Vol.2018-SE-198 No.20
2018/3/9

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

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

概要：パーサ作成支援系が提供するエラー報告機能は，言語処理系の利用者に提示するものとしては十分でない場合がある．利用者は構文誤りを修正する必要があり，エラー報告はその助けとなる情報を持つべきである．そのような視点でのエラー報告を本論文では「言語指向」と呼ぶ．本論文では，パーサコンビネータライブラリである Parsec に焦点をあて，ライブラリが提供するエラーコンビネータを利用した場合の，エラー報告を言語指向にするための改善の方法を体系化することを目標に検証する．事例研究として，Parsec を用いて C 言語のサブセットのパーサを作成し，エラーメッセージの置き換え機能を提供する標準の unexpected コンビネータを用いて，言語指向のエラー報告をさせる動作を追加した．

1. はじめに

プログラミング言語で記述されたプログラムを計算機上で実行するためには，言語処理系を用いて実行形式に変換を行う．言語処理系はレクサ（字句解析器），パーサ（構文解析器），意味解析器，コード生成器または解釈実行系など複数の要素から成り立っている．機構の 1 つであるパーサは，レクサで字句解析されたトークンを受け取り，構文木を生成する．パーサを作成する場合，構文規則と受理アクションからプログラムを自動生成するパーサジェネレータを利用することが一般的である．

近年発展しているパーサジェネレータとして，Parsec[1] などのパーサコンビネータライブラリがある．Yacc[2] のように文法の記述からパーサのソースプログラムを生成するのではなく，Parsec は小さなパーサを Haskell の関数として作成し，それらをコンビネータで組み合わせて複雑な構文を受理するパーサを実行時に構築する．

パーサは，構文誤りのために受理できない入力が与えられた時に構文エラーを報告する．そのエラー報告は解析を続けることが不可能となった位置，受理できなかったトークン，期待した入力などからなる．このエラー報告の内容そのものは，パーサにバグがない限り正しいものである．しかし，言語処理系の利用者に提示するものとしては適切でない場合がある．

パーサジェネレータは与えられた構文規則の粒度で動作し，エラー報告を行う．一方利用者は対象言語の構造の粒度で思考する．この粒度の違いが適切でないエラー報告につながる．利用者は構文誤りを修正する必要があり，エラー報告はその助けとなる情報を持つべきである．そのような視点でのエラー報告を本論文では「言語指向」と呼ぶ．

言語指向のエラー報告を行うパーサを作成するためには，パーサの作成者が適切な情報を構文に注釈付けしたり，構文エラー時の異常系処理を丁寧に実装したりする必要がある．また，パーサ作成支援系もその作業を支援するべきである．

本論文では，パーサコンビネータライブラリの Parsec に焦点をあてる．ライブラリが提供するエラーコンビネータを用いて，エラー報告の改善の方法を体系化することを目標とし検証を行う．

2. Parsec

Parsec は，Haskell で記述されたパーサコンビネータライブラリである [1], [3]．派生版として，速さに重点を置いた attoparsec[4]，全体的な改善を目指した Megaparsec[5] などが存在する．

2.1 言語パーサの作成

言語を解析するパーサを実装する際の流れについて述べる．Parsec では小さなパーサを組み合わせて，最終的に複雑な構文を受理するより大きなパーサを構築する．主に次の 3 つの段階に分けて説明する．

[1] 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

(1) 構文定義に対応する内部表現を抽象データ型として定義する
(2) makeTokenParser 関数を用いて，そのパーサの仕様を反映させたトークンパーサを作成する
(3) 解析しようとする言語の LR 文法定義をなぞる形でパーサを構築する

一般的な「式」に対するパーサを構築するには，LR 文法を作成したり，演算子の優先度を考慮したりする必要がある．Parsec には，その役割を担う buildExpressionParser というユーティリティが提供されている．makeTokenParser 関数，buildExpressionParser 関数についての詳しい説明はそれぞれ 2.1.2 節，2.1.3 節で行う．また，パーサのエラー処理において重要なバックトラック，本論文で主に扱っているエラーコンビネータについても述べる．

