プログラム理解パターン抽出のための構文木と視線移動の自動マッピング手法

Automatic Mapping of Syntax Trees and Eye Movement for Program Comprehension Pattern Extraction

吉岡 春彦 * 上野 秀剛 †

あらまし ソフトウェア開発者がソースコードを理解する過程を明らかにすることは，開発作業や学習の効果・効率を改善するために重要な研究である．これまでに多数の研究が効率的な読み方を分析するために，ソースコードに対する開発者の視線移動を計測している．効率的に読む方法を理解するために，研究者はディスプレイの座標として記録された視線移動の情報からソースコードのどこを読んでいるか対応づける必要がある．しかし，エディタ上でのスクロールやウィンドウの移動などのためディスプレイ上の座標をソースコード上の単語と対応づけることは難しい．また，異なるソースコードに対する同じ意味の理解行動を抽出するためには，制御フローやフォーマット，識別子の違いを考慮する必要があるため分析には時間がかかる．本論文では，視線移動をディスプレイ上の座標から，注視した単語と対応する構文情報に変換する手法を提案する．

1 はじめに

ソフトウェア工学の分野の 1 つに，プログラムを読んでいる間の理解過程を対象としたプログラム理解の研究がある．よりよい理解手法を明らかにすることは開発者に効率的な読み方を教え，実装やデバッグの効率を改善する事で開発コストの削減に寄与する．これまでに多数の研究がプログラム理解の分析を目的にソースコードを読む過程を計測している [1] [2] [3] [4]．そのような研究の一部は，ディスプレイに対する視線移動を視線計測装置によって計測し，注視箇所の傾向や視線移動のパターンを分析している．視線計測装置が計測する視線移動は，ディスプレイ上の座標の時系列情報として記録されるため，複数の先行研究がソースコードを読んでいるときの座標単位の視線移動に基づいて開発者の理解過程を分析している [1] [2] [3] [4]．例えば，Hauser ら [1] と Busjahn ら [5] は熟練者が初心者よりも視線を上下に移動する傾向にあることを発見している．開発者がソースコードを読む際の視線移動を異なる開発者が見ることでバグ発見効率が上がることを確認した研究もあり [6]，視線移動から有用な情報を抽出し，優れた開発者が持つ理解パターンや理解戦略を開発者に提示することは，開発効率の向上や開発者の教育に有用である．

しかし，ディスプレイ上の座標に基づいた視線移動の分析は，座標とソースコードの対応をとる必要がある．表 1 に視線計測装置によって記録されたデータの例を示す．表の各行はある時刻に作業者が注視していたディスプレイ上の座標を表す．視線移動は視線計測装置の計測周期によって 1 秒間に 30〜300 点が計測され，分析の際には一定の範囲に一定時間以上留まった視線を停留としてまとめることが多い．ソースコードを対象とした視線移動の場合，ソースコードはエディタや IDE（統合開発環境）に表示される．そのため，ウィンドウ位置の移動やソースコードのスクロール，タブの切り替えなどによって同じ座標に表示されるソースコードは変化する．

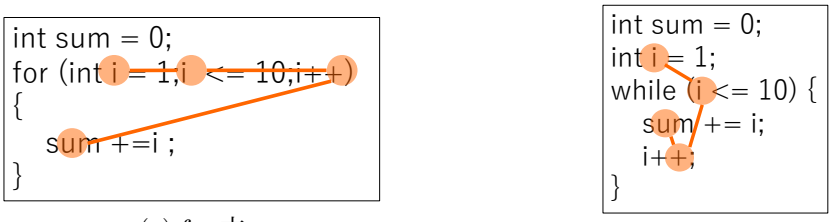
一部の研究は実験用プログラムや IDE のプラグインによって視線移動と操作履歴を組み合わせ，視線移動をソースコード上の行と列に変換している [7] [8]．これらの手法を用いることで同一のソースコードに対する，異なる被験者の視線を比較し，特徴を抽出することができる．一方で，異なるソースコードの場合，制御フローや

