メタプログラミング班
前期活動報告書

西見元希 [*1] 小泉孝弥 [*2] 中尾龍矢 [*3] 松本幸大 [*1] 平井柊太 [*4] 服部瑠斗 [*5] 八木田裕伍 [*3] 吉田享平 [*6]

[*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 画像・音メディアコース 二回生
[*5] 情報理工学部 知能情報コース 二回生
[*6] 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前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．活動は週2回行い，1日目は個人研究のための時間とし，2日目は班員の内一名が成果を報告した．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．

2.メタプログラミング概論

文責：西見元希

　メタプログラミングとは，

> メタプログラミングとは，ロジックを直接コーディングするのではなく，あるパターンをもったロジックを生成する高位ロジックを定義する方法のこと．
> 主に対象言語に埋め込まれたマクロ言語によって行われる．
> *(Wikipedia:メタプログラミングより引用)*

というものであり，本プロジェクトでは

- ソースコードを生成するコードを書くプログラミング手法
- 言語仕様を変更することで機能拡張を行う手法

と定義する．
プログラミング言語の種類によってどちらを行うことができるかは変わってくる．前者を行うものにはC++， Haskell，Rustなどがあり，後者を行うものにはRuby，Pythonなどがある．LispやJavaScriptは両方を行うことのできる言語機能を持つ．メタプログラムの実用例として有名なものにはRuby on Railsやコンパイラなどがある．

DSLとは，Domain Specific Language：ドメイン固有言語の略称であり，特定の問題領域(ドメイン)のために作られた言語のことを指す．対義語としてC++， Javaなどの汎用言語がある．Railsやシェルスクリプトのようにドメインが大きなものから，C++のSTLやCommon Lispのループマクロのように小さいものまで様々なものが存在するが，ホスト言語と呼ばれるそのDSLを実装する言語がDSLを含む場合，メタプログラミングを活用している場合が多い.

3.各研究内容成果

3.1 Lisp

Common Lispにおける競技プログラミング用DSLの構築

文責：西見 元希

I. 背景

競技プログラミングにおいてCommon Lispはややインターフェースにおいて使いにくい部分が存在するので，ユーティリティ群としてのDSLを実装し，コンテストに活かそうとしたものである.

II. 実装項目

実装したユーティリティマクロの代表として次のものがある.

- arrayの参照と破壊的変更を同時に行うvref
  - (vref v 2)のようにするとvの2番目の要素を参照し，(vref v 2 5)のようにするとvの2番目の要素を5に変更する.
  - マクロの性質上変数名が展開後のコードと干渉する可能性があり，最も実装が難しいものであったが，同時に最も利便性の高いものとなった.
- オブジェクトを標準出力し改行するprintln
  - 競技プログラミングにおいて出力は必須の機能であり，便利なインターフェースがほしいところであった.
  - (println a)とするとaの評価結果を出力する.
- 一次arrayのラップとしてのvec
  - Common Lispのarray型は極めて多機能であるが指定すべき項目が多くなるのでマクロを使わない場合コードが冗長になりやすい．ここではマクロで冗長性を回避するというLispの方向性に従って一次配列のインターフェースを実装した.
  - (vec ‘integer 100)を評価するとinteger型で要素数100の配列オブジェクトが作られる．初期値の設定は(vec ‘integer 100 0)のように引数を増やすことで実行できる.
- integer型のvecのラップvint
  - vecでは型を指定する必要がある(ように設定した)ため，最も頻繁に使用するintegerの配列としてvintを用意した.
  - (vint 10 5)のように型の指定を省略することができる.

