OMetaのための
衛生的マクロ機構導入方式

星野友宏　高桑健太郎　渡部卓雄
東京工業大学

概要

- OMetaで実装された言語処理系（構文解析系）に，衛生的マクロを定義するための機構を簡便に導入するためのフレームワークOMetaMacroの提案
- Cに似たブロック構文を持つ言語を対象として，いくつか制限はあるが衛生的なマクロを定義できることを確認

マクロ

- よく用いられるプログラム片に新しい名前（あるいは構文）を与えたもの
- 実行前（コンパイル時）に元のプログラム片に展開される
- 用途：記述の簡素化，構文の追加，DSL
- マクロ定義機構を持つ言語の例
  - Lisp, Scheme
  - Scala, Elixir
  - C, C++, 各種アセンブリ言語

動機

自動化できないか？

言語Lの処理系（パーザ）

L

Lにマクロ定義機構を追加した

言語L_{macro}の処理系

L_{macro}

```
int a = 1;
int k = 1;
while (k <= n) {
    a = a * k;
    k++;
}
```

Lのプログラム

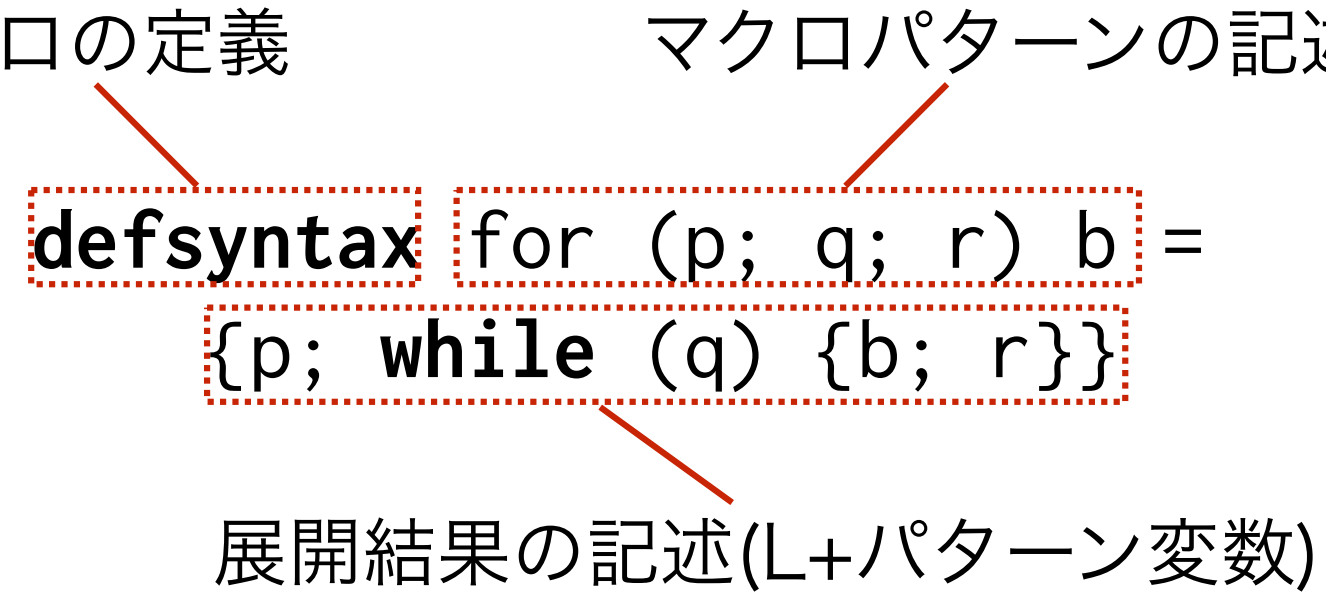
```
defsyntax for (p; q; r) b =
    {p; while (q) {b; r}}

int a = 1;
for (int k = 1; k <= n; k++)
    a = a * k;
```

L_{macro}のプログラム

自動化

- 様々な拡張構文解析器の導入



- 衛生的マクロ展開機構の導入

他の手法

- 処理系に手を入れて構文を追加する
  - プログラマに新しい構文の追加を解放したい
- 構文拡張用API (Common-Lispのdefmacro)
  - ASTの内部構造を知る必要がある / 非衛生的
- プリプロセッサ(m4, cpp等)
  - 定義可能な構文に制約
  - 非衛生的
- その他（演算子定義，テンプレート，パターンマッチ，メタプログラミング，etc.）
  - これらの機能のない言語も対象としたい

衛生的マクロ(Hygienic Macro)

- 名前の衝突やスコープの誤りが生じないマクロ
  - 例：展開時に導入した名前が衝突する

```
#defmacro SWAP(x, y) { int t = x; x = y; y = t; }
int t = 1, u = 2;
SWAP(t, u)  // { int t = t; t = u; u = t; }
```

  - 例：展開後に異なるスコープの名前を参照する

```
#defmacro COUNT { counter++; }
... COUNT ...
{ int counter = 0; ...; COUNT }
```

