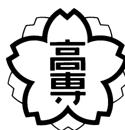

卒 業 研 究 報 告 書

令和3年度

研究題目

意味に基づいた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

指導教員　　上　野　秀　剛　准教授

氏　　名　　吉　岡　春　彦

令和4年1月25日　提出

奈良工業高等専門学校　情報工学科

意味に基づいた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

上野研究室　吉岡春彦

作業効率の高い開発者がソースコードをどのように理解するかを知るためにソースコードに対する視線移動の特徴を分析する研究が行われている．しかし，視線移動は停留点と呼ばれる画面上の座標として計測されるため，その位置に表示されたプログラムの内容や停留点間の処理上のつながりを研究者が個別に理解する必要がある．そのため従来手法では異なる処理構造やフォーマットを持つソースコード間で共通した理解のパターンを抽出することが難しい．本研究はソースコードを構文解析し，構文木のノードと対応する文字列の座標情報を生成することで，座標として記録された視線移動を構文木のノードに対する視線移動として分析する手法を提案する．提案手法は開発者の視線移動を意味上の遷移パターンとして表現することで，ソースコードの処理構造やフォーマットに影響を受けることなく，開発者自身の理解パターンを分析できる．本研究は構文解析を使った新しい分析手法の提案と実装を行う．

目次

1 はじめに **2**

2 関連研究 **5**

2.1 ソースコードを対象とした視線の分析 5

2.2 iTrace 6

3 提案手法 **8**

3.1 概要 8

3.2 処理 8

3.3 分かること 8

4 実装 **11**

5 ケーススタディ **16**

6 おわりに **18**

謝辞 **20**

参考文献 **21**

付録 **22**

A Dog.java の構文木 22

1 はじめに

ソフトウェア開発において，優れた開発者がソースコードを理解する様子（プログラム理解）を調べることで，効率の良いソースコードの読み方を初心者や学習者に伝達することが可能になる．効率の良い読み方を理解することでソースコード理解の効率が向上し，開発コストの削減につながる．プログラム理解の様子を分析する手段として，開発者の視線移動を計測する研究がこれまでに行われている[1, 2, 3, 4]．視線移動は視線計測装置によって記録された，ディスプレイの座標で表される位置の時系列情報である．視線移動を分析することで，理解の様子を調べることができる例として，例えば，視線がある特定の場所にとどまっていると，その場所を注視していることが分かる．また，ある2点を頻繁に交互に見ていた場合，2点の関係を重要視していることが分かる．そのため，開発者がソースコードを読んで，内容を理解する際の視線移動のパターンから開発者の理解パターンを分析する研究が行われている[1, 2, 3, 4]．

プログラム理解を対象に視線移動を計測する研究における従来手法の問題点として，異なるソースコード間で意味における開発者の理解パターンを比較しにくい問題がある．ソースコードが表示されるディスプレイに対する視線移動は画面上の座標として計測される．記録されるデータの例を表1に示す．timeは計測した時刻，xとyは，視線の位置を示す2次元の座標情報である．座標単位の移動情報を使った従来の研究においては，視線移動の方向や，停留時間（視線がある範囲に留まっている時間）についての開発者の特徴を分析している．例えば，熟練者は上下の視線移動が多いことが分かっている[1, 5]．

表1 座標単位の視線移動

time	x	y
2021/12/3 5:24:53.992	3222	457
2021/12/3 5:24:54.018	3222	457
2021/12/3 5:24:54.050	3222	457
2021/12/3 5:24:54.081	3222	457
2021/12/3 5:24:54.111	3222	457
2021/12/3 5:24:54.141	3222	458
2021/12/3 5:24:54.173	3218	463
2021/12/3 5:24:54.204	3216	469
2021/12/3 5:24:54.236	3210	479
2021/12/3 5:24:54.268	3200	497
2021/12/3 5:24:54.300	3179	525
2021/12/3 5:24:54.331	3151	551
2021/12/3 5:24:54.363	3102	582

しかし，異なるソースコードにおいて同じ処理内容だが処理構造やフォーマットが異なる場合，同じ処理内容であるにも関わらず，字句の位置関係が異なる．図1に同じ処理内容を持つ，異なる処理構造の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．図のソースコードはいずれも1~10までの和を求めているが，左はfor文，右はwhile文と用いている処理構造が異なる．左に示したfor文と右に示したwhile文において，字句の位置関係が異なるため，赤ふきだしの座標に基づいた分析は，for文は「右，右，左下」，while文は「右下，右下，右上」のように異なる移動に見える．一方で，緑ふきだしの意味に基づいた分析は，異なるソースコード（for文とwhile文）において「iの宣言，iの利用，iの利用，sumの利用」と，同じ意味の読み方をしていることが分かる．さらに，異なるソースコードにおいて，フォーマットが異なることによって名前（変数名，メソッド名，クラス名）が異なっても，意味が同じであるなら同じものとして分析できる．