III. 実装前と実装後のコードの比較

ここでは要素数10の一次配列を宣言し，各要素に整数値を格納してそれを表示するLispプログラムの，DSLを用いた場合と用いずに実行した場合の両方を示す.

```
;;DSLを使用しない場合
(defvar v
  (make-array 10 :element-type 'integer :initial-element   0
:adjustable nil :fill-pointer 10))
(dotimes (i 10)
  (setf (aref v i) i))
(dotimes (i 10)
  (format t "~a~%" (aref v i)))

;;DSLを使用した場合
(defvar v (vint 10 0))
(dotimes (i 10) (vref v i i))
(dotimes (i 10) (println (vref v i)))
```

IV. 成果

上の比較でも分かる通り，素のコードのままだとかなり冗長な式がDSLによって簡潔かつ直感的なものにすることができた.このユーティリティは競技プログラミング以外の分野にも用いることのできる汎用的なものであり，今後の活用も可能であると考えられる.また，このDSLを実装して，数回の競技プログラミングのコンテストにCommon Lispで出場したが，C++で解いた方が解きやすいことが分かった.

3.2 OCaml

文責：松本 幸大

I. 背景

OCaml はシンプルな関数型プログラミング言語である．アドホック多相に相当する言語機能を制限することにより，強力な型推論機構を有している．しかし，その徹底した言語仕様ゆえに，かえって実装の自由度を下げてしまうケースがいくつか存在する．今回は，PPX という技術を用いたこの問題の解決方法について研究した．PPX は，OCaml の各種コンパイラで利用可能なプリプロセッサとしての機能である．この PPX の実体となる実行可能ファイル (以下，PPX 拡張) を OCaml で開発すること，すなわちソースコードを受け取り新たなソースコードを生成してコンパイラに渡す OCaml プログラムを開発することで，自由にソースコードの前処理を行うことができる.

II. 内容

まず既存の PPX 拡張を利用した開発を体験し，その後実際に PPX 拡張を

自作した.

- PPX の基礎知識

  OCaml では，PPX がソースコードを処理するうえで，抽象構文木 (以下，AST) の変形処理の起点とするための「エクステンションポイント」「アトリビュート」という２つの構文が用意されている．アトリビュートはPPXに処理されないまま残った状態でコンパイラに渡されても無視されるのに対し，エクステンションポイントが残った状態でコンパイラが読み込むとコンパイルに失敗する．前者は AST に情報を付与し，後者は前者を含めた AST をもとに必ず展開されなければならない.

- ppx_variants_conv の利用

  ppx_variants_conv (https://github.com/janestreet/ppx_variants_conv) は，OCaml で用いられる主要なデータ構造であるヴァリアントに対して，それを便利に扱うための関数群を提供する．ここではその関数群のなかでも，ヴァリアントの値を文字列に変換する関数を利用する.

  このように，type 構文 (ヴァリアントとその値を定義する構文) を就職する形でアトリビュート [@@deriving variants] を記述している.

  OCaml ソースコードの補完機能を提供する ocamlmerlin を用いて，プリプロセス時に生成される Variants_of_color モジュールの内容を表示させると以下のようになる.

```
6
7  let () =
8    [Green; Red; Blue]
9    |> List.iter (fun color ->
10     print_endline @@ Variants_of_color.to_name color)
11
```

```
sig
  val red : color Variantslib.Variant.t
  val green : color Variantslib.Variant.t
  val blue : color Variantslib.Variant.t
  val fold :
    init:'a ->
    red:('a -> color Variantslib.Variant.t -> 'b) ->
    green:('b -> color Variantslib.Variant.t -> 'c) ->
    blue:('c -> color Variantslib.Variant.t -> 'd) -> 'd
  val iter :
    red:(color Variantslib.Variant.t -> unit) ->
    green:(color Variantslib.Variant.t -> unit) ->
```

to_name 関数も含まれているため，それを呼び出す式を書くだけで文字列への変換が可能になっている．実際にプリプロセス・コンパイル・実行すると以下のようになる．

```
~/github/ocaml/ppx master*
❯ ocamlfind ocamlopt -linkpkg -package ppx_variants_conv variants.ml
~/github/ocaml/ppx master*
❯ ./a.out
Green
Red
Blue
~/github/ocaml/ppx master*
❯
```

通常，ヴァリアントを定義しただけでは値を標準出力に出力することはできないが，この PPX 拡張を導入したことによって，毎回ヴァリアントの値を文字列に変換する関数を定義する必要がなくなった．

