クラスオブジェクトを用いたJava言語用マクロ処理系

立 堀 道 昭[†]　千 葉　滋[††,†††]　板 野 肯 三[††]

本稿では著者らの開発した Java 言語のためのマクロ処理系 OpenJava について述べる．Java のようなオブジェクト指向言語では，手続きや関数でなく，クラスやメソッドが主要な言語要素となる．このため，従来のようなマクロ処理系では，オブジェクト指向プログラミングで本来必要とされるマクロ展開をうまく記述できない．本稿では，まず，この問題を指摘し，次に OpenJava がこの問題にどのように対処しているかを述べる．本研究では，従来のマクロ処理系の問題点が，トークン列や構文木を操作の対象としている点に起因すると考え，OpenJava ではクラスオブジェクトというデータ構造を処理の対象として採用した．オブジェクト指向言語に基づく高度なマクロの例として，デザインパターンを用いたプログラミングを支援するマクロを OpenJava で記述した例を示す．

A Macro System with Class Objects for the Java Language

Michiaki Tatsubori,[†] Shigeru Chiba[††,†††] and Kozo Itano[††]

This paper presents OpenJava, which is a macro system the authors have developed for Java. Writing a number of typical macros in object-oriented programming is difficult with traditional macro systems designed for non object-oriented languages. This is because the primary language constructs of object-oriented languages uch as Java are not procedures or functions but they are classes and methods. This paper first points out this problem and then shows how OpenJava is addressed to the problem. One of the drawbacks of traditional macro systems is that syntax trees are used for representing source programs. For OpenJava, therefore, class objects were chosen instead of syntax trees. As high-level macros for an object-oriented language, this paper shows a few macros in OpenJava which help programming with design patterns.

1. はじめに

従来から，マクロ機構は，プログラムの可読性やコードの再利用性を向上させる言語機構として様々なプログラミング言語に採り入れられてきた．本稿では，著者らが開発した，Java 言語[10] 用のマクロ処理系 OpenJava について述べる．その特徴は，ソースプログラムの変換の枠組みとして，クラスオブジェクトと呼ぶデータ構造を提供していることである．これによって，従来は難しかったオブジェクト指向言語用の高度なマクロを記述することが容易になった．たとえば，デザインパターンを利用するプログラミングを支援するためのマクロはクラスに関する構文的に複雑な情報を調査しなければならないが，OpenJava ではクラスオブジェクトのおかげでそのようなマクロを簡潔に記述することができる．著者らは，このようなマクロ機構が，オブジェクト指向言語による複雑なプログラミングを支援する有効な手段の 1 つとなると考えている．

マクロ機構　Brown[3] によれば，マクロとは記号のある系列を別の系列に置き換える機能のことである．OpenJava のマクロは，その中でも基底言語のコンパイラへの完全なプリプロセッサとして動作する種類のマクロであり，従来のコンパイラや Java 仮想機械をそのまま活用できる．この種のマクロの単純な例は C/C++の `#define` マクロにみることができる．C/C++では，

```
#define E E'
```

とすることによって，以降のプログラム中で現れる字句列 E は字句列 E' により置き換えられる．ある意味を表す表現 E' を表現 E に割り当てることができるので，マクロはソースコード変換により実現される言語拡張機構の一種である．

上の例では，元の表現に対して置き換わる表現は固

† 筑波大学大学院工学研究科
Doctral Program in Engineering, University of Tsukuba
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba
††† 科学技術振興事業団さきがけ研究 21
PRESTO, Japan Science and Technology Corporation

定であるが，Common Lisp [20] にみられるようなプログラム可能なマクロ機構では，元の表現をどのように置き換えるかを，処理対象となる言語自体を使ってプログラムすることができる．たとえば，Common Lisp のマクロは，元の表現を表す構文木を受け取り，これをもとに作った別の構文木を結果として返すプログラムであり，Common Lisp で記述される．マクロ処理系は，マクロの返した構文木が表す表現で，元の表現を置き換える．このように元の表現をどう置き換えるか，つまりマクロ展開，を記述したプログラムのことを本稿では**メタプログラム**と呼ぶ．

オブジェクト指向言語のマクロ Java のようなオブジェクト指向言語のプログラムでは，C や Common Lisp でのプログラムで典型的にみられるようなマクロ展開よりも，より高度なマクロ展開が必要になることが多い．たとえばデザインパターンに沿ったプログラミングを考えてみる．プログラマは，デザインパターンカタログ[7] のパターンに沿ってプログラミングをすることにより，Java 言語にない高度な抽象モデルを洗練されたデザインで実現することができ，その有用性は広く受け入れられている．一方で，実装の際には似たようなコードを繰り返し書かなければならないなどの問題が存在し[1,19]，デザインパターンを支援する構文を新たに言語に導入することで解決する提案がなされてきている[1,8]．

そのような解法を採用する場合，特定のデザインパターンの利用を支援するための構文は言語に直接埋め込むのではなく，必要に応じてマクロとして提供できれば便利である．多岐にわたるデザインパターンごとの支援構文をすべてあらかじめ言語に組み込んでおくことは，言語仕様の肥大化を招く．またその場合，後から別の支援構文を採り入れるたびに言語仕様を変えなければならなくなる．

しかしながら，現状のマクロ処理系では，Common Lisp のマクロのような比較的強力なマクロ処理系であっても，デザインパターンの利用を支援できるような高度なマクロを記述することは困難である．これは，メタプログラムの処理する対象が構文木であるためである．このような高度なマクロは，あるクラスでどのようなメソッドが定義されているかなど，プログラムの論理構造を読み取って，プログラムの変換を行わなければならない．少なくとも，局所的な構文木だけを用いてこのような変換を行うのは不可能である．また，構文木に関する環境情報が得られるような処理系でも容易ではない．

OpenJava 著者らはソースコードプログラムを表す抽象構文木というデータ構造がオブジェクト指向言語で書かれたプログラムの論理構造から離れてしまっていることを問題視した．そこで OpenJava のマクロ機構では，プログラムの論理構造を表すために，**クラスオブジェクト**というデータ構造を提供する．OpenJava ではメタプログラムを，Java 言語を使いオブジェクト指向で記述する．メタプログラムは，クラスオブジェクトのメソッドを通してソースプログラムの論理構造にアクセスすることができる．またクラスオブジェクトの標準クラスのサブクラスを書くことで，特定のクラスに関係する部分のプログラムだけを選択的にマクロ展開することができる．

