あるトークンを消費した構文規則番号に基づく構文木の安定性および IDE によるエラー報告の改善

岩田 風多　山崎 徹郎　千葉 滋

本論文では，小さな編集を行った際の統合開発環境のエラー報告の大幅な変化を防ぐ手法を提案する．この手法では，編集の前後で構文木が大きく変化する場合は，新たに得られた構文木を却下し，編集前に得られていた構文木に基づいてエラーを報告する．構文木の変化の程度を計算するために，トークンを消費した構文規則番号に基づく指標を用いる．この手法によって，小さな編集を行っただけにもかかわらず，表示されるエラーが大幅に変化する場合があるという問題を防げることが期待される．提案手法がエラー報告を安定させることを確かめる実験のため，細粒度の編集履歴を収集する実験を計画している．

1　はじめに

現代のソフトウェア開発において，IDE（統合開発環境）は開発者の生産性向上に不可欠なツールである．IDE は，開発者がコードを編集している最中にリアルタイムでエラーや警告を表示することで，開発プロセスを支援する．

編集中のコードはしばしば構文的に不完全であるが，IDE は構文エラーリカバリ機能により有益なエラーを表示することができる．有益なエラーを表示するためには構文木の解析が必要であるが，構文的に不完全なコードでは構文エラーのために構文木の生成に失敗する．構文エラーリカバリを備えた IDE は，不完全なコードでもトークンを挿入・削除・置換することで構文解析を成功させ構文木を生成できるため，有益なエラーを表示できる．このエラー報告機能により，開発者は問題を迅速に把握し修正することができるため，開発効率の大幅な向上が実現されている．

しかしながら，現在の IDE におけるリアルタイムエラー報告では，小さな編集を行っただけにもかかわらず，表示されるエラーが大幅に変化することがある．例えば，単一の文字を削除しただけで，エラーの種類や数が大きく変わり，編集箇所とは無関係な部分にエラーが報告されるようになる場合がある．このような予期しないエラーの変化は，開発者の認知負荷を増加させ，開発効率を低下させる要因となる．

本論文では，IDE におけるリアルタイムエラー報告の安定性を向上させる手法を提案する．具体的には，編集後のトークン列を構文解析して得られた構文木が解析前のトークン列ないし編集前のトークン列を構文解析して得られていた構文木から大きく異なる場合は，新たに得られた構文木を却下し，編集前に得られていた構文木を採用する．これにより，小さな編集による大きな構文木の変化を抑制する．この手法により，開発者は編集の影響を直感的に理解でき，より効率的な開発が可能になると期待される．提案手法では，構文解析時に各トークンを消費した構文規則の情報 (token interpretation) を記録し，この情報を用いて構文木の変化の程度を定量化する．

本論文の貢献は，主に 3 点ある．まず，IDE におけるエラー報告の不安定性の問題を具体的に特定し，その原因を構文エラーリカバリによる構文木の大幅

Improvement of Stability of Syntax Trees and Error Reporting by IDE Based on the Syntax Rule Number Consuming a Token

Futa Iwata, Tetsuro Yamazaki, Shigeru Chiba, 東京大学大学院情報理工学系研究科創造情報学専攻, Graduate School of Information Science and Technology, The University of Tokyo.

```
1 import { defineConfig } from "vite";
2 export default defineConfig({
3   resolve: {
4     ailas: {
5       "@": path.resolve(__dirname, "src
            "),
6     },
7   },
8 });
```

プログラム 1　TypeScript コード

```
1 import { defineConfig } from "vite";
2 export default defineConfig(
3   resolve: {
4     ailas: {
5       "@": path.resolve(__dirname, "src
            "),
6     },
7   },
8 });
```

プログラム 2　編集後の TypeScript コード（{を削除）

図 1　編集前の IDE によるエラー報告

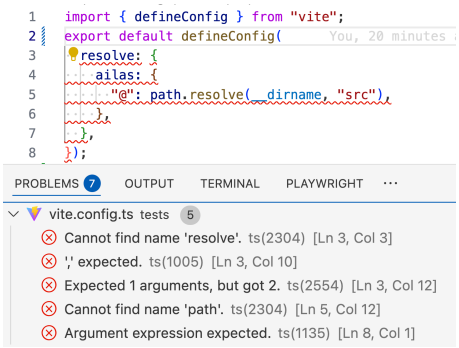

図 2　編集後の IDE によるエラー報告

な変化に求めている点．次に，構文木の変化を定量的に測定する token interpretation という指標の提案．最後に，編集前の構文木の情報を活用してエラー報告の安定性を向上させる手法の提案．

