GHCコンパイラのエラーメッセージの簡易化について

島根大学大学院
総合理工学研究科　数理・情報システム学専攻
発表者　田中陽子
岩見研究室

平成25年2月20日

目　次

- 背景
- 目的
- 関連研究
- システム概要
- エラーメッセージの収集とパターン化
- 解析器と照合器の実装
- 実行結果
- 考察とまとめ
- 結論と今後の課題
- 参考文献

背　景　①

- プログラミング言語の種類

  ―命令型言語　（C, Java, Perl, Ruby etc…）

  ・・・計算をプログラムの状態を変化させる文で記述することでプログラミングを行う

  ―宣言型言語 (Haskell, ML, OCaml, Scala etc…)

  ・・・処理方法ではなく対象の性質などを宣言することでプログラミングを行う

- プログラミングの流れ

  プログラムの記述　→　コンパイル　→　実行ファイルの生成

  エラー発生（エラーメッセージが表示）

  エラーの原因を明確に示していることが望ましい

しかし．．．　Haskellのエラーメッセージはわかりにくい

背　景　②

・文字列表示プログラムについて

言語	プログラム	エラーメッセージ
Java	「‘)’がありません」	HelloWorld.java:3: ')' がありません. System.out.print("HelloWorld";
Haskell	よく分らない・・・	error.hs:1:28: parse error (possibly incorrect indentation)

・Haskell

・・・関数を中心にプログラムを組み立てることを目的とした関数型言語.
型推論や遅延評価といった特徴をもつ.

・Haskellの代表的なコンパイラ：Glasgow Haskell Compiler (以降GHC)