2.1.1 内部表現の定義

内部表現の定義は，代数的データ型を用いて行う．作成したい言語の構文定義と 1 対 1 に対応するように構文木のデータ型を定義する．パーサがトークンを受理した際，対応したコンストラクタを返すことにより構文木を作成する．

2.1.2 トークンパーサの作成

Parsec には字句解析を担う関数 makeTokenParser がある．ソースコード内に任意に存在する空白などを読み飛ばすために，この関数で生成したトークンパーサを用いて構文解析を行う．この関数の型を以下に示す．

```
makeTokenParser :: Stream s m Char =>
  GenLanguageDef s u m ->
  GenTokenParser s u m
```

最初に，コメントや識別子などのルール，予約語などの仕様をレコードのフィールドとして設定する．これは，Parsec 内では GenLanguageDef 型として定義されている．識別子で使用できる文字や，大小文字の区別など，そのパーサにおける細かいルールを決定する．

定義したレコードを makeTokenParser に渡すと，GenTokenParser という型のレコードが返される．レコード GenTokenParser のフィールドはいずれも小さなパーサである．例として，自然数を解析する natural や，識別子を解析する identifier，セミコロンを解析する semi などがある．それらを用いることにより，トークンレベルで構文解析を行うことが可能になる．

2.1.3 パーサの作成

トークンパーサを最小単位の部品として，それらを組み合わせて式・文・トップレベル定義を構文解析するパーサを作成する．

式パーサは，Parsec に用意されている buildExpressionParser 関数を用いることにより簡単に作成することができる．この関数の型を以下に示す．

```
1  exprparser = (-省略-) --式パーサ
2
3  stmtparser = do { expr <- exprparser
4                  ; semi
5                  ; return StmtExpr expr
6                  }
7              <|> do { (-省略-) }
8              <|> do { (-省略-) } --文パーサ
9
10 toplevelparser = (-省略-) --トップレベル定義パーサ
11
12 parser = whiteSpace >> many1 toplevelparser
13                        <* eof --パーサ本体
```

図 1 パーサ作成例

```
buildExpressionParser :: Stream s m t =>
  OperatorTable s u m a ->
  ParsecT s u m a ->
  ParsecT s u m a
```

ParsecT は，ストリーム型 s，ユーザ状態型 u，基底モナド m，戻り型 a を持つパーサである．演算子の優先順序，結合性，それらの演算子をパースした場合のアクションを定義した Operator 型のリストのリストである OperatorTable 型，項をパースする方法を定義した ParsecT 型のパーサ，これら 2 つを buildExpressionParser に渡すと，式パーサを得ることができる．代入や関数呼び出しなど，単項演算と二項演算以外に定義が必要な場合は，それらのパーサを適宜作成し選択コンビネータ <|> で結合する．

式パーサを作成したら，**図 1** のように文・トップレベル定義パーサを構文規則に則って記述していく．パーサ一つ一つを，受理したのちに対応したコンストラクタを return で返す do 構文とし，それらを選択コンビネータ <|> で結合することで，最終的に構文木を返すパーサを作成する．先頭の空白を読み飛ばさなければならないことに注意が必要である．

パーサが完成したら，parseTest 関数を使って正常に解析が行われるかをテストすることができる．入力として与えた文字列で失敗せずに最後まで解析が行われた場合は，構文木が返される．途中で失敗した場合は，失敗した行と列，受理できなかったトークンと期待していたトークンを示すエラー報告が返される．

2.2 バックトラック

パースに失敗したとき，構文解析の状態を戻してやり直すことをバックトラックという．Parsec においては，try コンビネータを用いてこのバックトラックを行うことができる．

例えば，数字を 1 つ受理する digit，英文字を 1 つ受理する letter を用いたパーサ (digit >> letter) <|> (digit >> digit) があるとする．>>の左辺で解析が成功した場合はその結果を破棄し，右辺の実行結果を返す．

左辺で失敗した場合はそのままエラー報告が行われる．

このパーサに入力として “12” を与えた場合，まず左側にある (digit >> letter) において数字 “1” が消費される．その次に “2” をパースしようとするが，これは letter で失敗する．そのままの状態で (digit >> digit) に移ると入力 “1” が消費され，“2” だけが残ったままパースしようとしてしまい，正しくパースすることができない．