開発者は文法に従ってソースコードを記述しているため，開発者の視線移動は文法上の意味に影響を受けていると考えられる．したがって，開発者の視線移動を文法上の意味に基づいて分析することでより詳細に理解パターンを抽出できると考えられる．本研究における文法上の意味とは，例えば，メソッド宣言，宣言された変数，演算子などを指す．より詳細に開発者の理解パターンを抽出することを目的に，意味単位の移動パターンを分析したい．効率のよい見方を知るには，開発者の共通する特徴を抽出する必要があり，2つのソースコードで視線移動を比較したい．しかし，処理構造やフォーマットが異なると，同じ処理内容が記述されていても字句自体（メソッド名や変数名）や字句の位置関係が異なるため，意味の位置関係も異なる．そのため，座標単位の分析手法である従来手法ではソースコード間における意味単位における理解パターンの比較は難しい．ソースコードの字句単位の移動であれば比較しやすい．しかし，座標単位の移動情報から字句単位の移動情報に変換するには，各字句に対応する座標領域を求め，ソースコード上にある開発者の視線が，どの字句を見ているのかを算出する必要がある．各字句に対応する座標領域を求める必要があるため，比較したいが手動以外では難しい．したがって，従来手法の座標単位の分析はソースコードの処理構造とフォーマットに影響を受け，ソースコード間で共通した理解パターンを抽出することが難しい．

複数のソースコードにおいて，同じ処理内容の場合，開発者に対して共通した理解パターンが見られると考えられる．ソースコードを見るとき，開発者は処理内容に影響を受けて視線移動していると考えれるため，理解パターンはソースコードに使用した言語に関する文法上の意味単位のパターンであると考えられる．開発者に対して共通した理解パターンが見られると考えられるが，従来の研究では手動で意味単位の移動情報を生成してきた．生成の具体的な方法として，視線移動の座標をソースコードにプロットし，研究者が座標に対応するソー

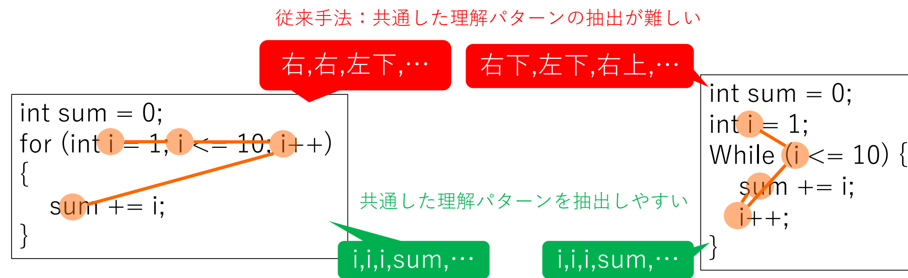

図 1 字句の位置関係が異なる例

スコード上の文字を見て，座標と文字の対応をとっていた．また文字に対応する字句の対応をとり，字句に対応する意味を研究者が手動で生成していた．従来手法は，研究者が手動で意味単位の移動情報を生成するため，ソースコードの数が増えるほど研究者の負担が増える．そのため，複数のソースコードで共通した理解パターンを抽出することは難しい．

本研究は開発者の視線移動を文法上の意味に基づいて自動で分析できるようにすることを目的とする．そのためには，従来，研究者が手動で行ってきた座標単位の移動情報から文法における意味単位の移動情報への変換を自動化する必要がある．

本研究の提案手法は，構文解析を利用することで座標単位の移動情報を意味単位の移動情報に自動で変換する．具体的には視線の座標に対応する字句を求め，ソースコードから構文木を生成し，字句と構文木のノードの対応をとることで，字句に対して文法上の意味をマッピングする．提案手法は異なるソースコード間で同じ処理内容が含まれる場合，共通した意味単位の移動として視線情報を出力する．複数のソースコードで共通した意味単位の移動パターンを抽出することによって，ソースコードの処理構造とフォーマットという特徴に影響されない，開発者自身の理解パターンが分かる．

2 関連研究

2.1 ソースコードを対象とした視線の分析

視線の計測は初心者と熟練者の違いを分析することを目的によく用いられており，プログラム理解の分析においてもソースコードを対象とした視線移動の分析に用いられる．視線がある範囲に一定の時間長を超えて留まっている状態を停留(Fixation)といい，その中心座標を停留点という．停留点は読み手の関心が高い場所とその時系列の変化を明らかにするために用いられる．Hauserらは，コンピュータサイエンスコースの学部生を初心者，既に学士号を取得し初めての職業経歴を持つものを上級者，博士課程の学生または異なる協力企業のプロプログラマを専門家とし，各グループの視線から停留点移動の差を比較した[1]．結果，初心者に対して上級者と熟練者はソースコードをレビューする際に，コードを非線形に読む傾向があることが分かった．

視線の停留を判断する範囲として，見る対象物（絵やソースコード）に複数の領域(AOI: Area Of Interest)を設定し，視線がその領域内に留まった時間長と回数を算出する手法がある．プログラム理解を対象とした研究においてもソースコードに出現する字句の中で処理内容を持つ最小単位のかたまり(以降，トークンと呼ぶ)に対してAOIを定義することで開発者の理解パターンを分析する研究がある[2, 3, 4, 6]．Rodegheroらは，メソッドのシグネチャ，メソッドの呼び出し，制御フロー，その他の領域，にAOIを割り当て，ソースコードの内容を要約するプログラマの視線移動を分析した[2]．分析の結果，メソッドのシグネチャをメソッドの呼び出しよりも詳細に読み，呼び出しを制御フローより読むことが分かった．Crosbyらはソースコードの中の各コメントと処理内容に著者らが手作業でAOIを割り当て視線移動を分析した．分析の結果，熟練者は初心者よりコメントに費やす時間が低いことが分かった[3]．