本稿の流れ 以後，2 章で従来のマクロシステムの問題点を提起し，3 章でその問題点を解決する OpenJava の設計と実装を説明する．このとき，いくつかのマクロの応用例をあげて本処理系の有用性を示す．4 章で他の研究との比較を論じ，5 章で全体をまとめる．

2. 従来のマクロとその問題点

マクロは言語を拡張する普遍的な言語機構であり，古くから存在する．ANSI C/C++の `#define` マクロでは，シンボルまたは関数呼び出し形式の式を指定して，そのシンボルがプログラム中に現れたとき，別の字句列で置き換えることができる．しかし，置き換えられる字句列はシンボルごとに固定である．Common Lisp にみられる種のマクロではより強力なメタプログラムを記述できるが，オブジェクト指向言語を対象とする場合には必ずしも適さないことが分かってきた．

2.1 プログラム可能な構文マクロ

C/C++のマクロと異なり，Common Lisp にみられるようなマクロでは，マクロ展開の結果，元の字句列がどのような字句列で置き換えられるかを Common Lisp を用いてプログラムすることができる．Common Lisp のマクロ関数は，元の字句列に対応する構文木を引数として受け取り，別の構文木を返す関数である．マクロ処理系は返された構文木で元の字句列を置き換える．このようなマクロ処理系は非常に強力であり，たとえば Common Lisp のオブジェクト機構（CLOS）は，マクロ処理系を使って Common Lisp の上に実装されている．

同様のプログラム可能なマクロ処理系は，C 言語のような Common Lisp よりも複雑な構文を持つ言語についても開発されている．たとえば Weise ら[21] による MS^2 が例としてあげられる．MS^2 のマクロは構文木を組み込みデータ型として持つ擬似的な C 言語で記述される．ユーザはマクロを使って新しい構文を

定義し，それが実際にどのように展開されるかを記述することができる．マクロの引数は，新しく定義した構文に対応する構文木である．

プログラム可能なマクロを設計するうえで重要な点の1つは，マクロ引数として何を与えるか，すなわち元のプログラムをどのようなデータ構造で表現するか，である．そしてもう1つ重要な点は，マクロをプログラム中のどこに適用するか，その適用場所の指定の仕方である．前者についてはほとんどの処理系が構文木を採用しているが，後者についてはいくつか異なる方法が提案されている．

Common Lisp や MS^2 のマクロでは，あらかじめ指定されたマクロ名から始まる構文すべてについてマクロが適用される．たとえばマクロ名として `unless` を指定すれば， `unless` で始まるすべての文がマクロ展開される．ところがこの方法では，マクロ名の挿入なしに，非明示的にマクロ展開を指定することができない．たとえば，プログラム中に現れる `+` 式をいくつか選択的にマクロ展開するように指定することは困難である．

マクロ処理系によっては，マクロの適用場所の条件を詳細に記述できるようになっているものもある．たとえば A^* [16] では，マクロはマクロ名から始まる文や式に適用されるのではなく，あらかじめ BNF を使って指定されたパターンに一致した文や式に適用される．また，EPP [11] ではマクロは， `if` 文や `+` 式など，指定された構文要素ごとに適用される．マクロ名を挿入する必要はない．

2.2 オブジェクト指向言語への適用

オブジェクト指向言語の分野では，デザインパターンを使いこなしてプログラミングすることが重要になってきている．しかしながらいくつかのパターンは，その言語に標準で用意されている構文，言語機構だけを使って，直観的に記述することができない．デザインパターンの利用を支援する方法はいろいろありうるが，パターンごとの専用構文を用意することでプログラミングを支援する方法が提案されている[1),8)]．ここではマクロでそのような支援を行うことを考える．プログラマには，より直観的で簡潔な記述をさせ，それをマクロ機構によって，標準の構文と言語機構だけを使ったプログラムに変換すればよい．

ここでは，一種の Observer [7] パターンが用いられた Java 標準のクラスライブラリのイベントハンドラを実装するプログラミングを考えてみる．Observer パターンは，あるオブジェクトの状態が変化したときにそれに依存する複数のオブジェクトに通知がされ自動的に更新されるようにするパターンである．通常，通知に使われるメソッドはオブジェクトにつき1つ用意されるが，このライブラリでは，大まかなイベントの種類で分けるために複数用意されている．

ここでは，Java 標準のクラスライブラリで用意されたインタフェース MenuListener が，Observer パターンで示される Observer にあたり，状態変更の通知に使われる複数のメソッドのインタフェースを定めている．プログラマは，パターンの ConcreteObserver にあたるクラス MyMenuListener でこれらのメソッドを実装することになる．クラス MyMenuListener はインタフェース MenuListener 中に宣言されているメソッドをすべて実装しなければならない．ところが，少数のメソッドで重要な処理を行うだけの場合，ほとんどのメソッドは何もしないメソッドでよい．

そこで，たとえば**図 1** に示すように特別な構文 `follows` を導入して記述を簡略化できると便利である．この構文をマクロを使って実装するとすれば，そのようなマクロは図1のプログラムを**図 2** のように展開することになるであろう． MenuListener で宣言されているメソッドのうち，図1で実装されていないメソッドは自動的に追加される．このマクロの特徴は，クラス MyMenuListener とインタフェース MenuListener の定義を調べて，挿入しなければならないメソッドを決定している点である．オブジェクト指向言語の観点からは単純にみえるこの処理だが，従来の処理系で記述することは決して容易ではない． MyMenuListener はスーパークラス MyObject からメソッドを継承しているので， MyObject の定義も調べなければならないからである．

```
class MyMenuListener follows ObserverPattern
   extends MyObject
   implements MenuListener
{
   void menuSelected(MenuEvent e) {
      //something to do
   }
}
```

図 1 マクロ展開されるソースプログラム
Fig. 1 Source program to be expanded by macro.

```
class MyMenuListener
   extends MyObject
   implements MenuListener
{
   void menuSelected(MenuEvent e) {
      //something to do
   }
   void menuDeselected(MenuEvent e) { return; }
   void menuCanceled(MenuEvent e) { return; }
}
```

図 2 マクロ展開されたプログラム
Fig. 2 Program expanded by macro.

