メタプログラミング

村野 翔太 冨田 龍太郎
Shota MURANO，Ryutaro TOMITA

1　はじめに

近年，変化に強く，柔軟に変化できるプログラムの構成に，注目が集まっている．プログラムの構成にはいくつか原則が存在しているが，重要な原則の一つとして，DRY（Don't Repeat Yourself）原則が存在する．DRY 原則は，同じ機能をもつ情報やデータを複数の場所に重複して記述するのを避けるべきという原則である．1 度記述したソースコードを再利用できるようプログラムの構成を行うことで，DRY 原則を満たす．DRY 原則を満たす方法には，オブジェクト指向プログラミングやジェネリックプログラミングがある．オブジェクト指向でクラスを用いると重複を避けることが可能だが，常に重複を避けられるわけではない．コードに合わせた幾つかの技法を組み合わせて解決することが必要である．幾つかの技法を組み合わせ，ソースコードの重複を避ける方法の 1 つにメタプログラミングが存在する．本稿では，メタプログラミングについて詳しく述べる．

2　メタプログラミング

2.1　概要

メタプログラムはコード群を生成するプログラムである．例えば，コンパイラは高水準言語のプログラムを操作し，アセンブリ言語やマシン語のプログラムを生成しているため，メタプログラムに該当する．コンパイラの高水準言語を変換する処理は以下の 4 つの段階に分類される．

- 字句解析：文字列を字句列に変換
- 構文解析：字句列を抽象構文木に変換
- 意味解析：抽象構文木を中間コードに変換
- 最適化　：中間コードを命令列に変換

メタプログラミング [1] は，動的プログラミングにおける実行時，静的プログラミング言語におけるコンパイル時の字句解析・構文解析・意味解析の段階に，ある機能をもつコード群を生成するコードのプログラミング技法である．動的プログラミング言語は，コンパイルを行わず，直接プログラムを実行する言語である．静的プログラミング言語は，実行を行う前にコンパイルが必要な言語である．メタプログラミングを行うためには，使用する言語に備わっている言語機能が必要である．動的プログラミング言語は，文字列を式として評価する eval 系の機能がある．静的プログラミング言語は，コンパイル時，どの段階で作用するかによって言語機能の分類が可能である．字句列で変換を行う C プリプロセッサマクロ，構文解析で変換を行う Lisp マクロ，意味解析で変換を行う C++ テンプレート，D テンプレートなどがある．作用する段階が，機械語に近いほど表現力が高くなる．メタプログラミングは，プログラムの実行時・コンパイル時に，クラス・モジュール・オブジェクト・メソッドを記述または操作することを可能とする．メタプログラミングの構造を Fig. 1 に示す．

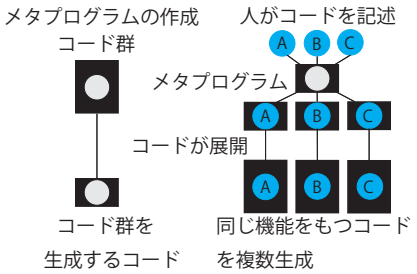

Fig.1　メタプログラミングの構造

ある機能をもつコード群を生成するコードのプログラミングを行い，メタプログラムを作成する．プログラム内でメタプログラムを呼び出すコードを使用すると，実行時・コンパイル時に，呼び出したコードにコード群が展開される．メタプログラムを呼び出すコードをプログラム内で使用することで，コード群の機能を得る．

2.2　マクロ

マクロ [2] は，C 言語，C++，Lisp などで使用が可能である．. マクロとは，既定のソースコードを置き換えるルールやパターンを生成する技術である．コンパイル時，ソースコードのデータ型にマクロを発見すると，マクロで定義されたソースコードを展開する．C++ マクロを例に述べる．C++ マクロを使用すると，文字列操作によってソースコードを生成することが可能である．マクロはプログラム内から呼び出すことができるため，関数と同じと，とらわれる傾向にある．しかし，マクロの動作は関数と異なる．関数は，レジスタの退避・回復・スタック調節を行いデータの値を送信している．マクロは，呼び出した場所にコードを生成するインライン展開を行う．リスト 1 に，C++ マクロの例を示す．

リスト 1　C++ マクロ

```
1. #include <stdio.h>
2. #define Add(x,y) (x+y)

3. int main(void){
4. printf("%d\n",Add(2,3));
5. printf("%f\n",Add(2.2,3.3));

6. return 0;
7. }
```

リスト 1 のマクロ式では，プログラム内のコードに Add() というソースコードが存在した場合，「x+y」の式が展開される．Add(x,y) の展開イメージを Fig. 2 に示す．

