遷移の順序(step)を表す. a)に着目するとstep2,step3で2行目のvoidからstaticへ移動し, 次に3行列目の=へ移動している. b)に着目するとstep2のPath Context(青の波線)とstep3のPath Context(緑の波線)における最小共通祖先（黄色）はいずれもclassBodyDec.で, フィールドとメソッド宣言を含む構文単位である. 同様に, step5やstep6の最小共通祖先もclassBodyDec.であった. これらの視線移動は主に2行目に対するものであり, mainメソッドの宣言部void main(String arg[])は以降のブロックに記述された内容がプログラム起動直後に呼び出される部分であることを理解するために重要な内容である. また, publicやstaticといった修飾はいずれも対象メソッドのアクセス性を表しており, クラス内におけるメソッドの役割を理解するために重要な要素である. 以上の点から, step2〜6の視線移動はクラス内構造という大きな粒度に着目していると言える.

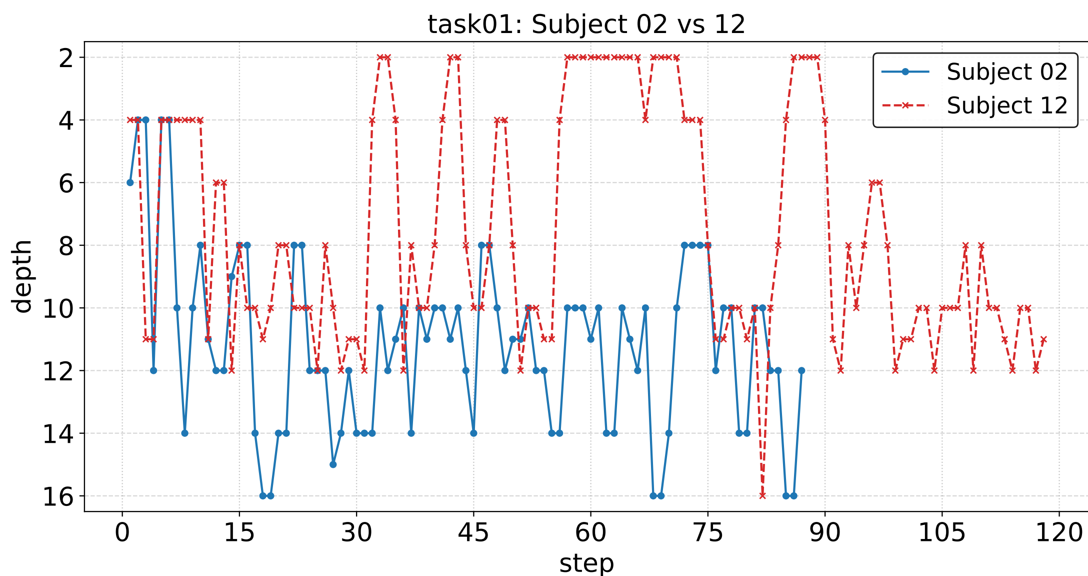

図 11 Task1: 最小共通祖先の深さの時系列変化グラフ

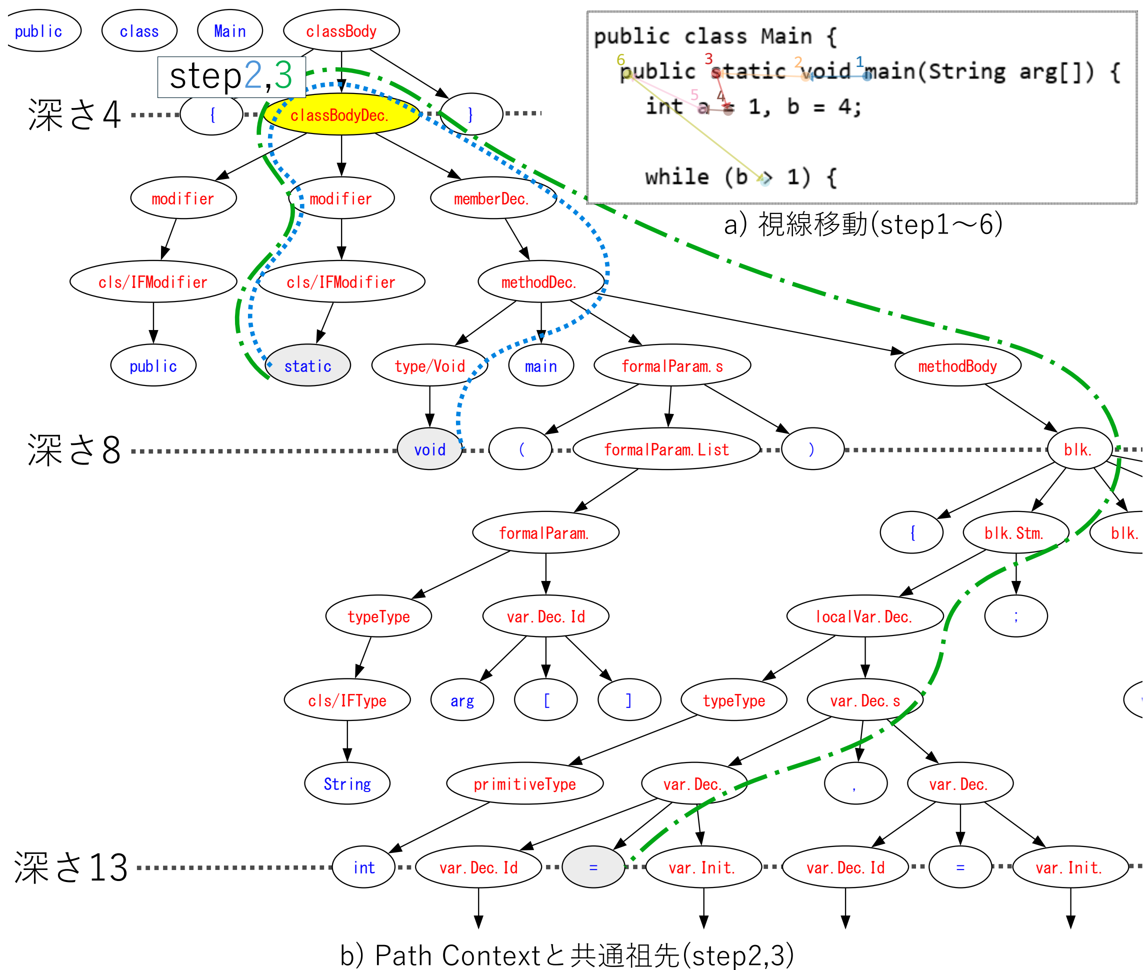

図 12 step1～6 の視線移動と最小共通祖先(被験者02，Task1)

図 13 に step6～13 の a) 視線移動と b) 最小共通祖先を示す．a) に着目すると，main メソッド内の変数宣言と変数が利用されている行に視線が移動しており，特に変数 b 付近に視線が集中している．b) 最小共通祖先を見ると step7 は 1 つの文を表す stm.（オレンジ色）であり，すぐ下の葉ノードから while に相当することが分かる．続く step8,step9 も同じ while の構造内に留まっており，読み手が while 内に着目していると言える．

その後，step10 で 3 行目の変数宣言に遷移しており，最小共通祖先は blk.（青色）である．blk. は複数の文をひとまとまりとして扱う構文単位で，ここでは main ブロック全体を指す．この視線移動は while 文と int 文という異なる文をまたいだ，main のメソッド単位の処理を見ていると言える．また，step11 は変数宣言の並びを表す var.Dec.s（ピンク色）であり，main ブロック内の int 文全体に着目していることが分かる．続く step12,step13 でも変数宣言の構造内に留まっており，int 文の構成要素を詳細に確認していると考えられる．以上の点から，step11～13 の視線移動は main ブロック内の変数宣言という細粒度の構文単位に着目していると言える．