2.3 従来のマクロの問題点

オブジェクト指向プログラミングでは，上で示したようなマクロが望まれているにもかかわらず，従来のマクロ機構では，そのようなマクロを簡単に実現することが難しい．従来のマクロ機構は，手続きや関数が主要な要素であるプログラミング言語を対象に開発されており，マクロ処理の対象がトークン列や構文木など，比較的低レベルのものであるためである．

2.2 節の例では，マクロはクラス MyMenuListener の定義を解釈して，どのメソッドが実装されていないか判別しなければならない．ところがこのような比較的高次の解釈をマクロで行うための支援は，従来のマクロ機構では提供されてこなかった．また実装されていないと分かったメソッド `menuDeselected()` を MyMenuListener 中の正しい位置に挿入する処理も，クラス定義を表す複雑な構文木を操作して行わなければならないので，その記述は決して容易ではない．このような処理は BNF によるパターンマッチなどではうまく記述できない．

このようなマクロを記述できるようにするためには，マクロ処理系が何らかの形で，プログラムの意味的な情報，たとえば名前と型の束縛情報など，直接扱えるようにする必要がある．ところが，我々の知る限り，意味的な情報を扱うマクロの研究は多くない．Maddox による XL [17] はそのようなマクロの 1 つであるが，対象言語はオブジェクト指向言語ではない．

3. OpenJava のマクロ機構

従来のマクロ処理系が操作の対象とする抽象構文木というデータ構造は，対象言語が Lisp や C など主な言語要素が手続きや関数であるものについてはたまたま問題が少なく，十分に有用であった．しかし，Java のような言語用のマクロでオブジェクト指向言語に基づく高度な変換を行うためには，メタプログラムがクラスという言語要素を直接扱えることが重要になってくる．クラスは継承やメンバ，メソッドなどの記述により定義され，宣言的である．これらの言語要素は構文木として扱うには，意味的にも構文的にもあまりに複雑である．クラスの定義は宣言的であるために，従来のマクロでこれを調べて変更しようとすると，少なくとも，クラス宣言全体の巨大な構文木を取り扱わなければならなくなる．しかも，クラスの継承という言語機構が存在するために，そのクラスにどのようなメソッドが定義されているかは，クラス宣言の構文情報だけから調べるのは不可能である．この問題を解決するために著者らは，クラスオブジェクトと呼ぶデータ構造を採用したマクロ処理系，OpenJava を開発した．

3.1 OpenJava のマクロプログラミング

OpenJava の処理系は，ソースコード中に新たなクラスが現れるごとに，そのクラス定義の論理構造を表すオブジェクトを生成する．このオブジェクトをクラスオブジェクトと呼ぶ．クラスオブジェクトは，同時に，ソースコード中のそのクラスに直接関連する部分のマクロ展開を司る．プログラマは，クラスオブジェクトのクラス（メタクラス）の定義を修正し，マクロ展開を実行するメソッドを実装することによって，マクロを記述する．処理系は，必要に応じてそれらのメソッドを呼び出し，ソースコードを変換していく．

本稿で，あるクラスに直接関連する部分とは，そのクラス宣言の部分（呼ばれる側）と，クラスとそのオブジェクトを利用している部分（呼ぶ側）の 2 つの部分であるとする．まずは，呼ばれる側のマクロ展開を取り上げて，OpenJava のマクロ処理系について述べる．

マクロの適用

図 3 に示すのは OpenJava のマクロをクラス MyMenuListener に適用したソースコードである．

クラス宣言のソースコード中，クラス名の直後に `instantiates M` という節を付加することによって，そのクラスを表すクラスオブジェクトがメタクラス M のインスタンスであることを指示することができる．マクロ展開はメタクラス M の定義に従ってクラスオブジェクトが行う．OpenJava では，クラスオブジェクトはクラス定義の論理構造を表すだけでなく，マクロ展開を行うメタプログラムをメソッドとして持つ．上の例では，メタクラスは ObserverClass であり，**図 4** のように Java または OpenJava により拡張された Java を使って記述される．

すべてのメタクラスは OpenJava が提供するメタクラス OJClass を継承しなければならない．図 4 の `translateDefinition()` は，処理系からマクロ展開を行うために呼び出されるメソッドの 1 つである．処理系は，クラス宣言の中に `instantiates` 節を見つけると，その節で指定されたメタクラスのオ

```
class MyMenuListener
    instantiates ObserverClass
    extends MyObject
    implements MenuListener
{ .... }
```

図 3 OpenJava のマクロ適用
Fig. 3 Application of a macro in OpenJava.

```
class ObserverClass instantiates Metaclass
    extends OJClass
{
    void translateDefinition() {
        OJMethod[] m = this.getMethods(this);
        for (int i = 0; i < m.length; ++i) {
            OJModifier modif = m[i].getModifiers();
            if (modif.isAbstract()) {
                OJMethod n = new OJMethod(this,
                    m[i].getModifiers().removeAbstract(),
                    m[i].getReturnType(), m[i].getName(),
                    m[i].getParameterTypes(),
                    m[i].getExceptionTypes(),
                    makeStatementList("return;"));
                this.addMethod(n);
            }
        }
    }
}
```

図 4 図 2 の変換を実装した OpenJava のマクロ
Fig. 4 An OpenJava macro implementing the translation in Fig. 2.

ブジェクトをこのクラスの定義を表すクラスオブジェクトとする．そして後に，そのクラスオブジェクトの translateDefinition() を呼び出す． OJClass で定義されている translateDefinition() は何も行わないが，サブクラスではこのメソッドを上書きして，必要なマクロ処理を行わせることができる．たとえばこのメソッドの中から，別のメソッドを呼び出して，クラスオブジェクトに新たなメソッドを追加することができる．クラスオブジェクトに加えられた変更は，処理系によって最後にソースコードに反映される．

図 4 では，メタクラス ObserverClass のメタクラスが Metaclass となっている． Metaclass は処理系に標準で用意されているメタクラスのためのメタクラスで，やはり， OJClass のサブクラスである． Metaclass で定義されているマクロにより，メタプログラムの記述が支援される．このマクロは，本処理系のフレームワークで利用されるコンストラクタなどを自動的に実装するなどの支援を行う．

メタプログラムの記述

2.2 節の例を実現するメタクラス ObserverClass のメソッド translateDefinition() は，**図 4** のようになる．このメタプログラムは，まずクラスオブジェクトに対して getMethods() を呼び出してクラスの持つメソッドを取り出し，それぞれのメソッドが未実装の場合，何もしないメソッドを生成して addMethod() によりクラスに追加している．

クラスがオブジェクトで表されるだけでなく，メソッドもオブジェクトで表されていることが分かる．OpenJava では，クラス，メソッド，フィールド，コンストラクタが，それぞれ OJClass，OJMethod，OJField，OJConstructor クラスのオブジェクトで表される．これらのオブジェクトは，クラスやそのメンバの定義の論理構造を表し，クラス宣言の巨大な構文木の情報とプログラム中に分散する情報をまとめ，抽象化して取り扱いやすくしている．

