4J-1

自己拡張可能な構文解析器生成系

舞田 純一 †　　中井央 ††
筑波大学　　筑波大学

1. 序　　論

近年、オブジェクト指向言語が広く使われるようになったが、構文解析器生成系にはオブジェクト指向に基づく機能付加を考慮したものは依然少ない。そのため、生成された構文解析器に対する機能付加は困難であった。この観点から[1] では、構文解析器に対する Decorator パターン[2] による機能付加及びそのための生成系について報告されている。

しかし、この方法では構文解析器の拡張性は生成系の持つ能力 (記述力) に依存し限定される。例えば、構文解析器に対し構文木作成の機能付加を行うには、生成系に入力記述から構文木作成に関連する情報を読み取り、構文木の作成に対応するコードを生成する機能が必要となる。そのため、[1] では生成系自体が多数の機能を抱えていた。

本研究では[1] を基に、さらに汎用的な機能付加のための生成系自体の機能拡張について提案する。また、その生成系を Ruby[3] を用いて実現した。

2. システムの概要

2.1 入 力 記 述

本システムへ与える入力記述は、宣言部、文法部、拡張部から構成される。宣言部は入力記述の始まりから最初の%%までである。宣言部には、生成されるクラス名や利用する拡張機能についての情報を記述する。その直後から次の%%までが文法部記述と拡張部記述である。文法部には Yacc[4] 風の BNF を記述する。拡張部は文法部中にあり拡張機能を用いる記述を記述する。

```
%class TinyCalc
%extend Action ('action.rb')
%%
  expr :
      expr '+' term {val[0] + val[1]}
    | expr '-' term {val[0] - val[1]}
  ;
  ...
%%
```

図 1　入力記述例

図 1 に、還元時にアクションを行う拡張を用いる入力記述の例を示す。図 1 の 2 行目が使用する拡張機能の指定で、これにより各生成規則中各右辺の最後に{と}で囲んだ拡張部の記述が可能になる。5、6 行目の{ …}が、この拡張機能を用いた拡張部の記述である。

2.2 生成系と拡張

本システムは、入力記述中の宣言部、文法部を処理する「生成系」と生成系の機能を拡張する「拡張」から構成される。

生成系は、LALR(1) 構文解析器を生成する機能及び入力記述に基づき自身を動的に拡張する機能のみを有する。生成される構文解析器は[1] に基づいて Decorator パターンによる機能付加が可能となっている。

拡張は一般に字句・構文解析器を持ち、入力記述中の拡張記述部を処理する。拡張は生成系と関連する以下の操作を行うことが出来る。

- 処理中の構文規則に関する情報の取得、更新
- 生成される構文解析器の情報の取得、更新
- 生成されるコードへの任意の文字列の追加

本システムでは、拡張 (及び対応する Decorator) として還元時アクション機能、字句解析器生成系、状態

An Extensible Compiler Generator
† Junichi Maita・University of Tsukuba
†† Hisashi Nakai・University of Tsukuba

付き字句解析器生成系、解析木作成機能、構文木作成機能、巡回法の異なるもう一つの構文木作成機能を用意しており、これらは自由に組み合わせることができる。また、本システムを用いて容易に新たな拡張を作成することができる。

2.3 拡張の仕組み

生成系は入力記述の宣言部を処理する宣言処理部、文法部を処理する文法処理部から構成される。

宣言処理部は宣言部から、%extend 節に記されている利用する拡張についての情報を収集する。そして、その情報を基に拡張をファイルから読み込み、そのインスタンスを作成する。文法処理部には拡張に処理を移すタイミング (文法全体の始まり、各右辺の終わり等) 毎にフックというリストがあり、宣言処理部は作成した拡張に対し拡張自身をフックに登録させる。

文法処理部は文法部を処理しながら、規定のタイミング毎にフックに登録された拡張に制御を移す。拡張は文法処理部から入力記述を引き継ぎ、拡張部を処理する。拡張部の処理が終わると文法処理部に制御が戻り、再び文法部の処理を行う。

3. 適 用 例

3.1 字句解析器生成系拡張

字句解析器生成系拡張を用いることで、文法中に字句解析記述を記述できるようになる。この拡張は、記述から字句解析メソッドを実装したコードを生成する。図 2 では、%LEX{ … %}がこの拡張を用いた記述である。

```
%extend Lexer ('lex.rb')
%%
%LEX{
  /[ \t]+/  { }
  /[a-z_]+/ { yield :IDENT, $& }
  /./       { yield $&, $& }
%}
```

図 2 字句解析器生成系拡張の例

3.2 構文木作成拡張

構文木作成拡張は、文法中に構文木ノードの定義と構文規則と作成される構文木ノードの対応付けを記述できるようにする。この拡張は記述から、Composite パターン[2] と Visitor パターン[2] に基づいて、複数の構文木ノードクラスと構文木を巡回する Visitor クラスを実装したコードを生成する。図 3 では、%AST{ … %}が構文木ノード (及び Visitor) の定義部であり、各構文規則の後の=>…が構文規則と作成される構文木ノードの対応付けである。

```
%extend ASTBuilder ('ast.rb')
%%
%AST{
  add(e0,e1) { ~val=~e0.val+~e1.val }
  sub(e0,e1) { ~val=~e0.val-~e1.val }
%}
  expr:
      expr '+' term
      =>add(expr, term)
    | expr '-' term
      =>sub(expr, term)
  ;
```

図 3 構文木作成拡張の例

4. 結　　論

本研究ではオブジェクト指向に基づいて構文解析器を構成する方法に対し、その生成系をどのように構築すべきかと言う観点から、生成系自体を拡張可能にする方法について考察し、実装を行なった。

今後の課題としては、より実用的な拡張の作成や現在は Ruby に限定されている出力コードの Java や C#等の他言語への対応等がある。

参 考 文 献

1) 佐竹力, 中井央. オブジェクト指向に基づいた構文解析器構成法の提案. 情報処理学会論文誌：プログラミング, Vol. 45, No. SIG.12 (PRO 23), pp. 25–38, 2004.
2) Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
3) まつもとゆきひろ, 石塚圭樹. オブジェクト指向スクリプト言語 Ruby. アスキー, 1999.
4) Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, Vol. 2, pp. 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report July 31, 1978.