令和4年度 学士論文

# プログラミングエラーメッセージの読解を促進する初学者向け演習形式の提案

東京工業大学 情報理工学院
数理・計算科学系

学籍番号 18B10103

角田 和広

指導教員

増原 英彦 教授

令和4年2月3日

# 概要

プログラミング時のエラーメッセージは初学者にとって理解しづらい。この問題に対してメッセージの工夫やグラフィカルな表示などの研究が行われてきた。しかし未だにエラーメッセージの改善や初学者の支援について多くの研究が必要とされている。

本論文では、プログラミング教育で使用できるエラー生成クイズという新たな問題形式を提案する。この形式の特徴は、学習者がエラーのあるコードを修正するのではなく、正しいコードを変更しエラーを発生させるという点にある。

我々はこの問題形式により、初学者がエラーメッセージを読み、理解する力を養うことを期待している。また、実際にプログラミングを学んでいる学生にエラー生成クイズを解いてもらい、その様子の観察とインタビューを行った。その結果から、エラー生成クイズという問題形式の特徴を考察した。

# 謝辞

本研究を進めるにあたり多くのアドバイスやご指導をしていただいた増原英彦教授、叢悠悠助教に心より感謝申し上げます。

また、日頃から多くの助言をしていただき、本研究の観察実験の予行練習にも協力いただいた増原研究室の皆様にも感謝いたします。特に研究の方針や進め方について親身に相談に乗っていただいた田辺裕大さん、伊澤侑祐さんのお二方には感謝の念が絶えません。

# 目 次

# 図 目 次

# 表 目 次

# 第1章　はじめに

我が国の小・中学校でプログラミングが必修科目となった[1] [2]ことに代表されるように、プログラミングを学ぶ人は今後ますます増えることが予想される。そのため、初学者がプログラミングに取り組みやすくなることの必要性は高まっている。

エラーの解決はプログラミング初学者が苦労する点の 1 つである。エラーを解決するためには、エラーが発生する場所と誤りの原因を発見し、問題が起こらぬように修正しなければならない。そのため文法知識などが不十分な初学者にとって自力でエラー修正を行うのは難しく、初学者に向けた支援が重要である。

プログラミングエラーメッセージとは、ソースコードに存在する誤りをプログラマーに伝える言語処理系からの出力である。一般にエラーメッセージは、ソースコード上のエラーの原因箇所や誤りの理由を保持している。そのためプログラマーにとってはエラーを解決するための重要な手がかりである。しかしプログラミング初学者は、エラー解決の際にエラーメッセージを活用できず、そもそも内容を理解していないことがある。例えば Nienaltowski [18] は、初学者にとってはエラーメッセージが過剰に簡潔で、十分に視覚的でなく専門的すぎると述べている。本来、エラーメッセージは初学者が活用できる、プログラミングをする上での良きパートナーとなるべきである。

初学者のエラー解決に関する既存研究はいずれも初学者がエラーメッセージを注意深く読むことを想定している。例えばエラーメッセージの単純化・詳細化の研究 [6] や、デバッグ型演習 [13] などである。

本研究は初学者がエラーメッセージを注意深く読まない点に注目する。例えば、Marceau ら [15] が指摘したように、初学者はソースコード内のエラー箇所としてハイライトされている部分を変更する傾向がある。この際、初学者がエラーメッセージを全く読まずに、「少なくともハイライトされた部分に『間違い』があることは真実」だと考えて変更を加えることが確認されている。また、エラーメッセージが複雑な場合、初学者はエラーメッセージの主要でない一部のみを読むことが予想される。実際に本

[1] https://www.mext.go.jp/a_menu/shotou/zyouhou/detail/1375607.html

[2] https://www.mext.go.jp/component/a_menu/education/micro_detail/__icsFiles/afieldfile/2019/03/18/1387018_009.pdf

Q2 : ソースコード

```
1 abstract class FTree
2 case class End(name: String) extends FTree
3 case class Child(n: String, f: FTree, m: FTree) extends FTree
4
5 def attach(tree: FTree): FTree = {
6   tree match {
7     case End(name)       ⇒ End(name + "san")
8     case Child(n, f, m) ⇒ Child(attach(f), attach(m))
9   }
10 }
```

エラーメッセージ

```
[error]…/testq2.scala:8:31: not enough arguments for method apply:
(n: String, f: FTree, m: FTree)Child in object Child.
[error] Unspecified value parameter m.
[error]     case Child(n,f,m) => Child(attach(f),attach(m))
[error]                                ^
```

図 1.1: 複雑なエラーメッセージ

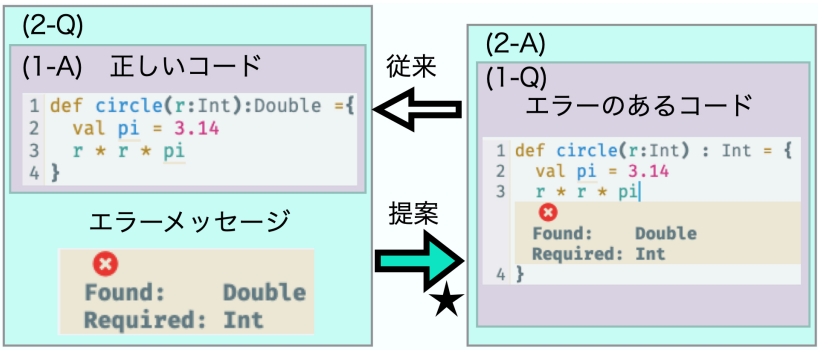



図 1.2: エラー生成クイズ

研究の予備実験内で、図 1.1 のエラーメッセージを見たプログラミング歴の短い学生は、このエラーメッセージの主要でない一部分のみを読んでいたことが確認された (詳細は 6.2.1 節)。

本研究では**エラー生成クイズ**を提案する。この問題形式は初学者へのエラーメッセージの理解の支援を目的としており、そのために解答者がエラーメッセージを必ず読むよう設計している。特徴として、エラー生成クイズでは、初学者は示されたエラーメッセージと正しいコード (2-Q) から、示されたエラーメッセージを**発生**させるコード (2-A) を作成する (図 1.2 ⋆)。これはデバッグ型演習で、初学者がエラーのあるコード (1-Q) を修正し正しいコード (1-A) を作成することと対称的である。

まず、エラーメッセージの読解を 5 つの段階に分解した**エラーメッセージ読解モデル**を仮定した。そして初学者がエラーメッセージ読解モデルの各段階を実行するように、エラー生成クイズを設計した。また、エラー生成クイズがエラーメッセージ理解支援を促すか確認するための第一歩と

して予備実験を実施した。観察の結果、エラー生成クイズは問題として成立し、初学者にエラーメッセージを読むことを促す効果があることを確認した。

以降では、第 2 章でエラーメッセージと初学者間の問題の背景について、第 3 章でエラーメッセージ読解モデルについて、第 4 章でエラー生成クイズについて述べる。その後、第 5 章で実施した予備実験の詳細について、第 6 章で予備実験の結果と考察を述べる。そして第 7 章で関連研究について、最後に第 8 章で将来の展望を述べ、9 章でまとめを述べる。

なお、本稿で登場するプログラムコードは全て Scala 言語で書かれたコードである。

# 第2章　背景

エラーメッセージがプログラミング初学者にとってエラー解決のための重要な支援の要素であることは先に述べたとおりである。しかし現実には、初学者がエラーメッセージを活用する際に問題となる点がいくつも指摘されている。代表的な問題点として、エラーメッセージ自体のわかりづらさと、初学者がエラーメッセージと遭遇した時の知識不足という 2 点がある。これらの問題点は、そのまま初学者のエラーメッセージ読解を困難にすることにもつながっていると考えられる。

## 2.1　エラーメッセージのわかりづらさ

まず問題視されているのはエラーメッセージ自体のわかりづらさである。これは過去数十年にわたる論文内で、現行のエラーメッセージについて肯定的な意見がほとんどない [6] ことからも読み取れる。その要因として、様々な点が挙げられている。

1 つの要因として、最も基本的な点であるエラーメッセージの文章自体の読みやすさが十分に研究されていないことが挙げられる。この読みやすさについて研究した Denny ら [8] は「専門用語の排除」、「完全な文章での提示」、「単純な語彙の使用」、「程よい語数」(筆者訳) を意識したエラーメッセージ作成のガイドラインが必要であると主張している。エラーメッセージの研究が長年行われてきたにも関わらず、文章として読みやすいエラーメッセージのガイドラインを示したのはこの論文が初めてである。

その他の大きな要因として、エラーメッセージの正しさと、有用さについての問題がある。

正しさについて図 2.1 に代表されるように、エラーメッセージの内容が実際の誤りの原因と必ずしも一致していないことが問題点として挙げられる。この図の場合、実際の誤りの原因は `isEqual` 関数 2 つの引数 `x` と `y` の間に “,”が不足していることによるものである。しかし発生しているエラーメッセージに “,”がないことに言及するものはない。

また有用さについて図 2.2 に代表されるように、エラーメッセージ内にエラー解決の手がかりとなる情報が不足していることが問題点として挙げられる。この図のエラーメッセージ内には、このエラーが型に関するもの

ソースコード：

```
def isEqual(x:Int b:Int): Boolean ={
  x == y
}
```

エラーメッセージ：

```
- identifier expected but ':' found.
    def isEqual(x:Int b:Int): Boolean ={
                      ^

- ')' expected but '}' found.
    }
    ^
```

図 2.1: 原因の不一致

ソースコード：

```
def append(x:Boolean, list:List[Int]): List[Int] ={
  if(x) x::list
  else list
}
```

エラーメッセージ：

```
- type mismatch;
  found   : Boolean
  required: Int
  if(x) x::list
        ^
```

図 2.2: 情報不足

で、Int が必要な場所で Boolean が見つかったという情報がある。一方で、具体的にこのエラーで言及している Boolean がどの要素の型なのかや、Int を要求しているのはどれなのかという情報は（曖昧な ^ マーク以外に）示されていない。

なお現実的には、これらの問題に対処できるような常に確実に正しくかつ詳細なエラーメッセージを表示することは困難である。実際、一般の状況ではエラーメッセージの正しさと詳細さはトレードオフの関係にある可能性が指摘されている [16]。これは誤りがある時に、その間違いがどのような文脈で発生しているかを前もって知ることができないためである。つまり、エラーメッセージを詳細にすると一部の文脈によっては正しさが損なわれ、どんな文脈でも常に内容が正しいことを目指すと内容を詳細にすることが難しくなる。