プログラムは文(statement)で構成されており，多くのプログラムにおいて1つの文は1行に書かれている．したがってレビュー作業者はプログラムを理解する際に1行を単位として読むと考えられる．そのため，視線の停留を判断する範囲としてソースコードの各行を設定し，開発者がどの行をどのような順で見ているか分析する研究がある[7, 8]．Uwanoらは，C言語で書かれた12~23行の小規模なソースコード6つを用意し，各ソースコードに1つのバグを埋め込んだ．そして3~4年のプログラミング経験を持ち1度以上はコードレビューを経験している5人の大学院生に対して，埋め込まれたバグを探させ視線を計測した．視線計測によって得た座標単位の移動情報を，ソースコード上の行番号単位の移動情報に変換し分析した[7]．結果として，被験者はまずソースコードの全行を上から下に向かって簡潔に読み，次に特定の部分を集中的に読む傾向があることが分かった．

Peitekらも同様に，オブジェクト指向の基礎を理解している人を初心者，コン

ピュータサイエンス専攻の大学院生を中級者とし，初心者12人と中級者19人に対して10個のソースコードを読ませて最終的な出力結果を考えて入力させ，視線移動を計測した．そして視線計測装置が出力した座標単位の移動情報を行番号単位の移動情報に変換し分析を行った[8]．結果，中級者は初心者よりソースコードの上の行へ視線移動する割合が大きいことが分かった．

座標単位の移動情報を用いた従来研究は，開発者がソースコード中のどの字句を見ているかを，視線座標から研究者が手動で判断していた．本研究で開発したシステムは視線移動からどの字句を見ていたかを自動で出力する．オープンソースソフトウェアiTraceは自動で視線座標を行番号と列番号に変換する．本研究は行番号と列番号から，ソースコード上のどの文字を見ているかを求め，座標単位の移動情報を字句単位の移動情報に自動で変換する．また，本研究は開発者の視線移動は，ソースコードにおける文法上の意味に影響を受けていると考える．例えば，文法上の意味である変数宣言をよく見る，といった特徴があると考えられる．従来，研究者がソースコードの処理内容を理解してAOIを割り当てて分析を行っている．しかし，ソースコードにおいて開発者は処理内容を文法に従って記述しているため，構文解析によってソースコードの字句に対して自動的に処理内容をマッピングできると考えた．視線移動を文法上の意味に基づいて分析できれば，より詳細に開発者の理解パターンを分析できると考える．優れた開発者の理解パターンを抽出することは，効率の良いソースコードの見方を初心者や学習者に伝達することにつながる．そのため，本研究はソースコードを構文解析し，構文解析によって得た文法上の意味を字句に対してマッピングすることで，従来，研究者が座標単位の移動情報から手動で生成していた意味単位の移動情報を自動で得る．

2.2 iTrace

開発者は多くのソースコードファイルから構成されるソフトウェアを扱う．そのため，プログラム理解やデバッグ時に，画面上に表示するファイルを頻繁に切り替える必要がある．しかし，従来の視線分析は，ソースコード上の座標情報で行われていたため，スクロールまたはファイルの切り替え時に視線を追跡することは難しく，研究者が視線の座標から行番号と列番号を手作業で求めていた．Uwanoらはスクロールを考慮し座標から行番号への変換を自動で行っている[7]が，実験専用のソフトを使ったものであり，開発者が一般的に使う環境と異なる．そのため普段の作業時に見られる視線移動と異なる可能性がある．iTrace[9]は一般の開発でよく用いられるIDEのひとつであるEclipse上に表示されたソースコードに対する視線移動からファイル名と行番号，列番号を自動算出する．

iTraceが出力するデータの例を表2に示す．plugin_timeは開発者の視線を計測した時刻のUnix時間(ms)，xとyは停留点の座標，source_file_lineとsource_file_colは座標

表 2 iTrace が出力するデータの例

plugin_time	x	y	source_file_line	source_file_col
1638509094839	751	655	35	48
1638509095090	645	552	29	34
1638509095121	618	546	29	32
1638509095152	611	545	29	31
1638509095182	604	545	29	30
1638509095214	595	547	29	29
1638509095246	592	549	29	28
1638509095277	589	551	29	28
1638509095310	581	555	29	27
1638509095341	568	563	30	28

に対応するソースコード上の行番号と列番号である．

視線座標に対応するソースコード上の行番号と列番号が分かると，ソースコードのどの字句を見ているかを求めることができる．本研究で作成した構文・視線結合モジュールは，iTrace から行番号 (source_file_line) と列番号 (source_file_col) の移動情報を得て，字句単位の移動情報に変換する．構文・視線結合モジュールはソースコード, 行番号と列番号の移動情報を入力とし，座標単位の移動情報を行番号と列番号の移動情報に変換する機能があれば，iTrace に限らずどのようなツールを入力としてもよい．

3 提案手法

3.1 概要

研究者は視線計測装置の利用によって開発者の視線移動を座標単位の移動情報として記録する．提案手法は開発者の理解パターンをより詳細に分析することを目的とし，視線計測装置が記録した座標単位の移動情報を文法における意味単位の移動情報に自動で変換する．手法に対する入力は視線計測装置から得た座標単位の移動情報であり,「座標単位の移動情報」→「字句単位の移動情報」→「意味単位の移動情報」と変換し，意味単位の移動情報を出力する．本提案手法は，ある開発者は意味上の特定の理解パターンが多い，といった分析を可能にすると考えられる．例えば，変数宣言のような意味を見ることが多い，メソッド宣言とメソッド呼び出しを頻繁に見る，などの開発者の意味上における理解パターンを抽出できると考えられる．