- 例：Schemeのdefine-syntax
- 実装方式
  - 古典：[Kohlbecker '86][Bawden '88][Clinger '91]
  - 最近のRacketでの実装：[Flatt '16]

研究目的

プリプロセッサ方式では物足りないが，構文解析より後には（できれば）手をつけたくない

- 構文解析器に衛生的マクロの定義機構を導入する上での諸条件，およびそれにもとづく導入手法の有用性を明らかにする
  - 様々な言語に対応可能にするには？
    - 適応可能な構文の性質
    - スコープ規則
    - 型システム，副作用，etc.
  - 記述可能なマクロパターン
    - 新しい構文規則の導入方法
  - 展開方法の記述方式

現時点での貢献

- OMetaで記述された，Cに似たブロック構文をもつ言語の構文解析器に，衛生的マクロの定義機構をモジュールとして（後付けで）導入する手法を提案・実装し，その有効性を確認した
  - 実装：OMetaMacro

- 問題点
  - 対応できる構文の制約がある
  - 非衛生的な部分が一部残っている

関連研究

- Schemeの衛生的マクロ機構
  - 歴史と伝統
    - [Kolhbecker '86] [Clinger '91] 〜 [Flatt '16]
  - S式(AST)を扱うので具象構文の扱いが楽
    - 本研究で問題になるような部分は無視できる
- EX-JS [甫水 '13]
  - JavaScriptのための衛生的マクロ機構
    - JSコードをSchemeに変換し，Scheme処理系を用いて衛生的マクロ展開をおこなう
  - 対象言語↔Schemeの相互変換が必要

OMeta [Warth '07]

- パターンマッチ機構を備えたOOP言語
  - パターンマッチ：PEGによる構文解析
  - 様々なホスト言語上に実現：Smalltalk, JS, etc.
- 例：算術式

```
ometa Exp {
  num = <digit+>:d          -> parseInt(d, 10),
  fac = num
      | '(' exp:x ')'       -> x,
  ter = ter:x '*' fac:y     -> (x * y)
      | ter:x '/' fac:y     -> (x / y)
      | fac,
  exp = exp:x '+' ter:y     -> (x + y)
      | exp:x '-' ter:y     -> (x - y)
      | ter
}
```