## 2.2　初学者の知識不足

初学者のエラーメッセージを理解するための知識不足は、初学者がエラーメッセージ活用を試みる際のもう一つの代表的な問題点である。Marceau ら [15] はエラーメッセージ内に頻出する 15 個の用語の理解度を調べた。すると 50%以上の学生が正解したのは 4 個のみという結果であった。これは回答者全員が少なくとも 2ヶ月間そのエラーメッセージが発生する環境を使用していたにもかかわらず、確認された。

このような知識の不足が起きてしまう最も大きな要因として、初学者がエラーメッセージと事前に遭遇する機会の不足が挙げられる。例えばプログラミングを学ぶ大学生の場合、プログラミングの講義では時間的制約のためエラーメッセージの例示や説明が行われることは稀である。実際、プログラミング教育では意図的に間違ったコードを見せることがあまり行われていないと指摘 [10] されている。このため、学生は特に演習の前半数回で、知識が十分に得られていない状態でエラーメッセージと遭遇する。こ

れにより一度知っておけば簡単に理解できるエラーメッセージであったとしても、学生はその知識を得る機会がないために大幅な苦労をする恐れがある。Marceau ら [14] は、学生は講義最初の 3 週間で特にエラー解決に苦労しており、また新たに学んだ文法に特有のエラーメッセージが出た場合にエラー解決に苦労する割合が高いという報告をしている。ここから、遭遇機会の不足が実際に問題となりうることが伺える。

そのほかの要因として、特に先に例示したような語彙についてのものがある。それはエラーメッセージ内に出現するプログラミング用語の一部が、講義等で使用されていない、あるいは別の表現で使用されている事である。実際に Marceau ら [15] の語彙の実験結果で、授業内で使用されていない、あるいは別の表現使われていた用語が複数あり、その用語の理解度が低くなることが確認されている。

そしてこの語彙の問題は特に講義を英語以外で受ける学生にとって大きな影響を与えることが考えられる。それは、ほとんどのプログラミング言語のエラーメッセージは英語であるため、講義が英語以外の時はほぼ全ての用語が他言語に言い換えられていることになるためである。そのため、英語以外を母語にするユーザへのプログラミング用語の語彙の負担軽減を目指すべきという提案がなされている [11, 8]。

# 第3章　エラーメッセージ読解モデル

エラーメッセージを必ず読む問題を作成するために、まずエラーメッセージの読解に明確な定義を与える。

本研究ではエラーメッセージの読解を以下の5つの段階からなると仮定する。これをエラーメッセージ読解モデルと呼ぶ。

**(i) エラー位置の確認**　コード上のエラー位置を確認する

例:エラーは2行目で発生している (図 3.1)

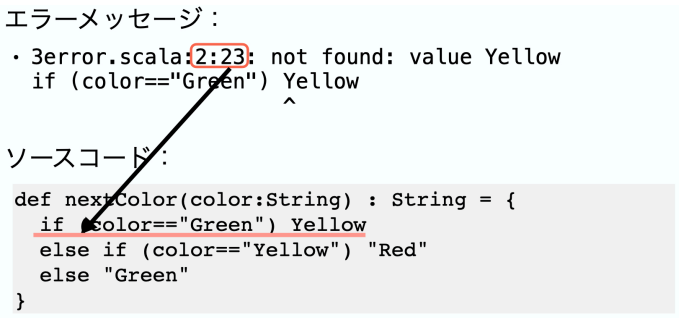



図 3.1: エラー位置の確認

**(ii) 語句の対応づけ**　エラーメッセージ内の語句とコードの語句を対応づける

例:メッセージ内の `value Yellow` とは2行目の `Yellow` を指す (図 3.2)

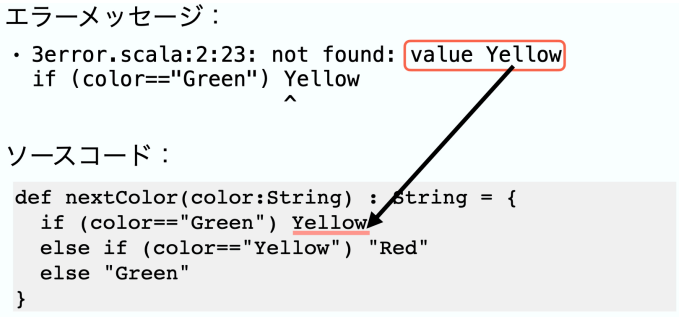



図 3.2: 語句の対応づけ

**(iii) 言語規則の推測** エラーを起こす言語規則 (文法や型規則) を推測する

例:事前に定義していない `value` の `Yellow` を使用していることによるエラー (図 3.3)

エラーメッセージ：

```
- 3error.scala:2:23: not found: value Yellow
  if (color=="Green") Yellow
                      ^
```

ソースコード：

```
def nextColor(color:String) : String = {
  if (color=="Green") Yellow
  else if (color=="Yellow") "Red"
  else "Green"
}
```

図 3.3: 言語規則の推測

**(iv) コードへの当てはめ** 推測した言語規則の誤りがコード上で実際に発生しているかを確認する

例:確かに `Yellow` は事前に定義されておらず、事前に定義していない `value` がある (図 3.4)

エラーメッセージ：

```
- 3error.scala:2:23: not found: value Yellow
  if (color=="Green") Yellow
                      ^
```

ソースコード：

```
def nextColor(color:String) : String = {
  if (color=="Green") Yellow
  else if (color=="Yellow") "Red"
  else "Green"
}
```

図 3.4: コードへの当てはめ

**(v) 修正の考案** 規則違反を解消するためにどのような修正を行えば良いか考える

例:`Yellow` の`""`がないため、`"Yellow"`とすれば未定義の `value` がなくなりエラーが解決する。(図 3.5)

エラーメッセージ：

```
- 3error.scala:2:23: not found: value Yellow
  if (color=="Green") Yellow
                      ^
```

ソースコード：

```
def nextColor(color:String) : String = {
  if (color=="Green") Yellow
  else if (color=="Yellow") "Red"
  else "Green"
}
```

図 3.5: 修正の考案

エラーメッセージに遭遇したときに全ての要素を実施していれば、普段からエラーメッセージを読解しているとみなす。

エラーメッセージ読解モデルは、Marceau らの提唱した概念モデル [14] の要素を具体化する形で仮定した。この概念モデルはエラーメッセージがどのようにして学生の役に立つかを示しており “*Read*”、“*Understand*”、“*Formulate*” という要素があると提唱した。また要素の具体化の際、特に (ii)、(iii) を仮定する際に関連研究を参考にしている。Wrenn ら [21] の実験で、対応する語句をハイライトで色付けしたエラーメッセージがエラー解決時に学生からの肯定的な評価が多かった。また Marceau ら [15] の実験で、エラー解決にわずかながら対応する語句のハイライトが有効だったという結果もある。ここから、(ii) はエラーメッセージを読み理解する上で重要な段階であると考えた。また、Marceau ら [14] の実験の結果に、学生が未知の文法を学んだ直後のエラーでより解決に苦労するという結果がある。これは、新たな文法を学ぶことで (iii) を実施するのが困難になったからではないかと考えられ、エラーメッセージの読解の中で (iii) も重要であると考えた。

# 第4章　エラー生成クイズ

本章で、我々が提案するエラー生成クイズの詳細を述べる。なお本稿にあるエラー生成クイズのエラーメッセージは、ブラウザ上で Scala を実行できる環境 Scastie[1]で発生するものを用いている。

## 4.1　エラー生成クイズの概要

図 4.1 がエラー生成クイズの概観であり、エラー生成クイズは (a) 元コードと (b) 目的エラーメッセージからなる。

エラー生成クイズは、初学者がエラーのあるコードを修正するのではなく、正しいコードを変更しエラーを発生させるというものである。解答者は (a) 元コードと (b) 目的エラーメッセージを受け取り、元コードの一部を変更して、正解となる (c) 誤りがあるコードを作成することを目指す。

(a) 元コードは、正しいソースコード、つまりコンパイルや動作時にエラーや警告が発生しないコードである。解答者はこのコードを元にしてクイズに取り組む。(b) 目的エラーメッセージは実際に存在するエラーメッセージ、もしくは警告メッセージ[2]である。解答者はこのメッセージを発

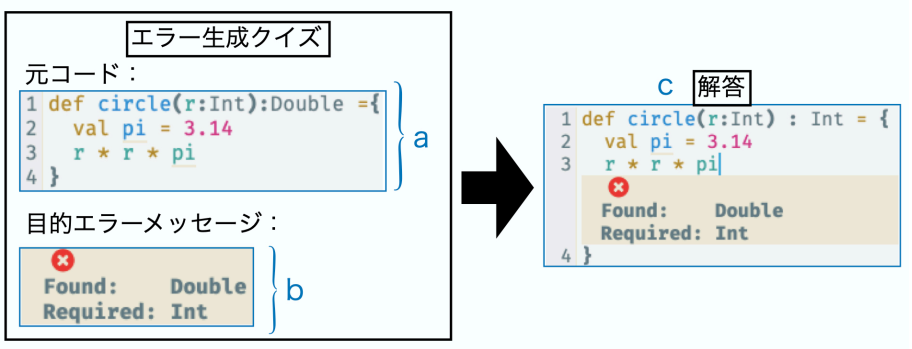



図 4.1: エラー生成クイズの概観

[1] https://scastie.scala-lang.org

[2] 以降エラー生成クイズについて言及するときは、エラーメッセージの場合のみ記すが、警告メッセージも同様に使用する。

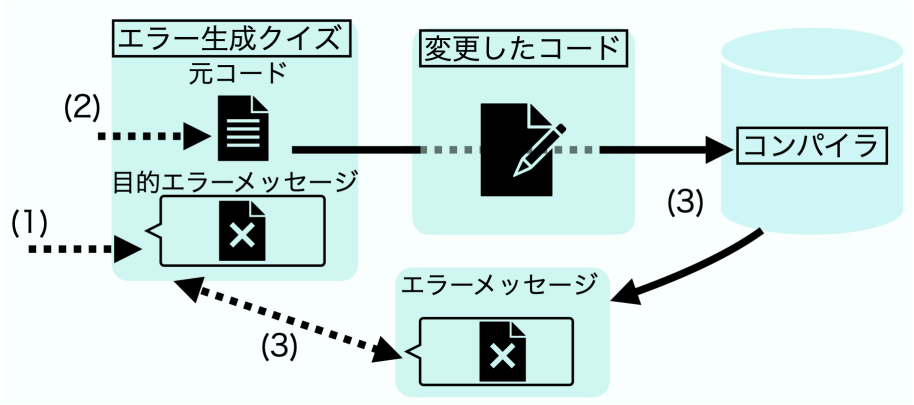



図 4.2: エラー生成クイズの取り組み方

生させることを目指す。

## 4.2　エラー生成クイズの取り組み方

エラー生成クイズの取り組み方を図示したものが図 4.2 である。実線がコンピュータ上で操作を行う内容で、破線は解答者が頭の中で考えることを表している。以下が手順の説明である。