背　景　③

- Haskellのコンパイル時のエラーメッセージはわかりにくい

簡単なデバッグ方法があるのでは？

- C言語やJava のように文単位の編集を用いたデバッグができない

- **Haskell初学者がHaskellを理解しにくいと考える原因の一つ**

草書のような
Haskellのコード

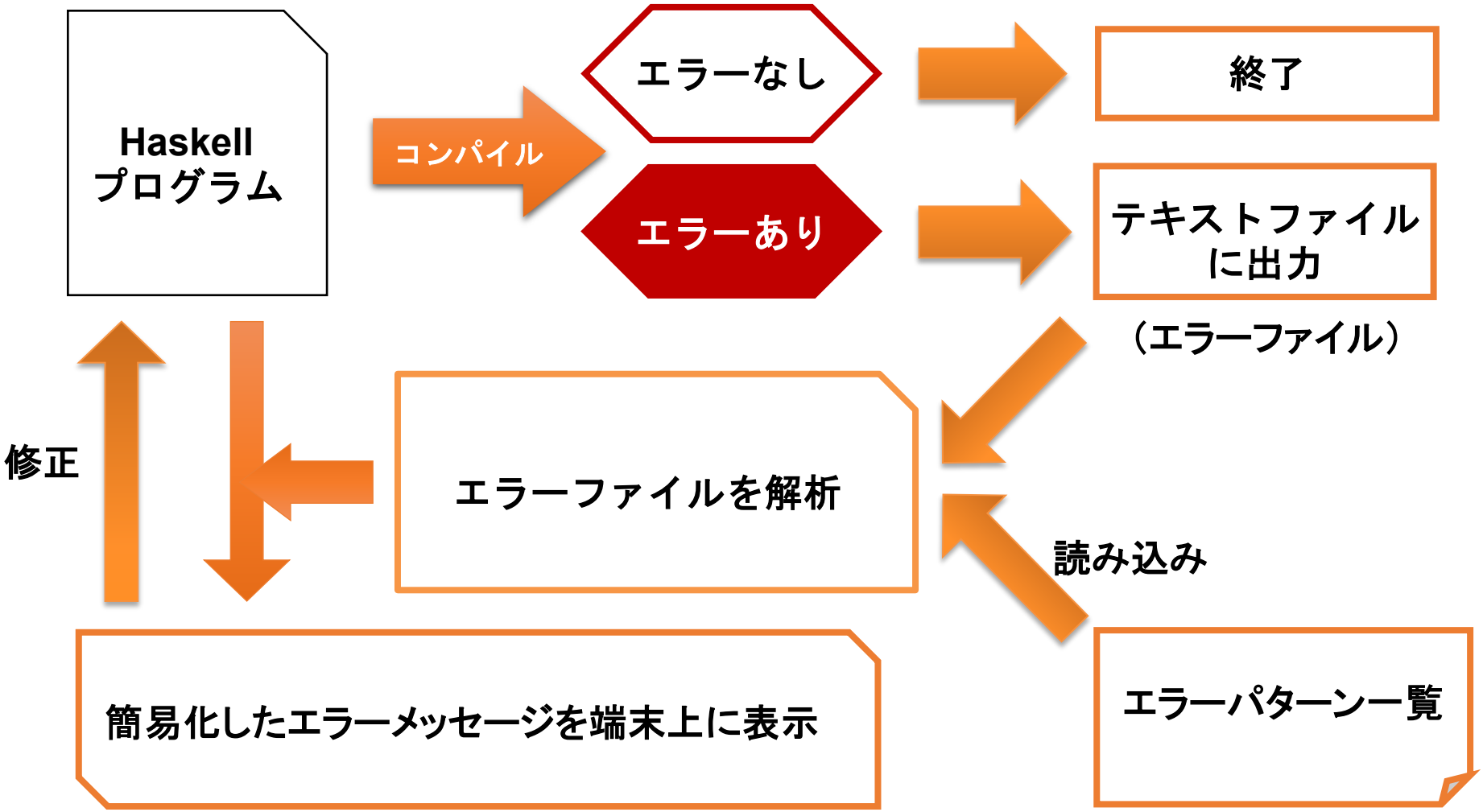
楷書のような
手続型言語のコード

目　的

HaskellのGHCコンパイルエラーを解析して，メッセージを簡易化する

初学者がより簡単にデバッグ作業を行えるようにする

簡易化システム全体の流れ



エラーメッセージの収集①

・単一エラーの場合のメッセージを収集
→ 関数のスペルミス，引数の欠如，メインメッセージの欠如等

“Hello World”を出力

```
mian  =  putStrLn "Hello World"
```

考えられるエラーをできる限り排出

① mainを削除したエラー
② mainのスペルミスのエラー
③ =を削除したエラー
…
…

初学者が間違えやすいエラーを中心に収集

エラーメッセージの収集は5個のプログラムで144個収集
（この状態で解析器を実装して，実装後エラーメッセージを追加）

エラーメッセージのパターン化①

・エラーメッセージと原因を関連づけてエラーのパターン化を行う

[エラーメッセージ例]

エラーメッセージ	エラー原因
The function `take' is applied to three arguments, but its type `Int -> [a0] ->[a0]' has only two In the second argument of `($)', namely `take n lines cs' In the expression: unlines $ take n lines cs	プログラム内のtake関数の引数の数が違う
The function `lines' is applied to two arguments, but its type `String -> [String]' has only one In the second argument of `($)', namely `lines cs cs‘ In the second argument of `($)', namely `length $ lines cs cs' In the expression: print $ length $ lines cs cs	プログラム内のlength関数の引数の数が違う

2つは関数が求める引数の数が違うというエラー原因は同じ

エラーメッセージのパターン化②

・2つのエラーメッセージを解析してパターンを抽出

[エラーメッセージのパターン化例]

パターン化したエラーメッセージ	エラー原因
The function `関数名1‘ is applied to 数 argumentsbut, but its type `型1’ has only 数 In the 番手 argument of `関数名2', namely `関数または引数名1‘ In the expression: 関数名3	関数名１の引数の数が違う

収集した単一エラーメッセージ一覧に対して，上記のようなパターン化を行い，エラーパターン一覧を作成

エラーメッセージの収集②

・複数のエラーが起こる場合のメッセージを収集

・複数が同時に起こる組み合わせ（$\sum_{r=2}^{n} nCr$）

ファイル名	単一エラー数	エラーの組み合わせ個数
error1_1.hs	9	502
error1_2.hs	14	16369
error1_3.hs	20	1048555
error1_4.hs	39	549755813848
error1_5.hs	62	4611686018427387841
合計	**144**	**4611686568184267115**

