オブジェクト指向に基づいた構文解析器構成法の提案

佐竹 力[†] 中井 央[††]

コンパイラにおける構文解析器の作成には生成系が用いられることが多い．構文解析器生成系である Yacc の生成する構文解析器では，アクションを記述することで構文解析時に意味動作を行うことができる．しかし，このような処理の付加の方法はきわめて局所的で，構文解析器への機能付加の要件を十分に満たしているとはいえない．別の方法として，出力された解析器プログラムを書き換えることで処理を追加することもできるであろうが，そのような作業は煩雑かつ危険で好ましくない．また，これらの方法では生成系の設計方針に基づいて解析器に装備されている処理が不要の場合，容易に取り外すことができないという不便性がある．たとえば，既存の生成系の出力する構文解析器ではエラー処理が次に期待されるトークンを出力して終了することが多いが，そのエラー処理を取り外してエラー修正機能を取り付けるようなことは容易ではない．本論文では，純粋な構文解析処理以外に構文解析器で行われる可能性のある処理を付加的な機能と見なし，生成された純粋な構文解析だけを行う構文解析器に後から機能を自由に付加できるような構成法を提案する．また，この構成法を基にした構文解析器生成系についても述べる．

An Approach to Parser Construction Based on Object-Oriented Method

Chikara Satake[†] and Hisashi Nakai[††]

Usually we use parser generator for compiler construction. There are some methods when we want to change the behavior of the generated parser. One of them is using actions with Yacc (or the similar generator), but for example, if we want to add error repairing method to the parser, we can not use them. Another method is to modify the generated parser by hand. But it is very dangerous operations. Or if we want to remove some function, it is not easy to remove it. We consider that functions except for basic parsing actions are optional. In this paper, we propose a parser construction method that we can easily augment a parser with adding functions using object oriented method. We also mention the implementation of the parser generator.

1. はじめに

近年，多種多様なニーズに対応するため，そのニーズにあった言語とその処理系を開発することは少なくない．処理系のうち，特に字句解析，構文解析の開発には生成系が用いられるのが一般的である．その代表的なものには Lex [11] と Yacc [9]，あるいはそれらの互換である Flex [14] や Bison [5] といったものをはじめとし，最近では Java が多く利用されるため，Java 言語によるソースプログラムを出力する，ANTLR [13]，SableCC [6]，CUP [1]，JavaCC [15] などがある．

コンパイラ生成系と呼ばれる多くのツールは，構文解析器生成系である．いずれも文法を入力として与えるとその文法に沿っているかどうかをチェックする構文解析器を出力するが，実際にはその機能だけを使うことはまずない．コンパイラの中で構文解析に続くフェーズとされる意味解析の動作として，Yacc に代表される，各生成規則にマッチした際に行われるアクションの機能を持つものか，構文解析器によって生成されたツリーを走査するもののいずれかが提供される．また，いずれのツールも構文解析時のエラー処理機能として満足のいくものを備えてはいない．

これらの点を考慮し，生成系が生成する構文解析器はごく単純な構造とし，必要に応じて上述の，アクション，ツリー生成，エラー処理などの機能が付加できるほうが，コンパイラ開発者にとって都合がよい．本研究ではこの立場に立って，機能付加を容易にし，機能拡張性の高い構文解析器を生成する方法を提案する．また，本論文ではこの方法に従って構文解析器生成系を実現する方法について述べ，実際に作成した生成系について紹介する．

† 図書館情報大学
University of Library and Information Science
†† 筑波大学
University of Tsukuba

2. 生成される構文解析器の設計

ここでは主として LR 構文解析法をもとに機能拡張性の高い構文解析器の構成法について論じる．

2.1 研究の動機と目的

本研究の最初の動機は，構文エラーの研究をする際，新たな構文エラー処理方法を考案したとしても，その有効性を確かめるためには，構文解析器生成系を実装するか，構文解析器生成系（の出力する構文解析器）を改造する必要があることであった．すなわち，たとえばフリーソフトとしてソースが公開されている Bison の LALR ドライバはある程度の最適化されたコードになっており，教科書が示すような単純な動作をするわけではなく，それゆえ，エラー処理を行う部分を特定し，かつ改造することは難しい．不可能ではないにしろ，そのようなソースプログラムの解析と，ソースプログラムを書き換えることによってバグが混入する可能性により，考案したエラー処理方法を確かめるまでにはいくつもの難関があることになる．

この問題を解決するには，構文解析器（のドライバ）の構成として，まず，純粋に（教科書どおりの）構文解析を実行するプログラム（**図 1**）を用意しておき，必要に応じて機能を付加することが可能なようにしておくことを思いついた．具体的な設計については以下で述べるが，この考えに基づくと，既存の生成系が提供する機能は，純粋な構文解析機能に対する付加機能である．たとえば，Yacc に代表される還元時のアクション機能がある．しかし，Yacc には構文解析の結果として構文解析木を出力する構文解析器を出力する機能はない．一方，SableCC などの生成系が生成する構文解析器は，構文解析時のアクションは（基本的に）機能として提供していないが，解析結果として解析木を生成する．SableCC ではこの解析木を走査するための機能（Visitor パターンの拡張）を提供している．また，Yacc では，構文解析器のデバッグを行うため，`yydebug` という static 変数を設けており，`YYDEBUG` というマクロと組み合わせて使うことで，構文解析時に解析器の状態を標準エラー出力へ出力する機能が備わっている．

このほか，Yacc では `error` トークンによる構文エラー処理の機能を提供しているが，この使いにくさは誰もが認めるところである．SableCC（2.18.2）には構文エラー処理機能はない．一方で構文エラーに関する研究は近年でもよく行われている [2)~4),8),10)]．

そこで本研究では基本機能としての構文解析とそれに機能を付加することで多様な構文解析器を構築できるようにする方法と，それに基づいた生成系の実装を目的とする．付加機能として，システムは以下の付加機能を用意することにした．

```
while(true){
  switch (action 表を引いた結果){
    shift: 入力をシフトする;
    還元：還元動作を行う;
    受理：入力を受理したため，呼び出し元へ返る;
    エラー：入力に誤りがあったため，
            エラー処理を行う;
  }
}
```

図 1 教科書的な LR 構文解析ドライバ
Fig. 1 LR driver algorithm in the text book.

