CCC——データの内部表現に依存しないオブジェクト指向

原田康徳[†] 山崎憲一[††]

C，C++ でリストを表現する際に，NULL でリストの終端を表現することが多い．しかし NULL は C++ のオブジェクトの構造をしていないので，クラスとして定義することはできない．そのため C++ では，if (pointer) pointer->message(); のように，if 文でチェックしてから呼び出さなければならない．C++ が提供するオブジェクトの構造に従わないデータをクラスとして定義することはできない．CCC はクラス階層を条件式を記述することで定義する．クラスに定義されたメソッドはそれが呼び出される前に，クラスに特徴付けている条件式を試され，それが真のときのみ呼び出される．クラス階層はそのまま条件式の階層となり，親クラスから順に条件式を調べられ，自分が真で真の子クラスがないクラスのメソッドが実行される．以上により，NULL のクラス化などが実現できる．CCC は C または C++ へのプリプロセッサとして実装され，条件式によるクラス定義のほかに，クラス内に閉じたマクロ，後からプログラムの断片をソース上の特定の位置に挿入する機構，クラスとは独立なモジュール構造などの拡張がなされている．これらの設計ポリシーは，単純な仕組みで，生成されるコードをプログラマが容易に想像できるようにした点である．現在 CCC ではある記号処理言語の処理系を記述している．CCC の特徴をそれらの経験もふまえて述べる．

CCC——An Object Oriented C Extention with Conditonally Defined Classes

Yasunori Harada[†] and Kenichi Yamazaki[††]

We sometimes use a NULL pointer to indicate a list structure termination in C or C++ programming. Because NULL is not an object of a C++, we can not define a NULL class. So in C++ programming, an if statement, if (pointer) pointer->message(...); is used for a NULL related method call. This problem is because of C++ and other object oriented languages use their own object structure for object implementation. When a programmer wants to treat a special data structure (ex. NULL, data packet, system call ...) as an object, such kind of problem occurrs. We propose a new language, CCC, that extends C and C++ to overcome the above problem. In CCC, a class is define by a condition, and a class hierarchy is a condition hierarchy. When a message is sent the system checks conditions that define classes, finds a class whose condition is satisfied and has no subclass whose condition is satisfied, and executes the method that is associated with the class. Using such a mechanism, we can treat a NULL pointer as a NULL class. CCC compiler is implemented as a C and C++ preprocesser. Its other features are class-local macro definition/extraction, a source modifying mechanism for inserting blocks, and class independent modules. This paper also describes an experience using CCC to develop a symbol manipulating language whose pointers include tag information.

1. はじめに

プログラミング言語は様々な研究や実用化により高機能化されているが，その結果，動作の直感がつかみにくくなってきている．様々なことを自動的にやってくれる言語が重要である一方で，必要なことを単純な機構の組合せで実現する，透明性を持つプログラミング言語も重要と考える．

たとえば，C++ の STL [1)]は Template や inline などの機能を巧みに使い，データ構造とアルゴリズムを分離した効率の良いライブラリである．その高度さの反面，処理系がどのようなコードを生成するのか，プログラマにとって分かりにくくもなっている．元々の C が持つ動作の直感を維持しつつ，オブジェクト指向プログラミングなどの高度なプログラミングを提供する，というのが研究の動機である．

オブジェクト指向言語を用いてプログラミングをする際に，1) オブジェクトのインタフェースを定義し，

† さきがけ 21，科学技術振興事業団
Presto JST
†† NTT 未来ねっと研究所
NTT Network Innovation Laboratories

次に，2) オブジェクトの内部構造（実装）を定義する，というのが一般的な順序である．しかし，この順序が逆になるようなプログラミングが存在する．たとえば，ファイル構造，プロトコルのパケット，システムコールやライブラリに与えるデータ構造といった，外部から与えられた構造を操作したり，記号処理言語の実装のように，構造のデザインが別の要因で決まっている場合などのプログラミングである．その場合データ構造が最初に決められ，それに合わせる形で手続きを定義していく．

外部から与えられた構造であっても，クラス階層は存在している．たとえば，X-window のイベント構造体はいくつかの種類のイベントの union として定義されているが，それらはクラス階層で表現するのに適している．そのような構造に対して，データ抽象はできないけれど，多相や継承を利用したプログラミングは行いたい．

より分かりやすい例としては，C や C++ のプログラミングにおいて，リスト構造などの終端を表現するのに NULL（値が 0 のポインタ）が伝統的に用いられているが，

```
if (pointer) pointer->message();
```

のような，NULL のチェックがいたるところで必要である．もし，NULL をそのままクラスとして扱うことができれば，このようなチェックを隠蔽することができる．このように，C++ はオブジェクトのデータ構造やクラス判定方法などが固定されているので，それからずれた構造に対するプログラミングは，オブジェクト指向プログラミングの機能を活かすことができない．

本論文は，新しいオブジェクト指向言語 CCC を提案する[6]．CCC 処理系は複数の CCC プログラムコードを入力し，C または C++ の 1 つのプログラムコードを出力するプリプロセッサである．入力されたプログラムから出力されるプログラムが容易に想像できるために，プログラマは限定した範囲で納得して CCC を使用することができる．CCC の個々の機能は単純ではあるけれども，それらを組み合わせて大きな効果が得られる．