➡ ランダムでエラーを組み合わせてエラーメッセージを排出

複数エラーがある場合，メッセージの表示方法は3通り存在することを確認

エラーメッセージの収集③

・複数エラーがある場合のメッセージの表示方法

① **エラーメッセージが複数表示される**

→ エラーの数に対応した，それぞれのエラーメッセージが表示される

② **排出の優先順位が高い方のエラーメッセージが表示される**

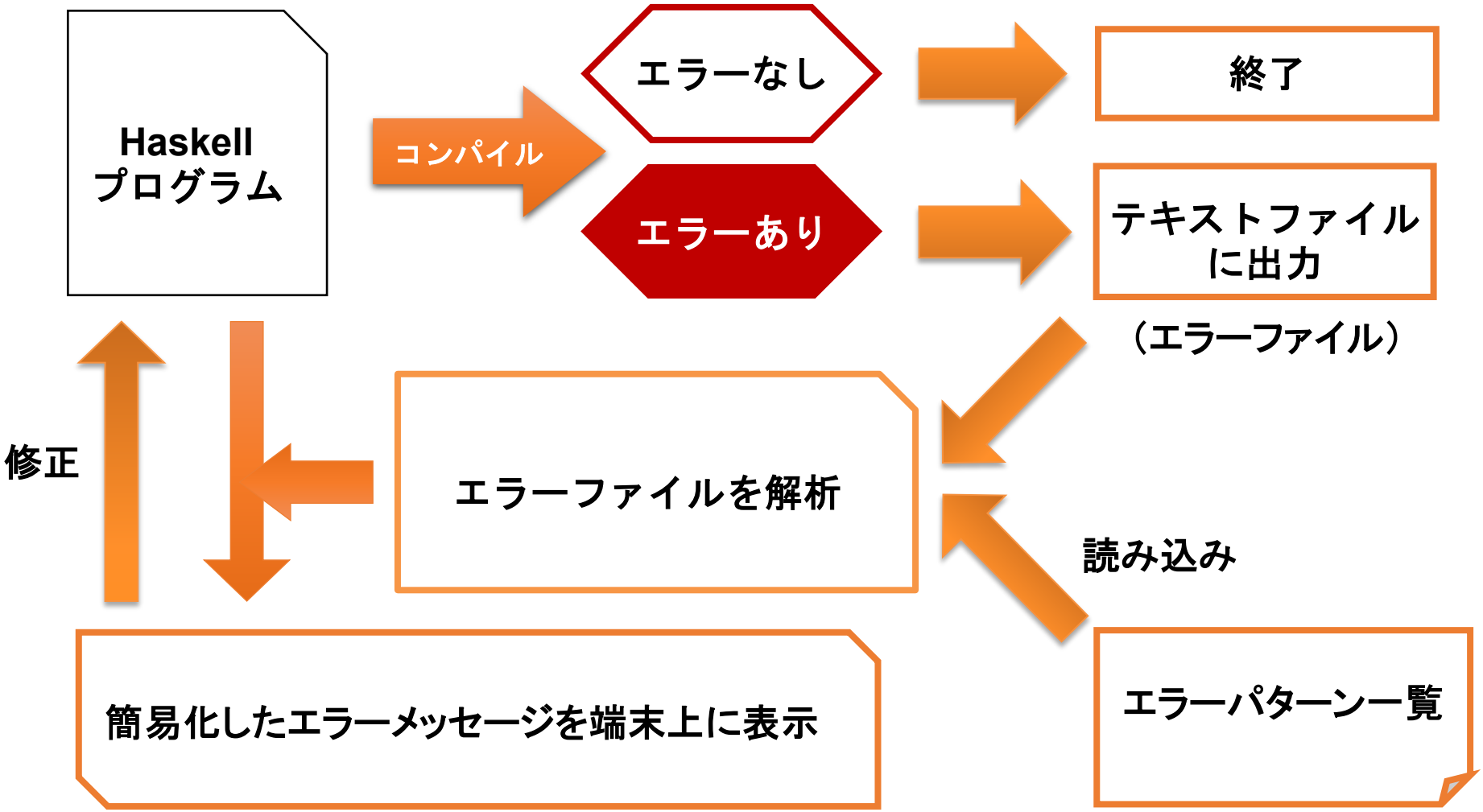
→ (例)プログラムにエラー1，エラー2が存在する場合，エラー1のメッセージのみ表示される