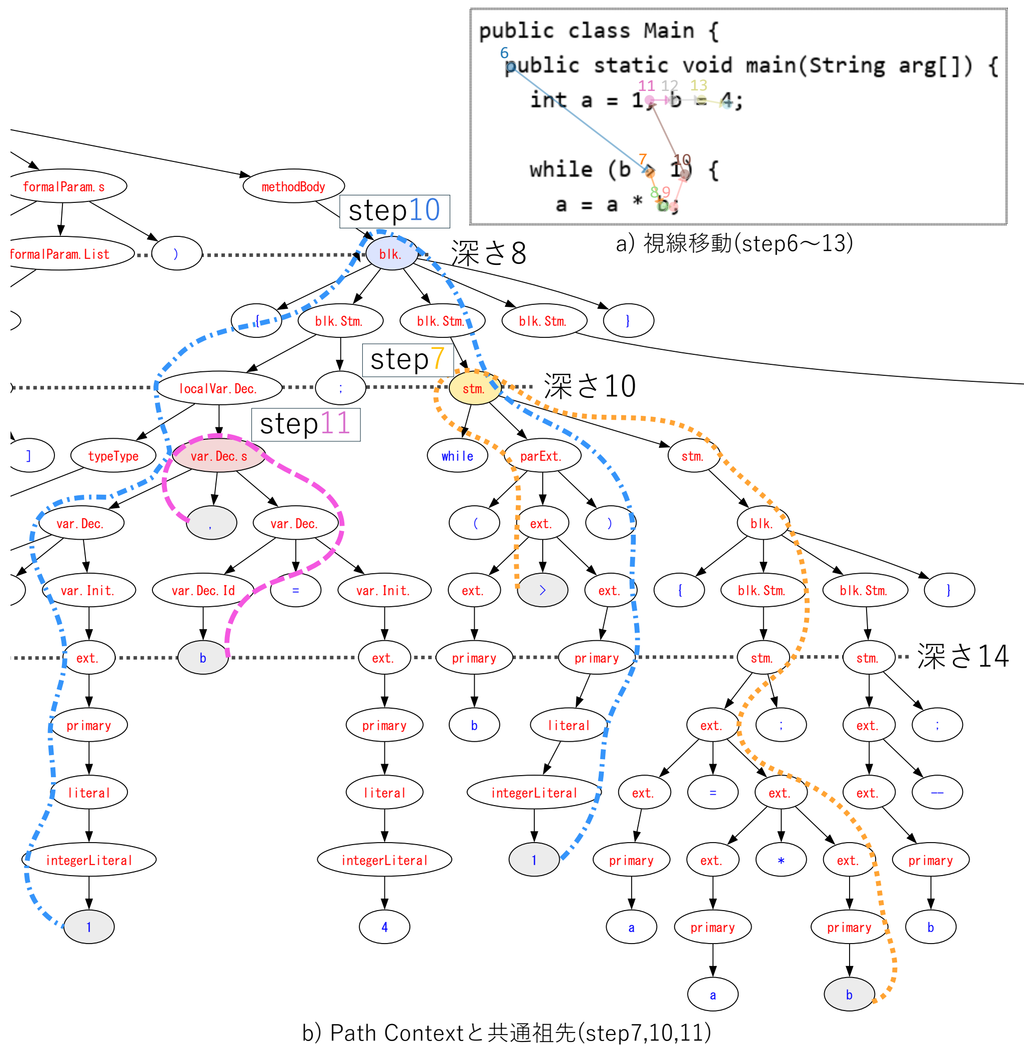

図 13 step6〜13 の視線移動と最小共通祖先 (被験者 02，Task1)

図 14 に step15〜17 の a) 視線移動と b) 最小共通祖先を示す. a) に着目すると step15,step16 で 3 行目の; から 10 行目の println へ移動し，6 行目の; へ移動している．また，step17 では 6 行目の b へ移動している．b) に着目すると step15 の Path Context（オレンジ色）と step16 の Path Context（緑色）における最小共通祖先はいずれも blk.（黄色）であり，step10 と同様に main ブロック全体を表す. 一方，step17 の最小共通祖先は stm.（青色）であり，while 文内部の代入文を表す構文単位である．これらの視線移動は，main の全体の流れを俯瞰的に確認し，その後 while 文内部の変数操作という細粒度な構文単位に着目していると言える．