CCC の最大の特徴は，オブジェクトの構造，クラス判定法をユーザが自由にプログラミングできる点である．2 章では，この仕組みについて述べる．3 章では，もう 1 つの特徴である，分散されたテキストを 1 つにまとめる仕組みである，挿入構文について述べる．さらに 4 章で実装，5 章で CCC による記述例を述べる．また，より詳しい CCC のプログラミング例を付録で述べる．

2. 条件クラス

条件クラスは条件式を用いてクラス階層を定義する仕組みで，データからクラス構造を抽出し同時にメソッドディスパッチを行う．

CCC のクラス定義は，

```
@class クラス名（クラス引数）if（クラス条件式）{ クラス本体 }
```

のように行う．クラス引数は，型と変数名のペアのリスト，クラス条件式は C の式である．クラス本体はサブクラスとメソッドの定義を行う．クラス引数で宣言された変数は，クラス条件式やクラス本体の中（すなわちメソッドとサブクラス）からアクセスすることができる．

例により説明する．

```
@class integer (int x) if (1) {
  int abs() { return x; }
  @class negint if (x < 0) {
    int abs() { return -x; }
  }
}
void foo() {
  int y = abs(-10);
}
```

この例では，2 つのクラス `integer` と `negint` が定義されており，それぞれに `abs` というメソッドが定義されている．`integer` のクラス条件式は `(1)` でありつねに成立する．`negint` にクラス条件式は `(x < 0)` であり，クラス引数 `x` が負のときだけ成立する．

メソッド呼び出しは，`abs(-10)` のように行い，クラス引数とメソッド固有の引数（この例では空）をつなげて呼び出す．ここでは，`-10` がクラス引数 `x` へ与えられ，それを用いて条件式が評価され，`negint` のメソッド `return -x;` が呼び出される．

クラス宣言で，クラス引数，クラス条件式は省略することができ，条件式がない場合はつねに成立 `(1)` の意味になる．また，クラス名を省略する場合は `_` を用いる．

2.1 ディスパッチ

CCC 処理系は上述のプログラムを入力すると，次のような C プログラムを生成する．

```
int integer_abs(int x) { return x; }
int negint_abs(int x) { return -x; }
int abs(int x) {
  if (1) {
```

```
    if (x < 0) {
      return negint_abs(x);
    }
    return integer_abs(x);
  }
}
```

このように，ルートのクラスから順に，自分が真で，真となるサブクラスがないようなクラスを探す．あるクラスにおけるサブクラス間には全順序が定義され，探索はその順序に従った深さ優先で行われる．そのようなクラスが見つかると，そのメソッドが呼び出される．

一般のオブジェクト指向言語のメソッドディスパッチでは，最初にそのオブジェクトのクラスが分かり，メソッドが見つかるまでルートクラスに向かって探していく．CCC と探索の向きが逆である点に注意してほしい．

各クラスのメソッドは "クラス名_メソッド名" という名前で定義されている．特定のクラスのメソッドを直接呼び出す場合は，この名前を使う．

メソッドディスパッチはメソッド名ごとに汎関数として生成される．このため，メソッド名ごとに異なったクラス階層を作る．

たとえば，

```
@class A (int x) {
  @class B if (x > 0) {
    int f(int y) { return y; }
  }
  @class C if (x < 0) {
    int g() { return 0; }
  }
  int f(int y) { return -y; }
  int g() { return 1; }
}
```

のようなプログラムは，

```
int f(int x, int y) {
  if (x > 0) {
    return B_f(x, y);
  }
  return A_f(x, y);
}
int g(int x) {
  if (x < 0) {
    return C_g(x);
  }
  return A_g(x);
}
```

のように展開される．ここで，クラス B の g() やクラス C の f() の定義がないため，それぞれの条件式を調べないコードが生成されている．

2.2 switch-case による条件記述

if-then-else スタイルだけではなく，switch-case スタイルの条件式も記述できるようになっている．

```
@class base (char com) switch(com) {
  @class A case ('A') {
     void f() { .. }
  }
  @class B case ('B') {
     void f() { .. }
  }
  void f() { .. }
}
```

これは，

```
void f(char com) {
  switch (com) {
    case 'A': A_f(); break;
    case 'B': B_f(); break;
    default: base_f();
  }
}
```

のように展開される．これにより，効率の良いメソッドディスパッチが可能になる．

2.3 クラスローカルなマクロ

オブジェクト指向言語には，インスタンス変数を定義するという機能が必須である．CCC ではそれに相当する機能として，クラスローカルなマクロ定義が用意されている．

C++ のインスタンス変数は，構造体のメンバとして実装されている．しかし，CCC が対象とするオブジェクトは構造体として表現できるようなものだけではない．

整数の配列に，図形データが格納されているような例を考えよう．たとえば，'L' の文字は直線を表し，それに続く 4 つの整数で始点と終点を表現する．また，'C' は色属性を表し，それに続く 3 つの整数で RGB 値を指定し，以降の図形描画に影響を与える（**図 1**）．このようなデータ構造が定義されていたとき，直線や色属性などをクラスとして定義する．

```
@class figure (int *array, int pos) {
  macro type {array[pos]}
  macro body(x) {array[pos+(x)+1]}
  @class line if (type == 'L') {
    macro x1 {body(0)}
```