(1) 目的エラーメッセージの理解：目的エラーメッセージを読み、違反した規則を仮定する。

(2) 元コードの変更：分析した原因を発生させるように、元コードの変更箇所・変更内容を考える。

(3) 出力の確認：変更後のコードでコンパイルを試行し、その出力が目的エラーメッセージと等しいか確認する。同じならその問題に正答できたことになるため、その問題を終了する。エラーメッセージが発生しない、発生したが目的エラーメッセージと異なるという場合は (1) に戻る。

## 4.3　エラー生成クイズにおける読解

エラー生成クイズは、初学者がエラーメッセージ読解モデルの各段階を忠実に実行するように設計している。これはエラー生成クイズで行なわれ

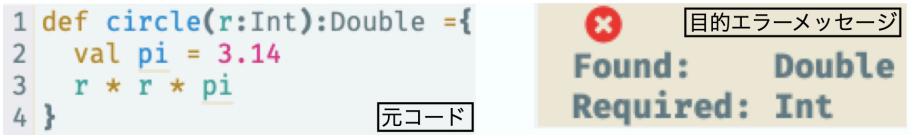



図 4.3: 問題例

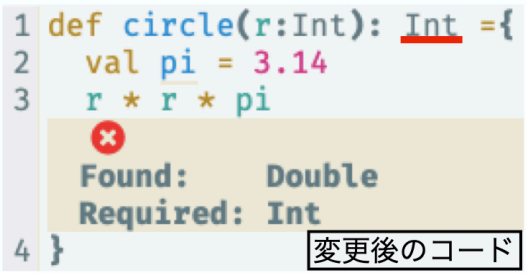



図 4.4: 問題例の解答

ることが想定される 3 つの行動にエラーメッセージ読解モデルの各段階が含まれていることによる[3]。図 4.3、図 4.4 を例としてこれを示す。

**目的エラーメッセージの理解**では、Int が要求されているが Double が見つかったと記述されていることを読み取る。ここから、作るべきエラーは型の不一致によって発生するという事を仮定することになる。この際、エラーを起こす言語規則の推測が行われておりエラーメッセージ読解モデルの **(iii)** が行われている。

**元コードの変更**では、Int が要求される場所に Double があるという型の不一致エラーだという分析をもとに考える。この図の場合元コードでは、関数の返り値の部分に Double 型が登場する事がわかるが、Int 型が要求されている場所は見つからないこともわかる。そのため、Int 型を要求するような場所を作れば良いと考える事ができ、返り値の Double を Int に変えれば良いと考えられる。この際、エラーメッセージに書かれた Double 型が登場するコード上の位置を探すことで読解モデルの **(ii)** が行われている。さらにエラー原因である、Int を要求している場所がないか探すことで **(iv)** が行われている。また、Int を要求している場所を作成するために返り値の型を Double から Int に変えれば良いと考えることで **(v)** の逆が行われている。

**出力の確認**では、返り値の部分で目的エラーメッセージと同じエラーメッセージが発生していることを確認する。この際に発生したエラーメッ

[3] 厳密には (v) のみ逆向きの行動になるが、実施される内容としては同じため、各段階が含まれるといえる。

セージを確認 (i) する。

## 4.4　エラー生成クイズの作成指針

現在、エラー生成クイズにはいくつか作成の指針がある。その内容と、それぞれの理由を示す。これらの指針は暫定的なものであるため、将来的に内容を変更する可能性がある。

**指針 1**　それぞれの問題で解答時は元コードの 1 箇所のみを変更する、という条件を設ける。これは、エラー生成クイズは問題としての難易度が全体的に高い可能性があり、想定する解答に複雑なコード変更を要求すべきでないと考えたためである。またコードの大部分を変更できてしまうと、予期していない別解が多く発生する可能性があり、それを防ぐことにもつながる。

**指針 2**　解答者が問題のエラーメッセージに集中できるよう、可能な限りそれぞれの問題内の元コードはシンプルにする。

**指針 3**　エラーメッセージが原因を表していない状態が想定解となる問題は作らない。これは、問題形式上エラーメッセージが原因を表していない状態 (図 2.1 のような状態。“,”がないことが原因だがエラーメッセージはそれに言及していない。) が想定解になる場合、元コードから想定解を導くことが著しく困難と思われるためである。

# 第5章　予備実験：アンケートとエラー生成クイズの実施観察

## 5.1　予備実験の目的

アンケートは、学生のエラーメッセージ読解モデルの実施率を調べることを目的とした。その中で、エラーメッセージ読解モデルがエラー生成クイズの取り組み方と関係性があるか調査することも目的とした。

エラー生成クイズの実施観察は、作成したエラー生成クイズを実施する際に、問題が成立しなくなるような重大な欠陥がないか確認することを目的とした。その上で、エラーメッセージの理解促進の一歩目として、エラー生成クイズがエラーメッセージを読むことを促進するか確認することも目的とした。

## 5.2　予備実験の流れ

初学者として、東京工業大学数理・計算科学系、学部2年生むけの講義「プログラミング第一」[1]の受講生を対象とした。講義「プログラミング第一」の中のある演習回で、受講生全体にアンケートの説明と協力の呼びかけを行なった。その後アンケート内でエラー生成クイズの観察実験に協力してよい、と答えた学生3名に対して連絡を取り観察実験を行なった。

## 5.3　アンケート

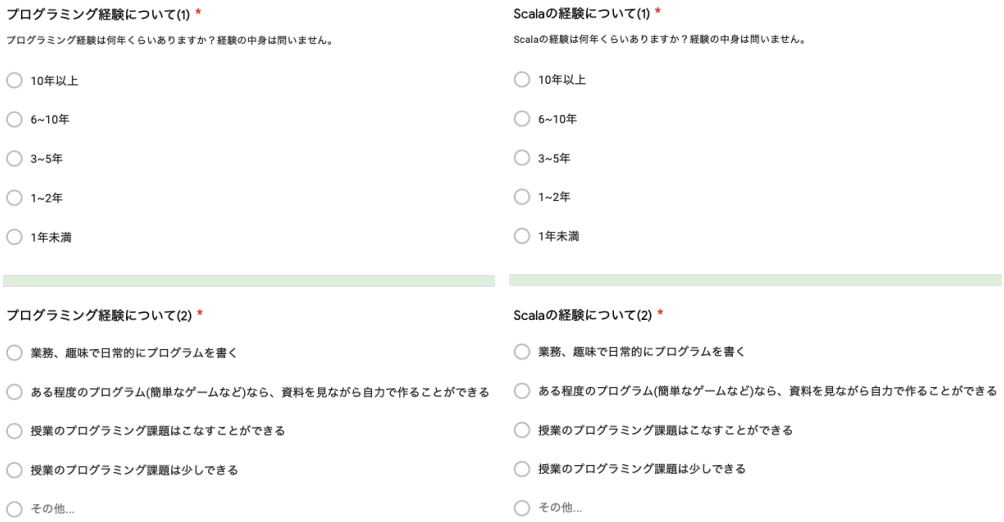
まずアンケート開始前に、エラー読解モデルの各要素について10分ほど学生に説明を行なった。

アンケート調査にはGoogleフォームを用い、以下の内容を尋ねた。

[1]「プログラミング第一」ではScalaを用いて、状態やイベント処理、高階関数などを学ぶ。また、「プログラミング第一」を受講している学生はこれ以前に「計算機科学概論」という講義でScalaの関数定義や条件文、パターンマッチなどを学んでいるか、それに準ずる知識を持っている。

**プログラミング歴**　言語不問と Scala について、それぞれ当てはまるプログラミング歴の選択肢を選ぶ形式 (図 5.1 上)。

**プログラミング経験の深さ**　言語不問と Scala について、それぞれ当てはまるプログラミング経験の深さの選択肢を選ぶ形式 (図 5.1 下)。




(a) 言語不問　　(b) Scala

図 5.1: プログラミング歴・経験

**エラーメッセージの印象について**　プラスの印象 4 種、マイナスの印象 4 種、その他 (自由記述)、の中から複数回答 (図 5.2)。

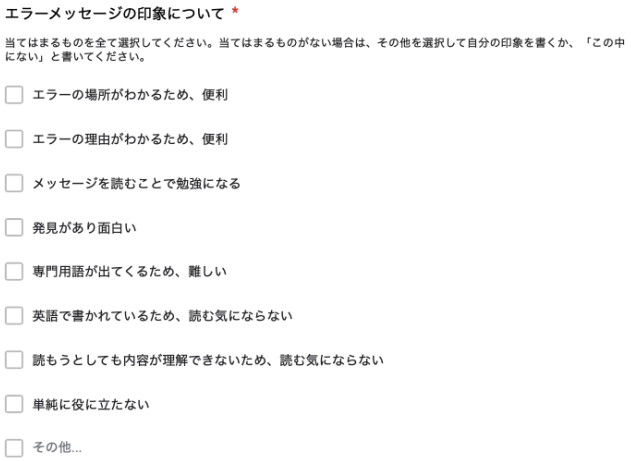



図 5.2: エラーメッセージの印象

**エラー読解モデルの実施割合**　演習中などに見る全てのエラーメッセージで平均して、それぞれの段階をどの程度の割合で実施しているかを%で記述 (図 5.3、例として (1) のみ)。

なお学生には、エラーメッセージを見たときに以前見たことがあるなどで解決法が読解途中、あるいは読解せずとも分かったという経験については、全ての読解の要素を実施して理解した扱いと考えるよう伝えた。


図 5.3: 実施割合

## 5.4　観察とインタビュー

アンケート内で観察実験に参加しても良いと答えた学生に後日連絡を取り、遠隔会議システム Zoom を用いて観察を行なった。協力者には実験当日以前に、実験の説明ビデオを視聴するよう伝えた。当日の観察は以下の手順で行った。なお全手順で第一著者が同席した。

