メタプログラミング班
後期活動報告書

西見元希 [*1] 小泉孝弥 [*2] 中尾龍矢 [*3] 服部瑠斗 [*4] 八木田裕伍 [*3] 吉田享平 [*5]

[*1] 情報理工学部 セキュリティ・ネットワークコース 二回生
[*2] 理工学部 数理科学科 三回生
[*3] 情報理工学部 情報理工学科 一回生
[*4] 情報理工学部 知能情報コース 二回生
[*5] 情報理工学部 実世界情報コース 三回生

目次

1. はじめに

文責：西見 元希

この報告書は2019年の通年プロジェクトメタプログラミング班の後期活動報告書である．このプロジェクトの目的はメタプログラミングやその活用方法を学習・研究するというものである．また，発足背景として，RCCで久しく行われていない，「研究」を主目的としたプロジェクトを発足させたいというものと，好きな言語で特定の製作対象物を持たないコーディングをすることのできる場をRCCに設けたいというものがあった．週一回の活動で，個人活動の成果報告を行った．班員は全員自身の研究や学習の成果を文書形式で提出することとしていた．上回生は前期活動で研究した言語と別の言語について学習した．

2. 各研究内容成果

2.1 Rust

文責：西見 元希

非常に長い文章となったので別文書に記述した．そちらを参照されたし．

2.2 Python3

文責：小泉 孝弥

I. 背景

普段，私はPython3を用いて，機械学習アルゴリズムの実装を行なっている．そこで，Python3のメタプログラミングを学ぶことで，普段の実装の効率をあげることができるのではないかと考えた．

II. 内容

A. クラスの継承

Aをクラスとする． この時Aを親クラスとする子クラスBはclass B(A) とすることで定義できる．以下にその具体的なコードを記す．

```
class hoge():
    def __init__(self, language):
        self.language = language
    def show_language(self):
        print(self.language)

class hoge2(hoge):
    def __init__(self, language, detail_version):
        self.language = language
        self.detail_language = detail_version
    def show_detail_language(self):
        self.show_language() # 親クラスの関数
        print(self.detail_language)
```

code 1. クラスhogeを継承してクラスhoge2を定義する.

B. type classとクラス定義について.

Pythonにはtypeという機能がある. 普段は型を調べるために使用されるが, typeを用いることでクラスを定義することができる. 例えば, call_helloをメンバ関数に持つクラスTest_classは以下のように定義できる.

```
def call_hello():
    print('Hello!')

Test = type('Test_class', (), {'call_hello': call_hello})
```

code 2. Test_classを定義して, インスタンス化.

したがって, クラスの定義はtypeクラスのインスタンス化であるということができる.

C. metaclassとクラス定義の拡張.

metaclassというものを用いてクラスの定義を拡張することができる. それは以下のようにかける.

```
def meta_test(self, number):
    return number
class Metapiyo(type):
    def __new__(cls, name, bases, dictionary):
        dictionary['meta_test'] = meta_test
        return type.__new__(cls, name, bases, dictionary)
class piyo(metaclass = Metapiyo):
    pass
```

code 3. Metaclassを指定して，新しくクラスを定義する.

D. 応用

最後に今まで紹介した技術を用いて機械学習のアルゴリズムである多項式回帰を実装した.

```
def h(self, x):
    res = np.poly1d(self.W[::-1])
    return res(x)

def fit(self, x_train, t_train):
    X = np.ones((len(x_train), self.degree + 1))
    for i in range(len(x_train)):
        for k in range(1, self.degree + 1):
            X[i][k] = np.power(x_train[i], k)
    # デザイン行列の生成.
    W = np.linalg.inv(np.dot(X.T, X))
    W = np.dot(np.dot(W, X.T), t_train)
    self.W = W
    print(W)

def score(self, x_test, t_test):
    accuracy = 0
    for i in range(len(x_test)):
        accuracy += np.power(self.h(x_test[i]) - t_test[i], 2)
    accuracy = accuracy * (1/len(x_test))
    return 100 - np.sqrt(accuracy)

class MetaPR(type):
    def __new__(cls, name, bases, dictionary):
        dictionary['h'] = h
        dictionary['fit'] = fit
        dictionary['score'] = score
        return type.__new__(cls, name, bases, dictionary)
class PR(metaclass = MetaPR):
    def __init__(self, degree):
        self.degree = degree

model = MetaPR(degree = 5)
model.fit([1, 2, 3], [4, 5, 6])
```

code 4. メタプログラミングを用いた多項式回帰の実装

III. 成果

結果として，Python3でメタプログラミングを用いて，機械学習アルゴリズムの多項式回帰を実装することができた．しかしながら，メタプログラミングを用いない実装よりもコード量が多くなってしまうという欠点があり，使用する場面は限られると感じた．

