ちょっと Eiffel 見物

香川考司

1 はじめに

Eiffel は Bertrand Meyer 氏が設計した強い型付きの (生まれつきの) オブジェクト指向言語です。ゴミ集め (garbage collection, GC) の機能も持っています。型システムは、柔軟性を失うことなく、”Message not understood” のエラーを出さないことを目標としていますが、今のところ、それに成功しているとはいえないようです。その他、“表明” や “例外処理” などの機構にも特徴があるようです。ここでは、言語が設計された際の哲学にはあまり深入りせずに、実際に動かすのに必要なことと、型システムの話について紹介したいと思います。なお参考文献として [2, 3, 4, 5] を挙げておきます。[5] は Eiffel を 80%理解したい人向きの日本語の文献です。お推めは [4] で、 Eiffel を 98%理解したい人向きの 100 ページちょっとの入門書です。 [2] は、 Eiffel を 101%理解したい人向きか、リファレンスマニュアルとしてお推めします。[3] は Eiffel を用いてオブジェクト指向の一般的な技法を説明した本だそうです。

2 環境の設定・コンパイル

この節の内容は言語の仕様には定められていないので、処理系依存です。ここでは数理解析研究所にある ISE Eiffel 3 について述べます。なお ISE Eiffel 3 は ebench という GUI を持った統合環境でプログラミング・コンパイル・実行・デバッグを行うのが本来の使用法なのですが、数研では 1 ユーザライセンスしかとっていないということもあり、バッチコンパイラ (es3) について説明します。これだと、ある人がソースファイルを編集している間に他の人がコンパイルを行なうことができます。この ebench そのものに興味のある方はマニュアルを読んでください。オンライン・ドキュメントは`$EIFFEL3/doc/*`にあります。また sample program は`$EIFFEL3/examples/`にあります。

2.1 環境変数

環境変数は次のように設定します。

```
setenv PLATFORM sparc
setenv EIFFEL3 /usr/local/Eiffel/Eiffel3
```

また、path に `$EIFFEL3/bench/spec/$PLATFORM/bin` を追加します。

2.2 Ace の記述の仕方

Ace とは、まぁ Eiffel 用の Makefile みたいなもんです。コンパイラにどこのディレクトリでクラスの定義を見つけるか、などの命令を与えるものです。Ace ファイルのデフォルト名は `Ace` です。図 1 に Ace ファイルの例を示します。

```
system
        calculator  -- executable の名前です。

root
        calculator (ROOT_CLUSTER): "make"
        -- 順に、
        --   root class (メインルーチンの存在するクラス)、
        --   root cluster (ルートクラスのあるクラスタ)、
        --   root creation procedure (メインルーチン)
        -- ここで (ROOT_CLUSTER) は省略可能です。


default
        precompiled ("$EIFFEL3/precompiled/spec/$PLATFORM/base")
        -- とにかくこう書きます。

cluster
        ROOT_CLUSTER:           ".";
        -- クラスタとディレクトリの対応を記述します、クラスタが複数
        -- ある場合は ;で区切って並べます。
```

図 1: Ace ファイルの例

2.3 コンパイルの仕方

Ace ファイルのあるディレクトリで `es3` を (引数なしで) 起動します。そうするとそのディレクトリの下に `EIFFELGEN` というディレクトリ、さらにその下に `COMP`, `W_code`, `F_code` という 3 つのディレクトリを生成します。そのあと、この `EIFFELGEN/W_code` で (最初に 1 度だけ) 以下のコマンドを実行して下さい。

`% ln -s` *precompilation_directory*`/EIFFELGEN/W_code/driver` *system_name*

ここで、*precompilation_directory* は Ace ファイル中で `precompiled` で指定したディレクトリ、*system_name* は実行形式の名前です。プログラムの実行にはこの *system_name* を実行して下さい。その他の `es3` のオプションについては `es3 -help` としてみて下さい。

ISE Eiffel 3 は**メルティングアイス (melting ice)** 技法[1]を用いています。これは (C を通して) コンパイルされたプログラムと、解釈実行 (interpret) される部分を必要に応じて組み合わせられる、というものです。これにより、プログラム開発時のコンパイル時間は短くでき、しかも、最終的には完全にコンパイルされた速いプログラムを動かすことができます。