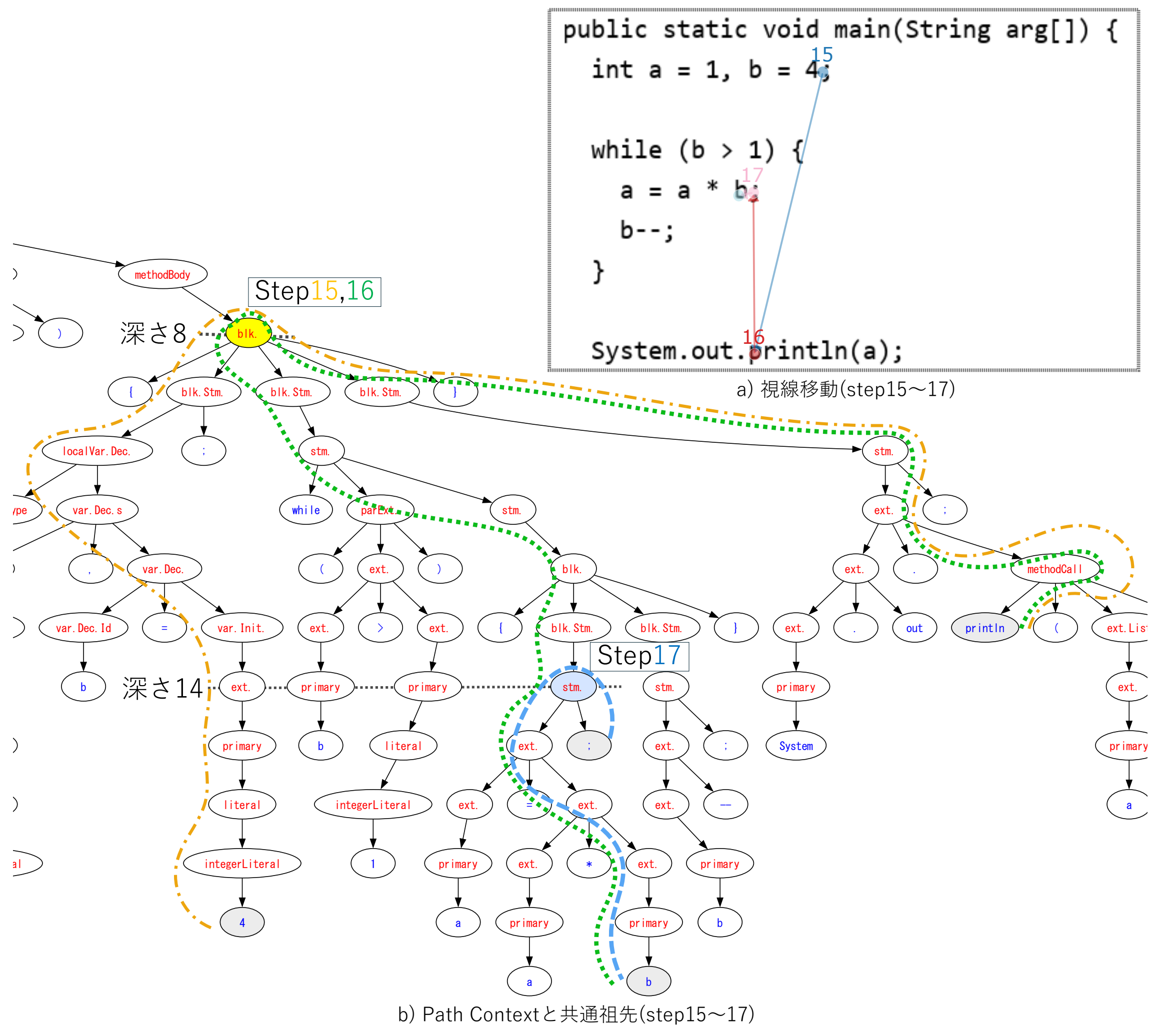

図 14 step15～17 の視線移動と最小共通祖先 (被験者 02，Task1)

図 15 に被験者 12 の step5～9 の視線移動の可視化を示す．main メソッドの宣言部分の static とブロック内の 1 を行き来する視線が見られ，main メソッドという大きな粒度に着目していることがわかる．また，図 11 から，step5～9 では深さの浅い遷移をしていることが確認できる．

図 16 に被験者 12 の step28～31 の視線移動の可視化を示す．int 文内で細かい部分への視線移動が見られ，代入式という細かい粒度に着目していることがわかる．図 11 から，step28～31 で深さの深い遷移が見られる．図 17 に被験者 12 の step82～83 の視線移動の可視化を示す．step82 では a = a * b; の*から b への視線移動が見られ，ブロック内の式に対しての視線移動が見られる．図 11 より，step82 では被験者 12 の視線移動で深さ 16 の最も深い視線が出現している．図 18 に被験者 12 の step103～106 の視線移動の可視化を示す．step103～106 への視線移動も while 文を見ているが，メソッド内の式を横に読むような視線移動はない．図 11 より，深さ 10～12 の視線移動では while 文というブロック単位に着目していることがわかる．深さ 12 より細部へ

```
public class Main {
  public static void main(String arg[]) {
    int a = 1, b = 4;
```

図 15 step5〜9 の視線移動の可視化 (被験者 12，Task1)

```
public static void main(String arg[]) {
  int a = 1, b = 4;
```

図 16 step28〜31 の視線移動の可視化 (被験者 12，Task1)

の視線移動は，step82 の深さ 16 の視線移動以外に見られず，深さ 12 が被験者 12 の細部への視線移動を示す代表的な粒度であると言える．被験者02では，深さ 12 より視線移動が複数観測され，while 文の各行まで細かく確認している．深さの時系列変化グラフより，被験者02 と 12 では，読み方の粒度を異なることが示され，被験者 12 は被験者 02 と比べて，より粗い粒度でソースコードを読んでいる傾向があると言える．

```
while (b > 1) {
  a = a * b;
  b--;
}
```

図 17 step80〜84 の視線動の可視化 (被験者 12，Task1)

```
while (b > 1) {
  a = a * b;
  b--;
}
```

図 18 step103〜107 の視線移動の可視化 (被験者 12，Task1)

