プログラミング言語設計論

2014年度

第7回: 型推論

The Expression Problem

担当: 増原英彦

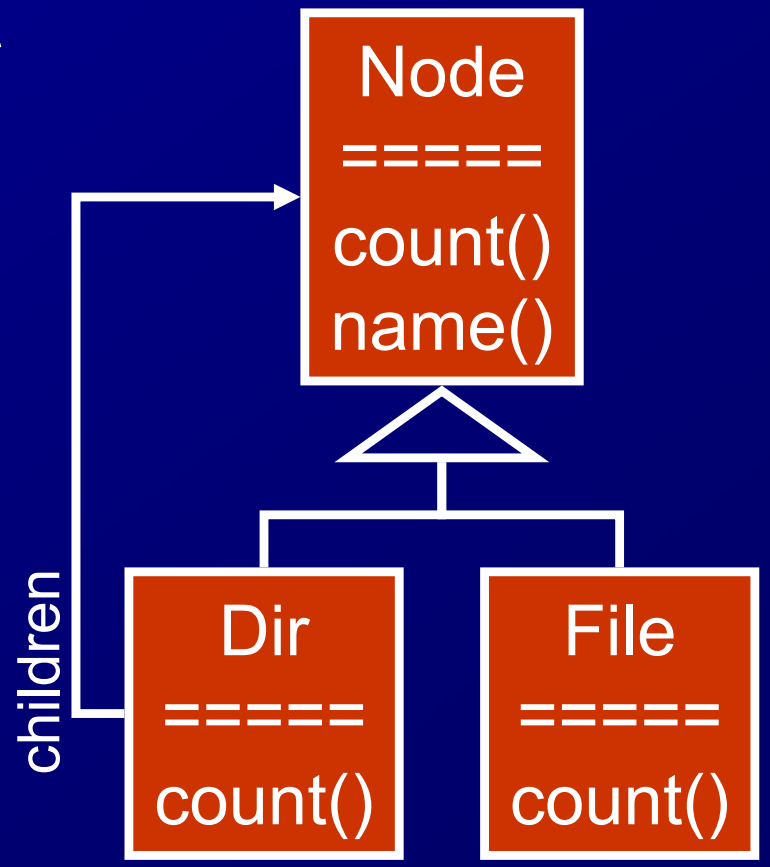
問題1/3 (5分)

1. 動的型付言語が静的型付言語に対して優れている点を挙げよ
2. 動的型付言語が静的型付言語に対して劣っている点を挙げよ

問題2/3 (10分)ファイルシステムのためのクラスライブラリを批判せよ

- countはファイルの総数を計えるメソッド

※ライブラリ利用者はライブラリ定義を変更できない



問題3/3 (10分) ファイルシステムのためのクラスライブラリを設計せよ

- Fileは名前と大きさを持つ
- Dirは名前と子供(複数のFileとDir)を持つ
- 利用者は以下を行いたい
  - ファイルの総数を数える
  - ファイルの大きさの合計を求める
  - 特定の名前のファイルを1つ見つけ、その大きさを答える

※ライブラリ利用者はライブラリ定義を変更できない

※「ファイル階層の走査」はライブラリが行うように

型推論

- 型宣言のないプログラム中の式の型を決定すること
  - 例:

```
def sumOf(f,n) {
   if (n==0) return 0;
   else return f(n) + sumOf(f,n-1); }
```

からsunOf : (int->int * int) -> int

  - 関数型言語(ML, Haskell)で成功

型推論の方法

古典的な言語の型検査

ここでは式と文
あわせて「式」と

- 型検査規則
  - 規則: 式の種類ごとに部分式および結果の許される型の組み合わせ
    - 例: if e1 then e2 else e3 ➔ e1の型はbool, e2とe3と結果の型は同一
  - 検査方法: 式に型付けをして、規則を使って確認
- 型付け
  - ~~変数・関数の型はプログラムに宣言してある~~
  - ~~型規則を「部分式の型から結果の型を求める」と読みかえる if e1 then e2 else e3 ➔ 結果の型はe2の型~~

```
def s(f,n) { return f(n+1); }
```

定数や組込関数の型は与えておく

規則を満たすような割り当てを探索

- 探索問題なので複数の解があり得る
- 最も一般的なものを選ぶ
  - それが唯一になるとき principal typeという
  - 総称型が必要になることが多い

オブジェクト指向言語の型推論

- 部分的に型を書く必要あり
- (例) Scala: 局所変数のみを推論

```
object InferenceTest1 extends Application {
  val x = 1 + 2 * 3
  val y = x.toString()
  def succ(x: Int) = x + 1
} (scala-lang.orgより)
```

オブジェクト指向言語の型推論

- Scalaはなぜ局所変数だけ推論する?
  - オブジェクトがあると型推論が難しくなる

```
class Person { String name; }
class File { byte[] name; }
def printName(obj) { print(obj.name); }
```

    - objの型: Person, Fileどちらも正しい
    - printNameの結果型: String? byte[]?

動的型付言語は型推論できるか?

- 動的型付言語
  - Lisp, Smalltalk, Python, Ruby, etc.
  - 実行時に型検査
    - 値に型情報を付加する
    - 引数の型を調べて適切な演算をする
      例: def double(x) { return x+x; }
      → +の所でxの型を調べてiadd, fadd, あるいはエラー
- 型推論できる→型検査が不要
  - 正しいプログラムは実行時に型誤りを起こさないはず

動的型付言語は型推論できるか?

