情報処理学会論文誌　プログラミング　Vol.10　No.1　1–13 (Jan. 2017)

拡張可能な構文解析器生成系による構文エラー処理機能の実装

細田 将大[1,a)]　中井 央[2,b)]　佐藤 聡[3,c)]　新城 靖[3,d)]

受付日 2016年5月10日, 採録日 2016年8月11日

概要：構文エラー処理はプログラム開発に不可欠であり，より優れた構文エラー処理アルゴリズムを求めて多くの研究がなされている．そのような研究に際して新たに考案された手法の評価には，その構文解析器の実装が必要となるが，実用的な規模の言語を想定した評価をするためには，構文解析器生成系の出力したコードもしくは構文解析器生成系の改造などをする必要があり，その実装コストは大きい．本論文では，我々の研究室で開発した拡張可能な構文解析器生成系を用いることで，新たな構文エラー処理手法を提案しようとする研究者が，比較的容易にそれを実装できることを示す．また，このことは，プログラミング言語処理系の開発者が，開発しようとする言語処理系にあった構文エラー処理手法を比較的容易に選択し，導入することができることにもつながる．本研究では，既存のいくつかの構文エラー処理手法を取り上げ，その特徴を分析し，共通して使用可能となる要素をライブラリとして実装を行った．そして，それぞれの構文エラー処理手法を我々の生成系の拡張機能として実装した．各エラー処理は，言語とは独立であるため，生成系に Java の文法を与えて構文解析器を生成し，そのエラー手法が提案された時点の論文における検証内容を再現することができた．機能拡張の実装は，200 行未満で行うことができた．これらから，本研究の手法により，我々の生成系が構文エラー処理手法の研究者に貢献できることを示せた．

キーワード：構文解析器生成系，構文エラー処理

Implementations of Syntax Error Handling with Extensible Parser Generator

Masahiro Hosoda[1,a)]　Hisashi Nakai[2,b)]　Akira Sato[3,c)]　Yasushi Shinjo[3,d)]

Received: May 10, 2016, Accepted: August 11, 2016

Abstract: Since syntax error handling is essential to program development, there has been much research for finding better error handling algorithms. An evaluation of a new algorithm requires implementation of a parser, but this can be very costly since evaluation of practical scale languages also requires altering a parser generator or outputting codes. This paper shows that, when a researcher wants to propose a new error handling method, extensible parser generator developed in our lab enables the researcher to implement it easily. Our parser generator also enables developers of language processors to skillfully choose and introduce more suitable error handling methods. In this research, we analyzed some already-existing syntax error handling methods, extracted common features from them, and then implemented these as libraries. We also implemented individual error handling methods as extensible functions on our parser generator. Since each of the error handling methods we treated was language-independent, we made a parser by giving Java grammar to the generator. As a result, we reproduced the same verification details of those methods as first proposed. The extensible functions could be implemented in fewer than 200 lines. Consequently, using our generator and methods will significantly contribute to the research of the syntax error handling method.

Keywords: parser generator, syntax error handling

[1] 筑波大学大学院システム情報工学研究科
Graduate School of System and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[2] 筑波大学図書館情報メディア系
Faculty of Library, Information and Media Science, University of Tsukuba, Tsukuba, Ibaraki 305–8550, Japan
[3] 筑波大学大学院システム情報系
Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
a) s1620674@u.tsukuba.ac.jp
b) nakai@slis.tsukuba.ac.jp
c) akira@cc.tsukuba.ac.jp
d) yas@cs.tsukuba.ac.jp

1. はじめに

開発に欠かせないコンパイラの機能として，エラー処理があげられる．構文エラー処理は様々なアプローチで研究が行われている．構文エラー処理は多岐にわたり，たとえばエラー検出後にも構文解析を継続するためにはエラーから何らかの形で回復する必要がある．構文エラーからの回復として，1 つの方法は，入力列中でエラーの原因となった終端記号の前に何らかの記号を挿入するか，誤りのある記号群を削除することで行われる．ここで誤った入力列から，プログラマが意図していると考えられる，構文上の誤りのない入力列をどのように導くかが問題となる．Kim ら [1] はトークンの挿入と削除にコストを定義したうえで，最も適切と考えられる最小コストの回復を探す問題をグラフ最短路問題に置き換えて導出する手法を提案している．彼らは提案手法の評価を行うため，既存の構文解析器生成系の 1 つである Bison [2] ベースの実装を行っている．その実装は，生成された構文解析器に対して提案手法のエラー処理を行うよう書き換えたか，そのエラー処理機能を持つ構文解析器が生成されるよう Bison 自体を書き換えたかのどちらかにより行ったと考えられる．しかし，Bison はそのようなエラー処理の置き換えに対応したインタフェースを持たないため，この実装は非常に煩雑であると考えられる．新しい構文エラー処理を実装する方法としては，他に構文解析器を手書きで作るといった方法も考えられるが，構文解析器自体の実装をともなうため，これには大きな労力がかかる．

この問題を解決するため，本研究では我々が開発した拡張可能な構文解析器生成系 [3] [*1]（以下，これを Depager と記す）に着目する．Depager の出力は純粋な構文解析器の機能をベースとしており，これに対し，テンプレートメソッドパターン，およびデコレータパターンを用いて複数の拡張機能を非破壊的に付加することが可能となっている．デフォルトの構文エラー処理としてはエラーメッセージの出力のみを行うが，拡張機能として構文エラー処理を実装することも想定されており，それにより新しい構文エラー処理機能の実装は非常に容易になると考えられる．

本研究では，これまでに様々な研究者により提案されてきた構文エラー処理アルゴリズムのいくつかを取り上げ，Depager の拡張機能として実装を行う．これにより，Depager によるエラー処理機能の実装が，エラー処理の研究を行う者にとって非常に有用であることを示す．

2. Depager

Depager は拡張可能な構文解析器生成系であり，原則として LALR(1) の構文解析器を生成する．

[*1] https://rubygems.org/gems/depager/

Depager はその拡張をオブジェクト指向に基づいて行うよう設計されており，その実現の方式として，純粋な構文解析器に対し拡張を付加していく形が採用されている．例として Yacc でいうアクションのようなものも拡張とすることができ，このためには Depager へ与える記述のフォーマットが拡張のために変更される必要があり，Depager ではこれにも対応している．

