模範解答を用いたコンパイルエラー箇所指摘の高精度化

中井 亮佑
大分大学
v1858239@oita-u.ac.jp

紙名 哲生
大分大学
kamina@oita-u.ac.jp

要旨

C 言語は今でも多く用いられ，プログラミングの入門用の言語としてもよく使われる．しかし，プログラミングの初学者にとって，コンパイラが表示するエラーメッセージは理解しにくい．本論文では，既存のコンパイラが，ブロック構造の閉じ括弧 '}' を忘れた際に誤りの原因となっている行番号を正しく指摘することができない点を改善するために，模範解答と初学者のプログラムを比較し，その差異によってプログラム中のブロック構造の閉じ括弧 '}' 忘れとその位置を指摘する方法を提案する．具体的には，構文解析器が処理するソースコード中の構文要素をイベントとして抽象化し，模範解答と初学者のプログラムのイベント列を比較する．提案手法を評価するために，コードベースを収集し，それらのブロック構造の閉じ括弧 '}' を筆者が適当に削除し，そのコンパイルエラーとその位置を正確に検出できるかを確認した．その結果，収集したコードベースの約半数のコンパイルエラーとその位置を検出することができた．また，被験者を用いた実験も行い，イベント列の抽象化の妥当性を確認した．

1. はじめに

C 言語は，現代でも幅広く使用されている言語であり，プログラミング初学者が最初に学ぶ代表的な言語のひとつである．C 言語初学者がプログラムを作る上で必ず使うものの一つにコンパイラがある．コンパイラはプログラム中の様々な問題に対して，エラーメッセージを表示する．しかし，そのエラーメッセージは初学者にとっては理解が難しいことが多い．例えば，ブロック構造の閉じ括弧 '}' を忘れたとき，既存のコンパイラでは誤りの原因となっている行番号を正しく指摘することはできない．エラーメッセージの理解が難しいことで思うようにプログラムが書けず，初学者の学習意欲をそぐ可能性がある．

本研究では，コンパイルエラーの箇所を適切に表示することで，経験の浅い初学者のプログラミング学習を支援する．本研究で想定している「初学者」は，プログラミングを初めて学ぶ人である．例えば，大学など学校のプログラミングの授業を初めて受ける学生や，入門書を買ってこれからプログラミングを始める人などを想定している．

具体的な手法として，模範解答と初学者のプログラムを比較し，その差異によってプログラム中のブロック構造の閉じ括弧 '}' 忘れとその位置を指摘する方法を提案する．具体的には，構文解析器が処理するソースコード中の構文要素をイベントとして抽象化し，模範解答と初学者のプログラムのイベント列を比較する．

提案手法を評価するために，コードベースを収集し，それらのブロック構造の閉じ括弧 '}' を筆者が適当に削除し，そのコンパイルエラーとその位置を正確に検出できるかを確認した．その結果，収集したコードベースの約半数のコンパイルエラーとその位置を検出することができた．また，被験者を用いた実験も行い，イベント列の抽象化の妥当性を確認した．

本論文の構成は次のとおりである，まず 2 節で関連研究について紹介し，3 節で本研究の提案について詳細に述べる．4 では，本研究で作成したツールの評価を行いその結果を考察する，最後に 5 節で結論と今後の課題を述べる．

2. 関連研究

C 言語からプログラミングを始める初学者のために，コンパイルエラーを分かりやすく表示する静的解析ツールとして，C-Helper [1] がある．従来のコンパイラでは，コンパイルエラー時のメッセージが初学者にとって理解が難しいこと，解決策が書かれていないことなど，プログラミング初学者向けではない点があった．C-Helper では，初学者が陥りがちな間違いを検出し，分かりやすいメッセージを表示する．このツールにより検出できる問題は，文献 [1] によると，インデント乱れ・`char` 型変数への文字配列の代入・`printf()` のパラメタ不整合・`return` 文の不足・関数定義の余分なセミコロン・構造体宣言のセミコロン不足・動的に確保した配列に対する `sizeof` の使用があげられる．

しかし，C-Helper では，`for` 文，`while` 文，`if` 文などを記述する際にありがちな閉じ括弧 '}' 不足を検出することはできない．この閉じ括弧不足は，既存のコンパイラが正確にエラー箇所を指摘しにくい問題の一つである．これらの点で，本研究は C-Helper とは異なり，また，初学者の支援にもなる．