- できる → 静的型付言語?
- 難しさ
  - 暗黙のユニオン型:

    ```
    def sq(x) {
    if (x<0) return "NG"; else return sqrt(x); }
    ```

  - 再帰的な関数:

    ```
    (lambda (f)
     ((lambda (x) (f (x x))) (lambda (x) (f (x x)))))
    ```

動的型付言語の型宣言 Common Lispのdeclare

- 目的: コンパイラへのヒント
- 例:

```
(defun double (x)
  (declare (type fixnum x))
  (+ x x))
```

- 意味: コンパイラはxが固定長整数だと思ってコード生成してよい
  - 間違った場合の結果は保証しない

実行時に型検査
- ◆値に型情報を付加する
- ◆引数の型を調べて適切な演算をする
  例: def double(x) { return x+x; }
  → +の所でxの型を調べてiadd, fadd, あるいはエラー

動的型付言語の型推論: soft typing [CF91]

- プログラムを部分的に型推論する体系
- 目的: 誤りの静的な検出、冗長検査の除去
- 方法:
  - 静的型付言語の型推論を適用
  - 1つに決まらないときはユニオン型を使う

    例: def sq(x) {
    if (x<0) return "NG"; else return sqrt(x); }
    → sq : int -> string + float
  - 分からない所は実行時型検査を残す

    例: sq(123)*2

段階的型付(gradual typing) [ST07]

- 動的型付言語は型のことを考えなくてよいのですばやく開発するのに向いている
- 静的型付言語は型に関する誤りを実行前に見つけてくれる
- 動的型付言語で開発して、静的型付言語に移行すればよい
  - 一気に移行するのは大変
  - 序々にやろう → gradual typing
  - module単位で型付・型なし
    - ◆難しさ: 型付の世界に型なしの関数を渡す、その逆

段階的型付の例 [ST07]

```
class Point {
  var x = 0
  function move(dx)
  { this.x = this.x + dx }
}
var a : int = 1
var p = new Point
p.move(a)
```

dxの型は動的

結果も動的

aは整数

pはPoint

aを動的型に変換

段階的型付の例 [ST07]

```
class Point {
  var x = 0
  function bool
    equal(o : [x:int]) {
     return this.x==o.x }}
var p = new Point
var q = new Point
p.equal(q)
```

oは整数フィールド
xを持つ

Pointの型の推論
[x:int,
 equal: [x:int]->bool]

静的に
正しいと分かる

段階的型付の例

```
class Point {
  var x = 0
  function bool
    equal(o : [x:int]) {
     return this.x==o.x }}
var p = new Point
p.equal("hello")
```

静的エラー

段階的型付け言語

- Newspeak [Bracha08]: Smalltalk似 (optional typingと呼んでいる)
- TypedClojure: Clojureに追加
- TypeScript: Javascriptに追加
- Dart: Java似

暗黙の型変換 (implicit type conversion)

- int i=...; float f=...; return i + f; は実数
  - 整数+実数->実数という規則がある
  - return itof(i)+f;に変換されている
    (upcast — より一般的な型への変換)

  どちらでもOK
- (Java)Person x=...; println("x=" + x);
  - 自動的にprintln("x=" + x.toString());に変換
  - PersonとStringには部分型関係なし
  - 予め決められた型への変換のみ

利用者定義の暗黙の型変換

(Scala) [OSV08]より

```
implicit def stringWrapper(s: String) =
    new RandomAccessSeq[Char] { ... }
def printWithSpaces(seq:
RandomAccessSeq[Char]) = ...

printWithSpaces("xyz")
```

自動的にstringWrapperが呼ばれる

The Expression Problem

[Wadler98]

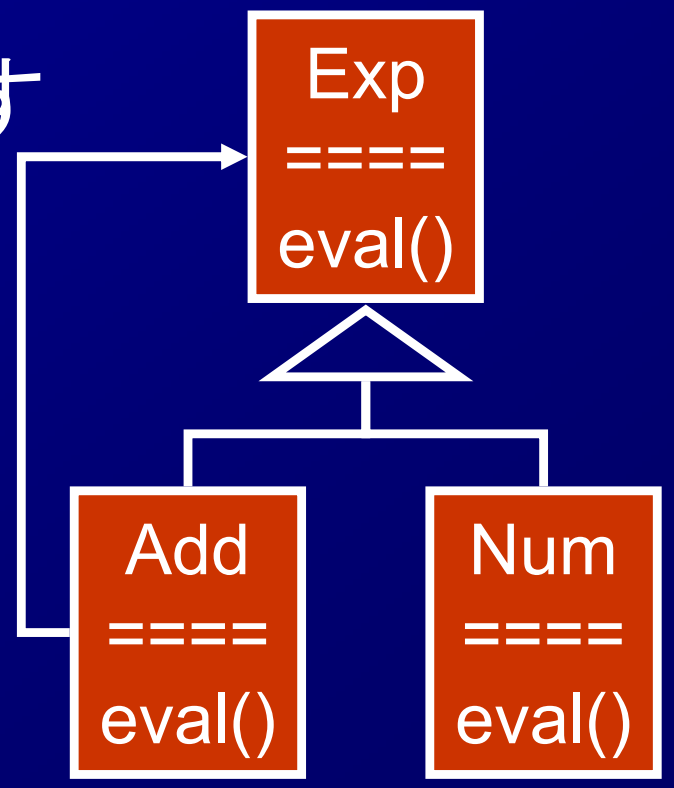
- データ構造とそれに対する操作を型安全に拡張する問題
  - 古くから知られていた (命名は[Wadler98])
- 拡張する = 既存の定義を変更せずに
  - 新しいデータの種類を追加
  - 新しい操作の種類を追加