1. 読解度測定クイズ (デバッグ形式)
2. エラー生成クイズ
3. インタビュー

それぞれの手順の詳細を示す。

**読解度測定クイズ**　実験当日、初めにエラーを修正するデバッグ型の問題を 2 問実施した。図 5.4, 図 5.5 の 2 つの問題を順に画面共有で提示し、ソースコードのどの部分をどのように変更すればエラーが発生しなくなるかを口頭で答えるよう求めた。1 問目 (図 5.4) は 3 分間の回答時間を設け、時間内に解けなかった場合は解説を行った。2 問目 (図 5.5) は、学生が解答がしたかどうかに関わらず 25 秒経過した時点で問題を隠し、2 問

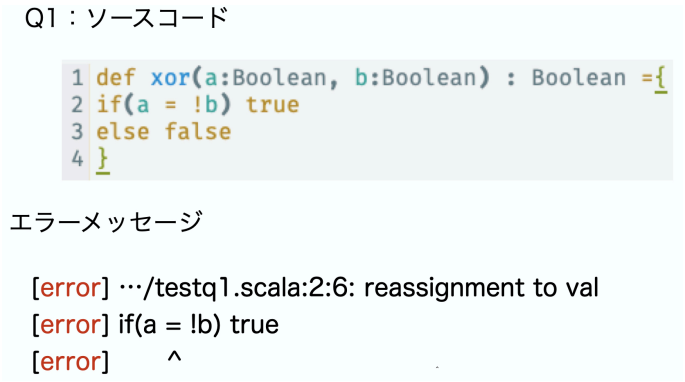



図 5.4: エラーデバッグクイズ 1(制限時間 3 分)

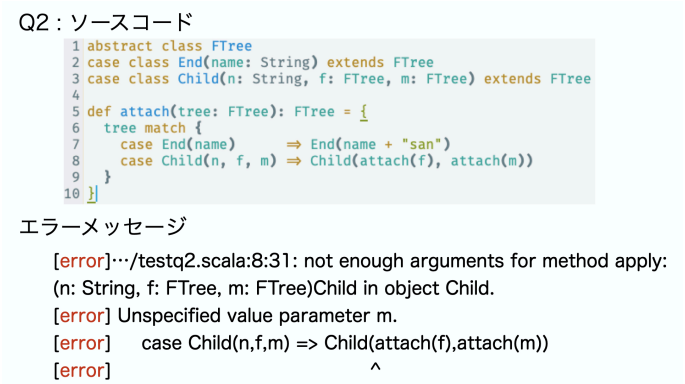



図 5.5: エラーデバッグクイズ 2(制限時間 25 秒)

目のエラーメッセージに何が書いてあったかを覚えている範囲で答えるよう求めた。その後 2 問目は答えなくて良いことを説明し、2 問目のエラー部分の解説をして次の手順に進む。

このクイズは参加者へのデバッグ型演習の紹介と、参加者の客観的なエラーメッセージ読解度を計測するために行った。

**エラー生成クイズ**　読解度測定クイズの後、エラー生成クイズを実施した。この手順では解答時の行動を観察するため、学生に Zoom でデスクトップの画面共有をするよう求めた。エラー生成クイズは学生 1 人につき 6 問実施し、各問題に 5 分の時間制限を設けた。今回使用した問題は図 5.6 のような形である。学生は Scastie 上で元コードの一部を変更し、実行することを繰り返す形で各問題を解いた。

各問題には 3 章で述べた指針に加え、以下のような特徴がある。

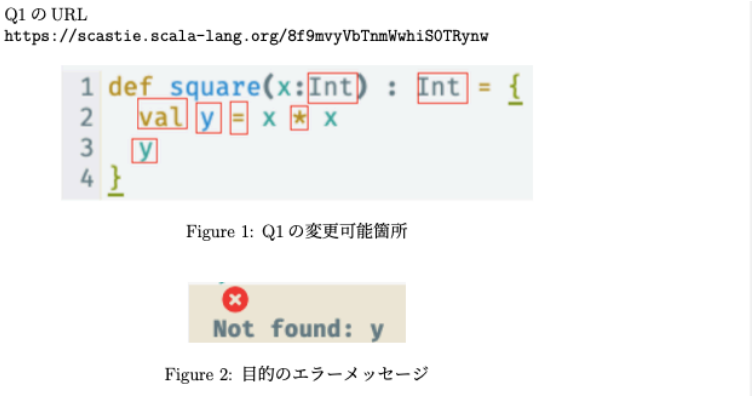



図 5.6: 観察時使用クイズ

**コンパイルエラーのみを扱う**

作成時や出題時のわかりやすさのために、目的エラーメッセージはコンパイル時に発生するものに限る。つまり、実行時エラーで発生するエラーメッセージは扱わない。これは、解答者に要求する動作を増やすことを避けるためである。

**既習の内容のみを扱う**

目的エラーメッセージは、解答者が既に講義で学習している内容の中で発生するものに限る。これは、学習していない文法や機能によって発生するエラーを意図して作成することは非常に難しいと考えられるためである。

**変更可能箇所の設定** (図 5.6 の赤枠)

それぞれの問題で元コードの中で変更して良い場所を指定した。つまり、学生はそれぞれの問題の変更可能箇所 (赤枠) の中で 1 箇所のみを変更して目的エラーメッセージを出すことを目指すという形式にした。これは、事前の観察から初学者に対しては、全ての箇所を変更可能にすると問題の難易度が高すぎると考えられたためである。

**エラーの発生行情報は与えない**

一般にエラーメッセージに含まれるエラーが何行目に発生するかという情報 (この場合は想定解では何行目にエラーが発生するかという情報) は、解答者に与えないこととした。これは、解答者の変更場所決定に影響し過ぎてしまうことを考慮したためである。

またそれぞれの問題実施時、解答者がエラーメッセージをインターネット検索することを許した。

**インタビュー**　最後に、エラー生成クイズについての質問を設けた。それぞれの問題についての質問と、エラー生成クイズ自体についての質問の 2 種類を聞いた。それぞれの問題についての質問では時間の都合上全問題についてではなく、その学生が解けなかった問題と解くまでにコード変更の試行回数が多かったものを優先して 3〜4 問分聞いた。

それぞれの問題に関する質問として、事前に以下を用意した。

- どのような手順で解いたか？
- 変更箇所を決定することに苦労したか？
- 目的エラーメッセージは演習等で見たことがあったか？
- 新たな知識を得られた、あるいは忘れていた知識を思い出せたという問題はあったか？

エラー生成クイズ自体についての質問として、事前に以下を用意した。

- エラー生成クイズを解いたことで、今後エラーメッセージを読むようになると思うか？
- 新たなプログラミング言語を学ぶとき、デバッグ型演習とエラー生成クイズのどちらを使いたいと思うか？
- エラー生成クイズに追加して欲しい要素はあるか？

その他、問題実施の観察時に解答者の行動で気になったものがあった場合、それぞれの問題についての質問時に併せて尋ねた。

# 第6章　実験結果と考察

## 6.1　アンケート結果と考察

講義に参加していた学生にアンケートの協力を求めたところ、30 件の回答を得た。

プログラミング歴、経験の深さについては表 6.1、表 6.2、表 6.3 のようになった。なお、Scala 歴については 30 名全員が 1 年未満だったため表からは省略した。 エラーメッセージの印象については表 6.4 の通りで、グラフにしたものが図 6.1 である。エラーメッセージ読解モデルの実施率についての回答とその平均は図 6.2 のようになった ((e) を 10%以下と答えた学生はいない)。今回のアンケート結果は母数が少ないため、あくまで参考程度となる。しかし興味深い傾向がいくつか見受けられる。

### 6.1.1　結果の詳細と考察

エラーメッセージ読解モデルの実施率についてと、実施率とその他のアンケート内容との関連性に言及する。

#### エラー読解モデルの実施割合

**結果** エラー読解モデルの中で、語句の対応づけと言語規則の推測の実施割合は相対的に低い学生が多くいる。

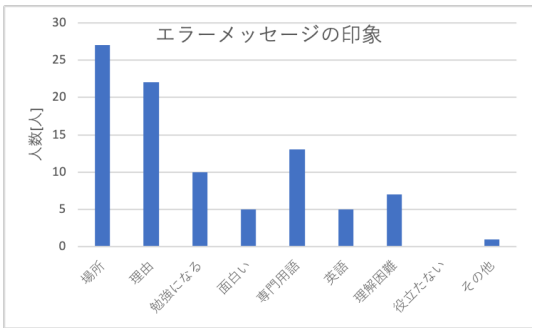



図 6.1: エラーメッセージの印象のアンケート

表 6.1: プログラミング歴 (言語問わず)

| 経験歴 (年) | 人数 |
|---|---|
| 11〜 | 0 |
| 6〜10 | 2 |
| 3〜5 | 5 |
| 1〜2 | 12 |
| 〜1 | 11 |

表 6.2: プログラミング経験 (言語問わず)

| 経験の深さ | 人数 |
|---|---|
| 趣味、業務として | 5 |
| 簡単なゲームなら | 2 |
| 授業課題なら | 11 |
| 授業課題少し | 12 |

表 6.3: Scala 経験

| 経験の深さ (年) | 人数 |
|---|---|
| 趣味、業務として | 0 |
| 簡単なゲームなら | 3 |
| 授業課題なら | 14 |
| 授業課題少し | 13 |

表 6.4: エラーメッセージの印象

| 印象 | 人数 |
|---|---|
| 場所 | 27 |
| 理由 | 22 |
| 勉強になる | 10 |
| 面白い | 5 |
| 専門用語 | 13 |
| 英語 | 5 |
| 理解困難 | 7 |
| 役に立たない | 0 |
| その他 | 1 |