③ **全く新しいエラーメッセージが表示される**

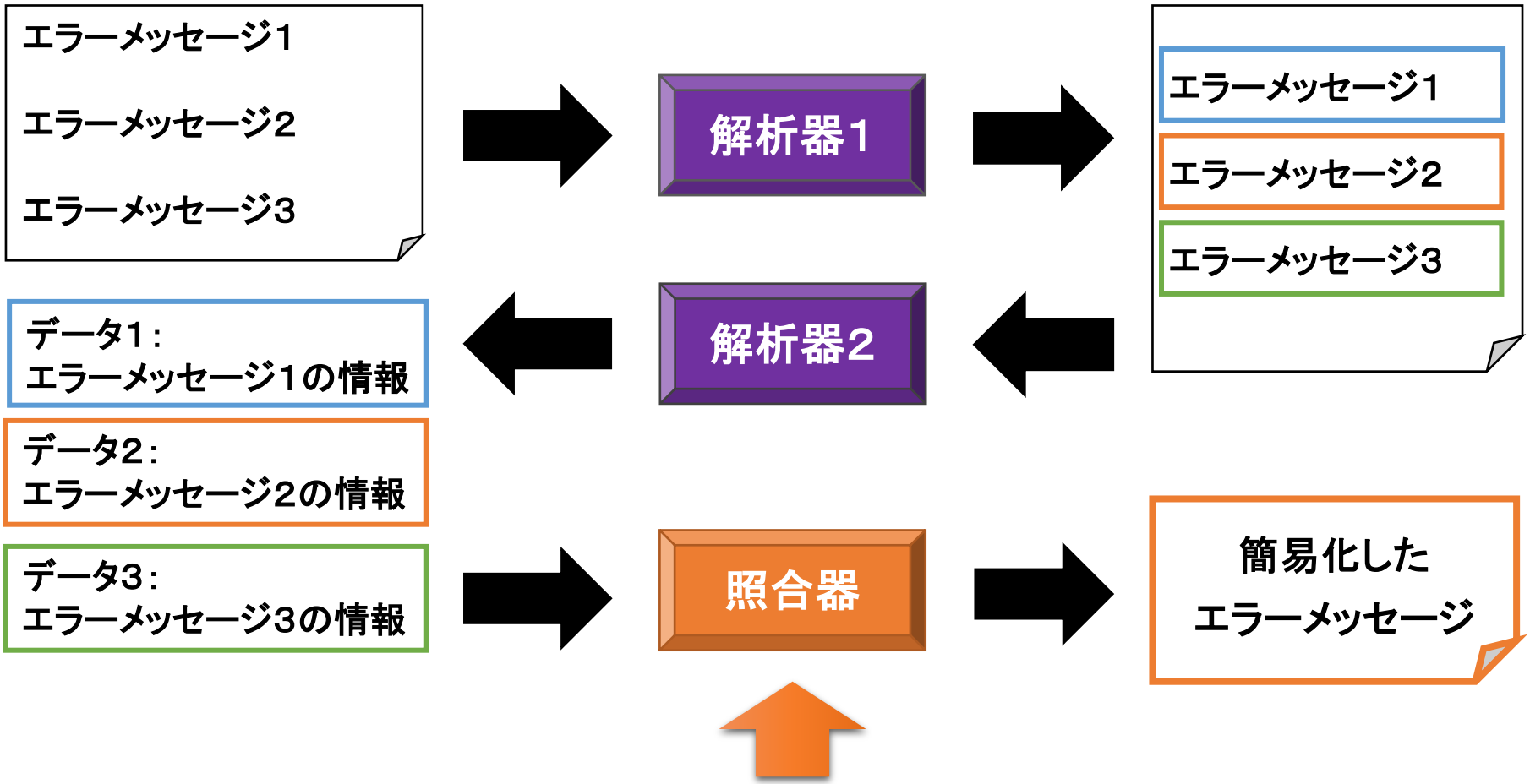
→ parse error (possibly incorrect indentation) のようなメッセージが表示される

