解析表現文法の開発支援のためのデバッガの実装と評価

本多　峻　倉光　君郎

パーサジェネレータは，形式文法に基づく生成的なパーサ開発法であり，今日、標準的に用いられている。しかし、パーサジェネレータの文法開発は，ベースとなる形式文法による特有の困難さがあり，必ずしも簡単な作業ではない．特に、解析表現文法 PEG は，LALR や LL 文法には存在しないネストされたバックトラックの発生が文法開発を困難にしている．本論文では，PEG 文法開発を動的に支援するため，PEG デバッガを提案する．PEG デバッガは，独自の PEG インタプリタ上で動作し，ブレークポイント，スタックトレース，ステップ実行などの機能がある．我々は，PEG デバッガを Nez 構文解析ツールライブラリの一部として実装し，いくつかの困難なデバッグに役立ったことを報告する．

Parser generator is a generative parser development method based on formal grammar, it has been used as a standard. However, grammar development of parser generator has unique difficulties in the based grammar, it is not necessarily easy tasks. Especially, the Parsing Expression Grammar is difficult to develop grammars because of nested backtrack that do not exist in the LALR and LL grammar. In this paper, we propose a PEG debugger to dynamically support the PEG grammar development. PEG debugger operates on its own PEG interpreter, and it has features such as breakpoints, stack trace and single stepping. We implement the PEG debugger as part of Nez parser libraries and report that it has served to some difficult debugging.

1　はじめに

パーサジェネレータは，形式文法に基づく生成的なパーサ開発法であり，多くの実用的なアプリケーションで用いられている．開発者は，パーサジェネレータを利用するために CFG や PEG [4] などの文法を定義し，定義した文法からパーサを生成する．しかし、パーサジェネレータの文法開発は，ベースとなる形式文法による特有の困難さがあり，必ずしも簡単な作業ではない．特に PEG は，他の LALR や LL 法とは異なる特徴が開発をより困難にすることがある．

PEG の文法を開発する際，開発者は EBNF で記述されたそれぞれの言語の定義から，PEG へと変換することが多い．これは自動化された技術ではなく，現在は手動で行われている．そのため，PEG と EBNF の違いから，開発した文法が受理すべき入力を受理しないという事態が発生する．図 1 を用いて EBNF と PEG の違いの一例を示す．図 1 の (1) ように EBNF を定義した場合は ab と a にマッチするが，図 1 の (2) のように EBNF を直接 PEG に変換すると，a にしかマッチしなくなる．このようなバグを文法中から発見し，修正することは非常に困難な作業である．

```
A = 'a' | 'ab'
   (1)EBNF
A = 'a' / 'ab'
   (2)PEG
```

図 1　PEG と EBNF の例

我々は，PEG の文法のバグ修正を支援するための，PEG デバッガを提案する．PEG デバッガは，独自の PEG インタプリタ上で動作し，ステップ実行によっ

Implementing and Evaluating a Debugger for Supporting Development of PEGs
Shun Honda, 横浜国立大学大学院工学府, Faculty of Engineernig, Yokohama National University.
Kimio Kuramitsu, 横浜国立大学大学院工学府, Faculty of Engineernig, Yokohama National University.

表 1　PEG の演算子

PEG	Type	Proc.	Description
’ ’	Primary	5	文字列リテラル
[]	Primary	5	文字クラス
.	Primary	5	任意の文字
A	Primary	5	非終端記号
(e)	Primary	5	グルーピング
$e?$	Unary suffix	4	オプション
$e*$	Unary suffix	4	0 回以上の繰り返し
$e+$	Unary suffix	4	1 回以上の繰り返し
$\&e$	Unary prefix	3	肯定先読み
$!e$	Unary prefix	3	否定先読み
$e_1 e_2$	Binary	2	シーケンス
e_1 / e_2	Binary	1	優先度付き選択

て詳細な PEG パーサの動作を確認することができる．また，PEG の特徴による文法開発の困難に対する支援機能を有する．

本論文の構成は以下の通りである．まず，第 2 節では PEG の説明とその文法開発の課題について述べる．第 3 節では，デバッガによる PEG の文法開発の支援，デバッガの設計思想について説明する．第 4 節では，PEG デバッガの詳細な実装について述べる．第 5 節では，いくつかのデバッグが困難な文法のバグについて，PEG デバッガが役立ったことを報告する．第 6 節では，関連研究について説明する．第 7 節で，本論文の結論を述べる．