3.2 クラスオブジェクト

著者らは，メタプログラムが扱う従来の抽象構文木というデータ構造がオブジェクト指向言語で書かれるプログラムの論理構造から離れていることを問題視している．Java のようなオブジェクト指向言語では，クラス定義がプログラムの論理構造として重要な役割を持つ．そこで OpenJava では，クラス定義の論理構造を表すデータ構造として，クラスオブジェクトというモデルを採用した．クラスオブジェクトは構文木を抽象化するのに加え，クラス階層の情報を含んだクラス定義の論理構造を直接表現する．これにより，図 4 のように高度な変換を行うメタプログラムの記述が簡潔になる．

構文情報の隠蔽

Java の文法では，言語の論理構造上同一の事柄を，複数の表現で記述することができる．これらの異なる表現は同一のデータ構造で表されたほうが，メタプログラムの記述は簡潔になる．たとえば，配列型のフィールドを宣言するには次の 2 通りがある．

```
String[] a;
String b[];
```

a と b はどちらも String の配列型である．メタプログラムでこの構文上の差異を意識しなければならないと記述が複雑になるので，OJField オブジェクトはフィールドを指定する型を操作するために， getType() と setType() の 2 つのメソッドのみを提供するべきである．a と b のどちらのフィールドについても getType() は String の配列型を表すクラスオブジェクトを返す．

また，言語の文法上は異なる構文要素であっても，言語の論理構造からは同一の要素であるものが多い．このようなものは，一括して変更されるように操作を定めるべきである．たとえば，クラスの名前を変更する setName() は，キーワード class に続くクラス名だけでなく，コンストラクタの名前もともに変更する必要がある．

クラス定義の論理構造の表現

構文木のデータ構造を整理，抽象化していくだけでは，クラス定義の論理構造を表現するのに適切なデータ構造にはならない．言語の文法ではなく，クラスやメソッドなどの言語の論理的な構成要素に直接対応するように，データ構造を注意深く設計する必要がある．

表 1 クラス以外の型を認識するための OJClass のメソッド
Table 1 Methods of OJClass to tell a non-class type.

メソッド	操作の内容
boolean isInterface()	インタフェース型を表しているかを調べる
boolean isArray()	配列型を表しているかを調べる
boolean isPrimitive()	プリミティブ型を表しているかを調べる
OJClass getComponentType()	配列の構成要素の型を得る

表 2 クラスを調べるために OJClass が持つメソッド（1）
Table 2 Methods of OJClass to look into the class (1).

メソッド		操作の内容
String	getName()	名前を得る
OJModifier	getModifiers()	クラス修飾子群を得る
OJClass	getSuperclass()	暗黙的または明示的に宣言されたスーパークラスを得る
OJClass[]	getDeclaredInterfaces()	宣言されたスーパーインタフェースのすべての並びを得る
StatementList	getInitializer()	静的初期化子の文の並びを得る
OJField[]	getDeclaredFields()	宣言されたフィールドをすべて得る
OJMethod[]	getDeclaredMethods()	宣言されたメソッドをすべて得る
OJConstructor[]	getDeclaredConstructors()	宣言されたコンストラクタをすべて得る
OJClass[]	getDeclaredClasses()	宣言された内部クラスをすべて得る
OJClass	getDeclaringClass()	このクラスを宣言している外側のクラスを得る

表 3 クラスに変更を加えるための OJClass のメソッド
Table 3 Methods of OJClass to modify the class.

メソッド		操作の内容
String	setName(String name)	名前を定める
OJModifier	setModifiers(OJModifier modifs)	クラス修飾子群を定める
OJClass	setSuperclass(OJClass clazz)	暗黙的または明示的に宣言するスーパークラスを定める
OJClass[]	setInterfaces(OJClass[] faces)	宣言するスーパーインタフェースのすべての並びを定める
OJField	removeField(OJField field)	宣言されたフィールドを取り除く
OJMethod	removeMethod(OJMethod method)	宣言されたメソッドを取り除く
OJConstructor	removeConstructor(OJConstructor constr)	宣言されたコンストラクタを取り除く
OJField	addField(OJField field)	宣言するフィールドを追加する
OJMethod	addMethod(OJMethod method)	宣言するメソッドを追加する
OJConstructor	addConstructor(OJConstructor constr)	宣言するコンストラクタを追加する

特に，名前と型の対応などの環境情報を内部データに組み込んで，適切にプログラムの論理的な情報を操作できるようにしなければならない．そのようなデータ構造は，従来，コンパイラの内部などで使われることはあったが，マクロ処理系には使われてこなかった．

たとえば，クラスオブジェクトの getMethods() を呼んで，そのクラスが持つすべてのメソッドを得ることを考える．クラスオブジェクトをクラス定義の論理的な構成要素と考えると，そのクラスが継承しているものも含めたすべてのメソッドを返すべきである．しかしながら，そのクラスの宣言に対応した構文木だけからでは，そのような getMethods() は実装できない．実装のためには，そのクラスのスーパークラスの定義を調べる必要があるので，スーパークラスの名前とその定義を対応づけた環境情報を，そのクラスオブジェクトが内部で保持している必要がある．

3.3 クラスオブジェクトの詳細

クラスオブジェクトのクラスは OJClass である．OJClass のメソッドのうち，クラスの情報を得るための基本的なものを**表 1**，**2** に示す．クラスのすべての属性を網羅している．OpenJava では， int などのプリミティブ型を含むすべての型にクラスオブジェクトが対応づけられるので，クラスでない型を識別できるように，表 1 のメソッドが用意されている．

表 3 は，クラスの定義を変更するメソッドである．メタプログラムは translateDefinition() を上書きして，その中からだけこれらのメソッドを呼んでクラス宣言のソースコードを変換することができる．図 4 では，必要に応じて新しく作った何もしないメソッドを addMethod() を使ってクラスに追加している．

クラスオブジェクトから得られるオブジェクト

クラスオブジェクトは，名前と型の対応などの環境情報を保持しているので，表 2 のスーパークラスを

表 4 OJMethod が持つ基本的なメソッド
Table 4 Basic methods of OJMethod.