なお，Depager や生成される構文解析器の実装言語は Ruby である．

2.1 処理を追加する拡張

純粋な構文解析器に対して処理を追加する拡張の形式は，デコレータパターンとテンプレートメソッドパターンに基づいている．そのため，処理の追加はその目的ごとに 1 つのクラスを作り，そのクラスの中に必要なメソッドを定義する形をとる．

Depager により生成される構文解析器の構成を**図 1** に示す．追加する処理から得られる機能を**拡張機能**と呼ぶことにするが，Depager では任意の数の拡張機能を付与することが可能である．それらの処理の追加を行うクラスと Basis クラスを継承する純粋な構文解析器のクラスは層をなすよう設計されており，それぞれ 1 つ内側のクラスに対して変数`@inside` による参照を持つ．

拡張機能を作成する際には，AdvancedParser クラスで中身が空の状態で定義されている以下のメソッドから必要に応じてオーバライドすることで，LR 法による構文解析の基本動作であるシフトや還元の前後に処理を追加することができる．

`before_error`, `after_error`
: エラー処理の前後にそれぞれ呼ばれる．

`before_shift`, `after_shift`
: シフト処理の前後にそれぞれ呼ばれる．

`before_reduce`, `after_reduce`
: 還元処理の前後にそれぞれ呼ばれる．

`before_accept`, `after_accept`
: 受理処理の前後にそれぞれ呼ばれる．

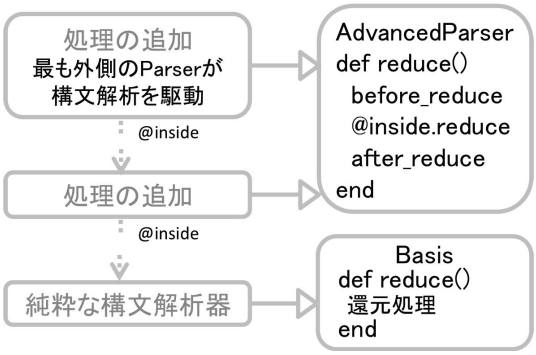


図 1　Depager が生成する構文解析器の概念図
Fig. 1　The concept of a parser Depager generated.

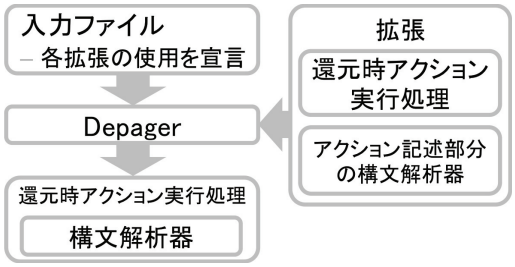


図 2　還元時アクション機能付与の概念図
Fig. 2　The concept of adding action functions.

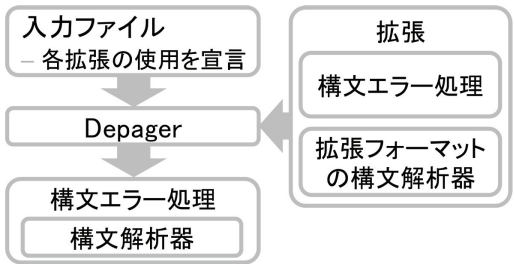


図 3　エラー処理拡張の概念図
Fig. 3　The concept of adding error handling functions.

たとえば還元後に Yacc のアクションに相当する処理を行う場合には，AdvancedParser クラスの子クラスを定義し，その `after_reduce` メソッドにアクションを実行するための処理を記述する．このクラスを拡張機能として利用すると，構文解析器は次のようにして還元時アクションを実行する．

1) 構文解析中の還元処理を行う際に，最も外側の拡張機能のクラスで `reduce` メソッドが呼ばれる．
2) `reduce` メソッドの定義は AdvancedParser クラスで図 1 のように定義されているため，まず自身の `before_reduce` メソッドを処理する．その後，層構造の 1 つ内側のクラスの `reduce` メソッドが呼び出される．
3) Basis の継承クラスの `reduce` メソッドが呼び出されると，還元処理が行われる．
4) 層構造の 1 つ外側のクラスに戻り，`after_reduce` メソッドが順に呼ばれていく．
5) 上述したように定義したクラスの `after_reduce` メソッドが呼ばれると，還元時アクションが実行される．
6) すべての拡張機能の `after_reduce` の処理が終わると，構文解析の処理の続きに戻る．

2.2　入力に対する拡張

Depager の拡張機能の種類によっては，Depager へ与える記述にその拡張に関する情報を含めるのが自然となる場合がある．たとえば，Yacc のアクションに相当する拡張の場合は，Yacc の記述と同様に `{` と `}` で囲んだコード断片を記述できるべきである．Depager ではこのために Depager への入力を拡張できる．拡張部分のフォーマットを解析するための構文解析器は，ユーザが手書きで作成することも不可能ではないが，Depager ではその生成もサポートしている．

図 2 は，還元時アクションの機能付与の概念図である．この機能の拡張は，生成される構文解析器が還元時にアクションを実行できるようにする処理の追加と，そのアクションのための記述を受け入れるための構文解析器から構成されている．Depager は実行時にこの拡張として与えられた構文解析器により Yacc と同様の `{` と `}` で囲まれた記述を解析し，生成される構文解析器（Basis を継承）実行時の各還元後にそれに対応したアクションが実行できる構文解析器（AdvancedParser を継承）を作り出す．

本研究では，この仕組みによって，エラー処理に必要となる情報を Depager に与えるようにし（**図 3**），エラー処理を拡張として作成する．

3.　既存のエラー処理手法の分析と共通する機能のライブラリとしての実装

本研究が目指すところは，Depager を用いることで，従来は，新たな構文エラー処理方法の研究において，そのアイデアの実装に多くの労力を注ぐ必要があったものが，最小限の労力でその実装や実験を行えるようになることを示すことである．

本章では，既知のエラー処理方法についてそのいくつかを取り上げて分析を行い，その結果に基づいて共通して必要な機能をライブラリとして実装することについて述べる．

3.1　実装の対象

本研究で対象とする構文エラー処理は，LALR を対象としたローカルなエラー処理とする．これは，Depager が原則として LALR の構文解析器を出力すること，およびローカルな構文エラー処理はエラー検出箇所以前の入力について正しいことを前提としていることから，比較的実装しやすいと考えられるためである．