2　背景と動機

2.1　Parsing Expression Grammars

PEG は，Ford により提唱された形式文法であり，バッカス・ナウア記法と似た構文を持つ [4]．PEG は A を非終端記号，e を解析表現としたときに，$A = e$ で表されるルールの集合である．解析表現は PEG のオペレータによって構成されている．

表 1 は PEG のオペレータの概要である．文字列リテラル ’abc’ は同じ入力とマッチし，[abc] はその中のどれか 1 つが同じ入力とマッチする．. は任意の 1 文字とマッチする．字句のマッチに成功した時，マッチした文字数分解析位置を進める．$e?$，$e*$，$e+$ は正規表現と正規表現と共通の動作をするが，これらの表現は貪欲であり，最長の位置でマッチする．二つの表現の連接を表す $e_1\ e_2$ は，どちらかのマッチに失敗した時開始位置までバックトラックする．優先度付きの選択を表す e_1 / e_2 は，最初に e_1 を試し，失敗した場合に e_2 を試す．肯定先読み $\&e$ は e を試すが，文字を消費しない．否定先読み $!e$ は e が成功したら失敗とし，e が失敗したら成功とする．

PEG は字句解析と構文解析を同時に行う．これにより，あらかじめ定義された字句パターンに左右されない柔軟な構文木の構築が可能となる．また，PEG は Packrat Parsing [2] によって入力長に対して線形時間での解析を保証する．

2.2　文法のバグ

文法のバグは，受理すべき入力を文法が受理しない状態のことである．これは，開発者が文法とそこから生成されるパーサの動作を結びつけられていないことが原因である．そのため，文法のバグ修正は，開発者がパーサの動作を理解することが重要になる．従来の文法のデバッグ作業では，文法エラーの情報をヒントに，開発者が文法を手探りで追うことになる．PEG のエラー報告は，最長マッチの位置情報に基づく報告 [3] が一般的だが，このエラー報告だけで文法のバグを発見することは難しい．

バグの発見が困難な PEG のエラー報告の例を以下で示す．まず，図 2 に文法のバグを含んだ四則演算を表現する PEG の文法の例を示す．

```
Expr     = Sum
Sum      = Product (( '+' / '-' ) S* Product )*
Product  = Value (S* ( '*' / '/' ) S* Value )*
Value    = [0-9]+ / '(' S* Expr S* ')'
S        = ' '
```

図 2　バグを含んだ四則演算を表現する PEG の文法

この例では，Sum のルールで’+’ と’-’ の演算子の前に空白を受理するルールが必要であるがそれを定義していない．図 3 に入力とそれに対するエラー報告を示す．

このエラーレポートを見ると，Product のルールのバグに見えるが，実際にバグがあるのは Sum のルー

```
Input: 2*(3 + 4)*7
Error Report:
 Line 1, column 6: Expected ' ', '*', '/' but found '+'
```

図 3　四則演算の PEG のエラー報告の例

ルであり，このようなバグを修正するのは困難な作業となる．

これは，図 2 の文法に対して，図に示した入力を入れた時のエラー報告である．

2.3　PEG の文法開発の困難

文法開発における困難さは，文法のバグの発見，修正にある．その中でも，特に文法のバグに繋がりやすく，修正が難しい要素がある．本節では，それらの PEG の文法開発の難所について詳細に説明する．

2.3.1　ネストされたバックトラック

PEG の文法の特徴として，ネストされた構造に強いことが挙げられる．しかし，その特徴は同時に複雑にネストされたバックトラックを発生させる．この複雑にネストされたバックトラックは PEG の文法開発上の課題の 1 つである．複雑にネストされたバックトラックの動作を，開発者が直感的に理解するのは難しい．そのため，開発者の予期していないパーサ動作が，文法のバグとなることがある．このような文法のバグを修正するのは，パーサの動作の理解が必要不可欠となる．

