2015 年度 ソフトウェア工学演習 第 9 回 *

横山 哲郎

1　算術式の解釈と翻訳 — Calc

本章では，プログラミング言語を設計して，その解釈器（インタープリタ）と翻訳器（コンパイラ）を実現する．解釈器は，ソースコードを一部ずつ逐次解釈をすることを繰り返しながら実行するプログラムである．翻訳器は，ソースコードをオブジェクトコードに変換するプログラムである．

本章で用いる表記法について説明する．ソースコードはタイプライタ体（例，`abc`）で表記する．数学的な対象は数式フォント（例，123）を用いる．集合には太字体（例，**Digit**）を用いる．ソースコードを引数に取る意味関数は二重括弧 $[\![\cdot]\!]$ を用いる．たとえば，$\mathcal{N}[\![\texttt{123}]\!] = 123$ は，ソースコード `123` を引数にとる意味関数 $\mathcal{N}$ が数 123 を返すことを意味する．

1.1　言語の設計

自然数と加算・減算・乗算といった二項演算子から構成される算術式を考える．たとえば，`1+2-3*4` は算術式である．本節では，算術式のみからなる簡単なプログラミング言語 Calc を設計する．まずは，プログラミング言語の構文と意味論を定める．

構文にはいくつかの定め方がある．本稿では，まず集合の上で帰納的に定義する方法を述べる．次に，より簡潔な BNF(Backus-Naur form) を用いる方法を述べる．

1.1.1　帰納的定義で定める構文

10 進表記の一桁の自然数を表す記号からなる集合

$$\mathbf{Digit} = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}\} \tag{1}$$

を考える．Digit は英語で一桁のアラビア数字のことを指す．**Digit** の要素は単なる記号であり，数の集合ではない．たとえば，タイプライタ体で書かれた `3` は数 3 ではない．**Digit** の要素がアラビア数字である必然性はない．後に関連づける数の類推がしやすいために用いているに過ぎない．類推しにくくてもいいのであれば，たとえば，$\mathbf{Digit} = \{$ 壱, 弐, ...$\}$ でも $\mathbf{Digit} = \{$ ひとつ, ふたつ, ...$\}$ でも構わない．

次に二桁以上の自然数を考慮に入れる．10 進表記の自然数を表す記号列からなる集合 **Num** を考える．非形式的には，$\mathbf{Num} = \{\texttt{0}, \texttt{1}, \ldots\}$ や

$$\{d_n \ldots d_2 d_1 \mid d_i \in \mathbf{Digit}, 1 \leq i \leq n, n \geq 1\} \tag{2}$$

* 最終更新日: 2016/01/19 16:31:27

と表現できる．しかし，この 3 点リーダ「...」による省略部分の意味が曖昧である．そこで，3 点リーダを用いずに帰納的定義を用いる．すなわち，**Num** は次を満たす最小の集合である[*1]:

$$\mathbf{Digit} \subseteq \mathbf{Num} \tag{3}$$
$$n \in \mathbf{Num} \wedge d \in \mathbf{Digit} \Longrightarrow nd \in \mathbf{Num} \tag{4}$$

なぜ「最小」という条件がついているのであろうか．**Num** が，式 3 と式 4 を満たすという条件のみである場合，「壱」という記号が集合 **Num** に含まれていても構わない．こうした余分な記号が **Num** に含まれないように「最小」という条件がついているのである．

同様にして帰納的定義を用いて算術式の集合 **Aexp** を定める．**Aexp** は，算術式を表す英単語 Arithmetic expressions を省略した名前である．自然数を表す記号 n は算術式であるとする:

$$n \in \mathbf{Num} \implies n \in \mathbf{Aexp} \tag{5}$$

たとえば，31 は自然数を表す記号であるので算術式である．すなわち，$\texttt{31} \in \mathbf{Num}$ であるので $\texttt{31} \in \mathbf{Aexp}$ である．