最終的なスタンドアローンのプログラムを作るには `es3 -finalize` です。

3 変数、オブジェクト、命令文

Eiffel では、プログラミングとはクラスを定義することと同値だと考えていただいてもかまいません。通常 1 つのファイルに 1 つのクラスを定義するようです。ちなみにファイルの拡張子には `.e` を用いるようです。

文法は Pascal 系の言語によく似ています。コメントは、“--” から行末までです。

3.1 用語のまとめ

エンティティ (entity) — いわゆる変数、(当然) C や Pascal の変数と同様に assignable

[1]だから Eiffel は potable な言語なんだそうな。

属性 (attribute) — インスタンス変数、当然代入可能 (assignable)

ルーチン (routine) — メソッドのこと、手続きと関数があります。

特性 (feature) — 属性とルーチンの 2 つをまとめたもの

3.2 変数とオブジェクト

`x : T`

`x` という変数が `T` という型 (クラスと同一視して良い) を持つことをあらわします。関数の引数、クラスの属性、局所変数の宣言の時にこの形を用います。

`x := y`

代入文。C や Pascal のものと大体同じです。`y` の型が `x` の型に適合 (conform, 後述) している必要があります。

`!!x`

生成文 (creation instruction)。どこかで `x : T` という宣言があるとすると、型 `T` の新しいオブジェクトを生成して `x` に束縛します。C の malloc、Pascal の new にだいたい対応します。

参照されなくなったオブジェクトは**ゴミ集め (garbage collection)** によって自動的に回収されます。それゆえ、Eiffel には “destructor function” はありません。ガーベッジコレクタは大抵の処理系でインクリメンタルなものが使われるようです。つまり、一度に実行をストップさせることなく、少しずつ GC を行ないます。

いくつかの基本型

`BOOLEAN, CHARACTER, INTEGER, REAL, DOUBLE, ARRAY, STRING` が用意されています。このうち `ARRAY` は詳しく言うと総称クラス (parametric な型) になります。また Eiffel は基本的に case insensitive なので、`boolean, Integer` などと書いても同じことです。

基本型の変数は最初は型ごとに決められたデフォルトの値に束縛されます。その他のユーザが定義した型の変数は、(代入されたり生成されたりする前は) `void` (C の `NULL`) に束縛されます。これに対して access を試みるとエラーになります。

いくつかの定義済みオペレータ

`BOOLEAN` `not, or, and, implies, or else, and then`
このうち、最後の 2 つは *lazy*(つまり、C の &&, || と同じ) です。

`INTEGER` `+, -, *, //, \\, ^, <, >, <=, >=`
この内 `//`は整数の割り算、`\\`は余りを表します。

`REAL` `+, -, *, /, ^, <, >, <=, >=`

当然かも知れませんが、関数型はありません。これらのほかに =, equal などというすべての型に対して適用できるオペレータもあります。

3.3 命令文

複合命令文 (compound instruction)

単なる命令文の列のことで、順番に実行されます。区切りのセミコロンは省略できます。図 2 の make 手続きの本体などがこの例です。

条件文 (conditional instruction)

```
if b_1 then    -- b_i は Boolean 型の式
   c_1         -- c_i は命令文
elseif b_2 then
   c_2
 ...
elseif b_k then
   c_k
else
   c_e
end
```

という形式です。ここで、elseif 節や else 節は省略できます。

繰り返し文 (iteration instruction)

```
from
   c_1     -- 初期化を行なうための文
until
   b       -- 終了条件 (Boolean)
loop
   c_2     -- 繰り返しの本体
end
```

ほかに分岐文 (multibranch instruction)、デバッグ文 (debug instruction) というのがありますが、あまり使うとはおもえません。

手続き (procedure)

関数とともにクラスの定義の中で定義されます。

```
pname (a_1 : T_1; a_2 : T_2; ...; a_n : T_n) is
local
        -- ここに局所変数 (local entity) の宣言を書く (assignable)
do
    c     -- ここに複合命令文を書く
end
```

