OMetaのための衛生的マクロ定義機構導入方式

星野 友宏　高桑 健太郎　渡部 卓雄

本研究では，OMeta を用いて定義された言語に衛生的マクロの定義機構を導入する手法を提案し，例題の記述を通してその有用性を明らかにする．OMeta は PEG による強力なパターンマッチング機構を備えた言語であり，構文解析器や言語処理系の記述に適している．OMeta によって定義された言語にマクロの定義機構を追加する際，マクロ定義およびマクロ展開の記述のために対象言語の構文をそれぞれ拡張する必要がある．また，マクロを衛生的にするためにはマクロ展開機構で識別子のスコープを管理する必要がある．提案手法は，これらに共通する機構を OMeta のモジュールとして提供し，既存の対象言語にマクロ定義機構を容易に導入できるようにするものである．

Programming languages with macro definition mechanisms often need special treatment for parsing macros. If a macro introduces a new syntax to a language, an extended parser is needed to recognize a code written using the macro. Also, the language might use some specific mechanisms such as pattern matching to define macros. In this paper, we propose a method of introducing hygienic macro definition mechanisms into languages defined in OMeta, an object-oriented language with PEG-based general-purpose pattern matching. Using the proposed method, extended parsers for macro definition are automatically generated from the specifications of macro definition mechanisms.

1　はじめに

プログラムにおいて、よく用いられるコード片に名前を付けて再利用を可能にするマクロは、プログラミングの可読性や効率の向上、およびある種のメタプログラミングを可能にする便利な言語機構である。現状では、マクロを備えている言語は限られている。C の`#define`のような単純な文字列置き換えによるマクロであれば、m4 などのプリプロセサを使うことで言語を問わず簡便に利用できるが、この方法にはいくつかの問題点があることが知られている [1]。その問題点を回避する仕組みを備えた衛生的マクロ機構というものが考えられているが、現状では衛生的マクロ機構を言語機能として持つ言語は Scheme などの一部の言語に留まる。

本論文では、OMeta [2] を用いて実装された言語処理系に対し、元の言語処理系にはほぼ手を入れずに衛生的マクロ機構を導入する手法を提案する。本手法を用いることで、DSL などを開発する言語開発者は開発中の言語に比較的容易に衛生的マクロ機構を導入することが可能になる。これにより、DSL の構文の変更等に比べて素早いプロトタイピングが可能になり、言語のユーザの利便性向上につながるといった効果が期待できる。

提案手法では、言語処理系の字句解析器に拡張を施し、さらに構文解析に関するデータフローの途中に、構文解析器拡張モジュール、マクロ定義管理モジュール、マクロ展開器モジュールを挿入することにより、最終的に元の文法に従う（マクロを含まない）抽象構文木を生成できるようにしている。我々はこの手法を

---

*A Method for Implementing Hygienic Macro Definers for OMeta

Tomohiro Hoshino, Takuo Watanabe, 東京工業大学・情報理工学院, School of Computing, Tokyo Institute of Technology.

Kentarou Takakuwa, 東京工業大学・工学部情報工学科, Department of Computer Science, Tokyo Institute of Technology.

例題として作成した簡単な手続き型言語に対して適用し、提案手法の有効性を確認した。

本論文の構成を記す。まず次節で本研究の背景について述べ、次に第 3 節では提案手法の概要について述べる。第 4 節では例題として作成した言語の仕様と実装について簡単な説明を述べ、続く第 5 節でその実装を通して提案手法の有用性について述べる。第 6 節では現状の制約と今後の課題について述べる。第 7 節で関連研究について述べ、第 8 節でまとめを行う。

2 背景

2.1 OMeta

OMeta[2] は、パターンマッチング機構を備えたオブジェクト指向言語であり、Parsing Expression Grammar (PEG) を基礎とする文法定義のための構文と、オブジェクト指向の特徴である継承やオーバーライド、親文法のルール呼び出しといった各種操作をサポートしている。また、OMeta は JavaScript (ECMAScript5) を含む様々な言語上で実装されている。OMeta による定義はそれぞれのホスト言語で動作するプログラムへと変換することが出来る。