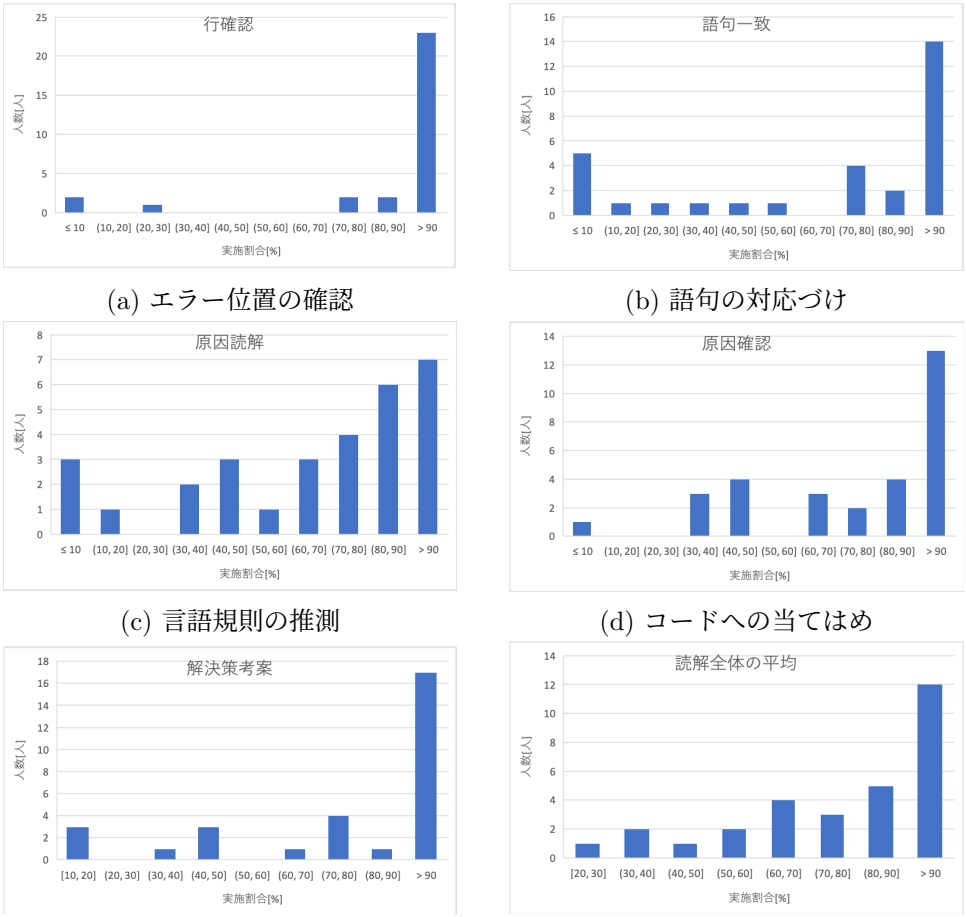
**考察** エラーメッセージ内とソースコードとの語句の一致は行う手間も大きさから敬遠する学生が多いと考えられるためそこまで不思議はない。一方、言語規則の推測はエラー修正において重要な手順であるにもかかわらず、実施している割合が他の手順より低いというのは興味深い結果である。この結果から言語規則の推測の実施率が、エラーメッセージ全体の読解においてより大きな意味を持つ可能性がある。

### プログラミング歴とエラー読解モデル

**結果** プログラミング歴とエラー読解モデルの間に相関関係が見られなかった。

プログラミング歴の長い方から 4〜1 と数字を割り当て、読解実施率の平均とプロットしたのが図 6.3a である。この時相関係数は-0.0030194 であり、相関は全くない。

**考察** これは、プログラミング初学者が読解をしていないのではないかという考えと反しているように見える。このような結果の原因として、




(a) エラー位置の確認　(b) 語句の対応づけ

(c) 言語規則の推測　(d) コードへの当てはめ

(e) 修正の考案　(f) 読解全体の平均

図 6.2: エラー読解モデルのアンケート

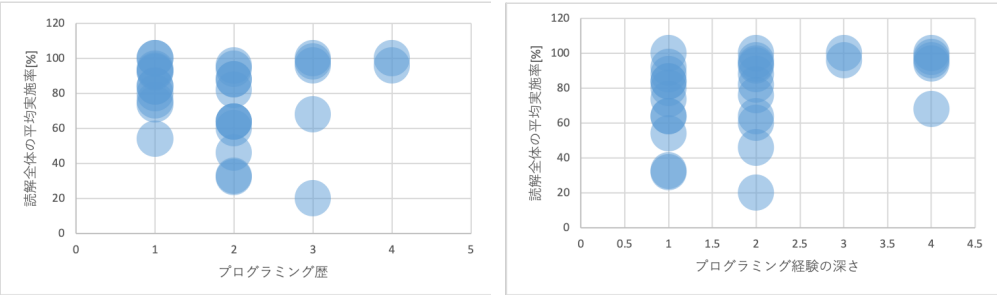
初学者が自身の読解率を正確に測れていない可能性が考えられる。つまり、今回のアンケートは自己評価で書くため、実際にはしっかりと読解ができていない初学者も、毎回できていると考えている可能性がある。その場合、実状に沿わない実施率を書いてしまっていることが考えられる。

### プログラミング経験とエラー読解モデル

**結果** 経験の深さとエラー読解モデルの間には正の相関関係が見られる。

プログラミング経験の深さが深い方から 4〜1 と数字を割り当て、読解実施率の平均とプロットしたのが図 6.3b である。この時相関係数は 0.3675685 であり、弱い正の相関がある (p 値は p=0.04568772 < 0.05)。

**考察**　これは、プログラム歴よりもプログラミング経験の深さの方がエラーメッセージ読解に影響する可能性を示している。または、プログラム経験量があると自己評価がより正確になる可能性を示しているともいえる。




(a) プログラミング歴と読解平均　(b) プログラミングの経験深さと読解平均

図 6.3: 読解度平均の関連性

### エラーメッセージへの印象と読解モデルの実施率

**結果**　エラーメッセージへの印象とエラー読解モデルの実施率に正の相関が見られる。

エラーメッセージの印象でプラスの選択肢 1 つにつき+1、マイナスの選択肢 1 つにつき-1 した合計を総合印象とし、エラー読解モデルの実施率とプロットしたものが図 6.4 である。この時相関係数は 0.506715242 であり、中程度の正の相関がある (p 値は 0.004270013 ＜ 0.05)。

**考察**　これは、エラーメッセージに悪い印象を持つ学生はエラーメッセージの読解を怠る可能性を示している。逆にエラーメッセージの印象改善ができれば、エラーメッセージの読解を促進できる可能性もある。

### エラー位置の確認の実施率との相関関係

**結果**　エラー読解モデルの実施率の中で、エラー位置の確認は経験の深さ、エラーメッセージへの印象いずれとも相関がない。

エラー位置の確認と経験の深さとのプロットが図 6.5a、エラーメッセージの総合印象とのプロットが図 6.5b である。それぞれ相関係数が 0.022770292、0.102553522 であり、上で見た読解率全体の平均と比べても相関がない。

**考察**　エラーメッセージ読解の中でも実施が簡単な段階については経験の影響が少ないため、初学者も自己評価通り毎回実施できている可能性がある。

なお、アンケート結果とエラー生成クイズとの関連性については、関連性を述べるには観察実験を実施できた人数が少なすぎるため言及しないこととする。

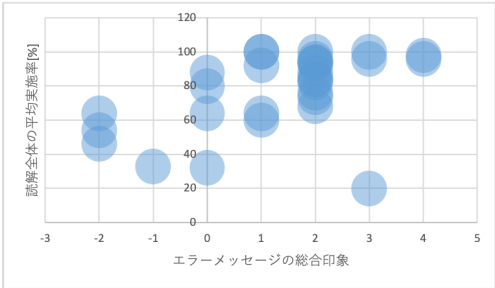
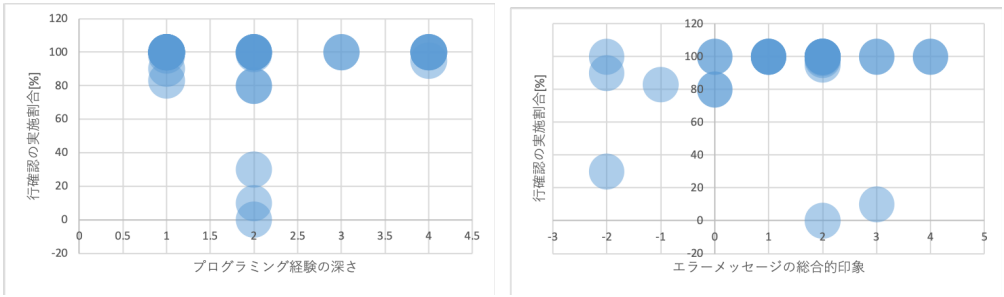



図 6.4: エラーメッセージ印象と読解平均




(a) 経験深さとエラー位置の確認　　(b) 印象とエラー位置の確認

図 6.5: エラー位置の確認との関連性

## 6.2　観察実験の結果と考察

アンケートの実施後、協力できると答えた学生と連絡を取り、計 3 名の学生に観察実験を行った。結果は表 6.5 の通りである。Scala の経験は全員 1 年未満であったため表からは省略した。またそれぞれの学生が解いたエラー生成クイズは同じ問題ではないため、正答数についての考察は行わない。母数が 3 名でありこちらも参考程度の結果だが、興味深い傾向がいくつか見受けられた。

### 6.2.1　読解度測定クイズからの観察と考察

読解度測定クイズでは、初学者のエラーメッセージ読解に関連した 1 つの興味深い傾向があった。

#### 初学者のエラーメッセージ読解実施率と実状との差

**結果** 読解度測定クイズの 2 問目 (図 5.5) を用いた読解度測定にて、プログラミング歴が長くない 2 人の学生はどちらも「パラメータ `m` が未定義/再割り当てされていると書いてあった」と答えた。両者の読解実施率の自己評価に大きく差があったにもかかわらず (実施率平均が 46%と 85%)、どちらもエラーメッセージの内容を主要でない部分から読み取った事になる。一方、プログラミング歴の長い学生は「`Child` に引数が足りていないと書いてあった」と、より主要な原因を答えた。

**考察** この結果は、初学者がエラーのあるコードを修正するときに、エラーメッセージの一部のみを読み、しっかりとは読解できていない可能性を示している。また、自己評価に差があったにもかかわらずどちらも主要でないパラメータ `m` についての情報のみを読み取っていた。これは、初学者が自身の読解実施率を正確に測れていない、もしくはエラーメッセージの一部のみに読解を行いそれだけでエラーメッセージを読解出来ているとみなしてしまっている可能性を示している。

### 6.2.2　実施の観察結果と考察

エラー生成クイズ実施の観察では、解答者の行動の傾向がいくつか確認できた。

表 6.5: 実験結果

| | 学生 1 | 学生 2 | 学生 3 |
|---|---|---|---|
| プログラミング歴 (年) | 〜1 | 1〜2 | 3〜5 |
| 自己評価の読解実施率 (%) | 85 | 46 | 98 |
| ミニクイズ 1 | 正解 | 正解 | 正解 |
| ミニクイズ 2 | 読解が不十分 | 読解が不十分 | 十分読解出来ていた |
| エラー生成クイズ正答数 (6 問中) | 3 | 3 | 6 |

