Vol.2020-SE-204 No.6
2020/3/2

パーサコンビネータにおける網羅的なエラー報告候補集合の生成とそれに基づくエラー報告

伊東 忠彦[1,a)]　大久保 弘崇[2,b)]　粕谷 英人[2,c)]　山本 晋一郎[2,d)]

概要：プログラムの構文誤りに対して言語処理系が提示するエラー報告は正確さを欠いていたり情報が不十分であることがある．誤りを修正するための情報としてより有用なものを返す，あるいは修正方法を誤認させるようなふさわしくないエラー報告を抑制することが望ましい．しかし，構文誤りを含むプログラムから作成者が望んだ正しいプログラムを推測することは，本質的に難しい問題である．本稿では，パーサの実装方法の 1 つであるパーサコンビネータに着目する．誤りのある入力に対する振る舞いを再検討し，部品パーサの返すエラー報告を集積することでエラー報告の候補集合を作成し，その中から誤りの修正作業に最も有用な選択肢を提示する手法を提案する．作成者が正しく書こうとしたが誤りが含まれてしまっているプログラムの真意の近似として，競技プログラミングの誤答プログラムとそれを修正した正答プログラムを用いて，提案手法の有効性を検証した．その結果，サンプルとして採用した誤答プログラムのうち，82%において有用な修正案の提示に成功した．

1. はじめに

1.1 背景

プログラムは，言語処理系により解釈実行あるいは実行形式にコンパイルされるが，プログラミングの過程では様々な誤りが発生する．それらの誤りは，ソースプログラムが構文に従わない場合に発生する構文エラー，関数定義における型宣言と呼び出しの実引数が矛盾するような意味エラー，配列の存在しない要素にアクセスするなどで実行が異常終了する実行時エラー，実行結果がプログラマの意図と異なるバグ，などを引き起こす．

本稿ではそれらの中でも，構文エラーに注目する．構文エラーは，言語処理系の要素である構文解析器が返すエラーである．パーサが提示するエラー報告は一般に，ソースプログラム中の受理できなかった文字またはトークンの位置と，その位置で構文から期待される正しい入力候補からなる．このエラー報告を，プログラマはプログラムの誤りを修正するヒントとして利用する．しかし実際には，エラー位置が本当に修正するべきプログラムの誤り箇所から大きく外れていたり，提示されている入力候補が有用でないような場合も多い．

構文解析器は，プログラムの誤りを修正するヒントとしてより有用性の高いエラー報告を返すべき，あるいは逆に修正の妨げになるようなふさわしくないエラー報告を抑制することが望ましい．しかし，構文誤りのあるひとつのプログラムから，プログラマが本当に望んだ正しいプログラムは一般には推測できないため，これは本質的に難しい問題である．

我々は [1] において，パーサコンビネータライブラリ Parsec におけるエラーメッセージの置き換えに関する研究を行った．C 言語のサブセットのパーサを作成し，`unexpected` パーサを用いた言語指向のエラー報告をさせる動作を追加した．置き換え前と置き換え後のメッセージを比較し，どのような改善を図ることができているかの考察を行った．

また，C 言語初学者を対象としたプログラミング学習を支援する既存研究として，静的解析ツール C-Helper の提案 [2] がある．初学者が既存ツールを利用する際の問題点に着目しその改善策を提示することにより，プログラミン

[1] 愛知県立大学大学院情報科学研究科
Graduate School of Information Science and Technology, Aichi Prefectural University
[2] 愛知県立大学情報科学部
School of Information Science and Technology, Aichi Prefectural University
[a)] tadahiko@yamamoto.ist.aichi-pu.ac.jp
[b)] ohkubo@ist.aichi-pu.ac.jp
[c)] kasuya@ist.aichi-pu.ac.jp
[d)] yamamoto@ist.aichi-pu.ac.jp

グ初学者にとってなるべく分かりやすいソースコードの問題点の指摘や解決策の提案を目的としている．

加えて，エラー報告改善の関連研究として，PEG 文法におけるラベル付き失敗 [3] がある．この手法では解析時に失敗してはならない構文となる部分に注釈付けを行い，エラー報告時にその情報を役立てるというものである．これにより，的確なエラー報告を出力することが可能になる場合も考えられる．