この場合に try (digit >> letter) <|> (digit >> digit) と修正すると，失敗して (digit >> digit) に移る際に，入力が (digit >> letter) によって消費される前と同じである “12” に戻すことができる．よって，このパーサは “1a” のような数字と文字が 1 つずつある文字列か，“12” のような数字が 2 つ並んでいる文字列を受理できるようになる．

この例のように，結合したパーサ同士で共通する部分とそうでない部分があった場合，try コンビネータでバックトラックを行うことによってそれぞれのパーサを正常に動作させることができる．

2.3 エラー処理

Parsec では構文解析エラーを表す ParseError というデータ型が定義されており，この中にはエラーのソース位置 (行番号，列番号，ファイル名) を表すデータ型 SourcePos と，エラーメッセージのリストを表すデータ型 Message という 2 種類の情報が格納されている．パーサで解析に失敗した場合には，ParseError の値を元にしてエラーメッセージが出力されるようになっている．

2.3.1 エラーメッセージの種類

Parsec では，エラーメッセージの置き換えなどの処理を追加するために<?>，unexpected，fail という 3 種類のコンビネータが用意されている．<?>は，期待していたトークンを表す expecting メッセージをパーサ実装者が設定したものに置き換えることができ，Parsec でパーサを実装する際，エラーメッセージをわかりやすいものにする為によく用いられる．unexpected コンビネータは，受理できなかったトークンを表す unexpected メッセージを置き換えることができるコンビネータである．fail は，Parsec が自動で生成するエラー報告に，新しいエラーメッセージを追加することができるコンビネータである．

Parsec のエラーメッセージは，先述の抽象データ型 Message のコンストラクタとして以下の 4 種類に分けられている．

SysUnExpect

satisfy コンビネータによって自動的に生成される．引数は予期しない入力である．

UnExpect

unexpected コンビネータによって生成される．引数は予期しない項目を表す．

パーサ unexpected msg は，受理できなかったトークンを表す unexpected メッセージを msg に置き換えるとともに常に失敗し，構文解析をその時点で終了する．

Expect

<?>コンビネータによって生成される．引数は期待される項目を示す．

パーサ p <?> msg はパーサ p が失敗した際，期待していたトークンを表す expecting メッセージを msg に置き換えることができる．

Message

fail コンビネータによって生成される．引数は一般的なパーサメッセージである．

パーサ fail msg は，常に失敗し構文解析をその時点で終了する．これを用いると，自動的に生成されて出力される unexpected，expecting エラーメッセージの後に，続けて msg が出力される．

コンストラクタごとの生成するコンビネータおよび，コンビネータが用いられていたパーサが失敗した時点でプログラムが終了する場合のエラー報告内容をまとめたものを表 1 に示す．

2.3.2 エラー報告の例

例として，digit で英文字 “ab” を解析しようとすると，次のエラー報告が表示される．

```
> parseTest digit "ab"
parse error at (line 1, column 1):
unexpected "a"
expecting digit
```

2 行目にエラーが起きた位置，3 行目にパーサが受理できなかったトークン，4 行目にパーサが期待していたトークンが出力されている．3 行目の unexpected エラーメッセージは，入力されたトークンから satisfy コンビネータが生成している．

digit パーサは，Parsec において次のように定義されている．

```
digit :: (Stream s m Char) =>
    ParsecT s u m Char
digit = satisfy isDigit <?> "digit"
```

satisfy は，digit などの主要なパーサの実装に使われているコンビネータである．パーサ satisfy f は，与えられた関数 f が True の値を返す任意の文字をパースし，解析された文字を返す．digit の実装においては，isDigit は数字 1 文字に True を返すため，数字 1 文字をパースすることができる．

このエラー報告の例では，digit パーサで呼び出されている satisfy コンビネータで失敗しているため，3

表 1 Message 型のコンストラクタ

名前	生成するコンビネータ	エラー報告内容
SysUnExpect	satisfy	unexpected （受理できなかったトークン） expecting （期待していたトークン）
UnExpect	unexpected	unexpected （置き換えたメッセージ） expecting （期待していたトークン）
Expect	<?>	unexpected （受理できなかったトークン） expecting （置き換えたメッセージ）
Message	fail	unexpected （受理できなかったトークン） expecting （期待していたトークン） （追加したメッセージ）