### エラーメッセージ読解と検索

**結果** 開始前に明示的にエラーメッセージの検索を許可したが、いずれの解答者もまずエラーメッセージを読んでおり、読んだ結果として理解できないと感じるまでは検索しなかった。

**考察** この結果は、エラー生成クイズという問題形式で解答者がまずエラーメッセージを読むという期待に即している。

### 目的エラーメッセージの検索とエラーメッセージの理解度

**結果** 解答者が目的エラーメッセージを検索した時、そのまま解に使用できる情報が載っていたことが 1 度あった。この時、解答者は書いてある通りに変更し目的エラーメッセージを出した。しかしインタビューで確認したところ、その変更でそのエラーが出る理由はわかっていないと回答した。

**考察** 答えとなる変更をそのまま検索で発見できたとき、解答者によってはそれ以上考察をしないことがわかった。これはこの問題形式で期待している、エラーメッセージの考察促進を妨げる可能性が高い。また、特に今回検索で見つかった変更は想定解と同じものではなく、解でない変更可能箇所にしていた部分を変更することで発生するものだった。そのため、変更可能箇所の設定時には特にインターネット検索の結果に注意を払う必要がある。

### 問題が解けない時の解答者の行動

**結果** 目的エラーメッセージから変更方法が思いつかない、または思いついたがその変更では目的エラーメッセージが出せなかった場合、解答者は基本的に手を止めた。そして目的エラーメッセージを見直し、どうすれば発生させられるかを考えることに時間を使っていた。そのため、解答者が当て推量で変更可能箇所を書き換えることはほとんどなかった。

**考察** この結果はエラー生成クイズが解答者にエラーメッセージをじっくり読解させることができる可能性を示している。特に目的エラーメッセージを出せなかった際に、解答者にエラーメッセージへのより深い洞察を促すことができる可能性がある。一方で読解した情報があまり有用でない問題は難問になってしまう恐れがある。特に指針で入れないこととした、目的エラーメッセージが原因を指していない問題はエラー生成クイズの問題に向かない可能性がある。

### 6.2.3　インタビュー結果と考察

準備していた質問から得られた結果と考察は以下の通りである。

#### 問題解答時の手順

**結果** 3 名の解答者全員が、エラー生成クイズの問題に、4.2 節で想定した行動と同じ行動をして取り組んだことが確認できた。また、問題を実施する際に進行できなくなるような大きな欠陥は見つからなかった。

**考察** 想定した行動は妥当なものであると思われる。ここから、エラー生成クイズに解答する際は、確実にエラーメッセージを読むといえる。

#### 変更可能箇所の量

**結果** 変更可能箇所が多すぎる、あるいは少なすぎると答えた解答者はいなかった。解答者が変更場所に苦労したことはあったが、その理由は主に「エラーメッセージの意味を読み違えた」、「コードの理解ができていなかった」ことによるものであった。

**考察** 変更可能箇所の数に関しては、極端に多い、あるいは少ないことがなければ解答には影響しない可能性が高い。

#### 目的エラーメッセージが既知かどうか

**結果** 解答者は目的エラーメッセージは演習で見たことがあるものが多かったが、見たことがないものもあった。そして、見たことがあるがエラーの出る状況を忘れていて解けなかった、見たことがなかったが問題中にじっくり読んで理解し問題を解けた、のどちらも確認された。

**考察** 出題するエラーメッセージは、既に習っている範囲で発生するものであれば、既知であるかどうかに関わらず問題として効果的な面が存在する可能性が高い。

#### 問題の知識への貢献

**結果** 解答者全員が新たな知識を得られた、もしくは忘れていた知識を思い出せた問題として複数の問題を挙げた。また得られた知識の内容はエラーメッセージに関するものだけでなく、文法についてのものもあった。

**考察** エラー生成クイズを通して、エラーメッセージの内容が理解できるだけでなく、エラーメッセージが発生する理由を理解することで文法を学ぶことができる可能性がある。

### エラーメッセージ読解の促進

**結果** 参加者全員が問題経験を通してエラーメッセージをより読もうと思える、と答えた。その理由について尋ねたところそれぞれ以下のようなコメントがあった。

- 普通に学習していると学べないことがエラーメッセージから得られる経験をできる。
- 単語を調べて和訳できるようにしたい、そのモチベーションが上がった。
- エラーの簡単/単純な例を知れたために、読む気になる。

**考察** エラー生成クイズはエラーメッセージ読解を促進できる可能性がある。また聞き取った理由が三者三様であったことから、この問題形式にはエラー読解を促進する要素が複数含まれている可能性がある。

### エラー生成クイズの実用性

**結果** エラー生成クイズは面白く勉強になるが難易度が比較的高いため、プログラミング学習の復習として使用したい、という意見が全員に共通していた。そのため参加者全員が基礎的な文法をデバッグ型で、授業の復習や応用としてエラー生成クイズを使えると良いのではと答えた。

**考察** この結果は予想通りであり、我々も学ぶ 1 歩目としてではなく授業の復習等で用いることに向いている問題形式であると考えている。

### 追加して欲しい要素

**結果** 今回、解けなかった問題と想定解と異なる解であった問題について口頭で 30 秒ほどの解説を行なった。それについて「解説を聞くことで納得がいった」などの感想があり、解説のおかげで解けなかった問題からも知識を得られたという感想が得られた。そして参加者全員が追加して欲しい要素として、想定解とエラーメッセージについての解説を挙げた。

**考察** この結果から、エラー生成クイズ作成の指針として解説の作成を追加することが考えられる。

その他、インタビューの中でエラー生成クイズの特徴的な点がいくつか見えた。

### エラーメッセージ内の不明な単語の検索

**結果** 解答者は、エラーメッセージ内に知らない単語があるとすぐに検索をしていた。インタビューで聞き取ると、ある学生は「何度か見かけたことがある単語だったが調べたことがなかった。今回調べて意味が知れたのは良い機会だった。」と言った。

**考察** この結果から、エラー生成クイズは知らない単語を自発的に調べることを促進する可能性が考えられる。これは問題形式から、エラーメッセージにわからない単語があると解くことが非常に困難になることが理由として考えられる。

### 問題の組み合わせによるエラーメッセージへの興味促進

**結果** 図 6.6 にある 2 問の問題はそれぞれ＞と＆＆の左側を異なる型にすることで目的エラーメッセージが発生するため、想定解となる間違い方はほぼ同一である。その一方で、発生するエラーメッセージは大きく異なる。この 2 つを両方解いたある学生はその違いが気になったようで、学生自らなぜエラーメッセージが違うのかという質問をこちらに聞いてきた。

**考察** エラー生成クイズの性質上、エラーメッセージを詳細に見るためその差異が気になるということが考えられる。この効果はエラー生成クイズの特徴となる可能性がある。

### エラーメッセージに対する誤解の観測と、誤解の修正

**結果** エラーメッセージに `list:List[Int]` と表示されたときに、ある学生は前半不思議に見える変更を行なった。インタビューしたところ `list` と `List` の間の `:` がリストの連結に用いる `::` が変化したものと思い込んだため、不思議に見える変更を明確な意図を持って行なっていたということがわかった。本人も解いている間に `:List[Int]` が型を表していることに気づくことができたと報告した。

**考察** `list:List[Int]` の `:` を `::` の変化した形だと思うという間違え方は、初学者がエラーメッセージを読もうとして初めて確認できるものである。このような、エラー生成クイズという形式ならではの発見が多く存在する可能性がある。また、初学者があるエラーメッセージに誤解を持ったとき、エラー生成クイズの問題を解く中で何度も同じエラーメッセージを見直すことになる。それによって誤解に気づくことができるというのはエラー生成クイズのメリットとなる可能性がある。

Q19 の URL
https://scastie.scala-lang.org/tLWtbIFpQyK7EkOiw7gcDQ

```
1 def whichlonger(list1:List[Int],list2:List[Int]) : String = {
2   if(list1.length > list2.length) "list1"
3   else "list2"
4 }
```

Figure 7: Q19 の変更可能箇所

```
None of the overloaded alternatives of method > in class Int with types
 (x: Double): Boolean
 (x: Float): Boolean
 (x: Long): Boolean
 (x: Int): Boolean
 (x: Char): Boolean
 (x: Short): Boolean
 (x: Byte): Boolean
match arguments ((list2 : List[Int]))
```

Figure 8: 目的のエラーメッセージ

Q20 の URL
https://scastie.scala-lang.org/3aRigfLtQxqKD59jIk9VPw

```
1 def int_and(x:Int,y:Int) : Boolean ={
2   int_to_bool(x) && int_to_bool(y)
3 }
4
5 def int_to_bool(x:Int) : Boolean = {
6   if (x == 0) false
7   else true
8 }
```

Figure 9: Q20 の変更可能箇所

```
Found:    (y : Int)
Required: Boolean
```

Figure 10: 目的のエラーメッセージ

図 6.6: 類似した間違い方が答えになるエラー生成クイズ例

# 第7章　今後の課題と展望

## 7.1　提案の有効性測定

現在までのところ、提案したエラー生成クイズとエラーメッセージ読解モデルが実際にどの程度の効果を持っているかの測定は完全には行っていない。エラーメッセージ読解モデルとエラー生成クイズの有用性を測定する方法を考察する。

**エラー生成クイズについて**　エラー生成クイズの効果を測定する１つの方法として、デバッグ型演習と比較することが考えられる。現段階ではエラー生成方式とデバッグ型演習を対象に、エラーメッセージの理解度に与える影響を比較することを計画している。デバッグ型演習はエラー生成クイズと構成要素がほとんど同じである。そのため、エラー生成クイズとデバッグ型演習を比較した際に、エラー生成クイズ側に大きく異なる利点が確認されれば、エラー生成クイズという形式の特徴を示すことができる。

**エラーメッセージ読解モデルについて**　エラーメッセージ読解モデルの有用性測定として、モデルの各段階で手に入れることができる情報を供給した場合のエラー解決率、エラーメッセージの理解度の変化を図ることが考えられる。このような実験により、エラーメッセージ読解モデルの中で特に影響が大きい段階を把握できる。また単純な単語の意味一覧や、エラーメッセージの和訳を追加で渡した場合と、モデルに基づいた情報を渡した場合を比較する実験も考えられる。

有効性への懸念点の１つとして、予備実験から読解度の自己評価と客観的な実施率の間に大きな差が存在する可能性が挙げられる。この場合、今回のアンケートのように単純に実施率を尋ねるだけでは正確な読解率を計測できず、エラーメッセージ読解モデルの実施率の数値が有効でなくなる恐れがある。今回実施した読解度測定クイズをアンケートに追加して実施するなど、主観的な評価だけでなく客観的な実施率を計測できる工夫を考察していく。