(1) 還元時のアクション
(2) 解析木の生成（およびそれを走査するクラス）
(3) デバッグ情報の出力
(4) 構文エラー処理機能

このような機能の拡張については，Decorator パターン[7)] を適用する．また，付加機能はあくまで付加機能であり，基本となる構文解析機能を妨げたり，上書きしたりしてはまずい．このため，その具体的な実装には Template Method パターン[7)] を利用する．

2.2 設計方針

ここで要求している機能拡張性には次の 2 つの制約がある．

(1) 複数の機能を付加することを可能とし，できる限り，お互いが影響を及ぼさないようにする．
(2) 基本となる構文解析動作はユーザによって改変されることがないようにする．

(2) は本来の動作を書き換えたり，上書きしたりするのではなく，保持することを目的とする．この 2 つの制約を満たすために Decorator パターンと Template Method パターンを用いて構文解析クラスを作成することにする．

まず，(1) の機能の付加については，Decorator パターンを用いて**図 2** のクラス階層とする．すなわち，基本となる構文解析クラスを Basis とし，付加機能を表すクラスを AdvancedParser クラスのサブクラスとする．システムとしては，AdvancedParser クラスを継承したクラスとして次のものを提供することにする．

Action Yacc のアクションに相当する，ユーザが指定した還元時の動作を扱う．
Tree 構文解析の結果として解析木を出力する．
Debug 構文解析器の動作を確認するための情報を

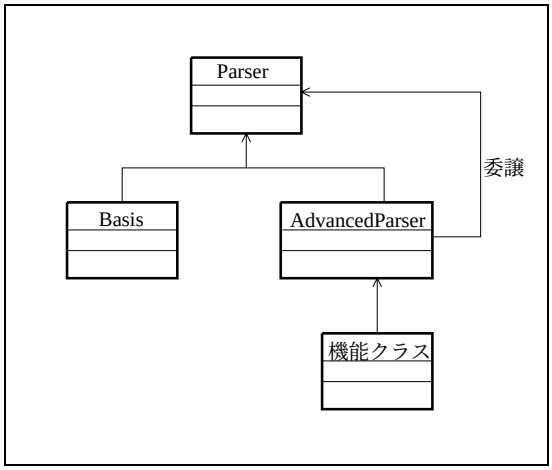

図 2 機能付加を可能にするクラス間の関係
Fig. 2 The relation among classes, which make possible to attach functions.

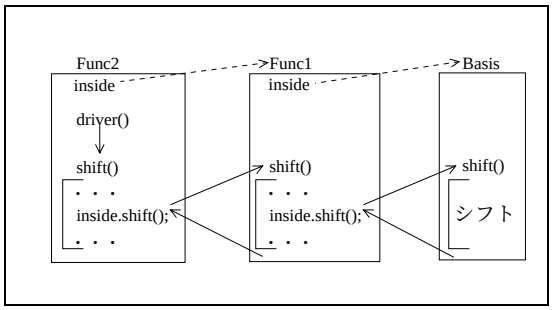

図 3 シフト時における各オブジェクトでの処理の様子
Fig. 3 A snapshot of processing on each object when shift occurs.

出力する．

ErrHandle 構文エラー処理を行う．

AdvancedParser クラスは次のコンストラクタを持つ．

```
public AdvancedParser(Parser inside);
```

これにより与えられた Parser クラスのインスタンスを内部で保持する．

例として機能 1（Func1），機能 2（Func2）を構文解析器に付加する場合，次のようなプログラム断片を記述することになる．

```
Parser p = new Func2(new Func1(new Basis(lexer)));
```

このときのシフト動作と構文解析の流れは**図 3** のようになる．inside から出ている点線矢印は指しているクラスのインスタンスであることを意味している．実線矢印は実行の流れである．上記のようにして生成した構文解析クラスのインスタンスに対し，構文解析の動作は，状態に応じて，シフト，還元，受理，エラー処理のいずれかになるが，機能付加を行った場合の動作の流れは，シフトを例にすると次のようになる．

(1) 構文解析の次の動作としてシフトであることが分かったら，自身の shift メソッドを呼ぶ．
(2) まず，自身の shift メソッドでは前処理をする．
(3) 次に自身が inside に保持するインスタンスの shift メソッドを呼び出す．
(4) そこから戻ると後処理をする．

この動作の流れは保証される必要がある．すなわち，オーバライドによって処理の流れが変わらないようにする必要がある．このために Template Method パターンを用いることにした．AdvancedParser クラスの shift メソッドを次のように定義している．

```
protected final void shift {
  this.beforeShift();
  this.inside.shift();
  this.afterShift();
}

protected void beforeShift(){}
protected void afterShift(){}
```

これらは，還元（reduce），受理（accept），エラー処理（error）の各メソッドについても同様である．これにより，機能クラスを作成するには，AdvancedParser クラスを継承し，beforeShift もしくは afterShift をオーバライドすればよい．

2.3 字句解析

構文解析器は，字句解析器を呼び出し，ソースプログラムから切り出されたトークンの情報を得る．このため，構文解析器と字句解析器の間には密接な関係があり，相互の情報のやりとりのためのインタフェースをよく吟味して設計しておく必要がある．

しかし，システムでは字句解析器の生成は行わないため，ユーザが字句解析器を用意する．そこで，生成される構文解析器で呼び出す字句解析器は以下の Lexer インタフェースを実装しなければならないことにする．

```
public interface Lexer {
  Attribute scan() throws java.io.IOException;
}
```

scan メソッドは，呼び出されるたびに Attribute インタフェースを実装したクラスのインスタンスを返す字句解析メソッドである．

トークンとして扱われるクラスは Attribute インタフェースを実装しなければならない．これを実装するクラスは構文解析器でトークンを識別するための数値であるトークンナンバとトークンの字面を保持するためのフィールドを持ち，Attribute インタフェースで定

められたメソッドによりそれらの値を返す．トークンナンバとしてセットする値は生成系が定義したものを使用する．システムの生成する TokenNumber インタフェース内では各トークンにトークンナンバが適切に割り当てられるように定数を宣言する．字句解析器ではこれを実装してトークンナンバを参照し，Attribute インタフェースを実装したクラスのインスタンスの生成を行う必要がある．システムでは Attribute インタフェースとそれを実装した Token クラスを用意する．

