Vol. 47 No. SIG 16(PRO 31) 情報処理学会論文誌：プログラミング Oct. 2006

機能拡張可能なコンパイラ生成系

舞田純一[†] 佐藤聡[†] 中井央[†]

計算機の多種多様な用途に応じて，それぞれの領域に適したプログラミング言語およびその処理系が開発されることも多く見受けられるようになってきた．このため，そのような開発を迅速かつ正確に行えるよう，コンパイラ生成系の研究は重要性を増している．このような目的に対応するため，我々はこれまでに純粋な構文解析処理とそれに対する機能付加をデコレータパターンに則って構成する方法を提案し，その生成系を試作した．付加する機能はユーザにより自由に定義できるが，生成系としてそのような機能の付加を行えるようにするには，用途に応じて生成系へ与える記述自体も拡張できる必要がある．本論文では，この観点から，生成系へ与える記述を自由に拡張できるようにした生成系について述べる．この生成系は，基本的には上述のデコレータパターンに基づいた構文解析器を生成する．しかし，実用的な構文解析器は，たとえば Yacc のアクションや SableCC における構文木とそのツリーウォーカのような機能による意味解析処理をあわせ持つことが求められる．本生成系では，このような機能をすべて拡張と見なし，生成系にどの拡張を使用するかを与えることで，動的にその拡張に対応し，その拡張のための記述を受理し，その拡張に対する記述に基づいた付加機能を持つ構文解析器を生成する．

An Extensible Compiler Generator

Jun'ichi Maita,[†] Akira Sato[†] and Hisashi Nakai[†]

According to various usages of the computer, often, we see many programming languages, each of them is suitable for its own area. Thus the study for compiler generators becomes more important. We developed a method for parser construction that can make the extension of pure parser easily based on the decorator pattern. The parser generator that implements this method should have the ability to accept the description for the attached functions dynamically. In this presentation, we present the parser generator that we developed. It generates basically pure parser based on above method. However, in practice, the parser is requied to have the ability of some semantic analysis, e.g., Yacc's action or SableCC's tree walker for AST generated during parsing, and so on. We regard these functions as 'extension'. When the user gives the specification that includes the description of which function the user wants to use, the generator accepts it and expands the generator itself so that it can process the following special desctiption for the function(s), and then the generator generates the parser with the function.

1. はじめに

近年，Java をはじめとしたオブジェクト指向言語が広く使われるようになったが，構文解析器生成系などのいわゆるコンパイラ生成系では，オブジェクト指向を考慮に入れた設計をしているものはまだ少ない．この観点から，我々は，デコレータパターンとテンプレートメソッドパターンを用いることにより，純粋な構文解析機能のみを持つ構文解析クラスに対して，ユーザの目的に応じて機能を容易に付加できる構文解析器の構成方法の提案を行った[7]（このようにして付加された機能を以降では付加機能という）．この方法により，ユーザは構文解析クラスに手を加えることなく，構文解析器を拡張することができる．実際には，この構成法に基づいた構文解析器生成系も試作し，提案を行った．

この生成系は，上述の枠組みのクラス群を生成するが，デフォルトの付加機能として，Yacc[3] のアクションに相当する付加機能，構文木（解析木）を作成し，その構文木上を走査してなんらかのアクションを起こすための Visitor パターンのクラス群（ツリーウォーカ）を生成する付加機能，エラーハンドリングを行うための基礎となる付加機能を提供している．もちろん，これらはデコレータになっている．

しかしながら，たとえば，Yacc のアクションの付

† 筑波大学
University of Tsukuba

加機能をユーザが利用するには，アクションについての情報を生成系へ与える必要があるのと同様に，付加機能によっては生成系へ情報を渡し，生成系はその情報に基づいて作成された付加機能を持つ構文解析プログラムを生成しなければならない．論文 7) で試作した生成系では，あらかじめ，システムとして用意した付加機能については，その付加機能への情報を生成系へ与える記述に入れることが可能であるが，独自の付加機能を作成した際に，そのような情報をシステムへの入力記述として与えたいならば，生成系を改造することになる．しかし，それではコストが高くなる．これは，生成系が自身に与えられる記述を，使用される付加機能によって動的に変更できる仕組みを持つことで解決できる．本研究ではこの観点から構文解析器生成系を構成する仕組みを考えた．

