不完全な文法からパーサを生成する手法の検討

奥田 勝己　千葉 滋

本論文では，一部を省略した不完全な文法からパーサを生成する手法を提案する．プログラミング言語を対象とするリファクタリングツールやソフトウェア解析ツールでは，特定のテキスト構造のみを抽出できれば十分であり，言語処理系が必要とする完全な構文木を必要としない．このため，これらの用途では特定のテキスト構造を grep のように手軽に抽出できると便利である．しかし，grep が使用する正規表現では再帰を扱うことができないため，プログラミング言語を対象とする場合，抽出可能なテキスト構造は限定的である．そこで，提案手法では，PEG（Parsing Expression Grammar）を拡張し部分的な省略を意味するワイルドカード記号を導入することで，必要とするテキスト構造のみを含む文法定義を可能とする．また，ワイルドカード記号の定義を通常の非終端記号の定義へと変換することで，不完全な文法からのパーサ生成を実現する．

This work-in-progress paper proposes a novel method that generates a parser from an incomplete grammar definition. Refactoring tools or software analysis tools do not require a fully detailed parse tree that is required by compilers or interpreters. Instead, they require only sub-parts of the parse tree. Hence, it is useful if we could extract these sub-parts of the parse tree by using grep-like tools. However, various kinds of lexical structures of source programs cannot be extracted by grep since it accepts a regular expression, which cannot handle recursive structures. To ease the difficulty in parser construction for programming tools, we extended PEG (Parsing Expression Grammar) to support a wildcard symbol. The wildcard symbol represents partial omission of PEG rules. We generate a parser by converting our extended PEG into conventional PEG.

1　はじめに

ソフトウェア開発の現場では，ソフトウェアのコード解析やリファクタリング，ソフトウェアメトリクスの計測など，特定の式や文の検索や書き換えが必要な場合がしばしばある．これらのタスクを人手で行うためには労力を要するが，汎用的なツールが存在しないため，プログラマ自身がちょっとした専用ツールを作成する必要がある場面は多い．専用ツールの開発には正規表現を使うことができるスクリプト言語が有力なアプローチである．しかし，正規表現では再帰的な構造を扱うことができないため，これらのタスクで適用できるケースは限定的である．たとえば，プログラムの比較的上位構造である式や文は再帰的に式や文を含むが，純粋な正規表現ではこれらを表現することができない．また，別のアプローチとしては，言語処理系を開発する場合と同様に LL や LR のパーサを生成することが考えられる．しかし，LL や LR のパーサ生成系は，プログラミング言語の完全な文法を必要とする．パーサを生成可能な文法定義は LL や LR などの構文解析アルゴリズムごとに異なるため，パーサを生成するための完全な文法を作成するには多大な労力が必要である．また，C/C++や Java のように複雑な文法を持つ実用言語が対象の場合，ベンダ拡張や最新の構文に対応した文法をオープンソースで入手することも困難である．

そこで，本論文では元の文法の一部を省略した文法を定義可能とし，省略した文法からパーサを生成する手法を提案する．提案手法では，PEG（Parsing

*Parser generation with an imcomplete grammar

Katsumi Okuda, 東京大学/三菱電機株式会社, The University of Tokyo/Mitsubishi Electric Corporation.
Shigeru Chiba, 東京大学, The University of Tokyo.

Expression Grammar）[1] に対し，規則の一部を省略した非終端記号を意味する新たなワイルドカード記号を導入する．ユーザは，ワイルドカード記号の完全な定義を与えることなく，ワイルドカード記号を他の非終端記号の定義に使用することができる．また，本手法では，ワイルドカード記号を含む PEG による文法を従来の PEG の文法に変換することで，パーサの生成を実現する．

提案手法を用いることで，ユーザは従来のパーサジェネレータのための完全な文法を作成するよりも少ない労力で文法を作成することができる．したがって，提案手法は小さなソフトウェアツール向けパーサの開発に適している．提案手法は，入力ファイルの文法エラー検出などを想定した手法ではなく，プログラミング言語のテキスト構造を抽出するための新しい `grep` ツールとしても位置付けることができる．