*Haruhiko Yoshioka, 奈良工業高等専門学校 システム創成工学専攻 情報システムコース

†Hidetake Uwano, 奈良工業高等専門学校 情報工学科

表 1: ディスプレイにおける座標単位の視線移動

経過時間	X 座標	Y 座標
24:54.1	322	457
24:54.1	322	458
24:54.2	328	463
24:54.2	326	469
24:54.2	320	479



(a) for 文　　(b) while 文

図 1: 2 つのソースコードに対する視線移動

表 2: 構文情報を用いた視線移動の出力例

ID	経過時間	視線移動
1	24:54.1	main メソッド / for 文 / 初期化式 / i
2	24:54.1	main メソッド / for 文 / 条件式 / i
3	24:54.2	main メソッド / for 文 / 変化式 / ++
4	24:54.2	main メソッド / for 文 / ブロック / sum

フォーマットが異なるため，2 つの視線が同一の処理内容に対する遷移であっても異なる座標や行・列番号として記録される．図 1 に同じ制御フローを持つが構文が異なる，2 つのソースコードと，それぞれに対する視線移動の例を示す．図の円と線は視線移動の連続を意味し，それぞれの円は停留を示す．2 つのソースコードは，いずれも 1 から 10 までの和を計算するが，(a) は for 文を，(b) は while 文を用いている．また，2 つの視線移動はいずれも同じ処理（インデックス変数の初期化，条件式，インデックスの増加，結果の計算）を同じ順序に見ている．しかし，2 つの文は異なる構造を持つため，座標単位の視線移動は for 文に対して「左から右へ」見ているものとして，while 文に対して「上から下へ」見ているものとして記録される．複数のソースコードを対象とした視線移動から，より一般的なパターンを抽出することはプログラム理解効率の向上に有用な知見を得るためにも重要であるが，異なる視線移動を手作業で解釈するためには多くの時間を必要とする．

本論文は視線計測装置が出力する座標単位の視線移動を，ソースコードから作成される構文木のノードに対する遷移に変換する手法を提案する．提案手法はレビュー対象のソースコードを構文解析し，行列番号と対応する構文木のノードを割り出す．その後，視線計測装置が出力した座標単位の視線移動を，ソースコード中の行・列番号に変換し，構文木のノードと対応づける．提案手法が出力する視線移動は，視線が停留した単語に対応するノードとその親ノード全ての情報を含む．

表 2 に図 1(a) の視線移動をソースコードの構文情報に基づいて変換した場合の出力例を示す．表の各行は視線の停留した単語に対応するノードの情報を表す．例え

ば ID1 の視線は for 文の初期化式（int i = 1）に対する視線を表す．視線移動の列は視線が停留した単語を表すノードとその親ノードを表し，ID1 の視線が main メソッド内にある for 文の初期化式の i を見ていることを示している．提案手法は構文木のノードに対する遷移として視線移動を表現することで，ソースコードの表示位置やフォーマットによる違いを取り除く．また，視線が停留した単語が属するブロックやメソッド，クラスの情報を含む情報として出力するため，分析目的に応じて異なる粒度で視線移動のパターンを抽出することを可能にする．

2 関連研究

視線移動の計測は初心者と熟練プログラマの違いを分析するためによく用いられる．一定の時間，一定の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 (fixation point) と呼ぶ．連続した停留点は読み手が興味を持つ場所の時間変化を表すとされる．Hauser らは初心者と熟練者の間における停留点に基づいた視線移動の違いを比較した [1]．結果，熟練者グループは非線形にソースコードを読む傾向にあるが，初心者は線形に読むことが分かった．