6.2.2　Task10

この節ではTask10に対する検証を示す．このタスクのソースコードは3つのメソッドから構成されている．図19に被験者03の最小共通祖先の深さの時系列変化グラフを示す．

図20にstep1〜3のa)視線移動とb)最小共通祖先を示す．a)に着目すると，step1（青線）ではmethod1メソッドからmainメソッドへの視線移動，step2（赤線）ではmainメソッドからmethod2への視線移動が見られ，異なるメソッド間を大きく移動していることがわかる．b)に着目するとstep1のPath Context（青色）より最小共通祖先(青色)はclassBodyで深さ3であり，step2の最小共通祖先もclassBodyで深さ3である．深さ3はフィールドやメソッド宣言が並ぶ構造に対応しており，メソッド全体を単位とした粗い粒度の視線移動であることを示している．

図21にstep18 22のa)視線移動とb)最小共通祖先を示す．a)に着目すると，step18ではmethod2メソッドのstaticからintへ，step19,step20ではintとmethod2（メソッド名）間で視線移動している．さらに，step21ではメソッド宣言部からブロック文内部へ，step22ではブロック文内の式へと視線移動しており，同一メソッド内で視線が段階的に移動していることがわかる．b)に着目すると，step18のPath Context（桃色）の最小共通祖先（桃色）はclassBodyDec.，step19およびstep20のPath Context(緑色)，step21のPath Context（オレンジ色）の最小共通祖先（黄色）はいずれもmethodDec.である．さらにstep22のPath Context（青色）の最小共通祖先は（青色）のext.で式構造を表す．step18〜22の視線移動は深さ4〜15に対応しており，これはmethod2において，

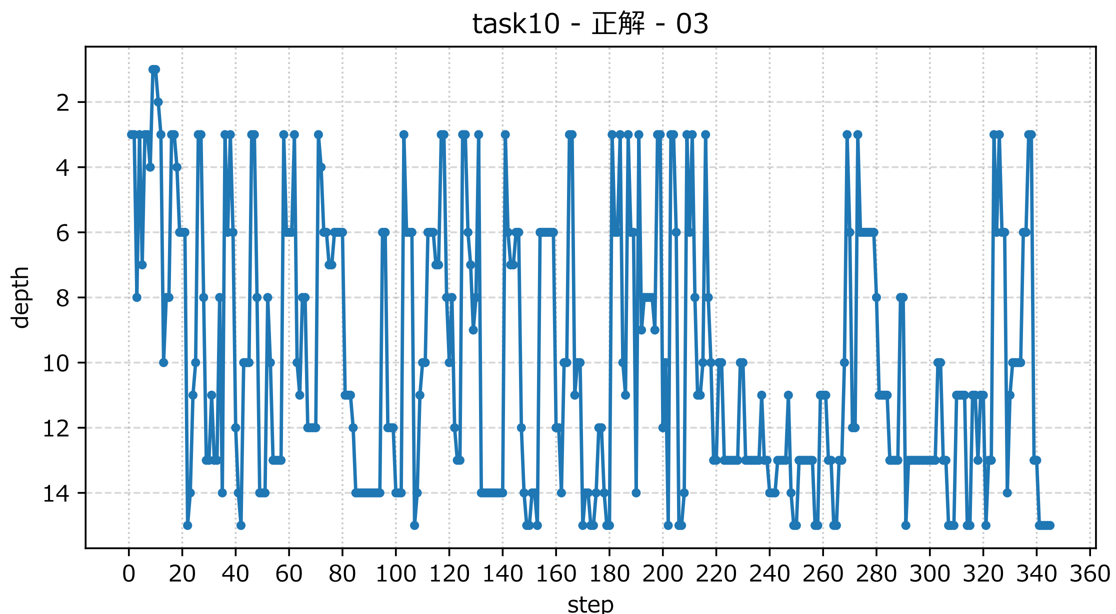

図19 最小共通祖先の深さの時系列変化グラフ(被験者03，Task10)

