神奈川大学理学部　情報科学科　馬谷研究室

Python 用マクロフレームワークの実現に向けて

吉野 友貴 201803451

1　はじめに

通常の Python では利用できない機能をマクロフレームワークを実現することによって追加できるようにするとプログラミング幅が拡がり，非常に便利になる．例えば，if 文の逆であり条件が成り立たない時に処理をする unless 文を追加したいときはユーザが関数を定義するだけで unless を使えるようにするというものである．

今回の研究では抽象構文木の自動変換を行うフレームワークを作るところまでを研究とする．ユーザには置き換える対象の抽象構文木を引数として受け取り，変換後の抽象構文木を返す処理を C 言語の関数として定義してもらい，フレームワークが自動で変換するというものである．

本研究では CPython を扱う．CPython とは C 言語によって書かれた Python 処理系で，最も主要な Python の処理系である．

2　Scheme マクロ

ここでは Scheme 言語を例にして，マクロについて説明を行う．マクロとはコードの変換のことで，コードが評価される前，またはコンパイル時にコードが変更され，変更後のコードが初めからソースコードにあったかのように実行される．マクロの簡単な例を Scheme 言語で以下に示す．

```
(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　CPython

ここでは Python 言語の標準の処理系である CPython においてソースコードがどのように解析されて抽象構文木になるのかと，抽象構文木の構造について説明する [1].

初めに，ソースコードからテキストを読み込み，字句解析器に渡す．字句解析器を使って具象構文木を作成する．具象構文木を構文解析器に渡し，構文解析器を使って具象構文木から抽象構文木を作成する．抽象構文木をバックエンドに渡し，バックエンドで抽象構文木を枝分かれなどのある複雑な構造を漏れなく辿っていき，コントロールフローグラフを作成する．最後にアセンブラでコントロールフローグラフ内のノードをバイトコードに変換することで Python を実行可能にする．

今回の研究では抽象構文木を変換させて，その変換した抽象構文木をバックエンドに渡すことでユーザーが新しい構文を追加できるようにする．

抽象構文木はプログラムの文法構造を木の形で表したものである．if 文を例に挙げると，test という条件式，body という test の条件が成り立つ時に行う処理，orelse という test の条件が成り立たない時に行う処理を if は木として持っている．マクロによって追加する構文の例として unless 文を考える．unless を追加するには，unless も if と同じように test，body，orelse という木を持っているので，test の条件式が成りたたない時は body を処理するようにして，test の条件式が成り立ちときは orelse を処理するようにすれば unless 文を追加できる．

4 設計

マクロを追加するユーザは，次のような定義を C 言語で行う．

```
stmt_ty
macro_stmt(stmt_ty s){
  /*unless の木である
        test, body, orelse の処理*/
  return
    If(new_test,s->v.Unless.orelse,
      s->v.Unless.body, s->lineno,
      s->col_offset, s->end_lineno,
      s->end_col_offset,arena);
}
```

上記のようにユーザが定義した関数を抽象構文木中の unless 文を引数にして呼び出すことにより，抽象構文木を変換させる．

5 実装

抽象構文木を変換させるには再帰的に木を探索しなければならない．switch 文を使い，場合分けをしながら抽象構文木を作成する処理を行う．文の型である stmt_ty 型の関数を例にして以下に示す．

```
stmt_ty macro_stmt(stmt_ty s)
{
  switch (s->kind) {
    case FunctionDef_kind:
      /*FunctionDef の AST を作成する処理*/
    case ClassDef_kind:
      /*ClassDef の AST を作成する処理*/
    /*その他 case 文*/
    case While_kind:
      /*While の AST を作成する処理*/
    case If_kind:
      /*If の AST を作成する処理*/
}
```

さらにそれぞれの case の中で再帰的に処理することにより，全ての木を探索する事ができる．ここでは if 文を例に上記の巡回処理を具体的に示す．

```
case If_kind:
  new_test = macro_expr(s->v.If.test);
  int i;
  asdl_seq *seq = (s->v.If.body);
  for(i = 0;i < asdl_seq_LEN(seq);i++){
    /*body で macro_stmt を
                再帰する処理*/
  }
  if (s->v.If.orelse){
    int i;
    asdl_seq *seq = (s->v.If.orelse);
    for(i = 0;i < asdl_seq_LEN(seq);i++){
      /*orelse で macro_stmt を
                    再帰する処理*/
    }
  }
  return If(new_test, s->v.If.body,
    s->v.If.orelse, s->lineno,
    s->col_offset, s->end_lineno,
    s->end_col_offset, arena);

  break;
```

if 文の条件式である test は式であるため macro_expr() を呼び出して，式から式への変換を行う．一方，body や orelse は文であるため，macro_stmt() を呼び出して文から文への変換を行う．if 文の内側にも別のマクロ呼び出しが書かれているかも知れないため，このように再帰的に変換を行う必要がある．最後に test や body，orelse のような引数にして If という関数を返すことによって If の AST を生成する事ができる．

6 まとめ

本研究では Python 用マクロフレームワークの実現に向けて，抽象構文木を自動で変換させるフレームワークを作成するところまで行った．抽象構文木の変換には switch 文で場合分けをして，その中で再帰関数を定義することで自動変換を行うようにさせた．今後の課題としては Python の対話環境にてユーザがライブラリを import して，追加したい関数を定義するだけでその構文を使えるように，マクロフレームワークを改善していく事である．

参考文献

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