- ppxlib (PPX 拡張の開発を支援するライブラリ) を使用した PPX 拡張の自作

Ast_mapper モジュールを使用すると，事前に登録された関数群を用いて入力のソースコード中の各種構文を再帰的に処理できる．そこに自作した関数 (パターンマッチ等でエクステンションポイントを検出して変形後のASTを返す関数) を登録することで，PPX 拡張が実装できる．ここでは，エクステンションポイント “addone” 付きの式 expr を検出して，expr + 1 に置換する関数を登録して PPX 拡張を紹介する．ソースコードは以下のようになる．

```
open Ast_mapper
open Ast_helper
open Asttypes
open Parsetree
open Longident

let rec expr_mapper mapper = function
  | { pexp_desc = Pexp_extension ({ txt = "addone"; loc }, pstr); pexp_loc = _; pexp_attributes = _ } ->
      (match pstr with
        | PStr [{ pstr_desc = Pstr_eval (expression, _); pstr_loc = _ }] ->
            Exp.apply
              (Exp.ident { txt = Lident "+"; loc = !(default_loc) })
              [(Nolabel, expr_mapper mapper expression); (Nolabel, Exp.constant (Pconst_integer ("1", None)))]
        | _ -> raise (Location.Error (Location.error ~loc "SyntaxError")))
  | x -> default_mapper.expr mapper x

let addone_mapper _ = { default_mapper with expr = expr_mapper }

let () = register "addone" addone_mapper
```

III. 成果

自作した PPX 拡張が前処理を担当するエクステンションポイント・アトリビュートを用いて，DSL として柔軟に開発ができるということ，そして PPX 拡張自体の開発では ppxlib の恩恵を受けながら型安全に抽象構文木の読み書き・変形を行うことが可能であることを知った．今後はさらに高等的な PPX 拡張の利用について研究していきたい．

3.3 Ruby

文責：平井 柊太

I. 背景

個人開発にて今まで活用していたRubyには強力な機能の1つとしてメタプログラミングが提供されている．これはRuby製フレームワークRuby on Rails(https://rubyonrails.org)や同じくRuby製の軽量フレームワークSinatra(http://sinatrarb.com *背景の笑顔が印象的である*)，Padrino(http://padrinorb.com)等の内部機能を記述することや，開発者オリジナルのDSLを記述することなどにも用いられる．活動内では メタプログラミング Ruby 第2版 のコードを参照しつつ具体的なメタプログラミングの有する**魔術**について調査した．

II. 内容

Rubyの有するメタプログラミングの機能や可能な記述方法を全て枚挙することは難しいため，3つの機能に絞って調査を行った．

- Dynamic Method

  動的メタプログラミングのうちでも強力な機能であり，実行時に呼び出すメソッドをプログラムを記述する人間が決定することができる．

  主な利点としては，複数のメソッドをある1つのインスタンスやブロックに対して呼び出したい際に，記述するコードの量をかなり減らすことが可能である点が挙げられる．

実際にPry(https://github.com/pry/pry ：Ruby製のデバッガ)のバージョン0.9.12.2では以下のようにDynamic Methodが用いられていることが確認できた.

```
 95    def refresh(options={})
 96      defaults   = {}
 97      attributes = [
 98                     :input, :output, :commands, :print, :quiet,
 99                     :exception_handler, :hooks, :custom_completions,
100                     :prompt, :memory_size, :extra_sticky_locals
101                   ]
102
103      attributes.each do |attribute|
104        defaults[attribute] = Pry.send attribute
105      end
```

このように実際のOSS開発にも用いられることから機能面や実行速度でもかなり実用レベルにあると考察することができる.