2 不完全な文法の定義

計算機のユーザは，テキストファイルを対象として検索や置換を行う場合，grep や sed などのツールを用いることができる．grep や sed は，対象テキストと抽出対象となるテキスト構造の文法を入力とし，入力テキストから文法にマッチするテキスト構造を抽出する．このとき文法は正規表現で表される．たとえば，ユーザは，ログファイルから特定の日時の項目を抽出することや，WEB ページから URL を抽出するといったことを，grep に日時や URL のパターンを与えることで行うことができる．

sed や grep による検索が便利な点は，抽出対象のテキストの文法を細部まで指定する必要がないことである．正規表現では，任意の文字にマッチするワイルドカードを使うことができる．したがって，ユーザはテキスト構造の文法の詳細を指定することなくその特徴的な部分のみを詳細化し，それ以外の部分にはワイルドカードや最短一致記号を用いることで，テキスト構造のパターンを記述することができる．たとえば，HTML の BODY に対応するパターンは，正規表現では`/<body>.*?</body>/`として記述することができる．このとき，BODY 中の細かい文法構造は指定する必要がない．

文法の細部まで指定不要な grep や sed の特徴は様々なテキストを対象として有益であり，対象がプログラミング言語の場合も例外ではない．このため，ソフトウェア開発の現場でも grep や sed に類するツールが用いられる場面は多い．しかし，プログラミング言語が対象の場合，その利用は限定的である．これは，プログラミング言語が一般に文脈自由言語であり，抽出対象のテキスト構造を正規表現で記述できないことがあるためである．

プログラミング言語を対象とする場合，正規表現では扱えないテキスト構造の検索や書き換えを行いたい場面は多い．ここでは例としてコーディング規約に違反しているコードを書き直す場合を考える．Java では，if 文の条件成立/不成立時に実行すべき文が 1 文の場合，波括弧を省略することができる．しかし，多くのコーディング規約では保守性の低下を防止することを目的としてこれを禁止している．

たとえば，下記のようなプログラムを対象とする場合，

```
1  double calcDailyWages(double hours, double
       hourlyWage)
2  {
3    double result = 0;
4
5    if (hours <= 8)
6      result = hourlyWage * hours;
7    else if (hours <= 12) {
8      double overtime = hours - 8;
9      result = hourlyWage * 8 + (1.5 * hourlyWage)
           * overtime;
10   }
11   else
12     warning();
13
14   return result;
15 }
```

プログラマは 6 行目と 12 行目の文の前後に波括弧を入れる必要がある．このようなソースコードの検索や書き換えは人手で行うと誤りがちであるため，専用のリファクタリングツールの作成が必要である．このリファクタリングツールの場合，if 文を検索することが必要である．しかし，if 文は別の if 文を入れ子にできるため，正規表現ではパターンを記述することができない．

このような場合，正規表現ではなく LL(k) や LR(k) など文脈自由文法のサブセットを扱うことができる

パーサジェネレータを用いることが考えらえる．しかし，既存のパーサジェネレータは，対象言語全体の完全な文法を必要とするため，些細な書き換えのためだけに Java のパーサを開発することは現実的ではない．また，Java のパーサはインターネットから入手可能な場合もあるが，それをツールの記述言語から使用できるとは限らない．

正規表現より強力で既存のパーサジェネレータよりも容易に扱うことができるアプローチはあるだろうか．リファクタリングツールやソフトウェア解析ツールは，コンパイラが必要とするような完全に詳細化された構文木や AST は不要である．これらのツールの場合，完全な構文木のうちツールにとって不要なサブツリーを省略した構文木が得られれば十分である．このような構文木を得るには，完全な文法は不要なはずである．

3 PEG の拡張

3.1 概要

提案手法では，PEG を拡張し，省略部分に対応するワイルドカード記号を導入することで，一部を省略した文法の定義を可能とする．さらに提案手法では，拡張した PEG を通常の PEG に変換することで，対象言語から構文木を作成可能なパーサを生成する．本論文で説明に使用する PEG の演算子は，表 1 のとおりである．

表 1　PEG 式の演算子一覧

演算子	優先度	概要
.	5	任意文字
(e)	5	グルーピング
$e?$	4	オプショナル
$e*$	4	0 回以上の繰り返し
$!e$	3	not predicate
e_1e_2	2	連接
e_1/e_2	1	ordered choice