その中で，広く知られた手法としてはエラー生成規則を利用したエラー処理と恐慌モードエラー処理を扱う [4]．エラー生成規則を利用したエラー処理は，構文規則中にエラートークンを含む規則を定義しておき，構文エラーの検出時に入力列中の誤っている箇所をエラートークンで読み替えて回復する手法である．恐慌モードエラー処理は，ここでは非終端記号のいくつかをキーとして，入力列の誤りを含む部分をキーのいずれかで置き換えて回復する手法である．

ローカルなエラー処理の中でも複雑な手法としては Corchuelo らの手法 [5] と McKenzie らの手法 [6] を扱う．Corchuelo らの手法は，入力列のエラー検出箇所を先頭として，そこから終端記号の挿入，削除，次の入力へ進む，

の 3 通りのいずれかの操作を繰り返し，挿入と削除の合計回数が少なく，回復後に次の入力へ進む操作を数回行ってもエラーが検出されないものを探す．McKenzie らの手法は，入力列のエラー検出箇所から挿入と削除を複数回行い，次の入力記号をシフトしたときにエラーとならないようなものを探す．このとき，各終端記号には挿入コストと削除コストを定義し，挿入時には挿入コスト，削除時には削除コストの総和が少ないものを選ぶ．

3.2　エラー処理手法の分析

上述した 4 つのエラー処理手法について，実装する観点から分析した．エラー処理のポイントとなるのは大きく次の 3 つである．

- エラー検出時点以降の入力列への操作
- エラー処理のための情報収集
- エラー処理のための基本動作

3.2.1　エラー検出時点以降の入力列への操作

ローカルなエラー処理方法では，エラーが検出された時点以降の入力を操作する．エラー生成規則によるエラー処理や恐慌モードエラー回復では，必要な数だけ入力を読み捨てる．

一方，上述した Corchuelo らや McKenzie らの方法では，エラーから回復するために試行錯誤的に最善の入力列を求める．このためには現時点以降の入力列を保持しつつ，入力列を適宜変更していく機能が必要となる．

3.2.2　エラー処理のための情報収集

エラー処理として，具体的な操作のために必要となる情報は，構文解析表に関連したものである．ここでは次の 2 点に着目した．

(1) ある状態において，エラーとならない先読み記号の集合
(2) ある非終端記号に対する FOLLOW 集合

(1) は，トークンの挿入や置換を考える際，ランダムにトークンを選ぶのではなく，その状態でエラーとならないものを選ぶ必要がある．

(2) は，恐慌モードエラー回復などで，特定の状態における先読み記号の集合が必要となる．

3.2.3　エラー処理のための基本動作

本研究では，LALR 構文解析におけるローカルな構文エラー処理を想定すると述べたが，LALR 構文解析では，正準 LR 構文解析に比べて，エラーの検出が遅れることがある．このため，エラーが見つかった時点の直前のシフトまでは少なくとも正しく解析が行われていたことになる．エラー処理のためには，その時点へ解析スタックを戻す必要がある．

エラー処理は，その時点の解析スタックに対して，なんらかの処置を施し，ある一定の基準を満たした時点で，元の解析動作へ復帰する．

3.3　構文エラー処理用ライブラリ

上述の分析に基づいて，共通している機能をライブラリとして実装した．ライブラリはその目的によって 3 つに分割して実装している．以下はライブラリの概要である．

LALex　入力列の先読みと書き換えを目的とするライブラリである．通常，字句解析器は 1 字句のみを切り出すだけであるが，エラー処理による繰返しの試行のために入力列を保持するための配列 `la_array` を使用する，字句解析器の拡張である．この拡張では，入力の記号列から任意の 1 記号を選択するためにインデックスの概念を用いる．インデックスは Depager デフォルトの記憶領域 `lookahead` を 0，配列 `la_array` の先頭を 1 とし，以降 $2, 3, \ldots$ と振られる．LALex により定義されるメソッドを次に示す．

- **lex_to_array()**　入力ストリームから次の記号を Depager デフォルトの先読み記号の記憶領域ではなく，配列の末尾に取得する．
- **lex_read(index)**　指定したインデックス index の記号を取得する．
- **lex_insert(item, index)**　item に指定した記号を入力列のインデックス index の位置に挿入する．
- **lex_delete(index)**　指定したインデックス index の記号を入力列から削除する．
- **get_tokens(until_index)**　入力列の先頭から指定したインデックス until_index までの記号を複製して返す．nil を指定した場合は現在記憶領域に保持している部分全体を返す．
- **replace_tokens(tokens)**　入力列の現在保持している部分全体を tokens で置き換える．

なお，`la_array` はすべての拡張機能に対して一意である必要があるため，`la_array` および上記のメソッドは Basis の継承クラスに定義される．すなわち，各拡張機能から呼び出す際には `basis.la_array` や `basis.method` のように記述する．

AfterErrContinue　Depager のデフォルトで構文エラー処理後に解析を終了する機能を無効化し，構文エラーからの回復処理を実装可能とする．

ErrHandle　構文エラー処理で必要とされるその他の機能を集約した．例として，ベースとなる構文解析器にエラーメッセージの表示の可否を切り替えるためのフラグである `errshow` を追加するといった機能がある．提供されるメソッドを以下に示す．

- **expected_tokens(id)**　状態を表す id を与え，その状態で次の先読み記号としてエラーとならない終端記号の集合を取得する．
- **expected_tokens_stack(stack)**　引数に解析スタックを与え，そのスタックの状況に対して

```
class Depager::LALR::ErrHandleParser <
      Depager::LALR::AdvancedParser; end

class Error_Production < Depager::LALR::ErrHandleParser
  def initialize # 初期化
    super
    yyerrok
  end
  def errhandle #  実際のエラー処理
    カウント変数を 3 に
    エラー状態フラグを立てる
    loop do
      action = 現在の状態とエラートークンにより
        アクションテーブルを引いた値
      action がエラーでも受理でもないなら，break
      解析スタックが空なら，exit
      解析スタックをポップ
    end
    basis.lex_insert(エラートークン，0)
    expected =
      expected_tokens(get_reduced_state(action))
    while basis.lex_read(1) が expected に含まれない
      先読みが入力列の末尾に達していれば，exit
      basis.lex_delete(1)
    end
  end
```

```
  def yyerrok # エラー状態からの回復
    カウント変数を 0 に
    エラー状態フラグを下げる
  end
  def yyerror
    errhandle # 強制的にエラー処理を呼び出す
  end
  def before_error
    basis.errshow = ! エラー状態フラグ
  end
  def after_shift
    if カウント変数が 0 以外 &&
        先読み記号がエラートークンでない
      カウント変数 -= 1
      カウント変数が 0 なら，エラー状態フラグを下げる
    end
  end
  def get_reduced_state(action) # errhandle で使用
    可能な限り還元し，最終的な解析スタックの状態を返す
  end
end

class Depager::LALR::Parser
  def yyerrok
    @inside.yyerrok
  end
  def yyerror
    @inside.yyerror
  end
end
```

図 4　エラー生成規則によるエラー処理を拡張として記述したもの
Fig. 4　An extension of the method using error production rules.