メソッド		操作の内容
String	getName()	名前を得る
OJModifier	getModifiers()	メソッド修飾子群を得る
OJClass	getReturnType()	返り値の型を得る
OJClass[]	getParameterTypes()	仮引数の型の並びを得る
String[]	getParameterVariables()	仮引数の変数名の並びを得る
OJClass[]	getExceptionTypes()	スローされうる例外の型をすべて得る
StatementList	getBody()	メソッド本体を得る
OJClass	getDeclaringClass()	このメソッドを宣言しているクラスを得る
String	setName(String name)	名前を定める
OJModifier	setModifiers(OJModifier modifs)	メソッド修飾子群を定める
OJClass	setReturnType(OJClass type)	返り値の型を定める
OJClass[]	setParameterTypes(OJClass[] types)	仮引数の型の並びを定める
String[]	setParameterVariables(String[] params)	仮引数の変数名の並びを定める
OJClass[]	setExceptionTypes(OJClass[] types)	スローされうる例外の型をすべて定める
StatementList	setBody(StatementList body)	メソッド本体を定める

表 5 クラスを調べるために OJClass が持つメソッド（2）
Table 5 Methods of OJClass to look into the class (2).

メソッド		操作の内容
OJClass[]	getInterfaces()	スーパーインタフェースの並びをすべて得る
boolean	isAssignableFrom(OJClass clazz)	与えたクラスのオブジェクトをこのクラスとして扱えるかを調べる
OJMethod[]	getMethods(OJClass situation)	与えたクラスから利用できるメソッドをすべて得る
OJMethod	getMethod(String name, OJClass[] types)	与えたシグネチャのメソッドを得る
OJMethod[]	getInvokedMethod(String name, OJClass[] types)	与えた名前と実引数の型の並びにより呼び出されるメソッドを得る

返す getSuperclass() などはクラス名（文字列オブジェクト）ではなく，クラスオブジェクトを直接返すようになっている．このため返されたクラスオブジェクトを操作して，そのクラスの情報を得ることができる．Java の場合，クラスは異なるソースファイル中で宣言されていたり，ソースコードが存在せず，そのクラスのバイトコードしか得られない可能性もある．いずれの場合でも，OpenJava の処理系はクラスオブジェクトを必要に応じて自動的に生成する．したがってメタプログラムでは，メソッドの実体部分などの詳細な実装部分を除けば，関連するクラスの情報を調べる際そのクラスがソースコードであるかバイトコードであるかを気にする必要はない．このようにして得られたクラスオブジェクトに対して，表 3 の，クラス定義の変更をともなう種のメソッドは呼べない．

表 2 の getModifiers() の返り値は OJModifier クラスのオブジェクトである．これは public，abstract，final といったクラスの修飾子群の論理構造を表す．このオブジェクトが修飾子の並び順を隠蔽するので，メタプログラムを書く際に修飾子の並び順を気にする必要がなくなる．

表 2 の getDeclaredMethods() が返す OJMethod オブジェクトも，メソッド定義の論理構造を表すので，OJClass 同様，属性を調べるためのメソッドや，それを変更するためのメソッドを持つ．OJMethod の持つメソッドのうち，基本的なものを**表 4** に示す．OJMethod のメソッドを呼び出して得られる型情報もやはり，クラスオブジェクトで表される．たとえば，表 4 の getReturnType() は，OJMethod オブジェクトの表すメソッド定義の返り値の型を OJClass オブジェクトとして返す．フィールド定義とコンストラクタ定義の論理構造を表す OJField と OJConstructor についても同様である．

OJMethod の getBody() が返す StatementList オブジェクトは，文を表すオブジェクトの並びであり，文や式を表すオブジェクトからなる．このオブジェクトは，従来のマクロ処理系と同様，抽象構文木とそれほど変わらない．ただし環境情報が組み込んであるので，変数の型などを調べることができる．型の情報以外，文や式の論理構造は，C 言語などとそれほど変わらず，著者らの経験からオブジェクト指向言語のマクロ記述には文や式の特別な情報が必要でないと判断したため，文や式に関してはクラスオブジェクトのような特別なデータ構造は新たに考案しなかった．

クラス定義の論理構造を表現するためのメソッド

表 5 は，クラスのより論理的な構造を知るためのメ

ソッドである．Java のクラス継承とメンバ隠蔽の仕組みを反映した情報を得ることができる．これらのメソッドの機能は，表 2 の基本的なメソッドを組み合わせて実現することもできる．しかしながら，クラスオブジェクトは，クラスの宣言文ではなく，プログラムの論理的な構成要素であるクラスに対応するという立場から，それらのメソッドを用意することは本質的に重要である．

クラスの継承機構を考えると，あるクラスに定義されているメソッドは，そのクラス宣言に記述されているものがすべてではなく，継承されているメソッドも考慮に入れなければならない．そこで，getMethods() を呼び出して得られるメソッドはそのクラスで明示的に宣言されているものだけでなく，スーパークラスやスーパーインタフェースから継承しているものを含むようになっている．

また，Java のクラスのメンバは，public，protected，private などの修飾子によって，そのメンバを利用できるクラスが限定され，また，それらの修飾子を省かれたメンバは同一パッケージのクラス以外からは利用できなくなる．そこで，getMethods() は引数として与えたクラスオブジェクトの表すクラスから利用可能なメソッドのみを返す．たとえば，別のパッケージにあり継承関係のないクラスを引数として与えると，getMethods() は修飾子が public のメソッドのみを返す．表 4 では，getMethods(OJClass) にそのクラスオブジェクト自身（this）を引数として与えて，そのクラス内部で利用可能な（継承されているものを含む）すべてのメソッドを得ている．

getMethods(OJClass) の実装は，Java のクラスの継承機構とメンバの隠蔽機構の論理的な仕様に依存する比較的複雑なものとなる．このような getMethods(OJClass) などのメソッドは，getDeclaredMethods() と getSuperclass() を組み合わせてプログラマに実現させるのでなく，OJClass のメソッドとして提供することがメタプログラムの記述をより簡潔にするうえで重要となる．

3.4 型駆動による変換

OpenJava のマクロ展開は，各クラス（型）に結び付けられたメタクラスによって制御されるため，著者らはこれを型駆動による変換と呼んでいる．先の例では，OJClass の translateDefinition() を上書きして，特定のクラスのクラス宣言の内容だけを変更（呼ばれる側の変換）した．OJClass はこのほかに，クラス・インスタンス生成式（new 式）や，メソッド呼び出し式，あるいはフィールド・アクセス式のうち，特定のクラスに関係するものだけを選んでマクロ展開する（呼ぶ側の変換）機能も備えている．