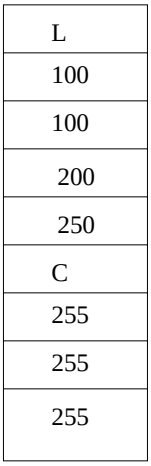

図 1 図形データ
Fig. 1 Figure data.

```
    macro y1 {body(1)}
    macro x2 {body(2)}
    macro y2 {body(3)}
    macro next {pos+5}
    int nextFig() { return next; }
  }
  @class color if (type == 'C') {
    macro r {body(0)}
    macro g {body(1)}
    macro b {body(2)}
    macro next {pos+4}
    int nextFig() { return next; }
    void darken() {
      r = r/2; g = g/2; b = b/2;
    }
  }
}
```

macro から始まる行がマクロ定義である．この例では，整数への配列へのポインタとそのインデックスによって，オブジェクトを表現している．type や x1，r などのマクロはそれぞれのメソッドから，あたかもインスタンス変数であるかのように使うことができる．たとえば，メソッド darken() では色の成分（r，g，b）を参照し，更新している．また，body(x) のように引数を与えることもできる．

マクロ呼出しが，いつでも代入の左辺に使えるわけではない（たとえば next など）．しかし，マクロ展開の透明性により，それが可能かどうかは，容易に判断することができる．

マクロは外部から多相的には使えないので，next Fig() のようにメソッドの皮をかぶせることで，多相的に呼び出せるようにする．

2.4 マクロのクラス外からの呼出し

クラスローカルなマクロをクラス階層の外から呼び出したいことがある．インスタンス変数を直接触らせることは，オブジェクト指向のマナーに違反することではあるが，その危険性（オブジェクトの実装の変化に弱いなど）を熟知していれば問題はない．まして，CCC ではオブジェクトの実装を自由に触らせるプログラミングを対象にしているので，この問題にはあてはまらない．

クラスローカルなマクロを外から呼び出すには，

@@ クラス名（クラス引数）. マクロ名

のようにする．マクロ名には引数が付くこともある．たとえば，

```
int makeLine(int *array, int pos,
    int X1, int Y1, int X2, int Y2) {
  @@figure(array, pos).type = 'L';
  @@line(array, pos).x1 = X1;
  @@line(array, pos).y1 = Y1;
  @@line(array, pos).x2 = X2;
  @@line(array, pos).y2 = Y2;
  return @@line(array, pos).next;
}
```

のようにして用いる．ここでは，クラス line のオブジェクトを生成している．最初の @@figure ... はクラスのタグを代入しているが，これが実行される以前はまだ line 型のオブジェクトではないため，このような初期化はクラスの外で行わなければならない．

マクロに与えられたクラス引数でクラス条件が成立するしないにかかわらず，マクロは展開される．そのため，クラス条件が成立していなければ，予期しない結果を生じる．

2.5 クラス条件式の外部からの参照

前節の問題を安全に行うために，クラス条件式を外部から呼び出せる必要がある．

それぞれのクラス固有のクラス条件式はマクロが展開された形で，

```
#define _line_condition(array, pos) \
   (array[pos] == 'L')
#define _vline_condition(array, pos) \
   (array[pos+1] == array[pos+2])
```

のように C のマクロとして定義されている．ここでクラス vline は line のサブクラスで if (x1 == x2) とクラス条件式が定義されているとしよう．この例で分かるように，条件式 _vline_condition はすでに

親クラス _line_condition が成立しているときだけ意味を持つ．

親クラスの条件までも含めた完全な条件式は，

```
int vline_condition(int *array,
                    int pos) {
  return _line_condition(array, pos)
      && _vline_condition(array, pos);
}
```

という関数として定義されている．この関数は，前提条件なしに呼び出すことができ，正しくクラスの判定を行う．

2.6 柔軟な条件式

条件式は C の一般の式であるから，クラス引数のほかに大域変数の参照や関数呼出しなども行える．これにより，きわめて柔軟なメソッドディスパッチが可能である．

大域変数を参照することで，実行時のモードをクラスにすることができる．たとえば，モバイル環境でバッテリで駆動している場合には呼び出されるメソッドが異なる，という使い方ができる．

```
@class base {
  void f1() { .. }
  @class _ if (gBattery == LOW) {
    void f1() { .. }
  }
}
```

この例では，プログラマはクラス階層のことを気にせずに，一般の関数として f1() を呼び出すことができる．しかし，大域的な条件が成立することで，緊急用の f1() にディスパッチされる．欠点としては，通常の f1() の呼出しでも，つねに大域的な条件を調べるオーバヘッドがかかってしまうという点である．

条件式で副作用のある関数を呼び出すことで，ディスパッチの計算を節約できる．たとえば，各クラスの条件式の評価の順番が決まっているため，

```
@class a (int x) if (cond_a(x)) {
  @class b if (cond_b(x)) {
    @class c if (cond_c(x)) {
 } } }
int global_state;
int cond_a(int x) {
  global_state = aProc(x);
  return global_state > 0;
}
int cond_b(int x) {
  return global_state > 10;
}
int cond_c(int x) {
  return global_state > 20;
}
```