ソースコードを解析する方法として，コードクローン [2] がある．コードクローンとは，ある特定の字句などをクローンとし，ソースコードにそのクローンと類似したものが含まれているかを検出する手法である．具体的な手法としては，文字列レベルのクローンを作成する手法 (手間がかかり，大規模なプログラムの解析には向かないためツールはない)・プログラムの字句解析を用いて字句をクローンとし，クローン解析する手法・さらに粗く行ごとを，ハッシュ関数を用いて一定の長さの文字列に圧縮し，その列をクローンとしクローン発見問題を解く手法・プログラム中の文を，識別子などをパラメタ化したクローンとし，クローン発見を行う手法・関数や手続き，クラス定義全体をクローンとし，等価な要素対を見つけることでクローン発見問題を解く手法がある．

本研究は，3 節で述べるとおり，ソースコードを抽象化したうえで模範解答とユーザのプログラムを比較するという点において，様々なコードクローン検出手法と通ずるものがある．しかしながら，本研究で行う比較方法はエラー箇所検出に特化したものであり，クローンを検出するためのものではない．本研究の提案をコードクローン検出に用いるのが難しいのと同様に，既存のコードクローン検出手法を本研究の用途に用いるのもまた難しいと考えられる．

C 言語初学者向けのインテリジェントな指導システムとして，C-Tutor[3] がある．C-Tutor は，サンプルプログラムから，リバースエンジニアリングによりプログラマの意図をプログラム記述（program description）として抽出する．その後，知識ベースのプログラム解析を行い，入力されたソースコードを実行しながらプログラム記述中の満たされていない目標を調べ，その目標を実装するプランの集合を知識ベースに問い合わせるとともに，プランとソースコードの違いを検出し，初学者に報告する．

この手法は，プログラマの意図をサンプルプログラムとして与える点は本研究と同じであるが，テストケースを予め準備しておく必要があり手間がかかる．また，コンパイル・実行可能なソースコードが対象であり，構文エラーを対象としている本研究とは適用範囲が異なる．

静的解析を用いて初学者向けのプログラミング支援を行う手法として，初学者向けの Java の静的解析フレームワーク [4] がある．この手法は，主にロジック上の誤りの検出や品質のチェックを行うために用いられる．具体的な手法は，模範解答と初学者のプログラムを，正規化したうえで構造の類似性（ループ構造，代入やメソッド呼出の順序など）の比較を行う．また，ソフトウェアメトリクスの計測も行う．

しかしこの手法は，ギャップを埋める問題を前提としており，最初から自由にソースコードを書かせるタイプの問題に同様の類似性の解析が適用できるかは不明である．

開発されたプログラムがコーティング規約を満たすか確認するためのツールとして，CX-Checker[5] がある．CX-Checker は，まずソースコードを構文解析し XML で表現する．ユーザはそれをもとに，XPath や DOM 形式でルールとして与えられたコーティング規約をもちいてプログラム中の規約違反を検出する．

このツールは，構文エラーや初学者の陥りがちなエラーを検出するためのものではないが，ルールの書き方は汎用的なので，模範解答の意図をルールとして記述するといった使い方ができる可能性はある．しかし，閉じ括弧忘れのような初歩的なエラーまで検出できるかは不明である．また，出題意図はルールとして準備しなければならず，模範解答をそのまま用いることができない点で本研究とは異なっている．

3. 提案手法

3.1. 前提

本節では，初学者の陥りがちなコンパイルエラーのなかでも，for，while，if 文などのブロック構造の閉じ括弧 '}' 不足の位置を適切に検出する手法を提案する．この手法では，プログラミング初学者を対象に，C 言語のコンパイルエラーを検出するために，ユーザのプログラムと模範解答のプログラムを用いる．この手法を用いる状況としては，学校の講義や，模範解答が用意されているようなプログラミングの教本や問題集を解くような場面を想定している．

また，この手法は，プログラム中にある程度の深さのネストがある場合に閉じ括弧を書かなかった場合（あるいは編集中にうっかり消してしまった場合）など，構文エラーはあるものの構文解析器が「それっぽい」構文木を作ることができた場合を想定している．意味のある構文木がそもそも作られない場合もあるため，本手法は既存のコンパイラとの併用も前提としている．