2. システムの概要

本システムの実装は Ruby [4] を用いて行った．本システムは基本的には文法を受け取り，それに基づいて LALR(1) 構文解析器クラスを生成する．前述のように付加機能を導入する場合，システムはその付加機能に関する情報を扱えなければならない．このため，本システムは，付加機能に関する情報を扱うために本システム自体を動的に拡張できる．以下では，ユーザの指示により拡張される本システム自体の機能のことを拡張と呼ぶ．すなわち，本システムでは，何も拡張しなければ単純に文法のみ受け付け，純粋な構文解析器クラスだけを生成する．ユーザが拡張を指定することで，たとえば，アクションについての記述を受け付けるよう，本システムは自身を拡張する．

2.1 本システムへ与える記述

本システムへ与える記述は Yacc 同様，全体が `%%` によって 3 つの部分に分けられる．最初の `%%` までは宣言部と呼ばれ，以下のものを記述できる．

`%class` 生成される構文解析器クラスの名前を与える．`%class` の後ろにクラス名を書く．

`%defext` 拡張を本システムを用いて作成する際に `%class` の代わりに指定する．`%defext` の後ろに拡張のクラス名を記述する．

`%inner` 生成される構文解析器クラスのコードに埋め込むコードを記述する．

`%extend` 使用する拡張の指定をする．`%extend` に続いて，拡張名，その実装ファイルを記述する．

`%prec` 演算子の順位を指定する．

`%decorate` 本システムにより生成される構文解析器クラスはデコレータパターンに基づいて実装される．既存または拡張により生成されるデコレータを利用するのに `%decorate` を指定する．

`%hook` 本システムを用いて拡張を作成する際，拡張が呼ばれるタイミングとそのときにどのような動作をするかを記述する．これは後に詳述する．

最初の `%%` から次の `%%` までの間には基本的には文法を記述する．この部分を文法部という．拡張を使うことにより，文法の記述の前後および各記号の間に独自の記法による拡張を使うための記述を埋め込むことができる．それらの部分は次のとおりである．

(a) 文法全体の始まりと終わり
(preRuleList, postRuleList)
(b) 各左辺・右辺ペアの始まりと終わり
(preRule, postRule)
(c) 各左辺の終わり (postLhs)
(d) 1 つの左辺に対する右辺全体の始まりと終わり
(preRhsList, postRhsList)
(e) 1 つの左辺に対する各右辺の始まりと終わり
(preRhs, postRhs)
(f) 各右辺の各記号の終わり
(postSym)

図示するとこれらは以下のようになる（f は省略）．

```
(a)
(b)expr(c) : (d)
          (e)expr '+' term(e')
        | (e)expr '-' term(e') (d')
(b')(a')
```

つまり，拡張はこれらの任意の部分で処理を行うことができる．以下，拡張を記載する位置のことをポジションと呼ぶ．

最後の `%%` 以降には任意のコードを書くことができる．

2.2 簡単な使用例

拡張なしで，簡単な算術式の文法を生成系に与える例を**図 1** に示す．行番号は説明のためのものである．

この例では `%inner` を使用して，2 行目から 22 行目までの間に字句解析器となる Ruby のコードを埋め込んでいる．25 行目から 40 行目が文法の定義である．文法はごく簡単な算術式のものである．

なお，`#` から行末まではコメントとなる．

41 行目の `%%` 以降は Yacc と同様に生成されるプログラムを記述している言語で書かれたプログラム断片の記述となる．ここではパーザプログラムを呼び出す，いわゆるメインプログラムに相当する部分のコードを記述している．