行目の unexpected メッセージは 2.3.1 節で示したうちの SysUnExpect に該当する．また，<?>コンビネータによって expecting メッセージの置き換えがされているため，4 行目のメッセージは 2.3.1 節で示したうちの Expect に該当する．

3. 実験

本論文では，Parsec を用いて C 言語のサブセットのパーサを作ったのち，言語指向のエラー報告が行えるようエラー処理を加えていった．本章ではその具体的な方法の提示，改善前・改善後のメッセージを比較を行い，次章にて結果の考察をする．

3.1 エラーコンビネータを用いた言語指向のエラー報告

2 章で述べたように，Parsec には<?>，unexpected，fail という 3 つのエラーコンビネータが用意されている．その中でも，本論文では unexpected コンビネータに着目した．unexpected コンビネータは，受理できなかったトークンを報告する unexpected メッセージを別のものに置き換えることができる．これを用いて，あらかじめユーザの間違いを想定し，実際にその失敗をしたユーザに対して独自に設定したエラーメッセージを表示することで，言語指向のエラー報告を行うことを提案する．

以降で述べる，検証したエラー処理は以下の 2 つである．

p <|> unexpected "msg"
: 成功するべきパーサ p と unexpected コンビネータを選択コンビネータ<|>で結び，p が失敗したらエラーメッセージを msg に置き換えて失敗させる．

p <|> (q >> unexpected "msg")
: 成功するべきパーサ p と，成功するべきでないパーサ q を選択コンビネータ<|>で結び，q が成功したら，結果を破棄してエラーメッセージを msg に置き換えて失敗させる．

必要に応じて，期待していたトークンを示す expecting メッセージを置き換える<?>コンビネータも使用している．設定したメッセージの内容は，主に確認を促すものや構文のルールを再確認するものである．

成功するべきパーサ p とは，構文規則からその時点で受理させたいパーサのことを指す．成功するべきでないパーサ q とは，ユーザの間違いとして想定されるトークンを受理するパーサである．unexpected コンビネータと成功するべきでないパーサ q を>>で結合することで，q で成功した場合に結果を破棄して unexpected コンビネータによるエラー処理を行うことができる．

このエラー処理における「失敗」とは，構文解析をその時点で失敗させ，メッセージを置き換えたエラー報告をしたのちに終了するという意味である．例えば，p1 <|> p2 というパーサがあったとする．通常，p1 で解析が失敗した場合はそのまま p2 で構文解析を続けて行おうとする．ただし，p1 の中に unexpected コンビネータが用いたエラー処理があり，それが動作した場合は p2 で構文解析を行うことはなく，エラーメッセージが置き換えられたエラー報告が出力されてプログラムが終了する．

3.2 サンプルプログラム

今回，構文解析を行うサンプルとして，C 言語のサブセットのプログラム sample.c を用意した（**図 2**）．

3.3 検証したエラー処理

3.3.1 セミコロンのつけ忘れ

C 言語のサブセットやそのオリジナルである C 言語において，セミコロン (;) は文の終わりを表す．セミコロンが文末で付け忘れられていると，パーサは文が継続していると解釈し，ユーザが別々の文として記述している次の文を継続した文として解析することでエラーになってしまう．この問題に対処するために，ユーザにセミコロンが足りないことを伝える率直なメッセージを出せるよう，式文の解析を行うパーサ部分に**図 3** の 2 行目のようなエラー処理を加えた．

sample.c において，11 行目の式文のセミコロンがない場合のエラーメッセージを比較する．

このエラーは，11 行目の文にセミコロンがないことによって，パーサが 11 行目と 12 行目の文が繋がっていると認識することから発生する．

```
1  int sum;
2
3  int main() {
4      int k;
5      k = 1;
6      sum = 0;
7
8      while (k < 10) {
9
10         if (k % 2) {
11             sum = sum + k;
12         }
13
14         k = k + 1;
15
16     }
17
18     sum = twice(sum);
19
20     return sum;
21 }
22
23 int twice(int x) {
24     x = x * 2;
25     return x;
26 }
```

