神奈川大学理学部　情報科学科　馬谷研究室

Python のための疑似引用とそれを用いた手続き的マクロ

馬場 優菜 202103335

1　はじめに

プログラミング言語におけるマクロ機能は，コードの再利用性や記述の簡潔性を高める手段として広く利用されている．しかし，Python はマクロ機能を提供していない．本研究では，Python におけるマクロ機能の実現を目指し，新たな構文「s 文字列構文」を提案する．この構文は，文字列中の式を抽象構文木（Abstract Syntax Tree, 以下「AST」と呼ぶ）として生成する機能を持つ．また，この構文内で「疑似引用中の評価」を表す印を利用することで，特定のコード片のみを評価し，その結果を AST ノードの一部として埋め込む仕組みを実現する．この s 文字列を用いることにより，さらに Python 上の手続き的マクロが実現できる．

2　s 文字列構文

s 文字列構文は，Python の既存の f 文字列構文に着想を得て設計した新しい構文である．この構文は，文字列中の式を解析し，Python オブジェクトとして表現された AST を生成する機能を持つ．AST は，コードの構造を表現する木構造データであり，Python の内部でプログラムの解析時や実行時に利用される重要なデータ構造である．

s 文字列構文の特徴的な機能として，疑似引用の概念を採用している．疑似引用とは，s 文字列構文内で特定の印（‘{|’ と ‘|}’）を用いることで，その部分のみを動的に評価し，その結果を AST ノードの一部として埋め込む仕組みである．これにより，通常の Python コードによってコード変換や動的なコード生成が可能となる．

以下に，s 文字列構文を利用したコード例を示す．

```
x = s"3"
s"1 + x"
s"1 + {|x|}"
```

この例の 2 行目の場合では x が評価されずそのままの状態で AST ノードとして組み込まれる．一方，3 行目の場合では{|x|}が評価され，その結果の 3 が AST ノードとして組み込まれる．よって，それぞれ図 1 のような抽象構文木が返ってくる．

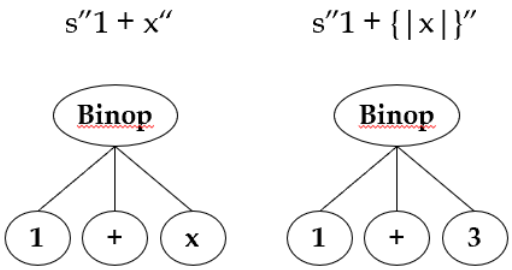

図 1　抽象構文木のイメージ図

3　マクロ機能

本研究では，s 文字列構文の仕組みを活用して，さらにマクロ機能を実現する．ここでは，そのためのマクロ定義構文について説明する．マクロは，s 文字列構文を作成する際に用いた仕組みを基盤として構築されている．通常の関数定義とマクロ定義であることを区別するために ‘!’ を用いる．また，マクロの呼び出し時にも ‘!’ を用いることで区別する．

以下にマクロ定義構文を利用したコード例を示す．

```
def mf!(n, m):
    return s"{|n|}+{|m|}"

def p(a, b):
    return mf!(a, b)

p(1, 2)
```

マクロ展開によって，上の定義構文 p(a, b) は以下に置き換わる．

```
def p(a, b):
    return a+b
```

この展開後の関数定義 p が Python に登録され，p(1, 2) と呼び出すと p には return a+b としか書かれていないため，1+2 の計算結果である 3 が返って

くる．

4 実装

ここでは，s 文字列構文とマクロ定義構文の実装について説明する．

s 文字列構文は，f 文字列構文と同様にクォートで囲まれた部分を文字列として認識し，構文解析や AST 生成のプロセスを通じて動作する．

s 文字列構文の解析では，文字列を解析する際に以下の処理を行う：

1. クォートで囲まれた部分全体を QuotedValue という名前のノードとして扱う．
2. QuotedValue ノードの下に，クォートで囲まれた式の AST を生成する．
3. クォート内で ‘{|’ . . . ‘|}’ を検出した場合，それを UnquotedValue という名前のノードとして扱い，その下に. . . 部分の AST 生成する．

この処理により，s 文字列構文内の式を適切に AST として分離し，後続のコンパイラで利用可能な構造を作成する．

コンパイル段階では，QuotedValue および UnquotedValue を他のノードと同様に式の一部として扱う．それぞれについて，以下のようにバイトコード命令を命令列に追加する．

```
static int
compiler_visit_expr1(struct compiler *c,
                     expr_ty e)
{
    switch(e->kind) {
    case BinOp_kind:
        /* 二項関数を計算するための
           バイトコードを積む処理 */
    ・・・
    case QuotedValue_kind:
        /* s 文字列構文のための関数に移動 */
    case UnquotedValue_kind:
        / *s 文字列構文の外なのでエラー */
    }
}

static int
compiler_visit_sstring1(struct compiler *c,
                        expr_ty e)
{
    switch(e->kind) {
    case BinOp_kind:
        /* BinOp の AST を生成する
           バイトコードを追加する処理 */
    ・・・
    case UnquotedValue_kind:
        /* 通常の AST を辿る関数に移動 */
    }
}
```

s 文字列構文の実装では，Python 標準の ast モジュールの機能を利用して AST を生成する．そのため，s 文字列構文がコード内で使用される場合，暗黙的に `import ast` を挿入する仕様となっている．

マクロと通常の関数の最も大きな違いは，コンパイラに渡される AST の形式にある．通常の関数では，関数呼び出しの際に Call という名前のノードの AST が生成される．一方，マクロでは関数内のブロックそのものが AST として生成され，直接コンパイラに渡される．この仕組みにより，マクロは関数とは異なる方法で AST を操作し，コード生成を可能にしている．

ユーザによりマクロが呼び出されると，まず引数として渡された各式が解析され，それぞれ AST に変換される．マクロ内で，引数の AST の埋め込みが完了した式全体が AST として生成され，コンパイラに渡される．これにより，マクロは実行時ではなく，コンパイル時にコード変換を行う．

5 まとめ

本研究では，Python におけるマクロ機能の実現を目的として，新たな構文「s 文字列構文」を提案し，その設計と実装について述べた．s 文字列構文は文字列中の式から AST を生成する仕組みを提供する．また，疑似引用の概念を導入することで，文字列内の一部を評価し，その結果を AST の一部として埋め込むマクロ機能を実現した．

s 文字列構文は Python における柔軟なコード生成の可能性を拡げる機能である．しかし，AST 生成に伴う性能面での課題や汎用性の高さについては，さらなる検討が必要である．

参考文献

[1] Anthony Shaw and The realpython.com tutorial team, *CPython Internals: Your Guide to the Python 3 Interpreter,* Real Python, 2021.

[2] Python 公式サイト，`https://www.python.org`.