プログラミング言語のスコープ階層を反映した構造化解析器

神谷 年洋† 石尾 隆‡ 井上 克郎‡

ソフトウェアの開発では，コンパイラやリバースエンジニアリングツールなど，ソースコードを処理するための種々のツールが用いられる．これらのツールが必要とする字句解析や構文解析をサポートするために，これまでに lex と yacc を初めとする多くのコンパイラ・コンパイラが開発・利用されてきた．コンパイラ・コンパイラによって，これらのツールの開発の生産性や保守性が向上する一方で，処理対象のプログラミング言語の構文がコンパイラ・コンパイラの能力を超える部分では，複雑なプログラミングが必要とされる．本稿では，オブジェクト指向プログラミング言語に特徴的なスコープの階層性，構文エラーからの回復処理，繰り返し型開発，回帰テスト等への対応を取り入れた解析器（字句解析，構文解析，意味解析を含む）の新しい設計を提案する．設計段階からこれらを取り入れることで，複雑なプログラミングを一掃し，ディペンダブルな解析器の開発を可能にすることを目指す．具体的な適用例として，オブジェクト指向プログラミング言語の一つである Featherweight Java を対象とする解析器を実装し，その有効性や適用可能性について評価する．

1 はじめに

ソフトウェアの開発では，ソースコードを入力とする様々なツールが用いられる．コンパイラを筆頭として，リバースエンジニアリングツール，lint などのソースコード診断ツール，変更波及解析やスライスなどの依存関係解析を求めるツールなど，数多くのツールが存在する．また，diff や grep，バージョン管理システムといった，基本的には任意のファイルを処理可能なツールにおいても，ソースコードの意味を考慮することでより高度な処理を行うことを目指したものが登場してきている [8] [14] [19].

これらのツールは，そのツールが必要とするソースコードの意味情報の程度によって，字句解析，プリプロセッシング，構文解析，意味解析等を行う．これまでに，字句解析や構文解析をサポートするため，lex と yacc [13]，JavaCC [11]，ANTLR [1]，Boost Spirit [4] などの字句解析器，構文解析器生成ツール（以降，「コンパイラ・コンパイラ」）が開発・利用されてきた．

コンパイラ・コンパイラを用いることで，正規表現や拡張 BNF で記述された字句規則および構文規則から，字句解析器および構文解析器のプログラムを生成することが可能になる．また同時に，それら字句規則や構文規則を，ソースコード処理ツールの意味解析部分と分離して記述することが可能になるなど，ツールを開発する労力が削減され，可読性や保守性が向上する．その一方で，プログラミング言語の構文規則は，しばしばこれらのコンパイラ・コンパイラの制限を超え，そのような場合には適用に困難が伴う．

本稿では，オブジェクト指向プログラミング言語（以降「OOPL」）に特徴的なスコープの階層性を，クラスの継承の階層によって実装する解析器（字句解析，構文解析，意味解析を含む）の新しい設計を提案

†Toshihiro Kamiya, 科学技術振興機構 さきがけ, PRESTO, Japan Science and Technology Agency.
‡Takashi Ishio, Katsuro Inoue, 大阪大学大学院情報科学研究科, Graduate School of Information and Science and Technology, Osaka University.

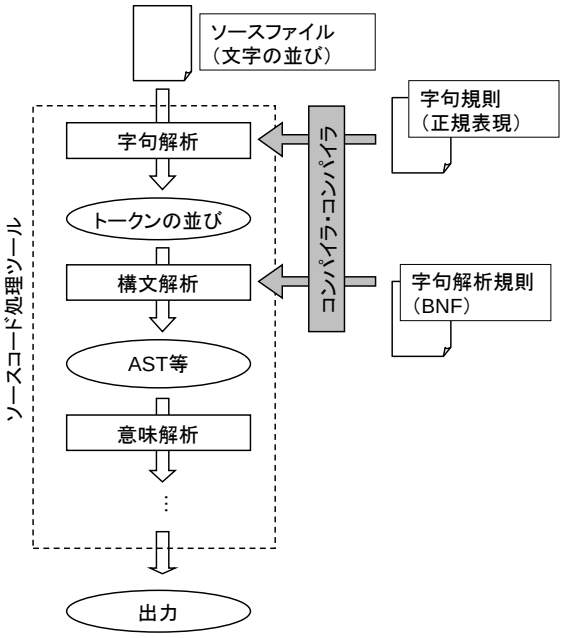

図 1　ソースコード処理ツールの構成とコンパイラ・コンパイラの適用

する．具体的な適用例として，小規模ながらもクラスやインスタンス変数，メソッドといった概念を備えるプログラミング言語 Featherweight Java [7] を対象とする解析器を，JavaCC [11] を利用して実装する．