図 2 サンプルプログラム sample.c

```
1  do { e <- exprparser
2     ; semi <|> unexpected "letter other than ;. 前の式に；は付いていますか."
3     ; next <- stmtparser
4     ; return (StmtExpr e next)
5     }
```

図 3 セミコロンのつけ忘れに対するエラー処理（式文パーサ）

```
Left "sample.c" (line 12, column 9):
unexpected "}"
expecting operator, "+" or ";"
```

図 4 式文パーサにエラー処理を加える前のメッセージ

```
Left "sample.c" (line 12, column 9):
unexpected letter other than ;. 前の式に；は付いていますか.
expecting operator, "+" or ";"
```

図 5 式文パーサにエラー処理を加えた後のメッセージ

エラー処理を加える前の**図 4** のメッセージでは}を予期せぬトークンだとするメッセージが表示される．これは，エラーの原因とは直接関係ないものである．expecting メッセージから補足するべきトークンをある程度予測することはできるものの，ソースコードを確認するまでは特定することは難しい．

これに対して，エラー処理を加えた後の**図 5** のメッセージではセミコロン以外の予期せぬトークンが見つかったことを報告したのち，確認を促すメッセージを出力している．

```
1  do { reserved "while"
2     ; symbol "("
3     ; expr <- exprparser
4     ; symbol ")" <|> unexpected "letter other than ). ) のとじ忘れはありませんか. "
5     ; stmt <- stmtparser
6     ; next <- stmtparser
7     ; return (While expr stmt next)
8     }
```

図 6 () のとじ忘れに対するエラー処理（while 文パーサ）

```
Left "sample.c" (line 8, column 18):
unexpected "{"
expecting operator, "<" or ")"
```

図 7 while 文パーサにエラー処理を加える前のメッセージ

```
Left "sample.c" (line 8, column 18):
unexpected letter other than ). ) のとじ忘れはありませんか.
expecting operator, "<" or ")"
```

図 8 while 文パーサにエラー処理を加えた後のメッセージ

3.3.2 () のとじ忘れ

if 式や while 式などでは，条件式を丸括弧で囲う．こうした括弧を閉じ忘れてしまうと，セミコロンのつけ忘れのように文が継続していると解釈してしまいエラーの原因となる．この対策を行うために，while 文パーサ中の閉じ括弧を解析する部分に**図 6** の 4 行目のようなエラー処理を加えた．

sample.c において，8 行目の while 文の条件式を囲う括弧が閉じられていない場合のエラーメッセージを比較する．

このエラーは，条件式の括弧が閉じられていないまま{という文を囲うための括弧を受理しようとしていることから発生する．よって，エラー処理を加える前の**図 7** のメッセージでは{を予期せぬトークンだとするメッセージが返ってきている．

これに対して，エラー処理を加えた後の**図 8** のメッセージでは，セミコロンのつけ忘れの場合と同様に，) 以外の予期せぬトークが見つかったことを報告したのち，確認を促すメッセージを出力している．

3.3.3 C 言語のサブセットの制約に関するエラー

検証のために実装した C 言語のサブセットには，次のような局所変数に関する制約がある．

- 局所変数の宣言は関数の先頭においてのみ許される．
- 2 つ以上の変数を同時に宣言することはできない．
- 変数宣言において初期化を行うことはできない．

```
1 stmtparser :: Parser Stmt
2 stmtparser = symbol "{" *> stmtparser <*
      symbol "}"
3                 (-一部省略-)
4                 <|> (reserved "int" >>
                      unexpected "int. 局所変数
                      は関数の先頭でのみ宣言でき
                      ます. " <?> "")
5                 <|> return (StmtNil)
```

図 9 局所変数の宣言場所に対するエラー処理（文パーサ）

```
1 do { var <- vardecl
2    ; semi <|> (symbol "," >> unexpected ",␣2
         つ以上の変数を同時に宣言することはでき
         ません. ")
3           <|> (symbol "=" >> unexpected "=.␣
                宣言時に代入はできません. ")
4           <|> unexpected "letter␣other␣than␣
                ;.␣変数宣言のおわりに;はついて
                いますか. "
                <?> ";"
5    ; return (var)
6    }
```