  の両方を実現

EP: 簡単なインタプリタ

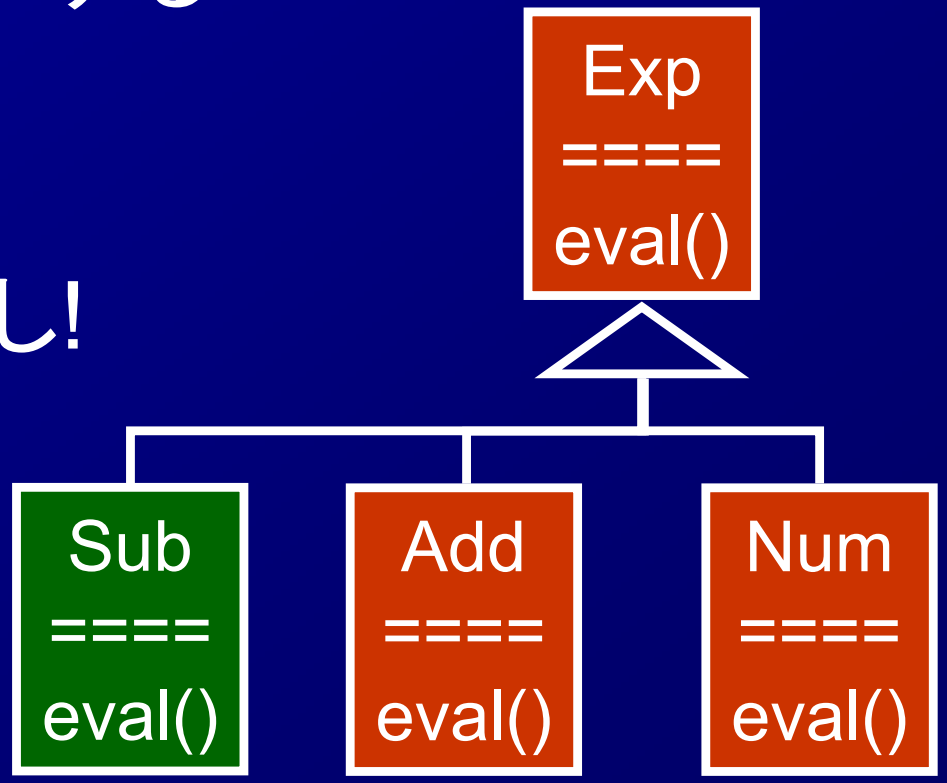
(単一継承を仮定)

- Expは構文木ノードを表わす
  eval()は式の計算をする
- new Add(new Num(1),
  new Num(2)).eval()
  ➔ 3



EP: データ構造の追加

- 引き算も扱えるようにする
  - Subクラスを追加
    evalメソッドを定義
- 既存クラスに変更なし!