視線移動の分析に用いられる別の定義として Area of Interest (AOI) がある．研究者は着目の対象である画像やソースコードに矩形，または円形の領域を AOI として定義する．AOI はその範囲内を注視した場合，同一の物を見ていると見なされる領域である．同じ AOI に連続した視線が見られた場合，その AOI に対する長時間の注視として視線移動を要約する．例えばソースコードを対象とする場合，各メソッドに AOI を定義することでメソッド間の視線移動の移り変わりを分析する．複数のプログラム理解に関する研究がソースコードの字句に AOI を定義することで開発者の理解パターンを分析している．Rodeghero らはメソッド宣言，メソッドの呼び出し，制御フロー，その他の 4 種類の AOI をソースコード上に定義し，それぞれに対する注視時間の長さを分析した [2]．分析から熟練の Java プログラマはメソッド呼び出しと制御フローをメソッド宣言よりも長い時間見ていることが分かった．Crosby らは初心者と熟練者の間における，各 AOI に対するレビュー時間の配分を分析した [3]．結果，熟練者は初心者よりもコメントを見る時間が短いことが分かった．ソースコードを対象とした研究においては，ソースコードの各行を AOI とすることで，行単位の視線移動を分析している．Peitek らは初心者と中級者の読み方を比較するために，座標単位の視線データを行単位に変換した [7]．結果，中級者は初心者より，より頻繁にソースコードの上の行に視線移動をすることが分かった．

プログラム理解を対象とした視線計測を行っている先行研究の多くは，ソースコードを画像としてディスプレイ上に表示し，停留点がソースコード内のどの単語に相当するか，手作業で抽出するか，各単語・各行に対して AOI を定義することで分析している．そのため，1 画面内に収まらないソースコードや複数のソースコードを対象とした分析はわずかしか行われていない．一部の研究では，実験用のプログラムを作成し，スクロールや複数のファイルを対象とした分析が行われている．オープンソースソフトウェアの 1 つである iTrace は，座標単位の視線情報を，自動的にソースコードの行と列番号に変換する [1]．iTrace は Eclipse のプラグインとして実装されており，iTrace を用いた視線移動の分析も行われている [8] [9] [10]．

3 提案手法

提案手法は視線計測装置が出力する座標単位の視線移動をソースコードから生成される構文木のノードに対する遷移に変換する．図 2 に提案手法の処理概要を示す．四角はシステムを構成するモジュールを表し，細い矢印は情報の流れを表す．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る．視線計測装置は各時点における注視点をディスプレイ上の座標（例えば，X:121, Y:313）と

[1] https://www.i-trace.org

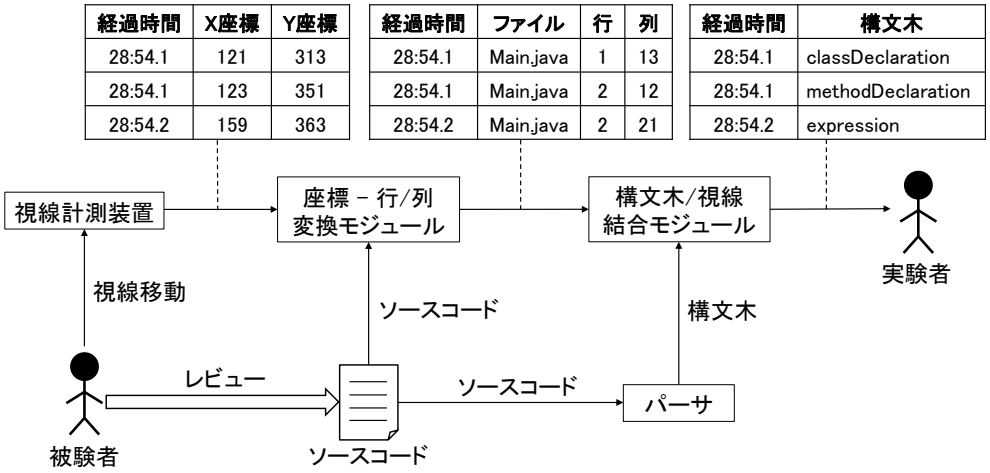


図 2: 提案手法

```
1 public class Main {
2     public static void main(String[] args) {
3         int sum = 0;
4         for (int i = 1; i <= 10; i++) {
5             sum+=i;
6         }
7         System.out.println(sum);
8     }
9 }
```