ここで局所変数を使わない場合は local で行なわれる局所変数の定義は省略可能です。C と同じで手続きの中で仮引数に代入を行なっても実引数に影響はありません[2]。

[2] もしかしたら warning が出るかもしれない。

関数 (function)

fname (*a*$_1$: *T*$_1$; *a*$_2$: *T*$_2$; ...; *a*$_n$: *T*$_n$) : *T* is
local
　　-- ここに局所変数 (local entity) の宣言を書く
do
　c　-- ここに複合命令文を書く
end

返り値の型を宣言する必要があることと、*c* のなかに `Result := obj` という式が必要なことが異なります。この *obj* がこの関数の返り値になります。

例: GCD

下に示すのは最大公約数を計算する関数です。

```
gcd (m, n : Integer) : Integer is
  do
     if n = 0 then
        result := m
     else
        result := gcd (n, m \\ n)
     end -- if
  end -- gcd
```

ところで、 Eiffel では関数や手続きは必ずあるクラスに属していなければいけないので、このような関数をどのクラスに定義するのが適当か、という問題があります。ようするに Eiffel ではクラスとモジュールの概念がわかれていないのです[3]。

外部関数の呼び出し

C など[4]で書かれた関数を呼び出す方法もあって、

```
pname (a1 : T1; a2 : T2; ...; an : Tn) is
  external "C"
  alias "C の関数の名前"

  end
```

のように書くだけのようです。

4　クラスとオブジェクト

オブジェクト指向とは何か、という根源的な疑問はありますが、それはさておき、クラスとオブジェクトというのが必要なのは間違いないようです[5]。

[3] また Eiffel で**末尾再帰 (tail recursion)** がどのような扱いをされるのか、ということも疑問になりますが、調べていません。
[4] 今のところ C **だけ**らしい。
[5] もちろん例外はあります。

4.1 クラスの定義

属性 (attribute) と**ルーチン (routine)** をまとめて**特性 (feature)** と呼びます。クラスの定義はこれらを定義することです。そして、Eiffel ではクラスを定義することが、プログラミングにほかなりません。また Eiffel ではクラス (class) と型 (type) を同一視して構いません。

クラスの定義は次のように行ないます。

```
class C

inherit
   --ここに ;で区切られた親クラスのリスト

creation
   -- 生成手続きのリスト

feature
   -- 特性の宣言と定義

invariant
   -- クラス不変量 (表明と例外の項を参照)

end -- class C
```

feature 節以外は optional です。この feature 節に属性とルーチンの宣言と定義を書きます。クラスが異なれば同じ名前の特性を定義できます。実際にどの定義を用いるかは、そのメッセージが送られたオブジェクトの実際の型によって決定されます。これがオブジェクト指向言語の本質の一つです。feature というキーワードの後ろにクラス名を {}でくくって並べて、これらの特性を export する範囲を明示することができます。何も書かなければ feature { ANY }と書くのと同じです。また feature{ NONE }と書くと、この特性を持つオブジェクトの中でのみ参照可能になります。(同じクラスでも、別のオブジェクトからは見えません。)

図 2 に示すのは、クラス定義の例です。($EIFFEL3/examples/base/calculator/calculator.e)

```
class CALCULATOR inherit SET_UP

creation make

feature
   make is
     do
        io.putstring ("%N*********************************%N");

        io.putstring ("Calculator in reverse Polish form%N");
        io.putstring ("*********************************%N");

        initialize;
        session
     end;
   ...
end -- class CALCULATOR
```

図 2: クラス定義の例

creation 節には feature で定義されたルーチンの名前を並べます。この節がある時にはオブジェクトを !! を用いて生成するときにこのルーチンのうちの 1 つを呼ばなければいけません。make というルーチンが変数 x の属するクラスの creation 節にあるとすると

!!x.make(e_1, ..., e_n)

