多言語に対応した
衛生的マクロ定義機構導入方式

高桑 健太郎　渡部 卓雄
東京工業大学

概要

- OMetaで記述された構文解析器を持つ言語処理系に対して、衛生的マクロ機構を導入する対象言語に依存しない手法を提案
- 衛生的マクロの実装は、RacketのScope Setモデルがベース
- JavaScriptサブセットおよびMinCamlの構文解析器をOMetaで設計し、これらに本手法を適用することで有効性を確認

衛生的マクロ(Hygienic Macro)

- マクロ展開による思わぬ識別子の衝突を防止するマクロ
  - マクロ引数とマクロ内部の変数との名前が衝突する

```
#define swap(a, b) { int t = a; a = b; b = t; }
int main(void) {
    int t = 10, u = 20;
    swap(t, u)
    printf(“%d %d\n”, t, u);
}
```

  - マクロ内の変数参照が異なる変数束縛を指す

```
int count = 0;
#define COUNTER (++count)
int main(void) {
    { int count = 50; printf(“%d\n”, COUNTER); }
}
```

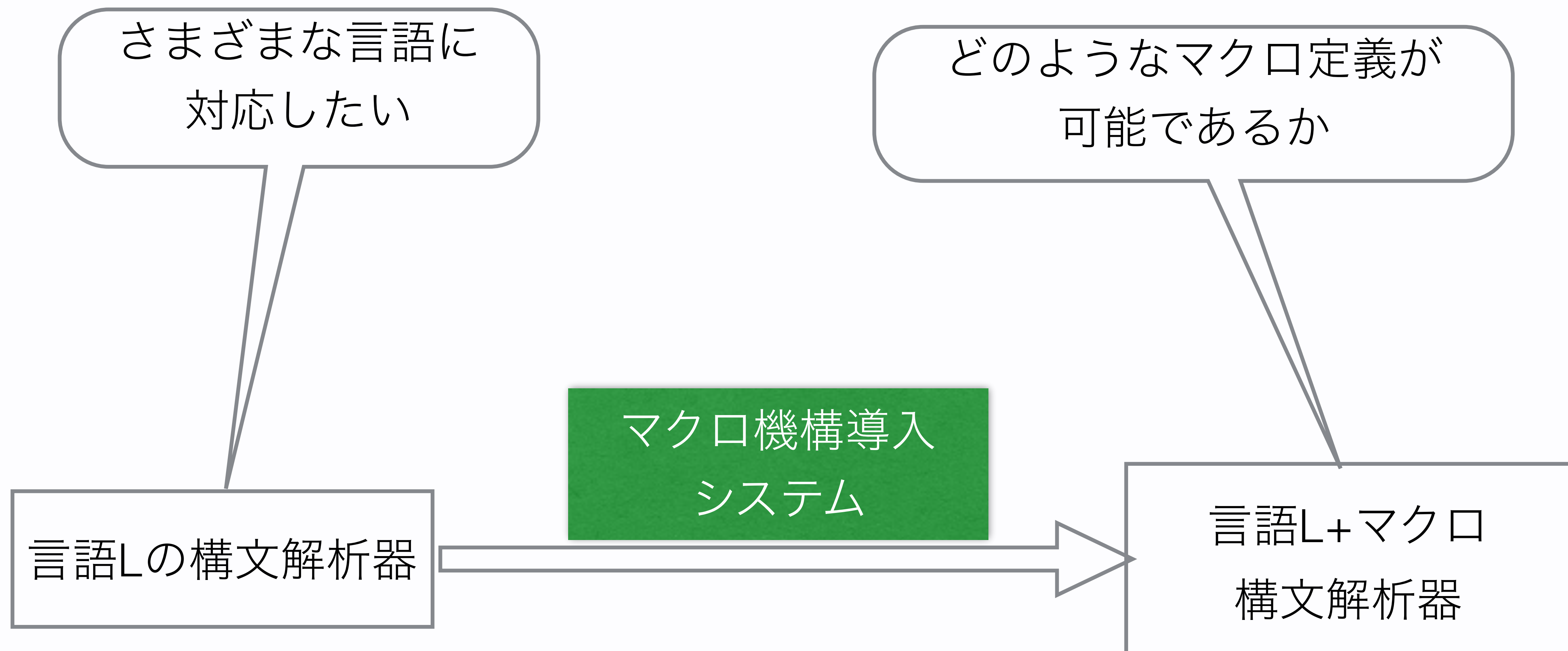
- 衛生的マクロを持つ言語: Scheme, Racket, Elixir, Rust etc.