ふたつの算術式の加算式は

a_0 および a_1 が算術式であるならば，a_0+a_1 は算術式である

という規則から構成する．この規則を言い換えると，

$$a_0, a_1 \in \mathbf{Aexp} \implies a_0\texttt{+}a_1 \in \mathbf{Aexp} \tag{6}$$

である．たとえば，1 や 2 は算術式であるので，1+2 は算術式である．ここでメタ変数 a_0 と a_1 は，任意の算術式を表すのに用いられている．「メタ」とは，「超えて」「高次の」といった意味を表す接頭語である．ここでは，（そのようなものは定義していないが）算術式に出現する変数ではなく，算術式を記述するときに使われている変数という意味で，メタ変数という用語を使っている．減算式や乗算式も同様の規則によって定まっているものとする．

以上の規則をまとめる．算術式は規則

$$n \in \mathbf{Num} \implies n \in \mathbf{Aexp} \tag{7}$$
$$a_0, a_1 \in \mathbf{Aexp} \implies a_0\texttt{+}a_1 \in \mathbf{Aexp} \tag{8}$$
$$a_0, a_1 \in \mathbf{Aexp} \implies a_0\texttt{-}a_1 \in \mathbf{Aexp} \tag{9}$$
$$a_0, a_1 \in \mathbf{Aexp} \implies a_0\texttt{*}a_1 \in \mathbf{Aexp} \tag{10}$$

からのみ定められるものとする．なぜ「からのみ」と限定したのであろうか．これは **Num** を定義するときに「最小」という条件をつけたのと同様に余分な要素が **Aexp** に含まれないようにするためである．たとえば，1+2*3 は算術式であることは次のように確かめられる．式 1 より，$\texttt{1}, \texttt{2}, \texttt{3} \in \mathbf{Digit}$ である．よって，式 3 より $\texttt{1}, \texttt{2}, \texttt{3} \in \mathbf{Num}$ である．さらに式 7 より，$\texttt{1}, \texttt{2}, \texttt{3} \in \mathbf{Aexp}$ である．したがって，式 8 と 10 より，$\texttt{1+2} \in \mathbf{Aexp}$ であり，$\texttt{1+2*3} \in \mathbf{Aexp}$ である．

このように帰納的定義によって構文を形式的に定めることができるが，現在，論文やテキストではこの方法を目にすることはあまりない．これは帰納的定義をより簡潔に記述することができる BNF が広く使われているためである．

[*1] 定義を簡単にするために，ここでは 013 のような二桁以上で 0 から始まる記号列も **Num** に含んでいる．

1.1.2 BNF で定める構文

構文を定めるときに BNF(Backus-Naur form) やそれを一部改変した記法が良く用いられる．ここではこれらを総称して BNF と呼ぶ．BNF を用いると上記のような帰納的定義を簡潔に記述することができる．たとえば，前小節の 10 進表記の自然数を表す記号列は

$$d ::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$
$$n ::= d \mid nd$$

と記述することができる．“::=” は “can be”，“|” は “or” と読むことができる．すなわち，これらの構文規則は

- d can be 0 or 1 or ... or 9.
- n can be d or n followed by d.
- d と n はこれ以外の値をとることはない.

ということを表している.

問題 1. n が 23 であることはあるか．あるとしたらなぜか．また，n が「壱」であることはあるか．ないとしたらなぜか.

BNF を用いると，前小節の算術式の帰納的定義は

$$a ::= n \mid a_1\texttt{+}a_2 \mid a_1\texttt{-}a_2 \mid a_1\texttt{*}a_2$$

と表される.

この構文規則は，記号列 2+3*4 を (2+3)*4 と 2+(3*4) のどちらの構文に解釈すべきかについて何も述べていないことに注意して欲しい．この構文規則は，記号列の解釈を与えているのではなく，算術式の木構造，すなわち**抽象構文木**を定めているのである.