のようにして，関数間で大域変数を通じて情報を受け渡すことが可能になり，クラス条件式間での計算を再利用できる．

同様に，条件式の計算の一部分をキャッシュに入れるなどして，呼出しごとの再計算を減らすことが可能となる．

3. 挿 入 構 文

3.1 サブクラスメソッドの挿入

これまでの例は，サブクラスやメソッドの定義をクラスの本体（内側）で行っていた．しかし，これでは後からサブクラスやメソッドを追加する際に不便である．

そこで，CCC では挿入構文を導入した．挿入構文は，後から自由にテキストブロックを目的の位置に挿入できるものである．

```
@+line {
  @class vline if (x1 == x2) { }
  @class hline if (y1 == y2) { }
}
```

これは，2 つのクラスをクラス line のサブクラスとして宣言している．それぞれのクラスの本体は空であるが，後から，

```
@+vline {
  void move(int dx, int dy) {..}
}
```

のようにして，メソッドなどを挿入できる．

挿入構文により，プログラムの局所性を向上させることができる．

たとえば，最初にクラス階層のスケルトンだけを

```
@class A (char *txt) {
  @class B if (txt[0] == 'B') {
    @class C if (txt[1] == 'C') { }
    @class D if (txt[1] == 'D') { }
  }
  @class E if (txt[0] == 'E') { }
}
```

のように定義する．これは，たとえばクラス引数の txt が B から始まる文字列で 2 文字目が C のとき，クラス C のように判定される．さらに，その後で，

```
@+C { int m1() { .. } }
@+E { int m1() { .. } }
```

のように，メソッドを追加していく．

複数のファイルにまたがって，定義を挿入することができるので，1 つのクラスが 1 つのファイルに対応している必要はない．同じ名前のメソッドを 1 つのファイルにまとめて宣言する，クラス階層のスケルトンだけを 1 カ所に集めるなどといった，プログラマが管理しやすいような構成にできる．

ファイルにまたがった挿入構文の実装のために，CCC は与えられたすべての CCC ファイルを走査して，それからクラス階層やメソッドの構造を作り出す．そのため，あるファイルを CCC のコマンド呼出しの引数に含めたり，含めなかったりすることで，生成されるクラス階層やメソッドなどをコントロールすることができる．たとえば，

```
@+D { int m1() { .. } }
```

を別のファイルに記述していたとする．このファイルを CCC の引数に含めると，生成される `m1` のメソッドディスパッチはクラス `D` の条件を調べる．しかし，このファイルを CCC の引数に含めなければ，生成される `m1` のメソッドディスパッチはクラス `D` の条件を調べない．

3.2 任意のテキストブロックの挿入

挿入構文は任意のテキストブロックに対しても拡張されている．

まず，

```
@position initialize
@position finalize
```

によって，テキスト型の変数を宣言する．次に，

```
@+initialize {
  /* initialize code segment */
}
```

のような記述で，これらの変数にプログラムの断片を追加し，

```
main() {
  @ref initialize
   /* ... */
  @rref finalize
}
```

によって，すべての断片を連結して挿入する．ここで，`@ref` は追加した順序で，`@rref` は追加した逆の順序で，断片を挿入する．この例は，各ファイルで局所的な初期化，終了化ルーチンを記述し，それをまとめて呼び出している．

1 つのテキスト型変数を，複数の箇所で引用することができる．それらはつねに同じものを挿入する．しかし，挿入するコンテキストが異なっているので，その解釈も異なる．たとえば，各ファイルで，コマンド名，その本体，コマンドのヘルプからなるデータを定義する．

```
@+command {
  cmd("cmd1", "help for cmd1",
      x = cmd1())
}
@+command {
  cmd("cmd2", "help for cmd2",
      x = cmd2())
}
```

それらを，コマンドインタプリタの本体から次のように引用する．

```
@class _ (char *cmdName) {
  macro cmd(name, help, body) {
    if (!strcmp(name, cmdName)) {
      { body; }
      return 1;
    }
  }
  int execCommand() {
    @ref command
    return 0;
  }
}
```

これによって，各ファイルで定義されたコマンドを 1 カ所に集めて，`strcmp` で調べて呼び出すことができる．さらに，

```
@class _ () {
  macro cmd(name, help, body) {
    printf("%s: %s\n", name, help);
  }
  void printHelpMessage() {
    @ref command
  }
}
```

とすることで，各ファイルで定義されたコマンドのヘルプメッセージを 1 カ所に集めて，印字することができる．

このようなテキストの追加，参照はきわめて高機能であるが，アドホックでもある．テキスト型変数の宣言時に引数（マクロ名 `cmd`）も宣言し，その参照時にそのマクロの定義を与えるようにすればアドホックさは解消される．

4. 実　　装

CCC 処理系は複数の CCC ファイルを入力し，C

または C++ のソースファイルを 1 つにまとめて出力する．C のプリプロセッサ CPP と同様に，テキストブロックの中身をほとんど見ないで処理をすすめるので，そこに C か C++ のプログラムを記述することで，どの言語のプログラムを出力するかが決められる．