衛生的マクロ機構の導入

- プログラミング言語に対して衛生的マクロ機構を導入するためには次のような機能が必要
  - マクロを定義するための記法の導入
    - C言語: `#define`, Scheme: `define-syntax`等
  - マクロ定義から構文解析器を拡張する機構
  - 衛生的にマクロを展開する機構

本研究の目的

- さまざまな言語の構文解析器に対して衛生的マクロ機構を提供するための一般的な方法を考案し、その有用性を示す



関連研究

伝統的な衛生的マクロの研究(1986~)の多くはSchemeが対象

JavaScript向け実装：**EX-JS**[甫水 2013], **Sweet.js**[Disney 2014]

他にも、特定の言語向けの実装がいくつか存在している

OMetaMacro[星野 2016]

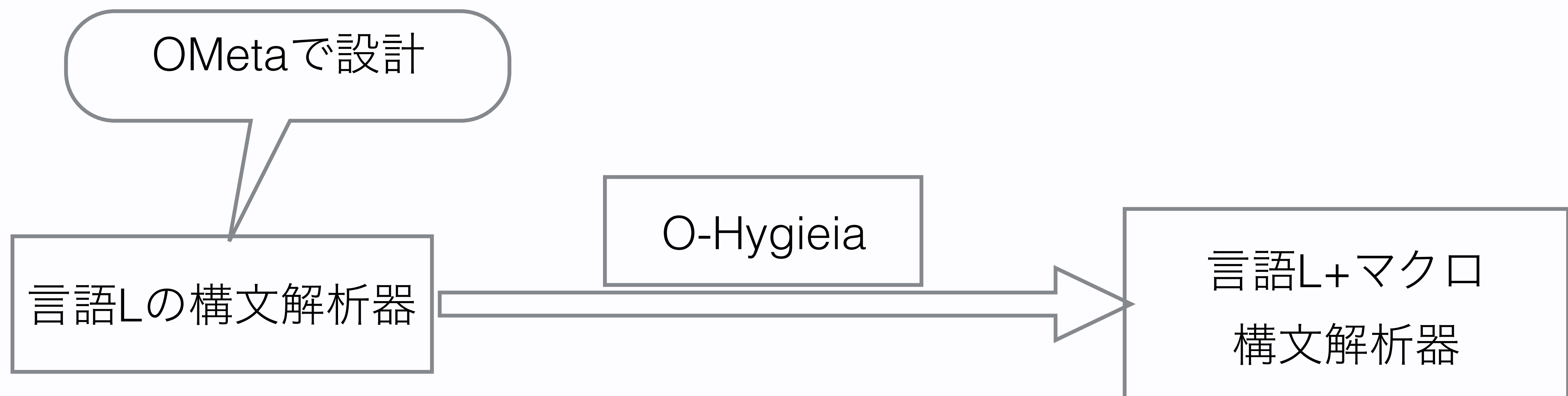
- **OMeta**[Warth 2007]で書かれた字句解析器・構文解析器に対し、一から実装するよりも容易に衛生的マクロを導入できるツール

OMetaMacroの課題

- 衛生的なマクロ展開に失敗するケースがある
- 検証用の言語の仕様に依存した箇所が見られる

現在の成果

- OMetaで書かれた構文解析器に衛生的マクロ機構を導入する手法を考案し実装した → **O-Hygieia**
- JavaScriptサブセットおよびMinCamlの構文解析器をOMetaで設計し、O-Hygieiaを適用した
  - 複数のテストケースで実験を行い、有効であることを確認した



OMeta [Warth 2007]

- PEGベースのパターンマッチ機構を備えたプログラミング言語
  - 構文解析器の設計に適している

```
ometa Calc { // 四則演算を OMeta/JS で実装した例
    Expr = Term:l '+' Expr:r -> (l + r)
         | Term:l '-' Term:r -> (l - r)
         | Term,
    Term = Factor:l '*' Term:r -> (l * r)
         | Factor:l '/' Term:r -> (l / r)
         | Factor,
    Factor = '(' Expr:e ')' -> e
           | Number,
    Number = <digit+>:digs -> parseInt(digs, 10)
}
```