新たに導入するワイルドカード記号は，非終端記号の拡張であり，文法中で非終端記号として扱うことができる．本論文では，<で始まり>で終わる非終端記号をワイルドカード記号とする．ワイルドカード記号は，完全な文法のうち省略対象の非終端記号として考えることができる．ユーザは，ワイルドカード記号を非終端記号と同様に定義することで，その一部を詳細化することも可能である．

ワイルドカード記号を用いると簡略化した Java の if 文の抽出に必要な非終端記号の大部分は，下記のように定義することができる．

```
1  block <− '{' stmt* '}'
2  stmt <− if_else_stmt / if_stmt / exp_stmt / block
3  exp_stmt <− <expr> ';'
4  if_else_stmt <− = if_stmt else_clause
5  if_stmt <− 'if' '(' <expr> ')' stmt
6  else_clause <− 'else' stmt
```

3 行目と 5 行目の規則の右辺で使用されている<expr>は，ワイルドカード記号であり，; または) 以外の文字の列にマッチする．

ここで，<expr>でマッチする文字列が) を含まないことは構文解析時に問題となる．<expr>が対応する式の中には，丸括弧で囲まれた別の式が入れ子になることがあるが，<expr>でマッチする文字列が) を含まない場合，パーサが正しく式に対応する文字列を消費することができない．そこで，提案手法では，ワイルドカード記号<expr>を下記規則のように部分的に詳細化することができるようにする．

```
1  <expr> <− '(' <expr> ')'
```

本規則は，ワイルドカード記号がマッチする文字列の中に右辺で示した PEG 式がマッチするサブ文字列が存在しても良いことを意味する．なお，ワイルドカード記号は，従来の非終端記号と異なり，規則の右辺で示した PEG 式にのみ一致するとは限らない点に注意が必要である．

プログラム全体をパースし，if 文を抽出するには，以下の規則を文法に追加し，compilation_unit を開始記号とすることで if 文抽出のための文法を定義することができる．

```
1  compilation_unit <− <program> EOF
2  <program> <− stmt
```

ここでは，プログラム全体に対応する<program>の中

を先述の `stmt` を用いて部分的に詳細化している．

3.2 PEG の変換

本節では，提案手法による PEG の変換について示す．ワイルドカード記号を通常の PEG に変換することで，不完全な文法から対象言語を構文解析可能な Packrat パーサ [2] を生成することができる．

提案手法では，ワイルドカード記号の定義から通常の非終端記号の定義への変換を下記の変換規則で行う．X をワイルドカード記号，e を任意の PEG 式とするとき，提案手法ではワイルドカード記号の下記定義

$$X \leftarrow e$$

を

$$X \leftarrow ((e)/!(sep(X)).)*$$

のように変換する．ここで，$sep(X)$ は X の Follow 集合の記号を PEG の ordered choice 演算子で繋げた PEG の式である．X の Follow 集合である $Follow(X)$ は，非終端記号 X の後に来る可能性がある記号の集合であり，LL パーサや LR パーサの生成で利用される Follow 集合と概ね等しい．ただし，従来の Follow 集合が非終端記号を要素とするのに対し，提案手法の Follow 集合は対象言語のトークンと対応した PEG の終端記号または非終端記号である．これは，LL パーサなどの生成に利用する文法がトークンを終端記号としているのに対し，スキャナレスを前提とする PEG の場合，終端記号は文字や記号であり，複数文字から構成されるトークンは非終端記号で定義されるためである．

例として前節の`<expr>`は，下記のように変換される．

```
1  <expr> <- ((('(' <expr>')') / !(')' / ';') .)*
```

ここで，$sep($`<expr>`$)$ は，集合 $Follow($`<expr>`$)$ の要素`')'` と`';'` を ordered choice 演算子で連結した PEG 式となっている．変換後の`<expr>`は，丸括弧で囲まれた`<expr>`，または閉じ丸括弧とセミコロン以外の任意文字の 0 個以上の繰り返しとして定義される．