- Ghost Method
  　存在しないメソッドを呼び出した際にmethod_missingがRubyでは呼び出されることになっている．それをオーバーライドしたメソッドをGhost Methodと呼ぶ.
  主な利点としては，同じようなメソッドを大量に作ることなく，ある名前空間の中のメソッドに対して，当該メソッドに対応するものが呼び出されなかった際に，ある処理を走らせるといったようなコードの縮小化ができる点が挙げられる.
- Around Alias
  　モンキーパッチのように既存のメソッドをオーバーライドすることで
  機能を変化させるという方法ある．それよりも，より安全にメソッドのオーバーライドをする方法として，存在しているメソッドのエイリアスを作成することで元のメソッドも使用ができる，というようなものがAround Aliasである.
  メタプログラミングRubyからの引用であるが，以下のようなコードがAround Aliasの一例である.

wrapper_around_alias.rb

```ruby
class String
  alias_method :real_length, :length

  def length
    real_length > 5 ? 'long' : 'short'
  end
end

p "Love and Peace".length
p "Love and Peace".real_length
```

このようにreal_lengthという名前で元の文字列長を測定するlengthメソッドのエイリアスを作成し，lengthメソッドにはそれがある値の長さよりも，長いか短いかを判別する機能にオーバーライドしている．主な利点としては，先程も挙げたように安全にメソッドのオーバーライドが行えるという点がある．

III. 成果

正しいメタプログラミングの運用によって，開発の中でも安全に且つ大きな成果を挙げることが可能であることが調査の結果分かった．また，Ruby自体のメタプログラミングとはコードを生成するためのコードを記述する静的メタプログラミングではなく，機能を拡張するためのコードを記述する動的メタプログラミングというスタイルを取るため，闇雲にコードを短縮するようなものではなく開発者が利益を被るようなコードを書くことが可能であるということを知ることが出来た．

IV. 今後の展望

Sorbet(https://sorbet.org)のようにRubyにおける型を付与するというサードパーティ製ライブラリが開発されるなど，Rubyへの型付けに関する開発が行われていることも調査によって分かった．メタプログラミングによってそのような型付けが行われていると予想できるため，どのような実装になっているのかを調査し，自身でも簡単な型チェックのための機構のようなものを作成したい．

3.4 Rust

文責：小泉 孝弥

I. 背景

Rustで競技プログラミングを行う上でネックになるのが入力である．

それを解決する方法としては関数を定義する方法と新しいマクロを定義する方法がある．今回はメタプログラミングをするために後者でのマクロを定義することにより，問題の解決を試みようとしてが，Rustを触るのが初めてだったので，基礎文法の勉強から入った．

II. 内容

まず，出力，条件文，繰り返し文などの基礎部分を学んだ．次にRustの特徴の一つである変数の可変性についてや所有権の概念について学んだ．その次に関数について学び，関数やマクロの定義の仕方について学び，簡単なものの実装をしたり，既存のマクロにはどのようなものがあるのかを学んだ．

III. 成果

結果として自分で新しく入力マクロを実装することはできなかった．しかしながら，既存のマクロ(https://qiita.com/tanakh/items/0ba42c7ca36cd29d0ac8)を用いてAtCoderのA, B問題はスラスラ実装ができるようになってきておりRustの基礎文法はしっかりと定着することができた．

3.5 C++

3.5.1

文責：中尾龍矢

I. 背景

メタプログラミングという単語を初めて聞いて，興味を持ったので勉強することにした．

II. 内容

勉強した内容は以下の箇条書きにある通りである．

- C＋＋言語のメタプログラミングの基礎
    - テンプレート関数

```
//関数の宣言
int Add_int(int a) {
 return a + 1;
}
float Add_float(float a) {
 return a + 1.0;
}
//関数の呼び出し
int a = Add_int(1);
int b = Add_float((float)1.2);

//出力結果
 a = 2
 b = 2.2000000
```

このような単に型の違いだけで二つも関数を作らなければいけない場合，templateを用いるだけで二つの関数を一つにまとめることができる．

```
//関数の宣言
template<typename T> //テンプレート引数Tを生成
  T Add(T a) {         //Tを用いて引数aとTの型の変数を返す関数Addを宣言
   return a + 1;
  }

//関数の呼び出し
int a = Add(1);
int b = Add((float)1.2);

//出力結果
 a = 2
 b = 2.2000000
```