マクロ定義の構文

- 以下のようなマクロ定義の構文を提供する

```
defsyntax `pattern` rule => `template`
```

- *pattern*: マクロ呼び出しの形式
  - 引数は %[*param rule*] の形式で書く
- *rule*: 定義するマクロをどの規則に追加するか(文、式など)
  - OMetaで記述したものを入れる
- *template*: マクロをどのように展開するか
  - 引数は %[*param*] の形式で書く

導入のための拡張

- さまざまな言語に対して衛生的なマクロ展開を行うためには、それぞれの言語のスコープの振る舞いに適応する必要がある
- 構文解析器の結果にスコープの振る舞いが含まれるように拡張する必要がある
  - 「この文法要素では変数が宣言されている」
  - 「この文法要素は変数を参照している」
  - 「スコープはこの文法要素で有効である」

導入のための拡張

- 例: OMetaで書かれたMinCamlの構文解析器 (一部抜粋)

```
ometa MinCamlParser {
  ...
  Ident = Type(‘IDENT’):id -> {
    type: ‘Var’, name: id
  },
  Exp = Token(‘LET’) Ident:id Token(‘EQUAL’) Exp:val_exp
        Token(‘IN’) Exp:body -> {
    // let <id> = <val_exp> in <body>
    type: ‘Let’,  id: id,  val_exp: val_exp,  body: body
  }
  ...
}
```

導入のための拡張

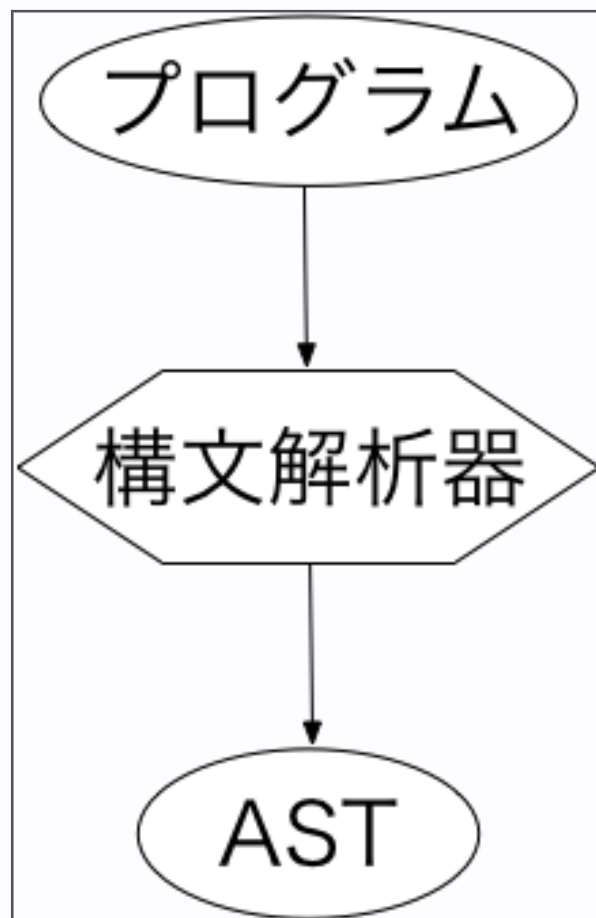
- 例: OMetaで書かれたMinCamlの構文解析器 (一部抜粋)