構文解析器では字句解析器から Attribute インタフェースを実装したクラスのインスタンスを受け取り，そこからトークンナンバを得て解析を進める．アクションにおいて文法記号に対応するインスタンスを参照する場合に得られるものもこれと同一のインスタンスである．アクションでそのようなクラスを扱う場合には，トークンナンバや字面以外の情報を持たせたいことがあるため，Attribute インタフェースを実装してユーザが用意したクラスを構文解析器で扱うこともできる．

2.4 機能クラス

前述のとおり，システムでは Action，Tree，Debug，ErrHandle の 4 つの機能クラスを用意する．機能クラスの動作には構文解析の経過の情報，たとえば還元の対象となった生成規則や現在の状態スタックのトップなどを必要とするものがある．そのため，機能クラスは構文解析器のインタフェースを定義する Parser クラスからそれらの情報へのアクセッサを継承する．

2.4.1 Action クラス

Action クラスは Yacc におけるアクションと同様の振舞いをするクラスである．内部ではスタックを持ち，構文解析スタックと連動する形で値を保持する．ここでいう値とは字句解析器から取得するオブジェクトのことである．

シフトの直後には字句解析器から返されたオブジェクトをスタックにプッシュする．

還元の直後にはまず，構文解析器から還元の対象となった生成規則の右辺の長さを取得し，その長さ分をスタックからポップして右辺の各文法記号に対応するインスタンスを得る．その後，還元の対象となった生成規則の番号を構文解析器から取得して対応するアクションを実行する．

2.4.2 Tree クラス

Tree クラスは解析木を構築するクラスである．解析木を構成するノードを表すクラスは Node クラスを継承しなければならない．非終端記号に対応するノードは生成規則ごとに生成され，そのクラス名は以下のようになる．

"P" + 非終端記号名 + 数字

クラス名の先頭には “P” がくる（production のつもりである）．非終端記号名は対応する生成規則の左辺の非終端記号名になる．数字は，ある非終端記号を左辺とする生成規則の中で該当する生成規則が何番目であるかを示している．以下のような生成規則があるとする．

```
statements : statements st
           | st
           ;
```

このとき，“statements : statements st” に対応するノードのクラス名は “Pstatements1” となる．また，終端記号に対応するノードのクラスは字句解析器から受け取るインスタンスとなるため，解析木を生成するときに扱うトークンのクラスは Node クラスを継承している必要がある．システムでは Node クラスを継承していてトークンとして扱える NToken クラスを用意する．

Tree クラスは，シフト直後に字句解析器から受け取ったトークンのクラスをスタックにプッシュし，還元直後には還元の対象となった生成規則の右辺の長さの分だけスタックからポップし，それらを子ノードとしてその生成規則に対応する非終端記号ノードのインスタンスを生成してスタックにプッシュする．ここで生成した解析木はシステムの用意する TreeWalker クラスを用いて走査される．

2.4.3 Debug クラス

Debug クラスはシフト時に字句解析器から得られたオブジェクトにより字面を得て出力し，還元時にはその対象となった生成規則を出力するものである．各生成規則の字面は Debug クラス内に java.lang.String の配列として保持し，生成規則を出力する際は，還元直後に還元の対象となった生成規則の番号により，その配列を参照して出力する．

2.4.4 ErrHandle クラス

ErrHandle クラスは抽象クラスである．Basis クラスは構文エラー処理が付加されていない場合，構文エラーが見つかると自身で解析を終了しなければならない．逆に，構文エラー処理が付加されている場合はエラー時の扱いをエラー処理機能に任せなければならないため，Basis クラスは自身で解析を終了してはならない．そのため，Basis クラスの afterError メソッドは以下のようになっている．

```
protected void afterError(){
  if(this.getEH() == null){
```

```
    System.out.println("syntax error.");
    System.exit(1);
  }
}
```

getEH は構文解析器からエラー処理クラスを取得するメソッドである．ErrHandle クラスのコンストラクタでは自身をフィールドにセットするようにする．すなわち，構文解析器にエラー処理機能が付加された場合には Basis クラスはそれを判定し，自身で解析を終了しないような仕様になっているため，構文エラーが起こってもエラー処理機能は妨害されることはない．システムでは構文エラー処理をするクラスに次に期待されるトークンを出力して終了するクラス，トークンの読み飛ばしによりエラー回復を試みるクラス，ユーザと対話的にエラー処理を行うクラスを用意する．これらはすべて ErrHandle クラスを継承する．

2.4.5 Action クラスと Tree クラスの併用

Action クラスと Tree クラスの両方を付加することでアクションと解析木の構築を協調して行うことができるようにする．ここでいう協調とは，解析木を構築しながら，それを構成するインスタンスを対象にアクションを行うことである．解析終了後には，解析木を走査して同じインスタンスを参照していくことができる．これは，還元直後に Tree クラスが還元の対象となった生成規則の部分木を構築し，その部分木を Action クラスが参照して対応したノードのインスタンスを扱うことで実現する．

これを実装するには Action クラスが Tree クラスのインスタンスを知っている必要がある．そのため，機能付加は以下のようにする．

```
Parser p = new Action(new Tree( ... ));
```

つまり，Action クラスのコンストラクタに Tree クラスのインスタンスを与える．これにより Action クラスは Tree クラスをコンストラクタで受け取ったときにはアクションで扱うインスタンスを Tree クラスから取得し，それ以外のときはアクセッサを用いて構文解析器から取得したトークンのクラスのインスタンスを扱えばよい．

3. 構文解析器生成系の試作および生成系の利用例

Java を用いて構文解析器生成系を実装した．この章では，まず，生成系への入力について述べ，次に本生成系の使用例をいくつか示す．

3.1 生成系への入力

図 4 のように本生成系に与える記述は 3 つの部分から成り立っている．ここでは部分ごとに記述できる内容を詳しく説明していく．

```
ユーザコード部
%%
宣言部
%%
生成規則部
```

図 4 生成系に与える記述の構成
Fig. 4 The structure of specification to the system.