その他にも，この問題に対する他のアプローチは複数考えられる．

機械学習を用いて，誤りのあるプログラムとそれに対する正しいプログラムを対として学習させ，提示するアプローチもありうる．これはそのような学習を実現するための大量の教師データと，構文誤りを含むプログラムを ML の入出力に移す双方向の写像が必要となる．

構文誤りのあるプログラムに対して編集距離として近傍にあり構文誤りのないプログラムを探索あるいは生成する方法も考えられる．このアプローチも追加のコストを要する．

本稿では，言語処理系の開発プロセスを変更せずに適用できるアプローチを目指す．ここで，構文解析器はパーサコンビネータライブラリを利用して実装することを想定する．ライブラリの持つエラー報告機構を改良することにより，よりユーザの助けとなる修正案を提示する手法を提案する．

2. パーサコンビネータ

パーサコンビネータとは，パーサの実装方法の 1 つである．基本パーサとそれを加工し組み合わせるコンビネータにより複雑な構文を受理するパーサを作ることができる．

2.1 パーサの構築

基本パーサは，数字，アルファベット，指定された文字列，与えられた述語を満たす任意の文字，などを受理するものが提供されている．基本パーサは文脈自由文法の終端記号に相当する．

例えば `string "ABC"` は `"ABC"` を受理し，`"X123"` も `"ABz"` も受理しない．受理できない入力に対して，パーサはエラー報告を結果として返す．

パーサはモナドアクションとして定義され，値として名前に束縛できる．この名前が文脈自由文法の非終端記号に相当する．コンビネータは文脈自由文法の個々の機能に対応している．連接はモナドの順次実行が対応し，選択は優先度付きの選択をするコンビネータが提供される．複数のパーサを順に試し，最初に成功したものを結果とする．また，0 回以上の繰り返しを受理するなどのより高度なコンビネータもある．

例えば，

```
p1 >> p2
```

は `p1` が入力を受理し，続きを `p2` が受理したとき成功する．

また，

```
q1 <|> q2
```

は，`q1` が入力を受理したときその結果を返す．`q1` を受理しなかった場合に，`q2` が入力を受理したときはその結果を返す．

2.2 Megaparsec

Megaparsec[4] は，関数型言語 Haskell で記述されたパーサコンビネータライブラリである．部品となる単純な構文のパーサを Haskell の関数として作成し，それらを組み合わせてパーサを構築する．本稿ではバージョン 6.5.0 を対象としている．

Megaparsec におけるパーサの構築方法は 2.1 節のとおりであるため，この節では，主に Megaparsec におけるエラー報告に関する機能について述べる．

2.2.1 `ParseError` 型

Megaparsec には，パースの失敗内容そのものを表す `ParseError` データ型がある．これらを図 1 に示す．エラーメッセージを出力するための関数などの引数として用いられる．

`ParseError` 型の中にある `NonEmpty` 型は base パッケージで定義されており，常に少なくとも 1 つの要素を持つことを保証するリストを示している．`SourcePos` 型にはソースファイル名，エラー時の行・列番号が格納されている．

`ParseError` 型の値コンストラクタの内容は次のとおりである．

TrivialError

Megaparsec の機構によって自動的に生成されるエラーである．

入力におけるエラーとなった位置，存在する場合には受理できなかったトークン，および受理したかったトークンがデータとして含まれている．`ErrorItem` 型は何らかのトークン，エラーに対して名前付けされたラベル，入力終了 (EndOfInput) のいずれかをとるデータ型である．

FancyError

カスタムエラーメッセージ機能を利用した際に生成されるエラーである．

カスタムエラーメッセージとは，Megaparsec の標準機能として用意されているものであり，部品パーサに注釈付けするような形で任意の場所でエラー報告を行うことができるようになる．本稿ではこの機能を直接扱わないので，詳細は省略する．

```
data ParseError t e
   = TrivialError (NonEmpty SourcePos) (Maybe (ErrorItem t)) (Set (ErrorItem t))
   | FancyError (NonEmpty SourcePos) (Set (ErrorFancy e))
```