①，②の場合は，単一エラーの場合と解析方法は同じ．
③の場合は，ソースコードと対応させながらエラー原因を特定．

簡易化システム全体の流れ



エラーファイル解析の流れ



解析器1

・エラーファイルをエラーメッセージごとに分割する

[エラーファイルの中身(2つのエラーメッセージがある場合)]

[“error.hs:1:30: lexical error in string/character literal at character ”,
“error.hs:14:1: Couldn't match expected type `IO t0'”,
“with actual type `[Char]'”,
”In the expression: main”,
”When checking the type of the function`main'”]

行ごとに分割された区切りを，エラーごとの区切りに再分割する

[**“error.hs:1:30: lexical error in string/character literal at character ”**,
“error.hs:14:1: Couldn‘t match expected type `IO t0’ with actual type `[Char]’
In the expression: main When checking the type of the function`main'”]

解析器2

・解析器1から送られてきたエラーメッセージをデータ型に保存する

[送られてきたエラーメッセージ例]

“error.hs:1:30: lexical error in string/character literal at character ”

[ERデータ型に保存]

```
ER { filename = “error.hs”,
     row = 1,
     column =30,
     error_sent = analysis $ bound (mydropWhile
                  (unwords“lexical error in string/character literal at character”)),
     error_num  =1}
```