!!x.make -- 無引数の場合

のようにして、オブジェクトを生成します。

ルーチンを呼び出すには、御想像の通り x.p, x.f という形 (無引数の場合) か、x.p(e_1, ..., e_n), x.f(e_1, ..., e_n) の形 (引数のある場合) を用います。また、属性の値を読み出すためには x.a です。ここで、無引数の関数を呼び出すのと、属性の読み出しは全く同じ形をしていることに注意して下さい。これを**参照の統一(?)** というそうです。これを用いて memo-function (つまり、一度計算した結果をメモしておく) のような技法を行なえます。また、最初は属性として定義しておいて、後で残りの部分を変更せずに、関数に実装を変更することができます。

またオブジェクトの属性に外部から代入を行なうことはできません。つまり x.a := *obj*ということはできません。このような代入を行なえるのはそのクラスのメソッドのみとなります。これは、契約によるプログラミングという観点からは当然のことでしょう。

Smalltalk の self や C++の this にあたる変数 (つまり、現在ルーチンを実行しているオブジェクトを参照する変数) は、Eiffel では Current です。Current の特性を呼び出す時には Current. をその前につける必要はありません。

例: リスト

下のクラスは cons セルを表すクラスです。

```
class NODE

feature
   item : ELEMENT
   next : NODE      -- 最初は VOID

   set_item (x : ELEMENT) is
      do
         item := x
   end

   set_next (n : NODE) is
      do
         next := n
      end
end -- class NODE
```

ここで ELEMENT は、実際に必要な型 (Integer, Real など) に変更しても構いません。これらを一度に定義する方法は後で紹介します。

問題: 上で定義されたクラス NODE を用いて次のような特性を持つクラスを定義せよ

```
    LIST   count : INTEGER
           first : NODE
           empty : BOOLEAN
           has (x : ELEMENT) : BOOLEAN
           add (x : ELEMENT)
           remove (x : ELEMENT)

   QUEUE   add (x : ELEMENT)
           remove
           front : ELEMENT
           empty : BOOLEAN
```

その他

infix operator, constant, array, unique, once

4.2 継承 (inheritance)

inherit 節は具体的には次のように書きます。

```
inherit
   C1
      rename
         f1 as g1,
         f2 as g2,
         ...
      redefine
         h1, h2, ...
      end;
   C2
      ...
      ;
   ...
```

$C_1, C_2, \ldots$が親クラスです。rename は多重継承をした時に名前の衝突を避けるために行ないます。また再定義する特性は redefine 節の中に並べて書かなければなりません。また、creation は継承されません。invariant 節については、あとで述べます。

注: 省略できないセミコロン (;)

大抵のセミコロンは省略できますが、

1. 手続き (procedure) と関数 (function) を定義する際の、型の異なる仮引数の宣言の間の区切り
2. クラス定義の際の親クラスの区切り

の 2 つだけは省略できません。

多重継承

これからわかるように Eiffel では**多重継承 (multiple inheritance)** を持ちます。これにより生じる名前の衝突や曖昧さは rename や select を用いて明示的に解決します。また rename は super class の同じ名前のメソッドを呼び出したい時にも必要になります。

Anchored declaration

属性やルーチンの引数の型を like という予約語を用いて、他の属性や引数と同じ型であるというふうな宣言の仕方をすることができます。これを**関連付けによる宣言(declaration by association** または **anchored declaration)** と言います。

```
class TREE

feature
```

```
   parent  : TREE
   child   : like parent
```

このクラスを継承する時に、アンカーの型が変わればこのように anchored declaration された属性の型も自動的に変わるので便利です。例えば

```
class WINDOW

inherit
   TREE ...
        redefine parent
        end;
   ...

feature
   parent : WINDOW
```

とすると WINDOW クラスでは child の型も自動的に変わります。特に like current という形は良く用いられます。

型適合 (conformance)

簡単に言えば、子孫クラスは先祖クラスに適合します。もちろん、正式な定義は、総称クラスやアンカーが絡むので少し複雑です。

型適合は次のような場合に用いられます。例えば Extra は Base に、ColoredPoint は Point に適合しているとします。そして今 Foo というクラスを継承して、Bar というクラスを定義しようとしているとします