これに対して，字句解析器を自動で生成する拡張を

```
 1 %class TinyCalc
 2 %inner{
 3   def lex
 4     until @file.eof?
 5       @line = @file.gets
 6       until @line.empty? do
 7         case @line
 8         when /\A\s+/, /\A\#.*/, /\A\n/
 9           #skip blank and comment
10         when /\A[0-9]+/
11           yield :NUM, $&
12         when /\A./
13           yield $&, $&
14         else
15           raise RuntimeError,
             "must not happen #{line}"
16         end
17         @line = $'
18       end
19     end
20     yield nil, nil
21   end
22 %}
23 %%
24
25 #begin-rule
26   expr :
27       expr '+' term
28     | expr '-' term
29     | term
30   ;
31   term :
32       term '*' fact
33     | term '/' fact
34     | fact
35   ;
36   fact :
37       NUM
38     | '(' expr ')'
39   ;
40 #end-rule
41 %%
42
43 begin
44   parser = TinyCalc.new()
45   if ARGV[0]
46     File.open( ARGV[0] ) do |f|
47       r, = parser.yyparse(f)
48     end
49   else
50     r, = parser.yyparse($stdin)
51   end
52   puts r
53 rescue
54   $stderr.puts "#{File.basename $0}: #{$!}"
55   exit 1
56 end
```

図 1 calc.dr
Fig. 1 calc.dr

```
 1 %class TinyCalc
 2 %extend Lexer ('depager/lex.rb')
 3 %%
 4
 5 %LEX{
 6   /\s+/, /\#.*/, /\n/  { }
 7   /[1-9][0-9]*/        { yield :NUM, $&.to_i }
 8   /./                  { yield $&, $& }
 9 %}
10
11 #begin-rule
   以下省略
```

図 2 calc.lex.dr
Fig. 2 calc.lex.dr

使用した記述を**図 2** に記す．ただし，後半部分は図 1 と同じであるので省略する．

この例では，最初の **%%** から 11 行目までに新たな記述ができるように拡張したわけである．この記述に対しての処理は図 2 の 2 行目のように **%extend** を使用してどの拡張がそれを行うかを指定する．後に述べるように拡張の開発自体をこの生成系を使って行うことが可能である．

この例では，正規表現とそれに対応するアクションを Lex のように記述することができる拡張を示している．

この生成系では，文法定義部分（たとえば，図 1 の 25 行目からと 40 行目まで）の記述も拡張可能である．具体的には，Yacc などのように { と } で囲った部分に Ruby のコードを書くようなことを可能とする．実際は，拡張の作り方によって，この部分に書ける内容を変えることができる．

我々が作成した Yacc 風のアクションの拡張の場合，次のように記述できる．

```
expr :
    expr '+' term { val[0] + val[2] }
  | expr '-' term { val[0] - val[2] }
  | term          { val[0] }
```

同様に我々が作成した，AST を構築し，その上で Visitor パターンに基づいてアクションを実行するような付加機能のための拡張の場合，生成規則に付随す

```
 1 %defext PseudoActionExtension
 2 %extend Lexer ('depager/lex.rb')
 3 %extend Action ('depager/action.rb')
 4 %decorate Action ('depager/action.rb')
 5
 6 %hook postrhs
 7 %%
 8 %LEX{
 9   /\s/ { }
10   /./  { yield $&, $&, @lineno }
11 %}
12  start:
13    '{' '}' { warn 'HIT' }
14  ;
15 %%
```

図 3 paction.dr
Fig. 3 paction.dr

```
1 %class PseudoActionTest
2 %extend PseudoAction ('paction.rb')
3 %%
4 #begin-rule
5   expr:
6       expr '+' fact { }
7   ;
8   fact:
9       ID { }
10  ;
11 #end-rule
12 %%
13 p = PseudoActionTest.new()
14 #p.yyparse(STDIN)
```

図 4 pactiontest.dr
Fig. 4 pactiontest.dr

るアクションを次のように記述できる．

```
%AST{
  Node [val]       { @val = nil }
  Visitor          { }
  add(l, r) { ~val = ~l.val + ~r.val }
  sub(l, r) { ~val = ~l.val - ~r.val }
  mul(l, r) { ~val = ~l.val * ~r.val }
  div(l, r) { ~val = ~l.val / ~r.val }
  literal(-n)      { ~val = ~n }
%}

#begin-rule
  expr :
      expr '+' term
      => add(expr, term)
```

