プログラム変換を用いた右括弧が欠落したプログラムに対する補正手法に関する研究

2012SE233 杉江駿介　2013SE191 白井将也　2013SE199 杉山雄哉
指導教員：吉田敦

1　はじめに

情報系の学科ではプログラミング実習を行うが，その実習では，個々の学生の進捗状況を把握するのは難しい．そこで，各学生の進捗状況を定期的に把握する手段として WebIDE を用いる方法がある [1][2]．WebIDE を用いれば，学生のプログラムのスナップショットを定期的に取得できる．ただし，スナップショットは編集途中なので，構文規則を満たさない場合がある．一般に，そのようなプログラムに対しては構文木を構成できず，進捗状況の把握のためのプログラム解析を適用できない．

一方，不正確さを許容できるのであれば，構文エラーを含んだプログラムの解析は可能であり [4]，そのような構文解析器の 1 つに TEBA[3] がある．小規模な不正確さであれば，進捗状況の把握への影響は少なく，近似解を求められる．

構文エラーが起こる例として以下のように複数の状況がある．

- 括弧の欠落
- 識別子を途中まで入力した状態
- 被演算子の入力前の状態
- セミコロンの欠落
- 制御文の予約語のみを書いた状態

このような状況でも，括弧は重要な要素である．括弧のような構文要素間の包含関係を表現する字句が抜けると，構造が大きく異なることがあり，進捗状況の近似解すら求められなくなる．そこで，本研究では欠落した括弧を補正する方法について検討する．

欠落した括弧を補うには，括弧内に入る式や文などの構文要素を識別する必要がある．TEBA を用いれば，欠落した括弧の挿入箇所を特定でき，構文木を操作すれば，正しい構文木に変換できる．

括弧の補正には，次の 2 つの方法が必要である．

- 括弧を入れるべき適切な位置を特定する方法
- 括弧を挿入した後の構文木を構文的に正しい構造に変換する方法

また，括弧の欠落に対する補正方法が正しく処理することを確認するには，字句が抜けたプログラムを用意してテストする必要がある．

本研究では次の 2 つの方法を提案する

- 欠落した括弧を追加し、構文木を補正する方法
- その方法に対するテストケースの生成方法

本研究では，括弧は右括弧の欠落のみを扱う．プログラムを書く際は左括弧を右括弧よりも先に書くので，編集途中で右括弧の方が欠落しやすいからである．左括弧についても，同様に実現は可能と想定される．

構文解析およびプログラム変換には TEBA を利用する．以降では，TEBA で解析した構文木と，TEBA が提供する字句書き換え系を用いて，字句補正ルールによる補正を行う．

2　欠落した括弧の補正方法

2.1　括弧の欠落

括弧の存在箇所を構文要素ごとに整理した．結果を表 1 に示す．

表 1　構文要素ごとの括弧の出現箇所

構文要素の種類	構文要素	括弧の種類
制御文	if 文	制御式の丸括弧 (条件演算子を用いた条件式の丸括弧)
	switch 文	制御式の丸括弧
	do 文	制御式の丸括弧
	while 文	制御式の丸括弧
	for 文	前処理、制御式、後始末を囲む丸括弧
複合文		文をまとめる波括弧
式	キャスト式	キャスト演算子の丸括弧
	計算式	計算時に用いる丸括弧
配列		宣言時の配列要素を囲む角括弧
		初期化する際の波括弧
		宣言時以外の添字を囲む角括弧
関数	関数定義	関数名の後の引数を囲む丸括弧
		中身を囲む波括弧
	関数呼び出し	関数名の後の引数を囲む丸括弧
	プロトタイプ宣言	関数名の後の引数を囲む丸括弧
その他	構造体の宣言	メンバを囲む波括弧
	列挙体	列挙定数を囲む波括弧
	共用体	メンバを囲む波括弧

2.2　TEBA による仮想字句の補正

TEBA は，括弧の不足に対し仮想的な括弧を挿入するが，構文的に正しい方法ではない．はじめに，プログラム内を上から下に走査し，不足する左括弧を右括弧の直前に挿入する．次に，逆に走査し，不足する右括弧を左括弧の直後に挿入する．