OMeta は言語設計とそのプロトタイピングにおいて強力な基盤を提供するが、OMeta そのものには、OMeta 上で作成された処理系に対してマクロ機構を付与するためのサポートはない。そこで、本研究では OMeta の JavaScript 実装を用いて、動的に OMeta の定義を生成し言語処理系を拡張することにより、OMeta 上で定義された言語処理系に対してマクロ機構を付与することを試みた。

2.2 衛生的マクロ機構

コード片に名前を付けて、あたかも新たな言語機構であるかのように再利用可能にする言語機構はマクロと呼ばれ、いくつかのプログラミング言語で利用可能になっている。例えば C では`#define`により、Scheme では `define-syntax`によってマクロを定義できる。

C におけるマクロは単純な文字列置き換えに基づくものであるため、以下の例に示すように、展開後の名前の衝突が生じることがある。まず、C のマクロ `SWAP`をコード 1 のように定義する。このマクロはコード 2 のように使って 2 つの変数（ここでは `a`と `tmp`）の内容を交換することを意図して定義したものである。

```
#define SWAP(x,y) \
  { int tmp = x; x = y; y = tmp; }
```

コード 1　SWAP マクロ

```
int main(void) {
    int a = 0;
    int tmp = 5;
    SWAP(a, tmp);
    printf("%d,␣%d\n", a, tmp);
    return 0;
}
```

コード 2　SWAP マクロの利用例

```
int main(void) {
    int a = 0;
    int tmp = 5;
    { int tmp = a; a = tmp; tmp = tmp; }
    printf("%d,␣%d\n", a, tmp);
    return 0;
}
```

コード 3　コード 2 の展開例

コード 2 を展開して得られたコード 3 は「`0, 5`」という結果を出力するが、これは意図した結果である「`5, 0`」と等しくない。この原因は、マクロが展開される直前で定義されている変数 `tmp`を、マクロ内で一時変数として定義される `tmp`が隠蔽してしまうことにある。これ以外にも、単純な文字列置き換えによるマクロ展開にはいくつかの問題があることが知られている [1]。

そのような問題を回避する仕組みを備えたマクロ機構として、**衛生的マクロ** *(hygienic macro)* がある。衛生的マクロの実現方式の一つである Syntactic Closure[1] は、「マクロ定義時の環境を保存する」というアイデアを元にしている。本研究ではこのアイデアを利用し、提案方式で作られるマクロ定義機構によって定義されるマクロが衛生的マクロとなるようにしている。

3 提案手法

本手法でマクロ機構を既存言語の処理系へ実装するためには、主に以下の手順を行う。

1. 字句解析器の拡張
2. 構文解析器拡張モジュールの実装
3. マクロ展開器モジュールの実装

この手順によって拡張された言語処理系（構文解析器）の構造は図 1 のようになる。構文解析器拡張モジュールにより元の構文解析器がマクロによる新規構文を取り扱えるように拡張され、マクロ展開器モジュールによりマクロの衛生的展開が行われる。

本手法で拡張された言語においてマクロを定義するには、図 2 のような構文を用いる。ここで《 マクロ構文定義 》は OMeta で記述されたマクロ構文の定義である。この定義では OMeta のキャプチャ機能を利用し、部分構文木をメタ変数へと束縛することが出来る。一方《 コードテンプレート 》は展開後のコードの記述である。この記述はこのマクロ定義が出現するより前の（マクロによる拡張が導入される前の）構文によって行われる。加えて、《 マクロ構文定義 》内でキャプチャされた部分構文木は、束縛するメタ変数を@...@ で囲むことにより取得することができるようにしている。