また、今回仮定したエラーメッセージ読解モデルが、エラーメッセージの読解を適切に分解できているかという考察も続けていく必要がある。エラーメッセージ読解モデルが妥当なものであれば、初学者がエラーメッ

セージを読解できているかどうかを測定するために活用できる。これはエラー生成クイズとデバッグ型演習の比較実験にも活用できる可能性がある。

## 7.2　エラー生成クイズの各問題について

**問題の難易度**　観察の中で、エラー生成クイズの問題は難易度が高いという課題が確認された。そこでエラー生成クイズの難易度を適切に段階化する方法の考察を進める。例として、エラーメッセージ読解モデルの各段階で得られる情報を、順々に開示するヒントの追加が考えられる。

また、エラー生成クイズ全ての問題に解説をつけることも考えられる。実験のインタビューにて、解説を聞くことで解けなかった問題であってもエラーメッセージの理解ができたというコメントが確認された。そのため解説があれば初学者が問題を解けなくとも、エラー生成クイズが目的とする効果を発揮できる可能性がある。今後の研究では、エラー生成クイズの各問題に解説を準備することを考慮する。

**エラー生成クイズに適したエラーの調査**　現在のエラー生成クイズでは、対応しているエラーメッセージの種類を限定している。例えば構文エラーは基本的に問題から除外している。これは、構文エラー時のエラーメッセージは、誤りの原因と直接対応していないことが多いためである。どのような誤りがエラー生成クイズに適しているのかを調査する。

一方、初学者にとって構文エラーの重要性が大きい可能性が示唆されている [17]。そのためエラー生成クイズの形式で、これらのエラーを効果的に使用できる工夫も考察する。

## 7.3　エラー生成クイズの作成支援/自動化

現在エラー生成クイズは手作業で作成している。エラー生成クイズの研究を効率的に進めるために、問題作成に必要な手順のコストを削減したい。そのため、エラー生成クイズの作成支援、もしくは支援を行うソフトウェアの作成を考察している。

ソフトウェア作成のために、まずエラー生成クイズの作成手順をアルゴリズム化する。手順は、コードの収集、エラーメッセージの収集、問題の元コード作成、変更可能箇所の設計からなる。これに今回の実験結果から追加の必要性が見受けられた解説の作成を加えると、5 つの手順がエラー生成クイズの問題作成に必要である。

### 7.3.1　作成手順の詳細

**コードの収集**　エラー生成クイズを実施する講義、もしくはその講義の受講者が既に受講済みの講義内で使用する教科書内に登場する正しい (エラーのない) コードや講義の演習問題の模範解答のプログラムコードを集める。対象をこのようなコードにしているのは、受講者が実際に出会う可能性の高いエラーメッセージを集めるためである。つまり、講義で登場する文法や機能の範囲で発生するようなエラーメッセージを集めるために、これらを対象にしている。

なお演習の形式によっては、模範解答に受講者が手をつけない部分 (図 7.1 の点線部分のような、テストを行うための部分など) があることが考えられる。その場合、コードの収集時にそのような部分は取り除く。

ソースコード：

```
def connect(a:List[Int],b:List[Int]):List[Int] ={
a match {
   case Nil => b
   case x :: xs => x::connect(xs,b)
  }
}

val list1 = List(1,2,3)
val list2 = List(3,2,1)
val test1 = connect(list1,list2) == List(1,2,3,3,2,1)

test1
```

図 7.1: コード収集時

**エラーメッセージの収集**　収集したコードの一部を正しくなくなるように変更し、その変更によって発生するエラーメッセージを集める。この時に収集するエラーメッセージが、エラー生成クイズで用いる目的エラーメッセージとなる。

例として図 7.2 のコードを考える。この場合、4 行目の append の左にある x を a に書き換え、図 7.3 の状況にして発生するエラーメッセージや、2 行目の Nil を x に書き換え、図 7.4 の状況にして発生するエラーメッセージを収集する。

ソースコード：

```
def connect(a:List[Int],b:List[Int]):List[Int] ={
a match {
   case Nil => b
   case x :: xs => x::connect(xs,b)
 }
}
```

図 7.2: エラーメッセージ収集 1

ソースコード：

```
def connect(a:List[Int],b:List[Int]):List[Int] ={
a match {
   case Nil => b
   case x :: xs => a::connect(xs,b)
 }
}
```

エラーメッセージ：

```
Found:    (a : List[Int])
Required: Int
```

図 7.3: エラーメッセージ収集 2

ソースコード：

```
def connect(a:List[Int],b:List[Int]):List[Int] ={
a match {
   case x => b
   case x :: xs => x::connect(xs,b)
 }
}
```

エラーメッセージ：

```
Unreachable case
```

図 7.4: エラーメッセージ収集 3

**問題の元コード作成**　収集したエラーメッセージを 1 つ選択し、そのエラーメッセージを 1 箇所のみの変更で発生させられるような正しいコードを作成する。これがエラー生成クイズの、選択した目的エラーメッセージ

に対する元コードとなる。元コードは、エラーメッセージ収集の際に変形させた正しいコードの原型が基本となる。もしその正しいコードに選択したエラーメッセージと関係のない部分があれば、それを消してより単純な元コードを作成する。

例として図 7.3 で発生していたエラーメッセージを目的エラーメッセージとする。この場合、元コードとしては図 7.2 のコードが基本となるため、図 7.5 のような形になる。ただしこの場合 `match case` の部分は目的エラーメッセージと関わらないため、元コードをより単純にして、問題を図 7.6 の形にすることができる。この時想定解は 2 行目の `x` を `a` に書き換えた図 7.7 となる。

元コード例(複雑)：

```
def connect(a:List[Int],b:List[Int]):List[Int] ={
a match {
  case Nil => b
  case x :: xs => x::connect(xs,b)
 }
}
```

目的エラーメッセージ：

```
Found:    (a : List[Int])
Required: Int
```

図 7.5: 元コード作成 1

元コード：

```
def cons(x:Int,a:List[Int]) : List[Int] = {
  x :: a
}
```

目的エラーメッセージ：

```
Found:    (a : List[Int])
Required: Int
```

図 7.6: 元コード作成 2

想定解：

```
def cons(x:Int,a:List[Int]) : List[Int] = {
  a :: a
}
```

図 7.7: 元コード作成 3

**変更可能箇所の設計**　作成した元コードの中から、問題として使用する際に指定する変更可能箇所をいくつか選択する。選択する際は以下の点に注意する。

I 想定解で変更する場所を変更可能箇所とする

II 想定解と異なる変更場所で、目的エラーメッセージを発生させることができる場所は変更可能箇所には選択しない

III 定義時に使用する関数名や、関数定義時の定型部分 (括弧など) は変更可能箇所には選択しない

IV 目的エラーメッセージとは異なるが、系統が同じ (型エラー同士、match case に関連する警告同士など) エラーが発生する場所は優先して変更可能箇所に選択する

例として図 7.8 の変更可能箇所の選択を考える。想定解は図 7.9 である。この時注意点は図 7.10 のように反映される。I の部分は図 7.9 にあるように想定解の変更場所である。II の部分は `Int` を `Boolean` に書き換えることで、目的エラーメッセージと同じエラーメッセージが発生する。III の部分は定義時の関数名や定型部分の括弧である (見やすさのため一部にのみ × をつけている)。IV の部分は型に関連し、変更することで型エラーが発生しやすい部分になっている。

元コード：

```
def int_and(x:Int,y:Int) : Boolean ={
  int_to_bool(x) && int_to_bool(y)
}

def int_to_bool(x:Int) : Boolean = {
  if (x == 0) false
  else true
}
```

目的エラーメッセージ：

```
Found:    (y : Int)
Required: Boolean
```

図 7.8: 変更可能箇所の設計 1

想定解：

```
def int_and(x:Int,y:Int) : Boolean ={
  int_to_bool(x) && y
}
                          Found:    (y : Int)
                          Required: Boolean
def int_to_bool(x:Int) : Boolean = {
  if (x == 0) false
  else true
}
```

図 7.9: 変更可能箇所の設計 2

```
def int_and(x:Int,y:Int) : Boolean ={
  int_to_bool(x) && int_to_bool(y)
}                                          I
        III
def int_to_bool(x:Int) : Boolean = {
  if (x == 0) false      II          IV
  else true
}
```

図 7.10: 変更可能箇所の設計 3

**解説の作成**　その想定解でなぜ目的エラーメッセージが発生するのかの解説を作成する。

例として、図 7.5 の問題の解説を考える。この場合「このコードでは::の左側で Int が要求されている。これは、::が List に要素を連結する演算子であるため、::の左側の要素の型が、右側の List の中身と同じでない

と連結できないためである。」というような解説を作成する。

### 7.3.2　自動化案

作成手順の中で、特に「コードの収集」と「解説の作成」以外の手順の自動化、もしくは支援を考察しており、現時点の見通しは以下の通りである。

**エラーメッセージの収集**　正しいコードを受け取りその中で数箇所を指定すると、それぞれの指定箇所を書き換えたときに発生しうるエラーメッセージを自動で収集するソフトウェアを考えている。これはコンピュータファジングの技術やミュータント生成の応用 [22] を用いることでの実装を見込んでいる。

**問題の元コード作成**　目的エラーメッセージと無関係の部分をコードから消去し、元コードを単純化する作業の自動化を考えている。これにはコンパイラファジングの分野で用いられる単純化の技術 [23] が利用できると見込んでいる。

**変更可能箇所の設計**　作成した元コードの中で、想定解にしたい部分以外に目的エラーメッセージが発生する変更ができる場所がないかを自動的に調べるツールを考えている。これはエラーメッセージの収集で使用できるソフトウェアを実装できれば、その機能で同様に実現できると考えている。

# 第8章　関連研究

初学者とエラーメッセージの間の問題点を解決しようとしている研究は近年様々なものが行われている。初学者がエラー解決のためにエラーメッセージを利用する際の代表的な問題点は、エラーメッセージ自体のわかりづらさと、初学者がエラーメッセージと遭遇した時の知識不足の2点である。そのため行われている研究は主に、エラーメッセージ自体をわかりやすくしようという研究と、知識不足の初学者がエラーを修正できるように直接的な支援をする研究に分けられる。