図 1　ParseError 型定義

```
pPlus :: (Ord e, Stream s)
  => ParsecT e s m a
  -> ParsecT e s m a
  -> ParsecT e s m a
pPlus m n = ParsecT f where
  f s cok cerr eok eerr = unParser m s cok
      cerr eok meerr where
    meerr err ms = unParser n s cok ncerr
        neok neerr where
      neok x s' hs = eok x s' (toHints (
          statePos s') err <> hs)
      ncerr err' s' = cerr (err' <> err) (
          longestMatch ms s')
      neerr err' s' = eerr (err' <> err) (
          longestMatch ms s')
```

図 2　pPlus 関数の定義

2.2.2　エラーメッセージの決定方法

Megaparsec の選択コンビネータ<|>を用いて構築されたパーサが解析失敗となった場合，出力されるエラーは 1 つに絞られる．主な影響があるのは，選択コンビネータ<|>の動作として定義されている pPlus の中に含まれている longestMatch 関数である．その内容を図 2 に示す．

pPlus 関数は，2 つの ParsecT e s m a 型のパーサを組み合わせて新たなパーサを返す．ncerr と neerr はそれぞれ，トークンを消費してエラーとなった場合と，トークンを消費せずエラーとなった場合に返す関数である．この中に含まれている longestMatch 関数は，あるトークンを引数にとった 2 つのパーサで解析した後の状態 ms と s' を引数にとり，より長くマッチした方を返している．

この仕組みにより，パーサが解析失敗となった場合は，パーサが結合している順番に関わらず「その中で最もトークンがマッチした」後のエラーが出力される．

3.　構文エラーに関する調査・考察

本稿の目的を達成するために，前手順として改善すべきエラー報告パターンの抽出を行う．それに伴い，ユーザが求めるエラー報告情報とパーサが提示する情報に相違がある場合について考察する．また，Megaparsec において破棄されているエラー報告の事例についても述べる．

3.1　構文エラーとエラー報告の整合性調査

3.1.1　実例となる構文エラーの必要性

本稿では，構文誤りを修正するためのエラー報告としてより有用なものを出力する，あるいは修正方法を誤認させるようなふさわしくないエラー報告を抑制することが研究目的の 1 つである．後の有効性の検証の過程において，あらかじめ作成しておいた正しいプログラムにランダムで構文エラーを紛れ込ませたサンプルを利用するのは適切とは言えない．プログラマが正しく書こうとしたものの誤ってしまった構文エラーを含むプログラムを利用することが重要であるため，それを取得するための調査を行った．

3.1.2　調査内容

競技プログラミングコンテストを開催している代表的な Web サイトとして，AtCoder[5] がある．問題量が豊富であることや，常設のコンテストがあることなどから，さまざまな言語のプログラムの投稿が盛んに行われている．

本稿では，AtCoder 上で出題されている問題に対して提出されている解答プログラムから，ユーザが起こした間違いとエラー報告の整合性について，次のような流れで調査を行った．

(1) 同じ問題に対する 1 人のユーザの正答と誤答の解答プログラムの組み合わせを取得する
(2) それぞれの解答プログラムの差分からユーザの修正内容を割り出す
(3) 修正内容とエラー報告内容が一致しているか確認する

上記の 3 番目の項目にある「修正内容とエラー報告の内容が一致」している場合に整合性があるとし，そうでない場合は整合性がないと定義する．対象とした問題は一番易しいレベルである Beginner Contest カテゴリの A・B 問題，対象言語は C 言語 (GCC5.4.1)，誤答のエラー種類はコンパイルエラーとした．

取得した構文エラーが含まれるプログラムの一例として，図 3 を示す．また，このプログラムに対して GCC が出力している構文エラーに関する報告を図 4 に示す．これは，for 文中の “;”が “,”になっているという構文エラーである．

AtCoder における調査においてはワードとして「expected」を含むエラー報告の回数の中でも上位 4 種類であった以下のメッセージに絞って調査した．(1)，(2)，(3) の末尾の「...」の部分には任意のトークンが入る．