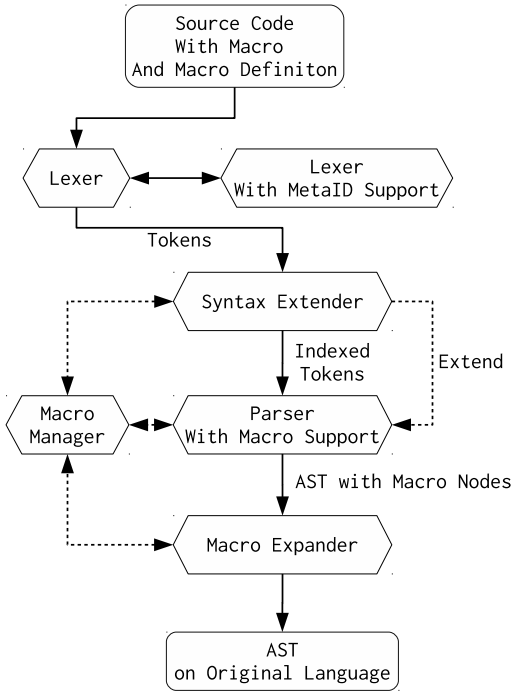

図 1　マクロ機構の構造

```
MACRO_FROM
  《 マクロ構文定義 》
MACRO_TO
  《 コードテンプレート 》
MACRO_END
```

図 2　マクロ定義のための構文

3.1 字句解析器の拡張

まず、先述のマクロ定義構文を正しく受け取り処理できるように、字句解析器の拡張を行う。具体的には、オリジナルの字句解析器をもとに、メタ変数（@...@で囲まれた識別子）および図 2 で示したキーワード **MACRO_FROM**, **MACRO_TO** および **MACRO_END** をマクロ定義トークンとして認識できるように拡張した字句解析器を定義する。

拡張された字句解析器は、マクロ定義トークンの組を見つけた場合、MacroDefinition というトークンを生成する。このトークンには次のデータが含まれるようにする。

- マクロルール文字列：
  《 マクロ構文定義 》による文法定義 (OMeta の処理系に渡すため、文字列のまま格納)
- マクロ展開木用トークン列：
  《 コードテンプレート 》を、メタ変数を扱える拡張字句解析器を利用してトークン化したもの

3.2 構文解析器拡張モジュール

図 1 中の Syntax Extenderが構文解析器拡張モジュールに当たる。このモジュールでは、あらかじめ入力されたソースコード中に出現する全てのマクロを

適切な条件を付け加えた上で処理してしまい、適切なブロックスコープでマクロを処理できる構文解析器を生成する。

Syntax Extender は、以下の項目に関する追跡・処理を行う。

- 全トークンへのインデックス付与
- ブロックへのラベル付けとブロックスコープの追跡
- MacroDefinitionトークンの処理
- 全トークンの処理後にマクロ定義トークンを処理できるように構文解析器を拡張

全トークンへのインデックス付与 とブロックへのラベル付けとブロックスコープの追跡は、構文解析器がマクロの有効範囲を判定するために必要となる。

MacroDefinitionトークンの処理により、マクロを処理できるように構文解析器が拡張される。ここでは以下のような手続きが行われる。

- マクロルール文字列を OMeta 文法定義の文字列へ必要に応じて変換
- マクロルール文字列から取得するべきメタ変数の列を取得
- 取得したメタ変数の列を利用し、メタ変数を適切に受け取れるように拡張した構文解析器を生成
- 生成した構文解析器を利用して、マクロ展開木用トークン列を解析し、マクロ展開用の抽象構文木を得る
- マクロを処理できるように現在の構文解析器を拡張し、拡張した構文解析器で現在の構文解析器を置き換える
- MacroManagerに、現在のブロックのラベル名、マクロ展開用の抽象構文木、メタ変数名の列を通知・登録

この処理は、MacroDefinitionトークンが検出される度に行われる。これにより、別のマクロ定義内でそれまでに定義したマクロを利用可能にしている。ここで拡張された構文解析器は、マクロのルールにマッチするトークン列を見つけると、マクロ展開に必要な情報 (マクロ名、メタ変数の内容) を含むノードを生成する。(以下、マクロノード とする)

全トークンの処理が終わった後、その時点での構文解析器を MacroDefinitionトークンを適切に処理できるように拡張し、それを Parser with Macro Supportとして扱う。以上の処理により、入力されたソースコード中に出現する全てのマクロを適切な有効範囲で処理可能な拡張済み構文解析器が生成される。

3.3 マクロ展開器モジュール

図 1 中の Macro Expanderがマクロ展開器モジュールに当たる。このモジュールでは、拡張済み構文解析器が生成した抽象構文木中のマクロ定義ノード・マクロノードを、マクロ内の環境がマクロ以外のプログラム上の環境と衝突しないように対策を行いながら処理を行い、最終的にマクロを含まない元の文法によっても生成が可能な抽象構文木を生成する。