2　コンパイラ・コンパイラの利用

図 1 に，構文解析を行うソースコード処理ツールの典型的なシステム構成を示す．字句解析は，ソースファイル中の文字の並びを，トークンと呼ばれるプログラミング言語で意味を持つ最小の単位の並びに変換する．その後，構文解析や意味解析によって，トークンの並びから，式，文，関数定義，クラス定義といった，より大きな構成要素を取り出す．ツールはその後，これらの構成要素に対する実行コードや依存関係を生成したり，型チェックや最適化といった処理を行う．

従来のコンパイラ・コンパイラは，字句解析器や構文解析器の生成を容易にする一方で，現状では以下のような制限や問題点がある．

1. ソースコード中に存在する複数の構文エラーを一度に検出することが必要とされている場合，構文解析器は構文エラーからの回復処理を行う必要がある．このようなエラー回復処理を記述するためには，構文解析器の内部処理に関する知識が必要であり，また，場合によってはエラー回復処理を見越して構文規則を定義する，などの手法も併用する必要がある [2].
2. 構文解析器は何らかの方法で前方参照を解決す

```
class C extends Object {
  Object value;
  C() {
    super();
    this.value = new Pair(new A(), new B()).setfst(new B());
  }
}
class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super();
    this.fst = fst;
    this.snd = snd;
  }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
}
（以下省略）
```

図 2　Fetherweight Java のソースコードの例

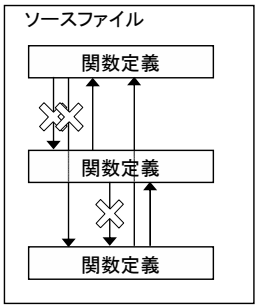

図 3 レキシカルスコープ

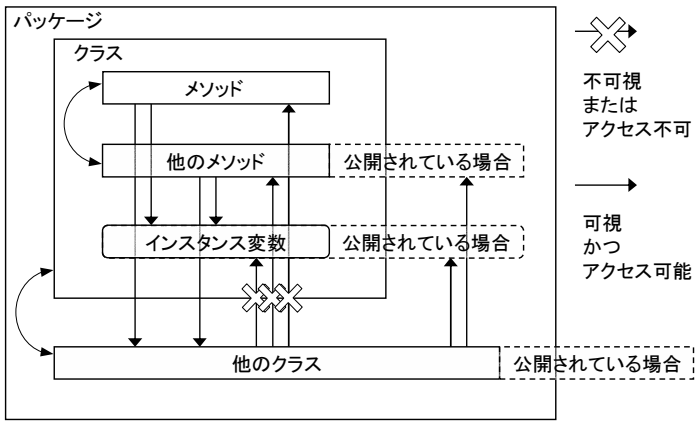

図 4 OOPL のスコープおよびアクセス制御

る必要がある．図 2 に，Java [10] の文法を簡略化した Featherweight Java [7] で記述されたサンプルプログラムを示す．下線を引いた部分は，ソースコード中に定義が現れる前に利用されているクラスやメソッドである．オブジェクト指向プログラミング言語では，手続き型言語 Pascal や C で基本となっているレキシカルスコープ (図 3) に代わって，クラスやパッケージによるスコープやアクセス制御 (図 4) を多用する傾向にあり，このような前方参照が，インスタンス変数，メソッド，クラスなどに対して起こる．

3. C++ [18] や Java といった OOPL の文法には，特定の識別子が型やパッケージの名前であることを字句解析器が区別できない場合に，構文規則が曖昧にな部分がある．

   a b(c); c が型ならプロトタイプ，c が変数なら変数宣言 (C++)

   t<a.b.c.d.e> t が汎用型であれば型，さもなければエラー (C++, Java 1.5 [9])

   (a.b.c.d.e) a.b.c.d.e が型であればキャスト演算子，e がインスタンス変数の名前であれば変数の参照 (C++, Java)