1.1.3 Calc の構文

プログラミング言語 Calc には自然数を表すための記号列が有限個あるものとする：

$$n ::= \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{10} \mid \texttt{11} \mid \cdots \mid \texttt{2147483647}$$

なお，$2147483647 = 2^{31} - 1$ である．Calc の算術式は，構文規則

$$a ::= n \mid a_1\texttt{+}a_2 \mid a_1\texttt{*}a_2 \mid a_1\texttt{-}a_2 \mid \texttt{(}a_1\texttt{)}$$

から構成されるもののみとする．ここで構文の中に括弧 () が含まれていることに注意してほしい．プログラミング言語 Calc の実装を簡単にするため，ここでは構文解析をして得られる**具象構文木**を定めているのである.

問題 2. 次の文字列の中からプログラミング言語 Calc の算術式として適当なものを選びなさい.

- 1
- 1+2

- `-1`
- 1 （←数式フォント）
- `a1+a2`
- `1+(2-3)`
- `((1))`
- `1+2-3`

問題 3. `1+2-3` および `1+2*3` をそれぞれ 2 種類の構文木で表しなさい．

1.1.4 Calc の意味論

記号列を数という抽象的な対象に対応づけるのに意味関数を用いる．意味関数 $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ は

$$\mathcal{N}[\![\mathtt{0}]\!] = 0$$
$$\mathcal{N}[\![\mathtt{1}]\!] = 1$$
$$\vdots$$
$$\mathcal{N}[\![\mathtt{2147483647}]\!] = 2147483647$$

というように，記号を数に対応づける[*2]．

次のメタ変数は，それぞれの集合の範囲の値をもつものとする:

$$n \in \mathbf{Num}$$
$$a \in \mathbf{Aexp}$$
$$v \in \mathbb{Z}$$

ここでメタ変数 v は値を表す英単語 value の頭文字から命名している．

算術式と整数値の関係は

$$a \to v$$

と表す．この関係は，算術式 a が整数値 v に評価される，と読む．この関係が成立するかは以下の意味規則により定める．

自然数を表すための記号列が算術式である場合，それを意味関数で評価した整数値に評価される．これを表現する規則は

$$\text{NUM}\,\frac{}{n \to v} \quad \text{where } \mathcal{N}[\![n]\!] = v$$

である．ここで NUM はこの規則の名前である．水平線の上に書かれる関係を前提，下に書かれる関係を結論と呼ぶ．読みやすくするため，前提が空の場合は，水平線は省略する．水平線を省略すると前提が空である規則 NUM は

$$\text{NUM} \quad n \to v \quad \text{where } \mathcal{N}[\![n]\!] = v$$

という形になる．

ふたつの算術式の加算式 $a_1\texttt{+}a_2$ は，a_1, a_2 の算術式を評価した結果をそれぞれ v_1, v_2 とすると $v_1 + v_2$ に評価されることにする．これを表現する規則は

$$\text{PLUS}\,\frac{a_1 \to v_1 \qquad a_2 \to v_2}{a_1\texttt{+}a_2 \to v} \quad \text{where } v = v_1 + v_2$$

[*2] $\mathbb{Z}$ は，整数の集合．

である．減算式や乗算式についての規則も同様にして表現することができる．

Calc の意味規則をまとめると以下のようになる：

$$\text{NUM} \quad n \to v \quad \text{where } \mathcal{N}[\![n]\!] = v$$

$$\text{PLUS}\frac{a_1 \to v_1 \quad a_2 \to v_2}{a_1\texttt{+}a_2 \to v} \quad \text{where } v = v_1 + v_2$$

$$\text{MINUS}\frac{a_1 \to v_1 \quad a_2 \to v_2}{a_1\texttt{-}a_2 \to v} \quad \text{where } v = v_1 - v_2$$

$$\text{MULT}\frac{a_1 \to v_1 \quad a_2 \to v_2}{a_1\texttt{*}a_2 \to v} \quad \text{where } v = v_1 \cdot v_2$$