2.3　括弧が欠落した構文木

括弧の欠落により構造がどのように変わるかを説明する．ソースコード 1 に対し，TEBA が構成する構文木を図 1 に，構文的に正しい構文木を図 2 に示す．

ソースコード 1　欠落が発生しているプログラム

```
int score[5][5[5];
```

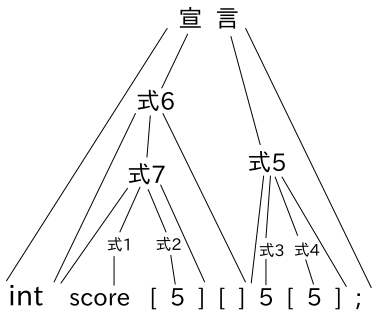

図 1 TEBA が自動補正した構文木

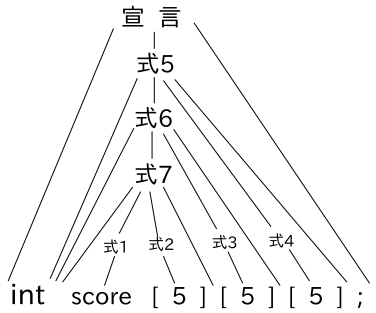

図 2 構文的に正しい構文木

2.4 括弧の位置の特定

TEBA の構文木と構文的に正しい構文木を見比べ，括弧の補正位置を特定し，構文木の変形方法を検討した．その結果，以下の 4 通りの方法で補正できた．

(A) **直後の兄弟と結合** 括弧内に入るべき構文要素が直後に存在する兄弟となる構文要素の子に入っている状態を補正する方法

(B) **親の兄弟の子と結合** 括弧に入るべき式が直後に存在する兄弟となる構文要素の子孫になっている状態を補正する方法

(C-1) **親の兄弟の構文要素の移動 (構文の制約なし)** 括弧内に入るべき構文要素が親の子になる，かつ括弧を含む構文的に閉じた式と兄弟になっている状態を補正する方法

(C-2) **親の兄弟の構文要素の移動 (構文の制約あり)** 括弧を含む構文的に閉じた構文要素の兄弟になっている括弧内に入るべき構文要素を構文的に許されるまで括弧で囲む方法

2.5 テストケースの生成方法

括弧の欠落が，その欠落を含む構文要素の兄弟や親子の構造にも影響を与える可能性があるので，各構文要素ごとに兄弟関係や親子関係に基づいてテストケースを作成する．すべての括弧の欠落に対して，対象とする構文要素の兄弟関係，親子関係に基づき検討する．さらに，正しい構造を誤って書き換える可能性を考慮して，テストケースを作成した．

1. 前に文，宣言や式がある
2. 後に文，宣言や式がある
3. 前と後に文，宣言や式がある
4. 他の文，宣言や式の中にある
5. 括弧が連続して登場する

上記の 3 の具体例をソースコード 2 に示す．補正したい宣言の前と後に宣言があることで，括弧の不足する宣言以外の箇所に挿入しないことや，括弧が不足していない宣言を書き換えないことを検証できる．