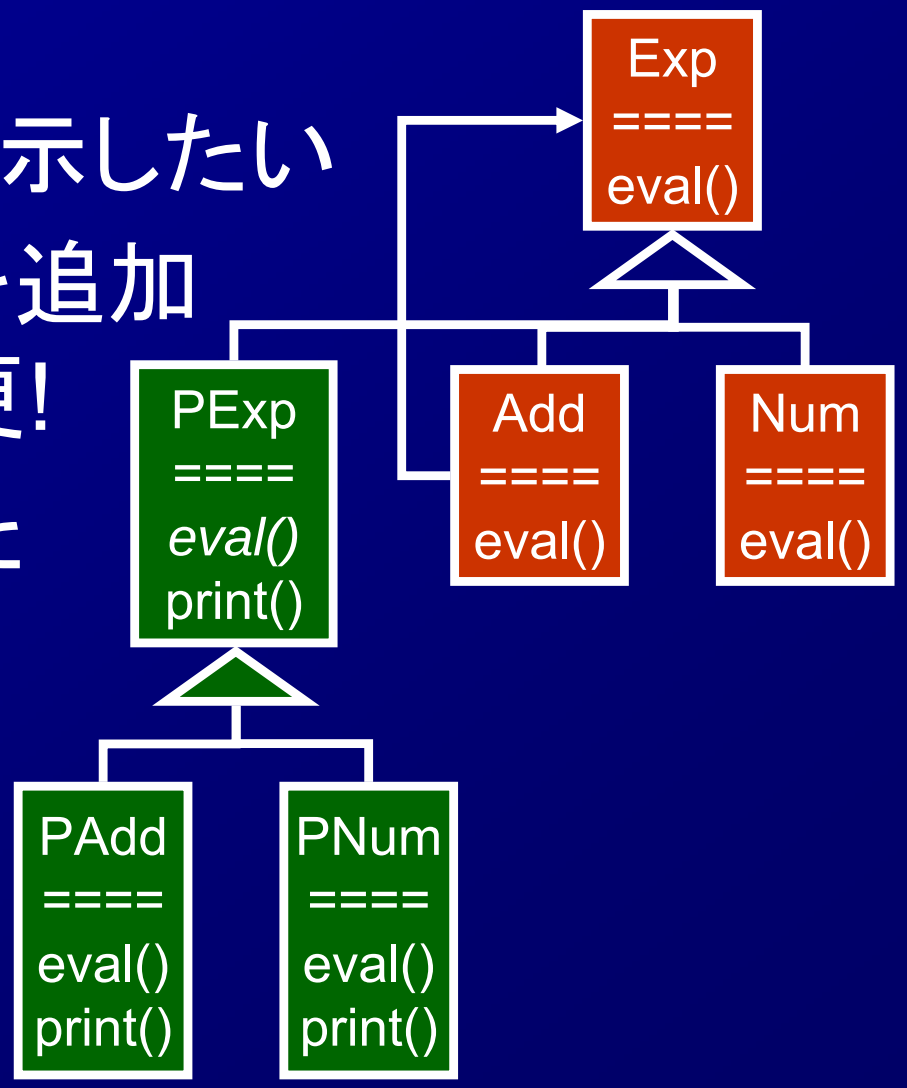


EP: 操作の追加

- 式を文字列にして表示したい

1. Expクラスにprint()を追加
   → 既存クラスの変更!
2. Expクラスを継承した
   PExpクラスを定義
   → 問題?

   evalの定義を
   継承できない

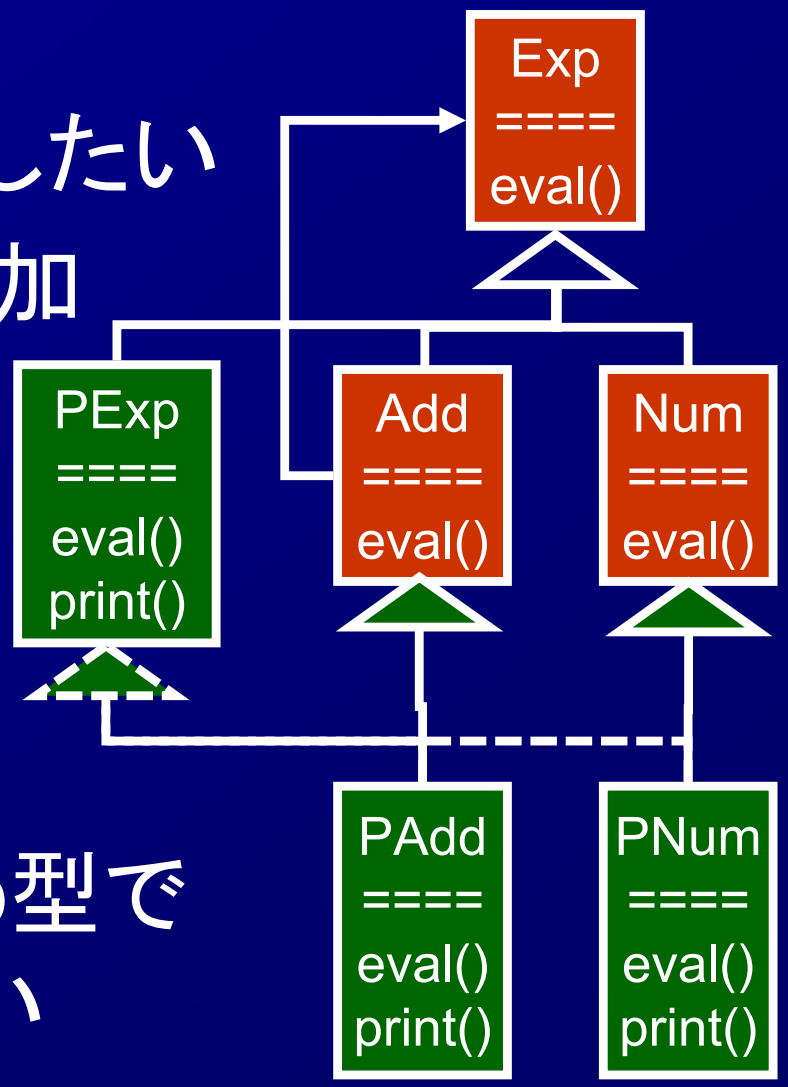


EP: 操作の追加

- 式を文字列にして表示したい