図 10 局所変数の宣言方法に対するエラー処理（局所変数宣言パーサ）

こうしたルールに反していたときの指摘を行うために，文パーサに**図 9** のソースコードのようなエラー処理を加えた．一部省略した部分には，式パーサ，if 文などを受理するパーサがある．

実装した C 言語のサブセットのパーサでは，予約語 int を用いた変数宣言は他のパーサで受理しており，文パーサで int を受理することがない．したがって，文を解析している途中に int が検出されたら，ユーザが文と文の間で変数宣言しようとしていることを予測し，宣言は関数の先頭でのみ行えることを伝える．

また，局所変数を解析する部分に**図 10** のようなエラー処理を加えた．

図 10 の局所変数宣言パーサでは，2 行目で，を検出した時点でユーザが変数を同時宣言しようとしていることを読み取り，3 行目で=を検出することにより宣言時に代入をしようとしていることを読み取り，最後に 4 行目でそれら以外のものが検出されたら; がついていないことを読み取り，状況に応じたエラー報告を返す．

なお，この場合のような「ある特定のパーサが成功したら失敗させる」という手法をとっている場合，"int"以外の間違いがあった場合に予期されるトークンとして"int"が出力されてしまう．これを防ぐために，<?>を用いて expecting メッセージの置き換えを行っている．

C 言語サブセットの制約に対して加えたエラー処理は，以上の 3 種類である．それぞれの場合についてメッセージを比較する．局所変数宣言の終わりにセミコロンがついていなかった場合のエラー処理については，前述のものと同じであるため，ここでは省略する．

```
Left "sample.c" (line 17, column 8):
unexpected reserved word "int"
expecting expression
```

図 11 文パーサにエラー処理を加える前のメッセージ（宣言場所について）

```
Left "sample.c" (line 17, column 9):
unexpected int. 局所変数は関数の先頭でのみ宣言できます.
```

図 12 文パーサにエラー処理を加えた後のメッセージ（宣言場所について）

```
Left "sample.c" (line 4, column 10):
unexpected ","
expecting letter or digit or ";"
```

図 13 局所変数宣言パーサにエラー処理を加える前のメッセージ（同時宣言した場合ついて）

```
Left "sample.c" (line 4, column 11):
unexpected ,. 2 つ以上の変数を同時に宣言することはできません.
```

図 14 局所変数宣言パーサにエラー処理を加えた後のメッセージ（同時宣言した場合ついて）

局所変数の宣言場所

図 2 のサンプルプログラム sample.c において，空行になっている 17 行目に変数宣言を行う int a; を挿入した場合のエラーメッセージを比較する．

エラー処理を加える前の**図 11** のメッセージでは，予期せぬトークンとして予約語 int を，予期したトークンとして式を挙げている．C 言語のサブセットでは局所変数宣言が自由にできると考えているユーザがいたことを仮定してこのメッセージを見ると，あまり親切でないメッセージだと考えられる．

対して，エラー処理を加えた後の**図 12** のメッセージでは，C 言語のサブセットでは局所変数は関数の先頭でのみ宣言できる，という制約を明示的にユーザに伝えることが可能になっている．

2 つ以上の変数を同時宣言

サンプルプログラム sample.c において，4 行目の変数宣言の式を int k,l; に変更した場合のエラーメッセージを比較する．

エラー処理を加える前のメッセージでは，予期せぬトークンとして，を挙げ，予期したトークンとして識別子の続きとしてまだ英数字をとるか，宣言の終わりを示すセミコロンを挙げている．

```
Left "sample.c" (line 4, column 11):
unexpected "="
expecting ";"
```

図 15　局所変数宣言パーサにエラー処理を加える前のメッセージ（宣言時に初期化をした場合について）

```
Left "sample.c" (line 4, column 13):
unexpected =.  宣言時に代入はできません.
```

図 16　局所変数宣言パーサにエラー処理を加えた後のメッセージ（宣言時に初期化をした場合について）

```
1 exprparser :: Parser Expr
2 exprparser = try funcall <|> try assign <|>
      buildExpressionParser table term <?> "
      expression"
```