メソッドという同一構文単位内で，ブロック宣言からブロック文，さらに式へと段階的に細かい粒度に対して着目していることを示している．

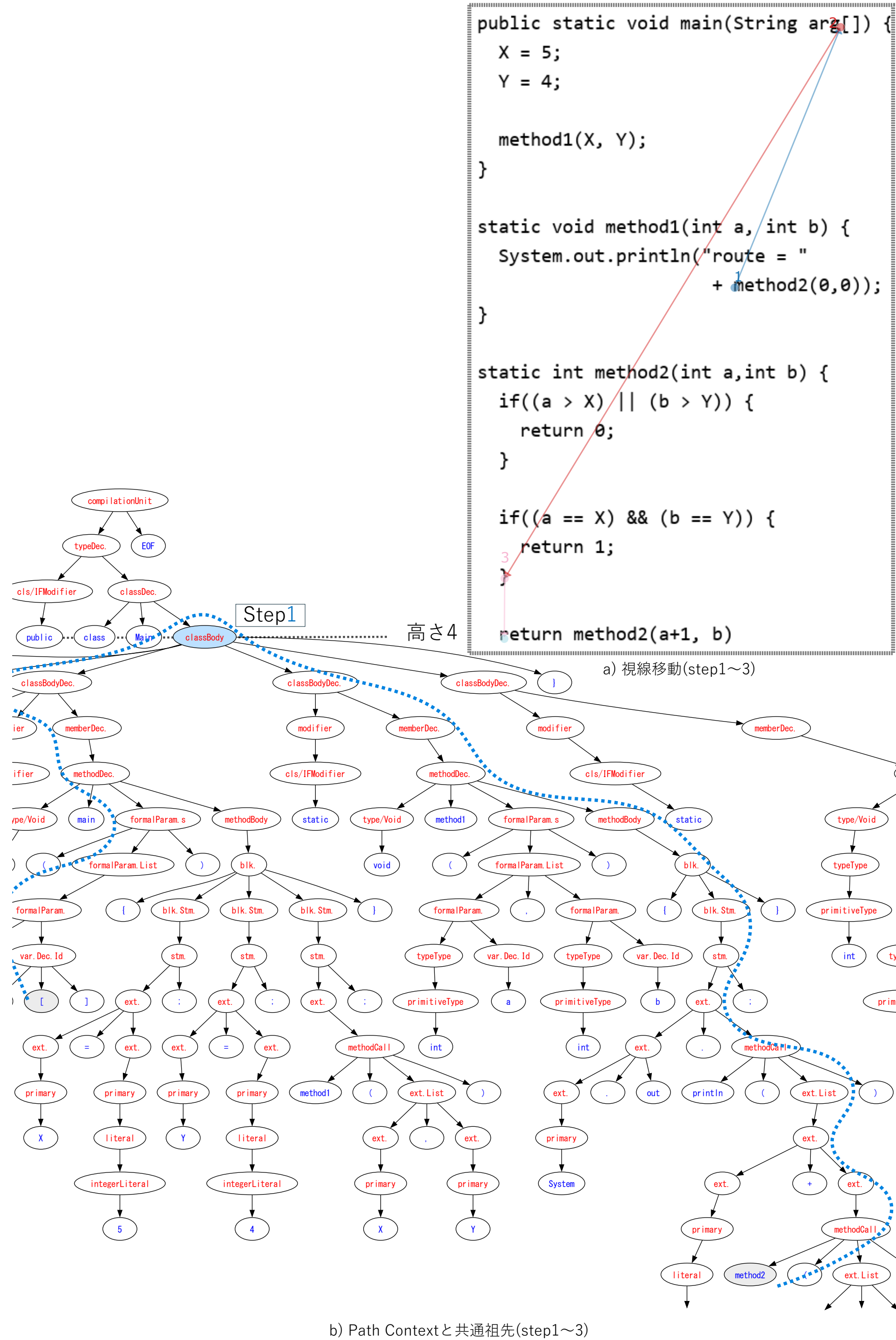

図 20 step1〜3 の視線移動と最小共通祖先 (被験者 03，Task10)