ここでは，Flyweight [7] パターンを支援するマクロの例を取り上げて，この機能を説明する．このデザインパターンは，等価なオブジェクトを多数生成しなければならないとき，それらを 1 つのオブジェクトで代用して，メモリ効率を改善するためのパターンである．このパターンの利用を支援するマクロは，たとえば Glyph オブジェクトの生成式：

```
new Glyph('c')
```

を，次のようなクラスメソッドの呼び出し：

```
GlyphFactory.createCharacter('c')
```

にマクロ展開する必要がある．createCharacter() は，与えられた引数に対応する Glyph のオブジェクトが，すでに生成されていればそれを返し，そうでなければ新たに生成するメソッドである．これによって，ある文字 c を表す Glyph のオブジェクトをいくつも生成せずに，1 つの Glyph のオブジェクトを自動的に共有するようになる．パターンに従った従来のプログラミングでは，Glyph オブジェクトを生成するプログラムで明示的に createCharacter() を記述していた．マクロによる支援により，パターンを適用されたオブジェクト生成を従来の new 構文で記述することが可能になる．

OpenJava では，このようなマクロ展開は，メタクラス FlyweightClass を定義して，クラス Glyph に適用すれば実現できる．このメタクラスは OJClass のメソッド expandAllocation() を**図 5** のように上書きする．このメソッドは，クラス・インスタンス生成式を受け取り，変換した式を返す．OpenJava の処理系は，ソースコード全体を調べ，そこに現れる Glyph のクラス・インスタンス生成式それぞれについてこのメソッドを実行してマクロ展開を実行する．

expandAllocation() は，クラス・インスタンス生成式を表す AllocationExpression オブジェクトと，その式の環境を表す Environment オブジェクトを引数にとる． Environment オブジェクトはその式のスコープにおける変数の型の情報など，名前の束縛情報を持っている．

OpenJava は，型駆動による変換を導入することで，

```
Expression expandAllocation(
    AllocationExpression expr, Environment env
) {
    ExpressionList args = expr.getArguments();
    return new MethodCall(this, "createCharacter", args);
}
```

図 5 クラス・インスタンス生成式の置き換え
Fig. 5 Replacement of class instance expressions.

表 6 適用場所別のマクロ展開のメソッド
Table 6 Macro-expansion methods for each place.

メソッド	適用場所
translateDefinition()	クラス宣言
expandAllocation()	クラス・インスタンス生成式
expandArrayAllocation()	配列生成式
expandTypeName()	クラス名
expandMethodCall()	メソッド呼び出し式
expandFieldRead()	フィールドの読み出し式
expandFieldWrite()	フィールドへの書き込み式
expandCastedExpression()	他の型へキャストされるこのクラスの型を持つ式
expandCastExpression()	このクラスの型へのキャスト式

プログラムのあちこちに分散したコードの断片を，包括的にマクロ展開することを可能にした．オブジェクト指向言語のためのマクロでは，しばしば，単純にクラス宣言を変換するだけでなく，オブジェクトの生成式など，そのクラスを使っている式もあわせて変換する必要がある．OpenJava では，メタクラスを定義し， expandAllocation() のようなメソッドを上書きすることで，そのメタクラスに結び付けられたクラスを使っている式を選択的にマクロ展開することができる．このような機能は，OpenC++ [4] など一部の処理系を除き，従来のマクロ処理系にはみられない．OJClass のメソッドのうち，マクロ展開のためにサブクラスが上書きできる主要なものを**表 6** に示す．

3.5 変換の仕組み

OpenJava の処理系は，ソースコードを与えられて起動されると，次のように動作する．

（1） ソースコードを解析し，各クラスごとにクラスオブジェクトを作成する．
（2） マクロ展開のためにクラスオブジェクトのメソッドを順次呼び出す．
（3） クラスオブジェクトに加えられた変更をソースコードへ反映する．
（4） 通常の Java コンパイラを呼び，バイトコードを生成する．

変換順序の一貫性

マクロ展開を行う OJClass のメソッドのうち，expand から始まる名前のものは，呼ぶ側の変換を行うため，他のクラス C を宣言しているソースコード中の式などに作用する．しかしそれらの式は， C のクラスオブジェクトの translateDefinition() から，OJMethod オブジェクトを介して変換される可能性もある．

OpenJava はこの曖昧性を解消するために，つねに translateDefinition() を先に実行し，呼ばれる側の変換がすんだクラス宣言に対して，呼ぶ側の変換を適用する．ただしこの仕様では，translateDefinition() がクラス X のクラス・インスタンス生成式をクラス Y のものに変換する場合，クラス X のメタクラスの expandAllocation() が実行されなくなってしまう．

また，親子関係にあるクラスに関して，OpenJava は必ずスーパークラスの translateDefinition() から順に実行する．クラスの定義はスーパークラスの定義に強く依存するため，変換の内容もスーパークラスの定義に依存する可能性がある．したがってまずスーパークラスの定義を確定するために，スーパークラスの translateDefinition() を実行する．

さらに，親子関係にないクラスの間でも，実行の順序が変換結果に影響を与える場合がある．たとえば，あるクラス C の変換がそのクラスの持つフィールドの型 D に依存するような場合，クラス D の変換後に C を変換する必要がある．このような場合には， C を変換するメタプログラム中の適切な時点で，OpenJava に用意されているメソッド waitForTranslation() に先に定義を確定しておきたいクラスオブジェクトを与えて呼び出すことにより，明示的に実行順序を指定することができる．このとき処理系は， C の変換を中断し，先に D の変換を終了した後， C の変換の続きを実行する．依存関係に循環が検出されると処理系はその旨を知らせて終了する．

複数の変換の変換結果に依存関係が存在する場合について，OpenJava では変換の順序に一貫性を持たせることによって曖昧性を解消している．このような問題について OpenC++では考慮されていなかった．

分割コンパイルへの対応

Java ではコンパイルの際， import しているクラスについては，ソースコードかバイトコード（クラスファイル）のどちらかがあればよいことになっている．しかし import されているクラスのソースコードが存在しないと，そのままではそのクラスのメタクラスが分からず，たとえば，そのクラスのオブジェクト生成式について，適切なメタクラスの expandAllocation() を適用できなくなる．

そこで OpenJava の処理系は，1 つのクラスの呼ばれる側の変換の処理をするたびに，メタクラスの名前など，そのクラスのメタレベル情報を定数に変換し，その定数をクラスフィールドに持つクラスを作成，コンパイルして，そのバイトコードを保存する．これによって，後にソースコードがなくなっても，そのバイトコードから必要なメタレベル情報を OpenJava の処理系が得られるようにしている．なお，保存されるメ