`expected_tokens()` より正確に，エラーとならない記号の集合を得る．

follow(n)　Depager では各終端記号と非終端記号に ID を割り振っており，引数に受け取る整数は各記号を表す．$n > 0$ ならば，n 番の終端記号の後に続いても構文エラーとならない終端記号の集合を返し，$n \leq 0$ ならば $-$n 番の非終端記号の FOLLOW 集合を返す．

4. 既存のエラー処理手法の Depager による実装

本章ではこれまでに述べた 4 つのエラー処理手法の Depager による実装について述べる．

4.1 エラー生成規則を利用したエラー処理

4.1.1 概要

エラー生成規則を利用する手法は Yacc でも採用されている手法であり，今回の実装ではそれを参考とした．この手法では，あらかじめ構文規則中にエラートークンと呼ばれる特別なトークンを含む規則であるエラー生成規則を追加しておく．構文規則を上記のように書き換えたとしても，エラートークンを含む規則は誤りのない入力については還元されないため，本来の文法の意味が損なわれることはない．エラー検出時には，入力列中の誤った終端記号列をエラートークンに対応させてシフトすることでエラーからの回復を試みる．エラーからの回復後はエラー状態の扱いとし，エラー処理による回復後の入力列から 3 記号が連続してエラーを検出されることなくシフトされるまではエラー状態を維持する．エラー状態の間は回復処理の影響による構文エラーが検出される可能性が高いと考えられることから，エラーメッセージを表示しない．

4.1.2 実装

拡張のための記述を図 4 に示す．実装は非常にシンプルであり，構文エラー検出時には `errhandle` メソッドが呼ばれ，ここでエラー処理がなされる．その内容は，構文解析器をエラー状態にあるとして設定し，先読み記号としてエラートークンを許す状態が見つかるまで解析スタックをポップしながらループすることである．そのような状態

```
class Depager::LALR::ErrHandleParser < Depager::LALR::AdvancedParser; end

class PanicMode_Recovery < Depager::LALR::ErrHandleParser
  DEBUG = true
  def initialize *args # 初期化
    super
    @keys = basis.class::PanicMode_KEYS を対応する ID の配列に変換したもの
  end
  def errhandle # 実際のエラー処理
    super
    removed = [] if DEBUG
    loop do
      @keys.each do |key|
        v = stack と key で GOTO テーブルを引いた値
        next if v がエラー
        expected = expected_tokens(v)
        if 入力列の末尾までに expected に含まれる終端記号が存在する
          stack << [key, :NT] << v # 還元と同様の扱い
          basis.lex_read(0) が expected に含まれない間，basis.lex_delete(0)
          removed << basis.lex_delete(0) により削除した記号列 if DEBUG
          puts "succeeded: #{key の記号},\n  removed: #{removed}" if DEBUG
          return
        end
      end
      解析スタックが空なら，exit
      removed.unshift(解析スタックの末尾の記号) if DEBUG
      解析スタックをポップ
    end
  end
end
```

図 5　恐慌モードエラー処理を拡張として記述したもの
Fig. 5　An extension of the method of panic mode.

が見つかった場合には，拡張機能 LALex により提供される `lex_insert()` によりエラートークンを入力列の先頭に挿入する．その後，拡張機能 ErrHandle により提供される `expected_tokens()` によりエラートークンの次に許される記号を取得し，それらのいずれかが見つかるまで入力列のエラートークンの次の記号を `lex_delete()` により削除していき，見つかった時点で回復処理を終了し，構文解析に復帰する．

`before_error()`（エラー処理の直前に呼ばれるメソッド）では，エラー状態にあれば，エラーメッセージの表示を行わない（エラー状態でなければ，エラーメッセージの表示を行う）設定をしている．

還元時のアクションを行うための拡張と一緒に使うことを想定して，Yacc における `yyerrok` や `yyerror` を実装した．これらはそれぞれエラー状態からの強制的な回復と，強制的なエラー処理を行うメソッドである．この実装例では，`Parser` クラスに `yyerrok()` と `yyerror()` を定義することで，還元時アクションにおいてそれらをメソッドとして呼び出すことを可能としている．拡張はデコレータにより実装されるので，各オブジェクトは `@inside` により 1 つ内側にあるオブジェクトを参照する．

この拡張機能の実際のコードは 60 行程度に収まっている．

4.1.3　検証

Yacc によるエラートークンを用いたエラー処理の例として，文献 [7] に掲載されている 2 つの例を Depager で実装し，実行した．1 つは `ERROR` トークンを用いた生成規則が機能するかどうかのみを調べるためのサンプルであり，もう 1 つは，算術式においてゼロによる除算の際，アクションによって `yyerrok` の使用により回復するものである．

いずれも文献 [7] に掲載されている実行結果と同じものを得た．

4.2　恐慌モードエラー処理

4.2.1　概要

恐慌モードエラー処理では，キーとなる非終端記号をあらかじめいくつか定めておく．ここでキーとして定めた非終端記号を左辺とする規則の右辺に対応する入力に誤りが

見つかった場合，その右辺に該当する部分すべてを強制的にキーの非終端記号として還元することで構文エラーから回復する．

4.2.2　実装

恐慌モードエラー処理本体の記述は，**図 5** のようにして，約 30 行で作成することができた．この実装は，エラーが検出され `errhandle` メソッドが呼ばれると，解析スタックをポップしながら置き換えが可能なキーを探し，見つかれば置き換えを行う簡単なものである．なお，解析スタックをポップする操作は，シフトや還元によりスタックに積まれた状態をポップすることで，そのシフトや還元を誤りと見なし，キーによる置き換えの対象とするものである．