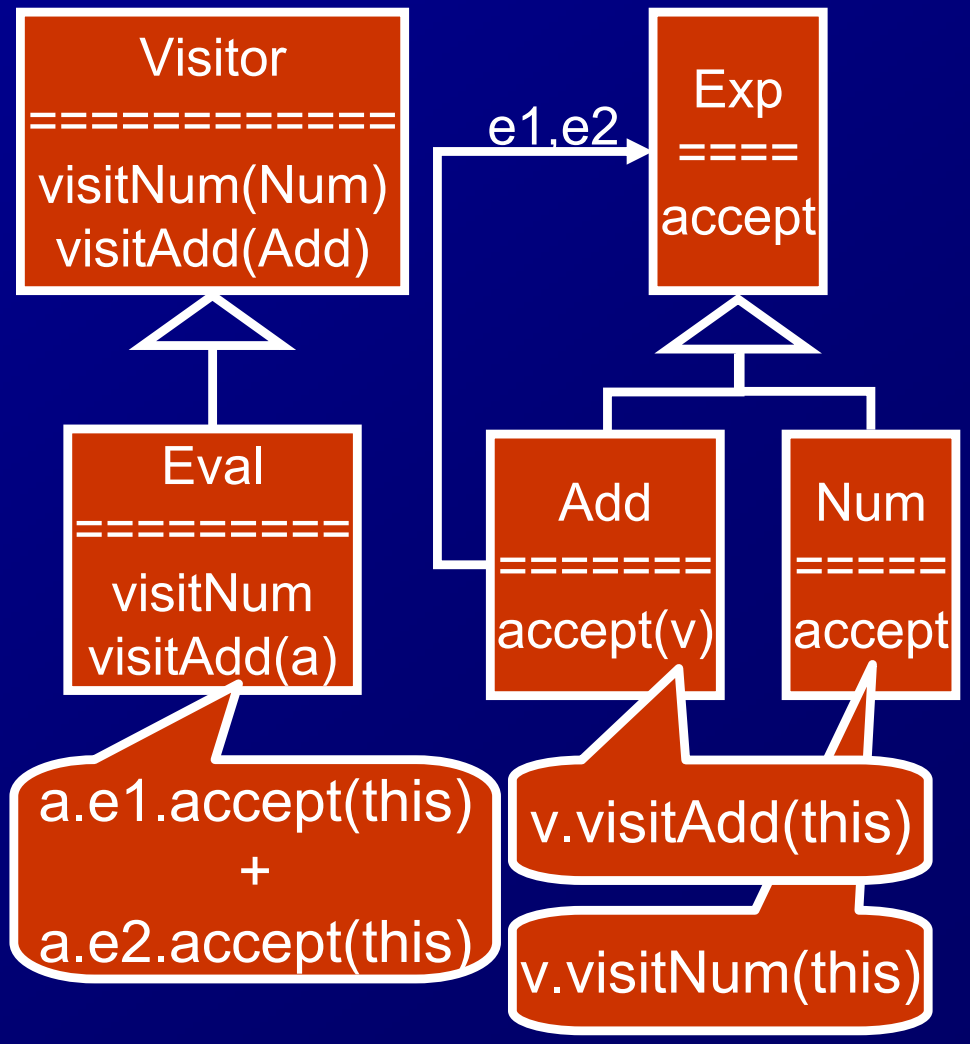
1. Expクラスにprint()を追加
   → 既存クラスの変更!
2. Expクラスを継承した
   PExpクラスを定義
   → 問題?

   PAddの部分式がPExp型で
   ない→printを呼べない



EP: Visitorパターン

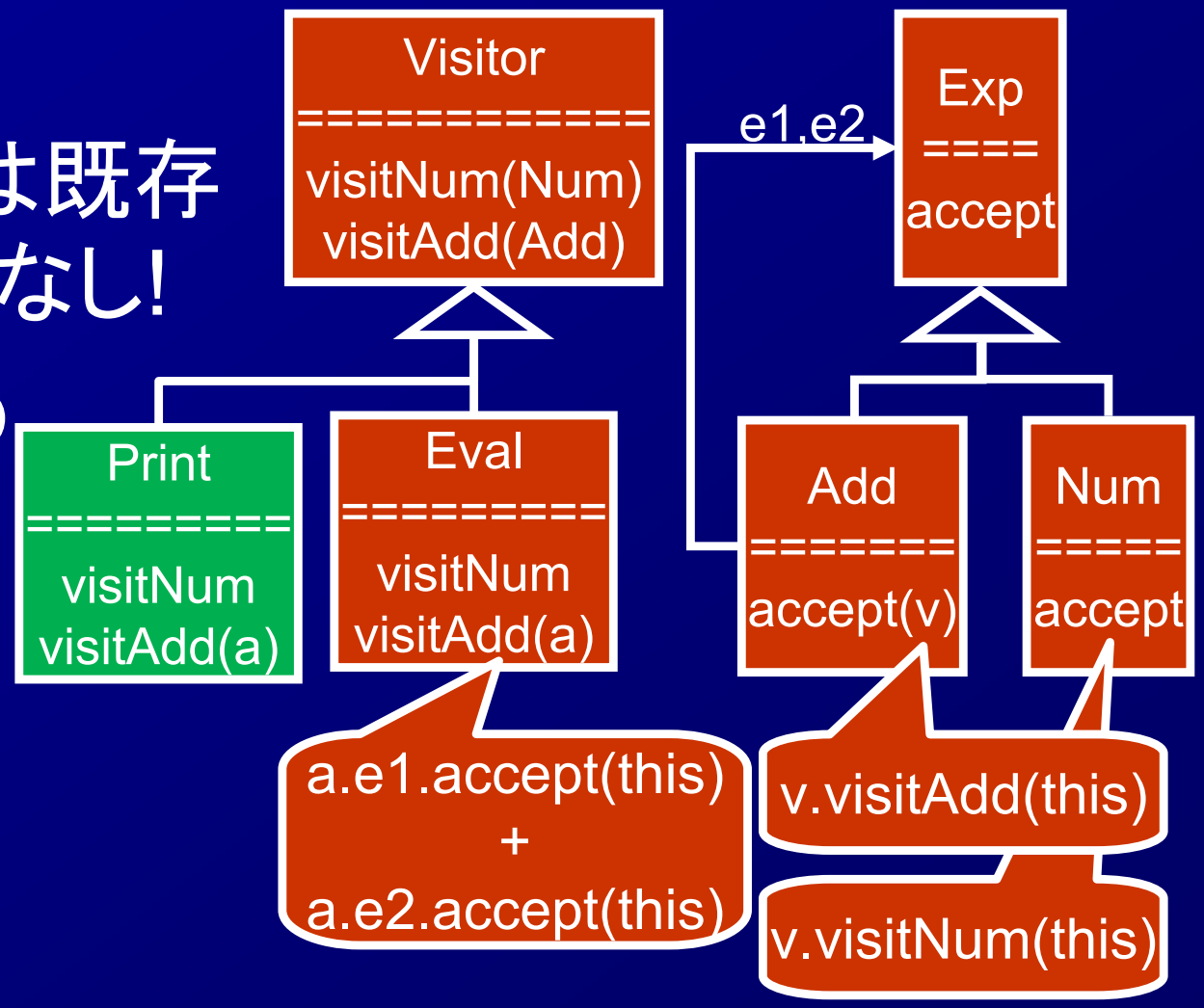
- GoFデザインパターンの1つ
- データ構造からそれに対する操作を分離
- 操作の追加は既存コードの変更なし!