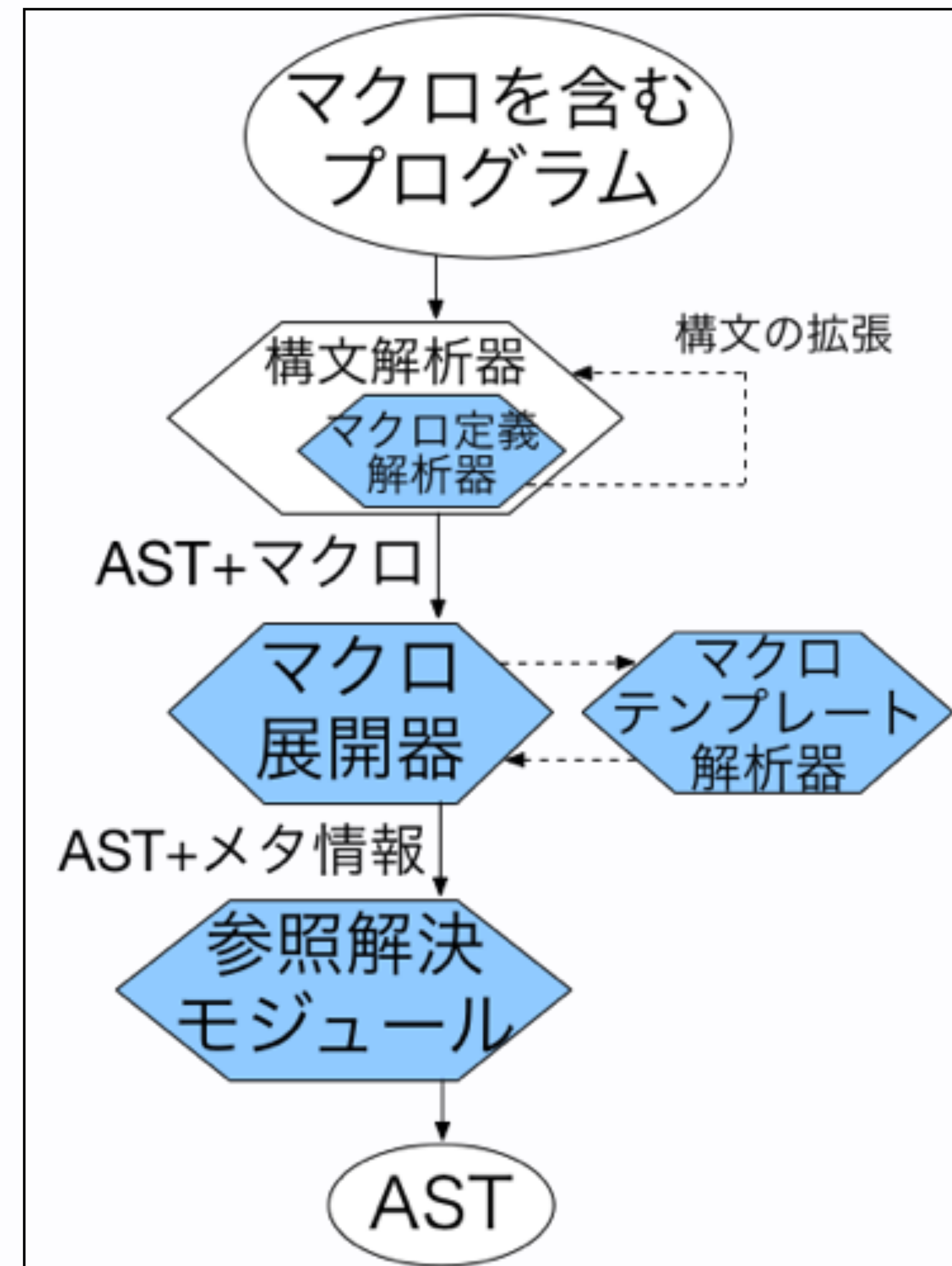
```
ometa MinCamlParser {
  ...
  Ident = Type('IDENT'):id -> {
    type: 'Var', name: id, _x_ref: 'name' //変数ノードを表す
  },
  Exp = Token('LET') Ident:id Token('EQUAL') Exp:val_exp
        Token('IN') Exp:body -> {
    // let <id> = <val_exp> in <body>
    type: 'Let',  id: id,  val_exp: val_exp,  body: body,
    _x_create_scopes: ['id'], // 変数束縛を記述する
    _x_scope_specs: [
      { type: 'inner', from: '', to: ['body'] }
    ] // 変数束縛がどこで有効なのかを記述する
  }
  ...
}
```

O-Hygieiaの設計

O-Hygieiaは4つのモジュールで
構成されている

O-Hygieia 適用後





マクロ定義解析器

- マクロ定義の構文 **defsyntax** \`*pattern*\` *rule* => \`*template*\` を解析し、*pattern*を解析する規則を構文解析器に動的に追加

プログラム中の文

```
defsyntax `SWAP %[a Variable] %[b Variable];` Stmt
=> `{ let t = %[a]; %[a] = %[b]; %[b] = t; }`
// 以降で新しい構文が利用できる


let t = 10, u = 20;
SWAP t u;
print(t, u);
```

マクロテンプレート解析器

- マクロ定義の構文のテンプレートの部分を解析する
  - 規則 *rule* で解析を行うが、中には引数 %[*param*] も含まれているためそのための拡張も行う
  - パターンに存在しない %[*param*] などがあればエラー

```
defsyntax `SWAP %[a Variable] %[b Variable];` Stmt
=> `{ let t = %[a]; %[a] = %[b]; %[b] = t; }`
```