この例では，%AST の部分および文法記述部分における => を用いた記法について，拡張で定義している．

2.3 拡張を本システムで作る例

本システムを用いて拡張を作成する例を示す．ここでは生成規則の右辺の最後に { と } が来た場合に画面上に HIT と表示する拡張を作成する．本システムに与える記述を**図 3** に示す．

最初の %defext によってこの記述が拡張についてのものであることを示している．%defext の後ろには拡張として生成されるクラスの名前を記す．2 行目は，システムで標準として用意している字句解析拡張を使うことを宣言している．これにより，この記述中では 8〜11 行目のように Lex 風の字句解析用の記述が利用できるようになっている．同様に 3 行目では Yacc のアクション風の拡張を使用することを宣言しており，それは，現在与えられている記述に対する拡張部分を解析するパーザへのデコレータの形になっているので，4 行目でデコレータとして使用することを宣言している．

6 行目の %hook から 15 行目の %% までが指定したポジションに対する解析動作である．この例では postrhs が指定されているため，各生成規則の右辺の最後に記述される内容を解析し，処理するためのプログラムを記述していることになる．実際には 8 行目から 11 行目の間が，その部分で受け付ける字句の定義となり，12 行目から 14 行目が受け付ける文法となる．

1 つの記述ファイルには複数の %hook を並べて書くことができる．

この記述を本システムに与えると指定したポジション（portrhs）に記述された拡張記述を解析するための構文解析器が生成される．得られた解析器がファイル paction.rb に入れられているとすると，それを使用する例は**図 4** のようになる．

1 行目では生成されるパーザクラス名を指定している．2 行目で図 3 によって生成された拡張を使うことを宣言している．これにより，この記述に書かれた文法部分で右辺の終わりを見つけるごとに HIT を画面に表示するので，本システムに図 4 を与えると HIT が 2 回表示される．

本来は図 3 の 13 行目でそこに記述されている内容を取り出し，加工して，生成される構文解析プログラムに埋め込むような使い方をする．

3. 本システムの実装

本章では，本研究の目的である，これまでに提案した構文解析器の構成方法[7] に対して，その考え方を発揮できるような構文解析器生成系を実現する方法につ

いて述べる．まず，システムの設計方針について述べ，構成する要素のそれぞれを述べた後，本システムにおける問題点について考察する．

3.1 本システムの設計方針

本システムの概要については前章で述べた．拡張は複数組み合わせて使われることもある．これに対し，拡張どうしの衝突や拡張と本システム自体への入力との衝突についても考慮すべきであるが，本研究では，基本となるアイデアである，入力記述を拡張できるということの実装に重点をおき，現時点では拡張を設計する際，記述する者が使用上の制約によって，衝突を回避する方針とし，システムとして衝突を回避する方法の導入は後回しとした．このことは 3.5 節で述べる．

本システムには拡張を作る場合と通常の構文解析器クラスを生成する場合の 2 通りの使用パターンがある．この 2 つで動作の仕組みが異なる．本システムでは 3 つの構文解析器を用いて与えられた記述を解析していく．これらは次のようになっている．

宣言部パーザ システム起動時に生成され，記述の宣言部を解析し，以降の動作を決定する．

文法部パーザ 記述の文法部を拡張部パーザと協調して解析する．

拡張部パーザ 拡張が使用された際，文法の前後，各記号の間などに記述された部分の解析を行う．各拡張は，1 つのマスタパーザと各ポジションに対応した複数のスレーブパーザから構成される．詳細は 3.3 節で述べる．