図 3: Main.java

して時系列に出力する．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，ソースコード名と行・列番号 (Main.java, 行:1, 列:13) として視線を出力する．このとき，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 つにまとめられる．

ソースコードはパーサにも送られ，構文木が生成される．構文木/視線結合モジュールは構文木と，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上の型が含まれる．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結果の各ノードが持つ行・列番号と対応付けすることで，視線移動を構文木上のノード単位に変換する．

図 3 のソースコードから生成された構文木の例を図 4 に示す．紙面の都合，main メソッドに対応するノード (main method @9) 以下のみを示し，一部の中間ノードを削除している．図 4 において，葉ノードはソースコード上の単語を表し，内側のノードは子ノードの属する構文上の要素を表す．例えば，右側にある内側のノード (block @19) は，図 3 の 5 行目にある 3 つの単語 (sum, +=, i) から構成されるブロックを表し，4 行目の for 文 (statement @13) の一部である．提案手法は視線移動を停留点上にある単語と，行・列番号，および単語が属する構文上の要素全てを組み合わせた情報に変換する．より上位の構文要素の情報を含むことで，例えば，図 3 の 4 行目 30 文字目に対する視線を，以下の異なる粒度で捉えることができる．

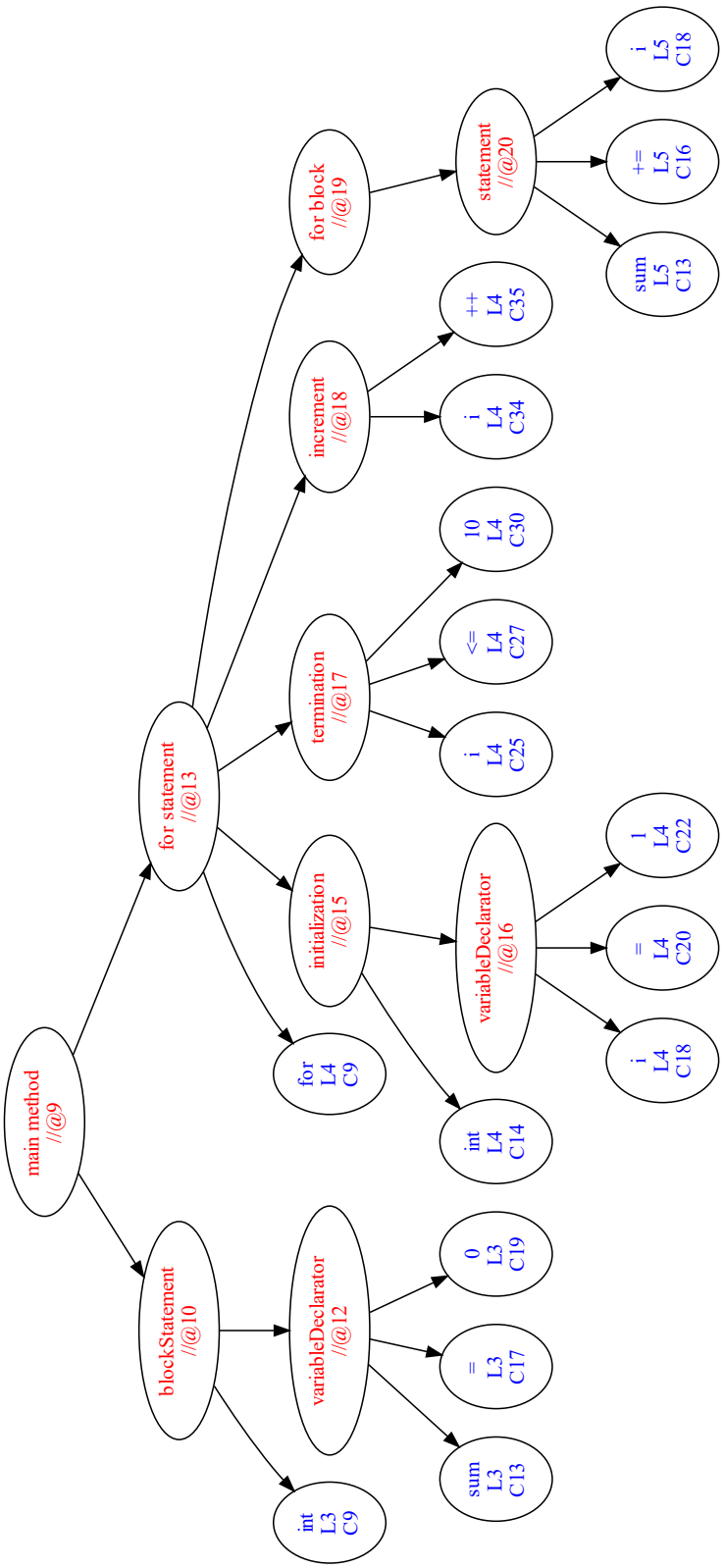

main method
//@9
blockStatement
//@10
int
L3
C9
variableDeclarator