3.3 Follow 集合の計算

提案手法では，PEG の変換に Follow 集合を用いる．これまでの例では，$Follow($`<expr>`$) = \{$`')', ';'`$\}$ のように文法規則で`<expr>`の後ろに現れる記号のみである．以下では，Follow 集合の計算が複雑となるケースを示し，提案手法における Follow 集合の計算手法について示す．

ここでは，先ほど下記のようにワイルドカード記号として定義した式を通常の非終端記号として再定義し，代入式を扱えるまで詳細化する場合を考える．

```
1  <expr> <- '(' <expr> ')'
```

代入式まで扱えるようにする場合，式は通常の非終端記号 `expr` として下記のように定義される．

```
1  expr <- assign_expr
2  assign_expr <- <cond_expr> ('=' <cond_expr>)*
3  <cond_epx> <- '(' expr ')'
```

上記の定義では，代入式より下位のテキスト構造を省略するため，新たにワイルドカード記号`<cond_expr>`を導入している．

ここで，$Follow($`<cond_expr>`$)$ には，上記 2 行目の規則から`=`が含まれることがわかる．しかし，$Follow($`<cond_expr>`$)$ には `exp` および `assign_expr` の Follow 集合である`;` と`)` も含まれる．このような計算は，従来の Follow 集合と同様に First 集合を用いた不動点計算によって行うことができる．

従来の First 集合および Follow 集合の計算と提案手法による計算の相違は，従来の First 集合および Follow 集合を非終端記号ごとに計算するのに対し，提案手法では PEG の式ごとに計算する点である．PEG の式は再帰的に部分式を含むため，ツリーとして表現することができる．提案手法ではツリーのリーフおよびノードごとに First 集合と Follow 集合を計算する．たとえば，先述の $Follow($`<cond_expr>`$)$ の計算では，PEG の式 `('=' <cond_expr>)*` に対する First 集合が必要である．同様に式 `('=' <cond_expr>)*`の First 集合の計算には，式`'=' <cond_expr>`の First 集合が必要である．

4 関連研究

これまでに Yacc や ANTLR など多数のパーサ生成系が開発されている．パーサ生成系は，対象言語の文法定義を入力とし，パーサを生成する．従来のパーサ生成系は，対象言語の完全な文法を必要とする．このため，言語の規模に比例して文法記述に要する労力も増加する．また，パーサ生成アルゴリズムごとに入力可能な文法は異なるため，パーサ生成系に完全な文法を与えることは容易でない．

ANTLRWorks [3] では，GUI を提供することで，ユーザによる文法記述を支援する．ユーザは，ANTLRWorks を用いることで，作成中の文法が入力ソースプログラムを期待通りに構文解析できない理由を理解できる．

Cocke-Younger-Kasami （CYK） [4]，GLL [5]，GLR [6] などの CFG（Context Free Grammar）パーサ生成系を用いることで，ユーザは任意の CFG で文法記述を行うことができる．このため，ユーザは，shift/reduce 衝突や左再帰などを気にする必要がない．しかし，CFG パーサの出力は構文木ではなく，構文森である．このため，ユーザは構文森から構文木を抽出する手段を別に用意する必要がある．

5 まとめ

本論文では，対象言語の完全な文法を必要とすることなく，部分的に詳細化した文法から構文木を作成可能なパーサを生成する手法の概要を示した．提案手法では，PEG を拡張し部分的な省略を意味するワイルドカード記号を導入することで，必要とするテキスト構造のみを含む文法定義を可能とする．また，ワイルドカード記号の定義を通常の非終端記号の定義へと変換することで，不完全な文法からのパーサ生成を実現する．アルゴリズムの詳細化，形式化，効果の評価が今後の課題である．

参 考 文 献

[1] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.

[2] Bryan Ford. Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47, New York, NY, USA, 2002. ACM.

[3] Jean Bovet and Terence Parr. Antlrworks: An antlr grammar development environment. *Softw. Pract. Exper.*, 38(12):1305–1332, October 2008.

[4] Daniel H. Younger. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189 – 208, 1967.

[5] Elizabeth Scott and Adrian Johnstone. Gll parsing. *Electron. Notes Theor. Comput. Sci.*, 253(7):177–189, September 2010.

[6] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.