複数の CCC ファイルは，CCC 処理系によって 1 つのソースファイルにまとめられる．このことは，CCC のファイルと C とでは，ファイル内のスコープなどの意味が異なる点に注意しなければならない．すなわち，ヘッダファイルの読み込みや static 宣言によるファイルローカルな名前などは，各 CCC ファイルで閉じているのではなく，CCC ファイル全体にまたがって有効である．

サブクラスの定義やテキストブロックの挿入は，その順序が重要である．条件を満たすクラスが複数あった場合は，先に定義されたサブクラスが優先される．またテキストブロックも挿入の順序（または逆順序）のとおりに引用される．CCC に与える複数のファイルはコマンドラインの順序で処理をする．

CCC の処理系は C で記述され，約 2400 行である．CCC は C や C++ の構文をほとんど知らないので，CCC のために拡張された構文はあまり美しくない．

5. 記 述 例

ここでは，CCC を用いた記号処理言語の核のプログラミングの例を述べる．その他の例に関しては付録を参照してほしい．ここで述べているプログラミングには，次のような要求事項があった．

(1) 共有メモリ上でセルなどのデータを表現し，複数のプロセスからアクセスする．
(2) データは階層的な構造を持ち，それぞれにクラスを定義する．
(3) 基本的なデータ構造は，すべてのプロセスで共通であるが，それを応用したデータの構造は，必要なプロセスだけで定義する．

(1)，(2) から，オブジェクトのデータ構造の定義などを自前で行わなければならず，C++ のオブジェクトの機能は使うことができず，CCC を使う必要がある．

```
int *gMemory;
@class object (int self) {
  @class nil if (self == 0) { }
  @class fixNum if (self < 0) { }
  @class data if
    (self >= MINMEM &&
     self <  MAXMEM) { }
  }
```

共有メモリは整数の配列 gMemory で表現されている．オブジェクト ID（self）は符号付き整数で表され，0 のときは nil，負のときは fixNum，MIMMEM 以上で MAXMEM 未満のものが data で ID がそのまま gMemory のインデックスとなっている．

```
@+data {
  macro type {
    gMemory[(self >> SEGSFT)+TYPEBASE]}
  @class _ switch (type) {
    @class cons case (tagCONS) { }
    @class str  case (tagSTR) { }
  }
}
```

data は 2 の SEGSFT 乗個の整数を 1 セグメントとする BIBOP で管理されている．gMemory の TYPEBASE 以降に各セグメントの型情報が格納されている．ここには示していないが，gMemory の TYPEBASE 未満の領域には，各種の共有情報が格納されている．また，セグメントの型に応じて，cons や str などの構造が定義されている．

```
@+cons {
  macro car {gMemory[self]}
  macro cdr {gMemory[self+1]}
}
```

cons 型には car，cdr が定義されている．

基本データに関する印字関数は 1 つのファイルにまとめて記述する．

```
@+nil {void print() {printf("nil");}}
@+fixNum {void print() {
            printf("%d", value);}}
@+cons {
  void print() {
    printf("("); print(car);
    printf(" . ");
    print(cdr); printf(")");
  }
}
@+str { void print() {
          printf("%s", text);}}
}
```

ここで，value や text はそれぞれのクラスローカルなマクロである．

car，cdr のアクセスをチェックしてからアクセスする安全な関数を定義する．

```
@+object {
  int Car() { return 0; }
  int Cdr() { return 0; }
}
@+cons {
  int Car() { return car; }
  int Cdr() { return cdr; }
}
```

また，リストの先頭が特殊な値であるとき，特殊な処理をしたい．

```
@+cons {
  @class point if (car == gPoint) {
    macro X {Car(cdr)}
    macro Y {Car(Cdr(cdr))}
  }
}
```

これは，(point 10 20) のようなリストを CCC のクラスとして扱うものである．

以上のように，CCC では上で述べた要求を満たして，オブジェクトの特殊な構造を抽象化して実装することができた．これらは，13 本の CCC ファイル（合計で約 3000 行）であるが，CCC 処理系を通すことで約 6000 行のコードが得られた．

CCC が生成するプログラムは普通の C プログラムであるから，それを見て効率的に満足できない点があれば，データの表現法などを変更すればよい．たとえば，上の Car の呼出しでは，クラス data を判定に self の範囲チェックをし，type の計算（シフトと加算とメモリ参照）をして switch-case の分岐をすることが必要なコストである．

さらに，CCC は他の C や C++ のプログラムと容易にリンクできるため，CCC の使用が適している場合にのみ選択的に使える．CCC には構造体をオブジェクトの構造とし，継承で構造体が拡張されるような支援機構を持っているが，ほとんど使われていない．そのような機能は C++ と同等（またはそれ以下）だからである．

6. 考　　察

Cecil [2),3)] は条件式をクラスの定義に使えるオブジェクト指向言語である．関数型言語において，条件式を型として使用できる．Cecil にはそれをオブジェクトの型にまで拡張した predicate クラスが導入されている．

CCC と Cecil とでは，そもそも言語設計の動機が異なっているので，Cecil ではその言語が用意しているオブジェクトの構造しか扱えないという点で異なっている．さらに，Cecil では明示的にオブジェクトが存在するが CCC ではそのようなものはない．その違いがオブジェクトの関係をクラスで表現する際に現れる．CCC では，

```
@class gt(int x, int y)
   if (x > y) { }
```