マクロ展開モジュール

- マクロを含んだ抽象構文木の中をマクロを展開しながら、スコープの情報を抽象構文木に追加 (Scope Setモデル[Flatt 2016])

```
let t^{bind0} = 10, u^{bind0, bind1} = 20;
{
  let t^{macro2, bind4} = t^{bind0, bind1, use3};
  t^{bind0, bind1, use3, bind4} = u^{bind0, bind1, use3, bind4};
  u^{bind0, bind1, use3, bind4} = t^{macro2, bind4};
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

bind0: 始めの変数tのスコープ　*bind1*: 変数uのスコープ

macro2: マクロのテンプレートから導入された変数に与えるスコープ

use3: マクロの引数にあった変数に与えるスコープ

bind4: マクロ内から導入された変数tのスコープ

参照解決モジュール

- 変数にスコープの情報が与えられた抽象構文木に対して、適切に変数名の変更を行う
- 自由変数には変更を行わない
- 現在は (元の変数名) + '_' + (通し番号) が基本方針
  - 自由変数とは重複しないようにする

処理終了後のAST

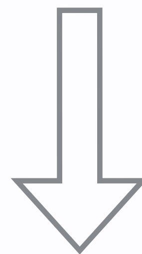
```
let t_0 = 10, u_2 = 20;
{
    let t_1 = t_0;
    t_0 = u_2;
    u_2 = t_1;
}
print(t_0, u_2);
```

適用例

- MinCamlの構文解析器への適用例

```
defsyntax `PRINT! %[x SimpleExp]` Exp
=> `print %[x]`

let print = 1234 in PRINT! print
```



展開後のAST

```
let print_0 = 1234 in print print_0
```

まとめ

- OMetaで書かれた構文解析器に対して
  容易に衛生的なマクロを導入する O-Hygieia を提案した
- JavaScriptサブセットおよびMinCamlの構文解析器に対して
  O-Hygieiaを適用し、その有効性を示した

今後の課題

- マクロ定義構文において、構文解析の規則名を直接指定することは
  設計者以外の利用者にとって難しい
- より柔軟なマクロ定義
  - 任意長のパラメーター ／ 意図的に識別子を衛生的でなくする
- 出力が抽象構文木 → コードに逆変換等で扱いやすく
- 提案手法で導入されるマクロ定義機構の性質を明らかにする

補足: SWAPの衛生的なマクロ展開の過程

```
let t = 10, u = 20;
SWAP t u;
print(t, u);
```

let $t^{\{bind0\}}$ = 10, $u^{\{bind0,\ bind1\}}$ = 20;

SWAP $t^{\{bind0,\ bind1\}}$ $u^{\{bind0,\ bind1\}}$;

$print^{\{bind0,\ bind1\}}$($t^{\{bind0,\ bind1\}}$, $u^{\{bind0,\ bind1\}}$);

bind0: 始めの変数tから導入されたスコープ

bind1: 変数uから導入されたスコープ

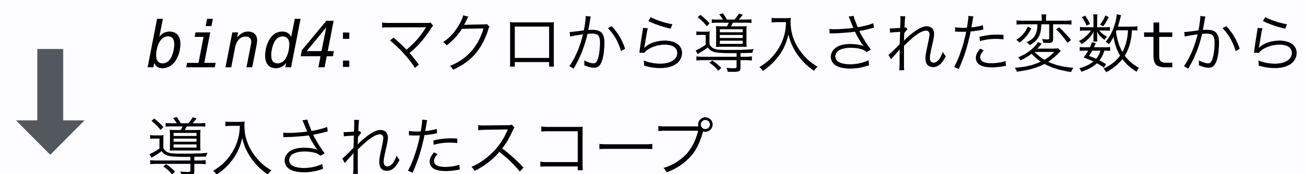
```
let t^{bind0} = 10, u^{bind0, bind1} = 20;
SWAP t^{bind0, bind1} u^{bind0, bind1};
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

```
let t^{bind0} = 10, u^{bind0, bind1} = 20;
{
  let t^{macro2} = t^{bind0, bind1, use3};
  t^{bind0, bind1, use3} = u^{bind0, bind1, use3};
  u^{bind0, bind1, use3} = t^{macro2};
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

macro2: マクロのテンプレートにあった引数に与えられるスコープ

use3: マクロの実引数にあった引数に与えられるスコープ