なお，記述にはコメントを入れることを考慮して，複数行にまたがるコメントを可能にする/**/と行単位のコメントのための//を使用できるようにしている．

3.1.1 ユーザコード部

最初の “%%” までの部分がユーザコード部である．ここには import 文などを記述する．ここに記述された Java コードは，“%tree” が宣言されているときは TreeWalker クラスに，宣言されていないときは Action クラスの最上部，package 宣言の後にそのままコピーされる．

3.1.2 宣 言 部

2 つの “%%” に挟まれた部分が宣言部である．ここでは生成する構文解析器のクラスや動作に関連するものなどを定義する．ここで記述される宣言を以下にあげる．なお，“%package” と “%compress” 以外は宣言される順番は任意である．

- %compress …LR 構文解析時に使用される解析表を圧縮することを指示する．“%method” に続けて指定される構文解析の方法をあげた直後に記述する．
- %debug …これを記述すると構文解析器クラスに付加できる Debug クラスを生成する．Debug クラスはデバッグを補助する情報を標準出力に出力するクラスである．
- %function パージングメソッド名…パージングメソッド名の指定をする．宣言されていない場合，パージングメソッド名は “parse” になる．
- %implements インタフェース名 [, インタフェース名]*…Action または TreeWalker クラスが実装するインタフェース名を指定する．複数記述する場合はコンマで区切る．“%tree” が宣言されているときは TreeWalker クラス，宣言されていないときは Action クラスに implements に続いて記述される．

- %member{ Java コード } …括弧内に Java コードを記述する．たとえば，構文解析の結果として Action クラスから値を得たい場合に get メソッドなどを記述できる．“%tree” が宣言されているときは TreeWalker クラス，宣言されていないときは Action クラス内のフィールド部に，括弧内に記述したものがコピーされる．
- %method 解析法…構文解析法の指定を行う．LR 解析法の場合，直後に “%compress” が記述できる．“%method” を宣言していない場合は LALR 解析になる．以下から選択して記述する．
  - SLR …SLR 解析法
  - CLR …正準 LR 解析法
  - LALR …LALA 解析法
  - LL …LL 解析法

  ただし，LL 解析法は現段階では不完全な実装である．
- %node_head { パッケージ名 }…各非終端記号ノードクラスのファイルの先頭にコピーされる．
- %node_pkg ノードパッケージ名…指定したパッケージ下に生成されるノードパッケージ名を指定する．ノードパッケージにはノードクラスに関係のあるクラスが生成される．宣言しないと “node” になる．
- %node_type ノードクラス名…ツリーを構成するノードのクラスを指定する．ここに指定されるクラスは Node クラスを継承している必要がある．宣言されていない場合は Node クラスが設定される．
- %package パッケージ名…生成される構文解析器を扱うパッケージ名を指定する．1 つ目の “%%” の直後に必ず記述されなければならない．
- %parser_name 構文解析器クラス名…構文解析クラスの名前を指定する．宣言されていない場合は “Parser” になる．
- %parser_pkg 構文解析器パッケージ名…指定したパッケージ下に生成される構文解析器クラスに関係のあるクラス群のパッケージ名を指定する．宣言しないと “parser” になる．
- %stype トークンクラス名…構文解析器が字句解析器から受け取るトークンのクラスを指定する．ここで指定されるクラスは Attribute インタフェースを実装している必要がある．“%stype” が宣言されていなくて，かつ “%tree” が宣言されていなければ Token クラスが設定される．“%stype” が宣言されていなくて，かつ “%tree” が宣言されていれば NToken クラスが設定される．
- %token 終端記号名…生成規則内のトークンを宣言する．1 つも宣言されていないとエラーになる．
- %tree 木の種類…ツリーを作るクラスを使用することを宣言する．木の種類には以下のいずれかを指定する．
  - CST …解析木
  - AST …構文木

  ただし，構文木を作る仕組みは未実装なのでどちらを指定しても解析木を作るクラスが生成される．

3.1.3 生成規則部

2 つ目の “%%” 以降が生成規則部である．生成規則の記述には BNF 記法に従ったものを用いる．ある 1 つの非終端記号（Nonterminal）に対する生成規則の文法は以下のようである．’Id’ には終端記号か非終端記号名がくる．終端記号はクォーテーションで括って示している．

本生成系では最も上に記述された生成規則の左辺記号を開始記号として扱う．また，本生成系では拡大文法を導入するため，開始記号を右辺に，真の開始記号となる “#start#” を左辺に持つ生成規則を文法に加える．拡大文法は解析の終了を明確にするために，開始記号を左辺とする生成規則がただ 1 つになるようにするものである．

```
production : 'Nonterminal' ':' rhs ';'
rhs        : rhs  '|' one_rhs
           | one_rhs
           ;
one_rhs    : symbols semantics
           ;
symbols    : symbols 'Id'
           | 'Id'
           ;
```

3.1.4 意味動作の記述

上の文法の “semantics” の部分には意味動作を記述する．“semantics” の文法は以下のとおりである．これらの文法から分かるとおり，1 つの生成規則に対して 3 つまで意味動作を記述することができる．実際には，ツリーを生成することを宣言していれば，異なるタイミングを指定して意味動作を記述する必要がある．ツリーを生成することを宣言していなければ，タイミングを記述しないで 1 つだけ意味動作を記述する．

```
semantics   : field semantic1 semantic2 semantic3
            |
            ;
field       : '%{' Java_source '%}'
            |
            ;
semantic1   : node_action
```

```
            |
            ;
semantic2   : node_action
            |
            ;
semantic3   : node_action
            |
            ;
node_action : timing '{' Java_source '}'
            ;
timing      : '[in]'
            | '[mid]'
            | '[out]'
            |
            ;
```

in，mid，out はそれぞれ意味動作が起動されるタイミングを示している．in は行きがけ時に，out は帰りがけ時に，mid は各子ノードを訪問する際，それぞれの間に動作することを意味する．

次に意味動作について “%tree” を宣言している場合としていない場合について説明する．

"%tree"を宣言していない場合

%tree を宣言していない場合，記述された意味動作は構文解析時に還元が起こった際に実行される．semantic1・semantic2・semantic3 の “timing” より後を 3 つまでならいくつ記述してもエラーにはならないが，実際に実行されるのは始めに記述したものだけで後のものは無視される．[in]・[mid]・[out] を記述するとエラーになる．また “%{...%}” を記述した場合，これはノードのための記述であるので還元時の意味動作では使用されないため，警告されるが無視して解析は続行される．記述した意味動作は Action クラス内にコピーされる．