//@12
sum
L3
C13
=
L3
C17
0
L3
C19
for statement
//@13
for
L4
C9
initialization
//@15
int
L4
C14
variableDeclarator
//@16
i
L4
C18
=
L4
C20
1
L4
C22
termination
//@17
i
L4
C25
<=
L4
C27
10
L4
C30
increment
//@18
i
L4
C34
++
L4
C35
for block
//@19
statement
//@20
sum
L5
C13
+=
L5
C16
i
L5
C18

図 4: 構文木の例

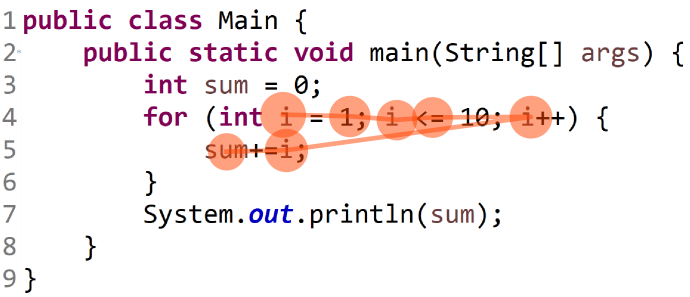


図 5: Main.java に対する視線移動

- 単語 “10” に対する視線
- for 文の条件式に対する視線
- for ブロックに対する視線
- main メソッドに対する視線
- Main クラスに対する視線

提案手法の出力は，分析の目的に応じた粒度（例えばメソッド単位）による視線の要約を最小限の変換処理で可能にする．また，1 つの単語に対する異なる粒度の情報をベクトルとして表現することで，単語が持つ構文上の意味に対する視線の遷移として特徴分析やパターンマイニングが行えると考えられる．

提案手法を Python により実装した．座標単位の視線移動から行・列番号を抽出する “座標 - 行/列変換モジュール” の実装として iTrace を用いた．“構文木/視線結合モジュール” の実装にはオープンソースの parser generator である，ANTLR を使った [2][3]．本論文では対象言語として Java を選択したが，Java 以外の言語を対象とするパーサを用いることで他のプログラミング言語に対する視線移動の分析が可能である．また，分析を補助するためソースコード上に視線移動のデータを可視化する機能を実装した．次章に実装した提案手法を用いた出力例と分析例を示す．