既存の IDE の中には，閉じ括弧 '}' を自動補完するものもある．しかし，初学者が IDE を使用しない場面 (いきなり IDE を使わせない授業も根強く残っている) も想定される．また，編集中に閉じ括弧 '}' をうっかり消してしまう事態も想定されるが，既存の IDE ではそれを指摘することはできない．

3.2. アプローチの概要

本手法は，文献 [6] に記述された手法を拡張したものであるため，まずはそこで採用されているアプローチの概要を説明する．まず，ユーザのプログラムと模範解答のプログラムをそれぞれ構文解析する．そして，if，for，while の各抽象構文木 (AST) ノードを深さ優先で訪問して何らかの処理 (例えばエラーメッセージの収集) を行うプログラムのそれぞれのノードへの訪問 (visit) と退去 (leave) を記録したイベント列[1]をそれぞれ作成・比較し，そのイベント列の違いで閉じ括弧の不足を検出する．

イベント列の生成手順は次のとおりである．まず，プログラムの AST を visitor パターンで辿る．訪問したノードが if 文だった際には，if 文の開始であることと

[1]各ノードへの訪問や退去を，AST 探索中に遭遇する事象としてイベントと呼んでいる．以後，イベントの出現順序をプログラムの構文要素に対応させ，それぞれを「開始」「終了」と呼ぶ．

```
1  #include <stdio.h>
2  void main() {
3    int i, j, k;
4    for (i = 1; i <= 10; i++) {
5      for (j = 0; j < 10; j++) {
6        printf("%d", j);
7      }
8      if (i > 10) {
9        printf(",");
10     }
11   }
12   k = 0;
13   while (k < 10) {
14     printf("%d", k);
15     k++;
16   }
17 }
```

図 1. 模範解答のプログラム例

その if 文の出現順序を表すイベントを生成し，イベント列に追加する．そのノードから leave するときには if 文の終了を表すイベントを生成し，イベント列に追加する．for 文や while 文も同様にしてイベント列を作成する．イベントの種類は以下のとおりである．

- i (if 文の開始)
- iend (if 文の終了)
- f (for 文の開始)
- fend (for 文の終了)
- w (while 文の開始)
- wend (while 文の終了)

この操作をユーザと模範解答のプログラムで行い，それぞれのイベント列を生成する．

例として，図 1 の模範解答に対して，ユーザのプログラムが図 2 に示すものであった場合について説明する．ユーザのプログラムでは，4 行目から始まる for 文の閉じ括弧が不足している．gcc などの既存のコンパイラでは，これは main 関数の閉じ括弧が不足していると検出されるが，このアプローチでは以下のようにイベント列

```
1  #include <stdio.h>
2  void main() {
3    int i, j, k;
4    for (i = 1; i <= 10; i++) {
5      for (j = 0; j < 10; j++) {
6        printf("%d", j);
7      }
8      if (i > 10) {
9        printf(",");
10     }
11   k = 0;
12   while (k < 10) {
13     printf("%d", k);
14     k++;
15   }
16 }
```

図 2. ユーザのプログラム例

を比較することにより，12 行目の while 文の前に閉じ括弧が不足しているとして検出することができる．

模範解答	ユーザプログラム
f	f
f	f
fend	fend
i	i
iend	iend
fend	w <- '}' が不足している
...	...

以上が本論文で示す手法の基本的な方針であるが，この方法には次の問題点が存在する．

まず，この方法では閉じ括弧不足の位置を細かくは検出できない．上の例では本来閉じ括弧が不足しているのは while 文の前ではなく，11 行目の代入文の前である．しかし，図 2 のようにイベント間にイベントと関係のない文があった場合，本来イベントと関係のない文の前に閉じ括弧を置かなければいけない場合でも，イベントの直前（図 2 の場合は w の直前）としか指摘することができない．これでは，使用者は間違った位置に '}' を書いてしまい，コンパイルエラーは無くなるが求めていた結果が得られずに混乱させてしまう恐れがある．