また Java_source 内には Java コードを記述できる．この中で “$” 記号を使用して生成規則の文法記号に対応するインスタンスを参照することができる．“$$” を使用すると左辺のインスタンスを参照することができ，“$[1-9][0-9]*” を使用すると右辺の文法記号に対応したインスタンスを参照できる．数字は右辺のインデックスで，右辺の生成規則の長さより大きい数を指定するとエラーになる．以下のようなインデックスになる．

```
E   :   E   PLUS   T   ;
$$      $1     $2     $3
```

前述のとおり，インスタンスとなるクラスは “%stype” で設定することができる．還元時の意味動作の後で左辺 “$$” の値が null であるかチェックされ，null であると $$ = $1; に相当する処理が行われる．このような処理が行われるのは，意味動作がスタック

```
例 1:
     {System.out.pritnln("hoge1");}← in になる
[out]{System.out.pritnln("hoge2");}← out になる
     {System.out.pritnln("hoge3");}← mid になる
例 2:
     {System.out.pritnln("hoge1");}← in になる
     {System.out.pritnln("hoge2");}← mid になる
     {System.out.pritnln("hoge3");}← out になる
例 3:
[out]{System.out.pritnln("hoge1");}← out になる
 [in]{System.out.pritnln("hoge2");}← in になる
     {System.out.pritnln("hoge3");}← mid になる
例 4:
     {System.out.pritnln("hoge1");}← mid になる
 [in]{System.out.pritnln("hoge2");}← in になる
```

図 5 意味動作のタイミングの割り振り例
Fig. 5 An example of assignment of timing of semantic actions.

を用いて実現されているためである．

還元時の意味動作では “%field” を用いて記述したメソッドやフィールド変数を使用できる．“%field” 内の Java コードは意味動作とともに Action クラス内にコピーされるので，“%field” で

```
private void execute(){
  System.out.println("action!");
}
```

のようにメソッドを記述すれば，還元時の意味動作では

```
this.execute();
```

として使用することができる．

"%tree"を宣言している場合

記述された意味動作は TreeWalker クラスのインスタンスに対し traverse メソッドを呼び出すことで実行される．[in]・[mid]・[out] を semantic1 などに記述することで意味動作の起こるタイミングを明示的に指示できる．ただし，以下のように 1 つの生成規則に対して同じタイミングを複数記述するとエラーになる．

```
[out]{System.out.pritnln("out");}
[out]{System.out.pritnln("out");}
[in]{System.out.pritnln("in");}
```

タイミングを指示していない場合，まだタイミングを指示されていないものから順に in，mid，out が割り振られる（**図 5**）．

1 つの生成規則に対し 1 つのノードクラスが生成される．各ノードクラスごとにフィールド変数やメソッドを加えたい場合，“{% Java_source %}” の

```
 1  /* ここに Java コードを記入する */
 2  %%
 3  %package arithmetic
 4  %stype Opr
 5
 6  %token PLUS
 7  %token MINUS
 8  %token MULT
 9  %token DIV
10  %token Num
11  %token L_PAR
12  %token R_PAR
13  %%
14  S : E
15  {
16    System.out.println("result: "+
                          $1.getValue());
17  }
18    ;
19  E : E PLUS  T
20  {
21    $2.execute($1, $3); $$ = $2;
22  }
23    | E MINUS T
24  {
25    $2.execute($1, $3); $$ = $2;
26  }
```

```
27    | T
28  {
29    $$ = $1;
30  }
31    ;
32  T : T MULT F
33  {
34    $2.execute($1, $3); $$ = $2;
35  }
36    | T DIV  F
37  {
38    $2.execute($1, $3); $$ = $2;
39  }
40    | F
41  {
42    $$ = $1;
43  }
44    ;
45  F : Num
46  {
47    $$ = $1;
48  }
49    | L_PAR E R_PAR
50  {
51    $$ = $2;
52  }
63    ;
```

図 6 arithmetic.y
Fig. 6 arithmetic.y.

Java_source の部分に記述された Java コードはそのノードクラスのフィールドにコピーされる．そのような Java コードに import が必要なパッケージに含まれるクラス（Java の標準の API も含む）を使用したい場合は “%node_head” を用いる．

ノードの意味動作内でも還元時の意味動作と同様に “$” を使用した参照ができる．ただし，ノードの意味動作の場合に “$$” で参照されるのはここで記述された意味動作を行うノード自身（this と同値）であり，“$数字” で参照されるのはその子ノードである．

3.2 本生成系の使用例

本生成系を用いて次のものを作成した．

(1) 簡易電卓（Action クラスを利用したもの）
(2) 簡易電卓（構文解析時に解析木を作り，構文解析後，解析木を走査するもの）
(3) 簡易なプログラミング言語のコンパイラの実装（Action クラスと Tree クラスの併用）
(4) Yacc の error トークンを使用する場合と同様の振舞いをするエラー処理クラス
(5) 本生成系の再実装

以下では，まず最初の 3 つについて簡単に説明し，ErrHandle クラスを使用する例を説明した後に Yacc の error トークンを使用する場合と同様の振舞いをするエラー処理クラスの実装について述べる．

3.2.1 簡易電卓（Action クラスによるもの）

まず，生成系に与える記述を**図 6** に示す．

以下，簡単に説明する．

4： 字句解析器から受け取るオブジェクトのクラスとして Opr （ユーザが定義したもの）を指定している．

6〜12： トークンの宣言である．

14〜53： 生成規則とアクションの宣言である．ここでは，四則演算を扱っている．Yacc でアクションを利用する場合とほぼ同様の記述になる．ただし，ここではスタックに積まれるのは字句解析器から返されたオブジェクトであり，計算結果もそのオブジェクト内に格納される．Yacc と同様に各文法記号に対応するオブジェクトを$を使った記法で参照できる．