(1) セミコロンの追加を提案 (error: expected ‘;’ before ...)
(2) 閉じかっこの追加を提案 (error: expected ‘)’ before ...)
(3) 複数の修正候補を提案 (error: expected ‘=’, ‘,’, ‘;’, ‘asm’ or ‘__attribute__’ before ...)

```
1  #include<stdio.h>
2
3  int main(void){
4      char s[6];
5      scanf("%s",s);
6      int i,counter = 0;
7      for(i=0,i<4;i++){
8          if(s[i] == '+'){
9              counter ++;
10         }
11         else{
12             counter --;
13         }
14     }
15     printf("%d\n",counter);
16     return 0;
17 }
```

図 3 取得した誤答プログラムの一例

```
./Main.c: In function ‘main‘:
./Main.c:7:20: error: expected ‘;‘ before ‘)‘
    token
     for(i=0,i<4;i++){
                    ^
```

図 4 取得した誤答プログラムに対する構文エラーに関する報告

表 1 提出件数の内訳

エラー種類	提出総件数	誤答と正答の組が取得できた数
1	1232 件	139 件
2	376 件	79 件
3	487 件	98 件
4	482 件	100 件

(4) 関数定義が閉じられていない場合の提案 (error: expected declaration or statement at end of input)

3.1.3 調査結果

このエラー報告が出現していた A 問題の提出，B 問題の提出，それらのうち正答と誤答の組み合わせを取得できた数を表 1 にまとめた．

次に，各エラー報告において，正答と誤答の比較から，修正内容を調査するとともに，エラー報告と修正内容が一致する割合を確認する．

セミコロンの追加を提案

セミコロンがないことを指摘されている「error: expected ‘;’ before ...」メッセージが出力されているプログラムは，表 2 のように修正されていた．

整合性があるものとして，GCC の指摘どおりにセミコロンを追加して修正しているものは 58%であり，ユーザが求めていたメッセージを出せていることがわかる．その他整合性がないものとしては 3 つに分けることができた．

- **誤字を修正**
  “;”を “:”にしてしまう，C 言語の予約語である return の綴りを間違えている，二項演算の “*”を全角文字の “× ”にしてしまう，余分な “)”を付けてしまうという事例であった．
- **文法間違いを修正**
  else に条件式を付けてしまっているのを修正している，for 文の中を修正している，という事例であった．
- **かっこを閉じて修正**
  “)”やタブルクオーテーションを閉じる” がない部分を修正している事例であった．
  同時に出ているエラー報告では閉じかっこがないことを指摘できていたが，閉じられていない関係で次に “} ”が来るタイミングでこのメッセージが出されてしまっていた．

表 2 セミコロンの追加を提案した場合の修正内容

ユーザの修正内容	件数	割合
セミコロンをつけて修正	81	58%
誤字を修正	34	25%
文法間違いを修正	8	6%
かっこを閉じて修正	16	12%
合計	139	

表 3 閉じかっこの追加を提案した場合の修正内容

ユーザの修正内容	件数	割合
かっこを閉じて修正	18	23%
条件式の中を修正	18	23%
カンマを追加して修正	25	32%
タブルクオーテーションを修正	4	5%
その他	14	18%
合計	79	

閉じかっこの追加を提案

閉じかっこを追加することを提案している「expected ‘)’ before ...」メッセージが出力されているプログラムは，表 3 のように修正されていた．

メッセージ通り閉じかっこを追加して修正している場合は 23%にとどまっていた．その他の修正内容として，以下のものがあった．

- **条件式の中を修正**
  条件式において “==”が入るはずの部分を “=”にしてしまっている，“/”が入るはずの部分をバックスラッシュにしてしまっている，などの問題を修正をしていた．
- **カンマを追加して修正**
  関数の引数として区切りが必要な場合に printf("%d\n"temp) などと記述してしまった場合の修正である．
  なお，カンマ (“,”) を追加するような提案はエラー報告に含まれていなかった．
- **ダブルクオーテーションを修正，その他**
  printf の中身を”” で囲っていない，printf の外に改行文字を入れている，elseif 文を囲む時の波かっこを “(”

表 4　複数の修正候補を提案した場合の修正内容

ユーザの修正内容	件数	割合
変数宣言を修正	28	29%
include 行を修正	48	49%
int main() を修正	11	11%
その他	11	11%
合計	98	