タレベル情報の種類は，メタプログラムが自由に拡張できる．

メタレベル情報は，そのクラスのバイトコードの特別な属性として保存することもできる．しかしながら独立したバイトコードとして保存する方が，マクロ展開後のプログラムを Java 仮想機械が実行する際，読み込むバイトコードの総量が小さくなると考え，現在の実装ではバイトコードの特別な属性として保存する方法を採用していない．

3.6 文法拡張

OpenJava のマクロでは，メタクラスで指定すれば，制限された特定の場所に新規の修飾句を導入することが可能である．1 つのメタクラスで新規の修飾句を複数導入する場合，それぞれの修飾句はメタクラス内で固有の識別子から始まらなければならない．追加された修飾句はそのメタクラスに対応するクラスのみに有効である．新規の修飾句の導入が許される部分は，メタクラスに対応するクラスの宣言文中（呼ばれる側）の

- クラス宣言中のメンバ宣言のブロックの直前
- メソッド宣言およびコンストラクタ宣言中のメソッド本体のブロックの直前
- フィールド宣言中のフィールド変数の直後

および，他のクラス宣言中（呼ぶ側）の

- 対応するクラス名の直後

に限られる．

新規に導入できる修飾句の形式と導入可能な場所が制限されているので，複数の文法拡張が衝突して，構文解析が不可能になることはない．またメタプログラマは，修飾句を導入する際に他のメタクラスで導入される修飾句との衝突を考慮しなくてよい．

図 6 は Adapter [7] パターンの利用を支援するマクロを実装したメタクラス AdaperClass を適用している例である．メタクラスにより `adapts` で始まる特別な修飾句が導入されており，クラス宣言に Adapter パターン用の注釈を記述させている．図 6 中の `adapts` 句では，クラス VectorStack が，クラス Vector からクラス Stack へのアダプタであることが宣言されている．この注釈は，メタクラス AdapterClass のクラスオブジェクトがマクロ展開を行うときにのみ利用される．したがって， VectorStack 以外のクラスオブジェクトからは， `adapts` 句の表す意味情報が，それに基づいて変換された結果である通常の Java のクラスとしてのみ，みえることになる．

この `adapts` 句を導入するためには，メタクラス AdapterClass にメソッド `getDeclSuffix()` を**図 7** のように実装する．メソッド `getDeclSuffix()` は，識

```
class VectorStack instantiates AdapterClass
    adapts Vector in v to Stack
{
    ....
}
```

図 6 OpenJava による文法拡張例
Fig. 6 An example of syntax extension in OpenJava.

```
static SyntaxRule getDeclSuffix(String keyword) {
    if (keyword.equals("adapts")) {
        return new CompositeRule(
            new TypeNameRule(),
            new PrepPhraseRule("in", new IdentifierRule()),
            new PrepPhraseRule("to", new TypeNameRule()) );
    }
    return null;
}
```

図 7 新規の修飾句を導入するメタプログラム
Fig. 7 A meta-program for a customized suffix.

別子句に対応して，それに続く構文を表す SyntaxRule オブジェクトを返す． SyntaxRule クラスのオブジェクトは再帰的下向き構文解析器を実装しており，処理系により与えられる字句列を解析して対応する抽象構文木を構築する．処理系はこのメソッドを呼び出して得た SyntaxRule オブジェクトを用いて構文解析を行う．

OpenJava は標準の Java の構文要素の構文解析器オブジェクトや，構文解析器オブジェクトを組み合わせた，構文の合成，繰返し，選択を行う構文解析器オブジェクトのためのクラスライブラリを備えている．メタプログラマは，このクラスライブラリを利用したり新しく下向き構文解析器を実装したりすることにより，望みの修飾句を定義できる．

3.7 OpenJava のメタクラス機構

OpenJava では 1 つのクラスに 1 つのメタクラスのみが対応する．1 つのクラスに複数のメタクラスを適用できると便利であるが，メタクラス間でマクロ展開の衝突が起きる問題があるため，現在は実装されていない．また，クラス A に対応するメタクラスは A のサブクラス A′ には及ばない仕様になっている．つまり，A′ のクラス宣言で明示的に指定しない限り，A′ の呼ばれる側と呼ぶ側のマクロ変換に A のメタクラスは関与しない．

OpenJava のモデルでは，インタフェースを含むすべてのクラスはメタクラスのインスタンスである．Java 言語では内部クラス（メンバクラス，局所クラス，匿名クラス）が存在するが，これらのクラスもやはりメタクラスのインスタンスである．OpenJava では内部クラスにも新たに定義したメタクラスを適用することができる．

4. 関連研究

従来のマクロ処理系については2章で取り上げた．クラスオブジェクトを抽象データ構造として採用したシステムは，3-KRS [18]，ObjVlisp [6]，CLOS MOP [15]，Smalltalk-80 [9] など数多い．Java 言語標準の Reflection API [13] でも，実行時にプログラムの論理構造を調査し，自己反映計算を行うためにクラスオブジェクトのモデルが採用されており，クラスオブジェクトのメソッドを呼び出して，クラスの名前やメソッドの一覧を得ることができる．これらのシステムでは，クラスオブジェクトを実行時のプログラムの振舞いを制御する自己反映機構のために用いている．著者らはクラスオブジェクトというモデルをマクロ処理系に応用し，プログラムの論理構造を操作するために用いた．

マクロ処理系にクラスオブジェクトを採用したシステムとしては，OpenC++ [4] がある．OpenJava は型駆動によってマクロを適用するといった設計上のいくつかの特徴を OpenC++から継承している．しかしながら，OpenJava と異なり，OpenC++でプログラムを表現するのに使われるデータ構造は，構文木であった．MPC++ [12] や EPP [11] などの類似のシステムについても同様である．このことは，2 章で述べたような高度なマクロを記述する際に，プログラミングを難しくする．たとえば，あるクラスの利用可能なメソッドを調べるためには，継承を考慮し，そのクラス宣言の構文木を調べ，さらにスーパークラスの構文木を入手して調べるというメタプログラムを記述しなければならない[5]．このため OpenC++では，ユーザからのフィードバックに応じて，そのようなプログラミングを簡略化するようなメソッド群を後からライブラリとして提供しなければならなかった．一方，OpenJava では設計の段階から，構文木ではなく，クラスの論理構造を表すクラスオブジェクトを中心的なデータ構造に採用している．したがって，OpenC++で問題となっていた多くの事例が OpenJava では解消されている．OpenJava のような高度なマクロ処理系を設計する場合には，プログラムの論理構造を表すデータ構造を主たるデータ構造にすることが重要であると，著者らは考える．