Macro Expander は、基本的にはマクロを含む抽象構文木を訪問しながら、以下の項目に関する追跡・処理を行う。

- プログラム上で定義される全てのシンボルの収集
- プログラム上のブロックスコープを含む環境の追跡
- マクロ定義ノードを発見した時に必要なシンボルのエイリアスを生成
- マクロノードを発見した時にマクロ内一時変数のシンボル名の変更を行いマクロ展開を実行

プログラム上で定義される全てのシンボルの収集は、マクロ定義ノードを発見した時に必要なシンボルのエイリアスを生成する時、プログラム上全体で衝突しないシンボル名の生成を行うために必要になる。この時は、ブロックを考慮せずにプログラムのコード上に現れる全ての変数・関数宣言を追跡し、収集を行う。

プログラム上のブロックスコープを含む環境の追跡は、マクロの展開を行う時に衝突しないシンボル名の生成を行うために必要になる。ブロックスコープは階層構造も含めて追跡を行う。

マクロ定義ノードを発見した時、まず該当するマクロの展開用構文木を訪問し、「マクロ内で定義されてなく、且つマクロ内で使用されているシンボル」を全

て列挙する。これらのシンボルをマクロが定義時の環境に依存しているシンボルとし、それらのシンボルを定義時の状態で展開されたマクロ内で使用可能にするために、それらのシンボルにプログラム上で定義される全てのシンボルと衝突しない別名シンボルを与える。その後、マクロ内の定義時依存シンボルをその別名を利用するように適切に変更し、マクロ定義ノードの位置にエイリアスを生成するノード (つまり、エイリアスを定義する変数定義ノード) を追加する。これにより、まずマクロ内で定義されずに使用されている変数・関数などに関する環境を、マクロ定義時の状態に固定する。

マクロノードを発見した時、まず該当するマクロの展開用構文木を訪問し、マクロ内で定義されるシンボルを、ブロックスコープに関する情報と共に全て列挙する。これらのシンボルをマクロ内一時変数とし、それらのシンボルを定義及び使用するノードを、マクロノードが出現した時点で定義されている全てのシンボルと衝突しない別名シンボルを生成した上で、そのシンボルを定義・使用するように、ブロックスコープを考慮しながら適切に書き換える。これにより、マクロ内一時変数が誤ってプログラム上のシンボルを隠蔽してしまうことを防ぐ。その後、マクロの展開用構文木に含まれるメタ変数に関するノードを適切に該当する文法要素や識別子に置き換え、マクロ展開後の構文木をマクロノードの位置へ挿入する。

以上の処理により、衛生的なマクロ展開が行われ、元の文法に従う抽象構文木が生成される。

4 実装

本研究のケーススタディーのため、提案手法を用いてマクロ機構を実装する言語を設計した。本言語は手続き型言語の構造を持ち、以下のような特徴を持つ。

- 変数に型はない
- 関数は第一級オブジェクト
- ブロックによるレキシカルスコープを持つ

サポートされる構文は以下のとおりである。

- 関数定義
- 関数呼び出し
- 変数定義
- 変数参照
- 変数代入
- if 文
- while 文
- ブロック
- 四則演算・括弧による計算優先順位の明示

ここで、衛生的マクロを実現するために、注意して追跡を行うべき 2 種類の抽象構文木ノードがある。

1 つ目が「環境に対して変更を加える抽象構文木ノード」である。これは、実行された時点での環境に対して、シンボルを新たに加える操作を行うようなノードを指す。変数定義、関数定義などがこれに該当する。これを本論では環境変更要素 と呼ぶこととする。

2 つ目は「環境に対して問い合わせを行う抽象構文木ノード」である。これは、実行された時点での環境に対して、シンボルの内容を問い合わせたり、シンボルへの参照を得る操作を行うようなノードを指す。変数参照、変数代入、関数呼び出しなどがこれに該当する。これを本論では環境利用要素 と呼ぶこととする。