表 5　関数定義が閉じられていない場合の修正内容

ユーザの修正内容	件数	割合
main 関数の波括弧を閉じて修正	63	63%
for 文・if 文を閉じて修正	25	25%
タブルクオーテーションを閉じて修正	4	4%
その他	8	8%
合計	100	

にしてしまっている，などの問題を修正していた．

複数の修正候補を提案

複数の修正候補を提案している「`error: expected '=', ',', ';', 'asm' or '__attribute__' before ...`」メッセージが出力されているプログラムは，表 4 のように修正されていた．

これまでの結果とは異なり，`int x = 1;` などの変数宣言に関する誤りを指摘できていたのは 29%に留まり，`#include` ディレクティブを忘れた影響でこのメッセージが出力されてしまっているパターンが一番多かった．その他の修正内容は，以下のとおりである．

- **int main() を修正**
  関数宣言である `int main()` のかっこを付け忘れてしまっているという問題を修正していた．
- **その他**
  C++の機能である `using namespace` を使ってしまっている，変数宣言時と `for` 文の初期化式の “=”を “==” にしてしまっている，コメントが正しく閉じられていない，というパターンがあった．

関数定義が閉じられていない場合の提案

「`error: expected declaration or statement at end of input`」というエラー報告は自体は，宣言もしくは文を入力の最後に期待していたという内容を表す．しかし，実際多くの場合は関数や `if` 文の条件式などの閉じ波括弧が不足しているときにみられるエラー報告である．このメッセージが出力されているプログラムは，表 5 のように修正されていた．

プログラムの最後に “}”を追加して修正しているのは 63%であった．

- **for 文・if 文を閉じて修正**
  “}”を入力の最後ではなく途中に追加して修正している例である．`for` 文・`if` が閉じていない場合にやむを得ずこのメッセージが出力されているパターンである．
- **タブルクオーテーションを閉じて修正，その他**
  ダブルクオーテーションが閉じていないことによる対応関係の修正，関数定義の閉じる波括弧を “]”にしている，`char s1[100;` のように変数宣言で失敗している，`else if` を使用したのにもかかわらず条件式を書き忘れてしている，という間違いの修正が含まれていた．

```
1  exStmtParser = do { expr <- exExprParser
2                    ; semi
3                    ; return StmtExpr expr
4                    }
5                <|> do { (...) }
6                <|> do { (...) }
7
8  exExprParser = try funcCall <|> try assign
       <|> opTerm
9
10 assign = do
11   id <- identifier
12   symbol "="
13   expr <- exExprParser
14   return $ Assign id expr
15
16 funcCall = do
17   id <- identifier
18   symbol "("
19   expr <- commaSep exExprParser
20   symbol ")"
21   return $ FuncCall id expr
22
23 opTerm = makeExprParser term opTable
```

図 5　文パーサと式パーサ (一部省略)

3.2　破棄されているエラー報告

Megaparsec の性質上，破棄されてしまっているエラー報告がある事例について述べる．

例として図 5 のパーサを示す．この文パーサ `exStmtParser` は，構成の一部として式パーサ `exExprParser` を持ち，どちらも複数のパーサが選択コンビネータ<|>で結合されている．また，式パーサは関数呼出を受理する `funcCall`，代入を受理する `assign`，単項演算・二項演算または項そのものを受理する `opTerm` から構成されている．`term` と `opTable` の定義については省略する．

式パーサにおいて関数呼出，代入で失敗した場合は，`try` コンビネータによる入力の状態の巻き戻しを行い，結合している次のパーサで解析を試みる．このようなパーサがある場合に，間違った入力 `sum = xfold(10 sum);` を文パーサで解析すると，次のような結果が得られる．

```
1:12:
  |
1 | sum = xfold(10 sum);
  |             ^
unexpected '('
expecting ';', alphanumeric character, or
```

```
1:16:
  |
1 | sum = xfold(10 sum);
  |                 ^
unexpected 's'
expecting ',', ), or operator
```