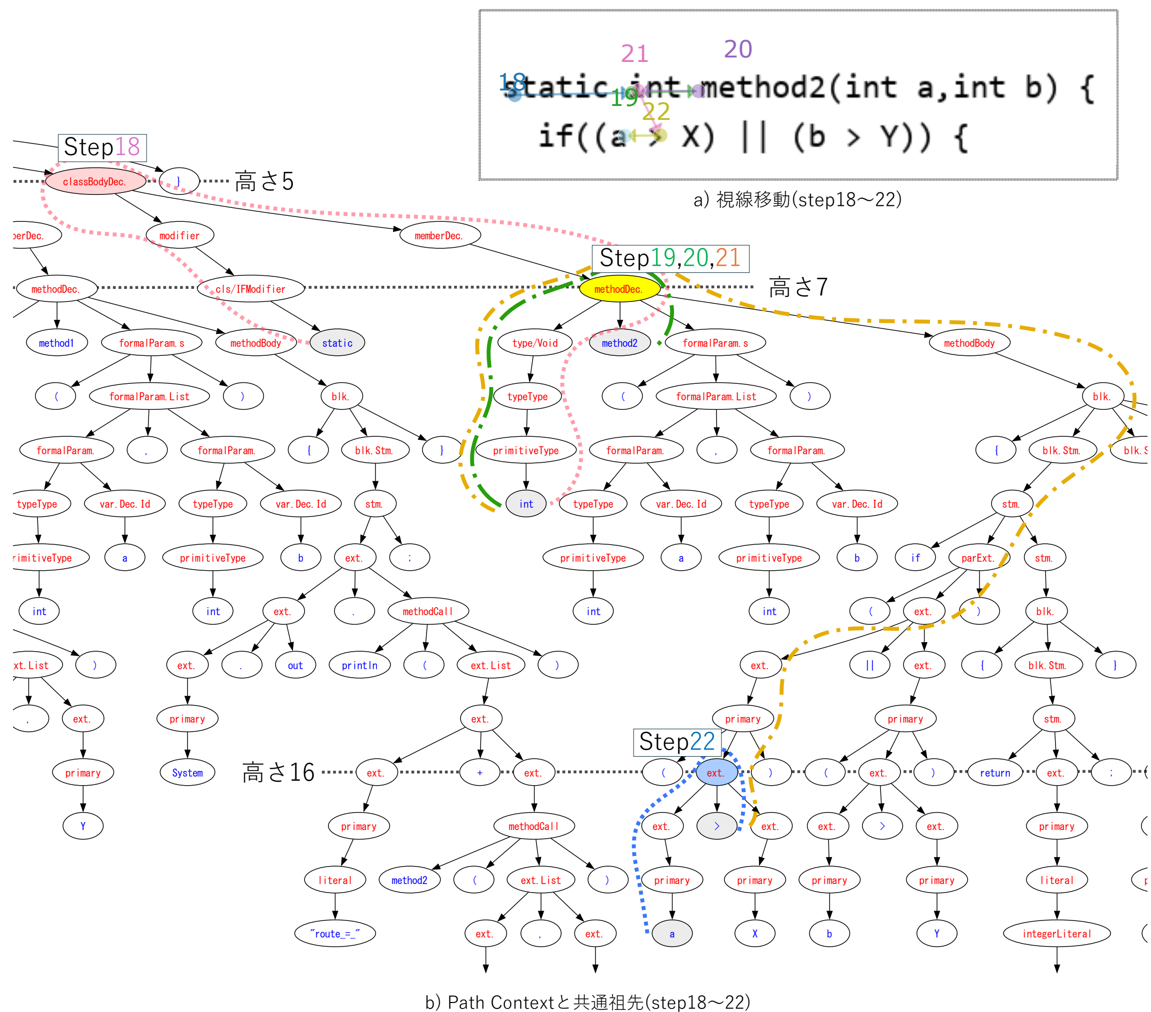

図 21 step18〜22 の視線移動と最小共通祖先 (被験者 03, Task10)

図 22 に step21〜26 の視線移動の可視化を示す.

step21〜26 で視線はメソッド宣言部からメソッド内部へ視線を移した後, 異なる

```
static void method1(int a, int b) {
  System.out.println("route = "
                              + method2(0,0));
}

static int method2(int a,int b) {
  if((a > X) || (b > Y)) {
```

図 22 Task10:step21〜26 の視線移動 (被験者 03)

メソッドへ移動している．図19より，step21〜27では深さが浅いところから深いところへ，さらに再び浅い深さへと遷移しており，粗い粒度から細かい粒度，再び粗い粒度へと視線の着目単位が変化していることがわかる．

以上の結果から，RQ2に回答する．

RQ2: 最小共通祖先の時系列変化から読み手の興味範囲の変化を識別できるのか？

RQ2への回答：最小共通祖先の深さの変化から粗い粒度から細粒度へ，あるいはその逆へと視線の着目単位が変化していることを確認した．最小共通祖先の深さの時系列変化により読み手の興味範囲の変化を識別できた．

7 おわりに

本研究はディスプレイ上の座標単位で記録されたプログラム理解時の視線移動を構文木上のノードに対する遷移に変換し，それらをPath Contextとして表現したときの最小共通祖先とその深さを算出する手法を提案した．学生16名を被験者としたJavaのソースコードの処理内容を理解する実験で得られた視線データを対象に本手法を適用した．その結果，最小共通祖先とその深さを自動的に抽出できることを確認した．また，最小共通祖先の深さを時系列変化グラフで表現することで，読み手の興味範囲の変化をプログラム構造のどの粒度にあるのかを識別できるのか検証した．その結果，最小共通祖先が浅い視線移動は粒度が大きく，クラスやメソッドといった大きな構造に注目しているのに対して，最小共通祖先が深い視線移動は粒度が小さく，式やブロック文など局所的な構文要素に注目していることがわかった．また，深さの変動が見られた場合には，興味範囲が異なる構文単位に移っていることがわかった．これにより，従来は人間による目視による解釈に依存していた「どの粒度単位に着目して読んでいるか」を自動的かつ定量的に抽出できることが示された．