3.2 処理

視線計測装置は開発者の視線移動を記録し座標単位の移動情報を出力する．本提案手法は視線計測装置で得た座標単位の移動情報を入力とし，意味単位の移動情報を自動で出力するが，直接，座標から意味への変換はできず, 座標からソースコードにおける字句に変換する必要がある．なぜなら，座標はディスプレイの左上を[0,0]とするピクセルを使った位置情報である．座標単位の移動情報を使った従来手法において，研究者はソースコード上に視線移動をプロットし，目視で座標と字句の対応を取っていた．しかし，本提案手法は自動で意味単位の移動を出力する必要があるため，字句単位の移動情報も自動で生成する必要がある．したがって，ソースコードのどの字句を見ているかを知るためには，フォントサイズ，ウィンドウの位置，ソースコードがどの程度スクロールされているかを元に算出する必要がある．従来，研究者は字句に対応する意味を手作業で生成している．しかし，構文解析をすると算出した字句に対応する文法上の意味が自動で分かる．そのため，本提案手法は構文解析を使うことで，字句単位の移動情報を意味単位の移動情報に自動で変換することができる．

3.3 分かること

本システムによって分かると考えられることを以下に示す.

分かること1: 開発者のより詳細な理解パターン

分かること2: 意味上における開発者自身の理解パターン

優れた開発者の理解パターンを抽出し，初心者や学習者に伝達することは，開発効率を高め，開発コストの削減につながる．そのため，座標単位の移動情報を使った従来手法は，視線移動の方向と，注視する場所に関して開発者の視線移動を分析している．例えばPeitekらによって，中級者は初心者よりソースコードの上の行へ視線移動する割合が大きいことが分かっている[8]．しかし，開発者の視線移動を文法の意味に基づいて分析することができれば，開発者の理解パターンをより詳細に分析できると考えられる．例えば，ある開発者は変数宣言という意味をよく見る，またある開発者はメソッド宣言とメソッド呼び出しを頻繁に見る，などが分かると考えられる．

座標単位の移動情報を使った従来手法において，研究者は手作業で座標に対応する意味を生成し，ソースコードの意味に基づいた分析を行ってきた．本システムは，開発者の視線移動を意味単位の移動情報として自動で出力するため，研究者が意味を生成する必要がなくなり，負担を減らすことができる．

また，異なるソースコード間で意味上における理解パターンを比較できる．なぜなら，クラス名や関数名，変数名が異なっても，異なるソースコード間において共通した文法の意味を持つためである．座標単位の移動情報を使った従来手法において，自動で意味に基づく比較をすることは難しかった．なぜなら，従来の研究手法である，座標単位の移動情報を使った視線移動の分析は，被験者の視線移動はソースコードの処理構造やフォーマットといった特徴に影響を受け，字句の位置関係が変わるためである．例えば，for文において継続条件式と増減式は同じ行にあるが，while文において継続条件式と増減式は同じ行にない．このように，意味の位置関係が異なること（以降，意味の位置関係問題と呼ぶ）があるため，座標単位の移動情報を使い異なるソースコード間で意味に基づく分析をすることは難しい．

あるソースコードが特定の意味上における理解パターンをしやすい特徴を持っていたとき，特定の視線移動が見られても被験者の理解パターンであるか判断が難しい．例えば，変数宣言が大量にあるソースコードの場合，被験者の理解パターンは変数宣言をよく見る特徴が出ると考えられる．しかし，ソースコードの特徴に影響されたためであり，開発者の特徴と関係ない．被験者の理解パターンに，ある特徴が出た場合，ソースコードの特徴に影響されたためか，被験者自身の理解パターンが判断が難しい．問題を解決するには，複数のソースコードで共通する被験者の視線移動のパターンを抽出する必要がある．なぜなら，あるソースコードは特定の意味上における理解パターンをしやすい特徴をもっていても，他のソースコードすべてが特定の意味上における理解パターンをしやすい特徴をもつとは考えられない．したがって，複数のソースコードで共通した意味上における理解パターンであれば，被験者自身の特徴であると考えられる．

共通した理解パターンを抽出するには，異なるソースコード間で理解パターン

を比較することができる必要がある．座標単位の移動情報を使った従来手法は，は研究者が手動で複数のソースコードに対して意味を生成するという負担の面で比較は難しい．また，意味の位置関係問題があるため，自動で比較することは難しい．

しかし従来の手法がソースコードの処理構造やフォーマットに影響を受けたのは，位置関係，つまり座標上の問題であり，意味上の移動情報は影響を受けない．そのため，意味上の移動情報であれば，複数のソースコードで共通した理解パターンを抽出できると考える．したがって，意味上における被験者自身の理解パターンを抽出できる．

4 実装

開発者の視線移動を意味単位の移動情報として出力し，研究者が開発者の理解パターンを分析するために提案手法を実装したシステムを開発した．システムは1) ソースコードに対する視線移動を，視線計測装置が出力する座標単位から意味単位に変換する機能に加えて，2) 視線移動をソースコード上にプロットし可視化する機能，視線移動から生成した停留点単位の移動をソースコード上にプロットする機能を持つ．