1. 代入 x := y という文がある時、y の型は x の型に適合していなければいけません。つまり x : Point; y : ColoredPoint の時は許されますが、逆はいけません。
2. 生成文 !T!x という文は、型 T のオブジェクトを生成して、変数 x に束縛しますが、この時 T は x の型に適合してなければいけません。つまり !ColoredPoint!x のようなことを行なうことができます。
3. ルーチン呼び出しの時、実引数の型は仮引数の型に適合しなくてはいけません。すなわち proc (x : BASE) というルーチンがあって y : EXTRA のとき proc (y) は可能です。
4. 属性を再定義する時、元の型に適合する型に置き換えなくてはいけません。Foo に x : Base という属性がある時、Bar で x : Extra に再定義することはできますが、その逆はできません。
5. 関数を再定義する時、返り値の型は元の返り値の型に適合する型に置き換えなくてはいけません。
6. ルーチンを再定義する時、引数の数は変えられません。また、仮引数の型は元の対応する仮引数の型に適合する型に置き換えなくてはいけません。Foo に proc (x : Base) というルーチンがある時、Bar で proc (x : Extra) に再定義することはできますが、その逆はできません。

特に怪しいのは、最後のルールです。しかし、下の "<"の定義のように、これができないと困る場合も出てきます。

また、最初の 2 つのルールも怪しいですがヘテロなコンテナ (heterogeneous container) を実現するためには必要かもしれません。例えば Point のリストに ColoredPoint を代入してリスト全体を move したいという場合です[6]。

[6]高階関数があるわけではないので、こういうことを簡単にできるわけではありませんが。

Eiffel の型システムは現時点では完全ではありません [1]。(つまり、runtime に型エラーが起こり得ます。) comp.lang.eiffel の FAQ によると、仕様ではこのような型エラーはコンパイル時に捕捉されることになっているようですが、少なくとも effective ではないようです[7]。(現実の処理系ではコンパイル時に型エラーを捕捉しません。)

暫定クラス　(deferred class)

メソッドの定義を持たず、親クラスとしてのみ用いられるクラスのことです。例えば大小比較を行なうルーチン ("<"など) を定義する親クラスでは、このルーチンの定義を行なうことはできません。この場合 deferred というキーワードを用います。

```
deferred class COMPARABLE

feature
   infix "<" (other : like current) : BOOLEAN is
      deferred
      end

   infix ">" (other : like current) : BOOLEAN is
      do
         result := (other < current)
      end
```

この例では、infix operator の定義の仕方や anchored type の使い方も見ることができます。また ">"の定義でわかるように、暫定クラスのすべてのルーチンが deferred である必要はありません。また、暫定クラスのインスタンスは生成することはできません。

4.3　総称クラス (generic class)

パラメータ化されたクラスのことです。リストや配列などのコンテナとして良く用いられます。このパラメータは

class C [G_1, G_2, ..., G_k]

のように [と] で囲みます。この G_n はクラスの定義の中で普通の型のように使うことができます。

例: 総称リスト

以前、定義した NODE クラスを総称リストに書き換えると、

```
class NODE [G]
   item : G
   next : NODE [G]
   ...
```

のようになります。ようするに ELEMENT を G に書き換えるだけであとは何も変更しません。
問題: List や Queue も総称クラスに書き換えよ。

[7]プログラム全体で使われ方を調べたりしなければいけないらしい。

Constrained genericity

たとえば、binary tree などを実装する時にパラメータがある特性 (例えば、全順序を持つ、COMPARABLE) を持っていることを仮定したい時があります。このような場合、次のような記法を用います。

class C [G_1 -> T_1, G_2 -> T_2, ...]