$$\text{PARENT}\frac{a_1 \to v_1}{\texttt{(}a_1\texttt{)} \to v_1}$$

算術式の評価は**導出木**によって行うことができる．例えば，

$$\dfrac{\dfrac{\dfrac{\texttt{2} \to 2 \quad \texttt{3} \to 3}{\texttt{2+3} \to 5}}{\texttt{(2+3)} \to 5} \quad \dfrac{\dfrac{\texttt{4} \to 4 \quad \texttt{6} \to 6}{\texttt{4-6} \to -2}}{\texttt{(4-6)} \to -2}}{\texttt{(2+3)*(4-6)} \to -10}$$

という導出木が得られるので，式 `(2+3)*(4-6)` は整数値 -10 に評価される．

以上で，高水準言語である Calc の意味を具体的な計算機を仮定せずに形式的に定めることができた．

1.2 字句解析器と構文解析器の実現

本節では BNFC によってプログラミング言語 Calc の字句解析器（レクサ，スキャナ）と構文解析器（パーサ）の実現を行う方法を説明する．

ほぼ BNF の記法にしたがったファイル `Calc.cf` を用意する：

Calc.cf
```
EAdd. Exp  ::= Exp  "+" Exp1 ;
ESub. Exp  ::= Exp  "-" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
EInt. Exp2 ::= Integer ;

coercions Exp 2 ;
```

BNFC を用いて Java または Haskell の字句解析器と構文解析器の実現をするには，それぞれ

```
bnfc -m -java    Calc.cf
bnfc -m -haskell Calc.cf
```

と打鍵する．ここで，オプション`-m` の指定により `Makefile` が生成される．

1.2.1 GNU make について

カレントフォルダに生成された `Makefile` には，GNU make で用いるルールが記述されている．コマンド

```
make
```

を打鍵すると，Makefile に記述された一番上のルールが実行されて，処理系のコンパイルが行われる．ここで

```
make clean
```

と打鍵すると，ターゲット clean に記述されたコマンドが実行されて，コンパイルで生成されたファイルが削除される．

GNU make の公式ページは http://www.gnu.org/software/make/ である．少し古いバージョンの GNU make は，有志によって日本語訳がされている（http://www.ecoop.net/coop/translated/GNUMake3.77/make_toc.jp.html）．

1.2.2 Java の場合

JLex で構文解析器を生成して，CUP で字句解析器を生成して，それらをソースコードに追加して，Java コンパイラである javac で解釈器のテンプレートをコンパイルし，コンパイル結果を実行する．これらを行うコマンドは Make ファイルに記述されているので，実際に打鍵するコマンドは

```
$ make
```

のみである．

以下のような実行結果が出力されたら成功である：

```
$ echo "1 + 3 * (4 - 2)" | java Calc/Test

Parse Succesful!

[Abstract Syntax]

(EAdd (EInt 1) (EMul (EInt 3) (ESub (EInt 4) (EInt 2))))

[Linearized Tree]

1 + 3 * (4 - 2)
```

1.2.3 Haskell の場合

Happy で構文解析器を生成して，Alex で字句解析器を生成して，GHC で解釈器のテンプレートをコンパイルして，コンパイル結果を実行する：

```
$ happy -gca ParCalc.y
$ alex -g LexCalc.x
$ ghc --make TestCalc.hs -o TestCalc
$ echo "1 + 3 * (4 - 2)" | ./TestCalc
echo "1 + 3 * (4 - 2)" | ./TestCalc

Parse Successful!

[Abstract Syntax]
```