このようにint型の変数でもfloat型の変数でも，それぞれに同じ処理を行う事が出来る．
複数の任意の型の変数を宣言する場合は

```
template<typename T>
  ... Func(T a, T b) {
   ...
   return ...;
  }
```

ただこの場合，a，bは二つともTを使って宣言されているので，同じ型の変数扱いとなる．
異なる任意の型の変数を宣言する場合，

```
template<typename T, typename U>
 ... Func(T a, U b) {
   ...;
   return ...;
 }
```

というふうに置く．

- クラステンプレート
  関数と同様にクラスにのtemplateを使用できる．

```
//クラスの宣言
      template<typename T>
      class Myclass {
public:
  T Value;
  Myclass(T num) {          //コンストラクタ
         Value = num;
         cout << Value << endl;
        }
};
//インスタンスの生成
      Myclass<int> myclass;

//出力結果
1
```

- テンプレート関数の特殊化
  テンプレートを使用において，<>の中に使用する型を記述する際，特定の型を入力したときのみ内容を変えること（特殊化）が出来る.

```
//関数の宣言
template<typename T>
 T Add(T a, T b) {
  return a + b;
 }
//特殊化
 template<>  //特殊化を行う時は<>の中身は空欄
 float Add (float a, float b) { //この時の変数の型は明示する
  return a - b;
 }

//関数の呼び出し
Add(3, 1);//特殊化されてない関数が使用される
Add(3.2, 0.2)//特殊化された関数が使用される
//出力結果
4
3.000000000
```

- クラステンプレートの特殊化

```
//クラスの宣言
template<typename T>
```

```
 class Myclass {
 public:
  T Value;
  Myclass(T num) {          //コンストラクタ
   Value = num;
   cout << Value << endl;
  }
 };
//クラスの特殊化
 template<>
 class Myclass<int> {
  int Value;
  Myclass(int num) {        //コンストラクタ
   Value = -num;
   cout << Value << endl;
  }
 };

//インスタンスの生成
Myclass<float> myclass((float)0.1);
Myclass<int> myclass(1);
//出力結果
 0.1
 -1
```

- 任意の型の可変数の引数を持つ関数の作成

```
//関数の宣言
template<typename ...A>
 void CreateInt(A...args) {
  const int size = sizeof ...(A);
  int Array1[size] = { args... }; //args...で展開
  for (int i = 0; i < size; i++) {
   printf("Array[%d] = %d\n", i, Array1[i]);
  }
  return;
 }

 //関数の呼び出し
  CreateInt(1, 2, 3, 5, 6, 7, 8, 9);

//出力結果
```

```
Array[0] = 1
Array[1] = 2
Array[2] = 3
Array[3] = 4
Array[4] = 5
Array[5] = 6
Array[6] = 7
Array[7] = 8
Array[8] = 9
```

このようにA ...argsに１から９までの整数が渡される.

- 任意の型の変数の型の判定をするプログラム

```
//クラスの宣言
class Inspection {
public:
  template<class First, class Check>  //テンプレート関数
    int inspect(First first, Check check) {
     cout << "不一致" << endl;
     return-1;
  }
  template<class Scan>      //特殊化
  int inspect(Scan scan1, Scan check) {
    cout << "一致" << endl;
    return 1;
  }
};
//メイン関数
int a = 0;
float b = 0.0;
char c = 'c';
int num;

num = Inspection::inspect(a, a);
cout << num << endl;
num = Inspection::inspect(b, a);
cout << num << endl;
num = Inspection::inspect(c, c);
cout << num << endl;

//出力結果
一致
```

```
1
不一致
-1
一致
1
```

sclapboxに（未完成故に）投稿していない学習内容としては,

- tipleを用いた複数の任意の型の変数の制御
- 任意の型の変数の筒底をするプログラム

以上を勉強した.

III. 成果

今までに知らなかった知識が入ったので，かなり新鮮な気持ちで勉強が出来た.
そして，今私が制作中のプログラムの一部に採用し，さらなる理解の獲得，効率化していこうと思っている.