図 17　式パーサ部分

対して，エラー処理を加えた後のメッセージでは，,というトークンからユーザが同時宣言をしようとしていることを予測し，それはできないということを伝えることが可能になっている．

3.3.4　変数宣言における初期化

サンプルプログラムにおいて，4 行目の変数宣言の式を `int k = 0;` に変更した場合のエラーメッセージを比較する．

エラー処理を加える前の**図 15** のメッセージでは，予期せぬトークンとして=を挙げ，予期したトークンとして; を挙げているシンプルなものである．予期したトークンは一種類であるため，メッセージに従って修正すれば問題ないソースコードになるが，ここではエラーの理由が述べられていない．

対して，エラー処理を加えた後の**図 16** のメッセージでは，=というトークンからユーザが宣言時に代入しようとしていることを予測し，それが不可能であることを伝えることが可能になっている．

3.4　try コンビネータによるバックトラックの影響

C 言語のサブセットのパーサでは `try` コンビネータを用いて適宜バックトラックを行っているが，このバックトラックの仕組みにより，`unexpected` コンビネータを挿入しても意図通りの動作がなされないということがあった．

例として，**図 17** に示す式パーサ `exprparser` においては次のように関数呼び出し (`funcall` 関数) と代入 (`assign` 関数) を `try` コンビネータで囲っている．

また，`funcall` 関数と `assign` 関数の内部はそれぞれ，**図 18**，**図 19** のようになっている．

`assign` 関数と `funcall` 関数は，両方とも最初に

```
1 assign :: Parser Expr
2 assign = do { id <- identifier
3             ; reservedOp "="
4             ; expr <- exprparser
5             ; return (Assign id expr)
6             }
```

図 18　代入式を解析するパーサ部分

```
1 funcall :: Parser Expr -- 関数呼び出し
2 funcall = do { id <- identifier
3              ; symbol "("
4              ; expr <- commaSep exprparser
5              ; symbol ")"
6              ; return (Funcall id expr)
7              }
```

図 19　関数呼び出しを解析するパーサ部分

`identifier` パーサで識別子を解析するという共通の動作を持ち，`identifier` で失敗した場合はバックトラックをしてからでないと正しく構文解析が行えない．そういった理由から `try` が必要となるのだが，これら 2 つの関数内に何らかの間違いを想定して `unexpected` コンビネータを挿入して失敗させても，その時点で `try` コンビネータによってバックトラックがなされてしまい，設定したエラーメッセージを表示することができない．

3.5　unexpected を用いたエラー処理のまとめ

ここまで述べてきたエラーコンビネータ `unexpected` を用いたエラー処理に関してまとめる．

- `p <|> unexpected "msg"`の場合は，unexpected，expecting メッセージ両方出力され，unexpected メッセージが `msg` に置き換わる．
- `p1 <|> (p2 >> unexpected "msg")` の場合は，`msg` に置き換えられた unexpected メッセージのみがエラーメッセージとして表示される．この場合，`p2` 以外の間違いがあったときに expecting メッセージにおける予期していたトークンの候補として `p2` が出力されてしまうので，<?>を用いて `expecting` メッセージを書き換える必要がある．
- `try` コンビネータが用いられているパーサに `unexpected` コンビネータを用いたエラー処理の挿入は現状ではできない．

4.　考察

4.1　エラーメッセージの考察

本論文では，`unexpected` コンビネータによるエラーメッセージの置き換えを中心に検証した．置き換え後の内容は，セミコロンのつけ忘れにおける置き換え例の「`unexpected letter other than ;.`」ように，まずは予期したトークン以外のものが現れたということをそ

のまま英文で示し，次に「前の式に；はついていますか．」と日本語でソースコードを修正するためのヒントを与える，というものである．比較例を見ると，エラー処理を加えた後のメッセージでは，エラー処理を加える前のものにはない原因の指摘をある程度できていて，本研究で提示した言語指向のエラー報告に近づけられていると考えることができる．また，C 言語のサブセットの制約に関するエラー処理については，あくまでこの言語に限られたものであるが，フルセットの C 言語に慣れた利用者がサブセットで対応していない構文を誤って使用することが予想される．そうした誤り方を予測したエラー処理の挿入方法の 1 つとしてみることができる．