Opr クラスおよびそのサブクラスを**図 7** に示す．

そして JFlex を利用して字句解析クラスを作成した．JFlex へ与えた入力を**図 8** に示す．以下，簡単に

```
 1  package arithmetic;
 2  import arithmetic.parser.*;
 3
 4  public class Opr extends Token {
 5    protected int value;
 6
 7    public Opr(int tokenNumber,
                    String lexeme){
 8      super(tokenNumber, lexeme);
 9    }
10
11    public Opr(int tokenNumber){
12      super(tokenNumber);
13    }
14
15    public int getValue(){
16      return this.value;
17    }
18
19    public void execute(Opr op1, Opr op2)
      {}
20  }
21
22  class PlusOpr extends Opr {
23    public PlusOpr(int tokenNumber,
                       String lexeme){
24      super(tokenNumber, lexeme);
25    }
26
27    public void execute(Opr op1, Opr op2){
28      this.value=op1.getValue()+
                      op2.getValue();
29    }
30  }
31
32  class MinusOpr extends Opr {
33    public MinusOpr(int tokenNumber,
                       String lexeme){
34      super(tokenNumber, lexeme);
35    }
36    public void execute(Opr op1, Opr op2){
37      this.value = op1.getValue()-
                         op2.getValue();
38    }
39  }
40
41  class MultOpr extends Opr {
42    public MultOpr(int tokenNumber,
                         String lexeme){
43      super(tokenNumber, lexeme);
44    }
45    public void execute(Opr op1, Opr op2){
46      this.value = op1.getValue()*
                         op2.getValue();
47    }
48  }
49
50  class DivOpr extends Opr {
51    public DivOpr(int tokenNumber,
                        String lexeme){
52      super(tokenNumber, lexeme);
53    }
54    public void execute(Opr op1, Opr op2){
55      this.value = op1.getValue()/
                         op2.getValue();
56    }
57  }
58
59  class Digit extends Opr {
60    public Digit(int tokenNumber,
                       String lexeme){
61      super(tokenNumber, lexeme);
62      this.value = Integer.parseInt(lexeme);
63    }
64  }
```

図 7 Opr.java
Fig. 7 Opr.java.

説明する．

6： 前節で説明したように字句解析クラスは TokenNumber と Lexer という 2 つのインタフェースを実装する必要がある．

8： 字句解析器メソッドが返すクラス型を宣言している．

10〜12： 入力の終わりに達したときに行う動作を指定している．

18〜26： 正規表現とそれにマッチした際の動作を定義している．各々，その字面にあった Opr クラスのサブクラスのインスタンスを生成し，返している．

最後に main メソッドを含むクラスを**図 9** 示す．

6 行目で字句解析器クラスのオブジェクトを作成し，7 行目では，それを与えて Basis クラスのオブジェクトを作成し，それを与えて Action クラスのオブジェクトを作成している．

具体的にはコマンドラインから以下の手順でコンパイラの作成を行う．

```
% java dpg.Depage arithmetic.y
% jflex arithmetic.l
% javac arithmetic/*.java
```

3.2.2 簡易電卓（Tree クラスを利用するもの）

簡易電卓は前節で説明したように構文解析時のアクションを用いることで実現できるが，ここではツリーを生成し，それを走査することで動作するものを作成

```
 1  package arithmetic;
 2  import  arithmetic.parser.*;
 3  %%
 4  %public
 5  %class Scanner
 6  %implements TokenNumber, Lexer
 7  %function scan
 8  %type Opr
 9
10  %eofval{
11    return new Opr(-1);
12  %eofval}
13
14  digit = [1-9][0-9]* | "0"
15  ws    = [ \r\n\t\f]
16  other =  .
17  %%
18  "+"     { return new PlusOpr(PLUS, "+"); }
19  "-"     { return new MinusOpr(MINUS, "-"); }
20  "*"     { return new MultOpr(MULT, "*"); }
21  "/"     { return new DivOpr(DIV, "/"); }
22  "("     { return new Opr(L_PAR, "("); }
23  ")"     { return new Opr(R_PAR, ")"); }
24  {digit} { return new Digit(Num, yytext());}
25  {ws}    { }
26  {other} { System.out.println
              ("Illegal character!"); }
```

図 8 arithmetic.l
Fig. 8 arithmetic.l.

```
 1  package arithmetic;
 2  import arithmetic.parser.*;
 3
 4  public class Main{
 5    public static void main(String[] args){
 6      Lexer lexer = new Scanner(System.in);
 7      Parser parser =
          new Action(new Basis(lexer));
 8
 9      parser.parse();
10    }
11  }
```

図 9 Main.java
Fig. 9 Main.java.

する．

この例では，生成系に与える記述で，`%tree` による宣言をし，解析木を生成する付加機能をつけ，生成される解析木のノードクラスを出力するよう指示する．生成される解析木のノードクラスは`%node_type` によって指定することができる．すなわち，デフォルトで用意している Node クラスを継承して，独自のノードクラスを定義し，それを扱うノードクラスとすることができる．

```
 1  %%
 2  %package postfix
 3  %tree CST
 4  %node_type OprNode
 5
 6  %token PLUS

...

13  %%
14  S : E
15  [out]{
16    System.out.println();
17    System.out.println("result: "+
                           $1.execute());
18  }
19    ;
20  E : E PLUS  T
21  %{
22    public int execute(){
23      return $1.execute() + $3.execute();
24    }
25  %}
26  [out]{System.out.print("+ ");}
27    | E MINUS T
...
```

図 10 postfix.y
Fig. 10 postfix.y.

生成系へ与える記述の一部を**図 10** に示す．

図 10 では 3 行目で`%tree CST` を指定している．これにより，生成系は構文解析時に解析木を構築する機能クラスを出力する．そして，構築された解析木を走査し，各ノードを訪問した際に行われる動作については各生成規則のところへ記述する．たとえば 21 行目の`%{`から 25 行目の`%}`でくくられた部分は，生成系がそのノードクラスを生成する際，そのクラス（この例では `PE1`）内にコピーされる．26 行目では動作の記述の前に `[out]` がつけられている．これは，深さ優先で訪問した際，帰りがけにその動作を行うことを意味する．

この例では OprNode クラスを定義し，生成系が生成する各ノードクラスはこのクラスを継承するようにしている（図 10 の 4 行目）．OprNode クラスを**図 11** に示す．生成規則ごとの動作の記述では，生成されるノード内で OprNode で定義されている execute メソッドをオーバライドするように記述している．

この例では main メソッド内で次のようにして構文解析器の生成と生成された解析木の走査を指示して

```
package postfix;
import postfix.node.*;

public class OprNode extends Node {
  protected int value;

  public int execute(){
    return this.value;
  }
}
```

図 11 OprNode.java
Fig. 11 OprNode.java.