ソースコード 2 3 の具体例

```
int score[5][5];
int score1[5[5];
int score2[5][5];
```

このようなテストケースを構文要素が持つそれぞれの括弧に対してに用意し，補正方法を検証できるようにした．

3 括弧補正ツールの設計

3.1 括弧補正ツールの流れ

括弧が欠落したプログラムを，TEBA で構文解析する．得られた構文木にプログラム変換器を用いて補正ルール群を適用し，補正後の構文木をプログラムのテキストに戻す．

3.2 親の兄弟の子と結合 に対する補正方法

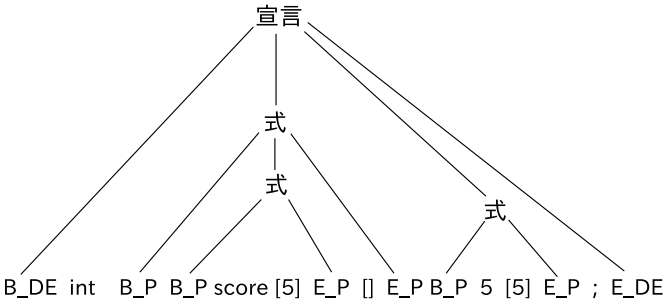

図 3 配列の字句列を含めた構文木

図 3 は TEBA が右角括弧が抜け落ちている配列を構文解析したときにできる構文木である．

ソースコード 3 はソースコード 1 のような配列要素が 3 つの配列で，2 つ目の要素の右角括弧が抜け落ちているプログラムに対する補正ルールである．TEBA で構文解析をすると，2 つ目の要素の左角括弧の直後に仮想的な右角括弧を挿入するので，A_L A_R となる．左角括弧を A_L，右角括弧を A_R と表す．式の位置が 2 章の図 2 のように score の直前から 3 つなければいけないが，図 3 のように B_P が score の直前に 2 つしかなく，囲まれていない 5 の直前に B_P がくるので本来の構文木とは違う形になる．仮想的な A_L の直後の B_P，E_P を正しい位置に移動させ，仮想的な A_L を直後の要素である 5 の直後に移動させている．

表 2 ルールの種類

構文要素	欠落字句	補正ルールの種類	補正方法の分類	
if 文	条件式右丸括弧	2 種類	(A)	ルールを定義できた
switch 文	制御式右丸括弧	1 種類	(A)	ルールを定義できた
while 文	制御式右丸括弧	1 種類	(A)	ルールを定義できた
do 文	制御式右丸括弧	1 種類	(A)	ルールを定義できた
for 文	制御式などを囲む右丸括弧	3 種類	(A)	一部の状況で定義できた
複合文	文を囲む右波括弧	1 種類	(C-2)	一部の状況定義できた
キャスト式	キャスト演算子の右丸括弧	1 種類	(C-1)	一部の状況で定義できた
算術式	算術式に使われる右丸括弧	0 種類		ルールを定義できない
関数定義	引数を囲む右丸括弧	1 種類	(C-1)	ルールを定義できた
関数定義	本体を囲む右波括弧	1 種類	(C-2)	一部の状況で定義できた
関数呼び出し	引数を囲む右丸括弧	1 種類	(C-1)	ルールを定義できた
プロトタイプ宣言	引数を囲む右丸括弧	1 種類	(C-1)	ルールを定義できた
配列	添字演算子の右角括弧	2 種類	(A)	一部の状況で定義できた
配列	宣言時の右角括弧	7 種類	(B)	一部の状況で定義できた
配列	初期化する際の右波括弧	3 種類	(C-1)	一部の状況で定義できた
構造体	メンバを囲む右波括弧	1 種類	(C-2)	ルールを定義できた
列挙体	列挙定数を囲む右波括弧	1 種類	(C-2)	ルールを定義できた
共用体	メンバを囲む右波括弧	1 種類	(C-2)	ルールを定義できた

ソースコード 3 3 次元配列の補正ルール

```
{
$bde:B_DE $id:ID_TP $sp:SP $bp#b:B_P
$bp2#c:B_P $var:ANY $al#paren:A_L
$lin1:ANY $ar#paren:A_R $ep2#c:E_P
$al1#paren1:A_L $ar1#paren1:A_R
$ep#b:E_P $bp3#d:B_P $lin2:ANY
$al2#paren2:A_L $lin3:ANY $ar2#paren2:A_R
$ep3#d:E_P $sc:SC $ede:E_DE
}
=>
{
$bde $id $sp $bp#b:B_P $bp2#c:B_P
$bp3#d:B_P $var $al $lin1 $ar
$ep3#d:E_P '['#paren1:A_L
$lin2 ']'#paren1:A_R $ep2#c:E_P $al2 $lin3
$ar2 $ep#b:E_P $sc $ede
}
```

4 ツールの実装と評価

4.1 実装

表 2 は，表 1 で示した構文要素に対し，何種類の補正ルールが構文の制約に基づいて定義できたかを示すものである．

以下の 5 点は，一部の状況でのみ定義できたものである．

- for 文は，括弧で囲む 3 つの式の初期化式，条件式，後始末のうち，後始末の式を書かない状況以外の場合でルールを定義できた
- 配列は三次元までしか対応させない三次元までであれば補正ができる
- 関数定義の本体を囲む右波括弧は，構文規則上関数の中に関数は存在しないので，関数の直前とプログラムの終わりまでという状況であれば補正できる
- キャスト式がソースコード 4 のような場合補正ができた
- 複合文は文の終わりが return 文のときのみ補正できる

