Packrat Parsingを用いたRubyの構文解析

山口大貴† 前田敦司‡ 山口喜教‡

†筑波大学第三学群情報学類

‡筑波大学大学院システム情報工学研究科

1 はじめに

近年 Web アプリケーション開発などの分野において Ruby[1] などのスクリプト言語が急速に普及してきている．しかし多くのスクリプト言語同様，Ruby は記述性の向上のために C や Java といった言語と比較して文法が複雑である．そのため，構文解析を行う際に現在主流の bison と lex を組み合わせる手法では文法の記述が複雑化し，メンテナンスが困難となってしまう．例えば Ruby では，文字列リテラルに任意の式を埋め込み可能な構文に対応するため等の理由で，約1800 行の字句解析器を手書きしている．これは従来の構文解析アルゴリズムが字句解析をベースとしているために，字句解析器に状態を付加し，構文解析器から字句解析器の状態を変更する等の手法を用いなければ構文を解析できないためである．そこで本研究では Parsing Expression Grammar(PEG)[2] をベースとした構文解析アルゴリズムである Packrat Parsing[3] を用いることで Ruby のこのような問題を解消し，保守性を向上させることを目的とする．

2 Parsing Expression Grammar

PEG の文法は BNF に類似した記法を用いて表現され，その構文規則は $N \leftarrow e$ という形を取る．ここで N は非終端記号，e は Parsing Expression という式を表している（表 1）．PEG における選択には文脈自由文法と違い，優先度が存在する．つまり e_1/e_2 は単に e_1 もしくは e_2 のどちらかにマッチするという意味ではなく，最初に e_1 にマッチするかどうか試して，失敗したら Backtrack して e_2 にマッチするか試すという動作を表している．&e は入力が式 e とマッチするときに成功するが入力を消費せず，同じ入力位置から解析を続ける．また，!e は入力が式 e とマッチしないときに成功し，これもまた入力を消費しない．

表 1 Parsing Expression の主要な構成要素

"s"	文字列リテラル
N	非終端記号
e_1 e_2	式の列
e_1 / e_2	(優先度付)選択
e*	繰り返し(0回以上)
e+	繰り返し(1回以上)
e?	省略可能
&e	AND-predicate
!e	NOT-predicate

3 Packrat Parsing

Packrat Parsing は Backtrack Recursive Descent Parsing (Backtrack Parsing) に対してメモ化を追加した構文解析アルゴリズムである．Backtrack Parsing では構文規則を解析対象の文字列を引数に取る解析関数とみなし，構文規則に複数の選択肢があった場合は最初の選択肢を試して，成功した場合はその結果を返し，失敗した場合は Backtrack して次の選択肢を試すという動作を行う．Packrat Parsing では Backtrack Parsing に加え，各部分文字列に対する解析結果をメモ化しておき，以前呼び出された部分文字列に対して同じ解析関数が呼び出された場合にはメモ化しておいた結果を返す．メモ化により，Backtrack Parsing では入力サイズに対して解析に最悪指数関数時間かかるものを，Packrat Parsing では線形時間で解析を行うことができる．また，Packrat Parsing のアルゴリズムは無制限の先読みが可能であるため，あらかじめ字句解析器を用いてシンボルをトークンにまとめたり，コメントや空白を入力から取り除いたりしておく必要がない．そのため，構文レベルの処理と字句レベルの処理を統合して扱うことができ，前述の式を埋め込み可能な文字列リテラルといったようなものも比較的簡単に解析することができる．

4 実装方針

本研究では *Rats!*[4] を使用する．*Rats!* は，PEG に基づく構文定義を入力として与えるとJava による Packrat Parser を生成する Parser

A Packrat Parser for Ruby.
Daiki Yamaguchi†, Atusi Maeda‡, Yoshinori Yamaguchi‡
†College of Information Sciences, Third Cluster of Colleges, University of Tsukuba
‡Graduate School of Systems and Information Engineering, University of Tsukuba

生成系である．Ruby の文法を表現する構文定義ファイルを PEG を用いて作成し，それを *Rats!*の入力として与える．構文定義の作成にあたり，まず従来の字句解析器に相当する処理を行う部分から作成する．具体的には，図 1 のようなプログラムから図 2 のような PEG が作成できる．その際，AND(NOT)-predicate を適切に使用することでトークンをより正しく判別することができる．

```
c = read();
switch(c){
  case '>'
    c = read();
    switch(c){
      case '=' : return Tokens.tGEQ;
      case '>' :
        if((c = src.read()) == '=' ){
          return Tokens.tOP_ASGN;
        }
        unread();
        return Tokens.tRSHFT;
      default:
        unread();
        return Tokens.tGT;
    }
}
```

図 1　lexer の構造例

```
tGEQ      = ">="
tOP_ASGN  = ">>="
tRSHFT    = ">>"  ! "="
tGT       = ">"   !( "=" / ">" )
```

図 2　PEG の例

次に構文定義を bison の入力形式から PEG へと書き換える．書き換えの際は，前述した選択の順序，元は字句解析器で行っていた操作，再帰的な構文などに注意する．元は字句解析器で行っていた操作とは，空白(改行)の読み飛ばしや文字列リテラルの処理等である．Packrat Parsing には字句解析器の概念が無いため，このような操作は構文規則に埋め込むことで対応する．

5　評価・考察

Ⅰ．従来の字句解析器
Ⅱ．Ⅰを PEG に書き換えた構文定義ファイル
Ⅲ．Ⅱから生成した Packrat Parser
これらのコード行数を計測した．*Rats!* のバージョンは 1.14.3 である．

表 2 コード行数

	Ⅰ	Ⅱ	Ⅲ
行数(行)	170	1774	4666

Ruby の字句解析器は約 1800 行にも及び，更に状態付きであるため，一般的にその処理内容を把握することは難しい．対して提案手法では，構文定義ファイルとして PEG を用いているため同様の機能をおよそ 1/10 程の行数で表現できる．生成された Parser の行数は従来の字句解析器に比べ約 2.6 倍大きいが，提案手法では文法の把握に Parser 自体を読む必要が無いことや現在の計算機の性能・容量等を考慮すれば，行数の増加は大きな問題にならないと考えられる．また，提案手法では構文定義の中にトークン情報も埋め込めるため，従来の字句解析器と構文解析器を分離する手法よりも文法の把握・変更が容易になる．これらにより，可読性や保守性の向上が見込める．

6　今後の課題

現在実装済みの構文定義は，Ruby 文法の一部には対応しているものの，解析できない構文が多い．そのため，今後の課題としてはより多くの文法に対応することが挙げられる．

7　おわりに

本研究では，Packrat Parsing を用いた Ruby の構文解析について述べた．Ruby の文法を PEG に書き換えて Packrat Parser を生成することで，構文定義ファイルの可読性や保守性の向上が見込めることを示した．今後の課題は，より多くの文法に対応する PEG を作成することである．

参考文献

[1] まつもとゆきひろほか: オブジェクト指向スクリプト言語 Ruby. http://www.ruby-lang.org/ja/

[2] Bryan Ford, Parsing expression grammars: a recognition-based syntactic foundation. Proc. POPL '04, pp.111-122, 2004.

[3] Bryan Ford, Packrat parsing: simple, powerful, lazy, linear time, functional pearl. Proc. ICFP '02, pp.36-47, 2002.

[4] Robert Grimm, Better extensibility through modular syntax. Proc. PLDI '06, pp.38-51, 2006