Depager を用いることで，構文解析器の生成時に作成者がキーとなる非終端記号を与えることができるようにできる．このためには，Depager が受け取る入力を拡張するための拡張機能も作成する必要がある．

キーとなる非終端記号の集合は次のフォーマットを生成規則の前に記述できるようすることにする．

```
%PanicMode_KEYS{
非終端記号名 A，非終端記号名 B，...
%}
```

このように入力を拡張するための Depager の拡張は，40 行未満のコードから生成できた．

4.2.3　検証

このエラー処理手法は，入力列からのトークンの削除のみによって行われるため，比較的多くのトークンを切り捨てることになる．このため，ここでは比較的大きい文法で検証を行うこととし，その文法としては Java (1.4) のものを選んだ．エラーを含んだサンプルプログラムを**図 6** に示す．

ここでは実験のため，図 6 に関連した非終端記号を次のように Depager への記述として与えた．

```
%PanicMode_KEYS{
  :variable_initializers, :expression_statement
%}
```

このサンプルプログラムを生成した構文解析器に与えた際の出力を**図 7** に示す．この結果から，(E1) については抜けているコンマの両端にあたる文字列の削除により空の配列の定義とみることで回復していることが分かる．(E2) と (E3) については，expression_statement，すなわち式全体をキーとしているため，いずれも末尾のセミコロンが欠けている式全体を削除することで回復している．これらから，サンプルプログラム中の 3 カ所の構文の誤りをすべて検出し，それぞれエラーから回復することで入力列の末尾まで解析が行われたことを確認した．

```
public class SampleJava {
  public static final String[] S = {
    "Hello," /* , コンマ抜け（E1）*/  "World" };
  public static void main(String[] args) {
    // セミコロン抜け（E2）
    System.out.print(S[0])//;

    // セミコロンをコロンで打ち間違え（E3）
    System.out.println(S[1] + '!')/* ; */:
  }
}
```

図 6　誤りを含んだ Java のサンプルプログラム
Fig. 6　A sample program contains three errors in Java.

```
SampleJava.java:2: syntax error(...),
  unexpected :STRING_LITERAL, expecting  ... .
  succeeded: variable_initializers,
    removed: [:STRING_LITERAL, :STRING_LITERAL]
SampleJava.java:5: syntax error(...),
  unexpected :IDENTIFIER, expecting ... .
  succeeded: expression_statement,
    removed: [:name, :L_PARENTHESE, :argument_list_q,
    :R_PARENTHESE]
SampleJava.java:5: syntax error(...),
  unexpected :COLON, expecting ... .
  succeeded: expression_statement,
    removed: [:name, :L_PARENTHESE, :argument_list_q,
    :R_PARENTHESE, :COLON]
accept
```

図 7　恐慌モードエラー処理による出力例
Fig. 7　The output of the method of panic mode.

4.3　Corchuelo らの手法

4.3.1　概要

Corchuelo らの手法は，エラー時の解析スタックとエラーを検出した箇所を先頭とする入力列に対し，後述する回復遷移を複数回適用することによりエラーからの回復を試みる手法である．回復の基準としては，回復後に N 個の終端記号をエラーなくシフトできるか，あるいはそのシフトの途中で入力列の末尾に至った場合に成功となる．

回復遷移とは，簡単には入力列への終端記号の挿入，入力列の記号の削除，入力列からシフトするといった 3 つの操作のいずれかをとり，解析スタックや残りの入力列が変化する遷移を表す．しかし，たとえばある終端記号の挿入後に同記号を削除する遷移の繰返しを考えると，エラーから回復可能な遷移の組合せ（本節ではこれを回復遷移列と呼ぶことにする）は無限に存在するといえる．そこでこの手法では，そのような回復遷移列の中から，最も少ない数の挿入と削除によるもの（その数の和を本節ではコストと呼ぶことにする）を選び，適用する．その数が同一であるものについては，特に選定の基準は定められていない．

4.3.2　実装

設計の例を図 8 に示す．この例では構文エラー検出後に呼ばれる errhandle メソッドにより，まず try_repair メソッドを呼び出して適切な回復遷移列を探し，もし見つかればそれを残りの入力列に実際に適用して回復を行い，エラー処理を終了する流れとなる．

遷移の探索では，最初に空の遷移列を 1 つ用意し，それをキューに入れる．以後，キューから最もコストの低い遷移列を 1 つ取り出し，それに対し次に行ってもエラーとならない遷移（挿入，削除，入力のシフト）のいずれかを 1 回だけ行った遷移列のすべてを候補としてキューに格納することを繰り返す．この繰返しの終了条件は，キューから取り出した時点で遷移列が回復に成功している場合か，キューが空となった場合となる．後者の場合は遷移列の探索失敗を表すため，exit を呼び出して構文解析をそこで終了する．

探索により回復可能な遷移列が得られた場合には，それを入力列に適用して回復させる．これはライブラリ LALex の機能である lex_insert()，lex_delete() を用いることで非常に容易に実装が可能であった．

この実装は 170 行未満に収めることができた．

4.3.3　検証

この手法は，実装上の制約を無視すればどのような構文エラーからでも回復が可能であると述べられているため [5]，検証は比較的大きい文法として，Java (1.4) により図 6 に示したコードを用いて行った．