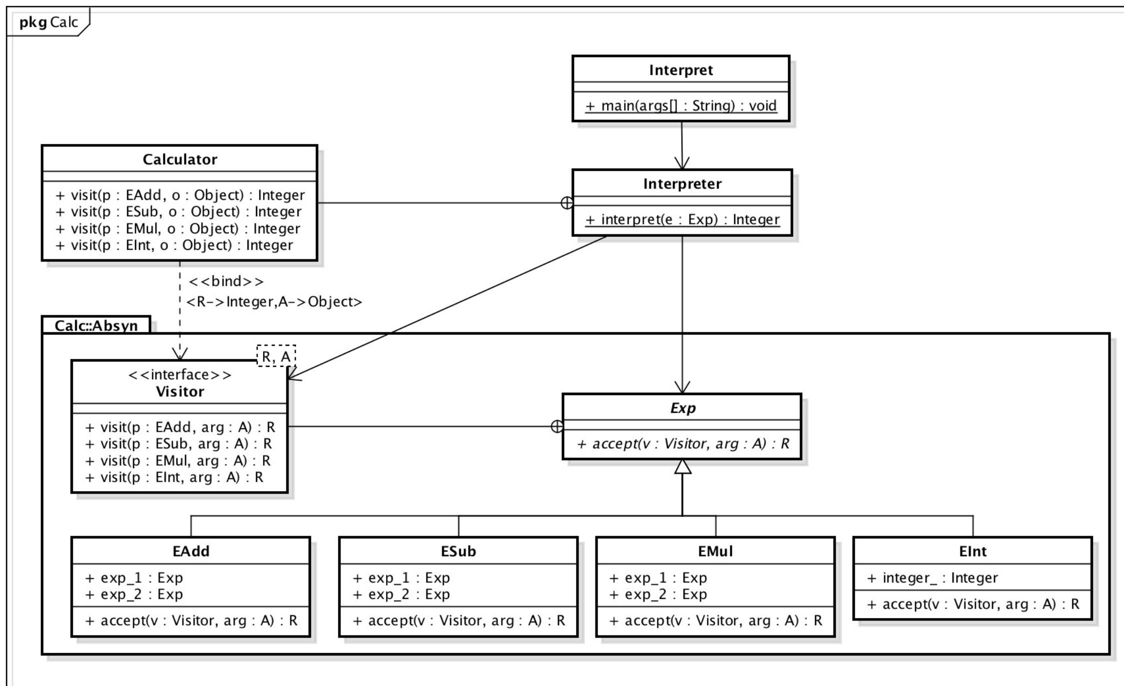

図 1　Calc のクラス図

```
EAdd (EInt 1) (EMul (EInt 3) (ESub (EInt 4) (EInt 2)))

[Linearized tree]

1 + 3 * (4 - 2)
```

1.3　解釈器の実現

1.3.1　Java の場合

図 1 のような Visitor パターンを用いる．Exp クラスは，「訪問者」を受け入れる accept メソッドと，「訪問者」を表す Visitor インタフェースをメンバとしてもつ．抽象クラスである Exp クラスは，加算式，減算式，乗算式，整数値をそれぞれ表すクラスによって継承される．「訪問」を受けたときの振る舞いはそれぞれの継承クラスの accept による．「訪問者」は，Calculator クラスの多相メソッド visit によって訪問する継承クラスによって振る舞いを変える．

参考までに，クラス図の読み方を簡単に述べる．クラスやインタフェースは長方形で表される．長方形は水平線で 3 つに分割される．上から順にクラス名やインタフェース名などの情報，フィールドの情報，メソッドの情報が書かれる:

クラス名やインタフェース名などの情報
フィールドの情報
メソッドの情報

抽象クラスの名前は斜体で書かれる．たとえば，抽象クラスである Exp の名前が斜体で書かれている．抽象メソッドの名前は斜体で書かれる．たとえば，Exp クラスの accept メソッドの名前が斜体で書かれている．静的メソッドには下線が引かれる．たとえば，Interpret クラスの main メソッドや Interpreter クラスの interpret メソッドには下線が引かれている．フィールド名やメソッド名の前に書かれる「+」は，そのメソッドが public であることを表す．インタフェース名の上には，明示的に << interface >> というステレオタイプが書かれる．引数や返却値のクラスが総称型になることでテンプレート化（パラメトライズド）されたインタフェースは右上の破線で囲まれた箱の中に総称型が列挙される．総称型の束縛は，実現関係を表す矢印の付近に << bind >> と書いてその下に束縛関係を列挙する．たとえば，Visitor クラスは総称型 R, A でパラメータ化されて Calculator クラスで実現するときに R が Integer に A が Object に束縛されている．クラスのメンバとしてクラスやインタフェースがあることは矢印の先が ⊕ である関連によって表される．たとえば，Interpreter クラスは Calculator クラスをメンバにもつ．この場合，Calculator クラスは Interpreter クラスで宣言されたもので，ネストしたクラスである．

以下，コードの詳細をみていく．Exp クラスでは算術式の抽象構文木の構成と木のノードへの「訪問」を扱う:

Calc/Absyn/Exp.java

```
package Calc.Absyn; // Java Package generated by the BNF Converter.

public abstract class Exp implements java.io.Serializable {
  public abstract <R,A> R accept(Exp.Visitor<R,A> v, A arg);
  public interface Visitor <R,A> {
    public R visit(Calc.Absyn.EAdd p, A arg);
    public R visit(Calc.Absyn.ESub p, A arg);
    public R visit(Calc.Absyn.EMul p, A arg);
    public R visit(Calc.Absyn.EDiv p, A arg);
    public R visit(Calc.Absyn.EInt p, A arg);
  }
}
```

「訪問」を受け入れるためのメソッド accept と「訪問先」のクラスの種類によって振る舞いを変える多相メソッド visit は，Exp を継承したクラス（後述）で実装される．

Calc の処理は，ラッパークラスである Interpret の main メソッドから始まる:

Interpret.java

```
package Calc;
import java_cup.runtime.*;
import Calc.*;
import Calc.Absyn.*;
import java.io.*;

public class Interpret {
    public static void main(String args[]) throws Exception {
        Yylex l  = new Yylex(System.in);
        parser p = new parser(l) ;
        Calc.Absyn.Exp parse_tree = p.pExp() ;
        System.out.println(Interpreter.interpret(parse_tree)) ;
    }
}
```

標準入力 System.in から得られた文字列を字句解析し l を得る．この結果に構文解析を行って構文木 parse_tree を得る．この構文木を Interpreter クラスの interpret メソッドに渡す．

Interpreter クラスの interpret メソッドでは，「訪問」を受け取った Exp クラスに accept メソッドによって受け入れてもらう．Calc の解釈器は整数値を返すのでパラメータ `R` には `Integer` を用いる．また，引数には情報を渡す必要がないので `Null` オブジェクトを渡す．「訪問者」がどのような振る舞いをするかは Calculator クラスの多相メソッド visit で書く．たとえば，引数が EAdd クラスであった場合，その二つの子どもの式に「訪問」を accept メソッドで受け入れてもらって得られた Integer の結果の和を返却値とする．また，たとえば，引数が EInt クラスであった場合，（新たに「訪問」行うことなく）もっている整数値を返却値とする．accept メソッドが呼び出されているところでは，必ず第一引数に Calculator のオブジェクト自身が渡されていることに注意してほしい．以下でみるが，受け入れてもらった先で Calculator クラスの visit が再度呼ばれるようになっている．

Interpreter.java

```java
package Calc;
import Calc.Absyn.*;

public class Interpreter {
    public static Integer interpret(Exp e) {
        return e.accept(new Calculator(), null) ;
    }

    private static class Calculator implements Exp.Visitor<Integer,Object> {
        public Integer visit(Calc.Absyn.EAdd p,Object o) {
            Integer a = p.exp_1.accept(this, null);
            Integer b = p.exp_2.accept(this, null);
            return a + b;
        }
        public Integer visit(Calc.Absyn.ESub p,Object o) {
            Integer a = p.exp_1.accept(this, null);
            Integer b = p.exp_2.accept(this, null);
            return a - b;
        }
        public Integer visit(Calc.Absyn.EMul p,Object o) {
            Integer a = p.exp_1.accept(this, null);
            Integer b = p.exp_2.accept(this, null);
            return a * b;
        }
        public Integer visit(Calc.Absyn.EInt p, Object o) {
            return p.integer_;
        }
    }
}
```