ただし，行った検証内容ですべて過不足なくエラー報告を言語指向にできているわけではなく，改善の余地が存在する．例えば，文を囲う{}が閉じ忘れられていない場合のエラー報告の改善を行おうと検証した結果，思わぬところで unexpected コンビネータによるエラー処理が動作したり，そもそもエラー処理が動作しなかったりと多くの問題が発生した．

こうした結果から，unexpected コンビネータを用いてユーザのあらゆる入力パターンに備えることは難しいが，「想定される間違い」を優先して入れていくことにより，従来のパーサよりも親切なメッセージを発行できるようにすることが重要な点であると考えられる．

4.2 選択コンビネータ <|> と言語指向のエラー報告

p1 <|> p2 <|> p3 というパーサがあった場合，通常，p1 で解析が失敗した場合はそのまま p2，p3 で構文解析を続けて行おうとする．対して本研究では，p1 の中に unexpected コンビネータが用いたエラー処理があり，それが動作した場合は p2，p3 で構文解析が行われることなく，エラーメッセージを置き換えた上でプログラムを終了させる．それにより，結果的に言語指向のエラー報告を出力している．以上のことは，3.1 節でも述べた通りである．

しかし，これでは選択コンビネータ <|> で結ばれている，本来解析が行われるはずの p2，p3 の結果を見ないまま p1 で先に失敗させていることになる．もう少し視野の広い報告にするために，p2，p3 まで構文解析を試してその結果をまとめ，構文エラーに対して最もふさわしいエラーメッセージを出せるようにすれば，より良い言語指向のエラー報告にすることができると考えられる．ただし，そのような検証を行うためには 3.4 節でも述べた try コンビネータのバックトラックによる影響を考えなければならない場合もあるため，まずはこちらの問題の解決が必要となる．

4.3 発展

エラー報告を改善するための，エラーコンビネータを用いたエラー処理の挿入方法は一つではなく複数存在する場合がある．

例えば，図 9 のようなエラー処理は，文パーサ stmtparser における return 文を受理するパーサの後に挿入している．これは unexpected コンビネータの挿入を複数の場所（式文パーサの内部など）で試したのち，ここが最もふさわしいであろうというパーサ実装者の主観に基づいて行われている．本論文の実験で示したエラー処理の挿入方法が最もふさわしいと示すことは現段階ではできていない．

そのような場合に，それぞれの方法のうちどれが最もふさわしいかを計測して，比較する方法を確立できれば，1 つの指標を作ることができる．それが実現すれば，パーサ実装者が構文に対してより適切で効率的な注釈付けをしたり，異常系処理の実装の負担を軽減したりすることができる可能性がある．

あらかじめ想定されるユーザの間違いも幅があるため発展的内容といえるが，より客観的な視点で評価を行うためには，こちらも重要な研究課題といえる．

5. おわりに

本論文では C 言語のサブセットを Parsec で実装した上で，ライブラリに用意されているエラーコンビネータ unexpected を利用して言語指向のエラー報告を行うための方法の検証を，さまざまなコード例で行った．それを踏まえて，unexpected コンビネータを加える前と加えた後のメッセージを比較をすることで，どのような差別化を図ることができているかの考察を行った．

今後の課題としては，try コンビネータによるバックトラックの影響で，unexpected を用いたエラー処理を挿入することができなかった問題の解決が挙げられる．try コンビネータの動作についての理解を深め，この問題を克服することができれば，ユーザに親切なエラー報告を提供することのできる範囲が広がる可能性がある．

謝辞 本研究は JSPS 科研費 15K00488 の助成を受けたものである．

参考文献

[1] Leijen Daan. *Parsec, a fast combinator parser*. Department of Computer Science,Utrecht University（RUU）, 2001.
[2] Stephen C Johnson. *Yacc: Yet another compiler-compiler*. Murray Hill, NJ: Bell Laboratories, 1975.
[3] parsec: Monadic parser combinators. https://hackage.haskell.org/package/parsec.
[4] attoparsec: Fast combinator parsing for bytestrings and text. https://hackage.haskell.org/package/parsec.
[5] megaparsec: Monadic parser combinators. https://hackage.haskell.org/package/megaparsec.