本研究で実装したマクロ機構では、マクロ展開器モジュールにおいてこれらの要素内のシンボル名を適切に操作することにより、衛生的マクロを実現している。

本研究では、概ね図 1 に従うプログラムを、OMeta の JavaScript 実装である OMeta/JS、JavaScript 実行環境である Node.js、JavaScript をコンパイル結果として出力するオブジェクト指向言語の TypeScript を用いて実装した。

5 例題

本節では、マクロの衛生性が問題となる 2 つの例題を示し、本研究で設計したマクロ機構が衛生的となっていることを示す。

```
var flag = 1;

MACRO_FROM
alternateness ( Block:ctxa , Block:ctxb );
MACRO_TO
if (flag) {
    @ctxa@
    flag = 0;
}
if (not(flag)) {
    @ctxb@
    flag = 1;
}
MACRO_END

{
    var flag = 0;
    while (operationLT(flag, 10)) {
        alternateness( {
            print(5);
        }, {
            print(10);
        } );
        flag = flag + 1;
    }
}
```

コード 4　ブロックの交互実行マクロ

5.1　マクロが定義時の環境に依存する場合

マクロが定義時の環境に依存する例として、コード 4 のようなプログラムを考える。alternatenessマクロは、実行される度に指定された 2 つのブロックが交互に実行されることを意図している。そのような機能を実現するため、このマクロでは flagという変数をマクロ定義前に宣言し、利用している。flagは保存されないといけないため、マクロ内一時変数としてではなく、マクロの外に定義する必要がある。また、検証用言語には否定の演算子が無いため、not()という関数も中で使用されている。これもマクロ定義時の環境に依存する要素である。

しかし、このプログラム上では、マクロ外のプログラムにも flagという変数が定義されているため、単純なマクロ展開ではマクロ内からの flagへの参照が本来意図した変数への参照と異なってしまう。これにより、マクロの意図が損なわれる可能性が出てくる。提案方式で実装したマクロ機構では、コード 5 に示

```
var flag = 1;
var flag__at_2 = flag;
var not__at_2 = not;
{
    var flag = 0;
    while (operationLT(flag, 10)) {
        if (flag__at_2) {
            {
                print(5);
            }
            flag__at_2 = 0;
        }
        if (not__at_2(flag__at_2)) {
            {
                print(10);
            }
            flag__at_2 = 1;
        }
        flag = _operationAdd(flag, 1);
    }
}
```

コード 5　コード 4 の展開

すような対策付きのマクロ展開を行うことにより、この問題を回避する。

このように、マクロが依存する flagと notに対して、マクロ定義の出現箇所にエイリアスを作成し、それをマクロ内で利用することにより、プログラム上で定義されうるシンボルとの衝突を回避するため、マクロが定義時の環境に依存していても、依存するシンボルについては環境が保存されることを保障する。また、別名の作成時には作成した別名が本当に衝突していないかを再度チェックしているので、マクロ外のプログラムで flag__at_2のような変数を定義しても問題は起こらない。

5.2　マクロ内で一時変数を使う場合

次に、マクロ内で一時変数を使う例としてコード 6 のようなプログラムを考える。これは、第 2 節で示した SWAPマクロを提案方式で実装した言語とマクロ機構に適応するように書きなおしたものである。第 2 節で触れたように、このプログラムはマクロ内で定義された一時変数が意図せずマクロ外のプログラム上の変数を隠蔽してしまい、マクロの意図が損なわれる可能性を持っている。提案方式で実装したマクロ機構

```
MACRO_FROM
swap ( Identifier:lhs , Identifier:rhs ) ;
MACRO_TO
{
    var temp = @lhs@;
    @lhs@ = @rhs@;
    @rhs@ = temp;
}
MACRO_END

var temp = 2;
var other = 3;

swap(temp,other);

print(temp);
print(other);
```

コード 6　マクロ内で一時変数を使う例

```
var temp = 2;
var other = 3;
{
    var temp__at_5 = temp;
    temp = other;
    other = temp__at_5;
}
print(temp);
print(other);
```