二つ目の問題点として，同じ結果を得る for 文と while 文の比較ができない点があげられる．繰り返し処理である for と while は，どちらで行っても同様の結果を得ることができる．そのため，指定がない限りはプログラムを書く側がどちらを用いるかが分からない．模範解答のプログラムを用意しても，模範解答とユーザのプログラムで使用する繰り返し処理が違う場合がある．上で示した手法では，for 文や while 文の出現をイベントとして，イベント列を生成することはできる．しかし，模範解答とユーザのプログラムで使用している繰り返し処理の方法が異なる場合，それらを異なるイベント列として比較するので誤りであると検出してしまう．

3.3. イベント列

本研究では，上記の問題を解決するため，イベントの種類に以下のものを加える．

- return (return 文)
- s (宣言を含めた，return 文以外のブロック構造を持たない文)

これは，イベント列の中でブロック構造とは無関係の文を識別できるようにするためである．return を特別に区別したのは，ブロック構造と同様制御構造に関わる文であるからであるが，本論文ではこれ以上扱わない．

また，s は構文解析の際，高い頻度で出現するが，その出現回数には意味がなく，むしろ有害となる場合もあるため（例えば複数の変数を単一の文でまとめて宣言するか，別々の文を用いて宣言するかによって，同じプログラムでも異なるイベント列として検出される），s の出現は 1 回以上の繰り返しとし，正規表現の記法を用いて {s}+と表記する．

この拡張により図 1 と図 2 のイベント列を比較した結果は以下のとおりである．

```
1  #include <stdio.h>
2  void main() {
3    int i, j, k;
4    i = 1;
5    while (i <= 10) {
6      j = 0;
7      while (j < 10) {
8        printf("%d", j);
9        j++;
10     }
11     if (i > 10) {
12       printf(",");
13     }
14     i++;
15   }
16   k = 0;
17   while (k < 10) {
18     printf("%d", k);
19     k++;
20   }
21 }
```

図 3. ユーザのプログラム例（while 使用）

模範解答	ユーザプログラム
{s}+	{s}+
f	f
{s}+	{s}+
f	f
{s}+	{s}+
fend	fend
i	i
{s}+	{s}+
iend	iend
fend	{s}+ <- '}' が不足している
…	…

これより，閉じ括弧不足がブロック構造を持たない文の直前であることが分かり，図 2 の 11 行目より前に閉じ括弧が必要なことがわかる．

3.4. for 文と while 文の扱い

本手法では，模範解答とユーザのプログラムに使用する繰り返し文 for と while の違いがある場合でも比較することを可能にするために，イベント列の比較方法を変更し，fend と wend を比較する際に '}' 不足を指摘せず，さらに wend の前にのみ余分な {s}+ がある場合は無視するようにした．これは単純な方法ではあるが，ある程度は機能する．例えば，図 1 のプログラムに対し，そこで用いられている for 文を while 文に書き換えたプログラムを用意し（図 3），両者のイベント列を比較すると以下のようになる．

模範解答	ユーザプログラム
{s}+	{s}+
f	w
{s}+	{s}+
f	w
{s}+	{s}+
fend	wend
i	i
{s}+	{s}+
iend	iend
	{s}+
fend	wend
…	…

4. 評価

4.1. コードベースを用いた評価

本節では，コードベースを用いて，閉じ括弧不足があるプログラムの不足箇所を本提案手法がどれくらい正確に検出できるかについて評価を行った．コードベースは，AtCoder[2]内の公式解説と，大分大学の初年次プログラミング科目である「基礎プログラミング」の例題集より，筆者が例題を適当に 20 個ずつ選ぶことによって収集した．扱った例題については，main 以外の関数を使わず，且つ main 内にネストされたブロックがあるものを中心に収集した．なお，AtCoder は競技プログラミングサイトのひとつである．

評価方法は，まず，ソースコード内の '}' を第一著者が適当に[3]一つ削除し，本手法を実装したツールで解析してそのエラーを確認する．次に，表示された位置が正しい位置であるかどうかを確認する．評価項目は，エラー

[2] https://atcoder.jp/home

[3] コードベースの収集や '}' の削除にバイアスが入らないよう，できる限り何も考えずに行うことを意識して行ったという意味である．

表 1. 検出成功数

コードベース	検出数
AtCoder	8 (20 のうち)
基礎プログラミング	11 (20 のうち)