解決策としては，構文解析器はバックトラックを行うか（再帰下降構文解析器の場合），処理中にシンボルテーブルを生成するなどの手段によって字句解析器を制御するか，あるいは，構文解析の結果生成された AST(抽象構文木) を修正するなどの処理を行う必要がある．しかし，これらの解決策はいずれも構文解析器の開発に制約を課すことになる．バックトラックを用いるためには，いったん受理された部分のアクションの副作用をどのように取り消すかを考える必要がある．構文解析器が字句解析器を制御する方法では，構文解析器と字句解析器の間に相互依存が生じるために，ソースコード処理ツールのプログラムが複雑になる．AST を修正する方法では，構文解析が構文解析器（の拡張 BNF による記述）と AST 修正ルーチン（のプログラミング言語による記述）とに分散することになり，やはり複雑になる．

この問題の一部は，BNF や構文図（といった自由文脈文法を記述する [16] 記法）によって定義されたプログラミング言語を，より弱い文法クラスを処理する構文解析器を用いて構文解析することに起因する．例えば，代表的なコンパイラ・コンパイラである Yacc（あるいは Bison) [3] や SableCC [17] の構文解析器は $LALR(1)$, ANTLR [1] や JavaCC [2] の構文解析器は $LL(k)$ である．また，文献 [5] では，“The grammar for Java presented piecemeal in the preceding chapters is much better for exposition, but it cannot be parsed left-to-right with one token of lookahead (中略). These problems and the solutions adopted for the LALR(1) grammar

are presented below” と述べられている．

3 OOPL に適した構文解析

上述の制限や問題点は，従来のコンパイラ・コンパイラによって生成される構文解析器が，ソースコード内の出現順序に従ってトークンを処理する方法を用いていることに起因するところが大きい．この処理方法は，Pascal や C 言語で用いられるレキシカルスコープ (図 3)，つまり，そこに至るまでに出現したものが可視となるようなスコープを処理するのに都合がよい．一方で，OOPL のクラスやパッケージによるスコープとアクセス制御 (図 4) は，入れ子のレベルが同じかより外側のものは可視であり，内側のものは公開されない限り不可視であるという，外側から内側への階層性を持っている．

本稿では，(処理対象となっているプログラミング言語の) クラスやパッケージといったスコープと可視性制御の「外側から内側に向かう階層性」を，構文および意味解析を実装するクラスの継承階層によって実装する手法を提案する．このアプローチにより，上述の制限や問題点は以下のように解消・解決される．

1. 構文エラーからの回復処理を構造化例外処理として組み込むことが可能になり，単一のソースコードに含まれる複数個のエラーを検出することが容易である．
2. 前方参照の問題が生じない．
3. 字句解析器が変数名と型とを区別することが可能になり，曖昧または無限個の先読みが必要な文法を解消するための変形処理を容易に組み込むことができる．

また，本手法にはさらに以下のような利点もある．

- プログラミング言語の構文規則を（最初から全部作り込むのではなく）部分的な構文規則から順に作り込んでいくことで，繰り返し型開発に対応する．
- ツールの開発を通じて，個々の部分的な構文規則に対するテストケースを，開発の最後まで回帰テストのために用いることができる．

3.1 漸進的解析器

本稿で提案する構文および意味解析器（以降，構文および意味の解析を合わせたものを単に「解析」，本稿で提案する解析器を「漸進的解析器」と呼ぶ）の設計は，OOPL の構文および意味規則を，パッケージやクラス，メソッドといった，入れ子になったスコープを作り出す言語要素を境界として分割し，それぞれの分割に対応する構文および意味規則の部分（以降「部分規則」）を継承関係にある別個のクラスに実装するものである．さらに,「内側」の言語要素（たとえばメソッド）の解析を担当するクラスは,「外側」の言語要素（たとえばクラス）の解析を担当するクラスから導出される．実行時には，基底クラスが「外側」の言語要素を先に解析した後で，導出クラスがメソッドオーバーライドにより「内側」の言語要素の解析を行う．この設計により，基底クラスの意味解析により生成したスコープに基づいてトークン列を修正してから導出クラスが構文解析を行うことが可能になり，シンボルテーブルなどを利用した字句解析器を用いることなく，型や名前を区別した構文規則をそのまま利用することができる．また，導出クラスが構文エラーを例外として送出することにより，構文エラーの回復処理を構造化例外処理によって行うことが可能になる．

3.2 漸進的解析器の構成

図 5 に，クラス，インスタンス変数，メソッド（およびパラメータ），自動変数を持ち，強く型付けされた（すなわち，明示的な型の宣言を必要とする）OOPL を対象とする漸進的解析器のクラス階層と，処理全体の実行順序を示す．この図で，partialSrc とあるのは，ソースファイルの一部，全体，または複数のソースファイルの内容を保持するオブジェクトである．構文解析器は，部分規則に対応する 4 つのクラスから構成されている[†1]．解析は，以下の順序で進行する．