2.3.2　到達不能な選択

到達不能な選択は．PEG の優先度付き選択の特徴によって発生する文法のバグである．優先度付き選択では，最初の選択にマッチした場合，次の選択は試さない．よって，最初の選択に次の選択の先頭の文字列が含まれる時，先の選択で入力が受理されてしまう．次のような例で考える．

```
A = 'a' / 'ab'
```

この例では ab を入力として与えた時，先の選択で'a' にマッチし解析が終了する．そのため，ab 全体にマッチすることができない．ab にマッチする文法とするためには，文法を次のように定義する必要がある．

```
A = 'ab' / 'a'
```

このように，PEG の選択は，正しく定義されていない場合，到達不能な選択ができてしまうことがある．

2.3.3　コードレイアウト

コードレイアウトとは，空白や改行，インデントのことである．PEG は字句解析を同時に行う性質上，コードレイアウトの定義を正確に行うことが文法開発上の課題となる．これは，PEG の文法を定義する上での難所の一つである．我々のこれまでの文法開発の経験から，コードレイアウトに関する文法のバグは，発生頻度が高いことが分かっている．また，コードレイアウトのルールは，繰り返しなどの表現が組み合わされて定義されることが多いため，解決がさらに困難になる．

3　デバッガによる文法開発支援

デバッガを用いるアプローチは，開発者にパーサの挙動を理解させることに役立つ．また，動的な支援を行うことで，文法のバグの発見をより容易にする．本節では，デバッガによる文法開発支援の概要と，PEG の文法デバッガの設計について述べる．

3.1　概要

開発者は．これまでパーサから出力される文法エラーの情報を元に，文法のバグを探すという方法でデバッグを行ってきた．しかし，この方法によるデバッグでは，開発者はどのルールで入力のどの部分を受理したのか，どのようにバックトラックが発生しているのかなど，パーサの動作を理解することが難しい．そのため，開発者は，なぜ文法がその入力を受理しないのかを直感的に理解できない．また，第 2.3 で説明したような PEG の特徴から発生するバグを発見することも難しい．

そこで，本論文ではデバッガによる動的な文法開発支援を行うアプローチを提案する．デバッガは，開発者が実際に解析の処理をトレースできるので，開発者

表 2　基本デバッグイベント

イベント	処理の説明
Run	デバッガの実行
Exit	デバッガの終了
BreakPoint	ブレークポイントの設置
StepOver	次の解析表現へ進む
StepIn	次の解析表現へ進む (非終端記号に入る)
StepOut	現在のルールを抜ける
Continue	次のブレークポイントまで進む
PrintPos	現在の解析位置を表示
PrintNode	現在の解析結果の AST を表示
PrintRule	指定したルールを表示
BackTrace	スタックトレースの表示

がパーサの動作を理解することができる．

我々は以下の内容を実現する文法デバッガを設計した．

- PEG パーサの動作を理解できる
- PEG 特有の文法のバグを発見できる
- 伝統的なデバッガの機能を備えている

これらの設計思想は，開発者がより文法開発をしやすい環境を提供することにつながる．

3.2　デバッグイベント

デバッグイベントは，開発者がデバッガに送信するイベントである．これらは，開発者が直感的に使うことができ，かつ PEG の文法に特化した操作を行えることが求められる．そのため，伝統的なデバッガの機能と，PEG の文法をデバッグすることに特化した機能を共存させる形で，デバッグイベントを設計した．表 2，表 3 はデバッガがサポートするデバッグイベントの一覧である．

3.3　到達不能な選択の検知

第 2.3.2 節で述べた通り，PEG は優先度付き選択を採用しているので，到達不能な選択が発生する可能性がある．しかし，これを静的な文法解析によって評価することは難しい．本論文では，動的に到達不能な選択の評価を行う手法を選択する．

この手法では，先の選択で受理した入力を，後に定義された選択でも受理できた時にその選択が到達不能な選択であると認識する．よって，開発者が入力した入力において，その文法が到達不能な選択を有しているのかを判定することができる．