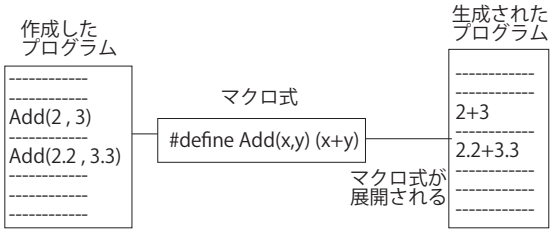

Fig.2　Add(x,y) の展開

C++ マクロは，文字列操作によるソースコードの展開であるので，型推論が不可能であり，型安全性もない．例えば，「1.1+Add(2.2,3.3)」という計算は不可能である．

2.3　テンプレートメタプログラミング

テンプレートメタプログラミング [3] は，C++，Eiffel，ML などで使用が可能である．テンプテートメタプログラミングでは，コンパイラがテンプレートを使い，ソースコードを生成する．それを自ら生成したソースコードと結合しコンパイルする技術である．テンプレートには，関数テンプレートとクラステンプレートがある．関数テンプレートはデータ型に応じて関数を生成する．テンプレートクラスはクラス内に，データ型に応じてメソッドを生成する．リスト 2 に，C++ の関数テンプレートの例を示す．

リスト 2　C++ の関数テンプレート

```
1. #include<stdio.h>

2. template <typename T>
3. T add(T x,T y){
4.  return x + y;
5. }
6. int main(void){
7.  printf("%d\n",add(2,3));
8.  printf("%f\n",add(2.2,3.3));
9.  return 0;
10. }
```

7・8 行目で，異なるデータ型でテンプレート関数呼び出しを行っている．add という足し算を行うテンプレート関数を呼び出した結果，コンパイル時，データ型に応じた関数が生成される．C++ テンプレートメタプログラミングでは，型の定義まで行われるため，「1.1+add(2.2,3.3)」の計算結果を得ることが可能である．コンパイル時，テンプレート関数を各データ型専用の関数に変換する工程をインスタンシエートと呼ぶ．今回のインスタンシエートを Fig. 3 に示す．

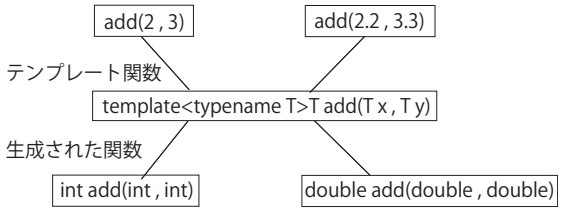

Fig.3　インスタンシエート

3　メタプログラミングのメリット・デメリット

3.1　メリット

コードに特定のコードを記述させることにより，DRY 原則を満たす．それにより，プログラム内のコード量が削減され，可読性も向上するため，プログラマーへの負担が軽減される．結果，開発効率の向上へとつながる．プログラム内の呼び出したコードにコード群が展開されるため，全てのデータ型に対応することが可能である．関数やクラスの引数の型の違いなどによるエラーが解消される．アルゴリズムなど全てのデータ型に対し同じ処理を行う場合に有効である．

3.2　デメリット

メタプログラムを呼び出す度，コード群が生成されるため，メモリ容量が増加する．動的にコードを生成するため，静的言語において，コンパイル時の時間は増加する．動的言語においては，実行速度が低下する．メタプログラミングを利用しすぎると読み解くことが困難なソースコードとなってしまう．複雑なソースコードは，デバックが困難となるため，開発効率が低下する．メタプログラミングは，使用するべき場所を判断することが重要となる．

4　今後の展望

例えば,Ruby の言語には，Lisp から受け継がれたメタプログラミングの機能が多く備わっている．それにより，コードもシンプルとなっている．さらに，自らメタプログラムを作成するための機能も備わっている．これにより，プログラマが特定の機能をもった新たなメタプログラムを作成し，言語の機能がそれぞれに拡張されていくようになる．近年，企業の大規模開発が発展している．大規模開発では，ソースコードの共有を行うため，変化に強く，柔軟に変化できるメタプログラミングは多くの場面で活用される．

参考文献

1) あかさた，平凡なエンジニアの独り言，http://akasata.com/articles/218
2) 野田　開，On　Lisp—マクロ，http://www.asahi-net.or.jp/ kc7k-nd/onlispjhtml/macros.html
3) デビッド・エブラハムズ.，アレクセイ・グルトヴァイ.，C++ テンプレートメタプログラミング，2010 年 3 月 11 日