(1) PackageParser は，ソースコード中でクラス

†1 漸進的解析器は，構文解析だけではなく意味解析も行うため，クラス名やメソッド名が HogeParser, parseHoge() などとなっているのは不適切かもしれないが,「意味解析をする」を意味するうまい動詞が見つからないため，このままにしておく．

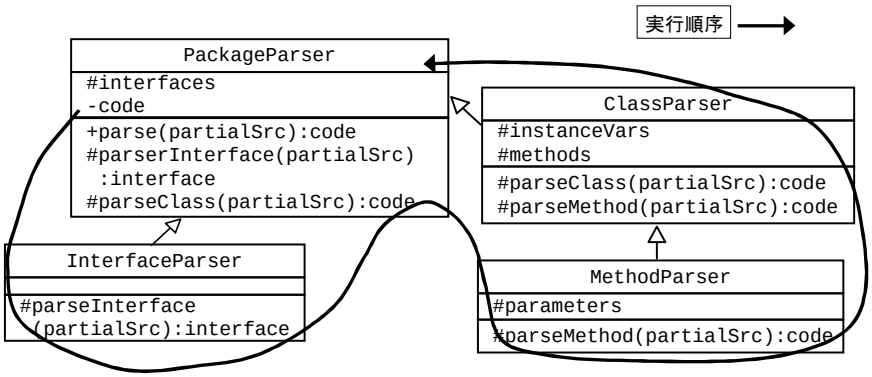

図 5 漸進的解析器の典型的なクラス階層

を定義している部分すべて見つけだし，parseInterface() を呼び出すことにより，それらを InterfaceParser に渡す.

(2) InterfaceParser はクラスのインターフェイス（公開メソッドのシグネチャや公開インスタンス変数の型）を調べて PackageParser に返す. PackageParser はその情報を interfaces に追加していく.

(3) PackageParser は，クラスを定義するソースコードの部分を，parseClass() を呼び出すことにより，ClassParser に渡す.

(4) ClassParser はすべてのメソッドのシグネチャとインスタンス変数の型を調べて，個々のメソッドを定義するソースコードの部分を，parseMethod() を呼び出すことにより，MethodParser に渡す.

(5) MethodParser はそれぞれのメソッドのオブジェクトコードを生成し，ClassParser に返す.

(6) ClassParser は受け取ったオブジェクトコードと，自らが生成したオブジェクトコードをあわせて，PackageParser に返す.

(7) 最後に，PackageParser は受け取ったオブジェクトコードをまとめて最終的なオブジェクトコードとする.

クラス階層間のデータの受け渡しは，オーバーライドされるメソッド (parse(), parseInterface(), parseClass(), parseMethod()，以降，これらをまとめて「parse*()」) のパラメータや戻り値 (partialSrc, interface, code) によるものと，protected インスタンス変数 (interfaces, instanceVars, methods) によるものがある．前者はメソッドが呼び出されるたびに違う内容が生成されて渡されること，後者はそのクラスの parse*() メソッドによって内容が生成された後，導出クラスの parse*() メソッドを（複数回）呼び出す間は変化しないことを意図している（従って，実装の際に必ずしもオーバーライドされるメソッドと，そのパラメータや protected インスタンス変数を用いる必要はなく，オーバーライドされるメソッドの代わりに委譲を用い，個々のデータに対応する getter/setter を用意しても良い）.

個々の構文解析器クラスの parse*() メソッドの内部処理は，以下の通りである.

(1) 受け取った partialSrc が保持するソースコードの部分を，基底クラスの protected インスタンス変数が保持するデータを参照しながら，構文解析を行う.

(2) 途中，そのクラスの導出クラスに渡すべき部分が出現した場合には，直ちにその部分を新たな partialSrc オブジェクトにして導出クラスの parse*() を呼び出すか，または，後で導出クラスを呼ぶためにその部分を記録しておく.

(3) 途中，構文エラーが検出された場合には，その構文エラーを例外として送出する.