本システムの構成を図2に示す. 四角は装置とソフトウェア, ソースコード, 人間，矢印は処理と情報の流れを表す. システムは視線計測装置，被験者，iTrace, ソースコード, 構文･視線結合モジュール，ANTLR, 実験者で構成される．構文･視線結合モジュールは行番号と列番号単位の移動情報を意味単位の移動情報に変換する．

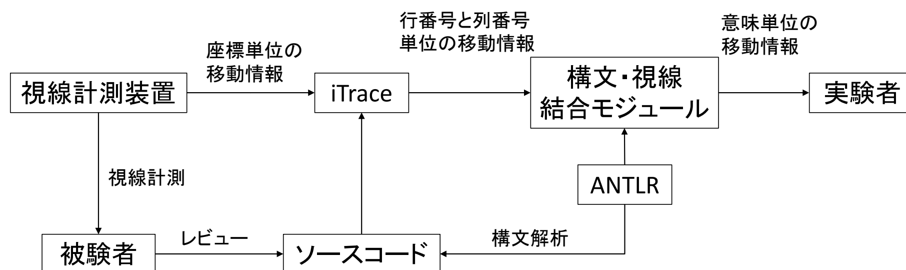

図2システム図

図において，まず開発者はディスプレイ上のソースコードに対してレビューする．レビュー中の開発者の視線移動は視線計測装置を使って計測し，視線移動は連続した座標単位の位置情報として出力される．iTraceは視線移動とソースコードを入力とし，行番号と列番号単位の移動情報を構文･視線結合モジュールに出力する．iTraceの出力例を表3に示す. 各行のtimeは視線を計測した時刻をUNIX時間(ms)で表したものである．xとyは視線の座標，lineとcolumnはソースコードにおける行番号と列番号である．ANTLRは構文解析器を作成するためのオープンソースソフトウェアである．以降，ANTLRによって生成した構文解析器をANTLRと呼ぶ．ANTLRによる構文解析の結果を利用し構文･視線結合モジュールは行番号単位の移動情報を意味単位の移動情報に変換する．

ANTLRで作成した構文解析器に対して，入力するソースコードの例を図3に，出力例を表4に示す. 表において，meaningは文法上の意味，textはソースコード上のテキスト, line, columnはソースコードにおける行番号と列番号，text_lengthは文字の長さである．meaningは一定の種類しかないため，異なるtextに対して同じmeaningが出ることがある．meaningの@[番号]は対応するtextを識別するためにある．例えば，表においてtext列の4~8行目にある"java",".","util",".","Random"は同じ

表 3 行番号と列番号単位の移動情報の例

time	x	y	line	column
1638509094839	751	655	35	48
1638509095090	645	552	29	34
1638509095121	618	546	29	32
1638509095152	611	545	29	31
1638509095182	604	545	29	30
1638509095214	595	547	29	29
1638509095246	592	549	29	28
1638509095277	589	551	29	28
1638509095310	581	555	29	27
1638509095341	568	563	30	28
1638509095388	558	569	30	27
1638509095421	554	569	30	26
1638509095451	547	569	30	25
1638509095483	537	566	30	24
1638509095513	530	565	30	23

qualifiedName（修飾名）という意味を持つ．しかし，9 行目以降に qualifiedName がある場合，4~8 行目にある qualifiedName とは異なり，@[番号] によって意味を識別した．図の 1 行目にある import は，表の 2 行目を見ると，importDeclaration という意味を持つことが分かる．

```
1   import java.util.Random;
2
3   public class Dog extends Thread {
4       private String name;
5       private double stride;
6       private int interval;
7
8       Random rand = new Random();
9
10      public Dog(String name, double stride, int interval) {
11          this.name = name;
12          this.stride = stride;
13          this.interval = interval;
14      }
15
16      @Override
17      public void run() {
18          int fine[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
19
20          for (int i = 0; i <= 600; i += (int) stride) {
21              boolean val = rand.nextInt(100) == 0;
22
23              if (val == true) {
24                  for (int l = 0; l < 10; l++) {
25                      fine[l]++;
26                  }
```

```
27                 }
28                 try {
29                     Thread.sleep(interval); // 処理を(約)1000
                           ms止める. 例外の可能性あり
30                 } catch (InterruptedException e) {
31                     System.out.println("エラーが起きたよ");
32                 }
33 
34                 if (i % 200 == 0) {
35                     System.out.println(name + "が" + i + "mに到達！");
36                 }
37 
38                 if (val == true) {
39                     System.out.print(name + "は絶好調\t(" + i + "m)");
40                 } else {
41                     System.out.print(name + "は不調\t(" + i + "m)");
42                 }
43 
44                 if (fine[0] > 0) {// 問題あり
45                     System.out.println(" ── 強化 × " + fine[0] + " ──");
46                 } else {
47                     System.out.println();
48                 }
49 
50                 for (int k = 0; k < 9; k++)
51                     fine[k] = fine[k + 1];
52                 fine[9] = 0;
53 
54                 if (fine[0] > 0) {
55                     i += stride * Math.pow(2, fine[0]);
56                     i -= stride;
57                 }
58             }
59         }
60 }
```

