神奈川大学理学部　情報科学科　馬谷研究室

ast.py を用いて記述可能な Python 用手続き的マクロ

栗巣　健一 201903009

1　はじめに

本来 Python には定義されていないマクロシステムを実現することにより、Python のプログラミングの幅がより広がり、便利になる。例えば今は Python の構文にはない sum!() や defmacro のようなマクロ構文を追加することである。

今回の研究では新たな構文を追加し、実際に追加した構文を用いてマクロシステムの作成の検討を行う。ユーザーが defmaco の関数を作成したときに def の AST に変更させ、辞書にその関数を追加させるような AST もフレームワークが自動的に作成できるようにする。

本研究での構文追加に関して CPython を用いて、CPython を実装しているコードの変更、追加をしながらマクロシステムを実装する。

2　CPython

Python 言語の標準の処理系である、CPython についてどのようにコードが実行されていくかの流れを説明する。　コードを実行するとき初めにファイルからテキストが読み込まれ、字句解析が行われ具象構文木が生成される。ここから構文解析をし、抽象構文木を作っていく。抽象構文木をコンパイラに渡し、木のノードを一つ一つ辿っていき CFG(コントロールフローグラフ) を作成する。この CFG をアセンブラに渡して、バイトコードに変換することにより Python コードを実行可能できるようにする。[1]

今回は作成された抽象構文木がコンパイラに渡される間の中で抽象構文木を変換させ、マクロシステムをユーザーが使えるようにしていく。抽象構文木というのはプログラムの文法構造を木にしたものであり、具体的な例で表すと、関数定義をする際のコードを抽象構文木にすると、name という関数名が入っている木、引数がはいいている args という木、そして body という関数の中のコードを入れる木のリストがある。

今回の研究では、関数定義をする際には def args():... という形で作られるがそれと同じ動きをする。defmacro args():... という文法を考える (マクロ関数と呼ぶことにする)。そしてこのマクロ関数を呼び出すときには args!!()（マクロ呼び出しと呼ぶ）と呼び出すことにし、この構文も追加していく。

2.1　マクロ

マクロとはコード変換のことであり、コードがコンパイルされる前や評価される前にもともと書いていたコードが変更され変更後のコードが最初から書かれていたかのような振る舞いをする。下記は Scheme 言語でのマクロの例である。

```
(define-syntax nil!
  (syntax-rules ()
    ((_ x)
     (set! x '()))))
```

`((_ x) (set! x'()))` は，もとの式と変換後の式を記述した組である．上記のプログラムは `(nil! x)` という式を `(set! x '())` に変換するという意味になる．

3　設計

ユーザーが Python でマクロ関数を定義して呼び出すコードの例

```
defmacro x():
    pass
```

```
x!!()
```

この上記のコードの AST が作成されて変換させた後のコードの構想は下記のようになる。

```
def x():
    pass
macro_table["x"] = x
```

抽象構文木の変換を通してマクロ呼び出しの部分を辞書に関数名を追加するコードに変換していく。

4 実装

抽象構文木の変換というのはどういうものなのかの例を示す。x + y というコードがあったとしてこれを抽象構文木にすると

```
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

[2]

これは、BinOp クラスに left,op,right というノードがありそれぞれにクラスが入っている。op のノードを Add() クラスから Sub() クラスに変えると抽象構文木は、

```
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Sub(),
        right=Name(id='y', ctx=Load())))
```

となり、このコードは x - y となる。これが抽象構文木を変換させるということである。

抽象構文木を変換させる前に、今 Python ではマクロ関数とマクロ呼び出しの構文を定義させる必要がある。まず初めに、CPython にある FunctionDef クラスと同じ構造の MacroDef クラスを追加し、Call クラスとほとんど同じような構造の CallMacro クラスを追加する。この 2 つは def の代わりに defmacro、args() の呼び出しの代わりに args!!() と記述できるように定義する。この定義で MacroDef と CallMacro それぞれの AST が作成できるようになった。

抽象構文木を変換させるためには、木を再帰的に探索して変換させたいノードを発見したときにそのノードを変換させるような処理を書く必要がある。ソースコードの抽象構文木を作成して、コンパイラに渡される間で変換処理を加える。一つの関数ですべてまとめて処理するには少し煩雑なので 2 に関数を分けて処理をさせることにする。まずソースコード内に MacroDefKind があるかを for 文を使って調べる。あった場合、makeassignast という関数を呼び出す。この関数は MacroDefKind を FunctionDefKind に入れ替えて、関数名の情報を抜き出しながら、macrotable[”func”] = func という AST を生成する。これで AST 変換を行いつつ、マクロの関数名の登録まで行うことが出来る。

5 まとめ

今回の研究では CPython に CallMacro と MacroDef という構文の実装、この二つがソースコードん中に現れた時に、MacroDef を FunctionDef に、CallMacro を Assign に変更させ、関数名を辞書に追加するという部分まで進めた。AST の変換は木を for 文で巡回しながら特定のノードを見つけた時にそれぞれの AST 変換を行うように実装した。本来のマクロ展開とは違うような形になってしまったので、今後の課題としては疑似クォート記法を書けるようにすることで、より簡潔にマクロを定義が出来るように改善したいと考えている。

参考文献

[1] Anthony Show, CPython Internals: Your Guide to the Python 3 Interpreter, Real Python, 2021.

[2] https://docs.python.org/3/library/ast.html