表 3　PEG デバッグイベント

イベント	処理の説明
PrintPackrat	メモ化の情報を表示
StartRule	開始ルールを指定
StartPosition	開始時の解析位置を指定
Consume	指定した解析位置まで解析を進める
Goto	指定したルールまで解析を進める

この手法の限界は，開発者が入力した入力に関して到達可能であるかを言うことはできるが，一般的にその選択が到達可能であるという証明はできない点である．しかし，今回は PEG の文法のバグを発見，修正する事が目的である．この手法は，入力をこの時の入力に限定する事で，到達不能な選択を発見する事ができる．よって，入力の限定されるデバッグに対する用途としては，本手法は十分に効果を発揮する．

3.4　コードレイアウトのバグに対する支援

コードレイアウトは PEG の文法開発の難所の 1 つである．我々はコードレイアウトの問題によって発生する文法のバグに対する支援を行うために，新たな操作を加える．我々はこの新たな操作を FailOver と呼ぶことにした．

FailOver は，スペース，改行文字が入力であり，それを受理できなかった時，一度そのスペース，改行文字を無視して解析を続ける．スペースや改行を飛ばして入力を受理できたのであれば，それはコードレイアウトがバグの原因となっていると分かり，先ほど無視した箇所をバグとして認識することができる．

4　実装

本節では，PEG デバッガの実装について述べる．

4.1　構成

PEG デバッガは，PEG の文法開発を動的に支援するためのデバッグ支援ツールである．PEG デバッガ

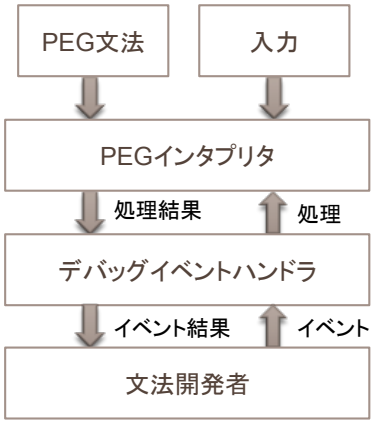

図 4　デバッガの構成

はの全ての機能は Java で記述されており，Nez パーサライブラリ [5] の一部として実装されている．Nez パーサライブラリでは，構文木の構築や，シンボルテーブルを用いた文脈依存な入力の解析を行うための記法を備えている．

図 4 に PEG デバッガの構成を示している．PEG デバッガは，PEG の解析を行う PEG インタプリタ上で動作し，デバッグイベントハンドラによってユーザが指定したデバッグ処理を実行する．入力を受理しながら，その解析の様子を開発者にフィードバックする．

4.2　PEG インタプリタ

PEG デバッガは，パーサジェネレータではなく，独自のインタプリタ上で動作する．本論文では，PEG インタプリタを仮想マシンインタプリタとして実装した．PEG の文法は，仮想マシン命令列にコンパイルされ，仮想マシンインタプリタ上で動作する．また，PEG インタプリタは Nez パーサライブラリでサポートしている拡張構文にも対応しており，それぞれの構文は仮想マシン命令にコンパイルされる．図 5 に仮想マシンのソースコードから一部抜粋したものを示す．仮想マシンは，各命令で対応する処理を行った後，その返り値として次に実行する命令を返し，それを繰り返すことで実行される．例えば，Ichar は 1 文字のマッチングを行う命令であり，マッチに成功した時は解析位置を進め，失敗した時は結果を false とする．

```
class Machine{
    String inputs;
    long pos;
    boolean result;
    StackEntry[] stack;
    MemoTable memoTable;

    Instruction Ichar(Ichar pc) {
      if (pc.c == inputs[pos]) {
        pos++;
      } else {
        result = false;
      }
      return pc.next;
    }

    Instruction Iiffail(Ichar pc) {
      if (!result) {
        return pc.jump;
      }
      return pc.next;
    }
}
```

図 5　仮装マシンのソースコード (抜粋)

第 3.3 節，第 3.4 節で述べた手法は，特別な仮想マシン命令として実装されている．これらは，後述するデバッグイベントハンドラと連携して，検知された場合に解析動作を一時停止する．これによって，デバッグ困難なバグの位置を特定し，そこからデバッグを再開することができる．