4 分析例

提案手法が開発者の視線移動から理解パターンや理解戦略を抽出するために有用であるか分析例を用いて定性的に評価する．

図 5 に Main.java（図 3）に対する視線移動の一部を実装したシステムによって可視化した結果を示す．図の丸は視線の停留点を示し，丸を結ぶ直線は連続する停留点を示す．視線は 4 行目にある for 文内の i の宣言から条件式，i の増加，sum への追加を順に読んでいることを示す．9 行のソースコードに対する 15.2 秒の視線データを可視化するのに必要な処理時間は CPU が i5-4210U@1.70GHz の PC で 9.9 秒，構文情報に基づいた視線データの出力は 2.4 秒だった．

表 3 に提案手法の出力例を示す．各行は図 5 で可視化された停留点に対応する視線を表す．トークンは停留点上に表示された単語を表し，構文木はトークンに対応する構文木ノードと根ノードの間にある全てのノードの ID を連結した文字列を表す．各ノードの ID は図 4 に対応している．可視化された視線移動を見ると（図 5），4 行目にある for 文の初期化式，条件式，変化式，ブロック内での演算を順に見ていることが容易に理解できる．しかし，表 3 の行・列番号やトークンからは被験者が何を読んでいるかソースコードなしで理解するのは難しい．一方，提案手法により出力された構文木を見ると，一連の視線移動がすべて for 文 (for statement @13) に集中しており，また，初期化式 (initialization @15)，条件式 (termination @17)，変

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

[3]https://github.com/antlr/grammars-v4/tree/master/java/java

表 3: 提案手法の出力

ID	時間	停留点	行	列	トークン	構文木
1	02:15	705, 226	4	18	i	block@9/ for statement@13/ initialization@15/ var.Dec.@16/ i
2	02:16	800, 235	4	22	1	block@9/ for statement@13/ initialization@15/ var.Dec.@16/ 1
3	02:18	877, 234	4	25	i	block@9/ for statement@13/ termination@17/ i
4	02:19	928, 228	4	27	<=	block@9/ for statement@13/ termination@17/ <=
5	02:21	1076, 237	4	34	i	block@9/ for statement@13/ increment@18/ i
6	02:23	708, 283	5	18	i	block@9/ for statement@13/ for block@19/ statement@20/ i
7	02:24	618, 282	5	13	sum	block@9/ for statement@13/ for block@19/ statement@20/ sum

化式 (increment @18)，ブロック内 (for block @19) を順に見ていることが示されている．従来の視線移動の表現方法と比較して，提案手法は以下の利点がある．

- **構文上の文を単位とした分析が容易**
  従来の行・列番号で表される視線移動は単語を単位とした表現である．テキスト上の行を単位とした分析 [7] も行われているが，複数の文が含まれてる場合や，複数行で 1 文を構成することもある．そのため，視線が停留した行がどのような内容か研究者が読み取る必要がある．
  一方で提案手法は構文上の “文” に対する一連の視線として解釈可能な文字列を出力する．図 5 で可視化した視線移動を構文木上に表現した結果を図 6 に示す．細い点線は視線移動が停留したトークンに対応するノードの順序を示す．太い点線はトークン単位の視線移動を親ノードを単位とした形で要約した視線移動を示す．提案手法は構文上の親ノードと視線を対応づけることで，構文上の “文” に対する視線として扱うことができる．太い点線が示す視線移動は初期化式，条件式，変化式，ブロック内を順に見ていることが容易に理解できる．プログラムの基本的な単位である文を単位とした視線移動はその解釈が容易であると考えられる．また，提案手法は表 3 に示したように，視線が停留したトークンが属する文が文字列で出力されるため，パターンマイニングの手法を用いることで理解パターンや理解戦略の抽出が可能である．