**エラーメッセージの内容を改善する研究**　視覚的な改善として、Marceau ら [14] や Funabiki [9] らは IDE 内でエラー発生場所をハイライトで表現することを提案・実装している。Barik [2] や Wrenn ら [21] はエラーメッセージ内の単語と対応するコード断片を同色でハイライトすることや矢印で繋ぐことによる視覚的改善を提案している。またエラーメッセージの有用性を向上させるために、Becker [5] や Prather ら [20] はエラーメッセージ内でより詳細なエラー原因の情報を提供するツールを実装した。Barik ら [4] はエラーの原因を理解できるように最もシンプルなエラー解決方法の提案をエラーメッセージとともに行うことを提唱している。そのほかに、わかりやすいエラーメッセージのためのガイドラインの提案 [8, 3, 15] も行われている。

これらの研究によって初学者が簡単にエラーメッセージを読解できるようになる可能性がある。しかし一方で、視覚情報を追加することで学生が何も考えずに誤った変更を行う可能性 [15] や、エラーメッセージ内の情報を詳細にしたが良い効果が確認できなかったという結果 [19, 7, 18] もあり、この方針がどこまでの効果をもたらせるかはいまだに不明瞭である。またエラーメッセージ自体の改善は処理系自体を変更する必要があるため、変更の影響やコストの大きさも問題点として考えられる。

エラー生成クイズでは、エラーメッセージ自体の改善を行うわけではないためこれらの問題点の影響は受けない。その一方でエラー生成クイズは、エラーメッセージ改善の研究の効果を増幅できる可能性がある。これは仮にエラーメッセージ単体の改善では初学者の理解度を向上させることが難しくとも、エラーメッセージの改善がエラー生成クイズを取り組みや

すくする可能性が存在するためである。取り組みやすくなったエラー生成クイズを通してエラーメッセージの読解を促進できれば、初学者がエラーメッセージと遭遇する際の問題を解決できる。このようにエラー生成クイズにはエラーメッセージ改善の研究との相互作用が期待できる。

**初学者を直接支援する研究**　エラー修正提案の発展型として、Hartman ら [12] や Ahmed ら [1] は発生したエラーと類似している例を以前書かれたコードから発見し、そのエラーを修正するときに行われた変更を提示するという手法を実装している。また、Lee ら [13] はエラーのあるコードを修正することを繰り返すデバッグ型演習の提案、実装を行い、蜂巣ら [24] はデバッグ型演習の自動化システムの提案や実装を行なっている。

これらはエラー生成クイズと同様に初学者に知識そのもの、あるいは知識を身につける機会を提供している。しかし修正方法の提案という手法の場合、誤った提案をするというリスクが常に存在し、コンピューター側が誤った提案を出す可能性は問題視 [15] されることもある。また修正方法を提案するツールがあると、そのツールに依存しすぎてしまうという懸念もある。

またデバッグ型演習では、エラーメッセージを読解せずに問題を解くという懸念点がある。これは、エラーメッセージが複雑な際に特に懸念される。例として、図 8.1、図 8.2 の状況が挙げられる。このコードを問題にした場合、初学者はエラーメッセージをしっかり読まずにエラー発生箇所のみ見てもエラーを解決することが可能である。つまり、エラー発生箇所のコードを見ると＞の左右の要素が対称でなく、そこからなんとなく対称になるように変更すれば良いのではないかと考えることができてしまう。この解き方によって、図のどちらの問題も解くことができる。しかしそのような解き方をした場合、初学者がそれぞれの問題でエラーメッセージが異なることに注目する可能性は低く、エラーメッセージの理解が促進されないことが考えられる。するとエラーメッセージから判断が難しくなるため、以降に同様のエラーが発生しても、対称性を用いて解決できない状況 (図 8.3) の場合はそのエラー解決に苦労する可能性がある。

エラー生成クイズでは、解決策の提案をする必要がない。また、初学者にエラーメッセージ読解力をつけてもらうための形式であるため、エラー生成クイズがない環境に移行しても知識を活用できる。その点ではデバッグ演習も同様であるが、問題を通してエラーメッセージ読解の要素を必ず行うことはエラー生成クイズに特有で、この特徴が有用な可能性がある。


図 8.1: 間違い 1

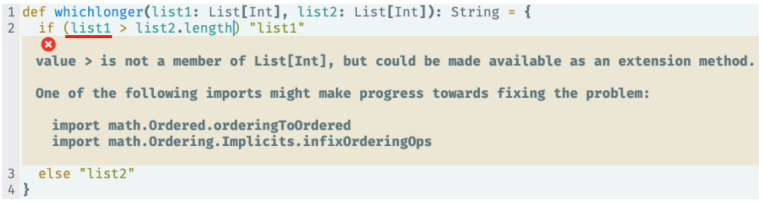



図 8.2: 間違い 2

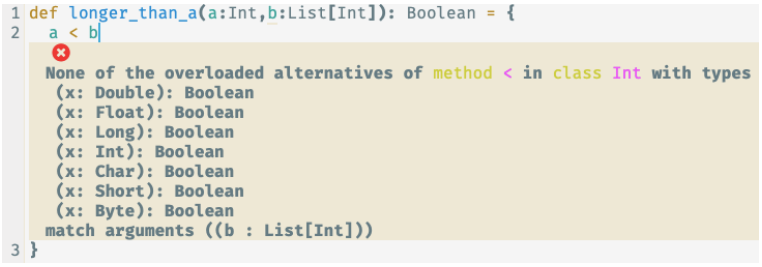



図 8.3: 対称性のない間違い

# 第9章　まとめと将来の展望

我々は本論文で、エラー生成クイズというプログラミング教育で使用できる問題形式を提案した。エラー生成クイズの目的は、初学者へのエラーメッセージの理解の支援である。本論文ではこの目的のために、エラーメッセージの読解モデルを提案し、またエラー生成クイズの試行として観察実験を行なった。実験人数が3名と少ない中ではあるが、実験を通してエラー生成クイズには従来の問題形式やエラーメッセージ改善手法にはない特徴と効果がある可能性を確認した。

現在の我々の目標は、実際の講義等で使用できるエラー生成クイズのドリルを作成することである。その目標に向け、エラー生成クイズのそれぞれの課題点に対応していく。特に作成コストの低減は、解決が比較的容易と思われるため対応を進めて行く。

初学者がエラーメッセージを利用する際の問題を解決するために様々な手法の研究が行われているが、いまだにこの問題は解決していない。提案したエラー生成クイズがこの問題を解決する新たな一つの道筋となることを期待している。

# 参考文献

[1] Ahmed, U. Z., Sindhgatta, R., Srivastava, N. and Karkare, A.: Targeted example generation for compilation errors, *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, pp. 327–338 (2019).

[2] Barik, T.: Improving error notification comprehension through visual overlays in IDEs, *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, pp. 177–178 (2014).

[3] Barik, T., Ford, D., Murphy-Hill, E. and Parnin, C.: How should compilers explain problems to developers?, *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 633–643 (2018).

[4] 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*, pp. 536–539 (2014).

[5] Becker, B. A.: An effective approach to enhancing compiler error messages, *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp. 126–131 (2016).

[6] Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P.-M. et al.: 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*, pp. 177–210 (2019).

[7] Denny, P., Luxton-Reilly, A. and Carpenter, D.: Enhancing syntax error messages appears ineffectual, *Proceedings of the 2014 confer-*

*ence on Innovation & technology in computer science education*, pp. 273–278 (2014).

[8] Denny, P., Prather, J., Becker, B. A., Mooney, C., Homer, J., Albrecht, Z. C. and Powell, G. B.: On Designing Programming Error Messages for Novices: Readability and its Constituent Factors, *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–15 (2021).

[9] Funabiki, N., Matsushima, Y., Nakanishi, T., Watanabe, K. and Amano, N.: A Java programming learning assistant system using test-driven development method, *IAENG International Journal of Computer Science*, Vol. 40, No. 1, pp. 38–46 (2013).

[10] Griffin, J. M.: Designing intentional bugs for learning, *Proceedings of the 1st UK & Ireland Computing Education Research Conference*, pp. 1–7 (2019).

[11] Guo, P. J.: Non-native english speakers learning computer programming: Barriers, desires, and design opportunities, *Proceedings of the 2018 CHI conference on human factors in computing systems*, pp. 1–14 (2018).

[12] Hartmann, B., MacDougall, D., Brandt, J. and Klemmer, S. R.: What would other programmers do: suggesting solutions to error messages, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1019–1028 (2010).

[13] Lee, G. C. and Wu, J. C.: Debug It: A debugging practicing system, *Computers & Education*, Vol. 32, No. 2, pp. 165–179 (1999).

[14] Marceau, G., Fisler, K. and Krishnamurthi, S.: Measuring the effectiveness of error messages designed for novice programmers, *Proceedings of the 42nd ACM technical symposium on Computer science education*, pp. 499–504 (2011).

[15] Marceau, G., Fisler, K. and Krishnamurthi, S.: Mind your language: on novices' interactions with error messages, *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 3–18 (2011).

[16] Mccall, D.: *Novice Programmer Errors-Analysis and Diagnostics*, PhD Thesis, University of Kent, (2016).

[17] McCall, D. and Kölling, M.: A new look at novice programmer errors, *ACM Transactions on Computing Education (TOCE)*, Vol. 19, No. 4, pp. 1–30 (2019).

[18] Nienaltowski, M.-H., Pedroni, M. and Meyer, B.: Compiler error messages: What can help novices?, *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pp. 168–172 (2008).

[19] Pettit, R. S., Homer, J. and Gee, R.: Do Enhanced Compiler Error Messages Help Students? Results Inconclusive., *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 465–470 (2017).

[20] Prather, J., Pettit, R., McMurry, K. H., Peters, A., Homer, J., Simone, N. and Cohen, M.: On novices' interaction with compiler error messages: A human factors approach, *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pp. 74–82 (2017).

[21] Wrenn, J. and Krishnamurthi, S.: Error messages are classifiers: a process to design and evaluate error messages, *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 134–147 (2017).

[22] 亀井亮汰, 吉塚大浩, 篠埜功, 古宮誠一: 写経型学習の欠点を補う摂動を用いた理解度確認問題生成手法—二項演算子の事例に基づく有効性評価, コンピュータ ソフトウェア, Vol. 38, No. 1, pp. 1_111–1_139 (2021).

[23] 菜岐佐石浦: コンパイラのファジング, 電子情報通信学会 基礎・境界ソサイエティFundamentals Review, Vol. 9, No. 3, pp. 188–196 (2016).

[24] 蜂巣吉成, 吉田敦, 阿草清滋ほか: プログラムの誤り修正課題および正誤判定プログラムの自動生成, 情報処理学会論文誌教育とコンピュータ (TCE), Vol. 3, No. 1, pp. 64–78 (2017).