EP: Visitorパターン

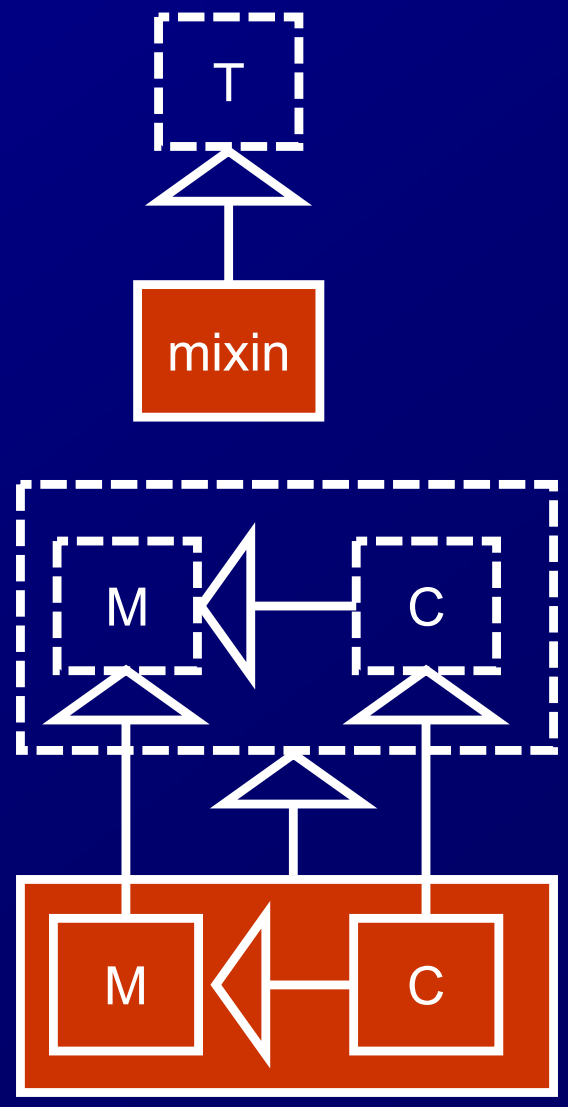
- 操作の追加は既存コードの変更なし!
- データ構造の追加は?

→既存のVisitorを全て変更!