(4) 導出クラスの parse*() の呼び出しが正常終了

```
1: PARSER_BEGIN(FWJParser)
2: public class FWJParser {
3:     public static void main(String[] args)
   throws ParseException {
4:         new FWJParser(System.in).Input();
5:     }
6: }
7: PARSER_END(FWJParser)
8: SKIP : {
9:     " " | "¥t" | "¥n" | "¥r" | "¥f"
10: }
11: TOKEN : {
12:     < LP: "(" > | < RP: ")" >
13: |   < LB: "{" > | < RB: "}" >
14: |   < COMMA: ","> | < SEMICOLON: ";" >
15: |   < DOT: "." > | < EQUAL: "=" >
16: |   < CLASS: "class" > | < EXTENDS: "extends" >
17: |   < NEW: "new" > | < RETURN: "return" >
18: |   < SUPER: "super" > | < THIS: "this" >
19: |   < ID: ( <LETTER> )+ >
20: |   < #LETTER: ["¥u0041"-"¥u005a", "¥u005f",
      "¥u0061"-"¥u007a" ] >
21: }
22: void Input() : { } { // 開始シンボル
23:     ( CL() )* <EOF>
24: }
25: void CL() : { } { // クラス定義
26:     <CLASS> <ID> <EXTENDS> <ID>
27:     <LB>
28:     ( LOOKAHEAD(2) <ID> <ID> <SEMICOLON> )*
29:     K() ( M() )*
30:     <RB>
31: }
32: void K() : { } { // コンストラクタ
33:     <ID> <LP> ( ParamL() )? <RP>
34:     <LB>
35:     <SUPER> <LP> ( ArgL() )? <RP> <SEMICOLON>
36:     ( <THIS> <DOT> <ID> <EQUAL> e() <SEMICOLON> )*
37:     <RB>
38: }
39: void ParamL() : { } { // パラメータの並び
40:     <ID> <ID> ( <COMMA> <ID> <ID> )*
41: }
42: void ArgL() : { } { // 実引数の並び
43:     e() ( LOOKAHEAD(2) <COMMA> e() )*
44: }
45: void M() : { } { // メソッド定義
46:     <ID> <ID> <LP> ( ParamL() )? <RP>
47:     <LB> <RETURN> e() <SEMICOLON>  <RB>
48: }
49: void e() : { } { // 式
50:     a() ( LOOKAHEAD(2) ( LOOKAHEAD(2) <DOT> <ID> )+
51:     ( <LP> ( ArgL() )? <RP> )? )?
52: |   <LP> <ID> <RP> e()
53: }
54: void a() : { } { // 式の一部
55:     <THIS> | <ID> | <NEW> <ID> <LP> ( ArgL() )? <RP>
56: }
```

図 6　Featherweight Java の構文規則を示すため JavaCC で記述された従来型の構文解析器

せず，構文エラーが送出された場合には，直ちにその構文エラーを再送出するか，あるいは，(複数個の構文エラーを検出するためには）その構文エラーを記録しておき，後にそれらの構文エラーをまとめた例外を作って送出する．

漸進的解析器のクラス階層は，開発者がツールの開発に当たって，基底クラスから順に部分規則やオブジェクトコード生成を実装することも意図したものにもなっている．たとえば，図 5 の場合なら，PackageParser，InterfaceParser，ClassParser，MethodParser の順に開発される．従来から，プログラミング言語の処理系の開発においては，経験的にこのような部分規則への分割が行われることがあるが，漸進的解析器ではそれに加えて，部分規則のそれぞれに対応するテストケースのテスト用ルーチンを，オーバーライドされる parse*() メソッドに閉じこめることができる（4.1 で具体的な例を示す）．これにより，いったん作成されたテストケースを，構文解析器の開発が進んだ後も引き続き回帰テストとして用いることが可能になる．

図 5 では，プログラミング言語の構文規則や意味規則を 4 つの部分規則に分割することを想定しているが，扱う文法の複雑さに応じて，さらに，MethodParser を StatementParser や ExpressionParser などに分割することも考えられる．

4　例題: Featherweight Java

本章では，例題として，Java を簡略化した文法を持つ Featherweight Java [7]（以降「FWJ」）に漸進的解析器を適用する．FWJ は元々Java の汎用型拡張である GJ の形式的検証のために提案された．従って，OOPL の特徴であるクラスやメソッドといった入れ子のスコープを持つ強く型付けされた言語でありながら，その文法は少数の構文規則で構成されている（FWJ のサンプルプログラムは図 2 を参照）．

図 6 に，JavaCC で記述した FWJ の字句規則および構文規則を示す．これは同時に，従来の手法で実装した構文解析器の例でもある．この構文解析器では，

入れ子になったスコープや前方参照の解決を一切行っていないが，構文規則が単純であるため，特に問題なく構文解析を行うことができる．クラス名，メソッド名，変数名は区別されることなく，識別子を意味する終端シンボル ID（19 行目で定義されている）で受理されており，名前の衝突などの意味解析は一切行っていない（たとえば，28 行目ではインスタンス変数の宣言を「識別子 識別子 セミコロン」というトークンの並びとして受理している）．また，入力となるソースコード中に複数の構文エラーが含まれている場合には，最初の構文エラーを検出した時点で終了する．