図 6　破棄されているエラー報告

```
    operator
```

Megaparsec では，なるべく解析に成功した結果を優先して，それを最終結果とする仕組みがある．この入力は，代入の `sum = xfold(10 sum)` の右辺として，関数呼出パーサにて `xfold(10 sum)` を解析した際 `xfold(10` と `sum)` の間に，が無いため失敗する．このとき重要なのが，内部では `sum = xfold(10` まで解析すること自体には成功していることである．解析失敗までの流れをまとめると，以下のようになる．

(1) 文パーサ内部 (図 5 の 1 行目〜4 行目) の `exExprParser` で `sum = xfold(10 sum);` の解析を行う
(2) `exExprParser` における関数呼出パーサ `funcCall` が `sum = xfold(10 sum);` の `xfold(10` と `sum)` の間に“,”が無いことで失敗する
(3) `sum = xfold` で，代入パーサ `assign` が成功することで式パーサ `exExprParser` が成功する
(4) 文の終端として“;”を解析しようとするが，それと“(”が一致せずエラー報告が返される

パーサとしては `sum = xfold` で成功し，その後の失敗結果を出力してるに過ぎないが，ユーザにとっては，図 6 のような式パーサで解析を行っている途中で得られた `sum = xfold(10` までマッチしていた情報も知りたい場合がある．

こうした，ユーザが求める情報とパーサが提供する情報に整合性がない状況が発生してしまうことがわかった．

4. 提案手法

Megaparsec で作成したパーサが破棄してしまう複数のエラー報告を活用することにより，よりユーザの助けとなる修正案を提示するための提案手法について述べる．提案手法の流れとしては，

(1) エラー報告の蓄積
(2) 修正案に利用するエラー報告の決定
(3) 修正案の提示

という 3 段階に分けられる．

4.1 エラー報告の蓄積

構文解析中のエラー報告を蓄積するための手法について述べる．本稿では，Haskell の State モナド [6] を利用した，エラー報告が蓄積可能なコンビネータの実装を行った．基本的には純粋であり，値の変更等が行えない関数型言語

```
type ParserS e s a = ParsecT e s (S.State [(
    State s,(ParseError (Token s) e))]) a
type Parser = ParserS Void Text
```

図 7　Parser 型定義

```
putErr :: ParserS e s a -> ParserS e s a
putErr m = ParsecT $ \s cok cerr eok eerr ->
  let mcerr err s' = S.modify ((s',err):) >>
      cerr err s'
      meerr err s' = S.modify ((s',err):) >>
          eerr err s'
  in unParser m s cok mcerr eok meerr

alt' :: (Stream s,Ord e) => ParserS e s a ->
    ParserS e s a -> ParserS e s a
alt' p q = putErr p <|> putErr q
```

図 8　実装した関数

Haskell において，State モナドは可変な状態付きの計算を扱うことが可能になる機構である．

State モナドで扱う状態としては，Megaparsec における「エラー時の解析状態」を持つ `State s` 型の値，「発生したエラー報告」そのものである `ParseError (Token s) e` 型の値の組のリストである．この組を構文エラーが発生したときに追加して更新していくことで，エラー報告の蓄積を実現する．

4.1.1 型定義の修正

本稿で実装したコンビネータを利用するためには，パーサの型定義を修正する必要がある．一般的に，Megaparsec を用いてパーサを作成する場合，`Parser` 型の内部に定義されている型シノニム

```
type Parsec e s a = ParsecT e s Identity
    a
```

を利用し，

```
type Parser = Parsec Void Text
```

のように定義する．これは，内部モナドを示す `m` が `Identity` の場合がほとんどだからである．このような `Parser` 型があった場合は，図 7 のように修正を行う．

`Identity` モナドの部分が `State` モナドに変更され，前述の値の組を状態として持つことができるようになった (Megaparsec 内にパース状態を扱う `State` 型が既に定義されているため，修飾名をつける形で Control.Monad.State モジュールをインポートする必要がある)．

4.1.2 エラー報告蓄積のためのコンビネータ

エラー報告蓄積のための機能として，`putErr` 関数と，それを利用するためのコンビネータ `alt'` を作成した．これを図 8 に示す．

`putErr` 関数は，パーサを 1 つ引数にとり，Control.Monad.State モジュールで定義されている `modify` 関

数を利用して，状態のリストにそのとき発生したエラーを挿入して更新し，そのままパーサ自身を返す関数である．

これにより，エラー報告の蓄積は可能になったが，これだけではパーサのあらゆる箇所にこの関数を挿入しなければならない．それでは利便性に欠けてしまうため，putErr 関数を組み込んだ alt' コンビネータを作成した．このコンビネータを従来の選択コンビネータ<|>と置き換えることで，エラー報告の蓄積が可能となる．