2　IDE におけるエラー報告の不安定性

現代の IDE では，プログラムを書いている最中にリアルタイムでエラーが報告される．これにより開発者は，プログラムコードを編集しているその場で構文エラーや型エラーに気づくことができ，開発効率の大幅な向上を見込むことができる．

IDE がエラーを報告する具体的な例として，プログラム 1 に示す TypeScript コードを IDE で開いた場合を考える． 例えば図 1 に VS Code †1 によるエラー報告の例を示す．また，以下に報告されているエラーの一覧を示す．

- `ailas` が `alias` の typo であるというエラー
- `path` が未定義であるというエラー

これらのエラーは，プログラマにとって有益な情報でありコードの品質向上に寄与すると考えられる．

†1　https://code.visualstudio.com/

IDE によるエラー報告は有用であるが，その一方で，編集の仕方によってはエラー報告が役に立たないこともある．コードの構造を担う文字を誤って削除して多くのエラーが報告されるような場合もこれにあたる．コードの構造を担う文字を誤って削除しても，理想的には，編集した箇所以外で新しく修正すべき箇所が増えたり減ったりすることはないはずであるが，実際にはその影響がプログラマの意図しない形でコードの他の部分に波及する場合がある．

例えばプログラム 1 の TypeScript コードから 2 行目末尾の { を削除した場合，VS Code によって報告されるエラーは大きく変化してしまう．削除後のコードをプログラム 2 に示す． 例えば図 2 に VS Code によるエラー報告の例を示す．また，以下に報告されているエラーの一覧を示す．

- `resolve` が未定義であるというエラー
- `resolve` の直後に : は期待されないという構文エラー
- 8 行目冒頭に } は期待されないという構文エラー

```
1 import { defineConfig } from "vite";
2 export default defineConfig(
3   resolve,
4   {
5     ailas: {
6       "@": path.resolve(__dirname, "src
            "),
7     },
8   },
9 );
```

プログラム 3　IDE による解析結果（エラー回復後の構文木）

- `defineConfig` の引数の数が型と合わないというエラー
- `path` が未定義であるというエラー

このようなエラー報告の変化は，構文エラーリカバリが実際に加えられた編集とは異なる部分でトークンを挿入・削除・置換し，出力される構文木が加えられた編集以上に変化することで起こる．構文エラーリカバリでは，入力されたトークン列がその言語として受け入れられない場合に，トークン列に最小限の変更を行い，構文木を出力する．しかしながらその方式によっては，入力されたトークン列が少し変わるだけでその編集とは異なる部分でトークンを挿入・削除・置換し，大きく異なる構文木が出力される場合がある．このように出力された大きく異なる構文木をさらに分析することで得られる未定義エラーや型エラーは，やはり大きく変化すると考えられる．

プログラム 3 に，プログラム 2 を TypeScript トランスパイラ tsc でトランスパイルして得られる JavaScript コードを示す．tsc は不完全なコードであっても構文エラーリカバリによって JavaScript の生成とエラー報告を行うことができる．IDE においても同様の方法でエラー報告を行うものと考えられる．

プログラム 3 から推測される，プログラム 2 で行われた構文エラーリカバリを以下に示す．

- `resolve` の直後にあった`:` が`,` に置換された
- プログラム 2 の 8 行目の冒頭にあった`}`が削除された

これらの構文エラーリカバリがもたらした構文木の変化を以下に示す．

- `resolve` が，`defineConfig` の第一引数に渡される変数読み出しに変化している
- `resolve:` 以降のオブジェクトリテラルによる式が，`defineConfig` の第二引数に渡されている

これらの構文木の変化が図 1 と図 2 の間のエラー報告の変化を引き起こす原因である．これらの構文木の変化がもたらしたエラー報告の変化を以下に示す．

- `resolve` がオブジェクトのキーから変数読み出しに変化したため，直後の`:` が期待されないという構文エラーが報告され，`resolve` が未定義であるというエラーが新たに報告された
- `resolve:` 以降のオブジェクトリテラルによる式が `defineConfig` の第二引数に渡されているため，プログラム 2 の 8 行目の冒頭に`}`は期待されないという構文エラー，`defineConfig` の引数の数が型と合わないというエラーが新たに報告され，`ailas` が `alias` の typo であるというエラーが報告されなくなった．