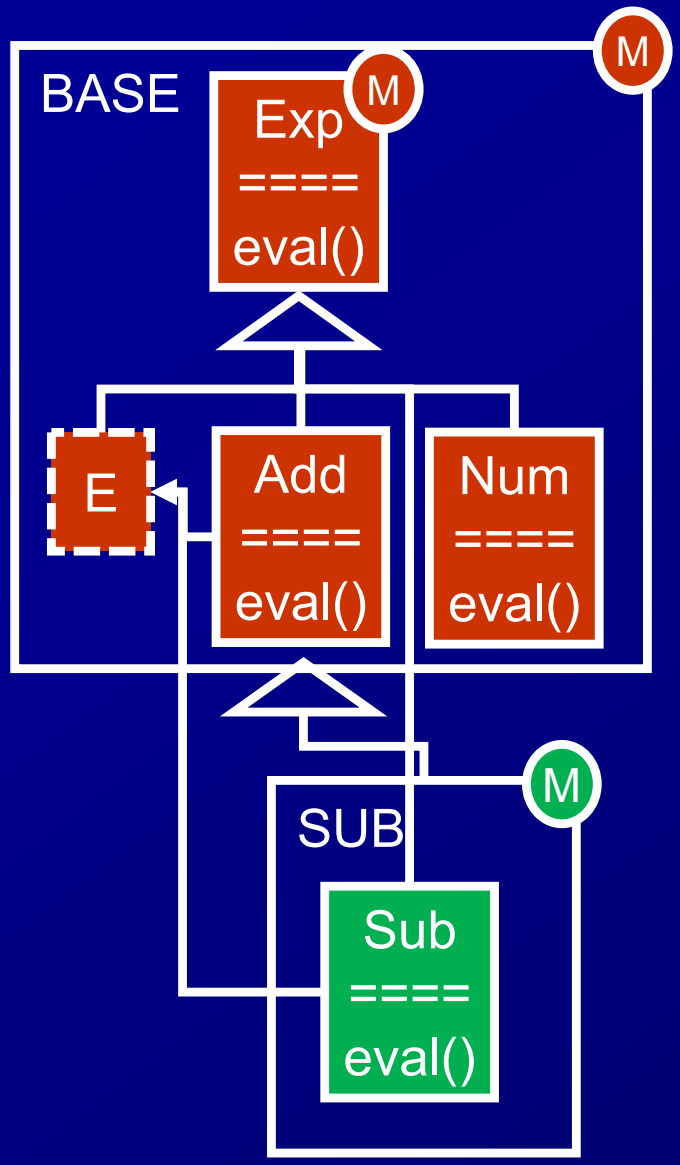


Mixin layersによる解決

- (復習)mixi: 親クラスをパラメタ化したクラス
- Mixin layers: mixinを入れ子にしたもの [SB98]



Mixin layers(と型変数)を使ったEPの解 [ZO04]