ファイル名

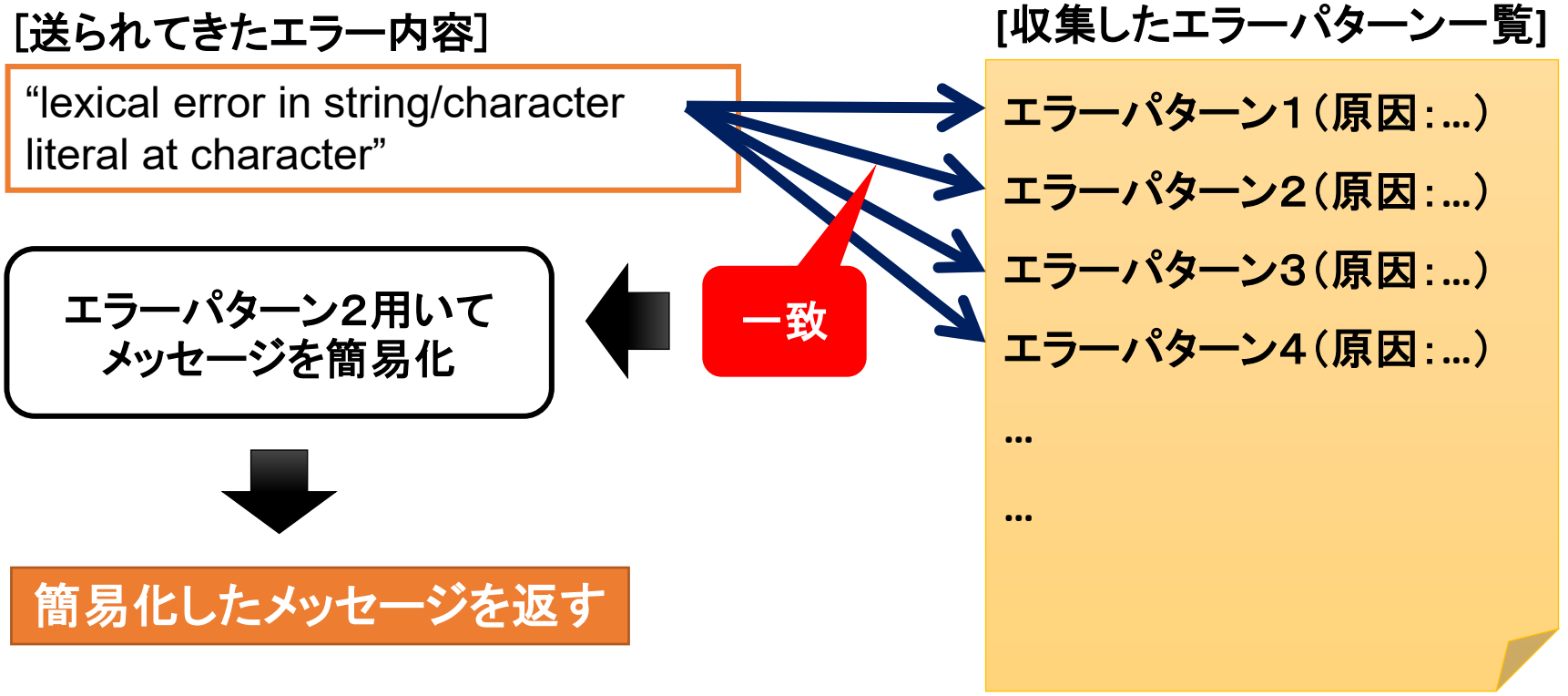
列数

エラー番号

・error_sentには簡易化したエラーの内容が束縛される

照合器

・解析器2から送られてきたエラー内容と収集しておいたエラーパターンを照合



合致しなかった場合は，特定できなかったことを表示して終了

実行結果

・簡易化の実行結果例

```
main    putStrLn "Hello World!!"
```

今回の簡易化

```
error.hs:1:7: Parse error: naked expression at top level
```

```
The error is detected.

[error1]
-------------------------------------------------------------------------
filename : error.hs
error point : (1,7) <- (row, column)

error contetnts :
There is not `='
Please check whether `=' exist ?
```

考　察

- 様々なプログラムにおいて，エラー数が1,2,3の場合でランダムにエラーを発生.
- それぞれ50回ずつ実装したシステムを用いて簡易化を試みた.

	エラー数1	エラー数2	エラー数3
簡易化メッセージの出力	47	36	22
エラー発生場所の特定	44	-	-
正確なエラー原因の出力	46	28	16
プログラムの修正	46	21	12
プログラムの修正確率	**92%**	**42%**	**24%**

- エラー数1の場合は，高い確率で修正に成功
- エラー数2，3の場合は修正確率が50%以下

まとめ

1. **GHCのコンパイルエラーメッセージの収集**

   → 現在は23個のプログラムにおいて726個のエラーメッセージを収集済み

2. **収集したエラーメッセージをパターン化**

   → 現在は34個のエラーパターンを生成済み

3. **エラーメッセージの解析**

   → エラーファイルをメッセージごとに分割してデータ型に保存

4. **エラー内容と収集したエラーパターンを照合**

   → 簡易化した結果をユーザに表示

結　論

・**単一エラー**に関しては，高い確率でメッセージを簡易化して表示し，プログラムの修正を行うことができた．

・**複数エラー**に関しては，エラー情報が完全ではなく，簡易化と修正を行うことがまだ難しい．

今後の課題

- 複数エラーの場合の解析の汎用性を高める
- Haskell初学者を対象としたユーザテストを行い，システムの有効性を検証
- システムの実行方法を改善する
  - ghcコンパイラ自体をシステム内部に組み込みテキスト形式で保存する手間を省略するなど
- 統合開発環境へプラグイン形式での提供方法の考察

関連研究

・コンパイラの型推論を利用した型デバッカの提案（2012,13）

（対馬，浅井）

[研究内容]

OCamlのコンパイラ内部の型推論器を用いて型デバッカを実装.

関連研究と本研究について

関連研究	型デバッカを作成し，OCamlのコンパイル時の型エラーの排出を改善
本研究	**コンパイラから排出されるエラーメッセージを解析して全体を簡易化**

エラーの改善という点では共通しているが，手法は異なる

参考文献

- 青木峰郎,”ふつうのHaskellプログラミング”,ソフトバンククリエイティブ,2006
- G.Hutton,”プログラミングHaskell”,オーム社,2009
- B.O’Sullivanら,”Real World Haskell”,オーム社、2009
- M.Lipovaca,”すごいHaskellたのしく学ぼう!”,オーム社,2012
- 対馬，浅井,”コンパイラの型推論を利用した型デバッカの提案”,PPL,2012
- 対馬，浅井,”コンパイラの型推論を使用した型デバッカの提案”，コンピュータソフトウェア,2013