ソースコード 4 キャスト式

```
b=(double a;
```

算術式については，正しい構文木を構成できる括弧の挿入位置が複数存在する場合があるので，算術式の補正ルールが定義できていない．

関数定義や配列のように括弧が複数存在するものは，三次元まで考えることとする．

4.2 評価

対象とする構文要素に対して本研究の補正ルールを適用することで，本来意図していた箇所に括弧が追加され，構文木の生成ができているかを検証する．

補正ルールの検証を行うにあたって，補正の対象とする箇所の構文木の構造を変えてしまう状況をテストケースとして分類した．それぞれのテストケースに補正ルールを適用し，その成功率をもとに本研究の補正ルールが妥当であるかどうかを評価する．評価に用いる対象は，表 1 で挙げている構文要素ごとの右括弧とする．

表 3 欠落字句とテスト成功回数

構文要素	欠落字句	前に文・式	後に文・式	前後に文・式	文・式の中に	括弧の中
if 文	条件式右丸括弧	3/3	3/3	3/3		1/1
switch 文	制御式右丸括弧	3/3	3/3	3/3		3/3
while 文 (do)	制御式右丸括弧	3/3	3/3	3/3		3/3
for 文	制御式などを囲む右丸括弧	3/3	3/3	3/3		3/3
複合文	文を囲む右波括弧	1/3				1/3
キャスト式	キャスト演算子の右丸括弧	3/3	3/3	3/3	0/3	3/3
算術式	算術式に使われる右丸括弧					
関数定義	引数右丸括弧	3/3	3/3	3/3		3/3
関数定義	本体右波括弧	3/3	2/3	1/3		3/3
関数呼び出し	引数右丸括弧	3/3	3/3	3/3		3/3
プロトタイプ宣言	引数右丸括弧	3/3	3/3	3/3		3/3
配列	宣言時右角括弧	3/3	3/3	3/3		3/3
配列	初期化右波括弧	3/3	3/3	3/3		3/3
配列	添字演算子右角括弧	3/3	3/3	3/3		2/2
構造体	メンバを囲む右波括弧	3/3	3/3	3/3		3/3
列挙体	列挙定数右波括弧	3/3	3/3	3/3		3/3
共用体	メンバを囲む右波括弧	3/3	3/3	3/3		3/3

表 3 は，構文要素ごとのテストケースに対し，本研究で作成した補正ルールを適用したときの成功率をまとめたものである．対象とする構文要素のまわり配置される字句によって，構文木の構造が異なることがあるようなテストケースでも，本研究で作成した補正ルールで，構文的に正しい箇所に括弧を追加できているか検証した結果を表している．

表 3 の縦軸は，構文要素ごとの右括弧を列挙している．第 3 列から第 7 列までの 1 行目は構文要素に対してのテストケースの種類である．第 3 列から第 7 列の 2 行目以降は，各構文要素のテストケース対し，補正ルールを適用したときの成功率を示す．

表 1 で挙げた括弧のうち，補正ルールが実現できたものは 4/5 程度であり，それぞれに起こる可能性のある状態のテストケースはすべて網羅している．

4.3 考察

式の中の括弧を補正する場合のように，正しい構文木を構成できる括弧の挿入位置が複数存在する場合がある．そのようなプログラムに対して，括弧を追加する箇所を決めることは困難である．右括弧と左括弧の対応関係をとるこ

とはできるが，プログラマが意図する位置を特定して，挿入することができない．

ソースコード 5 式の中の括弧

```
((a+b)*c*d)
```

ソースコード 6 式の中の括弧

```
((a+b*c)*d)
```

例として，ソースコード 5，ソースコード 6 を挙げる．右丸括弧が式 c を囲んでいるか，いないかのように，括弧の補正位置が変わることで，異なる構文木ができる．複合文の括弧を補正する場合，TEBA を用いると，文の区切り箇所を見つけることが困難なので，return 文が存在しない場合には括弧を補正する箇所が決められない．しかし，if 文の中に return 文が出現することがあり，その if 文を含む複合文が存在した場合，return 文で区切ると構文木が，本来意図していた構文木と異なる．