表 2. 既存のコンパイラとの比較

本研究\Visual Studio	正しい	正しくない
正しい	0	19
正しくない	8	13

```
1  #include <stdio.h>
2  int main() {
3    int s = 75;
4    if (s>=90) {
5      printf("あなたの成績はSです\n");
6    else if ((80<=s) && (s<90)) {//}削除
7      printf("あなたの成績はAです\n");
8    } else if ((70<=s) && (s<80)) {
9      printf("あなたの成績はBです\n");
10   } else {
11     printf("がんばりましょう\n");
12   }
13   return 0;
14 }
```

図 4. エラーを正しく検出できなかった例

を検出でき，且つ検出された箇所に閉じ括弧を挿入することによってプログラムが正しく動作した場合に「正しくエラーを検出できた」とする．

評価結果を表 1 に示す．AtCoder のソースコードに本ツールを用いた場合，40%はエラーを正しく検出できた．「基礎プログラミング」のソースコードに本ツールを用いた場合，55%はエラーを正しく検出できた．

ここで，エラーを正しく検出できなかったケースについて考える．図 4 は，模範解答から 6 行目の '}' を削除したプログラムである．これと模範解答のイベント列を比較すると以下のようになる．

```
模範解答    ユーザプログラム
{s}+        {s}+
i
{s}+
i
...
```

このように，ユーザのプログラムでは閉じ括弧がないために 4 行目の `if` 文に相当する AST ノードが作られる前に構文解析が失敗している．このように，構文解析が早期に失敗するようなケースでは本手法は役に立たないことがわかる．一方でこの事例は既存のコンパイラでは正しくエラーを検出してくれそうである．そこで，本ツールと既存のコンパイラを比較するために，評価で用いた閉じ括弧不足のあるユーザのソースコードを Visual Studio Community 2019[4]でコンパイルし，コンパイルエラーが正しく検出されるかどうかを確認した．評価基準は，閉じ括弧不足を指摘し行数を表示している，もしくは，閉じ括弧不足の次のトークンにエラーがあることを指摘しているなら正しく検出しているとし，閉じ括弧不足を指摘していないものは正しく検出していないものとする．

結果を表 2 に示す．本研究で正しく検出できたケース 19 件のうち，既存のコンパイラで正しく検出できたケースは 0 件であり，正しく検出できなかったものは 19 件であった．また，本研究で正しく検出できなかったケース 21 件のうち，既存のコンパイラで正しく検出できたケースは 8 件であり，正しく検出できなかったものは 13 件であった．これは，既存のコンパイラで正しく検出できなかった閉じ括弧不足の多くを本手法では正しく検出できており，逆に本手法で正しく検出できなかった閉じ括弧不足の一部は既存のコンパイラで正しく検出できたことを示している．つまり，構文木がある程度出来上がった段階で見つかるような構文エラーは既存のコンパイラは正しく検出できず，そのようなケースでは本手法が有効に働くこと，逆に既存のコンパイラでエラーが正しく見つかるようなケースでは本手法は役に立たないことをこの結果は示唆している．

4.2. 被験者実験

イベント列による抽象化の妥当性を確認するため，本研究では 1 名の被験者（情報系を専攻している大学学部 4 年生で，C 言語の経験有り）に協力してもらい，ある設問に対するあらかじめ用意した模範解答と，被験者の

[4] https://visualstudio.microsoft.com/ja/downloads/

[設問]
配列 { 1,3,5,5,7,9,11,13 } の最大値を出力するプログラムを C 言語で作ってください.
(main 文だけでお願いします)

図 5. 被験者実験に用いた設問

```
#include <stdio.h>
//模範解答
int main() {
  int max = 0;  //最大値
  int i = 0;
  int a[8] = { 1,3,5,5,7,9,11,13 };
  //最大値の仮定
  max = a[0];
  for (; i < 8; i++) {
    if (max < a[i]) {
      max = a[i];
    }
  }
  printf("最大値は%dです。\n", max);
  return 0;
}
```

図 6. 被験者実験で準備した模範解答

作成したプログラムに対して，イベント列生成・比較が正しく行えるかどうかについての実験を行った.