全体の処理の流れは次のようになる．

(1) 本システムが記述を読み込む．

(2) まず，宣言部パーザが起動される．宣言部パーザは次を行う．

  (a) 最初に `%class` もしくは `%defext` があるかを確認し，どちらであるかを記録する．

  (b) `%extend` が見つかるとその拡張名を記録しておく．

  (c) `%%` に到達すると，文法部パーザのインスタンスを生成する．そして，それぞれの拡張用のマスタパーザのインスタンスを作る．

  (d) 各拡張のマスタパーザのインスタンスは，各ポジションの記述を処理するスレーブパーザを生成し，それらを文法部パーザに登録する．

  (e) 文法部パーザが最初の `%%` 以降の文法部分の解析を始める．それぞれのポジションで該当する登録されているパーザを呼び出す．拡張が複数あり，同じポジションに対して `%hook` に相当するパーザを複数登録している場合は，登録順にそれらが起動される．登録は，記述された順番となる．

(3) 文法部パーザのすべての処理が終わったら，宣言部パーザが構文解析プログラムを出力する．

3.2 文法部パーザとその構文解析

文法部パーザは，Yacc 風の BNF 記述を受け付ける．文法部は以下の構文を持つ．ここで各 <〜> はポジションを示している．

```
grammar:
  <preRuleList> rulelist <postRuleList>
;
rulelist:
    <preRule> <postRule>
  | <preRule> rule <postRule> rulelist
;
rule:
  sym <postLhs> ':' <preRhsList>
    rhslist <postRhsList> ';'
;
rhslist:
  <preRhs> symlist <postRhs> rhslisttail
;
rhslisttail:
  | '|' rhslist
;
symlist:
  |  sym <postSym> symlist
;
sym: [A-Za-z_][A-Za-z0-9_] ;
```

文法部パーザは再帰的下向き構文解析を行い，その過程でポジションごとに登録された拡張部パーザへ制御を移す．

3.3 拡張とその構文解析

拡張は，基本的には生成系が生成する純粋な構文解析器に対する付加機能，すなわち，デコレータのコードを生成する．

拡張に対する記述については前章で述べた，文法部の最初と最後，文法記号間に入れることができる．本システムではこのそれぞれに構文解析器を用意し，必要に応じて呼び出されるようにする．これらを制御するため，各拡張は，マスタパーザとスレーブパーザから構成される．マスタパーザはスレーブパーザの管理を行い，スレーブパーザどうしの情報共有を支援する．

拡張はマスタパーザを必ず 1 つ持たなければならない．スレーブパーザはマスタパーザに従属し，1 つのマスタパーザに複数持たせることができる．スレーブパーザは本システムを用いて拡張を記述した場合，本システムにより生成される LALR パーザである．

文法部パーザへの解析処理の登録はマスタパーザが行う．拡張はマスタパーザに属するメソッドまたはスレーブパーザを文法部パーザへ登録できる．このとき，このマスタパーザに属するメソッドは構文解析を行っても行わなくてもよい．本生成系を用いてスレーブパーザを実装した場合，文法部パーザへの登録コードも自動で生成される．

拡張は，一般的には上述のようにデコレータのコードを出力する．たとえば，Yacc のアクションのような拡張の場合，文法記号間や末尾に書かれたコード断片を取り出して加工することで，生成される構文解析器が構文解析を行うときに，対応する生成規則が還元された時点で，記述されたコード断片が実行されるような付加機能を生成する必要がある．これは構文解析の動作と関連するため，構文解析テーブルと連動する動作用のテーブルを作ることになる．すなわち，このような動作用テーブルを含めた付加機能（デコレータ）を生成する．

3.4 拡張から利用可能な本システムの機能

拡張は拡張記述の構文解析を行う必要がある．そのため，入力記述にアクセスできる必要がある．入力記述および関連情報の取得インタフェースは，現在処理中のファイルハンドルや行，行番号などの入力記述に関する情報へのアクセスを提供する．

拡張がコードの生成などの処理を行う場合，拡張記述だけでなく，文法部パーザの持つ関連する情報が必要になる場合もある．文法情報の取得・更新インタフェースは処理済みの文法全体の情報および処理中の生成規則の情報を提供する．これにより，たとえば，処理済みの生成規則数や処理中の生成規則の情報（左辺の記号や右辺の記号列）などの情報を取得できる．また，文法全体や各生成規則などは更新も可能であり，拡張から対象文法の変更などが行える．