のような，2 つのオブジェクト x，y の関係を新しい構造を導入することなしに，クラスとして定義できる．

Cecil ではこの例のようなオブジェクトの関係は，直接表現できず，関係を表現するためのオブジェクトを定義し，そのオブジェクトの型として表現しなければならない．たとえば，

```
object pair;
  var field x(@pair);
  var field y(@pair);
predicate gt isa pair
     when pair.x > pair.y;
  method m1(p@gt) { -- method -- }
```

のように，2 つのフィールド x，y を持つオブジェクト pair を定義し，そのオブジェクトの 1 つの状態として gt クラスを定義し，それに対してメソッドを定義するのである．大域変数によるディスパッチのようなことを Cecil で行う場合も同様である．

Predicate Dispatching[4)] はシングルとマルチメソッドディスパッチ，Cecil の Predicate Classes, ML 風のパターンマッチ，classifier を一般化したディスパッチである．非常に広い概念なので，CCC の条件クラスもこれで表現できる．

MixJuice [5)] はクラス定義とモジュール定義とを分離できる言語である．同様な仕組みは，CCC の挿入構文によって，実現できている．

モジュールローカルなマクロは数多く研究されている．たとえば，文献 7) では，モジュール内のマクロとモジュールのインポートという単純な機能を組み合わせることで，様々なプログラミングの例を示している．

CCC は分割コンパイルをしていない．そのことで，コンパイルに時間がかかる，変更しない部分をライブラリとしてしまいソースを隠蔽することができない，といった欠点がある．しかし，これらの問題はそれほど重要ではない．

一方で，スタティックに全体で 1 つのファイルを生成することで，リンカに特殊な仕掛けをすることなく，分割されたファイル間にまたがった処理が可能となり，実行時の無駄なオーバヘッドをおさえられ，プログラマにとっての直感が得られやすい，という利点がある．

7. ま と め

CCCは，最初にデータ構造が決められているような従来の言語では扱えなかった問題に対して，オブジェクト指向プログラミングを適用できるようにした言語である．クラスローカルなマクロ，プログラムの挿入などの機能があるが，それらは処理系の動作を単純で見通しの良いものになるように設計されている．

謝辞 電総研の一杉氏，NTT CS 研の櫻田氏との有益なディスカッションに感謝する．

参 考 文 献

1) Musser, D.R. and Saini, A.: *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library*, Addison-Wesley (1996).
2) Chambers, C.: Object-Oriented Multi-Methods in Cecil, *Proc. ECOOP'92*.
3) Chambers, C.: The Cecil Language: Specification and Rationale, Technical Report 93-03-05, University of Washington (1993).
4) Ernst, M., Kaplan, C., and Chambers, C.: Predicate Dispatching: A Unified Theory of Dispatch, *ECOOP'98*.
5) 一杉裕志：MixJuice, SPA'2000 Poster.
6) 原田康徳，山崎憲一：条件クラス：データ抽象なしの多相と継承，プログラミング研究会 (Jan. 1996).
7) Waddell, O. and Dybvig, R.K.: Extending the Scope of Syntactic Abstraction, *POPL99*.

付録 CCC のプログラム例

ここでは，本文では触れにくいが，より詳細な例を用いて，CCC のプログラミングの特徴を見ていく．2つの例，X-Window における XEvent 構造体をクラス的に解釈するプログラム，グラフィック用ファイルフォーマット PNG を解釈するプログラムを紹介する．

X-Window の例では，まずイベント構造体の `type` フィールドの値によって，イベントの種類を判別しクラス分けを行っている．さらにマウスボタンのイベントのサブクラスとして，大域変数にイベントの時刻を格納し，判定することで，クリック，ダブルクリックのクラスを定義している．

CCC には多重継承や mixin の機能がないため，`BTN1`，`SFT` のような条件式のマクロを定義して，それをサブクラスの定義で再利用している．

クリックのサブクラスとして，どのウインドウでクリックされたかというクラスを定義している（`tool1_click` など）．本来なら効率や汎用性などを考慮すると，サブウインドウからオブジェクト ID へのハッシュ表を用いてディスパッチするところであるが，ここではクラス階層のバリエーションの例としてあげている．

各クラスでは `_event()` を定義することで，そのイベントが生じたときに起動するメソッドを定義できる．また，`XEvent` 構造体の直下のクラスにはデバック用の `dump` メソッドが定義されている．