4.2 修正案に利用するエラー報告の決定

構文エラーが発生し，それ以上の解析ができなくなった場合，修正案に利用するエラー報告を決定する．本手法では，「入力の最も先まで解析できた位置を抽出する」ことで決定を行う．蓄積されたエラー報告のうち，位置が同値であるが修正候補が異なる場合はそれぞれをマージし，その中から場合によっては複数の修正候補を提案する．

例として，あるプログラムの 33 行目で構文エラーが発生した場合に，図 9 のような 3 つのエラー報告が得られたとする．この場合は，左から 2 番目と 3 番目のエラー報告が修正案に利用される．

4.3 修正案の提示

最終段階として，4.2 節で選択したエラー報告の情報を利用し，元ソースコードの再解析を試みて，修正案を提示する．修正候補 (「expecting」の後ろに列挙されているトークン) となる n 個のトークンそれぞれを挿入，置換し，再び構文解析を行う．また，これとは別に，エラーとなったトークンそのものを削除することでも再解析を行う．再解析に成功したら，そのパターンを対象プログラムの修正案として提示する．

いくつかのパターンで再解析が成功した場合は，元のソースコードと修正後と比べて差分が小さいものを修正案として採用する．

図 9 の左から 2 番目と 3 番目のようなエラー報告があった場合，挿入の修正を行った例として，

```
printf(foo (bar);
printf(foo =bar);
printf(foo )bar);
printf(foo ,bar);
```

置換の修正を行った例として，

```
printf(foo (ar);
printf(foo =ar);
printf(foo )ar);
printf(foo ,ar);
```

削除の修正を行った例として，

```
printf(foo ar);
```

表 6 実験結果の内訳

総件数	エラーサンプル	成功数	失敗数	成功の割合
93	68	56	12	82%

が考えられる．

この例では，挿入では printf(foo ,ar); と printf(foo ,bar); が，置換では printf(foo =bar); と printf(foo =ar); が再解析に成功する．この場合ではこれらのうち printf(foo ,bar); もしくは printf(foo =bar); を採用し，“,”もしくは “=”を追加する修正案を提示する．

5. 実験と評価

提案手法の有効性を確かめるための実験と，その結果を踏まえた評価・考察を行う．本稿では，ある程度の大きさを持つ構文で実験を行うために，要点となる機能を残して簡潔な言語体系とした C 言語のサブセットパーサを作成した．このサブセット言語パーサでは，使用する型や制御構造に制限があり，式・文・関数定義・トップレベル構文定義のみが受理できる．

5.1 実験

本手法で選択したエラー報告と提示する修正案がふさわしいかどうかを評価するための実験を行った．サンプルとして使用するプログラムは，3.1.2 節にて収集した正答と誤答の組み合わせが存在する一般ユーザが記述した解答プログラムである．ユーザの修正とエラー報告に整合性がある組に関してはそもそもエラー報告を修正する必要がないと判断し，実験の対象外とした．

また，サンプルプログラムはフルセットの C 言語で記述されているため，実装した C 言語のサブセットパーサでも構文解析が行えるような形に手動でプログラムの変換を行った．主な項目は次の通りである．

- #include 文を削除
- int main(void) の void を削除
- printf や scanf の引数 (書式文字列) を識別子に変更・ダブルクオーテーションを削除
- ポインタ型変数をポインタ型でない変数に変更
- 一行の変数宣言を複数行に分割する
- インクリメンタル演算子を代入式に変換
- char 型を int 型に，配列を削除
- 文字列の代入は識別子の代入に変換

これらのサンプルプログラムについて，誤答プログラムに対して提示した修正案が組で取得した正答プログラムの内容と一致している場合に実験成功，一致していない場合に実験失敗とした．

5.2 評価と考察

実験結果は表 6 の通りである．

```
33:9:
33 | printf(foo bar);
           ^

unexpected '('
expecting ';', alpha
numeric character, or operator
```