Exp クラスを継承したクラスでは，「訪問者」を受け入れる抽象メソッド accept が実装される．accept の中では，visit メソッドにそのクラスのオブジェクト自身（すなわち Exp クラスのサブクラス）と引数が渡される．

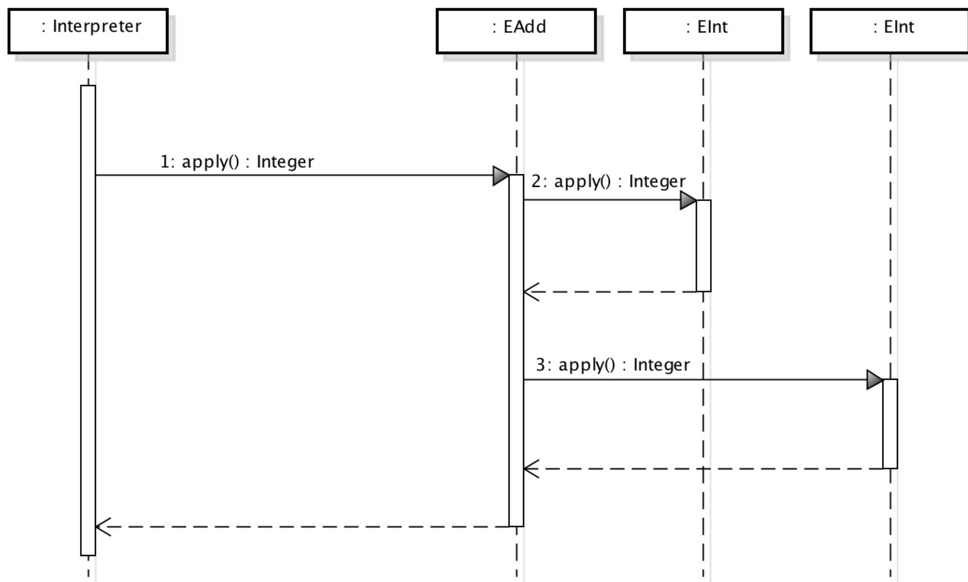

図 2　データ構造と処理を同一のクラス Interpreter で定義したと仮定したときに 2+3 を解釈したときのシーケンス図

Calc/Absyn/EAdd.java

```
package Calc.Absyn; // Java Package generated by the BNF Converter.

public class EAdd extends Exp {
  public final Exp exp_1, exp_2;

  /* Java のコンストラクタのメソッド名には，クラス名を用いる． */
  public EAdd(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }

  public <R,A> R accept(Calc.Absyn.Exp.Visitor<R,A> v, A arg) {
    return v.visit(this, arg);
  }

  public boolean equals(Object o) {
    if (this == o) return true;
    if (o instanceof Calc.Absyn.EAdd) {
      Calc.Absyn.EAdd x = (Calc.Absyn.EAdd)o;
      return this.exp_1.equals(x.exp_1) && this.exp_2.equals(x.exp_2);
    }
    return false;
  }

  public int hashCode() {
    return 37*(this.exp_1.hashCode())+this.exp_2.hashCode();
  }
}
```