C++ ではこのようなクラス階層を表現するためには，イベント列を解釈して独自のイベントオブジェクトに変換する必要があった．仮にそのようなイベントオブジェクトを定義したとしても，この例のように特定のサブウインドウで生じたイベントを特定のクラスとして定義するためには，新しいクラスを定義するだけでなく，イベント列を解釈するプログラムも修正しなければならない．

```
#include <X11/X.h> #include <X11/Xlib.h>
#include <stdio.h>

/* 直前にボタンが押された時刻 */
Time gLastButtonDown[4] = {0};
/* 直前にクリックされた時刻 */
Time gLastClick[4] = {0};

/* クリックの判定用 */
Time gClickInterval = 100;
/* ダブルクリック判定用 */
Time gDoubleClickInterval = 300;

Window gTool1;
Window gTool2;

@class XEv (XEvent *ev) {
  @class _ switch (ev->type) {
    @class cKeyPress
        case (KeyPress) { }
    @class cKeyRelease
        case (KeyRelease) { }
    @class cButtonPress
        case (ButtonPress) { }
    @class cButtonRelease
        case (ButtonRelease) { }
     /* 以下省略 */
  }
  void event() {
```

```
    before_event(ev);
    _event(ev);
    after_event(ev);
  }
  void before_event() { }
  void _event() { }
  void after_event() { }
}

@+cButtonPress {
       /* イベントが発生した時刻 */
  macro TIME {ev->xbutton.time}
  macro BTN {ev->xbutton.button}
  @class _ {
    void after_event() {
      gLastButtonDown[BTN] = TIME;
    }
  }
}
@+cButtonRelease {
       /* イベントが発生した時刻 */
  macro TIME {ev->xbutton.time}
       /* どのボタンが離されたか */
  macro BTN {ev->xbutton.button}

  macro BTN1 {BTN == Button1}
  macro BTN2 {BTN == Button2}
  macro BTN3 {BTN == Button3}
  macro STATE {ev->xbutton.state}
  macro SFT  {STATE & ShiftMask}
  macro CTRL {STATE & ControlMask}

     /* イベントが生じたサブウインドウ */
  macro SUBWINDOW {ev->xbutton.subwindow}

     /* ボタンが押された時刻と離された
        時刻の差でクリックを判定 */
  @class click
    if ((TIME - gLastButtonDown[BTN])
          < gClickInterval) {
    void after_event() {
      /* 直前のマウスクリックの時刻 */
      gLastClick[BTN] = TIME;
    }
    /* このクリックと 1 つ前のクリックの
       時刻の差でダブルクリックを判定 */
    @class doubleClick
      if ((TIME - gLastClick[BTN])
            < gDoubleClickInterval) {
      }
  }
}

/* 各ボタンごとのクラス定義 */
@+click {
  @class click1 if (BTN1) { }
  @class click2 if (BTN1) { }
  @class click3 if (BTN1) { }
}

@+doubleClick {
  @class dclick1 if (BTN1) { }
  @class dclick2 if (BTN2) { }
  @class dclick3 if (BTN3) { }
}

/* ボタン 1 に対するモディファイア
   キー用のクラス */
@+click1 {
  @class click1s if (SFT) { }
  @class click1c if (CTRL) { }
}

/* イベントの動作の定義 */

@+dclick1 {
  void _event() {
    printf("double click!\n");
  }
}
@+click1 {
  void _event() {
    printf("click!\n");
  }
}
@+click1s {
  void _event() {
    printf("shift click!\n");
  }
}
@+click1c {
  void _event() {
```

```
    printf("control click!\n");
  }
}

/* 特別なサブウインドウ上で生じたクリック */
@+click1 {
  @class tool1_click
      if (SUBWINDOW == gTool1) { }
  @class tool2_click
      if (SUBWINDOW == gTool2) { }
}

@+tool1_click {
  void _event() {
    printf("tool1 click\n");
  }
}
@+tool2_click {
  void _event() {
    printf("tool2 click\n");
  }
}

/* デバッグ用のイベントダンプ */
@+cKeyPress {
  void dump() {
    printf("KeyPress %d %d\n",
      ev->xkey.keycode, ev->xkey.state);
  }
}
@+cKeyRelease {
  void dump() {
    printf("KeyRelease %d %d\n",
      ev->xkey.keycode, ev->xkey.state);
  }
}
@+cButtonPress {
  void dump() {
    printf("ButtonPress %d %d %d %d\n",
      ev->xbutton.button,
      ev->xbutton.state,
      ev->xbutton.x, ev->xbutton.y);
  }
}
@+cButtonRelease {
  void dump() {
    printf("ButtonRelease %d %d %d %d\n",
      ev->xbutton.button,
      ev->xbutton.state,
      ev->xbutton.x, ev->xbutton.y);
  }
}

main() {
  Display *d;
  Window  win;
  XEvent ev;
  d = XOpenDisplay(0);
  win = XCreateSimpleWindow(d,
            RootWindow(d, 0),
            0, 0, 100, 100, 0,
            WhitePixel(d, 0),
            BlackPixel(d, 0));

  XSelectInput(d, win, ExposureMask |
            KeyPressMask | KeyReleaseMask |
            ButtonPressMask |
            ButtonReleaseMask);

    /* クラス tool1_click 用ウインドウ */
  gTool1 = XCreateSimpleWindow(d, win,
            5,  5, 30, 30, 2,
            WhitePixel(d, 0),
            BlackPixel(d, 0));
    /* クラス tool2_click 用ウインドウ */
  gTool2 = XCreateSimpleWindow(d, win,
            5, 55, 30, 30, 2,
            WhitePixel(d, 0),
            BlackPixel(d, 0));

   /* 以下省略 */

}
```

PNG フォーマットを扱うプログラム例を述べる．PNG フォーマットは先頭からの 8 バイトにマジックナンバーなどの固定情報が格納され，その後，同じ構造をした複数のチャンクデータが続いている．ここではこのチャンクデータをクラスとして定義している．