いる．

```
Tree parser = new Tree(new Basis(lexer));

parser.parse();

new TreeWalker().traverse(parser.getRootNode());
```

3.2.3 Cmm コンパイラの作成

次の例は，Cmm（C−− の意味）言語のコンパイラの実装である．Cmm は，筆者の 1 人がコンパイラの授業用に開発したものである．C 言語ライクな構文を持ち，関数の再帰呼び出しなどが可能である．実行は PL/0' インタプリタ[12] を関数の再帰呼び出しが可能なように改造したものを用いている．なお，型は整数型のみである．本来，この言語のコンパイラはコード生成までを 1 パスで行うように作成することが可能であるが，この例では，構文解析を行いながら解析木を作りつつ，意味解析までを行い，コード生成を解析木の走査によって行う．すなわち，還元時の意味動作（Action）と解析木の出力（と走査）を併用している．このためには，還元時の意味動作用と，解析木の走査用に 2 つの記述を用意する必要がある．

1 つ目の記述には`%tree` を宣言しない，Action クラスを使用するための内容を記述する．2 つ目の記述には`%tree` を宣言し，構文解析によって生成されたツリーを走査する際の動作を記述する．このような生成系の使われ方を考慮し，生成系には`-a` オプションを用意した．これは（FIRST や FOLLOW を求めるなどの）構文解析器の生成に必要な計算を省略するものである．コマンドラインからの入力は次のように行う．

```
% java dpg.Depage symbol_check.y
% java dpg.Depage -a code_generate.y
```

これによって，構文解析に必要なクラス，Action クラス，Tree クラスおよびその走査クラスが生成される．これらを利用して，ここでは次のように構文解析を行うコードを記述している．

```
Tree t = new Tree(new Basis(new Scanner ...));
Parser P = new Action(t);

p.parse();
```

3.2.4 機能クラスの作成

通常の機能クラスを作成するにはシステムの用意する AdvancedParser クラスを継承して afterShift メソッドなどをオーバライドすればよいが，構文エラー処理を行う機能クラスを作成する場合，前述した理由により ErrHandle クラスを継承して作成する．ここではシステムの用意する OutputErr クラスの例を用いて機能クラスの作成法について述べる．

OutputErr クラスは，エラーが見つかった場合，そこまでの解析の流れに沿って，次に来るべきトークンの種類を出力して終了する．この機能クラスでは LR 解析表を利用して，次に期待されるトークンを集めておく．OutputErr クラスの処理の流れが分かるよう擬似コードで**図 12** に示す．

以下，簡単に説明する．

4： 先読みとして期待されるトークン名を保持する．

6〜9： ErrHandle クラスを継承し，そのコンストラクタを用いることで，構文解析器に機能付加したときに構文エラー処理クラスとして認識させることができる．8 行目は初期状態について次に来るべき（すなわち入力の最初に期待される）トークンの候補を集めている．

11〜14： シフト後の処理を行う．新たなトークンをシフトした後なので，そこまでに集めた先読み集合は捨てて，先読み候補を集め直す．

16〜18： 還元前の処理を行う．還元することが分かってから状態が変わる前に先読み候補を集めておく．

20〜23： エラー発見後の処理を行う．lookaheads に集められた要素をすべて出力し，プログラムを終了する．

25〜32： 先読み候補を集めるメソッドである．引数に与えられた状態とすべてのトークンから解析表を引き，エントリのあるトークン名をフィールドの集合に追加していく．

機能クラスを実装するには，構文解析器の持つ様々な情報を利用できる必要がある．このような情報を取得するためのメソッドは Parser クラスに用意されている．以下，その一部を示す．“`#stype#`” には “`%stype`” で指定されたトークンのクラスが入る．

- `Table getTable()` …構文解析表クラスを取得する．
- `int getReducePrdc()` …現在までの解析で最後

```
 1  import java.util.LinkedHashSet;
 2
 3  public class OutputErr extends ErrHandle {
 4    private LinkedHashSet lookeaheads;
 5
 6    public OutputErr(Parser parser){
 7      super(parser);
 8      this.collectCandidates( 初期状態 );
 9    }
10
11    protected void afterShift(){
12      lookeaheads の要素をクリア;
13      this.collectCandidates( スタックトップ );
14    }
15
16    protected void beforeReduce(){
17      this.collectCandidates( スタックトップ );
18    }
19
20    protected void afterError(){
21      lookeaheads に保持されている
        トークン名を出力;
22      プログラムの終了;
23    }
24
25    protected void collectCandidates(LR 状態){
26      for(int token=0; token <
                    トークンの種類数; token++){
27        token と LR 状態から解析表を引く;
28        if(エントリーがある){
29          lookeaheads に token に対応する
            トークン名を追加;
30        }
31      }
32    }
33  }
```

図 12 OutputErr.java
Fig. 12 OutputErr.java.

に還元された生成規則の全生成規則におけるインデックスを返す.

- `int getReduceTimes()` …現在までの解析で最後に還元された生成規則の右辺の長さを返す.
- `void setLexer(Lexer lexer)` … 字句解析器クラスを受け取り，構文解析器クラス内にセットする.
- `MyStack getStack()` …現在の解析スタックのインスタンスをそのまま返す.
- `int getState()` …現在の解析スタックのトップの値を返す.
- `void setState(int state)` …Parser クラスの static 変数 state に LR 状態をセットする．state はスタックトップに積まれている状態を表す値である.

```
afterError():
    if (yyrecovering == true){
      エラートークン (ERR) を生成;
      while(true){
        if (s_m と ERR で action 表を引いた結果
            がエラーでも受理でもない) break;
        状態スタックをポップ;
        if (状態スタックが空) エラーを返す;
      }
      最後に読み込んだトークンを
        先読みキューに入れる; // (※)
      yyrecovering = false;
    }
    yyerrok = 3;
afterShift():
    if (yyerrok != 0) yyerrok--;
    if (yyerrok == 0) yyrecovering = true;
```

図 13 Yacc の error トークンを使用したときと同様の振舞いをする機能クラス
Fig. 13 The function class of Yacc's error token.