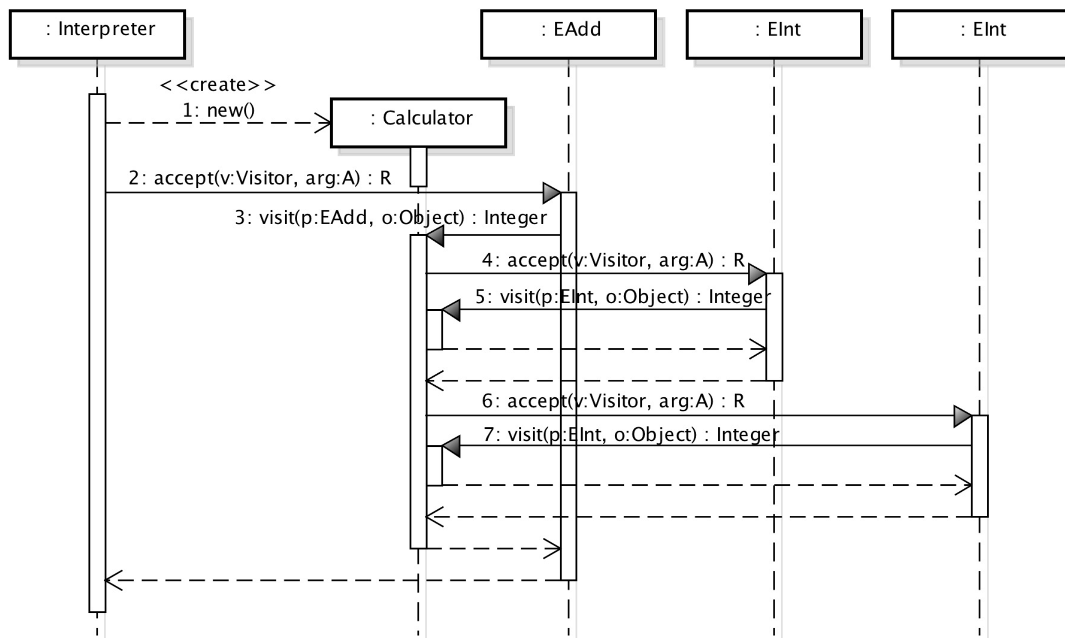

図 3　2+3 を解釈したときのシーケンス図

シーケンス図を用いて解釈器の処理の流れをみてみる．図 3 は，式 2+3 を理するときの流れを示している．ただし Interpret のインスタンスから Interpreter が呼ばれ値が返却される部分は省略してある．なお，たとえば，:Interpreter は Interpreter のインスタンスを表す．

1. :Interpreter は:Calculator を作る．（interprete メソッドにおいて `new Calculator()` でこれを行う．）
2. :Interpreter は，:EAdd に対して，accept メソッドを呼び出す．このとき第一引数には 1 で作成した:Calculator を渡す．
3. :EAdd は，渡された:Calculator の visit メソッドを呼び出す．このとき第一引数には:EAdd 自身を渡す．
4. :Calculator は，:EInt に対して，accept メソッドを呼び出す．このとき第一引数には:Calculator 自身を渡す．この:EInt は部分式 2 に対応する左の子どものフィールドである．
5. :EInt は，渡された:Calculator の visit メソッドを呼び出す．このとき第一引数には:EInt 自身を渡す．続いて，整数値 2 が:Calculator → :EInt → :Calculator という順に返却される．
6. 4 と同様に，:Calculator は，:EInt に対して，accept メソッドを呼び出す．
7. 5 と同様に，:EInt は，渡された:Calculator の visit メソッドを呼び出す．:EInt から:Calculator に整数値 3 が返却される．5 で返却された整数値 2 とここで得られた整数値 3 の和 5 が最終的に:Interpreter の返却値になる．

■Java プログラムのコンパイルと実行　Java コンパイラのコマンド名は javac である．パッケージ Calc の中の Interpret.java をコンパイルするには，Calc の含まれたディレクトリにおいて

```
$ javac Calc/Interpret.java
```

とタイプする．

サンプルセッションは以下の通りである：

```
$ echo "1 + 3 * (4 - 2)" | java Calc/Interpret
7
```