図 9 に示す結果から，3 つすべての構文エラーから回復することに成功し，期待どおりのエラー処理が行われていることを確認した．ただし今回行った実装では，3 つの

```
module Corchuelo_Module
  DEBUG = true # メッセージ表示
  # アルゴリズムのパラメータ
  N  = 3
  Nt = 10
  Ni = 4
  Nd = 3
  # 遷移の表現
  DELETE = false
  FMOVE  = nil
end

class Depager::LALR::ErrHandleParser <
      Depager::LALR::AdvancedParser; end

class Corchuelo_Recovery <
      Depager::LALR::ErrHandleParser
  include Corchuelo_Module
  def initialize(*args) # 初期化
    super
    @queue = []
  end
  def errhandle # 実際のエラー処理
    super
    @queue << Corchuelo_Configuration.new(解析スタック)
    result = try_repair
    if result && result.repaired?
      if DEBUG # 回復に成功した場合のメッセージ
        puts "repair succeeded #{result の repair}"
      end
      index = 0
      result.repair_each do |r|
        case r
        when DELETE # 遷移：削除
          basis.lex_delete(index)
        when FMOVE # 遷移：入力を進める
          index += 1
        else # 遷移：挿入
          basis.lex_insert(ID が r の終端記号, index)
          index += 1
        end
      end
    else
      puts "repair failed" if DEBUG # 回復失敗時
      exit
    end
    @queue を空にする
  end
  def next_conf # 次の配置をひとつ取り出す
    @queue を配置のコストで昇順にソートし，
    先頭を取り出して返す
  end
  def add(conf) # 新しい配置をキューに追加する
    @queue << conf if
      conf.repaired? || conf.more_repair?
  end
  def try_shift(conf, act, repair) # 遷移中のシフト
    conf.spush(act).rpush(repair)
    還元が可能な間，try_reduce(conf, v)
  end
  def try_reduce(conf, act)
    conf.reduce による還元操作
  end
  def try_repair # 可能な遷移の探索ループ
    while (conf = next_conf) && !conf.repaired?
      try_ER1(conf)
      try_ER2(conf)
      try_ER3(conf)
    end
    conf
  end
  def try_ER1(conf) # 挿入による遷移の評価
    return unless conf.more_insertion?
    shifti = basis.lex_read(conf.index)
```

```
    expected_tokens(conf.state).each do |i|
      next if i == shifti # try_ER3 で行う
      c = conf.clone
      loop do
        a = c.state と i でアクションテーブルを引いた値
        if a.nil? # エラー
          break
        elsif a == ACC # 受理
          c.index が入力の末尾なら,
            add(c.spush(a).rpush(i).make_repaired)
          break
        elsif a > 0 # シフト
          try_shift(c, a, i)
          add(c)
          break
        else # 還元
          try_reduce(c, a)
        end
      end
    end
  end
  def try_ER2(conf) # 削除による遷移の評価
    add(conf.clone.lex_forward.rpush(DELETE)) if
      conf.more_deletion?
  end
  def try_ER3(conf) # 入力を次に進めることによる遷移の評価
    return unless t = basis.lex_read(conf.index)
    c = conf.clone.lex_forward
    loop do
      a = c.state と t でアクションテーブルを引いた値
      if a.nil? # エラー
        break
      elsif a == ACC # 受理
        add(c.spush(a).rpush(FMOVE).make_repaired)
        break
      elsif a > 0 # シフト
        try_shift(c, a, FMOVE)
        c.judge unless c.repaired?
        add c
        break
      else # 還元
        try_reduce(c, a)
      end
    end
  end
end

class Corchuelo_Configuration # 配置
  include Corchuelo_Module
  attr_reader :index
  def initialize(stack, index = 0, repair = nil)
    @stack = stack.clone
    @index = index
    @repair = repair ? repair.clone : []
    @repaired = false
  end
  def state(n = 0) # スタックの状態を返す
    @stack[-n - 1]
  end
  def cost
    @repair 内の挿入と削除操作の数を返す
  end
  def more_repair?
    @repair.size < Nt
  end
  def more_insertion?
    @repair 内の挿入の数 < Ni
  end
  def more_deletion?
    @repair 内の削除の数 < Nd
  end
  def repaired?
    @repaired
  end
  def judge
    make_repaired if @repair.size > N &&
      @repair の末尾 N 要素が全て FMOVE
  end
  def clone
    self.class.new(@stack, @index, @repair)
  end
  def reduce(v, x)
    @stack.pop x
    @stack << v
  end
  def spush(v)
    @stack << v; self
  end
  def rpush(v)
    @repair << v; self
  end
  def lex_forward
    @index += 1; self
  end
  def make_repaired
    @repair の末尾に連続する FMOVE を削除
    @repaired = true
    self
  end
  def repair_each(&block)
    @repair.each &block
  end
end
```

図 8　Corchuelo らの手法を拡張として記述したもの
Fig. 8　An extension of Corchuelo's method.

構文エラーのうち，図中の E1，E2 については構文上は正しいが意味上は Java (1.4) に沿わない記号としてそれぞれ‘*’(STAR)，‘.’(DOT) が挿入される回復がなされた．これは 4.3.1 項で述べたように，この手法において挿入さ

```
SampleJava.java:2: syntax error ... .
repair succeeded ["ins STAR"]
SampleJava.java:5: syntax error ... .
repair succeeded ["ins DOT"]
SampleJava.java:5: syntax error ... .
repair succeeded ["ins SEMICOLON", "del"]
accept
```

図 9　Corchuelo らの手法による出力例
Fig. 9　The output of Corchuelo's method.

れる記号の判断基準がコストのみであり，それが互いに等しい遷移群からどれが選択されるかは実装に依存するためである．

4.4　McKenzie らの手法

4.4.1　概要

McKenzie らの手法は，エラーを検出した箇所を先頭とする入力列に，終端記号の挿入と削除をそれぞれ複数回適用してエラーからの回復を試みる手法である．この回復は，先読み記号 1 つをシフトできれば成功の扱いとなる．ただし，挿入を行った記号は削除の対象としないことを前提としており，挿入と削除の順は結果に影響しない．

各終端記号には挿入コストと削除コストがあると仮定し，回復に成功する組合せの中から挿入する記号の挿入コストと，削除する記号の削除コストの合計が低いものが選択される．

4.4.2　実装

実装としては，終端記号のコストを入力の拡張で定義する．コストは 0 以上の整数とし，

```
%COST{
終端記号Aのシンボル　挿入コストA, 削除コストA
終端記号Bのシンボル　挿入コストB, 削除コストB
:
%}
```

というフォーマットを生成規則の前に記述できるようにし，

```
INSERTION_COST = {
終端記号Aのシンボル => 挿入コストA,
終端記号Bのシンボル => 挿入コストB,
:
}
DELETION_COST = {
終端記号Aのシンボル => 削除コストA,
終端記号Bのシンボル => 削除コストB,
:
}
```

という形式で構文解析器のクラスに書き込むようにする．このための入力の拡張は 50 行未満のコードから生成することができた．

一方，実際にこのコストの定義を利用するエラー処理の本体は，図 **10** のように記述でき，まず適切な回復を探し，それが見つかれば入力列に適用してエラー処理を終了する流れとなる．