拡張は，一般的には，その処理の結果としてコードを生成する．拡張が生成したコードを保持するため，システムは 2 つのコードキューを提供する．生成される構文解析器クラスの内側に埋め込まれるコードのためのキューとその外側に埋め込まれるコードのためのキューである．

そのほか，システム本体は，生成される構文解析器クラスのクラス名などのメタ情報へのアクセスのためのインタフェースも提供している．

3.5 問題点の検討

この章の冒頭でも述べたように，本システムでは，拡張どうしの衝突，拡張と本システム自身が持つ記述との衝突についても考えなければならない．

ここでは構文に関する面と意味に関する面について考察する．

3.5.1 構文面から見た衝突に関して

本システムの設計にあたって，衝突について考慮する点は次である．

(1) 本システムは，構文解析器生成系という性質上，拡張記述が挿入される位置は，文法全体の前後か各文法記号の前後ということになる．このため，拡張記述は文法記号とは区別がつく必要がある．

(2) 複数の拡張を使用する場合，同じ位置へ複数の拡張記述を与えることがある．

(1) についてはその拡張記述の開始の記号と終了の記号と，文法記号を区別する．我々は本システムにおける文法記述に用いられる記号は，`[A-Za-z0-9_\n\t :|;]` のみとする．拡張はこれら以外の文字で始まることとすることで文法記述と拡張記述での衝突は回避することができる．ただし，現時点の実装ではこれらの衝突があるかどうかをチェックすることはしていないため，実際に使用する際には，生成系へ与える記述を作成する者が注意する必要がある．

(2) については，拡張どうしの衝突について考える必要がある．拡張 A と拡張 B の両方を使用する場合を考えよう．まず，原則として拡張は宣言された順に登録され，登録された順に解析が行われる．このため，本システムに与える記述中で複数の拡張を使用する際は，それぞれのポジションで宣言順に対応する拡張の記述を行う必要がある．次に，いずれかの拡張記述が省略可能である場合を考慮しなければならない．本生成系を用いてこれらの拡張を作る際には，省略可能である場合，`%hook` を記述するときに，次のようにその先頭となる記号についての正規表現を併記することとした．

```
%hook postrhs /[/
```

これにより，入力の先頭がその正規表現と一致するときのみ，その拡張の解析が行われる．拡張 A，拡張 B とも，同じ記号で記述が始まり，いずれかの記述を省略する場合は，後者が省略されると見なされる．

本システムにおける拡張の記述は，上述のように文法記号と区別がつくように開始されることが求められ

るが，同様にその拡張記述の終わりと次の文法記号との間が判別できることも求められる．これは一般的には各拡張記述はその始まりと終わりの記号が区別がつけばよいと考えた．ただし，そのような記述が並列できるようにならないことが要求される．これはたとえば，その拡張記述の文法が次のようになっていてはならないという意味である．

```
START : lists ;

lists : lists a_list
      | a_list
      ;

a_list : '{' A '}' ;
```

以上の検討の結果，一般的なプログラミング言語における構文の拡張とは異なり，本システムにおいては拡張どうしまたは拡張とシステムへの記述との衝突に関しては，拡張の起動順序と拡張を設計するにあたっての各拡張の開始記号／終了記号の区別をつけられるようにしなければならないという使用上の規則で対応することとする．後者は本システムを用いて拡張を作成する際には，システムによってチェックする機構を導入することも考えられるが，拡張自体は本システムを用いなくても作成できるため，現時点ではシステムにはその機能を持たせてはいない．また，各拡張は，それぞれ個別に静的に作成され，必要時に本システムへの記述に記載されることで本システム起動時に実行されることから，実行時にその拡張の文法に遡って衝突のチェックを行うことはしないこととする．

3.5.2 意味の面から見た衝突に関して

今度は意味の面から見た衝突に関して述べる．ここでいう意味の面から見た衝突とは，複数の拡張を指定した際，拡張どうしあるいは拡張と本システムの挙動で干渉しないかということである．