以降では，FWJ の漸進的解析器を，3.2 で示した漸進的解析器の構成とその実装順序に従って実装する．具体的には，クラスを境界として構文規則および意味規則を分割し，PackageParser と ClassParser を作成する．本稿ではソースコードの説明に必要な部分のみを示すが，ソースコード全体は http://sel.ist.osaka-u.ac.jp/ ~kamiya/ pparser/から入手可能である．

4.1 PackageParser の作成

図 7 の上段に，実際に作成した FWJPackageParser の (JavaCC の拡張 BNF によって記述された) 部分規則の構文定義を示す．字句規則は図 6 と同じである．BlockLike() は，中括弧の対応を数えることにより個々のクラスの定義に相当するソースコード部分を切り出すための規則である．図 7 の下段に，メソッド parse() および parseClass() の処理内容を示す．BlockLike() によって受理されるソースコード部分が，変数 classDefs に格納されていて，それらを引数として，protected メソッド parseClass() が呼ばれる（27 から 29 行目）．FWJPackageParser の parseClass() は，クラスのテストに用いることを目的として，切り出した個々のクラスの名前と基底クラスの名前，クラス定義の内部を印字する（37 から 52 行目）．次に作成する ClassParser では，このメソッドがオーバーライドされ，クラス定義の解析を行うルーチンに置き換えられる．

```
Input() :
    ( CL() ) * <EOF>
CL() :
   <CLASS> <ID> <EXTENDS> <ID> BlockLike()
BlockLike() :
   <LB>
   ( <ID> | <LP> | <RP> | <SUPER> | <THIS>
   | <DOT> | <EQUAL> | <SEMICOLON> | <COMMA>
   | <RETURN> | <NEW> | BlockLike() )*
   <RB>
```

```
1: PARSER_BEGIN(FWJPackageParser)
（省略）
9: public class FWJPackageParser
10: {
（省略）
21:   protected HashMap classDefs;
22:
23:   public final void parse() {
24:     try {
25:       classDefs = Input();
26:       for (Iterator i = classDefs.entrySet().iterator();
      i.hasNext(); ) {
27:         Map.Entry e = (Map.Entry) i.next();
28:         MyClassCode code = (MyClassCode)e.getValue();
29:         parseClass(code);
30:       }
31:     }
32:     catch (Exception e) {
33:       e.printStackTrace();
34:     }
35:   }
36:
37:   /* FWJPackageParserのテスト用。継承クラスでオーバー
      ライドすること */
38:   protected void parseClass(MyClassCode code) {
39:     try {
40:       System.out.println("class: " + code.className);
41:       System.out.println("super: " + code.superName);
42:       MyReader reader = new MyReader(code.body);
43:       int ch;
44:       while ((ch = reader.read()) > 0) {
45:         System.out.print((char)ch);
46:       }
47:       System.out.println("");
48:     }
49:     catch (Exception e) {
50:       e.printStackTrace();
51:     }
52:   }
53: }
(以下省略)
```

図 7 FWJPackageParser

4.2 ClassParser の作成

図 8 の左側に FWJClassParser の部分規則の構文定義を示す．字句規則に新しい種類の終端シンボル TYPEID が導入されていることに注意する．TYPEID はクラスの名前を表す識別子であり，FWJClassParser が構文解析を始める前に FWJPackageParser によって検出されていたすべてのクラス名が TYPEID とし

```
Input() :
   <LB>
   ( LOOKAHEAD(2) V() )*
   K() ( M() )*
   <RB>
V() :
   <TYPEID> <ID> <SEMICOLON>
K() :
   <TYPEID> <LP> ( ParamL() )? <RP>
   BlockLike()
ParamL() :
   <TYPEID> <ID>
   ( <COMMA> <TYPEID> <ID> )*
M() :
   ( <TYPEID> | <VOID> ) <ID>
   <LP> ( ParamL() )? <RP>
   BlockLike()
BlockLike() :
   <LB>
   ( <ID> | <TYPEID> |  <LP> |  <RP>
   |  <SUPER> |  <THIS>
   | <DOT> | <EQUAL>
   | <SEMICOLON> | <COMMA>
   | <RETURN> | <NEW> | BlockLike() )*
   <RB>
```

```
1: PARSER_BEGIN(FWJClassParser)
（省略）
9: public class FWJClassParser
10: {
11:   public static void main(String[] args) {
12:     try {
13:       FWJPackageParser parser = new FWJPackageParser(System.in) {
14:         private FWJClassParser parser;
15:         protected void parseClass(MyClassCode code) {
16:           try {
17:             TypeRecognizeReader reader
      = new TypeRecognizeReader(code.body);
（省略）
23:             MyClassData classData = parser.Input();
24:             parser.parseMethod(classData.ctor);
25:             for (Iterator i = classData.methods.entrySet().iterator(); i.hasNext(); ) {
26:               Map.Entry e = (Map.Entry) i.next();
27:               MyMethodCode methodCode = (MyMethodCode)e.getValue();
28:               parser.parseMethod(methodCode);
29:             }
30:           }
31:           catch (ParseException e) {
32:             System.err.println("error: in method " + code.className + ": "
      + e.toString());
33:           }
(省略)
37:         }
38:       };
39:       parser.parse();
40:     }
41:     catch (Exception e) {
42:       e.printStackTrace();
43:     }
44:   }
(以下省略)
```