- **異なる抽象度による視線移動の分析が可能**
  視線を文単位で要約する事と同様に，より上位の構文要素で視線を要約することで，ブロックやメソッド，クラスを単位とした視線移動の生成が可能である．従来研究においては，実験者が分析対象とする粒度で AOI を設定することで同様の分析が可能であったが，座標情報や行・列番号で個別に定義する必要があることから規模の大きいソーコードを対象とした視線分析においては手間が大きい．また，実験で計測した結果に基づいた探索的な分析が困難である．
  提案手法は構文情報に基づいて，AOI の事前定義無しにプログラム理解時における理解単位を構成する文やブロック，メソッド，クラスと視線移動を関連付ける事ができる．そのため，従来分析が困難であった規模の大きなソースコードに対する，長時間にわたる視線移動に対しても分析が可能である．

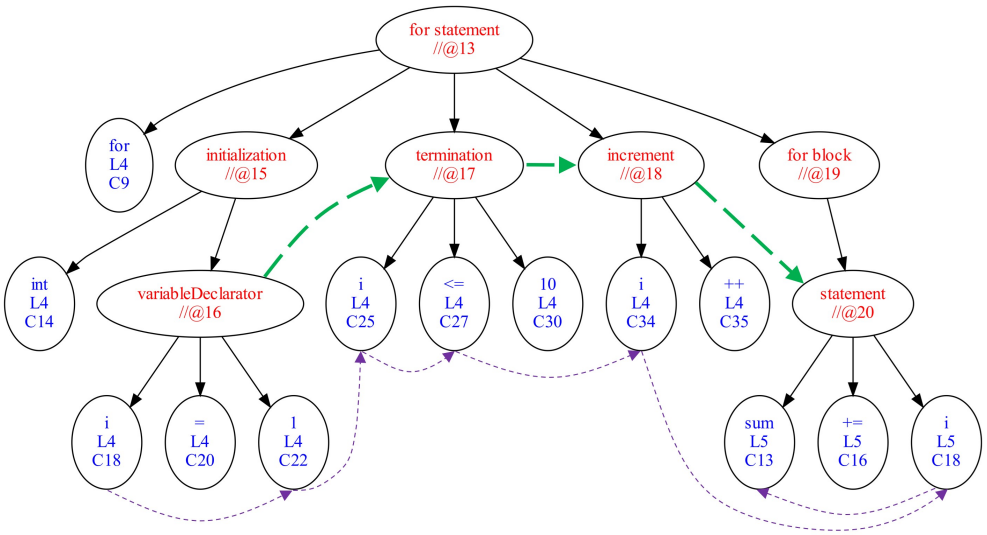


図 6: 構文木を使った視線移動の可視化

- **構文情報のベクトル化によるパターンマイニングとラベル予測**
  提案手法が出力する視線情報は，視線が停留した単語と，単語が属する構文要素（文やブロック，メソッド，クラス）の双方を含む．ある単語に対する視線は，単語が持つ異なる粒度における意味に対する注視として解釈が可能である．例えば，あるメソッド内の for 文で定義されている index 変数への注視は “変数名，型の理解”，“代入値の理解”，“index を用いているループの処理回数の理解”，“メソッド引数に対応する出力値の理解” など異なる粒度の理解過程の一部を構成している可能性がある．ある注視や連続した視線移動がどのような理解過程の一部か理解することは，従来，研究者による手動で行われてきた．提案手法が出力する構文情報から，特定の注視や視線移動を特徴付けるベクトルを生成することで，共通した理解を表すパターンのマイニングや機械学習によるラベル予測が可能になる．

- **異なるソースコードを対象とした分析**
  異なるソースコードの一部に同じ処理内容（例えば配列の合計を計算）が含まれる場合，同一の被験者が，または異なる被験者が各ソースコードに対して同じ読み方をする可能性がある．しかし，同じ処理であっても異なるフォーマットや変数名などを用いている場合，視線の停留位置を表す行・列番号や単語からパターンを抽出することは難しい．提案手法は構文情報を含む視線移動を出力するため，同じ構文要素を持つ単語に対する視線移動を識別子や行・文字数の影響を受けずに抽出することができる．これによって，異なるソースコードに対する視線移動から共通する理解パターンの機械的な抽出が可能になる．