本システムにおいて，各拡張の各パーザはそれぞれ独立したクラスとして実現される．このため，それぞれが持つ情報はカプセル化されるので，それぞれのインスタンスどうしで明示的に情報を破壊するようにコードを作らない限り，問題はない．

拡張どうしの依存については現時点では著者らは必要性を感じておらず，生成系を用いた拡張の作成では，それをサポートする機能は存在しない．一方，拡張は一般的には，生成される構文解析器への付加機能となるデコレータを出力する．複数の拡張を用いると複数のデコレータを出力することになるが，そのそれぞれのコードが生成された構文解析器の実行時に干渉するかどうかについては，関与していない．これは，たとえば C プリプロセッサにおいて，関数のように振る舞うマクロを定義する際，書き方が十分でない（括弧をつけてやるなど）と書き手の意図したものと違う挙動をするコードに置き換えてしまうことと同様であると考える．これについては，今後，運用していく中で，問題になりそうな点をシステムで解決できることもあるかもしれないが，それは今後の課題とする．

3.5.3 他研究との比較

まず，本研究との類似研究であるが，構文解析器（もしくはコンパイラ）生成系という観点で見た場合，これまでに類似する研究はない．これは，従来の構文解析器の構成法では，生成系に与える記述を動的に拡張する必要性がなかったためである．一方，本研究は，論文 7) における構文解析器の構成法を実現するために拡張という考え方を導入した．すなわち，論文 7) の考え方を発揮できる生成系にするには拡張という考え方が必要であったわけである．

しかしながら，（文法を受け取り，構文解析プログラムを出力するという）プログラミング言語処理系という範疇で本システムを眺めると，これまでの歴史の中で，その拡張を考えたものは存在する．

本システムでは，文法部パーザの実装は再帰的下向き構文解析法を用いている．これは拡張部の解析処理が必要になった際，その処理プログラムを登録するという動作を容易に実装できるためである．この点では，一杉の手法[2]，Camlp4 [1] などと同様である．ただし，構文レベルにおける衝突の回避という観点では，上述したとおりであり，関連研究に見られるバックトラックなどの手法は不要であると考え，実装していない．

一方，コンパイラを再利用可能なモジュールに分けることで，拡張可能にする佐伯の手法[6] では，拡張における，実行時の意味における衝突回避についても述べられているが，この観点についても上述のように，本システムでは，その用途が構文解析器生成系であることから重要性が低いととらえ，実装はしていない．この点は今後の課題としたい．

3.6 例

この節ではいくつかの標準拡張において，どのような処理が行われるかを例示する．

3.6.1 Action 拡張

Action 拡張は，各右辺の終わりで，以下のような処理を行う．

(1) アクション部分 {…} の取り込み，および構文解析

(2) 取り込んだアクションをメソッドの形に修正して，この拡張自身が持つコードのためのキューに追加する．また，この拡張自身が持つ，生成するデコレータのためのリデュースステーブルをそのメソッドを実行するように更新する．
(3) 生成したコードとリデュースステーブルをデコレータクラスの形に修正して生成されるパーザクラスの外側に埋め込まれるコードのためのキューに追加する．

3.6.2 Lex 拡張

Lex 拡張は，文法全体の始まりで，以下のような処理を行う．

(1) 字句定義部 `%LEX{…%}` を構文解析し，字句解析メソッドのコードを生成する．
(2) 生成した字句解析メソッドのコードを，生成されるパーザクラスの内側に埋め込まれるコードのためのキューに追加する．

3.6.3 AST 構築拡張

AST 構築拡張は，文法全体の始まりで，以下のような処理を行う．

(1) 構文木定義部 `%AST{…%}` を構文解析し，構文木ノードクラス群と Visitor クラスのコードを生成する．
(2) 生成したコードを，生成されるパーザクラスの外側に埋め込まれるコードのためのキューに追加する．