図 8　FWJClassParser

て受理される．図 8 の右側に，FWJClassParser の parserClass() メソッドの内容を示す（3.2 の説明とは異なり，FWJClassParser は FWJPackagePaser の導出クラスになっていない．理由は後述）．17 行目の TypeRecognizeReader は，parseClass() の引数として受け取ったソースコードを変形するフィルタであり，型を表す識別子を修飾して，字句解析器に TYPEID として受理させる役目を持つ．24 から 29 行目で，K() および M() によって切り出されたコンストラクタやメソッド定義のソースコード部分を，parseMethod() に渡している．31 から 33 行目は構文エラーを処理する部分であり，parseClass() が構文エラーを送出したとき，それを標準エラー出力に印字して処理を続行する処理を記述している．

図 9 はこの解析器をテストするために作成されたエラーを含む入力の例，および，この解析器の出力である．この例には，型とそれ以外の名前とを区別する構文規則を用いた場合には構文エラーとして検出される 2 つのエラーが含まれている．出力例では期待通り 2 つのエラーが検出されていることがわかる（図 6 の構文解析器を用いた場合，これらのエラーは検出されない）．実際には，Java の言語仕様はクラス名とメソッド名前が同じであることを許すため，コンパイラ javac [10] は後者（12 行目）を構文エラーとしないが，もう一つの独立に実装された Eclipse のコンパイラ [12] は警告を出力する．

実装を通じて判明した技術的な問題点としては以下のものがある．

- 3.2 の説明とは異なり，FWJClassParser を FWJPackageParser の導出クラスにせず，代わりに，匿名内部クラスを用いて parseClass() をオーバーライドしている．これは，両クラスを

```
 1: class A extends Object {
 2:   C() { super(); }
 3: }
 4: class Pair extends Object {
 5:   Object fst;
 6:   Object snd;
 7:   Pair(Object fst, Object snd) {
 8:     super();
 9:     this.fst = fst;
10:     this.snd = snd;
11:   }
12:   Pair Pair(Object newfst) {
13:     return new Pair(newfst, this.snd);
14:   }
15: }
```

```
error: in method Pair:
classparser.ParseException:
Encountered "type:Pair" at line 1,
column 151.
Was expecting:
    <ID> ...

error: in method A:
classparser.ParseException:
Encountered "C" at line 1, column 3.
Was expecting:
    <TYPEID> ...
```

図 9 エラーを含む入力例と FWJClassParser の出力

JavaCC によって生成した場合，両クラスに継承関係があると名前の衝突が起きるためである．

- 図 9 のエラーメッセージの中で不正な行番号が表示されている．これは，構文解析器によって切り出された部分的なソースコードの特定の行の，元のソースコードでの行番号を調べる簡便な手段が見あたらなかったためである．

5 まとめと課題

本稿で提案した構文および意味解析器の新しい設計「漸進的解析器」により，OOPL の入れ子になったスコープルールをクラス階層によって実現し，構文エラーからの回復を構造化例外によって処理することが可能になる．さらに，シンボルテーブルなどの手段を用いることなく特定の識別子が型であるかを字句解析器が判定することが可能になり，識別子が型であるか否かで異なった構文であると見なす構文規則を利用することができる．漸進的解析器の設計は，これらの問題を，オブジェクト指向パラダイムの枠組みの中で見通しよく解決するものであり，また同時に，繰り返し型開発，テスト駆動型開発とも親和性がある．