エラー報告の位置:
33 行 9 列目
修正候補:3 個

```
33:14:
33 | printf(foo bar);
                ^

unexpected 'b'
expecting '(' or '='
```

エラー報告の位置:
33 行 14 列目
修正候補:2 個

```
33:14:
33 | printf(foo bar);
                ^

unexpected 'b'
expecting ')', ',', or operator
```

エラー報告の位置:
33 行 14 列目
修正候補:3 個

図 9 蓄積したエラー報告の比較

実験の対象とした誤答と正答の組み合わせ提出は 93 件ある．

この 93 件は，3.1.3 の調査で得られた以下の提出の組を使用している．

- 表 2「セミコロンの追加を提案した場合の修正内容」のうち，「誤字を修正」の 34 件，「かっこをとじて修正」の 16 件
- 表 3「閉じかっこの追加を提案した場合の修正内容」のうち，「条件式の中を修正」の 18 件，「カンマを追加して修正」の 25 件

これら以外の組み合わせについては，誤答プログラムを C 言語のサブセット言語に変換する過程で構文エラーが修正されてしまい，ほとんどがサンプルプログラムとして利用できないものになってしまうため，実験の対象外とした．

93 件の中でエラーサンプル (C サブセットプログラムに変換できたもの) としては 68 件得られた．これらはすべて GCC で整合性がないエラー報告が出力されていたものであり，そのうち 56 件が再解析に成功し，正答と同じ修正案の提示を行うことができた．

一方で，失敗したサンプルは 12 件であり，内容としては大きく 2 つにわけられた．1 つは全角の “× ”やバックスラッシュ等，パーサ側で対応していないトークンによる構文エラーである．こういったエラーはトークンの削除を行っても修正が難しい場合が多いということが見受けられた．

もう一つは，`if(foo == 0 bar == 0)` のような入力の `foo == 0` と `bar == 0` の間でエラーが出てしまう場合である．現在は 1 つの文字に着目して修正案の提案を行っている．しかし，このようなエラーに対応するには何らかの手段をもってトークンの前後関係を考慮する必要があるため，現在の仕組みでは修正が困難である．

6. おわりに

本稿では，パーサコンビネータにおける構文解析の途中で生成される複数のエラー報告から，よりユーザの助けとなる修正案を提示する手法を提案した．

競技プログラミングコンテストサイトを対象とした，プログラムの修正内容とエラー報告の整合性を調査し，改善すべきエラー報告パターンの抽出を行った．

パーサコンビネータライブラリの 1 つである Megaparsec で作成したパーサには破棄されているエラー報告があることに着目した．本稿にて実装したコンビネータでエラー報告の蓄積をしたのち，蓄積されたエラー報告を利用したプログラムの再解析の結果から修正案を提示する手法を提案した．また，その効果を測るための実験・評価を行った．調査結果に基づく整合性のないエラー報告に対して 82%を整合性のある修正案として提示することができた．

今後の課題としては，既存研究との比較，C 言語サブセットパーサよりも広い構文パターンで実験を重ねていくことや，Megaparsec 以外のパーサ作成支援系にも本手法が適用できるかどうかを検証する，等が挙げられる．

謝辞 本稿は JSPS 科研費 15K00488 の助成を受けたものである．

参考文献

[1] 伊東忠彦, 大久保弘崇, 粕谷英人, 山本晋一郎. コンビネータパーサによる構文解析器における言語指向のエラー報告. 情報処理学会研究報告, Vol. 2018-SE-198, No. 20, pp. 1–8, 2018.
[2] 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. Vol. 2013, pp. 153–160, 2013.
[3] Andre Murbach Maidl, et al. Error reporting in parsing expression grammars. In *Science of Computer Programming 132*, pp. 129–140, 2016.
[4] megaparsec: Monadic parser combinators. `https://hackage.haskell.org/package/megaparsec-6.5.0`. 2020 年 2 月 5 日閲覧.
[5] Atcoder. `https://atcoder.jp/`. 2020 年 2 月 5 日閲覧.
[6] Control.monad.state. `http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State.html`. 2020 年 2 月 5 日閲覧.