T_i はクラス名です。

例: Sorted list

要素を必ずソートされた状態で持つリストを考えてみましょう。

```
class SORTED_LIST [G -> COMPARABLE]

feature
   first : NODE [G]
   count : INTEGER
   found : BOOLEAN
   add (x : G) is
      ...
```

このようにしておくと、ルーチン add のなかで x が COMPARABLE であるという事実を用いることができます。つまり、x < first.item のような文を書くことができます。

問題: 総称クラス SET を binary search tree として実装せよ。

5 表明と例外

ルーチンには ensure により**事後条件 (postcondition)**、require により**事前条件 (precondition)** を付加することができます。これは Bertrand Meyer のいうところの**契約によるプログラミング (programming by contract)** という考え方に基づくものです。これらの**表明 (assertion)** は実行時に (optional に) チェックされます。表明が満たされない時は通常その旨を表示してプログラムは終了します。しかし、rescue 節と retry 文を用いて表明が満たされなかった時にどうするかを明示的にプログラムすることもできます。

ルーチンの定義の完全な形式は次のようになります。

```
proc (e1, ..., en) is
   require
      br -- 事前条件
   local
         -- 局所的変数の宣言
   do
      cd
   ensure
      be -- 事後条件
   rescue
      cr -- エラー処理
   end
```

このルーチンが子孫クラスで再定義される時、これらの表明も継承されます。さらに子孫クラスで表明を付け足すこともできます。このときには require else と ensure then というキーワードを用います。事前条件では継承された条件との OR、事後条件では継承された条件との AND になります。

その他の表明として、ループ不変量 (loop invariant)、ループ変量 (loop variant)、クラス不変量 (class variant)、チェック文などがあります。クラス不変量も当然継承されます。

以下は繰り返し文の完全な形式です。

```
from
   c_i -- 初期化
invariant
   b_i -- ループ不変量
variant
   i_v -- ループ変量 (Integer 型、非負で単調減少)
until
   b_t -- 終了条件
loop
   c_b -- ループ本体
end
```

6 Eiffel Vision について

ISE Eiffel 3 は Eiffel Vision という GUI ライブラリも持っています。C のツールキット (Motif か OPEN LOOK) に基づいているので使い勝手はそんなに変わらないはずです (実はよく知らない)。
`$EIFFEL3/examples/vision/` のサンプルを参照して下さい。この場合 Ace ファイルに

```
precompiled ("$EIFFEL3/precompiled/spec/$PLATFORM/mvision")
```

と書くことに注意して下さい[8]。

また、`ebuild` という GUI ビルダーもあります。こちらはオンラインドキュメント `$EIFFEL3/doc/12.ps` を参照して下さい。とりあえず、path に `$EIFFEL3/build/spec/$PLATFORM/bin` をくわえて、

```
% xrdb  -load $EIFFEL3/build/app-defaults/XeiffelBuild
```

をしておけば、あとは `ebuild` を起動するだけです。

[8] `mvision` は Motif のライブラリを用いる場合。実は今のところちゃんと動かない。

参考文献

[1] William R. Cook. A proposal for making eiffel type-safe. In *ECOOP '89*, pages 57–70, 1989. Nottingham.

[2] Bertrand Meyer. *Eiffel: Language and Environment*. Prentice Hall, 1991. $36.00 ISBN:0-13-247925-7.

[3] Bertrand Meyer. *Object-Oriented Software Construction (Second Edition)*. Prentice Hall, 1994. To appear.

[4] Robert Switzer. *Eiffel: Introduction*. Prentice Hall, 1993. $24.00 ISBN:0-13-105909-2.

[5] 酒匂 寛. Eiffel — 仕様記述能力を持つオブジェクト指向言語 —. **情報処理**, 35(3):204–214, May 1994.

A 付録

A.1 予約語

`alias all and as BIT BOOLEAN CHARACTER check class creation Current debug deferred do DOUBLE else elseif end ensure expanded export external false feature from frozen if implies indexing infix inherit inspect INTEGER invariant is like local loop NONE not obsolete old once or POINTER prefix REAL redefine rename require rescue Result retry select separate STRING strip then true undefine unique until variant when xor`

A.2 演算子の結合順位

10 `.`
9 `old strip not` (unary)+ (unary)- (All free unary operators)
8 (All free binary operators)
7 `^`
6 `* / // \\`
5 (binary)+ (binary)-
4 `= /= < > <= >=`
3 `and and then`
2 `or or else xor`
1 `implies`

同じ優先順位の演算子は左に結合します。