2.3 C++

文責：中尾 龍矢

I. 背景

前期でのメタプログラミングの活動は，テンプレート関数やテンプレートクラスの基本的な使い方や，基礎的な部分を固める事を行いました．なので後期はその発展形として，関数ポインタの扱い方に着目して活動を進めていく事にしました．

II. 内容

A. 関数ポインタ

```
int add(int a, int b) {
 return a + b;
}

int (*add_pt)(int, int) = add;

int main(void) {
 printf("%d", add_pt(1, 2));

 return 0;
}
```

まず基本として関数ポインタの型の構文は，

返り値の型 (* 関数名)(第一引数, 第二引数, ...)

というように決められている．そして，関数アドレスは関数名で得ることができます．

B. 引数がテンプレート

```
template<typename T>
T add(T a, T b) {
 return a + b;
}
template<>
int add(int a, int b) {
 return a * b;
}

int main(void) {
 cout << add(1, 3) << endl;
 cout << add(1.5, 2.0) << endl;
 _getch();
 return 0;
}
```

少し前期の内容の復習，このようにしてテンプレート関数は定義されます．以上のようにtemplate<>のように書くことで特殊化が行える．

C. テンプレート関数をテンプレート関数で管理

```
template<typename T>
T addT(T a) {
 return a += (T)1;
}

template<typename T>
T method(T a) {
 T(*addT_pt)(T) = addT;
 return addT_pt(a);
}

int main(void) {
 cout << addT(3) << endl;
 cout << method(3) << endl;
}
```

以上のようにテンプレート関数ポインタもテンプレートパラメータFで変数宣言することによって代入を可能にしています.

D. 関数ポインタを引数に持つテンプレート関数

```
template<class T>
void Temp(T (*call_func)() ) {
 cout << call_func() << endl;
 return;
}

int function1() {
 cout << "This function1 was called" << endl;
 return 0;
}

int function2() {
 cout << "This function2 was called" << endl;
 return -1;
}

int main(void) {
 Temp(function1);
 Temp(function2);
}
```

このように，テンプレート関数の引数に関数ポインタを用意する場合は（このとき引き渡す関数の引数は0）返り値の変数をテンプレートパラメータFで変数宣言することによって可能にします.

E. 可変引数のテンプレート関数

```
template<class T>
T sum(T init, int count, ...) {
 va_list ap;
 va_start(ap, count);
 T sum = init;
 for (int i = 0; i < count; i++) {
  sum += va_arg(ap, T);
 }
 va_end(ap);
 return sum;
}

int main(void) {
 cout << sum(0, 3, 1, 2, 3) << endl;
 cout << sum(0.0, 3, 1.2, 4.4, 2.5) << endl;
 string str;
 str = sum((string)"ab", 3, (string)"bc", (string)"cd",
(string)"de");
 printf("%s", str.c_str());
}
```

与えられた引数をすべて総和をとり表示する可変引数関数を，テンプレートで実現したい場合，このように（第一引数（初期値）をとり，第二引数に個数を指定することにしました）va_arg(ap, T)にあるとおり，第二引数にテンプレートパラメータTを与えるこで対応できました.

F. 可変引数関数のポインタ

```
int sum(int count, ...) {
 va_list ap;
 va_start(ap, count);
 int sum = 0;
 for (int i = 0; i < count; i++) {
  sum += va_arg(ap, int);
 }
 return sum;
};

int main(void) {
 int (*pt)(int, ...) = sum;
 printf("%d", pt(3, 1, 2, 3));
}
```

可変引数のテンプレート関数を作ったのに対し，今度は可変引数のアドレスを取り出したいとき，普通の関数アドレスの取得方法と変わりありませんでした.