4.3　デバッグイベントハンドラ

デバッグイベントハンドラは，ユーザが指定したデバッグイベントを実行するための処理機構であり，第 3.2 節で挙げたイベントをサポートしている．デバッグイベントハンドラに送られてきたデバッグイベントの情報をもとに，PEG インタプリタを動作させる．これにより，解析中の情報を開発者にフィードバックすることができる．

5　評価

本節では，我々が開発を行ってきた文法の中で特に修正が困難であったバグについて，PEG デバッガによるデバッグが有効であったことを報告する．

我々が開発した文法に，どこかで無限ループが発生

し，解析が終了しないものがあった．このバグは解析が終了しないために，どのルールで発生したバグか判別できなかった．そこで，PEG デバッガを用いてステップ実行で解析をトレースしたところ，間接的な左再帰が生まれている箇所を発見した．これは，EBNF から PEG に手作業で変換する際に，左再帰を消しきれなかったために発生したバグであった．このような，結果が出力されない困難なバグに対して，デバッガは有効に作用した．

次に，到達不能な選択が原因で発生したバグ修正について報告する．このバグ修正の困難さは，文法エラーの情報が正しく出力されていないことにあった．そのため，エラー報告で得られた情報からの修正は困難であった．そこで，本論文で実装した到達不能な選択の検出機能使用したところ，バグの箇所を特定することができた．その周辺をステップ実行でデバッグすることで，バグを修正を行った．

最後に，コードレイアウトのバグ修正について報告する．EBNF から PEG へ手作業で変換する際に，空白のルールを入れ忘れるという事態がしばしば発生した．このバグは一見単純なバグに見えて修正が難しい．これは，第 2.2 節で述べた通りである．特に，Python のようなインデントベースのブロックを持つ言語の解析では，改行を許容する空白と，許容しない空白を分けて定義しなくてはならず，定義のミスが多く出た．これらのバグを，PEG デバッガの機能である FailOver とステップ実行を組み合わせることで修正することができた．

6 関連研究

プログラムのデバッグを行うためにデバッガを用いることは多々あるが，文法開発にこれを適用した例は少ない．ANTLRWorks [1] は文法デバッガを有する貴重な環境である．ANTLRWorks は，ANTLR [6] に特化した文法開発環境である．主な機能として，リファクタリング機能付きのエディタ，文法インタプリタ，シンタックスや Lookahead DFA のビジュアライザ，文法デバッガを有する．ANTLRWorks の文法デバッガは，ブレークポイントの設置やステップ実行を行うことが可能である．これらの伝統的なデバッガの機能と，解析処理のビジュアライズによってデバッグ支援を実現している．

7 結論

PEG はパーサジェネレータで用いられる形式文法であるが，その文法開発は固有の困難を持つ．我々は，PEG の文法開発の支援を行うために，新たなアプローチとして PEG デバッガを提案した．PEG デバッガはブレークポイントや，ステップ実行，スタックトレース機能などがある．さらに，PEG の特徴から発生する文法のバグに対して支援を行う機能を持つ．PEG デバッガは，これまで困難であった文法のデバッグに役立った．

謝辞 本論文の初期の版について議論していただいた須藤建氏 (横浜国立大学)，田村健介氏 (横浜国立大学)，山口真弥氏 (横浜国立大学)，佐藤正典氏 (横浜国立大学) に感謝する．

参 考 文 献

[1] Bovet, J. and Parr, T.: ANTLRWorks: An ANTLR Grammar Development Environment, *Softw. Pract. Exper.*, Vol. 38, No. 12(2008), pp. 1305–1332.

[2] Ford, B.: Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, New York, NY, USA, ACM, 2002, pp. 36–47.

[3] Ford, B.: Packrat Parsing:a Practical Linear-Time Algorithm with Backtracking, Master's thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2002.

[4] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, New York, NY, USA, ACM, 2004, pp. 111–122.

[5] Kuramitsu, K.: Nez - Declrative Parsing Expression Grammar Toolkit. http://peg-nez.github.io/.

[6] Parr, T. and Fisher, K.: LL(*): The Foundation of the ANTLR Parser Generator, *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, New York, NY, USA, ACM, 2011, pp. 425–436.