- M: mixinとして定義
- E: Expの部分型であるような型変数
  - Addの部分式はE型

- データの種類の追加は素直にできる

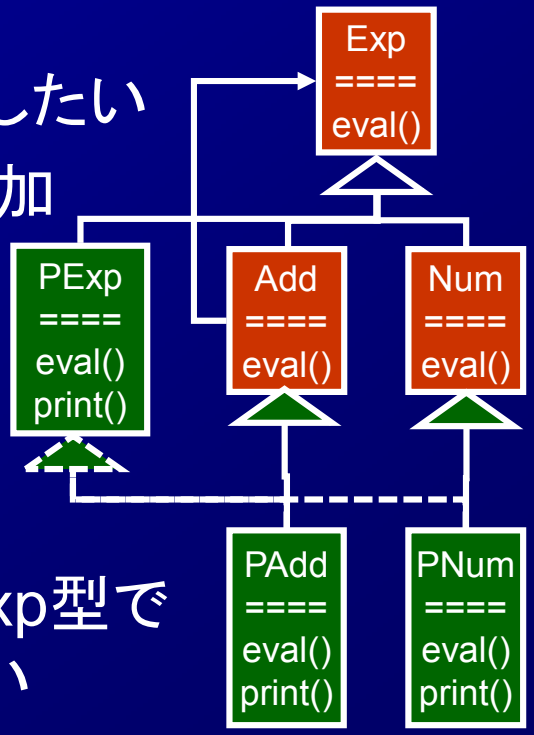
Mixin layersによるEPの解: 操作の追加

EP: 操作の追加

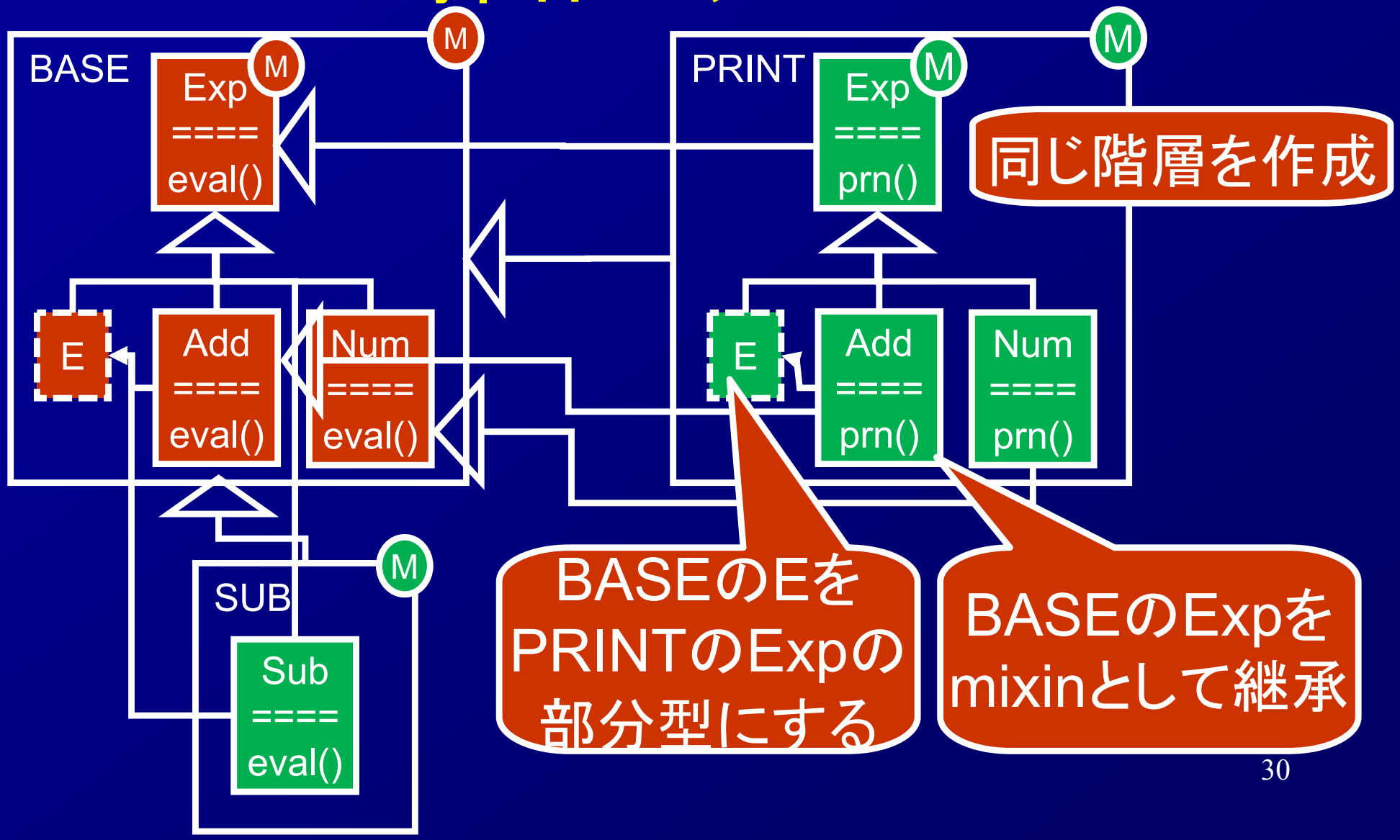
- 式を文字列にして表示したい

1. Expクラスにprint()を追加
   → 既存クラスの変更!
2. Expクラスを継承した
   PExpクラスを定義
   → 問題?

   PAddの部分式がPExp型でない→printを呼べない



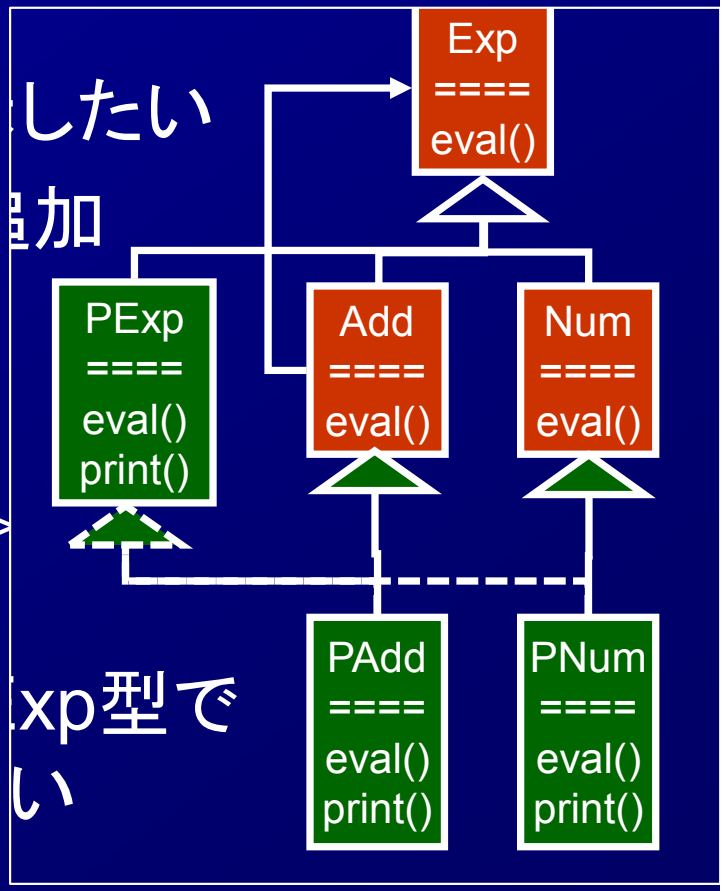
Mixin layersによるEPの解: 操作の追加



総称型と型パラメタへの制約によるEPの解

```
interface Exp<E extends Exp<E>> { int eval(); }
class Add<E extends Exp<E>> implements Exp<E>
  E e1, e2;
  int eval() { ...e1.eval()... } }
class Num<E extends Exp<E>> implements Exp<E>
  int n;
  int eval() {return n; } }

interface PExp<E extends PExp<E>> extends Exp<E> {
  String print(); }
class PAdd<E extends PExp<E>> extends Add<E> implements PExp<E> {
  String print() {...e1.print() ...} }
```



参考文献

[CF91] Cartwright, Robert, and Mike Fagan. "Soft typing." PLDI'91 (1991): 278-292

[ST07] Siek, Jeremy, and Walid Taha. "Gradual typing for objects." ECOOP 2007–Object-Oriented Programming. Springer Berlin Heidelberg, 2007. 2-27.

[Bracha08] Bracha, Gilad, et al. "The newspeak programming platform." Cadence Design Systems (2008).

[OSV] Martin Odersky, Lex Spoon, and Bill Venners, "Implicit Conversions and Parameters" in Chapter 21 of Programming in Scala, First Edition, 2008

[Wadler98] Philip Wadler, “The Expression Problem”, Java Genericity Mailing List, 1998.

[SB98] Smaragdakis, Yannis, and Don Batory. "Implementing layered designs with mixin layers." ECOOP’98—Object-Oriented Programming. 1998. 550-570.

[ZO04] Zenger, Matthias, and Martin Odersky. Independently extensible solutions to the expression problem. No. LAMP-REPORT-2004-004. 2004.