G. テンプレート関数のポインタ

```
template<class T>
T func(T a) {
 return a;
}

int main(void) {
 int(*pt1)(int) = func;
 float(*pt2)(float) = func;
 cout << pt1(5) << endl;
 cout << pt2(1.5) << endl;
}
```

テンプレート関数アドレスをテンプレートパラメータを用いずに変数に代入したい場合，戻り値の型，引数の型を指定する必要あります.

H. 可変引数のテンプレート関数のポインタ

```
<class T>
T func(T init, int count, ...) {
 va_list ap;
 va_start(ap, count);

 T sum = init;
 for (int i = 0; i < count; i++) {
  sum += va_arg(ap, T);
 }
 va_end(ap);
 return sum;
}

int main(void) {
 int(*pt1)(int, int, ...) = func;
 double(*pt2)(double, int, ...) = func;
 string(*pt3)(string, int, ...) = func;
 cout << pt1(0, 3, 1, 2, 3) << endl;
 cout << pt2(0.5, 3, 1.0, 2.0, 3.0) << endl;
 cout << pt3((string)"a", 3, (string)"b"
  , (string)"c", (string)"d").c_str() << endl;
}
```

Gで行ったことを可変引数の関数で行おうとすると，以上のようになります．

I. 関数のポインタをメンバに持つクラス

```
int func(int a) {
 return a + (5);
}

template<class T>
class C {
public:
 T(*func)(T);
};

int main(void) {
 C<int> c;
 c.func = func;
 cout << c.func(5) << endl;
}
```

テンプレートクラスでテンプレート関数ポインタを持ちたいとき，インスタンス生成時に型を指定する必要があります．

J. テンプレート関数のポインタをメンバに持つクラス

```
template<class T>
T func(T value) {
 return value + (T)5;
}

class C {
public:
 int(*func)(int);
};

int main(void) {
 C c;
 c.func = func;
 cout << c.func(5) << endl;
}
```

クラスの宣言部分でテンプレートパラメータに指定する型を明記する必要があります.

K. テンプレート関数のポインタを持つテンプレートクラス

```
template<class T>
T func(T value) {
 return value + (T)5;
}

template<class T>
class C {
public:
 T (*func)(T);
};

int main(void) {
 C<int> c1;
 C<double> c2;
 c1.func = func;
 c2.func = func;
 cout << c1.func(5) << endl;
 cout << c2.func(1.5) << endl;
}
```

この場合もインスタンス生成時にテンプレートパラメータに与える型を指定しなければいけません.

III. 成果

前期の基礎勉強のおかげで発展的な物事を調べていくのに大変役立ちました. 加えて今回のテンプレートと関数との深い関係性を学んでいけばいつか役に立つ日が来ると思っています.

2.4 Python3

文責：服部 瑠斗

I. 背景

最近Pythonを触ることが増えて,より効率のいい・読みやすいコードを作成したいという思いからPythonでのメタプログラミングの学習に着手した.

II. 内容

2.2 と同じくPython上でのメタプログラミングについて学習した. 2.2で既に説明がされている部分は省略する.

A. metaclass

metaclassとはmetaclassのインスタンスを生成するとそのインスタンスがclassであるものを指す.通常のclassはインスタンスを生成すると，そのインスタンスはobjectクラスである.インスタンスがobjectクラスかどうかは, isinstance関数を用いればわかる.

B. typeを用いてクラスを生成する

typeを用いてクラスを生成する場合と継承を用いてクラスを作成する場合の大きな差について説明していく.

```
class Base_A(type):
    def __new__(cls, name, bases, dict):
        print("クラスを生成しました")
        return type.__new__(cls, name, bases, dict)

class A(metaclass=Base_A):
    # コンストラクタ
    def __init__(self,message="Hello World!!!"):
            self.message = message

    def print_message(self):
            print(self.message)

# => クラスを生成しました
a = A()
a.print_message()
# => Hello World!!!
another = A("test")
another.print_message()
# => test
```

このようにtypeを用いてクラスを生成する場合は，クラスが生成される時のみprintが実行される.次に継承を用いた場合である.

```
class Base_B():
    def __init__(self):
        print("クラスを生成しました")

class B(Base_B):
    # コンストラクタ
    def __init__(self,message="Hello World!!!"):
            super().__init__()
            self.message = message

    def print_message(self):
            print(self.message)

b = B()
# => クラスを生成しました
b.print_message()
# => Hello World!!!

another = B("test")
# => クラスを生成しました
another.print_message()
# => Hello World!!!
```

継承を用いた場合では,親クラスの__init__関数が呼び出される度にprintが実行される.また､親クラスの__init__関数を呼び出さない場合は当然printは実行されない.