3.5.2

文責：服部瑠斗

I. 背景

別サークル(RoboCup Soccer 2D sim)での開発の効率を上げたいと思いメタプログラミングを勉強するための前提知識であるテンプレートについて学習した. またテンプレートを用いてどのように開発の効率を上げるかという実践にも取り組んだ.

II. 実装内容

テンプレートの基礎知識については前項で述べられているので省略する.この項では　主に実践について触れていこうと思う.今回私が実装したものは，2つの引数が持つ座標の距離を返す関数である.(Vector2D型などの独自のクラス・関数群はこのリンクにて配布されている: http://rctools.osdn.jp/pukiwiki/index.php?librcsc).

III. 結果

```cpp
template<typename T, typename U>
double getDist( T * t, U * u)
{
    // u->pos()で座標にアクセスすることが出来る
    // 座標->dist( 別の座標 )で2つの座標の距離を求めることが出来る
    return u->pos()->dist( t->pos() );
}

// Vector2Dクラス == 座標なので、
// Vector2Dクラスは座標にアクセスする必要が無いため
// 別関数を実装してオーバーロードを行うことによって処理内容を切り替える
double getDist( const Vector2D * t, const Vector2D * u )
{
    return u->dist( (*t) );
}
```

今回実装した関数を呼び出すときに想定している引数のクラスではほぼ全て上記のような記述方法で座標にアクセスすることが出来る.しかし，Vector2Dクラスのみは座標へのアクセス方法が違う.そこで，オーバーロードという仕組みを用いることによって引数の型がVector2D型かそれ以外の型で処理内容を分けている.しかしこのコードはコンパイルエラーが発生してしまう.解決を試みたが知識不足により断念した.

IV. 今後の展望

今回実装した内容は，元々テンプレートの特殊化を用いて実装予定だったが知識不足により断念してしまった.また片方の引数がVector2D型だった場合の処理もテンプレートの部分特殊化を用いて実装しようとしたが同じく知識不足により断念した.なので後期ではテンプレートの部分特殊化や，元々の目的であるメタプログラミングの知識の勉強を行っていきたい.

3.6 JavaScript

3.6.1 動的なコード生成ソフトウェアs2sのプラグインの作成

文責：八木田 裕伍

I. 背景

JavaScriptでWebアプリを作るときのデータ管理ライブラリであるReduxを使用する場合のデメリットとして，実装しようとしている機能に対して記述量が多くなり，単純にタイプ数が増える上，保守性も下がることもあるというものがある．s2sは開発者が書いたソースコードを元に動的にソースコードを生成することで，開発者の負担を削減するソフトウェアである．今回はその仕組みの研究を行った.

II. 内容

Babel Handbook(https://github.com/jamiebuilds/babel-handbook)を読んでBabelの基本的な概念を学習し，Babelのコード(https://github.com/babel/babel)や型定義

(https://github.com/DefinitelyTyped/DefinitelyTyped)，s2sのコード(https://github.com/akameco/s2s)や読みながらs2sの仕組みを学習すると共に簡単なプラグインを作成した.

III. 成果

当初の目的であったTypeScriptでRe-ducksを書くためのs2sプラグインは完成できなかったが，Babel及びs2sの仕組みは理解でき，既存のプラグインのソースコードを読んで意味を理解できるようになった．途中まで作成しているので後期ではその完成とBabelを使用したさらに発展的なプラグインを作成したい.

4.今後の展望

文責：西見 元希

このプロジェクトは通年プロジェクトであるため，前期活動で得られた言語仕様に関する知見を活かして更に発展的な技術を学習したりDSLを実装したりしていく．また，一定の成果が既に得られた者に関しては研究する言語を別のものに変えていく.

文責：小泉 孝弥

残念ながら，前期のプロジェクトは言語使用の勉強に時間を取られてしまい，成果物をあげることはできなかったが，後期では前期で学んだ知識を活かして新しいマクロを自分で定義していきたい．また，Rustにはまだまだライフタイムなどの難解な概念も残されているため，引き続き言語使用の勉強も進めていきたい.