図 3 Dog.java

構文・視線結合モジュールはiTraceの出力とANTLRの出力を組み合わせることで，開発者の視線移動を意味単位の移動情報として実験者に出力し，本システムの出力にあたる．具体的には，iTraceは開発者の視線移動をソースコード上の行番号と列番号の移動情報として表2のように出力し，35行目の48列目，29行目の34列目，...，と，開発者の視線が移動していることが分かる．ANTLRは表4のように構文解析の結果を出力し，ソースコード上のテキストに対応する意味と，行番号と列番号を使った位置を表す領域（以降，行列番号領域と呼ぶ）が分かる．例えば，表の2行目にあるimportはimportDeclarationという意味を持つことが分かる．そして，text_lengthは文字の長さを意味し，列番号における幅を表すので，importはline，column，text_lengthから1行目の1~7列にあることが分かる．構文・視線結合モジュールはiTraceが出力した行番号と列番号の移動情報において，35行目の48列目が，ANTLRの各テキストに対応する行列番号領域のどれにあてはまるかマッチングを行い，意味単位の移動情報を出力する．

表 4 ANTLRで作成した構文解析器の出力例

meaning	text	line	column	text_length
importDeclaration@1	import	1	1	6
space	␣	1	7	1
qualifiedName@2	java	1	8	4
qualifiedName@2	.	1	12	1
qualifiedName@2	util	1	13	4
qualifiedName@2	.	1	17	1
qualifiedName@2	Random	1	18	6
importDeclaration@1	;	1	24	1
newLine	\r\n	1	25	2
newLine	\r\n	2	1	2
classOrInterfaceModifier@4	public	3	1	6
space	␣	3	7	1
classDeclaration@5	class	3	8	5
space	␣	3	13	1
classDeclaration@5	Dog	3	14	3
space	␣	3	17	1
classDeclaration@5	extends	3	18	7
space	␣	3	25	1

構文･視線結合モジュールの出力例を表5に示す. meaningは文法上の意味, textはソースコード上のテキスト, lineはtextの行番号, columnはtextにおける左端の列番号, text_lengthは文字の長さである. 開発者の視線がソースコード上の特定のtextにとどまると, 特定のtextに対応するmeaning,line,column,text_lengthは全く同じであるため, 表において同じ行を複数出力することになる. 冗長であるため, 重複回数をtime_countとし, 複数の同じ行を1行として出力する. ANTLRの出力 (表4) と構文･視線結合モジュールの出力 (表5) は似ているが, ANTLRの出力はソースコードの構文情報であり, 構文･視線結合モジュールの出力は文法上の意味とソースコード上の字句に基づいた開発者の視線情報である.

表5 構文・視線結合モジュールの出力例

meaning	text	line	column	text_length	time_count
space	␣	35	48	1	1
primary@282	interval	29	30	8	4
methodCall@279	(	29	29	1	1
methodCall@279	sleep	29	24	5	3
qualifiedName@285	InterruptedException	30	22	20	9
primary@278	Thread	29	17	6	8
indent	\t\t\t\t\t	25	1	20	10
primitiveType@113	int	18	9	3	1
indent	\t\t	12	1	8	3
indent	\t\t	11	1	8	4

5 ケーススタディ

本提案手法を用いることによって開発者の視線移動を意味単位の移動情報として分析できるか確認するためにケーススタディをする．

本システムの入力するソースコードの例を図3に，視線計測装置が出力した座標単位の移動情報の例を表4に示す．表において，meaning は文法上の意味，text はソースコード上のテキスト，line，column はソースコードにおける行番号と列番号，text_length は文字の長さである．

本システムの出力例として，視線移動をソースコード上にプロットし図4に，視線移動から生成した停留点単位の移動をソースコード上にプロットし図5に，座標単位の移動情報から変換した意味単位の移動情報を表6に示す．表6において，text はソースコード上のテキスト，meaning は text に対応する文法上の意味，line と column は text に対応する行番号と列番号，text_length は text の文字列長である．time_count は出力の重複回数である．

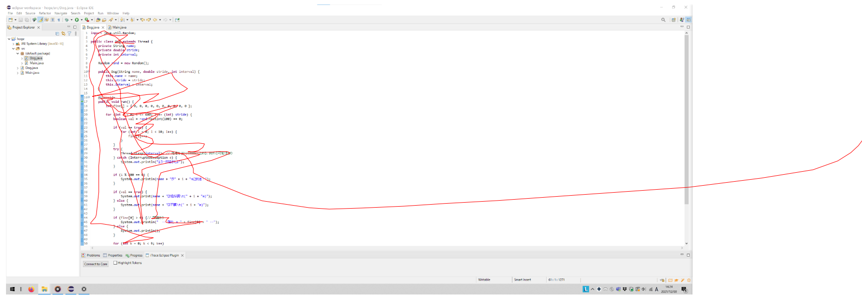

図4 視線計測装置で得たデータ

構文･視線結合モジュールは視線計測装置で得た座標単位の移動情報を表6にある meaning 列のように，意味単位の移動情報に変換できる．したがって，被験者の意味における理解パターンを分析できる．例えば，表6にある meaning 列より，被験者の視線は，スペース（space），primary，methodCall（メソッド呼び出し），...，というように移動していることが分かる．time_count は重複する回数であるため，被験者