回復の候補となるクラス Mckenzie_Configuration には，挿入する記号の列と削除する記号数といった情報を持たせ，最初は挿入する記号列が空，削除する記号数が 0 の候補をキューに入れておく．以後，キューから最もコストの合計

```
class Depager::LALR::ErrHandleParser <
      Depager::LALR::AdvancedParser; end

class Mckenzie_Recovery <
      Depager::LALR::ErrHandleParser
  DEBUG = true # メッセージ表示
  AFTER_READ = 3 # 処理後に立ち戻る記号数
  def initialize(*args) # 初期化
    super
    @recovering = 0 # 立ち戻りの記号数のカウンタ
    @icost = [nil, nil] # 挿入コスト
    @dcost = [nil, nil] # 削除コスト
    each_term do |i|
      @icost[i] =
        basis.class::INSERTION_COST[ID が i の終端記号]
      @dcost[i] =
        basis.class::DELETION_COST[ID が i の終端記号]
    end
  end
  def errhandle # 実際のエラー処理
    super
    new_recovery = @recovering == 0
    conf = recover # 探索
    save_configuration(@recovering, new_recovery)
    conf.deletion_times.times{ basis.lex_delete(0) }
    conf.insertion_each_with_index do |t, index|
      basis.lex_insert(ID が t の終端記号, index)
    end
    puts "ins: #{conf の inserted}, " +
      "del: #{conf.deletion_times}" if DEBUG
  end
  def after_shift
    super
    @recovering -= 1 if @recovering > 0
  end
  def before_error # 立ち戻り時はメッセージ表示しない
    super
    basis.errshow = @recovering == 0
  end
  def save_configuration(index, new_recovery)
    if new_recovery # 立ち戻りでない：新しく保持
      @error_stack = stack.clone
      la_array の要素数が index になるまで,
        basis.lex_to_array
    else # 立ち戻り：必要なら更新
      old_size = basis.la_array.size
      la_array の要素数が index になるまで,
```

```
      basis.lex_to_array
    return if old_size == basis.la_array.size
  end
  @error_read = basis.get_tokens(index)
end
def restore_configuration # 保持した配置の復元
  basis.stack = @error_stack.clone
  basis.replace_tokens(@error_read)
end
def clear_queue
  @queue = []
end
def enqueue(conf)
  @queue << conf
end
def delete_min
  @queue をコストの昇順でソートし,
  先頭要素を取り出して返す
end
def recover
  if @recovering == 0 # 立ち戻りでない
    clear_queue
    enqueue(Mckenzie_Configuration.new(解析スタック))
  else # 立ち戻り
    restore_configuration
  end
  until @queue.empty?
    conf = delete_min
    if conf.deleted_empty?
      # コストが定義されているのが前提
      expected = expected_tokens(
        conf.state).select{|i| @icost[i]}
      expected.each do |i|
        a = conf.state と i でアクションテーブルを
        引いた値
        if a.nil? || a == ACC # エラーか受理
        elsif a > 0 # シフト
          enqueue(conf.insert(a, i, @icost[i]))
        else # 還元
          do_reduce(conf.clone, a, i)
        end
      end
    end
    if i = basis.lex_read(conf.index)
      enqueue(conf.delete(@dcost[i])) if @dcost[i]
      return conf.tap{
       @recovering = AFTER_READ + conf.count } if
       expected_tokens_stack(conf.stack).include?(i)
    end
  end
  puts "recovery failed" if DEBUG
  exit
end
def do_reduce(conf, a, i)
  conf.reduce による還元
    a = conf.state と i でアクションテーブルを引いた値
    if a.nil? || a == ACC # エラーか受理
    elsif a > 0 # シフト
      enqueue(conf.insert(a, i, @icost[i]))
    else # 還元
      do_reduce(conf, a, i)
    end
  end
end

class Mckenzie_Configuration # 配置
  attr_reader :index, :cost, :stack
  def initialize(stack, index = 0, inserted = [],
    cost = 0)
    @stack = stack
    @index = index
    @inserted = inserted
    @cost = cost
  end
  def clone
    self.class.new(@stack.clone, @index,
      @inserted.clone, @cost)
  end
  def insert(dstack, dinserted, dcost) # 挿入した配置
    self.class.new(@stack + [dstack], @index,
      @inserted + [dinserted], @cost + dcost)
  end
  def delete(dcost) # 削除した配置
    self.class.new(@stack, @index + 1,
      @inserted, @cost + dcost)
  end
  def reduce(v, x)
    @stack.pop x
    @stack << v
  end
  def deleted_empty?
    @index == 0
  end
  def state(n = 0)
    @stack[-n - 1]
  end
  def insertion_each_with_index(&block)
    @inserted.each_with_index &block
  end
  def deletion_times
    @index
  end
  def count
    @inserted.size
  end
end
```

図 10　McKenzie らの手法を拡張として記述したもの
Fig. 10　An extension of McKenzie's method.

が低い候補を 1 つ取り出し，それに対し入力列の削除や，エラーとならない記号の挿入を行った場合を示す候補のすべてをキューに格納することを繰り返す．挿入と削除の順は結果に影響しないため，挿入については，削除する記号数が 0 である場合のみ考慮する．この繰返しの終了条件は，キューから取り出した回復の候補が回復に成功している場合（実際には後述するオプションの実装のため，取り出した候補に対し挿入や削除を行ったものをキューに格納した後に判定を行う）か，キューが空となった場合である．

探索により得られた回復の適用操作については，その回復の削除する記号数だけ `lex_delete()` を繰り返し呼び出した後，挿入する記号列をそれぞれ `lex_insert()` を用いて入力列に挿入するのみである．

ただしこの手法では，回復後に構文エラーが起きないことが保証されているのは，残りの入力列の先頭の 1 つのみである．そのため，入力列の 2 番目以降の記号により，回復に起因する構文エラーが検出される可能性が比較的高いといえる．この問題については，回復後 3 記号以内で再び構文エラーが検出される場合には，直前の回復前の配置（解析スタックと入力列の残りの組）に戻し，その回復時に選択された候補を除いた候補群から再度適切な回復を探索し直すことで対処する．この対処方法は文献 [6] 中ではアルゴリズム本体に含まれず，アルゴリズムの性能を向上させるためのオプションとして紹介されているものであるが，今回は実装に含めることとした．