C. 応用

私が別サークルの方で開発を行っている強化学習用のライブラリの新機能を実装する時にメタプロを用いる予定だった．しかし，以下の理由で断念した.

- メタプロを用いて実装する難易度の高さ
- わざわざメタプロを用いて実装する必要がない
- 可読性が落ちる
- デバッグが行いにくい

III. 成果

具体的な制作物を作成することは出来なかったが,metaclassやtypeなどの知見を得られることが出来た.今後もメタプログラミングの知見を得られるように日々精進しようと考えている.

2.5 D言語

文責：吉田 享平

I. 背景

本プロジェクトを本格的に取り組もうとした際，SNS上で「D言語はメタプログラミングができる」という言説を目にした．D言語自体を触ったことがないかつメタプログラミングができるという理由からD言語を選択した.

II. 内容

A. 基本文法

1. ライブラリのインポートとmain関数

ライブラリのインポートはimport文で行う．Pythonと似ている．またプログラムのエントリーポイントはmain関数で行い，こちらはC/C++に似ている.

2. 変数宣言

D言語による変数宣言を以下に示す．一般的なプログラミング言語の変数宣言と似ている.

```d
void main() {
    // 変数定義
    int a = 810; // 整数型
    float f = 114.514f; // 小数型
    char c = 'c'; // 文字型
    string s = "文字列"; // 文字列型
    int[8] arr1; // 静的配列
    int[] arr2 = new int[8]; // 動的配列
}
```

3. 条件分岐

D言語による条件分岐を以下に示す．switch文においては簡潔に記述できる.

```
// 条件分岐
if(a < 10) {
    writeln("10未満だよ");
} else if(a < 20) {
    writeln("10以上20未満だよ");
} else {
    writeln("20以上だよ");
}

// これは上と同様
switch(a) {
    case 0: .. case 9:
        writeln("10未満だよ");
        break;
    case 10: .. case 19:
        writeln("10以上20未満だよ");
        break;
    default:
        writeln("20以上だよ");
        break;
}
```

4. ループ

D言語はwhile，do while，forといった基本的なループに加えてforeachもサポートしている．

```
    // while
    while(true) {
        // doSomething
    }
    // do while
    do {
        // doSomething
    } while(true);
    // for
    for(int i = 0; i < 10; i++) {
        writeln(i);
    }
    // foreach
    int[] arr = [1, 2, 3];
    arr.writeln;
    // セミコロンであることに注意
    foreach(int val; arr) {
        writeln(val);
    }
```

5. 関数

D言語による関数定義は以下の通りに行える．D言語ではデフォルト引数も利用できる．

```
int add(int a, int b) {
    return a + b;
}
// デフォルト引数も可
// e.g. add(10) => 20
int add(int a, int b = 10) {
    return a + b;
}
```

6. 構造体

D言語は構造体を定義できる．後述するクラスと似ているが構造体は常に値コピーされる．

```
struct Pokemon
{
    string name;
    int HP;

    // クラスでいうコンストラクタ
    this(string name, int HP) {
        this.name = name;
        this.HP = HP;
    }

    // メソッドも定義できる
    void damage(int AP){
        HP -= AP;
    }

    private void privateFunc(){
        writeln("this is private");
    }
}
```

7. class

classは構造体と似ているが，挙動が異なる．classは暗黙的にObjectクラスを継承する．またコピーするときは，参照コピーされる．

```
class Pokemon
{
    private string name;
    private int HP;

    // クラスでいうコンストラクタ
    this(string name, int HP) {
        this.name = name;
        this.HP = HP;
    }

    // メソッドも定義できる
    void damage(int AP){
        HP -= AP;
    }

    private void privateFunc(){
        writeln("this is private");
    }
}
```

B. D言語によるメタプログラミング

1. 簡単な例

メタプログラミングの簡単な例として汎用的な足し算の関数を定義した．まず一般的な足し算の関数を示す．

```
import std.stdio;

/// 足し算する関数
int plus(int a, int b) {
    return a + b;
}

void main() {
    immutable int a = 10, b = 20;
    // 30
    writeln(plus(a, b));
}
```