AST 構築拡張はまた，各右辺の終わりで，以下のような処理を行う．

(1) 対応付け部分 `=>…` を構文解析し，処理コードを生成する．
(2) 生成した処理コードをメソッドの形に修正して，この拡張自身が持つコードのためのキューに追加する．また，この拡張自身が持つ，生成するデコレータのためのリデュースステーブルをそのメソッドを実行するように更新する．
(3) 生成したコードとリデュースステーブルをデコレータクラスの形に修正して生成されるパーザクラスの外側に埋め込まれるコードのためのキューに追加する．

4. おわりに

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いう観点から，生成系自体を拡張可能にする方法について考察し，実装を行った．

本システムは，自身へ与える記述を比較的容易に変更できるため，コンパイラフロントエンドの研究者にとっては，自身が考えた記法に基づいた生成系を比較的容易に作り出し，実験することが可能となる．また，現在の多種多様な用途には，自身の目的にあった言語を比較的容易にかつ迅速に実装できることも要求される．その際に，本システムの拡張可能性は開発者へのカスタマイズの手段を与えることになる．このためには，本システムは，さまざまな拡張を備えていく必要もあり，どのような拡張を作ればよいのかは今後の課題でもある．

最後に，本システムの 2 つの応用について述べる．

まず，1 つ目は属性文法の研究への寄与である．属性文法およびその応用の研究はいまなお，さかんに行われているが，研究者が属性文法を自身の領域に応用しようと考えた際，一からその生成系（または処理系）を開発するにはかなりの労力を要するが，本システムにより，拡張（作成）したい部分のみに集中して，そのような処理系を開発することができる．たとえば，具体的な文法記述の前に，各文法記号に対する属性を列挙するようにし，各生成規則では文法記号とその属性をドットなどでつないだ表現を用いて，属性評価規則を記述する，オーソドックスな属性評価器生成系へ与える記述を作ることが考えられる．

もう 1 つは，バックエンドまで含めたコンパイラ生成系への発展である．並列化コンパイラ向け共通インフラストラクチャCOINS[5] は，高性能なバックエンドを構成するための枠組みを提供している．COINS で提供される機能を本システムで使用できるようにライブラリを作成するとともに本システムの拡張として組み入れることで，バックエンドまでをサポートするコンパイラ生成系として，本システムを利用できるようになる．著者らは現在この拡張の作成にとりかかっている．

参考文献

1) de Rauglaudr, D.: Camlp4—Tutorial. http://caml.inria.fr/pub/docs/tutorial-camlp4/index.html
2) 一杉裕志：高いモジュラリティと拡張性を持つ構文解析器，情報処理学会論文誌：プログラミング, Vol.39, No.SIG.1 (PRO 1), pp.61–69 (1998).
3) Johnson, S.C.: Yacc: Yet Another Compiler Compiler, *UNIX Programmer's Manual*, Vol.2, Holt, Rinehart and Winston, New York, NY, USA, pp.353–387 (1979). AT&T Bell Laboratories Technical Report July 31, 1978.
4) まつもとゆきひろ：オブジェクト指向スクリプト言語 Ruby，アスキー (1999).

5) 中田育男ほか：21 世紀のコンパイラ道しるべ—COINS をベースにして，情報処理，Vol.47, No.4〜7 (2006).
6) 佐伯 豊：再利用可能な拡張機構を備えた言語処理系，博士論文，北陸先端科学技術大学院大学 (2001).
7) 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).

(平成 18 年 1 月 16 日受付)
(平成 18 年 8 月 8 日採録)

舞田 純一（学生会員）
1982 年生．2006 年筑波大学図書館情報専門学群卒業．筑波大学図書館情報メディア研究科在学中（2006 年現在）．

佐藤 聡（正会員）
1991 年筑波大学第三学群情報学類卒業．1996 年同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 年筑波大学システム情報工学研究科講師．現在，同大学学術情報メディアセンター勤務．キャンパスネットワークの企画管理運用，ネットワーク，データベース，言語処理等の研究に従事．電子情報通信学会，ACM-SIGMOD-JAPAN 各会員．

中井 央（正会員）
1968 年生．筑波大学第三学群情報学類卒業，同大学大学院工学研究科修了〔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）．日本ソフトウェア科学会，ACM，ACM-SIGMOD-JAPAN 各会員．