```
let t^{bind0} = 10, u^{bind0, bind1} = 20;
{
  let t^{macro2} = t^{bind0, bind1, use3};
  t^{bind0, bind1, use3} = u^{bind0, bind1, use3};
  u^{bind0, bind1, use3} = t^{macro2};
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```




```
let t^{bind0} = 10, u^{bind0, bind1} = 20;
{
  let t^{macro2, bind4} = t^{bind0, bind1, use3};
  t^{bind0, bind1, use3, bind4} = u^{bind0, bind1, use3, bind4};
  u^{bind0, bind1, use3, bind4} = t^{macro2, bind4};
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

参照解決

```
let t_0 = 10, u_2 = 20;
{
  let t_1 = t^{bind0, bind1, use3};
  t^{bind0, bind1, use3, bind4} = u^{bind0, bind1, use3, bind4};
  u^{bind0, bind1, use3, bind4} = t^{macro2, bind4};
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

以下のように新しい変数表を作る

$t^{\{bind0\}}$ → t_0

$t^{\{macro2,\ bind4\}}$ → t_1

$u^{\{bind0,\ bind1\}}$ → u_2

```
let t_0 = 10, u_2 = 20;
{
  let t_1 = t^{bind0, bind1, use3};
  t^{bind0, bind1, use3, bind4} = u^{bind0, bind1, use3, bind4};
  u^{bind0, bind1, use3, bind4} = t^{macro2, bind4};
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

$t^{\{bind0\}}$ → t_0

$t^{\{macro2,\ bind4\}}$ → t_1

$u^{\{bind0,\ bind1\}}$ → u_2

$t^{\{bind0,\ bind1,\ use3\}}$
が参照するのは
$t^{\{bind0\}}$

```
let t_0 = 10, u_2 = 20;
{
  let t_1 = t_0;
  t^{bind0, bind1, use3, bind4} = u^{bind0, bind1, use3, bind4};
  u^{bind0, bind1, use3, bind4} = t^{macro2, bind4};
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

$t^{\{bind0\}}$ → t_0

$t^{\{macro2,\ bind4\}}$ → t_1

$u^{\{bind0,\ bind1\}}$ → u_2

$t^{\{bind0,\ bind1,\ use3,\ bind4\}}$
が参照するのは
$t^{\{bind0\}}$

```
let t_0 = 10, u_2 = 20;
{
  let t_1 = t_0;
  t_0                                  = u^{bind0, bind1, use3, bind4};
  u^{bind0, bind1, use3, bind4} = t^{macro2, bind4};
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

$t^{\{bind0\}}$ → t_0

$t^{\{macro2, bind4\}}$ → t_1

$u^{\{bind0, bind1\}}$ → u_2

$u^{\{bind0, bind1, use3, bind4\}}$
が参照するのは
$u^{\{bind0, bind1\}}$

```
let t_0 = 10, u_2 = 20;
{
  let t_1 = t_0;
  t_0                          = u_2;
  u_2                          = t^{macro2, bind4};
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

$t^{\{bind0\}}$ → t_0

$t^{\{macro2,\ bind4\}}$ → t_1

$u^{\{bind0,\ bind1\}}$ → u_2

$t^{\{macro2,\ bind4\}}$

が参照するのは

$t^{\{macro2,\ bind4\}}$

```
let t_0 = 10, u_2 = 20;
{
  let t_1 = t_0;
  t_0                                     = u_2;
  u_2                                     = t_1;
}
print^{bind0, bind1}(t^{bind0, bind1}, u^{bind0, bind1});
```

$t^{\{bind0\}}$ → t_0

$t^{\{macro2,\ bind4\}}$ → t_1

$u^{\{bind0,\ bind1\}}$ → u_2

$print^{\{bind0,\ bind1\}}$
が参照する束縛は存在しない

→自由変数なので変更しない

```
let t_0 = 10, u_2 = 20;
{
  let t_1 = t_0;
  t_0                                    = u_2;
  u_2                                    = t_1;
}
print(t_0, u_2);
```

$t^{\{bind0\}}$ → t_0

$t^{\{macro2,\ bind4\}}$ → t_1

$u^{\{bind0,\ bind1\}}$ → u_2

完成形