この 3 記号のカウントには変数を用意し，`after_shift` メソッドでデクリメントしていく．これが 0 となる前に再び構文エラーが検出された場合には，回復後に 3 記号をシフトする前であるため，回復前の配置に立ち戻る．

立ち戻りに必要である，構文エラー検出直前の配置を構成する解析スタックと入力列の内容は，回復後の解析で書き換えられてしまうため，この拡張機能内に保持しておく．入力列の保持には LALex により提供される `get_tokens()` が利用でき，現在の入力列を保持した入力列で書き換える際には `replace_tokens()` を利用できる．

この実装は 140 行未満で行うことができた．

4.4.3　検証

この手法についても，結果の比較のため，Java (1.4) として図 6 のコードを解析させることで検証を行った．この手法は終端記号にコストの概念があるため，コストの定義により結果は異なる．今回対象とするコードにおいて構文だけでなく，図 6 で期待される意味上でも正しくなるような記号の挿入または削除を優先的に行うようにコストを設定した．そのときの出力が**図 11** である．この結果より，E1 については抜けているコンマを挿入，E2 については抜けているセミコロンの挿入，E3 についてはエラー検出直後の 1 記号（コロン）を削除し，抜けているセミコロンの挿入を行うことでエラーから回復していることが分かる．

```
SampleJava.java:2: syntax error(...),
  unexpected :STRING_LITERAL, expecting ... .
  ins: [:COMMA], del: 0
SampleJava.java:5: syntax error(...),
  unexpected :IDENTIFIER, expecting ... .
  ins: [:SEMICOLON], del: 0
SampleJava.java:5: syntax error(...),
  unexpected :COLON, expecting ... .
  ins: [:SEMICOLON], del: 1
accept
```

図 11　McKenzie らの手法による出力例
Fig. 11　The output of McKenzie's method.

これは，すべて期待どおりの回復を行うことができたことを示す．

5.　おわりに

本論文では，構文エラー処理を，拡張可能な構文解析器生成系 Depager の拡張機能として実装する手法について述べた．既知のエラー処理方法を分析し，ライブラリとしてまとめたうえで，この方法を用いることで，4 つの構文エラー処理手法を数十行から百数十行程度で記述することができた．この結果，新しい構文エラー処理に関する研究や，または複数の構文エラー処理の比較をする際にともなう実装を容易にすることができることを示した．

このような，構文エラー処理の実装をサポート可能なツールとして，ANTLR [8] があげられる．ANTLR では，エラー処理クラスのオブジェクトを置き換えることで標準とは別のエラー処理を実装できる．本研究の場合は，拡張として開発されているエラー処理手法をその開発者と別の言語処理系作成者が容易に用いることができることと，Depager へ与える入力も変更できることにより，その拡張としてのエラー処理へのパラメータを言語処理系の開発者の意向により容易に調整できる点が ANTLR と比較した場合の利点である．

今後の課題は，より多くの構文エラー処理を実装することにより，ライブラリに必要とされる機能を洗練させていくことである．

なお，今回実装したライブラリやエラー処理の拡張は http://www.slis.tsukuba.ac.jp/˜nakai.hisashi.gt/に公開している．

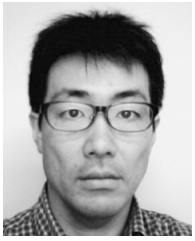
参考文献

[1] Kim, I.-S. and Choe, K.-M.: Error Repair with Validation in LR-based Parsing, *ACM Trans. Programming Languages and Systems* (*TOPLAS*), Vol.23, No.4, pp.451–471 (2001).
[2] Donnelly, C. and Stallman, R.: Bison. The YACC-compatible Parser Generator, Free Software Foundation (2004).
[3] 舞田純一，佐藤　聡，中井　央：機能拡張可能なコンパイ

ラ生成系，情報処理学会論文誌．プログラミング，Vol.47, No.16, pp.1–9 (2006).
[4] 佐々政孝：プログラミング言語処理系，岩波書店 (1989).
[5] Corchuelo, R., Pérez, J.A., Ruiz, A. and Toro, M.: Repairing Syntax Errors in LR Parsers, *ACM Trans. Programming Languages and Systems* (*TOPLAS*), Vol.24, No.6, pp.698–710 (2002).
[6] McKenzie, B.J., Yeatman, C. and de Vere, L.: Error Repair in Shift-Reduce Parsers, *ACM Trans. Programming Languages and Systems*, Vol.17, No.4, pp.672–689 (1995).
[7] 五月女健治：yacc/lex プログラムジェネレータ on UNIX，テクノプレス (1996).
[8] Parr, T. and Fisher, K.: LL (*): the foundation of the ANTLR parser generator, *ACM SIGPLAN Notices*, Vol.46, No.6, ACM, pp.425–436 (2011).

細田 将大　（学生会員）

1994 年生．2016 年筑波大学情報学群卒業．筑波大学システム情報工学研究科在学中（2016 年現在）．



中井 央　（正会員）

1968 年生．筑波大学第三学群情報学類卒業，同工学研究科修了（博士（工学））．1997 年 10 月図書館情報大学助手，2001 年 8 月同総合情報処理センター講師，2002 年 8 月同助教授，2002 年 10 月の筑波大学との統合により，筑波大学図書館情報メディア研究科助教授（学術情報メディアセンター勤務），2007 年 4 月同准教授．日本ソフトウェア科学会，ACM，ACM-SIGMOD-JAPAN 各会員．

佐藤 聡　（正会員）

筑波大学システム情報系准教授（学術情報メディアセンター勤務）．1996 年筑波大学大学院工学研究科単位取得退学．同年広島市立大学情報科学部助手．2001 年筑波大学講師．2013 年より現職．博士（工学）．主に，キャンパスネットワークの企画管理運用に関する研究に従事．

新城 靖　（正会員）

1965 年生．1988 年筑波大学第三学群卒業．1993 年筑波大学大学院工学研究科博士課程修了．同年琉球大学工学部情報工学科助手．1995 年筑波大学電子・情報工学系講師，2003 年同助教授，2004 年同大学院システム情報工学研究科助教授．2007 年同准教授．オペレーティング・システム，並行システム，仮想化，情報セキュリティの研究に従事．博士（工学）．ACM，IEEE，USENIX，日本ソフトウェア科学会各会員．