漸進的解析器は再帰下降構文解析器によっても開発可能であるが，従来のコンパイラ・コンパイラを用いても開発できることを示すために，4 章の例題では JavaCC を用いて Featherweight Java の漸進的解析器を作成した．開発の途中で判明した実装上の問題は，コンパイラ・コンパイラをその開発者が想定しない方法で用いたため生じた技術的な問題であり，いずれ解決するものと楽観している．

同例題では，本質的に曖昧な名前（例えば，クラスとメソッドが同じ名前を持つ場合など）は，漸進的解析器ではうまく扱えないことも判明した．ただし，このような状況は Java 言語の名前付け規約に従う限りほとんど発生しない [6] し，また，構文エラーにはならないまでもコンパイラによっては警告が出力されるものである．なお，C/C++においては，型とメンバが同じ名前を持つ場合等にも，このような曖昧な名前が生じる．

漸進的解析器の開発では，構文規則と意味規則の全体が用意されておらず，部分規則に対応する構文解析器の開発を通じて段階的に構文規則を生成していくことを想定している．課題としては，4 章の例題のように構文規則と意味規則の全体が与えられているときに，それを部分規則に分割する作業をサポートする手法の開発があげられる．特に，前段の構文解析器では識別子として受理されていたものが，後段の構文解析器では型名や変数名，メソッド名として受理される部分では，Data Reification [15] 等の技術を応用することが考えられる．

6 謝辞

本論文の匿名査読者の方々に，心より感謝いたします．ご指摘により，本論文に含まれていた重大な誤りや術語の混乱を修正することができました．

参考文献

[1] ANTLR Home Page, http://www.antlr.org/.
[2] Appel, A. and Palsberg, J., *Modern Compiler Implementation in Java, 2nd Ed.*, Cambridge University Press (2002).
[3] Bison 1.35

http://www.gnu.org/software/bison/manual/, Free Software Foundation.
[4] Boost.Spirit Home, http://spirit.sourceforge.net/.
[5] Gosling, J., Joy, B., Steele, G., Java Language Specification, First Edition, Chapter 19, LALR(1) Grammar, Taken from http://java.sun.com/docs/books/jls/first_edition/html/19.doc.html
[6] ゴスリン, G., ジョイ, B., スティール, G., ブラーハ, G.(著), 村上雅章 (訳), Java 言語仕様 第 2 版, ピアソン・エデュケーション (2000), Gosling, J., Joy, B., Steele, G., *Java Language Specification, 2nd Ed.*, Pearson Education (2000).
[7] Igarashi, A., Pierce, B., and Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ, *ACM Transactions on Programming Languages and Systems*, 23(3), pp. 396-450 (2001).
[8] 泉田聡介, 植田泰士, 神谷年洋, 楠本真二, 井上克郎: ソフトウェア保守のための類似コード片検索ツール, 電子情報通信学会論文誌, vol. J86-D-I, no. 12, pp. 906-908 (2003).
[9] J2SE 1.5, http://java.sun.com/j2se/1.5/.
[10] Java Technology, http://java.sun.com/.
[11] JavaCC Home, https://javacc.dev.java.net/.
[12] JDT, Eclipse Project, http://www.eclipse.org/jdt/.
[13] Levine, J., Mason, T., and Brown, D., *Lex & Yacc, 2nd Ed.*, O'Reilly & Associates (1992).
[14] Ohst, D., and Kelter, U. : A Fine-grained Version and Configuration Model in Analysis and Design, *Proceedings of the IEEE 18th International Conference on Software Maintenance (ICSM2002)*, pp. 521-527 (2002).
[15] Ozaki, H., Ban, S., and Gondow, K.: An Environment for Evolutionary Prototyping Java Programs Based on Abstract Interpretation, *Proceedings of the IEEE 5th Asia Pacific Structural Engineering & Construction Conference (APSEC2003)*, pp. 362-370 (2003).
[16] 都倉信樹, オートマトンと形式言語, 昭晃堂 (1995).
[17] SableCC, http://www.sablecc.org/.
[18] Stroustrup, B. 著, 株式会社ロングテール, 長尾高弘訳, プログラミング言語 C++ 第 3 版, アジソン・ウェスレイ (1998). Stroustrup, B., *The C++ Programming Language 3rd Ed.*, Addison-Wesley, (1997).
[19] 吉田敦, 山本一郎, 阿草滋: 意味を考慮した差分抽出ツール, 情報処理学会論文誌, vol. 38, no. 6, pp.1163-1171 (1997).