このようなエラーの変化は，開発者の認知負荷を増加させ，開発効率を低下させる要因となる．構文エラーが変化しているのは，それが構文エラーリカバリで得られた構文木と元のトークン列を比較して出力されているからだと考えられる．未定義エラーや引数の型エラーなどのエラーは，構文解析器が出力した構文木をさらに分析することによって出力されるため，その構文木が変化したことにより変化していると考えられる．

3　Token Interpretation に基づく構文木安定化手法

本論文では，token interpretation の変化の程度によって，新たに得られる構文木を採用あるいは却下する手法を提案する．提案の全体図を図 3 に示す．提案手法では，編集前のトークン列および編集後のトークン列から得られた構文木を編集後のトークン列と比較し，編集後の構文木の方が変化が小さければ採用し，そうでなければ却下して，代わりに編集前の構文木を解析結果として採用する．構文木がどれほど異なるかを評価するために，我々が新たに設計した指標 token interpretation を使用する．

この手法によって，エラー報告のために利用する構文木が編集前の構文木と大きく変化することを防ぎ，エラー報告の変化を抑えることが期待される．編集後のコードを構文解析して得られた構文木の変化が大きければ，却下して編集前の構文木を再利用するため，エラー報告も編集された箇所くらいしか変化しないことが期待できる．編集後のコードを構文解析して得られた構文木が採用されるならば，構文木の変化は小さいはずであり，エラー報告の変化も小さいことが期待される．

3.1 Token Interpretation

構文木がどれくらい変化したかを測る指標として，token interpretation を導入する．ある構文解析における token interpretation とは，あるトークンが構文解析中にどの構文規則によって消費されたかを示す．例えば変数宣言文 `const x = 1;` を解析する場合，`const` トークンは TypeScript の標準的な文法では変数宣言規則によって消費され，`x` は束縛識別子規則によって消費され，`=` は変数初期化規則によって消費される．このように各トークンがどの構文規則によって消費されたかの情報が token interpretation である．

図 4 に token interpretation によって構文木の変化の程度を計算する例を示す．図左は編集前のトークン列の一部とその token interpretation を示し，図中央は編集後のものを示す．図右上は編集後のトークン列に既存の構文エラーリカバリを適用することで得られるトークン列およびその token interpretation を示し，図右下はエラー報告を安定させるために我々が期待する構文エラーリカバリの結果得られるトークン列およびその token interpretation を示す．token interpretation が途中で変化する `resolve` トークンのみ，右上に token interpretation を表示している．token interpretation の異なるトークンはそのテキストが同じだったとしても異なるトークンだとみなすと，従来の構文エラーリカバリによるトークン列ではその解析元であるトークン列からの変化は 2 トークンであるが，期待するエラーリカバリによるトークン列では変化は 1 トークンである．図中では下線が引かれたトークンが編集後のトークン列から変化したものである．`resolve` については，テキストは変化していないが token interpretation が異なるため，変化していると考える．そのため下線も token interpretation に引かれている．このように，token interpretation を使用することで構文木の変化の程度を測ることができる．

3.2 候補の中から出力する構文木を選択する

編集後のコードを構文解析して得られた構文木と編集前の構文木のそれぞれについて，編集後のトークン列との差分を計算し，小さい方を採用する．そのため，まず編集後のトークン列の各トークンに期待する token interpretation を決定する．構文木の変化の程度を計算するためには，第 3.1 節で示したように，それぞれのトークンの token interpretation が付与されている必要がある．

編集によって変化していないトークンの token interpretation は，編集前の構文解析から変化しないことを期待する．実装上は，同じトークンでも編集の前後で異なるオブジェクトとなるため，同一のトークンを特定するには対応を計算する必要がある．IDE が提供する編集情報を参照することで，この対応関係は簡単に計算することができる．

編集によって変化しているトークンの token interpretation は，ワイルドカードとする．構文木の変化の程度を計算する際に，ワイルドカードは任意の token interpretation と一致すると考える．これは，プログラマが意図的に編集したトークンの周辺では構文木が変化することが期待されるためである．

token interpretation を考慮した差分の計算にあたっては，2 つのトークン列に対して，挿入・置換・削除による編集距離を求める．ただし，期待ないし記録されている token interpretation が異なる場合は，異なるトークンとみなす．付与されている token interpretation がワイルドカードである場合，テキストが一致していれば同じトークンとみなす．これにより，表面的には同じトークンであっても，構文的役割が異なる場合には適切にペナルティが課されると期待される．例えば，`x` というトークンが変数参照から変