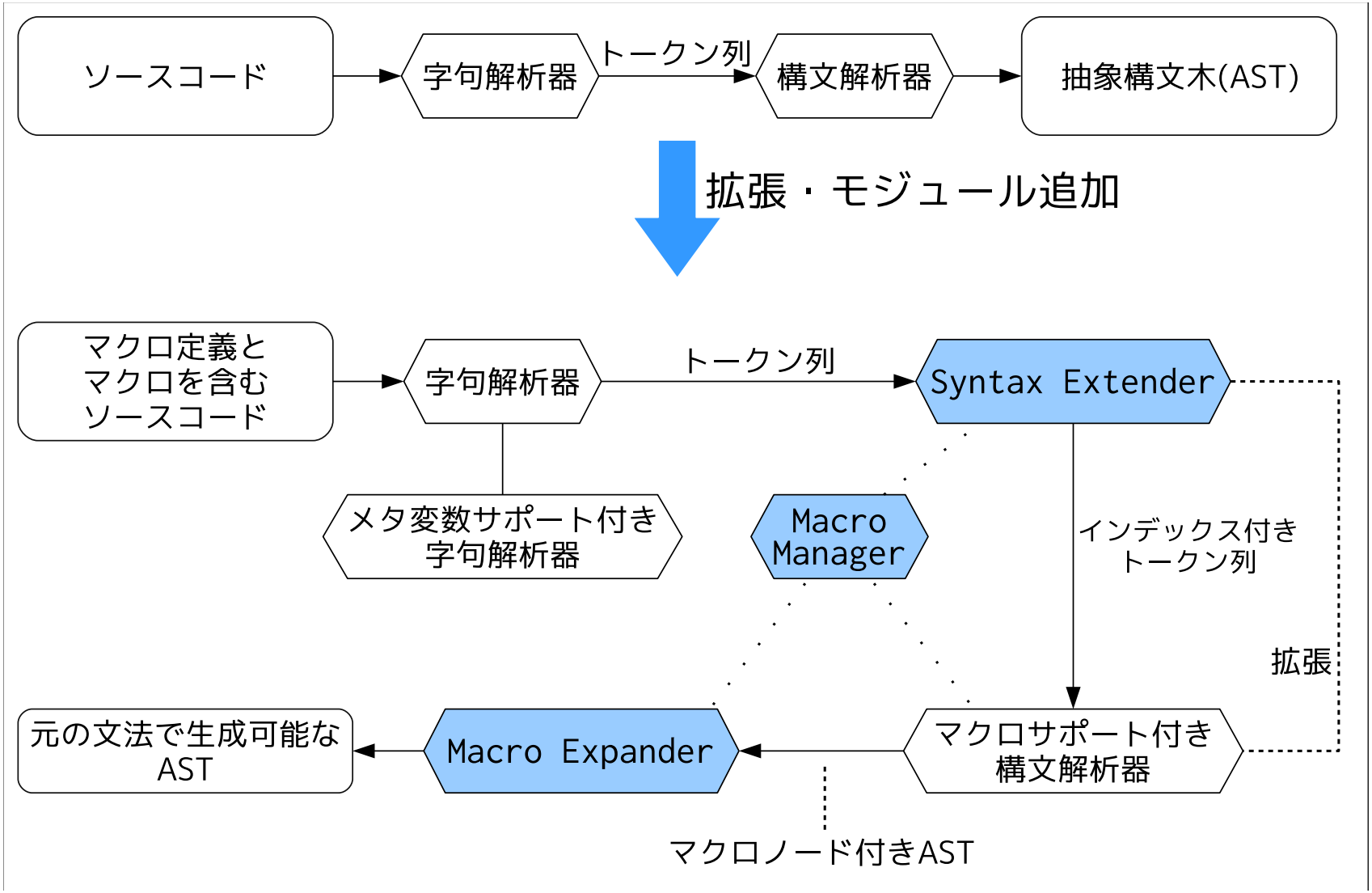
構文定義の拡張

- 他の構文規則を継承して拡張できる

```
ometa UExp <: Exp {
  fac = '-' fac:x          -> (- x)
      | '+' fac:x          -> x
      | super(`fac)
}
```

- 提案手法では，構文規則の継承を活用してマクロ定義と使用に関する各場所の構文解析器を定義している

提案手法


ソースコード
字句解析器
トークン列
構文解析器
抽象構文木(AST)
拡張・モジュール追加
マクロ定義と
マクロを含む
ソースコード
字句解析器
トークン列
Syntax Extender
メタ変数サポート付き
字句解析器
Macro
Manager
インデックス付き
トークン列
拡張
元の文法で生成可能な
AST
Macro Expander
マクロサポート付き
構文解析器
マクロノード付きAST

マクロ定義構文

- 以下のような構文を用いる（言語独立）

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

- 例

```
MACRO_FROM
  swap(Identifier:x, Identifier:y);
MACRO_TO
  { var t = @x@; @x@ = @y@; @y@ = t; }
MACRO_END
```

マクロ定義構文の記述方法

- 《マクロ構文定義》
  - OMetaのパターンとして記述
  - 元の言語の構文定義における名前が使える
    - 例：Identifier
- 《コードテンプレート》
  - 元の言語の構文＋パターン変数
    - パターン変数：@x@, @y@

字句解析器の拡張

- マクロ定義構文のためのキーワードを認識
  - MACRO_FROM, MACRO_TO, MACRO_END
- マクロ定義を一つのトークンとして出力
  - マクロ構文定義は文字列として格納
    - これをもとにOMetaで拡張パーザを作成
  - コードテンプレートはトークン列として格納
    - こちらはマクロ展開器に渡される

構文解析器の拡張

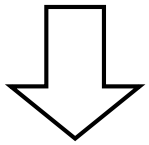
- Syntax Extender
  - マクロ定義を扱うことのできる構文解析器（OMetaモジュール）を実行時（対象言語の構文解析前）に生成し，マクロを含む構文解析を行う
    - OMeta/JSにおけるOMetaモジュールはJSのオブジェクトであるので，拡張した構文解析モジュールを動的に生成できる.
- Macro Expander
  - Syntactic Closure[Bawden '88]に類似のアルゴリズムにより，ASTをスキャンしてマクロを展開

例題

- Cに似た構文の簡単な言語をOMeta/JSで定義
  - 代入文，関数定義・呼出，if, while, ブロック
  - ブロックによるレキシカルスコープ
  - 一級関数，動的型検査
- 拡張部分：Syntax Extender, Macro Manager, Macro Expander
  - OMeta/JSおよびTypeScriptで実装

例題：swapの展開

```
var t = 1;
var u = 2;
swap(t, u);
```



```
var t = 1;
var u = 2;
{ var t__at_5 = t; t = u; u = t__at_5; }
```

- t__at_5 はマクロ展開アルゴリズムによって衝突を回避できるよう生成された名前
  - もちろん，t__at_5 が使われていたらそれとも衝突しないように名前を生成する

現在の実装における問題点・制約

- 構文
  - マクロ定義のための構文が固定
  - マクロ構文定義にOMetaをそのまま使っている
- 拡張モジュール
  - 手でカスタマイズしたものを用意する必要がある
  - スコープの導入はブロック構文にのみ対応
    - 例えば，Lisp風の構文で導入されるスコープはかなり書き換えないと難しい
  - マクロ展開に一部非衛生的なところがある

まとめ

- OMeta/JSを用いて実装された構文解析器に衛生的マクロの定義機構を導入するためのフレームワークOMetaMacroを提案・実装
- 今後の課題
  - 適用可能な構文に関する制限を明確にする
  - 導入されるマクロ定義機構の改善
  - 様々な言語に適用して有効性を評価
  - 構文解析以降の処理への応用
    - 最適化，安全性検査，etc.