本研究の今後の課題としては，最小共通祖先の深さだけでは読み手の興味内容を十分に特徴づけられない場合があることが確認された．このため，今後は最小共通祖先の深さに加えてノード種別も考慮することで，読み手の興味範囲をより具体的に解釈できるような改善が考えられる．さらに，最小共通祖先の深さのみを興味範囲の指標としたが，Path Contextの長さを組み合わせることで，浅い階層のノード間の移動と，深い階層のノード間における構造的に大きな移動を区別できるようになると考えられる．また，修飾子や宣言要素の違いによって構文木の階層構造が変化し，分析結果の解釈が直感に反する場合がある点も課題として挙げられる．

謝辞

本研究を進めるにあたり, 研究の進め方から論文の執筆に至るまで, 終始丁寧なご指導と多くのアドバイスを賜りました指導教員の上野秀剛准教授に, 深く御礼申し上げます. また, 査読教員である岡村真吾教授には, 貴重なご意見を賜り, 御礼申し上げます.

参考文献

[1] 栗山進, 大平雅雄, 門田暁人, 松本健一, ”プログラム理解度がコードレビュー達成度に及ぼす影響の分析”, 電子情報通信学会技術研究報告; 信学技報, Vol.104, No.571, pp.17-22 (2005).

[2] N.Peitek, J.Siegmund, S.Apel, ”What Drives the Reading Order of Programmers? An Eye Tracking Study”, Proceedings of the 28th International Conference on Program Comprehension, pp.342-353 (2020).

[3] M.Turčáni, Z.Balogh, M.Kohútek, ”Evaluating Computer Science Students Reading Comprehension of Educational Multimedia-enhanced Text using Scalable Eye-tracking Methodology”, Smart Learning Environments, Vol.11, No.1 (2024).

[4] H.Yoshioka, H.Uwano, ”An Analysis of Program Comprehension Process by Eye Movement Mapping to Syntax Trees”, Networking and Parallel/Distributed Computing Systems, Vol.18, pp.137-152 (2024).

[5] 堀川恭吾, ”プログラム理解時の構文要素に対する注視特徴に基づいた効果的な読み”, 奈良工業高等専門学校情報工学科令和6年度卒業研究論文 (2025).

[6] M.E.Crosby, J.Stelovsky, ”How Do We Read Algorithms? A Case Study”, Computer, Vol.23, No.1, pp.25-35 (2002).

[7] H.Uwano, M.Nakamura, A.Monden, K.Matsumoto, ”Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement”, Proceedings of The 2006 Symposium on Eye Tracking Research & Applications, pp.133-140 (2006).

[8] 花房亮, 松本慎平, 林雄介, 平嶋宗, ”視線運動を用いたプログラム読解パターンのデータ依存関係に基づく分析ーー代入演算と算術演算で構成されるプログラムを対象としてーー”, 教育システム情報学会誌, Vol.35, No.2, pp.192-203 (2018).

[9] T.Busjahn, R.Bednarik, A.Begel, M.Crosby, J.H.Paterson, C.Schulte, B.Sharif, S.Tamm, ”Eye Movements in Code Reading: Relaxing the Linear Order”, 2015 IEEE 23rd International Conference on Program Comprehension, pp.255-265 (2015).

[10] S.Aljehane, B.Sharif, J.Maletic, ” Determining Differences in Reading Behavior Between Experts and Novices by Investigating Eye Movement on Source Code Constructs During a Bug Fixing Task”, ACM Symposium on Eye Tracking Research and Applications, pp.1-6 (2021).

[11] 浦井慧, 寺田実, 中山泰一, ”Path Context を用いたプログラム部分点の算出方式とその評価”, 情報教育シンポジウム論文集, pp.118-123 (2024).

[12] U.Alon, M.Ziberstein, O.Levy, E.Yahav, ”Code2vec: Learning Distributed Representations of Code”, Proceedings of the ACM on Programming Languages, Vol.3, pp.1-29 (2019).

[13] 北庄司亮, 松下誠, 肥後芳樹, ”機械学習を用いて模範コードを提示する初学者向けプログラミング学習システムの構築”, 研究報告ソフトウェア工学 (SE), pp.1-8 (2023).

[14] 石田豊美, ”脳波と視線の同時計測によるプログラム理解状況の把握”, 奈良高専特別研究論文 (2020).