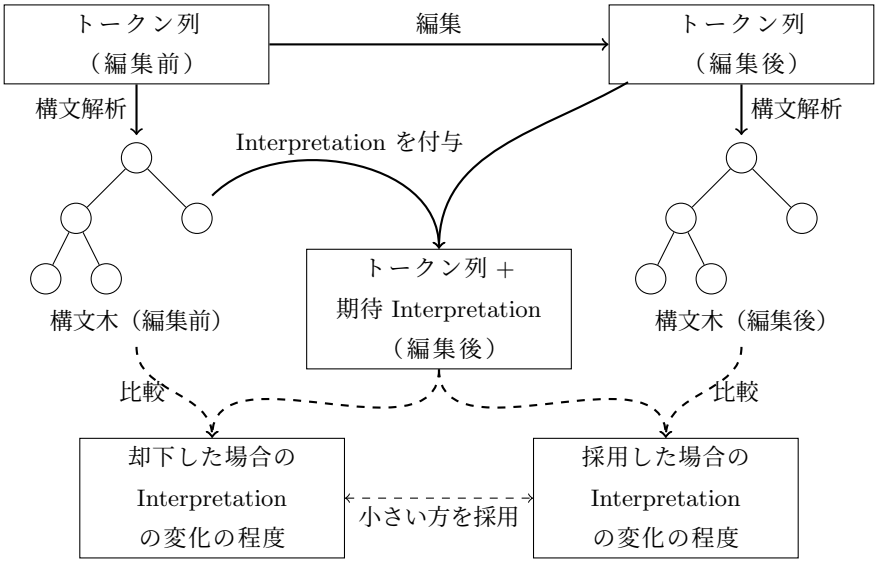

図 3　提案手法の全体図

図 4　Token Interpretation の変化の例

数宣言に変わった場合，同じ文字列であっても異なる token interpretation を持つため，異なるトークンとして扱われる．

4　細粒度編集履歴収集実験

提案手法がエラー報告を安定させることを確かめる実験のため，細粒度の編集履歴を収集する実験を計画している．提案手法によって構文木を得るためには，編集前の構文木の情報や，編集前後のトークンの対応が必要であるが，既存のデータではこれらの情報を得ることができない．既存の多くのデータセットには，完全なコードしか含まれないため細粒度の編集情報が残っておらず，公開されているリポジトリからデータセットを作成する場合にも同様の問題がある．

我々の計画する実験では，参加者にプログラミング課題を行ってもらい，その編集情報を収集することでデータセットを作成する．細粒度の編集情報を収集するため，プログラミング課題を行う間に IDE が言語サーバー送る編集情報を記録する．これによって収集されたデータに匿名化加工を施し，細粒度の編集情報を含むデータセットを作成する．

この実験に際しては，検討すべき点がいくつか存在した．まず，細粒度の編集情報を収集する方法を検討した．次に，どのようなプログラミング課題における編集情報を収集するか検討した．最後に，収集したデータのプライバシー保護と倫理的な取り扱いについて考慮した．

細粒度の編集情報を収集するために，簡単な IDE のプラグインを作成する．IDE が言語サーバーに提供しているインターフェースを前提とし，同じ

ものを利用して実装する．具体的には VS Code Web Extension を実装する．この拡張機能は，Language Server Protocol (LSP) で定義されている `textDocument/didChange` イベントを監視し，編集の開始位置，削除される文字数，挿入される文字列の情報をタイムスタンプとともに記録する．

プログラミング課題としては，単に新しくプログラムを書くものだけでなく，意図的にバグが含まれた既存のプログラムを修正するものや，新機能を既存のプログラムに追加するものも用意する．既存のプログラムの編集では，編集されない領域が大きく，そこでのエラー報告が得られることが期待できる．これにより，編集前後でのエラー報告の変化を詳細に観察することができる．

提案手法の試験に要求される細粒度の編集情報は，情報倫理的に扱いが難しい．プライバシーに障る情報が一度でも書かれるとデータに含まれてしまうため，それを取り除く必要がある．適切な匿名化処理と参加者の同意取得を行った上でデータを収集する．