- `void pushLookAhead(#stype# token)` …先読みキューにトークンをエンキューする.
- `void setEH(ErrHandle eh)` …ErrHandle クラスのインスタンスをフィールド ErrHandle eh にセットする.
- `ErrHandle getEH()` …ErrHandle クラスのインスタンスを返す.
- `#stype# getLastToken()`…現在までの解析で最後に読み込んだトークンを返す.

これらのメソッドはすべてスーパクラスから継承している.

3.2.5 error トークンによるエラー回復クラス

エラー回復の 1 つの例として，Yacc で導入されている error トークンを用いたエラー回復を実現した.

まず，生成系への記述の中で Yacc における `error` トークンに相当するトークンも`%token` を使って宣言する．生成規則としては次のように `error` トークンを使用したものを含める.

```
st : WRITE E SEMI
  ...
   | ID COLEQ E SEMI
   | ID COLEQ ERROR SEMI
     { System.out.println("error"); }
```

エラー処理用のクラスは次のように実現する．まず，エラー後のシフト回数を記録する `yyerrok` と現在，回復を試みているのかどうかを表すフラグである `yyrecovering` を用意する.

具体的な処理は**図 13** のようになる．状態スタックのトップは s_m で表現する.

(※) については，エラーが見つかった時点での先読みはまだ消費されておらず，回復処理後の入力として利用するために先読みキューに入れておく．構文解析器は先読みキューが空の場合，字句解析器に次のトークンを要求する．

この例では Yacc の error トークンによるエラー回復の方法を模倣したが，このような機能をデフォルトのエラー処理として構文解析器プログラムに埋め込むのではなく，着脱可能な 1 つの機能として実現することができた．

4. おわりに

構文解析器に対して，容易に機能を付加することを可能にする構文解析器の構成法をオブジェクト指向に基づいて提案し，それに基づいて作成した生成系について述べた．この方法により，純粋な構文解析動作と，ツリーの生成や Yacc でいうアクションなどの構文解析にともなう動作を切り離すことができるようになった．このことはソースコードを改変するという改造ではなく，クラスによる機能の付加によって構文解析器を改造することを可能にした．特にコンパイラの研究者は，新たなアイデア（たとえば新しい構文エラー処理方式）が浮かんだ場合，それを実装するには必要最小限の労力で済むようになる．

今後の課題はいくつもあるが，大きく次のものがあげられる．

解析木の走査： 本システムの生成する TreeWalker クラスでは，Visitor パターンを使用していない．構文解析によって解析木が得られた後，多パスでこれを走査することを想定すると Visitor パターンの導入も検討する必要がある．

LL 構文解析のサポート： 本システムでは，構文解析法の選択肢の 1 つとして LL 構文解析もサポートしている．これまで LR 解析をベースとして機能付加について述べてきたが，LL（再帰的下向き）構文解析を対象としても同様の方式で機能付加を扱えると考え，現在，その実装方法を検討中である．

字句解析器のサポート： 本システムを使用する際，字句解析器は別途用意する必要がある．しかし，多くの情報は本システムに与える記述から得ることができ，それに基づいてたとえば JFlex への入力記述を自動で生成することも可能である．

構文エラー処理機能： 実用的な構文エラー処理機能を開発し，それを付加機能クラスとして提供するようにする．

参考文献

1) Appel, A.W.: *Modern Compiler Implementation in Java*, Cambridge University Press, Cambridge, UK (1998).
2) Cerecke, C.: Repairing Syntax Errors in LR-based Parsers, *25th Australasian Computer Science Conference* (*ACSC2002*), Oudshoorn, M.J. (Ed.), Conferences in Research and Practice in Information Technology, Vol.4, Melbourne, Australia, ACS, pp.17–22 (2002).
3) Corchuelo, R., Perez, J.A., Ruiz, A. and Toro, M.: Repairing Syntax Errors in LR Parsers, *ACM Trans. Prog. Lang. Syst.*, Vol.24 (2002).
4) Corchuelo, R., Pérez, J.A., Ruiz, A. and Toro, M.: Repairing syntax errors in LR parsers, *ACM Trans. Prog. Lang. Syst.*, Vol.24, No.6, pp. 698–710 (2002).
5) Donnelly, C. and Stallman, R.M.: BISON The YACC-compatible Parser Generator, Technical report, Free Software Foundation (1988).
6) Gagnon, É.: Sablecc, an Object-Oriented Compiler Framework, Master's thesis, McGill University, Montreal (1998). http://www.sablecc.org/
7) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional (1995). 本位田真一，吉田和樹（訳）：オブジェクト指向における再利用のためのデザインパターン改訂版，ソフトバンクパブリッシング株式会社 (1999).
8) Jeffery, C.L.: Generating LR syntax error messages from examples, *ACM Trans. Prog. Lang. Syst.*, Vol.25, No.5, pp.631–640 (2003).
9) 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).
10) Kim, I.-S. and Choe, K.-M.: Error repair with validation in LR-based parsing, *ACM Trans. Prog. Lang. Syst.*, Vol.23, No.4, pp.451–471 (2001).
11) Lesk, M.E.: Lex—A Lexical Analyzer Generator, Technical Report CS-39, AT&T Bell Laboratories, Murray Hill, NJ, USA (1975).
12) 中田育男：コンパイラ，オーム社 (1995).
13) Parr, T. and Quong, R.: ANTLR: A predicated LL (k) parser generator (1995). http://www.antlr.org/
14) Paxson, V.: flex fast lexical analyzer generator (1988).
15) 五月女健治：JavaCC コンパイラ・コンパイラ

for Java，株式会社 テクノプレス (2003).

(平成 16 年 2 月 20 日受付)
(平成 16 年 7 月 25 日採録)

佐竹　力

1981 年生．図書館情報大学 4 年次在学中（2004 年 3 月現在）．

中井　央（正会員）

1968 年生．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学系助教授（学術情報処理センター勤務）．ソフトウェア科学会，ACM，ACM-SIGMOD-JAPAN 各会員．