実験の手順は，まず被験者に図 5 に示す設問を解いてもらう．設問は，main 以外の関数を使わず，且つ main のブロック中にネストされたブロックができるようなものにした．次に，あらかじめ用意した模範解答と被験者のプログラムに対して，本ツールを使用する．図 6 に模範解答のプログラムを示す．それに対し，被験者は図 7 のようなプログラムを記述した．どちらも同じ実行結果になる．両者を，本ツールを用いてイベント列同士で比較した結果は以下のとおりである.

```
#include <stdio.h>

int main() {
  int a[8]={1, 3, 5, 5, 7, 9, 11, 13};
  int i, max;

  max = 0;
  for(i=0; i<8; i++) {
    if(a[i]>max){
      max = a[i];
    }
  }
  printf("最大値は%dです。\n", max);
  return 0;
}
```

図 7. 被験者の書いたプログラム（印刷の都合で一部改変）

模範解答	ユーザプログラム
{s}+	{s}+
f	f
{s}+	{s}+
i	i
{s}+	{s}+
iend	iend
fend	fend
{s}+	{s}+
return	return

模範解答とユーザのプログラムで，変数宣言の順序や変数 max の初期化方法などの違いはあるものの，イベント列として比較すると全く同じになり，適切な比較が行われていることが確認できる.

4.3. 議論

コードベースを用いた評価の結果から，既存のコンパイラで正しくエラー箇所を検出できないケースでも本ツールでは正しく検出できる場合が多く，逆に本ツールで正しくエラー箇所を検出できない場合でも既存のコンパイラだと比較的正しくできる場合があることが確認できた．とくに，両方のツールで正しく行えた場合は，今

回の実験では一つもなかった．これらのことから両者は相補的な関係にあり，場合に応じて適切に使い分けるのがよいと考えることができる．

なお，被験者実験については現在１名のみのデータしか集まっていない．本ツールは，大学初年次教育などの入門的なプログラミング教育で用いられることを想定しており，イベント列レベルで違いが出るような様々なバリエーションが存在する場面はそう多くないと著者らは考えているが，それでも結果の一般性については大きな疑問が残されている．特に、閉じ括弧 '}' 忘れ以外のエラー (余計な {s}+ブロックを置いてしまうなど) が入りうる環境においての，本手法の評価は課題の一つである．今後より多くの被験者を集めて結果の検証を重ねる必要がある．

5. 結論

本論文では，既存のコンパイラが，ブロック構造の閉じ括弧 '}' を忘れた際に誤りの原因となっている行番号を正しく指摘することができない点を改善するために，模範解答と初学者のプログラムを比較し，その差異によってプログラム中のブロック構造の閉じ括弧 '}' 忘れとその位置を指摘する方法を提案した．具体的な実現方法として，構文解析器が処理するソースコード中の構文要素をイベントとして抽象化し，模範解答と初学者のプログラムのイベント列を比較するツールを実装し，収集したコードベースによる評価と被験者実験による評価を行った．その結果，既存のコンパイラが正しくエラー箇所を検出できない場合において，本ツールが正しくエラー箇所を検出できることが多いことが確認できた．また１名の被験者実験においてはイベント列の抽象化の妥当性を確認できた．なお本ツールは閉じ括弧忘れに特化しているが，他の種類の誤りについても同様の方法で検出できるものがないか，今後検討する必要がある．

参考文献

[1] 内田 公太, 権藤 克彦, C 言語初学者向けツール C-Helper の現状と展望, 第 54 回プログラミング・シンポジウム, pp.153–160, 2013.

[2] 井上 克郎, 神谷 年洋, 楠本 真二, コードクローン検出法, コンピュータソフトウェア, 18(5), pp.47–54, 2001.

[3] J. S. Song, S. H. Hahn, K. Y. Tak, and J. H. Kim, An intelligent tutoring system for introductory C language course, *Computers & Education*, 28(2), pp.93–102, 1997.

[4] Nghi Truong, Paul Roe, and Peter Bancroft, Static analysis of students' Java programs, In *Proceedings of the Sixth Australasian Conference on Computing Education (ACE'04)*, pp.317—325, 2004.

[5] 大須賀 俊憲, 小林 隆志, 渥美 紀寿, 間瀬 順一, 山本 晋一郎, 鈴村 延保, 阿草 清滋, CX-Checker：柔軟にカスタマイズ可能な C 言語プログラムのコーディングチェッカ, 情報処理学会論文誌, 53(2), pp.590–600, 2012.

[6] 西村 渉, 模範解答を用いたコンパイルエラー出力法, 令和 2 年度大分大学理工学部卒業論文, 2021.