5 関連研究

プログラムに対して報告されるエラーを改善する研究は多々あるが，IDE におけるリアルタイムエラー報告の不安定性に直接的に取り組んだものは我々の調査の限りは見られない．IDE に表示されるエラーを改善する研究はあるが，エラーの変化を抑え安定させるものは見られず，しかも言語依存性の低いものは特に少ない．また，特に IDE の編集に伴うエラー報告の不安定性に焦点を当てるものも見られない．以下に代表的な研究を挙げる．

Layman ら [5] は，開発者が興味を持ちそうなエラーだけを表示する手法を提案している．この研究では，エラーの重要度，開発者の現在の作業コンテキストとの関連性，プログラミング環境での開発者の行動に基づいてエラーの関心度を推定し，関心の高いエラーのみを表示する MimEc システムを開発している．この研究はエラー報告の不安定性に直接取り組むものではない．

Barik ら [2] は，エラーを分類してそれぞれに適した表示をする手法を提案している．この研究では，コンパイラエラー通知とその解決方法について，インタラクション中心のアプローチを通じて開発者がより効果的に理解し解決できるよう支援している．エラー通知とその解決方法のための新しい分類体系を提案し，またこれらの分類体系に基づくエラーの可視化を行っている．この研究は出力されたエラーを分類するものであるが，出力されるエラー自体の不安定性に直接取り組むものではない．

Kohn と Manaris [4] は，Python の教育環境において強化されたエラーメッセージを持つ IDE を開発し，初心者プログラマにとってより理解しやすいエラー報告を実現している．構文エラーについては，扱っているプログラムを分析して統計的に尤もらしいエラーメッセージを表示している．例えば，`form caption` と書かれていたら本来は `form_caption` ないし `form.caption` などであろうとし，空白部分にその旨のメッセージを表示する．この研究はエラー報告の不安定性に直接取り組むものではない．

コンパイラが出力するエラーを改善する研究も多々ある．Becker ら [3] は，構文解析器を改善することでコンパイラによる構文エラーを改善する研究を包括的に調査している．この研究では，テキストベースのプログラミングにおけるエラーメッセージ研究の全体像を示し，半世紀にわたる研究成果を体系化している．しかし，IDE 上で編集中の未完成なコードに対応するものではない．このレビュー論文は IDE 上でのリアルタイムなエラー報告について Live Compilation という言葉で問題を提起しているが，この問題に特に取り組んだ研究を挙げていない．

Algaraibeh ら [1] は，スコープに注目した新たな構文解析手法によってエラーを改善する研究を行っている．この研究では，学生プログラマのためのコンパイラ構文エラーメッセージを強化する解析技法を提案している．この研究は構文エラーの報告の改善を行っているが，それ以外のエラーについては言及していない．

6 まとめ

本論文では，IDE におけるリアルタイムエラー報告の不安定性を解決するため，トークンごとに消費し

た構文規則（token interpretation）を記録し，編集前後の構文木の変化を定量化して安定したエラー報告を実現する手法を提案している．今後は，細粒度編集履歴の収集実験を通じて本手法の有効性を検証し，より直感的で信頼性の高いエラー報告の実現を目指す．

参考文献

[1] Algaraibeh, S., Jeffery, C., Soule, T., and Dousay, T.: A Parsing Technique for Enhancing Compiler Syntax Error Messages for Student Programmers, *2024 IEEE Frontiers in Education Conference (FIE)*, 2024, pp. 1–7.

[2] Barik, T., Witschey, J., Johnson, B., and Murphy-Hill, E.: Compiler error notifications revisited: an interaction-first approach for helping developers more effectively comprehend and resolve error notifications, *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, New York, NY, USA, Association for Computing Machinery, 2014, pp. 536–539.

[3] Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P.-M., Pearce, J. L., and Prather, J.: Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research, *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, New York, NY, USA, Association for Computing Machinery, 2019, pp. 177–210.

[4] Kohn, T. and Manaris, B.: Tell Me What's Wrong: A Python IDE with Error Messages, *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, New York, NY, USA, Association for Computing Machinery, 2020, pp. 1054–1060.

[5] Layman, L. M., Williams, L. A., and St. Amant, R.: MimEc: intelligent user notification of faults in the eclipse IDE, *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '08, New York, NY, USA, Association for Computing Machinery, 2008, pp. 73–76.