各チャンクは先頭にデータ部の長さが 4 バイトで格納され，次に 4 バイトからなるデータ型，データ部，最後に CRC が格納されている．

この例では，構造を判定したあと，その内容を印字

するメソッドを定義している．

また，ファイルの終了を示すチャンク IEND と CHUNK クラスとにメソッド endp() が定義されている．

PNG はこの基本的な構造を保持したまま，拡張が行われる．その拡張に対しては，CCC の挿入構文によって後からチャンク型の追加が容易である．

このプログラムは，PNG のフォーマットを理解するためにも有用である．たとえば C で書かれた典型的な PNG を操作するプログラムでは，いくつかの定数や構造体が定義されているが，それらの関係はプログラムを見るだけではなかなか理解できない．しかし，このように CCC で書かれたプログラムは，定数の局所化，条件式と構造体の関係が明示されていることから，ある程度の理解を助けるものと思われる．

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

@class CHUNK (char *obj) {
  /* 符号無し文字で参照する */
  macro uobj {((unsigned char*)obj)}
  /* 4バイト整数 */
  macro asUINT(x) {
         ((uobj[x]<<24)+(uobj[x+1]<<16)
         +(uobj[x+2]<<8)+(uobj[x+3]))}
  /* データ部のバイト数 */
  macro length    {asUINT(0)}
  /* タイプ名 (4バイト) */
  macro name      {(obj+4)}
  /* データ部の後に続く */
  macro CRC       {(obj+length+8)}
  /* 次のデータチャンクの先頭アドレス */
  macro NEXT      {(obj+length+12)}

  macro _print() {
    printf("%x %c%c%c%c ", obj,
      name[0], name[1], name[2], name[3])
  }
  void print() {
    _print();
    printf("%d\n", length);
  }
  char *next() { return NEXT; }

  /* データタイプのチェックマクロ */
  macro check(c1,c2,c3,c4)
    {(name[0]==c1 && name[1]==c2
      && name[2]==c3 && name[3]==c4)}

/* ヘッダー型 */
@class IHDR if (check('I','H','D','R')) {
  /* 図形のサイズ */
  macro width     {asUINT(8)}
  macro height    {asUINT(12)}
  /* 各種属性 */
  macro coldepth  {uobj[16]}
  macro ctype     {uobj[17]}
  macro comp      {uobj[18]}
  macro filter    {uobj[19]}
  macro interlace {uobj[20]}
  void print() {
    _print();
    printf(
      "%d %d %d %d %d %d %d\n",
      width, height, coldepth, ctype, comp,
      filter, interlace);
  }
}
/* ガンマ値の設定 */
@class gAMA if (check('g','A','M','A')) {
  macro igamma {asUINT(8)}
  void print() {
    _print();
    printf("Image gamma= %d\n", igamma);
  }
}
/* データ本体 */
@class IDAT if (check('I','D','A','T')) {
  /* data(x) の中身を zlib によって伸長すると,

     ピクセル値の列が得られる．ピクセルの並
     び方は，ヘッダの属性によって決まる */
  macro data(x) {uobj[8+x]}
  void print() {
    int i;
    _print();
    printf("IDAT lenght=%d (", length);
    for (i=0;i<length;i++) {
      printf("%02x ", data(i));
    }
    printf(")\n");
  }
}
```

```
  /* ファイルの終了 */
  @class IEND if (check('I','E','N','D')) {
    void print() {
      _print();
    }
    /* ファイル終了の判定 */
    int endp() { return 1; }
  }
  /* ファイル終了の判定 */
  int endp() { return 0; }
}

char *fileOpen(char *fileName) {
  FILE *fp;
  char *body;
  int len;
  fp = fopen(fileName, "r");

  fseek(fp, 0, SEEK_END);
  len = ftell(fp);
  fseek(fp, 0, SEEK_SET);
  body = (char *)malloc(len + 1);
  fread(body,len,1,fp);
  fclose(fp);
  return body;
}
main(int argc, char **argv) {
  char *obj = fileOpen(argv[1]);
  int i;
  obj+=8; /* magic ナンバーをスキップ */
  /* ファイルの終了チャンクが現れるまで */
  while (!endp(obj)) {
    /* 各チャンクを表示し */
    print(obj);
    /* 次のチャンクに進む */
    obj = next(obj);
  }
}
```

(平成 12 年 5 月 26 日受付)
(平成 12 年 9 月 8 日採録)

原田 康徳（正会員）

1963 年生．1992 年 3 月北海道大学大学院工学研究科情報工学専攻博士後期課程卒業．博士（工学）．同年 4 月日本電信電話（株）入社．1998 年 10 月科学技術振興事業団さきがけ研究員．ビジュアルプログラミング言語の研究に従事．日本ソフトウェア科学会，ACM 各会員．

山崎 憲一（正会員）

1961 年生．1984 年東北大学工学部通信工学科卒業．1986 年同大学院情報工学科修士課程修了．同年，日本電信電話（株）入社．現在，NTT 未来ねっと研究所ネットワークインテリジェンス研究部主任研究員．記号処理専用計算機，記号処理プログラミング言語，日本語文書処理の研究に従事．ACM 会員．