上記の利点はいずれも従来手法では時間を要した視線移動の分析をより短い時間で実行する点で有用であり，開発者の視線移動から理解パターンや理解戦略を抽出することを容易にする．

5 おわりに

本論文で著者らはディスプレイの座標単位で記録される視線移動を，ソースコードから作成される構文木のノードに対する遷移に変換する手法を提案した．提案手法は視線が停留する行・列番号を求めた上で，ソースコードの構文解析結果と対応を取ることで視線が停留した単語の構文要素を視線情報に付与する．分析例を通じて，従来手法と比較した時の提案手法の利点を示した．

本研究の今後の課題として，提案手法を用いた視線移動の分析が挙げられる．4章で述べたように，異なるフォーマット，識別子で書かれているソースコードに対する視線移動から，同じ処理内容を読む様子を抽出することは従来手法では難しい．提案手法を用いて複数の視線移動から同じ処理内容を読むパターンが抽出できれば，視線移動の分析効率を高めることができる．視線パターンの分析によってプログラム理解の能力が高い開発者と低い開発者の特性を明らかにすることは初学者の教育に有用である．また，多くの開発者に広く見られるパターンが見つかれば，より開発効率を高めるための IDE やプログラミング言語の開発に有用な知見となる．

また，提案手法は AOI を定義せずに異なるソースコードに対する視線から構文情報に基づいたパターン抽出ができるため，従来難しかった多数の視線データを対象とした分析を可能にする．ソースコードを対象とした視線移動のオープンなデータセットが少数ながら公開されており [4]，今後，複数の研究によって計測された視線データに共通する理解パターンの分析を行うことは本研究の発展の 1 つである．

謝辞 本研究は JSPS 科研費 JP21K11842 の助成を受けたものです．

参考文献

[1] F. Hauser, S. Schreistter, R. Reuter, J. H. Mottok, H. Gruber, K. Holmqvist, and N. Schorr, “Code Reviews in C++: Preliminary Results from an Eye Tracking Study”, In Proc. Symposium on Eye Tracking Research and Applications (ETRA), pp. 1–5, 2020.

[2] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D’Mello, “Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers”, In Proc. 36th International Conference on Software Engineering (ICSE), pp. 390–401, 2014.

[3] M. E. Crosby and J. Stelovsky, “How Do We Read Algorithms? A Case Study”, Computer, Vol. 23, No. 1, pp. 24–35, 1990.

[4] I. Bertram, J. Hong, Y. Huang, W. Weimer, and Z. Sharafi, “Trustworthiness Perceptions in Code Review: An Eye-Tracking Study”, In Proc. 14th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–6, 2020.

[5] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm, “Eye Movements in Code Reading: Relaxing the Linear Order”, In Proc. 23rd International Conference on Program Comprehension (ICPC), pp. 255–265, 2015.

[6] R. Stein, and S. E. Brennan, “Another Person’s Gaze as a Cue in Solving Programming Problems”, In Proc. 6th International Conference on Multimodal Interface (ICMI), pp. 9-15, (2004).

[7] N. Peitek, J. Siegmund, and S. Apel, “What Drives the Reading Order of Programmers? An Eye Tracking Study”, In Proc. 28th International Conference on Program Comprehension (ICPC), pp. 342–353, 2020.

[8] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, “itrace: Eye Tracking Infrastructure for Development Environments”, In Proc. Symposium on Eye Tracking Research and Applications (ETRA), pp. 1–3, 2018.

[9] A. Abbad-Andaloussi, T. Sorg and B. Weber, “Estimating Developers’ Cognitive Load at a Fine-Grained Level Using Eye-Tracking Measures”, In Proc. 30th International Conference on Program Comprehension (ICPC), pp. 111-121, 2022.

[10] B. Sharif and N. Mansoor, “Humans in Empirical Software Engineering Studies: An Experience Report”, In Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1286-1292, 2022.

[4] http://www.emipws.org/corrected-dataset/