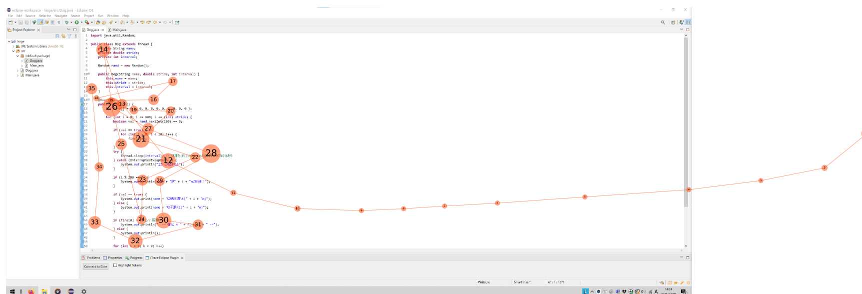

図5 停留点

表 6 意味単位の移動情報

meaning	text	line	column	text_length	time_count
space	␣	35	48	1	1
primary@282	interval	29	30	8	4
methodCall@279	(	29	29	1	1
methodCall@279	sleep	29	24	5	3
qualifiedName@285	InterruptedException	30	22	20	9
primary@278	Thread	29	17	6	8
indent	\t\t\t\t\t	25	1	20	10
primitiveType@113	int	18	9	3	1
indent	\t\t	12	1	8	3
indent	\t\t	11	1	8	4

がどの程度注視したかが分かる．time_count 列を見ると，9 回，8 回，10 回になっているところが特に多く，対応する meaning は，qualifiedName@285，primary@278，indent であり，注視していることが分かる．

また今後の予定として，パターンマイニングの手法を使うことで，意味単位の移動情報から被験者の意味における移動パターンを抽出する．例えば，methodCall から methodDeclaration への移動が多いなどが分かると考える．

異なるソースコードにおいて表6のように，各ソースコードに対応する意味単位の移動情報を生成する．意味単位の移動情報において，異なる名前であっても処理内容が同じであれば同じ意味になる．例えば，宣言している変数 A と B は両方とも，variableDeclaratorId となる．したがって，異なるソースコードにおいて，変数名，メソッド名，クラス名が異なっても意味単位の移動情報を使いソースコードの比較ができる．また，座標を使った従来の分析手法は処理構造，フォーマットが異なると，字句の位置関係が異なるため，ソースコード間の比較が難しい．しかし，意味単位の移動情報は，座標と関係がないため，処理構造，フォーマットが異っても，比較できる．したがって，従来手法と比べて意味単位の移動情報を使った提案手法はソースコード間の比較がしやすい．

6 おわりに

本研究では異なるソースコード間に対するレビューの比較分析を目的とし，視線移動を文法における意味単位の移動として分析する手法を提案した．提案システムは座標単位の移動情報を入力とし，意味単位の移動情報に変換して出力することで，研究者が開発者の理解パターンを分析することを支援する．視線計測装置は被験者の視線移動を座標単位の移動情報として出力し，本システムは以下の処理によって意味単位の移動情報に変換する．

処理1: iTraceを使うことによって視線計測器から得た座標単位の移動情報をソースコード上の行番号，列番号単位の移動情報に変換する．

処理2: 構文･視線結合モジュールとANTLRを使うことによって行番号，列番号単位の移動情報を字句単位の移動情報に変換する．

処理3: 構文･視線結合モジュールとANTLRを使うことによって字句単位の移動情報に対して文法上の意味をマッピングする．

開発者の視線移動を座標単位の移動情報として分析する従来手法は，視線の移動方向と停留時間に関する開発者の理解パターンを抽出している．しかし，開発者の視線移動を意味に基づいて分析することで開発者の理解パターンをより詳細に知ることができる．座標単位の移動情報を使った従来手法において，開発者の視線移動を意味に基づいて分析するには以下の問題がある．

- 研究者は座標に対応するソースコードにおける文法上の意味を手動で生成するため，研究者に負担がかかっている．
- 研究者が手作業で意味を生成しているため，複数の異なるソースコードに対して意味のマッピングが難しい．

本システムは構文解析を利用することで座標に対応する意味を自動で生成し，研究者の負担を減らすことができる．また，意味を自動で生成するため，複数のソースコードに対して意味のマッピングができる．本システムが出力する意味単位の移動情報は，異なるソースコード間で意味に関する理解パターンを比較しやすい．そのため，複数のソースコードで比較し共通する理解パターンを抽出しやすく，従来難しかったレビュー作業者自身の理解パターンを抽出しやすい．優れた開発者の理解パターンを抽出し，初心者や学習者に伝達することで，ソースコードを理解する効率が向上すると考えられる．

本研究の今後の課題として，提案手法を使った以下の分析を行うことである．

- 複数のソースコードに共通する開発者の理解パターンを抽出．
- 共通する理解パターンを抽出することで，開発者自身の理解パターンを抽出

- 熟練者と初心者に対する意味における理解パターンの違い

謝辞

本研究を進めるにあたり，多くの方々にご指導，ご協力を頂きました．この場を借りてお礼申しあげます．

本研究を直接指導して頂いた，指導教員の上野秀剛准教授には，常日頃より，開発したシステムが必要とする機能，スライドの作り方，論文の書き方について適切なご助言を頂きました．心より感謝申し上げます．

査読教員である内田眞司教授には，中間発表で大変貴重な意見を頂きました．心より感謝申し上げます．

参考文献

[1] Florian Hauser, Stefan Schreistter, Rebecca Reuter, Jurgen Horst Mottok, Hans Gruber, Kenneth Holmqvist, and Nick Schorr, "Code Reviews in C++: Preliminary Results from an Eye Tracking Study", ETRA '20 Short Papers, (2020).