5 関連研究

コンパイラは，構文エラーが存在する場合はコードを生成できない．ただし，1 回のコンパイルで，できるだけ多くのエラーを発見できるよう，構文エラーが生じた構文要素を無視して，次の字句から解析を継続する．また，JDT[6] など，エディタに組み込まれる構文解析器は、構文エラーに対する回復処理として，エラーを含むブロックを特定し，それを無視することで，コード全体を解析を達成している．

C 言語では前処理を利用することが一般的であり，2 種類の言語が混在する．プログラマは前処理前のプログラムを編集するので，開発支援ツールは，前処理前のプログラムに対する構文木を必要とするが，コンパイラなどが持つ前処理後のプログラムに対する構文解析器ではエラーが生じる．前処理の典型的な書き方などを分析し，前処理を伴う構文エラーを回避する解析器としては，srcML[7]，yacfe[8]，TEBA[3][9][10] などが存在する．ただし，これらの手法であっても，本来，プログラマが意図した構文木を生成するとは限らず，近似的な構文木を生成する．本論文の手法は，これらの手法を拡張する形で，欠落する字句を補正し，より正確な構文木を生成する方法を提案している．なお，本論文と同じような手法には知る範囲においては提案されていない．

6 おわりに

本研究では，括弧の欠落は構文木に影響を与えることから，プログラミング学習において，進捗状況を把握し，指導をするうえで重要であるので，プログラム変換を用いた右括弧が欠落したプログラムに対する補正手法を提案した．また，構文解析器である TEBA を用いて，右括弧の欠落が生じているプログラムを構文解析し，その結果を元に正しい構文木を生成できるように括弧を追加するプログラムを作成した．さらに，括弧の欠落により，進捗状況を正しく把握できない場合のために欠落している右括弧を自動的に補正できるようにした．今後の課題は，インデントの読み取りや，括弧以外の欠落がある複合的なプログラム，式の中の括弧に対する補正ルールの検討，実装を行うことである．

参考文献

[1] 浅井裕太, 石川航, 松井暁生: 『プログラミング演習における進捗状況把握方法の提案』, 南山大学情報理工学部 2013 年度卒業論文, 2013.

[2] 浅野勝, 小林悟: 『WebIDE を用いたプログラミング初学者向けのテスト評価支援方法の提案』, 南山大学情報理工学部 2015 年度卒業論文, 2015.

[3] 吉田敦, 蜂巣吉成, 沢田篤史, 張漢明, 野呂昌満: 『属性付き字句系列に基づくソースコード書き換え支援環境』, 情報処理学会論文誌, Vol. 53, No. 7, pp. 1832–1849, 2012.

[4] L. Moonen: “Generating Robust Parsers using Island Grammars,” Proc. 8th Working Conf. Reverse Engineering, IEEE Computer Society Press, pp.13-22, 2001.

[5] G. J. Holzmann: “Brace Yourself,” in IEEE Software, Vol. 33, No. 5, pp. 34–37, 2016.

[6] The Eclipse Foundation: JDT Core Component, http://www.eclipse.org/jdt/core/index.php.

[7] M. L. Collard, and J. I. Maletic: Document-Oriented Source Code Transformatin using XML, Proc. 1st International Workshop on Software Evolution and Transformation, Vol.75, No.12, pp. 11–14, 2004.

[8] Y. Padioleau: Parsing C/C++ Code without Preprocessing, Proc. 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, CC ’09, Berlin, Heidelberg, pp. 109–125, 2009.

[9] 前林達也, 吉田敦, 蜂巣吉成, 張漢明, 野呂昌満: 『前処理前プログラムに対する記号表の構成手法』, 情報処理学会論文誌, Vol. 54, No. 2, pp. 912–921, 2013.

[10] 吉田敦, 蜂巣吉成: 『前処理指令に対する制約のない前処理前コードの構文解析手法』, 情報処理学会論文誌, Vol. 56, No. 2, pp. 593–610, 2015.