コード 7　コード 6 の展開

では、コード 7 のように対策付きのマクロ展開を行うことによりこの問題を回避する。

このように、マクロ内で定義された一時変数 temp を、その時点で可視である環境に存在するシンボル、及びメタ変数で指定された変数・関数宣言・識別子と被らないようにリネームすることにより、プログラム上で定義された変数との衝突を回避し、マクロの意図を保護している。

6　制約・今後の課題

本研究で設計・実装したマクロ機構には、以下のような制約が存在する。

- 実装対象の言語がレキシカルスコープを持つことを仮定しているため、ダイナミックスコープを持つような言語には適用できない
- ポインタ・参照の仕組みを持つ言語の場合、ポインタ・参照が示すオブジェクトの保存を行っていないため、そのオブジェクトに関してマクロ定義時の状態が保存されることを保証しない
- 式として取り扱えるマクロを定義できない
- 再帰的定義は出来ない
- マクロ内でマクロを定義することは出来ない

また、現在の設計では字句解析直後のトークンレベルでブロックスコープの追跡を行っているが、言語の構造によってはトークンレベルでは厳密なブロックスコープの追跡が難しい可能性があることが分かっている。

どのような言語に対して本手法が適用可能かについてもまだ実装例が足りておらず、手続き型言語以外の言語や実用言語に対して本手法が適用可能か十分には分かっていない。本論文の執筆時点で Lisp 風の構造を持つ言語に対して本手法が適用できるかを試行しているが、本手法で提案したマクロ機構の構造をそのまま使用できない可能性があることが判明している。そのため、本手法が提案するマクロ機構の構造が十分一般的でないことも考えられ、今後マクロ機構の構造についてより議論を重ねていく必要があると考えている。

本手法はマクロ機構の実装の方針と仕様について述べるに留まっており、字句解析器の拡張はほぼそのまま本研究での実装例と同じようにできるが、構文解析器拡張モジュールとマクロ展開器モジュールについては実装先の言語の構造・仕様に依存しているため、再利用できる箇所が少なく、実装コストはまだ高いと考えている。そのため、今後実装例を増やしていく中で再利用可能な箇所を増やし、自動化可能な箇所を見つけていくことが必要であると考えている。

7　関連研究

甫水らによる研究 [3] では、動的に拡張できるパーザと Scheme 処理系に含まれる衛生的マクロ機構を用いて、JavaScript に対して表現力に優れた衛生的マクロ機構を付与する手法を提案している。この研究では、Scheme 処理系が持つ衛生的マクロのシステム

を利用するため、ホスト言語と S 式間の変換を行う。一方本研究のマクロ機構では、拡張後のパーザによって得られた抽象構文木に対して直に訪問処理を行い、マクロ定義時のマクロ展開時のそれぞれで決められた対策を行う。これによって条件付きで衛生的なマクロ展開を実現している。まだ表現力は豊かでなく、対象言語に対応する OMeta 定義 (字句解析器、構文解析器) や構文木訪問処理が書けることが必要になるが、マクロ機構を実現するのに必要な実行環境は 1 つで良い (本研究では JavaScript のみ) という利点がある。

8 まとめ

本論文では、OMeta で実装された字句解析器・構文解析器からなる言語処理系に対して衛生的なマクロ機構を追加するための手法を提案し、例題として作成した言語に対して提案手法を適用することで、衛生的マクロ機構を実現できることを示した。今後様々な構文の言語に対する適用例を増やして提案手法の有用性を示すとともに、より洗練されたマクロ定義機構の導入方式を確立することを目指す。

謝辞

本研究の一部は JSPS 科研費 26330079 および 15K00089 の助成を受けている.

参 考 文 献

[1] Bawden, A. and Rees, J.: Syntactic Closures, *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, New York, NY, USA, ACM, 1988, pp. 86–95.

[2] Warth, A. and Piumata, I.: OMeta: An Object-Oriented Language for Pattern-Matching, *Symposium on Dynamic Languages (DLS 2007)*, ACM, 2007, pp. 11–19.

[3] 甫水佳奈子, 脇田建, 佐々木晃: 解析表現文法と Scheme マクロ展開器を用いた JavaScript 向け Hygienic 構文マクロシステムの実装, 情報処理学会論文誌プログラミング（*PRO*）, Vol. 6, No. 2(2013), pp. 85–101.