その他，デザインパターンを利用したプログラミングを言語レベルで支援する方法としては，GJ [2] にみられるパラメータ化された型のような，より言語に即した機構を使う方法も考えられる．しかしながら，これまでのところ，パラメータ化された型を活用した目立った支援技術は発表されていない．ほとんどの研究は，冒頭で紹介したような新しい言語機構を言語に導入してプログラミングを支援するものである．GJ ではクラスの定義をパラメータ化することができ，実パラメータとして与えた型ごとに異なるクラスの定義を得られる．しかし実パラメータとして与えられた型は，元のクラス宣言中に現れる仮パラメータを置き換えるだけで，そのパラメータに応じて宣言全体が大きく変更されることはない．デザインパターンの利用を支援するためには，パラメータの単純な置換ではなく，OpenJava のように，メタプログラムがパラメータを解析してクラスの定義を柔軟に変更できる能力が必要であるように思われる．

5. まとめ

本稿では，クラスオブジェクトと呼ぶデータ構造を提供する Java 言語用のマクロ処理系 OpenJava について述べた．これまで多くの研究者により，高度なマクロ展開を容易に記述できるように，マクロ記述言語および抽象構文木などのデータ構造が工夫されてきた．本研究も，そのような研究の流れの中に位置付けられる．OpenJava では，オブジェクト指向プログラムの論理構造を表すデータ構造を主たるデータ構造に据えたマクロ処理系を設計した．これにより，従来のマクロ処理系では記述が難しかった，オブジェクト指向プログラミングに典型的なマクロを記述することがより容易になった．

著者らは，オブジェクト指向言語用のマクロの応用例として，デザインパターンを利用したプログラミングを支援するマクロを OpenJava 上に実装してきた．クラスオブジェクトのデータ構造がこのようなマクロの記述に有用であることが分かる一方，デザインパターンの利用を支援するうえでいくつかの問題点がみえてきた．一般に，1 つのデザインパターンは複数のクラスから構成されるため，複数のクラスを包括的に表現したいという要求がある[19]．しかし，OpenJava ではマクロをクラス単位に適用するため，このような支援を行うマクロを記述するのが難しい．アスペクト指向プログラミング技術[14] を取り入れるなどして，この問題を解決していくのが将来の課題である．

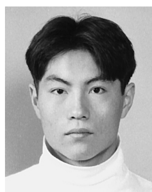
参考文献

1) Bosch, J.: Design Patterns as Language Constructs, *Journal of Object Oriented Programming* (1997).

2) Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P.: Making the future safe for the past:

Adding Genericity to the Java Programming Language, *SIGPLAN Notices*, Vol.33, No.10, pp.183–200 (1998).
3) Brown, P.J.: *Macro Processors and Techniques for Portable Software*, Wiley (1974).
4) Chiba, S.: A Metaobject Protocol for C++, *SIGPLAN Notices*, Vol.30, No.10, pp.285–299 (1995).
5) Chiba, S.: Macro Processing in Object-Oriented Languages, *Proc. TOOLS Pacific '98*, Australia, IEEE Press (1998).
6) Cointe, P.: Metaclasses are First Class: the ObjVlisp Model, *SIGPLAN Notices*, Vol.22, No.12, pp.156–162 (1987).
7) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994).
8) Gil, J. and Lorenz, D.H.: Design Patterns and Language Design, *IEEE Computer*, Vol.31, No.3, pp.118–120 (1998).
9) Goldberg, A. and Robson, D.: *Smalltalk-80: The Language*, Addison-Wesley (1989).
10) Gosling, J., Joy, B. and Steele Jr., G.L.: *The Java Language Specification*, Addison-Wesley (1997).
11) Ichisugi, Y. and Roudier, Y.: Extensible Java Preprocessor Kit and Tiny Data-Parallel Java, *Proc. ISCOPE'97*, California (1997).
12) Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H. and Kubota, K.: Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach, *Proc. Reflection'96*, pp.153–166 (1996).
13) JavaSoft: Java Core Reflection API and Specification, online publishing (1997).
14) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M. and Irwin, J.: Aspect-Oriented Programming, LNCS, Vol.1241, pp.220–242 (1997).
15) Kiczales, G., Rivières, J. and Bobrow, D.G.: *The Art of the Metaobject Protocol*, The MIT Press (1991).
16) Ladd, D.A. and Ramming, J.C.: A* : A Language for Implementing Language Processors, *IEEE Transactions on Software Engineering*, Vol.21, No.11, pp.894–901 (1995).
17) Maddox, W.: Semantically-sensitive macroprocessing, Master's Thesis, ucb/csd 89/545, University of California, Berkeley (1989).
18) Maes, P.: Concepts and Experiments in Computational Reflection, *SIGPLAN Notices*, Vol.22, No.12, pp.147–155 (1987).
19) Soukup, J.: Implementing Patterns, *Pattern Languages of Program Design*, chapter 20, pp.395–412, Addison-Wesley (1995).
20) Steel Jr., G.L.: *Common Lisp: The Language*, 2nd edition, Digital Press (1990).
21) Weise, D. and Crew, R.: Programmable Syntax Macros, *SIGPLAN Notices*, Vol.28, No.6, pp.156–165 (1993).

(平成 11 年 9 月 2 日受付)
(平成 12 年 6 月 1 日採録)



立堀　道昭（学生会員）

1974 年生．1997 年筑波大学第三学群情報学類卒業．1997 年より同大学院博士課程工学研究科．1999 年同大学院工学修士号取得．言語処理系，プログラミング，システムソフトウェアに関する研究に従事．日本ソフトウェア科学会，ACM 各学生会員．

千葉　　滋（正会員）

1968 年生．1991 年東京大学理学部情報科学科卒業．1993 年同大学院理学系研究科情報科学専攻修士課程修了．1996 年同専攻博士（理学）取得．1996〜97 年同専攻助手．1997 年より筑波大学電子・情報工学系講師．言語処理系およびオペレーティングシステム等システムソフトウェアの研究に従事．日本ソフトウェア科学会，ACM 各会員．

板野　肯三（正会員）

1948 年生．1977 年東京大学大学院理学系研究科物理学専門課程博士課程単位取得後退学．1993 年より筑波大学電子・情報工学系教授．計算機のアーキテクチャ，分散処理システム，プログラミングシステム等に関する研究に従事．理学博士．日本ソフトウェア科学会，電子情報通信学会，ACM，IEEE 各会員．