[2] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello, "Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers", ICSE 2014, pp. 390–401, (2014).

[3] Martha E. Crosby and Jan Stelovsky, "How Do We Read Algorithms? A Case Study", *Computer*, Vol. 23, No. 1, pp. 24–35.

[4] Ian Bertram, Jack Hong, Yu Huang, Westley Weimer, and Zohreh Sharafi, "Trustworthiness Perceptions in Code Review: An Eye-Tracking Study", ESEM '20, (2020).

[5] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm, "Eye Movements in Code Reading: Relaxing the Linear Order", ICPC '15, pp. 255–265, (2015).

[6] K R Chandrika, J Amudha, and Sithu D Sudarsan, "Recognizing eye tracking traits for source code review", In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, (2017).

[7] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, "Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement", pp. 133–140, (2006).

[8] Norman Peitek, Janet Siegmund, and Sven Apel, "What Drives the Reading Order of Programmers? An Eye Tracking Study", ICPC '20, pp. 342–353, (2020).

[9] Drew T. Guarnera, Corey A. Bryant, Ashwin Mishra, Jonathan I. Maletic, and Bonita Sharif, "ITrace: Eye Tracking Infrastructure for Development Environments", ETRA '18, (2018).

付録

A　Dog.java の構文木

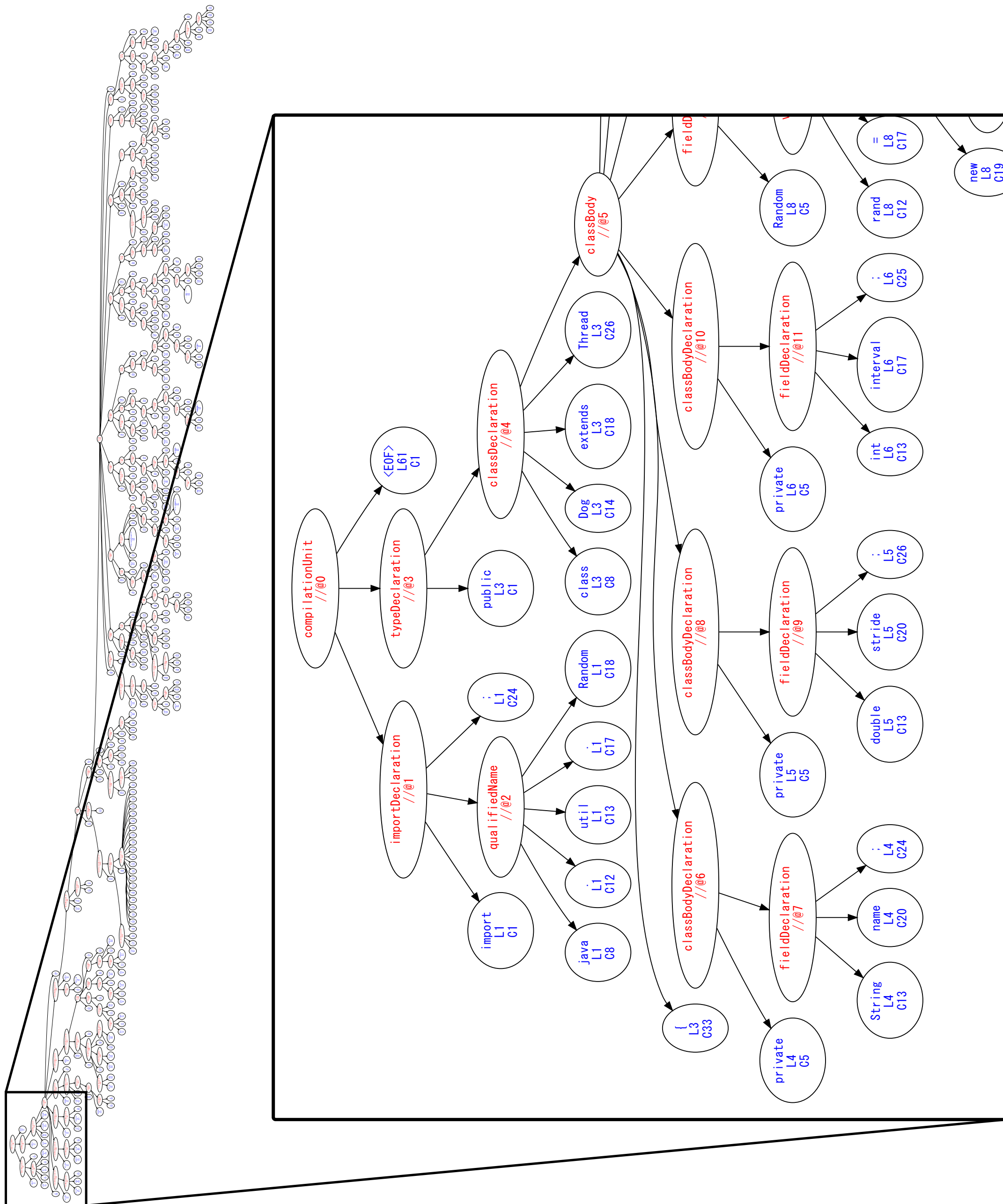

図 6 構文木