しかしながらこの例では，整数型（int型）の足し算しか演算できず，汎用性がない．これを解決するためにメタプログラミングの技術を駆使した，改善したコードを以下に示す．

```
/// 足し算する関数（改）
auto plusKai(T)(T a, T b) {
    return a + b;
}

void main() {
    immutable int a = 10, b = 20;
    // 30
    writeln(plusKai(a, b));

    // 小数型も可能
    immutable float c = 10.5f, d = 20.3f;
    // 30.8
    writeln(plusKai(c, d));
}
```

このように小数型にも対応でき，汎用性の高い関数が記述できた．

2. リストの実装

前節ではメタプログラミングの簡単な例を示した，本節ではこれを応用して汎用的になリストの実装を行う．仕様は以下の通りである．

- 様々な型の値が格納できる
- リストに追加ができる
- 任意のindexの要素が取得できる

リストを実装したものを以下に示す．

```
import std.stdio;
import std.conv;

/// リストのアイテム
class ListItem(T)
{
    private T value;
    private ListItem!(T) next;

    /// コンストラクタ
    this(T value) {
        this.value = value;
        this.next = null;
    }

    public T getValue() {
        return value;
    }
}

/// リスト
class List(T) {
    private ListItem!(T) head;

    this() {
        head = null;
    }

    /// 要素を追加します
    public void add(T newValue) {
        // headがnullの場合は先頭に追加
        if(head is null) {
            head = new ListItem!(T)(newValue);
            return;
        }
        ListItem!(T) current = head;
        while(current.next !is null) {
            current = current.next;
        }
        ListItem!(T) next = new ListItem!(T)(newValue);
        current.next = next;
    }

    /// index目の要素を返却します
    public T get(int index) {
```

```
        int currentIndex = 0;
        ListItem!(T) current = head;
        while(current !is null) {
            if(currentIndex == index) {
                return current.getValue();
            }
            current = current.next;
            currentIndex++;
        }
        // 見つからなかった
        throw new Exception("要素ないよ");
    }
}

struct Pokemon {
    string name;
    int HP;

    this(string name, int HP) {
        this.name = name;
        this.HP = HP;
    }
}

void main() {
    // 整数型
    List!(int) list = new List!(int)();
    list.add(1);
    list.add(2);
    list.add(3);
    // 2
    writeln(list.get(1));

    // 文字列型
    List!(string) listS = new List!(string)();
    listS.add("114");
    listS.add("514");
    listS.add("1919");
    // 1919
    writeln(listS.get(2));

    // 構造体
    List!(Pokemon) listP = new List!(Pokemon)();
    listP.add(Pokemon("ヒコザル", 10));
    listP.add(Pokemon("ﾎﾟｯﾁｬﾏ……", 20));
    listP.add(Pokemon("萎えとる", 0));
```

```
    // 萎えとる 0
    Pokemon result = listP.get(2);
    writeln(result.name ~ " " ~ to!string(result.HP));
}
```

このように様々な型に対応したリストが実装された.

III. 成果

D言語の基本的な文法と多少のメタプログラミングに関する知識を獲得できた. メタプログラミングはコードの汎用性を高める1つの手法である. しかしながら, メタプログラミングを多用すると可読性が著しく低下する可能性もあり, 諸刃の剣である.

これは余談だが, D言語の文法は使いにくいわけでもなく, 普及していないことに多少の疑問を抱いた. 今後もD言語を使用し, D言語の普及に取り組みたい.

3. 終わりに

文責：西見 元希

　このプロジェクトを通じ，班員は様々な言語においてその言語仕様を深いところまで調査し活用方法を調べたり考えたりするという貴重な機会を得た．あまり現実的な活用方法が見つからない言語も中にはあったが，実用的か否かにかかわらず今回の活動は班員の大きな経験値になったことだろう．また，活動においては進捗の遅れが多々見られたが，他の班よりも個人の責任の度合いが大きい中，追い込み合宿やその後の数日で報告書をこのような形でまとめることができたのは非常に喜ばしいことであると考える．これからのRCCの活動やそれ以外のコーディングにおいて今回の活動内容が活きることがあれば幸いである．